Kokkos and Performance Portable Languages

Bob Robey, AMD

EuroCC-AMD Workshop

May 3rd, 2023

AMD

together we advance_

Performance Portability Languages

- Performance, Portability and Productivity (PPP) is critical for today's HPC application developer
  - Custom implementations for each new computer hardware vendor/type is not sustainable
  - Single-source application code is a necessity
- Department of Energy (DOE) has sponsored PPP conferences and workshops on this topic
- Two popular PPP languages have emerged in the DOE
  - RAJA – LLNL C++ performance portability layer
    - Modular in structure with separation of compute and data management
    - Adaptable for how each application team implements in their code
  - Kokkos – SNL C++ performance portable programming model
    - Comprehensive approach to performance portability
    - Parts being integrated into C++ standard

Both RAJA and Kokkos are great choices for C++ HPC applications. They share more similarities than differences. Both are well-supported for AMD GPUs. AMD staff support both RAJA and Kokkos.

Why Kokkos?

- For C++ applications, Kokkos is an attractive PPP development language

- Kokkos provides a single source capability for C++ codes to run on a variety of parallel CPU and GPU architectures
- Kokkos is well-supported and relatively mature
  - Been around for 10 years
  - Used in many critical applications at Sandia National Laboratories
  - Gaining use in lots of other applications worldwide
  - Selected for Exascale Computing Project funding
- Kokkos generated code performs nearly as well or better than lower-level languages

HIP backend

- Kokkos has long had a HIP backend for selected AMD Processors
  - Both CPUs and GPUs
- The Kokkos team has aggressively developed their implementation for new AMD systems coming online
- In the Fall of 2022, the HIP backend was promoted to production status

- Kokkos handles many of the unique attributes of the AMD GPUs for you

What is Kokkos and How does it work?

- Kokkos (κόκκος) is greek for “grains”, “seed” or “kernels” as in grains of sand or kernels in an ear of corn
- Library based on C++ templates
  - Libraries are quicker to implement and distribute
  - Eventually these techniques can migrate to compilers
    - but this is one-by-one for each compiler
    - additions to language standards takes even longer
    - The concept of multi-dimensional arrays from Kokkos will be implemented as “mdspan” in the C++ 23 standard
- Developed by a team of computer scientists at Sandia National Laboratory
  - Original purpose was to provide an abstraction layer for mathematical solvers
- Supports many backends including OpenMP threading, CUDA, HIP, and others

Kokkos abstractions for GPUs (and parallelism on CPUs)

- Two basic requirements for a GPU programming language
  - Actually, for any fine-grained parallel language that runs on either GPUs or CPUs

- **Execution capability** – this handles how to generate the execution code within a program to run on the target architecture. Generally, this is for loops, but may also include single lines of computation.
- **Memory handling** – the control of the allocation and movement of memory between the CPU and GPU or other memory locations.

- Kokkos, as a portability layer for various fine-grained programming languages, must have an abstract representation of these two requirements.

Execution and Memory abstractions in Kokkos

Execution Spaces -- compute hardware where computations are done

- Execution Patterns
  - Simple loops -- parallel_for
  - Reductions -- parallel_reduce
  - Scans -- parallel_scan
- Execution Policies
  - Range policies -- basically index sets that need to be operated on
  - Team policies – grouping threads into teams as a subset of the execution space for hierarchical parallelism.

Memory Spaces – memory hardware where the data is stored

- Memory Layout
  - LayoutRight vs LayoutLeft or automatic conversion between the two for different execution spaces
- Memory Traits
  - atomic access, random access (shader memory), streaming stores

Kokkos has two main build options for cmake

External build

- Modify CMakeLists.txt
  - **find_package**(Kokkos)
  - **target_link_libraries**(<my_application> Kokkos::kokkos)
- export Kokkos_DIR=<kokkos_path>/lib/cmake/Kokkos

In-line build

- Retrieve a copy of Kokkos
  - git clone https://github.com/kokkos/kokkos Kokkos, or
  - download a zip or tar file of kokkos from https://github.com/kokkos/kokkos
- or create a submodule
  - git submodule add https://github.com/kokkos/kokkos Kokkos
- Modify CMakeLists.txt
  - **add_subdirectory**(Kokkos)
  - **target_link_libraries**(<my_application> Kokkos::kokkos)

Kokkos Examples with HIP backend

We'll demonstrate how Kokkos works with some examples

- Stream Triad
- Shallow Water

It is recommended that you try these out on your own to learn the most effectively.

Stream Triad Example

Stream Triad application - the steps

We'll work through these one at a time

1. First do an external Kokkos build with OpenMP backend
   a. also a HIP backend if you have the appropriate GPU
2. Modify CMakeList.txt to add Kokkos headers and library
3. Add Kokkos views for memory allocation of arrays
4. Add Kokkos execution pattern – parallel_fors
5. Add Kokkos timers
6. Run and measure performance for OpenMP

Portability exercises

1. Rebuild for AMD Radeon GPUs
2. Run and measure performance for AMD Radeon GPU
3. Rebuild Stream Triad using Kokkos build with CUDA backend
4. Run and measure performance for Nvidia GPU

Step 1: Build a separate Kokkos package

- `git clone https://github.com/kokkos/kokkos Kokkos_build`
- `cd Kokkos_build`
- Build Kokkos with OpenMP backend
  - `mkdir build_openmp && cd build_openmp`
  - `cmake -DCMAKE_INSTALL_PREFIX=${HOME}/Kokkos_OpenMP \`
    `-DKokkos_ENABLE_SERIAL=On \`
    `-DKokkos_ENABLE_OPENMP=On ..`
  - `make -j 8`
  - `make install`
  - `cd ..`

Step 1: Build a separate Kokkos package (cont)

- Build Kokkos with HIP backend
  - `module load rocm/5.3.0`
  - `mkdir build_hip && cd build_hip`
  - ```
    cmake -DCMAKE_INSTALL_PREFIX=${HOME}/Kokkos_HIP \
        -DKokkos_ENABLE_SERIAL=ON -DKokkos_ENABLE_HIP=ON \
        -DKokkos_ARCH_ZEN=ON -DKokkos_ARCH_VEGA90A=ON \
        -DCMAKE_CXX_COMPILER=hipcc \
        ..
    ```
  - `make -j 8; make install`
  - `cd ..`

- Set Kokkos_DIR to point to external Kokkos package to use

  `export Kokkos_DIR=${HOME}/Kokkos_HIP`
    ```

Step 2: Modify build

- `git clone –recursive https://github.com/EssentialsOfParallelComputing/Chapter13 Chapter13`
- `cd Chapter13/Kokkos/StreamTriad`
- `cd Orig`
- Test serial version with `mkdir build && cd build; cmake ..; make; ./StreamTriad`
- `If run fails, try reducing the size of the arrays`
- Add to `CMakeLists.txt`

```
find_package(Kokkos REQUIRED)
target_link_libraries(StreamTriad Kokkos::kokkos)
```

- Retest with `cmake ..; make` and run `./StreamTriad` again
- Check Ver1 for solution. These modifications have already been made in this version.

Step 3: Add Kokkos views for memory allocation of arrays

Add include file

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

Add initialize and finalize

```
Kokkos::initialize(argc, argv);  {
} Kokkos::finalize();
```

Replace static array declarations with Kokkos views

```
int nsize=80000000;
Kokkos::View<double *> a( "a", nsize);
Kokkos::View<double *> b( "b", nsize);
Kokkos::View<double *> c( "c", nsize);
```

Rebuild and run

Kokkos Syntax: Initialization of Kokkos

- The first requirement for using Kokkos is to include a header file

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

- The next requirement is to initialize and finalize the Kokkos environment

```
Kokkos::initialize(argc, argv);
Kokkos::finalize();
```

- The initialize call should follow the MPI_Init call, if present, and should be near the start of the program
- You should add scope guards to these calls so that the memory that Kokkos allocates gets deallocated before the finalize call

```
Kokkos::initialize(argc, argv);
{
…
}
Kokkos::finalize();
```

Kokkos Syntax: Kokkos memory (views)

```
Kokkos::View<double *> x("data label", N0);
```

Data can be accessed with either x[i] or x(i)

Kokkos handles deallocation automatically

By default, Kokkos views are initialized. This can be overridden by adding an optional parameter.

```
Kokkos::View<double *> x (Kokkos::ViewAllocateWithoutInitializing
(label), N0);
```

You can also create an unmanaged view of a raw pointer, x_raw

```
Kokkos::View<double*, Kokkos::HostSpace,
  Kokkos::MemoryTraits<Kokkos::Unmanaged> > x_view (x_raw, N0);
```

Step 4: Add Kokkos execution pattern – parallel_for

Change for loops to Kokkos parallel fors.

- At start of loop

```
Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
```

- At end of loop, replace closing brace with

```
});
```

Rebuild and run. Add environment variables as Kokkos message suggests:

```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_PROC_BIND=true
```

How much speedup do you observe?

Step 5: Add Kokkos timers

Add Kokkos calls

```
Kokkos::Timer timer;
timer.reset(); // for timer start
time_sum += timer.seconds();
```

Remove

```
#include <timer.h>
struct timespec tstart;
cpu_timer_start(&tstart);
time_sum += cpu_timer_stop(tstart);
```

Completed version of Kokkos StreamTriad

```
#include <Kokkos_Core.hpp>

int main(int argc, char *argv[]){
   Kokkos::Timer timer;
   int nsize=80000000; int ntimes=16;
   double scalar = 3.0, time_sum = 0.0;

   Kokkos::initialize(argc, argv); {

   // initializing arrays
   Kokkos::View<double *> a( "a", nsize);
   Kokkos::View<double *> b( "b", nsize);
   Kokkos::View<double *> c( "c", nsize);

   Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
      a[i] = 1.0;
      b[i] = 2.0;
   });

   for (int k=0; k<ntimes; k++){
      timer.reset();
      // stream triad loop
      Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
         c[i] = a[i] + scalar*b[i];
      });
      time_sum += timer.seconds();
   }

   printf("Average runtime is %lf msecs\n", time_sum/ntimes*1000.0);

   }  Kokkos::finalize();
}
```

6. Run and measure performance with OpenMP

Find out how many virtual cores are on your CPU

```
lscpu
```

First run with a single processor:

- Average runtime _____

Then run the OpenMP version:

- Average runtime _____

Portability Exercises

1. Rebuild Stream Triad using Kokkos build with HIP

- Set Kokkos_DIR to point to external Kokkos build with HIP

```
export Kokkos_DIR=${HOME}/Kokkos_HIP/lib/cmake/Kokkos_HIP
cmake ..
make
```

2. Run and measure performance with AMD Radeon GPUs

- HIP build with ROCm 5.2.0+
- Ver4 - Average runtime is _____ msecs

3. Rebuild Stream Triad using Kokkos build with CUDA

- Set Kokkos_DIR to point to external Kokkos build with CUDA

```
export Kokkos_DIR=${HOME}/Kokkos_CUDA
cmake -DCMAKE_CXX_COMPILER=${Kokkos_DIR}/bin/nvcc_wrapper
make
```

4. Run and measure performance with Nvidia GPU

- CUDA build
- Ver4 - Average runtime is _____ msecs

Kokkos: performance profiling

Build kokkos tools

```
git clone https://github.com/kokkos/kokkos-tools kokkos-tools
cd kokkos-tools/src/tools/simple-kernel-timer
make
```

Run application with tool

```
./StreamTriad --kokkos-tools-library=<path to kokkos tools>/
    src/tools/simple-kernel-timer/kp_kernel_timer.so
```

or

```
KOKKOS_PROFILE_LIBRARY=<path to kokkos tools>/
  –   src/tools/simple-kernel-timer/kp_kernel_timer.so ./StreamTriad
```

Print out results of tool

```
<path_to_tool_directory>/kp_reader
```

Review

We covered:

- How to use an external Kokkos build (pre-built)
- How to add the Kokkos dependency to a cmake build
- How to initialize and finalize Kokkos in your application
- How to convert arrays to Kokkos views
- How to express simple loops in Kokkos parallel_for syntax

Shallow Water Application

Shallow Water application – the steps

1. Add Kokkos build to cmake
2. Add Kokkos initialization
3. Add Kokkos views
4. Add Kokkos parallel_for
5. Add Kokkos parallel_reduce
6. Swap views
7. Run and compare versions
8. Profile with Kokkos tools

Shallow Water application – retrieve and build

- git clone --recursive https://github.com/EssentialsOfParallelComputing/Chapter13 Chapter13
- cd Chapter13/Kokkos/ShallowWater
- `./build_[hip|cuda|openmp|serial]_version.sh`
  - Each of these versions will create a directory [cuda|hip|openmp|serial]_build
  - A kokkos version will be built for the “Execution Space”
  - The ShallowWater examples will be built and run
    - ShallowWater.cc – Original code
    - ShallowWater_par1.cc – Add Kokkos views
    - ShallowWater_par2.cc – Add Kokkos parallel_for
    - ShallowWater_par3.cc – Add Kokkos parallel_reduce
    - ShallowWater_par4.cc – Add Kokkos timers

Step 1: Add kokkos build to CMakeLists.txt

In-line build – this has already been done in the example code

```
git submodule add https://github.com/kokkos/kokkos Kokkos
```

- cd Kokkos
- git checkout 3.7.00
  – current release is 3.7
- cd ..
- Modify CMakeLists.txt

```
add_subdirectory(Kokkos)
target_link_libraries(ShallowWater_par1 Kokkos::kokkos -lm)
```

Step 2: Add Kokkos initialization

- Add kokkos include

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

- Remove graphics code
- Add kokkos initialize and finalize

```
Kokkos::initialize(argc, argv);  {
}Kokkos::finalize();
```

Step 3: Add Kokkos views

- **Change mallocs to Kokkos views**
  - **Change**

```
double** __restrict H = malloc2D(ny+2, nx+2);
```

to

```
Kokkos::View<double **> H("H", ny+2, nx+2);
```

  - Remove frees
  - Change `H[j][i]` to `H(j,i)`
    - replace `[ ]` with `,`
    - replace `[` with `(`
    - replace `]` with `)`
- Change SWAP_PTR to explicit loops

```
for(int j=1;j<=ny;j++){
  for(int i=1;i<=nx;i++){
    H(j,i) = Hnew(j,i);
    U(j,i) = Unew(j,i);
    V(j,i) = Vnew(j,i);
  }
}
```

Kokkos syntax: memory (views) - multidimensional arrays

```
Kokkos::View<double **> x("data label", N0, N1);
```

Data is then accessed with parentheses syntax (i,j)

Data layout can be controlled with an optional parameter, LayoutLeft or LayoutRight. Layout left is Fortran ordering and layout right is C/C++ ordering. By default, CudaSpace (includes HIP – really GPU) is LayoutLeft and OpenMP is LayoutRight.

```
Kokkos::View<double**, Kokkos::LayoutLeft> x ("x", N0, N1);
Kokkos::View<double**, Kokkos::CudaSpace> x ("x", N0, N1);
```

Step 4: Add Kokkos parallel_for

- Replace 2D nested loops – use MDRangePolicy

```
Kokkos::parallel_for("State Init",
Kokkos::MDRangePolicy<Kokkos::Rank<2>>({0,0},{ny+1,nx+1}),KOKKOS_LAMBDA(int j, int i){
});
```

- Replace 1D loops – use RangePolicy

```
Kokkos::parallel_for("BC", Kokkos::RangePolicy<>(1,ny+1), KOKKOS_LAMBDA(int j){
});
```

Step 5: Add Kokkos parallel_reduce

- Convert sum reductions to kokkos

```
origTM=0.0;
Kokkos::parallel_reduce("Sum Original Mass",
   Kokkos::MDRangePolicy<Kokkos::Rank<2>>({1,1},{ny+1,nx+1}),
   KOKKOS_LAMBDA(int j, int i, double &local_sum){
      local_sum+=H(j,i);
}, origTM);
```

- Convert to min reductions kokkos – note the change in the local loop variable

```
Kokkos::parallel_reduce("Calc DT",
    Kokkos::MDRangePolicy<Kokkos::Rank<2>>({1,1},{ny,nx}),
    KOKKOS_LAMBDA(int j, int i, double &local_deltaT){
       …
       double my_deltaT = sigma/(xspeed+yspeed);
       if (my_deltaT < local_deltaT) local_deltaT = my_deltaT;
}, Kokkos::Min<double>(deltaT));
```

Kokkos parallel_reduce

- Parallel reduction – sum of 1D array
  - needs a local variable for performing the sum - local_xsum
  - needs a result variable - xsum

```
double xsum = 0.0;
Kokkos::parallel_reduce("Sum", N, KOKKOS_LAMBDA (int i, double& local_xsum) {
  local_xsum += x(i);
  }, xsum);
```

- Parallel reduction – sum of 2D array

```
double xsum = 0.0;
Kokkos::parallel_reduce("Sum",Kokkos::MDRangePolicy<Kokkos::Rank<2>>({1,1},{ny+1,nx+1}),
  KOKKOS_LAMBDA (int j, int i, double& local_xsum) {
  local_xsum += x(j,i);
  }, xsum);
```

- Parallel reduction – minimum of 2D array

```
double xmin = 1.0e30;
Kokkos::parallel_reduce("Min",Kokkos::MDRangePolicy<Kokkos::Rank<2>>({1,1},{ny+1,nx+1}),
  KOKKOS_LAMBDA (int j, int i, double& local_xmin) {
  if (x(j,i) < local_xmin) local_xmin = x(j,i);
}, Kokkos::Min<double>(xmin);
```

Step 6. Swap views

- Replace copy of old to new with a swap of Kokkos views

```
#define SWAP_PTR(xnew,xold,xtmp) (xtmp=xnew, xnew=xold, xold=xtmp)

   Kokkos::View<double **>TempView;

   // Swapping views
   SWAP_PTR(H, Hnew, TempView);
   SWAP_PTR(U, Unew, TempView);
   SWAP_PTR(V, Vnew, TempView);
```

Kokkos syntax: shallow copies and deep copies

Kokkos does a shallow copy by default

- Shallow copy just assigns the pointer for the view and does not copy the data in the array

```
Kokkos::View<double *> xnew(“xnew”, 100);
Kokkos::View<double *> xold(“xold”, 100);
xnew = xold;// this is a “shallow copy”
```

- xnew now points to the xold view (array)

To do a deep copy, you have to specifically ask for it

```
Kokkos::View<double *> xnew(“xnew”, 100);
Kokkos::View<double *> xold(“xold”, 100);
Kokkos::deep_copy(xnew, xold);// this is a “deep copy”
```

- The data from the xold view is copied into the xnew view

Step 7. Run and compare versions

- Serial
  - Orig SWAP_PTR: Flow finished in _____ seconds
  - Copy new to old: Flow finished in _____ seconds – will be about 2x longer
- OpenMP
  - Flow finished in _____ seconds
- HIP
  - Flow finished in _____ seconds
- CUDA
  - Flow finished in _____ seconds

par5 with reimplementation of SWAP_PTR

- Serial - Flow finished in _____ seconds
- OpenMP - Flow finished in _____ seconds
- HIP - Flow finished in _____ seconds
- CUDA - Flow finished in _____ seconds

Step 8. Profile with Kokkos tools

Build all the kokkos tools

```
git clone https://github.com/kokkos/kokkos-tools kokkos-tools
cd kokkos-tools
sh build_all.sh
```

Run application with tool

```
./ShallowWater_par4 --kokkos-tools-library=<path to kokkos tools>/
      src/tools/simple-kernel-timer/kp_kernel_timer.so
  or
KOKKOS_PROFILE_LIBRARY=<path to kokkos tools>/
      src/tools/simple-kernel-timer/kp_kernel_timer.so ./ShallowWater_par4
```

Print out results from data collected with tool

```
<path_to_tool_directory/kp_reader
```

Output from kp_reader
Columns are Name, Total Time, # Calls, Time/Call, % of Kokkos Time, % of Total Time

```
>> kp_reader *.dat
Regions:

---------------------------------------------------------------------------
Kernels:

- Calc DT
 (ParRed)   0.094028 2001 0.000047 31.041607 28.491250
- Second Pass
 (ParFor)   0.070621 2000 0.000035 23.314244 21.398761
- BC
 (ParFor)   0.042495 4000 0.000011 14.028908 12.876303
- X Direction
 (ParFor)   0.032122 2000 0.000016 10.604346 9.733100
- Y Direction
 (ParFor)   0.031640 2000 0.000016 10.445353 9.587170
- Swap
 (ParFor)   0.031170 2000 0.000016 10.290059 9.444635
- Sum Mass
 (ParRed)   0.000609 20   0.000030 0.201024  0.184508
- Sum Original Mass
 (ParRed)   0.000060 1    0.000060 0.019835  0.018205
- ……
---------------------------------------------------------------------------
Summary:

Total Execution Time (incl. Kokkos + non-Kokkos):                0.33002 seconds
Total Time in Kokkos kernels:                                    0.30291 seconds
   -> Time outside Kokkos kernels:                               0.02711 seconds
   -> Percentage in Kokkos kernels:                                91.78 %
Total Calls to Kokkos Kernels:                                     14036

---------------------------------------------------------------------------
```

Review

Learned how:

- To handle multi-dimensional arrays
- How to do reductions with parallel_reduce
  - sum
  - minimum
- How to swap view pointers

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.


AMD