Experience with Kokkos for Lattice QCD Code Bridge++

Kokkos tea-time
at 7AM PST, 8AM MST, 10AM EST, 3PM UTC, 4PM CET, 0AM JST

February 19(20), 2025

Keigo Nitadori　似鳥 啓吾
Technical Scientist, RIKEN R-CCS
Software Development Technology Unit

Abstract

Development Experience with Kokkos
(Practical insights from recent months)

- **First time using the Kokkos framework**
  - Prior experience with C++ (templates, lambdas) and CUDA
- **Target**
  - Mini benchmark code from Bridge++
  - Community lattice QCD code from Japan
- **Focus**
  - Domain-wall Fermions (5D spins in 4D gauge)
  - Even-odd decomposition
- **Progress**
  - Implemented ”mult” kernel in fp32 (solver implementation pending)
- **Notes**
  - Insights are based on practical development experience rather than deep expertise

1. Introduction: Hello Kokkos
2. Bridge++: Lattice QCD code
3. Performance and conclusion

Motivation/Background

- **Accelerators were not applied for K (2012–2019) or Fugaku (2020–)**
  - Priolizerd the ecosystem compatibility of application codes
  - People don’t like to maintain a diverged source code for accelerators
- **Seeking platforms for unified code development**
  - Directive-based: OpenACC, OpenMP
  - Modern C++ library: Kokkos
  - Balancing Porting cost and performance

「京」K computer (2012–2019)


「富岳」Fugaku (2020–)

What is Kokkos?

- **Overview**
  - A modern C++ library for ***Performance Portability***
  - Designed for high-performance computing (HPC) applications
- **Key Features**
  - Abstraction for multi-platform parallelism (e.g., CPUs, GPUs)
  - Supports multiple backends: OpenMP, CUDA, HIP, etc.
  - Enables unified code for diverse hardware architectures
- **My Perspective**
  - Exploring Kokkos for the first time (not an evangelist)
  - Balancing performance and portability
- **Insights**
  - Mostly for writing new code, not for modifying existing codes
  - Requires explicit host-device data transfer
  - Demands C++ expertise (e.g., lambda and value/reference capture)
  - Free from vendor's proprietary statements in the users' codes

Hello world in Kokkos

In CUDA

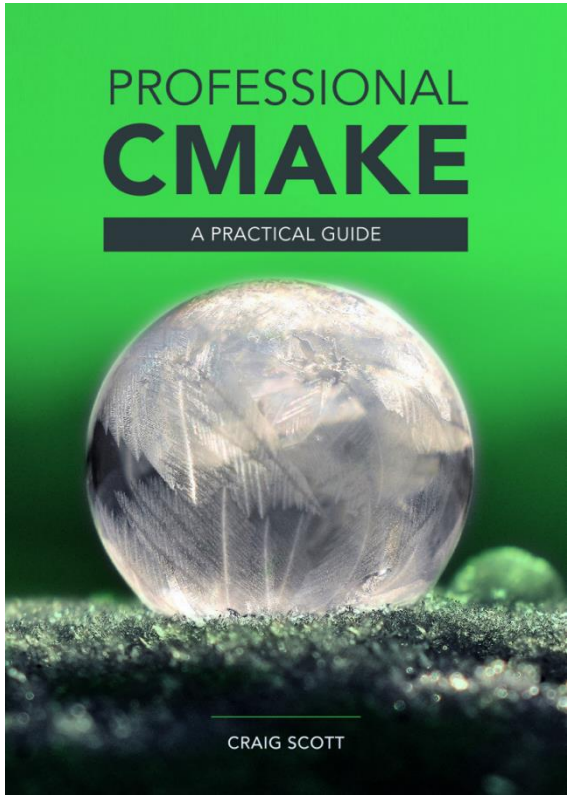
Getting started with CUDA is simple:
a few lines of code and one command

```
keigo@gpu1:~$ nvcc hello.cu
keigo@gpu1:~$ ./a.out
Hello, world!
keigo@gpu1:~$ _
```

- The first and the greatest barrier to Kokkos is **CMAKE**
  - By nature, Kokkos is designed to support multiple platforms
  - If there were `kokkoscc` command,,,
- Starting with Kokkos often feels like mastering a second tool—**CMAKE**—before writing actual code

In Kookkos

With Kokkos, learning CMake becomes an essential step—even for minimal examples



Best 40 dollars you will spend

Minimal startup of Kokkos with CMAKE

- **The common.cmake file in the kokkos-tutorials repository turned out to be helpful**

https://github.com/kokkos/kokkos-tutorials/blob/main/Exercises/common.cmake

```
cmake_minimum_required(VERSION 3.16)
project(caxpy)
include(../common.cmake)

add_executable(caxpy caxpy.cpp)
target_link_libraries(caxpy Kokkos::kokkos)

set(CMAKE_VERBOSE_MAKEFILE ON)
```

Minimum CMakeLists.txt

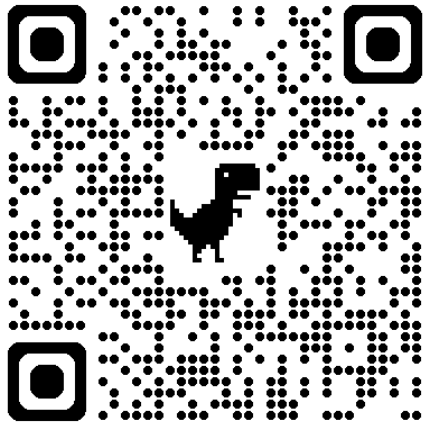
- Kokkos' source is downloaded from GitHub when you build first time
- In each build directory, e.g., build_openmp or build_cuda, Kokkos-runtime is built as a dependent library

```
$ cmake -B build_cuda
      -DKokkos_ENABLE_CUDA=ON
      -DKokkos_ARCH_HOPPER90=ON
$ make –C build_cuda -j
```

Build your C++ code in CUDA configuration together with **libkokkoscore.a** et al.

New Version of Quick Start

- **Simplest example of CMakeLists.txt in https://kokkos.org/kokkos-core-wiki/quick_start.html**



```
cmake_minimum_required(VERSION 3.16)
project(MyProject)

include(FetchContent)
FetchContent_Declare(
  Kokkos
  URL https://github.com/kokkos/kokkos/archive/refs/tags/4.5.01.zip
)
FetchContent_MakeAvailable(Kokkos)

add_executable(HelloKokkos HelloKokkos.cpp)
target_link_libraries(HelloKokkos Kokkos::kokkos)
```

Just type

```
$ cmake -B build_openmp -DKokkos_ENABLE_OPENMP=ON
$ make –C build_openmp -j
```

Case of using Spack

- **Application will be configured as the configuration of the Kokkos found in Spack**

```
[keigo@qc-gh200-01 ~]$ spack find -lx
-- linux-rocky9-neoverse_v2 / gcc@11.5.0 ----------------------
2cnzhf4 gh@2.58.0  mqw6c46 kokkos@4.4.01  qpcwipg kokkos@4.4.01
==> 3 installed packages
```

```
$ spack load kokkos+openmp
$ cmake –B build_openmp
$ spack unload kokkos
$ spack load kokkos+cuda
$ cmake –B build_cuda
$ spack unload kokkos
```

Kokkos packages in Spack only need to be loaded when you run cmake -B

```
cmake_minimum_required(VERSION 3.16)
project(MyProject)

find_package(Kokkos REQUIRED)
add_executable(HelloKokkos HelloKokkos.cpp)
target_link_libraries(HelloKokkos Kokkos::kokkos)
```

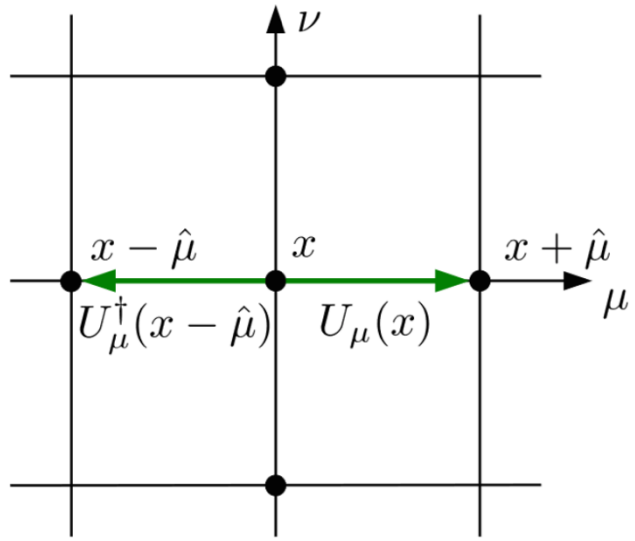
Minimum CmakeFiles.txt

1. Introduction: Hello Kokkos
2. **Bridge++: Lattice QCD code**
3. Performance and conclusion

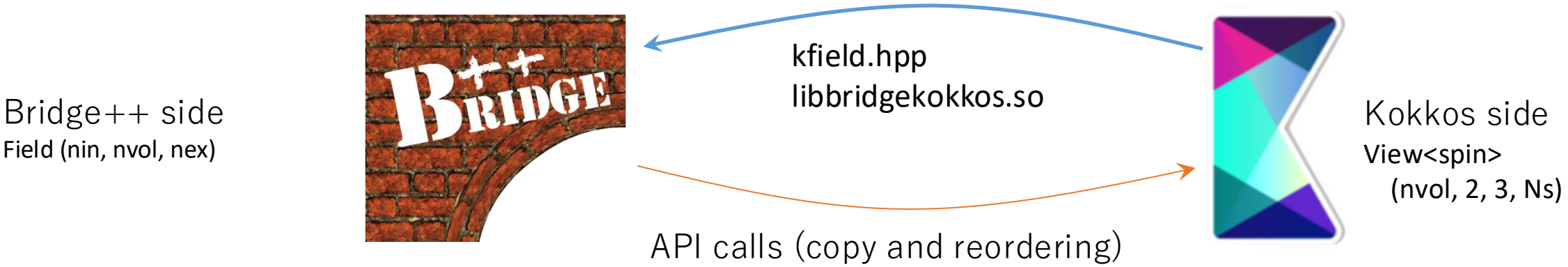
Bridge++: Community Lattice QCD code

https://bridge.kek.jp/Lattice-code/index_e.html

- **An object-oriented LQCD simulation code in C++**
  - Development started in 2009, with version 1.0 released in 2012, and 2.0 in February 2024
  - Developed within the Japanese high-energy physics community
  - Experimental accelerator branch in OpenACC and CUDA
- **Mini benchmark code on Domainwall Fermion kernel**
  - The fermion (or spinor or quark) field has a 5th dimension to keep chiral symmetry well
  - 4-dimensional SU(3) gauge field as link variables
  - Benchmarking matrix (gauge) vector (spinor) multiplication performance



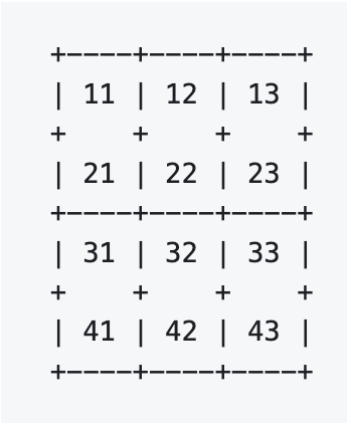
Connecting Two Worlds Through Modular Design



- **We need to connect two C++ codes with various class and macro definitions for each.**
  - Impractical to let Bridge++ include Kokkos headers.
  - Use kernel programs in Kokkos as a ***plug-in***
  - The module serves Bridge++ only two files: Kfield.hpp and libbridgekokkos.so
    - Employed pImpl idiom for this purpose
    - 1 line change in CMakeLists.txt, to switch a.out to liba.so

Data structure

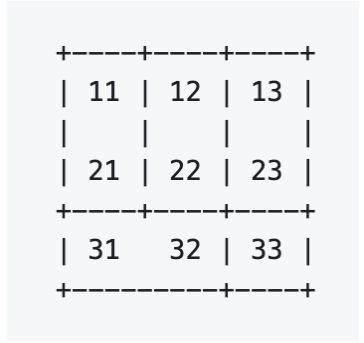
- **Best memory performance on GPU is achieved by *coalesced access* of float4 datatype**
  - SoA (structure of arrays) is a natural layout for GPUs
  - But fixed size structure in 8- or 16-byte enables more efficiency
    - Smaller number of streams and access locality
    - cuobjdump to confirm the 16-byte load/store instruction
- **We assigned 2-spinor (pair of complex<float>) to the float4 datatype**
  - Spin: 3 colors, upper/lower 2-spinor, 6 streams
  - Gauge: 3 or 4 streams + reconstruction



Spinor:
- 3 color, 4 complex (dirac)
- Ns for the 5th dimension

View<TwoSpin****> spin("spin", nvol/2, 2, 3, Ns);

(sizeof(TwoSpin) == 16)
(Fortran like ordering)



Gauge:
- 3×3 complex matrix, 4 directions
- 3rd row can be reconstructed

View<TwoSpin****> gauge("gauge", nvol/2, 4, 5, 2);

Nvol = Nx × Ny × Nz × Nt
nvol/2 for even-odd decomposition

Even odd decomposition

Lexical

3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7

Even

3,1	3,3	3,5	3,7
2,0	2,2	2,4	2,6
1,1	1,3	1,5	1,7
0,0	0,2	0,4	0,6

Odd

3,0	3,2	3,4	3,6
2,1	2,3	2,5	2,7
1,0	1,2	1,4	1,6
0,1	0,3	0,5	0,7

$$\begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix} \begin{pmatrix} x_e \\ x_o \end{pmatrix} = \begin{pmatrix} b_e \\ b_o \end{pmatrix}$$

$$\underbrace{\left(1 - D_{ee}^{-1} D_{eo} D_{oo}^{-1} D_{oe}\right)}_{\equiv D} x_e = D_{ee}^{-1} \left(b_e - D_{eo} D_{oo}^{-1} b_o\right)$$

$$x_o = D_{oo}^{-1} \left(b_o - D_{oe} x_e\right)$$

- Split the lattice points into two parts
  - D_{ee}, D_{oo}: Self (site local) operations
  - D_{eo}, D_{oe}: Stencil operations
- A new D matrix is defined to be solved
  - Nearly the same cost as the original, but rapid convergence

Kernel properties

- **$D^\dagger D$ operation for CG solver consists of two different types of kernels**
  - **Stencil kernel**
    - Called 4 times
    - Spatial hopping in 4D spacetime, as in the usual Wilson-type kernel
    - Needs halo exchange in MPI environment (not implemented in this study)
    - Potential reusability of data (no explicit shared memory tuning is applied)
  - **Stream kernel**
    - Called 5 times, before and after the stencil kernels
    - Site-local, element-by-element operations for the 5th dimension
    - No data reusability (good benchmark for the bandwidth)
    - 3x 1R/1W, 1x 2R/1W, 1x 2R/2W

Even-odd decompisition

$$\begin{pmatrix} b_e \\ b_o \end{pmatrix} = \begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix} \begin{pmatrix} x_e \\ x_o \end{pmatrix}$$

It is transformed to solve

$$D = 1 - {D_{ee}}^{-1} D_{eo} {D_{oo}}^{-1} D_{oe}$$

Diagonal D_{ee} and D_{oo} are tridiagonal matrices of size N_s, of which inverse is available in $O(N_s)$ cost

Four-dimensional Wilson kernel

$$D_{x,y} = [1 + F(x)]\delta_{x,y} - \kappa \sum_{\mu=1}^{4} \left[(1-\gamma_\mu) U_\mu(x) \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U_\mu^\dagger(x-\hat{\mu}) \delta_{x-\hat{\mu},y} \right],$$

Use of template for on-register kernels

- **GPU compilers can assign registers for fixed-size small arrays, like float x[3];**
- **But users don't like the lattice size parameters fixed as compile-time constants**
  - Nx, Ny, Nz, Nt, Ns
  - At least in the C++ era
- **I set finite numbers of *presets* and chose chose appropriate number at runtime**
  - Assigned 12 threads per lattice point, for 3 colors, and 4 real numbers in 2-spinor are independent in the stream kernels

```
struct Fops5D_Base;

template <int Ns>
struct Fops5D_dirac : public Fops5D_Base {
  struct Kernel{
    float upper[Ns];
    float lower[Ns];
  };
};

constexpr int Ns_presets[] = {8, 10, 12};
constexpr int Len_presets  = sizeof(Ns_presets)
                           / sizeof(Ns_presets[0]);

// Recursive template
template <int I>
Fops5D_Base *new_Fops5D_dirac(int Ns){
  if(Ns == Ns_presets[I]){
    fprintf(stderr, "Ns=%d found in presets ", Ns);
    return new Fops5D_dirac<Ns_presets[I]>;
  }else{
    return new_Fops5D_dirac<I+1>(Ns);
  }
}

// Terminate the recursion
Template <>
Fops5D_Base *new_Fops5D_dirac<Len_presets>(int Ns){
  fprintf(stderr, "Ns=%d NOT found in presets ", Ns);
  return new Fops5D_Base;
}
```

1. Introduction: Hello Kokkos
2. Bridge++: Lattice QCD code
3. Performance and conclusion

Performance (GH200)

- **32×32×32×32(×8) Lattice**
- **6.7 TFLOPS (Bridge++ output)** [peak: 67 TFLOPS]
- **2.4 TB/s** [peak: 4TB/s]
- **0.23 sec for Stencil and 0.21 sec for Stream kernels**

```
Elapsed time: mult: total       0.44 sec, count    1, average       0.44 sec
  FLOP per mult : 29796335616.000000
  performance  : 6741.344648 GFlops
  elapsed      : 4.419940e-03 sec/iter
Elapsed time: DdagD: total      0.44 sec, count    1, average       0.44 sec
            DdagD : 4.420481e-03 sec/iter
Elapsed time: DdagD_dummy1: total       0.23 sec, count    1, average       0.23 sec
       DdagD_dummy1 : 2.298491e-03 sec/iter
Elapsed time: DdagD_dummy2: total       0.21 sec, count    1, average       0.21 sec
       DdagD_dummy2 : 2.147071e-03 sec/iter
Bandwidth : 2.437968 Tbyte/s
```

Performance (MI250)

- After developing in the NVIDIA environment, the benchmark code ran without any modification
- **0.8 TFLOPS** [peak: 23 TFLOPS]
- **0.30 TB/s** [peak: 1.6 TB/s]
- Still, the stencil and stream kernels are well-balanced

```
Elapsed time: mult: total       3.65 sec, count    1, average       3.65 sec
  FLOP per mult : 29796335616.000000
  performance  : 816.755729 GFlops
  elapsed      : 3.648133e-02 sec/iter
Elapsed time: DdagD: total      3.70 sec, count    1, average       3.70 sec
            DdagD : 3.698752e-02 sec/iter
Elapsed time: DdagD_dummy1: total       2.01 sec, count    1, average       2.01 sec
       DdagD_dummy1 : 2.012270e-02 sec/iter
Elapsed time: DdagD_dummy2: total       1.70 sec, count    1, average       1.70 sec
       DdagD_dummy2 : 1.704034e-02 sec/iter
Bandwidth : 0.307182 Tbyte/s
```

Insights and Reflections

- **Challenges, or steep learning curve**
  - CMAKE: a new environment to me inevitable for Kokkos
  - pImpl: connecting two worlds with minimum dependency
- **Currently remaining problem**
  - Poor performance on AMD, Kokkos, HIP, or hardware issue?
- **Developing with Chat GPT-4o**
  - He/she knows CMAKE, HIP, CUDA, Kokkos, and C++
  - Generated codes were not always perfect but I enjoyed the conversations
  - It is essential to refer to the ***primary source*** finally, but generative AI is beneficial as a ***reverse lookup*** tool

Supplemental materials for implementation detail

Gamma matrices and spin projections

Dirac (energy) form:

$$\gamma_k = \sigma_2 \otimes \sigma_k = \begin{pmatrix} 0 & -i\sigma_k \\ i\sigma_k & 0 \end{pmatrix} \quad (k = 1, 2, 3)$$

$$\gamma_4 = \sigma_3 \otimes 1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\gamma_5 = \sigma_1 \otimes 1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Weyl (chiral) form:

$$\gamma_4 = -\sigma_1 \otimes 1 = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$$

$$\gamma_5 = \sigma_3 \otimes 1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Spin projections:

$$1 + \gamma_k = \begin{pmatrix} 1 & -i\sigma_k \\ i\sigma_k & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ i\sigma_k \end{pmatrix} \begin{pmatrix} 1 & -i\sigma_k \end{pmatrix}$$

$$1 - \gamma_k = \begin{pmatrix} 1 & i\sigma_k \\ -i\sigma_k & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -i\sigma_k \end{pmatrix} \begin{pmatrix} 1 & i\sigma_k \end{pmatrix}$$

For implementers:

$$i\sigma_1 = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} : (a, b, c, d) \to (-d, c, -b, a)$$

$$i\sigma_2 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} : (a, b, c, d) \to (-c, -d, a, b)$$

$$i\sigma_3 = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} \mapsto \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix} : (a, b, c, d) \to (b, -a, -d, c)$$