Kokkos-FFT: a newcomer library in the Kokkos ecosystem

Yuuichi Asahi[1], Paul Zehner[1], Baptiste Legouix[2], Thomas Padioleau[1], Julien Bigot[1]

1. Maison de la Simulation

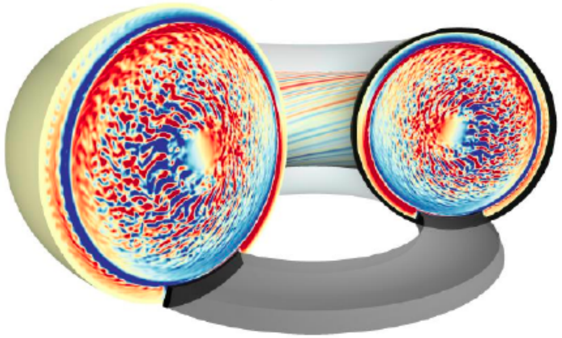
2. CEA

https://github.com/kokkos/kokkos-fft

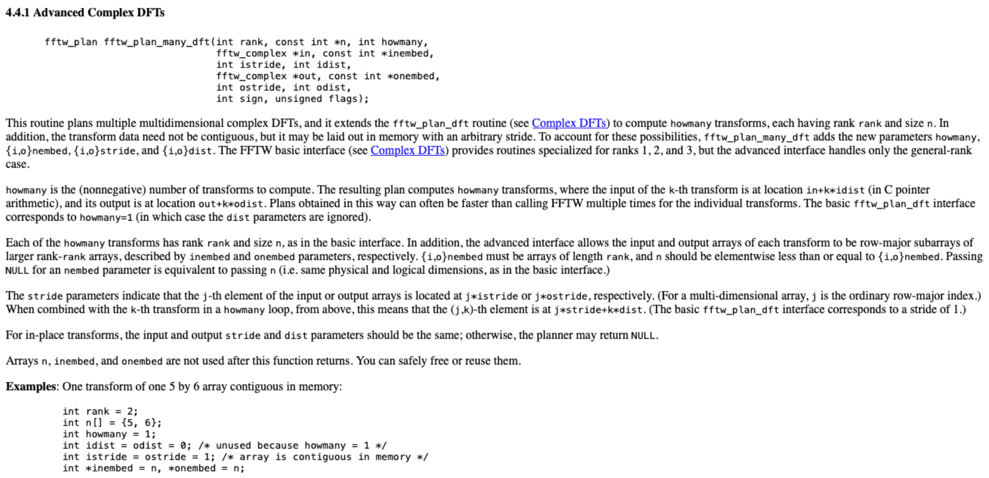
Why KokkosFFT, who needs that?

- Using Kokkos to port a legacy application which relies on FFT libraries
  - Fluid simulation codes with periodic boundaries, **Plasma turbulence**, etc
- Having a Kokkos code and willing to integrate in-situ data processing with FFTs
  - **Spectral analyses**, etc.
- **NOT willing** to get through documentations of de facto standard FFT libraries
  - Benefit from powerful FFT libraries as simple as **numpy.fft**

GYSELA-X (plasma turbulence)
Periodic along toroidal direction



FFTW Documentation



4.4.1 Advanced Complex DFTs

```
fftw_plan fftw_plan_many_dft(int rank, const int *n, int howmany,
                             fftw_complex *in, const int *inembed,
                             int istride, int idist,
                             fftw_complex *out, const int *onembed,
                             int ostride, int odist,
                             int sign, unsigned flags);
```

This routine plans multiple multidimensional complex DFTs, and it extends the `fftw_plan_dft` routine (see Complex DFTs) to compute `howmany` transforms, each having rank `rank` and size `n`. In addition, the transform data need not be contiguous, but it may be laid out in memory with an arbitrary stride. To account for these possibilities, `fftw_plan_many_dft` adds the new parameters `howmany`, `{i,o}nembed`, `{i,o}stride`, and `{i,o}dist`. The FFTW basic interface (see Complex DFTs) provides routines specialized for ranks 1, 2, and 3, but the advanced interface handles only the general-rank case.

`howmany` is the (nonnegative) number of transforms to compute. The resulting plan computes `howmany` transforms, where the input of the `k`-th transform is at location `in+k*idist` (in C pointer arithmetic), and its output is at location `out+k*odist`. Plans obtained in this way can often be faster than calling FFTW multiple times for the individual transforms. The basic `fftw_plan_dft` interface corresponds to `howmany=1` (in which case the `dist` parameters are ignored).

Each of the `howmany` transforms has rank `rank` and size `n`, as in the basic interface. In addition, the advanced interface allows the input and output arrays of each transform to be row-major subarrays of larger rank-`rank` arrays, described by `inembed` and `onembed` parameters, respectively. `{i,o}nembed` must be arrays of length `rank`, and `n` should be elementwise less than or equal to `{i,o}nembed`. Passing `NULL` for an `nembed` parameter is equivalent to passing `n` (i.e. same physical and logical dimensions, as in the basic interface.)

The `stride` parameters indicate that the `j`-th element of the input or output arrays is located at `j*istride` or `j*ostride`, respectively. (For a multi-dimensional array, `j` is the ordinary row-major index.) When combined with the `k`-th transform in a `howmany` loop, from above, this means that the (`j`,`k`)-th element is at `j*stride+k*dist`. (The basic `fftw_plan_dft` interface corresponds to a stride of 1.)

For in-place transforms, the input and output `stride` and `dist` parameters should be the same; otherwise, the planner may return `NULL`.

Arrays `n`, `inembed`, and `onembed` are not used after this function returns. You can safely free or reuse them.

Examples: One transform of one 5 by 6 array contiguous in memory:

```
int rank = 2;
int n[] = {5, 6};
int howmany = 1;
int idist = odist = 0; /* unused because howmany = 1 */
int istride = ostride = 1; /* array is contiguous in memory */
int *inembed = n, *onembed = n;
```

kokkos-fft

numpy.fft

```
xr2c_hat = np.rfft(xr2c, axis=-1)
```

↓

kokkos-fft

```
KokkosFFT::rfft(exec, xr2c, xr2c_hat, /*axis=*/-1);
```

Key features of Kokkos-fft

As simple as numpy.fft, as fast as vendor libraries

- 1D, 2D, 3D standard and real Fast Fourier Transforms over 1D to 8D Kokkos Views
  - Batched plans are automatically used if View Dim > FFT Dim
- Simple interfaces like **numpy.fft** (out-place only)
  - View is all we need: No need to access the complicated FFT APIs
- Supporting multiple CPU and GPU backends (FFTs are executed on the stream/queue used in the Execution space)
  - **SERIAL, THREADS, OPENMP, CUDA, HIP and SYCL**
- Supported data types: float, double and Kokkos::complex
  - Limited to contiguous layout only: **LayoutLeft and LayoutRight**
  - **DefaultExecutionSpace and DefaultHostExecutionSpace supported**

APIs (numpy.fft + FFT Plan)

```cpp
constexpr int n0 = 128, n1 = 128, n2 = 16;

// 1D batched R2C FFT
View3D<double> xr2c("xr2c", n0, n1, n2);
Kokkos::Random_XorShift64_Pool<> random_pool(12345);
Kokkos::fill_random(xr2c, random_pool, 1);

View3D<Kokkos::complex<double> > xr2c_hat("xr2c_hat", n0, n1, n2/2+1);
KokkosFFT::rfft(exec_space(), xr2c, xr2c_hat, KokkosFFT::Normalization::Backward, -1);
```

```python
import numpy as np

n0, n1, n2 = 128, 128, 16

# 1D batched R2C FFT
xr2c = np.random.rand(n0, n1, n2)


xr2c_hat = np.fft.rfft(xr2c, axis=-1)
```

APIs

- KokkosFFT::<func> equivalent to numpy.fft.<func>
- Namespaces: KokkosFFT (APIs) and KokkosFFT::Impl (implementation details)
- Macros: KOKKOSFFT_*

Implementations

- Internally, (maybe) transpose + FFT plan creation + FFT execution + normalization
- Errors if there is inconsistency between exec-space and Views
- FFT plans can be reused (important for cufft and rocfft)

Implementation details

Class to handle FFT Plans

```
explicit Plan(const ExecutionSpace& exec_space, InViewType& in,
              OutViewType& out, KokkosFFT::Direction direction, int axis)
    : m_fft_size(1), m_is_transpose_needed(false), m_direction(direction) {
  static_assert(Kokkos::is_view<InViewType>::value,
                "Plan::Plan: InViewType is not a Kokkos::View.");
  static_assert(Kokkos::is_view<OutViewType>::value,
                "Plan::Plan: OutViewType is not a Kokkos::View.");
  static_assert(
      KokkosFFT::Impl::is_layout_left_or_right_v<InViewType>,
      "Plan::Plan: InViewType must be either LayoutLeft or LayoutRight.");
  static_assert(
      KokkosFFT::Impl::is_layout_left_or_right_v<OutViewType>,
      "Plan::Plan: OutViewType must be either LayoutLeft or LayoutRight.”);


  …


  m_axes                       = {axis};
  m_in_extents                 = KokkosFFT::Impl::extract_extents(in);
  m_out_extents                = KokkosFFT::Impl::extract_extents(out);
  std::tie(m_map, m_map_inv) = KokkosFFT::Impl::get_map_axes(in, axis);
  m_is_transpose_needed        = KokkosFFT::Impl::is_transpose_needed(m_map);
  m_fft_size = KokkosFFT::Impl::_create(exec_space, m_plan, in, out, m_buffer,
                                        m_info, direction, m_axes);
}
```

fft APIs (alias to _fft)

```
template <typename ExecutionSpace, typename InViewType, typename OutViewType>
void fft(const ExecutionSpace& exec_space, const InViewType& in,
         OutViewType& out,
         KokkosFFT::Normalization norm = KokkosFFT::Normalization::backward,
         int axis = -1, std::optional<std::size_t> n = std::nullopt) {
  static_assert(Kokkos::is_view<InViewType>::value,
                "fft: InViewType is not a Kokkos::View.");
  static_assert(Kokkos::is_view<OutViewType>::value,
                "fft: OutViewType is not a Kokkos::View.");


  …


  KokkosFFT::Impl::Plan plan(exec_space, _in, out,
                             KokkosFFT::Direction::forward, axis);
  if (plan.is_transpose_needed()) {
    InViewType in_T;
    OutViewType out_T;

    KokkosFFT::Impl::transpose(exec_space, _in, in_T, plan.map());
    KokkosFFT::Impl::transpose(exec_space, out, out_T, plan.map());

    KokkosFFT::Impl::_fft(exec_space, plan, in_T, out_T, norm);

    KokkosFFT::Impl::transpose(exec_space, out_T, out, plan.map_inv());

  } else {
    KokkosFFT::Impl::_fft(exec_space, plan, _in, out, norm);
  }
}
```

- _create() and _fft() are defined for each Kokkos device backend (wrappers for FFT libs)
- In _create(), an appropriate FFT plan is created based on Views and axis
- In _fft(), appropriate FFT execution functions are called

Build systems (dependency)

CMake option	Description	Backend FFT library	Compilers
Kokkos_ENABLE_SERIAL	Serial backend targeting CPUs	fftw (serial)	gcc/icpx
Kokkos_ENABLE_THREADS	C++ threads backend targeting CPUs	fftw (threads)	gcc/icpx
Kokkos_ENABLE_OPENMP	OpenMP backend targeting CPUs	fftw (openmp)	gcc/icpx
Kokkos_ENABLE_CUDA	CUDA backend targeting NVIDIA GPU	cufft	nvcc/(nvc++)
Kokkos_ENABLE_HIP	HIP backend targeting AMD GPUs	hipfft/rocfft	hipcc
Kokkos_ENABLE_SYCL	SYCL backend targeting Intel GPUs	oneMKL	icpx

- gcc 8.3.0+ (CPUs), IntelLLVM 2023.0.0+ (Intel GPUs), nvcc 12.0.0+ (NVIDIA GPUs), hipcc 5.3.0+ (AMD GPUs)
- CMake 3.22+ and Kokkos 4.2.0+
- Include as a subdirectory or use as an installed library (spack not ready)
- CMake options are **KokkosFFT_ENABLE_<something>**
- With KokkosFFT_ENABLE_HOST_AND_DEVICE=ON, KokkosFFT APIs can be called from both host and device. (fftw needed)
- For HIP backend, we use hipfft as a default (rocfft is optional).

Quick start (as subdirectory)

- CMake project

```
---/
 |
 └──<project_directory>/
    |--tpls
    |     |--kokkos/
    |     └──kokkos-fft/
    |--CMakeLists.txt
    └──hello.cpp
```

- Compile (for A100 GPU)

```
cmake -B build \
      -DCMAKE_CXX_COMPILER=g++ \
      -DCMAKE_BUILD_TYPE=Release \
      -DKokkos_ENABLE_CUDA=ON \
      -DKokkos_ARCH_AMPERE80=ON
cmake --build build -j 8
```

- CMakeLists.txt

```
cmake_minimum_required(VERSION 3.23)
project(kokkos-fft-as-subdirectory LANGUAGES CXX)

add_subdirectory(tpls/kokkos)
add_subdirectory(tpls/kokkos-fft)

add_executable(hello-kokkos-fft hello.cpp)
target_link_libraries(hello-kokkos-fft PUBLIC Kokkos::kokkos KokkosFFT::fft)
```

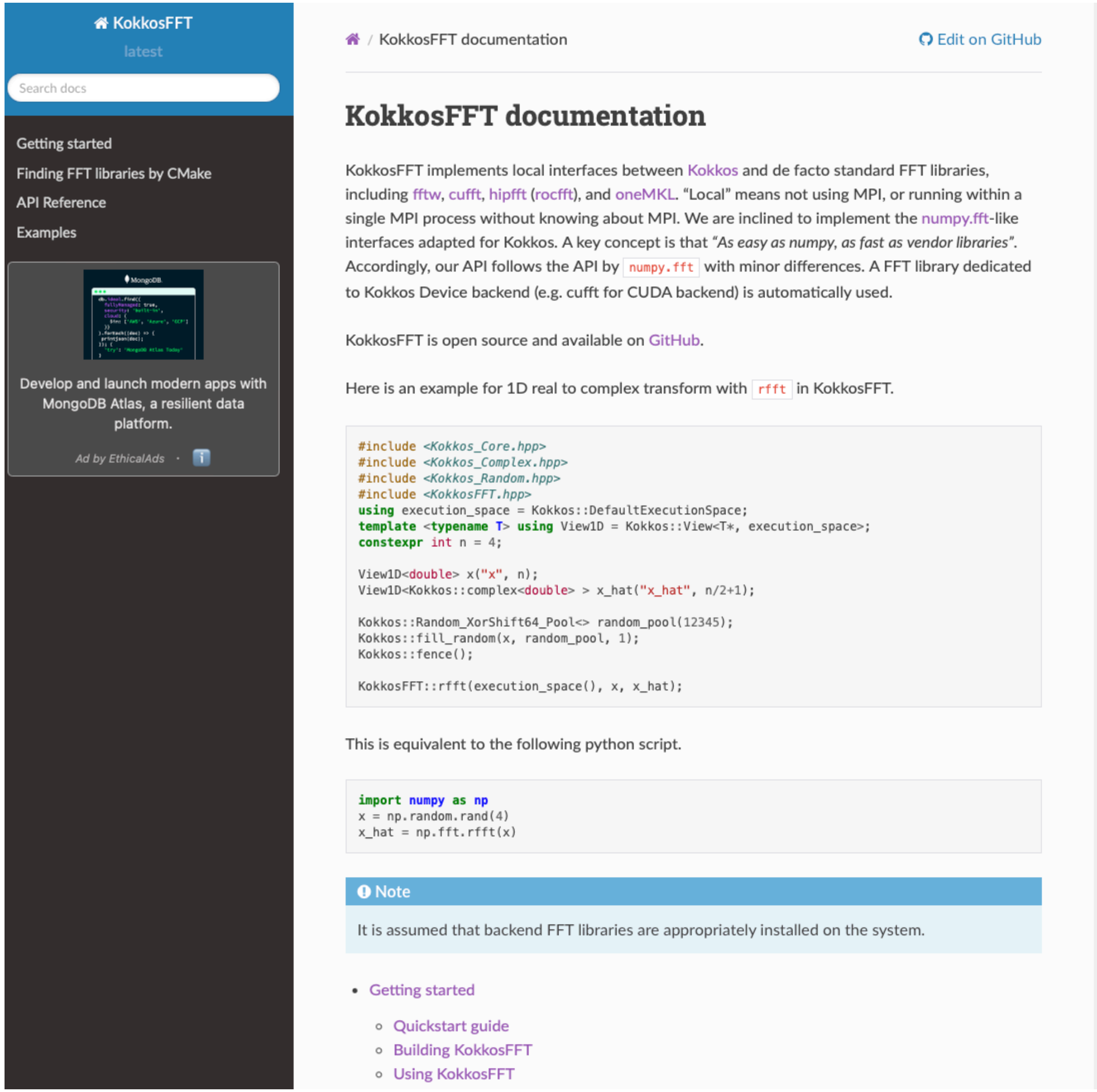
CI/CD (GitHub Actions)

github actions (docker + singularity)

Device	SERIAL	THREADS	OPENMP	CUDA	HIP	SYCL
Build tests	X	X	X	X	X	X
Install tests	X	X	X	X	X	X
Run tests	X	X	X	X		

- Build tests: tests, examples, and benchmarks
- Install tests: Using Kokkos-fft as a library or a subdirectory in CMake project
- Run tests: Mostly 1D-3D FFTs on Views for different precision and layout (282 tests)
- For CUDA, run tests are performed on our internal server (**self-hosted runner**)
- For HIP/SYCL, run tests are made manually on Frontier and Intel PVC testbed
- For SYCL, relying on oneapi-basekit with -DKokkos_ENABLE_SYCL=ON -DKokkos_ARCH_INTEL_GEN=ON (does not compile with PVC on container)
- Issue to build with nvc++ inside Nvidia hpc sdk container (can compile on our server)

Documentations (readthedocs)


KokkosFFT
latest
Search docs

Getting started
Finding FFT libraries by CMake
API Reference
Examples

Develop and launch modern apps with MongoDB Atlas, a resilient data platform.
Ad by EthicalAds

KokkosFFT documentation   Edit on GitHub

KokkosFFT documentation

KokkosFFT implements local interfaces between Kokkos and de facto standard FFT libraries, including fftw, cufft, hipfft (rocfft), and oneMKL. "Local" means not using MPI, or running within a single MPI process without knowing about MPI. We are inclined to implement the numpy.fft-like interfaces adapted for Kokkos. A key concept is that *"As easy as numpy, as fast as vendor libraries"*. Accordingly, our API follows the API by `numpy.fft` with minor differences. A FFT library dedicated to Kokkos Device backend (e.g. cufft for CUDA backend) is automatically used.

KokkosFFT is open source and available on GitHub.

Here is an example for 1D real to complex transform with `rfft` in KokkosFFT.

```cpp
#include <Kokkos_Core.hpp>
#include <Kokkos_Complex.hpp>
#include <Kokkos_Random.hpp>
#include <KokkosFFT.hpp>
using execution_space = Kokkos::DefaultExecutionSpace;
template <typename T> using View1D = Kokkos::View<T*, execution_space>;
constexpr int n = 4;

View1D<double> x("x", n);
View1D<Kokkos::complex<double> > x_hat("x_hat", n/2+1);

Kokkos::Random_XorShift64_Pool<> random_pool(12345);
Kokkos::fill_random(x, random_pool, 1);
Kokkos::fence();

KokkosFFT::rfft(execution_space(), x, x_hat);
```

This is equivalent to the following python script.

```python
import numpy as np
x = np.random.rand(4)
x_hat = np.fft.rfft(x)
```

Note

It is assumed that backend FFT libraries are appropriately installed on the system.

- Getting started
  - Quickstart guide
  - Building KokkosFFT
  - Using KokkosFFT

Doxygen: Docstrings for C++ code

Sphinx: Generate html

Breathe: conf.py to build sphinx using generated doc strings

Contents

- Getting started
  - Quickstart, building, using
- Finding FFT libraries by CMake
- API Reference
- Examples
  - KokkosFFT and python

Summary/Future plans

1D batched transform needed for GYSELA-X code (FFT along toroidal direction)

$$(N_r, N_\theta, N_\varphi) = (256, 256, 256)$$

Device	Icelake (36 cores)	A100	MI250X (1 GCD)	PVC
axis=0	60.7 [ms] (transposed)	37.1 [ms]	1.1 [ms]	3.99 [ms]
axis=1	60.0 [ms] (transposed)	38.1 [ms] (transposed)	2.9 [ms] (transposed)	7.39 [ms] (transposed)
axis=2	6.56 [ms]	38.4 [ms] (transposed)	4.2 [ms] (transposed)	11.67 [ms] (transposed)

- Summary
  - Kokkos-fft: performance portable FFT for Kokkos as easy as numpy
- Issues/Future plans
  - Distributed plans with MPI support
  - In-place transform support
  - Optimization (memory, performance)