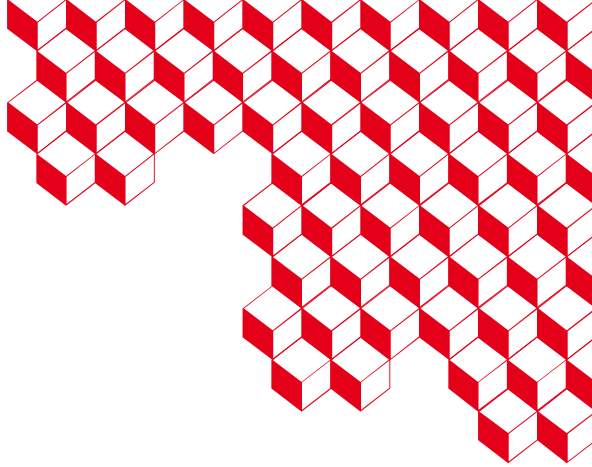


KokkosFFT: Performance-Portable Fast Fourier Transform interface for Kokkos Applications

Yuuichi Asahi, Trévis Morvany, Thomas Padioleau, Julien Bigot

Maison de la Simulation, CEA, France

Outline

- Introduction
  - Use case of distributed FFTs in simulations
  - FFT libraries and Kokkos performance portable ecosystem
- Performance portable solution for distributed FFTs
  - kokkos-fft: Interface between Kokkos and vendor libraries
  - APIs and functionalities of distributed version
  - Using in 3D Navier Stokes solver
  - Performance analyses
- Summary

Outline

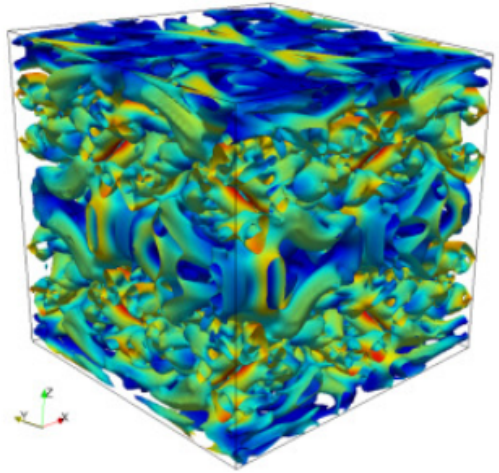
- Introduction
  - Use case of distributed FFTs in simulations
  - FFT libraries and Kokkos performance portable ecosystem

- Performance portable solution for distributed FFTs
  - kokkos-fft: Interface between Kokkos and vendor libraries
  - APIs and functionalities of distributed version
  - Using in 3D Navier Stokes solver
  - Performance analyses

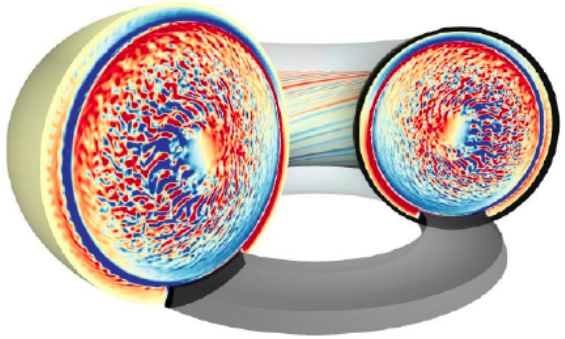
- Summary

Performance portable solution for distributed FFT

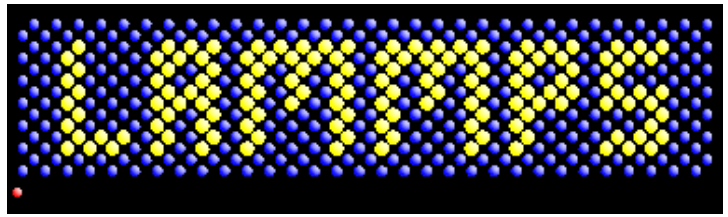
3D Fluid simulation [1]



Plasma turbulence [2]



Molecular dynamics [3]



- Many scientific applications rely on (distributed) FFTs
  - 3D direct numerical simulations
  - Plasma turbulence simulations for nuclear fusion
  - FFT-based particle-particle/particle-mesh (PPPM) method in Molecular dynamics simulations

- Performance portable solution for distributed FFTs needed
  - High performance over NVIDIA, AMD and Intel GPUs
  - Good scalability

[1] M. Uh. Zapata, et al, Computers & Fluids (2023)
[2] V. Grandgirard, et al, CPC (2016)
[3] Aidan P. Thompson, et al, CPC (2022)

Preparation for exascale systems

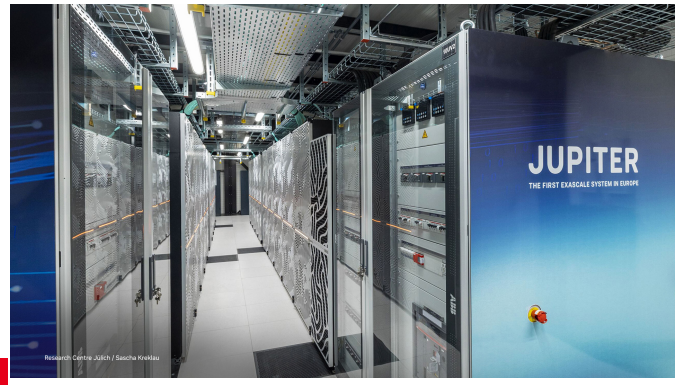
Directives/Higher level abstraction

OpenACC

OpenMP

SYCL™

JUPITER (H200)



Frontier (MI250X)

Aurora (PVC)

Preparation for exascale systems

Directives/Higher level abstraction

OpenACC

OpenMP

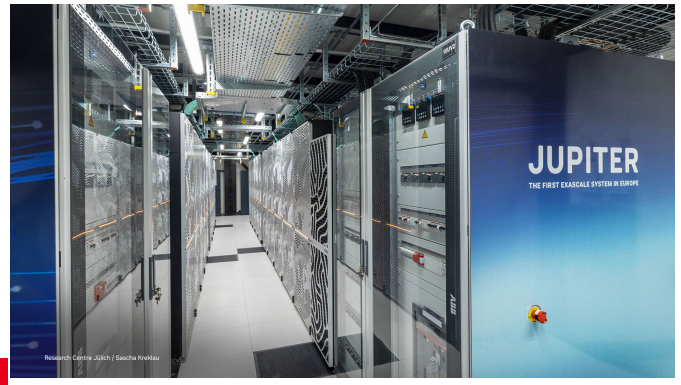
SYCL™

Kokkos ecosystem under HPSF

- Kokkos Kernels
- Kokkos Tools
- Kokkos FFT
- Kokkos Comm

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

JUPITER (H200)



Frontier (MI250X)

Aurora (PVC)

Existing distributed FFT libraries

Library name	Language	CPU	NVIDIA GPUs	AMD GPUs	Intel GPUs
2Decomp&FFT	Fortran/OpenACC	X	X		
cuDecomp	C++/CUDA		X		
cuFFTMp	C++/CUDA		X		
heFFTe	C++/CUDA/HIP/SYCL	X	X	X	X
kokkos-fft	C++/Kokkos	X	X	X	X

- Most existing libraries are highly vendor locked
  - Distributed FFT libraries: FFTs + Communications + Transpose kernels
  - heFFTe offers performance portability by managing distinct specific backends

- Distributed FFT interface on top of the Kokkos ecosystem
  - Benefit from Kokkos: Performance portability, Data structure, consistent build system
  - Simpler to use in Kokkos codes

Kokkos: A HPSF project

Kokkos ecosystem

- Kokkos Kernels
- Kokkos Tools
- Kokkos FFT
- Kokkos Comm

- Kokkos is a HPSF project
  - OSS project which is vendor neutral
  - It has its own ecosystem: linear algebra, FFT, MPI, and tools

- Kokkos key features
  - **A single code runs everywhere**
  - Kokkos Views: multi dimensional arrays accessible from CPUs and GPUs
  - Kokkos parallelization: for, reduce, scan can operate on CPUs and GPUs

[1] C. Trott, et al, CSE (2021)
[2] C. Trott, et al, IEEE Tpds (2022)

Outline

- Introduction
  - Use case of distributed FFTs in simulations
  - FFT libraries and Kokkos performance portable ecosystem

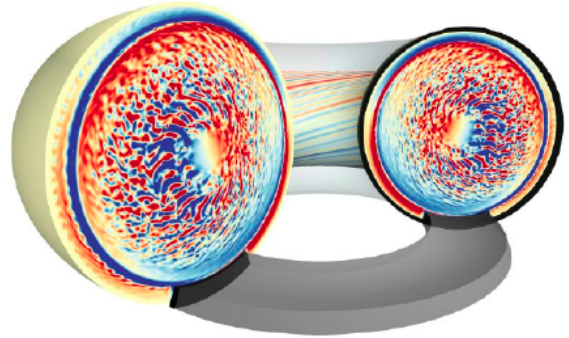
- Performance portable solution for distributed FFTs
  - kokkos-fft: Interface between Kokkos and vendor libraries
  - APIs and functionalities of distributed version
  - Using in 3D Navier Stokes solver
  - Performance analyses

- Summary

Why kokkos-FFT? Who needs it?

- 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**, Low pass filter, 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



Non-uniform interfaces for vendor libraries

FFTW

```
plan = fftw_plan_many_dft(
 rank, fft_extents.data(), howmany,
 idata, in_extents.data(), istride, idist,
 odata, out_extents.data(), ostride, odist,
 sign, FFTW_ESTIMATE);
```

rocFFT

```
rocfft_plan_create(&plan, place,
                   fft_direction, precision,
                   fft_extents.size(),
                   fft_extents.data(),
                   howmany,
                   description);
```

cuFFT

```
cufftPlanMany(&plan, rank, fft_extents.data(),
              in_extents.data(), istride, idist,
              out_extents.data(), ostride, odist,
              type, howmany);
```

oneMKL

```
plan = std::make_unique<PlanType>(fft_extents);
plan->set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,
                in_strides.data());
plan->set_value(oneapi::mkl::dft::config_param::OUTPUT_STRIDES,
                out_strides.data());
// Configuration for batched plan
plan->set_value(oneapi::mkl::dft::config_param::FWD_DISTANCE, idist);
plan->set_value(oneapi::mkl::dft::config_param::BWD_DISTANCE, odist);
plan->set_value(oneapi::mkl::dft::config_param::NUMBER_OF_TRANSFORMS,
                static_cast<std::int64_t>(howmany));
// Data layout in conjugate-even domain
int placement = is_inplace ? DFTI_INPLACE : DFTI_NOT_INPLACE;
plan->set_value(oneapi::mkl::dft::config_param::PLACEMENT, placement);
plan->set_value(oneapi::mkl::dft::config_param::CONJUGATE_EVEN_STORAGE,
                DFTI_COMPLEX_COMPLEX);
sycl::queue q = exec_space.sycl_queue();
plan->commit(q);
```

kokkos-FFT

numpy.fft

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

kokkos-FFT

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

[1] Y. Asahi, et al, JOSS (2025)

Kokkos-fft v1.1 released

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-of-place and in-place)
  - View is all we need: No need to access the complicated FFT APIs
  - Much safer APIs: No need to use raw pointers directly
  - Compile time or runtime errors for invalid usage

- 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**
  - FFT libraries dedicated to Kokkos backends are automatically enabled

- Supported data types: float, double and Kokkos::complex
  - Limited to contiguous layout only: **LayoutLeft** and **LayoutRight**
  - **DefaultExecutionSpace** and **DefaultHostExecutionSpace** supported

kokkos-fft
[1] Y. Asahi, et al., JOSS (2025)

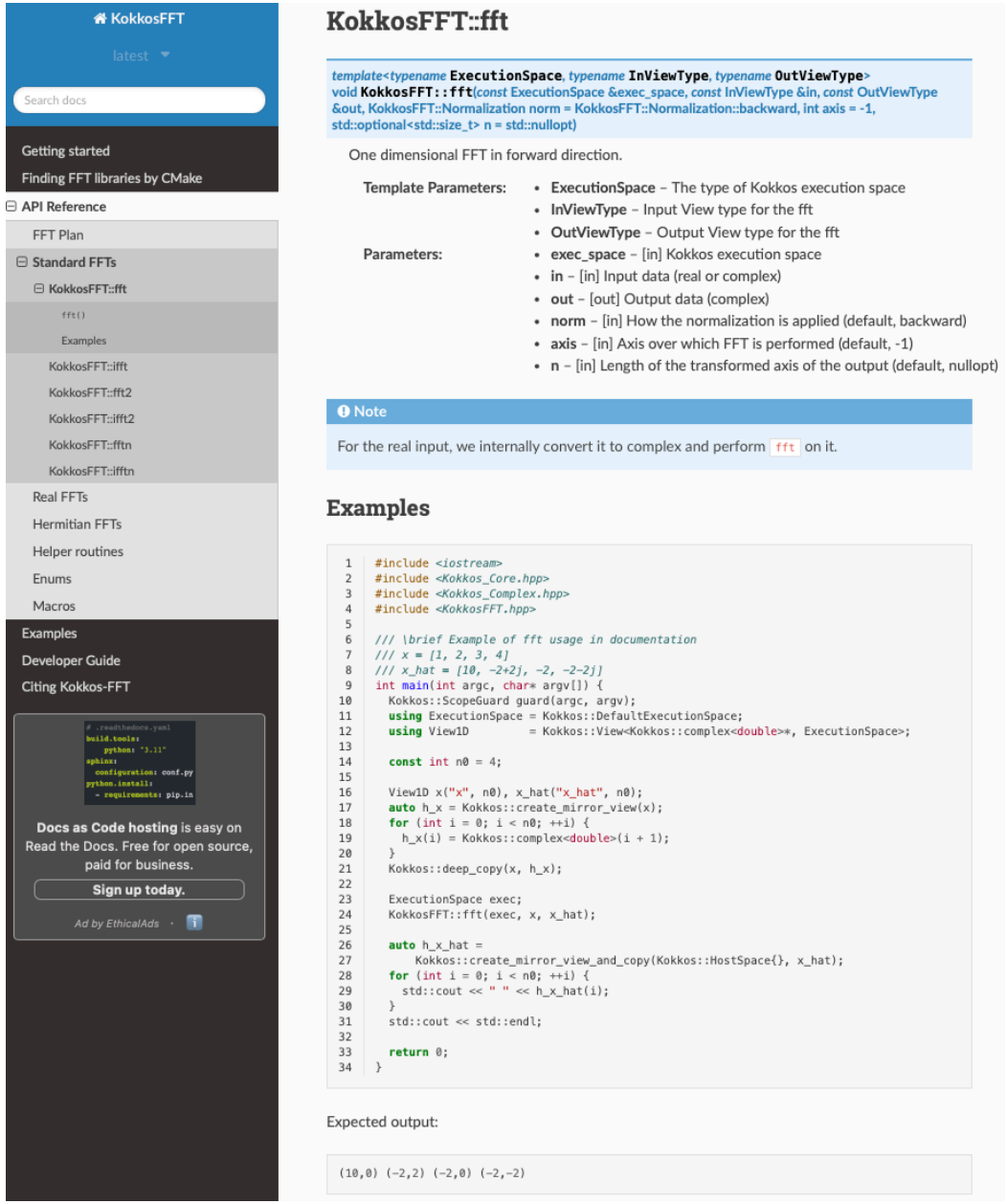
Use cases: DDC, gyselalib++
[2] T. Padioleau, et al., JOSS (2025)
[3] E. Bourne, et al., JOSS (2025)

Reusable FFT plans

```
template <typename T> using View2D = Kokkos::View<T**, Kokkos::LayoutLeft, execution_space>;
const int n0 = 4, n1 = 8, n2 = 5, n3 = 10;

View2D<Kokkos::complex<double> > x("x", n0, n1), x_hat("x_hat", n0, n1);
View2D<Kokkos::complex<double> > y("y", n0, n1), y_hat("y_hat", n0, n1);
View2D<Kokkos::complex<double> > z("z", n2, n3), z_hat("z_hat", n2, n3);

// Create a plan for 1D FFT
int axis = -1;
KokkosFFT::Plan fft_plan(execution_space(), x, x_hat,
                         KokkosFFT::Direction::forward, axis);

// Perform FFTs using fft_plan
KokkosFFT::execute(fft_plan, x, x_hat);

// [OK] Reuse the plan for different data
KokkosFFT::execute(fft_plan, y, y_hat);

// [NG, Run time error] Inconsistent extents
KokkosFFT::execute(fft_plan, z, z_hat);
```

- FFT plans can be reused multiple times on different data given that they have the same properties
- Runtime or compile time errors if the data do not fit the plan
- Inplace FFT plan can also be reused as long as the data are for inplace transform

Documentations (readthedocs)



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
  - kokkos-fft and python
- Developper Guide

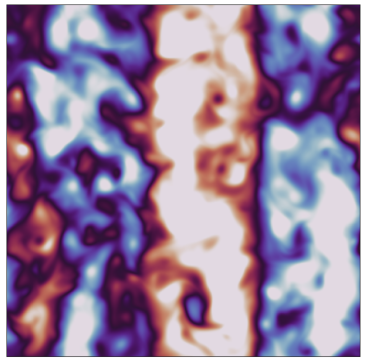
As simple as numpy, as fast as vendor libraries

```cpp
KokkosFFT::irfft2(execution_space(), m_buffer, dfgdx_all);

// Convolution in real space
auto dfdx = Kokkos::subview(dfgdx_all, pair_type(0,2), ALL, ALL);
auto dfdy = Kokkos::subview(dfgdx_all, pair_type(2,4), ALL, ALL);
auto dgdx = Kokkos::subview(dfgdx_all, 4, ALL, ALL);
auto dgdy = Kokkos::subview(dfgdx_all, 5, ALL, ALL);

Kokkos::parallel_for(
  range, KOKKOS_LAMBDA(int iy, int ix) {
    for(int in=0; in<nb_vars; in++) {
      conv(in, iy, ix) = dfdx(in, iy, ix) * dgdy(iy, ix)
        - dfdy(in, iy, ix) * dgdx(iy, ix);
    };
  });

KokkosFFT::rfft2(execution_space(), m_conv, m_buffer);
```

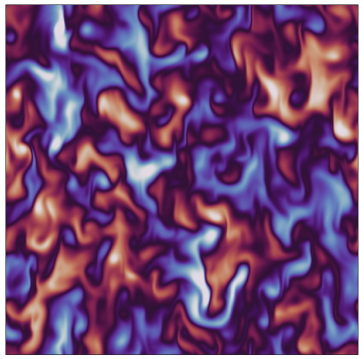
```python
# Inverse FFT complex [ny, nx/2+1] => real [ny, nx]
dfdx = np.fft.irfft2(ikx_f)
dfdy = np.fft.irfft2(iky_f)
dgdx = np.fft.irfft2(ikx_g)
dgdy = np.fft.irfft2(iky_g)

# Convolution in real space
conv = dfdx * dgdy - dfdy * dgdx



# Forward FFT real [ny, nx] => [ny, nx/2+1]
poisson_bracket = np.fft.rfft2(conv)
```

Experiment in 2D Hasegawa-Wakatani solver relying on convolution with FFT (1024 x 1024), 100 iterations (w/o I/O)

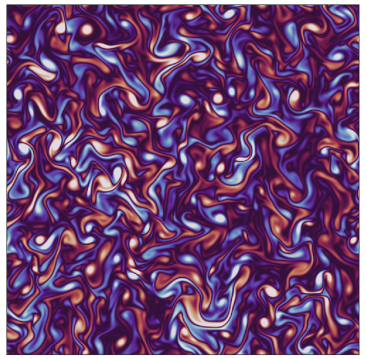
Potential



Density



Vorticity



Device	Icelake (python)	Icelake (36 cores)	A100	H100	MI250X (1 GCD)	PVC
LOC	**568**	**738**	738	738	738	738
GB/s	205	205	1555	3350	1600	3276.8
Elapsed time [s]	463	9.28	0.25	0.14	0.41	0.30
Speed up	x 1	**x 49.9**	**x 1852**	**x 3307**	**x 1129**	**x 1562**

KokkosFFT::Distributed capability

- 1D, 2D, 3D standard and real Fast Fourier Transforms over 1D to 7D Views
  - Batched plans are automatically used if View Dim > FFT Dim

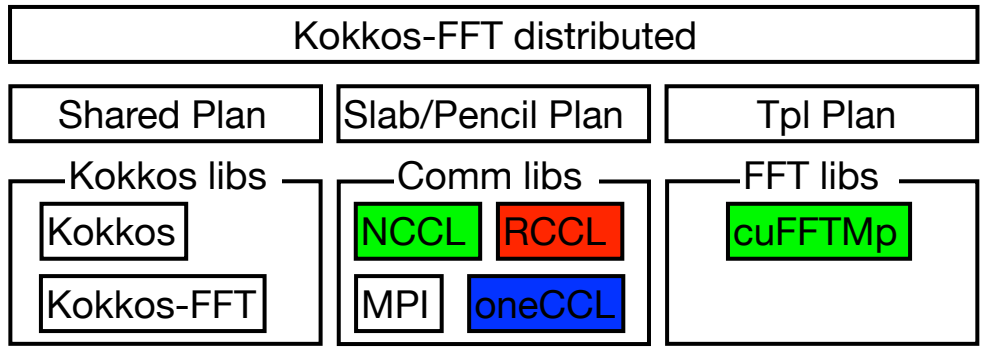
- Simple interfaces like **kokkosFFT::Plan & execute APIs**
  - View is all we need: No need to access the complicated FFT APIs
  - Much safer APIs: No need to use raw pointers directly
  - Compile time or runtime errors for invalid usage

- Supporting multiple CPU and GPU backends
  - **SERIAL, THREADS, OPENMP, CUDA, HIP and SYCL**
  - FFT libraries dedicated to Kokkos backends are automatically enabled

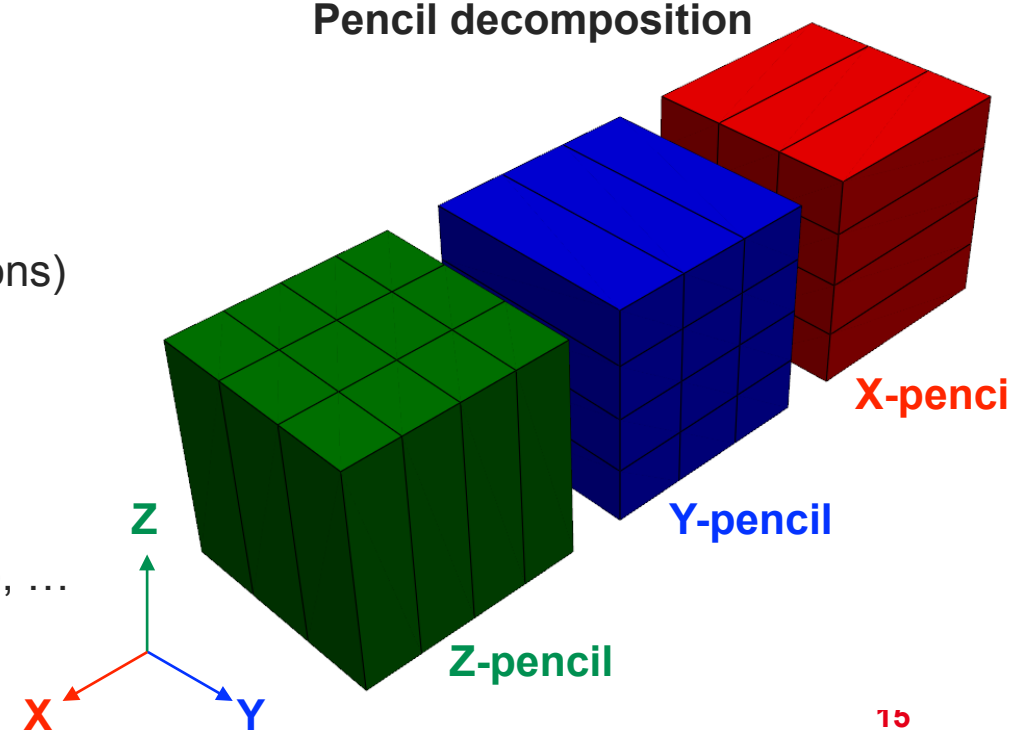
- Parallelization (Seamless integration of Shared, Slab, Pencil decompositions)
  - Device aware MPI (All2All)
  - Collective communication interface: NCCL, RCCL and oneCCL

- Third Party Library (TPL) support
  - Vendor native distributed FFT support: cuFFTMp, rocfftMPI, cuDecomp, …

Global architecture



Pencil decomposition



KokkosFFT::Distributed: API

Shared

```cpp
#include <Kokkos_Core.hpp>
#include <Kokkos_Complex.hpp>
#include <Kokkos_Random.hpp>
#include <KokkosFFT.hpp>

using execution_space = Kokkos::DefaultExecutionSpace;
using axes_type = KokkosFFT::axes_type<2>;
using ComplexView2DType =
    Kokkos::View<Kokkos::complex<T>**, execution_space>;

int main(int argc, char* argv[]) {
  {

    execution_space exec;
    int n0 = 8, n1 = 8;
    ComplexView2DType u("u", n0, n1), u_hat("u_hat", n0, n1);

    // Plan creation
    Plan plan(exec, u, u_hat, KokkosFFT::Direction::forward,
              axes_type{0, 1});

    // Execute the plan
    execute(plan, u, u_hat);
  }
};
```

Distributed

```cpp
#include <mpi.h>
#include <Kokkos_Core.hpp>
#include <Kokkos_Complex.hpp>
#include <Kokkos_Random.hpp>
#include <KokkosFFT.hpp>
#include "Distributed.hpp"

using execution_space = Kokkos::DefaultExecutionSpace;
using extents_type = std::array<std::size_t, 2>;
using axes_type = KokkosFFT::axes_type<2>;
using ComplexView2DType =
    Kokkos::View<Kokkos::complex<T>**, execution_space>;

int main(int argc, char* argv[]) {
  ::MPI_Init(&argc, &argv);
  int rank, nprocs;
  ::MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ::MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  {
    Kokkos::ScopeGuard(argc, argv);
    execution_space exec;
    extents_type topo0 {1, std::size_t(nprocs)},
                 topo1 {std::size_t(nprocs), 1};
    int n0 = 8, n1 = 8;
    ComplexView2DType u_0("u_0", n0, n1/nprocs);
    ComplexView2DType u_hat_1("u_hat_1", n0/nprocs, n1);

    // Plan creation
    Plan plan(exec, u_0, u_hat_1, axes_type{0, 1},
              topo0, topo1, MPI_COMM_WORLD);

    // Execute the plan
    execute(plan, u_0, u_hat_1, KokkosFFT::Direction::forward);
  }
  ::MPI_Finalize();
};
```

- Unified Plan: A single Plan object handles both forward and backward transform
- Distributed version needs: MPI topology
- Local Data Views: Input and Out views are local data residing on a specific MPI rank

KokkosFFT::Distributed: Implementation details

- C++ **dynamic** polymorphism to select from four distinct internal plan types
  - Shared Plan: An alias to the standard kokkos-fft API (non-MPI aware)
  - **Slab Plan**: Manages distributed transforms using a 1D slab decomposition
  - **Pencil Plan**: Manages distributed transforms using a 2D pencil decomposition
  - **TPL Plan**: An interface to third-party distributed FFT libraries (e.g. cuFFTMp)

- Core routines
  - **All2all**: Manages the MPI communications required for global data transpositions using GPU-aware MPI_Alltoall
  - **Pack/Unpack**: packs data into a contiguous send buffer before communication and unpacks it from a receive buffer
  - **Transpose**: local, in-memory transpose to align data for FFTs
  - **Normalization**: 1/N scaling for FFT

Make communications as simple as possible

MPI Wrapper

```
template <typename ViewType>
void all2all(const ViewType& send, const ViewType& recv,
             const MPI_Comm& comm) {
  using value_type = typename ViewType::non_const_value_type;
  using LayoutType = typename ViewType::array_layout;
  int size_send = std::is_same_v<LayoutType, Kokkos::LayoutLeft>
                      ? send.extent_int(ViewType::rank() - 1)
                      : send.extent_int(0);
  auto mpi_data_type = mpi_datatype_v<value_type>;
  int send_count     = static_cast<int>(send.size()) / size_send;
  ::MPI_Alltoall(send.data(), send_count, mpi_data_type, recv.data(),
                 send_count, mpi_data_type, comm);
}
```

- Keep MPI tasks simple
  - Communications are the bottlenecks
  - Use the MPI_Alltoall which is the simplest
  - All the **complexity** is handled on **GPUs**
- Generic GPU kernels
  - Same kernel for 2D - 7D Views
  - Parallelized with MDRange policy

Pack kernel

```
struct PackInternal {
  using LayoutType = typename DstViewType::array_layout;
  using ValueType  = typename SrcViewType::non_const_value_type;
  SrcViewType m_src;
  DstViewType m_dst;
  std::size_t m_axis, m_merged_size, m_nprocs = 1;
  ShapeType m_map;
  ShapeType m_dst_extents;
  ShapeType m_src_extents;

  PackInternal(const SrcViewType& src, const DstViewType& dst,
               const ShapeType& map, const std::size_t axis,
               const std::size_t merged_size)
      : m_src(src),
        m_dst(dst),
        m_axis(axis),
        m_merged_size(merged_size),
        m_map(map) {
    for (std::size_t i = 0; i < SrcViewType::rank(); ++i) {
      m_dst_extents[i] = std::is_same_v<LayoutType, Kokkos::LayoutRight>
                             ? dst.extent(i + 1)
                             : dst.extent(i);
    }
    m_nprocs = std::is_same_v<LayoutType, Kokkos::LayoutRight>
                   ? dst.extent(0)
                   : dst.extent(DstViewType::rank() - 1);
    for (std::size_t i = 0; i < SrcViewType::rank(); ++i) {
      m_src_extents[i] = src.extent(i);
    }
  }

  template <typename... IndicesType>
  KOKKOS_INLINE_FUNCTION void operator()(const IndicesType... indices) const {
    iType dst_indices[DstViewType::rank()] = {
        static_cast<iType>(indices)...};
    m_dst(indices...) = get_src_value(
        dst_indices, std::make_index_sequence<DstViewType::rank() - 1>{});
  }
};
```

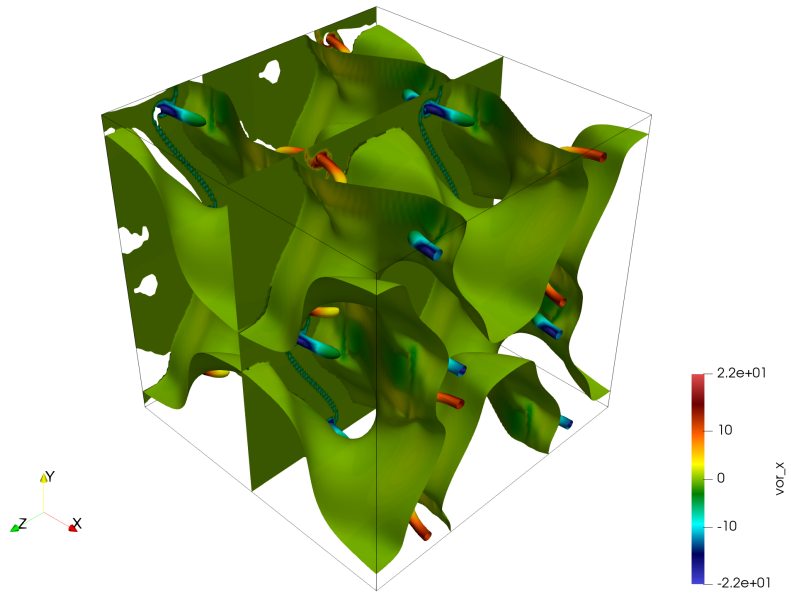
Solving Navier-Stokes equations

Non-batched

```
template <typename ViewType>
void rhs(const ViewType& uk, const ViewType& vk,
         const ViewType& wk, const ViewType& dukdt,
         const ViewType& dvkdt, const ViewType& dwkdt) {
  // Apply dealiasing filter before inverse transform
  dealias(uk, m_grid.m_alias_mask);
  dealias(vk, m_grid.m_alias_mask);
  dealias(wk, m_grid.m_alias_mask);

  // Calculate velocity derivatives in Fourier space
  derivative(uk, m_dukdx, m_dukdy, m_dukdz);
  derivative(vk, m_dvkdx, m_dvkdy, m_dvkdz);
  derivative(wk, m_dwkdx, m_dwkdy, m_dwkdz);

  // Inverse FFT to get derivatives in real space
  execute(*m_plan, uk, m_u, Direction::backward);
  execute(*m_plan, vk, m_v, Direction::backward);
  execute(*m_plan, wk, m_w, Direction::backward);
  execute(*m_plan, m_dukdx, m_dudx, Direction::backward);
  execute(*m_plan, m_dukdy, m_dudy, Direction::backward);
  execute(*m_plan, m_dukdz, m_dudz, Direction::backward);
  execute(*m_plan, m_dvkdx, m_dvdx, Direction::backward);
  execute(*m_plan, m_dvkdy, m_dvdy, Direction::backward);
  execute(*m_plan, m_dvkdz, m_dvdz, Direction::backward);
  execute(*m_plan, m_dwkdx, m_dwdx, Direction::backward);
  execute(*m_plan, m_dwkdy, m_dwdy, Direction::backward);
  execute(*m_plan, m_dwkdz, m_dwdz, Direction::backward);

  // Calculate nonlinear advection terms in real space
  advection(m_u, m_v, m_w, m_dudx, m_dudy, m_dudz,
            m_dvdx, m_dvdy, m_dvdz, m_dwdx, m_dwdy, m_dwdz);

  // FFT of nonlinear terms (stored in dukdt)
  execute(*m_plan, m_u, dukdt, Direction::forward);
  execute(*m_plan, m_v, dvkdt, Direction::forward);
  execute(*m_plan, m_w, dwkdt, Direction::forward);

  dealias(dukdt, m_grid.m_alias_mask);
  dealias(dvkdt, m_grid.m_alias_mask);
  dealias(dwkdt, m_grid.m_alias_mask);

  // Calculate viscous diffusion term in Fourier space
  // And combine terms (stored in dukdt)
  combine(dukdt, dvkdt, dwkdt, uk, vk, wk);

  // Project the RHS onto the divergence-free space
  projection(m_grid, dukdt, dvkdt, dwkdt);
}
```

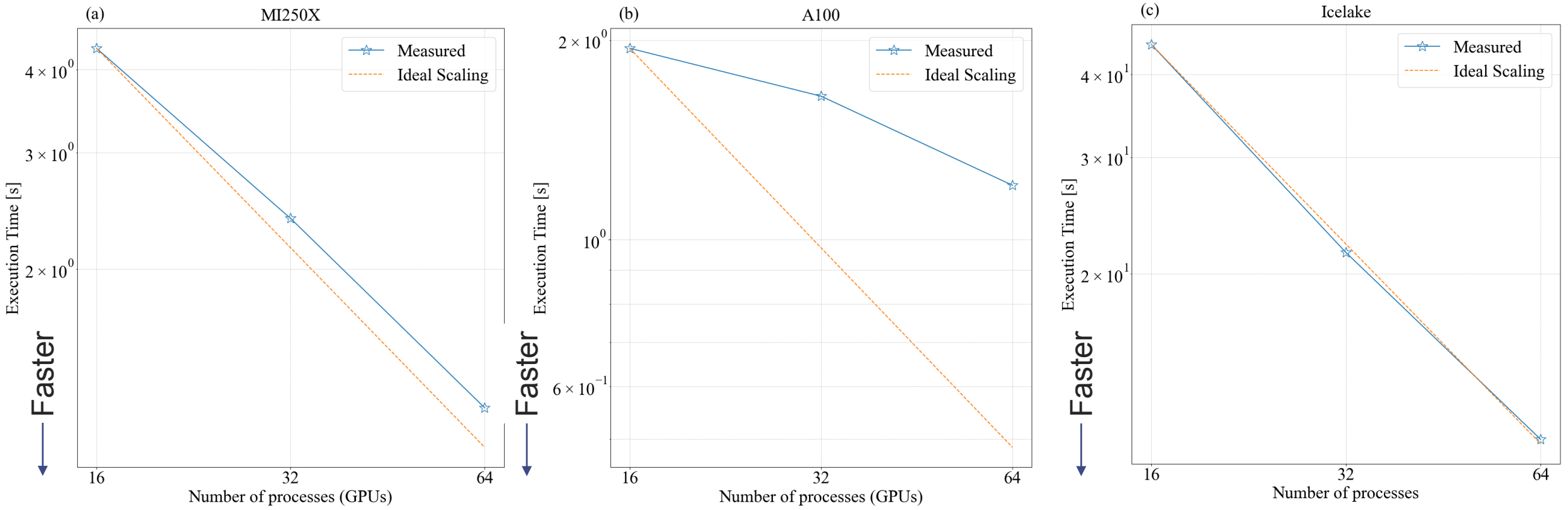
Batched

```
template <typename ViewType>
void rhs(const ViewType& uk, const ViewType& dukdt) {
  // Apply dealiasing filter before inverse transform
  dealias(uk, m_grid.m_alias_mask);

  // Calculate velocity derivatives in Fourier space
  derivative(uk, m_dukdx, m_dukdy, m_dukdz);

  // Inverse FFT to get derivatives in real space
  execute(*m_plan, uk, m_u, Direction::backward);
  execute(*m_plan, m_dukdx, m_dudx, Direction::backward);
  execute(*m_plan, m_dukdy, m_dudy, Direction::backward);
  execute(*m_plan, m_dukdz, m_dudz, Direction::backward);

  // Calculate nonlinear advection terms in real space
  advection(m_u, m_dudx, m_dudy, m_dudz);

  // FFT of nonlinear terms (stored in dukdt)
  execute(*m_plan, m_u, dukdt, Direction::forward);

  dealias(dukdt, m_grid.m_alias_mask);
  // Calculate viscous diffusion term in Fourier space
  // And combine terms (stored in dukdt)
  combine(dukdt, uk);
  projection(m_grid, dukdt);
}
```



- Problem size fixed (1024^3, FP64)
- 12 C2R + 3 R2C repeated in 4 Runge-Kutta steps
- In batched implementation, (u, v, w) is stored in a single 4D View
- At least useful for delta-F gyrokinetic simulations which rely on 2D batched distributed FFTs

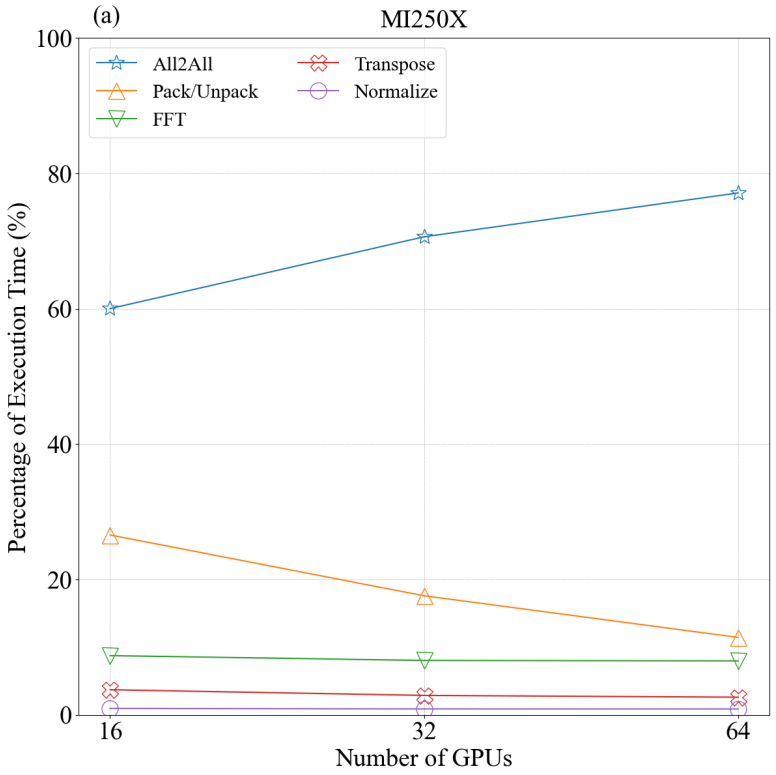
Strong Scaling of NS code (1024^3, 2-8 nodes)

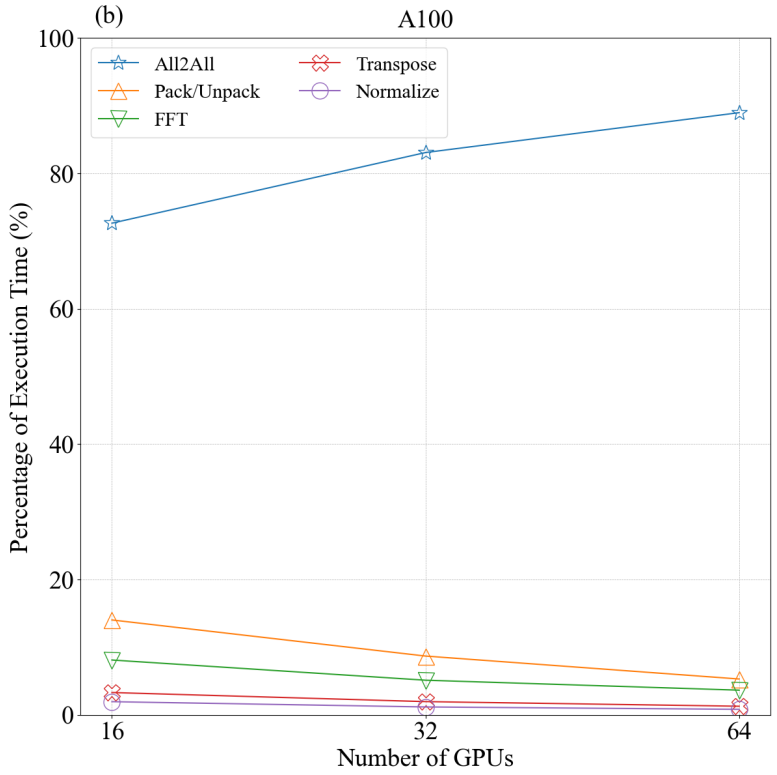


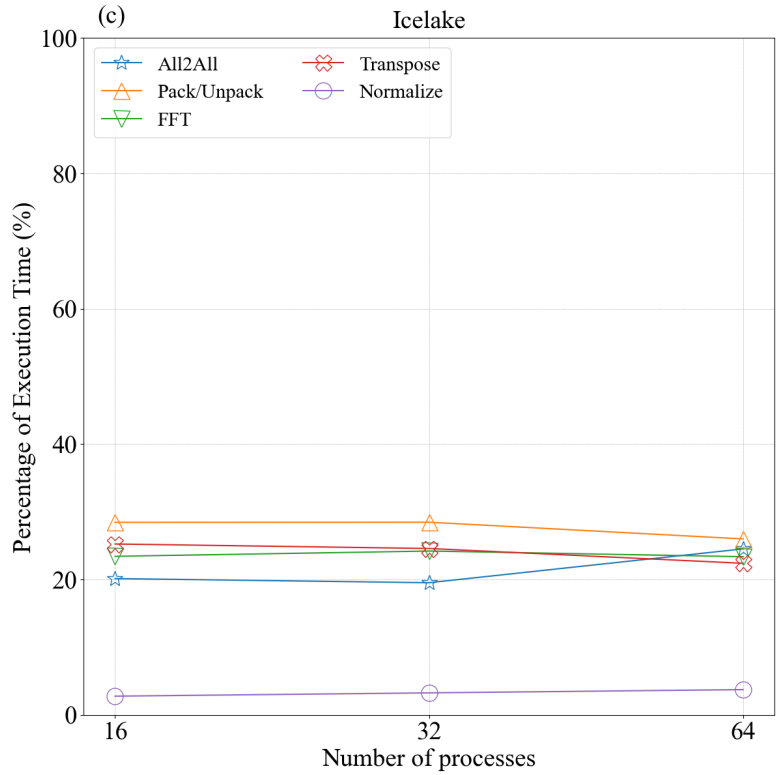
- Highly scalable up to 64 processes on MI250X and Icelake
- On A100, the MPI costs about 90 %

System	Adastra	Wisteria (GPU)	Wisteria (CPU)
Processor	MI250X	A100	Icelake
Number of processors per node	8	8	2
Number of cores (FP64) per processor	-	3456	36
Shared Cache [MB] per processor	16/2	40	54
Peak performance [GFlops] per processor	26500/2	9700	2764.8
Peak B/W [GB/s] per processor	1600	1555	204.8
Intranode bandwidth [GB/s]	200 (Infinity fabric)	300 (NVLink)	25
Internode bandwidth [GB/s]	25	25	25
Compilers	rocm 6.3.3	CUDA/12.0.0	Intel LLVM/2023.0.0
MPI	Cray MPICH 8.1.3	OpenMPI 4.6.1	Intel MPI 2021.8

Breakdown: MI250X, A100 and Icelake







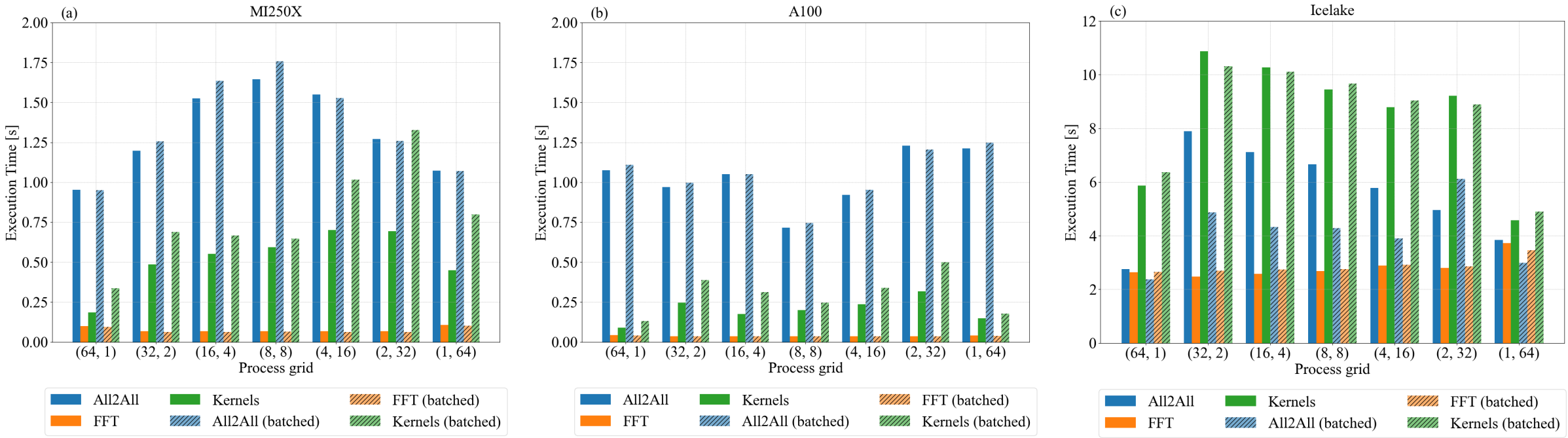
- MPI All2all costs 60-90 % of the execution time on MI250X and A100
- Packing/Unpacking can be improved a little more by improving Kokkos MDRange
- Room for overlapping on Icelake. MPI is not a bottleneck. Mapping may not be the best : 9 Threads per process

Comparison with cuFFTMp

# MPI	Architecture	All2All	Unpack	Pack	FFT	Transpose	Normalize
16	MI250X	2.5869005	0.6046888	0.5407781	0.3771451	0.1599698	0.0410012
16	Icelake	8.9302868	4.9891716	7.6298695	10.390203	11.1979041	1.2257816
16	A100 (MPI)	1.4119186	0.1384448	0.1340588	0.1574669	0.0641185	0.0378279
16	A100 (cuFFTMp)	1.4891016 (cufftXtExecDescriptor) + 0.0591815 (deep_copy)					0.0375082
16	H200 (MPI)	1.3911636	0.0313026	0.0352117	0.0642570	0.0240311	0.0158418
16	H200 (cuFFTMp)	1.2969191 (cufftXtExecDescriptor) + 0.0384167 (deep_copy)					0.0157858
64	MI250X	0.9532557	0.0728108	0.0686664	0.0985339	0.0322849	0.0107661
64	Icelake	2.7603145	1.2328665	0.0321609	2.6308734	2.5209463	0.4199633
64	A100 (MPI)	1.0743411	0.0318285	0.0321609	0.0440656	0.0154079	0.0098744
64	A100 (cuFFTMp)	**0.7914976** (cufftXtExecDescriptor) + 0.0143040 (deep_copy)					0.0094754
64	H200 (MPI)	0.8290491	0.0067909	0.0076418	0.0178711	0.0062001	0.0043301
64	H200 (cuFFTMp)	**0.5940724** (cufftXtExecDescriptor) + 0.0100590 (deep_copy)					0.0042695

- Input: X-slab geometry (N, 1, 1), Output: Y-Slab geometry (1, N, 1)
- Roughly the same performance on MI250X, A100 (and PVC) at scale (All2All is a major bottleneck)
- We initially encountered an issue in cuFFTMp on A100 machine (8 GPUs per node) due to inappropriate **CPU bindings** (every process needs access to at least 2 cores)

Impact of distributed topology and batching



- Thanks to NVlink inside a node, pencil (8x8) decomposition gives the best performance on A100.
- This is not the case on Adastra. Internode communication is not fully covered by Infinity fabric
- Further optimization needed for Icelake. Vectorization issues in MDRange.
- Batching did not help for this case except on Icelake

Coming features (batched APIs)

```
execution_space exec_space;

// Create a batched plan on host
KokkosFFT::Batched::Plan plan(exec_space, Kokkos::subview(x_in, Kokkos::ALL(), 0),
                              Kokkos::subview(x_out, Kokkos::ALL(), 0));

// Execute small FFTs on device
Kokkos::parallel_for(
    Kokkos::RangePolicy<execution_space, Kokkos::IndexType<std::size_t>>{0, nbatch},
    KOKKOS_LAMBDA(const int& i) {
      auto sub_x     = Kokkos::subview(x, Kokkos::ALL(), i);
      auto sub_x_out = Kokkos::subview(x_out, Kokkos::ALL(), i);
      KokkosFFT::Batched::execute(plan, sub_x, sub_x_out);
    });
```

- Highly inspired by batched BLAS/LAPACK APIs in Kokkos-kernels
- Improve data locality
- Kernel fusion including FFTs

Summary and future work

- Performance portable libraries needed for scientific applications
  - Distributed FFT interface on Kokkos ecosystem: performance portability, FFTs, CMake build system

- Key features of KokkosFFT::Distributed
  - **Simple and safe interface** thanks to Kokkos
  - **Good performance** on NVIDIA, AMD and Intel GPUs while keeping a reasonable performance on CPUs
  - Batched capability and Vendor Native distributed FFT library supports

- Future Work
  - Better integration of GPU collective libraries (NVSHMEM)
  - Optimize MDRange: sub-optimal default tile size, registers (Done for Device)

https://github.com/yasahi-hpc/distributed-FFT-for-kokkos.git