Evaluating the performance of distributed FFT over multiple vendor GPUs

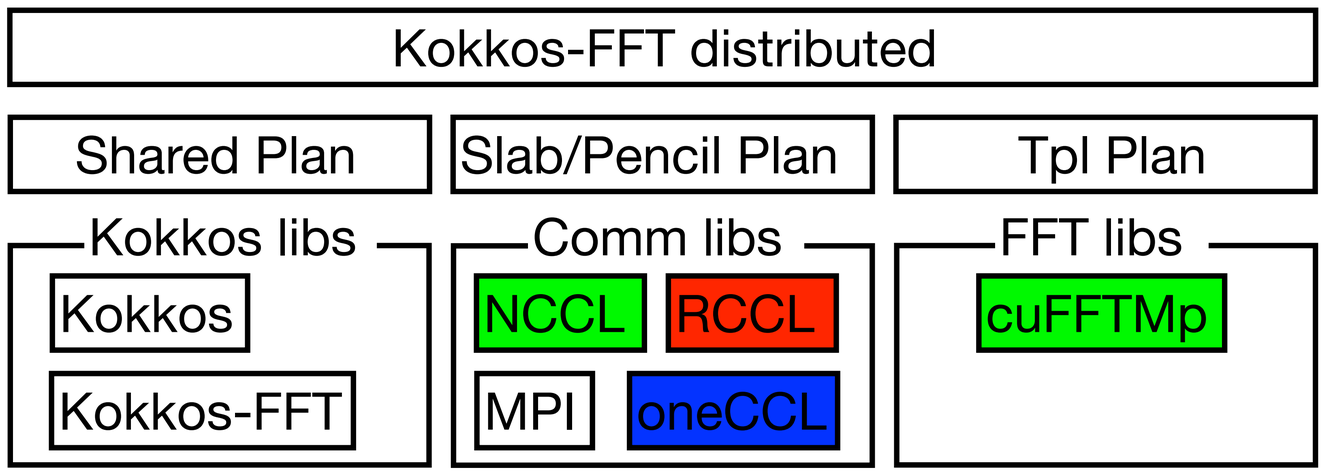
Y. ASAHI[1], T. Morvany[1], T. Padioleau[1], J. Bigot[1]
Maison de la Simulation, Université Paris-Saclay, UVSQ, CNRS, CEA[1]

In this work, we demonstrate a performance portable distributed FFT implementation on top of the Kokkos performance portable ecosystem. The performance portability of our distributed FFT library comes for free as part of the Kokkos ecosystem. Instead of spending time to implement across multiple backends, we focus on developing the unique features such as the batched capability, collective communication layer (e.g. NCCL, RCCL, and oneCCL) and an interface to third party distributed FFT libraries. Our library supports slab and pencil domain decompositions with a batched capability. We have demonstrated reasonable performance on AMD, NVIDIA and Intel GPUs.

Distributed FFT on exascale systems

Library name	Language	CPU	NVIDIA GPUs	AMD GPUs	Intel GPUs
2Decomp&FFT	Fortran/OpenACC	X	X		
cuDecomp	C++/CUDA		X		
cuFFTMp	C++/CUDA	X	X		
heFFTe	C++/CUDA/HIP/SYCL	X	X	X	X
Kokkos-FFT [1]	**C++/Kokkos**	**X**	**X**	**X**	**X**

- Most existing libraries are highly vendor locked
  - Distributed FFT libraries: FFT + Communication + Transpose kernels
  - heFFTe offers performance portability by managing distinct backend specific backend
- Distributed FFT interface on top of the Kokkos ecosystem [2]
  - Benefit from Kokkos: Performance portability, and consistent build system
  - Simpler to use in Kokkos codes



[1] Y. Asahi, et al., JOSS, 2025
[2] Y. Asahi, et al., P3HPC @SC25

KokkosFFT::Distributed features and API

- 1D, 2D, 3D standard and real Fast Fourier Transforms over 1D to 7D Views (can be **batched**)
- Simple interfaces like **KokkosFFT::Plan & execute APIs**
  - View is all we need: No need to access the complicated FFT APIs
  - Compile time or runtime errors for invalid usage (much safer)
- 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, …

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[]) {
  {
    Kokkos::ScopeGuard guard(argc, 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_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 guard(argc, argv);
    execution_space exec;
    int n0 = 8, n1 = 8;
    extents_type topo0 {1, std::size_t(nprocs)}, topo1 {std::size_t(nprocs), 1};
    ComplexView2DType u_0("u_0", n0, n1/nprocs), 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();
};
```

Benchmark: solving 3D Navier-Stokes equation

Algorithm 1 Computation of the RHS

1: **Input:** $\mathbf{u}_k$, **Output:** $\frac{\partial \mathbf{u}_k}{\partial t}$
2: **for all** All grid points (k_x, k_y, k_z) **do**
3: Dealiasing: Apply numerical filters on $\mathbf{u}_k$
4: Derivative: Compute $ik_x\mathbf{u}_k$, $ik_y\mathbf{u}_k$, and $ik_z\mathbf{u}_k$
5: Inverse FFT: Convert $\mathbf{u}_k$, $ik_x\mathbf{u}_k$, $ik_y\mathbf{u}_k$ and $ik_z\mathbf{u}_k$ into $\mathbf{u}_k$, $\frac{\partial \mathbf{u}}{\partial x}$, $\frac{\partial \mathbf{u}}{\partial y}$ and $\frac{\partial \mathbf{u}}{\partial z}$
6: Nonlinear advection: Compute $\mathbf{u} \cdot \nabla \mathbf{u}$ in real space
7: Forward FFT: Convert $\mathbf{u} \cdot \nabla \mathbf{u}$ into Fourier representation followed by dealiasing
8: Add viscous term: Add $\nu \nabla^2 \mathbf{u}$ computed in Fourier space
9: Projection: subtract the pressure gradient from $\frac{\partial \mathbf{u}_k}{\partial t}$
10: **end for**

3D NS equation

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u},$$
$$\nabla \cdot \mathbf{u} = 0,$$

Non-batched

```cpp
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

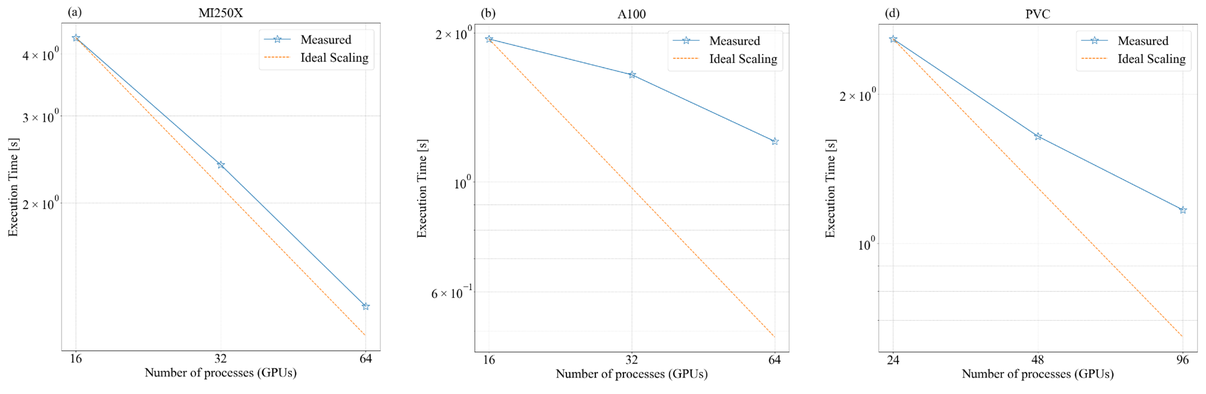
```cpp
template <typename ViewType>
void rhs(const ViewType& uk, const ViewType& dukdt) {
  // Apply dealiasing filter before inverse transform
  dealias(uk, m_grid.m_alias_mask);

  // 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);
}
```

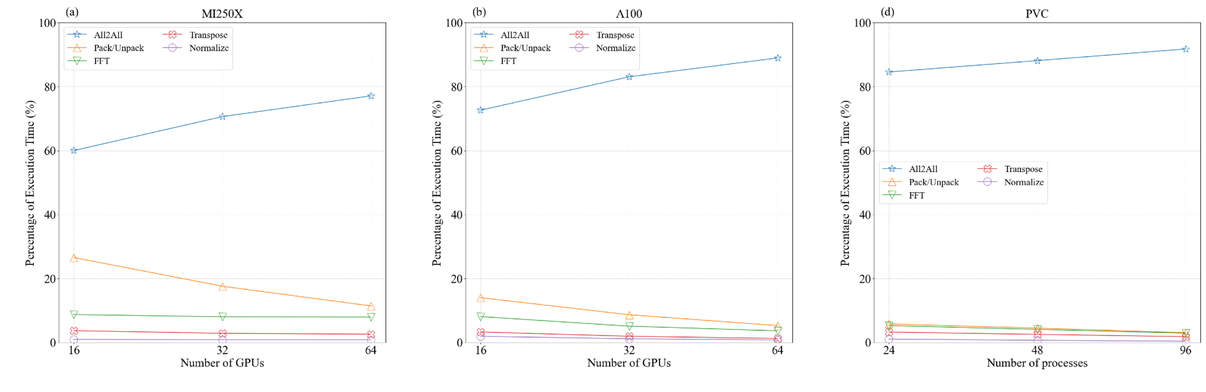
- In batched implementation, (u, v, w) is stored in a single 4D View
- Making a room for further parallelization and potentially optimization
- At least useful for delta-F gyrokinetic simulations which rely on 2D batched distributed FFTs

Testbed description

System	Adastra	Wisteria	Aurora
Processor	MI250X	A100	PVC
Number of processors per node	8	8	6
Peak B/W [GB/s] per processor	1600	1555	3276
Intranode bandwidth [GB/s]	200 (Infinity Fabric)	300 (NVLink)	28 (Xe-Links)
Internode bandwidth [GB/s]	25	25	50
Compilers	rocm 6.3.3	CUDA/12.0.0	Intel LLVM/2025.2.0
MPI	Cray MPICH 8.1.3	OpenMPI 4.6.1	MPICH 4.1

Strong scaling of NS code on AMD, NVIDIA and Intel GPUs (slab decomposition)





Detailed comparison with third party library (cuFFTMp MPI backend)

# MPI	Architecture	All2All	Unpack	Pack	FFT	Transpose	Normalize
16	MI250X	2.5869005	0.6046888	0.5407781	0.3771451	0.1599698	0.0410012
16	A100	1.4119186	0.1384448	0.1340588	0.1574669	0.0641185	0.0378279
24	PVC	2.1882714	0.0745944	0.0755608	0.1367731	0.0844676	0.0279205
16	A100 (cuFFTMp)	5.6404234 (cufftXtExecDescriptor) + 0.0615868 (deep_copy)					0.0398363
32	MI250X	1.6854427	0.2153435	0.2045072	0.1917983	0.0685456	0.0208990
32	A100	1.3671452	0.0736637	0.0692853	0.0842405	0.0321864	0.0192575
48	PVC	1.4499639	0.0367731	0.0374491	0.0675543	0.0419294	0.0116997
32	A100 (cuFFTMp)	6.2026824 (cufftXtExecDescriptor) + 0.0279685 (deep_copy)					0.0189584
64	MI250X	**0.9532557**	0.0728108	0.0686664	0.0985339	0.0322849	0.0107661
64	A100	**1.0743411**	0.0318285	0.0321609	0.0440656	0.0154079	0.0098744
96	PVC	**1.0724439**	0.0180935	0.0177808	0.0341698	0.0214499	0.0047145
64	A100 (cuFFTMp)	6.7232816 (cufftXtExecDescriptor) + 0.0143585 (deep_copy)					0.0093825

- Keep MPI tasks simple. All the complexities are handled on GPUs.
- cuFFTMp does not scale at all. Need further investigation
- Roughly the same performance on MI250X and A100 (Internode bandwidth is 25 GB/s on these machines)
- With 64 (or 96) MPI processes, All2All is a major bottleneck

Conclusions

- Performance portable implementation of distributed FFTs: running efficiently on MI250X, A100 and PVC
- Batched capability and vendor native distributed FFT library supports