

cuFINUFFT: a load-balanced GPU library for general-purpose nonuniform FFTs

Yu-hsuan Shih

*Courant Institute of Mathematical Sciences
New York University
New York, NY, USA*

Garrett Wright

*PACM
Princeton University
Princeton, NJ, USA*

Joakim Andén

*Department of Mathematics
KTH Royal Institute of Technology
Stockholm, Sweden*

Johannes Blaschke

*National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA, USA*

Alex H. Barnett

*Center for Computational Mathematics
Flatiron Institute
New York, NY, USA*

Abstract—Nonuniform fast Fourier transforms dominate the computational cost in many applications including image reconstruction and signal processing. We thus present a general-purpose GPU-based CUDA library for type 1 (nonuniform to uniform) and type 2 (uniform to nonuniform) transforms in dimensions 2 and 3, in single or double precision. It achieves high performance for a given user-requested accuracy, regardless of the distribution of nonuniform points, via cache-aware point reordering, and load-balanced blocked spreading in shared memory. At low accuracies, this gives on-GPU throughputs around 10^9 nonuniform points per second, and (even including host-device transfer) is typically 4–10 \times faster than the latest parallel CPU code FINUFFT (at 28 threads). It is competitive with two established GPU codes, being up to 90 \times faster at high accuracy and/or type 1 clustered point distributions. Finally we demonstrate a 5–12 \times speedup versus CPU in an X-ray diffraction 3D iterative reconstruction task at 10^{-12} accuracy, observing excellent multi-GPU weak scaling up to one rank per GPU.

Index Terms—Nonuniform FFT, GPU, load balancing.

I. INTRODUCTION

Nonuniform (or nonequispaced) fast Fourier transforms (NUFFT) are fast algorithms that generalize the FFT to the case of off-grid points. They thus have a wealth of applications in engineering and scientific computing, including image reconstruction from off-grid Fourier data (e.g. MRI gridding [1], optical coherence tomography [2], cryo electron microscopy [3]–[7], radioastronomy [8], coherent diffraction X-ray imaging [9]); wave diffraction [10]; partial differential equations [11], [12]; and long-range interactions in molecular [13] and particle dynamics [14]. For reviews, see [15]–[18].

In 2D, the type 1 NUFFT [15] (also known as the “adjoint nonequispaced fast Fourier transform” or “adjoint NFFT” [17]) evaluates the uniform $N_1 \times N_2$ grid of *Fourier series coefficients* f_{k_1, k_2} due to a set of point masses of arbitrary strengths c_j and locations $(x_j, y_j) \in [-\pi, \pi)^2$, $j = 1, \dots, M$:

$$f_{k_1, k_2} := \sum_{j=1}^M c_j e^{-i(k_1 x_j + k_2 y_j)}, \quad (k_1, k_2) \in \mathcal{I}_{N_1, N_2}, \quad (1)$$

where the 1D integer Fourier frequency grid is defined by

$$\mathcal{I}_N := \{k \in \mathbb{Z} : -N/2 \leq k < N/2\}, \quad (2)$$

and we use the notation $\mathcal{I}_{N_1, N_2} := \mathcal{I}_{N_1} \times \mathcal{I}_{N_2}$ for a 2D grid of Fourier frequencies.

The type 2 NUFFT (or “NFFT” [17]) is the adjoint of the type 1. Given a grid of Fourier coefficients f_{k_1, k_2} it evaluates the resulting Fourier series at arbitrary (generally nonuniform) targets $(x_j, y_j) \in [-\pi, \pi)^2$, to give

$$c_j := \sum_{(k_1, k_2) \in \mathcal{I}_{N_1, N_2}} f_{k_1, k_2} e^{i(k_1 x_j + k_2 y_j)}, \quad j = 1, \dots, M. \quad (3)$$

In contrast to the FFT, type 1 is generally *not* the inverse of type 2: inverting a NUFFT usually requires iterative solution of a linear system [17], [19]. Definitions (1) and (3) generalize to 1D and 3D in the obvious fashion [15].

Naively the exponential sums in (1) and (3) take $\mathcal{O}(NM)$ effort, where $N := N_1 \times \dots \times N_d$ is the total number of Fourier modes, and d the dimension. The NUFFT uses *fast algorithms* [15], [20] to approximate these sums to a user-prescribed tolerance ε , typically with effort of only $\mathcal{O}(N \log N + M \log^d(1/\varepsilon))$, i.e., quasi-linear in the data size. Most algorithms internally set up a “fine” grid of size $n_1 \times \dots \times n_d$, where each $n_i = \sigma N_i$, for a given upsampling factor $\sigma > 1$. Then the type 1 transform has three steps:

- i) *spreading* (convolution) of each weighted nonuniform point by a localized kernel, writing into the fine grid,
- ii) performing a d -dimensional FFT of this fine grid, then
- iii) selecting the central N output modes from this fine grid, after pointwise division (deconvolution) by the kernel Fourier coefficients.

Type 2 simply reverses (transposes) these steps, with i) becoming kernel-weighted *interpolation* from the fine grid to the nonuniform target points. See Sec. II for details.

The NUFFT is often the rate-limiting step in applications, especially for iterative reconstruction [1], motivating the need for high throughput. Spreading and interpolation are often the

dominant steps of NUFFTs, due to scattered memory writes and reads of kernel-sized blocks. Since it demands high memory bandwidth, yet is data parallel, it is a task well suited for acceleration by a general-purpose GPU [21].

This potential of GPUs to accelerate the NUFFT has of course been noted, and to some extent exploited, in prior implementations [22]–[25]. However, the present work shows that it is possible to increase the efficiency significantly beyond that of prior codes, in the same hardware, via algorithmic innovations. With few exceptions [22], most prior GPU NUFFT implementations are packaged in a manner specific to a single science application (e.g. MRI [23]–[25], OCT [2], MD [13], or cryo-EM [5]), have unknown or limited accuracy, and lack mathematical documentation and testing, rendering them almost inaccessible to the general user. This motivates the need for an efficient, tested, general-purpose GPU code.

A. Contributions of this work

We present cuFINUFFT, a general-purpose GPU-based CUDA NUFFT library. It exploits a recently-developed kernel with optimally small width for a full range of user-chosen tolerances (10^{-1} to 10^{-12}), yet more efficient to evaluate than prior ones [18], [26]. Its throughput is high—and largely insensitive to the point distribution.

One main contribution is to accelerate spreading in the type 1 NUFFT. Broadly speaking there have been two styles of parallelization in prior work: “input driven” [27] (or scatter [5]), which assigns one thread to each nonuniform point, and “output driven” [5], [17], [25], [28] (gather), which assigns each thread a distinct portion of the fine output grid to spread to. The input driven scheme, accumulating to GPU global memory, has been used in many prior GPU codes [14], [22], [29]; we will refer to our implementation of this baseline method as **GM** (global memory). While load-balanced, the memory access is arbitrary and can suffer from atomic collisions between writes (see Sec. IV-A). Yet, a naive output driven approach, although collision-free, is poorly load-balanced for highly nonuniform point distributions [18, Rmk. 12]. To address these issues we propose two new spreading methods:

- **GM-sort** (global memory, sorted). This improves upon **GM** in that the work order of the nonuniform points is chosen by spatial sorting into bins (boxes) covering the fine grid. This regularizes the memory access pattern, enabling cache reuse.
- **SM** (shared memory). This sets up spreading “subproblems” executed in faster GPU shared memory. This is a hybrid scheme (see Fig. 1): each subproblem has a local copy of the fine grid lying within the kernel half-width of one bin (output driven), yet is load-balanced by capping its subset of nonuniform points (input driven). Local fine grids are added back into global memory using far fewer global atomic operations than **GM** methods, avoiding collisions. The result is 2–10× faster than **GM-sort** (depending on d and clustering).

Bin sizes and shapes have been hand-tuned for performance; this is crucial for **SM** to ensure optimal use of limited shared memory.

Turning to the interpolation task in type 2, we propose to use the adjoint version of **GM-sort**, where grid writes are replaced by reads. We will also refer to this algorithm as **GM-sort**.

For both tasks, while bin-sorting nonuniform points adds time, it accelerates the execution of spreading/interpolation. Thus our library uses a “plan, setup, execute, destroy” interface that allows efficient *reuse* of the same (sorted) nonuniform points with new strength vectors (e.g. new c_i in (1)). This use case is common, e.g. in iterative methods for NUFFT inversion.

We benchmark in detail the speedup of cuFINUFFT over existing NUFFT libraries, for a range of accuracies, problem sizes, and point distributions. For example, including GPU memory allocation and transfer time, for low accuracy and quasi-uniform points, cuFINUFFT is on average $8\times$ faster than FINUFFT [18] (28 threads), $5\times$ faster than CUNFFT [22] and $78\times$ faster than gpuNUFFT [24] for type 1 transforms. For type 2, cuFINUFFT is on average $6\times$ faster than FINUFFT, $5\times$ faster than gpuNUFFT, and performs similarly to CUNFFT but with 2–5× faster “execute” times.

The library also enables multi-GPU parallelism, essential for larger problems in HPC environments. In Sec. V we show this in the setting of 3D single particle reconstruction from coherent X-ray diffraction data, which demands thousands of 3D NUFFTs. For NSERC and OLCF supercomputer nodes, we demonstrate an order of magnitude speedup over the CPU version, and excellent weak scaling with respect to the number of GPU processes, up to one process per GPU.

The code and documentation for the library is available on GitHub¹ and installable as a PyPI package in Python through `pip install cufinufft`.

B. Limitations

Our library has a few limitations. (1) Both **GM-sort** and **SM** have some GPU memory overhead, due to sorting index arrays. Yet, for a large 3D transform ($N_i = 256$, $i = 1, 2, 3$, and $M = 1.3 \times 10^8$), this overhead is only around 20%. (2) The **SM** method, while providing a large acceleration for type 1, is currently limited to single precision, due to the small GPU shared memory per thread block (49 kB). (3) We fixed the upsampling factor $\sigma = 2$; reducing this could reduce memory overhead and run times [18]. (4) Our library does not yet provide NUFFTs in 1D, nor of type 3 (nonuniform to nonuniform) [30].

II. ALGORITHMS

We follow the standard three-step scheme presented in the previous section. Our Fourier transform convention is

$$\hat{\phi}(k) = \int_{-\infty}^{\infty} \phi(x) e^{-ikx} dx, \quad \phi(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{\phi}(k) e^{ikx} dk. \quad (4)$$

¹<https://github.com/flatironinstitute/cufinufft>

We fix the upsampling factor $\sigma = 2$, and use the “exponential of semicircle” (ES) kernel from FINUFFT [18], [26],

$$\phi_\beta(z) := \begin{cases} e^{\beta(\sqrt{1-z^2}-1)}, & |z| \leq 1 \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

Given a user-requested tolerance ε , the kernel width w in fine grid points, and parameter β in (5), are set via

$$w = \lceil \log_{10} 1/\varepsilon \rceil + 1, \quad \beta = 2.30w. \quad (6)$$

This typically gives relative ℓ_2 errors close to ε [18]. As in FINUFFT, for FFT efficiency, the fine grid size n_i is set to be the smallest integer of the form $2^{q_i}3^{p_i}5^{r_i}$, greater than or equal to $\max(\sigma N_i, 2w)$, in each dimension $i = 1, \dots, d$.

A. Type 1: nonuniform to uniform

We use the same algorithm as FINUFFT to compute \tilde{f}_{k_1, k_2} , an approximation to f_{k_1, k_2} in (1). We will write only the 2D case, the generalization to 3D being clear.

a) Step 1 (spreading): For each index (l_1, l_2) in the fine grid $0 \leq l_1 < n_1$, $0 \leq l_2 < n_2$, compute

$$b_{l_1, l_2} = \sum_{j=1}^M c_j \psi^{\text{per}}(l_1 h_1 - x_j, l_2 h_2 - y_j), \quad (7)$$

where $h_i := 2\pi/n_i$ is the fine grid spacing, and $\psi^{\text{per}}(x, y)$ is the periodized tensor product of rescaled ES kernels

$$\begin{aligned} \psi(x, y) &:= \phi_\beta(x/\alpha_1)\phi_\beta(y/\alpha_2), \quad \alpha_i = w\pi/n_i, \quad i = 1, 2, \\ \psi^{\text{per}}(x, y) &:= \sum_{(m_1, m_2) \in \mathbb{Z}^2} \psi(x - 2\pi m_1, y - 2\pi m_2). \end{aligned} \quad (8)$$

Note that each nonuniform point (x_j, y_j) only affects a nearby square of w^2 fine grid points.

b) Step 2: Use a plain 2D FFT to evaluate

$$\hat{b}_{k_1, k_2} = \sum_{l_2=0}^{n_2-1} \sum_{l_1=0}^{n_1-1} b_{l_1, l_2} e^{-2\pi i(l_1 k_1/n_1 + l_2 k_2/n_2)}, \quad (k_1, k_2) \in \mathcal{I}_{n_1, n_2}. \quad (9)$$

c) Step 3 (correction): Truncate the Fourier coefficients to the central $N_1 \times N_2$ frequencies, and scale them to give the final outputs

$$\tilde{f}_{k_1, k_2} = p_{k_1, k_2} \hat{b}_{k_1, k_2}, \quad (k_1, k_2) \in \mathcal{I}_{N_1, N_2}. \quad (10)$$

Here, correction (deconvolution) factors p_{k_1, k_2} are precomputed from samples of the kernel Fourier transform, via

$$p_{k_1, k_2} = h_1 h_2 \hat{\psi}(k_1, k_2)^{-1} = (2/w)^2 (\hat{\phi}_\beta(\alpha_1 k_1) \hat{\phi}_\beta(\alpha_2 k_2))^{-1}.$$

B. Type 2: uniform to nonuniform

To compute \tilde{c}_j , an approximation to c_j in (3), as in FINUFFT, the above steps for type 1 are reversed. The correction factors p_{k_1, k_2} and the periodized kernel ψ^{per} remain as above.

a) Step 1 (correction): Pre-correct then zero-pad the coefficients f_{k_1, k_2} to the fine grid, i.e., for all indices (l_1, l_2) ,

$$\hat{b}_{l_1, l_2} = \begin{cases} p_{k_1, k_2} f_{k_1, k_2}, & (k_1, k_2) \in \mathcal{I}_{N_1, N_2} \\ 0, & (k_1, k_2) \in \mathcal{I}_{n_1, n_2} \setminus \mathcal{I}_{N_1, N_2} \end{cases} \quad (11)$$

b) Step 2: Use a plain inverse 2D FFT to evaluate

$$b_{l_1, l_2} = \sum_{(k_1, k_2) \in \mathcal{I}_{n_1, n_2}} \hat{b}_{k_1, k_2} e^{2\pi i(l_1 k_1/n_1 + l_2 k_2/n_2)}, \quad l_i = 0, \dots, n_i - 1, \quad i = 1, 2. \quad (12)$$

c) Step 3 (interpolation): Compute a weighted sum of the w^2 grid values near each target nonuniform point (x_j, y_j) ,

$$\tilde{c}_j = \sum_{l_1=0}^{n_1-1} \sum_{l_2=0}^{n_2-1} b_{l_1, l_2} \psi^{\text{per}}(l_1 h_1 - x_j, l_2 h_2 - y_j), \quad j = 1, \dots, M.$$

III. GPU IMPLEMENTATION

This section shows how the above three-step algorithms are implemented on the GPU using the CUDA API. For the FFT in both types, we use NVIDIA’s cuFFT library. For the correction steps, since the task is embarrassingly parallel, we simply launch one thread for each of the $N_1 \times N_2$ Fourier modes. The factors p_{k_1, k_2} are precomputed once in the planning stage. The major work lies in the spreading (type 1) and interpolation (type 2), to which we now turn.

A. Spreading

Recall that we use **GM** to denote a baseline input driven spreading implementation in global memory (as used in CUNFFT [22]). This launches one thread per nonuniform point, i.e. M in total, in their user-supplied order. The thread given nonuniform point j spreads it to the fine grid b array:

$$b_{l_1, l_2} \leftarrow b_{l_1, l_2} + c_j \psi^{\text{per}}(l_1 h_1 - x_j, l_2 h_2 - y_j), \quad \forall (l_1, l_2).$$

This task comprises (a) reading x_j, y_j, c_j from GPU global memory, (b) $2w$ evaluations of the kernel function ψ , and (c) w^2 atomic adds to the b array residing in GPU global memory. One downside of this approach is that nonuniform points assigned to threads within a thread block and hence within a warp can reside far from each other on the grid, which results in uncoalesced memory loads. (Note that assigning one *thread block* per nonuniform point may alleviate this issue [14].) Another downside is that global atomic operations for clustered points can essentially serialize the method.

GM-sort is a scheme to address the issue of uncoalesced access. We partition the $n_1 \times n_2$ fine grid into rectangular “bins” R_i , $i = 1, \dots, n_{\text{bins}}$, each of integer sizes $m_1 \times m_2$ if possible, otherwise smaller. (A typical choice is $m_1 = m_2 = 32$.) Thus $n_{\text{bins}} = \lceil \frac{n_1}{m_1} \rceil \times \lceil \frac{n_2}{m_2} \rceil$. Bins are ordered in a Cartesian grid with the x axis fast and y slow, which echoes on a larger scale the ordering of the fine grid itself. We will say that nonuniform point j is “inside” bin R_i if the point’s rounded integer fine grid coordinates

$$l_1 = \lfloor (x_j + \pi)/h_1 \rfloor, \quad l_2 = \lfloor (y_j + \pi)/h_2 \rfloor,$$

lie in R_i . Suppose that there are M_i nonuniform points inside bin R_i for $i = 1, \dots, n_{\text{bins}}$. We set up a permutation t (a bijection from $\{1, \dots, M\}$ to itself), such that the nonuniform points with indices $t(1), t(2), \dots, t(M_1)$ are precisely those lying in bin R_1 , then those with indices $t(M_1 + 1), t(M_1 +$

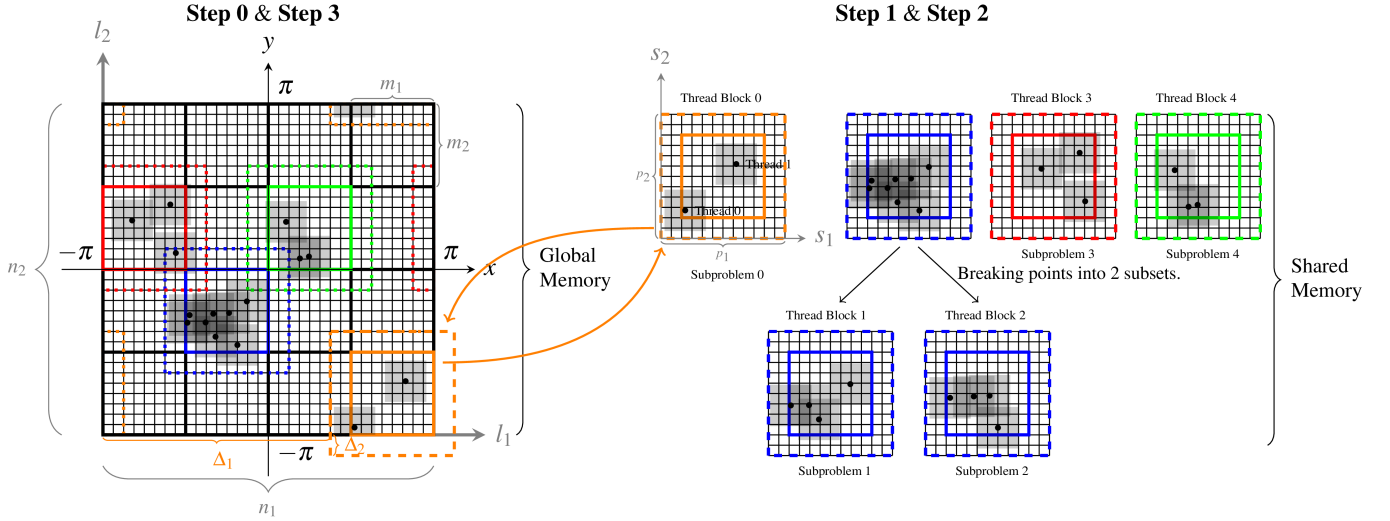


Fig. 1: **SM** spreading scheme. The shaded gray squares (each $w \times w$ fine grid points) show the support of the spreading kernel, centered on each nonuniform point (black dots). Colors indicate the different nonempty bins. **Step 0**: Divide the $n_1 \times n_2$ fine grid into bins of size $m_1 \times m_2$. **Step 1**: Assign subproblems by bin-sorting non-uniform points, and, if needed, further splitting so that there are at most M_{sub} points per subproblem. **Step 2**: Spread the points inside each subproblem into a padded bin copy in faster shared memory (size $p_1 \times p_2$). **Step 3**: Atomic add each padded bin result back into global memory.

2), ..., $t(M_1 + M_2)$ are precisely those in bin R_2 , etc. This is done in practice by first recording the bin index of each point, reading out this list in bin ordering, then inverting this permutation to give t . Nonuniform points are then assigned to threads in the permuted index order $t(1), \dots, t(M)$. This means that the threads within a warp now access parts of the b fine grid array that, most of the time, are approximately adjacent. The GPU therefore has a better chance of coalescing these accesses into fewer global memory transactions.

SM is a hybrid scheme which exploits GPU shared memory to further address the issue of slow global atomic operations. It partitions the fine grid into bins R_i as above, then has three remaining steps, as follows (see Fig. 1).

Step 1: Assign subproblems using bin-sorting and blocking of nonuniform points. The nonuniform point index list $1, \dots, M$ is broken into the union of disjoint subsets S_1, S_2, \dots , each of which we call a “subproblem”. This is done as follows. For bin R_1 , if the number of points M_1 is larger than M_{sub} , a parameter controlling the maximum subproblem size, then it is further broken into subsets (subproblems) of size at most M_{sub} . These one or more subproblems are all associated to the bin R_1 , in which their points lie. The same is repeated for the remainder of the bins R_i . Thread blocks are then launched, one per subproblem. Note that the cap on subproblem size is a (blocked) form of input driven load-balancing: if many points lie in a bin, their spreading tasks are well parallelized.

Step 2: Spread nonuniform points inside each subproblem to shared memory. By the above partition, within a subproblem (a thread block), the nonuniform points can only affect the fine grid array b within a *padded bin* of dimensions $p_1 \times p_2$, where

$$p_i = (m_i + 2\lceil w/2 \rceil), \quad i = 1, 2. \quad (13)$$

For the k th subproblem S_k , we accumulate its spreading result on a shared memory copy b^{shared} of its padded bin, local to its thread block,

$$b_{s_1, s_2}^{\text{shared}} \leftarrow b_{s_1, s_2}^{\text{shared}} + \sum_{j \in S_k} c_j \psi^{\text{per}}((s_1 + \Delta_1)h_1 - x_j, (s_2 + \Delta_2)h_2 - y_j),$$

$$s_i = 0, \dots, p_i - 1, \quad i = 1, 2, \quad (14)$$

where (s_1, s_2) are local indices within the padded bin copy, and (Δ_1, Δ_2) its offset within the fine grid (see Fig. 1).

Step 3: Atomic add the results back to global memory. Once the spreading subproblem result is accumulated in the shared memory padded bin, we atomically add it back to the corresponding region of global memory array b ,

$$b_{l_1(s_1), l_2(s_2)} \leftarrow b_{l_1(s_1), l_2(s_2)} + b_{s_1, s_2}^{\text{shared}}, \quad \forall (s_1, s_2) \quad (15)$$

where $l_i(s_i) := (s_i + \Delta_i) \bmod n_i$, $i = 1, 2$, maps each coordinate in the padded bin back to the fine grid, with periodic wrapping (see Fig. 1). This completes the spreading process. When there are many nonuniform points per bin, this incurs many fewer global atomic writes than **GM-sort**.

We generalize both the implementations **GM-sort** and **SM** to 3D by using cuboid bins of maximum dimension $m_1 \times m_2 \times m_3$.

Remark 1: In both methods **GM-sort** and **SM**, the performance is sensitive to the bin sizes. By hand-tuning (in powers of two), we have found 32×32 in 2D and $16 \times 16 \times 2$ in 3D to be optimal. This takes account of the area (or volume) ratio of bin to padded bin, the ordering of fine grid data, and the maximum size of GPU shared memory per thread block (49 kB), and are based on speed tests on an NVIDIA Tesla V100.

We similarly set $M_{\text{sub}} = 1024$, although we believe that its optimal value is problem-dependent.

Remark 2: We implemented **SM** in both dimensions and precisions, apart from 3D double precision where it is no longer advantageous. Here the shared memory constraint $16(m_1 + w)(m_2 + w)(m_3 + w) \leq 49000$, for widths $w > 8$ needed when $\varepsilon < 10^{-7}$, forces the bin volume to be tiny compared to the padded bin volume, dramatically increasing the number of global operations. We test only **GM-sort** in this case.

B. Interpolation

For interpolation, we use the same idea as **GM-sort** for spreading, with read and write memory operations reversed. Threads are assigned to nonuniform points in the permuted order $t(1), \dots, t(M)$ coming from bin-sorting. Thus the j th thread performs the task

$$\tilde{c}_{t(j)} = \sum_{l_1=0}^{n_1-1} \sum_{l_2=0}^{n_2-1} b_{l_1, l_2} \psi^{\text{per}}(l_1 h_1 - x_{t(j)}, l_2 h_2 - y_{t(j)}) .$$

We refer to the method by the same name, **GM-sort**, and use **GM** to refer to the unsorted version. The reason to bin-sort the nonuniform points is to coalesce the reads from the fine grid. Since there is no conflict between threads reading the same location in memory, this is fast; the benefit of applying an idea like **SM** to interpolation would be limited.

IV. PERFORMANCE TESTS

In this section, we report the performance of our GPU library, cuFINUFFT. We first show how the proposed spreading methods **GM-sort** and **SM** perform against **GM**. Then, we compare the performance of the interpolation step with (**GM-sort**) and without (**GM**) bin-sorting the nonuniform points. Finally, we show how the whole pipeline performance (spreading/interpolation, FFT, correction) of cuFINUFFT compares with the fastest known multithreaded CPU library FINUFFT [18], and two established GPU libraries CUNFFT [22] and gpuNUFFT [5].

All GPU timings are for a NVIDIA Tesla V100 (released in 2017), with memory bandwidth 900 GB/s. We compile all codes with GCC v7.4.0 and NVCC v10.0.130. Unless specified, single precision and $M_{\text{sub}} = 1024$ are used in all the tests.

Tasks. We present results for the following two nonuniform point distributions, which are extreme cases:

- “rand”: nonuniform points are independent and identically distributed uniform random variables across the entire periodic domain box $[-\pi, \pi]^d$, $d = 2, 3$.
- “cluster”: points are iid random in the small box $[0, 8h_1] \times \dots \times [0, 8h_d]$, recalling that h_i are the fine grid spacings.

We restrict to square/cube problems, i.e. $N_1 = N_2 (= N_3)$, being the most common application problem. We define problem *density* ρ to be the ratio of number of nonuniform points to number of upsampled grid points, i.e.,

$$\rho := \frac{M}{\prod_{i=1}^d n_i} = \frac{M}{\sigma^d \prod_{i=1}^d N_i} . \quad (16)$$

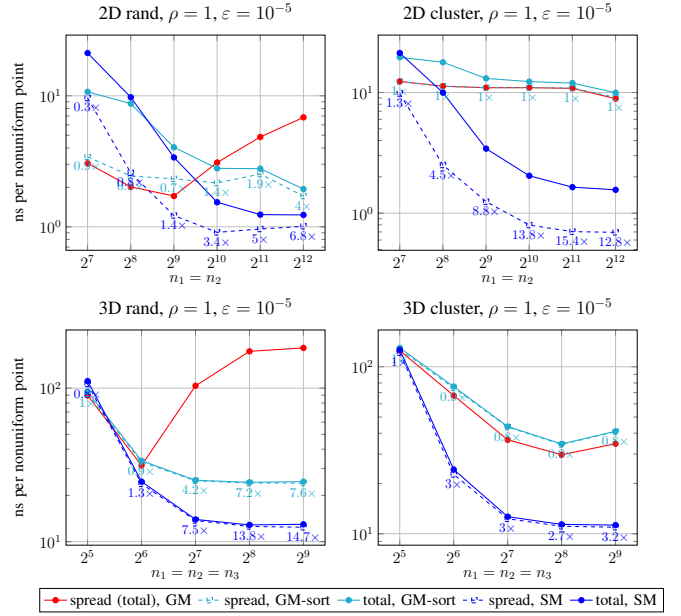


Fig. 2: Spreading method comparisons. Execution time per nonuniform point (smaller is better) is shown, for various fine grid sizes, and distributions “rand” and “cluster”, in 2D and 3D. For **GM-sort** (cyan) and **SM** (dark blue), the “total” time (solid lines) includes the precomputation (bin-sorting or subproblem setup), whereas the “spread” time (dotted lines) excludes this precomputation. The baseline **GM** (red) needs no precomputation. Annotations are the speedups over **GM**.

We report tests with ρ of order 1, since a) this is common in applications, and b) in the uncommon case $\rho \ll 1$ one ends up essentially comparing plain FFT speeds. In fact we have tested $\rho = 0.1$ and $\rho = 10$, as well as less extreme nonuniform point distributions; the conclusions are rather similar.

A. Spreading performance

For high-accuracy single precision ($\varepsilon = 10^{-5}$, i.e. $w = 6$), Fig. 2 compares our spreading methods **GM-sort** and **SM** against the baseline method **GM**, in 2D (top), 3D (bottom), and for “rand” (left) and “cluster” (right) distributions. Solid lines show total times (in nanoseconds per nonuniform point) including preprocessing (sorting) of new nonuniform points. Dotted lines show execution excluding this, so are relevant for *repeated transforms* with same set of nonuniform points.

We can see from the “rand” results that for large grids ($n_1 = n_2 \geq 2^{10}$ in 2D, or $n_1 = n_2 = n_3 \geq 2^7$ in 3D) bin-sorting brings a large gain. In the extreme case (the largest n_i tested), **GM-sort** is 3.9× faster than **GM** in 2D, and 7.6× faster in 3D. On the other hand, for small grids, because the memory accesses are already localized, we do not see any benefit of bin-sorting.

From the “cluster” results, since nonuniform points all reside in a small region, bin-sorting brings no benefit. However, we see the clear advantage of doing local spreading on shared memory, in that **SM** is up to 12.8× faster than **GM** in 2D,

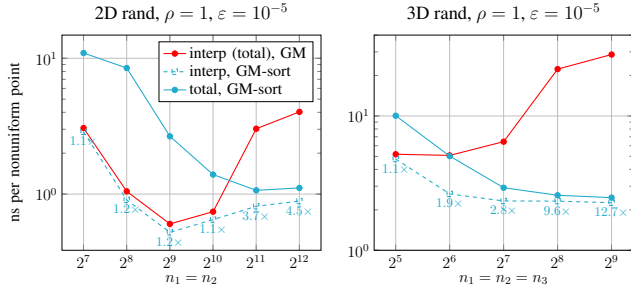


Fig. 3: Interpolation comparisons. Execution time per nonuniform point is shown, for various fine grid sizes, and distribution “rand”, in 2D and 3D. The “total” time (solid lines) includes the bin-sorting precomputation, whereas the “interp” time (dotted lines) excludes this precomputation.

and up to $3.2\times$ faster in 3D. The speedup in 3D is limited because the padding of the bins, especially in the z direction, grows the volume of global atomic adds needed in *Step 3*.

Comparing the dark blue curves in the left and right plots, we see a distribution-robust performance for **SM**, in that similar throughput is achieved for “rand” and “cluster” distributions. In comparison, **GM-sort** is on average $3.9\times$ slower in 2D comparing “cluster” and “rand” distributions. The 2D execution throughput (excluding precomputation) exceeds 10^9 points/sec for large grids, and is close even when including precomputation.

B. Interpolation performance

For the same accuracy, Fig. 3 compares interpolation method **GM-sort** against **GM** in 2D and 3D, for the “rand” distribution. (We exclude the “cluster” results, since, as with spreading, bin-sorting nonuniform points has no effect.) We see that again **GM-sort** improves the performance for large grids ($n_1 = n_2 \geq 2^{11}$ in 2D, or $n_1 = n_2 = n_3 \geq 2^6$ in 3D). It is $4.5\times$ faster than **GM** in 2D, and $12.7\times$ faster in 3D, for the largest n_i tested. A difference with spreading is that, because there are no global write conflicts, the execution time of **GM-sort** (excluding precomputation) never becomes slower than **GM**.

C. Benchmark comparisons against existing libraries

We now compare cuFINUFFT against the CPU library FINUFFT (which has already benchmarked favorably against other CPU libraries [18]), and GPU libraries CUNFFT [22] and gpuNUFFT [5]. For FINUFFT, we used a high-end compute node equipped with 512 GB RAM and two Intel Xeon E5-2680 v4 processors (released in 2016). Each processor has 14 physical cores at 2.40 GHz. We ran multithreaded FINUFFT with 28 threads (1 thread per physical core).

- cuFINUFFT version 1.0. We used host compiler flags `-fPIC -O3 -funroll-loops -march=native`.
- FINUFFT version 2.0.2. Compiler flags were `-O3 -funroll-loops -march=native -fcx-limited-range`. We fixed upsampling factor $\sigma = 2$ to match that used in cuFINUFFT.

- CUNFFT version 2.0. We compiled with fast Gaussian gridding (`-DCOM_FG_PSI=ON`). Default dimensions of thread blocks (`THREAD_DIM_X=16`, `THREAD_DIM_Y=16`) are used.

- gpuNUFFT version 2.1.0. We use its MATLAB interface. We used default host compiler flags (`-std=c++11 -fPIC`). We set `MAXIMUM_ALIASING_ERROR` to 10^{-6} to get more accurate results. We use the same sector width 8 and `THREAD_BLOCK_SIZE=256` as the demo codes.

We present three different timings for NUFFT executions:

- “total”: Execution time (per nonuniform point) for inputs and output on the GPU.
- “total+mem”: Execution time (per nonuniform point), including the time for GPU memory allocation plus transferring data from host to GPU and back.
- “exec”: Execution time (per nonuniform point) for a transform, after its nonuniform points have already been preprocessed. This is a subset of the “total” time. It is the relevant time for the case of multiple fixed-size transforms with a fixed set of nonuniform points, but new strength or coefficient vectors.

There is a constant start-up cost (about 0.1–0.2 second) for calling the cuFFT library, so to exclude it we add a dummy call of `cuFFTPlan1d` before calling `cuFINUFFT` or `CUNFFT`. For gpuNUFFT, in “total+mem”, we exclude the time for building the nonuniform FFT operator and creating the cuFFT plan. Note also that gpuNUFFT sorts the nonuniform points into sectors on the CPU and copies the arrays to the GPU when it builds the operator, so, to be generous, we do not include this in “total+mem” either. Finally, “total” is not shown for gpuNUFFT and CUNFFT, because gpuNUFFT takes CPU arrays as inputs and outputs, and CUNFFT allocates GPU memory in the initialization stage (`cunfft_init`) in a way that did not allow us to separate its timing.

We now discuss the results (Figures 4 to 7 and Table I). A wide range of relative ℓ_2 errors, ϵ , are explored by varying the requested tolerance, or kernel parameters (usually the width w), for each library. Error is measured against a ground truth of FINUFFT with tolerance $\epsilon = 10^{-14}$ for double precision runs, and $\epsilon = 6 \times 10^{-8}$ for single precision runs.

a) *Single precision comparisons*: Figs. 4 and 5 compare performance of both type 1 (top) and type 2 (bottom) in 2D (left), 3D (right), for all libraries in single precision. We can see from the top plots that for type 1, cuFINUFFT outperforms all other libraries. For type 1, the best performance is achieved using the **SM** method (dark blue). The “exec” time of cuFINUFFT (SM) in 2D is around $10\times$ faster than “exec” time of FINUFFT, independent of the accuracy; and in 3D, it is $3\text{--}12\times$ faster from high to low accuracy.

For type 2 (bottom plots), except for 2D low accuracy ($\epsilon > 10^{-2}$) where CUNFFT is comparable to cuFINUFFT, cuFINUFFT is again the fastest. Its “exec” time is $4\text{--}7\times$ and $6\text{--}8\times$ faster than the “exec” time of FINUFFT in 2D and 3D respectively.

In Fig. 6, we fix a tolerance $\epsilon = 10^{-2}$ (achievable by all libraries), and examine the effect of nonuniform point

| ϵ | $N_1 = N_2 = N_3$ | M | Method | “Exec” time (sec) | RAM (MB) | Speedup vs FINUFFT | Spread fraction (%) |
|------------|-------------------|--------------------|---------|-------------------|----------|----------------------|---------------------|
| 10^{-2} | 32 | 2.62×10^5 | GM-sort | 0.0009 | 381 | 5.9 \times | 94.7 |
| | | | SM | 0.0005 | 381 | 11.8 \times | 90.3 |
| | 256 | 1.34×10^8 | GM-sort | 0.33 | 6139 | 8.6 \times | 95.7 |
| | | | SM | 0.18 | 6141 | 16.1 \times | 91.9 |
| 10^{-5} | 32 | 2.62×10^5 | GM-sort | 0.0041 | 381 | 1.7 \times | 98.8 |
| | | | SM | 0.0028 | 381 | 2.6 \times | 98.5 |
| | 256 | 1.34×10^8 | GM-sort | 1.7 | 6139 | 2 \times | 99.2 |
| | | | SM | 0.87 | 6141 | 3.9 \times | 98.4 |

TABLE I: cuFINUFFT 3D type 1 NUFFT GPU memory usage, and “exec” time, for distribution “rand” and for two relative tolerances $\epsilon = 10^{-2}, 10^{-5}$. Speedup is computed relative to the “exec” time from FINUFFT. Spread fraction is the percentage of “exec” time spent on spreading. RAM is measured using `nvidia-smi`. For the baseline spreading method **GM**, RAM use is 381 MB for $N_i = 32$ and 5113 MB for $N_i = 256$).

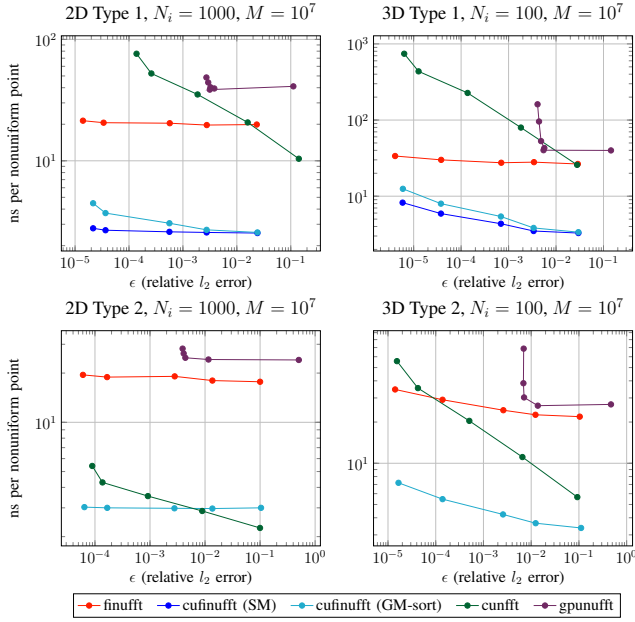


Fig. 4: Single precision NUFFT comparisons in 2D (left) and 3D (right), for type 1 (upper) and 2 (lower). “total+mem” (“total” for FINUFFT) time per nonuniform point vs accuracy is shown, for the named libraries, for the distribution “rand”.

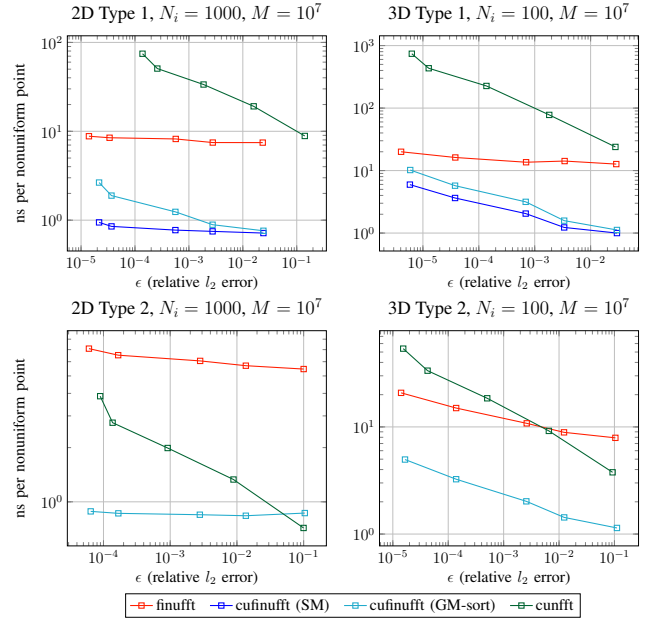


Fig. 5: Single precision comparisons in 2D and 3D. “exec” time per nonuniform point vs accuracy is shown for the tested libraries, except for `gpuNUFFT`. For explanation see caption of Fig. 4.

distribution and number of Fourier modes (fixing density $\rho = 1$) on library performance. From the top plots, for type 1, we observe distribution-robust performance in cuFINUFFT (SM), FINUFFT and `gpuNUFFT`. The others, cuFINUFFT (GM-sort) and CUNFFT, slow down when the points are clustered: for an intermediate problem size ($N_i = 2^9$), cuFINUFFT (GM-sort) “exec” is slowed by a factor of 3 when switching from “rand” to “cluster” to “rand”. Dramatically, CUNFFT is slowed by a factor of 200: it is very slow for clustered type 1 transforms.

For type 2 (lower plots in Fig. 6), the sensitivity to clustering is much weaker: all libraries tackle “cluster” at about the same speed they tackle “rand”, apart from cuFINUFFT which becomes 3–4 \times faster. While cuFINUFFT has similar “total+mem” time as CUNFFT, its “exec” time is 2–5 \times faster than that of CUNFFT. In 3D our detailed findings are quite

similar, and we do not show them.

Lastly, in Table I we detail the type 1 performance and RAM usage of cuFINUFFT in 3D, for two tolerances. We see again that higher speedup with respect to FINUFFT is achieved for *low accuracy* and *large problem sizes*. The spreading method **SM** achieves better performance, but at a cost of slightly more GPU RAM usage for large problems. Lastly, spreading is still the performance bottleneck for 3D type 1: it occupies over 90% of “exec” time for all accuracies and problem sizes.

b) *Double precision comparisons*: Fig. 7 compares the performance for both types in 2D and 3D, for all libraries (except `gpuNUFFT`, whose ϵ appears always to exceed 10^{-3}). We see from the top left plot that for 2D type 1, cuFINUFFT outperforms the others by 1–2 orders of magnitude. The best performance is achieved by **SM** (blue) for high accuracy ($\epsilon \leq$

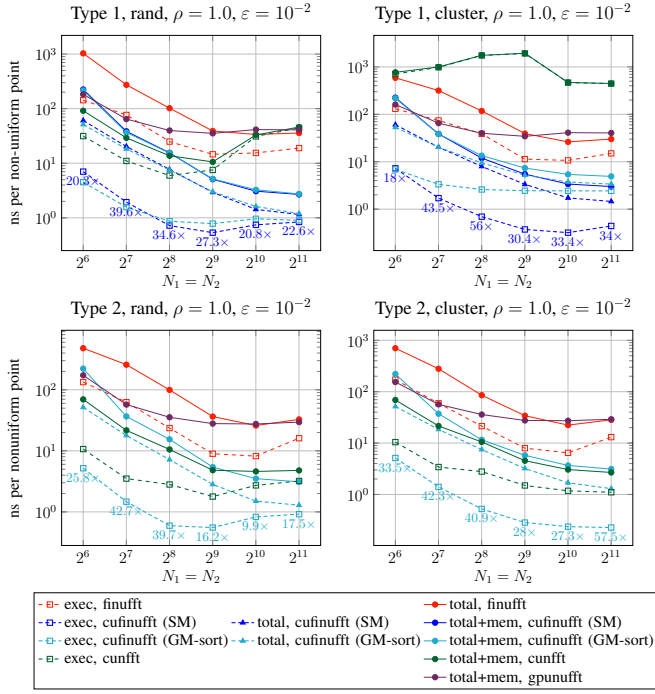


Fig. 6: Detailed 2D Type 1 (top) and 2 (bottom) NUFFT comparisons (single precision). Execution time per nonuniform point vs number of Fourier modes are shown for the named libraries, comparing “rand” (left) and “cluster” (right). Annotations give the speedup of “exec” of cuFINUFFT (SM) for type 1, cuFINUFFT (GM-sort) for type 2, vs “exec” of FINUFFT.

10^{-5}) and by **GM-sort** (cyan) for low accuracy. The “exec” speedup of cuFINUFFT (taking the faster of **SM** and **GM-sort**), vs FINUFFT, ranges from 4–11 \times . From the top right plot, for 3D type 1, cuFINUFFT is only faster than FINUFFT for relative error $\epsilon \geq 10^{-10}$, merely matching its speed at the highest accuracies. From the bottom plots, for type 2, cuFINUFFT is always the fastest, and by a large factor at high accuracies. The “exec” time of cuFINUFFT is on average 6 \times faster than that FINUFFT in both dimensions. In 2D, and low-accuracy 3D, we see that host-to-device transfer dominates: a several-fold speedup is available by maintaining data on the GPU.

V. MULTI-GPU APPLICATIONS

Finally, we illustrate the multi-GPU performance of cuFINUFFT in 3D coherent X-ray image reconstruction. Single particle imaging is a technique whereby the 3D electron density of a molecule may be recovered at sub-nanometer resolution from a large ($\leq 10^5$) set of 2D far-field diffraction images, each taken in a single shot of a free-electron laser with a random *unknown* molecular orientation [9]. Each 2D image measures the squared magnitude Fourier transform of the density on an (Ewald sphere) slice passing through the origin; see Fig. 8. The *multitiered iterative phasing* (M-TIP) algorithm is used for reconstruction [9]. Broadly speaking, one starts with a Fourier

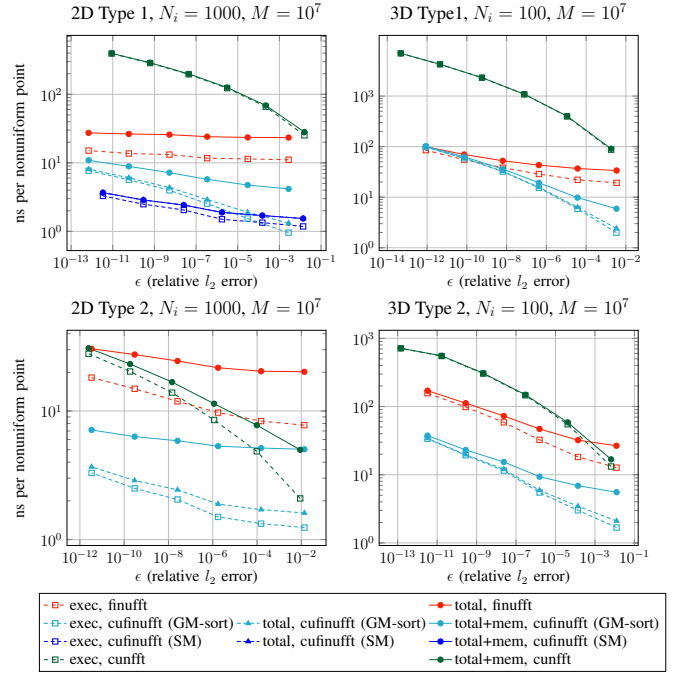


Fig. 7: Double precision comparisons. All three timings “exec”, “total”, and “total+mem” are shown. For more explanation see caption of Fig. 4.

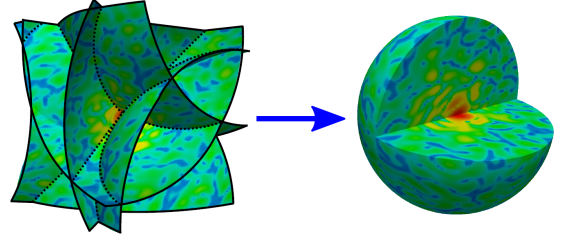


Fig. 8: M-TIP merging step: 3D Fourier transform data, collected on multiple Ewald sphere slices with arbitrary orientations, is merged onto a single uniform grid. Image credit: Jeffrey Donatelli (Lawrence Berkeley National Laboratory).

transform estimate on a 3D Cartesian grid, and estimated orientations, then iterates the following four steps:

- i) “Slicing”: a 3D type 2 NUFFT is used to evaluate the Fourier transform on a large set of Ewald sphere slices.
- ii) Orientation matching: adjust each slice orientation using its 2D image data.
- iii) “Merging”: solve for 3D Fourier transform data matching the 2D images on known slices, as in Fig. 8; this needs two 3D type 1 NUFFTs.
- iv) Phasing: find the most likely phase of the 3D Fourier transform to give a real-space density of known support.

The code acceleration strategy, a part of the Exascale Computing Project, has been to offload the intensive steps i)–iii) to GPUs.

A. Work management and the cuFINUFFT Python interface

We use MPI to manage parallel processes, via the mpi4py package. Each MPI *rank* (i.e. process) is assigned some data to handle and a GPU. Since slicing and merging are linear operations, we can scatter (`mpi4py.scatter`) before the slicing step, and reduce (`mpi4py.reduce`) after the merging step. In modern HPC environments each compute node is furnished with several GPUs—e.g. NERSC’s Cori GPU system has 8 NVIDIA V100 per node, while OLCF’s Summit system has 6 V100 per node. Thus, depending on the ratio of GPUs to CPU cores, we can have several MPI ranks share the same GPU. For M-TIP, load balancing is simple because each rank does (roughly) the same amount of work, thus we assign GPUs in a round-robin fashion. We use PyCUDA to transfer data between device and host. This allows cuFINUFFT to access `numpy.ndarray` objects as `double []`, hence requiring no specialized API calls to convert data between Python and cuFINUFFT. In order to ensure that the PyCUDA API sends the data to the correct device, we manually define the device context. Here is an example of taking a type 1 3D NUFFT with nonuniform coordinates *X*, *Y*, *Z*, and strengths *nuvect* (note: `mpi4py` provides `rank`):

```
from cufinufft import cufinufft
from pycuda.gpuarray import GPUArray, to_gpu

# Initialize GPU using round-robin assignment
GPUS_PER_NODE = 8 # Cori GPU
device_id = rank % GPUS_PER_NODE

pycuda.driver.init()
device = pycuda.driver.Device(device_id)
ctx = device.make_context()

ugrid_gpu = GPUArray(shape) # Memory for result

plan = cufinufft(1, shape, eps=eps,
                 gpu_device_id=device_id)
plan.set_pts(to_gpu(X), to_gpu(Y), to_gpu(Z))
plan.execute(to_gpu(nuvect), ugrid_gpu)
```

The cuFINUFFT Python interface allows the user to assign a `cufinufft` plan to a specific device by setting the `gpu_device_id` option. See the documentation on GitHub for details. The M-TIP code requires a tolerance of $\varepsilon = 10^{-12}$.

B. cuFINUFFT Performance on Cori GPU and Summit

To perform a single-node weak scaling study we used, per rank, the NUFFT problem sizes in Table II, which correspond to 10^3 images. The table shows the average wall-clock time spent performing NUFFTs during slicing (type 2 NUFFT) and merging (type 1 NUFFT) steps for one M-TIP iteration. Comparing the GPU wall-clock times (including data movement) to the equivalent problem running on 40 CPU threads on a single Intel Skylake Cori GPU node (using the FINUFFT code), we find that cuFINUFFT on a single GPU is roughly similar to the CPU times, while for the larger problem distributed over the whole node (multi-GPU) it is 5–12 \times faster than on the CPU.

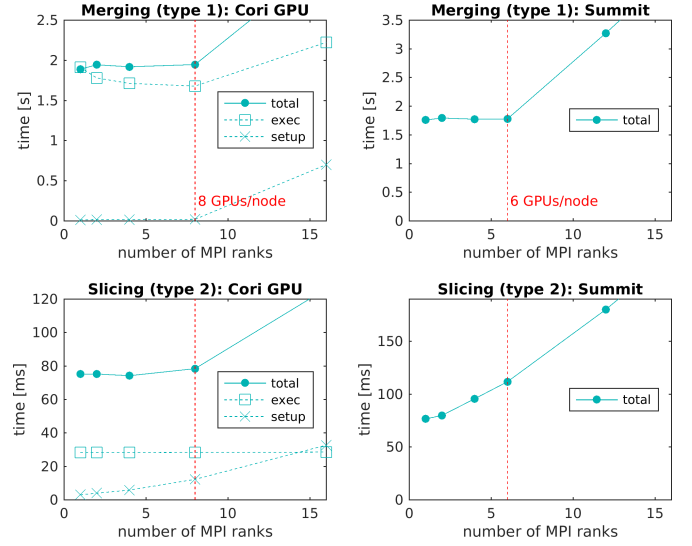


Fig. 9: Single-node multi-GPU weak scaling on NERSC Cori GPU (left) and OLCF Summit (right). We achieve close to ideal weak scaling (flat lines) up to a number of ranks matching the number of GPUs on the node (vertical dotted line).

Fig. 9 shows the weak scaling performance on single nodes of NERSC’s Cori GPU and OLCF’s Summit. Each rank is given the single-rank problem size from Table II. Solid lines show the total time including host-device transfer. Crosses show setup time (plan, input and nonuniform point sorting), while squares show NUFFT execution time. In all cases (except type 2 on Summit), we see ideal weak scaling up to one rank per GPU. We found that enabling multi-process service (MPS) made no measurable difference. We see rapid deterioration of weak scaling once each GPU is used by more than one rank, suggesting that cuFINUFFT uses each GPU to capacity.

VI. CONCLUSIONS

We presented a general-purpose GPU-based library for nonuniform fast Fourier transforms: cuFINUFFT. It supports both transforms of type 1 (nonuniform to uniform) and type 2 (uniform to nonuniform), in 2D and 3D, with adjustable accuracies. By using an efficient kernel function, sorting the nonuniform points into bins, and leveraging shared memory to reduce write collisions, cuFINUFFT obtains a significant speedup compared to established CPU- and GPU-based libraries. On average, we observe a speedup of one order of magnitude over the FINUFFT parallel CPU library. We also observe up to an order of magnitude speedup compared to the CUNFFT GPU library, and more in the case of clustered type 1 transforms. We also see excellent multi-GPU weak scaling in an iterative 3D X-ray reconstruction application.

There are several directions for future work. One is extending the library to include 1D and type 3 transforms, but also to use smaller upsampling factors σ , which can significantly reduce memory size. It is also worth exploring supporting other GPUs via a library that provides a unified hardware API, such as OCCA or Kokkos.

| Task | Uniform grid (per rank) $N_1 = N_2 = N_3$ | Nonuniform points (per rank) M | Density ρ | Parallelism | CPU time [s] (Intel Skylake) | GPU time [s] (Cori GPU) | GPU time [s] (Summit) |
|---------------------|--|-------------------------------------|-------------------|---------------------------|---------------------------------|-----------------------------|-----------------------------|
| Slicing (type 2) | 41 | 1.02×10^6 | 1.86 | single-rank whole-node | 0.11 0.95 | 0.075 (1.5×) 0.078 (12×) | 0.076 (1.5×) 0.11 (8.6×) |
| Merging (type 1) | 81 | 1.64×10^7 | 3.85 | single-rank whole-node | 1.62 9.97 | 1.89 (0.9×) 1.94 (5.1×) | 1.76 (0.9×) 1.76 (5.7×) |

TABLE II: Problem sizes and average NUFFT wallclock times for a representative M-TIP iteration: NERSC’s Cori GPU (Intel Skylake with 8 NVIDIA V100’s) and OLCF’s Summit (IBM Power9 with 6 NVIDIA V100’s). The problem sizes are fixed per MPI rank. The CPU code is FINUFFT v1.1.2, with 40 threads on the Intel Skylake. The speedup ratio of single- or multi-GPU cuFINUFFT over the CPU code is shown in parentheses. Rows labeled “whole-node” use problems scaled up by the number of GPUs per node—8 for Cori GPU and 6 for Summit—using this same number of ranks (i.e., one rank per GPU).

VII. ACKNOWLEDGMENTS

This research used resources of two U.S. Department of Energy Office of Science User Facilities: NERSC and OLCF (contract # DE-AC02-05CH11231 and DE-AC05-00OR22725). Johannes Blaschke’s work is supported by the DOE OOS in part through DOE’s ECP ExaFEL project, Project # 17-SC-20-SC, a collaborative effort of the Office of Science and the National Nuclear Security Administration. We thank Jeff Donatelli, Chuck Yoon, and Christine Sweeney for work on the M-TIP code, and Georg Stadler for helpful suggestions at various stages of this work. The work was conducted while Joakim Andén was a visiting scholar at CCM. Part of Yuhsuan Shih’s work was supported through the CCM internship program. The Flatiron Institute is a division of the Simons Foundation. The work of Garrett Wright was supported by the Moore Foundation Award #9121 Cryo-EM Software Grant.

REFERENCES

- [1] J. A. Fessler and B. P. Sutton. Nonuniform fast Fourier transforms using min-max interpolation. *IEEE Trans. Signal Process.*, 51(2):560–574, 2003.
- [2] K. Zhang and J. U. Kang. Graphics processing unit accelerated non-uniform fast Fourier transform for ultrahigh-speed, real-time Fourier-domain OCT. *Opt. Express*, 18(22):23472–87, 2010.
- [3] L. Wang, Y. Shkolnisky, and A. Singer. A Fourier-based approach for iterative 3D reconstruction from cryo-EM images. *arXiv preprint arXiv:1307.5824*, 2013.
- [4] A. H. Barnett, M. Spivak, A. Pataki, and L. Greengard. Rapid solution of the cryo-EM reconstruction problem by frequency marching. *SIAM J. Imaging Sci.*, 10(3):1170–1195, 2017.
- [5] D. Střelák, C. Ó. S. Sorzano, J. M. Carazo, and J. Filipovič. A GPU acceleration of 3-D Fourier reconstruction in cryo-EM. *Int. J. High Perform. Comput. Appl.*, 33(5):948–959, 2019.
- [6] J. Andén and A. Singer. Structural variability from noisy tomographic projections. *SIAM J. Imaging Sci.*, 11(2):1441–1492, 2018.
- [7] Z. Zhao, Y. Shkolnisky, and A. Singer. Fast steerable principal component analysis. *IEEE Trans. Comput. Imaging*, 2(1):1–12, 2016.
- [8] P. Arras, M. Reinecke, R. Westermann, and T. A. Enßlin. Efficient wide-field radio interferometry response, 2020. *arXiv:2010.10122; in press, Astron. Astrophys.*
- [9] J. J. Donatelli, J. A. Sethian, and P. H. Zwart. Reconstruction from limited single-particle diffraction data via simultaneous determination of state, orientation, intensity, and phase. *Proc. Natl. Acad. Sci.*, 114(28):7222–7227, 2017.
- [10] A. H. Barnett. Efficient high-order accurate Fresnel diffraction via areal quadrature and the nonuniform FFT. *J. Astron. Telesc. Instrum. Syst.*, 7(2):021211, 2021.
- [11] Z. Gimbutas and S. Veerapaneni. A fast algorithm for spherical grid rotations and its application to singular quadrature. *SIAM J. Sci. Comput.*, 5(6):A2738–A2751, 2013.
- [12] L. af Klinteberg, T. Askham, and M. C. Kropinski. A fast integral equation method for the two-dimensional Navier–Stokes equations. *J. Comput. Phys.*, 409:109353, 2020.
- [13] R. Salomon-Ferrer, A. W. Gotz, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. *J. Chem. Theory Comput.*, 9(9):3878–3888, 2013.
- [14] A. M. Fiore, F. Balboa Usabiaga, A. Donev, and J. W. Swan. Rapid sampling of stochastic displacements in Brownian dynamics simulations. *J. Chem. Phys.*, 146(12):124116, 2017.
- [15] A. Dutt and V. Rokhlin. Fast Fourier transforms for nonequispaced data. *SIAM J. Sci. Comput.*, 14:1369–1393, 1993.
- [16] L. Greengard and J.-Y. Lee. Accelerating the nonuniform fast Fourier transform. *SIAM Review*, 46(3):443–454, 2004.
- [17] J. Keiner, S. Kunis, and D. Potts. Using NFFT 3 – a software library for various nonequispaced fast Fourier transforms. *ACM Trans. Math. Software*, 36(4), 2009.
- [18] A. H. Barnett, J. Magland, and L. af Klinteberg. A parallel nonuniform fast Fourier transform library based on an “exponential of semicircle” kernel. *SIAM J. Sci. Comput.*, 41(5):C479–C504, Jan 2019.
- [19] L. Greengard, J.-Y. Lee, and S. Inati. The fast sinc transform and image reconstruction from nonuniform samples in k -space. *Commun. Appl. Math. Comput. Sci.*, 1(1):121–131, 2006.
- [20] G. Steidl. A note on fast Fourier transforms for nonequispaced grids. *Adv. Comput. Math.*, 9:337–352, 1998.
- [21] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.
- [22] S. Kunis and S. Kunis. The nonequispaced FFT on graphics processing units. *Proc. Appl. Math. Mech.*, 12:7–10, 2012.
- [23] J.-M. Lin. Python non-uniform fast Fourier transform (PyNUFFT): An accelerated non-Cartesian MRI package on a heterogeneous platform (CPU/GPU). *J. Imaging*, 4(3):51, 2018.
- [24] F. Knoll, A. Schwarzl, C. Diwok, and D. K. Sodickson. gnuNUFFT—an open-source GPU library for 3D gridding with direct MATLAB interface. *Proc. Intl. Soc. Mag. Reson. Med.*, 22:4297, 2014.
- [25] J. Gai et al. More IMPATIENT: A gridding-accelerated Toeplitz-based strategy for non-Cartesian high-resolution 3D MRI on GPUs. *J. Parallel Distrib. Comput.*, 73(5):686–697, 2013.
- [26] A. H. Barnett. Aliasing error of the $\exp(\beta\sqrt{1-z^2})$ kernel in the nonuniform fast Fourier transform. *Appl. Comput. Harmon. Anal.*, 51:1–16, 2021.
- [27] A. Gregerson. Implementing fast MRI gridding on GPUs via CUDA, 2008. NVIDIA Whitepaper.
- [28] D. S. Smith, S. Sengupta, S. A. Smith, and E. B. Welch. Trajectory optimized NUFFT: Faster non-Cartesian MRI reconstruction through prior knowledge and parallel architectures. *Magn. Reson. Med.*, 81(3):2064–2071, 2019.
- [29] S.-C. Yang, H.-J. Qian, and Z.-Y. Lu. A new theoretical derivation of NFFT and its implementation on GPU. *Appl. Comput. Harmon. Anal.*, 44(2):273–293, 2018.
- [30] J.-Y. Lee and L. Greengard. The type 3 nonuniform FFT and its applications. *J. Comput. Phys.*, 206:1–5, 2005.