The Flatiron Institute Nonuniform Fast Fourier Transform: Project Summary, December 2023

Joakim Andén,[*] Alex H. Barnett,[†] Robert Blackwell,[‡] Libin Lu,[†] and Yu-Hsuan Shih[§]

1 Nonuniform fast Fourier transforms and their uses

Nonuniform fast Fourier transforms (NUFFTs, or non-equispaced FFTs) generalize the fast Fourier transform (FFT) to irregularly spaced (nonuniform) sample points. Since non-Cartesian measurements or quadrature nodes are common in science and engineering, NUFFTs have wide applications, including Fourier imaging, image processing, diffraction modeling, time series analysis, interpolation between grids, high-order and spectral methods for partial differential equations (PDEs), fast N-body Ewald codes, oscillatory integral transforms, and statistics (e.g., Gaussian process regression).

In one dimension (1D), given arbitrary ("nonuniform") source points $x_j \in \mathbb{R}$, $j = 1, \ldots, M$, with strengths $c_j \in \mathbb{C}$, a desired number $N \in \mathbb{N}$ of Fourier coefficients, and a tolerance $\varepsilon > 0$, the type-1 NUFFT approximates the exponential sums

$$f_k = \sum_{j=1}^{M} c_j e^{ikx_j}, \qquad k \in \mathbb{Z},\ -N/2 \le k < N/2,$$

with relative accuracy ε. The d-dimensional version maps strengths at points in $\mathbb{R}^d$ to a cuboid of coefficients. The naive cost would be $\mathcal{O}(MN)$; yet NUFFTs use a fast algorithm (see below) to achieve a quasilinear cost of $\mathcal{O}(M \log^d \varepsilon^{-1} + N \log N)$, often only a few times slower than an FFT. The adjoint of the above transform is called the type-2 NUFFT; it evaluates a given d-dimensional Fourier series at arbitrary target points. In the more general, often more expensive, type-3 NUFFT, both source and target points are arbitrary (nonuniform). For background see definitions and references.

A major class of 2D and 3D applications is image/volume reconstruction from non-Cartesian Fourier measurements, including magnetic resonance imaging (MRI), cryo-electron microscopy (cryo-EM), synthetic aperture radar (SAR), coherent X-ray diffraction, and very long baseline interferometry (VLBI) in radioastronomy. In each case, an iterative solution to a least-squares linear or nonlinear system is needed, where each iteration combines type-1 and type-2 transforms involving all data. Thus NUFFTs dominate the cost, and their efficiency is crucial in order to guarantee a reasonable reconstruction time; this is especially true in time-critical clinical MRI workflows.

2 Our library: implementation, documentation, and tests

The Flatiron Institute NUFFT ("FINUFFT") library contains shared-memory parallel CPU (C++ OpenMP) and GPU (CUDA) implementations of type-1, type-2, and type-3 (for CPU only) transforms in 1D, 2D, and 3D, in both single- and double-precision arithmetic, plus many language bindings.

Implementation and performance. In broad strokes, we use well-known algorithms. Type-1 comprises: i) spreading onto a fine grid using a tensor-product kernel w grid points wide, ii) executing

[*]Department of Mathematics, KTH Royal Institute of Technology, Sweden
[†]Center for Computational Mathematics, Flatiron Institute, New York, USA
[‡]Scientific Computing Core, Flatiron Institute, New York, USA
[§]Nvidia, Santa Clara, USA (work done while at New York University, USA)

an FFT (we use FFTW3) on the fine grid, and iii) returning the low-frequency block of the fine grid, elementwise divided (deconvolved) by the kernel Fourier transform. Type-2 reverses these steps, while type-3 is a type-1 with its FFT replaced by a type-2. For reasons of efficiency we augmented the standard algorithms with several innovations, including:

1. A simple kernel function of the form $e^{\beta\sqrt{1-x^2}}$ with provable ℓ_1-ℓ_∞ error bounds $\mathcal{O}(e^{-cw})$, where the exponential rate c is optimal (see ACHA analysis paper).
2. Kernel evaluation via piecewise-polynomial Horner's rule on intervals aligning with the fine grid, enabling SIMD auto-vectorization with respect to the width w.
3. Improved fine-grid caching by traversing the points $\{x_j\}_{j=1}^M$ in an order given by first (parallel) bin-sorting into small cuboid bins.
4. A parallel spreader avoiding write collisions. Each thread spreads $\sim 10^4$ points to a private subgrid, then copy-adds this to the fine grid. This load-balances even when points are clustered.
5. SIMD-aware spread/interpolation loops in 2D and 3D, and point batching (we have explored SIMD intrinsics but avoid them for CPU flexibility).
6. The GPU code also introduced spreading using small custom subgrids that fit in fast *shared memory* (49 kB per thread block; see IPDSPW paper, which won PDSEC21 best paper prize).

We extensively benchmarked the CPU performance of FINUFFT against four other popular NUFFT codes, finding that FINUFFT is typically 10× faster than the closest competition, yet needs 10× less RAM overhead than codes relying on kernel precomputation. Throughput on a typical laptop is 10^6 to 10^8 points/sec, depending on type, density (ratio of nonuniform to uniform points), dimension, and ε. Last year a Julia code beat us in some cases, but by less than a factor of 1.7 when multithreaded. Our GPU benchmarks showed of order 10× speed-up over two popular GPU codes and our own CPU code on a workstation. On a gaming GPU (e.g., Nvidia GeForce RTX 3070), 2D transforms at $\varepsilon = 10^{-3}$ exceed 10^9 points/sec (excluding host–device transfer).

Interface design. We provide bindings to five languages, namely: C, Fortran, MATLAB/Octave, Python, and Julia for the CPU code; and C and Python for the GPU code. Our interfaces have carefully evolved over 6 years to provide the simplest possible interface for the new user, while allowing experts more control and performance. In most languages we thus have three styles:

- Single-function calls to the nine transforms (three dimensions × three types) in both precisions.
- "Vectorized" transforms on a *batch* of many strength or coefficient vectors with the same points. Batched FFTs and amortized planning and sorting can give several-times speed-ups here.
- A four-stage "guru" interface: create plan object (one transform or a batch), set nonuniform points, execute, destroy. This has the most flexibility and also simplifies language bindings.

In each case, users may change parameters from their defaults, e.g. upsampling factor or kernel evaluation method. That being said, we have chosen these defaults in order to achieve good performance for a wide range of problems. Crucially, the user is therefore *not forced to contemplate kernel types nor parameters* (as in some other codes): they simply request a tolerance $\varepsilon \in [10^{-15}, 10^{-1}]$, and the most efficient parameters are automatically chosen to achieve this expected accuracy.

Installation and maintenance. For local compilation, we maintain GNU `make` and `cmake` settings for three operating systems: Linux, macOS, and Windows. In Python our PyPI package allows one-line `pip` installation (similarly in Julia). Internal documentation is ubiquitous at the function level; user-facing APIs live only on the website (sourced from reStructuredText)—this follows a *single source of truth* (SSOT) idea for easy maintenance. We follow 14 of the 17 xSDK community policies.

Documentation. Extensive web-facing docs (26 pages, or as a 124-page PDF) cover installation, definitions, quick-start examples from all languages, APIs, tutorials, references, and more. Crucial features include: 1) a clear *mathematical description* of what the library computes[1] 2) guidance on when *not to use* an NUFFT; 3) troubleshooting speed issues and mathematical misunderstandings.

Examples and tutorials. The goal is to teach. Thus we avoid domain-specific jargon (MRI, etc.), start as simple as possible (see, e.g., Python), then build in complexity. More advanced tutorials blend discussion, code, output, plots, and further reading. Many of these have arisen out of discussions with users in the FINUFFT community.

Tests. We provide CPU and GPU test drivers for mathematical correctness (including edge/invalid cases and their error codes), and limited performance tests. Continuous integration (CI) on GitHub Workflows and Jenkins (for GPU) tests branches and pull requests for correctness of all executables built via `make` and via `cmake`, on the six platform-compiler combinations listed above.

3 Impact of the library

A variety of research results and codes rely on the FINUFFT CPU and/or GPU library. For example:

- The SMILI code used 2D FINUFFT (Fortran wrapper) to reconstruct the famous M87 black hole image from EHT data using iterative VLBI (K. Akiyama et al. at MIT).
- In MRI, Lecoeur et al. use GPU FINUFFT to accelerate 4D-MRI by 17×. Neurospin (Paris) users state "*finufft and cufinufft are the most stable implementations we can find out there.*"
- The GPU library is being run on 6144 Nvidia A100 GPUs (the NERSC-9 "Perlmutter" supercomputer) for a particle-in-Fourier method for plasma simulation by S. Muralikrishnan.
- In cryo-EM, fast processing/alignment of 2D images and 3D volumes is enabled by FINUFFT CPU and GPU (via Python): see ASPIRE (see support letter), and EM-Align.
- Exoplanet discovery uses power-spectrum estimation of nonequispaced time series, via FINUFFT 1D type-3: NWelch (Python wrappers) Multitaper.jl (Julia wrappers).
- Harness et al. use 2D FINUFFT for Fresnel diffraction modeling in an algorithm 10000× faster than pre-2021 edge-integral methods.
- Multi-GPU FINUFFT accelerates single-particle reconstruction from X-ray diffraction data at ExaFEL (LBNL) by 5 to 12× (see IPDSPW paper, Sec. 5).
- The DISCUS package manual explains (p. 161) that 3D type-1 FINUFFT (via Fortran) enables two orders of magnitude speed-up in X-ray crystal diffraction modeling.
- The community has written, and uses, machine learning style differentiable combined-CPU/GPU wrappers including: TensorFlow, JAX, and PyTorch.

While metrics should be taken with a large pinch of salt, we note 210 Google Scholar citations to our three papers, 67 GitHub forks and 224 stars (78 stars for the legacy GPU repository), and on the order of 1700 PyPI downloads/month and 200 unique GitHub clones/month, at the time of writing.

Acknowledgements. Apart from the five above authors, the FINUFFT development community includes about twenty other contributors. Especially credit-worthy are: **Jeremy Magland** (Flatiron Institute) who built an early multithreaded type-1 spreader; **Ludvig af Klinteberg** (now at Mälardalen U.) who added some SIMD acceleration and built the Julia wrapper; **Andrea Malleo** (now at Bloomberg) who prototyped the guru interface; **Garrett Wright** (Princeton) who helped with the dual-precision build and Python wrappers. Shih and Malleo were CCM summer interns mentored by Barnett, providing training of computational mathematicians from under-represented groups. The Flatiron Institute is a division of the Simons Foundation.

[1] We note that many open-source libraries—including NUFFTs—cannot be widely used because their documentation has no mathematical formulae stating exactly what they do.