

gNUFFTW: Auto-Tuning for High-Performance GPU-Accelerated Non-Uniform Fast Fourier Transforms

Teresa Ou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-90

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-90.html>

May 12, 2017



Copyright © 2017, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**gNUFFTW: Auto-Tuning for High-Performance
GPU-Accelerated Non-Uniform Fast Fourier Transforms**

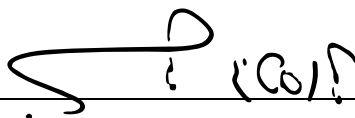
by Teresa Ou

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:



Professor Michael Lustig
Research Advisor

05/12/2017

(Date)

* * * * *



Professor Laura Waller
Research Advisor

5/12/17

(Date)

gNUFFT: Auto-Tuning for High-Performance GPU-Accelerated Non-Uniform Fast Fourier Transforms

Teresa Ou

Abstract—Non-uniform sampling of the Fourier transform appears in many important applications such as magnetic resonance imaging (MRI), optics, tomography and radio interferometry. Computing the inverse often requires fast application of the non-uniform discrete Fourier transform (NUDFT) and its adjoint operation. Non-Uniform Fast Fourier Transform (NUFFT) methods, such as gridding/regridding, are approximate algorithms which often leverage the highly-optimized Fast Fourier Transform (FFT) and localized interpolations. These approaches require selecting several parameters, such as interpolation and FFT grid sizes, which affect both the accuracy and runtime. In addition, different implementations lie on a spectrum of precomputation levels, which can further speed up repeated computations, with various trade-offs in planning time, execution time and memory usage. Choosing the optimal parameters and implementations is important for performance speed, but difficult to do manually since the performance of NUFFT is not well-understood for modern parallel processors. Inspired by the FFTW library, we demonstrate an empirical auto-tuning approach for the NUFFT on General Purpose Graphics Processors Units (GPGPU). We demonstrate order-of-magnitude speed improvements with auto-tuning compared to typical default choices. Our auto-tuning is implemented in an easy to use proof-of-concept library called gNUFFT, which leverages existing open-source NUFFT packages, cuFFT and cuSPARSE libraries, as well as our own NUFFT implementations for high performance.

Keywords—non-uniform, non-Cartesian, FFT, NUFFT, GPU, auto-tuning, image processing.

I. INTRODUCTION

Non-uniform sampling of the Fourier transform appears in many important applications such as magnetic resonance imaging (MRI) [1]–[3], optics [4]–[6], tomography [7]–[9], and radio interferometry [10]–[12]. When samples are acquired on a non-uniform grid in the frequency domain, computing the inverse non-uniform discrete Fourier transform (NUDFT) is generally non-trivial. Exact solutions require inverting the large NUDFT [13], [14] or repeatedly computing the NUDFT and its adjoint operations in iterative minimization algorithms [15]–[17]. Approximate solutions use the adjoint NUDFT with sampling density correction functions [18], or attempt to approximate the inverse NUDFT by other means [19].

The Fast Fourier Transform (FFT) exploits the regular structure of computing the DFT between Cartesian grids in both image and frequency domains to reduce computation. Such structure is broken when the frequency domain is not uniformly sampled. Non-Uniform Fast Fourier Transform (NUFFT) methods are a set of fast approximate algorithms for

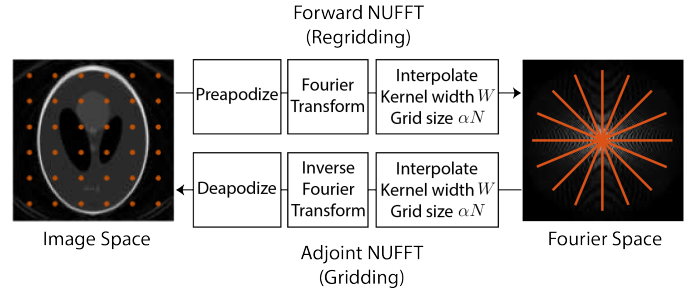


Fig. 1. Overview of gridding- and regridding-based approaches for NUFFT, which are fast approximate algorithms for computing NUDFT and its adjoint operation. These approaches leverage the highly optimized FFT with additional interpolation and weighting steps.

computing the NUDFT [15], [20]–[28]. Allowing for approximation error helps to reduce the algorithmic complexity of NUFFT. While several variants of NUFFT exist, the most commonly used are gridding/regridding-based approaches, shown in Fig. 1, which leverage the highly optimized FFT with additional interpolations and weighting steps [22], [26]–[29].

Gridding-based approaches require selecting parameters for the Cartesian grid size and interpolation kernel width, which affect both accuracy and runtime of the NUFFT. While the accuracy of the NUFFT is well-understood, the implementation and performance on modern parallel processors is still an active area of exploration. Here, we focus on Graphics Processing Units (GPU), which are increasingly used for image processing, due to their massively parallel architecture.

NUFFT implementations are less highly optimized than FFT libraries such as FFTW [30] and CUFFT [31]. Due to the complexity of modern processor architectures, FFTW uses auto-tuning to optimize FFTs by empirically searching over multiple decompositions, algorithms, and low-level optimizations, and measuring their execution times. Inspired by FFTW [30], we propose to optimize NUFFT over the space of NUFFT algorithm implementations and their associated parameters.

The main result of this work is that auto-tuning can accelerate the computation of NUFFT operations by an order of magnitude over default parameters for specific algorithms. Auto-tuning is therefore an easy way to speed up any existing NUFFT implementations. In addition, when memory restrictions exist, or when the same NUFFT is repeatedly executed, significant gains in speed can be obtained by optimizing over algorithms with different levels of precomputation. Finally, we

present a software library for computing NUFFTs on the GPU with a prototype auto-tuning framework. Similar to FFTW, our software library uses separate planning and execution phases. Our library uses the concept of empirical performance measurement during the planning phase to identify the parameter values and NUFFT implementations that result in the fastest runtime [32]. We provide implementations of NUFFT algorithms that span a range of levels of precomputation, which trade off planning time, execution time, and memory requirements. Additional data structures that facilitate execution may be precomputed during the planning phase and stored in memory.

The proposed library is portable, easy to use, and fast across various hardware architectures, and we leverage highly optimized routines from the cuFFT and cuSPARSE libraries [31], [33]. We analyze the performance of a range of NUFFT problems, to understand how to choose the parameter values and algorithms that will yield the fastest NUFFT computations.

II. RELATED WORK

The computation and accuracy of NUFFT have been well-studied in prior work [26], [27], [29], [34]. Jackson et al. [34] compare different interpolation kernels and evaluate the amount of aliased energy produced in NUFFT. Beatty et al. [29] develop a method to select oversampling ratios below the conventional value of two and use presampled interpolation kernels to reduce memory requirements and runtime. The notion of maximum aliasing amplitude is introduced as a way to estimate the NUFFT approximation error and choose the appropriate combination of parameters that achieve the desired accuracy. We draw from their techniques in our implementation of NUFFT.

Prior work on the implementation of NUFFT in software includes the acceleration of NUFFT on the GPU [35]–[38], methods for partial precomputation [36], [39], [40], and Toeplitz-based approaches [41]–[44]. Several software libraries for computing the NUFFT exist. NFFT [45], [46] and PNFFT [47] are software libraries for computing NUFFT on the CPU, with serial, multi-core and distributed implementations. NFFT allows the user to choose different interpolation kernel functions, levels of precomputation, and parameter values. `gpuNUFFT` [38] is a GPU-based library that implements partial precomputation with load balancing. `PowerGrid` [48] is a highly accessible library written in OpenACC, which allows it to be used on various accelerators, including GPUs, multi-core CPUs, and distributed systems, although this flexibility comes at the cost of decreased performance. `IMPATIENT` [44], [49] is a GPU-based library for accelerating iterative image reconstruction using a Toeplitz-based approach [41]–[44]. Section IV further discusses related work with respect to the spectrum of precomputation levels.

In all of these existing libraries, however, the user must manually choose parameter values for the oversampling ratio and kernel width. We propose a framework to automatically select these parameter values for the user to improve both performance and ease of use. NFFT and PNFFT libraries

for the CPU are well-developed and relatively complete in comparison to the existing libraries for GPU. However, GPUs can offer inexpensive significant speedup alternatives over multi-core CPU implementations. Therefore we focus here solely on GPU implementations of the NUFFT.

III. NUFFT ALGORITHMS AND PARAMETERS

Several algorithms for computing the NUFFT exist. The most commonly used are gridding-based methods which leverage the FFT, local interpolations, and weighting operations. In iterative algorithms, successive operations of the NUFFT and its adjoint are often needed. In that case, there is an interesting alternative often referred to as the Toeplitz approach, for computing both operations simultaneously. This method was first proposed by Wajer and Pruessman [41] and also appears in [42]–[44], and only uses FFTs and weighting operations without interpolations. Each of these mentioned algorithms has a different set of parameters and implementations, which comprise the optimization space for auto-tuning.

The runtime of NUFFT is a complex function of hardware parameters and the space of implementations and algorithm parameter values. Modern parallel processors are difficult to model, and hardware architectures can change significantly between machines and also over time. For example, GPUs may have different numbers of multiprocessors, memory bandwidth, available registers, shared memory, etc. Therefore, we propose to use auto-tuning to determine the implementations and algorithm parameters that would result in the fastest executions for each hardware architecture.

What the user would specify are the non-uniform sampling pattern, the desired accuracy level of the NUFFT, and amount of planning allowed. The auto-tuning library will automatically choose the implementations that meet the specified constraints and determine the optimal parameter values for each method to achieve the best performance. Auto-tuning may be performed over the space of algorithm parameters and implementations, or over a smaller search space of the algorithm parameters for a particular implementation. For different data sets with the same non-uniform sampling patterns, the same NUFFT plan can be reused (for example, in iterative reconstructions [1], [43], [44], [50]) and the cost of planning is amortized over each time the plan is reused.

As a proof of concept, we focus on a few algorithms to demonstrate auto-tuning of the NUFFT on the GPU. Additional algorithms may be included in the auto-tuning search space in the future. In this section, we describe the NUFFT problem in 1D for clarity; the equations can be extended to higher dimensions using separable functions.

A. Non-Uniform Discrete Fourier Transform

Several NUDFT problem types exist [51], where the image and/or Fourier domains have non-Cartesian sampling patterns. For simplicity, we focus on the case where the image domain is uniform and the Fourier domain is non-uniform, which occurs in applications where data are obtained in the frequency domain and a Cartesian image is desired for analysis and display purposes. Examples include non-Cartesian sampling

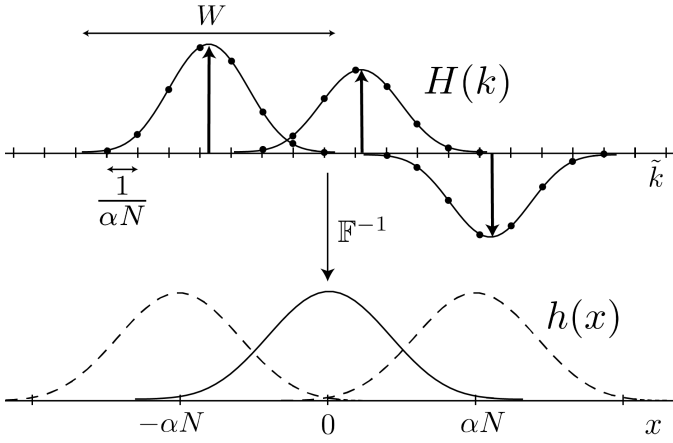


Fig. 2. In the gridding algorithm, or adjoint NUFFT, the non-uniform samples are convolved with an interpolation kernel $H(k)$ and resampled onto an oversampled Cartesian grid with $\Delta\tilde{k} = \frac{1}{\alpha N}$, where N is the desired image size. In the image domain, this corresponds to multiplying the image with $h(x)$, the inverse Fourier transform of the kernel, and replicating the image with a spacing of αN between copies. The overlapping of the image replications, or aliasing, results in the approximation error of the NUFFT.

trajectories in MRI, tomography and radio-astronomy. The opposite case, where the image domain is non-uniform and the Fourier domain is uniform, is a trivial extension and will not be discussed here.

Suppose we have a signal sampled at grid points $\{x_n\}$. Then, the NUDFT of the signal evaluated at arbitrary spatial digital frequencies $\{k_m\}$ is:

$$F(k_m) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f(x_n) e^{-i2\pi x_n k_m}. \quad (1)$$

Similarly, the adjoint operation is defined as follows:

$$\hat{f}(x_n) = \frac{1}{\sqrt{N}} \sum_{m=0}^{M-1} F(k_m) e^{+i2\pi x_n k_m}. \quad (2)$$

The above formulation uses a similar normalization that appears for the orthonormal DFT. However, unlike the orthonormal DFT, the adjoint and inverse NUDFT operations are not the same. Computing the inverse NUDFT is more difficult. Depending on the number of samples and their distribution, the inverse may be under- or over-determined and in general requires computing the inverse of the NUDFT matrix [13], [14] or computing the inverse through iterative algorithms, such as the conjugate gradient (CG) [15]–[17], that use the NUDFT and its adjoint operation internally.

FFT algorithms can achieve $O(N \log N)$ runtime by leveraging the symmetry of Cartesian sampling patterns [52], which does not apply to the NUDFT. Therefore, the algorithmic complexity of the direct computation of the NUDFT is $O(MN)$.

B. Gridding

We focus primarily on gridding algorithms, which approximate the forward and adjoint NUDFT by interpolating between

non-uniform samples and a Cartesian grid to allow use of the FFT. Table I defines the relevant variables and functions.

The forward NUFFT operation of a signal with length N is described by the following procedure:

- 1) Perform pre-apodization by dividing by $h(x)$.
- 2) Zero-pad to a grid αN and perform an FFT.
- 3) Interpolate to the non-Cartesian grid by convolution with the kernel function $H(k)$ with finite width W , and evaluate the result at frequencies $\{k_m\}$.

The adjoint NUFFT operation applies the operations in reverse and is described by the following procedure:

- 1) Convolve the non-uniform samples $F(k_m)$ with the interpolation kernel $H(k)$ of finite width W , and evaluate the result on an oversampled Cartesian grid with size αN .
- 2) Apply the inverse FFT and crop to size N .
- 3) Perform deapodization by dividing by $h(x)$.

We provide an in-depth discussion of the adjoint operation, since this operation used along with density compensation is often the method of choice for approximating the inverse NUFFT in practice. In the first step, the non-uniform samples $F(k_m)$ are convolved with an interpolation kernel $H(k)$ and resampled onto a Cartesian grid, which may be written as the following summation:

$$(F * H)(\tilde{k}_g) = \sum_{m: |\tilde{k}_g - k_m| \leq W/2} F(k_m) H(\tilde{k}_g - k_m). \quad (3)$$

The inverse FFT is then applied to the resampled data. As shown in Fig. 2, convolving $F(k_m)$ with $H(k)$ in the frequency domain corresponds to multiplying $f(x_n)$ by $h(x) = \mathbb{F}^{-1}\{H(k)\}$ in the image domain. Resampling on the oversampled Cartesian grid of size $G = \alpha N$ with $\Delta k = \frac{1}{\alpha N}$ corresponds to replication with spacing αN in the image domain, which results in aliasing. Finally, the uniform samples in the image domain are obtained by dividing by $h(x)$ and cropping the image. This deconvolution step is called deapodization and is implemented as a pointwise division. As shown in [29], the result of the adjoint NUFFT can be summarized as follows:

$$\hat{f}(x) = \mathbb{F}^{-1} \{ [F(k_m) * H(k)] \text{III}(Gk) \} (x_n) \frac{1}{h(x_n)} \quad (4)$$

$$= \left([f(x_n) h(x_n)] * \frac{1}{G} \text{III}\left(\frac{x_n}{G}\right) \right) \frac{1}{h(x_n)}. \quad (5)$$

The overlapping replications in the image domain can cause undesirable aliasing artifacts and degrade image quality. The level of aliasing is determined by the gridding parameters: oversampling ratio, kernel width, and interpolation kernel.

Commonly used interpolation kernels include Hamming, Hanning, Gaussian, and Kaiser-Bessel windows. We choose to use the Kaiser-Bessel kernel here, since it can be parameterized by a single parameter and has low relative sidelobe energy as a function of width W , as shown by Jackson et al. [34]. The Kaiser-Bessel kernel in one dimension is

$$H(k) = \frac{G}{W} I_0 \left(\beta \sqrt{1 - (2Gk/W)^2} \right), \quad (6)$$

with inverse Fourier transform

$$h(x) = \frac{\sin \sqrt{(\pi W x / G)^2 - \beta^2}}{\sqrt{(\pi W x / G)^2 - \beta^2}}, \quad (7)$$

where we use the shape parameter β from Beatty et al. [29]: $\beta(\alpha, W) = \pi \sqrt{\frac{W}{\alpha} (\alpha - \frac{1}{2})^2 - 0.8}$, and $I_0(\cdot)$ is the zero-order modified Bessel function of the first kind. In higher dimensions, separable interpolation kernels can be used.

The Kaiser-Bessel kernel width determines the amplitude of sidelobes of $h(x)$. A wider kernel results in smaller sidelobes and therefore lower error. However, the level of aliasing also depends on the oversampling ratio, which affects the spacing of the replications of the image. A higher oversampling ratio increases the distance between replications and allows for a transition region in $h(x)$, which reduces the amount of aliasing.

The approximation error of the NUFFT can be calculated by computing the exact NUDFT and subtracting the result of the NUFFT. However, this is impractical since the exact NUDFT may be expensive to compute and the error depends on both the sampling pattern and the sampled data. Instead, we desire a method for estimating approximation error that is independent of the sampling pattern and data. Beatty et al. [29] introduced the concept of the maximum aliasing amplitude ϵ , which they have shown to reliably estimate the order of magnitude of the approximation error. The maximum aliasing amplitude is defined as:

$$\epsilon = \max_x \sqrt{\frac{1}{h(x)^2} \sum_{p \neq 0} (h(x + Gp))^2}. \quad (8)$$

The relationship between maximum aliasing amplitude ϵ , oversampling ratio α , and kernel width W is shown in Fig. 3 for the Kaiser-Bessel kernel. Multiple (α, W) pairs that achieve the same desired maximum aliasing amplitude ϵ can be enumerated.

Given parameter values for (α, W) , we can analyze the NUFFT algorithmic complexity. The interpolation requires evaluating the kernel W^d times for each non-Cartesian sample. With an oversampling ratio α , an FFT with $\alpha^d N$ elements is evaluated. Finally, the deapodization step has N point-wise multiplications. The total runtime of gridding is

$$c_1 M W^d + c_2 (\alpha^d N) \log \alpha^d N + c_3 N, \quad (9)$$

where c_i are implementation and hardware dependent constants. The first two terms corresponding to the interpolation and FFT steps, respectively, dominate the overall runtime, while the last term for the deapodization is negligible.

To keep the maximum aliasing amplitude ϵ below a certain value, a high oversampling ratio α may be selected, at the cost of increased FFT runtime. Since a high oversampling ratio results in replications of the image spaced further apart with less aliasing, a small kernel width W corresponding to higher amplitude sidelobes is sufficient to achieve the desired accuracy level. The choice of a small kernel width decreases the interpolation runtime. Similarly, a lower oversampling ratio may be selected, which necessitates a larger kernel width to achieve the same maximum aliasing amplitude.

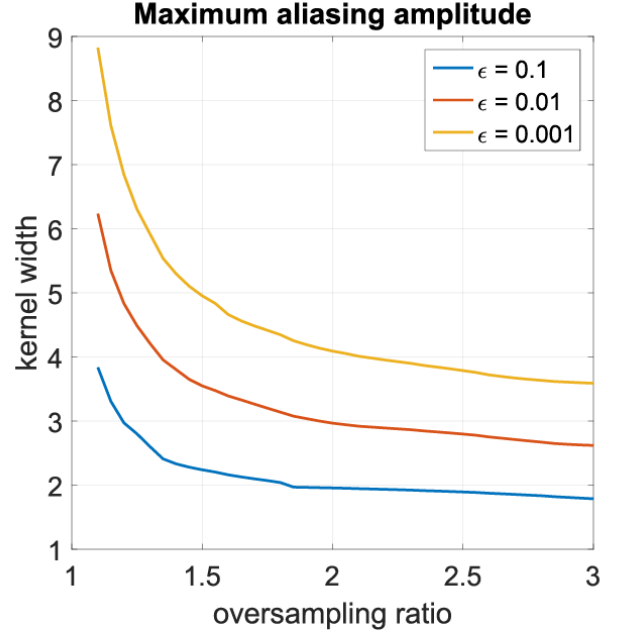


Fig. 3. Maximum aliasing amplitude ϵ as a function of oversampling ratio α and kernel width W , for linearly interpolated presampled Kaiser-Bessel kernel [29]. The maximum aliasing amplitude is a data-independent estimate of error between NUDFT and NUFFT. Lower error levels can be achieved by increasing α or W . Different sets of parameters (α, W) that achieve equal error level can be enumerated. The values shown here were calculated for $N = 128^3$.

Gridding implementations can trade off runtime between the interpolation and FFT, depending on the choice of oversampling ratio and kernel width. Since multiple (α, W) pairs achieve the same accuracy and it is unclear which set of parameters results in the fastest execution time, we can use empirical performance optimization to benchmark different sets of parameter values and then choose the fastest one.

The NUFFT gridding computations may be expressed compactly in terms of linear operators:

$$\text{NUFFT} = \Gamma \mathbb{F} D, \quad (10)$$

$$\text{NUFFT}^H = D^\top \mathbb{F}^H \Gamma^\top \quad (11)$$

where $\Gamma \in \mathbb{R}^{M \times G}$, $\mathbb{F} \in \mathbb{C}^{G \times G}$ and $D \in \mathbb{R}^{G \times N}$. The adjoint and forward NUFFT are conjugate Hermitian operations. D is a diagonal matrix with deapodization values which also performs zero-padding, and Γ is the interpolation matrix, where

$$\Gamma_{m,g} = \begin{cases} H(\tilde{k}_g - k_m) & |\tilde{k}_g - k_m| \preceq \frac{W}{2} \\ 0 & \text{else} \end{cases} \quad (12)$$

The interpolation operators Γ and Γ^\top can be implemented in several ways. The interpolation coefficients can be computed on-the-fly during execution or precomputed in advance. If the coefficients are computed on-the-fly, the operation is implemented directly as a convolution with the interpolation kernel. Alternatively, all coefficients may be precomputed and

TABLE I. GRIDDING VARIABLES

M	Number of non-uniform samples
N	Image size (total number of pixels)
G	Oversampled grid size (total number of pixels, $G = \alpha^d N$)
d	Dimensionality of image
x	Image coordinate, $x \in \mathbb{R}^d$
k	Spatial frequency coordinate, $k \in \mathbb{R}^d$
\tilde{k}	Spatial frequency coordinate on oversampled grid, $\tilde{k} \in \mathbb{R}^d$
n	Image grid index, $n \in \{0, \dots, N-1\}$
m	Spatial frequency index, $m \in \{0, \dots, M-1\}$
g	Oversampled grid index, $g \in \{0, \dots, G-1\}$
$f(x_n)$	Image with uniformly spaced samples
$F(k_m)$	Spatial frequency signal, with non-uniform sampling pattern
$\hat{f}(x_n)$	Gridding result from adjoint NUFFT
α	Oversampling ratio, for each dimension ($\alpha = \sqrt[d]{\frac{G}{N}}$)
W	Kernel width (in grid units)
ε	Maximum aliasing amplitude
R	Acceleration factor
α_T	Zero-padding ratio for Toeplitz method
$H(k)$	Kaiser-Bessel interpolation kernel
$h(x)$	Inverse Fourier transform of Kaiser-Bessel kernel $H(k)$
β	Shape parameter of Kaiser-Bessel kernel
$\mathbb{F}\{\cdot\}$	Fourier transform
Γ	Sparse interpolation matrix for the gridding-based NUFFT
D	Prepodization and zero-padding matrix for gridding-based NUFFT

stored in a sparse matrix. During execution, the interpolation is performed via sparse matrix-vector multiplication (SpMV), where the dense vector is the vectorized non-Cartesian samples in the frequency domain or the uniform samples in the image domain. These different implementation strategies are discussed further in Section IV.

C. The Toeplitz Method

In iterative reconstructions, the primary operation of interest is often the forward NUFFT directly followed by an adjoint NUFFT. In this case, a fast approach that combines the forward and adjoint NUFFTs into a single operation can be advantageous. As shown in [41]–[44], [49], [53], when using a grid oversampling larger than two, the joint forward and adjoint NUFFTs have a Toeplitz structure which reduces the joint operation to a convolution with the point spread function (PSF) $q(x_n)$ of the non-uniform sampling pattern. This linear convolution can be computed in practice, by using a zero-padding ratio of $\alpha_T \geq 2$ and performing the convolution in the DFT domain as follows:

$$g(x_n) = f(x_n) * q(x_n) = \mathbb{F}^{-1} \{ \mathbb{F} \{ f(x_n) \} \mathbb{F} \{ q(x_n) \} \}, \quad (13)$$

where $q(x_n) = \sum_{m=0}^{M-1} e^{-i2\pi x_n k_m} \mid 0 \leq n < \alpha_T N$, which may be computed exactly or via gridding. In the planning phase, $\mathbb{F} \{ q(x_n) \}$ is computed and stored in memory.

During execution, two FFTs of size $\alpha_T^d N$ are computed, where $\alpha_T \geq 2$, with $\alpha_T^d N$ pointwise multiplications. The runtime of the Toeplitz-based approach is

$$c_1 2(\alpha_T^d N) \log \alpha_T^d N + c_2 \alpha_T^d N. \quad (14)$$

The FFT step dominates the runtime while the pointwise multiplication is negligible. Unlike the gridding-based method, the Toeplitz-based approach has an execution runtime that is independent of the approximation error. Since the runtime of FFT is non-monotonic, auto-tuning can be applied to determine the value of α_T that yields the fastest execution time.

IV. NUFFT IMPLEMENTATIONS

Here we discuss further several implementations and optimization strategies on the GPU. In particular, we discuss different computation methods for interpolation and FFT in gridding.

A. CUDA Overview

The CUDA programming model provides a framework to extend C and write functions called kernels to launch on the GPU. Whereas CPUs are optimized for latency, GPUs have high throughput with many processors for parallel computation. Kernels are executed by a grid of blocks, which are groups of threads that execute in parallel. Threads execute in lockstep in groups of 32 called warps. To achieve high performance, kernels need to exploit sufficient parallelism to hide latency. Care must be taken to choose the grid and block sizes appropriately. High occupancy (number of active warps divided by maximum active warps) usually allows for latency-hiding and results in good performance.

GPUs have three main types of memory: global memory, shared memory, and local memory. Global memory is the largest and accessible to all threads, but has high latency. Shared memory is fast on-chip memory with the lowest latency, but it is a limited resource and is shared between threads in the same block. Local memory is private to each thread with latency similar to that of global memory.

General strategies for obtaining good performance include using memory efficiently, avoiding thread divergence, and minimizing data transfers. Accesses to global memory should be linear and aligned when possible, to encourage coalescing and caching. Thread divergence occurs when threads within the same warp take different execution paths. When a subset of threads in a warp proceed on an execution path, all other threads in the warp are idle, which degrades performance. Lastly, data transfers between CPU and GPU can take a significant amount of time relative to computation. Data transfers should be avoided when possible or overlapped with computation.

B. Interpolation and Resampling

NUFFT implementations span a spectrum of different levels of precomputation, shown in Figure 4, ranging from no precomputation to full precomputation. Toeplitz-based methods can be considered to fall in the category of full precomputation, since the location of the non-Cartesian samples are no longer required during execution, after the Fourier transform of the PSF has been precomputed. Table II shows existing software implementations and related work along the spectrum of precomputation levels.

Here we compare different implementations of gridding-based methods across the spectrum of precomputation levels. Precomputation-free methods require less memory and planning time for precomputation compared to fully precomputed methods, but may result in slower execution time. If a non-uniform sampling pattern is reused multiple times, then the cost of precomputation is amortized over multiple executions of the same NUFFT plan, and higher levels of precomputation with faster execution times may be preferred.

The relative spatial positions and ordering of the non-Cartesian samples in memory affect the interpolation time. In particular, the performance of GPU programs are highly sensitive to cache performance. Additional optimizations, which are not explored further here, can be made for methods of all precomputation levels by reordering the non-Cartesian samples to improve the pattern of memory accesses and caching behavior.

1) *No Precomputation:* In the precomputation-free approach, the interpolation is performed as a convolution, which may be parallelized over non-Cartesian samples or Cartesian grid points, referred to as input-driven or output-driven interpolation, respectively [36]. The precomputation-free approach is implemented in PowerGrid [48], NFFT [46], PNFFT [47], BART [50], and IRT [55].

Parallelization over Cartesian grid points is expensive without precomputation, since it is not known a priori which non-Cartesian samples contribute to a particular Cartesian grid point. Each thread corresponding to a Cartesian grid point must loop over all non-Cartesian samples to find the samples that lie within its neighborhood of width W , resulting in an $O(MN)$ algorithm.

Instead, parallelizing over non-Cartesian samples presents a lower-cost approach with algorithmic complexity $O(MW^d)$. Each thread corresponding to a non-Cartesian sample can enumerate its neighboring Cartesian grid points. Fig. 4a illustrates the precomputation-free method.

In the adjoint direction, each thread adds the non-Cartesian sample's contribution to the neighboring Cartesian grid points via atomic add operations, which are required to ensure correctness. Similarly, in the forward direction, each thread accumulates the contribution from neighboring Cartesian grid points to the non-Cartesian sample. Unlike the adjoint operation, the forward operation does not require atomic add operations. For each NUFFT computation, each thread must enumerate the neighboring grid points and calculate the interpolation kernel values based on distance to the grid points.

In NUFFT implementations, the interpolation kernel is usually finely presampled and stored in a lookup table [29]. Presampling the kernel allows us freedom to choose any kernel function, without needing to account for the cost of evaluating a particular function in the execution phase. The value of the kernel at a particular point is approximated using linear interpolation of the lookup table values. On the GPU, texture memory can be used to perform fast low-precision linear interpolation where the linear interpolation coefficients are represented with 9-bit fixed point format, as implemented in gpuNUFFT [38].

The precomputation-free approach requires no additional

memory besides the input, output, and interpolation kernel lookup table, and the planning phase is quick, with only the computation of the lookup table. However, this approach is optimal in the number of read operations and suboptimal in the number of write operations in the adjoint direction, and vice versa for the forward direction [39]. In the adjoint direction, each non-Cartesian sample is read once, while values are written to the surrounding Cartesian grid samples multiple times using atomic add operations. Similarly, in the forward direction, each non-Cartesian sample is written once, while the values of the neighboring Cartesian points are read multiple times.

Furthermore, in the adjoint direction, if two non-Cartesian samples are spatially close and their threads attempt to write to the same grid point simultaneously, the atomic add operations are serialized. That is, one thread must wait for the other to finish updating the grid point before it can write its value to the grid point. In sampling patterns such as the radial sampling pattern, non-Cartesian samples are densely located toward the center of k-space, resulting in serialization of atomics. Meanwhile, non-Cartesian samples at the edge of k-space are farther apart. Threads corresponding to these samples finish their computation earlier and must wait for threads at the center of k-space to finish their work. These issues suggest that some precomputation may be beneficial in providing a better balance between the number of read and write operations and/or preventing serialization of atomics.

2) *Partial Precomputation:* Various partially precomputed methods fall between the two ends of the spectrum. One approach, implemented in gpuNUFFT [38] and described by Sorensen et al. [54] and Gregerson [36], is to divide k-space into sectors, where each sector may be assigned to a processor, as shown in Fig. 4b. During the planning phase, non-Cartesian samples are sorted into sectors. In the adjoint operation, each non-Cartesian sample that contributes to a sector is read once from a precomputed list of samples. The contributions to Cartesian grid points are written to fast, low-latency shared memory multiple times, and the final result is written once to global memory. In this sector-based approach, the non-Cartesian samples at the boundaries of the sectors will contribute to Cartesian grid points in neighboring sectors, which necessitates extending the boundary of the corresponding shared memory grid by $W/2$ in each direction. In gpuNUFFT [38], load-balancing is also implemented by creating new virtual sectors when the number of non-Cartesian samples in a sector exceeds some limit.

Obeid et al. [40] and Gai et al. [49] describe a compact binning approach, where bins are allowed to have a variable number of non-uniform samples and load-balancing is achieved by partitioning work between the GPU and CPU. This approach is implemented in IMPATIENT [44]. Regular binning, where each bin contains an equal number of samples, is not used due to the potentially high level of extra padding and memory required for highly non-uniform sampling patterns.

3) *Full Precomputation:* Fully precomputed methods enumerate the neighboring Cartesian grid points and calculate interpolation kernel values in advance during the planning phase. The interpolation kernel values are stored in a sparse

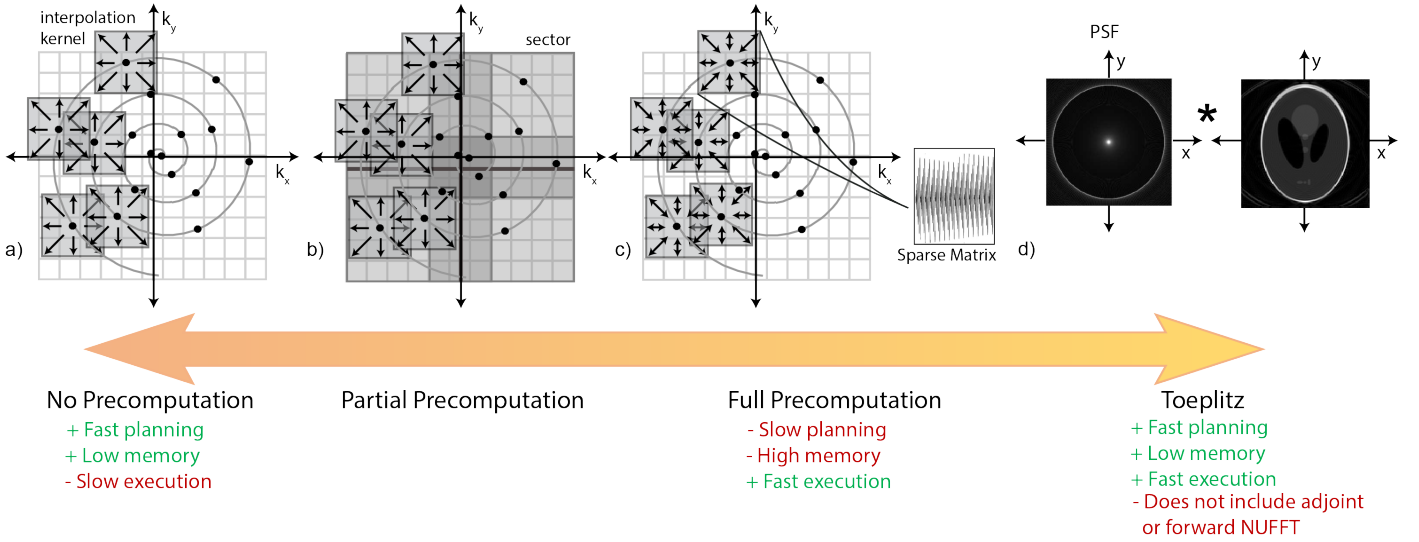


Fig. 4. Spectrum of precomputation levels for various NUFFT implementations. Different levels of precomputation present trade-offs with planning time, memory footprint, and execution time. a) With no precomputation, each non-Cartesian sample is assigned a thread, which will enumerate its neighboring Cartesian grid points and compute values to gather from or add to its neighbors, and hence requires atomic operations to prevent conflicts. b) Partial precomputation [36], [38], [40], [44], [54] divides k -space into sectors or bins during the planning phase and computes results for each sector in parallel. Within each sector, non-Cartesian samples are processed in parallel. c) The fully precomputed method computes in advance the interpolation coefficients, which are stored in a sparse matrix. During the execution phase, the interpolation is performed as an SpMV operation, which is parallelized along the non-Cartesian points in the forward operation and the Cartesian grid points in the adjoint operation. d) The Toeplitz-based method [41]–[44] precomputes the PSF of the non-uniform sampling pattern during the planning phase. During execution, the PSF and image are convolved using FFTs.

TABLE II. SOFTWARE IMPLEMENTATIONS AND RELATED WORK ALONG THE SPECTRUM OF PRECOMPUTATION LEVELS.

	Implementation Method			
	No Precomputation	Partial Precomputation	Full Precomputation	Toeplitz
Open-Source GPU Software	gNUFFTW PowerGrid [48]	gpuNUFFT [38] IMPATIENT [44]	gNUFFTW	gNUFFTW IMPATIENT [44]
Open-Source CPU Software	NFFT, PNFFT [45], [47] BART [50] IRT [55]		NFFT [45] IRT [55]	BART [50]
Research Publications	Beatty et al. [29]	Obeid et al. [40] Gregerson [36] Sorensen et al. [39] Gai et al. [49]	Murphy [32]	Wajer and Pruessman [41] Eggers et al. [42]

matrix (Fig. 4c). During the execution phase, the interpolation and resampling is performed as a sparse matrix-vector multiplication (SpMV). We rely on existing optimized libraries for sparse matrix operations, such as cuSPARSE [33]. Unlike the previously discussed methods, the fully precomputed method incurs a large memory footprint. Existing implementations of the fully precomputed approach can be found in NFFT [46] and IRT [55].

Multiple representations for sparse matrices exist, including compressed sparse row (CSR) and a hybrid format (HYB) of Ellpack-Itpack (ELL) and coordinate (COO) formats. COO stores the nonzero entries and their row and column indices. CSR compresses the vector of row indices into pointers to the beginning of each row. HYB stores a portion of the matrix in ELL format and the remainder in COO format. ELL stores the nonzero entries and column indices as block matrices, with row

length K . If a row has fewer than K nonzero entries, then zeros are represented as 0 and -1 for the value and column index, respectively. The ELL format is useful when the matrix has a regular structure with a similar number of nonzero entries in each row. Thus, the HYB format stores the regular part of the matrix as ELL and the irregular part as COO. On the GPU, CSR SpMV kernels have partial coalescing, whereas HYB SpMV kernels have full coalescing. Bell et al. [56] optimized and benchmarked various SpMV kernels on the GPU and found that no particular sparse matrix format outperforms the other formats for all classes of unstructured matrices that were benchmarked.

The fully precomputed method requires significantly more memory than precomputation-free and partially precomputed methods. Using the CSR sparse matrix format requires storing M row pointers and MW^d values and column indices

for the nonzero entries, for the adjoint operation. The sparse matrix for the forward operation is the transposed matrix, with G row pointers and MW^d values and column indices. Memory requirements for the HYB sparse matrix format depend on the selected value of K and the number of remaining entries stored in COO format. The ELL portion stores K values and column indices for each row, and the COO portion stores row index, column index, and value for all remaining entries. Transposing a sparse matrix is generally slow. To obtain the fastest execution, we store both the adjoint and forward matrices individually, which doubles the required memory footprint.

The values of the sparse matrix are real, while the dense vector is complex. The cuSPARSE library only supports SpMV operations where the matrix and vector have the same data type. One option is to perform a sparse matrix dense matrix multiplication, where the dense matrix consists of the real and imaginary parts of the complex data. cuSPARSE assumes dense matrices to be stored in column-major ordering, i.e. all of the real parts followed by all of the imaginary parts, whereas cuFFT for complex data expects the real and imaginary parts to be interleaved. Although cuSPARSE can also index into the dense matrix with row-major ordering, this yields suboptimal performance. We include in our implementation a modified version of CUSP SpMV routine [57] to multiply a real-valued sparse matrix with complex vector. Alternatively, we can store the sparse matrix with complex values where the imaginary part is 0. Although this further doubles the memory footprint, this option is straightforward and allows us to leverage the highly optimized cuSPARSE routines. This sparse matrix representation appeared to perform better than the modified CUSP routine and HYB format and was chosen for the fully precomputed method in our library. Future work may involve exploring optimizations for the modified CUSP routine and HYB format, such as choosing appropriate grid and block sizes or choosing the row length parameter K , respectively.

C. FFT Optimizations

1) *Pruned FFT*: Further optimizations to the FFT step can be made by pruning unnecessary operations. The output of the inverse FFT in the adjoint NUFFT is cropped, and the input to the FFT in the forward NUFFT is zero-padded. We can compute only a subset of the output in the adjoint operation, and similarly, only a subset of the inputs in the forward operation are used. Here we discuss only the cropped inverse FFT for the adjoint operation where we compute a subset of the output, and similar analysis also holds for the zero-padded FFT.

These pruning optimizations are most beneficial when the subset of interest is small relative to the full output. In the context of NUFFT, the size of the subset, K , is smallest for 3D problems with a high oversampling ratio, $\alpha = 2$, where we are interested in $1/8$ of the output. We will analyze the performance for this best-case scenario.

A multidimensional FFT can be decomposed into 1D FFTs along each dimension. One way to omit unnecessary operations is to compute only the 1D FFTs that contribute to the final result, which is the approach used in PNFFT [47]. For $\alpha = 2$ and

a 3D FFT with size n^3 , we perform n^2 1D FFTs of length n in the first dimension, $n^2/2$ 1D FFTs in the second dimension, and $n^2/4$ 1D FFTs in the third dimension. If the algorithmic complexity for the full FFT problem is $c \cdot 3n^3 \log(n)$, the algorithmic complexity in this approach is $c \cdot 1.75n^3 \log(n)$. This approach yields only a constant factor improvement in algorithmic complexity by 41.67%. Furthermore, performing a batch of 1D FFTs in this configuration requires custom FFT kernels. The cuFFT library [31] provides advanced data layout options, such as distance, stride, and embedded arrays; however, we require a non-constant distance between the first element of consecutive batches, which cannot be specified through the cuFFT interface.

Alternatively, FFT problems can be decomposed into smaller FFTs, which are then recombined with linear phase factors. We can directly eliminate all unnecessary operations that do not contribute to the subset of outputs of interest. This is referred to as a pruned FFT [54], which has algorithmic complexity $O(N \log K)$ where N is the size of the full output. Instead of performing a full 3D FFT of size n^3 , we can perform 8 smaller FFTs of size $(n/2)^3$, where the algorithmic complexity is $cn^3 \log \frac{n}{8}$. Compared to the previous approach, the pruned FFT has much lower algorithmic complexity, with an improvement in the log factor. As shown in Figure 5, we benchmarked these 3D FFT problems and compared the time to perform 8 smaller FFTs with the time for the full FFT problem. The timing measurements represent an upper bound on the potential speedup of the pruned FFT, as they do not account for recombination of the intermediate results with the linear phase factors.

In most cases, the potential speedup of the smaller FFTs over the full FFT is not substantial enough to justify using the pruned FFT, and so our library does not include pruned FFTs.

2) *Selection of Oversampling Ratio*: The runtime of the FFT is non-monotonic and highly dependent on the factorization of the problem size. For gridding, we choose the set of oversampling ratios α that result in oversampled grid sizes with fast FFT runtimes, of the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. This both reduces the search space and avoids pathological FFT sizes. For the same reason, we enumerate several values of α_T for the Toeplitz-based approach and select the optimal value empirically.

Finally, as shown in Figure 3, for a fixed maximum aliasing amplitude, the kernel width decreases as the oversampling ratio increases. However, at high oversampling ratios, increasing the oversampling ratio further will only result in negligible decreases in the kernel width. The FFT time will increase while the interpolation time decreases only by a small amount. Therefore, we restrict our parameter search space for the oversampling ratio to $\alpha \in (1, 2]$ for gridding.

V. METHODS

Given a target maximum aliasing error, our library empirically searches over the space of NUFFT implementations and their parameters, to choose the combination that results in the fastest execution time. The library includes precomputation-free, fully precomputed, and Toeplitz methods, which use

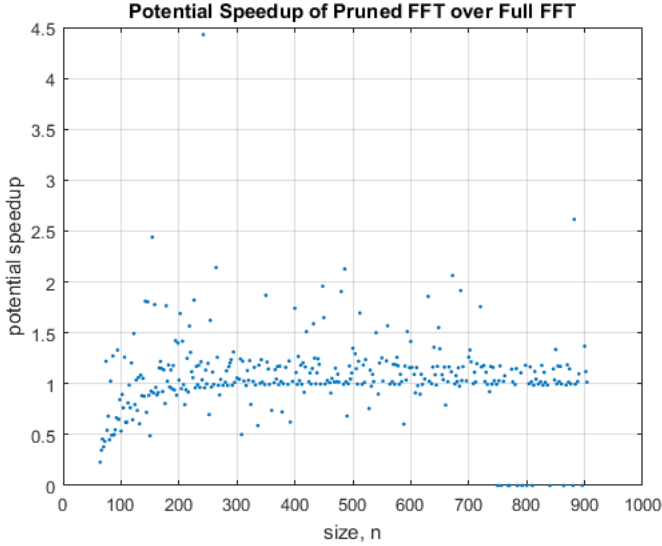


Fig. 5. Potential speedup of 3D pruned FFT compared to full FFT. NUFFT utilizes FFTs that are zero-padded or cropped, which can be exploited using the pruned FFT algorithm [54]. The potential performance increase is greatest for large-dimensional problems where the outputs of interest are a small fraction of the full FFT output, i.e. 3D FFT with $\alpha = 2$. Pruned FFT consists of 8 smaller $(n/2)^3$ FFTs, rather than a single n^3 FFT. The potential speedup is calculated by dividing the runtime of the full FFT by the runtime of the 8 smaller FFTs. The plotted data represents an upper bound on speedup and does not account for data movement.

cuFFT and cuSPARSE. Pruned FFT optimizations are not included. We also incorporate gpuNUFFT [38] into our auto-tuning framework, to utilize its implementation of partial precomputation. We use the recommended sector width of 8 for 3D NUFFTs. gpuNUFFT uses only integer kernel widths, which we accommodate by rounding the kernel width both up and down.

We evaluate the NUFFT for several different problem sizes, which are characterized by their image size and the size of the sampling pattern. The number of non-Cartesian samples is determined by the efficiency of the sampling pattern and the acceleration factor R , which is the level of undersampling below the Nyquist rate. A fully sampled non-uniform sampling pattern corresponds to acceleration factor $R = 1$, while under-sampled sampling patterns correspond to higher acceleration factors $R > 1$. The outcome of undersampling is that with increased R , the ratio between the number of samples in the frequency and that in the image domain decreases.

For each NUFFT implementation, we measure the execution time for isotropic 3D images with the following variables:

- image size $N = 62^3, 94^3, 126^3, 158^3, 190^3, 222^3, 254^3$
- radial, rotated stack of stars, cones [58], and uniform random sampling patterns
- acceleration factor $R = 1, 4, 16$
- maximum aliasing amplitudes $\varepsilon = 0.01, 0.001$
- varying oversampling ratios.

All benchmarking was performed on a Tesla K40 GPU. For each NUFFT problem, we perform the computation 50 times

and use the median execution time from the CUDA event timer. We use NUFFT from BART [50], an open-source library for MRI reconstruction, as a serial CPU baseline, where the oversampling ratio and kernel width are the fixed default values ($\alpha = 2, W = 3$).

Execution times do not include time for data transfer between CPU and GPU; we assume that input data is already present on the GPU and output data remains on the GPU. That is, we assume that the GPU has sufficient memory for the NUFFT problem and that there may be additional processing that will also be performed on the GPU. These assumptions are suitable for iterative reconstructions, where the input and output data can remain on the GPU between iterations, provided that the NUFFT problem fits in the GPU memory.

VI. PERFORMANCE ANALYSIS AND DISCUSSION

Here we present performance analysis of various NUFFT problems using auto-tuning. First, we examine the effect of the oversampling ratio on the gridding runtime and examine the optimal oversampling ratios that yield fast NUFFT computations. We show that auto-tuning provides substantial performance benefits over fixed parameter values. We then compare the performance of the different gridding- and Toeplitz-based implementations and analyze the effects of different non-Cartesian sampling pattern inputs.

A. Performance Improvement from Auto-Tuning Algorithm Parameters

The effects of the oversampling ratio on the NUFFT runtime are shown in Fig. 6. As the oversampling ratio increases, the Cartesian grid size and FFT runtime increase. A lower kernel width can be used to achieve the same desired accuracy, and the interpolation time decreases. The total time consists of a trade-off between the interpolation runtime and FFT runtime.

The trade-off in interpolation and FFT time varies with the acceleration factor R . With a lower acceleration factor, the sampling pattern has more samples and the overall runtime is dominated by the interpolation time, resulting in an optimal oversampling ratio that is close to 2. As the acceleration factor is increased, the interpolation runtime decreases while the FFT runtime remains the same as before. The optimal oversampling ratio is lower for high acceleration factors. Fig. 7 shows the optimal oversampling ratios, averaged over different image sizes N , for a range of sampling patterns, acceleration factors, and maximum aliasing amplitudes $\varepsilon = 0.01$ and $\varepsilon = 0.001$.

These effects are more pronounced for the full precomputation method than for the precomputation-free method, as the overall runtime is more closely distributed between SpMV and FFT than between direct convolution and FFT. In the precomputation-free method, interpolation time dominates over the FFT time, and the optimal oversampling ratio tends to be closer to 2. In the fully precomputed method, the runtime is generally more evenly balanced between interpolation and FFT, and the optimal oversampling ratio is more sensitive to the acceleration factor. In addition, these effects are also more pronounced at higher maximum aliasing amplitudes for the

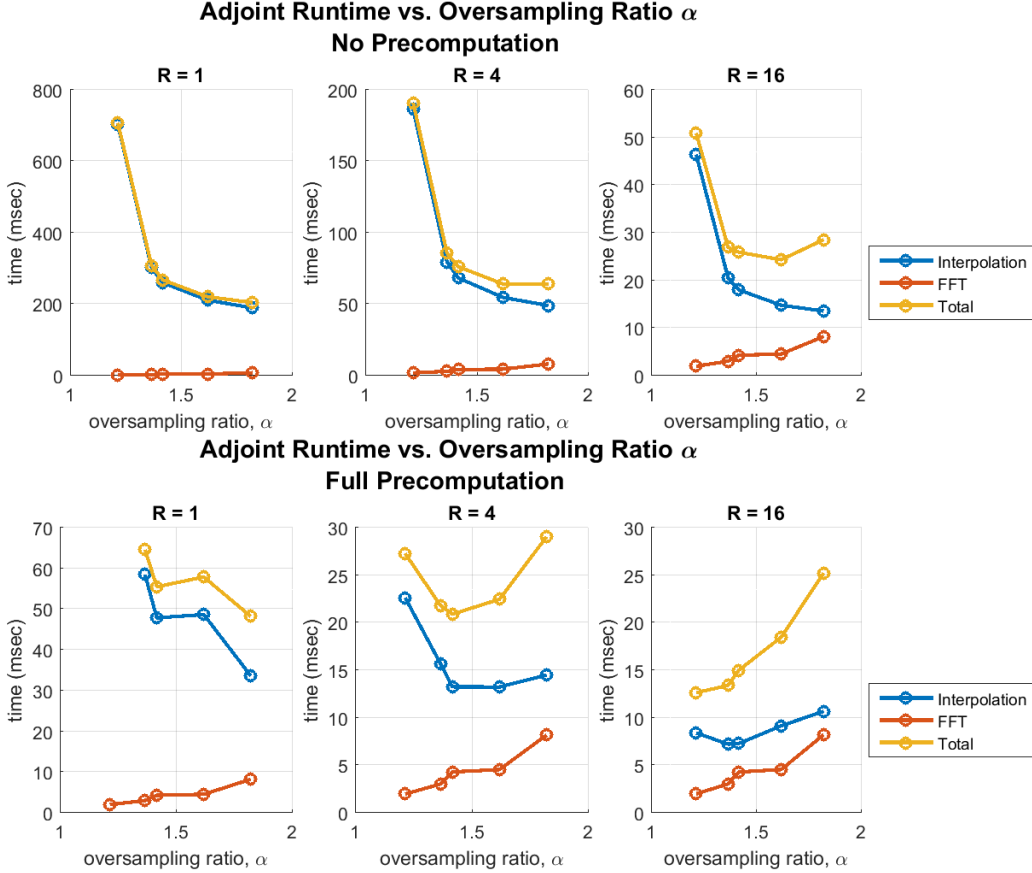


Fig. 6. Effects of oversampling ratio on execution time. As the oversampling ratio increases, the FFT time increases, while interpolation time (convolution or SpMV) decreases due to the decrease in kernel width. In the precomputation-free method, convolution time dominates the total execution time, whereas the runtime is more evenly balanced between SpMV and FFT for the fully precomputed method. The trade-off between interpolation and FFT times varies with the interpolation method and the acceleration factor. Parameters: $N = 158^3$, $\varepsilon = 0.01$, non-separable (i.e. rotated) stack of stars sampling pattern. (Refer to Fig. 14 in App. C for the corresponding graphs of the forward runtime, which are similar.)

same reasons; the interpolation and FFT times are more evenly balanced. Therefore, the optimal oversampling ratio depends on various factors, including maximum aliasing amplitude, acceleration factor, and non-uniform sampling pattern.

We compare performance with auto-tuned oversampling ratios versus fixed oversampling ratios in Fig. 8. We benchmarked the partially precomputed method in `gpuNUFFT` over different image sizes, with auto-tuning and with fixed oversampling ratios $\alpha_{\text{low}} = 1.125$ and $\alpha_{\text{high}} = 2$. Across all sampling patterns and acceleration factors, the speedup is highest with the auto-tuned oversampling ratio. The improvement in performance with the auto-tuned oversampling ratio compared to fixed oversampling ratios is significant, with up to an order-of-magnitude increase in speedup. These performance improvements arise from optimizing the trade-off between interpolation and FFT runtimes and choosing oversampled grid sizes that account for the non-monotonic nature of the FFT runtime.

B. Comparison of NUFFT Implementations

Auto-tuning over algorithm parameters was performed for a range of implementations, including the precomputation-free, fully precomputed, and Toeplitz methods implemented in `gNUFFT` and the partially precomputed method from `gpuNUFFT` [38]. Fig. 9-10 show the performance across different sampling patterns and acceleration factors for $\varepsilon = 0.01$ and $\varepsilon = 0.001$.

As expected, higher levels of precomputation generally correspond to higher speedups. Fewer operations are required during the execution phase for the higher levels of precomputation. We use the NVIDIA Profiler to further examine the performance of different implementations.

In the precomputation-free method, performance is limited by high register usage, which is required to store intermediate results for enumerating and calculating distances to the neighboring grid points. Each thread uses 79 registers, or 60,672 registers per shared multiprocessor (SM), out of a maximum of 65,536 registers per SM. The occupancy is limited to 24 active warps, out of a maximum of 64 active warps. Because the occupancy is independent of the maximum aliasing amplitude

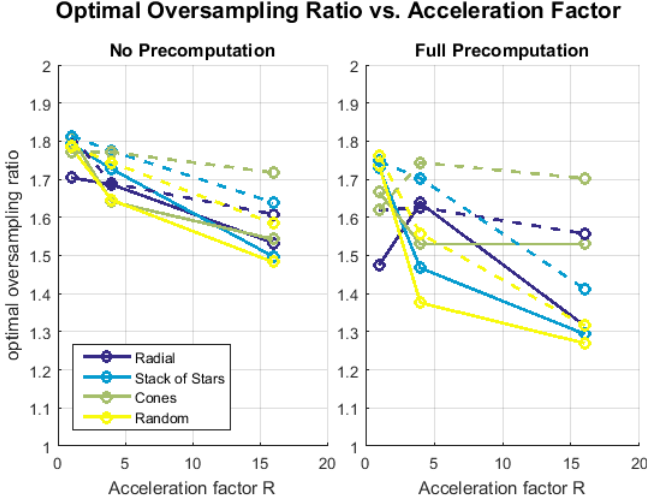


Fig. 7. Optimal oversampling ratios, averaged across image sizes N , for $\varepsilon = 0.01$ (solid) and $\varepsilon = 0.001$ (dashed), for NUFFT problems specified in Section V. The optimal oversampling ratio is close to 2 for low acceleration factors and decreases as the acceleration factor is increased, since the interpolation occupies a smaller portion of the overall runtime (see Fig. 6). The effect is more pronounced for the fully precomputed method and for higher values of ε , where the runtime is more evenly distributed between interpolation and FFT.

ε , the speedup of the precomputation-free method relative to the fully precomputed method is similar across $\varepsilon = 0.01$ and $\varepsilon = 0.001$.

The partially precomputed method performs well for low values of acceleration factor R . However, its performance suffers at higher acceleration factors where the number of non-Cartesian samples is low, or with smaller image sizes $N = 62^3, 94^3$. The partially precomputed method requires additional overhead to combine the intermediate results from each sector, which can decrease performance, especially for smaller problem sizes.

In addition, performance for the partially precomputed method decreases as ε is lowered, due to limitations on shared memory usage. The partially precomputed method requires shared memory for each sector which has padded dimensions $SW_{pad}^d = (SW + 2 \lfloor \frac{W}{2} \rfloor)^d$, where SW is the chosen sector width. The additional padding is necessary to compute the gridded values near the edges of the sectors. Since shared memory is a limited resource, shared memory usage can limit occupancy on the GPU.

Consider an oversampling ratio $\alpha = 1.5$ and sector width $SW = 8$ on the Tesla K40. To achieve a maximum aliasing amplitude $\varepsilon = 0.01$, the required kernel width is $W = 3.139$ which corresponds to $(8 + 2 * 1)^3$ complex floats, or 7.8125 KiB of shared memory out of a maximum of 48 KiB per shared multiprocessor. With block size SW_{pad}^2 , the occupancy is limited to 24 active warps. When the maximum aliasing amplitude is lowered to $\varepsilon = 0.001$, the required kernel width increases to $W = 4.346$ and 13.5 KiB of shared memory is required. Due to the increased shared memory usage, the occupancy is decreased to 15 active warps.

In the fully precomputed method, performance is limited by the memory bandwidth of the GPU, due to the high memory traffic incurred by SpMV operations. On different GPU architectures, differences in the available registers, shared memory, and memory bandwidth will result in different relative speedups between no precomputation, partial precomputation, and full precomputation methods.

The Toeplitz method achieves higher performance than gridding-based methods when the number of non-Cartesian samples is relatively high or when the desired maximum aliasing amplitude is low. As shown in Fig. 9-10, the Toeplitz method is slower than gridding-based methods when there are few non-Cartesian samples (i.e. $R = 16$). The Toeplitz method requires a larger FFT than gridding-based methods. For problems with few non-Cartesian samples, the time to perform the larger FFT of the Toeplitz method ($\alpha_T \geq 2$) is greater than the time to interpolate the samples onto a Cartesian grid and perform a smaller FFT ($\alpha \in (1, 2]$). When the number of non-Cartesian samples is relatively high or the required maximum aliasing amplitude is low, the Toeplitz-based approach may be the optimal choice of algorithm if the separate adjoint and/or forward operations are not needed.

In summary, higher levels of precomputation generally achieve higher speedups, since they require fewer operations during execution. However, when a low maximum aliasing amplitude is required, performance of the partially precomputed method decreases due to shared memory requirements. The Toeplitz method is favorable over gridding-based methods when the sampling pattern has a large number of non-Cartesian points or when a low maximum aliasing amplitude is required.

C. Effects of Non-Cartesian Sampling Pattern

The non-Cartesian sampling pattern affects the pattern of memory accesses in the interpolation step of gridding. Here we compare different sampling patterns by examining the interpolation runtime and the distribution of non-zero entries per row in the sparse matrices Γ and Γ^T as shown in Fig. 11, with the calculated standard deviations in Table III in App. C.

In the adjoint operation, the distribution of row lengths reflects the spatial distribution of the non-Cartesian samples. For example, the standard deviation of the row lengths is highest for the radial sampling pattern, where many samples are located near the center of k-space and few samples are located at the outer edges. In the case of the random sampling pattern, the standard deviation is low since samples have a uniform random spatial distribution. In the precomputation-free method, a high standard deviation corresponds to uneven serialization of atomics across different threads. In the fully precomputed method, each row is assigned a warp of threads. When the row lengths have an uneven distribution, some warps have more work than others, which will increase the overall runtime.

In the forward operation, each non-Cartesian sample is computed from approximately W^d neighboring grid points, where W may be rounded up or down in each dimension depending on the location of the non-Cartesian sample in the grid. For example, with a kernel width $W = 3.25$, each

Average Speedup for Forward + Adjoint NUFFT with Auto-Tuned and Fixed Oversampling Ratios

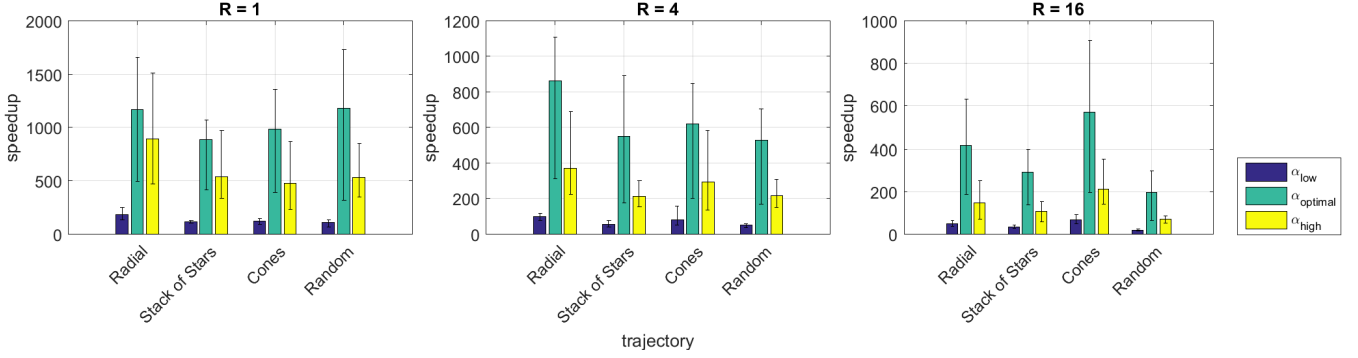


Fig. 8. Comparison of performance with auto-tuned and fixed oversampling ratios (α_{optimal} , $\alpha_{\text{low}} = 1.125$, $\alpha_{\text{high}} = 2$). Speedup over CPU implementation (BART) using partial precomputation (gpuNUFFT $\lfloor W \rfloor$). Auto-tuning can provide up to an order of magnitude improvement in performance over fixed parameters. Parameters: $\varepsilon = 0.01$; $R = 1, 4, 16$; $N = 126^3, 158^3, 190^3, 222^3, 254^3$

Average Speedup for Forward + Adjoint NUFFT, $\varepsilon = 0.01$

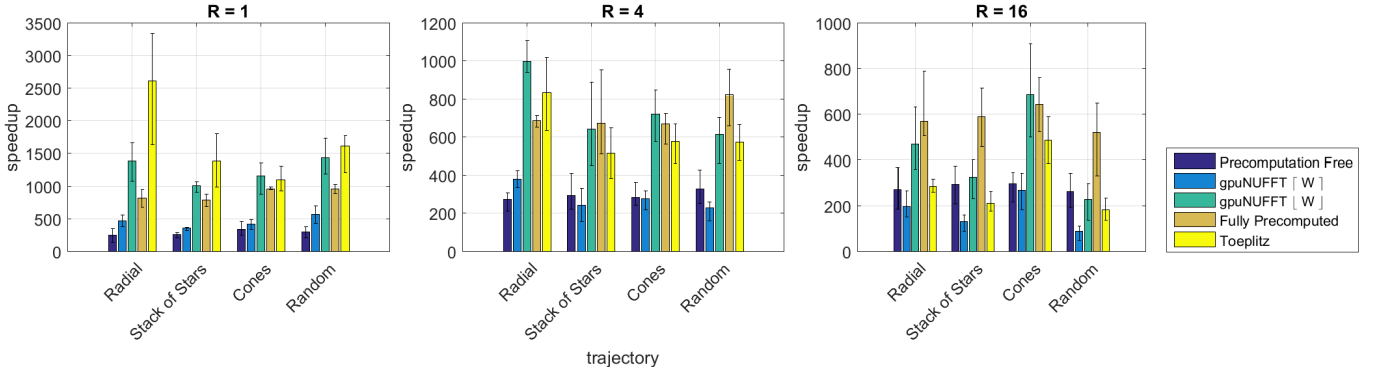


Fig. 9. Speedup over CPU implementation, averaged across image sizes, for isotropic 3D images with different non-uniform sampling patterns. Error bars show minimum and maximum speedup. The Toeplitz method achieves the best performance for small values of R , while the fully precomputed method is better for higher values of R . Parameters: $\varepsilon = 0.01$; $R = 1, 4, 16$; $N = 126^3, 158^3, 190^3, 222^3, 254^3$. (Speedups for $N = 62^3, 94^3$ are not included here, since the performance of partial precomputation is reduced for smaller problem sizes. See Fig. 15, 17, 19, 21 in App. C for complete results.)

Average Speedup for Forward + Adjoint NUFFT, $\varepsilon = 0.001$

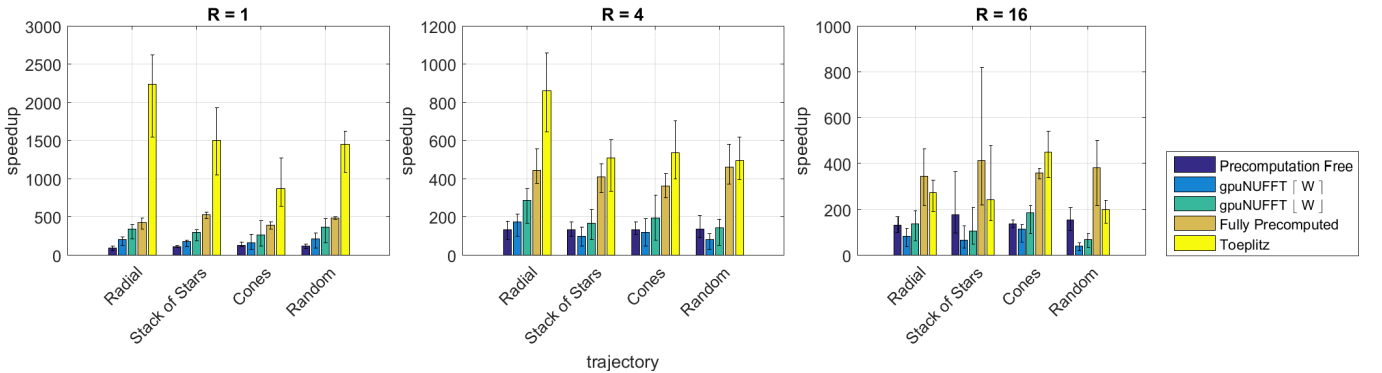


Fig. 10. Speedup over CPU implementation, averaged across image sizes. When the accuracy level is increased, the runtime of the Toeplitz method remains constant, whereas the runtimes of gridding-based methods increase. Parameters: $\varepsilon = 0.001$; all other parameters same as in Fig. 9. (See Fig. 16, 18, 20, 22 in App. C for complete results.)

non-Cartesian point may have $\{\lfloor W \rfloor, \lceil W \rceil\}^d$ neighboring grid points (3^3 , $3^2 \cdot 4$, $3 \cdot 4^2$, or 4^3 in 3D). The distribution of non-zero entries per row is similar for all sampling patterns.

As previously discussed, the runtime for interpolation is cMW^d , where c depends on the hardware, implementation, and sampling pattern. Fig. 12-13 show interpolation times with linear fits for various values of MW^d , corresponding to the problems represented in Fig. 9-10. Comparing Fig. 11 with Fig. 12-13, we observe that the non-Cartesian sampling pattern affects the interpolation runtime via 1) the spatial distribution of samples and 2) the ordering of the samples in memory.

First, the spatial distribution of samples affects the pattern of memory accesses, as reflected by the adjoint interpolation runtimes. Unequal spatial distribution of points may result in serialized atomics in the precomputation-free method or imbalanced row lengths in the fully precomputed method, both of which result in increased interpolation runtime. The slope of interpolation time vs. number of non-zero entries for the radial sampling pattern is higher than that of the stack of stars and cones sampling patterns, due to the unequal spatial distribution of samples.

Second, the ordering of the samples in memory relative to their spatial positions affects caching of global memory and the interpolation runtime. For example, the high slope for the random sampling pattern with precomputation-free method in the adjoint direction can be attributed to the random ordering of the non-uniform samples relative to their spatial positions. In the forward direction, although all sampling patterns have similar distributions of non-zeros per row, the slope is highest for the random sampling pattern. In the radial, stack of stars, and cones sampling patterns, consecutive non-Cartesian samples are spatially near one another and tend to access the same Cartesian grid points, which are loaded into cache. In the random sampling pattern, however, non-Cartesian samples have random ordering relative to their spatial location. Cartesian grid points are repeatedly entering and leaving cache with non-coalesced access, resulting in increased interpolation runtime. Reordering the non-Cartesian samples can increase cache performance and encourage coalesced memory accesses, although strategies for reordering are not implemented in this work.

VII. CONCLUSION

In this work, we have presented a fast, flexible NUFFT library for auto-tuning NUFFT computations on the GPU. The library contains implementations of gridding- and Toeplitz-based methods and can be easily modified to perform auto-tuning using other existing libraries. In addition, we present analysis of the different NUFFT implementations along a spectrum of precomputation levels. Choosing appropriate implementations as well as parameter values is crucial to NUFFT performance.

The auto-tuning approach in this library uses an exhaustive search over implementations and algorithm parameters, which can be time-consuming. Developing heuristics to select the implementations and/or algorithm parameters can reduce the planning time and improve the utility of the library.

Further improvements can be explored to increase the performance of the NUFFT implementations described in this work. Performance of the partial precomputation method can be improved by auto-tuning the sector width to increase occupancy or by handling the shared memory limitation in other ways. In the fully precomputed method, other storage formats such as HYB or novel storage formats can be explored to exploit specific non-uniform sampling patterns. Additional algorithms may be also added to the optimization search space, such as fast methods for evaluating NUDFT on the GPU [59].

Additional areas of exploration include optimizing oversampling ratios for anisotropic image sizes, using radial instead of separable kernels [60], or extending the library to include multi-GPU systems.

Distribution of Non-zeros Per Row in Sparse Interpolation Matrices

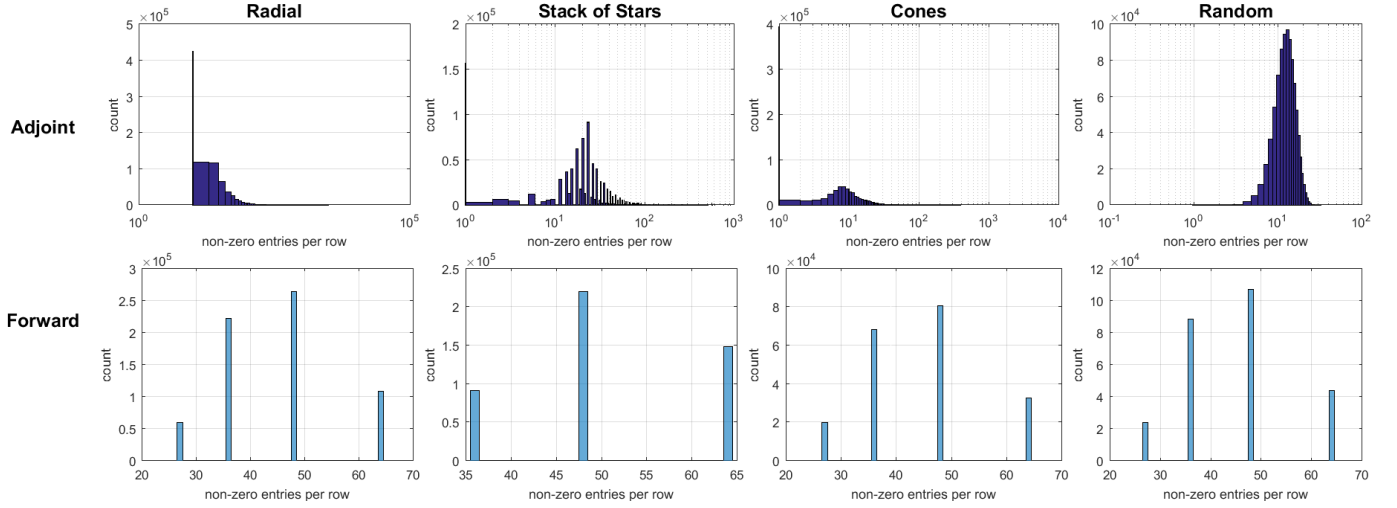


Fig. 11. Distribution of row lengths in fully precomputed sparse matrices Γ and Γ^\top , for $N = 64^3$, $\varepsilon = 0.01$, $R = 1$. The row length of Γ^\top indicates the number of neighboring non-Cartesian samples for each Cartesian grid point, whereas the row length of Γ indicates the number of Cartesian neighbors for each non-Cartesian sample. The sparse matrix Γ^\top for the adjoint operation has a heavy right-tailed distribution. Note that the distribution for the sparse matrix Γ^\top in the adjoint operation is shown on a semilog scale. The radial sampling pattern has the widest variance in the distribution of row lengths, followed by stack of stars, cones, and random sampling patterns. The sparse matrix Γ for the forward operation has a lower variance in the row lengths than Γ^\top . Since each row may have $\{[W], [W]^\top\}^d$ non-zero entries, the variation in row length is similar across all sampling patterns for Γ . (Refer to Table III in App. C for standard deviations of each distribution.)

No Precomputation: Interpolation Time vs. Number of Nonzero Entries

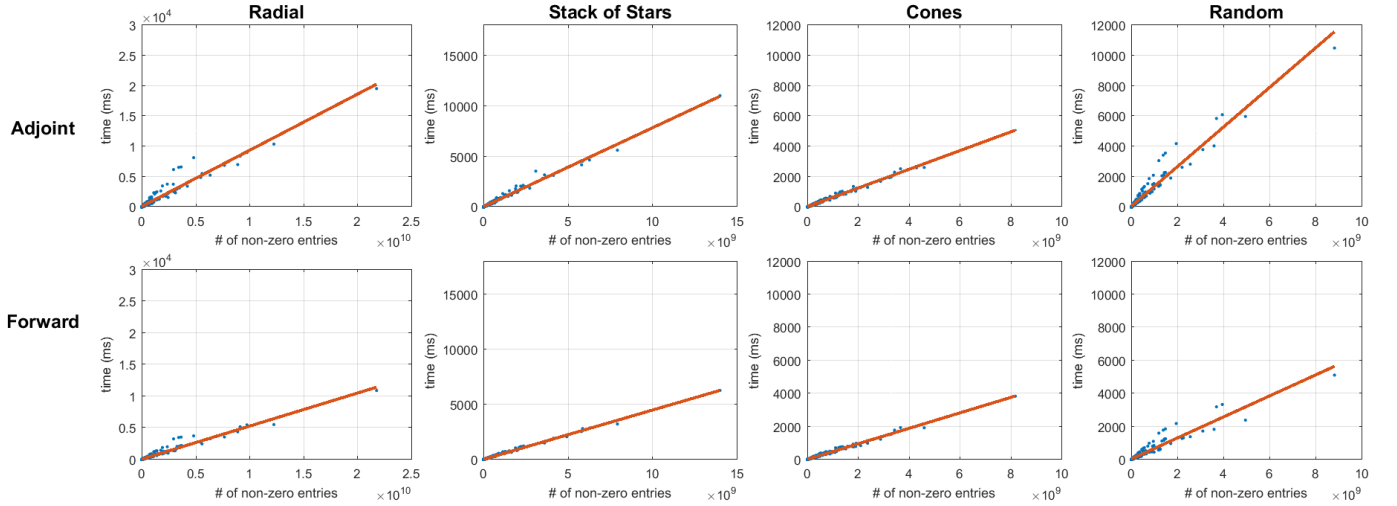


Fig. 12. The precomputation-free interpolation time is approximately linear with the number of grid contributions, MW^d . The slope of the linear fit in the adjoint operation shows the effect of the spatial distribution of non-Cartesian points from different sampling patterns. The slope is highest for the random sampling pattern, due to the random ordering of non-Cartesian samples which causes low cache performance. The next highest slope corresponds to the radial sampling pattern, which reflects the distributions shown in Fig. 11. The slope of the linear fit in the forward direction is similar for all sampling patterns except random, which is slightly higher, again due to the random ordering of non-Cartesian samples. Parameters: $\varepsilon = 0.01, 0.001$; $R = 1, 4, 16$; $N = 62^3, 94^3, 126^3, 158^3, 190^3, 222^3, 254^3$. (See Table IV in App. C for calculated slopes of linear fits.)

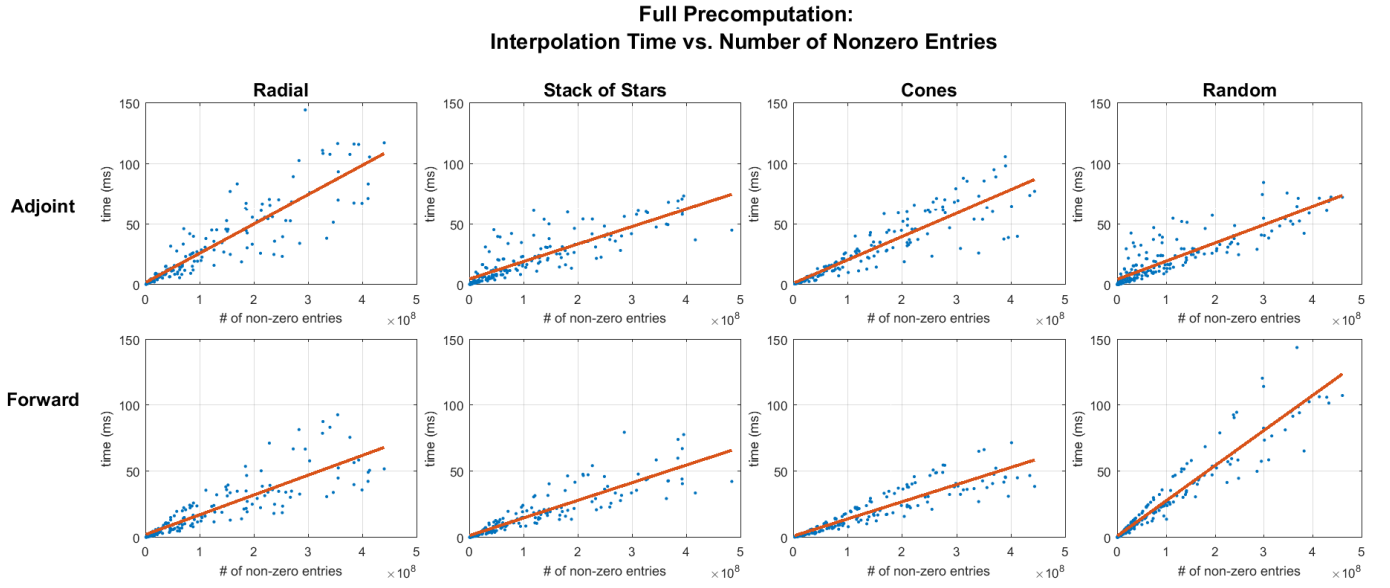


Fig. 13. Similar to the precomputation-free method, the fully precomputed interpolation time is also approximately linear with the number of non-zero entries in the precomputed sparse matrix. The slope for the radial sampling pattern in the adjoint direction is relatively high, due to the uneven distribution of row lengths. The random sampling pattern has a high slope in the forward direction, due to the random ordering of non-uniform samples. All parameters are the same as in Fig. 12. (See Table IV in App. C for calculated slopes of linear fits.)

APPENDIX A OPEN-SOURCE SOFTWARE LIBRARY

Our library performs auto-tuning across different NUFFT implementations and their parameters. We include precomputation-free, fully precomputed, and Toeplitz-based implementations, with the ability to include and perform auto-tuning on other implementations from external libraries. Our library computes 2D and 3D NUFFTs and allows for anisotropic image sizes. The library provides C and Matlab interfaces.

The user specifies the desired maximum aliasing amplitude and provides the coordinates of the non-uniform samples to `nufftPlan`, which performs auto-tuning and precomputes any required data structures on the GPU.

The space of auto-tuning is determined by a user-provided flag, similar to the planning flags in FFTW [30]. Flags include:

- `NUFFT_EXHAUSTIVE`: library auto-tunes over different implementations as well as algorithm parameters
- `NUFFT_CHOOSE_TYPE`: user specifies a desired implementation and library auto-tunes over algorithm parameters only
- `NUFFT_CHOOSE`: user specifies both implementation and oversampling ratio

Additional flags can be added to the library to specify different sets of implementations of interest, over which auto-tuning will be performed.

When using `NUFFT_CHOOSE_TYPE` or `NUFFT_CHOOSE`, the user must supply additional arguments for the desired implementation and/or parameter value (`type` and `a`). When the user specifies an implementation type, `type_args` may be used to indicate the operations where precomputation is used. For example, the user may specify that sparse matrices are precomputed for the adjoint operation but not the forward operation.

To summarize, the supplied arguments to `nufftPlan` include:

- `h_kcoord`: coordinates of non-uniform samples (array of $\text{dim} \times \text{Mtotal}$)
- `h_dcf`: density compensation factors (or `NULL`)
- `N`: image size (vector of length `dim`)
- `M`: scaling of non-Cartesian coordinates (vector of length `dim`), where all scaled coordinates `h_kcoord[i, :]/M[i]` lie within $[-0.5, 0.5]$
- `Mtotal`: total number of non-Cartesian samples
- `dim`: dimensionality of NUFFT
- `e`: desired maximum aliasing amplitude
- `isNoncartesianKspace`: true if frequency domain is non-uniform, false if image domain is non-uniform
- `Ntrials`: number of times to measure execution times during auto-tuning
- `flag`: auto-tuning flag
- Additional arguments
 - `type`: NUFFT implementation
 - `type_args`: specify directions where precomputation is used (vector of length 3)
 - `a`: oversampling ratio

After creating a NUFFT plan, the computation can be performed using `nufftExec`, which accepts pointers to input and output data on the GPU and the direction of the operation (`NUFFT_ADJOINT`, `NUFFT_FORWARD`, or `NUFFT_FORWARD_ADJOINT`). When the plan is no longer required, GPU and CPU resources may be released by calling `nufftDestroy`.

An example usage of the C library interface is shown here:

```
nufftHandle plan;
float *h_kcoord;
float e;
cuComplex *d_idata, *d_odata;
...
nufftPlan(&plan, h_kcoord, h_dcf,
          N, M, Mtotal,
          dim, e, isNoncartesianKspace,
          Ntrials, flag,
          [type, type_args, a]);
...
nufftExec(plan, d_idata, d_odata, direction);
...
nufftDestroy(plan);
```

APPENDIX B

ALIASING AMPLITUDE OF N-D PRESAMPLED KERNEL

Beatty et al. [29] derive the sampling density required for presampling a 1D kernel, which will be linearly interpolated. We extend the derivation to higher dimensions using a linearly separable kernel.

For example, in 2D we obtain the following equations. The inverse Fourier Transform of the linearly interpolated kernel is modulated by the envelope function

$$h(x, y) = \text{sinc}^2\left(\frac{x}{SG}\right) \text{sinc}^2\left(\frac{y}{SG}\right), \quad (15)$$

where S is the kernel sampling density.

The aliasing amplitude due to the linear interpolation and sampling density is

$$\varepsilon_1[i, j] = \sqrt{\left[\frac{\hat{h}(i, j)}{h(i, j)}\right]^2 - 1}, \quad (16)$$

where

$$\hat{h}(i, j) = \sqrt{\sum_{q, r} [h(i + SGq, j + SGr)]^2} \quad (17)$$

$$= \sqrt{\mathbb{E} \left\{ [h(x, y)]^2 \text{III} \left(\frac{x-i}{SG}, \frac{y-j}{SG} \right) \right\}} \quad (18)$$

$$= \sqrt{\left(\frac{2}{3} + \frac{1}{3} \cos \left(2\pi \frac{i}{SG} \right) \right) \left(\frac{2}{3} + \frac{1}{3} \cos \left(2\pi \frac{j}{SG} \right) \right)} \quad (19)$$

Substituting (19) into (16), we obtain

$$\varepsilon_1[i, j] = \sqrt{\frac{\left(\frac{2}{3} + \frac{1}{3} \cos\left(2\pi \frac{i}{SG}\right)\right) \left(\frac{2}{3} + \frac{1}{3} \cos\left(2\pi \frac{j}{SG}\right)\right)}{\text{sinc}^4\left(\frac{i}{SG}\right) \text{sinc}^4\left(\frac{j}{SG}\right)}} - 1 \quad (20)$$

$$\approx \sqrt{2} \frac{\pi^2}{3\sqrt{5}} \left(\frac{i}{SG}\right)^2 \quad (21)$$

when $i = j = -N/2$.

In n dimensions, the linear approximation of ε_1 is

$$\varepsilon_{1,\text{nD}} \approx \varepsilon_{1,\text{1D}} \sqrt{n} \quad (22)$$

where $\varepsilon_{1,\text{1D}} \approx 0.37/(\alpha S)^2$ when $i = -N/2$.

APPENDIX C SUPPLEMENTAL DATA

A. Effects of Oversampling Ratio on Execution Time

Here we show the effect of the oversampling ratio on the NUFFT execution time. Fig. 14 shows the NUFFT execution time in the forward direction, while Fig. 6 shows the execution time for the adjoint direction for $N = 158^3$ and $\varepsilon = 0.01$ with a rotated stack of stars sampling pattern. In both cases, as the oversampling ratio increases and the kernel width decreases, the FFT runtime increases and the interpolation time decreases. The total time consists of a trade-off between the interpolation runtime and FFT runtime.

Again, this trade-off varies with the acceleration factor and level of precomputation. At low acceleration factors, the optimal oversampling ratio tends to be close to two, while at high acceleration factors, the optimal oversampling ratio tends to be lower. When the overall runtime is more evenly balanced between interpolation and FFT (e.g. full precomputation or high maximum aliasing amplitude as shown in Fig. 7), the optimal oversampling ratio is more sensitive to the acceleration factor R .

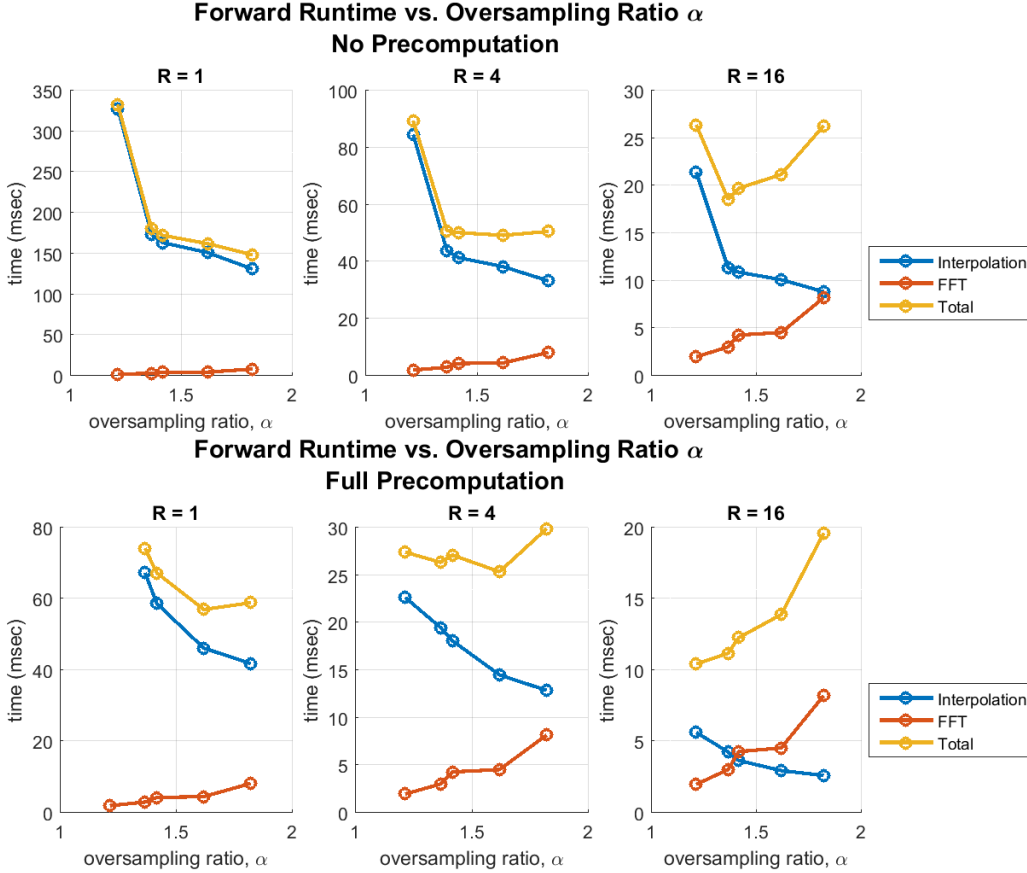


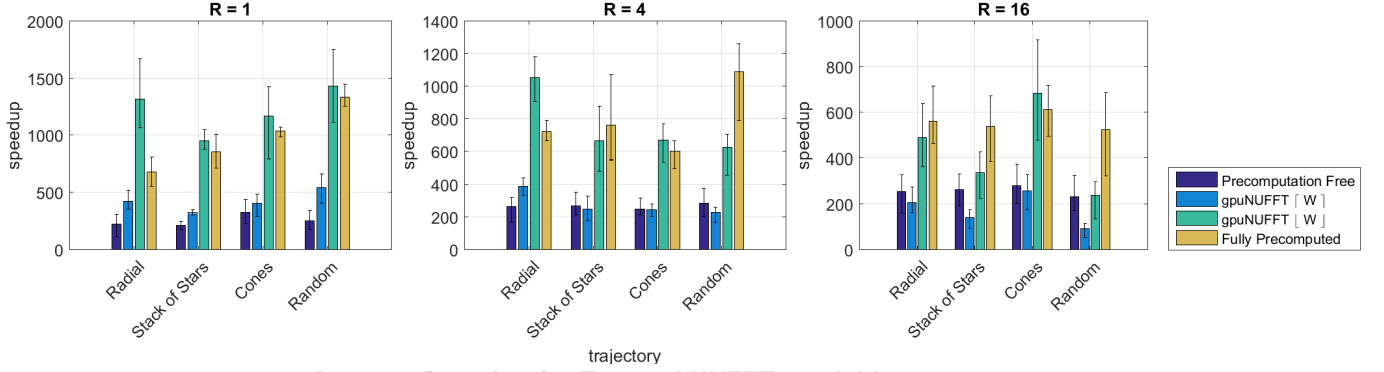
Fig. 14. Effects of oversampling ratio on execution time. As the oversampling ratio increases, the FFT time increases, while interpolation time (convolution or SpMV) decreases due to the decrease in kernel width. In the precomputation-free method, convolution time dominates the total execution time, whereas the runtime is more evenly balanced between SpMV and FFT for the fully precomputed method. Parameters: $N = 158^3$, $\varepsilon = 0.01$, stack of stars sampling pattern.

B. Average Speedup in Forward and Adjoint Directions for Various NUFFT Implementations

The results in Fig. 9-10 showed the speedup of the forward and adjoint NUFFT performed in succession. Here we show the speedups for the adjoint and forward NUFFTs separately. We compare different auto-tuned NUFFT implementations with radial, stack of stars, cones, and random sampling patterns over different acceleration factors $R = 1, 4, 16$ and maximum aliasing amplitudes $\varepsilon = 0.01, 0.001$.

We observe similar trends, e.g. higher levels of precomputation generally result in faster speedups, except in the case when the maximum aliasing amplitude is low ($\varepsilon = 0.001$) and shared memory usage in the partial precomputation method is increased, which lowers its performance.

Average Speedup for Adjoint NUFFT, $\varepsilon = 0.01$



Average Speedup for Forward NUFFT, $\varepsilon = 0.01$

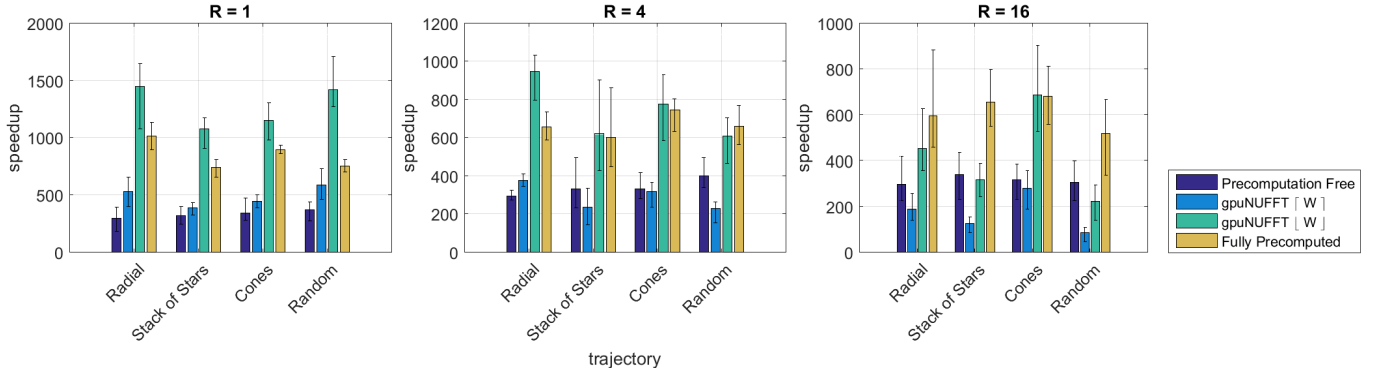
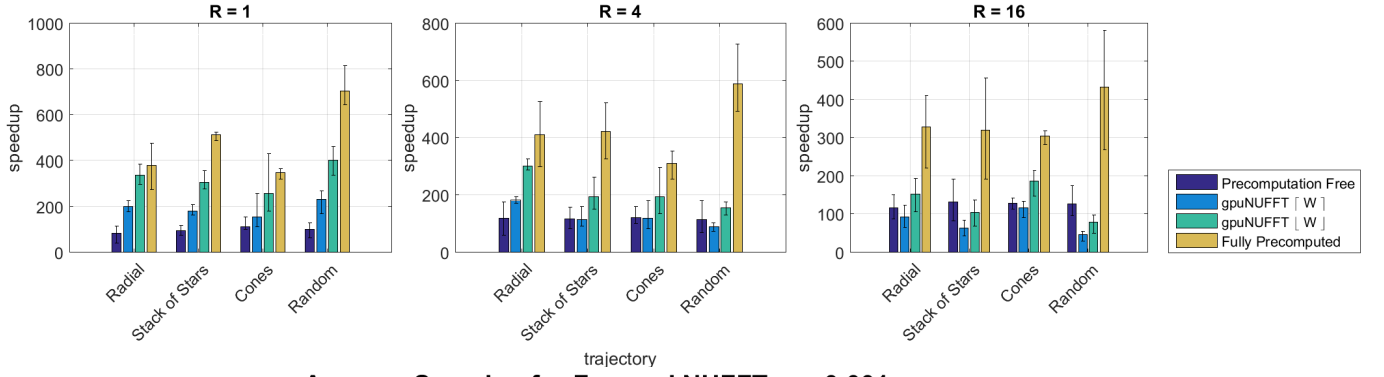


Fig. 15. Speedup over CPU implementation in forward and adjoint directions, averaged across image sizes, for isotropic 3D problems with different sampling patterns. Error bars show minimum and maximum speedup. Parameters: $\varepsilon = 0.01$; $R = 1, 4, 16$; $N = 126^3, 158^3, 190^3, 222^3, 254^3$. (Speedups for $N = 62^3, 94^3$ are not shown here, since the performance of partial precomputation is reduced for smaller problem sizes.)

Average Speedup for Adjoint NUFFT, $\varepsilon = 0.001$



Average Speedup for Forward NUFFT, $\varepsilon = 0.001$

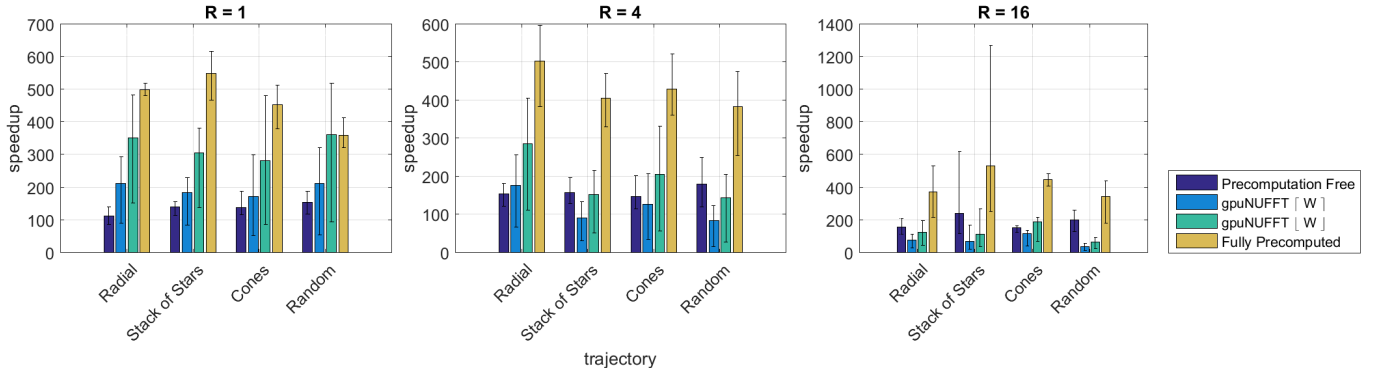


Fig. 16. Speedup over CPU implementation in forward and adjoint directions, averaged across image sizes, for isotropic 3D images with different non-uniform sampling patterns. Error bars show minimum and maximum speedup. Parameters: $\varepsilon = 0.001$; all other parameters same as in Fig. 15.

C. Speedup of NUFFT vs. Image Dimension

Fig. 9,10,15,16 showed speedups averaged over the image size N to summarize the timing measurements. Here we include the complete timing measurements, with the speedup plotted over the isotropic image dimension $N^{1/3}$ for all operations of interest (forward + adjoint, forward, adjoint). All parameters are the same as in Fig. 9,10,15,16.

The speedups shown here for $N = 126^3, 158^3, 190^3, 222^3, 254^3$ were used to calculate the average speedups in Fig. 9,10,15,16. $N = 62^3, 94^3$ were not used, since the performance for the partial precomputation method is lowered, due to overhead in recombining the intermediate outputs from each sector, which is less suitable for smaller problems. At larger image sizes, however, the individual speedups shown here have similar trends to the averaged speedups in Fig. 9,10,15,16.

Fig. 17-18 show speedups of the forward + adjoint NUFFT for $\varepsilon = 0.01$ and $\varepsilon = 0.001$, respectively. Similarly, Fig. 19-20 shows speedups of the adjoint NUFFT, and Fig. 21-22 shows speedups of the forward NUFFT, for $\varepsilon = 0.01$ and $\varepsilon = 0.001$.

Speedup of Forward + Adjoint NUFFT vs. Image Dimension, $\varepsilon = 0.01$

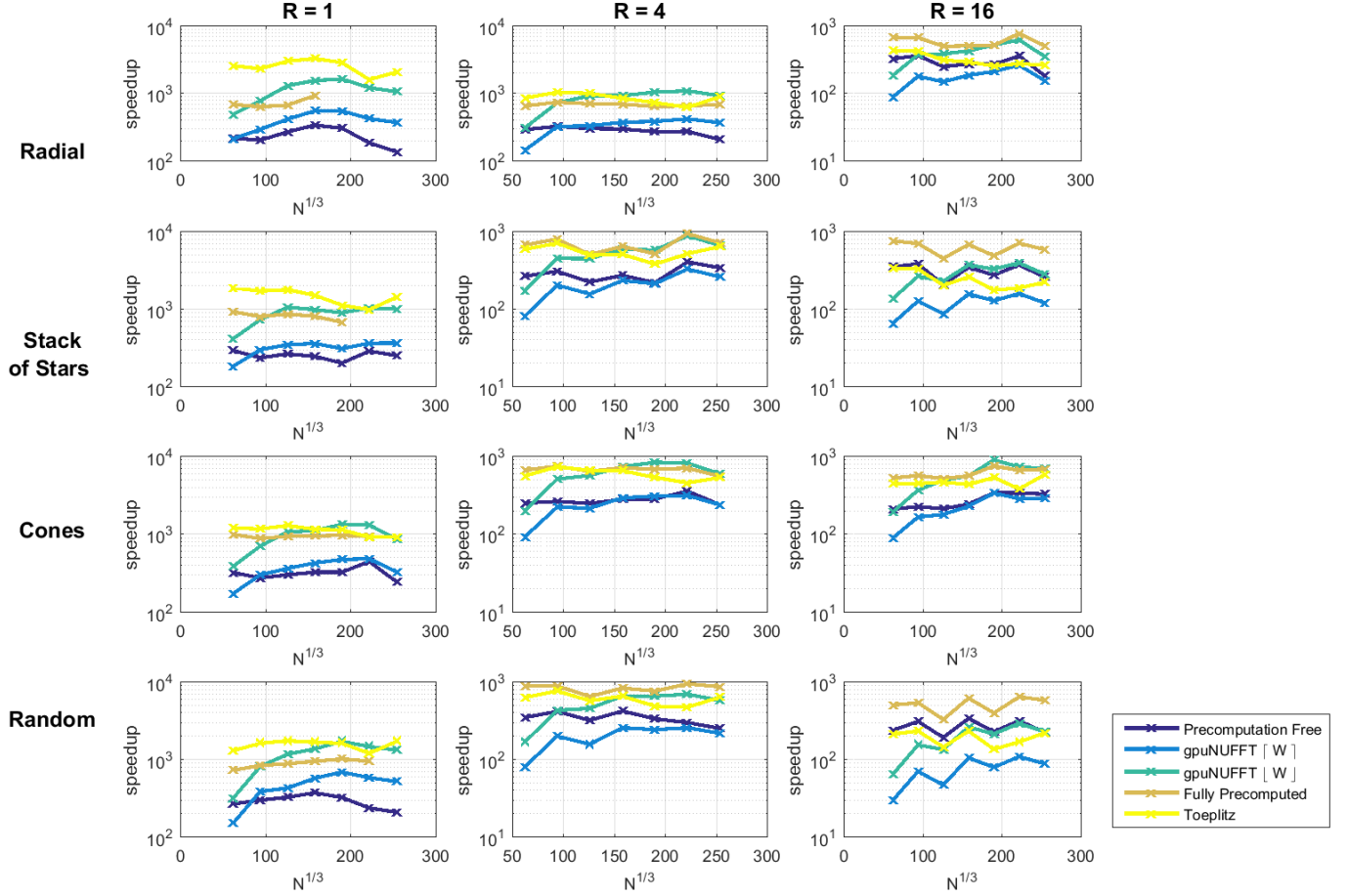


Fig. 17. Speedup over CPU implementation for forward + adjoint NUFFT, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. Parameters: $\varepsilon = 0.01$; $N = 62^3, 94^3, 126^3, 158^3, 190^3, 222^3, 254^3$.

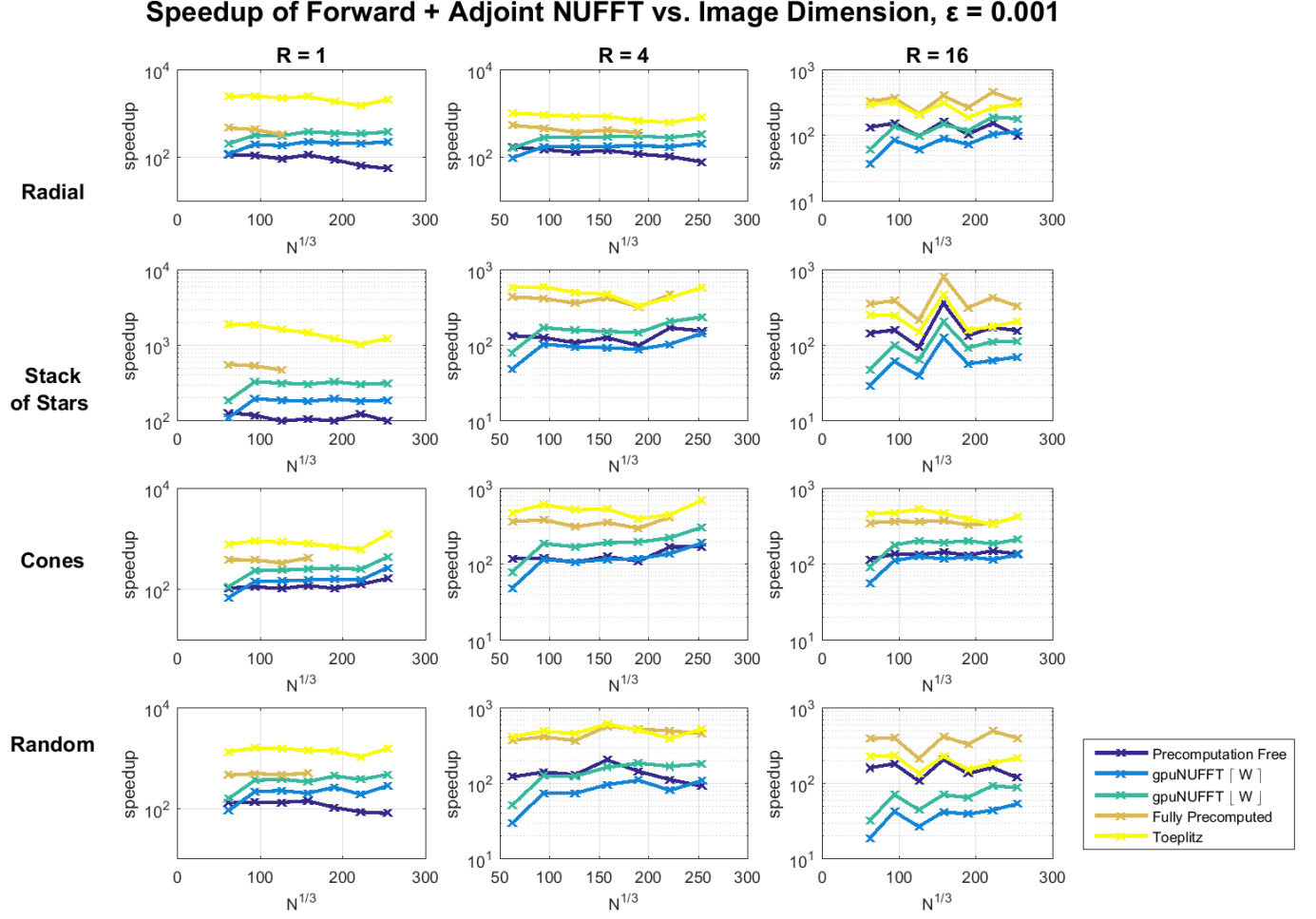


Fig. 18. Speedup over CPU implementation for forward + adjoint NUFFT, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. Parameters: $\varepsilon = 0.001$; all other parameters same as in Fig. 17.

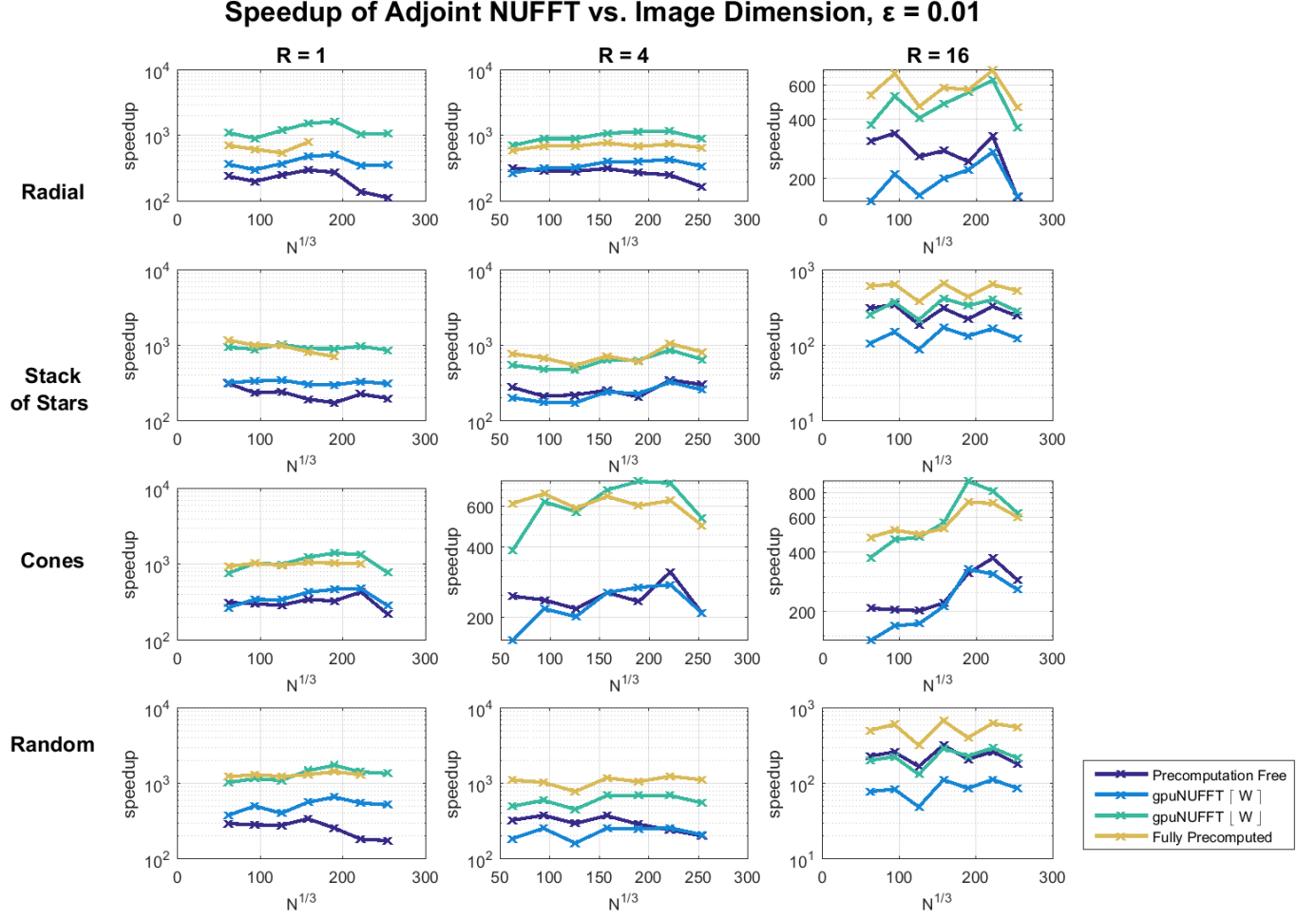


Fig. 19. Speedup over CPU implementation in adjoint direction, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. All parameters same as in Fig. 17.

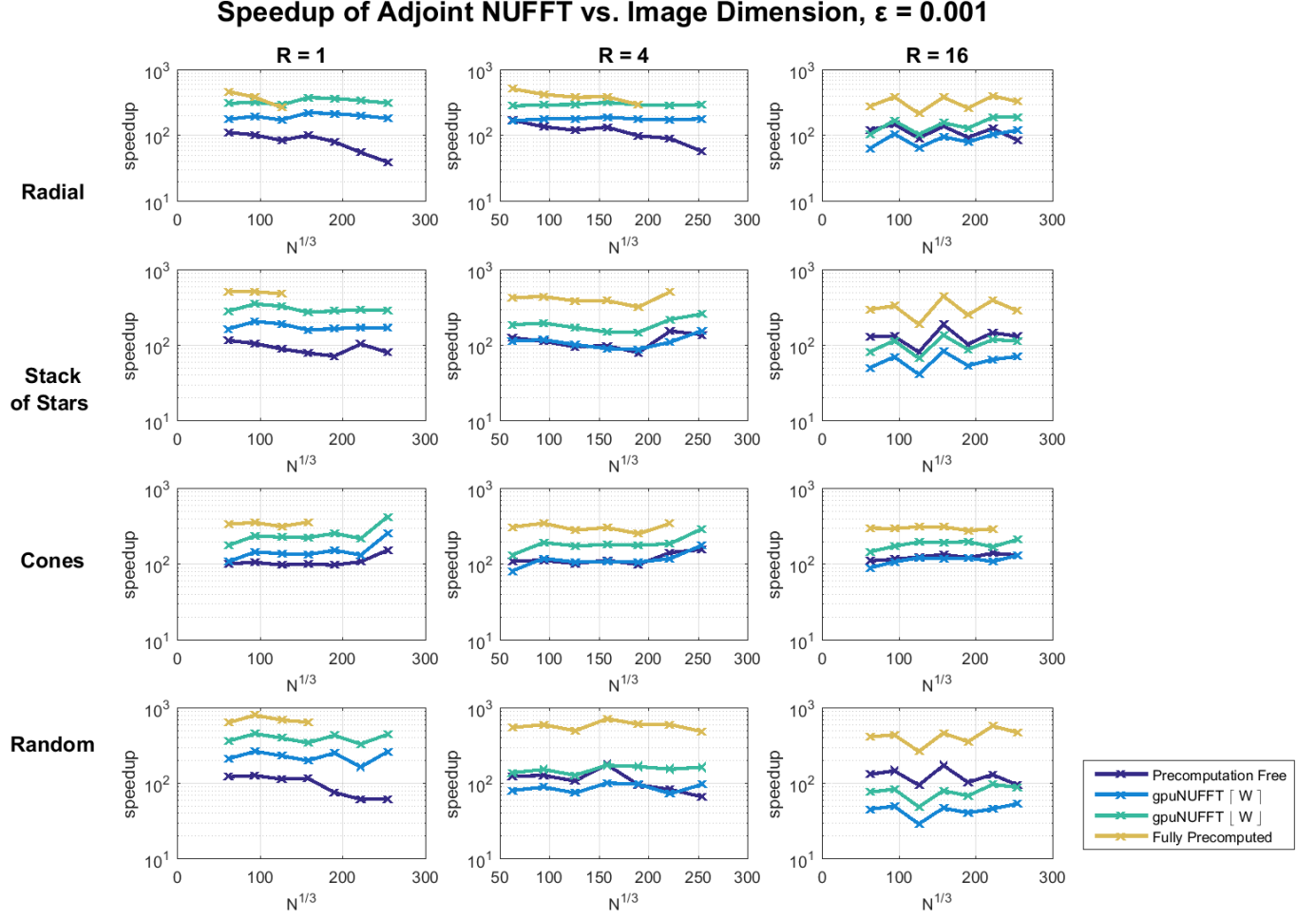


Fig. 20. Speedup over CPU implementation in adjoint direction, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. All parameters same as in Fig. 18.

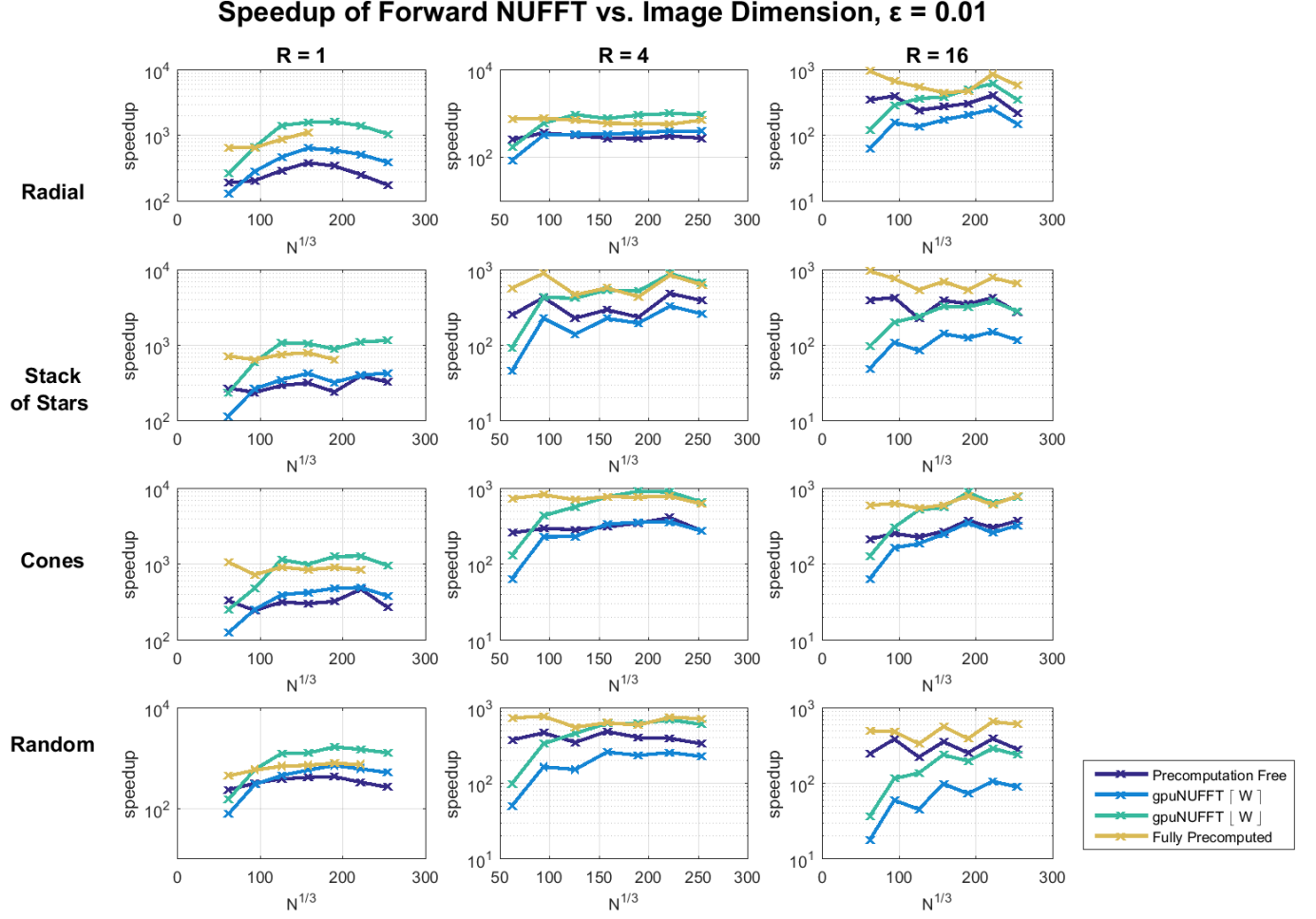


Fig. 21. Speedup over CPU implementation in forward direction, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. All parameters same as in Fig. 17.

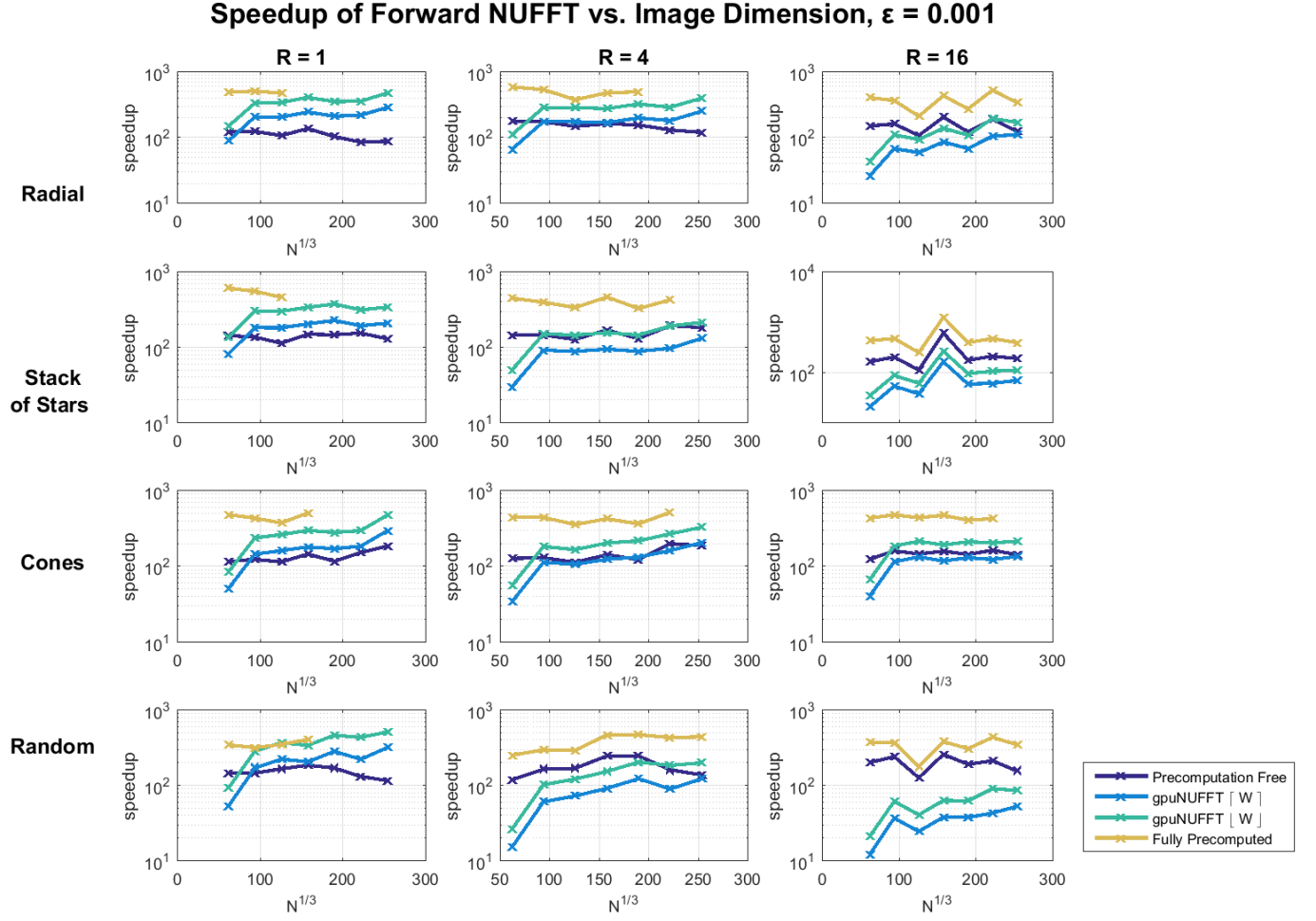


Fig. 22. Speedup over CPU implementation in forward direction, for isotropic 3D images with different non-uniform sampling patterns and acceleration factors, vs. size of image $N^{1/3}$. All parameters same as in Fig. 18.

D. Effects of Non-Cartesian Sampling Pattern on Performance of Interpolation

We analyze the effect of different non-Cartesian sampling patterns on the performance of interpolation, by comparing the interpolation runtime with the distribution of non-zero entries per row of the sparse matrices Γ and Γ^\top . The following tables correspond to Fig. 11, which shows representative distributions for each non-Cartesian sampling pattern, and Fig. 12-13, which include interpolation times from all experiments described in Section V.

Table III lists the standard deviations of the distributions shown in Fig. 11. Table IV lists the slopes of the linear fits of the interpolation runtime vs. total number of non-zero entries. In general, a high slope in the linear fit corresponds to a high standard deviation of the distribution of non-zero entries per row. If the rows are unevenly distributed, then the work distribution between threads in the precomputation-free method (or warps in the fully precomputed method) is unequal. Threads that are assigned the most work finish executing last, while other threads that finish earlier must wait, increasing the interpolation runtime. This correspondence between the linear fit slopes and the standard deviations generally holds true, except in the case of the random sampling pattern where the random ordering of non-Cartesian samples lowers cache performance and increases the interpolation runtime.

TABLE III. STANDARD DEVIATION OF NON-ZERO ENTRIES PER ROW OF PRECOMPUTED SPARSE MATRICES Γ AND Γ^\top , WITH DISTRIBUTIONS SHOWN IN FIG. 11.

Sampling Pattern	Adjoint	Forward
Radial	224.182	10.950
Stack of Stars	40.563	10.170
Cones	28.523	11.001
Random	3.647	10.974

TABLE IV. SLOPE OF LINEAR FIT BETWEEN INTERPOLATION RUNTIME AND NUMBER OF NON-ZERO ENTRIES MW^d IN PRECOMPUTED SPARSE MATRICES Γ , Γ^\top , WITH R^2 VALUES, CORRESPONDING TO FIG. 12-13.

Sampling Pattern	No Precomputation (ms/non-zero entry (R^2))		Full Precomputation (ms/non-zero entry (R^2))	
	Adjoint	Forward	Adjoint	Forward
Radial	9.266e-7 (0.933)	5.217e-7 (0.950)	2.420e-7 (0.790)	1.499e-7 (0.735)
Stack of Stars	7.814e-7 (0.987)	4.473e-7 (0.994)	1.444e-7 (0.733)	1.334e-7 (0.792)
Cones	6.172e-7 (0.989)	4.676e-7 (0.986)	1.929e-7 (0.811)	1.299e-7 (0.834)
Random	13.061e-7 (0.943)	6.350e-7 (0.905)	1.503e-7 (0.772)	2.662e-7 (0.912)

ACKNOWLEDGMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

REFERENCES

- [1] K. P. Pruessmann, M. Weiger, P. Börnert, and P. Boesiger, "Advances in sensitivity encoding with arbitrary k-space trajectories," *Magnetic Resonance in Medicine*, vol. 46, no. 4, pp. 638–651, 2001.
- [2] J. G. Pipe *et al.*, "Motion correction with propeller mri: application to head motion and free-breathing cardiac imaging," *Magnetic Resonance in Medicine*, vol. 42, no. 5, pp. 963–969, 1999.
- [3] K. L. Wright, J. I. Hamilton, M. A. Griswold, V. Gulani, and N. Seiberlich, "Non-cartesian parallel imaging reconstruction," *Journal of Magnetic Resonance Imaging*, vol. 40, no. 5, pp. 1022–1040, 2014.
- [4] K. K. Chan and S. Tang, "High-speed spectral domain optical coherence tomography using non-uniform fast fourier transform," *Biomedical optics express*, vol. 1, no. 5, pp. 1309–1319, 2010.
- [5] K. Wang, Z. Ding, T. Wu, C. Wang, J. Meng, M. Chen, and L. Xu, "Development of a non-uniform discrete fourier transform based high speed spectral domain optical coherence tomography system," *Optics express*, vol. 17, no. 14, pp. 12 121–12 131, 2009.
- [6] K. Zhang and J. U. Kang, "Graphics processing unit accelerated non-uniform fast fourier transform for ultrahigh-speed, real-time fourier-domain oct," *Optics express*, vol. 18, no. 22, pp. 23 472–23 487, 2010.
- [7] M. Haltmeier, O. Scherzer, and G. Zangerl, "A reconstruction algorithm for photoacoustic imaging based on the nonuniform fft," *IEEE transactions on medical imaging*, vol. 28, no. 11, pp. 1727–1735, 2009.
- [8] M. M. Bronstein, A. M. Bronstein, M. Zibulevsky, and H. Azhari, "Reconstruction in diffraction ultrasound tomography using nonuniform fft," *IEEE transactions on medical imaging*, vol. 21, no. 11, pp. 1395–1401, 2002.
- [9] S. Schaller, T. Flohr, and P. Steffen, "An efficient fourier method for 3-d radon inversion in exact cone-beam ct reconstruction," *IEEE transactions on medical imaging*, vol. 17, no. 2, pp. 244–250, 1998.
- [10] X. Zhou, H. Sun, J. He, and X. Lu, "Nufft-based iterative reconstruction algorithm for synthetic aperture imaging radiometers," *IEEE Geoscience and Remote Sensing Letters*, vol. 6, no. 2, pp. 273–276, 2009.
- [11] X. Y. He, X. Y. Zhou, and T. J. Cui, "Fast 3d-isar image simulation of targets at arbitrary aspect angles through nonuniform fast fourier transform (nufft)," *IEEE Transactions on Antennas and Propagation*, vol. 60, no. 5, pp. 2597–2602, 2012.
- [12] S. Wenger, M. Magnor, Y. Pihlström, S. Bhatnagar, and U. Rau, "Sparseri: A compressed sensing framework for aperture synthesis imaging in radio astronomy," *Publications of the Astronomical Society of the Pacific*, vol. 122, no. 897, p. 1367, 2010.
- [13] A. E. Yagle, "Closed-form reconstruction of images from irregular 2-d discrete fourier samples using the good-thomas fft," in *Image Processing, 2000. Proceedings. 2000 International Conference on*, vol. 1. IEEE, 2000, pp. 117–119.
- [14] B. C. Lee and A. E. Yagle, "A sensitivity measure for image reconstruction from irregular 2-d dtft samples," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 4. IEEE, 2002, pp. IV–3245.
- [15] J. A. Fessler and B. P. Sutton, "Nonuniform fast fourier transforms using min-max interpolation," *IEEE Transactions on Signal Processing*, vol. 51, no. 2, pp. 560–574, 2003.
- [16] Q. H. Liu and X. Y. Tang, "Iterative algorithm for nonuniform inverse fast fourier transform (nu-iff)," *Electronics Letters*, vol. 34, no. 20, pp. 1913–1914, 1998.
- [17] T. Knopp, S. Kunis, and D. Potts, "A note on the iterative mri reconstruction from nonuniform k-space data," *International journal of biomedical imaging*, vol. 2007, 2007.
- [18] J. G. Pipe and P. Menon, "Sampling density compensation in mri: rationale and an iterative numerical solution," *Magnetic resonance in medicine*, vol. 41, no. 1, pp. 179–186, 1999.
- [19] D. Rosenfeld, "New approach to gridding using regularization and estimation theory," *Magnetic resonance in medicine*, vol. 48, no. 1, pp. 193–202, 2002.
- [20] J. A. Fessler, "On nufft-based gridding for non-cartesian mri," *Journal of Magnetic Resonance*, vol. 188, no. 2, pp. 191–195, 2007.
- [21] L. Sha, H. Guo, and A. W. Song, "An improved gridding method for spiral mri using nonuniform fast fourier transform," *Journal of Magnetic Resonance*, vol. 162, no. 2, pp. 250–258, 2003.
- [22] V. Rasche, R. Proksa, R. Sinkus, P. Bornert, and H. Eggers, "Resampling of data between arbitrary grids using convolution interpolation," *IEEE transactions on medical imaging*, vol. 18, no. 5, pp. 385–392, 1999.
- [23] Q. Liu and N. Nguyen, "An accurate algorithm for nonuniform fast fourier transforms (nufft's)," *IEEE Microwave and guided wave letters*, vol. 8, no. 1, pp. 18–20, 1998.
- [24] N. Nguyen and Q. H. Liu, "The regular fourier matrices and nonuniform fast fourier transforms," *SIAM Journal on Scientific Computing*, vol. 21, no. 1, pp. 283–293, 1999.
- [25] A. Dutt and V. Rokhlin, "Fast fourier transforms for nonequispaced data," *SIAM Journal on Scientific computing*, vol. 14, no. 6, pp. 1368–1393, 1993.
- [26] J. O'sullivan, "A fast sinc function gridding algorithm for fourier inversion in computer tomography," *IEEE Transactions on Medical Imaging*, vol. 4, no. 4, pp. 200–207, 1985.
- [27] H. Schomberg and J. Timmer, "The gridding method for image reconstruction by fourier transformation," *IEEE transactions on medical imaging*, vol. 14, no. 3, pp. 596–607, 1995.
- [28] G. E. Sarty, R. Bennett, and R. W. Cox, "Direct reconstruction of non-cartesian k-space data using a nonuniform fast fourier transform," *Magnetic Resonance in Medicine*, vol. 45, no. 5, pp. 908–915, 2001.
- [29] P. J. Beatty, D. G. Nishimura, and J. M. Pauly, "Rapid gridding reconstruction with a minimal oversampling ratio," *IEEE transactions on medical imaging*, vol. 24, no. 6, pp. 799–808, 2005.
- [30] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.
- [31] C. Nvidia, "Cufft library," 2010.
- [32] M. Murphy, "Parallelism, patterns, and performance in iterative mri reconstruction," Ph.D. dissertation, University of California, Berkeley, 2011.
- [33] C. NVIDIA, "Cuspars library," *NVIDIA Corporation, Santa Clara, California*, 2014.
- [34] J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a convolution function for fourier inversion using gridding (computerized tomography application)," *IEEE transactions on medical imaging*, vol. 10, no. 3, pp. 473–478, 1991.
- [35] S. S. Stone, J. P. Haldar, S. C. Tsao, B. Sutton, Z.-P. Liang *et al.*, "Accelerating advanced mri reconstructions on gpus," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1307–1318, 2008.
- [36] A. Gregerson, "Implementing fast mri gridding on gpus via cuda," *Nvidia Tech. Report on Medical Imaging using CUDA*, 2008.
- [37] S. Nam and T. Basha, "A gpu implementation of compressed sensing reconstruction of 3d radial (kooshball) acquisition for high-resolution cardiac mri," 2011.
- [38] F. Knoll, A. Schwarzl, C. Diwoki, and D. K. Sodickson, "gpnufft - an open-source gpu library for 3d gridding with direct matlab interface," in *Proc. Intl. Soc. Mag. Reson. Med.* 22, 2014.
- [39] T. S. Sørensen, T. Schaeffter, K. Ø. Noe, and M. S. Hansen, "Accelerating the nonequispaced fast fourier transform on commodity graphics hardware," *IEEE Transactions on Medical Imaging*, vol. 27, no. 4, pp. 538–547, 2008.

- [40] N. Obeid, I. Atkinson, K. Thulborn, and W. Hwu, "Gpu-accelerated gridding for rapid reconstruction of non-cartesian mri," in *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2011.
- [41] F. Wajer and K. Pruessman, "Major speedup of reconstruction for sensitivity encoding with arbitrary trajectories," in *Proc. Intl. Soc. Mag. Reson. Med.* 9, 2001.
- [42] H. Eggers, P. Boernert, and P. Boesiger, "Comparison of gridding-and convolution-based iterative reconstruction algorithms for sensitivity-encoded non-cartesian acquisitions," in *Proceedings of the 10th Annual Meeting of ISMRM, Honolulu*, 2002, p. 743.
- [43] J. A. Fessler, S. Lee, V. T. Olafsson, H. R. Shi, and D. C. Noll, "Toeplitz-based iterative image reconstruction for mri with correction for magnetic field inhomogeneity," *IEEE Transactions on Signal Processing*, vol. 53, no. 9, pp. 3393–3402, 2005.
- [44] X.-L. Wu, J. Gai, F. Lam, M. Fu, J. P. Haldar, Y. Zhuo, Z.-P. Liang, W.-M. Hwu, and B. P. Sutton, "Impatient mri: Illinois massively parallel acceleration toolkit for image reconstruction with enhanced throughput in mri," in *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium on*. IEEE, 2011, pp. 69–72.
- [45] J. Keiner, S. Kunis, and D. Potts, "Nfft 3.0, c subroutine library," 2006.
- [46] —, "Using nfft 3—a software library for various nonequispaced fast fourier transforms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 4, p. 19, 2009.
- [47] M. Pippig and D. Potts, "Parallel three-dimensional nonequispaced fast fourier transforms and their application to particle simulation," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C411–C437, 2013.
- [48] A. Cerjanic, J. L. Holtrop, G. C. Ngo, B. Leback, G. Arnold, M. Van Moer, G. LaBelle, J. A. Fessler, and B. P. Sutton, "Powergrid: A open source library for accelerated iterative magnetic resonance image reconstruction," in *Proc. Intl. Soc. Mag. Reson. Med.* 24, 2016.
- [49] J. Gai, N. Obeid, J. L. Holtrop, X.-L. Wu, F. Lam, M. Fu, J. P. Haldar, W. H. Wen-mei, Z.-P. Liang, and B. P. Sutton, "More impatient: A gridding-accelerated toeplitz-based strategy for non-cartesian high-resolution 3d mri on gpus," *Journal of parallel and distributed computing*, vol. 73, no. 5, pp. 686–697, 2013.
- [50] M. Uecker, F. Ong, J. I. Tamir, D. Bahri, P. Virtue, J. Y. Cheng, T. Zhang, and M. Lustig, "Berkeley advanced reconstruction toolbox," in *Proceedings of the 23rd Annual Meeting ISMRM, Toronto*, 2015, p. 2486.
- [51] K. R. Rao, D. N. Kim, and J. J. Hwang, *Fast Fourier transform-algorithms and applications*. Springer Science & Business Media, 2011.
- [52] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
- [53] J. A. Fessler and D. C. Noll, "Iterative reconstruction methods for non-cartesian mri," in *Proc. ISMRM Workshop on Non-Cartesian MRI*, 2007.
- [54] H. V. Sorensen and C. S. Burrus, "Efficient computation of the dft with only a subset of input or output points," *IEEE transactions on signal processing*, vol. 41, no. 3, pp. 1184–1200, 1993.
- [55] J. Fessler *et al.*, "Image reconstruction toolbox," *Available at website: <http://www.eecs.umich.edu/fessler/code>*, 2012.
- [56] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
- [57] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: <http://cusplibrary.github.io/>
- [58] P. T. Gurney, B. A. Hargreaves, and D. G. Nishimura, "Design and analysis of a practical 3d cones trajectory," *Magnetic resonance in medicine*, vol. 55, no. 3, pp. 575–582, 2006.
- [59] K. Natarajan and N. Chandrathoodan, "Non-uniform dft implementation for channel simulations in gpu," in *Communications (NCC), 2015 Twenty First National Conference on*. IEEE, 2015, pp. 1–6.
- [60] M. Bydder, W. Zaaraoui, and J.-P. Ranjeva, "Use of a radial convolution kernel in the non-uniform fourier transform," in *Proc. Intl. Soc. Mag. Reson. Med.* 24, 2016.