nuFFTWC: The Fastest Non-Uniform FFT from the West Coast

Mark Murphy, Michal Zarrouk, Kurt Keutzer, and Michael Lustig

Abstract

We present a strategy for performance optimization of the Gridding non-uniform Fast Fourier Transform (nuFFT) algorithm, and evaluate our approach with workloads from Magnetic Resonance Imaging (MRI). The Gridding algorithm used in MRI reconstruction computes an approximate Fourier transform of a signal sampled at non-equispaced locations. Our optimization strategy relies on empirically-driven search over a range of error-equivalent gridding implementations. Empirical search has been called *auto-tuning* in prior works, and has proven effective for a number of other numerical libraries. We also adopt the split planning/execution programming interface used by these libraries. The range of implementations we consider is drawn from prior work on performance optimization of the gridding nuFFT. Our planner requires the transform to be specified via a desired Fourier transform approximation accuracy, a set of sampling locations, and potentially a restriction on memory use. Performance optimization can usually be performed off-line before the values of samples are known. We discuss an effective heuristic to reduce time spent in the planner when off-line optimization is impractical. Compared to an optimized sequential baseline, our multi-core CPU implementation achieves up to 50× performance improvement. Our implementation for graphics processors (GPUs) achieves up to 100× speedup over baseline.

1 Introduction

The Gridding algorithm is a non-uniform Fast Fourier Transform (nuFFT) implementation used in MRI [1] to approximate an inverse Fourier transform of non-equispaced k-space samplings. Due to the efficiency and image quality achievable by Gridding, it is the most commonly used algorithm for non-Cartesian reconstructions. As we'll discuss in detail in Section 2, the gridding nuFFT approximates the NDFT by re-sampling the data at equispaced locations on an over-sampled grid. The FFT is then used to compute the Fourier transform of the grid. Re-sampling k-space produces aliasing in the image domain, and over-sampling increases the field of view (FOV) of the reconstructed image. The interpolation kernel and degree of over-sampling are chosen so that aliasing artifacts appear in the extended FOV, rather than the central region containing the reconstructed image.

The accuracy and image quality of gridding in MRI applications has been well-studied in prior literature. Jackson *et al.* [2] evaluate a number of different interpolation kernels to be used during the re-sampling, determining that Kaiser-Bessel windows (with appropriate free-parameter selection) achieve very close to optimal reconstruction quality. Fessler *et al.* [3] propose a min-max criterion for gridding optimality that provides a closed-form solution for the optimal re-sampling interpolation kernel. Their kernel exhibits several desirable properties, including several options for partial precomputation to reduce memory footprint and runtime. Beatty *et al.* [4] discuss the selection of gridding parameters to substantially reduce memory footprint and runtime, while maintaining clinical-quality images. Additionally they propose the *maximum aliasing amplitude* metric, which provides a data-independent measure of gridding accuracy.

Efficient implementation of gridding nuFFT on modern parallel processors has been a topic of much recent study. These works focus on the resampling interpolation, and differ primarily in the amount of precomputation they perform. Sørensen *et al.*[5] describe a clever parallelization of resampling that pre-computes the set of non-equispaced samples nearby each equispaced grid location in order to obviate synchronization among parallel threads of execution. As their work was implemented on an early General-Purpose Graphics Processing Unit (GPGPU), this precomputation step was necessary for correctness. More recently Obeid *et al.* [6] describe a spatial "binning" step which sorts the non-Cartesian samples by their k-space coordinates, also to obviate inter-thread synchronization in a GPGPU implementation. Their approach permutes the non-equispaced samples, and potentially improves cache performance as well. In the same conference proceedings, Nam *et al.* [7] present a substantial GPGPU speedup for gridding of a 3D radial trajectory.

In this work, we propose an implementation of the gridding nuFFT and its adjoint, that uses a self-tuning optimization strategy to ensure high performance on modern parallel processing platforms. Our strategy permits a split plan/execute programming interface, similar to those used by other self-tuning libraries. The transform is specified to the planner as a pattern of non-uniform

samples, a final image matrix size, and a desired level of Fourier transform approximation accuracy. We note that if appropriate data structures are precomputed, all of the expensive computations in the nuFFT can be performed via extant highly optimized libraries. Much of the efficiency of our implementation is due to these libraries. We evaluate our approach using a variety of transforms drawn from the application of gridding nuFFT in MR Image reconstruction. A distributable version of the nuFFTWC is currently in development.

We rely heavily on the existence of highly optimized libraries for the FFT and sparse matrix operations, which have been well-studied in the scientific computing literature. Modern processor architectures are highly complex, and it is difficult to model performance accurately. For FFTs and sparse matrix operations, empirical search over a space of functionally equivalent implementations has proven a fruitful approach for designing high-performance libraries. These "self-tuning" libraries determine the optimal set of optimization parameters via a combination of benchmarking and heuristics. The ATLAS [8] (Automatically Tuned Linear Algebra Software) library is a self-tuning library for the Basic Linear Algebra Subroutines (BLAS) used heavily in most numerical applications. The well-known Fastest Fourier Transform in the West[1] (FFTW) [9] generates high-performance implementations of FFTs by evaluating alternative Cooley-Tukey factorizations, among a number of other implementation parameters. The Optimized Sparse Kernel Interface (OSKI) [10] is a self-tuning library for several common sparse matrix operations, including the matrix-vector multiplication we use in this work. The NFFT 3.0 library [11] is a very flexible implementation of the non-uniform FFT, suitable for use in a variety of numerical computing applications, including MRI reconstruction [12, 13]. The NFFT library provides a split plan/execute interface, similar to that of the FFTW library [9]. NFFT's planner provides a variety of precomputation options, including full precomputation of the matrix we define as $\mathbf{\Gamma}$ in Section 2, although their planner performs no empirically-driven performance optimization.

FFTW was the first library to perform performance optimization in a separate "planner" routine. Transforms are computed via a separate "execute" routine. The existence of a planning phase greatly simplifies the design and use of high-performance libraries. The optimizations that auto-tuning libraries perform require specification of some properties of the operation to be computed, such as FFT size or the locations of nonzeros in a sparse matrix. The planner phase takes as input only the specification of the operation, and returns an opaque "plan" object. Applications can re-use the same plan to perform multiple transforms, and amortize any time spent determining the optimal plan.

[1]The title of our work, nuFFTWC, is a tribute to FFTW

2 Gridding nuFFT Algorithm

Let $f(x)$ be a signal that we have sampled at M locations $\{x_m\}$. The signal is often complex-valued, and our index space is usually multi-dimensional. We denote by $F(k)$ the Fourier transform of f. The Fourier transform of f evaluated at a spatial frequency k_n is defined as the inner product with the appropriate Fourier basis function:

$$F_n = \frac{1}{\sqrt{M}} \sum_{m=1}^{M} f_m e^{-2\pi\imath(x_m \cdot k_n)} \tag{1}$$

where we define $f_m \triangleq f(x_m)$, $F_n \triangleq F(k_n)$, and $\imath = \sqrt{-1}$ as the imaginary number. Applications typically require evaluation of the Fourier transform at a set of N different frequencies $\{k_n\}$. In general, $M \neq N$ and we cannot define a unique inverse of the linear transformation described by Equation (1). However we can easily define its adjoint operator as

$$f_m = \frac{1}{\sqrt{M}} \sum_{n=1}^{N} F_n e^{2\pi\imath(x_m \cdot k_n)}. \tag{2}$$

Normalization by $\sqrt{M}$ is motivated by the case of $M = N$ and equispaced sampling. In this case, the Fourier transform is the well-known Discrete Fourier Transform (DFT) and the adjoint is an inverse DFT. Our work is motivated by the Fourier inverse problems that arise in MRI reconstruction. Computation of the adjoint is required by some algorithms for the solution of these problems, for example the bi-Conjugate Gradient (BiCG) method [14] for solving non-symmetric linear systems. Historically in MRI literature, the term "Gridding" has been used to describe the reconstruction of images from non-equispaced Fourier-domain samples (i.e. the adjoint Fourier transform), and the terms "re-Gridding" or "inverse-Gridding" have been used to describe the computation of non-equispaced Fourier coefficients.

When both the spatial samples $\{x_m\}$ and the frequency samples $\{k_n\}$ lie at evenly-spaced positions, the well-known Fast Fourier Transform (FFT) algorithm can evaluate the Fourier transform very efficiently. In many applications, including the non-Cartesian MRI reconstructions that motivate our work, at least one of the the sample-point sets $\{x_m\}$ or $\{k_n\}$ is not equispaced. The discrete Fourier transform in this context is commonly known as a Non-uniform DFT (NDFT), and no log-linear algorithm exists for its exact evaluation. Computing the NDFT directly requires $O(MN)$ arithmetic operations, and for many applications is unfeasibly expensive.

Instead, these applications rely on a class of fast approximation algorithms referred to as Non-Uniform Fast Fourier Transforms (nuFFT). Most nuFFT algorithms resample the data onto an equispaced grid, enabling the use of the FFT to compute the Fourier transform. If both sets $\{x_m\}$ and $\{k_n\}$ are non-equispaced, then the output of the FFT must be re-sampled as well. Resampling is defined as the convolution of $f(k)$ with some real-valued interpolation kernel $g(k)$:

$$(f * g)(k) = \int_{||k-\kappa||_2 < \frac{1}{2}W} f(\kappa) g(k-\kappa) d\kappa. \qquad (3)$$

For computational feasibility, the convolution kernel is defined to be of finite width W – i.e. $g(k) = 0$ if $||k||_2 \geq \frac{1}{2}W$. The window function is usually chosen to be spherically symmetric $g(k) = g_s(||k||_2)$ or separable $g(k) = g_x(k_x) \cdot g_y(k_y) \cdot g_z(k_z)$. The convolution (3) is to be evaluated at an evenly spaced set of locations $\{\tilde{k}_p\}$, while $f(k)$ is only known at the sample locations $\{k_m\}$. Hence, the convolution is evaluated as the summation:

$$(f * g)(\tilde{k}_p) = \sum_{m \in \mathcal{N}_W(\tilde{k}_p)} f_m \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m \qquad (4)$$

where we define the neighbor sets $\mathcal{N}_W(\tilde{k}_p) \triangleq \{m : ||k_m - \tilde{k}_p||_2 < \frac{1}{2}W\}$ so that the summation is evaluated only over the kernel support window. The terms Δk_m are the discrete representation of the differential $d\kappa$ in (3), and are frequently called the density compensation factors (DCFs) in MRI applications. After computing the FFT of the equispaced samples $(f * g)(\tilde{k}_p)$, the desired equispaced samples of the approximate DFT of f are recovered by de-convolving by g. Via the Fourier-convolution identity, the de-convolution is computed as an inexpensive division by the Fourier transform of $g(k)$. This de-convolution step is also known as deapodization or roll-off correction.

The nuFFT achieves the log-linear arithmetic complexity of the FFT at the expense of accuracy. The resampling step can be understood as multiplication of the function $(f * g)(k)$ by a pulse-train function $\sum_j \delta(k - j \cdot \Delta\tilde{k})$, where $\Delta\tilde{k}$ is the spacing of the grid samples $\tilde{k}_p$ and $\delta(k)$ is the Dirac delta. Again via the Fourier-convolution identity, this is equivalent to a convolution in the Fourier domain with a pulse train with spacing $\frac{1}{\Delta\tilde{k}}$. This operation results in aliasing, as the n^{th} sample produced by the FFT is the sum $\sum_{j=1}^{\infty}(F \cdot G)(k_n - j \cdot \frac{1}{\Delta\tilde{k}})$, where G is the Fourier transform of the truncated resampling kernel $g(k)$. This aliasing is the source of NDFT approximation error. To reduce the effect of aliasing and increase NDFT approximation accuracy, the grid spacing must be finer than the Nyquist-rate of the signal f. If the FFT is computed at size $\tilde{N} > N$, then aliasing of the side-lobes of $F \cdot G$ achieves its maximum intensity in the margins rather than in the central N samples of the reconstructed signal. Thus the accuracy of the nuFFT is determined by a third parameter in addition to the two already mentioned ($g(\cdot)$ and W), usually defined as the grid oversampling ratio $\alpha = \tilde{N}/N$.

All three gridding steps (resampling, FFT, and deconvolution) are linear and can sensibly be written as matrix-vector products. Let $\mathbf{D}$ be an $N \times \tilde{N}$ diagonal matrix containing the deapodization weights, $\mathbb{F}$ be an $\tilde{N} \times \tilde{N}$ DFT matrix, and $\mathbf{\Gamma}$ be an $\tilde{N} \times M$ matrix containing the weights used during re-sampling for each grid/non-equispaced sample pair $\tilde{k}_p, k_m$:

$$\mathbf{\Gamma}_{p,m} = \begin{cases} g(\tilde{k}_p - k_m)\Delta k_m & ||\tilde{k}_p - k_m||_2 < \frac{W}{2} \\ 0 & \text{otherwise} \end{cases} \qquad (5)$$

With $\tilde{f} \in \mathbb{C}^M$ a vector of the non-uniform samples of the signal f and $y \in \mathbb{C}^N$ a vector of the pixels in the approximated DFT of f, the nuFFT is equivalent to computing the matrix-vector product:

$$y = \mathbf{D}\,\mathbb{F}\,\mathbf{\Gamma}\,\tilde{f} \qquad (6)$$

The adjoint nuFFT is equivalent to multiplication by the conjugate transpose, $\mathbf{\Gamma}'\,\mathbb{F}^*\mathbf{D}$.

The deapodization is very inexpensive, consisting of a single multiplication per grid point. The FFT is expensive, but highly efficient library implementations are widely available. Thus much of the difficulty in the efficient implementation and performance optimization of the nuFFT lies in the resampling step, or equivalently of the matrix-vector product $\mathbf{\Gamma}\,\tilde{f}$. Several implementation approaches to the resampling step, demonstrating different levels of precomputation, are described in Section 5.

2.1 Algorithmic Complexity

The following sections will discuss the implementation and performance optimization of the nuFFT. To motivate the discussion, it is useful to discuss the asymptotic runtime of the algorithm. We assume that the input is a non-equispaced sampling of f and that the desired result is an equispaced sampling, so that the resampling must only be performed once. The number of arithmetic operations to be performed depends on the input size M and output size N, but also on the parameters that control NDFT approximation accuracy: the kernel function $g(\cdot)$, the convolution truncation width W, and the grid oversampling ratio α (equivalently, the oversampled grid size $\tilde{N}$).

With d-dimensional coordinates (typically 2-D or 3-D), the resampling convolution performs approximately W^d kernel evaluations and multiplications per input sample,

so the number of arithmetic operations performed by the the nuFFT is

$$C_R M W^d + C_F \tilde{N} \log \tilde{N} + N \quad (7)$$

where C_R and C_F are constant factors associated with the implementation of resampling and the FFT, respectively. The resampling constant C_R depends on the method by which the implementation evaluates $g(\cdot)$, which may require evaluation of transcendental or trigonometric functions. C_R also depends on the dimensionality d, as the formula for the size of the convolution window differs for 2D and 3D. C_R also depends on the amount of precomputation the implementation performs as described in Section 5.

The linear deapodization cost – the third term in (7) – is negligible, and the majority of the nuFFT's runtime is spent in performing the resampling and FFT. However, it is unclear from Equation (7) which of these two operations will dominate the nuFFT's runtime. A well-known property of the nuFFT is that an implementation can effectively trade runtime between the two, and force either to dominate. To achieve a given level of accuracy, one can choose to make the window width W very large. This tends to decrease the amplitude of the sidelobes of G, and consequently a smaller grid size $\tilde{N}$ is tolerable. Alternately, a small window W produces large sidelobes and necessitates a larger grid. Section 3 describes our strategy for identifying the tradeoff between W and α that minimizes the total runtime. This strategy will also allow us to decide several other implementation options for which the asymptotic analysis provides no guidance.

3 Empirical Performance Optimization

Section 2.1 discusses the inherent runtime tradeoff between the resampling convolution and the Fourier transform. However, the asymptotic analysis is unable to suggest a strategy for selecting the parameters that guarantee the best performance. Many possible values of the nuFFT's parameters are able to achieve any given level of NDFT approximation accuracy, but this space of error-equivalent implementations will exhibit very different balances of runtime between resampling and the FFT. The actual runtime an implementation achieves is a complicated function not only of the nuFFT parameters, but also of the microarchitecture of the target platform and the non-uniform sampling pattern, which determines the pattern of memory access during resampling. The execution pipelines and memory systems of modern processors are highly optimized for particular types of instructions and patterns of memory access. Cache behavior in particular is very sensitive to the order in which data is accessed by a program, and cache performance is notoriously difficult to model or predict.

We propose empirical performance measurement as a strategy to resolve the ambiguity in nuFFT parameter selection. This approach has been called *auto-tuning* in other contexts, as it has been successfully applied to a number of other important numerical algorithms [15, 10, 16, 17, 9, 18]. In all these cases, one can enumerate a number of functionally-equivalent implementations of an algorithm but cannot *a priori* determine which will provide optimal performance. Auto-tuning approaches benchmark a number of the alternate implementations in order to make performance optimization decisions. This benchmarking phase is commonly called *planning*, a term first used by the well-known and widely used FFTW library [9]. In cases where the performance of the algorithm is input-dependent, planning may need to be performed on-line. In the case of the nuFFT, auto-tuning must be performed for each pattern of non-uniform sampling. However, the resulting auto-tuned implementation can be reused for multiple signals sampled at the same set of locations.

We design the planning phase for the Gridding nuFFT to take into account the size of the final signal N, the non-equispaced sample locations $\{k_m\}_{m=1}^M$, and the desired level of NDFT accuracy measured via the metric discussed in Section 4.1. Section 6 discusses the motivation of this decision by the application of the nuFFT in MRI reconstruction. However, the FFT and Sparse Matrix libraries on which we build our implementation also have planning phases, thus it is natural that we require a planner as well. Our functionally equivalent implementations differ in their choices of the resampling kernel $g(\cdot)$, the convolution window width W, and the size of the oversampled grid $\tilde{N}$. Additionally, Section 5 will discuss several approaches of implementing the resampling convolution on parallel systems. The optimality of one approach over the others depends on characteristics of the microarchitecture: cost of interthread synchronization, effective memory bandwidth, and floating-point execution rates.

Much of the efficiency of nuFFT algorithms is due to the use of the FFT for computing the Fourier transform. Similarly, Section 5 discusses the resampling can be implemented very efficiently as a matrix-vector multiplication. High-performance auto-tuning libraries are available for computing both the FFTs and matrix-vector products. The nuFFT planner for the nuFFT must invoke the planner for these libraries as well. The FFTW library [9] produces highly efficient FFTs by searching over possible Cooley-Tukey factorizations of the transform size. To avoid an exhaustive search over this space of implmenetations, FFTW relies on a very effective dynamic programming heuristic that avoids benchmarking a particular problem size more than once. When consid-

ering all possible recursive Cooley-Tukey factorizations of a problem size, the FFTW planner will encounter smaller transform sizes many times. Each time the planner encounters a particular transform, it will be performed at possibly different input and output strides. Although memory-system behavior depends on these strides, the dynamic programming heuristic provides close to optimal performance.

Similarly, the Optimized Sparse Kernel Interface (OSKI) [10] library provides a heuristically auto-tuned implementation of sparse matrix operations such as the matrix-vector multiplication that can be used to implement resampling in the nuFFT. Many of the optimizations that OSKI can perform involve modifying the sparse matrix data structure to achieve higher memory system performance during matrix products. For example, general-purpose matrix data structures such as Compressed Sparse Row (CSR) generally store at least one integer index per nonzero entry in the matrix. However, many matrices exhibit patterns of local density, and indexing overhead can be reduced by storing a block of nonzeros in a small dense matrix. Rather than storing a single integer per nonzero, the sparse matrix data structure can store an integer per dense block. Using this more parsimonious data structure reduces memory traffic and increases the length of unit-stride memory accesses during matrix-vector products, potentially improving performance.

4 Parameter Selection

Selection of the three nuFFT algorithmic parameters – interpolation kernel $g(\cdot)$, convolution width W, and grid oversampling factor α – determines the balance between NDFT approximation accuracy and runtime, and the balance of complexity between resampling and the FFT. Per Equation (7), the asymptotic complexity depends on W and α, but only constant factors are affected by the choice of $g(\cdot)$. Moreover most implementations pre-sample the kernel function and estimate its value during resampling by linear interpolation, so the choice of $g(\cdot)$ is of little import for the purposes of performance optimization. Consequently, for the remainder of this work we'll assume that $g(\cdot)$ is the Kaiser-Bessel window:

$$g(k) \quad = \quad \frac{\tilde{N}}{W} I_0(\beta\sqrt{1-(2\tilde{N}k/W)^2}) \qquad (8)$$

Where β is a scalar parameter, $I_0(\cdot)$ is the zero-order modified Bessel function of the first kind, and $\tilde{N}$ is the oversampled grid width. The inverse Fourier transform of Equation (8) to be used during deapodization is computed as:

$$G(x) \quad = \quad \frac{\sin\sqrt{(\pi W x/\tilde{N})^2-\beta^2}}{\sqrt{(\pi W x/\tilde{N})^2-\beta^2}}.$$

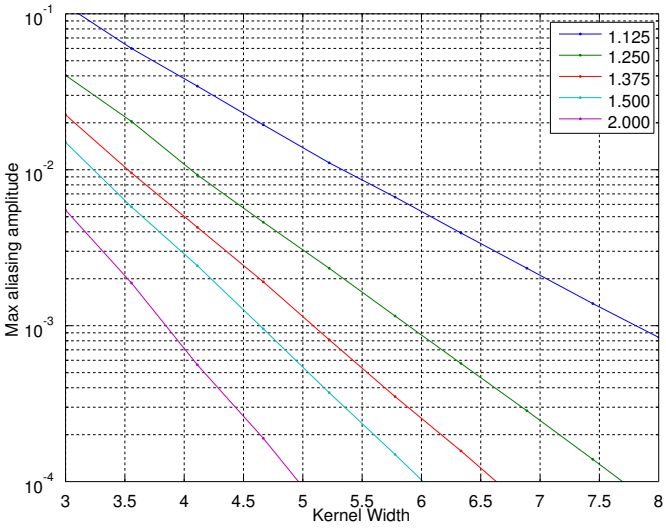

Figure 1: Plot of max aliasing amplitude vs. kernel width for a variety of α. Plots are generated by evaluating equation (9) in 1 spatial dimension for a variety of values of α and W. The infinite sum is truncated to $|p| \leq 4$.

Previous works [2, 4] have determined this gridding function (with appropriate selection of β) to provide near optimal image quality results. In this work, we use the β from Beatty *et al.* [4]: $\beta_{\alpha,W} = \pi\sqrt{\frac{W^2}{\alpha^2}\left(\alpha-\frac{1}{2}\right)^2-0.8}$. This expression is derived by positioning the first zero-crossing of $G(x)$ at $\tilde{N}-N/2$ as a heuristic to control aliasing behavior.

4.1 Aliasing Amplitude

Our auto-tuning approach requires that the desired NDFT approximation accuracy be specified to the planner, so that we can enumerate a set of values for α and W that explore the error-equivalent complexity tradeoff between resampling and the FFT. The most natural metric by which to measure NDFT approximation accuracy is the difference between the output of the nuFFT and the direct computation of the NDFT via (1). However, this metric is expensive to compute and dependent on the signal to be reconstructed. It is highly desirable to evaluate image reconstruction quality as a function only of the selected algorithmic parameters $g(\cdot)$, α, and W, especially since our planning approach does not require the sample values to be known. Also we desire a metric with a closed-form analytic expression which we can more eas-

ily use to constrain the space of parameters our planner must evaluate.

Beatty *et al.* [4] proposed the *aliasing amplitude* metric, which achieves these desired properties. Rather than measuring the reconstruction error for any particular signal, the aliasing amplitude metric estimates the amount of energy this aliasing produces at each pixel in the image:

$$\varepsilon(x_i) \quad = \quad \sqrt{\frac{1}{G(x_i)^2} \sum_{p \neq 0} \left[G(x_i + mp) \right]^2} \qquad (9)$$

In the MRI application that motivates this work, our primary accuracy-concern is the absence of reconstruction artifacts. Thus the maximum aliasing amplitude $\varepsilon^* = \max_{x_i} \varepsilon(x_i)$ is a fitting metric. Figure 1 demonstrates the relationship between this quality metric and the parameters W and α, using the Kaiser-Bessel kernel. Given a desired accuracy, fixing either one of W of α specifies a bound on the value of the other. In our implementation, the corresponding value is computed via binary search over solutions to (9). *E.g.* one can read from Figure 1 the minimum kernel width necessary to achieve a given error tolerance with a given oversampling ratio. For the purposes of planning an nuFFT implementation, the aliasing amplitude metric allows us to easily enumerate an arbitrarily large set of error-equivalent choices for W and α. The planner can evaluate as many (W, α) pairs as are feasible in the time allotted for planning.

5 Resampling Implementation

In this section we discuss the implementation of the convolution summation in Equation (4), or equivalently the multiplication by the matrix $\mathbf{\Gamma}$ defined in Equation (5). We'll review several previously proposed approaches from the MRI literature, and describe them via their relation (explicit or implicit) to the matrix $\mathbf{\Gamma}$. We organize this discussion according to the amount of precomputation performed by each approach. All precomputation approaches perform off-line analysis of the non-equispaced sample locations, and can be performed before the actual sample values are available. If the nuFFT is to be performed for multiple signals that have been sampled at the same set of locations, then the precomputation need only be performed once. The cost of the precomputation is amortized, and its performance benefit multiplied, over the multiple nuFFT executions.

To reduce runtime in many practical applications, the interpolation kernel $g(k)$ is pre-sampled in the range $-\frac{1}{2}W < ||k||_2 < \frac{1}{2}W$ and the values stored in a table. The size of this table is negligible compared to the samples of f. During resampling, values of $g(k)$ are estimated via linear interpolation between adjacent table entries. This linear interpolation requires only a few arithmetic operations, whereas many commonly used interpolation kernels are transcendental and relatively expensive to evaluate.

$\{f_m, k_m\}$: Non-equispaced samples of $f(x)$
Δk_m : Density Compensation Factors
$g(k)$: Resampling kernel
$\{h_p, \tilde{k}_p\}$: Equispaced samples of $(f * g)(k)$
$\tilde{\mathcal{N}}_W(k_m)$: Indices of equispaced samples that are within kernel support window w.r.t. sample location k_m: $\tilde{\mathcal{N}}_W(k_m) \triangleq \{p : ||k_m - \tilde{k}_p||_2 < \frac{1}{2}W\}$

Evaluation at equispaced locations
(1) for $m = 1, \ldots, M$
(2) for $p \in \tilde{\mathcal{N}}_W(k_m)$
(3) $h_p \leftarrow h_p + f_m \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m$

Evaluation at non-equispaced locations
(4) for $m = 1, \ldots, M$
(5) for $p \in \mathcal{N}_W(k_m)$
(6) $f_m \leftarrow f_m + h_p \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m$

Figure 2: Pseudocode of the precomputation-free resampling, which implements the matrix-vector multiplication $h \leftarrow \mathbf{\Gamma} f$ without storage besides the source data f and the output grid h.

5.1 Precomputation-free Resampling

One can directly translate Equation (4) into an algorithm for performing the resampling as a convolution, and pseudocode of this approach appears in Figure 2. The primary benefit of this approach is its lack of additional storage beyond the input and output data. Shared-memory parallel implementations of this approach partition the non-equispaced samples among processing threads. If the resampling is to be evaluated at the non-equispaced locations (line 6 in Figure 2), then this parallelization requires no synchronization among threads. However, when resampling onto the equispaced grid (line 3) it is not known *a priori* which grid locations h_p will be updated by a given sample f_m. If two threads are concurrently computing grid updates for sample locations whose distance is less than the kernel width W, then their grid updates will overlap. The memory systems of modern parallel processors cannot guarantee correct behavior in this case unless the grid updates are protected by mutual exclusion. The grid updates are very frequent, and primitives such as spin-locks and semaphores incur much too high a performance penalty to be used. Instead, low-level processor-specific read-modify-write operations must be used. Several prior works have described this implementation on

GPGPU systems, and it is equally applicable in multi-core CPU systems. In the case of a Cuda [19] implementation, the `AtomicAdd()` library function enables efficient parallelization. For x86 CPU implementations, equivalent functionality is provided by the `cmpxchg` instruction with a `LOCK` prefix inside of an execute-retry loop. Most compilers provide intrinsic functions to simplify use of these instructions.

5.2 Fully-Precomputed Resampling

Another implementation option provided by several previous nuFFT libraries [3, 11] is to perform the re-sampling as a multiplication of the $\mathbf{\Gamma}$ matrix defined in Equation (5) by the vector containing the sample values f_m. The values of the entries of $\mathbf{\Gamma}$ depend only on the locations $\{k_m\}$, of the non-equispaced samples, the density compensation factors $\{\Delta k_m\}$, and the nuFFT algorithmic parameters $g(\cdot)$, W, and α as defined in Section 2. Computing and storing all MN entries of $\mathbf{\Gamma}$ is unnecessary, since the kernel width W is typically much smaller than the size of the equispaced grid and the majority of entries $\mathbf{\Gamma}_{p,m}$ are zero – i.e. $\mathbf{\Gamma}$ is a sparse matrix. Typically in MRI applications W is chosen to be 2-10 times the spacing of grid samples, whereas the grid is 200-500 samples wide. Sparse matrices are used widely in numerical computing applications, and can be stored parsimoniously in data structures that permit efficient, synchronization-free implementation of matrix-vector products.

The potential benefit of the precomputed-matrix implementation over the precomputation-free implementation is two-fold. First, the widespread use of sparse matrices in scientific applications has produced a number of very efficient libraries of matrix-vector operations. While a detailed discussion of the performance optimization of sparse matrix operations is out of the scope of this work, several pertinent features of these libraries deserve our attention. These libraries typically support several storage formats, such as the general-purpose Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) that we use in our implementation. Additionally some libraries include storage formats optimized for particular patterns of nonzero locations, for example explicitly storing some zeros in order to store submatrices densely and reduce indexing overhead. Just as we rely on highly optimized extant implementations of the FFT algorithm, we can rely on highly optimized implementations of matrix-vector products. The existence of these libraries greatly simplifies implementation of the nuFFT, as all low-level performance optimization issues are managed by the FFT and sparse matrix libraries.

Second, sparse matrix-vector multiplication (SpMV) performs substantially fewer arithmetic operations than does the convolution-based implementation of the resampling described in Figure 2. Each nonzero entry $\mathbf{\Gamma}_{p,m}$ corresponds to a pair of samples, (f_m, k_m) from the non-equispaced data and $(h_p, \tilde{k}_p)$ from the equispaced grid, whose distance is less than the resampling kernel radius. A sparse matrix-vector multiplication performs a single multiplication and addition per nonzero entry in the matrix. On the other hand, the precomputation-free implementation must perform that same multiply-accumulate after enumerating the neighbor sets $\mathcal{N}_W(\cdot)$, evaluating the interpolation kernel $g(\cdot)$, and multiplying by the density compensation Δk_m. The number of arithmetic operations required depends on the implementation of $g(\cdot)$, but the commonly used approach is to linearly interpolate samples of a spherically symmetric function $g_s(\cdot)$. This approach requires 15 arithmetic evaluations (9 in computing the distance $||k_m - \tilde{k}_p||_2$ and 6 in the linear interpolation) as well as evaluation of the transcendental square root. An implementation that uses a separable interpolation kernel, rather than a spherically symmetric kernel, would perform slightly fewer arithmetic operations during convolution, but would still benefit from precomputation.

While precomputation of $\mathbf{\Gamma}$ decreases arithmetic operation count, the matrix datatsructure has a large memory footprint. Since each nonzero entry in this data structure must be accessed during resampling, the matrix-vector product incurs a large volume of memory traffic. As has been noted by recent literature on the optimization of sparse matrix operations [15], this memory traffic is the performance-limiting factor of SpMV on modern shared-memory parallel processors. Thus, the performance benefit of precomputation is contingent upon the ability of the processor's memory system to satisfy the memory traffic of SpMV faster than its floating-point pipelines can perform the convolution.

5.3 Partially Precomputed Resampling

Several previous works have presented approaches that perform some pre-analysis in order to more efficiently compute the resampling, but do not precompute the entire matrix $\mathbf{\Gamma}$. We do not include these approaches in the performance evaluation in Section 7, however the empirical search strategy we describe in Section 3 can easily accommodate them. We describe two relevant GPGPU implementations that have attempted to obviate the inter-thread synchronization needed by the precomputation-free implementation described in Section 5.1. A simple way to achieve this goal is to precompute only the locations of the nonzeros in $\mathbf{\Gamma}$, i.e. the pairs (p, m) for which $\mathbf{\Gamma}_{p,m} \neq 0$. The implementation of resampling would need to re-evaluate the values of the nonzeros. While this approach disambiguates grid-update conflicts, it does not achieve any performance gain: the size of the data structure required is almost as large as $\mathbf{\Gamma}$, and this approach

still incurs the runtime cost of evaluating the non-zeros.

The approach presented by Sørensen *et al.* [5] achieves a synchronization-free implementation of the convolution in Figure 2 by precomputing the locations of all the nonzeros in $\mathbf{\Gamma}$, but reducing the size of the data structure with two crucial transformations. First, they partition the grid samples $\tilde{k}_p$ (equivalently, partition the rows of $\mathbf{\Gamma}$) among the processing threads. By taking the union of column indices m in a partition, they find non-disjoint subsets of the non-equispaced samples from which each thread interpolates onto the grid. Second, they sort and run-length-encode the resulting column-index sets to further reduce their memory footprint. This approach is a highly attractive alternative to the precomputation-free convolution on systems that lack the highly efficient read-modify-write instructions mentioned in Section 5.1. This was the case for the early GPGPU system targeted by Sørensen *et al.*.

The "spatial binning" approach of Obeid *et al.* [6] is inspired by algorithms for molecular dynamics problems that compute with spatially-local interactions among points at arbitrary locations in $\mathbb{R}^d$. In MRI applications, the samples are usually stored in readout-order. Obeid *et al.* propose to partition the grid samples into contiguous rectangular regions (bins), and to permute the non-equispaced samples so that all samples within a given bin are stored contiguously. From this re-ordering, which is equivalent to a radix sort, one can infer the interacting grid/sample pairs (i.e. locations of nonzeros in $\mathbf{\Gamma}$).

The approach of Fessler *et al.* [3] achieves some of the arithmetic-operation reduction provided by the sparse matrix precomputation, rather than attempting to reduce inter-thread synchronization costs. Fessler *et al.* describe a particular choice of the resampling kernel $g(\cdot)$ that can be partially precomputed in one of several options, reducing the number of arithmetic evaluations that must be performed during resampling. Some of these options impact NDFT approximation accuracy.

6 Planning and Precomputation in MRI Reconstruction

Our work is primarily motivated by Gridding reconstruction in MRI, where the nuFFT is used to compute the adjoint NDFT defined in Equation (2). Our auto-tuning approach is highly applicable in this context, as the information required by the planner is known far in advance of the values of the samples. Additionally, it may be possible to perform much of the tuning during installation of the reconstruction system. For example, the auto-tuning of the equispaced FFT can be performed independently of the auto-tuning of the sparse matrices. The FFT can potentially be auto-tuned for all possible image matrix sizes to be used during clinical imaging, and the resulting plans saved in "wisdom" files [9]. In many clinical applications it may also be possible to enumerate the set of non-Cartesian trajectories that will be used, since the meaningful FOVs and resolutions are limited by patient anatomy. Sparse matrices can be precomputed for these trajectories and also stored in the file system.

If a pre-existing plan for a trajectory is not available, it may be possible to entirely hide the cost of planning and precomputation by performing it simultaneously with data acquisition. During an MRI scan, samples F_n are acquired in the Fourier domain (k-space) at a set of locations $\{k_n\}$ defined by the Gradient pulse sequence. The locations $\{k_n\}$ are known when the scan is prescribed, but the sample values F_n are not known until after the scan completes. MRI data acquisition is inherently slow, and the interval between prescription and completion allows ample time for precomputation. For example, data acquisition in high-resolution, physiologically gated/triggered 3D imaging can take several tens of minutes. Even in real-time 2D imaging applications, where the acquisition of each signal to be reconstructed can take only a few tens of milliseconds, the same trajectory is used for continuous data acquisition over a much longer period of time.

Any performance benefit provided by planning and precomputation will be multiplied by the number of nuFFT executions to be performed during reconstruction. Most MRI applications use Parallel Imaging, where the k-space samples are acquired simultaneously from multiple receiver channels. Each channel acquires the same set of k-space sample locations, but the signals are modulated by the spatial sensitivity profiles of the receiver coils. The nuFFT must be computed for each channel individually. Additionally, many recently proposed MRI reconstructions use iterative algorithms to solve linear systems that model the data acquisition process. These algorithms typically must compute two nuFFTs (a forward and an adjoint) per iteration for each Parallel Imaging channel. State-of-the-art parallel imaging systems utilize up to 32 channels, and iterative reconstructions can perform 10-100 iterations. Thus even moderate performance improvement can substantially decrease reconstruction runtimes.

7 Performance Results

We demonstrate the effectiveness of our implementation strategy via a number of sampling trajectories representative of non-Cartesian MRI reconstruction workloads. Figures 3, 4, and 5 describe in detail the results of performance tuning for three types of MRI sampling trajectories, and Figures 6, 7, and 8 describe the performance achieved for a wider range of trajectories.

All performance results are collected on our test system, which has dual-socket × six-core Intel Westmere CPUs at 2.67 GHz with 64 GB of system DRAM. GPU performance results are collected on an Nvidia GTX580 GPU with 3 GB of high-speed Graphics DRAM. We are not evaluating hybrid CPU-GPU implementations, and reported runtimes include no PCI-Express transfer overhead. All calculations were performed in single precision. Sparse matrices are represented in Compressed Sparse Row (CSR) format, and matrix-vector multiplications are performed via optimized libraries. Our CPU implementation uses the Optimized Sparse Kernel Interface version 1.01h (OSKI) [10], and our GPU implementation uses the CUSparse library [20] distributed with Cuda version 4.0. OSKI only provides sequential implementations, and we parallelize the matrix-vector multiplication by partitioning the rows of $\mathbf{\Gamma}$ among the 12 CPU cores. For load balance, we choose the partition to approximately equalize the number of non-zeros in each CPU's set. We use the FFTW 3.0 library [9] for computing FFTs on the CPUs, and tune with the `FFTW_MEASURE` flag. The sequential baseline against which we report speedups also uses tunes FFTs with `FFTW_MEASURE`. We use the CUFFT [19] library for the GPU implementation.

Performance Tuning Experiments Figure 3 shows nuFFT performance for a 3D cones [21] trajectory, and has $M = 14{,}123{,}432$ sample points and a maximum readout length of 512. Figure 4 shows performance for a very large radial trajectory [1] with $M = 26{,}379{,}904$ sample points. Figure 5 shows results for a 2D spiral [22] trajectory with $M = 38{,}656$ sample points. All three trajectories are designed for 25.6 cm field of view with 1 mm resolution, isotropic, and the final image matrix size N is 256^d. In all three cases, we set the maximum aliasing amplitude to be 1e-2. These figures show four panels each. The top-left and top-right panels show runtimes of convolution-based resampling (red), matrix-vector resampling (green), and FFT (blue). These runtimes are plotted versus a range of values for the grid oversampling ratio α between 1.2 and 2.0. The top-left panel shows multi-core CPU runtime, while the top-right panel shows GPU runtime. The total nuFFT runtime is the sum of resampling runtime, FFT runtime, and the negligible deapodization runtime. The bottom-left panel shows the memory footprint required by both the sparse matrix (green) and the oversampled grid (blue), also plotted versus α. We use the same CSR matrix data structure in both our CPU and GPU implementations, so the memory footprint is the same in both cases. The bottom-right panel compares the speedup achieved by the four types of implementations (convolution/sparse-matrix on CPU/GPU) over a baseline: a single-threaded convolution-based CPU implementation. All four implementations and baseline use a performance-optimal oversampling ratio. Figure 3 demonstrates several trends that correlate well with our expectations. Smaller values of α result in small grid sizes, but large sparse matrices and long convolution runtimes. Conversely, large values of α increase grid size but decrease resampling runtime. The CPU-parallelized convolution implementation consistently achieves approximately 10× performance improvement over the sequential baseline, while the sparse-matrix precomputation provides a substantial 5× further speedup. Interestingly, the precomputed sparse matrix provides performance comparable to the GPU-based convolution, and matrix precomputation provides a 2× speedup on the GPU. That the precomputation provides a relatively smaller benefit on the GPU verifies our claim that the relative memory bandwidth and instruction throughputs of a processor determine the usefulness of this optimization. GPUs have substantially higher arithmetic throughput than CPUs (relative to the memory bandwidths of the two systems), and thus trading memory traffic for floating-point computation is less beneficial on GPUs. FFT performance is highly sensitive to grid size. Most importantly, it is not monotonic: FFT libraries perform better when the grid size factors into small prime numbers. The anomalously bad performance of the GPU FFT in the second- and third- highest values of α correspond to matrix sizes of 491 (prime) and 502 (almost prime – 2×251). The same trends are visible in Figure 4. However the Radial trajectory's sampling density is much higher in the center of k-space, and consequently the trajectory has almost twice as many samples as the Cones trajectory. The memory footprint of the sparse matrix is correspondingly larger than that of the Cones trajectory, and overflows the GPU's 3GB memory capacity for all values of α that we evaluated. Since the CSR data structure requires an integer to be stored for each grid point, the size of the sparse matrix begins to increase as α grows further. Hence, the sparse-matrix implementation of this radial trajectory is infeasible for our GPUs. Note that even in Figure 3 the sparse matrix implementation is infeasible for small values of α. However, the much larger memory capacity of the CPUs in the system permits the fully-precomputed implementation in most cases. Thus CPU performance for this trajectory is about 15% better than the GPU performance.

Performance for a Range of Trajectories In Figures 6, 7, and 8, we have benchmarked the performance achieved by our nuFFT implementation for a number of 3D Cones trajectories with isotropic FOV varying from 16 cm to 32 cm. These figures show performance results for tuning on the CPUs of the system described above, using all 12 available processor cores. All these trajectories are designed for 1 mm isotropic resolution. Thus, as

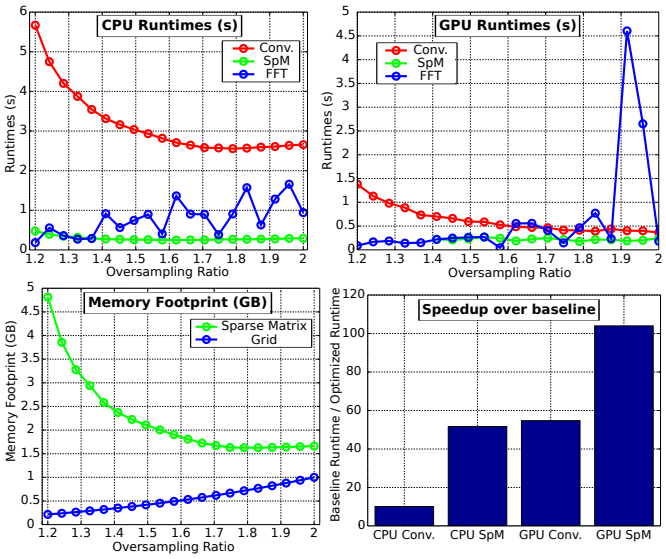

Figure 3: nuFFT auto-tuning performance for the 14M-sample 3D Cones trajectory. The top-left and top-right panels show runtimes vs. grid oversampling ratio α separately for convolution-based resampling (red), matrix-vector resampling (green), and FFT (blue). The top-left panel shows multi-core CPU runtime, while the top-right panel shows GPU runtime. The bottom-left panel shows the memory footprint of the oversamped grid (blue) and the sparse matrix (green). The bottom-right panel shows nuFFT speedup over optimized baseline of four implementations: on CPU with convolution-based resampling, on CPU with matrix-vector resampling, on GPU with convolution, and on GPU with matrix-vector.

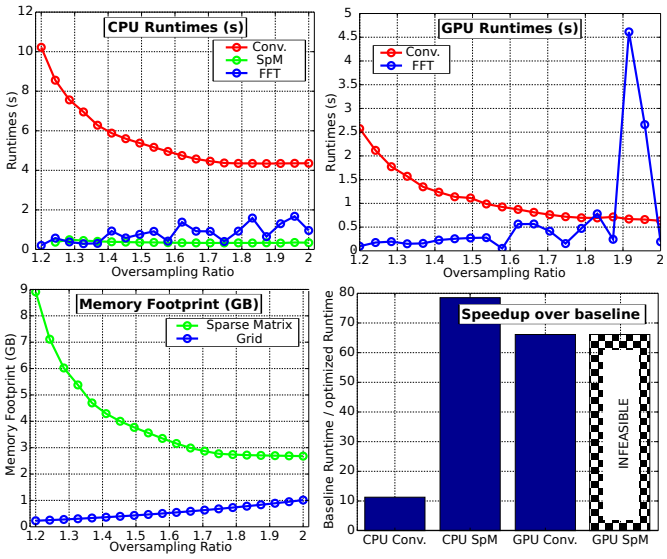

Figure 4: nuFFT auto-tuning performance for the 26M-sample 3D Radial trajectory. For description of the plots in the four panels, see Figure 3.

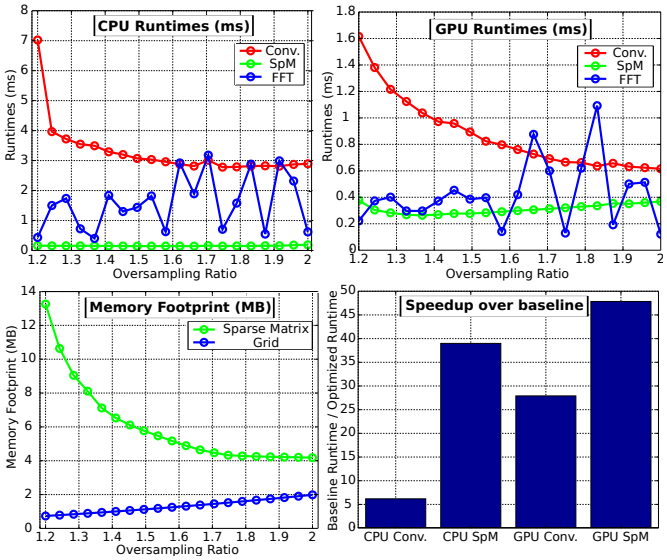

Figure 5: nuFFT performance for the 36K-sample 2D Spiral trajectory. For description of the plots in the four panels, see Figure 3. Note that runtimes are reported in milliseconds and memory footprints reported in megabytes, whereas in Figures 3 and 4 they were reported in seconds and Gigabytes.

the FOV increases, the sizes of the trajectory and final image matrix also increase. For each FOV, we evaluate oversampling ratios in the range of 1.2 to 2.0. Just as above, all nuFFTs are computed at 1e-2 maximum aliasing amplitude. Figure 6 plots three data series describing the performance-optimal precomputation-free implementations chosen by our approach. The blue points display the runtime of the FFT, the red points display the runtime of the resampling convolution, and the magenta points display the total runtime of the nuFFT. Figure 7 plots the runtimes achieved using a precomputed Sparse Matrix to perform the resampling. Similarly, the blue data series represents the runtime of the FFT for the chosen image matrix size, the green series shows the runtime of the corresponding matrix-vector multiplication, and the magenta series shows the total nuFFT runtime. Figure 8 compares the oversampling ratios that achieve optimal performance for the convolution-based and sparse matrix-based nuFFTs shown in Figures 6 and 7. The red points show the oversampling ratios chosen for the convolution-based nuFFT, and the green points show the ratios chosen for the sparse matrix nuFFT.

From Figure 6 it is clear that in a precomputation-free nuFFT implementation, the runtime of the convolution dominates that of the FFT. The performance-optimal oversampling ratio is very high, as shown in Figure 8, because it is beneficial to increase the image matrix size and

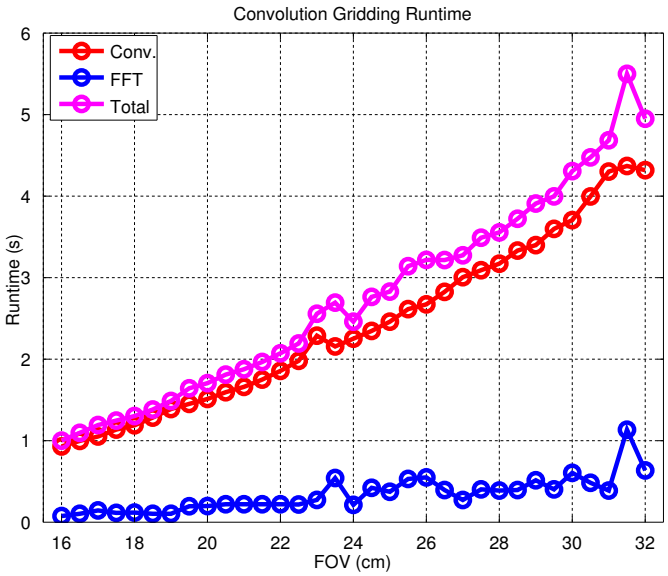

Figure 6: Tuned multi-core CPU runtime for Convolution-based nuFFT, for a range of 3D Cones trajectories with isotropic Field-of-View (FOV) varying from 16 cm to 32 cm, all with isotropic 1 mm spatial resolution.

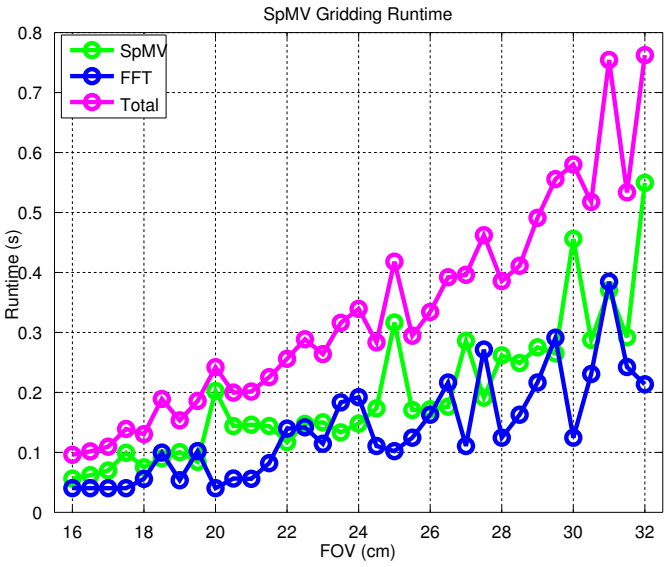

Figure 7: Tuned multi-core CPU runtime for Sparse Matrix-based nuFFT, for a range of 3D Cones trajectories with isotropic Field-of-View (FOV) varying from 16 cm to 32 cm, all with isotropic 1 mm spatial resolution.

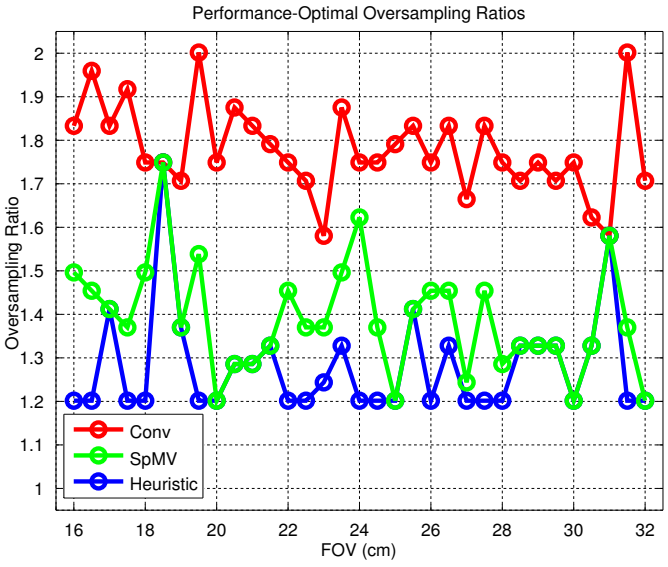

Figure 8: Performance-optimal oversampling ratios for the Convolution-based (red points) and Sparse Matrix-based (green points) multi-core CPU implementations of the nuFFTs shown in Figures 6 and 7, respectively. The blue line shows the alphas selected via a simple heuristic that minimizes FFT runtime, without evaluating sparse matrix-vector runtime.

FFT runtime while decreasing the convolution runtime. Still, the convolution runtime grows much more rapidly than the FFT runtime, and runtimes can be greatly improved by precomputing the sparse matrix. Figure 7 shows that all of the trajectories examined can reap the benefit of sparse matrix precomputation. Figure 8 shows that the tuning phase consistently selects a much smaller grid oversampling ratio for the sparse matrix implementation. Consequently, the runtime is much more evenly balanced between the FFT and the matrix-vector multiplication. Consistent with the results shown above in Figures Figures 3, 4, and 5, a precomputed sparse matrix enables substantially faster implementation of the nuFFT's resampling.

Heuristic Tuning In some applications, the time spent in exhaustive evaluation of a wide range of grid oversampling ratios is prohibitive. Note that evaluating each oversampling ratio requires the computation of the sparse matrix, which requires computing, storing, and sorting all nonzeros by row-index to convert to CSR format. Computing many such matrices can be time-consuming, and it is desirable to have simple heuristics to prune the search space. Since precomputing the sparse matrix shifts much of the computational burden onto the FFT, we propose a simple heuristic that selects the oversampling ratio that minimizes the FFT runtime. This heuristic does not take into consideration the runtime of the corresponding matrix-vector multiplication. Since the FFT can be benchmarked during system installation, this heuristic requires no computation during planning except for precomputation of a single sparse matrix. The blue data series in Figure 8 plots the oversampling ratios chosen by

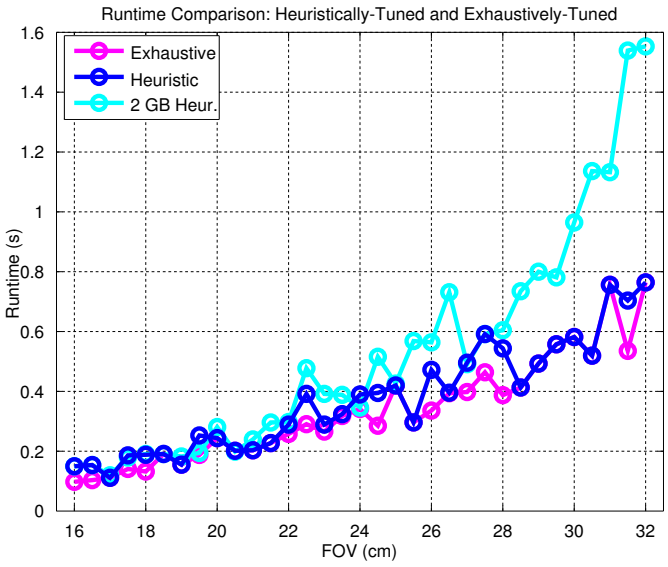

Figure 9: Comparison of multi-core CPU nuFFT runtimes resulting from exhaustive tuning (magenta series) and from the inexpensive heuristic (blue series).The cyan series plots the performance achieved by choosing the fastest FFT runtime subject to a 2 Gigabyte restriction on sparse matrix footprint.

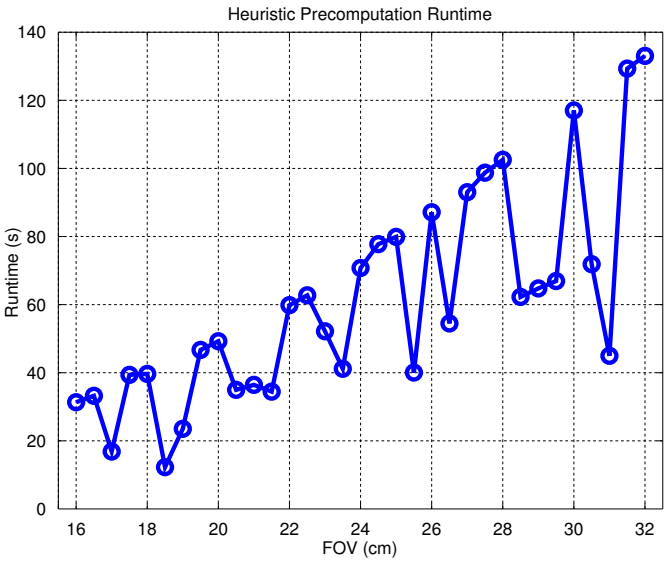

Figure 10: Runtime of sparse matrix precomputation for the matrices chosen by the heuristic described in Figure 9.

this heuristic, and Figure 9 compares the nuFFT runtimes resulting from this heuristic to those resulting from exhaustive evaluation of a range of grid oversampling ratios. In many cases, the heuristic chooses the minimum ratio evaluated – in our case, 1.2. However, due to the pathologically poor performance of the FFT for some transform sizes, this heuristic occasionally chooses a larger grid size. In some cases, the heuristic chooses the same grid size as the exhaustive search strategy. As shown in Figure 9, performance is comparable between the two approaches. Figure 10 shows the runtime of sparse matrix precomputation for the matrices chosen by the heuristic described in Figure 9. Precomputation requires enumerating and sorting all the nonzeros in the sparse matrix. Our current implementation is not optimized, and relies on C++ Standard Template Library (STL) container classes for memory management and sorting routines. Although further performance improvement is desirable for this phase, it can be performed off-line as described in Section 6. The 1–2 minute runtimes shown here can easily be overlapped with data acquisition in many MRI contexts.

Memory-Limited Heuristic Some applications may wish to limit the memory footprint of the Gridding algorithm. In this case, one can use a modified heuristic that chooses the grid oversampling ratio to minimize FFT runtime subject to a specified limit on matrix size. Less entries in the sparse matrix $\mathbf{\Gamma}$ generally require a smaller

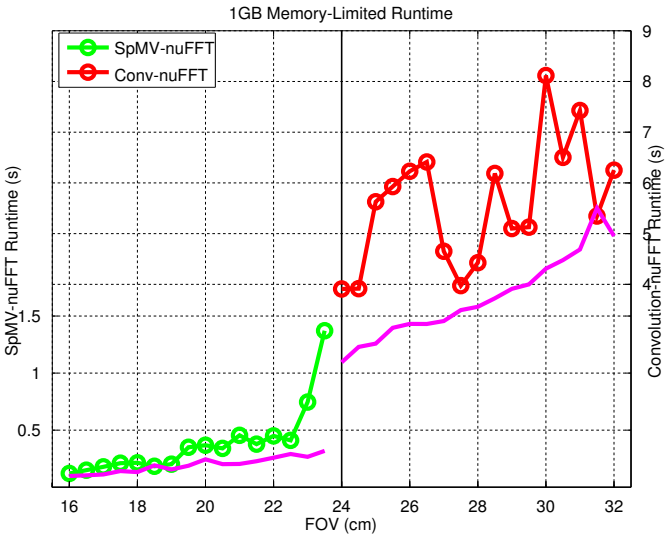

Figure 11: Multi-core CPU runtime of heuristically-tuned nuFFT when sparse matrix precomputation is infeasible for some trajectories. The green plot (measured via the left axis) shows nuFFT runtimes for the trajectories whose matrix can be stored in the 1 GB limit, and the red plot (right axis) shows runtimes for the larger trajectories for which no grid oversampling ratio produces a matrix smaller than 1 GB. The magenta line shows the performance achieved by optimal, exhaustive tuning.

kernel width W. This will force a larger grid oversampling ratio to be used and degrade performance relative for larger trajectories, as demonstrated by the cyan series in Figure 9. Note that the size of the grid and sparse ma-

trix can both be computed in $O(1)$ time, and this heuristic incurs very little cost during planning. However, we note that even for the largest trajectories we evaluate here, this memory-limited heuristic can achieve superior performance to the precomputation-free nuFFT when 2 GB of memory usage is allowed. In applications where memory is more severely restricted, sparse matrix precomputation may be infeasible. In these cases, a memory-limited heuristic can still be used to provide acceptable runtimes. In Figure 11, we demonstrate the runtime achieved for our range of 3D Cones trajectories when matrix memory footprint is limited to 1 gigabyte. For all trajectories with FOV 24 cm or larger, none of the evaluated grid oversampling ratios produces a sparse matrix that satisfies this severe memory limit, and the nuFFT must rely on a convolution-based implementation of resampling. Figure 11 evaluates a heuristic that chooses oversampling ratio in the range 1.5–2.0 to minimize FFT runtime when sparse matrix precomputation is infeasible.

8 Discussion and Conclusion

We have discussed the Gridding non-uniform Fourier transform approximation used in MRI reconstruction of non-Cartesian data, and described the runtime tradeoff between resampling and the FFT. This tradeoff, combined with the several methods of resampling implementation, motivates an *auto-tuning* approach to performance optimization. Following similar approaches used by other numerical libraries, we perform optimization in a *planner* routine that requires only the specification of the non-uniform sampling trajectory and desired level of Fourier transform approximation accuracy. The resulting optimized implementation can be re-used for the Fourier transforms of many signals sampled at the same set of locations. On modern processors, highest performance is always acheived by precomputing a sparse matrix whose matrix-vector multiplication performs the resampling. In this case, most of the arithmetic operations performed during resampling can be computed during the planner routine, although the data structure storing the matrix has a sizeable memory footprint. However, performing the nuFFT via a precomputed sparse-matrix is over 50× faster than an optimized, sequential baseline. A GPU implementation can be over 100× faster than baseline if the GPU's memory capacity permits storage of the sparse matrix. To guarantee optimal performance, matrices corresponding to a range of oversampling ratios must be evaluated. In some applications, it may be undesirable to perform this exhaustive evaluation. We have presented an inexpensive heuristic that achieves near-optimal performance and requires evaluation of only a single matrix.

A number of other implementation strategies and optimizations can readily be incorporated into the auto-tuning strategy. For example, we did not evaluate the performance of the partial-precomputation approaches discussed in Section 5.3. Moreover the implementation of resampling as a sparse matrix permits other optimizations as well, for example batching of multiple resampling instances into a single sparse-matrix/dense-matrix multiplication. Such strategies may improve performance beyond what we report here, and we leave their evaluation to future work. The high degree of flexibility afforded by the empirical-optimization, or auto-tuning, approach additionally future-proofs our library: empirical search will be able to account for microarchitectural changes in future processor generations that radically change the implementation parameters that achieve optimal performance. Thus, we can sensibly call our implementation the *fastest* nuFFT.

9 Acknowledgments

Many thanks to David Sheffield for providing the CPU implementation of the atomic grid update operation, to Holden Wu for providing the 3D Cones trajectory design software, and to Sara McMains for providing feedback on the manuscript.

References

[1] M. Bernstein, K. King, and X. Zhou, *Handbook of MRI pulse sequences*. Elsevier Academic Press, 2004.

[2] J. Jackson, C. Meyer, D. Nishimura, and A. Macovski, "Selection of a convolution function for Fourier inversion using gridding [computerised tomography application]," *Medical Imaging, IEEE Transactions on*, vol. 10, pp. 473 –478, sep 1991.

[3] J. Fessler and B. Sutton, "A min-max approach to the multidimensional nonuniform FFT: application to tomographic image reconstruction," in *Image Processing, 2001. Proceedings. 2001 International Conference on*, vol. 1, pp. 706 –709 vol.1, 2001.

[4] P. Beatty, D. Nishimura, and J. Pauly, "Rapid gridding reconstruction with a minimal oversampling ratio," *IEEE Trans Med Imaging*, pp. 799–808, Jun 2005.

[5] T. Sorensen, T. Schaeffter, K. Noe, and M. Hansen, "Accelerating the nonequispaced fast fourier transform on commodity graphics hardware," *Medical Imaging, IEEE Transactions on*, vol. 27, pp. 538 – 547, april 2008.

[6] N. Obeid, I. Atkinson, K. Thulborn, and W.-M. Hwu, "GPU-accelerated gridding for rapid reconstruction of non-Cartesian MRI," in *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2011.

[7] S. Nam, T. A. Basha, M. Akçakaya, C. Stehning, W. J. Manning, V. Tarokh, and R. Nezafat, "A GPU implementation of compressed sensing reconstruction of 3D radial (kooshball) acquisition for high-resolution cardiac MRI," in *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2011.

[8] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, (Washington, DC, USA), pp. 1–27, IEEE Computer Society, 1998.

[9] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, pp. 216 –231, feb. 2005.

[10] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, (San Francisco, CA, USA), Institute of Physics Publishing, June 2005.

[11] J. Keiner, S. Kunis, and D. Potts, "Using NFFT 3—a software library for various nonequispaced fast fourier transforms," *ACM Transactions on Mathematical Software*, vol. 36, no. 4, pp. Article 19, 1 — 30, 2009.

[12] 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, p. 24727.

[13] H. Eggers, T. Knopp, and D. Potts, "Field inhomogeneity correction based on gridding reconstruction for magnetic resonance imaging," *Medical Imaging, IEEE Transactions on*, vol. 26, pp. 374 –384, march 2007.

[14] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, and L. Restart, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1993.

[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *Supercomputing*, November 2007.

[16] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures," in *Supercomputing*, November 2008.

[17] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *Siam Review*, December 2008.

[18] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[19] Nvidia, "Compute Unified Device Architecture (Cuda)." `http://www.nvidia.com/object/cuda_get.html`. [Online; accessed 25 July, 2011].

[20] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–11, ACM, 2009.

[21] P. T. Gurney, *Magnetic resonance imaging using a 3D cones k-space trajectory*. PhD thesis, Stanford University, 2007.

[22] M. Lustig, S.-J. Kim, and J. M. Pauly, "A fast method for designing time-optimal gradient waveforms for arbitrary k-space trajectories.," *IEEE Transactions on Medical Imaging*, vol. 27, no. 6, pp. 866–873, 2008.