High Performance Non-uniform FFT on Modern x86-based Multi-core Systems

Dhiraj D. Kalamkar*, Joshua D. Trzasko‡, Srinivas Sridharan*, Mikhail Smelyanskiy†, Daehyun Kim†, Armando Manduca‡, Yunhong Shu§, Matt A. Bernstein§, Bharat Kaul* and Pradeep Dubey†
*Parallel Computing Lab, Intel Labs, Bangalore, KA, India
†Parallel Computing Lab, Intel Labs, Santa Clara, CA, USA
‡Department of Physiology and Biomedical Engineering, Mayo Clinic, Rochester, MN, USA
§ Department of Radiology, Mayo Clinic, Rochester, MN, USA

***Abstract*—The Non-Uniform Fast Fourier Transform (NUFFT) is a generalization of FFT to non-equidistant samples. It has many applications which vary from medical imaging to radio astronomy to the numerical solution of partial differential equations. Despite recent advances in speeding up NUFFT on various platforms, its practical applications are still limited, due to its high computational cost, which is significantly dominated by the convolution of a signal between a non-uniform and uniform grids. The computational cost of the NUFFT is particularly detrimental in cases which require fast reconstruction times, such as iterative 3D non-Cartesian MRI reconstruction.**

We propose novel and highly scalable parallel algorithm for performing NUFFT on x86-based multi-core CPUs. The high performance of our algorithm relies on good SIMD utilization and high parallel efficiency. On convolution, we demonstrate on average 90% SIMD efficiency using SSE, as well up to linear scalability using a quad-socket 40-core Intel® Xeon® E7-4870 Processors based system. As a result, on dual socket Intel® Xeon® X5670 based server, our NUFFT implementation is more than 4x faster compared to the best available NUFFT3D implementation, when run on the same hardware. On Intel® Xeon® E5-2670 processor based server, our NUFFT implementation is 1.5X faster than any published NUFFT implementation today. Such speed improvement opens new usages for NUFFT. For example, iterative multichannel reconstruction of a 240x240x240 image could execute in just over 3 minutes, which is on the same order as contemporary non-iterative (and thus less-accurate) 3D NUFFT-based MRI reconstructions.

***Keywords*-Non-uniform FFT; Parallelization; Scalability; Vectorization**

I. Introduction

NUFFT challenge: The Fast Fourier Transform (FFT) [1] is one of the most widely used tools in all of signal processing and numerical analysis. Whereas directly evaluating the discrete Fourier transform (DFT) of a length-N signal requires O(N^2) operations, the FFT uses a divide-and-conquer strategy to indirectly perform this transform with only O($N \log N$) operations. Certain applications, however, such as magnetic resonance imaging (MRI) [2], [3], [4], synthetic aperture radar (SAR) [5], multidimensional wavelet analysis [6], quality control in lumber production [7], digital filter design [8], and computing numerical solutions to elliptical partial differential equations (PDE) [9] may instead require discrete-time Fourier transformation (DTFT) of a uniformly-spaced length-N signal to a length-K signal corresponding to a set of non-uniformly spaced spectral indices. Although the DTFT cannot be exactly evaluated using only O($N \log N$) operations, it fortunately can be very accurately approximated with this complexity by means of the so-called non-uniform FFT (NUFFT) [10], [11], [12], [13], [3], [14].

Typical NUFFT based iterative solvers use two NUFFT operators: Forward NUFFT (**FWD**) and Adjoint NUFFT (**ADJ**). The forward NUFFT operates by applying a scaling function to the signal of interest, transforming the signal onto a uniformly-spaced (i.e., Cartesian) spectral grid by means of an (oversampled) FFT, and then judiciously interpolating that signal onto the desired set of non-uniformly spaced (i.e., non-Cartesian) indices. Conversely, the backward (i.e., adjoint) NUFFT first interpolates a signal associated with a set of non-uniformly spaced spectral indices onto a uniform spectral grid, then transforms that signal via (oversampled) inverse FFT, and finally applies the same scaling function used in the forward operation to the resulting signal. When a fixed width interpolation kernel is employed, the complexity of both the forward and adjoint transform is again only O($N \log N$) albeit with a larger "hidden constant" than its DFT-based analogue. The effective computational cost of the NUFFT is thus largely determined by how quickly this convolution interpolation operation can be performed, and correspondingly by the size and definition of the interpolation kernel [3].

Although advances in speeding up the NUFFT have been made, such as optimized minimum width interpolation kernels and use of lookup tables (LUT) (e.g., [15], [16]), it remains the computational bottleneck for many applications, and particularly those associated with "inverse problems" that employ iterative solvers requiring evaluation of one or more forward and adjoint DTFT/NUFFT's at each iteration. In many areas, such as 3D non-Cartesian MRI reconstruction, less accurate non-iterative reconstructions are often employed in lieu of more accurate iterative methods simply because the latter cannot provide results in a feasible amount of time. Loop-level parallelism can be readily exploited in the forward NUFFT, but parallelization of the adjoint

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.49

operator is non-trivial due to prevalent race conditions arising from multiple samples being "scattered" onto the same uniform grid location [17]. Privatization [18] can be used to circumvent this problem but the associated memory expense of this approach limits its scalability. Advanced hardware implementations, such as on graphics processing units (GPU) [17] or field programmable gate arrays (FPGA) [19], may be restricted to specific spectral sampling patterns [20], require significant code adaptation [17], or be memory limited [21].

Trends in parallel CPUs and algorithms: At the same time, modern central processing units (CPUs) have continued to evolve, as the number of on chip transistors continues to grow. Their compute capacity has increased through progressively higher core counts and, more recently, wider Single Instruction Multiple Data (SIMD) vector units. Current CPUs commonly feature more than 6-8 cores on the same die. SIMD units have increased from 128-bit SSE to 256-bit AVX [22]. Hence it is important to utilize available hardware resources efficiently. To improve this utilization, modern CPUs feature large caches with capacities of tens of Mega Bytes. Furthermore, they support cache coherence, memory consistency as well faster on-die synchronization and inter-core communication, compared to previous generations of CPUs. While these hardware features can simplify the design and improve the performance of parallel algorithms in general, the best performance is obtained in algorithms which avoid global communication. Such "locality-aware" algorithms, which include convolution, take advantage of local communication patterns inherent to many physical problems. This limits synchronization and communication to a small subset of cores on the chip. It also reduces problem's working set, which increases hit rate in smaller but faster first-level caches. Overall, this reduces or eliminates global communication. Correspondingly, the advantage of such algorithms will grow as number of cores, and therefore the cost of global communication will increase in future many-core architectures.

Our contributions: In this paper, we propose a new parallel algorithm for adjoint operator, which is one of the key NUFFT compute kernels. The key features of our algorithm are its ability to maintain even load-balance, local synchronization, locality-aware scheduling, as well as selective privatization and reduction. The last optimization restricts parallel reduction to small dataset regions, which are identified in the preprocessing stage, thus significantly reducing overhead of reduction. Furthermore, we decouple reduction operations from the rest of the computation within parallel task. This exposes larger amount of parallelism in datasets with highly irregular sample distributions. As a result, our algorithm achieves close to linear scalability on up to 40 cores across datasets with widely variable characteristics. Moreover, we demonstrate high SIMD efficiency of our implementation. Specifically, it scales up to 3.8X on SSE and is expected to scale to wider SIMD on future many-core architectures.

The rest of this paper is organized as follows. In Section II we present background and math behind NUFFT. Section III provides details of our implementation. Section IV and V provides details about our experimental setup, results and performance analysis. We discuss previous and related work in Section VI. Finally, we conclude in Section VII.

II. Background

We describe here the mathematics of the 1D NUFFT problem. The generalization to higher dimensions is straightforward, requiring only higher-dimensional FFTs and, for any spectral point, use of an interpolation kernel defined as the Kronecker product of the 1D kernels corresponding to each spectral dimension.

A. Non-Uniform Fourier Transforms

Recall that the DTFT of a length-N discrete signal, f, evaluated at spectral index $\omega \in [0, 2\pi)$ is

$$F(\omega) = \sum_{n=0}^{N-1} f[n] e^{-j\omega n}. \tag{1}$$

For the special case when $\omega = 2\pi m/N$, $m \in [0, N) \subseteq \mathbb{Z}^*$, (1) corresponds to the DFT which can be quickly and exactly evaluated in only O($N \log N$) operations via the FFT algorithm. However, when ω is not confined to a uniform grid, direct evaluation of (1) requires O(N) operations for each computed spectral index.

B. Non-Uniform Fast Fourier Transforms

Consider the M-point ($M \geq N$) oversampled DFT of f, with real-valued and spatially symmetric scaling function, $s > 0$,

$$G[m] = \sum_{n=0}^{N-1} s[n] f[n] e^{-j2\pi \frac{nm}{M}}, \tag{2}$$

whose adjoint and inverse are readily defined, and all of whom can be rapidly computed via the FFT. Using these constructions, note that, for $s[n] = 1$, (1) can be redefined as

$$F(\omega) = \sum_{m=0}^{M-1} G[m] I\left(\omega - \frac{2\pi m}{M}\right), \tag{3}$$

where I is a Dirichlet-like kernel (phase shifted) [3]. Thus, for any ω, $F(\omega)$ could be determined by interpolating from the Cartesian signal albeit with O(MK) operations. For an FFT-based approximator of the DTFT to be practical, interpolating from the uniform onto the non-uniform grid should be O(K). At the same time, this reduced interpolator should still permit a highly accurate approximation of the DTFT. Correspondingly, the FFT needs to be oversampled (i.e., $M \geq N$) so as to reduce the distance between a non-Cartesian spectral index and its nearest Cartesian neighbor.

Consider a compact, radius $W << M$, interpolator, $\hat{I}$. Replacing the "ideal" interpolator by $\hat{I}$ resorts (3) to

$$F(\omega) \approx \sum_{m \in \eta(\omega)} G[m \bmod M] \hat{I}\left(\omega - \frac{2\pi m}{M}\right), \quad (4)$$

where $\eta(\omega) = \left\{m \mid m \in \mathbb{Z}^* \wedge |\frac{M}{2\pi} w - m| \leq W\right\}$. Although there exist a wide variety of potential models for $\hat{I}$ (e.g., Gaussian [14], min-max estimated [3]), in practice and in this work the Kaiser-Bessel kernel is often employed since it is easily formed and can be parameterized to yield high performance even when M is only slightly larger than N (e.g., $M/N = 1.25$) [16], [3]. As spectral convolution by a compact kernel manifests signal domain apodization, the scaling function, s, is typically defined as the point-wise inverse of the inverse DFT of a delta function located at the spectral origin interpolated onto the uniform grid via $\hat{I}$. The scaling function thus precompensates for signal-domain apodization that results from spectral convolution, and is correspondingly often called "rolloff" correction. In summary, computing the forward NUFFT of a signal consists of:

1) point-wise scaling of f by s
2) execution of a M-point FFT
3) convolution interpolation onto the set of ω's

Step 3, which corresponds to (4), is typically realized as a gather operation and thus one that is readily parallelizable via standard loop partitioning.

Computing the adjoint NUFFT of a signal essentially constitutes a "reverse" of the operations of the forward NUFFT, namely:

1) convolution interpolation from the set of ω's
2) execution of an inverse M-point FFT
3) point-wise scaling of f by s and M

Step 1, the functional adjoint of (4), is typically realized in ensemble as

$$H \approx \sum_{\omega \in \Omega} F(\omega) \sum_{m \in \eta(\omega)} \hat{I}\left(\omega - \frac{2\pi m}{M}\right) \delta_{m \bmod M}, \quad (5)$$

where δ_* is the Kronecker delta function. Note that (5) comprises a scatter operation that is highly susceptible to race conditions arising from multiple ω's simultaneously scattering onto the same uniform grid location, n. Although thread privatization can circumvent this problem [18], when applied generically an increase in memory usage proportional to the number of threads is required, and is thus impractical for massive parallelization of large numerical problems.

In certain applications such as MRI, it may be necessary to shift the origin of the image and/or spectrum (akin to the `fftshift` command in Matlab ©) before and/or after any FFT calls. This linear-time operation can be achieved in the conjugate transform domain simply by modulating the transformed signal by a series of ± 1's, a process commonly referred to as "chopping".

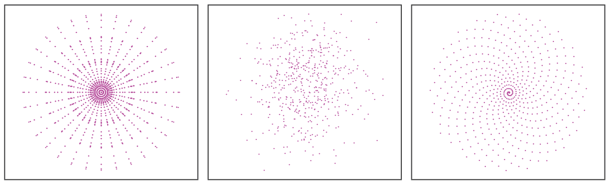
Figure 1. Three Types of Datasets: Radial, Random and Spiral

C. Example Spectral Sampling Strategies

One challenge faced by advanced parallel implementations of the NUFFT is that their performance may vary markedly for different spectral sampling distributions. At one extreme, an NUFFT implementation may be applicable to only a single sampling strategy. In this work, our goal is to provide improved performance for any potential sampling distribution. While we certainly cannot test every possible distribution (as there are an infinite number of possibilities), we have attempted to identify three most common distributions, as shown in Figure 1, that are representative of both those that can arise in the different specialty areas discussed in Section I, and contain different properties that may pose challenges to an advanced NUFFT implementation.

The first identified distribution is "radial", which consists of equispaced sampling along straight-line projections through the spectral origin. Typically, these projections are distributed equiangularly throughout the spectrum. This sampling distribution is regularly encountered in real-world applications, including in parallel-beam tomographic applications (implicit to the Radon transform), magnetic resonance imaging (e.g., the VIPR trajectory [23]), and multidimensional wavelet analysis [6].

The second identified distribution is "random", and particularly according to a variable density Gaussian distribution concentrated at the spectral origin. Random sampling is of growing interest in Compressive Sensing [24], [25], [26], [27] applications, which aim to accurately recover signals sampled at well below the Nyquist rate by exploiting certain information theoretic properties about them.

The third identified distribution is "stack-of-spirals", which is a hybrid sampling strategy that samples uniformly along one dimension (although not necessarily on the Cartesian grid) and along one or more Archimedian spirals in the transverse plane. This sampling model has been previously employed for rapid cardiac MRI [28]. For the sake of exposition, we employed a single, long spiral in each transverse plane, noting that in practice a series of interleaved spirals would likely be used to minimize off-resonance effects.

In many applications, batched or interleaved sampling is performed, where during each of S interleaves, K samples are acquired. For example, in MRI, each interleave may correspond to a single readout (i.e., TR) of the pulse sequence. Correspondingly, the total number of samples are simply SK. As sequential samples within a single interleave often

```
// Part 1: Common to FWD and ADJ
// $N2$: width of Cartesian Grid
// $wx[p]$: x co-ordinate of sample
// Form $x$ interpolation kernel
$kx = wx[p]$
$x1 = ceil(kx - W)$
$x2 = floor(kx + W)$
$lx = x2 - x1 + 1$
for $i = 0$ to $lx - 1$ do
    $nx = x1 + i$
    $kx[i] = mod(nx, N2)$
    $winX[i] = LUT(abs(nx - kx))$
end for
// Form $y$ and $z$ interpolation kernels
...
```

```
// Part 2a: Only in FWD Convolution
// Perform separable convolution
// $raw[p]$: Sample value
// $f[x, y, z]$: Cartesian Grid point
$raw[p] = 0$
for $x = 0$ to $lx - 1$ do
    for $y = 0$ to $ly - 1$ do
        for $z = 0$ to $lz - 1$ do
            $raw[p]$ += $f[kx[x], ky[y], kz[z]] * winX[x] * winY[y] * winZ[z]$
        end for
    end for
end for
```

```
// Part 2b: Only in ADJ Convolution
// Perform separable convolution
// $raw[p]$: Sample value
// $f[x, y, z]$: Cartesian Grid point

for $x = 0$ to $lx - 1$ do
    for $y = 0$ to $ly - 1$ do
        for $z = 0$ to $lz - 1$ do
            $f[kx[x], ky[y], kz[z]]$ += $raw[p] * winX[x] * winY[y] * winZ[z]$
        end for
    end for
end for
```

Figure 2. Convolution Code

exhibit some degree of spectral locality, data is often retained in its original $S \times K$ format to facilitate later preprocessing for NUFFT acceleration. Also, for a d dimensional space, if N^d is the size of reconstructed image, typically there is a relationship $K \times S = N^d \times SR$ where SR is the sampling rate.

III. Implementation

A. Base Algorithm Description

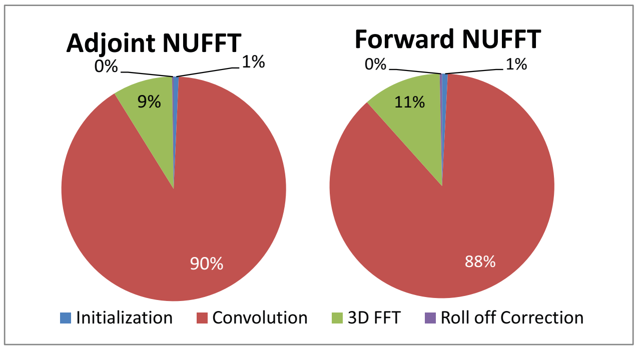

Figure 3. Typical Execution Time Breakdown for Scalar Sequential Code ($N = 256, K = 512, S = 24649, SR = 0.75, W = 4$) running on Intel® Xeon® E7-4870 Processor

For the two NUFFT operators, Figure 3 shows the four sub-kernels and their execution time breakdown for a scalar sequential code. Most of the time is spent inside the two convolution routines, which are the main focus of our optimizations. For 3D FFT, the second most time-consuming kernel, we use highly optimized Intel® MKL FFTW [29] library implementation.

As shown in Fig. 2, each convolution kernel consists of two parts. For a given sample point p, with coordinates $(wx[p], wy[p], wz[p])$, first part computes interpolation kernel coefficients, $winX$, $winY$, and $winZ$, using table LUT look-up. It also computes x, y and z coordinates of the neighbors in convolution window, kx, ky and kz. The second part performs actual convolution where forward convolution accumulates weighted values from Cartesian grid into sample value and adjoint convolution spreads sample value onto Cartesian grid, as shown in Fig. 2.

B. Parallelization of Adjoint Convolution

A parallelization strategy for convolution step is to divide total number of samples between all available threads. Note however that in adjoint convolution two or more samples may update the same grid point. If those two points belong to different threads, some form of mutual exclusion must be provided to guarantee that only a single update is performed at a time. There are several potential solutions to this problem. For example, one can use atomic update instructions and hence to rely on hardware support to guarantee mutual exclusion. Alternatively one can use privatization, wherein each thread maintains and updates private copy of Cartesian grid; a global reduction is performed at the end of convolution which aggregates all private copies into one grid. Both approaches have high overhead, and will not scale to a large number of threads. This overhead may be reduced if we spatially decompose the Cartesian grid into equal size sub-grids and divide them evenly among threads. This will limit simultaneous updates to the boundary of each partition. However, this can result in load imbalance for some datasets due to the fact that each sub-grid may contain different number of samples. This can happen for example in radial or spiral datasets, which are dense at the center and very sparse towards the edges. To address these challenges, we developed a novel parallelization scheme which greatly reduces overhead of reduction while maintaining good load balance. The scheme consists of three main components, which are described next.

1) Data Partitioning: First, we use geometric data partitioning scheme similar to one proposed in [30]. While [30] divides the grid recursively to improve load balance, we partition the grid into variable size sub-grids, as shown in Figure 4 and create a *task* for each partition. Each task processes samples contained in its partition. This has an advantage over recursive partitioning in that cost of

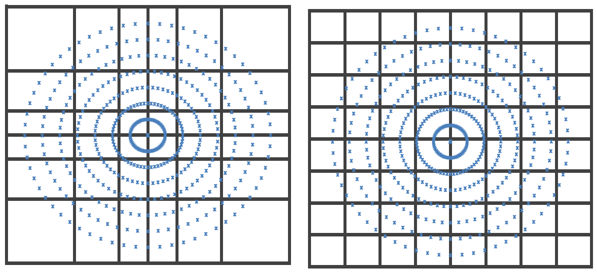

Figure 4. Variable Vs. Fixed Width Partitions

```
// $M$:number of samples
// $P$: desired number of partitions in each dimension
// $W$: convolution Radius
// $N$: width of Cartesian grid
// $hist_d(i)$: returns number of samples with coordinate value
   less than $i$ in dimension $d$
Compute the cumulative histogram
   of samples in each dimension
$avg = M/P$
$minWidth = 2W + 1$
for each dimension $d$ do
    $i = 0$
    $start = 0$
    $partition[d][0] = 0$
    while ($start < N$) do
        $end = start + minWidth$
        while (($hist_d(end) - hist_d(start)) < avg$) do
            $end = end + 1$
            if ($end >= N$) then
                $end = N$
                break
            end if
        end while
        $i = i + 1$
        $partition[d][i] = end$
        $start = end$
    end while
    $numPartitions[d] = i$
end for
```

Figure 5. Algorithm for computing width of partitions in each dimension

partitioning is only $O(M)$ compared to $O(MlogM)$ for recursive partitioning where M is the total number of sample points. Also, it enables Task scheduling without need for a barrier as described in Section III-B2. First we calculate cumulative histogram of samples in each dimension, e.g. for x dimension, $hist_x(i)$ would return number of samples having x coordinate less than i. Figure 5 shows an algorithm that can be used to calculate variable width of partitions in each dimension. If P is the desired number of partitions in each dimension, first we calculate average number of samples for each of these P partitions. P can be derived from number of parallel threads running in the system. Then we start with a minimum allowed width of a partition (which is one more than twice the radius of convolution kernel) and grow it until it contains number of samples more than the average. Then we move on to calculate width of next partition and so on until we reach to the end of the grid. Similar to width of partitions, number of partitions in each dimension need not be the same. For a datasets like radial, the variable size partitions enable us to use minimum size partitions at the center and larger partitions towards edges, thus helping with load balancing.

2) Task Queue Scheduling: To avoid simultaneous updates from multiple threads, one can perform a simple coloring of partitions, such that no two adjacent partitions will have the same color. This will require 8 colors in 3D and partitions of the same color can be computed in parallel. However, the coloring scheme might require a barrier between partitions of different colors, as shown in the previously proposed algorithms [30]. We propose a new algorithm that does not require a global barrier using a task dependency graph (TDG). Tasks get scheduled as soon as they become ready for execution. Our TDG is built as a part of preprocessing step and reused every time NUFFT operator is called. We use a novel scheduling algorithm which keeps TDG simple, as described next.

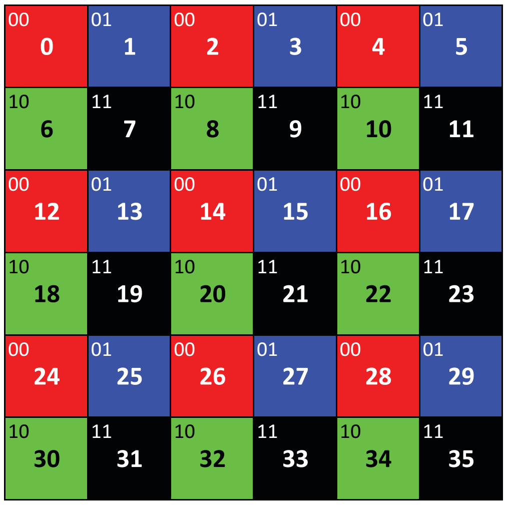

Figure 6. Task Scheduling Algorithm

We define *turn* of a task T as a number obtained by joining least significant bit of the partition number in each dimensions. Thus, for a d dimensional space, there would be total of 2^d turns. We use Gray Code [31] order for scheduling tasks with different turns e.g. ordering for 2-bit Gray Codes is $00, 01, 11, 10$. And for 3-bits it is $000, 001, 011, 010, 110, 111, 101, 100$. Initially, all the tasks with turn 00 are scheduled. For any task T with non-zero turn, task T can be scheduled only if the adjacent tasks of T with previous Gray Code value, as defined by the Gray Code ordering, have finished. This imposes implicit order on tasks and only requires 2 edges in each of forward and backward directions to be stored for each task to build TDG, thus making space requirements of TDG small. For example, as shown in Figure 6, the task 7 with turn value 11 depends on its two adjacent tasks 1 & 13 with turn 01. Thus, a task with turn value 11 cannot be scheduled until the two adjacent tasks with turn 01 have finished. However, tasks 7 and 9 are independent. Thus, if the two adjacent tasks of task 9 have finished, task 9 is ready to be scheduled, even if the adjacent tasks of 7 are still running. It is this property of TDG that gives the maximum concurrency and eliminates the need for global barrier. Once task 9 finishes, it only needs to check the successor tasks 8 & 10 with turn value 10 if they are ready to be scheduled, i.e. if there other adjacent task has finished or not e.g. for task 8, if task 7 is already finished then task 8 is ready for scheduling and will be enqueued, otherwise when task 7 would finish its execution, it would enqueue task 8.

3) Use of Priority Queue: We use a queue to hold all the tasks that are ready for execution. As soon as a thread becomes free, it dequeues a new task from the queue. This avoids parallel updates, as two ready tasks can never update the same grid point. In addition, it helps to maintain load balance. However, occasionally a task with large number of samples gets enqueued much later in the execution due to its dependence chain. Meanwhile, many smaller tasks get enqueued in front of large task. Hence, by the time large task moves to the front of the queue and starts executing, most of the smaller tasks have finished execution and soon queue becomes empty. Thus, except the thread executing larger task, all other threads become idle waiting for the large task to finish. This results in high load imbalance. We address this situation by using a priority queue instead of normal queue. Priority queues ensures that ready task with the largest number of samples is always at the front of the queue ahead of tasks with smaller number of samples. As a result, larger tasks get scheduled for execution before smaller tasks. This significantly improves load balance, especially with large number of threads.

4) Selective Privatization with Reduction: While the priority queue provides good load balancing, to maintain mutual exclusion it needs to execute adjacent tasks sequentially. If input samples have a very high density in small region compared to rest of the grid (similar to center region shown in Fig. 4), the overall speedup is limited by the sequential execution of tasks in the dense region. One way to avoid this situation is by having smaller width of partitions in the dense region. However, since minimum width of a partition is limited by size of W, beyond a certain limit, we can not shrink a partition and therefore, can not reduce number of samples in such tasks. The problem is more likely to happen for smaller sizes of N and for a system with large number of cores. To address this problem, we use selective privatization of tasks. A task which is too dense and can significantly increase the length of the critical path is privatized. Such task store all sample updates into a Private temporary buffer, which is latter merged into the global grid. This is significantly faster than privatizing the entire grid. We use a simple criteria to determine which tasks need privatization. If number of samples in a task exceeds certain threshold we privatize it. For M samples, P processors and d dimensions, threshold is calculated using Eq. 6.

$$Threshold = \frac{M}{P \times 2^{d+1}} \tag{6}$$

The reason behind this is as follows. We calculate average number of samples to be processed per thread. If a group of tasks that execute serially, have more samples than the average number of samples per thread, we expect that group to execute longer than others. In any group of adjacent tasks, there are minimum of 2^d tasks that need to execute serially (it is the number of *turns* in our task queue scheduling algorithm). Thus, the threshold is the ration between the average number of samples per thread to the minimum number of tasks that execute serially. We further divide it by 2 to add additional tolerance for handling delayed queuing.

As a part of preprocessing we identify which tasks need to be privatized. Once a task is marked for privatization, it allocates a private copy of that portion of the grid that this task needs to update. As a result, such task gets immediately executed, bypassing the queue, as soon as a free thread becomes available. When task finishes execution it gets enqueued, similar to other (non-private) tasks, to perform the reduction operation. Decoupling reduction from convolution operation allows more expensive convolution operation to proceed early. This reduces the length of the critical path and improves overall scalability. This effectively eliminates this task from the critical path.

C. Using SIMD for Convolution

Modern parallel architectures enjoy high fraction of their peak performance due to SIMD support. As SIMD width continues to increase, parallel algorithms should exploit SIMD effectively, or large fraction of the system will be underutilized.

A straightforward way to exploit SIMD in convolution is assign one sample to a SIMD lane. This however, has several drawbacks. First, this requires addressing the problem of simultaneous updates, described earlier, when two samples which modify the same grid point are processed in the same SIMD packet. Second, and most important, is the problem of irregular data accesses, where two sample points require accessing data in different cache line, called gather operation. Modern CPU offer no hardware support for gather. Even if such support is available, as in GPU architectures [32], the performance will still be low due to large amount of coalescing required.

To address this problem, we propose hybrid SIMD-friendly implementation. Fig. 7 shows the break down of execution time between Part 1 and Part 2 of convolution (see Figure 2). We see that Part 1, constitutes a small fraction of execution time, compared to Part 2, especially for larger

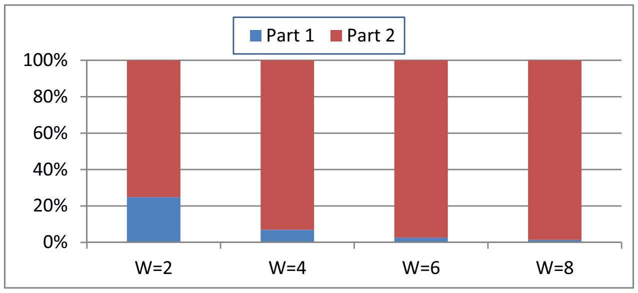

Figure 7. Relative Execution Time for Kernel values and Coordinates Computation (Part 1) and Interpolation (Part 2) of Convolution

W. In the Part 1 where grid coordinates and interpolation kernel values are computed for a given sample point, there is very little or no scope for using SIMD within a single sample processing. So we process one sample per SIMD lane/channel. In the second part, except for samples near eages, the inner most loop performs updates to consecutive grid cells, and hence maps well to SIMD. As a result, our hybrid implementation, uses SIMD across points for low overhead Part 1 and SIMD within a point for high overhead Part 2. Note for small W, amount of parallelism within inner loop over z is limited. In this case, we exploit SIMD across several y iterations to keep SIMD utilization high. While this require additional loads to access non-contiguous data across several y iterations, and thus reduces SIMD benefits, the number of such accesses and hence the loss in SIMD efficiency is much less, compared to across point approach.

D. *Samples Re-ordering for Better Cache Re-use*

In order to improve temporal and spatial locality of accesses to both samples as well as grid points, we re-order sample points. To do that, we use simple scan-line order with one level of tiling. While there exist more complicated reordering algorithms, such as space-filling curve [33], our simple reordering was able to reduce cache miss latency to 25% in the worst case of radial dataset, where samples are sparsely distributed further away from the center and amount of reuse is therefore limited.

IV. Experimental Setup

A. *Dataset types and sizes*

To evaluate performance of our NUFFT implementation, we used three different types of datasets, Radial, Random, and Spiral. These datasets, which correspond to three spectral sampling strategies, described in Section II-C, were identified both for their practical utility and breadth of characteristics. Transform absolute and relative dimensions were analogously selected to be representative of the domain that is currently considered challenging in practice. Table I shows values of dimension size of a 3D image (**N**), number of interleaves (**S**), samples per interleave (**K**) and sampling rate (**SR**) used to generate different sample datasets. Since

Table I
Dataset parameters used to generate 3 types of datasests

Dataset	N	K	S	SR
1	128	256	4096	0.5
2	256	512	24576	0.75
3	256	512	32768	1
4	256	512	40960	1.25
5	320	640	12800	0.25

combination of different types of datasets with different dataset parameter is too large, we use subset of datasets for each of our evaluations. We evaluated our implementation for four different convolution radii: $W = 2, W = 4, W = 6$ and $W = 8$. Unless otherwise stated, we use datasets with $N = 256, SR = 0.75$ and $W = 4$ for most of our experiments. The kernel used is the Kaiser-Bessel function and oversampling (α) is 2.

B. *Machine Configuration*

Our experimental testbed consists of a quad-socket Intel® Xeon® E7-4870 processor based server (henceforth called WSM40C). This is an x86-based multi-core architecture, which provides four sockets of ten cores each. Each core is running at 2.4 GHz. The architecture features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading. In addition to scalar units, it has 4-wide integer and single precision floating point SIMD units that support a wide range of SIMD instructions called SSE4 [34]. In a single cycle, it can issue a 4-wide floating-point multiply and add to two different pipelines. Compared to architectures which only have fused multiply-add unit, such as [32], it can achieve full hardware utilization, even in case when multiply and add can not be fused. Each core is backed by a 64 KiB L1 and a 256 KiB L2 cache, and all 10 cores share a 30 MiB last level L3 cache. Together, the forty cores can deliver a peak performance of 768 Gflop/s of single-precision arithmetic. The system has 256 GiB 1067 MHz DDR3 memory.

V. Results & Analysis

In this section we evaluate the performance of our implementation. We report performance of the entire NUFFT, as well as its main components, in particular convolution algorithms. Furthermore, we adopted a highly optimized FFT implementation from Intel's Math Kernel Library (*MKL*) 10.3.

First we show overall performance of our implementation on WSM40C. Then we show multi-core scaling of our algorithm with respect to optimized single core performance across different datasets, while varying image and convolution kernel sizes. Finally, we quantify the effect of each individual optimization.

A. *Overall Performance*

Table II shows execution time, averaged across all datasets, for our most optimized version (row 2) running on

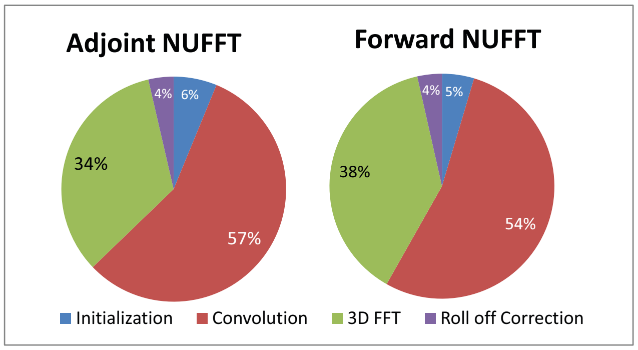

Figure 8. Typical Execution Time Breakdown for Optimized Parallel Code ($N = 256, K = 512, S = 24649, SR = 0.75, W = 4$) running on Intel® Xeon® E7-4870 Processor

all 40 cores, compared with sequential baseline code running on a single core. Note that FFT code is always optimized on any number of cores, since it is a library routine. As row three shows, our optimizations improved convolution performance by almost 150X. 3D FFT scales 28X on 40 cores. The entire NUFFT improved 93X.

Table II
OPTIMIZED PERFORMANCE COMPARED TO SCALAR SEQUENTIAL BASELINE FOR $W = 4$, $N = 256$ AND $SR = 0.75$

	Convolution	3D FFT	NUFFT
Baseline (sec)	71.2	5.9	78.0
Most Optimized (sec)	0.5	0.2	0.9
Speedup	147.5x	28.3x	92.8x

Figure 8 shows execution time breakdown after applying all of optimizations. Compared to Figure 3, we see that performance gap between FFT and Convolution has significantly reduced.

Next in Table III, we show performance of Adjoint and Forward convolution in Million samples processed per second (**SPS**) for different sizes of W. Since Adjoint convolution involves update to Cartesian grid its performance is always slightly less than performance of Forward convolution. We found that SPS are similar (within 10%) for various image sizes (N) and different sampling rates (SR). SPSs are also similar for different datasets when W is large. This makes sense, as convolution does $O(W^3)$ amount of work per pixel, which dominates, when W is large. On the other hand for smaller W, convolution performance is dataset dependent. For example, for $W = 2$, Radial achieves 145 SPS, while Spiral achieves 222 SPS: 1.5X faster. This is due to the fact that irregular datasets, such as Radial, suffer from more cache misses, compared to more regular datasets, such as Spiral, which lowers the performance. Assuming ˜$3 \times (2W)^3$ Flops per sample (and ignoring address computation overhead), our implementation achieves 0.7 to 3 $GFlops/s/core$ with an average of 2 $GFlops/s/core$ across all datasets and different values of W.

Table III
PERFORMANCE OF ADJOINT AND FORWARD CONVOLUTION IN MILLION SAMPLES CONVOLVED PER SECOND

	W=2		W=4		W=6		W=8	
	ADJ	FWD	ADJ	FWD	ADJ	FWD	ADJ	FWD
Radial	145.1	190.7	48.2	60.5	14.1	18.3	6.6	10.2
Random	169.3	194.0	54.2	60.9	14.7	19.3	6.8	10.3
Spiral	222.6	236.7	58.1	64.8	14.7	19.6	7.2	9.4

B. Scalability versus Problem Parameters

In this section we evaluate robustness of our algorithm by varying problem parameters. We report scalability with respect to the performance of optimized single core implementation that uses SMT. To save space, we only show scalability for 10, 20 and 40 Cores. Figure 10 shows performance scaling with select values of N and W. For most of other combinations of N and W as well as for different SR, we get speedup between 30x and 35x for Adjoint convolution and between 35x and 40x for forward convolution with average of more than **35x on 40 Cores**. We also observe that for large number of threads and smaller W and N both convolutions scales worse than for larger W and N. For example, for $W = 2$ and $N = 256$, ADJ scales 28X on Spiral dataset, while for $W = 8$ and $N = 256$, ADJ scales 32X on the same dataset. This is mainly due to decreased amount of work done per thread, and therefore higher overhead of task management.

C. Incremental Performance Results

Figure 9 shows performance improvement due to each individual optimization for convolution and the entire NUFFT over corresponding baseline implementation ('Base'). The results are averaged over all three datasets. Since we use library implementation of 3D FFT, its performance does not change with our individual optimizations. However, we show how performance of 3D FFT, over increasing number of cores, affects performance of entire NUFFT. Our first optimization, called 'Reorder', divides samples into tasks and reorders them withing each task, as described in Section III-D. On average this optimization results 7% improvement over Base.

Our next optimization, 'SIMD', vectorizes convolution code, as explained in Section III-C. On average SIMD

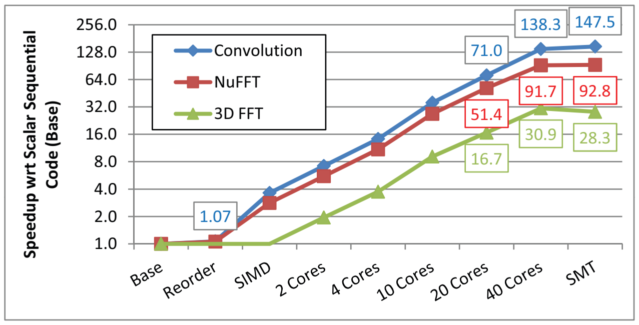

Figure 9. Speedup with successive optimizations

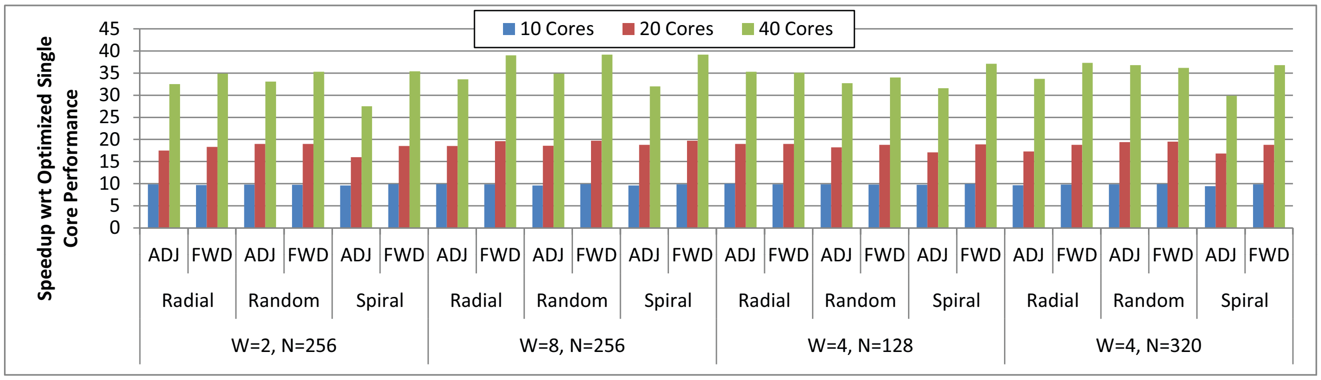

Figure 10. Performance Scaling Across Different W and N

improves performance by 3.4x. Next we parallelize the code using techniques discussed in Section III-B. The graph shows scalability from one to 40 cores. Finally, we get another 7% speedup for convolution by using SMT. The SMT speedup corresponds to overall scalability reported in Table II.

D. Optimizations Analysis

In this section we quantify the performance benefits of various optimizations in our parallel algorithm. Recall that main goal of these optimizations is to improve scalability for large number of cores. We present scalability data for 10, 20 and 40 cores, respectively.

1) Fixed versus Variable Width Partitions: Figure 11 shows scalability of our algorithm with respect to optimized single core performance when we use fixed width partitioning versus a variable width partitioning. For this evaluation, we use radial datasets with three different sizes: $N = 128$, $N = 256$, and $N = 320$. We picked radial datasets because they have very irregular distribution of samples: very high sample density at the center and very sparse density towards the edges. As a result, this dataset is the most challenging to scale to a large number of cores. As figure shows, fixed width partitioning does not scale well beyond 10 cores and for larger N scalability is poor even with 10 cores. This is due to the fact that in order to accommodate non-uniform density, fixed size partitioning creates large number of tasks with a few samples per task. As number of cores increases, this causes higher overhead of task enqueuing and de-queuing and results in poor scalability.

However, with variable width partitioning performance scales very well in all the cases. This is due to the fact that we have fewer tasks, and as a result larger number of samples per task, which reduces task queue overhead.

2) Selective Privatization: In Figure 12, Group A and B shows scalability of our algorithm without and with selective privatization. Again, we use radial datasets for evaluation. As discussed in Section III-B4, we expect smaller datasets to see higher benefits from privatization, Indeed for $N = 128$, even at 10 cores we get 73% additional performance by privatizing selective tasks. With 40 cores, performance advantage of privatization further increases to 3.5x. The advantage of privatization continues to show for larger values of $N = 256$ and $N = 320$, although it is reduced to 2.7x and 1.8x, respectively. Note however that privatization comes with an additional overhead of reducing privatized part into a global Cartesian grid. The high performance advantage of selective privatization stems from the fact that it breaks sequential dependencies among tasks in a few small but computationally expensive dense regions. By selectively breaking these dependencies only within such, we avoid high overhead of full reduction, if entire 3D grid was privatized. However, even small amount of reduction has its overhead. This is the reasons for lower scalability of adjoint convolution compared to forward convolution when using 40 cores (Figure 10). However, it is very obvious from Figure 12, that reduction overhead is much smaller compared to the loss in scalability without selective privatization.

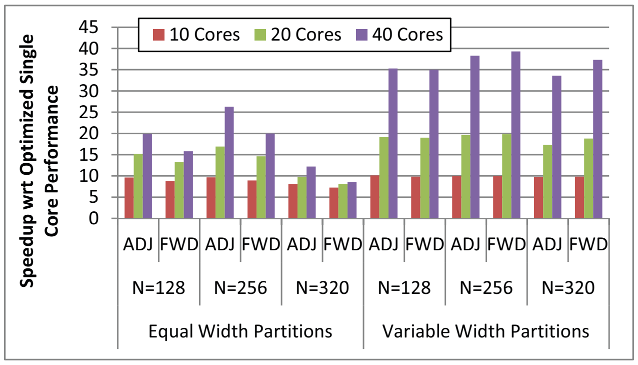

Figure 11. Fixed Vs. Variable Width Partitions

3) Normal versus Priority Queue (PQ): We also evaluate scalability with and without PQ, again using radial dataset. In Figure 12, bars in group B & C show scalability without and with PQ. We see that benefits of PQ increase with the number of cores. For example, while PQ does not offer any significant benefit for 10 cores, on average it improves

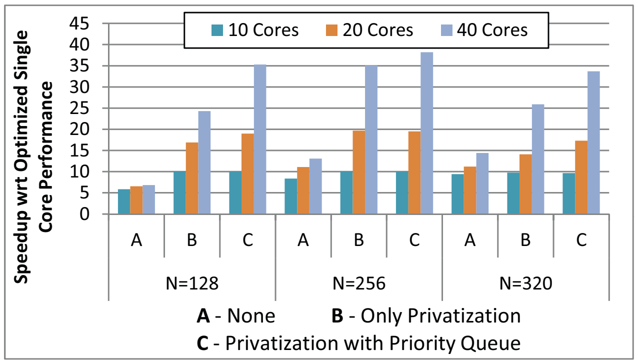

Figure 12. Scalability of algorithm for Adjoint Convolution with and without use of Priority Queue and Selective Privatization

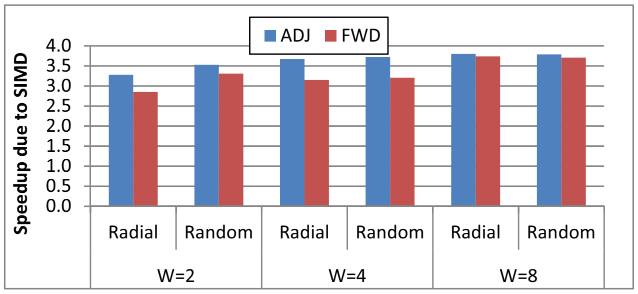

Figure 13. Speedup due to SIMD

scalability by about 10% for 20 cores and 30% for 40 cores. $N = 128$ enjoys as much as 45% improvement when running on 40 cores. In general, when there are large number of parallel threads running, PQ helps improves performance by scheduling longer running tasks at the earliest, thus reducing chances for a longer running task to keep running while all other tasks have finished execution and thus causing a load imbalance.

4) SIMD Performance: Figure 13 shows speedup due to SSE achieved over scalar code on a single core. The results are shown for Radial and Random datasets, $W = 2$, $W = 4$ and $W = 8$. Results for Spiral dataset are similar to Radial. We see that speedup increases with W. For example, in case of forward convolution, speedup increases from 3.2x for $W = 4$ to over 3.8x for $W = 8$. Speedup for $W = 2$ is more modest. Specifically, we achieve 3.2x for adjoint and 2.8x for forward convolution, respectively. As discussed in Section III-C, for smaller W amount of parallelism within inner loop is limited. As a result, we explore SSE across several iterations of the next outer-loop (over y), which introduces extra load instructions. This lowers SSE efficiency due to limited number of vector load ports. Finally, note that forward convolution scales worse with SSE than adjoint. This is due to the fact that, as shown in Figure 2 (middle), forward convolution requires additional reduction into $raw[p]$, which incurs extra overhead.

E. Pre-processing Overhead

As mentioned earlier, in order to get best speedup over multiple iterations, we perform preprocessing of input samples. We divide them into tasks and perform sample re-ordering for better cache utilization. All of this add-up to a one time pre-processing overhead. A comparison of execution time required for both the pre-processing and one iteration of NUFFT (which includes one call to Forward and Adjoint operators) is shown in Figure 14. Due to the fact that pre-processing involves a significant amount of serial code, it does not scale well with cores. As a result ratio of pre-processing execution time to one iteration of NUFFT increases from 0.16 on a single core to 1.67x on 40 cores. However, for an iterative solver, the cost of preprocessing is amortized over 10s to 100s iterations, each of which calls NUFFT. Additionally, due to the fact that the same spectral sampling patterns are often reused in practical applications (e.g., tomography), the preprocessing can be performed offline and reused, in the same manner that the FFTW library [35] reuses “wisdom”.

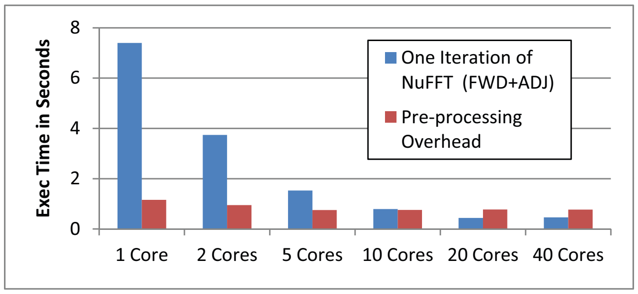

Figure 14. Pre-processing Overhead

VI. Previous Work

Due to the fact that the NUFFT is the computational bottleneck for many numerical problems, there have been a number of previous attempts to accelerate its performance. Early works by Dale et al. [15] and Beatty et al. [16] investigated the use of lookup tables and optimal minimum width interpolation kernels to reduce the number of operations required by an NUFFT evaluation, respectively. Schiwietz et al. [20] were the first to utilize specialized hardware to accelerated NUFFT computation. By taking advantage of the native image rotation capabilities of their ATI Radeon™ X1800 XT card, they were able to accelerate adjoint NUFFT computations for radially-sampled 2D spectral data by up to 3.5x compared to the implementation running on a single 3.0GHz Pentium 4 processor system. Sorensen et al. [17] later considered acceleration of both the forward and backward NUFFT for generic spectral sampling distributions, and explicitly noted the challenge associated with parallelizing the adjoint transform while avoiding race conditions. In that work, the authors propose to divide 2D Cartesian grid into rectangular regions and process each region on a distinct processor. They also preprocess the raw samples to identify set of samples that convolve into specific region of Cartesian grid. Their approach has an overhead where single sample may need to be processed by multiple processors and overhead increases significantly for larger size of convolution window(W). Also, unlike our approach, their approach can significantly suffer from load balancing issues if samples have highly irregular distribution. On ATI FireStream™ 2U platform, using radial and spiral datasets with $W = 2$, their 2D NUFFT implementation achieved up to 20x speedup for both the forward and adjoint operators, compared to 2.13 GHz dual-core CPU. Similarly, in [30], Zhang et al. described two approaches for parallelizing NUFFT. Both approaches use recursive partitioning for better cache utilization. Their "source driven" parallelization is similar to our approach but the task scheduling requires multiple barriers across phases. Their second approach is similar to the one proposed by Sorensen et al. and suffers from similar overheads.

Several groups have also focused on accelerating the 3D NUFFT, with particular focus on non-Cartesian MRI. In [36], Shu et al. described a C-based multithreaded implementation of the 3D NUFFT that ran natively on the MRI scanner console, and was able to perform a $N = 240, K = 512, S = 8047, W = 2.5, OF \sim 1.25$ adjoint NUFFT in about 15s. A more recent implementation of this group's work on a dual socket 12-core Intel® Xeon® X5670 based server machine (WSM12C) running at 2.93GHz with 12MB cache and 24GB DDR3 1333MHz memory that employed thread privatization for parallelization of the adjoint operator was able to execute in only about 1.4s for the adjoint and 0.9s for the forward NUFFT. We ran our implementation using the same parameters except that we set $W = 4$. Table IV shows a comparison of performance on the same hardware (WSM12C) as well on the latest dual socket 16-core Intel® Xeon® E5-2670 processor based server (SNB16C). Even with larger W, our implementation runs about 4.26x faster on the same hardware and about 6.69x faster on SNB16C.

In [21], Nam et al. described 3D NUFFT implementation on the recent NVIDIA® GeForce® GTX 480 hardware. For $N = 344, S = 9000, K = 344$ their implementation finished forward and adjoint NUFFT in only 0.66s and 0.94s, respectively. Table V compares their performance with WSM12C and SNB16C. We see that while the performance of our WSM12C implementation is within 10% of theirs, our SNB16C is 1.44x faster. Finally, Obeid et al. [37] described an NVIDIA® C2050 implementation of the adjoint convolution of 3D NUFFT for $K \times S = 2655910$ and $N = 256$, that finishes in 1s by utilizing a gather rather than scatter approach for updating Cartesian grid, and using a set of preprocessed proximal bins for the neighbor search. Even this approach requires multiple visits to same sample point and does not scale with large convolution window sizes.

Table IV
COMPARISON OF PERFORMANCE BETWEEN MULTI-CORE CPU BASED IMPLEMENTATIONS

	Our Implementation		Shu et al.
	WSM12C	SNB16C	WSM12C
ADJ NUFFT (sec)	0.28	0.18	1.40
FWD NUFFT (sec)	0.26	0.17	0.90
Total (sec)	0.54	0.34	2.30
Speedup	4.26x	6.69x	1.00x

Table V
COMPARISON OF PERFORMANCE BETWEEN MULTI-CORE CPU BASED IMPLEMENTATIONS AND GPU BASED IMPLEMENTATION

	Multi-core CPUs		GPU
	WSM12C	SNB16C	GTX480
ADJ NUFFT (sec)	0.93	0.58	0.94
FWD NUFFT (sec)	0.87	0.54	0.66
Total (sec)	1.79	1.11	1.60
Speedup	0.89x	1.44x	1.00x

VII. CONCLUSIONS

In this paper, we have proposed a novel and highly scalable algorithm for performing NUFFT on x86-based multi-core CPUs. Our algorithm achieves high parallel efficiency and makes good use of SIMD. We demonstrate on average 90% SIMD efficiency using SSE as well up to linear scalability on a quad-socket 40-core Intel® Xeon® E7-4870 processor based server. On dual socket Intel® Xeon® X5670 based server, our NUFFT implementation is more than 4x faster when compared to the best available NUFFT3D implementation, when run on the same hardware. On Intel® Xeon® E5-2670 , our NUFFT implementation is almost 1.5X faster than the best available NUFFT implementation today. Such speed improvement opens new usages for NUFFT, as well as enables practical application of iterative algorithms which require many successive NUFFT calls, such as 3D MRI reconstruction.

REFERENCES

[1] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19, no. 1, pp. 297–301, 1965.

[2] H. Schomberg and J. Timmer, "The gridding method for image reconstruction by Fourier transformation," *IEEE Trans. Med. Imag.*, vol. 14, no. 3, pp. 596–607, 1995.

[3] J. Fessler and B. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation," *IEEE Trans. Sig. Proc.*, vol. 51, no. 2, pp. 560–564, 2003.

[4] B. Sutton, D. Noll, and J. Fessler, "Fast, iterative image reconstruction for MRI in the presence of field inhomogeneities," *IEEE Trans. Med. Imag.*, vol. 22, no. 2, pp. 178–1888, 2003.

[5] H. Choi and D. Munson, "Direct-Fourier reconstruction in tomography and synthetic aperture radar," *Int. J. Imag. Syst. Tech*, vol. 9, no. 1, pp. 1–13, 1998.

[6] E. Candés and D. Donoho, "Ridgelets: a key to higher-dimensional intermittency?" *Phil. Trans. R. Soc. Lond. A.*, vol. 357, no. 1760, pp. 2495–2509, 1999.

[7] S. Bhandarkar, X. Luo, R. Daniels, and E. Tollner, "Automated planning and optimization of lumber production using machine vision and computed tomography," *IEEE Trans. Auto. Sci. Eng*, vol. 5, no. 4, pp. 677–695, 2008.

[8] E. Angelidis and J. Diamessis, "A novel method for designing FIR digital filters with nonuniform frequency samples," *IEEE Trans. Sig. Proc.*, vol. 42, no. 2, pp. 259–267, 1994.

[9] J. Strain, "Fast potential theory II: Layer potentials and discrete sums," *J. Comp. Physics*, vol. 99, no. 2, pp. 251–270, 1992.

[10] A. Dutt and V. Rokhlin, "Fast Fourier transforms for nonequispaced data," *SIAM J. Sci. Comp.*, vol. 14, no. 6, pp. 1368–1393, 1993.

[11] V. Rokhlin and A. Dutt, "Fast Fourier transforms for nonequispaced data, II," *Appl. Comp. Harm. Anal.*, vol. 2, no. 1, pp. 85–100, 1993.

[12] A. Ware, "Fast approximate fourier transforms for irregularly spaced data," *SIAM Rev.*, vol. 40, pp. 838–856, 1998.

[13] Q. Liu and N. Nguyen, "An accurate algorithm for nonuniform fast fourier transforms (NUFFTs)," *IEEE Micro. Guided Wave Let.*, vol. 8, no. 1, pp. 18–20, 1998.

[14] L. Greengard and J. Lee, "Accelerating the nonuniform fast Fourier transform," *SIAM Rev.*, vol. 46, no. 3, pp. 443–454, 2004.

[15] B. Dale, M.Wendt, and J. Duerk, "A rapid look-up table method for reconstructing MR images from arbitrary k-space trajectories," *IEEE Trans. Med. Imag.*, vol. 20, no. 3, pp. 207–217, 2001.

[16] P. Beatty, D. Nishimura, and J. Pauly, "Rapid gridding reconstruction with minimal oversampling ratio," *IEEE Trans. Med. Imag.*, vol. 24, no. 6, pp. 799–808, 2005.

[17] T. Sorensen, T. Schaeffter, K. Noe, and M. Hansen, "Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware," *IEEE Trans. Med. Imag.*, vol. 27, no. 4, pp. 538–547, 2008.

[18] Y. Shu, J. Trzasko, J. Huston, A. Manduca, and M. Bernstein, "Single phase 3D contrast-enhanced intracranial magnetic resonance angiography with undersampled SWIRLS trajectory at 3T," in *Proc. of the ISMRM*, May 2011, p. 2656.

[19] S. Kestur, P. Sungho, and K. I. V. Narayanan, "Accelerating the nonuniform fast Fourier transform using FPGAs," in *Proceedings of FCCM*, May 2010, pp. 19–26.

[20] T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction on the GPU," in *Proc. of the SPIE: Medical Imaging*, February 2006, p. 61423T.

[21] S. Nam, T. Basha, M. A. C. Stehning, W. Manning, V. Tarokh, and R. Nezafat, "A GPU implementation of compressed sensing reconstruction of 3D radial (kooshball) acquisition for high-resolution cardiac MRI," in *Proc. of the ISMRM*, May 2011, p. 2548.

[22] "Intel Advanced Vector Extensions Programming Reference," 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[23] A. Barger, W. Block, Y. Toropov, T. Grist, and C. Mistretta, "Time-resolved contrast-enhanced imaging with isotropic resolution and broad coverage using an undersampled 3D projection trajectory," *Magn. Res. Med.*, vol. 48, no. 2, pp. 297–305, 2002.

[24] E. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information," *IEEE Trans. Info. Theory*, vol. 52, no. 2, pp. 489–509, 2006.

[25] D. Donoho, "Compressed sensing," *IEEE Trans. Info. Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[26] M. Lustig, D. Donoho, and J. Pauly, "Sparse MRI: The application of compressed sensing for rapid MR imaging," *Magn. Res. Med.*, vol. 58, no. 6, pp. 1182–1195, 2007.

[27] J. Trzasko and A. Manduca, "Highly undersampled magnetic resonance image reconstruction via homotopic ℓ_0-minimization," *IEEE Trans. Med. Imag.*, vol. 28, no. 1, pp. 106–121, 2009.

[28] P. Irarrazabel and D. Nishimura, "Fast three dimensional magnetic resonance imaging," *Magn. Res. Med.*, vol. 33, no. 5, pp. 656–662, 1995.

[29] FFTW3 Interface to Intel Math Kernel Library, "http://software.intel.com/sites/products/documentation/hpc /compilerpro/en-us/cpp/win/mkl/refman/appendices/ mkl_appg_fftw3_intro.html."

[30] Y. Zhang, J. Liu, E. Kultursay, M. Kandemir, N. Pitsianis, and X. Sun, "Scalable parallelization strategies to accelerate nufft data translation on multicores," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par'10, 2010, pp. 125–136.

[31] C. Savage, "A survey of combinatorial gray codes," *SIAM Rev.*, vol. 39, pp. 605–629, December 1997.

[32] N. Leischner, V. Osipov, and P. Sanders, "Fermi Architecture White Paper," 2009.

[33] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *Int. J. Parallel Program.*, vol. 29, pp. 217–247, June 2001.

[34] "Intel SSE4 programming reference," 2007, http://www.intel.com/design/processor/manuals/253667.pdf.

[35] FFTW3 Library, "http://www.fftw.org."

[36] Y. Shu, M. Bernstein, J. Huston, and D. Rettmann, "Contrast-enhanced intracranial magnetic resonance angiography with a spherical shells trajectory and online gridding reconstruction," *J. Magn. Res. Imag.*, vol. 30, no. 5, pp. 1101–1109, 2009.

[37] N. Obeid, I. Atkinson, K. Thulborn, and W. Hwu, "GPU accelerated gridding for rapid reconstruction of non-cartesian MRI," in *Proc. of the ISMRM*, May 2011, p. 2547.