Accelerating the Nonequispaced Fast Fourier Transform on Commodity Graphics Hardware

Thomas Sangild Sørensen*, Tobias Schaeffter, Karsten Østergaard Noe, and Michael Schacht Hansen

***Abstract*—We present a fast parallel algorithm to compute the nonequispaced fast Fourier transform on commodity graphics hardware (the GPU). We focus particularly on a novel implementation of the convolution step in the transform as it was previously its most time consuming part. We describe the performance for two common sample distributions in medical imaging (radial and spiral trajectories), and for different convolution kernels as these parameters all influence the speed of the algorithm. The GPU-accelerated convolution is up to 85 times faster as our reference, the open source NFFT library on a state-of-the-art 64 bit CPU. The accuracy of the proposed GPU implementation was quantitatively evaluated at the various settings. To illustrate the applicability of the transform in medical imaging, in which it is also known as gridding, we look specifically at non-Cartesian magnetic resonance imaging and reconstruct both a numerical phantom and an *in vivo* cardiac image.**

***Index Terms*—Discrete Fourier transforms, magnetic resonance imaging (MRI), parallel algorithms, parallel architectures.**

I. Introduction

THIS PAPER describes a parallel implementation of the nonequispaced fast Fourier transform (NFFT) that utilizes modern graphics cards (GPUs) for general purpose computation to achieve a significant speedup compared to even the fastest CPU implementations available. The NFFT is an important transform used in a number of application areas. In medical imaging the adjoint transform $\mathrm{NFFT}^{\mathrm{H}}$ is often referred to as "gridding" and used particularly in magnetic resonance imaging and computed tomography when the sampled data do not conform to a Cartesian grid [1]–[4]. For non-Cartesian magnetic resonance imaging (MRI) in particular, the speed of the most commonly used reconstruction algorithms is defined primarily by the speed of the gridding and inverse gridding implementations [1], [5]–[7]. Gridding alone can be used to reconstruct non-Cartesian MRI and optimized CPU implementations exist [1]–[3], [8]. For real-time applications using a high number of receiver coils they are however not always sufficiently fast. Both gridding and inverse gridding are applied repeatedly in many iterative reconstruction schemes for fast imaging, e.g., parallel imaging [5] and schemes exploiting temporal and spatial correlations [6]. Unfortunately, many potential applications of these imaging protocols are currently not clinically feasible due to unacceptably long reconstruction times. Thus by reducing reconstruction times significantly, a whole new range of imaging sequences could potentially make their way to clinical practice.

The name "nonequispaced fast Fourier transform" stems from our CPU reference implementation, the open source NFFT library [8], [9]. This name was also chosen in [10], but otherwise it varies between publications and application areas. As described in the NFFT tutorial [8], [9] it is known also as the nonuniform fast Fourier transform [11], the generalized fast Fourier transform [12], the unequallyspaced fast Fourier transform [13], the fast approximate Fourier transforms for irregularly spaced data [14], and gridding [1], [3], [4].

Manuscript received July 31, 2007; revised September 24, 2007. The work of T. S. Sorensen was supported by the Danish Research Council under Grant 2059-03-0004. The work of M. S. Hansen was supported by the BHF Cardiovascular Initiative CI/05/010. *Asterisk indicates corresponding author.*

*T. S. Sørensen is with the Division of Imaging Sciences at King's College London, London WC2R 2LS, U.K. He is also with the Centre for Advanced Visualization and Interaction, University of Aarhus, Aarhus DK 8200, Denmark, and with the Centre for Medical Image Computing, Department of Medical Physics and Bioengineering, University College London, London WC1E 6BT, U.K. (e-mail: sangild@cavi.dk).

T. Schaeffter is with the Division of Imaging Sciences at King's College London, London WC2R 2LS, U.K. (e-mail: tobias.schaeffter@kcl.ac.uk).

K. Ø. Noe is with the Department of Computer Science, University of Aarhus, Aarhus DK 8200, Denmark (e-mail: kn@daimi.au.dk).

M. S. Hansen is with the Institute of Child Health at University College London, London WC1N 1EH, U.K. (e-mail: michael.hansen@ucl.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TMI.2007.909834

II. Theory

This section is comprised of several subsections. First, we briefly describe the NDFT. This is followed by an introduction to the NFFT. We then briefly review and discuss some overall concepts of general purpose computation on GPUs before analyzing and describing our parallel GPU implementation of the NFFT in the subsequent section.

A. NDFT

Using the notation from [8], [9] let M denote the number of nonequidistant samples $\mathbf{x}$ in a given sampling set. Let f_j denote the complex Fourier coefficient corresponding to the sample positioned at $\mathbf{x}_j$. Let I_N denote the set of equidistant Cartesian grid cells of dimension d and let $|I_N|$ denote the number of cells in this set. For a finite number of complex Fourier coefficients $\hat{f}_{\mathbf{k}}$ corresponding to grid cells $\mathbf{k}$ we are interested in computing

$$f_j = \sum_{\mathbf{k}\in I_{\mathbf{N}}} \hat{f}_{\mathbf{k}} e^{-2\pi i \mathbf{k}\mathbf{x}_j} \text{ and } \hat{f}_{\mathbf{k}} = \sum_{j=0}^{M-1} f_j e^{2\pi i \mathbf{k}\mathbf{x}_j}. \quad (1)$$

The complexity of this computation is $\mathrm{O}(M|I_N|)$ arithmetic operations. If the sampling set is equispaced however and $M = |I_N|$ this special case can be computed in $\mathrm{O}(|I_N|\log|I_N|)$ operations using the well known fast Fourier transform (FFT) [15].

B. NFFT

The generalization of the FFT to nonequidistant samples is an approximate algorithm consisting of three steps; rolloff correction, an FFT, and a convolution. The FFT is used to transform the input grid to a grid in frequency space. This is followed by a convolution of the computed grid onto the points in the sampling set. The convolution kernel is of fixed size independent of the grid resolution. For convolution rolloff correction (or deapodization) [4] the input grid is initially divided by the Fourier transformed convolution kernel. A formal derivation of the algorithm can be found in [8] and [9]. Outlines of the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ algorithms are shown in Fig. 1. Their complexity is $\mathrm{O}(|I_N|\log|I_N| + \mathrm{M})$ arithmetic operations. An oversampling factor $\sigma \geq 1$ (which is multiplied both to the input grid size and the convolution kernel size W) is used to reduce aliasing artifacts. In [4], an oversampling factor of two is suggested and kernel parameters minimizing the average aliasing errors are determined for a range of convolution kernels. It has since been shown that the maximum aliasing error in an image can be maintained at reduced oversampling factors by increasing the convolution kernel size only slightly [3].

Of the three steps in the two algorithms, the FFT and the convolution steps constitute the significant part of the execution time as rolloff correction is merely a division. The NFFT reference library [8], [9] comes with a benchmark application from which we can deduce the relationship between the cost of the FFT and the convolution for a range of image sizes. For images and sample sets sized 128^2, 256^2, and 512^2 ($\sigma = 2$, $W = 4$) the convolution step constitutes between 83% and 95% of the overall execution time. By reducing σ and increasing W as suggested in [3] the balance is shifted even further towards the convolution step, making it the single most important part to speed up. The aim of this work is exactly this; to significantly improve the convolution (and hereby the entire NFFT and $\mathrm{NFFT}^{\mathrm{H}}$) execution times by an efficient parallel GPU implementation.

NFFT algorithm

Input: Complex time domain coefficients $\hat{f}_{\mathbf{k}}$ corresponding to the equispaced grid cells $\mathbf{k}$.

Output: Approximate complex frequency domain coefficients f_j corresponding to the non-equispaced samples $\mathbf{x}_j$.

1. **Kernel rolloff correction**. Compute $\hat{g}_{\mathbf{k}}$.
   Divide $\hat{f}_{\mathbf{k}}$ with the coefficients of the Fourier transformed convolution kernel.
2. **FFT**. Compute $g_{\mathbf{l}}$.
   Compute the fast Fourier transform of $\hat{g}_{\mathbf{k}}$.
3. **Convolution**. Compute f_j.
   Compute the convolution of the complex values $g_{\mathbf{l}}$ at the equispaced grid cells $\mathbf{k}$ onto the non-equispaced samples $\mathbf{x}_j$.

$\mathrm{NFFT}^{\mathrm{H}}$ algorithm

Input: Complex frequency domain coefficients f_j corresponding to the non-equispaced samples $\mathbf{x}_j$.

Output: Approximate complex time domain coefficients $\hat{f}_{\mathbf{k}}$ corresponding to the equispaced grid cells $\mathbf{k}$.

1. **Convolution**. Compute $g_{\mathbf{l}}$.
   Compute the convolution of the complex coefficients f_j at the non-equispaced samples $\mathbf{x}_j$ onto the equispaced grid cells $\mathbf{k}$.
2. **FFT**. Compute $\hat{g}_{\mathbf{k}}$.
   Compute the inverse fast Fourier transform of $g_{\mathbf{l}}$.
3. **Kernel rolloff correction**. Compute $\hat{f}_{\mathbf{k}}$.
   Divide $\hat{g}_{\mathbf{k}}$ with the coefficients of the Fourier transformed convolution kernel.

Fig. 1. Algorithm outlines for computing the NFFT and the $\mathrm{NFFT}^{\mathrm{H}}$, respectively [8], [9].

C. General Purpose Computation on the GPU

General purpose computation on the GPU (GPGPU) is still an emerging research area, but many algorithms have been accelerated significantly compared to their CPU counterparts already at present. We refer to the introductory papers [16]–[18] for a thorough overview. Until very recently GPGPU was implemented by exploiting an established application programming interface (API), i.e., OpenGL or DirectX, for computation rather than graphics. One would create an off-screen memory buffer and "render" geometry to invoke computation herein. So-called fragment shaders were programmed to calculate a quadruple (i.e., color) for each pixel in the memory buffer *in parallel*. One of the most recent GPUs, the Geforce 8800 GTX (Nvidia, Santa Clara, CA), has 128 "stream processors" for these computations. Hence, a significant acceleration can potentially be obtained if a computational problem can be solved in parallel. It has been a limiting factor though to be restricted by graphics APIs designed for rendering rather than computation. Fortunately, this is now changing as both major hardware manufacturers (ATI Technologies, Marham, ON, Canada and Nvidia) have both released new APIs dedicated for GPGPU—namely CTM [19] and CUDA [20]. The GPU can now be used for computation through high-level C-like programming languages without knowledge of graphics programming in general. Although different restrictions do apply in the two APIs, one can consider the GPU as a multiprocessor where each individual processor can read from and write to a shared memory pool.

An important concept when analyzing the performance of a GPU-based algorithm is its *arithmetic intensity*. This is defined as the "amount of computational work" that is performed per off-chip (global) memory access (e.g., [21]). Applications with high arithmetic intensity are most likely compute bound while a low arithmetic intensity is an indication of a memory bound algorithm. To see why this concept is important, Fig. 2

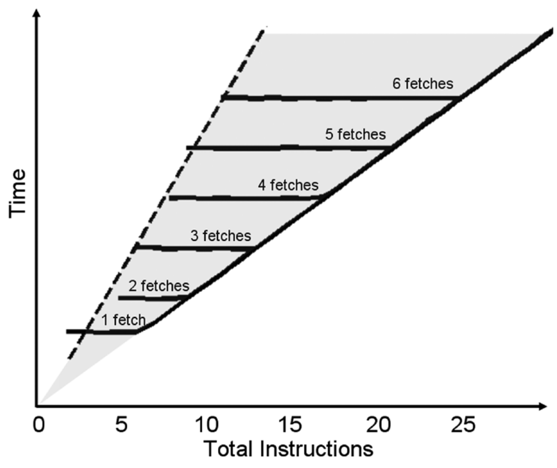

Fig. 2. Memory access costs as a function of the number of shader instructions (solid lines). Number of memory fetches is varied from 1 to 6. Data was obtained using GPUBench [22] on an ATI Radeon X1800 XT.

shows the execution time of six programs with different arithmetic intensities as a function of the number of instructions. The data making up the figure was obtained using the GPU benchmark suite GPUBench [22]. The figure is comprised of six subtests that perform one to six memory cache accesses each (solid lines). Notice the horizontal line segments. They show that for each memory access, a number of "free" computations can be made without influencing the overall execution time if they are independent of the result of the prior memory fetches. Only as the diagonal part of the graph is reached, there is a cost associated to issuing additional instructions. From the figure, we can predict the execution time of an application consisting entirely of memory reads by following the stippled line. Notice that the slope of this line is much steeper than the slope of the diagonal solid line, which constitutes the border between a memory bound and a truly compute bound application. An application with an arithmetic intensity that places it between the stippled line and the solid diagonal line is memory bound, while an application with an arithmetic intensity that places it on (or close to) the diagonal would be compute bound. The performance of a compute bound application will grow with the rapid increase in arithmetic performance from each new generation of GPUs, whereas a memory bound application will increase speed as a function of memory bandwidth, which unfortunately is much slower growing. Whenever possible, one should use on-chip memory (e.g., registers or dedicated fast memory) to avoid the cost of expensive global memory fetches. As it is often impossible to interleave all memory fetches with computations independent of their results, the GPU APIs handle several concurrent threads on each processor. When a thread requires memory access on which the subsequent instruction relies on, the processor will switch to a different thread to avoid being idle. Once the data arrives it can again switch back to the original thread. The number of concurrent threads is determined by the amount of on-chip storage for each processor. We refer to [21] for further discussion of this topic.

To summarize the discussion above, it is crucial to minimize the overall number of memory accesses for a given algorithm in order to obtain maximum performance. Ideally, each memory access should be interleaved with computations independent of its result to hide its cost completely. When at all possible, one should use on-chip memory such as registers for computations as there is no penalizing cost associated to accessing these.

III. NFFT on the GPU

As described in the previous section, the NFFT and the $\mathrm{NFFT}^{\mathrm{H}}$ consist of three steps each: rolloff correction, the FFT, and a convolution. Prior to this work the convolution step was the far most time consuming and consequently the step we have focused on improving. Its implementation is described shortly. Several GPU accelerated implementations of the FFT have been published previously, e.g., [23]–[26]. In this work, we used the approach described in [24]. As rolloff correction is simply a division at each grid cell, it is straightforward to parallelize.

To analyze and discuss potential parallel convolution algorithms we focus on the convolution step in the $\mathrm{NFFT}^{\mathrm{H}}$ algorithm as this makes graphical illustration simpler. With minor modifications only, the algorithms that we deduce are however valid for the NFFT also. Hence, we describe a parallel convolution of a randomly distributed sampling set onto a Cartesian grid. From our previous discussion, it was clear that the arithmetic intensity, i.e., the amount of computation per memory access, is an important factor for the overall performance. We thus analyze the amount of memory access required for each algorithm we describe. Two algorithms will be optimal with respect to *either* the number of sample reads or the number of grid cell writes—but not both. We consequently introduce instead a novel third algorithm which overcomes this limitation.

The first method to be discussed is a naive parallelization of the conventional CPU approach in which the points from the sampling set are processed one by one and sequentially convolved onto the Cartesian grid. Given p processors we split the sampling set into p subsets and let each processor be responsible for the convolution of exactly one subset. Compared to a machine with just one processor, this approach could potentially provide a speedup factor of p. One needs to ensure however that no two processors will attempt to write to the same memory address concurrently as this would produce undefined results. This approach is illustrated in Fig. 3 (left). The overall number of memory accesses required for the p processors is proportional to $M + 2W^dM$. As previously described, W denotes the convolution kernel width and M and d are the number of samples and data dimension, respectively. The leftmost part of the sum constitutes the number of memory reads used to process the M samples. The rightmost part of the sum describes the number of memory accesses required to store the result of the convolution. For each sample, the kernel size determines the number of grid cells to write to. The additional factor of 2 follows from the fact that we must accumulate the contribution of the samples. For each grid cell, we thus require both a read and a write to account for each sample. Please note however that support for hardware accelerated additive blending would eliminate this factor 2. The leftmost part of the sum is optimal since each sample is only read once. For the rightmost part of the sum, unfortunately the cost is (close to) optimal for very sparse sampling sets only. As M

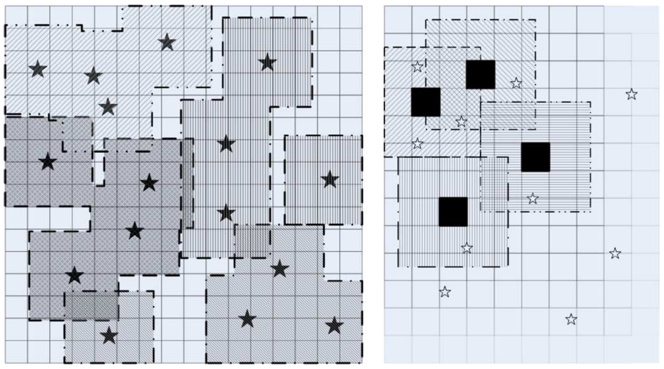
Fig. 3. Two parallel algorithms for the convolution of a sampling set (stars) onto a Cartesian grid in two dimensions. On the left, the samples are gathered in four groups to be processed by four processors concurrently. Each group is illustrated by a unique shading pattern highlighting a convolution kernel of size of 4^2 around each sample. On the right, one processor is assigned to the computation of each grid cell. Convolution kernels of size of 4^2 are highlighted around four selected grid cells marked in solid black.

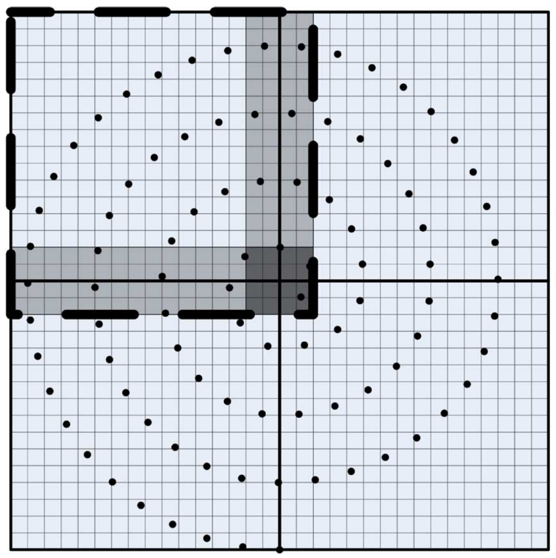
Fig. 4. Our hybrid algorithm for the convolution of a sampling set (spiral dots) onto a Cartesian grid in two dimensions. In this example, the grid cells are divided into four groups (thick solid black lines) to be processed by four processors concurrently. With a convolution kernel size of 4^2 all sampling nodes within the stippled square are read by the processor associated to the top left quadrant. Nodes in the rectangles filled with light grey shading are read by two processors, while nodes in the darkly shaded square are read by four processors.

approaches the number of grid cells, we will be updating each cell repeatedly.

The excessive amount of accumulative memory writes in the previous approach leads us to an alternative algorithm in which we only write to each grid cell once. This approach has previously been used for reconstruction of MRI images [26]. Each processor is now responsible for the computing the convolution onto a given grid cell. If the number of cells exceeds the number of processors, each processor computes the convolution onto several cells sequentially. This is illustrated in Fig. 3 (right). A utility input data structure is used to identify the list of sampling points that influence a given grid cell. Alternatively, for certain sampling trajectories this list can be computed online as part of the algorithm. The accumulated value of the convolution of the list entries is written to the associated grid cell. This algorithm requires only the optimal $|I_N|$ memory writes—one for each Cartesian grid cell. The number of times each sample is read however is determined by the convolution kernel size and data dimension. The total number of sample reads is proportional to W^dM. This is far from optimal even for modest convolution kernel sizes.

The two parallel algorithms presented above have been optimal with respect to either the number of sample reads or the number of grid cell writes. The final algorithm we present can be considered a hybrid of the two previous algorithms that performs well with respect to both the number of memory reads and writes. It is in fact a generalization of the latter of the two previous algorithms to which it reduces in a worst case scenario. In the previous algorithm each processor was responsible for computing the convolution onto a given grid cell. In our generalization, each processor is instead responsible for computing the convolution onto a set of neighboring points, i.e., a rectangular area of the grid. The input data structure which in the previous algorithm mapped each processor to a list of samples from the sampling set is extended to cover all samples that convolve into in the associated rectangle. Each processor now iterates through this list, accessing each sample only once. Each sample is convolved onto the cells in the associated rectangle when it is processed. *The key here being that the cells are stored in local on-chip memory, e.g., registers.* When the input list has been processed, the registers holding the convolution result are written to global memory. The algorithm is illustrated in Fig. 4. As in the previous algorithm it requires only the optimal $|I_N|$ global memory writes. We have, however, significantly improved the overall number of sample reads. All samples that are interior to a single rectangular grid area, i.e., samples that do not convolve into other areas, are only read once which is optimal. Some "boundary samples," i.e., those inside the areas with light or dark grey shading in Fig. 4, convolve into several regions and are consequently read multiple, say n times. However, as $n \leq W^d$, only if the rectangular area is chosen as unit size will the hybrid algorithm perform an overall of W^dM sample reads as in the previous algorithm. Section III-B below contains a description of the necessary utility data structure, which has only a small additional cost.

A. Implementation Details

We implemented the hybrid convolution algorithm on an ATI FireStream 2U graphics card with 1 GB of memory through CTM [19] on a PC with 2 GB of memory and an 2.13 GHz dual core processor running Windows XP. The GPU code was written in HLSL, a high-level C-like language, and compiled with the Microsoft FXC compiler shipped with the DirectX 9 SDK [27]. The algorithm requires each processor to write multiple outputs, i.e., scatter. As scatter is currently not supported in HLSL, it was necessary to add the scatter statements manually in the compiled Pixel Shader 3 assembly code [27]. To make this easy, we compute the scatter values and addresses in the HLSL code. To avoid compiler optimizations changing these computations, we insert a multiplication of these with a dummy variable whose value is unknown at compile time. This allows us to quickly insert some "global" code snippets to handle the scatter requirements after each compilation. Another limitation induced by the FXC compiler is the number of registers it will allow in the resulting assembler code (currently 32 quadruple floating point values). With our present implementation this limits each processor to handle rectangles of eight complex valued cells. The GPU based convolution using a precomputed kernel for the NFFT is an exception from this statement. Due to lack of registers, we have omitted realizing this specific scenario.

Our GPU implementation of the FFT was obtained by directly translating the source code accompanying [24] to CTM–no attempts were made to reduce the memory bandwidth requirements for the FFT.

Our CPU reference implementation, the open source NFFT library [8], [9], was compiled on a 64 bit Linux machine running Fedora Core 6 on an Intel Xeon 2.33 GHz dual core processor with 4 GB of memory. All optimization flags were turned on.

B. Preprocessing

The GPU-based $\mathrm{NFFT}^{\mathrm{H}}$ algorithm described above requires a preprocessing step to build an essential data structure: the mapping from each processor id to the set of sample points that convolve onto any grid cell the processor is responsible for updating. As the driving force for this work was non-Cartesian magnetic resonance image reconstruction we have chosen a data representation that performs best with sample trajectories common in this scenario (spiral and radial trajectories). A different representation should be developed for optimal performance if sample points were randomly distributed. We notice that series of successive sampling points convolve into a given rectangle on the Cartesian grid. We thus store each of these series by the sample index to the first sample followed by the number of successive samples. This representation is a compressed data format that reduces memory bandwidth by retrieving a set of sample indices by a single memory access. For the convolution step in the NFFT algorithm (as opposed to the $\mathrm{NFFT}^{\mathrm{H}}$) the mapping is between processor ids and sets of grid cells. We then store rows or columns of grid cells in a similar encoding. In either case the data structure can be computed in time $\mathrm{O}(\mathrm{M} + |I_N|)$, i.e., it is linearly dependant on the number of samples and the image size.

Density compensation was performed prior to initiating $\mathrm{NFFT}^{\mathrm{H}}$ computations by multiplication of the sample values and density weights. This could be done on either the CPU or GPU.

The CPU reference benchmark application also utilizes an optimization that is linear in the number of samples with respect to time: The convolution kernel is precomputed and accessed through a lookup table. A precomputed kernel can also be selected in our GPU algorithm depending on the desired configuration.

C. Optimizations

In most previous work on the non-Cartesian Fourier transform, it has been customary to precompute the convolution kernel and store it in a lookup table. This has been the preferred approach on the GPU also. A precomputed kernel provides freedom to use any kernel shape without changing the speed of the convolution. However, it also constitutes an additional memory bandwidth load, which we are trying to reduce. As an alternative, we also implemented the hybrid convolution algorithm to use a Gaussian kernel or a Kaiser–Bessel kernel computed online. When computing the kernel online on a GPU operating on quadruple data, it is important to express computations to utilize such vector operations whenever possible. Unfortunately, the ATI FireStream 2U GPU allows the exponential in the Gaussian kernel to be computed only on scalar values. It thus has to be repeated for each component of the vector. The Kaiser–Bessel kernel on the other hand, consists mostly of multiplications and additions, which can easily be implemented using the quadruple vector operations.

Just as we have a choice of either pre- or online computation of the kernel, the same can be said about the samples' positions when the sampling trajectory is known (as for e.g., spiral or radial MRI). By computing the sample positions online we effectively halve the memory bandwidth required to access the samples. Moreover, when fetching a quadruple of data from memory containing a sample's position and complex value, the sample position is required immediately afterwards to compute the convolution, which is dependant on the distance between the sample point and the grid cell positions. A lot of thread switching could potentially be avoided if we instead computed the sample position while independently looking up its associated sample value (for free). This optimization is only relevant for the convolution in the $\mathrm{NFFT}^{\mathrm{H}}$ computation. In this case, each processor is iterating over a potentially large number of sample points. For the convolution in the NFFT algorithm, the position of the grid cells are known from our data structure encoding.

D. Experiments

We implemented the proposed hybrid GPU implementation of the NFFT and the $\mathrm{NFFT}^{\mathrm{H}}$ algorithms and compared its speed to the CPU reference implementation. Consistently with this reference we chose an oversampling factor $\sigma = 2$ and kernel width $W = 4$ for these tests. Performance is reported for both spiral and radial trajectories with several convolution kernels at image and sample resolutions of 128^2, 256^2, and 512^2 keeping the number of samples and the number of grid cells equal. We also evaluated the performance implications for the varying optimizations suggested above.

E. MRI Application

We applied the proposed algorithm to non-Cartesian MRI of a numerical phantom and an *in vivo* acquisition.

The phantom was sampled using two non-Cartesian MRI trajectories; radial and spiral. The trajectories were designed to conform to the specifications of typical gradient hardware. Specifically for the spirals, a maximum gradient strength of 40 mT/m and a slew rate of 140 T/m/s were used. The spiral trajectories had approximately constant angular velocity at the center of k-space and close to constant linear velocity at the edges of k-space. A smooth transition between constant angular and constant linear velocities was used [28]. The radial acquisitions were sampled equidistant along each radial k-space profile.

TABLE I
SPEED MEASUREMENTS FOR THE NFFT ALGORITHM ON THE CPU AND THE GPU FOR DIFFERENT MATRIX SIZES AND SAMPLING TRAJECTORIES. OVERSAMPLING FACTOR $\sigma = 2$ AND KERNEL SIZE $W = 4$ WAS USED FOR ALL MEASUREMENTS. REPORTED TIMES ARE MEASURED IN SECONDS

NFFT - trajectories computed offline

		Convolution			FFT	Deapo-dization	Total		
		G	L	KB			G	L	KB
Spiral trajectory									
128x128 samples/matrix	**CPU**	-	3.3e^{-2}	-	1.7 e^{-3}	7.6 e^{-4}	-	3.7 e^{-2}	-
	GPU	1.0 e^{-3}	-	1.3 e^{-3}	3.6 e^{-3}	1.9 e^{-4}	5.4 e^{-3}	-	5.6 e^{-3}
	Factor	**33**	-	**25**	**0.5**	**4.0**	**7**	-	**7**
256x256 samples/matrix	**CPU**	-	1.7e^{-1}	-	1.6 e^{-2}	3.0 e^{-3}	-	1.9 e^{-1}	-
	GPU	3.5 e^{-3}	-	4.0 e^{-3}	1.1 e^{-2}	5.2 e^{-4}	1.6 e^{-2}	-	1.7 e^{-2}
	Factor	**49**	-	**43**	**1.5**	**5.8**	**12**	-	**11**
512x512 samples/matrix	**CPU**	-	1.0e^{0}	-	7.2 e^{-2}	1.2 e^{-2}	-	1.2 e^{0}	-
	GPU	1.3 e^{-2}	-	1.5 e^{-2}	4.5 e^{-2}	2.2 e^{-3}	6.3 e^{-2}	-	6.5 e^{-2}
	Factor	**77**	-	**67**	**1.6**	**5.5**	**19**	-	**18**
Radial trajectory									
128x128 Samples/matrix	**CPU**	-	3.3e^{-2}	-	1.7 e^{-3}	7.6 e^{-4}	-	3.7 e^{-2}	-
	GPU	1.4 e^{-3}	-	1.8 e^{-3}	2.9 e^{-3}	1.7 e^{-4}	4.9 e^{-3}	-	5.3 e^{-3}
	Factor	**24**	-	**18**	**0.6**	**4.0**	**8**	-	**7**
256x256 Samples/matrix	**CPU**	-	1.7e^{-1}	-	1.6 e^{-2}	3.0 e^{-3}	-	1.9 e^{-1}	-
	GPU	5.0 e^{-3}	-	5.8 e^{-3}	1.1 e^{-2}	5.2 e^{-4}	1.7 e^{-2}	-	1.8 e^{-2}
	Factor	**34**	-	**29**	**1.5**	**5.8**	**11**	-	**11**
512x512 Samples/matrix	**CPU**	-	1.0e^{0}	-	7.2 e^{-2}	1.2 e^{-2}	-	1.2 e^{0}	-
	GPU	1.8 e^{-2}	-	2.1 e^{-2}	4.4 e^{-2}	2.2 e^{-3}	6.8 e^{-2}	-	7.0 e^{-2}
	Factor	**56**	-	**48**	**1.6**	**5.5**	**18**	-	**17**

The *in vivo* cardiac MRI dataset was acquired using a radial acquisition. The sequence used was a steady state free precession acquisition with $\mathrm{TR} = 3.03$ ms and $\mathrm{TE} = 1.51$ ms. The matrix size was 128×128 pixels, field-of-view was 320 mm and 128 projections were acquired. Each projection was oversampled along the readout direction by a factor of two.

IV. RESULTS

CPU and GPU performance measurements of the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ implementations can be seen in Tables I–IV. In Tables I–III, we used an oversampling factor $\sigma = 2$ and a kernel width $W = 4$ for all measurements corresponding to the default settings of the CPU-based NFFT benchmark application. In Tables I–II, all sample positions were fetched from memory. In Table III, we report the impact of our proposed optimization of the $\mathrm{NFFT}^{\mathrm{H}}$ convolution, i.e., computing the sample trajectories online. The execution time of the different steps of the algorithm can be found in columns and the performance for the various image sizes and trajectories is shown in the different rows. On the GPU the performance of the convolution step was measured with three types of kernels; a Gaussian kernel computed online (G), a Kaiser–Bessel kernel computed online (KB), and using a lookup table (L). For the CPU implementation, we always used a precomputed lookup table as we would otherwise make an unfair comparison. Rows denoted "factor" provide the speedup factor between the CPU and the GPU. The columns named "Total" provide the overall execution time for the entire $\mathrm{NFFT}/\mathrm{NFFT}^{\mathrm{H}}$ algorithms. This column includes besides from the three main steps of the algorithm (convolution, FFT, and deapodization) also some "computational overhead" to connect these steps—such as wrapping and unwrapping image and frequency before and after the FFT, and zero filling of image space initiating the NFFT. Consequently, the total execution time in each row is possibly slightly higher as the sum of the contributions from the three main steps. Fig. 5 summarizes maximum speedup factors obtained at the various resolutions. Table IV shows the effect of varying the oversampling factor and kernel width for a radial acquisition. The column "Sum" denotes the execution time and acceleration factor for performing both a convolution and an FFT.

Fig. 6 compares the reconstruction of a numerical MRI phantom on the CPU and GPU using the $\mathrm{NFFT}^{\mathrm{H}}$ algorithm, i.e., gridding. We selected the best performing GPU configurations: spiral trajectories fetched from memory and radial trajectories computed online both with a 4×4 Gaussian kernel computed online. On the CPU, we used a precomputed Kaiser–Bessel kernel of the same width. To the eye there are no noticeable differences in the images. This is also true for the in vivo cardiac MRI example provided in Fig. 7. A quantitative evaluation of the reconstruction quality for various settings is presented in Table V. The table evaluates the reconstructions of both the numerical phantom and the *in vivo* cardiac acquisition shown in Figs. 6 and 7. The table lists the root mean square (rms) errors of the reconstruction computed on the CPU and the GPU, respectively, compared to reference images obtained from a double precision CPU implementation of the $\mathrm{NDFT}^{\mathrm{H}}$. To challenge our GPU implementation the most the numbers are based on a setting in which we computed online both a Gaussian kernel and the sample trajectories. The CPU reconstruction utilized a Gaussian kernel lookup table for comparison. Kernel control parameters were chosen according to [4]. Fig. 8 shows a difference image between a CPU and GPU reconstruction of the numerical phantom. The images differ only by random noise at a magnitude of 10^{-6}.

TABLE II
SPEED MEASUREMENTS FOR THE $\mathrm{NFFT}^{\mathrm{H}}$ ALGORITHM ON THE CPU AND THE GPU FOR DIFFERENT MATRIX SIZES AND SAMPLING TRAJECTORIES. OVERSAMPLING FACTOR $\sigma = 2$ AND KERNEL SIZE $W = 4$ WAS USED FOR ALL MEASUREMENTS. REPORTED TIMES ARE MEASURED IN SECONDS

NFFT$^{\mathrm{H}}$ - trajectories computed offline

		Convolution			FFT	Deapo-dization	Total		
		G	L	KB			G	L	KB
Spiral trajectory									
128x128 samples/matrix	**CPU**	-	3.8 e^{-2}	-	1.8 e^{-3}	7.5 e^{-4}	-	4.0 e^{-2}	-
	GPU	1.1 e^{-3}	1.8 e^{-3}	1.6 e^{-3}	3.9 e^{-3}	2.2 e^{-4}	5.2 e^{-3}	6.6 e^{-3}	6.2 e^{-3}
	Factor	**35**	**21**	**24**	**0.5**	**3.4**	**8**	**6**	**6**
256x256 samples/matrix	**CPU**	-	1.9 e^{-1}	-	1.8 e^{-2}	3.0 e^{-3}	-	2.1 e^{-1}	-
	GPU	3.5 e^{-3}	5.2 e^{-3}	5.1 e^{-3}	1.0 e^{-2}	5.0 e^{-4}	1.4 e^{-2}	1.9 e^{-2}	1.6 e^{-2}
	Factor	**54**	**37**	**35**	**1.8**	**6.0**	**15**	**11**	**13**
512x512 samples/matrix	**CPU**	-	1.1 e^{0}	-	7.4 e^{-2}	1.2 e^{-2}	-	1.3 e^{0}	-
	GPU	1.3 e^{-2}	1.8 e^{-2}	1.8 e^{-2}	4.5 e^{-2}	2.3 e^{-3}	6.4 e^{-2}	6.9 e^{-2}	6.9 e^{-2}
	Factor	**85**	**61**	**61**	**1.6**	**5.2**	**20**	**19**	**19**
Radial trajectory									
128x128 samples/matrix	**CPU**	-	3.8 e^{-2}	-	1.8 e^{-3}	7.5 e^{-4}	-	4.0 e^{-2}	-
	GPU	1.5 e^{-3}	2.0 e^{-3}	1.9 e^{-3}	3.9 e^{-3}	2.2 e^{-4}	5.9 e^{-3}	6.4 e^{-3}	6.7 e^{-3}
	Factor	**25**	**19**	**20**	**0.5**	**3.4**	**7**	**6**	**6**
256x256 samples/matrix	**CPU**	-	1.9 e^{-1}	-	1.8 e^{-2}	3.0 e^{-3}	-	2.1 e^{-1}	-
	GPU	4.3 e^{-3}	5.6 e^{-3}	5.4 e^{-3}	1.2 e^{-2}	5.7 e^{-4}	1.8 e^{-2}	1.9 e^{-2}	1.9 e^{-2}
	Factor	**44**	**34**	**35**	**1.5**	**5.2**	**12**	**11**	**11**
512x512 samples/matrix	**CPU**	-	1.1 e^{0}	-	7.4 e^{-2}	1.2 e^{-2}	-	1.3 e^{0}	-
	GPU	1.5 e^{-2}	1.8 e^{-2}	1.7 e^{-2}	4.5 e^{-2}	2.2 e^{-3}	6.5 e^{-2}	6.9 e^{-2}	6.8 e^{-2}
	Factor	**73**	**61**	**65**	**1.6**	**5.5**	**20**	**19**	**19**

TABLE III
SPEED MEASUREMENTS FOR THE $\mathrm{NFFT}^{\mathrm{H}}$ ALGORITHM WITH ONLINE COMPUTATION OF THE SAMPLING TRAJECTORIES. REPORTED TIMES ARE MEASURED IN SECONDS

NFFT$^{\mathrm{H}}$ - trajectories computed online

		Convolution	
		G	KB
Spiral trajectory			
128x128 samples/matrix	**CPU**	3.8 e^{-2}	
	GPU	1.4 e^{-3}	1.8 e^{-3}
	Factor	**27**	**21**
256x256 samples/matrix	**CPU**	1.9 e^{-1}	
	GPU	4.7 e^{-3}	6.1 e^{-3}
	Factor	**40**	**31**
512x512 samples/matrix	**CPU**	1.1 e^{0}	
	GPU	1.9 e^{-2}	2.4 e^{-2}
	Factor	**58**	**46**
Radial trajectory			
128x128 samples/matrix	**CPU**	3.8 e^{-2}	
	GPU	1.4 e^{-3}	2.2 e^{-3}
	Factor	**27**	**17**
256x256 samples/matrix	**CPU**	1.9 e^{-1}	
	GPU	4.0 e^{-3}	5.9 e^{-3}
	Factor	**48**	**32**
512x512 samples/matrix	**CPU**	1.1 e^{0}	
	GPU	1.3 e^{-2}	1.9 e^{-2}
	Factor	**85**	**58**

TABLE IV
INFLUENCE OF THE OVERSAMPLING FACTOR (α) AND KERNEL WIDTH (W) ON THE ACCELERATION FACTOR FOR THE $\mathrm{NFFT}^{\mathrm{H}}$ ALGORITHM ON A RADIAL ACQUISITION OF 256^2 SAMPLES ON A 256^2 MATRIX. THIS TABLE RELATES TO THE LOWER PART OF TABLE II. REPORTED TIMES ARE MEASURED IN SECONDS

NFFT$^{\mathrm{H}}$ - trajectories computed offline

		Gaussian Convolution	FFT	Sum
α = 2 W =3	**CPU**	1.4 e^{-1}	1.8 e^{-2}	1.6 e^{-1}
	GPU	3.7 e^{-3}	1.2 e^{-2}	1.6 e^{-2}
	Factor	**38**	**1.5**	**10**
α = 1 W = 4	**CPU**	6.3 e^{-2}	1.8 e^{-3}	6.5 e^{-2}
	GPU	2.9 e^{-3}	3.9 e^{-3}	6.8 e^{-3}
	Factor	**20**	**0.5**	**10**

V. DISCUSSION

The main contribution of this paper is a GPU-accelerated implementation of the convolution step in the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ algorithms. Significant improvements in speed of up to a factor of 85 were achieved compared to our CPU reference implementation. As the convolution step was previously the predominant component in the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ algorithms with respect to reconstruction times, up to twenty-fold reductions in the overall reconstructions times were achieved. For spiral imaging, we achieved convolution speedup factors of 85 and 77 using Gaussian kernels on image and sample sizes of 512^2. At resolution 256^2 these speedup factors were 54 and 49, and for a resolution of 128^2 they were 35 and 33. For radial imaging, the best performance was obtained when computing the trajectories online for the $\mathrm{NFFT}^{\mathrm{H}}$ algorithm. At resolution 512^2 the speedup factors were 85 and 56, at resolution 256^2 they were 48 and 34, and at a resolution of 128^2 they were 27 and 24. Performance was thus comparable for radial and spiral imaging at their respective optimal configurations. Generally speaking, the more samples to convolve, the larger an acceleration factor was obtained. From Table IV, it can be observed that the acceleration factor is also influenced by the effective kernel size (αW).

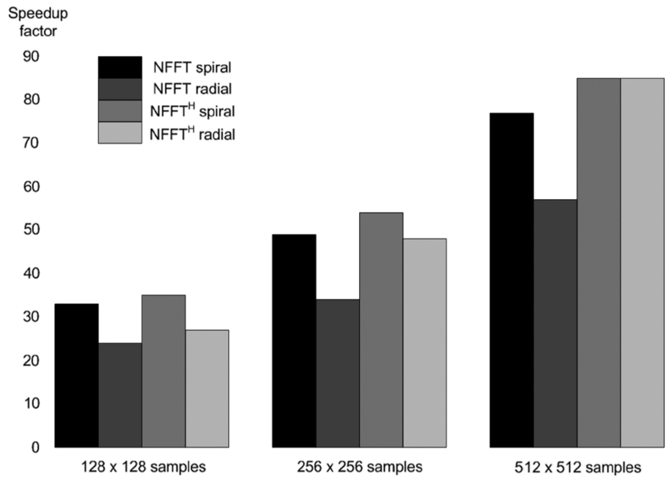

Fig. 5. Speedup factors obtained for our GPU based convolution algorithm at optimal configurations according to Tables I–III. Different shades refer to spiral and radial acquisitions for the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$, respectively. Number of samples is varied between 128^2, 256^2, and 512^2.

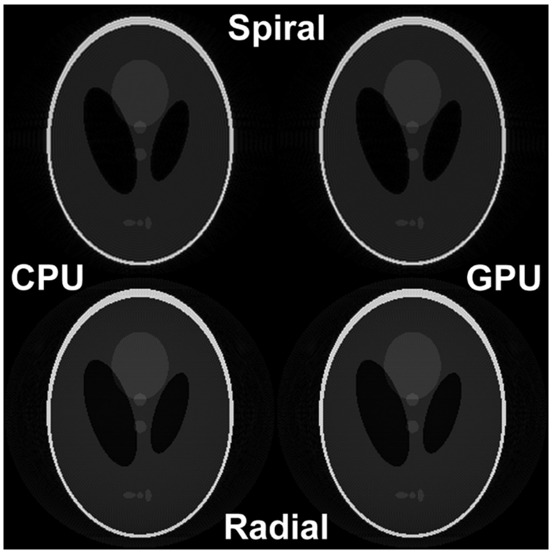

Fig. 6. Reconstruction of a numerical MRI phantom on the CPU (left) and GPU (right). 256^2 points were sampled on spiral trajectories (top) and radial trajectories (bottom). A 4 × 4 Kaiser–Bessel convolution kernel was used on the CPU and a 4 × 4 Gaussian kernel on the GPU. An oversampling factor of two was applied.

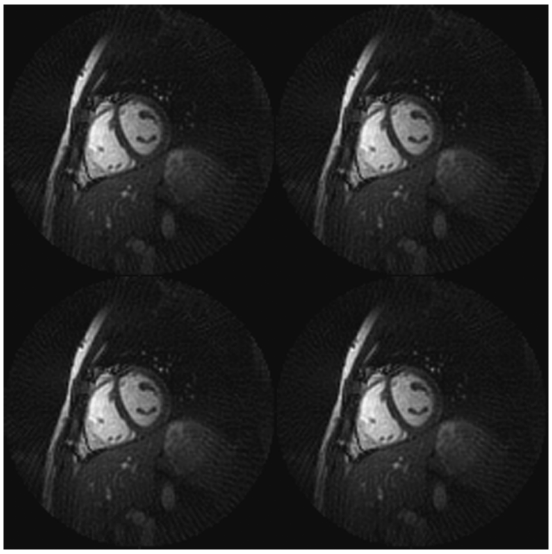

Fig. 7. Reconstruction of an *in vivo* cardiac MRI acquisition. Top left: reference $\mathrm{NDFT}^{\mathrm{H}}$ reconstruction. The remaining three images were reconstructed with the $\mathrm{NFFT}^{\mathrm{H}}$ algorithm (gridding) on the GPU using a Gaussian kernel with different oversampling factors (α) and kernel widths (W). Top right: $\alpha = 2$, $W = 4$. Bottom left: $\alpha = 2, W = 2$. Bottom right: $\alpha = 1, W = 4$.

TABLE V
ROOT MEAN SQUARE RECONSTRUCTION ERRORS FOR THE NUMERICAL MRI PHANTOM AND *IN VIVO* CARDIAC MRI RADIAL ACQUISITIONS PRESENTED IN FIG. 6 AND 7. IMAGES WERE RECONSTRUCTED BY THE $\mathrm{NFFT}^{\mathrm{H}}$ ALGORITHM (GRIDDING) WITH VARYING OVERSAMPLING FACTORS (α) AND KERNEL WIDTHS (W) USING A GAUSSIAN KERNEL ON THE CPU AND GPU, RESPECTIVELY

$\mathrm{NFFT}^{\mathrm{H}}$ - RMS reconstruction error

		Numerial phantom	In vivo MRI
α = 1 W = 2	CPU	**2.441e-01**	**5.581e-02**
	GPU	**2.198e-01**	**5.544e-02**
α = 1 W = 4	CPU	**2.071e-01**	**4.944e-02**
	GPU	**2.073e-01**	**4.993e-02**
α = 2 W = 2	CPU	**3.045e-02**	**1.562e-02**
	GPU	**2.990e-02**	**1.555e-02**
α = 2 W = 3	CPU	**9.349e-03**	**5.353e-03**
	GPU	**8.863e-03**	**5.351e-03**
α = 2 W = 4	CPU	**2.881e-03**	**1.561e-03**
	GPU	**2.914e-03**	**1.559e-03**

Again, the larger αW, the larger the acceleration factor. Another general observation is that it pays off computing the convolution kernel online, particularly when using a Gaussian kernel. Even though the difference is marginal only for the Kaiser–Bessel kernel it is still advantageous as it increases the arithmetic intensity of the algorithm. It did not pay off computing the spiral trajectories online however. From this we can conclude that it is advantageous to compute online only the simplest trajectories, e.g., equidistant radial sampling.

The FFT and deapodization steps generally experienced a modest speedup of 1.5–6.0 at image sizes 256^2 and 512^2. The FFT however did halve its performance at resolution 128^2. This is the result of a severely memory bound implementation of the FFT adopted directly from [24] without much optimization. It would be worth investigating a more efficient GPU implementation of the FFT as it is now the limiting factor in the reconstruction. Furthermore, since the FFT implementation is memory bound it will likely be even more dominant on future generations of hardware. As previously stated this work was implemented using the programming API CTM [19]. Recently, an alternative has appeared, namely CUDA [20]. CUDA ships with a highly optimized GPU implementation of the FFT that outperforms our implementation with at least a factor of 5. This shows that it is indeed possible to implement a faster FFT on a commodity GPU.

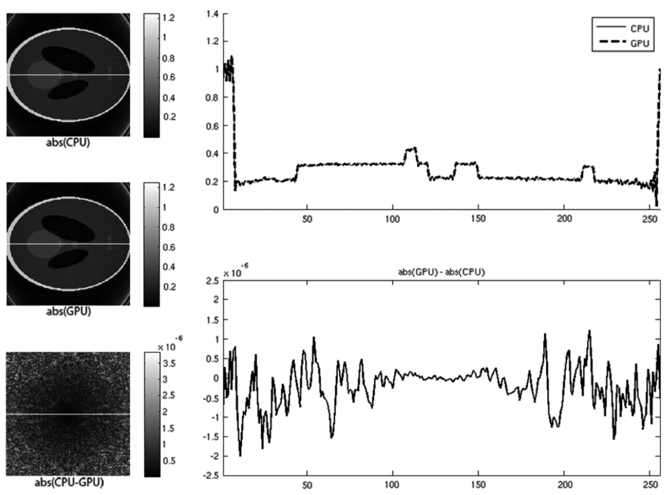

Fig. 8. Reconstruction differences between the CPU and the GPU. Left: CPU reconstruction (top), GPU reconstruction (middle), and difference image (bottom). Right: Image profiles along the grey horizontal line in the leftmost images (top) and corresponding image differences (bottom).

On the CPU the convolution operation constitute 80%–95% of the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ execution times ($\alpha = 2$, $W = 4$). On the GPU on the other hand, our proposed convolution algorithm is between two and four times as fast as our FFT. Hence, the FFT is now the limiting component of the algorithms. For the best overall performance it is important to balance the cost of the convolution and the FFT. This can be achieved by reducing the oversampling factor while increasing the kernel size. In [3], was shown that this can be done while maintaining the maximum aliasing amplitude for a given kernel. Table IV shows a rather exaggerated example setting $\alpha = 2/W = 3$ and $\alpha = 1/W = 4$, respectively. The convolution acceleration factor is lower for the latter case as αW takes the lowest value. However, the overall acceleration factor is maintained due to the shift in balance between the convolution and the FFT. The overall acceleration factor would even have increased had the kernel size been chosen such that $W > 4$ in the latter case.

GPUs currently operate with single precision floating point values (support for double precision has been announced for future generations of hardware). We must emphasize that the CPU reference library operates on double precision numbers only. Unfortunately, it is not straightforward to modify the code to use single precision floating point values instead. Since all CPU speed measurements were performed on a 64-bit machine however, the execution time differences between single and double precision is limited.

A quantitative evaluation of the accuracy of our GPU $\mathrm{NFFT}^{\mathrm{H}}$ implementation was included in Table V showing no significant differences between the CPU and GPU reconstruction errors for any tested combination of oversampling factors and kernel widths. The marginally better GPU reconstructions are likely due to the fact that kernel weights are computed based on actual distances and not read from a discrete lookup table. From Fig. 8, it can be concluded that the error increases with the distance from the center. This is due to the division by the deapodization filter that has smaller values the further the distance from the center. When choosing a sufficient oversampling factor and kernel size the rms error is very low and shows that single precision reconstruction is adequate in the examples chosen here. It is perceivable however that if the MRI data was acquired with more meaningful bits (e.g., in a 3-D acquisition) that single precision would prove inadequate. Consequently, care should be taken to test that single precision reconstruction is sufficient for any new type of input data. From the *in vivo* rms errors, it is also evident that the reconstruction error is well below the general noise level for most of the selected reconstruction settings.

To utilize the GPU convolution algorithm, a small initial cost to setup our data structures prior to the reconstruction is necessary. For non-real-time scans, this raises no concern as the data structures can simply be computed prior to or during the acquisition. For real-time applications, we must assume that the data structure remains constant during acquisition and reconstruction to be reused over and over. Fortunately, the imaging plane position and orientation can be chosen freely without changing the data structure as the trajectories of the sample points are (or can be) expressed in coordinates relative to the imaging plane axes. However, changing certain acquisition paramters (e.g., the field-of-view) or reconstruction parameters (e.g., α and W) requires recomputation of the data structure. If a specific application cannot tolerate a brief stall in the reconstruction due to changing one such parameter, we suggest precomputing the data structure for a number of predefined settings, which can then to be stored on the host computer and uploaded to the GPU when required. This compares somewhat to previous CPU strategies for precomputing convolution kernel values per sample for a given set of trajectories [29]. We have not attempted any optimization of our preprocessing step but report a few observed running times nonetheless. For data containing 128^2, 256^2, or 512^2 samples the preprocessing costs are 0.1s/0.04s, 0.4s/0.2s, and 1.4s/0.9s for the NFFT and $\mathrm{NFFT}^{\mathrm{H}}$ algorithms, respectively.

Previous implementations of the $\mathrm{NFFT}^{\mathrm{H}}$ on the GPU have been restricted to the approach sketched to the right in Fig. 3 due to restrictions in OpenGL and DirectX [26]. Consequently only a four-fold increase in performance compared to the CPU could be obtained. The dedicated programming APIs for general purpose computation on GPUs that have recently emerged have fortunately removed most of the restrictions that have previously been hindering effective, compute bound algorithms in being implemented [19], [20]. The work presented in this paper thus constitutes one of the first algorithms to take advantage of these new opportunities. To verify that we are indeed compute bound in the proposed convolution algorithm we repeated some of our measurements at GPU memory and clock settings running at half speed. This confirmed our hypothesis since the speed of the convolution was reduced by up to 90% when decreasing the clock speed 100%, whereas reducing memory speed to half resulted in down to a 5% reduction in performance only.

In our convolution implementation, we were somewhat restricted by the DirectX 9 FXC compiler in that it only allows compilations of programs that will result in assembly code consisting of no more than 32 temporary registers. The GPU used in this work supported many more registers however. It will be interesting to get around this limitation in the future, as we most likely could not choose the optimal number of grid cells to associate each processor due to this restriction. A related discussion is the impact of nonuniformity of the sampling density to convolution speeds. Areas of high sampling density (for MRI typically the center of *k*-space) take longer to process than areas of low sampling density. This could reduce performance if some processors were partly stalled due to the lock stepping SIMD nature of the GPU. Although beyond the control of the GPU programmer, the GPU does minimize this problem by enforcing the SIMD model only subsets of its computational domain. This property suggests that it might be advantageous to limit each processors domain size somewhat.

Modern GPUs now offer affordable "parallel computers" providing a much higher "computational power per $" ratios as CPU clusters or multicore CPUs. At the same time, the introduction of dedicated programming "interfaces" for general purpose computation on GPUs from both major hardware vendors has removed the necessity of knowing about computer graphics before migrating to this new platform. GPGPU is thus expected to obtain an important role in many "time conscious" scientific computing applications in the future.

We conclude this paper with a brief discussion of the potential clinical impact of our results. When using gridding for real-time reconstruction of radial or spiral MRI we can now reconstruct acquisitions of 256^2 and 512^2 samples with an oversampling factor $\sigma = 2$ and kernel width $W = 4$. Acquisitions containing 512^2 samples went from reconstruction rates of 1 frame per second (fps) to 15 fps, i.e., a transition from non real-time to real-time reconstruction. For 256^2 samples we moved from 5 to 70 fps, i.e., from near real-time reconstruction to real-time reconstruction from multiple receiver coils. Finally, the GPU based gridding and inverse gridding algorithms hold a huge potential for many iterative reconstruction algorithms in non-Cartesian imaging, e.g., SENSE [5], $k-t$ BLAST and $k-t$ SENSE [6], and [7]. Most applications of these algorithms are not currently being used routinely in clinical settings due to considerable reconstruction times. As each iteration contains both a gridding and an inverse gridding step, which together constitute the main reconstruction time, the presented work can significantly speed up these reconstructions and hopefully make them clinically feasible.

ACKNOWLEDGMENT

The authors would like to thank Dr. O. Østerby of the Department of Computer Science at the University of Aarhus, Denmark for valuable discussion and feedback.

REFERENCES

[1] 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, Sep. 1995.

[2] S. Schaller, T. Flohr, and P. Steffen, "An efficient Fourier method for 3-D radon inversion in exact cone-beam CT reconstruction," *IEEE Trans. Med. Imag.*, vol. 17, no. 2, pp. 244–250, Apr. 1998.

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

[4] J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a convolution function for fourier inversion using gridding," *IEEE Trans. Med. Imag.*, vol. 10, no. 3, pp. 473–478, Sep. 1991.

[5] K. P. Pruessmann, M. Weiger, P. Bornert, and P. Boesiger, "Advances in sensitivity encoding with arbitrary k-space trajectories," *Magn. Reson. Med.*, vol. 46, no. 4, pp. 638–651, Oct. 2001.

[6] M. S. Hansen, C. Baltes, J. Tsao, S. Kozerke, K. P. Pruessmann, and H. Eggers, "k-t BLAST reconstruction from non-Cartesian k-t space sampling," *Magn. Reson. Med.*, vol. 55, no. 1, pp. 85–91, Jan. 2006.

[7] B. P. Sutton, D. C. Noll, and J. A. Fessler, "Fast, iterative image reconstruction for MRI in the presence of field inhomogeneities," *IEEE Trans. Med. Imag.*, vol. 22, no. 2, pp. 178–188, Feb. 2003.

[8] D. Potts, G. Steidl, and M. Tasche, "Fast Fourier transforms for nonequispaced data: A tutorial," in *Modern Sampling Theory: Mathematics and Applications*, J. J. Benedetto and P. Ferreira, Eds. Boston, MA: Birkhäuser, 2001, pp. 249–274.

[9] J. Keiner, S. Kunis, and D. Potts, NFFT 3.0—Tutorial 2007.

[10] K. Fourmont, "Non-equispaced fast fourier transforms with applications to tomography," *J. Fourier Anal. Appl.*, vol. 9, no. 5, pp. 431–450, 2003.

[11] J. A. Fessler and B. P. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation," *IEEE Trans. Signal Process.*, vol. 51, no. 2, pp. 560–574, Feb. 2003.

[12] A. Dutt and V. Rokhlin, "Fast fourier-transforms for nonequispaced data," *SIAM J. Sci. Comput.*, vol. 14, no. 6, pp. 1368–1393, Nov. 1993.

[13] G. Beylkin, "On the fast fourier-transform of functions with singularities," *Appl. Computational Harmonic Anal.*, vol. 2, no. 4, pp. 363–381, Oct. 1995.

[14] A. F. Ware, "Fast approximate fourier transforms for irregularly spaced data," *SIAM Rev.*, vol. 40, no. 4, pp. 838–856, 1998.

[15] J. W. Cooley and J. W. Tukey, "An algorithm for machine calculation of complex fourier series," *Math. Computation*, vol. 19, no. 90, p. 297, 1965.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[17] R. Strzodka, M. Doggett, and A. Kolb, "Scientific," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 667–681, 2005.

[18] T. S. Sorensen and J. Mosegaard, *An Introduction to GPU Accelerated Surgical Simulation*. Berlin, Germany: Springer, 2006, vol. 4072, pp. 93–104.

[19] M. Segal and M. Peercy, A performance-oriented data parallel virtual machine for GPUs 2006, p. 184.

[20] NVIDIA Corporation, NVIDIA CUDA: Compute unified device architecture—Programming guide 2007, pp. 1–83.

[21] M. Pharr, *GPU Gems II*. Boston, MA: Addison-Wesley, 2005.

[22] I. Buck, K. Fatahalian, and P. Hanrahan, "GPUBench: Evaluating GPU performance for numerical and scientific application," in *ACM Workshop General Purpose Computing Graphics Processors*, Los Angeles, CA, 2004, p. C-20.

[23] K. Moreland and E. Angel, "The FFT on a GPU," in *Proc. SIGGRAPH/Eurographics Workshop Graphics Hardware*, 2003, pp. 112–119.

[24] T. Sumanaweera and D. Liu, "Medical Image Reconstruction with the FFT," in *GPU Gems II*, M. Pharr, Ed. Boston, MA: Addison-Wesley, 2005.

[25] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha, A memory model for scientific algorithms on graphics processors Univ. North Carolina, Chapel Hill, Tech. Rep. 108, 2006.

[26] T. Schiwietz, T.-C. Chang, P. Speier, and R. Westermann, "MR Image Reconstruction Using the GPU," in *Proc. SPIE*, 2006, vol. 6142, pp. 1279–1290.

[27] K. Gray, Microsoft DirectX 9 Programmable Graphics Pipeline 2003.

[28] M. T. Vlaardingerbroek and J. A. den Boer, *Magnetic Resonance Imaging Theory and Practice*, 3rd ed. New York: Springer, 2003.

[29] B. Dale, M. Wendt, and J. L. 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, Mar. 2001.