MR Image Reconstruction Using the GPU

Thomas Schiwietz[a,c], Ti-chiun Chang[a], Peter Speier[b], Rüdiger Westermann[c]

[a]Siemens Corporate Research, 755 College Road East, Princeton, NJ 08540, USA
[b]Siemens AG Med, Karl Schall Str. 6, 91052 Erlangen, Germany
[c]Technische Universität München, Computer Graphics & Visualization Group, Boltzmannstrasse 3, 85748 Garching, Germany

ABSTRACT

Magnetic resonance (MR) image reconstruction has reached a bottleneck where further speed improvement from the algorithmic perspective is difficult. However, some clinical practices such as real-time surgery monitoring demand faster reconstruction than what is currently available. For such dynamic imaging applications, radial sampling in k-space (i.e. projection acquisition) recently revives due to fast image acquisition, relatively good signal-to-noise ratio, and better resistance to motion artifacts, as compared with the conventional Cartesian scan. Concurrently, using the graphic processing unit (GPU) to improve algorithm performance has become increasingly popular. In this paper, an efficient GPU implementation of the fast Fourier transform (FFT) will first be described in detail, since the FFT is an important part of virtually all MR image reconstruction algorithms. Then, we evaluate the speed and image quality for the GPU implementation of two reconstruction algorithms that are suited for projection acquisition. The first algorithm is the look-up table based gridding algorithm. The second one is the filtered backprojection method which is widely used in computed tomography. Our results show that the GPU implementation is up to 100 times faster than a conventional CPU implementation with comparable image quality.

Keywords: MR Image reconstruction, Radial Trajectory, Gridding, Filtered Backprojection, GPU, FFT

1. INTRODUCTION

Fast MR image reconstruction has become more and more important for real-time applications such as interventional imaging. An imaging speed of at least 5 frames per second is necessary in order to provide immediate feedback to the physicians. This motivates faster image acquisition and reconstruction. In MR imaging (MRI), raw data measured from the scanner correspond to the Fourier coefficients of the target image; and the Fourier space is referred to as k-space. Traditional MRI acquires k-space samples on a Cartesian grid. This scan pattern is relatively slow in the phase encoding direction.[1] Fortunately, higher acquisition speed can be achieved by using a non-Cartesian scan in k-space such as radial or spiral trajectories. In this paper, we focus on radial trajectories which offer the following advantages over the Cartesian scan. Due to their high sampling density near the center of k-space (i.e. low frequency components), the reconstructed images achieve relatively high signal to noise ratio (SNR) and contrast. They are also less sensitive to motion and flow; and this prevents from producing unacceptable artifacts. Finally, they allow the scan of tissues or objects with very short T2.

A popular technique to reconstruct images from non-Cartesian trajectories in k-space is the so-called gridding method.[2,3] This is the method of choice, among various possible MR image reconstruction methods such as iterative approach[4] and pseudoinverse calculation,[5] if one considers high computation efficiency with reasonable reconstructed image quality. The basic idea of gridding is to resample the raw measurement data on the Cartesian grid. Then, the fast Fourier transform (FFT) is used to reconstruct the target image. An important procedure called the density compensation is necessary in the very beginning to account for the non-uniform sampling. We

Further author information: (Send correspondence to Thomas Schiwietz)
Thomas Schiwietz: Thomas.Schiwietz@siemens.com
Ti-Chiun Chang: Ti-chiun.Chang@siemens.com
Peter Speier: Peter.Speier@siemens.com
Rüdiger Westermann: westermann@in.tum.de

will review the gridding method in Sec. 4.1.1. Of particular interest, Dale et al.[6] proposed a fast implementation which exploits table look-up operations. This approach is the foundation on which our implementation is based. Given a reasonable window size for interpolation (usually 3×3 or 5×5), this algorithm provides acceptable image quality with high computational efficiency. However, on currently available MR image reconstruction hardware, this algorithm is still a performance bottleneck for real-time imaging applications.

To improve the reconstruction speed, we can take advantage of the computational power of modern graphics processing units (GPUs). Although GPUs are not Turing-complete, they provide a powerful subset for parallel single instruction multiple data (SIMD)-type operations which outperform current CPUs even with streaming SIMD extension (SSE) 2 or 3 optimization. Furthermore, there is a second type of parallelism available on modern graphics cards. Today there are 16 to 24 pixel units on board executing SIMD operations in parallel. To better understand the architecture and advantages of using the GPU, we will present a brief overview of GPU programming in Sec. 2.

A core technique for MR image reconstruction is the implementation of the FFT. For the CPU-based FFT, the FFTW library[7] is claimed to be the fastest implementation using all possible optimizations like SSE2/3 and hyper threading. It will be used as a reference to compare with our results. For the GPU-based FFT, Moreland et al.[8] presented the first implementation with the performance being slower than the FFTW library. Schiwietz et al.[9] and Jansen et al.[10] presented different implementations with the performance being comparable to that of the FFTW. In our current experiment, we accomplish the fastest FFT implementation among all mentioned above. This is an important factor for achieving very fast MR image reconstruction.

To the best of our knowledge, there is virtually no work in the literature investigating GPU-based MR image reconstruction for the case of projection acquisition. The most relevant work is presented by Thilaka et al.,[11] who described one variation of GPU FFT implementations and applied it to MR image reconstruction for the case of Cartesian sampling. Interestingly, the GPU implementation of the filtered backprojection algorithm has been more widely investigated in the computed tomography (CT) literature.[12] In our work, we also implement the filtered backprojection technique as the radial samples can be also treated as projection data, thanks to the central slice theorem[1].[13] We compare the performance of our GPU-based filtered backprojection and gridding algorithms with that of the same algorithms implemented on the CPU. Our results show that the proposed implementations outperform the CPU-based implementation by a factor of about 100 in speed.

It is worth noting that our work enables new real-time applications for MR parallel imaging. Real-time MRI always strives for higher frame rates. MR image acquisition can be sped up further by using multiple receivers and exploiting the differences in their spatial sensitivity profiles to create images from only partially sampled k-space.[14] However, the algorithms for reconstructing these data sets are also quite computational intensive and pose another bottleneck for real-time MR image reconstruction. With existing commercial hardware, either parallel image reconstruction[15] (GRAPPA[16] in particular), or gridding can barely be achieved in real-time, but not both together. Moving just the FFT and gridding steps to the GPU thus frees enough computational resources on the CPU to implement real-time radial GRAPPA MR image reconstruction on commercially available hardware.

The remainder of this paper is organized as follows. We present an overview of GPU programming in Sec. 2. In Sec. 3, we derive an efficient GPU implementation of the FFT following the approach of Schiwietz et al..[9] Then, a GPU implementation of an gridding algorithm and filtered backprojection is described in Sec. 4. Experimental results are shown in Sec. 5. Finally, we conclude our work in Sec. 6.

2. GPU PROGRAMMING

As we will show in Sec. 4, using a graphics processor for MR image reconstruction provides superior computational power as compared with that of the CPU. We briefly review the use of the GPU as a co-processor for arbitrary algorithms, known as *general-purpose* GPU (*GPGPU*) programming. For more a comprehensive introduction, we refer the readers to the books[11, 17–19] and to the website.[20]

We first describe how graphics cards handle data storage. Graphics cards provide readable and writable data structures in GPU memory. Basically, there are three types of data structures available: 1D, 2D, and 3D arrays, all referred to as *textures*. Among them, 2D textures provide the fastest update performance. The array elements

of a texture are called *texels*. Each texel can have up to four float-typed components. The four components are often called the *RGBA* channels, as they are originally used to represent the red, green, blue, and alpha intensities of a color for rendering.

To set values in a texture, the GPU processing *pipeline* consisting of several stages is utilized. This procedure is also referred to as "rendering" which is carried over from the original convention when screen pixels are drawn by the pipeline. Figure 1 illustrates three important stages in the graphics card pipeline, which we elaborate below.

- *Vertex shader*: A stream of *vertices* enters the vertex shader stage. A vertex is a data structure on the GPU with attributes such as position vectors for certain coordinate systems. In this stage, vertex attributes are manipulated according to the objectives of the applications. Traditionally, user-defined vertex shader programs are used to transform position vectors from one space to another. In one instance of our work, vertices are used to address the k-space coordinates.
- *Geometry shader (triangle setup)*: In this stage, sets of three vertices are selected to setup triangles according to a predefined GPU state. These triangles will be used by the next pipeline stage.
- *Rasterization / Pixel shader*: Using the three vertices of each triangle defined above as knot points, this rasterization stage bilinearly interpolates all vertex attributes for the texels circumscribed by this triangle. This is done by the hardware and is highly computational efficient. We use this mechanism to setup a coordinate system on the triangles addressing k-space and indexing the measurement data. User-defined pixel shader programs are executed for each rasterized texel, which can access and work with the interpolated vertex attributes. The following types of instructions are available in this stage.
  - *Arithmetical instructions* perform addition, multiplication, exponent, etc.
  - *Data access instructions* allow reading values from GPU memory.
  - *Control flow instructions* allow branching and loops.

  Each pixel is rasterized and processed independently and potentially in parallel. However, no processing order can be assumed by the programmer. The return value of a pixel shader program is a four-component vector, which is explained in detail next.

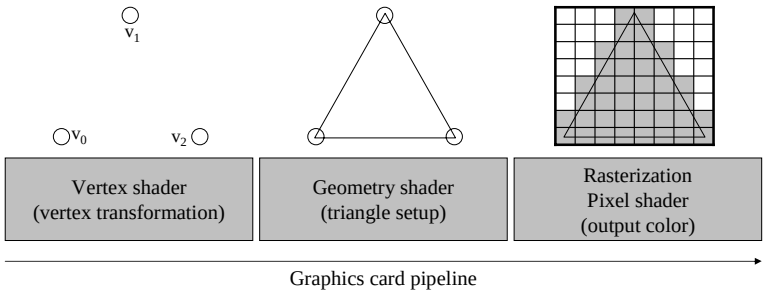


Figure 1. The graphics card pipeline with the three most important stages: the vertex shader, the geometry shader and the rasterizer/pixel shader.

Particularly, the texture to be updated is set as the *render target*. This implies that the pixel shader now writes values to the texture instead of the screen. Usually, a quadrilateral which is defined by two triangles and covers all texels to be updated is processed according to the graphics card pipeline above. In this way, a pixel shader program is executed for each texel in the target texture. With this procedure, arbitrary calculations can be performed for each texel. In the context of MR image reconstruction, the measurement data and the k-space are represented as textures and updated using pixel-shader programs.

Currently, two major graphics programming APIs are *DirectX* and *OpenGL*. Both APIs provide similar functionality for graphics cards programming. Additionally, there are so called *shading languages* to program the vertex and pixel shader units on the graphics card. The DirectX shading language is called high-level shading language (*HLSL*); and the one for OpenGL is OpenGL shading language (*GLSL*). Both languages provide similar functionality and their syntax is closely related to the C language. Our implementations have been written in DirectX with HLSL.

3. GPU-BASED FAST FOURIER TRANSFORM

One important algorithm that is common to virtually all MR image reconstruction algorithms is the Fast Fourier Transform (FFT). In this section we review the deviation of the FFT from the DFT as well as an efficient GPU implementation. A performance comparison of our implementation with the FFTW library[7] is presented in Sec. 5.

3.1. Theory

The discrete Fourier Transform (DFT) $F(n)$ of a discrete signal $f(k),\ k = 0, \cdots, N-1$ is expressed as a superposition of complex sinusoids

$$F(n) = \sum_{k=0}^{N-1} f(k) e^{\frac{-i2\pi kn}{N}}, \tag{1}$$

where $2\pi n/N,\ n = 0, \cdots, N-1$ are the sampled frequencies in the Fourier plane.

Direct computation of the DFT has complexity $O(N^2)$. The FFT reduces the numerical complexity of the DFT to $O(N\,log\,N)$. As described by Brigham,[21] Eq. (1) can be written as a matrix-vector product

$$\begin{pmatrix} F(0) \\ F(1) \\ F(2) \\ \vdots \\ F(N-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2N-2} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(N-1) \end{pmatrix} \tag{2}$$

where $\omega^k = e^{\frac{-i2\pi k}{N}} = \cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right)$; and the $N \times N$ matrix is referred to as the DFT matrix.

Due to periodicity, we have $\omega^{nk} = \omega^{n[k]_N}$, where $[k]_N$ denotes k mod N. Consecutively using this identity and matrix shuffling (column-wise permutation) that groups the even and odd columns, the DFT matrix in Eq. (2) can be split into a chain of $log\,N$ sparse matrices,[21] referred to as the FFT matrices. Each row of any FFT matrix contains exactly two non-zero entries, with one entry being always 1 which need not to be stored explicitly. We illustrate this factorization for a four-element input signal

$$\begin{pmatrix} F(0) \\ F(2) \\ F(1) \\ F(3) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ 0 & 0 & 1 & \omega^1 \\ 0 & 0 & 1 & \omega^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & \omega^2 & 0 \\ 0 & 1 & 0 & \omega^2 \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{pmatrix}. \tag{3}$$

Finally, the output signal has to be rearranged to yield the Fourier coefficients in the right order. The 2D FFT can be computed by consecutive 1D FFTs first along the rows (columns) and then along the columns (rows). Referring to Eq. (3), for a 2D signal of size $N \times N$, the FFT can be completed by exactly $2N^2 \cdot log\,N$ multiplications.

Table 1. Four-component texture layout for the precomputed FFT table for the first matrix on the right hand side of Eq. (3). Here, ω_r and ω_i denote, respectively, the real and imaginary parts of ω defined in Eq. (2).

$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} \omega_r^2 \\ \omega_i^2 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} \omega_r^1 \\ \omega_i^1 \\ 3 \\ 2 \end{pmatrix}$	$\begin{pmatrix} \omega_r^3 \\ \omega_i^3 \\ 3 \\ 2 \end{pmatrix}$

3.2. GPU Implementation

With the particular structure of the FFT matrices, we employ a special representation on the GPU. Every FFT matrix is represented as a 1D RGBA texture which contains, in the i-th component, the value of the complex non-zero entry in the i-th row of the matrix and the absolute positions (with index starting from 0) of both non-zero entries in this row. As an example, Table 1 shows the texture content for the first matrix on the right hand side of Eq. (3). The real and imaginary components are stored in the R and G channels, and the two position indices in the B and A channels of a four component texture.

The 1D signal to be transformed is stored in one line of a 2D texture with the real part in the R channel and the imaginary part in the G channel. In this way, a stack of 1D signals can be transformed efficiently. To perform the matrix-vector multiplication at a particular FFT stage, a quadrilateral defined by four vertices covering $N \times N$ pixels is rendered. This quadrilateral, along with the 2D texture containing the signal and the 1D texture containing the FFT table, are used as input parameters to the shader program (see Figure 2).

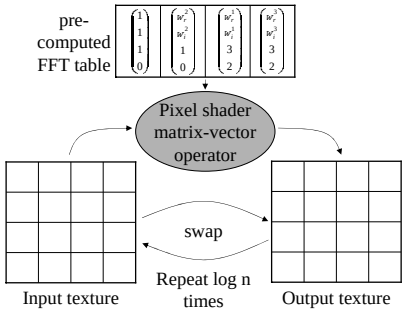


Figure 2. The FFT shader samples the FFT matrix entries of a specific stage. Then, it samples the two vector components indexed by the table and performs the butterfly operation. Input and output textures are swapped in each stage for efficient memory usage.

To avoid switching to a different FFT table in every FFT stage, all these textures are combined into one single 2D texture. The respective row to be accessed in each stage is specified in a constant parameter to the shader program. In every pass, we swap the roles of the *input texture* and the *output texture* as illustrated in Figure 2. After $\log N$ passes, the row-wise FFTs have been computed, and the results have to be reordered using a reorder texture. This texture stores an index to the untangled position of each vector component. A pixel shader program queries the reorder texture to get the correct position of the vector component. Finally, a 2D transformation can be achieved by a consecutive column-wise 1D transformation.

The FFT performance can be doubled if a second data set of the same size is available, since both sets can be transformed in parallel by storing the second set in the BA channels and by using vector instructions on the GPU. This effectively reduces in half the number of texture fetches and arithmetic instructions. With this highly efficient FFT implementation, the next section describes reconstruction approaches and the implementation on the GPU.

4. RECONSTRUCTION FROM RADIAL SAMPLES IN *K*-SPACE

Measurement data from MRI scanners can be nicely interpreted as complex-valued samples in the frequency space, more commonly referred to as k-space in the MR literature. The switching patterns of the magnetic field gradients applied during the measurement determine the sampling trajectories in k-space.[22] Conventional MRI adopts Cartesian trajectories as illustrated on the left of Figure 3. When the entire k-space is sampled in this fashion, the image is reconstructed by a straight forward application of the FFT to the measurement data. Other trajectories such as radial, spiral, and Lissajou are possible and generally called non-Cartesian or non-uniform sampling. In this work, we concentrate on the radial trajectories, which is shown on the right of Figure 3.

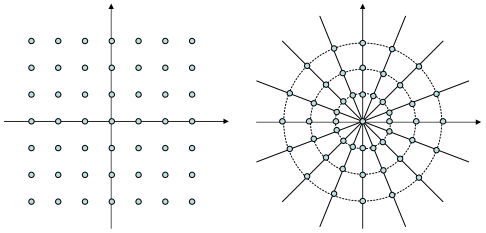


Figure 3. Cartesian and radial sampling trajectories in k-space.

A popular approach to reconstruct MR images from non-Cartesian k-space signals is the gridding method. In this section, we will give a brief review of this method and its GPU implementation. Because radial trajectories in k-space can also be considered as projection data according to the central slice theorem[1],[13] we will also present the filtered backprojection method, again with both a brief review and the GPU implementation.

4.1. The Gridding Algorithm

4.1.1. Overview

Gridding algorithms[2, 3] are computationally efficient for reconstructing images from MR measurement data that have arbitrary nonuniform trajectories. This class of algorithms consist of four major steps.

1. Density compensation: To account for nonuniform sampling in k-space, each sample point should be multiplied by a compensation factor. There are several algorithms aim at approximating the optimal density compensation functions.[23, 24]

2. Interpolation: Each sample point contributes, according to a certain interpolation window such as Kaiser-Bessel window,[3] to the neighboring Cartesian coordinates. In this step, one can choose to oversample the original k-space data and make the grid denser than the original. This oversampling can reduce aliasing significantly and allows the use of smaller interpolation kernels.

3. Inverse FFT: After the interpolation step, a standard inverse FFT is used to generate a reconstructed image.

4. Deapodization: This is a postprocessing operation to compensate for the non-ideal roll-off (non-square) of the frequency responses of the interpolation kernels. This step is sometimes referred to as "roll-off correction".

We describe the implementation of a look-up table based gridding algorithm in the following subsection.

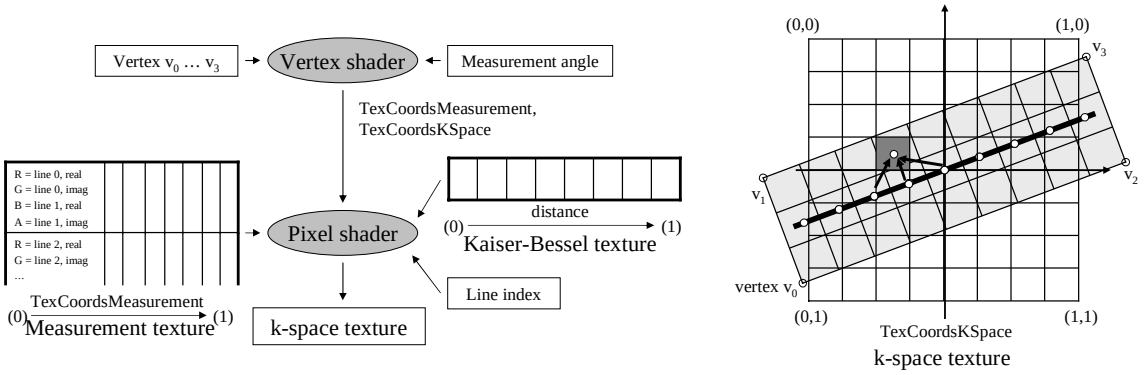


Figure 4. Left: An overview of our GPU gridding implementation. Right: A $N \times 3$ pixels quadrilateral defined by the vertices $v_0 ... v_3$ is rotated by the vertex shader according to the measurement angle. Two coordinate systems are defined: one addressing k-space (*TexCoordsKSpace*), the other addressing the measurement data (*TexCoordsMeasurement*). The pixel shader performs a grid-driven interpolation from the three nearest measurement samples with weights computed according to the density compensation function and the distance.

4.1.2. GPU implementation using a Table-based Kaiser-Bessel Function

As described by Dale *et al.*,[6] the table-based Kaiser-Bessel window gridding algorithm distributes each measurement sample on a radial line to a small area (3×3) around the nearest cell on the Cartesian grid. Contrarily, our GPU implementation gathers surrounding measurement samples for each Cartesian grid cell since distribution operations are not supported by the current generation of GPUs.

In our work, all GPU textures are float-typed. The measurement data is stored in the *measurement texture*. All measurement lines are stacked up into the 2D texture. Each measurement sample is complex-valued, therefore two channels are required to stored the signal. In order to increase memory efficiency, even-numbered measurement lines are stored in the RG channels and odd ones are stored in the BG channels. The gridded results in the Cartesian k-space is represented in the k-*space texture*. Again, two channels are required to store this complex-valued signal. The remaining two channels can be used to represent another set of the gridded results. This is useful when there are several measurement channels as two sets of data can be reconstructed in parallel.

Conceptually, we map the radial measurement line with a thickness of three pixels to the Cartesian grid. This is illustrated by the skewed light gray box on the right hand side of Figure 4. Then, for each covered Cartesian grid cell (k_x, k_y), we compute the contribution $c_i(k_x, k_y)$ from the three nearest measurement samples $m_i(k_x, k_y)$ located at $\mathcal{N}_i(k_x, k_y)$, where $i = 0, 1, 2$. This procedure is exemplified by the grid cell colored with the dark gray in Figure 4 and explained below.

- The density compensation factor $\rho(D_{\mathcal{N}_i}(k_x, k_y))$, which is inversely proportional to the distance $D_{\mathcal{N}_i}(k_x, k_y)$ between $\mathcal{N}_i(k_x, k_y)$ and the k-space origin, is multiplied with $m_i(k_x, k_y)$.
- Then, we use the distance $d_{m_i}(k_x, k_y)$ between $\mathcal{N}_i(k_x, k_y)$ and (k_x, k_y) to look up the precomputed Kaiser-Bessel table and obtain the weighting coefficient $w(d_{m_i}(k_x, k_y))$ so that we have

$$c_i(k_x, k_y) = w(d_{m_i}(k_x, k_y)) \cdot \rho(D_{\mathcal{N}_i}(k_x, k_y)) \cdot m_i(k_x, k_y). \tag{4}$$

In our GPU implementation, we store the Kaiser-Bessel lookup table in a 1D texture (*Kaiser-Bessel texture*) as it is computationally inefficient to evaluate the Kaiser-Bessel function on the fly. This table is precomputed at program initialization. Furthermore, we provide the measurement and k-space texture as input data to the

shader program. As the measurement data is stacked up in a 2D texture, a *line index* parameter is passed as a constant to the shader program in order to index a specific measurement line in the stack.

A quadrilateral covering $N \times 3$ pixels is rotated by a vertex shader program according to the measurement angle in order to rasterize the correct Cartesian grid cells in k-space. Two coordinate system are required in the pixel shader later. One coordinate system addresses the Cartesian grid of k-space, called the *TexCoordsKSpace*. The other coordinate system addresses along the radial measurement line, called the *TexCoordsMeasurement*. Both coordinate systems are fixed at each vertex as attributes and interpolated in each Cartesian cell bilinearly.

The gridding is performed in the pixel shader program. The pixel shader gets the interpolated coordinates TexCoordsMeasurement and TexCoordsKSpace. Using TexCoordsMeasurement, we can compute $D_{\mathcal{N}_i}(k_x, k_y)$ and thus $\rho(D_{\mathcal{N}_i}(k_x, k_y))$ for each Cartesian grid cell. Next, $\mathcal{N}_i(k_x, k_y)$ is evaluated and $d_{m_i}(k_x, k_y)$ is calculated. The distance $d_{m_i}(k_x, k_y)$ is used to index the Kaiser-Bessel texture to retrieve $w(d_{m_i}(k_x, k_y))$. Finally, the measurement sample $m_i(k_x, k_y)$ indexed by TexCoordsMeasurement is weighted to obtained $c_i(k_x, k_y)$ as defined in Eq. (4). This procedure is repeated for $i = 0, 1$, and 2. Using the coordinate system TexCoordsKSpace, these $c_i(k_x, k_y)$ are accumulated into the k-space texture from different quadrilaterals that encompass (k_x, k_y). The overall contributions that a Cartesian grid cell (k_x, k_y) receives are

$$C(k_x, k_y) = \sum_{v \in \mathcal{R}(k_x, k_y)} \sum_{i=0}^{2} c_i^v(k_x, k_y), \tag{5}$$

where $\mathcal{R}(k_x, k_y)$ denotes the collections of radials lines that contain neighboring measurement samples of (k_x, k_y); and $c_i^v(k_x, k_y)$ is the i_{th} contribution from the v_{th} radial line. We note that since the blending operation is not supported for 32-bit floating by the current generation of graphics hardware, this step must be programmed explicitly.

After all measurement lines have been gridded, we use our FFT implementation to transform the gridded samples to obtain an image. Finally, the deapodization is applied to the image. We precompute the deapodization coefficients on the CPU and store them into a texture. Then, a pixel-wise multiplication of the image and the deapodization texture is performed on the GPU to render the final image.

4.2. Filtered Backprojection

4.2.1. Overview

According to the central slice theorem, a line passing through the k-space origin can be inversely Fourier transformed to obtain projection data in the image domain with the projection angle perpendicular to the k-space line. When the projection data is arranged in 2D with the angle being one axis and the projection position being the other, they are called a sinogram $s(\phi, x)$. Each radial line in our measurement data can be considered as the projection data $s(\phi_k, x)$ with a fixed angle ϕ_k. Therefore, filtered backprojection algorithms can also be used to reconstruct our MR images. Basically, each element in the sinogram $s(\phi, x)$ is a line integral of a given 2D signal along the angle ϕ at position x of the projection. This process is known as the Radon transform. Filtered backprojection, which consists of the following main steps, is a direct method to reconstruct the 2D signal.

1. High-pass/band-pass filtering: According to the derivation of the inverse Radon transform, a high-pass filter has to be applied to projection data. Several commonly used filters are *Ram-Lak*, *Shepp-Logan*, or *Hamming* filter. In our work, we simply use the frequency response of the Ram-Lak filter to multiply with each radial line in our measurement data. This procedure is exactly the same as the density compensation in our gridding implementation.

2. FFT: As the above filtering operation is performed directly in k-space, we then use 1D FFT to transform the filtered results to the projection data in the spatial domain.

3. Backprojection: Finally, the projection data with angle ϕ is backprojected onto the image domain. In this step, interpolation is necessary to obtain desirable values on the Cartesian grid. In our work, this is automatically done by the GPU using bilinear interpolation.

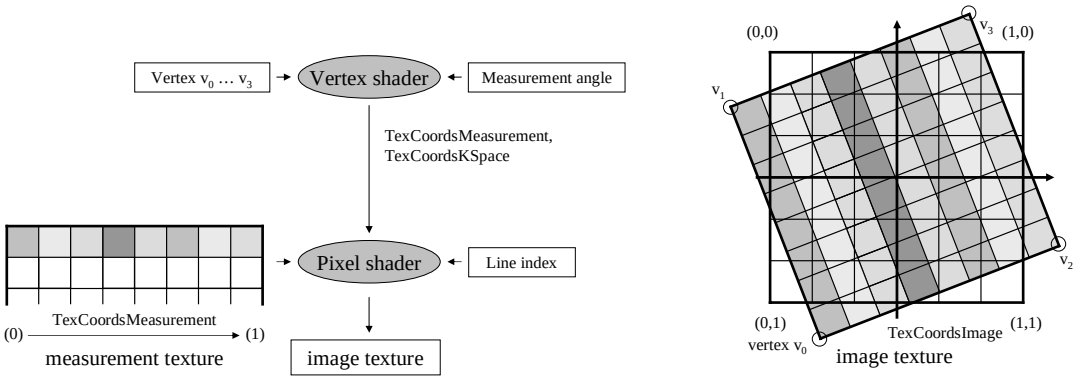


Figure 5. Left: An overview of our GPU backprojection implementation. Right: A quadrilateral covering the image domain is rotated by a vertex shader program according to the measurement angle. Two coordinate systems are defined: one addressing k-space (*TexCoordsKSpace*), the other addressing the measurement data (*TexCoordsMeasurement*). The pixel shader samples the measurement data at the interpolated TexCoordsMeasurement position and writes it to the interpolated TexCoordsKSpace position. The measurement line is repeated over the entire quadrilateral.

4.2.2. GPU implementation

We follow the procedures described in the previous subsection and point out important details of our GPU implementation.

1. High-pass/band-pass filtering: The k-space Ram-Lak filter is stored in a 1D floating point texture, called the filter texture, whose length is equal to that of a measurement line. A pixel shader program multiplies the measurement texture by the filter texture line by line.

2. FFT: The k-space measurement data is transformed to the spatial domain by our GPU-FFT implementation described in Sec. 3. The high-pass filtering and the transformation to the spatial domain is done for all available measurement lines at the same time, as the measurement lines are stacked up in a single 2D texture.

3. Backprojection: Figure 5 illustrates the backprojection implementation on the GPU. Four vertices $v_0 ... v_3$ setup a quadrilateral covering $N \times N$ pixels. The quadrilateral is rotated according to the measurement angle by a vertex shader program. Similar to the gridding implementation, two coordinate systems are required. The coordinate system *TexCoordsImage* addresses the image domain (*image texture*) for the accumulation of the previous backprojections. The other coordinate system *TexCoordsMeasurement* addresses k-space samples in the measurement texture. The backprojection is executed in a pixel shader program. The sinogram $s(\phi_k, x)$ corresponds to our measurement texture where ϕ_k is the *line index* and x is the *TexCoordsMeasurement*. One measurement line is reused inside the rotated quadrilateral to achieve the backprojection. This procedure is repeated for each measurement line. Similar to the implementation of the gridding algorithm, the backprojection results have to be accumulated. Using TexCoordsImage, the previous backprojected and accumulated values can be accessed and accumulated. Future generations of GPUs will provide a floating-point blending operator for this purpose.

This approach is easy to implement and very GPU friendly. No look-up tables or dependent textures fetches are required.

5. RESULTS

All our experiments were run under Windows XP on a P4 3.0 GHz processor equipped with an ATI Radeon X1800 XT graphics card.

5.1. Performance of FFT on the GPU

To verify the effectiveness of the proposed FFT implementation, we investigate its performance for different image sizes. In particular, we compare the performance of the GPU-FFT with that of the FFTW,[7] an efficient CPU implementation of the discrete FFT leveraging various acceleration strategies like SSE parallelization, cache optimization, and precomputed FFT tables. To conduct a fair comparison, the FFTW_MEASURE setting was enabled and the code was run in 32-bit floating point precision. In all our experiments, the time it takes to perform the FFT of a discrete complex 2D signal is measured. Table 2 essentially shows the GPU-FFT to be

Table 2. FFT performance in milliseconds.

	256^2	512^2	1024^2
FFTW	2	15	66
GPU-FFT	2	8	38

able to process even high resolution images at interactive rates. The performance doubles instantly as the two complex signals are stored in one RGBA texture. With these two transformations conducted in parallel, our implementation is clearly superior to that of the FFTW library.

5.2. MR image reconstruction on the GPU

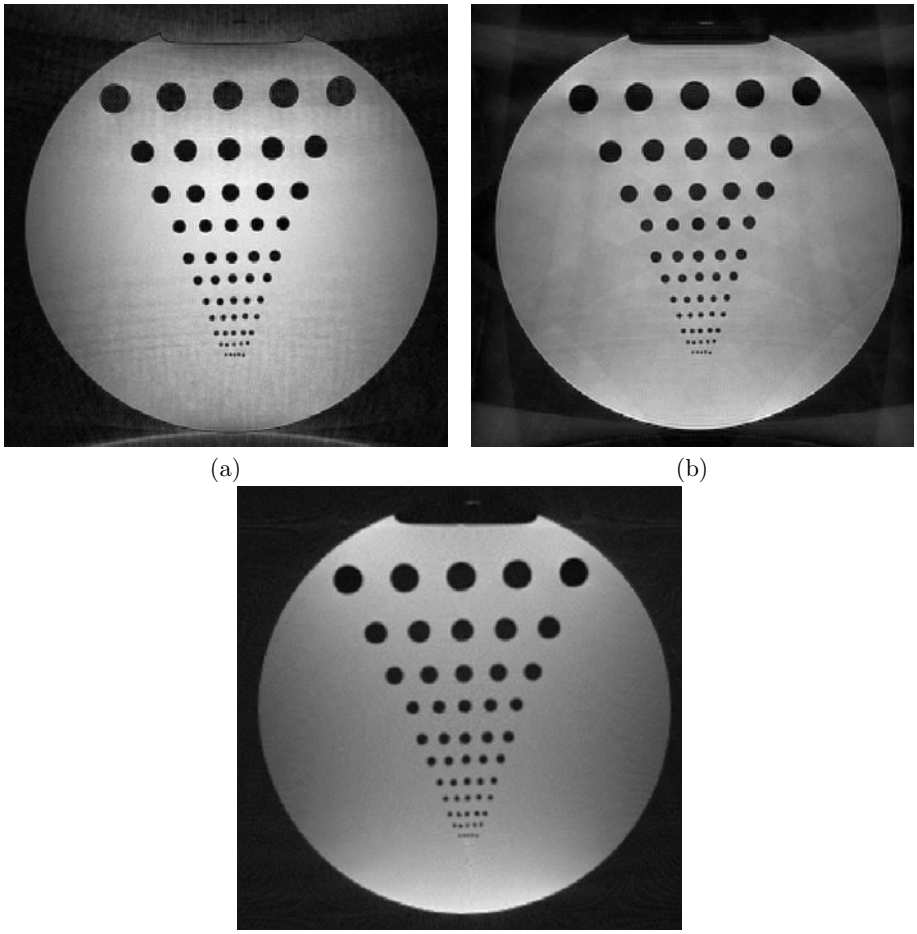
We measure the reconstruction performance for two sets of MR measurement data, which were obtained from a Siemens Magnetom Avanto 1.5T scanner using a trueFISP pulse sequence with a radial trajectory in k-space. For the phantom image shown in Figure 6, there are a total of 504 radial lines with 512 samples each. During the MR scanning, three coils/channels were used; and the scanning parameters are TR=4.8ms, TE=2.4ms, flip angle =60°, and FOV=206mm with a resolution of 256 pixels. For the other data set (a head image) which is not shown due to space limit, there are a total of 248 radial lines again with 512 samples each. In this data set, four channels were used; and the scanning parameters are TR=4.46ms, TE=2.23ms, flip angle =50°, and FOV=250mm with a resolution of 256 pixels. The final image is obtained by taking the square root of the sum of the square of each channel. Table 3 shows the performance of our various implementations. We have implemented

Table 3. MR reconstruction time in milliseconds on the CPU and GPU using backprojection and gridding.

	Backprojection		Gridding	
	CPU	GPU	CPU	GPU
Phantom	16700	130	730	200
Head	8400	60	600	170

the backprojection and gridding algorithms on both the CPU and GPU for speed comparisons. The speed up from the CPU to the GPU is about 3.5 times for the gridding algorithm. For the filtered backprojection, we observe a speed up of about 100 times. Figure 6 shows the reconstructed images for the scanned phantom. The results obtained from our filtered backprojection implementation are identical on the CPU and on the GPU. For the gridding implementations, the results show slight differences but comparable image quality.

For the GPU implementations, we measure the bus transfer time between main memory and GPU memory in both directions. Basically, the measurement data is uploaded to GPU memory and one reconstructed image is downloaded. The upload time for the phantom image is 6.2 milliseconds for 504 measurement lines with 512 samples each in three channels. For the head image the upload time is 4.4 milliseconds for 248 measurement lines with 512 samples each in four channels. The download time for a reconstructed 256×256 image is 0.4 milliseconds.



(a) (b)

(c)

Figure 6. The reconstructed images by using (a) gridding algorithm on the CPU, (b) gridding algorithm on the GPU, and (c) filtered backprojection which yields identical results on the CPU and GPU. The measurement data were obtained from 3 MR coils/channels. For each channel, there are 511 measurement lines with each containing 512 complex samples. We show the reconstruction speeds in Tables 2 and 3.

6. CONCLUSION

We have presented the GPU implementations of gridding and filtered backprojection algorithms for MR image reconstruction with measurement data sampled along radial trajectories. A particular efficient GPU implementation of the FFT was developed. This GPU-based FFT is faster than the CPU-based FFTW for images that is larger than 256×256. With the help of this implementation and the architecture of the GPU, we are able to improve the speed performance of the gridding algorithm and filtered backprojection, respectively, by a factor of three to four and more than 2 orders of magnitude. In addition, the resulting image quality is virtually the same for both CPU-based and GPU-based implementations. From our experiments, we have shown that the GPU inherent 4-channel textures and hardware-accelerated interpolation is very suitable for MR image reconstruction, especially for radially sampled measurement data. With increasingly more real-time applications in MRI, it is possible that the GPU-based MR image reconstruction will be indispensable in the future.

ACKNOWLEDGMENTS

We would like to thank Christine Lorenz for making this project possible.

REFERENCES

1. Z.-P. Liang and P. C. Lauterbur, *Principles of magnetic resonance imaging*, IEEE Press, 2000.
2. J. O'Sullivan, "Fast sinc function gridding algorithm for fourier inversion in computer tomography," *IEEE Transaction on Medical Imaging* **M1-4**(4), pp. 200–207, 1985.
3. J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a convolution function for fourier inversion using gridding," *IEEE Transaction on Medical Imaging* **10**(3), pp. 473–478, 1991.
4. B. P. Sutton, D. C. Noll, and J. A. Fessler, "Fast, iterative image reconstruction for mri in the presence of field inhomogeneities," *IEEE Transaction on Medical Imaging* **22**(2), pp. 178–188, 2003.
5. R. V. de Walle, H. H. Barrett, K. J. Myers, M. I. Altbach, B. Desplanques, A. F. Gmitro, J. Cornelis, and I. Lemahieu, "Reconstruction of mr images from data acquired on a general nonregular grid by pseudoinverse calculation," *IEEE Transaction on Medical Imaging* **19**(12), pp. 1160–1167, 2000.
6. B. Dale, M. Wendt, and J. L. Duerk, "A rapid look-up table method for reconstructing mr images from arbitrary k-space trajectories.," *IEEE Transaction on Medical Imaging* **20**(3), pp. 207–217, 2001.
7. M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," 1998. URL: `http://www.fftw.org`.
8. K. Moreland and E. Angel, "The FFT on a GPU," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119, Eurographics Association, (Aire-la-Ville, Switzerland, Switzerland), 2003.
9. T. Schiwietz and R. Westermann, "GPU-PIV," in *VMV*, pp. 151–158, 2004.
10. T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on the gpu using a split-stream-fft," in *VMV*, pp. 395–403, 2004.
11. S. Thilaka and L. Donald, *GPU Gems 2*, ch. Medical Image Reconstruction with the FFT, pp. 765–784. Addison-Wesley, 2005.
12. F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Transaction of Nuclear Science* , 2005.
13. Z.-H. Cho, J. P. Jones, and M. Singh, *Foundations of Medical Imaging*, Wiley, New York, NY, USA, 1994.
14. A. Oppelt, *Imaging Systems for Medical Diagnostics*, Publicis Corporate Publishing, Erlangen, Germany, 2005.
15. S. Müller, M. Bock1, C. Fink, S. Zhlsdorff, P. Speier, and W. Semmler, "truefisp mri with active catheter tracking and real-time parallel image reconstruction," in *Proc. Intl. Soc. Mag. Reson. Med, 13th Scientific Meeting & Exhibition*, p. 2158, 2005.
16. M. A. Griswold and et al., "Generalized autocalibrating partially parallel acquisitions (grappa)," *Magnetic Resonance in Medicine* **47**(6), pp. 1202–1210, 2002.
17. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics (2nd ed. in C): principles and practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
18. M. Woo, Davis, and M. B. Sheridan, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
19. K. Gray, *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, Redmond, WA, USA, 2003.
20. Information about GPGPU programming. `http://www.gpgpu.org`.
21. E. O. Brigham, *The fast Fourier transform and its applications*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
22. M. A. Bernstein, K. F. King, and X. J. Zhou, *Handbook of MRI Pulse Sequences*, Elsevier Academic Press, Burlington, MA, USA, 2004.
23. J. G. Pipe and P. Menon, "Sampling density compensation in mri: rationale and an iterative numerical solution," *Magnetic Resonance in Medicine* **41**(1), pp. 179–186, 1999.
24. H. Sedarat and D. G. Nishimura, "On the optimality of the gridding reconstruction algorithm," *IEEE Transaction on Medical Imaging* **19**(4), pp. 306–317, 2000.