A CUDA-based reverse gridding algorithm for MR reconstruction

Jingzhu Yang\textsuperscript{a,*,1}, Chaolu Feng\textsuperscript{b,1}, Dazhe Zhao\textsuperscript{a,c}

\textsuperscript{a}Key Laboratory of Medical Image Computing of Ministry of Education, Northeastern University, Shenyang, CO 110179, China
\textsuperscript{b}School of Information Science and Engineering, Northeastern University, Shenyang, CO 110179, China
\textsuperscript{c}Neusoft Group Co., Ltd., Shenyang, CO 110179, China

Received 23 February 2012; revised 29 June 2012; accepted 29 June 2012

Abstract

MR raw data collected using non-Cartesian method can be transformed on Cartesian grids by traditional gridding algorithm (GA) and reconstructed by Fourier transform. However, its runtime complexity is $O(K \times N^2)$, where resolution of raw data is $N \times N$ and size of convolution window (CW) is $K$. And it involves a large number of matrix calculation including modulus, addition, multiplication and convolution. Therefore, a Compute Unified Device Architecture (CUDA)-based algorithm is proposed to improve the reconstruction efficiency of PROPELLER (a globally recognized non-Cartesian sampling method). Experiment shows a write–write conflict among multiple CUDA threads. This induces an inconsistent result when synchronously convoluting multiple $k$-space data onto the same grid. To overcome this problem, a reverse gridding algorithm (RGA) was developed. Different from the method of generating a grid window for each trajectory as in traditional GA, RGA calculates a trajectory window for each grid. This is what “reverse” means. For each $k$-space point in the CW, contribution is cumulated to this grid. Although this algorithm can be easily extended to reconstruct other non-Cartesian sampled raw data, we only implement it based on PROPELLER. Experiment illustrates that this CUDA-based RGA has successfully solved the write–write conflict and its reconstruction speed is 7.5 times higher than that of traditional GA.

Keywords: CUDA; Fourier transform; Gridding; Reverse gridding; Sampling trajectory

1. Introduction

Since magnetic resonance imaging (MRI) was first developed by Paul C. Lauterbur in 1971 [1–4], where MR data were sampled using radial scanning method [5] and reconstructed through X-CT back-projection [6,7], and the first MRI experiment result was published in *Nature* in 1973 [8], numerous research studies have been done in this field. Mezrich et al. [9] devised a phase-encoding method in $k$-space for phase direction and frequency direction, which reconstructs images with two-dimensional Fourier transform (2DFT) [10,11]. Mansfield [12] developed echo-planar imaging (EPI) in 1977. It is a fast imaging method and improves the practicability of applying MRI to clinical application [13]. Although EPI can finish sampling in less than 1 s [14,15] and use fast Fourier transform (FFT) to reconstruct images directly [16–18], it requires a professional gradient system. Meyer et al. [19] sampled MR data in $k$-space making use of spiral trajectory in 1992. It is a fast MRI scheme under a general gradient field, but sampled data are distributed at unequal spaced intervals [20–22].

In 1999, a nonhomogeneous center oversampled method called Periodically Rotated Overlapping ParallEL Lines with Enhanced Reconstruction (PROPELLER) was firstly proposed by Pipe [23] to improve imaging quality. PROPELLER has been applied successfully to brain MRI [24–26] and adopted by General Electric (GE) as an important technique for use in its high-end MRI devices. However, data collected by PROPELLER are still unequally spaced. So it needs to be resampled on Cartesian grids [27] for FFT reconstruction [28,29].
This transformation method is called gridding algorithm (GA) [30,31].

GA is essentially a convolution interpolation method [32]. It convolutes each data with a convolution kernel to distribute its energy on different grids [33]. O’Sullivan [34] indicated in 1985 that an optimal GA should adopt infinite sinc function as the convolution kernel, but a finite convolution kernel is used in practice.

The key point of GA is the weighting function and interpolation function [35]. The former function is also named density compensation function (DCF), which is used to compensate for nonuniformity sampled data in k-space [36]. The latter is also called convolution function or convolution kernel, which is used to interpolate the nonhomogeneous k-space data on Cartesian grids.

The objective of this article was to develop a fast GA for MR reconstruction. In Section 2, a particular analysis of GA is presented. In Section 3, the flow of optimal GA with the Compute Unified Device Architecture (CUDA) is proposed. In Section 4, a reverse GA is devised to solve the write–write conflict caused by CUDA GA. A comparison experiment is shown in Section 5. Finally, conclusion and discussion are provided in Section 6.

2. General gridding algorithm

PROPELLER samples data in k-space with multiple blades, which are named phase-encoded lines. The center of the blades is fixed in the whole sampling process and the current blade rotates through a certain angle along an assigned direction (counterclockwise) to generate the next blade. Once the edges of these blades could connect sequentially and form a whole circle, a k-space filling is finished. Although one blade only sweeps one circle area, an overlap filling is made for the k-space center. Because the central part determines the image contrast and signal-to-noise ratio (SNR), data sampled by PROPELLER hold a higher contrast and SNR. Therefore, our research mainly focuses on data collected by PROPELLER, and the detailed principle is described in the previous reference [23].

2.1. Sampling trajectory

The first blade of sampling trajectory is given below:

\[ S(u, v)_x = \frac{k_{max} x}{2} + \frac{k_{max} x}{SaSiz} \times u \]  \hspace{1cm} (1)

\[ S(u, v)_y = \frac{k_{max} y}{2} + \frac{k_{max} y}{ScMat} \times \left( \frac{ScMat-ETL}{2} + v \right) \]  \hspace{1cm} (2)

where \( k_{max} x \) and \( k_{max} y \) respectively represent the normalized length of the x-axis and y-axis in Cartesian space. SaSiz is the number of sampling points and \( u \) is in the range of 0 and SaSiz. ScMat is the scanning matrix size and ETL denotes the echo train length and \( v \) is in the range of 0 and ETL.

Assume BladeNum denotes the number of blades, then rotation angle for each trajectory blade is equal to \( \theta = \pi / \text{BladeNum} \). The rotation of current blade with angle (\( \theta \)) can be achieved by multiplying it a unit matrix in the right side. This matrix can be written as:

\[
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}.
\]  \hspace{1cm} (3)

By storing these BladeNum blades into \( S(u, v) \) in turn, we obtain the sampling trajectory.

2.2. Gridding principle

Jackson et al. [37] indicated that the Kaiser–Bessel window function is a good choice of convolution functions, which has been widely recognized and has become one of the most important parts of GA.

The formula of Jackson’s GA can be expressed as follows:

\[
M_{RS} = \left[ M(u, v) \cdot \left( \frac{S(u, v)}{S(u, v) \cdot C(u, v)} \right)^*C(u, v) \right] \cdot R(u, v),
\]  \hspace{1cm} (4)

where \( M \) represents k-space data, \( M_{RS} \) indicates MR data after resampling, \( S \) denotes the sampling trajectory at unequal spaced intervals, \( R \) is a two-dimensional Cartesian grid, \( C \) is the convolution function, \( S(u, v)[S(u, v) \cdot C(u, v)] \) is the DCF and \( * \) is the convolution operator.

The schematic flow diagram of GA is shown in Fig. 1. Firstly, k-space raw data and sampling trajectory are loaded into memory. Secondly, the convolution window (CW) for each trajectory point is computed in Cartesian space. Thirdly, a k-space Euclidian distance between each grid point in CW and this trajectory point is calculated. Let dk and kwidth respectively denote the k-space Euclidian distance and the operator which controls the size of CW. The contribution of this trajectory point will be obtained by convolution if it satisfies \( dk < \text{kwidth} \). Finally, we get the gridding result. The detailed parameter information is given as follows.

Assume that \( (\text{cwidth}.x, \text{cwidth}.y) \) means the size of a CW in equal-interval Cartesian space, \( (\text{gsize}.x, \text{gsize}.y) \) denotes

![Fig. 1. Schematic diagram of the gridding algorithm.](image-url)
the size of the grid matrix and \((kwidth.x, kwidth.y)\) indicates the corresponding CW size in \(k\)-space. Then \((kwidth.x, kwidth.y)\) can be calculated using Eqs. (5) and (6).

\[
kwidth.x = kmax.x \times cwidth.x / gridsize.x \tag{5}
\]

\[
kwidth.y = kmax.y \times cwidth.y / gridsize.y \tag{6}
\]

Let \((gcenter.x, gcenter.y)\) represent the center of the Cartesian coordinate. The range of CW \([\text{ixmin, iymax}]) \) corresponding to trajectory \(S(u, v)\) can be computed as below:

\[
\text{ixmin} = \left(\frac{S(u, v).x - kwidth.x}{kmax.x}\right) \times \text{gsizex} + gcenter.x \quad \text{if } (\text{ixmin} < 0) \times \text{ixmin} = 0
\]

\[
\text{ixmax} = \left(\frac{S(u, v).x + kwidth.x}{kmax.x}\right) \times \text{gsizex} + gcenter.x + 1 \quad \text{if } (\text{ixmax} \geq \text{gsizex}) \times \text{ixmax} = \text{gsizex} - 1
\]

\[
\text{iymin} = \left(\frac{S(u, v).y - kwidth.y}{kmax.y}\right) \times \text{gsizey} + gcenter.y \quad \text{if } (\text{iymin} < 0) \times \text{iymin} = 0
\]

\[
\text{iymax} = \left(\frac{S(u, v).y + kwidth.y}{kmax.y}\right) \times \text{gsizey} + gcenter.y + 1 \quad \text{if } (\text{iymax} \geq \text{gsizey}) \times \text{iymax} = \text{gsizey} - 1
\]

Let \(\text{grid}(p, q)\) denote the grid point in \([\text{ixmin}, \text{iymax}]) \). Then

\[
dk_x = kmax.x \times \frac{\text{grid}(p, q).x - gcenter.x}{\text{gsizex}} - S(u, v).x \tag{11}
\]

\[
dk_y = kmax.y \times \frac{\text{grid}(p, q).y - gcenter.y}{\text{gsizey}} - S(u, v).y \tag{12}
\]

\[
dk = \sqrt{dk_x^2 + dk_y^2}. \tag{13}
\]

If \(dk\) is less than \(kwidth\) (Euclidian distance of \(kwidth.x\) and \(kwidth.y\)), \(k\)-space data at point \((u, v)\) are convoluted to generate its contribution to \(\text{grid}(p, q)\). The homogeneous resampling result for this grid is the accumulation of all the contribution values at \((p, q)\).

2.3. Algorithm complexity

The following is the pseudo-code of GA.

Gridding algorithm.

\(1\) for \(u \leftarrow 0\) to \(\text{dataw}\)

\(2\) for \(v \leftarrow 0\) to \(\text{datah}\) calculate window \([\text{ixmin}, \text{iymax})\) of trajectory \(S(u, v)\) under condition of \([\text{kwidth}.x, \text{kwidth}.y]\).

\(3\) for \(p \leftarrow \text{ixmin}\) to \(\text{iymax}\)

\(4\) for \(q \leftarrow \text{ixmin}\) to \(\text{iymax}\)

\(5\) calculate Euclidian distance \(dk\) between \((p, q)\) and \((u, v)\) in \(k\)-space.

\(6\) if \(dk\) is less than \(kwidth\)

\(7\) then convolute \(S(u, v), M(u, v),\) and \(R(u, v)\) with \(C(u, v)\) to accumulate the contribution to \(\text{grid}(p, q)\).

Suppose \(\text{dataw}\) is the column resolution and \(\text{datah}\) is the row resolution of \(k\)-space raw data and they are both equal to \(N\). In addition, assume the size of the CW is represented as \(K\), which is smaller than \(N\); it is obvious that the runtime complexity for GA achieves \(\mathcal{O}(K \times N^2)\). Besides that, a large amount of matrix calculation is involved including modulus, addition and multiplication. A CW and convolution for each \(k\)-space point need to be calculated in GA; therefore the efficiency of the algorithm will significantly reduce as data size increases. It will take more than 2 s to reconstruct \(408 \times 1024\) \(k\)-space MR data to a \(512^2\) image on a 3.0-GHz Intel Core 2 Duo processor. Therefore, it is necessary to develop a method to accelerate GA [38].

3. CUDA-based gridding optimization

In recent years, rapid development of graphic processing units (GPUs) has significantly improved the processing speed and the quality of computer graphics especially for medical image processing technology [39–43].

3.1. Data distribution strategy

CUDA developed by NVIDIA supports general-purpose computation on GPUs (GPGPU) [44–48]. CUDA-based GPUs consist of a certain number of parallel processing units. There are \(N\) streaming multiprocessors (SMs) in each CUDA-supported GPU chip. Each SM contains a certain capacity shared memory. A SM comprises \(M\) streaming processors (SPs) and each SP has a register. Thread is the minimum execution unit; multiple threads form a block in CUDA. On account of blocks corresponding to SMs and threads corresponding to SPs in hardware, threads in the same block can read from and write to the shared memory.

In order to maximize computing power, the optimal CUDA based method is essentially to find a balance between the number of blocks and the number of threads in one block. Although more threads in a block are beneficial to use the shared memory, the number of SPs is limited in one SM. So the more threads there are in a block, the more thread-scheduling times will be consumed. On the other hand, the maximum number of threads in a single block is also limited, for example, 512 in the GeForce 8800 GT GPU. On the contrary, thread number reduction can improve block runtime efficiency, but resource inside the block may not be used adequately. Furthermore, too many blocks will generate pressure on resource scheduling of SMs, which may seriously degrade the performance.

The following is the processing step for some CUDA-based calculations performed on a \(DW \times DH\) data set.
In Fig. 2, data matrix SourceData is divided into multiple blocks. They are all BS×BS and BS is 16 in this article. As mentioned above, the value of BS is relevant to hardware resource. Thus, the block matrix \( D_{(i,j)} \) can be expressed as follows:

\[
\begin{bmatrix}
D_{(0,0)} & D_{(0,1)} & \cdots & D_{(0,k)} \\
D_{(1,0)} & D_{(1,1)} & \cdots & D_{(1,k)} \\
\vdots & \vdots & \ddots & \vdots \\
D_{(m,0)} & D_{(m,1)} & \cdots & D_{(m,k)}
\end{bmatrix}
\]

If \( DH \) and \( DW \) cannot be divisible by \( BS \), the remainder boundary needs to be filled with 0 or other values depending on the requirement.

SourceData is transferred to video memory by the cudaMemcpy command using the macro cudaMemcpyHostToDevice. In the end, the result is copied back into system memory from video memory by cudaMemcpy using cudaMemcpyDeviceToHost.

### 3.2. CUDA-based gridding algorithm

These days, many desktop PCs have dual-, quad-, or even eight-core processors, and gridding can be easily implemented generally using multithreading to get acceleration on multicore CPUs. But the best speedup ratio should not be greater than the core number. The float-point performance of CPUs is only dozens of GFLOPS/s. But GPUs has reached or even exceeded thousands of GFLOPS/s. By the way, GPUs are much cheaper than CPUs with the same computing ability. Therefore, we designed a CUDA-based method to accelerate GA in this section. As shown in Fig. 3, this algorithm includes four steps: initialization, task allocation, calculation and results return.

#### 3.2.1. Initialization

Algorithm initialization includes two key steps: environment measurement and data loading. The former is mainly used to establish a CUDA device context. It consists of GPU parameters test, hardware selection (multiple GPUs in one system) and execution parameters’ configuration. The second step needs to allocate video memory for \( k \)-space MR data, trajectory and convolution kernel. These data are loaded from system memory to this allocated space.

#### 3.2.2. Task allocation

This proposed CUDA-based GA establishes thread blocks based on the quantity of sampling trajectory. Assume that the size of \( k \)-space data is \( TDW\times TDH \) and the number of threads inside a block is \( BS\times BS \), the total number of thread blocks will be \([TDW/BS]\times[TDH/BS]\). The ceiling operators here mean that the upper bound operation must be performed; they can also be seen as the 0 filling representation.

#### 3.2.3. Calculation

The CUDA-based GA is encapsulated into kernels, which includes the CW calculation function, distance calculation function and convolution calculation functions. Matrix modulus calculation, matrix addition and matrix multiplication will be invoked inside these kernels. Invoking these kernels requires some CUDA-based configuration. This configuration includes the block number, thread number and shared memory size inside each block, and processing stream number.

#### 3.2.4. Results return

Similar to the first step, gridding results are copied back to system memory, where it is reconstructed by FFT.

### 3.3. Algorithm analysis

In order to verify the correctness of the CUDA-based GA, water phantom with parameters of \( ScMat=256 \), \( SaSize=960 \), \( ETL=24 \) and \( BladeNum=17 \) is resampled on a 512×512 grid. The algorithm is implemented based on the flow shown in Fig. 3, and the gridding result is reconstructed by FFT. Fig. 4A and B represents respectively the FFT images based on the CPU and on CUDA. It is obvious that the CUDA-based GA image is more blurry than the CPU-based GA image at the axial center and at the boundaries between hollow sectors and solid sectors.

Some representative results are compared between CPU-based GA and CUDA-based GA in Table 1, where column 1 is the grid coordinates, columns 2 and 3 are the dual-calculation results of CPU-based GA, and columns 4 and 5 are the dual-calculation results of CUDA-based GA. It can be seen that two CUDA-based GA results are inconsistent and the difference between them is random.
CUDA-based GA calculates CWs for every trajectory point on a Cartesian grid and allocates the power of these \(k\)-space data on grids inside the CW. The distance between each grid in this CW and the trajectory point is considered as the convolution weight. Fig. 5 is the CW calculation process for GA, where XOY is the trajectory plane (or grid coordinate plane) and \(Z\) is the corresponding values. Therefore, trajectory matrix and gridding matrix construct two conicoids. SV0 and SV1 are the trajectory values of points S0 and S1, respectively. Each trajectory point generates a CW in GA. So S0 corresponds to a grid area R0, which is composed of four grid points: G0, G1, G2 and G3, where GV0, GV1, GV2 and GV3 are their values, respectively, and a conicoid C0 is constructed. Similarly, S1 also corresponds to a window area R1 composed of grid points G4, G5, G6 and G7, and their values are GV4, GV5, GV6 and GV7, respectively. And another conicoid, C1, is constructed there. The raw data of S0 affect the grid points in R0 and those of S1 affect the grid points in R1. Thus, an intersection R2 may possibly exist between R0 and R1, which is composed of G8, G9, G10 and G11. For this reason, a write–write conflict occurs at R2 during the parallel calculation of S0 and S1 in CUDA-based GA.

Based on the analysis above, the most important reason is the write–write conflict during CW calculation in the CUDA-based GA. Hence, Table 1 has the following characteristics.

It is obvious that no write–write conflict exists if there is no intersection. For example, the dual-calculation results of the CUDA-based GA are the same at grid (340,106), which means there might be no intersection at this point.

For the intersection, a write–write conflict appears at corresponding grid (234, 228) in Table 1. CUDA-based thread interaction is decided by thread-scheduling strategy and SP computation speed. Therefore, an intersection might not induce an inconsistency. In other words, theoretical write–write conflict may be sequential in practice. For example, the first result of the CUDA-based GA is the same as that of the CPU-based GA at grid (14, 202), but there is a great difference between the second result and the first one.

### 4. Reverse gridding algorithm

The write–write conflict that happens in GPU memory can lead to inconsistency in CUDA-based GA. In order to eliminate the conflict, we can adjust the thread allocation

<table>
<thead>
<tr>
<th>Position</th>
<th>CPU-based GA 1</th>
<th>CPU-based GA 2</th>
<th>CUDA-based GA 1</th>
<th>CUDA-based GA 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13,327)</td>
<td>0.697381+14.125287i</td>
<td>0.697381+14.125287i</td>
<td>5.245859+9.457215i</td>
<td>−4.595843+9.958951i</td>
</tr>
<tr>
<td>(14,202)</td>
<td>−9.497768−0.931772i</td>
<td>−9.497768−0.931772i</td>
<td>−9.499759−0.933009i</td>
<td>−3.301559−0.078084i</td>
</tr>
<tr>
<td>(14,329)</td>
<td>−6.330377−5.144327i</td>
<td>−6.330377−5.144327i</td>
<td>−3.912250−2.354891i</td>
<td>−5.089190−5.805894i</td>
</tr>
<tr>
<td>(154,209)</td>
<td>−34.936935+0.931772i</td>
<td>−34.936935+0.931772i</td>
<td>−33.445343+101.700485i</td>
<td>−28.070139+74.780998i</td>
</tr>
<tr>
<td>(234,228)</td>
<td>−119.441650+131.354736i</td>
<td>−119.441650+131.354736i</td>
<td>−111.075531+120.896126i</td>
<td>−88.876022+92.113686i</td>
</tr>
<tr>
<td>(276,236)</td>
<td>−440.715088−81.186028i</td>
<td>−440.715088−81.186028i</td>
<td>−400.623810−74.810127i</td>
<td>−379.718231−73.184990i</td>
</tr>
<tr>
<td>(340,106)</td>
<td>4.932378−5.873862i</td>
<td>4.932378−5.873862i</td>
<td>4.933623−5.874030i</td>
<td>4.933623−5.874030i</td>
</tr>
<tr>
<td>(438,207)</td>
<td>−52.370735−59.816231i</td>
<td>−52.370735−59.816231i</td>
<td>−52.370815−59.816929i</td>
<td>−47.514206−51.860348i</td>
</tr>
<tr>
<td>(479,233)</td>
<td>40.921024+1.009181i</td>
<td>40.921024+1.009181i</td>
<td>38.144155+2.000100i</td>
<td>40.920715+1.008464i</td>
</tr>
</tbody>
</table>

Fig. 4. Water phantom FFT images of (A) CPU-based gridding and (B) CUDA-based gridding.

Fig. 5. Schematic diagram of gridding convolution window.
strategy. If the CW of one trajectory point crosses that of another trajectory point, then these two trajectories should be assigned to the same thread. But it is hard to ensure whether a CW cross is generated or not for each trajectory. Therefore, a reverse gridding algorithm (RGA) was developed to deal with this problem in this article.

4.1. Reverse gridding principle

Similar to GA in Fig. 1, the schematic flow diagram of RGA is shown in Fig. 6. MR raw data and sampling trajectory are loaded into memory at first. Then, a CW is computed for each grid point in k-space. And it should be noted that this process is exactly opposite to GA. This is the reason why we named it with “reverse”. Thirdly, k-space Euclidian distance \( dk \) between each trajectory point in the CW and this grid point is calculated. It can be seen that this has the same meaning as the parameter \( dk \) in GA. Finally, grid value is obtained by convolution.

Because sampling trajectory is made up of \( \text{BladeNum} \) blades, each of which is the result of the previous blade with certain angle rotation, the edges of which sequentially connect to form a whole circle. For each grid, RGA has to calculate a CW for each blade. As the coordinate range of CW should be identified, we need to reversely rotate (clockwise) it at a certain angle which is equal to that of GA. It is clear that RGA is more complicated than GA.

Different from CW calculation in GA, this proposed RGA computes CWs from Cartesian space to k-space as shown in Fig. 7. GV0 and GV1 are the grid values of points G0 and G1, respectively. Each grid corresponds to a CW in RGA. So G0 corresponds to a grid area R0, which is composed of four grid points: S0, S1, S2 and S3. Let SV0, SV1, SV2 and SV3 be the values of points S0, S1, S2 and S3, respectively. A conicoid C0 can be constructed. Similarly, S1 also corresponds to a window area R1 composed of trajectory points S4, S5, S6 and S7, the values of which are SV4, SV5, SV6 and SV7, respectively. And a conicoid C1 appears in XYZ space. Because of the grid data, GV0 is affected by R0 and GV1 is affected by R1. In Fig. 7, R0 and R1 may appear at intersection R2, which is made up of S8, S9, S10 and S11. Although there is a read–read conflict in CUDA-based RGA in R2, there is no negative influence.

Although this RGA can be easily extended to reconstruct other non-Cartesian sampled raw data, we implement it only based on PROPELLER in this work. Let \( \text{grid}(u, v) \) be in the range of \([0, \text{gsizer}.x] \) and \( \text{grid}(u, v).y \) be in the range of \([0, \text{gsizer}.y] \). \( \text{MIN} \) and \( \text{MAX} \) are the minimum value function and maximum value function, respectively. Then the CW in blade \( i \) is shown as follows:

\[
\theta = i \times \left( \frac{\pi}{\text{BladeNum}} \right) \tag{14}
\]

\[
\text{xmin} = \frac{\text{grid}(u, v).x - \text{gcenter}.x \times \text{cwidth}.x}{\text{gsizer}.x} \times \text{kmax}.x \tag{15}
\]

\[
\text{ymax} = \text{grid}(u, v).y - \text{gcenter}.y \times \text{cwidth}.y}{\text{gsizer}.y} \times \text{kmax}.y \tag{17}
\]

The reverse CW rotation is expressed in Eqs. (19)–(22), where \( \text{tmp}0–\text{tmp}7 \) are eight coordinate values for four corners of CW. \( \text{tmp}0–\text{tmp}3 \) belong to the x-direction, whereas \( \text{tmp}4–\text{tmp}7 \) belong to the y-direction.

\[
(\text{tmp}0, \text{tmp}4) = (\text{xmin}, \text{ymin}) \times \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \tag{19}
\]

\[
(\text{tmp}1, \text{tmp}5) = (\text{ymax}, \text{ymin}) \times \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \tag{20}
\]

\[
(\text{tmp}2, \text{tmp}6) = (\text{xmin}, \text{ymax}) \times \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \tag{21}
\]

\[
(\text{tmp}3, \text{tmp}7) = (\text{ymax}, \text{ymax}) \times \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \tag{22}
\]

In order to get the window range, the eight new coordinates generated above need to be used to calculate the top-left corner and bottom-right corner, according to

---

Fig. 6. Schematic diagram of the reverse gridding algorithm.
Eqs. (23)–(26).
\[
x_{\min} = \text{MIN}(\text{tmp}_0, \text{tmp}_1, \text{tmp}_2, \text{tmp}_3)
\]
\[
x_{\max} = \text{MAX}(\text{tmp}_0, \text{tmp}_1, \text{tmp}_2, \text{tmp}_3)
\]
\[
y_{\min} = \text{MIN}(\text{tmp}_4, \text{tmp}_5, \text{tmp}_6, \text{tmp}_7)
\]
\[
y_{\max} = \text{MAX}(\text{tmp}_4, \text{tmp}_5, \text{tmp}_6, \text{tmp}_7)
\]

Thus four corner values of CWs are in $k$-space. CW can be obtained in trajectory space in Eqs. (27)–(30).

This procedure is exactly the opposite of Eqs. (1) and (2) used in GA, which also explains the meaning of “reverse” to some extent.

\[
\begin{align*}
  u_{\min} &= \left( x_{\min} + \frac{k_{\max}}{2} \right) \times \frac{\text{SaSiz}}{\text{max}.x}, \\
  u_{\max} &= \left( x_{\max} + \frac{k_{\max}}{2} \right) \times \frac{\text{SaSiz}}{\text{max}.x} + 1, \\
  v_{\min} &= \left( y_{\min} + \frac{k_{\max}}{2} \right) \times \frac{\text{ScMat} \cdot \text{ScMat-ETL}}{\text{max}.y} \frac{2}{2}, \\
  v_{\max} &= \left( y_{\max} + \frac{k_{\max}}{2} \right) \times \frac{\text{ScMat} \cdot \text{ScMat-ETL}}{\text{max}.y} \frac{2}{2} + 1
\end{align*}
\]

Assume $(p, q)\in\{(u_{\min}, v_{\min}), (u_{\max}, v_{\max})\}$, the $k$-space Euclidian distance between grid $(u, v)$ and trajectory point $(p, q)$ is calculated by Eqs. (31), (32) and (33), where $dk_x$ and $dk_y$ have the same meaning as used in Eqs. (11) and (12).

\[
\begin{align*}
  dk_x &= k_{\max} \times \frac{\text{grid}(u, v).x - gcenter.x}{\text{gsizex}}, \\
  dk_y &= k_{\max} \times \frac{\text{grid}(u, v).y - gcenter.y}{\text{gsizey}}. \quad (31, 32)
\end{align*}
\]

Then we convolute MR raw data to generate a grid value at $(u, v)$. These data satisfy the condition $dk < k_{\text{width}}$. 

4.2. CUDA-based reverse gridding optimization

In order to improve the calculation speed, a CUDA-based RGA is proposed based on the principle mentioned above. As shown in Fig. 8, there are four parts in the algorithm, which are initialization, task allocation, calculation and results return, just like in CUDA-based GA.

4.2.1. Initialization

This step is totally the same as that in CUDA GA.

4.2.2. Task allocation

Whereas CUDA-based GA generates thread blocks according to trajectory, CUDA-based RGA establishes thread blocks based on Cartesian grids. Then there will be $[\text{gsizex} \times \text{gsizey}]$ blocks. The CUDA-based RGA is also encapsulated into three kernels just like in CUDA-based GA, but the operations in them are very different.

4.2.3. Calculation

Calculation is the key step for CUDA-based RGA, where its result will be reconstructed into images by FFT. As indicated by the dash-dotted frame in Fig. 8 (right), the calculation is divided into three main substeps: CW calculation, $k$-space Euclidian distance calculation and MR data convolution. CW calculation is controlled by Eqs. (14) to (30) in CUDA threads. $k$-Space Euclidian distance calculation is performed in terms of Eqs. (31), (32) and (33). CW comprises square cells indicated by a red dot, each of which is representative of a trajectory. Each grid corresponds to a square cell (a CW). Then the $k$-space Euclidian distance calculation kernel is invoked. Matrix DK is made up of square cells related to CW marked by blue dots.

![Fig. 8. Flow diagram of the CUDA-based reverse gridding algorithm.](image-url)
points. The blue dot denotes the distance between each trajectory belonging to the window and the grid generating this CW. Finally, the CUDA convolution kernel convolutes trajectory $S$, $k$-space MR raw data $M$ and distance matrix $DK$ to generate grid results.

4.2.4. Results return

Similar to CUDA-based GA, results are copied back to system memory from video memory in the same method.

5. Results

All the algorithms we mentioned above are implemented on a desktop PC. Three groups of experimental data are made up of a water phantom, a static head and a moved head. They all come from Neusoft PHILIPS Superstar with the following scan parameters: 256 lines scan matrix, 960 sampling points, ETL of 24, and 17 blades.

With the same raw data as in Table 1, RGA results are shown in Table 2, where column 1 is the grid coordinates; columns 2 and 3 are the grid results of the CPU-based GA and CPU-based RGA, respectively; and columns 4 and 5 are the dual-calculation grid results of CUDA-based RGA.

Compared with Table 1, except for minute differences at the fifth or the sixth decimal place, the CPU-based RGA result is in accord with the CPU-based GA one. And differences between them may have been caused by accumulation errors in floating-point operations. Meanwhile, CUDA-based RGA can ensure consistency. That is to say, it is explicit and effective to resolve write–write conflict occurring in CUDA-based GA. But floating-point precision and round-off error make the result of CUDA-based RGA different from that of CPU-based GA at the third to the sixth decimal place. Reconstruction images are illustrated in Fig. 9. The FFT reconstruction result processed by the CUDA-based RGA on the water phantom is almost the same as that of the CUDA-based GA, which can meet application requirements.

FFT images with CPU-based GA on water phantom, static head, and moved head are shown in Fig. 9A, C and E, respectively. Similarly, FFT images with CUDA-based RGA are shown in Fig. 9B, D and F, respectively. It is obvious that blurs or artifacts near the axes and boundaries of hollow sectors and solid sectors in Fig. 4 have been corrected.

### Table 2

<table>
<thead>
<tr>
<th>Position</th>
<th>CPU-based GA</th>
<th>CPU-based RGA</th>
<th>CUDA-based GA</th>
<th>CUDA-based RGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13,327)</td>
<td>0.697381+14.125287i</td>
<td>0.697381+14.125287i</td>
<td>0.696859+14.124587i</td>
<td>0.696859+14.124587i</td>
</tr>
<tr>
<td>(14,202)</td>
<td>−9.497768−0.931772i</td>
<td>−9.497768−0.931772i</td>
<td>−9.499758−0.933008i</td>
<td>−9.499758−0.933008i</td>
</tr>
<tr>
<td>(14,329)</td>
<td>−6.330377+5.144327i</td>
<td>−6.330377+5.144327i</td>
<td>−6.329720−5.144278i</td>
<td>−6.329720−5.144278i</td>
</tr>
<tr>
<td>(154,209)</td>
<td>−34.936935+104.920227i</td>
<td>−34.936935+104.920235i</td>
<td>−34.936508+104.916725i</td>
<td>−34.936508+104.916725i</td>
</tr>
<tr>
<td>(234,228)</td>
<td>−119.441650+131.354736i</td>
<td>−119.441658+131.354721i</td>
<td>−119.434479+131.350266i</td>
<td>−119.434479+131.350266i</td>
</tr>
<tr>
<td>(276,236)</td>
<td>−440.715088+81.185628i</td>
<td>−440.715088+81.186020i</td>
<td>−440.728760+81.185410i</td>
<td>−440.728760+81.185410i</td>
</tr>
<tr>
<td>(340,106)</td>
<td>4.932378+5.876826i</td>
<td>4.932378+5.876826i</td>
<td>4.936242+5.874030i</td>
<td>4.936242+5.874030i</td>
</tr>
<tr>
<td>(438,207)</td>
<td>−52.370735+59.816231i</td>
<td>−52.370735+59.816235i</td>
<td>−52.370815+59.816929i</td>
<td>−52.370815+59.816929i</td>
</tr>
<tr>
<td>(478,233)</td>
<td>−40.921024+1.009181i</td>
<td>−40.921024+1.009181i</td>
<td>−40.920719+1.008464i</td>
<td>−40.920719+1.008464i</td>
</tr>
</tbody>
</table>
reconstruction images in Fig. 9C and E are totally the same as those in Fig. 9D and F, respectively.

Efficiency comparison among CPU-based GA, CUDA-based GA, CPU-based RGA and CUDA-based RGA is shown in Fig. 10. The CPU-based GA took 0.5 s to resample all of the three groups of experiment data. However, the CUDA-based GA spent only 0.03 s to do this, but some errors occurred as mentioned above. From Eqs. (14) to (30), it shows that RGA needs to calculate a CW in every blade for each grid by rotating a certain angle from the previous one. Thus it is more complicated than GA. So CPU-based RGA spends more than 1.3 s to resample the same data, which is 2.6 times longer than GA. But CUDA-based RGA can shorten it down to 0.067 s.

The speedup ratio comparisons between CUDA-based GA and CPU-based GA, between CUDA-based RGA and CPU-based RGA, and between CUDA-based RGA and CPU-based GA are shown in Fig. 11, respectively. The first speedup ratio is up to 16.87 and the second one is up to 19.55. As mentioned in Section 3, if the size of the source data matrix is indivisible by block size, the boundary will be filled with 0 or other values to make the redundant elements an unabridged block. ETL (24) and BladeNum (17) are neither divisible by 16. But SaSz (960) can be divided exactly by 16. So there are totally 1560 thread blocks. But for RGA, grid size (512) is divisible by block size and 1024 thread blocks will be generated. In CUDA, the fewer the thread blocks, the less cost in CUDA thread scheduling. That is, although RGA is more complex than GA, the speedup ratio of RGA is greater than that of GA. Finally, the ultimate speedup ratio between CUDA-based RGA and CPU-based GA is up to 7.5.

6. Discussion and future work

Because the MR raw data collected by PROPELLER are distributed at unequal space interval, GA is applied to compensate it on Cartesian grids. GA is too complex to be highly efficient. Therefore, we proposed a CUDA-based method to improve its efficiency. And its speedup ratio reaches up to 16.87. However, a write–write conflict at the same grid among multiple CUDA threads occurs when convoluting k-space data. That leads to reconstruction inconsistency. Therefore, we designed a reverse gridding algorithm (RGA). Although RGA is more complicated than GA, the number of thread blocks is less than that in GA. So its speedup ratio is much higher than that of GA. Experiment shows that the inconsistency has been eliminated by CUDA-based RGA, and CUDA-based RGA is 7.5 times faster than CPU-based GA. Although floating-point precision and round-off error make the result of our CUDA-based RGA different from that of CPU-based GA at the third to the sixth decimal place, our CUDA-based RGA is 7.5 times faster than traditional GA [49–51]. Therefore, our algorithm can
ensure the image quality. At the same time, it is much faster than traditional GA.

All data are directly copied from system memory to video memory or vice versa in a 1D array in our work. So the bottleneck is the bandwidth between them. Moreover, there are a considerable shared memory in SM and a register in SP. If CUDA-based algorithms could make full use of these resources, performance will increase significantly. Therefore, taking advantage of these resources by reasonable thread creation and task allocation strategy will be the main work that we will concentrate on next to get a further efficiency improvement. In addition, we will make the motion compensation performance improvement of PROPELLER reconstruction as our next main research content.

Acknowledgment

The authors would like to thank Neusoft’s Dr. Jinghui Li and Dr. Hongyu Gou for their support on the basic theory of magnetic resonance imaging and experiment data set.

References


