Implementation of GPU-Accelerated Back Projection for EPR imaging

Zhiwei Qiao[1,*], Gage Redler[2], Boris Epel[3], Yuhua Qian[1], Howard Halpern[3]

[1]School of Computer and Information Technology, Shanxi University, Taiyuan, Shanxi 030006, China

[2]Rush Hospital, Chicago, IL, ,USA (Please Gage complete it)

[3]Department of Radiation and Cellular Oncology, The University of Chicago, Chicago, IL 60637, USA

Abstract:

Electron paramagnetic resonance (EPR) Imaging (EPRI) is a robust method for measuring *in vivo* oxygen concentration (pO_2). For 3D pulse EPRI, a commonly used reconstruction algorithm is the filtered backprojection (FBP) algorithm, in which the backprojection process is computationally intensive and may be time consuming when implemented on a CPU. A multistage implementation of the backprojection can be used for acceleration, however it is not flexible (requires equal linear angle projection distribution) and may still be time consuming. In this work, single-stage backprojection is implemented on a GPU (Graphics Processing Units) having 1152 cores to accelerate the process. The GPU implementation results in acceleration by over a factor of 200 overall and by over a factor of 3500 if only the computing time is considered. Some important experiences regarding the implementation of GPU-accelerated backprojection for EPRI are summarized. The resulting accelerated image reconstruction is useful for real-time image reconstruction monitoring and other time sensitive applications.

Keywords: GPU, acceleration, back projection, EPR, EPR imaging

1. Introduction

Electron Paramagnetic Resonance Imaging (EPRI) is a technique which can image the spatial distribution of paramagnetic spin probes [1]. The magnetic resonance images of water soluble spin probes *in vivo* show high sensitivity to varied physiologic information [2]. Specialized spin probes are designed to report on specific physiology [3]. EPRI can be used to measure the spatial distribution of endogenous or introduced paramagnetic species, tissue redox status, pH, and microviscosity [4].

Images of oxygen concentration (pO_2) can provide prognostic insight for anticancer therapies. The oxygenation status of a tumor is an important determinant for the outcome of radiation therapy. EPRI images of pO_2 may provide necessary information for image guided dose painting, in which an appropriate spatial distribution of radiation dose is chosen to focus more strongly on radio-resistant hypoxic tumor regions [5].

There are two main types for EPRI, pulse EPRI and continuous wave (CW) EPRI [6]. Pulse EPRI is advantageous in certain situations compared to CW EPRI because of its faster imaging

* Corresponding author. Address: School of Computer and Information Technology, Shanxi University, Wucheng Road 92, Taiyuan, Shanxi 030006, China.
E-mail: zhiweiqiaook@nuc.edu.cn

speed. For tomographic 3D pulse EPRI, the classical image reconstruction algorithm is the 3D FBP algorithm [9][10], which derives from 3D inverse radon transform.

The FBP algorithm consists of two steps, the filtration process and the backprojection process. The filtration process is relatively fast, however the backprojection is very slow and, when implemented on a CPU, limits image reconstruction speed.

A multistage backprojection implementation can speed up the process. However, this method has multiple restrictions, including uniform linear angular projection sampling. As an example, the multistage method cannot directly reconstruct images from uniform solid angular projection sampling, which has been shown to be an optimal sampling pattern. For this projection scheme additional interpolation is required [8].

GPU (Graphics Processing Unit) implementation of the backprojection process is a known pathway for computation acceleration due to its favorable ratio of acceleration capability to cost [11][12][13][14][15][16][17][18].

In this work, the single-stage backprojection process is implemented on the GPU for accelerated image reconstruction. In Sec. 2, the 3D FBP algorithm is introduced and the computational complexity is analyzed. In Sec. 3, we demonstrate the GPU acceleration in detail. In Sec. 4, we discuss our approach and summarize some important experiences. In Sec. 5, a short conclusion is given.

2. The 3D FBP Algorithm and Its Time Complexity

The 3D FBP formula comes from the 3D inverse Radon Transform [19]. Without derivation, the formulation of 3D FBP is shown in Eq. (1) - (6).

$$f(\vec{r}) = f(x,y,z) = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} g(t,\varphi,\theta) \sin\theta \, d\varphi d\theta \tag{1}$$

here,

$$t = \vec{r} \cdot \vec{G} = (x,y,z) \cdot \left(G_x, G_y, G_z\right) \tag{2}$$

$$G_x = \cos\varphi \sin\theta \;\; ; \;\; G_y = \sin\varphi \sin\theta \;\; ; \;\; G_z = \cos\theta \tag{3}$$

$$g(t,\varphi,\theta) = p(t,\varphi,\theta) * h(t) \tag{4}$$

$$h(t) = \mathcal{F}^{-1}\{\omega^2\} = \int_{-\infty}^{+\infty} \omega^2 e^{j2\pi\omega t} d\omega \tag{5}$$

$$p(t,\varphi,\theta)=\iiint_{-\infty}^{+\infty} f(x,y,z)\delta\left(\vec{r}\cdot\vec{G}-t\right)dxdydz \tag{6}$$

In Eq. (1)-(6), $f(x,y,z)$ is a 3D object, $p(t,\varphi,\theta)$ is a 1D spatial projection at angle (φ,θ), $g(t,\varphi,\theta)$ is the filtered projection, and $h(t)$ is the unit impulse response of the parabola filter. In Eq. (2), t is the projection address of a point (x,y,z) at the angle(φ,θ). In Eq. (5), $\mathcal{F}^{-1}\{\cdot\}$ represents the inverse Fourier transform.

Assuming the spatial projections are distributed using a uniform solid angle pattern, we will use a single-stage backprojection method to implement the FBP process.

If we have Q spatial projections, each of which has S points, the projections can be stored in a 2D array 'proj' with size [S, Q]. If the reconstructed object has N rows, N columns and N slices, the object can be stored in a 3-D array 'object' with size [N, N, N].

The reconstruction pseudo-codes for the case above are shown below.

Step1, Parabola filtration of all projections to obtain an array of filtered projections: 'proj_filtered'.

Step2, Weighting of the filtered projections with weighting factors accounting for nonuniformities in projection distribution to obtain weighted projections: 'proj_wt'.

```
Step 3, Backprojection:
    for  m=1:N
        for n=1:N
            for k=1:N
                Compute the 3D coordinates [x,y,z]of the point [m,n,k].
                x=(m-N/2)*d_of _object;          % d_of_object is the sampling interval of the object.
                y=(n-N/2)*d_of_object;
                z=(n-N/2)*d_of_object;
                    for ii=1:Q
                        t=x*GX(ii)+y*GY(ii)+z*GZ(ii);      %compute the projection address
                        Do interpolation to get proj_wt(t,ii)
                        object(m,n,k)=object(m,n,k)+proj_wt(t,ii);
                    end
            end
        end
    end
```

In the pseudo-code, the complexity of different interpolation methods influences the speed of the reconstruction process. Zero-rank interpolation is the fastest interpolation method, whereas

cubic spline interpolation is very slow. A compromise is to use linear interpolation, which consist of 3 addition and 2 multiplication operations.

To quantify the time complexity of the backprojection process, we assume that a multiplication operation is equivalent to 4 addition operations. The time complexity of the backprojection process is $O(N^3Q)$, for it requires the equivalent of $15N^3 + 26N^3Q$ addition operations. For example, for the case of 208 projections and a 3D object of size 128*128*128, the backprojection process requires the equivalent of 85,983,440 addition operations. EPR image reconstruction for experimental data using the parameters from the above example requires 83 seconds for the backprojection process, which is not acceptable for real-time reconstruction.

3. The GPU Acceleration Program and the Resulting Speedup Effect

A GPU has hundreds of processor cores, which can simultaneously perform many operations in parallel. Because of its advantages in parallelizable computations and cheaper price compared to computer cluster or super computer, the GPU has often been a good choice for high performance computing, especially for SPMD (Single Program Multi data) problems [11].

For host application we used Matlab programming environment. Host application performed data preprocessing and executed CUDA GPU code written in C code for backprojection.

The part of the host program written in MATLAB language responsible for interaction between host and GPU kernel is shown below.

H.1) ***object=zeros(N,N,N);*** define a 3-D array to store the reconstructed object.

H.2) ***k=parallel.gpu.CUDAKernel('EPRI_Kernel.ptx', 'EPRI_Kernel.cu');*** define a CUDA kernel

H.3) ***k.ThreadBlockSize=N;*** set up the size of Block

H.4) ***k.GridSize=[N N];*** set up the size of Grid

H.5) ***object_GPU=feval(k, object, proj_wt, GX, GY, GZ, d_object, N, d_proj, Q, S);*** Execute CUDA kernel.

H.6) ***object=gather(object_GPU);*** Collect results from GPU Memory to Host Memory.

The kernel program of backprojection process is shown below.

```
__global__ void EPRI_Kernel(double *object, double *projection, double *GX, double *GY, double *GZ,
                            double d_object, double N, double d_proj , double Q, double S)
{
    double x, y, z,t,t0,value1,value2;   %define variables to use in the kernel function
    int kk,m,n,k,t1,t2;
    m=threadIdx.x;                       %select the data and execution unit according to Thread struct
```

```
K.4)  n=blockIdx.x;
K.5)  k=blockIdx.y;
K.6)  x=(m+1-N/2)*d_object;          %get the 3-D coordinate of the reconstructed points
K.7)  y=(n+1-N/2)*d_object;
K.8)  z=(k+1-N/2)*d_object;
for(kk=1;kk<=Q;kk++)     %reconstruct a point by summing all the filtered and weighted projections
{
   K.9)   t=x*GX[kk-1]+y*GY[kk-1]+z*GZ[kk-1];    %calculate the projecting address
    K.10)  t0=t/d_proj;    %get the discrete coordinate of the projecting address
    K.11)  t1=floor(t0);   %get the floor integer of the projecting address
    K.12)  t2=ceil(t0);     %get the ceil integer of the projecting address
    K.13)  t0=t0+(S/2);   % adjust the real discrete coordinates to be array index
    K.14)  t1=t1+(S/2);
   K.15)   t2=t2+(S/2);
   if(t1>=0&&t2<=S-1)   %linear interpolation
     K.16)  {value1=projection[(int)((kk-1)*S+t1)];
    K.17)   value2=projection[(int)((kk-1)*S+t2)];
    K. 18)   object[(int)(k*N*N+n*N+m)]+=value1+(value2-value1)*(t0-t1);}
  }
}
```

For comparing the speeds of different backprojection implementations, we designed 3 programs. The first one is a GPU-based single-stage 3D FBP algorithm (GS), the second one is a CPU-based multi-stage 3D FBP algorithm (CM), and the third one is a CPU-based single-stage 3D FBP algorithm (CS).

As a test object we used a bottle phantom. It was imaged using 208 projections distributed using uniform solid angle pattern. For every program, objects with sizes of 64^3, 100^3, 128^3, 150^3 and 200^3 are reconstructed.

The speed comparisons are shown in Fig. 1, 2 and 3 as well as Table 1.

4. Results and Discussion

4.1 Speedup effect and analysis of precision

From Fig. 1 and Table 1, we can see that the GS method provides appreciable reconstruction acceleration of up to a factor of 3523 compared to the CS method when transferring time is not considered. Transferring the computation result from the device (GPU memory) to the host (PC

memory) is a time consuming process. However, even if the time for the process is considered, the GS method is 208 times faster than the CS method. For current EPRI, this acceleration of over a factor of 200 is sufficient for real-time reconstruction. For example, one 128^3 object can be reconstructed in only 0.4s using the GS method, whereas 83s are required to reconstruct using CS method.

Multistage backprojection can reduce the complexity of the back projection process. However, as the size of the reconstructed object becomes large, the CM method becomes quite slow. From Fig. 1 and Table 1, it can be seen that the acceleration factor for GS vs. CM, is about 59 when transfer time is not considered and is 3.4 when considering transfer time. While this may argue that the GS method does not have a big speed advantage compared to the CM method, the CM method can only be used for the case of uniform linear angle sampling. Therefore, by comparison, the GS method is a fast and flexible backprojection algorithm.

From Fig. 2, we can see that the backprojection time is directly proportional to the size of the object (number of voxels) for the CS, CM, and GS (including transfer time) methods. However, the GPU backprojection computing time (GS method not including transfer time) is not proportional to the size of the object. For example, GS reconstruction time (including transferring time) for an object of size 200^3 is about 31 times that for an object of size 64^3, while GPU computing time for an object of size 200^3 is only about 5 times that for an object of size 64^3. This is a further proof that the GPU provides increasing acceleration with increasing size of the reconstructed object.

Note that total time for the GPU backprojection process, including data transferring time from the GPU memory to the PC memory, is also directly proportional to the size of object. Therefore, the acceleration factors for GS vs. CS and GS. vs. CM, considering transferring time, are not sensitive to the object size, which can be seen from Fig.3. For all object sizes, the average acceleration factor for GS vs. CS, considering transferring time, is about 208 and that for GS vs. CM is about 3.4.

The GPU can use double precision floating numbers, so the reconstructed object using the GS method should be the same with the reconstructed object using the CS method. However, the image precision of CM is a little lower than that of GS and CS, due to interpolation during the conversion from a uniform solid angle projection distribution to a uniform linear angle projection distribution, which can be seen in Fig. 4 and 5.

4.2 GPU parameters

For GPU CUDA program design, the GPU parameters are an important consideration because each type of GPU has a unique set of parameters. Table 2 [20][21] contains important parameters, which have important impact on the GPU speedup effect. The parameters given are for the unit used in this work. In the Table, SM means streaming multiprocessor and SP means streaming processor.

4.3 Design method

The Matlab host program design steps are shown below.

Step 1. define the CUDA kernel

Step 2. set up the Block size

Step 3. set up the Grid size

Step 4. Execute the CUDA kernel

Step 5. Collect result from GPU Memory to Host Memory.

The CUDA Kernel frame is shown below.

```
__global__ void KernelFunName( Parameters list)
{   (1) Variables declaring.
    (2) Select specific data according to specific thread ID and block ID.
    (3) Core program
}
```

If the host program is written in C language, we must allocate memory space for the GPU memory and copy the data to it before calling the Kernel function. However, we should note that it is not necessary to do this in the Matlab host program. In MATLAB one can directly use CPU variables as the parameters of the GPU Kernel function. If one designs the host program similarly to the C pattern, meaning allocating memory space for the GPU, the GPU acceleration will be reduced.

4.4 Important parameters computing

There are many factors which can impact the GPU acceleration efficiency. In order to achieve optimal efficiency, it is the most important to load maximum threads to the streaming multiprocessors (SM).

In our GPU program, we design the thread structure as shown in program statement H. 3 and 4. If the reconstructed object has a size of N^3, the number of threads in a block is N, and the

size of the grid is $[N, N]$. We should note that block and thread are logical concepts but SM and SP are physical concepts. For a specific GPU, there are logical confinements and physical confinements. Only by setting up the logical parameters optimally under the confinements of the physical parameters, one can obtain the optimal effect.

As an example, let's consider an object of size 128^3. From Table 2, we know that the maximum thread number in a SM here is 2048 and maximum number of blocks in a SM is 16. For the GPU program the thread number used is 128. Since $128 \times 16 = 2048$, the optimal number of threads (2048) will be loaded to the SM. However, for an object size of 64^3, only 1024 threads will be loaded to the SM ($64 \times 16 = 1024$). Therefore, 50% efficiency will be lost. For object size of 200^3, only 2000 threads will be loaded to the SM ($200 \times 10 = 2000\ ;\ 200 \times 11 = 2200 > 2048$). Therefore 48 threads will be idle.

Every SM has specific register memory, local memory and shared memory sizes. They are dynamically assigned to threads or blocks. The local variables of every thread and the shared memory of every block may become a limitation of speedup efficiency. The SM for the GPU card used here has 48KB shared memory, 512KB local memory and 64K 32bit register (equal to 256KB space). Register variables are much faster than memory variables[18]. It is therefore advisable that all local variables are loaded in the register space. The local variables in the Kernel function used here require 68 bytes, which can be calculated from program statement *K.1* and *2*. If 2048 threads were loaded in a SM, then the SM must assign $2048 \times 68B = 136KB < 256KB$ register space to the local variables of the 2048 threads. Clearly, 256KB register space is sufficient in this case, therefore the speedup effect will not be impacted by the available memory resources.

4.5 Application experiences

During the development process, we summarize some experiences which will be discussed below.

(1) Matlab arrays are stored according to column rather than row, which is the storage mode of C.

The majority of CUDA GPU programming literature focuses on CUDA C kernel functions, which are called by a C host program. However, in this work, the host program is written in Matlab which has a different layout of multi-dimensional arrays.

In C language, a multi-dimensional array is stored according to rows, but in Matlab language a multi-dimensional array is stored according to columns.

For example, if there is a 3D array a[M,N,K] in a CUDA kernel function and the host program is C, the index of the element a[m,n,k] is $kMN + mN + n$. However, if the host program is Matlab, the index is $kMN + nM + m$.

(2) It is not necessary to use the function 'gpuArray'.

The GPU function 'gpuArray' can copy variables to the GPU memory and produce GPU variables, which can be used in a GPU kernel function.

Matlab GPU kernel functions can use CPU variables as input parameters, so it is advisable and simpler to use CPU variables as input parameters for the kernel function rather than GPU variables after a transferring process by 'gpuArray'. We have found that use of 'gpuArray' hinders the resulting GPU acceleration.

(3) The function 'gather' is a time consuming function compared to the computing process on GPU.

From Table 1, it can be seen that 'gather' is a time consuming process. For a 128^3 object, the GPU computing time is just 0.022s, yet the 'gather' process requires 0.378s, which is more than 17 times greater than the computing time. Clearly, the transferring process from GPU to CPU is a main factor impacting the speedup efficiency. In this case, the GPU card is connected with the PC through PCIE 2.0 bus. PCIE 3.0 is much faster than PCIE 2.0, so using GPU cards with PCIE 3.0 may provide better efficiency.

(4) The number of threads in a block being an integer multiple of 32 is not important.

Wrap is a concept on execution of threads by a SM. Some manuals suggest that the number of threads in a block should be a multiple of 32 because one wrap has 32 threads. But through many experiments, we find that this is not necessary, i.e. the number of threads in a block does not have a strong impact on the speedup efficiency.

(5) Branch structures should not be used in the Kernel function.

Branch structures will impact the parallel effect of the 32 threads in a block. Therefore, branch structures should be avoided in the kernel function. This is especially important if the lengths of the branches in a branch structure are very different. This must be avoided to maintain an efficient parallel execution. To avoid such branches, we must write multiple kernel functions, each of which executes a particular branch.

(6) The GPU computing times are not direct proportional to the number of threads.

Form Table 2, we can see that backprojection for a 64^3 object takes 0.011s and that backprojection for a 128^3 object takes only 0.022s rather than 0.088s which would be expected from direct proportionality ($\frac{128^3}{64^3} = 8$). Clearly, GPU computing time is not direct proportional to the number of threads. This is a specific advantage of using the GPU program and is not true when using the CPU program.

5. Conclusion

In this work, we analyzed the computational complexity and resulting time requirements of the 3D FBP algorithm, designed a GPU-accelerated backprojection program, analyzed the speedup effect and image quality, proposed a CPU-Matlab host program design method and a GPU-CUDA C kernel program, described how to calculate the GPU thread loading efficiency and register allocation, and summarized our experiences on GPU programming.

If one uses a Matlab host program to call a CUDA C kernel function, one should focus on how to avoid branch structures, how to design the block and gird structure, and how to judge the memory distribution status.

One should note that the Matlab array is stored according to columns and that the function 'gpuArray' is not necessary. Also it is important to note that it is not necessary to restrict thread number in a block to be an integer multiple of 32.

The use of a GPU can help to speed up a SPMD problem enormously, and should therefore be considered as the first choice for the acceleration technic. In the future, attention should be paid to methods accelerating relatively time consuming data transfers between the CPU and GPU to further optimize the GPU reconstruction acceleration.

Acknowledgement: This work is supported by NIH grants (EB002034 and CA98575). Please Prof. Yuhua Qian write the grant number here.

References:

[1] Halpern, Howard J., David P. Spencer, Jerry van Polen, Michael K. Bowman, Alan C. Nelson, Elizabeth M. Dowey, and Beverly A. Teicher. "Imaging radio frequency electron-spin-resonance spectrometer with high resolution and sensitivity for in vivo measurements." *Review of Scientific Instruments* 60, no. 6 (1989): 1040-1050.

[2] Epel, Boris, Chad R. Haney, Danielle Hleihel, Craig Wardrip, Eugene D. Barth, and Howard J. Halpern. "Electron paramagnetic resonance oxygen imaging of a rabbit tumor using localized spin probe delivery." *Medical physics* 37 (2010): 2553.

[3] Epel, Boris, Subramanian V. Sundramoorthy, Eugene D. Barth, Colin Mailer, and Howard J. Halpern. "Comparison of 250 MHz electron spin echo and continuous wave oxygen EPR imaging methods for in vivo applications." *Medical Physics* 38 (2011): 2045.

[4] Som, Subhojit, Lee C. Potter, Rizwan Ahmad, and Periannan Kuppusamy. "A parametric approach to spectral–spatial EPR imaging." *Journal of Magnetic Resonance* 186, no. 1 (2007): 1-10.

[5] Redler, Gage, Boris Epel, and Howard J. Halpern. "Principal component analysis enhances SNR for dynamic electron paramagnetic resonance oxygen imaging of cycling hypoxia in vivo." *Magnetic Resonance in Medicine* (2013): 00-00.

[6] Ahn, Kang-Hyun, and Howard J. Halpern. "Spatially uniform sampling in 4-D EPR spectral-spatial imaging." *Journal of Magnetic Resonance* 185, no. 1 (2007): 152-158.

[7] Ahn, Kang-Hyun, and Howard J. Halpern. "Simulation of 4D spectral-spatial EPR images." *Journal of Magnetic Resonance* 187, no. 1 (2007): 1-9.

[8] Ahn, Kang-Hyun, and Howard J. Halpern. "Comparison of local and global angular interpolation applied to spectral-spatial EPR image reconstruction. "*Medical physics* 34 (2007): 1047.

[9] Ahn, Kang-Hyun, and Howard J. Halpern. "Object dependent sweep width reduction with spectral–spatial EPR imaging." *Journal of Magnetic Resonance*186, no. 1 (2007): 105-111.

[10] Ahmad, Rizwan, Bradley Clymer, Deepti S. Vikram, Yuanmu Deng, Hiroshi Hirata, Jay L. Zweier, and Periannan Kuppusamy. "Enhanced resolution for EPR imaging by two-step deblurring." *Journal of Magnetic Resonance* 184, no. 2 (2007): 246-257.

[11] Pratx, Guillem, and Lei Xing. "GPU computing in medical physics: A review." *Medical physics* 38 (2011): 2685.

[12] Uecker, Martin, Shuo Zhang, Dirk Voit, Alexander Karaus, Klaus-Dietmar Merboldt, and Jens Frahm. "Real-time MRI at a resolution of 20 ms." *NMR in Biomedicine* 23, no. 8 (2010): 986-994.

[13] Uecker, Martin, Shuo Zhang, and Jens Frahm. "Nonlinear inverse reconstruction for real‐time MRI of the human heart using undersampled radial FLASH." Magnetic Resonance in Medicine 63, no. 6 (2010): 1456-1462.

[14] Sorensen, Thomas Sangild, David Atkinson, Tobias Schaeffter, and Michael Sass Hansen. "Real-time reconstruction of sensitivity encoded radial magnetic resonance imaging using a graphics processing unit." Medical Imaging, IEEE Transactions on 28, no. 12 (2009): 1974-1985.

[15] Ryoo, Shane, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73-82. ACM, 2008.

[16] CUDA benchmark suite. http://www.crhc.uiuc.edu/impact/cudabench.html.

[17] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[18] Applied Parallel Programming. http://courses.engr.illinois.edu/ece498al/Syllabus.html

[19] Som, Subhojit, Lee C. Potter, Rizwan Ahmad, Deepti S. Vikram, and Periannan Kuppusamy. "EPR oximetry in three spatial dimensions using sparse spin distribution." *Journal of Magnetic Resonance* 193, no. 2 (2008): 210-217.

[20] Output of the Matlab function 'gpuDevice' running on a PC with Geforce GTX 760.

[21] CUDA. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/CUDA

Table1. Speed comparison of the backprojection process of the three reconstruction programs.

Object Size	**GS** Reconstruction Time (Computing time + gather time = total time) [s]	**CM** Reconstruction Time [s]	**CS** Reconstruction Time [s]	Acceleration Factor **GS vs. CM** *not considering transfer time*	Acceleration Factor **GS vs. CM** *considering transfer time*	Acceleration Factor **GS vs. CS** *not considering transfer time*	Acceleration Factor **GS vs. CS** *considering transfer time*
64^3	0.011+0.049=0.06	0.203	13	18.4	3.4	1181	216
100^3	0.016+0.229=0.245	0.624	44	39	2.55	2750	180
128^3	0.022+0.378=0.400	1.108	83	50.4	2.7	3772	207
150^3	0.029+0.605=0.634	1.910	142	65.8	3.01	4897	223
200^3	0.065+1.442=1.507	8.050	326	123	5.34	5015	216
Average	----	----	----	59.3	3.4	3523	208.4

Table 2. Some parameters of Geforce GTX 760

GPU Version	Geforce GTX 760	Number of SM	6
Global Memory	4GB	Number of cores in a SM	192
Number of cores (SP)	1152	Maximal Number of blocks in a SM	16
Compute capability	3.0	Maximal Number of threads in a SM	2048
Warp size	32	Register space of SM	64K*32bit
Maximal Number of threads in a Block	1024	Shared Memory of SM	48KB
Maximal size of block	[1024 1024 64]	Local Memory of SM	512KB
Maximal size of grid	[2.1475e+09 65535 65535]		

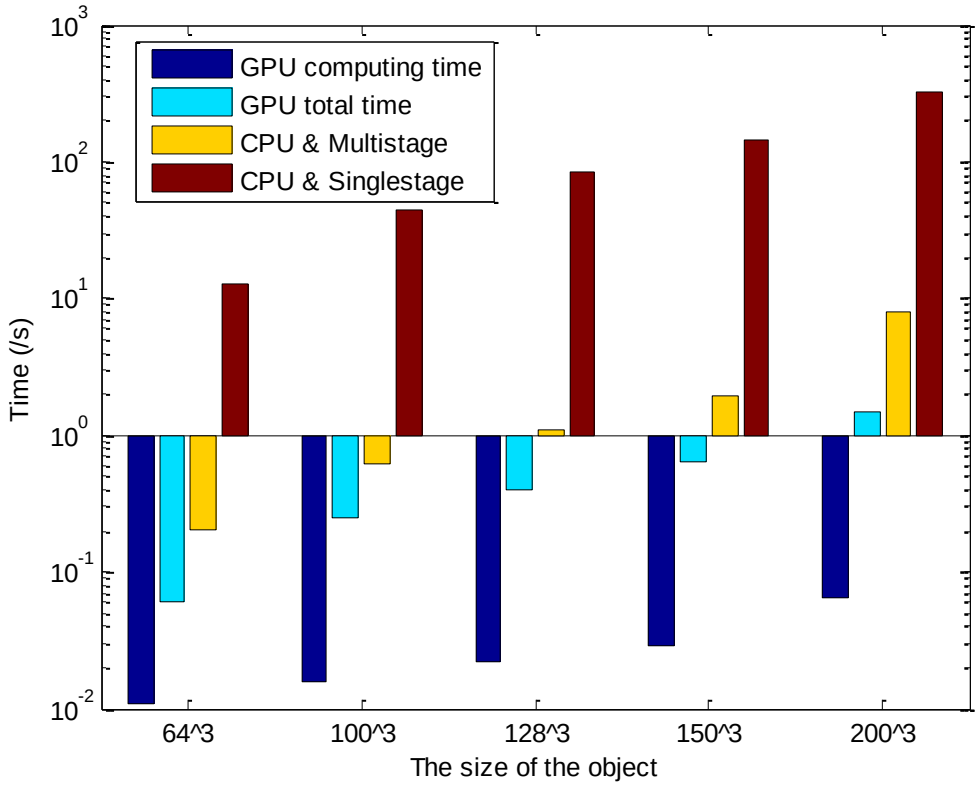

Fig. 1. Comparison of image reconstruction times for different backprojection implementation methods demonstrating significant acceleration with the GPU.

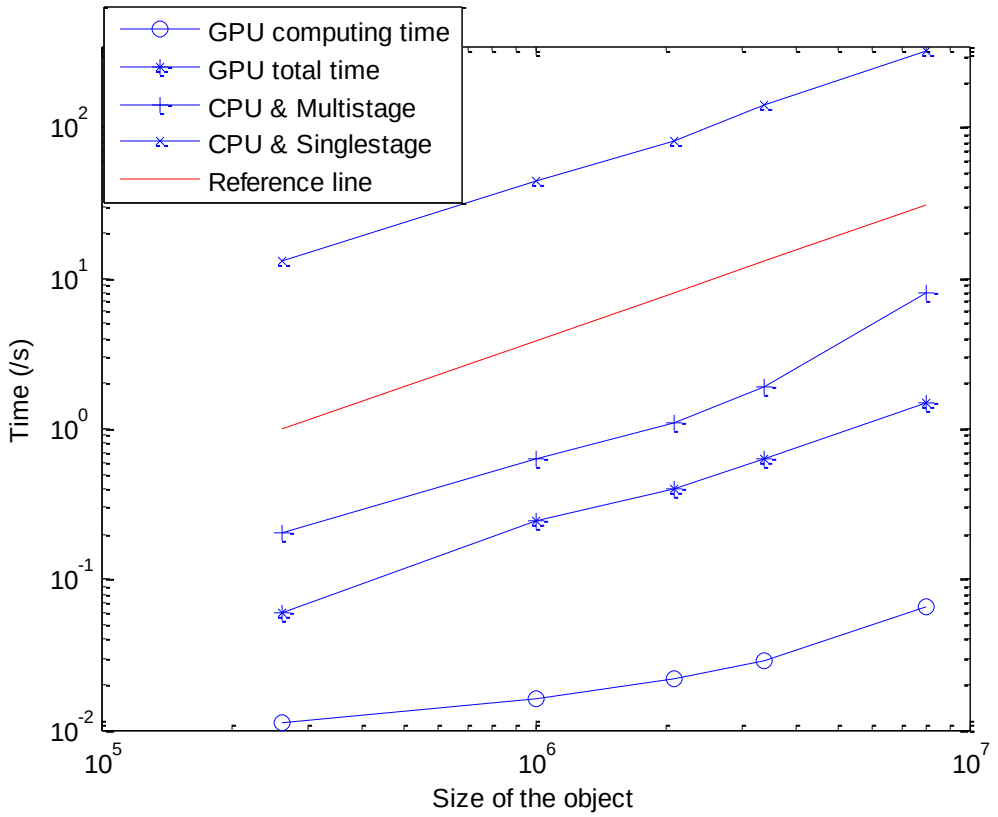

Fig. 2 The reconstruction time trend with the increase of the object size. Note that the x axis and y axis are both logarithmic scale. The red line is a reference line which is a direct proportion function. We can use it to test the four time changing trends intuitively.

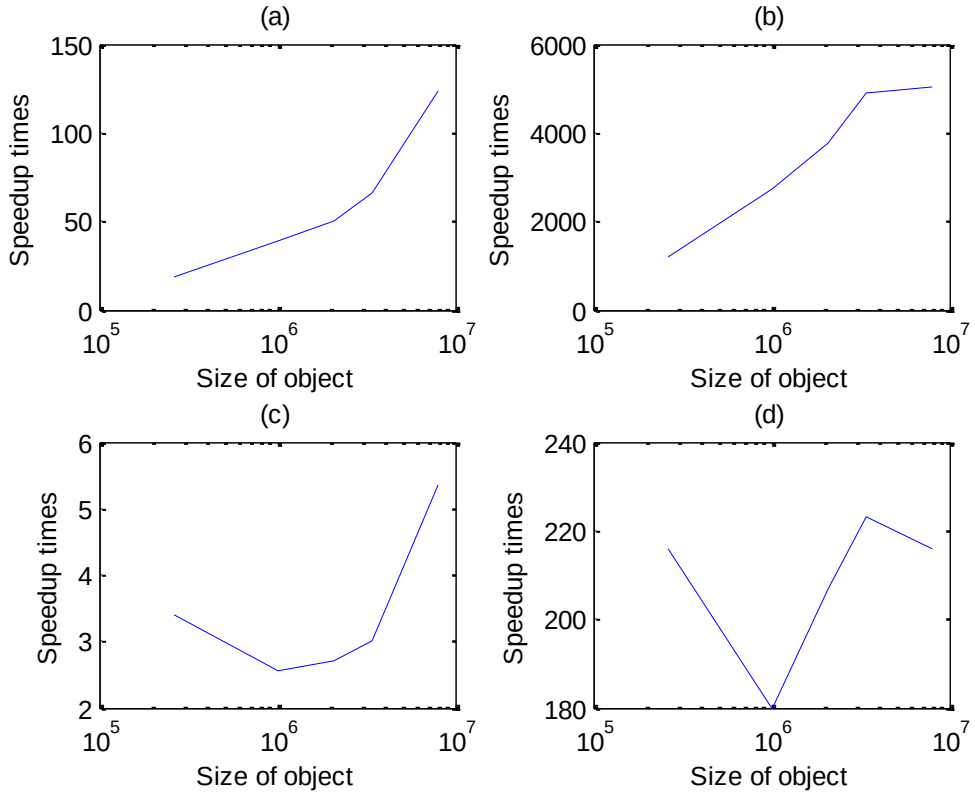

Fig. 3 The acceleration factor trend with increasing object size. Acceleration factors for (a) GS vs. CM without considering transfer time, (b) GS vs. CS without considering transfer time, (c) GS vs. CM considering transfer time, and (d) GS vs. CS considering transfer time.

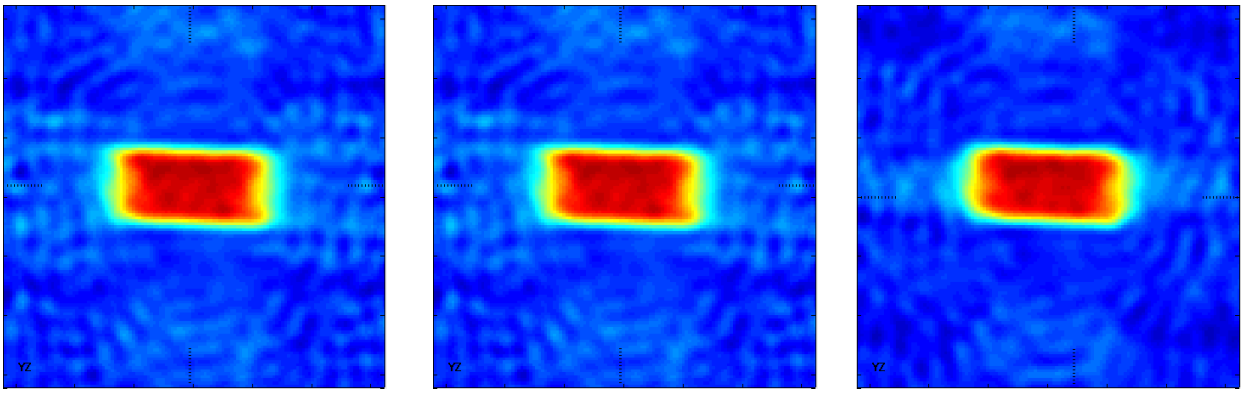

Fig. 4. On the left is a slice from a 3D EPRI image reconstructed using the GS method. In the middle and on the right are slices from 3D EPRI images reconstructed using the CS and CM methods respectively, Slightly decreased precision in the CM reconstructed object can be seen.

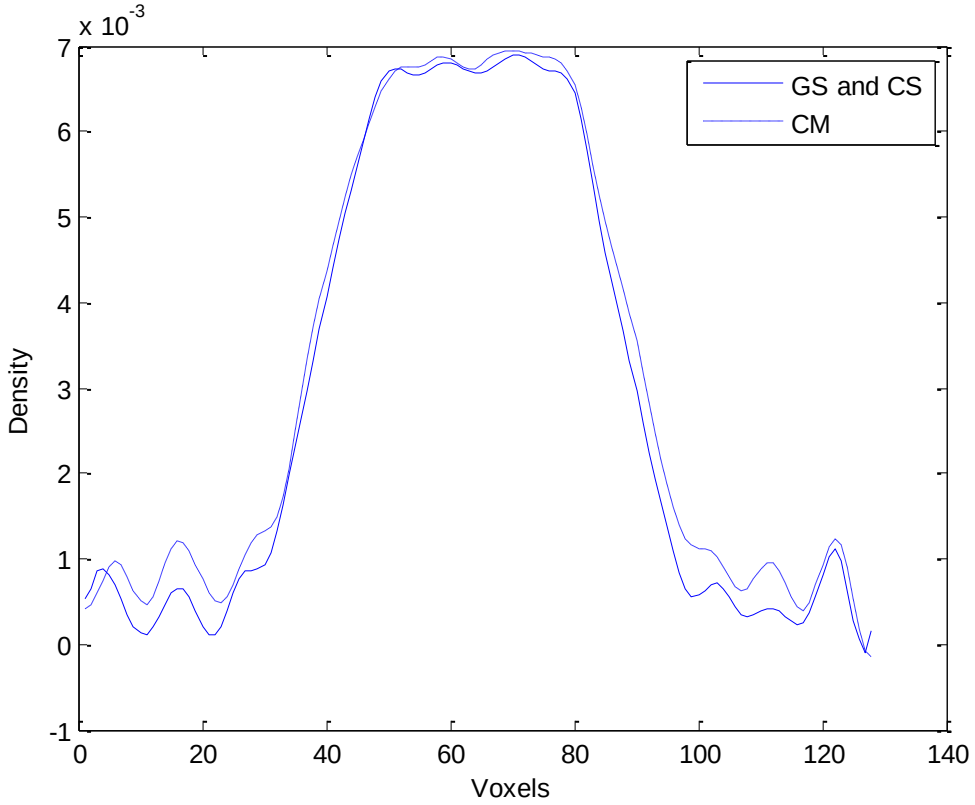

Fig. 5 A comparison of intensity profile across the objects reconstructed by GPU single-stage, CPU single-stage and CPU multistage FBP algorithms.