Int J CARS (2011) 6:187–199
DOI 10.1007/s11548-010-0499-3

ORIGINAL ARTICLE

Accelerating simultaneous algebraic reconstruction technique with motion compensation using CUDA-enabled GPU

Wai-Man Pang · Jing Qin · Yuqiang Lu · Yongming Xie · Chee-Kong Chui · Pheng-Ann Heng

Received: 15 January 2010 / Accepted: 31 May 2010 / Published online: 30 June 2010
© CARS 2010

Abstract
Purpose To accelerate the simultaneous algebraic reconstruction technique (SART) with motion compensation for speedy and quality computed tomography reconstruction by exploiting CUDA-enabled GPU.
Methods Two core techniques are proposed to fit SART into the CUDA architecture: (1) a ray-driven projection along with hardware trilinear interpolation, and (2) a voxel-driven back-projection that can avoid redundant computation by combining CUDA shared memory. We utilize the independence of each ray and voxel on both techniques to design CUDA kernel to represent a ray in the projection and a voxel in the back-projection respectively. Thus, significant parallelization and performance boost can be achieved. For motion compensation, we rectify each ray's direction during the projection and back-projection stages based on a known motion vector field.
Results Extensive experiments demonstrate the proposed techniques can provide faster reconstruction without compromising image quality. The process rate is nearly 100 projections s^{-1}, and it is about 150 times faster than a CPU-based SART. The reconstructed image is compared against ground truth visually and quantitatively by peak signal-to-noise ratio (PSNR) and line profiles. We further evaluate the reconstruction quality using quantitative metrics such as signal-to-noise ratio (SNR) and mean-square-error (MSE). All these reveal that satisfactory results are achieved. The effects of major parameters such as ray sampling interval and relaxation parameter are also investigated by a series of experiments. A simulated dataset is used for testing the effectiveness of our motion compensation technique. The results demonstrate our reconstructed volume can eliminate undesirable artifacts like blurring.
Conclusion Our proposed method has potential to realize instantaneous presentation of 3D CT volume to physicians once the projection data are acquired.

Keywords Simultaneous Algebraic Reconstruction Technique · GPU-accelerated SART · Tomography Reconstruction · Motion Compensation for Tomography Reconstruction · CUDA-enabled GPU acceleration

W.-M. Pang
Spatial Media Group, Computer Arts Lab, University of Aizu, Aizuwakamatsu, Japan

J. Qin (✉)
Department of Diagnostic Radiology, National University of Singapore, Kent Ridge, Singapore
e-mail: jqin@cse.cuhk.edu.hk

Y. Lu
Shenzhen Institute of Advanced Integration Technology, Chinese Academy of Sciences/The Chinese University of Hong Kong, Shenzhen, China

Y. Xie · P.-A. Heng
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong

C.-K. Chui
Department of Mechanical Engineering, National University of Singapore, Kent Ridge, Singapore

Introduction

Algebraic reconstruction technique (ART) [1] and its improved variation simultaneous algebraic reconstruction techniques (SART) [2] are well-known computed tomography reconstruction methods which view the result image values as an array of unknowns, and then solve them by formulating algebraic equations with a given set of projections acquired from tomography techniques. Compared with commonly used transform-based methods such as the Filtered Back-Projection (FBP) algorithm [3], ART and SART usually *perform better in terms of reconstruction quality*, especially when the available projections are sparsely and non-uniformly distributed in angles or the energy propagation paths between the source and receiver positions are subjected to ray bending because of refraction [4].

However, algebraic reconstruction techniques are not commonly employed in many medical applications requiring instantaneous image generation, due to their high computational complexity. For example, in the SART, the solving of algebraic equations requires a sequence of alternating volume projections and corrective back-projections to converge to the final reconstructed volume. This process is *time-consuming and difficult to complete instantaneously*, especially for high-resolution volumes and projections. To achieve realtime 3D images, reconstruction speed is suggested to be 30–50, projections per second [5]. This cannot be easily satisfied by SART implementation on contemporary CPUs, which usually requires hundreds or even thousands of seconds [6].

In this paper, we proposed an acceleration approach for SART based on the use of a consumer level GPU. We exploit the latest Computer Unified Device Architecture (CUDA) platform to implement our approach. Two core techniques are proposed to facilitate SART in the CUDA architecture: (1) *a ray-driven projection along with hardware trilinear interpolation*, and (2) *a voxel-driven back-projection that can avoid redundant computation by combining CUDA shared memory*. We also integrate an efficient compensation method to reduce artifacts caused by motion at the moment of acquiring projections. Significant performance boost and encouraging image quality are reported from experiments.

In the following sections, we first review some of previous works related to the topic of this paper. Then, we introduce basic knowledge on algebraic reconstruction algorithms, followed by implementation details of proposed methods, experiment results, as well as conclusions.

Related work

Hardware-accelerated CT reconstruction

There have been many attempts to accelerate algebraic algorithms with dedicated special purpose hardware platforms like digital signal processors (DSPs) [7], field programmable gate arrays (FPGAs) [8], and Cell BE processors [9]. While the performance is satisfactory, the capability in programming of these special devices is limited, and therefore modifications and adaptations are difficult. More importantly, these devices are not commonly available in consumer level PCs, which constrains the flexibility and extensibility of reconstruction techniques.

In recent years, more researchers have become aware of the highly parallelized computing power of GPUs for general purpose computations. In the context of tomography reconstruction, the earliest attempt dates back to 1994, when Calbral et al. utilized a custom texture mapping hardware for cone-beam CT reconstruction with filtered back-projection on a fixed point precision non-programmable high-end SGI workstation [10]. Chidlow and Möller implemented emission tomography with ordered subsets expectation maximization on a Nvidia GeForce 4 GPU card [11], which has similar limitations to Calbral's SGI solution. The standard texture mapping feature in GPUs was used to accelerate the SART algorithm [12], and it takes 36 seconds to reconstruct a 128^3 volume from 80 projections. By exploiting programmable GPUs, Riabkov et al. accelerated the back-projection step of the FDK cone-beam algorithm [13], and Schiwietz et al., tried to speed up the back-projection and FFT operations employed for MR k-space transfer [14]. Although both papers reported realtime performance, they are both transform-based algorithms and are not comparable to our SART solutions. In addition, most of these reconstruction methods were based on graphics-based approaches using OpenGL or shading languages, which many "non-graphics" programmers may find cumbersome.

Until recently Scherl et al. used the programmer friendly Compute Unified Device Architecture (CUDA) to implement the filtering and back-projection of the FDK algorithm on GPU [15]. It takes about 12 seconds to reconstruct a 512^3 volume from 414 projections. Okitsu et al. used a CUDA-enabled GPU to accelerate the entire FDK algorithm by two techniques: offchip memory access reduction and memory latency hiding [16]. Only 5.6 seconds are needed to reconstruct a 512^3 volume from 360×512^2 projections. However, both papers focus on FDK methods while the objective of our work is to accelerate algebraic algorithms. Note that there are fundamental differences in the reconstruction quality for various approaches. For example, the FDK algorithm commonly suffers from poorer reconstruction quality than the SART method [4]. However, most implementations of ART-based method require substantial processing time, which is undesirable in time-critical surgical and minimally invasive procedures. The necessity of a realtime SART implementation is therefore obvious.

Motion compensation in CT reconstruction

In most of the mentioned GPU-based CT reconstruction methods, the target object is assumed static during the acquisition. Motion such as respiration or cardiac motion during the acquisition of the projections are not taken into consideration. In medical practice, such motion may cause artifacts in the reconstructed images such as blurs, streaks, and bands, leading to misinterpretation and imprecise diagnosis, especially when small lesions are to be detected [17]. For example, significant tumor motion has been reported in several studies [18,19]. Nehmeh et al. also reported a significant volume increase in lung lesions in image reconstruction without respiratory motion compensation [20]. In this regard, to ensure the quality of reconstructed images, it is necessary to integrate motion compensation algorithm in our CUDA-based solution.

Overview of algebraic reconstruction

Figure 1 illustrates the main components involved in the algebraic tomographic reconstruction formulation. The target volume V at the center is superimposed with a 3D grid, and there is a total number of N voxels. Each of these voxel values v_i is assumed constant across time. Then, the objective of reconstruction is to estimate all v_i from a set of projections P acquired from different X-ray source directions along a circular trajectory. Similarly, each projection is discretized as pixel values p_j, while p_j is also the sum of voxel values passed by the jth ray. As a result, the relationship between v_i and p_j can be expressed as

$$\sum_{i=1}^{N} w_{ij} v_i = p_j, \quad j = 1, 2, \ldots, M, \tag{1}$$

where w_{ij} represents the contribution of the ith voxel to the jth ray integral. M is the total number of rays from all projections. Thus the reconstruction problem is transformed into an algebraic equation, which can be simply expressed as

$$\mathbf{WV} = \mathbf{P}. \tag{2}$$

In practice, N and M are usually so large that conventional matrix methods cannot solve the equation efficiently.

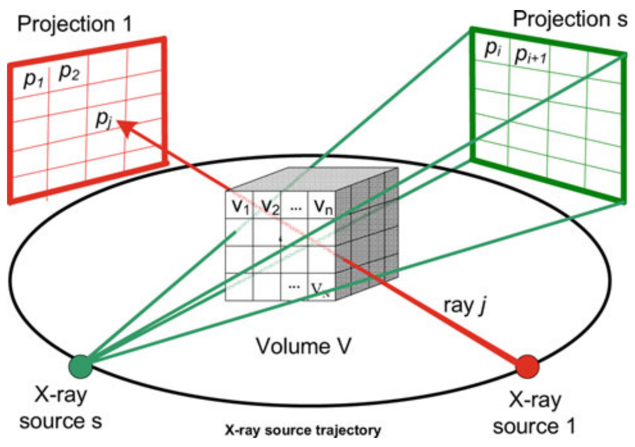

Fig. 1 Illustration of algebraic tomographic reconstruction formulation

Gordon et al. proposed an iterative scheme [1] based on the famous “method of projections”, first proposed in [21], which iteratively corrects the voxels in a ray by ray manner. Later, Andersen and Kak [2] noticed that the correction can be simultaneously performed on all the rays in one projection and this can avoid the salt and pepper noise and striping artifacts caused by the ray by ray correction manner. So SART is proposed. In SART, an iterative implementation can be carried out by employing the following formula:

$$c_i = \frac{p_i - \sum_{n=1}^{N} w_{in} v_n^{(k)}}{\sum_{n=1}^{N} w_{in}} \tag{3}$$

$$v_j^{(k+1)} = v_j^{(k)} + \lambda \frac{\sum_{i \in I_\varphi} (c_i) w_{ij}}{\sum_{i \in I_\varphi} w_{ij}} \tag{4}$$

where $v_j^{(k+1)}$ and $v_j^{(k)}$ are the jth voxel value at $k+1$ and k step; λ is a relaxation factor; P_s represents projection s; p_i and $\sum_{n=1}^{N} w_{in} v_n^{(k)}$ are the real ray-sum values of ith ray in projection data and line integral along ith ray, respectively; and w_{ij} is the weighted factor. We recommend readers to refer to [4] for the derivation details of ART and SART iterative formulas.

Note that in the above equations, we always assume the voxel values are constant across time. However, due to respiration and cardiac motion, there is no guarantee that this assumption is always valid. Thus, the matrix W should comprise not only the cone-beam transformation between the volume and projections but also the motion of the volume. To tackle this problem, our solution is integrated with a motion compensation algorithm to determine W. More detail is given in section “Motion compensation with inverse mapping”.

CUDA-accelerated algebraic reconstruction

An overview of the proposed approach is shown in Fig. 2. The process is iteratively applied to obtain the final volume

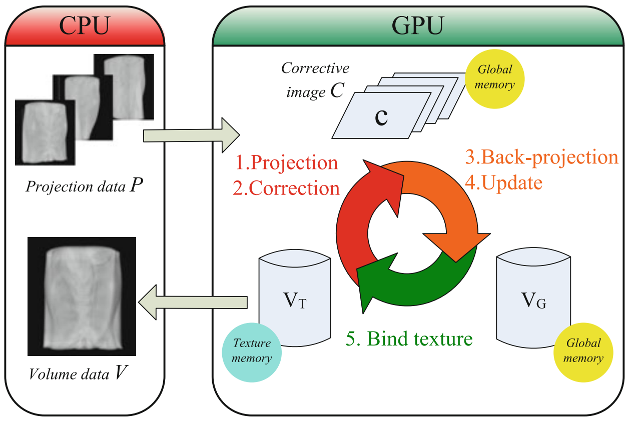

Fig. 2 Flowchart of the proposed CUDA-enabled GPU approach

data V. In each iteration, there are four main steps: projection, correction, back-projection, and update. We choose a CUDA-based ray-driven method for the projection and correction stages, while using a voxel-driven technique for the back projection and image update procedures. Both methods favor current graphics hardware architecture to avoid global summation which usually reduces the parallelism and degrades GPU performance.

As a result, the first two steps can be combined in implementation, since the corrective image is now computed in a ray by ray manner which matches with the projection procedure. These correspond to Eq. 3, which involves the computation of c_i (corrective image). Similarly, the last two steps can also be combined due to the fact that the image update can be applied to a voxel once its corresponding back-projection is completed. These processes update voxel value v_j based on c_i as in Eq. 4.

The whole process starts with loading of the projection data P from the CPU to global memory in the GPU, together with an initialization of volume V_T in the texture memory. As the volume V_T stores the most updated volume, it is initialized with all-zero. Then, the volume V_T is projected in the viewing angle consistent with projections P, so that their difference forms the corrective image C. Later, the corrective image C is back-projected to the volume V_G in global memory to update the volume. Storing the updated volume in global memory V_G is due to the limitation on read-only access of the texture memory. Later, V_G is copied back to V_T in texture memory. The process repeats for other projections until the volume V_T converges, and we can transfer the volume back to the CPU for output.

Figure 3 shows the intermediate results extracted when increasing number of iterations are performed for reconstruction of the CT CHEST data. The results are generated from 80 projections each with 4.5-degree difference in viewing angle, and go through 10 iterations with parameters $\lambda = 0.3$ and $\epsilon = 0.005$. We can find that when the number of projections accumulated increases, the reconstructed volume becomes more and more complete. When number of projections reached about 240, the resultant volume starts to converge and the reconstruction quality is rather satisfactory. We also give a result of 800 projections in Fig. 3 for reference. This demonstrates that our proposed projector and back-projector pairs converge properly and quickly.

Projection and corrective image computation

In the projection step, the classical Siddon's method [22] or the ray-tracing-based method are commonly employed. The major concern regarding Siddon's method is that it is a voxel-based method and the accuracy heavily depends on the resolution of the volume. When the voxel sampling is coarse, it may lead to a discontinuity or aliasing problems.

In contrast to the Siddon's method, the ray-tracing-based method samples along a ray passing through the volume. The sampling points are chosen between the entry point and the exit point in equal intervals. The intensity of each sampling point is computed using trilinear interpolation of the surrounding voxel values. All sampled intensities are then integrated to obtain the projected value. Therefore, the sampling is independent of the volume resolution and usually avoids artifacts in the projection result.

However, the ray-based method has its own drawback: the trilinear interpolation operations are time-consuming, especially when the number of rays is large. Therefore, the Siddon's method is more popular in many previous works. To overcome this drawback, we take advantages of the new features of CUDA 2.0 which support 3D texture and the capacity

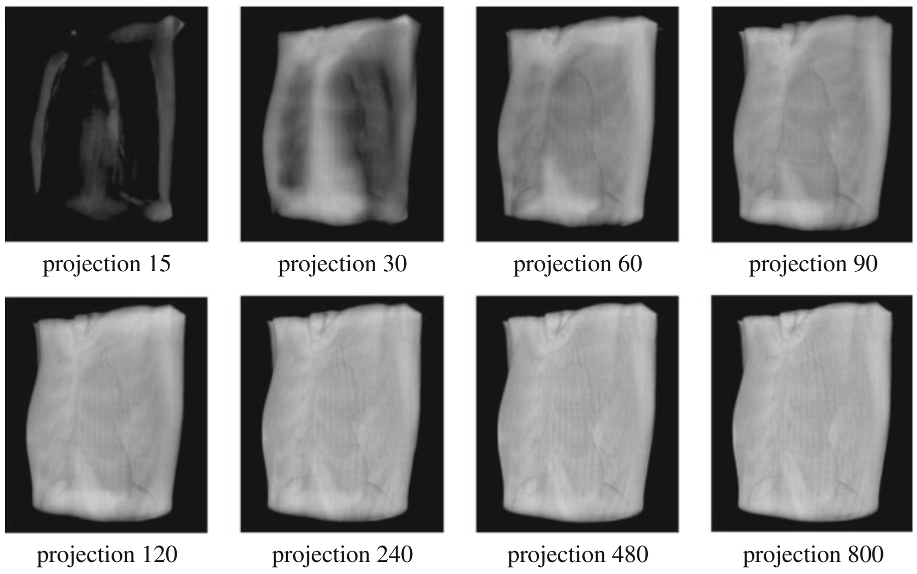

Fig. 3 Intermediate results during reconstruction of CT CHEST data. A total number of 80×10 projections is used, with $\lambda = 0.3$ and $\epsilon = 0.005$

Algorithm 1 Pseudocode of CUDA kernel doing projection and correction. Each kernel runs for a particular ray.

```
1: /* boxMin, boxMax : the boundary of volume.
   intersectBox : function to detect a ray has pass through a box.
   ray : a ray from the source to a pixel in projection plane.
   accumProj : accumulated intensity along a ray
   texProj : the current converging 3D texture
   $\epsilon$ : sampling interval
   proj : the 2D projection data (p)
   corrImg : texture to store the corrective image
   */
2: boxMin $\leftarrow$ (-1,-1,-1); boxMax $\leftarrow$ (1,1,1) ;
3: hit $\leftarrow$ intersectBox(ray, boxMin, boxMax);
4: if hit then
5:   // accumulate intensity along ray from far to near
6:   accumProj $\leftarrow$ 0; t $\leftarrow$ 0
7:   for i = 0 to maxSample do
8:     pos $\leftarrow$ ray.farPos - ray.dir $\times$ t;
9:     sample $\leftarrow$ tex3D(texProj, pos); // read from 3D texture
10:    accumProj $\leftarrow$ accumProj + sample;
11:    t $\leftarrow$ t + $\epsilon$;
12:  end for
13:  accumProj $\leftarrow$ accumProj $\times\epsilon$;
14:  corrImg[ray.ID] $\leftarrow \frac{\text{(proj[ray.ID] - accumProj)}}{\text{ray.Length}}$
15: end if
```

of hardware built-in trilinear interpolation. As a result, thousands of trilinear interpolation can be performed within a second.

In our CUDA implementation, each kernel corresponds to a ray, and all kernels run in parallel threads within GPU. Therefore, the kernel for projection computes both entry and exit points of the ray passing the volume, and then, accumulates the values, sampled in an interval of ϵ, to calculate the projection value of this ray as the term $\sum_{n=1}^{N} w_{in} v_n^{(k)}$ in Eq. 3. Note that usually $\sum_{n=1}^{N} w_{in}$ is taken as the ray length (i.e. distance between entry and exit point). The pseudocode of the corresponding CUDA kernel is provided in Algorithm 1.

Back-projection and image update

The critical issue of back-projection and image update is how to find the rays passing through a specified voxel, as exhaustively trying out all ray-voxel combinations leads to a complexity of O(N×M), which is computationally heavy and not really necessary. We solve this using similar method to Li et al. [23] by quickly filtering undesired rays. As it is found that the projection of a voxel (a small cube) on a plane can only be a hexagon, a pentagon or a quadrilateral, and the rays passing through a voxel must also have its projected point within these polygons on the plane.

As demonstrated in Fig. 4, we first calculate the projection of the eight vertices of the voxel, and form a minimum bounding rectangle (MBR) of these vertexes. Only the rays whose projection point P locates inside the MBR are tested to figure out whether they hit the voxel or not. This significantly improves performance by culling out a large number of unwanted rays.

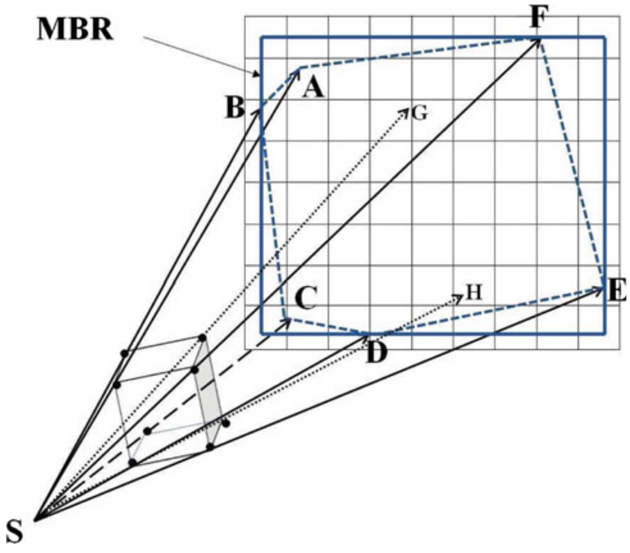

Fig. 4 MBR is employed to find the rays which pass through the specified voxel

One may notice the redundant computation of vertex projection is due to the fact that a vertex is commonly shared with 8 voxels. To make our computation efficient, we utilized the shared memory of CUDA-enabled GPU to store the projection of each vertex. Since the shared memory units can only be accessed by threads in the same block, a special structure of each block is required. In our implementation, a 3D block is formed with a dimension of 4 × 4 × 4. In this case, we only have to calculate the projection of their corresponding vertexes, that is 5^3 vertexes. The pseudocode of the corresponding CUDA kernel is provided in Algorithm 2.

Motion compensation with inverse mapping

If the volume remains static during acquisition of the projections, the matrix W in Eq. 2 is obtained by modeling only the transformation between the volume and the set of projections in terms of the cone-beam sources and angles. Unfortunately, in many situations, organs and tissues are unable to remain still for involuntary motions like respiration and cardiac motion. Therefore, the matrix W should not only consider the cone-beam transformation but also include motion transformation.

We integrate a simple but efficient algorithm [24,25] based on inverse mapping into our proposed implementation. As the volume now varies in the time domain, so do the acquired projections. In this regard, in case that we want to reconstruct a volume $V_{t_{\text{ref}}}$ at time step t_{ref}, we need to rectify the projections obtained during other time steps t. The rectification

Algorithm 2 Pseudocode of CUDA kernel for back-projection and update. Each kernel processes a particular voxel

```
1: /* λ : relaxation factor
   ray : a ray from the source to a pixel in detector plan.
   voxel : currently processing voxel
   mbr : the minimum bounding rectangle
   ComputeMBR : Compute the MBR based on voxel position
   accumErr : accumulated error of all rays via the voxel
   accumWeight : accumulated weight (w) of all rays
   */
2: accumError ← 0; accumWeight ← 0; λ ← 0.1;
3: mbr ← ComputeMBR(ray,voxel.Pos);
4: // only loop for all rays within MBR
5: for x = mbr.MinX to mbr.MaxX do
6:   for y = mbr.MinY to mbr.MaxY do
7:     ray.dir ← normalize(x,y,Cam.focalLength);
8:     hit ← intersectBox(ray, voxel.boxMin, voxel.boxMax);
9:     if hit then
10:       ray.ID = GetRayID(x,y);
11:       accumErr ← accumErr + ray.Length × corrImg[ray.ID];
12:       accumWeight ← accumWeight + ray.Length;
13:     end if
14:     voxel.intensity ← voxel.intensity + λ × $\frac{accumError}{accumWeight}$;
15:   end for
16: end for
```

requires a motion field or motion model $\Psi_{t_{\text{ref}}\to t}$ of the volume from time t_{ref} to t to be known in advance. In practice, the motion model is usually determined by non-rigid registration performed on two sets of images acquired at breath holding in expiration and inspiration. Please refer to [25] for the details of the process. We assume here such a preprocessing step is performed in advance and all the motion fields are known. As the motion fields are transitive, if we can have all motion fields between t and $t+1$ (i.e. $\Psi_{t\to t+1}$), we may obtain any $\Psi_{t_{ref}\to t}$ by summing all in between motion fields.

In our implementation, the rectification of projections is not directly applied in the image domain, but performed during the SART algorithm. Modifications on both the projection and back-projection steps are required. As the motion vector field Ψ is known, we can map a voxel position x at reference time t_{ref} to its corresponding position x' at time t. We can simply write the relationship as follows:

$$\begin{aligned} x' &= \Psi_{t_{\text{ref}}\to t}(x), \\ x &= \Psi^{-1}_{t_{\text{ref}}\to t}(x') = \Psi_{t\to t_{\text{ref}}}(x'), \end{aligned} \tag{5}$$

where Ψ^{-1} is the inverse of the motion field. Then, during the projection stage, the sampling points along the ray are being rectified by the inverse motion Ψ^{-1}. Correspondingly, in the back-projection, the ray direction is adjusted with the motion field Ψ. As a result, the motion difference between t_{ref} and t is compensated.

A few modifications on the projection and back-projection kernels are required. In the projection kernel, we need to read from the 3D texture storing the motion vector field Ψ to offset the ray positions. For instance, in Algorithm 1, we should modify line 8 to offset variable "pos" by the motion field. Similarly, in the back-projection kernel, we need the inverse motion Ψ^{-1} to rectify the variable "mbr" and "ray" in line 3 of Algorithm 2. The source codes of key modules of the proposed CUDA-based implementation are attached as Appendix 2.

Results

In order to evaluate the performance of the proposed method, we carried out a series of experiments. First, we compared the time performance between our CUDA-based SART and CPU-based SART as well as tested the time performance of our method on various numbers of projections and projection sizes. Second, we quantitatively evaluated the image quality of the proposed solution with various datasets. Third, we conducted a set of experiments to investigate the effects of main parameters on the reconstructed image quality. Fourth, we performed experiments to compare the reconstructed results with and without motion compensation. All experiments were conducted on a desktop PC equipped with a Core2 Quad Q6600 CPU (2.4 GHz), 4 GB memory and an nVIDIA 8800 GTX GPU with 768 MB RAM and CUDA 2.0.

Time performance

Table 1 shows the time statistics for reconstruction of the Shepp-Logan phantom. A CPU implementation takes nearly 18 min (1,121 s) to reconstruct a 128^3 volume from a set of 80 projections with 128^2 in size. In contrast, our CUDA-based method takes only 7.52 s with 10 iterations. This is equivalent to over 100 projections per second (pps) and nearly 150 times

Table 1 Execution time of reconstruction process using 80 or 160 projections and 10 iterations, with CPU and our CUDA-based implementation, and with different numbers of projections and projection sizes

Method	No. of projections	Projection size	Time (s)
CPU-based	80×10	128×128	1112.1
Our GPU approach	80×10	128×128	7.52
		256×256	11.25
		512×512	22.97
		768×768	46.88
		1024×1024	68.28
Our GPU approach	160×10	128×128	15.31
		256×256	22.35
		512×512	46.25
		768×768	93.6
		1024×1024	137.03

All GPU versions run on the nVidia 8800GTX faster than CPU counterpart. For larger projections with size of 512^2 and 1024^2, the reconstruction can be done within 47 and 138 s with up to 160 projections, which are much faster than CPU-based SART and can fulfill the requirement of many instantaneous clinical applications. These reveal that our proposed approach can efficiently accelerate the SART method.

Reconstruction quality

While achieving significant reduction in reconstruction time, the image quality should not be sacrificed. We test our method on various datasets. Figure 5 shows the reconstruction result by our method on the CT CHEST dataset. In this experiment, We use 80 projections and 10 iterations and the lambda is set as 0.1. The original slice is also presented alongside for ease of comparison. We can find that they are visually non-differentiable.

Quantitative measurements of the reconstruction quality are essential for a more objective evaluation. We first compare the ground truth and reconstructed images in Fig. 5 with peak signal-to-noise ratio (PSNR). The formula of PSNR we used are stated in Appendix 1 for reference. A high PSNR of 50.57 dB is reported, which suggests that they are very similar to each other. We also examine our solution on images with both low contrast and high contrast. Figures 6 and 7 show slices of the original data and reconstructed data as well as the line profile of CRANI and SKULL data, respectively. The line profiles chosen commonly across some significant regions in the data (indicated in yellow in original slices).

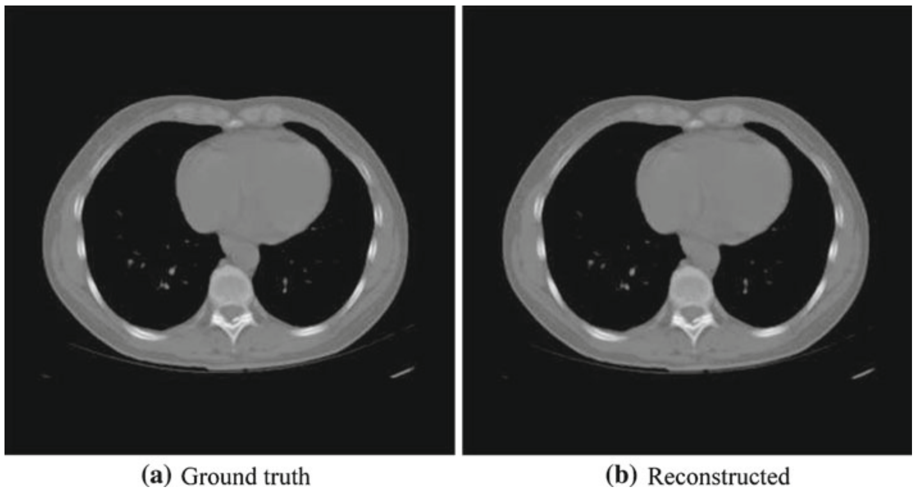

Fig. 5 Slices across the ground truth and reconstructed volume data of CT CHEST dataset. Peak signal-to-noise ratio (PSNR) = 50.57 dB

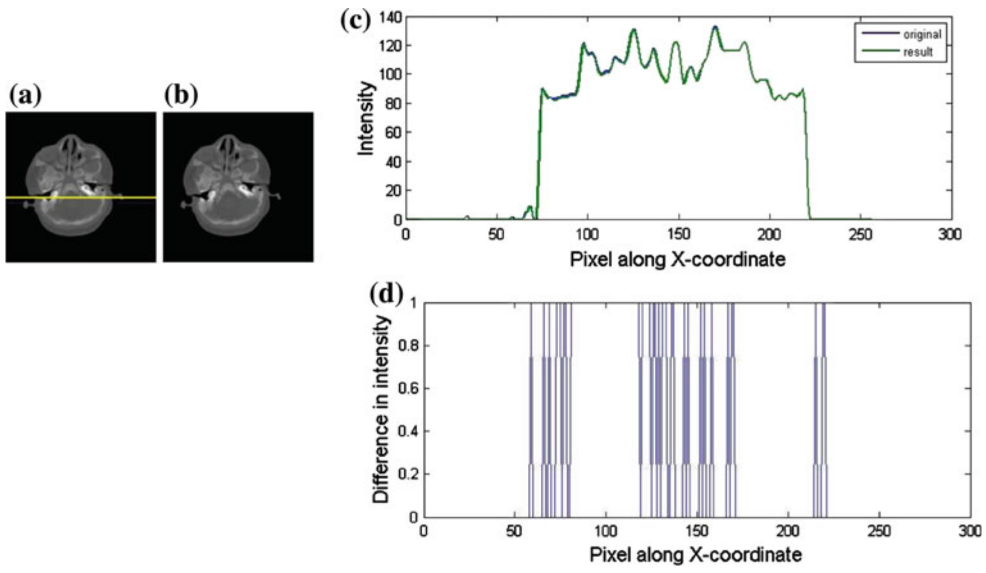

Fig. 6 Slice across of the 3D CT of CRANI: **a** original image, **b** reconstructed image and **c** line profiles across the slice image [indicated in *yellow* in (**a**)], and **d** the intensity differences between the original image and the reconstructed image along the line

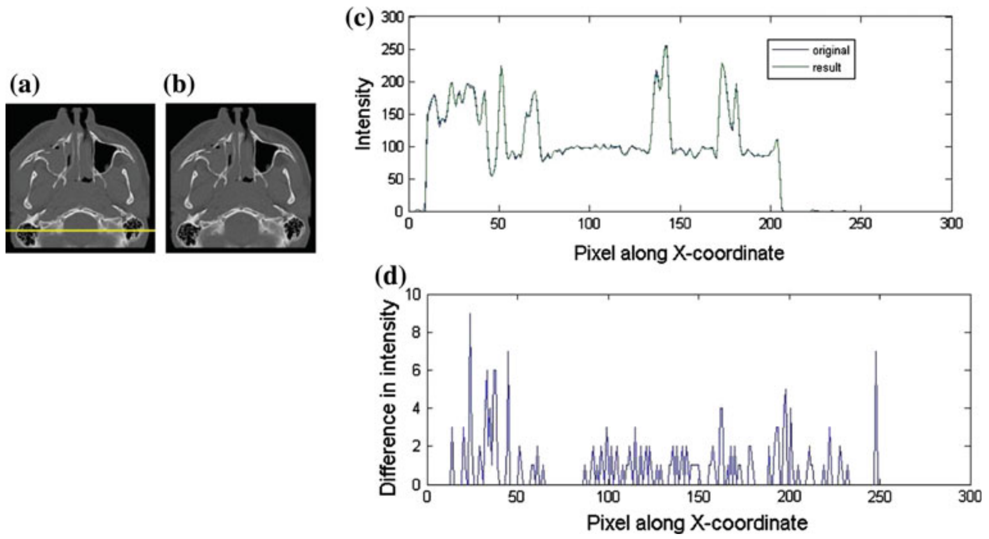

Fig. 7 Slice across of the 3D CT of SKULL: **a** original image, **b** reconstructed image and **c** line profiles across the slice image [indicated in *yellow* in (**a**)], and **d** the intensity differences between the original image and the reconstructed image along the line

Table 2 SNR and MSE of reconstructed volume obtained from various datasets

	CHEST	CRANI	HANDS
SNR (dB)	24.76	32.1	20.65
MSE	11.04	1.62	5.14

From the profiles, we find that the reconstructed profiles are highly consistent to the original profiles, which suggest that our proposed method performs well for both low contrast (i.e. CRANI) and high contrast (i.e. SKULL) data.

Apart from comparison on lines and slides of voxels, a quantitative study based on the whole volume is also conducted. We examine quality of reconstruction by signal-to-noise ratio (SNR) and mean-square-error (MSE). The formula of SNR and MSE are also stated in Appendix 1 for reference. We repeat the experiment on various CT and MRI datasets as listed in Table 2. The SNRs are ranging from 20.65 to 32.1, with an average of 25.84. MSEs are also not larger than 11.04 (of 255 dynamic range). The results reveal that the reconstructed quality is satisfactory.

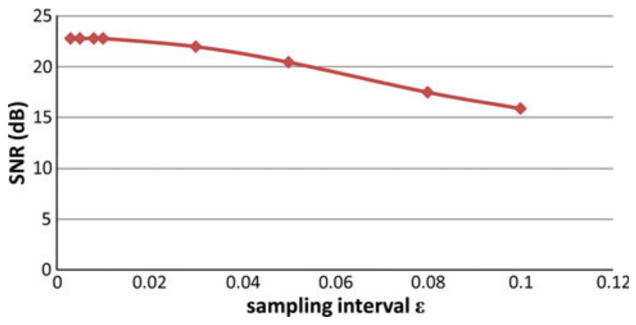

Fig. 8 Plotting of sampling interval ϵ versus SNR of reconstructed volume

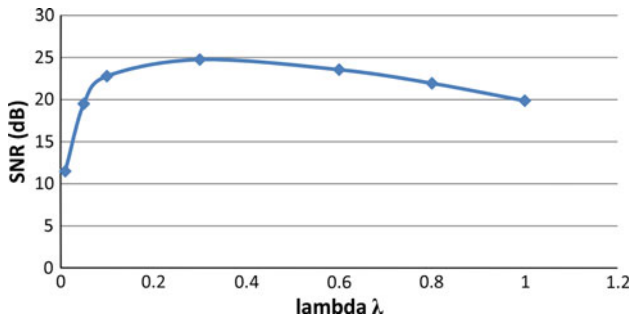

Fig. 9 Plotting of relaxation factor λ versus SNR of reconstructed volume

Effect of parameters

To investigate the effect of major parameters involved in the algorithm, we conducted several control experiments. The target parameters are the ray sampling interval ϵ and relaxation parameter λ. Since the computation complexity is not significantly affected by the setting of parameters, our experiments focus on figuring out the quality variations instead. Again, we use the CT CHEST dataset with 160 projections in all the experiments. Figure 8 and 9 show the SNR of the whole volume after reconstruction by using various values.

In our experiment, we set values between 0.003 to 0.1 for ϵ. These values are with reference to the scale of world coordinate system where the volume is defined within a cube of $2.0\times2.0\times2.0$. From Fig. 8, we observe that the corresponding SNR decreases as expected. The SNR remains about 22.8

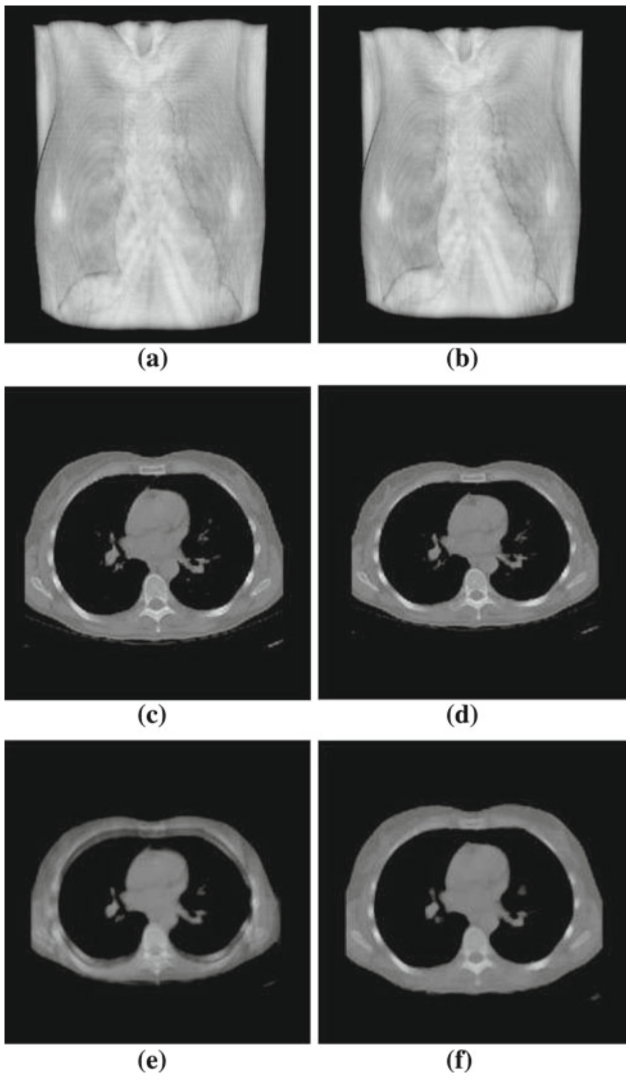

Fig. 10 Reconstruction with and without motion compensation. **a** Dataset CHEST at time 0, **b** Dataset CHEST at time 0.5 (**c**) slide from volume in (**a**), **d** slide from volume in (**b**), **e** reconstructed result with original method , **f** reconstruction with compensation

when ϵ is small enough (<0.01). As the sampling interval increases up to 0.03, the quality begins to drop in a magnitude around 1 dB. The reconstructed result becomes noisy when the sampling rate is too low.

The relaxation factor λ is plotted against reconstruction quality in Fig. 9. Experiments take range of values between 0.01 to 1.0. The quality is reduced whenever λ is too large or too small. An optimal value of 0.03 is observed from our experiments.

Motion-compensated results

Finally, we evaluate the performance of motion compensation with a simulated motion applied on CT CHEST dataset.

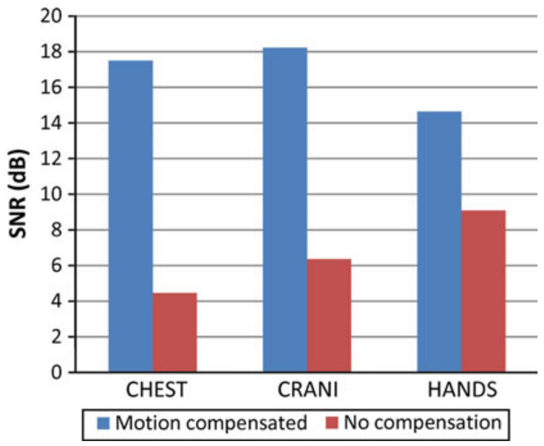

Fig. 11 Reconstruction qualities of motional datasets

We deform the original volume with the following scaling factor:

$$\begin{aligned} S_x &= 1.0 - 0.1 \times \sin(t/T), \\ S_y &= 1.0 - 0.3 \times \sin(t/T), \\ S_z &= 0.95 \end{aligned} \quad (6)$$

where T is the total acquisition time required for a 360 degree scanning cycle. The scaling is relative to the center of the object. The original CHEST volume is shown in Fig. 10a while corresponding deformed volume at $t/T = 0.5$ is shown in Fig. 10b. The same slices extracted from both volumes are shown in Fig. 10c, d respectively. Figure 10e shows the reconstructed result of a slice using the original SART algorithm without motion compensation. Serious blurring artifacts can be observed when compared to its ground truth in Fig. 10c. A motion-compensated result shown in Fig. 10f demonstrates that our method can resist the artifacts successfully during reconstruction.

The Bar chart in Fig. 11 shows a numerical evaluation of SNR on various datasets by applying our motion compensation technique. For the CT CHEST data, there is a significant difference (13.04 dB) in SNR when employing the compensation. Other data have similar improvement in quality; these results prove our method can effectively remove blurry and artifacts cause by motions and improve the image quality.

Conclusion

We employ a CUDA-enabled GPU to accelerate SART for instantaneous 3D cone-beam CT reconstruction. Two key techniques, used in ray-driven projection and voxel-driven back-projection, are proposed. In the projection stage, we utilize the hardware built-in trilinear interpolation of 3D texture to accelerate the time-consuming interpolation operations; while shared memory of GPU hardware is used in back-projection to avoid redundant computation. An efficient compensation method is implemented to reduce artifacts caused by motion. Experimental results demonstrate that our method can achieve satisfactory results in both time performance and image quality, therefore, has the potential to be used in clinical applications requiring instantaneous reconstruction of CT volumes. In the future, we plan to apply our techniques to accelerate more reconstruction algorithms, especially those from few-view and limited-angle data [26]. On the other hand, we are currently working on integrating the proposed solution into image-guided liver ablation procedures as well as other clinical applications.

Acknowledgments The work described in this paper was supported in part by a grant from the Research Grants Council of the Hong Kong Special Administrative Region under General Research Fund (Project no. CUHK 412108), a research grant from A*STAR Biomedical Research Council and Singapore Bioimaging Consortium Singapore, and a regular research grant from University of Aizu (No. R-17-2).

Appendix 1: Quality measuring metrics

Here, we define the calculation of quality measuring metrics used. N_x, N_y, N_z represent the number of voxel/pixel values in x, y and z directions respectively. $\sum\sum\sum$ is short form of $\sum_{x=1}^{N_x}\sum_{y=1}^{N_y}\sum_{z=1}^{N_z}$, and $\sum\sum$ represents $\sum_{x=1}^{N_x}\sum_{y=1}^{N_y}$. While f is the original data, and $\hat{f}$ is the reconstructed image/volume.

$$PSNR = 10\cdot\log_{10}\left[\frac{\sum\sum 255^2}{\sum\sum\left(f(x,y)-\hat{f}(x,y,z)\right)^2}\right] \quad (7)$$

$$SNR = 10\cdot\log_{10}\left[\frac{\sum\sum\sum f(x,y,z)^2}{\sum\sum\sum\left(f(x,y,z)-\hat{f}(x,y,z)\right)^2}\right] \quad (8)$$

$$MSE = \frac{\sum\sum\sum\left(f(x,y,z)-\hat{f}(x,y,z)\right)^2}{N_xN_yN_z} \quad (9)$$

Appendix 2: Source codes of key modules

Following are major source codes of the proposed CUDA-based algebraic algorithm:

```
/************ Functions to setup in OpenGL
     ******************/
/**
 *   @brief Initialize memories for volumes and corrections
 */
void InitReconstruction(){
  cudaExtent volumeSize = make_cudaExtent(volume.width,
       volume.height, volume.depth);
  cudaChannelFormatDesc channelDesc =cudaCreateChannelDesc<
       float>();

  // malloc global memoery to store the volume in cuda
  cudaMalloc((void**)&(cudavolume), volume.width * volume.
       height * volume.depth * sizeof(float));
  cudaMemset(cudavolume,0,volume.width * volume.height *
       volume.depth * sizeof(float));
  // allocate 3D array to bind 3D texture in cuda
  cudaMalloc3DArray(&d_volumeArray, &channelDesc,
       volumeSize);
  // allocate 3D array to bind motion field in cuda
  cudaMalloc3DArray(&d_motionArray, &channelDesc,
       volumeSize);
  cudaMalloc3DArray(&d_invmotionArray, &channelDesc,
       volumeSize);
  // malloc global memory to store correction data
  cudaMalloc((void**)&correction, PROJECTION_WIDTH *
       PROJECTION_HEIGHT * sizeof(float));
  cudaMemset(correction, 0, PROJECTION_WIDTH *
       PROJECTION_HEIGHT * sizeof(float));
}
/**
 *   @brief Preform the real constructions by providing all
       projections one by one, called within OpenGL display
       function
 *   @param h_proj The current projection image stored in
      main memory with size (PROJECTION_WIDTH,
      PROJECTION_HEIGHT)
 *   @param invViewMatrix  Inverted ModelView Matrix
 *   @param invRotationMatrix Inverted Rotation Matrix
 */
void DoReconstruction(float *h_proj, float* invViewMatrix,
     float* invRotationMatrix){
  float *d_proj;
  float volsize[4] = {volume.width, volume.height, volume.
       depth,0};
  dim3 blockSize(16,16);
  dim3 gridSize(PROJECTION_WIDTH / blockSize.x,
       PROJECTION_HEIGHT / blockSize.y);
  float ray_stepsize = 0.005f; // sampling interval
  float lamda = 0.1;   // relaxation factor

  cudaMemcpyToSymbol(c_invViewMatrix, invViewMatrix, sizeof
       (float4)*3);
  cudaMemcpyToSymbol(c_invRotationMatrix, invRotationMatrix
       , sizeof(float4)*3);
  cudaMemcpyToSymbol(c_volsize, volsize, sizeof(float4));
  cudaMemcpyToSymbol(c_step, &ray_stepsize, sizeof(float));
  cudaMemcpyToSymbol(c_lamda, &lamda, sizeof(float));
    // Copy the current projection to cuda
  cudaMalloc((void**)&d_proj, PROJECTION_WIDTH *
       PROJECTION_HEIGHT * sizeof(float));
  cudaMemcpy(d_proj, h_proj, PROJECTION_WIDTH *
       PROJECTION_HEIGHT * sizeof(float),
       cudaMemcpyHostToDevice);
  // bind 3D texture and motion field
  BindTextureToGlobalMem();
  //projection and correction
  cuda_projection<<<gridSize,blockSize>>>(correction,
       d_proj);
  //back-projection and update
  cuda_update<<<backGridSize,backBlockSize>>>(volume,
       correction);
  cudaFree(d_proj);
  free(h_proj);
}

void BindTextureToGlobalMem(){
  // copy data to 3D array
  cudaMemcpy3DParms copyParams = {0};
  copyParams.srcPtr   = make_cudaPitchedPtr((void*)volume.
       data, volume.width *sizeof(float), volume.width,
       volume.height);
  copyParams.dstArray = d_volumeArray;
  copyParams.extent   = volumeSize;
  copyParams.kind     = cudaMemcpyDeviceToDevice;
  cudaMemcpy3D(&copyParams);
  // set texture parameters
  texProj.normalized = true;
  texProj.filterMode = cudaFilterModeLinear;
  texProj.addressMode[0] = cudaAddressModeClamp;
  texProj.addressMode[1] = cudaAddressModeClamp;
  // bind texture array to 3D volume
  cudaBindTextureToArray(texProj,d_volumeArray,
       channelDescProj);

  cudaMemcpy3DParms copyParams = {0};
  copyParams.srcPtr   = make_cudaPitchedPtr((void*)motion.
       data, volume.width *sizeof(float), volume.width,
       volume.height );
  copyParams.dstArray = d_motionArray;
  copyParams.extent   = volumeSize;
  copyParams.kind     = cudaMemcpyHostToDevice;
  cudaMemcpy3D(&copyParams);
  motfield.normalized = true;
  motfield.filterMode = cudaFilterModeLinear;
  motfield.addressMode[0] = cudaAddressModeClamp;
  motfield.addressMode[1] = cudaAddressModeClamp;
  // bind the motion field
  cudaBindTextureToArray(motfield,d_motionArray,channelDesc
       );

  copyParams.srcPtr   = make_cudaPitchedPtr((void*)
       invmotion.data, volume.width *sizeof(float), volume.
       width, volume.height );
  copyParams.dstArray = d_invmotionArray;
  copyParams.extent   = volumeSize;
  copyParams.kind     = cudaMemcpyHostToDevice;
  cudaMemcpy3D(&copyParams);
  invmotfield.normalized = true;
  invmotfield.filterMode = cudaFilterModeLinear;
  invmotfield.addressMode[0] = cudaAddressModeClamp;
  invmotfield.addressMode[1] = cudaAddressModeClamp;

  // bind the inverted motion field
  cudaBindTextureToArray(invmotfield,d_invmotionArray,
       channelDesc);
}

/*********CUDA kernels doing projection and correction
     ************/
// 3D texture for simulation
texture<DATATYPE, 3, cudaReadModeNormalizedFloat> tex;
// 1D transfer function texture
texture<float4, 1, cudaReadModeElementType> transferTex;
// 3D texture for projection
texture<float, 3, cudaReadModeElementType> texProj;
// 3D motion fields for motion compensation
texture<float4, 3, cudaReadModeElementType> motfield;
texture<float4, 3, cudaReadModeElementType> invmotfield;
/**
 *  @brief Ray-driven Projection and Correction
 *  @param correction Buffer storing the result of
      corrections (output)
 *  @param d_proj Current projection (input)
 */
void cuda_projection(float *correction, float *d_proj){
  float3 boxMin = make_float3(-1.0f, -1.0f, -1.0f);
  float3 boxMax = make_float3(1.0f, 1.0f, 1.0f);
  // decide the pixel to handle based on thread ID
  uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
  uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

  // calculate eye ray in world space
  Ray eyeRay;
  eyeRay.o = make_float3(mul(c_invViewMatrix, make_float4
       (0.0f, 0.0f, 0.0f, 1.0f)));
  float u = (x / (float) PROJECTION_WIDTH)*2.0f-1.0f;
  float v = (y / (float) PROJECTION_HEIGHT)*2.0f-1.0f;
  eyeRay.d = normalize(make_float3(u, v, -2.0f));
  eyeRay.d = mul(c_invViewMatrix, eyeRay.d);

  // find intersection with bounding box of volume
  float proj=0; float tnear, tfar;
  int hit = intersectBox(eyeRay, boxMin, boxMax, &tnear, &
       tfar);
  if (!hit) return;
  if (tnear < 0.0f) tnear = 0.0f;     // clamp to near
       plane

  // march along ray from back to front, accumulating color
  float4 sum = make_float4(0.0f);
  float t = tfar;
  // compute the line integral p'
  for(int i=0; i<maxSteps; i++)
  {
    float3 pos = eyeRay.o + eyeRay.d*t;
    pos = pos*0.5f+0.5f;    // map position to [0, 1]
         coordinates
    // obtain motion compensation vector
    float4 motcomppos = tex3D(motfield,pos.x, pos.y, pos.z)
         ;
    pos.x =  pos.x + motcomppos.x;
    pos.y =  pos.y + motcomppos.y;
    pos.z =  pos.z + motcomppos.z;
        // obtain sample from volume using texture
             coordinate
    float sample = tex3D(texProj, pos.x, pos.y, pos.z);
    proj +=sample;
    t -= c_step;
    if (t < tnear) break;
  }
  if ((x < PROJECTION_WIDTH) && (y < PROJECTION_HEIGHT)) {
    uint i = __umul24(y, PROJECTION_WIDTH) + x;
    proj = proj * c_step;
    // write to the correction image
    correction[i] = (d_proj[i] - proj)/(tfar - tnear);
  }
}
/**
 *  @brief Voxel-driven Back-projection and Update
 *  @param volume The reconstructed volume (output)
 *  @param correction Correction image obtained from
      projection step (input)
 */
void cuda_update( float *volume,float *correction ){
  // determine the volume index to process
  uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx
       .x;
  if( index>= c_volsize.x * c_volsize.y *c_volsize.z)
    return;
  int indexZ=index/((int)c_volsize.x * (int)c_volsize.y);
  int indexY=(index - indexZ*((int)c_volsize.x * (int)
       c_volsize.y)) /(int)c_volsize.x;
  int indexX= index - indexZ*((int)c_volsize.x * (int)
       c_volsize.y) - indexY * (int)c_volsize.x;
  float3 projection;
  float minX=1000,minY=1000,maxX=-1000,maxY=-1000;
  float3 sp; float4 motcomppos; float3 volcoord;

  // loop through the 8 corners of the voxel
  for(int i=0;i<2;i++)
  for(int j=0;j<2;j++)
  for(int k=0;k<2;k++)  {
    // obtain motion compensation   vector
    motcomppos = tex3D(invmotfield,((float)indexX + i)/
         c_volsize.x, ((float)indexY + j)/c_volsize.y, ((
         float)indexZ + k)/c_volsize.z);
    volcoord.x = indexX + i + (motcomppos.x*c_volsize.x);
    volcoord.y = indexY + j + (motcomppos.y*c_volsize.y);
    volcoord.z = indexZ + k + (motcomppos.z*c_volsize.z);

    if (volcoord.x <0 || volcoord.x >= c_volsize.x) return;
    if (volcoord.y <0 || volcoord.y >= c_volsize.y) return;
    if (volcoord.z <0 || volcoord.z >= c_volsize.z ) return
         ;

    projection = getPhyCoord(volcoord.x , volcoord.y,
         volcoord.z);
    sp = projection - make_float3(0,0,4); // camear center
         at (0,0,4)
    projection.z = 2;  // projection plane at (0,0,2)
    projection.x =sp.x / sp.z * (-2.0);
    projection.y =sp.y / sp.z * (-2.0);
    // obtain the bounding box in the projection plane
    minX = min(minX,projection.x);
    maxX = max(maxX,projection.x);
    minY = min(minY,projection.y);
    maxY = max(maxY,projection.y);
  }

  int2 mbrMin,mbrMax;
    // based on similar triangles, the projection can at
         most equal to sqrt(2)/2 , i.e. between 0.7-0.8,
    // so we assume the range is between [-0.8, 0.8] , and
         now transform it to [0,PROJECTION_WIDTH]
    mbrMin.x = ( minX - (-0.8))/(1.6 / PROJECTION_WIDTH);
    mbrMin.y = ( minY - (-0.8))/(1.6 / PROJECTION_HEIGHT);
    mbrMax.x = ( maxX - (-0.8))/(1.6 / PROJECTION_WIDTH);
    mbrMax.y = ( maxY - (-0.8))/(1.6 / PROJECTION_HEIGHT);
  Ray eyeRay;
  eyeRay.o = make_float3(mul(c_invViewMatrix, make_float4
       (0.0f, 0.0f, 0.0f, 1.0f)));
  float tnear=0 ;
  float tfar=0 ;
  float3 boxMin,boxMax;
  boxMin.x = (indexX -(motcomppos.x* c_volsize.x))* (2.0 /
       c_volsize.x) -1.0;
  boxMin.y = (indexY - (motcomppos.y* c_volsize.y)) * (2.0
       / c_volsize.y) -1.0;
  boxMin.z = (indexZ- (motcomppos.z* c_volsize.z)) * (2.0 /
        c_volsize.z) -1.0;
  boxMax.x = (indexX-(motcomppos.x * c_volsize.x)+1) * (2.0
        / c_volsize.x) -1.0;
  boxMax.y = (indexY-(motcomppos.y * c_volsize.y)+1) * (2.0
        / c_volsize.y) -1.0;
  boxMax.z = (indexZ-(motcomppos.z * c_volsize.z)+1) * (2.0
        / c_volsize.z) -1.0;

  float sumError=0,sumWeight=0;
    // try all rays within the MBR
    for(int x=mbrMin.x; x<= mbrMax.x;x++)
    for(int y=mbrMin.y; y<= mbrMax.y;y++){
            float3 p = {x * (1.6/ PROJECTION_WIDTH) - 0.8,
                 y * (1.6/ PROJECTION_HEIGHT) - 0.8, -2};
            eyeRay.d = normalize(p);
            eyeRay.d = mul(c_invViewMatrix, eyeRay.d);
            int hit = intersectBox(eyeRay, boxMin, boxMax,
                 &tnear, &tfar);
            if (hit){
                sumError += (tfar - tnear) * correction[(y*
                      PROJECTION_WIDTH + x)];
                sumWeight += (tfar - tnear);
            }
    }
    if(sumWeight != 0 )
      // accumulate result in the volume
        volume[index] += c_lamda*sumError / sumWeight;
}
/**
 *  @brief getPhyCoord Compute the world coordinate of a
      voxel (x,y,z)
 */
float3 getPhyCoord(int x,int y,int z){
    float3 result;
    result.x = -1.0 + (float)x / c_volsize.x * 2.0;
    result.y = -1.0 + (float)y / c_volsize.y * 2.0;
    result.z = -1.0 + (float)z / c_volsize.z * 2.0;
    result = mul(c_invRotationMatrix, result);
    return result;
}
/**
 *  @brief intersectBox Compute ray-box intersections
 */
int intersectBox(Ray r, float3 boxmin, float3 boxmax, float
      *tnear, float *tfar){
  // compute intersection of ray with all six bbox planes
  float3 invR = make_float3(1.0f) / r.d;
  float3 tbot = invR * (boxmin - r.o);
  float3 ttop = invR * (boxmax - r.o);
  // re-order intersections to find smallest and largest on
        each axis
  float3 tmin = fminf(ttop, tbot);
  float3 tmax = fmaxf(ttop, tbot);
  // find the largest tmin and the smallest tmax
   float largest_tmin = fmaxf(fmaxf(tmin.x, tmin.y), fmaxf(
        tmin.x, tmin.z));
   float smallest_tmax = fminf(fminf(tmax.x, tmax.y), fminf(
        tmax.x, tmax.z));
     *tnear = largest_tmin;
     *tfar = smallest_tmax;
     return smallest_tmax > largest_tmin;
}
```

References

1. Gordon R, Bender R, Herman GT (1970) Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography. J Theor Biol 29:471–481
2. Andersen AH, Kak AC (1984) Simultaneous algebraic reconstruction technique (sart): a superior implementation of the art algorithm. Ultrason Img 6(1):81–94
3. Feldkamp LA, Davis LC, Kress JW (1984) Practical cone beam algorithm. J Opt Soc Am 1(6):612–619
4. Kak A, Slaney M (1988) Principles of computerized tomographic imaging. IEEE Press
5. Xu F, Mueller K (2007) Real-time 3-d computed tomographic reconstruction using commodity graphics hardware. Phys Med Biol 52(12):3405–3419
6. Mueller K, Yagel R (2000) Rapid 3d cone-beam reconstruction with the simultaneous algebraic reconstruction technique (sart) using 2-d texture mapping hardware. IEEE Trans Med Imaging 19(12):1227–1237
7. Huh Y, Jin SO, Park JB (1999) Fast image reconstruction from fan beam projections using paralleldigital signal processors and special purpose processors. In: Proceedings of the IEEE region 10 conference (TENCON), pp 1558–1561
8. Gac N, Mancini S, Desvignes M (2006) Hardware/software 2d-3d backprojection on a sopc platform. In: Proceedings of 21st ACM Symp. Applied Computing (SAC). 222–228
9. Kachelrieβ M, Knaup M, Bockenbach O (2006) Hyperfast perspective cone-beam backprojection. In: IEEE Medical Imaging Conference Records
10. Cabral B, Cam N, Foran J (1994) Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: VVS '94: Proceedings of the 1994 symposium on volume visualization, New York, NY, USA, ACM, pp 91–98
11. Chidlow K, Möller T (2003) Rapid emission tomography reconstruction. In: VG '03: Proceedings of the 2003 Eurographics/IEEE TVCG workshop on volume graphics, New York, NY, USA, ACM 15–26
12. Xu F, Mueller K (2005) Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. IEEE Trans Nuclear Sci 52(3):654–663
13. Riabkov D, Xue X, Tubbs D, Cheryauka A (2007) Accelerated cone-beam backprojection using GPU-CPU hardware. In: Proceedings of 9th international. Meeting fully three-dimensional image reconstruction in radiology and nuclear medicine (fully 3D 2007), pp 68–71
14. Thomas Schiwietz, Ti-chin Chang PSRW (2006) MR image reconstruction using the GPU. In: Flynn M, Jansen HJ (eds) Proceedings of SPIE medical imaging 2006, vol 6142, San Diego, CA, SPIE, pp 1279–1290
15. Scherl H, Keck B, Kowarschik M, Hornegger J (2007) Fast GPU-based ct reconstruction using the common unified device architecture (CUDA). In: Proceedings of nuclear science symposium and medical imaging conference (NSS/MIC), pp 4464–4466

16. Okitsu Y, Ino F, Hagihara K (2008) Accelerating cone beam reconstruction using the CUDA-enabled GPU. In: Proceedings of high performance computing (HiPC), pp 108–119
17. Bonnet S, Koenig A, Roux S, Hugonnard P, Guillemaud R, Grangeat P (2003) Dynamic x-ray computed tomography. In: Proceedings of the IEEE, vol 91, pp 1574–1587
18. Balter J, Haken RT, Lawrence T, Lam K, Robertson J (1996) Uncertainties in CT-based radiation therapy treatment planning associated with patient breathing. Int J Radiat Oncol Biol Phys 36:167–174
19. Seppenwoolde Y, Shirato H, Kitamura K, Shimizu S, Herk Mvan , Lebesque J, Miyasaka K (2002) Precise and realtime measurement of 3d tumor motion in lung due to breathing and heartbeat, measured during radiotherapy. Int J Radiat Oncol Biol Phys 53:822–834
20. Nehmeh S, Erdi Y, Ling C, Rosenzweig K, Schroder H, Larson S, Macapinlac H, Squire O, Humm J (2002) Effect of respiratory gating on quantifying pet images of lung cancer. J Nucl Med 43:876–881
21. Kaczmarz S (1937) Angenaherte auflosung von systemen linearer glerichungen. Bull Acad Pol Sci Lett A 6-8A:355–357
22. Siddon RL (1985) Fast calculation of the exact radiological path for a three-dimensional ct array. Med Phys 12(2):252–255
23. Li N, Zhao HX, Cho SH, Choi JG, Kim MH (2008) A fast algorithm for voxel-based deterministic simulation of x-ray imaging. Comput Phys Commun 178(7):518–523
24. Rit S, Sarrut D (2005) Cone-beam projection of a deformable volume for motion compensated algebraic reconstruction. In: 29th annual international conference of the IEEE engineering in medicine and biology society (EMBS), pp 22–26
25. Reyes M, Malandain G, Koulibaly PM, González-Ballester MAJD (2007) Model-based respiratory motion compensation for emission tomography image reconstruction. Phys Med Biol 52:3579–3600
26. Sidky EY, Kao CM, Pan X (2006) Accurate image reconstruction from few-views and limited-angle data in divergent-beam ct. J X-Ray Sci Technol 14(2):119–139