High-Performance Cone Beam Reconstruction Using CUDA Compatible GPUs

Yusuke OKITSU[a], Fumihiko INO[*,a], Kenichi HAGIHARA[a]

[a] *Graduate School of Information Science and Technology, Osaka University, 1-5 Yamada-oka, Suita, Osaka 565-0871, Japan*

Abstract

Compute unified device architecture (CUDA) is a software development platform that allows us to run C-like programs on the nVIDIA graphics processing unit (GPU). This paper presents an acceleration method for cone beam reconstruction using CUDA compatible GPUs. The proposed method accelerates the Feldkamp, Davis, and Kress (FDK) algorithm using three techniques: (1) off-chip memory access reduction for saving the memory bandwidth; (2) loop unrolling for hiding the memory latency; and (3) multithreading for exploiting multiple GPUs. We describe how these techniques can be incorporated into the reconstruction code. We also show an analytical model to understand the reconstruction performance on multi-GPU environments. Experimental results show that the proposed method runs at 83% of the theoretical memory bandwidth, achieving a throughput of 64.3 projections per second (pps) for reconstruction of 512^3-voxel volume from 360 512^2-pixel projections. This performance is 41% higher than the previous CUDA-based method and is 24 times faster than a CPU-based method optimized by vector intrinsics. Some detailed analyses are also presented to understand how effectively the acceleration techniques increase the reconstruction performance of a naive method. We also demonstrate out-of-core reconstruction for large-scale datasets, up to 1024^3-voxel volume.

Key words: Cone beam reconstruction, acceleration, GPU, CUDA

1. Introduction

Cone beam (CB) reconstruction is an imaging technique for producing a three-dimensional (3-D) volume that gives us insight of patients for clinical purposes. This technique generates volume data from a series of 2-D projections acquired by a computed tomography (CT) scan with a flat-panel detector. For the sake of surgical assistance, the technique is usually integrated into mobile C-arm CT systems to help the operator during image-guided surgery. Since the surgical procedure has to be stopped

[*] Corresponding author. Tel.: +81 6 6879 4353; fax: +81 6 6879 4354.
Email address: ino@ist.osaka-u.ac.jp (Fumihiko INO)

until the end of the reconstruction procedure, a reconstruction task should be completed within 10 seconds to minimize such interrupted time. However, it takes at least 3 minutes to obtain a 512^3-voxel volume on a single 3.06 GHz Xeon processor [1]. Accordingly, many researchers are trying to accelerate CB reconstruction using various accelerators, such as the graphics processing unit (GPU) [2, 3, 4, 5, 6, 7], Cell Broadband Engine (CBE) [1], and field programmable gate array (FPGA) [8].

Among the accelerators mentioned above, the GPU [9] and the CBE [10] are commodity chips designed to accelerate multimedia applications such as gaming and computer-aided design (CAD) systems. However, these chips can provide a low-cost solution to compute-intensive problems not only in multimedia area but also in medical area [11, 12]. Actually, as far as we know, the fastest CB reconstruction is presented by Scherl *et al.* [3]. They develop a GPU-accelerated reconstruction using compute unified device architecture (CUDA) [13], which enables us to run C-like programs on the nVIDIA GPU. A reconstruction task of a 512^3-voxel volume from 414 1024^2-pixel projections takes 12.02 seconds on an nVIDIA GeForce 8800 GTX, namely a high-end graphics card.

In contrast to this non-graphics based approach, a graphics based approach is proposed by many researchers [2, 4, 14]. Using the OpenGL graphics library [15], their implementation runs as a graphics application but realizes real-time reconstruction. It takes 8.1 seconds to reconstruct a 512^3-voxel volume from 360 512^2-pixel projections [14], which is only 1% slower than the non-graphics based approach [3]. Thus, it is still not clear whether the CUDA-based approach is significantly faster than the graphics-based approach, though the CUDA platform is designed for general purposes rather than graphics purposes. In particular, optimization techniques are of great interest to computational scientists, because such techniques determine the performance of CUDA programs.

In this paper, we propose a CUDA-based method for accelerating CB reconstruction on nVIDIA GPUs. Our main goal is to present key optimization techniques that enable the CUDA-based approach to achieve higher reconstruction performance than the graphics-based approach. We extend our preliminary work [16] by developing a multithreading mechanism capable of exploiting multiple GPUs on a single desktop PC. Our method is based on the Feldkamp, Davis, and Kress (FDK) reconstruction algorithm [17], which is used in many prior projects [1, 2, 3, 4, 5, 6, 7, 8, 18]. We optimize the method using three acceleration techniques: (1) off-chip memory access reduction for saving the bandwidth of video memory; (2) loop unrolling for hiding the memory latency with data-independent computation; and (3) multithreading for exploiting multiple GPUs. We also show how effectively these techniques contribute to higher performance, making it clear that the memory bandwidth mainly limits the performance of the proposed method. Furthermore, an analytical model is presented to understand the reconstruction performance on multi-GPU environments.

The rest of the paper is organized as follows. We begin in Section 2 by introducing related work. Section 3 gives a brief summary of CUDA and Section 4 shows an overview of the FDK algorithm. Section 5 describes our CUDA-based method and Section 6 presents experimental results. Finally, Section 7 concludes the paper.

2. Related Work

A naive solution to accelerate heavy computation is to employ a cluster computing approach [19, 20, 21] that runs applications on multiple PCs. However, the advances in integrated circuits (ICs) make it possible to implement hundreds of processing elements into a single IC chip. For example, the GPU currently has at least 128 processing elements on a single card, reducing the energy consumption with the physical size. These characteristics are desirable for surgical environments, where systems have to be robust against power failures.

To the best of our knowledge, the basic idea of graphics-based implementations has been introduced by Cabral *et al.* [22] who firstly used texture mapping hardware for CB reconstruction. This texture-based idea is then extended by Mueller and Xu [23, 24] to implement various reconstruction methods on the GPU: the FDK algorithm [17]; ordered subsets expectation maximization (OSEM) [25]; and simultaneous algebraic reconstruction technique (SART) [26]. Since their work is done before the dawning of CUDA, their methods are implemented using graphics libraries such as OpenGL [15] and C for graphics (Cg) [27].

Xu *et al.* [2] finally propose an OpenGL-based method that maps the FDK algorithm onto the graphics pipeline [11] in the GPU. Their graphics-based approach has an advantage over CUDA-based methods in terms of using graphics optimization techniques. For example, they realize a load balancing scheme by moving instructions from fragment processors to vertex processors, each composing a stage of the pipeline [11]. This code motion technique also reduces the computational complexity [28]. Furthermore, their method uses the early fragment kill (EFK) technique to restrict computation to voxels within the region of interest (ROI). Although this fragment culling technique achieves further acceleration, we cannot obtain the correct data outside the ROI, where fragments are culled from rendering. In contrast, our goal is to achieve rapid reconstruction for the entire volume. Another graphics-based implementation is proposed by Riabkov *et al.* [4].

Scherl *et al.* [3] show a CUDA-based method with a comparison to a CBE-based method. They claim that their method reduces the number of instructions and the usage of registers. On the other hand, our acceleration techniques focus on reducing the amount of off-chip memory accesses and on hiding the memory latency with computation. We think that such memory optimization is important to improve the performance of the FDK algorithm, which can be classified into a memory-intensive problem. A similar approach is proposed by Noël *et al.* [7] who use on-chip shared memory to save the bandwidth of off-chip memory. Although this is important for many scientific applications, shared memory does not have a fast interpolation mechanism, which improves the reconstruction quality. On the other hand, our method uses texture memory, which provides a hardware-accelerated interpolation mechanism to offload processing elements. Yang *et al.* [18] develop a fast backprojection method for CB reconstruction. Their CUDA-based method takes 4.7 seconds to perform backprojection of 360 600^2-pixel projections into a 512^3-voxel volume. Our method differs from their work in that the method accelerates the entire FDK algorithm. With respect to the backprojection performance, our method achieves slightly higher performance than their method.

In contrast to the implementation-level optimization mentioned above, rebinning

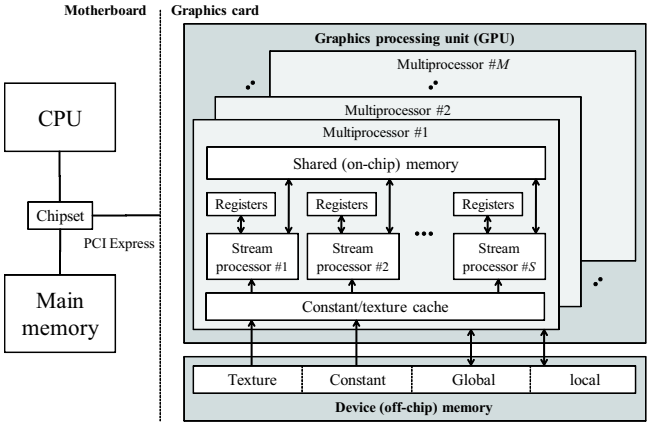


Figure 1: Architecture of CUDA-enabled GPU. There are roughly two groups of memory on a graphics card. On-chip (shared) memory is almost fast as registers. On the other hand, off-chip (device) memory takes 400–600 clock cycles to fetch/store data. A GeForce 8800 GTX card has $MS = 16 \cdot 8 = 128$ stream processors, where M and S represent the number of multiprocessors and that of stream processors, respectively.

algorithms [29, 30] convert the geometry to perform optimization at the algorithm level. For example, rebinning from CB projections to parallel beam projections simplifies the backprojection operation needed for the FDK reconstruction. One drawback of this rebinning strategy is that it can create artifacts in the final volume.

The main contribution of this paper is to show key optimization techniques that are essential to achieve the full reconstruction performance on CUDA compatible GPUs. We also show that our effective performance is reasonable in terms of the theoretical performance rather than the relative performance over CPU-based methods.

3. Overview of Compute Unified Device Architecture (CUDA)

CUDA [13] is a development platform designed for writing and running general-purpose applications on the nVIDIA GPU. Similar to graphics applications, CUDA applications can be accelerated by data-parallel computation [31] of millions of threads. A thread here is an instance of a kernel, namely a program running on the GPU. The main advantage of this platform is that it allows us to regard the GPU as a single instruction, multiple data (SIMD) parallel machine [31] rather than graphics hardware. Thus, there is no need to understand the graphics pipeline to execute programs on this highly-threaded architecture. It also should be noted that CUDA exposes the memory hierarchy to developers, allowing them to maximize application performance by optimizing data access.

Figure 1 illustrates the architecture of the CUDA compatible GPU. The GPU is implemented on a graphics card with video memory, called device memory. Since this off-chip memory is separated from the GPU, it takes at least 400 clock cycles to fetch data from that memory. The GPU consists of M multiprocessors (MPs), each having S stream processors (SPs). Every MP has small on-chip memory, called shared memory, which can be accessed from internal SPs as fast as registers. However, this memory

is not shared between different MPs. Due to this constraint, threads are classified into groups and each group is called as a thread block (TB), which is the minimum allocation unit assigned to an MP. Therefore, developers have to write their code such that there is no data dependence between threads in different TBs. On the other hand, threads in the same TB are allowed to have data dependence because they can exchange data using shared memory. The consistency of shared data must be kept by application developers using a synchronization function.

As shown in Fig. 1, off-chip memory consists of texture memory, constant memory, local memory, and global memory. Texture memory and constant memory have a cache mechanism but they are not writable from SPs. Therefore, developers are needed to transfer (download) data from main memory in advance of a kernel invocation. Texture memory differs from constant memory in that it provides a hardware mechanism that returns linearly interpolated texels from the surrounding texels. This hardware is called as the texture unit, which is separated from SPs. On the other hand, local memory and global memory are writable from SPs but they do not have a cache mechanism. Global memory achieves almost the full memory bandwidth if data accesses can be coalesced into a single access [13]. Local memory cannot be explicitly used by developers. This memory space is implicitly used by the CUDA compiler in order to avoid resource consumption. It is better to eliminate such inefficient accesses hidden in the kernel code because local memory cannot be accessed in a coalesced manner. For example, an array will be allocated to local memory if it is too large for register space.

In most applications, assigning multiple TBs to every MP is important to hide the latency of device memory. This is due to the architectural design of the GPU, which switches the current TB to another TB when it has to wait for data from device memory. To realize such latency hiding, we must write the kernel code such that it minimizes the usage of shared memory and registers, because such resource usage limits the number of concurrent TBs. Note here that this does not always apply to all cases especially if the kernel code should use texture memory (as shown later in Section 6).

4. Feldkamp, Davis, and Kress (FDK) Reconstruction

Consider a reconstruction task that produces an N^3-voxel volume F from a series of $U \times V$-pixel projections $P_1, P_2, \ldots, P_K$ obtained by a scan rotation of a detector. The FDK algorithm [17] solves this problem by two processing stages: the filtering stage and the backprojection stage. At the filtering stage, the algorithm applies the Shepp-Logan filter [32] to each projection P_n to obtain filtered projection Q_n, where $1 \leq n \leq K$. This filter gives a smoothing effect to minimize noise propagation during the succeeding backprojection stage. Let $P_n(u, v)$ and $Q_n(u, v)$ be the pixel value at point (u, v) of P_n and that of Q_n, respectively, where $1 \leq u \leq U$ and $1 \leq v \leq V$. The filtering stage converts $P_n(u, v)$ to $Q_n(u, v)$ such that:

$$Q_n(u, v) = \sum_{r=-R}^{R} \frac{2}{\pi^2(1 - 4r^2)} W_1(r, v) P_n(r, v), \tag{1}$$

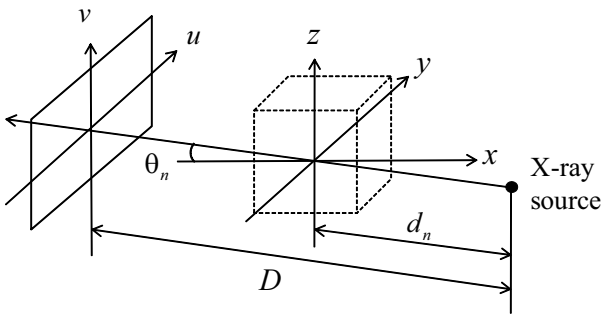


Figure 2: Coordinate system for backprojection. The xyz space represents the volume while the uv plane represents a filtered projection that is to be backprojected to the volume.

where R represents the filter size and $W_1(r, v)$ represents the weight value given by:

$$W_1(r, v) = \frac{D}{\sqrt{D^2 + r^2 + v^2}}, \tag{2}$$

where D represents the distance between the X-ray source and the origin of the detector (projection), as shown in Fig. 2.

A series of filtered projections $Q_1, Q_2, \ldots, Q_K$ are then backprojected to the volume F. In Fig. 2, the xyz space corresponds to the target volume F while the uv plane represents the n-th filtered projection Q_n that is to be backprojected to volume F from angle θ_n, where $1 \leq n \leq K$. Note here that the distance d_n between the X-ray source and the volume origin should be parameterized for each projection, because it varies during the rotation of a real detector. On the other hand, distance D can be modeled as a constant value in C-arm systems.

Using the coordinate system mentioned above, the voxel value $F(x, y, z)$ at point (x, y, z), where $0 \leq x, y, z \leq N - 1$, is computed by:

$$F(x, y, z) = \frac{1}{2\pi K} \sum_{n=1}^{K} W_2(x, y, n) Q_n(u(x, y, n), v(x, y, z, n)), \tag{3}$$

where the weight value $W_2(x, y, n)$, the coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are given by:

$$W_2(x, y, n) = \Big(\frac{d_n}{d_n - x\cos\theta_n + y\sin\theta_n}\Big)^2, \tag{4}$$

$$u(x, y, n) = \frac{D(x\sin\theta_n + y\cos\theta_n)}{d_n - x\cos\theta_n + y\sin\theta_n}, \tag{5}$$

$$v(x, y, z, n) = \frac{Dz}{d_n - x\cos\theta_n + y\sin\theta_n}. \tag{6}$$

The coordinates $u(x, y, n)$ and $v(x, y, z, n)$ here are usually real values rather than integer values. Since projections $P_1, P_2, \ldots P_K$ are given as discrete data, an interpolation mechanism is essential to obtain high-quality volume.

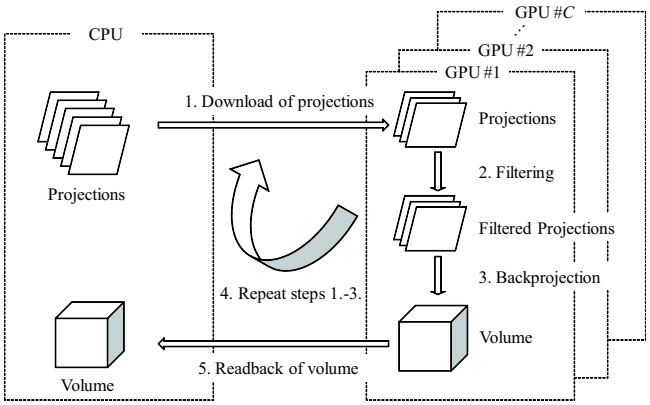


Figure 3: Overview of the proposed method. Projections are serially sent to the GPU in order to accumulate their pixels into the volume in video memory.

5. Proposed Method

We now describe our CUDA-based method that runs on a multi-GPU environment. We first present data distribution and parallelization strategies and then explain how the backprojection performance can be maximized on the GPU. An analytical model is also presented to analyze the theoretical performance of our method.

5.1. Data Distribution and Parallelization

Equation (3) indicates that there is no data dependency between different voxels as well as between different projections. Therefore, voxels can be independently assigned to different GPUs for parallel backprojection. In addition, the backprojection operation in Eq. (3) has an associativity that allows us to process projections in an arbitrary order. Thus, projections as well as voxels can be processed in parallel. This full parallelism means that the volume and projections can be arbitrary divided into small portions without significant performance degradation because it does not need data exchange between SPs during parallel backprojection.

It is not easy for commodity graphics cards to store both the entire volume and projections in device memory, because a 512^3-voxel volume consumes at least 512 MB of memory space. Therefore, either the volume data or the projection data must be divided into small portions and then be transferred from main memory to device memory when needed. We have decided to store the volume data rather than the projection data in device memory, because each projection can be removed immediately after its backprojection. In other words, this decision allows us to structure the reconstruction procedure into a pipeline that can return the volume immediately after the end of a scan rotation. Figure 3 shows an overview of our reconstruction method. The method sends a series of projections to global memory, which is then filtered and backprojected into the volume in global memory. This operation is iteratively applied to the remaining projections to obtain the final accumulated volume.

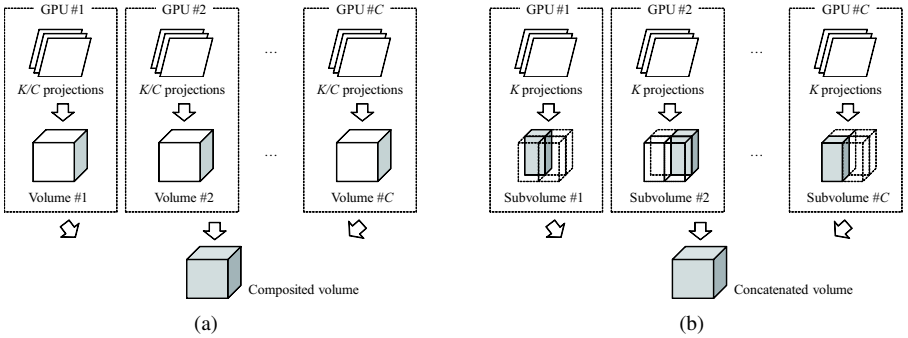


Figure 4: Two parallelization schemes. (a) Projection parallelism scheme and (b) voxel parallelism scheme. The former scheme involves volume composition to obtain the final volume. Instead of this, the latter scheme involves subvolume concatenation.

With respect to the parallelization scheme, there can be two variations for multi-GPU environments, as shown in Fig. 4. Let C be the number of GPUs. The first one is the projection parallelism scheme, which assigns K/C projections to each GPU. As shown in Fig. 4(a), this scheme produces C volumes after backprojection, which must be composited into a single volume at the final stage. The other one is the voxel parallelism scheme, which assigns all K projections to GPUs but each GPU generates the responsible portion of the volume as shown in Fig. 4(b). Thus, the difference to the former scheme is that the latter scheme directly generates subvolumes of the final volume. Therefore, the voxel parallelism scheme does not require the final composition stage but involves concatenation of C subvolumes into a single volume. This concatenation overhead is smaller than the composition overhead needed for the former scheme. We discuss on the pros and cons of each scheme later in Section 5.3.

Figure 5 shows a pseudocode for the proposed method. In Fig. 5, the filtering stage is parallelized in the following way. Equation (1) indicates that this stage performs a 1-D convolution in the u-axis direction. Thus, there is no data dependence between different pixels in a filtered projection Q_n. However, pixels in the same row v refer partly the same pixels in projection P_n: $P_n(r, v)$, where $-R \leq r \leq R$. Therefore, it is better to use shared memory to save the memory bandwidth for such commonly accessed pixels. Thus, we have decided to write the filtering kernel such that a TB is responsible for a row in Q_n. On the other hand, a thread is responsible for computing a pixel in the row. As shown in Fig. 5, threads in the same TB cooperatively copy a row v to shared memory at line 24, which are then accessed instead of the original data in global memory at line 26.

Note that the filtered projection data is accessed as textures during backprojection (line 14). As we mentioned in Section 4, the coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are usually real values. Therefore, we load the data $Q_n(u, v)$ from a texture, which returns a texel value interpolated by hardware. This strategy contributes to a full utilization of the GPU, because the interpolation hardware is separated from processing units.

5.2. Accelerated Backprojection

As we mentioned earlier, there is a full parallelism in the backprojection stage. Therefore, a naive solution may assign every voxel to different threads. However, this is not a good strategy because some computational results can be reused between different voxels. For example, Eqs. (4) and (5) indicate that $W_2(x, y, n)$ and $u(x, y, n)$ do not depend on z. Such a z-independent computation should be assigned to the same thread in order to perform data reuse for complexity reduction. Furthermore, this data reuse technique can also be applied to reduce the complexity of Eq. (6). Although $v(x, y, z, n)$ depends on z, it can be rewritten as

Input: Projections $P_1 \ldots P_K$, filter size R and parameters $D, d_1 \ldots d_K, \theta_1 \ldots \theta_K, I, J$
Output: Volume F

Algorithm Reconstruction()
1: Initialize volume F;
2: float $G[R+1]$;
3: **for** $r = 0$ **to** R **do**
4: $G[r] \leftarrow 2/(\pi^2(1-4r^2))$; // precomputation for filtering
5: **end for**
6: Transfer array G to constant memory;
7: $n \leftarrow 1$;
8: **while** $n \leq K$ **do**
9: **for** $i = 0$ **to** $IJ-1$ **do**
10: $d_n[i] \leftarrow d_{n+i}$;
11: $\theta_n[i] \leftarrow \theta_{n+i}$;
12: Transfer projection P_{n+i} to global memory;
13: $Q[i] \leftarrow$ FilteringKernel(P_{n+i}, R);
14: Bind filtered projection $Q[i]$ as a texture;
15: **end for**
16: $F \leftarrow$ BackprojectionKernel($Q[IJ], D, d_n[IJ], \theta_n[IJ], n$);
17: $n \leftarrow n + IJ$;
18: **end while**
19: Transfer volume F to main memory;

Function FilteringKernel(P, R)
20: __shared__ float $array[U]$; // U: projection width
21: $u \leftarrow \mathrm{index}(\mathrm{threadID})$; // returns responsible u
22: $v \leftarrow \mathrm{index}(\mathrm{blockID})$;
23: Initialize $Q(u, v)$;
24: $array[u] \leftarrow W_1(u, v) * P(u, v)$;
25: **for** $r = -R$ **to** R **do** // unrolled
26: $Q(u, v) \leftarrow Q(u, v) + G[|r|] * array[u + r]$;
27: **end for**

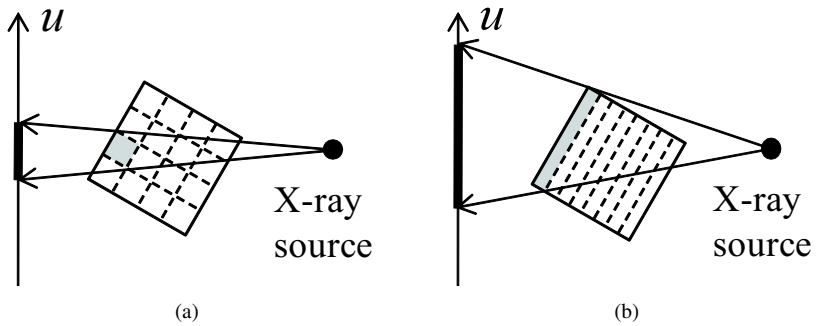
Function BackprojectionKernel($Q[IJ], D, d_n[IJ], \theta_n[IJ], n$)
28: var $u[I], v[I], v'[I], w[I]$;
29: $x \leftarrow \mathrm{index}(\mathrm{blockID}, \mathrm{threadID})$;
30: $y \leftarrow \mathrm{index}(\mathrm{blockID}, \mathrm{threadID})$;
31: **for** $j = 0$ **to** $J-1$ **do** // unrolled
32: **for** $i = 0$ **to** $I-1$ **do**
33: $w[i] \leftarrow W_2(x, y, Ij + i + n)$; // Eq. (4)
34: $u[i] \leftarrow u(x, y, Ij + i + n)$; // Eq. (5)
35: $v[i] \leftarrow v(x, y, 0, Ij + i + n)$; // Eq. (6)
36: $v'[i] \leftarrow v'(x, y, Ij + i + n)$; // Eq. (8)
37: **end for**
38: **for** $z = 0$ **to** $N-1$ **do**
39: $F(x, y, z) \leftarrow F(x, y, z) + w[0] * Q[Ij](u[0], v[0])$
$+w[1] * Q[Ij+1](u[1], v[1])$
$\cdots$
$+w[I-1] * Q[Ij+I-1](u[I-1], v[I-1])$;
40: **for** $k = 0$ **to** $I-1$ **do**
41: $v[k] \leftarrow v[k] + v'[k]$;
42: **end for**
43: **end for**
44: **end for**

Figure 5: Pseudocode of the proposed method. The CPU iteratively invokes two kernels that run on the GPU to perform filtering and backprojection of IJ projections at a time. Loops are not unrolled in this code for the simplicity of explanation.

$$v(x, y, z, n) = v'(x, y, n)z, \tag{7}$$

where

$$v'(x, y, n) = \frac{D}{d_n - x\cos\theta_n + y\sin\theta_n}. \tag{8}$$



(a) (b)

Figure 6: Thread block organization. (a) Our method structures thread blocks into squares to achieve higher locality of data access. (b) Otherwise, texels can be fetched from a wider region of the filtered projection, reducing the hit ratio of texture caches. A thread is responsible for z-axis aligned voxels.

Therefore, we can precompute $v'(x, y, n)$ for any z (line 36), in order to incrementally compute Eq. (6) at line 41. In summary, a thread is responsible for z-axis aligned voxels: voxels $(X, Y, 0), (X, Y, 1), \ldots, (X, Y, N-1)$, where $0 \leq X, Y \leq N-1$. On the other hand, a TB is responsible for a set of such aligned voxels.

Figure 6 illustrates how we organize threads into a TB. As shown in Fig. 6(a), we structure TBs into squares to maximize the cache efficiency. Such closely located voxels can be projected into a small region of a projection. Therefore, it improves the locality of texture accesses, increasing the cache hit ratio as compared with other organizations. The TB size is mainly determined by hardware limitation such as the number of available registers. We currently structure 16×16 threads into a TB on the GeForce 8800 GTX card. We also store the volume data in global memory so that the memory accesses can be coalesced into a single contiguous, aligned memory access [13].

Finally, we explain how our method maximizes the effective memory bandwidth to accelerate the memory-intensive backprojection operation. We maximize the effective bandwidth by two techniques which we mentioned in Section 1.

1. Off-chip memory access reduction. We write the kernel such that it performs backprojection of I projections at a time, where I (≥ 2) represents the number of projections processed by a single kernel invocation. This technique reduces the number of global memory accesses to $1/I$ because it allows us to write temporal voxel values to registers before writing the final values to global memory, as shown at line 39 in Fig. 5. Note that it requires more registers as we increase the value of I. We currently use $I = 3$, which is experimentally determined for the target GPU.
2. Loop unrolling. As shown in Fig. 5, we pack J successive kernel calls into a single call by unrolling the kernel code. A kernel invocation now processes every I projections J times. This modification is useful to hide the memory latency with data-independent computation. For example, if SPs are waiting for memory accesses needed for the first I projections, they can perform computation for the remaining $I(J-1)$ projections if there is no data dependence between them.

Table 1: Time complexity and space complexity of the proposed method. Space complexity is presented as per GPU.

Complexity	Stage	Filtering -based scheme	Backprojection -based scheme
Time	Download	$O(KUV)$	$O(KUVC)$
	Filtering	$O(KUV/C)$	$O(KUV)$
	Backprojection	$O(KN^3/C)$	$O(KN^3/C)$
	Readback	$O(N^3C)$	$O(N^3)$
	Post-processing	$O(N^3C)$	$O(C)$
Space	—	$O(N^3)$	$O(N^3/C)$

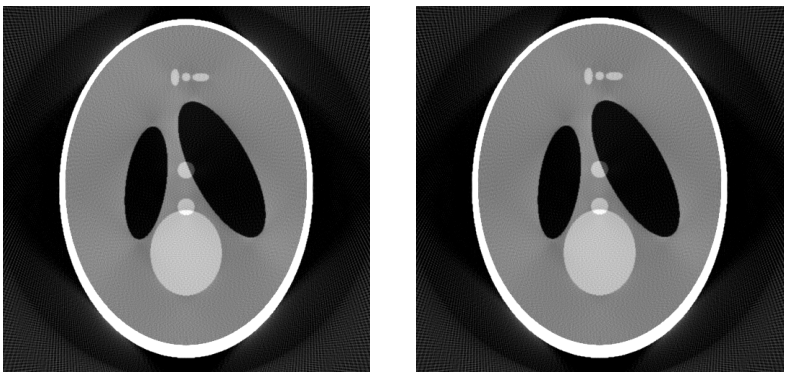
In general, the compiler assists this overlap by using different registers to avoid causing data dependence between unrolled sentences. As we did for I, we have experimentally decided to use $J = 2$.

5.3. Analytical Performance Model

We present an analytical model to understand the performance of our parallelization schemes. This model assumes serial data transfer between main memory and video memory. Actually, current graphics driver cannot exchange data with multiple graphics cards at the same time.

Table 1 shows the time and space complexities for each parallelization scheme. The space complexity is presented as per GPU. The time complexity can be explained as follows.

1. Download stage. The projection parallelism scheme serially sends K/C projections to C cards. Since each projection contains UV texels, it takes $O(KUV)$ time. In contrast, the voxel parallelism scheme sends K projections to C cards. Therefore, the time complexity is given by $O(KUVC)$.
2. Filtering stage. This stage deals with the same data amount as the download stage. However, filtering operations are processed in parallel using C cards. Therefore, the time complexity of this stage is given by dividing that of the download stage by C.
3. Backprojection stage. This stage is also processed in parallel. The projection parallelism scheme takes $O(KN^3/C)$ time to accumulate values into N^3 voxels K/C times. The voxel parallelism scheme also takes the same $O(KN^3/C)$ time because it processes N^3/C voxels K times.
4. Readback stage. Similar to the download stage, this stage is serially processed by the CPU. The projection parallelism scheme takes $O(N^3C)$ time to receive N^3 voxels from C cards. In contrast, the voxel parallelism scheme takes $O(N^3)$ time to receive N^3/C voxels from C cards.
5. Post-processing stage. This stage is also serially processed on the CPU. It takes $O(N^3C)$ time to accumulate C volumes into a single volume. On the other hand, concatenation of C volumes can be processed in $O(C)$ time.



(a) (b)

Figure 7: Sectional views of the Shepp-Logan phantom [32] reconstructed (a) by the GPU and (b) by the CPU.

The main difference between two schemes is that the voxel parallelism scheme requires a smaller amount of video memory because it distributes a portion of the volume to each GPU. This contributes to have smaller overheads in the readback and post-processing stages. Instead of these advantages, the overheads in the download and filtering stages are higher than those of the projection parallelism scheme.

6. Experimental Results

We now show experimental results to demonstrate the performance advantages of the proposed method. To make the explanation easier, we first show an analysis on a single-GPU environment and then that on a multi-GPU environment. Five previous implementations and results are used for comparison: OpenGL-based methods [2, 14]; a previous CUDA-based method [3]; a CBE-based method [1]; and a CPU-based method [1].

For the single-GPU environment, we use a desktop PC equipped with a Core 2 Quad Q6700 CPU, 8GB main memory, and an nVIDIA GeForce 8800 GTX GPU with 768MB video memory. We also use two Tesla C870 cards each with 1.5GB video memory for the multi-GPU environment. Our implementation runs on Windows XP 64-bit edition with CUDA 1.1 [13] and ForceWare graphics driver 169.21.

Figure 7 shows the Shepp-Logan phantom [32], namely a standard phantom widely used for evaluation. The data size is given by $U = V = N = 512$, $K = 360$, and $R = 256$.

6.1. Performance Comparison on Single-GPU Environment

Table 2 shows the execution time needed for reconstruction of the Shepp-Logan phantom. Since the number K of projections differs from previous results [1, 3], we have normalized them to the same condition as previous work [1, 2] did in the paper. The proposed method achieves the fastest time of 5.6 seconds, which is 45% and

Table 2: Performance comparison with previous methods. Throughput is presented by projections per second (pps).

Method	Hardware	Execution time (s)	Throughput (pps)
CPU [1]	Xeon 3.06 GHz	135.4	2.8
CBE [1]	Mercury CBE	19.1	18.8
OpenGL [2]	GeForce 8800 GTX	8.9	40.4
OpenGL [14]	GeForce 8800 GTX	8.1	44.4
Previous CUDA [3]	GeForce 8800 GTX	7.9	45.6
This work	Tesla C870	5.8	62.1
This work	GeForce 8800 GTX	5.6	64.3

Table 3: Breakdown of execution time on GeForce 8800 GTX.

Breakdown	This work (s)	Previous CUDA [3] (s)
Initialization	0.1	N/A
Projection download	0.1	0.2
Filtering	0.8	0.7
Backprojection	4.2	6.1
Volume readback	0.4	0.9
Total	5.6	7.9

41% faster than the fastest OpenGL-based method [14] and the previous CUDA-based method [3], respectively. With respect to the throughput, this performance is equivalent to 64.3 projections per second (pps) while the image acquisition speed in recent CT scans ranges from 30 to 50 pps [2]. Thus, the performance bottleneck now moves from the reconstruction stage to the image acquisition stage, making it possible to produce the entire volume immediately after a scan rotation.

Table 3 shows a breakdown analysis of execution time comparing our method with the previous CUDA-based method [3]. The main acceleration is achieved at the backprojection stage. As compared with the previous method, our method processes multiple projections at each kernel invocation in order to reduce the amount of global memory accesses, as shown at line 39 in Fig. 5. This reduction technique cannot be applied to the previous method, which deals with a single projection at a time. Since we use $I = 3$, the proposed method achieves 67% less data fetches between MPs and global memory. We also can see that the proposed method transfers the volume two times faster than the previous method. We think that this is due to the machine employed for the previous results, because the transfer rate is mainly determined by the chipset.

We next analyze the effective performance of the filtering and backprojection kernels in terms of the computational performance and the memory bandwidth. Figure 8 shows the measured performance with the theoretical peak performance of GeForce 8800 GTX and Tesla C870 cards. Both cards have the same arithmetic performance but GeForce has 11% higher memory bandwidth than Tesla. The measured values are determined according to the number of instructions in parallel thread execution (PTX) code, namely assembly code running on the GPU. For example, the texture unit in

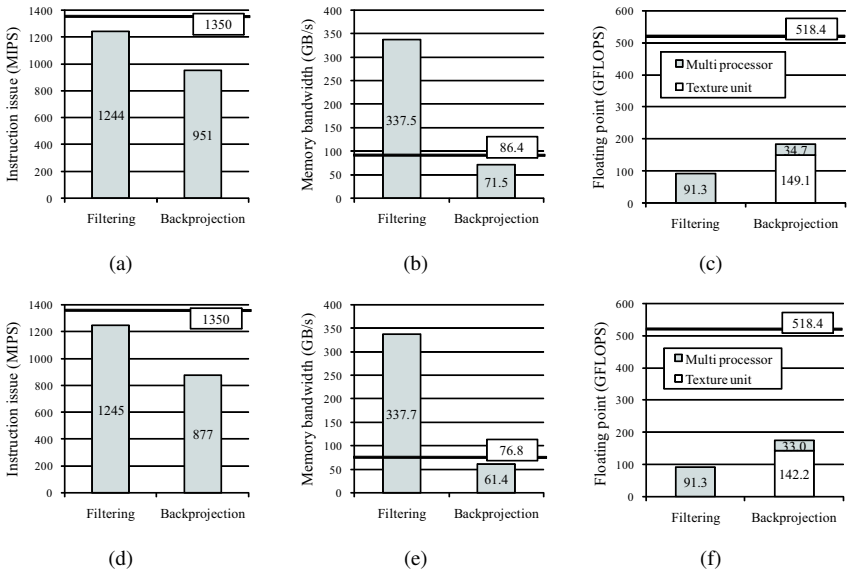


Figure 8: Effective performance in terms of instruction issue rate, memory bandwidth, and arithmetic performance. (a)–(c) Results on GeForce 8800 GTX and (d)–(f) those on Tesla C870. Theoretical values are also shown as lines. The theoretical memory bandwidth is given by device memory. The effective memory bandwidth can be higher than the theoretical value due to cache effects.

the GPU processes $13KN^3$ operations in total, because textures are accessed KN^3 times during backprojection and a texture access involves 13 floating-point operations to obtain an interpolated texel [13].

Figure 8 indicates that the instruction issue rate limits the performance of the filtering kernel. Due to this performance bottleneck, the floating point performance results in 91.3 GFLOPS, which is equivalent to 18% of the peak performance. On the other hand, the effective memory bandwidth reaches 337 GB/s, which is higher than the theoretical value. This is due to the fast shared memory and the cache mechanism working for constant memory. As compared with the variable data in global memory, the filtering kernel accesses 260 and 130 times more shared data and constant data, namely arrays $array$ and G at line 26 in Fig. 5, respectively.

In contrast, the memory bandwidth is a performance bottleneck in the backprojection kernel. This kernel has more data access to global memory, which does not have cache effects. Actually, global memory is used for 40% of total amount. Thus, the backprojection kernel has lower effective bandwidth than the filtering kernel. However, our backprojection kernel achieves almost the same bandwidth as bandwidthTest, which is a benchmark program distributed with the nVIDIA sample code [13]. In addition, the backprojection kernel achieves relatively higher floating point performance because it exploits the texture unit for linear interpolation. The effective performance reaches 183.8 GFLOPS including 149.1 GFLOPS observed at texture units. Exploiting this hardware is important to offload workloads from SPs.

Table 4: Backprojection performance on GeForce 8800 GTX with different acceleration techniques. Methods #1 and #2 are subsets of the proposed method.

Acceleration technique	Method			
	Naive	#1	#2	Proposed
1. Data reuse	no	yes	yes	yes
2. Off-chip memory access reduction	no	no	yes	yes
3. Loop unrolling	no	no	no	yes
Backprojection time (s)	27.1	24.3	5.6	4.2

6.2. Breakdown Analysis

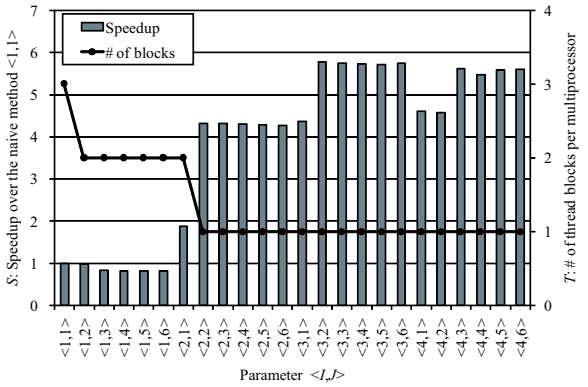
We measure the performance of two subset implementations to clarify the impact of each acceleration technique in terms of backprojection performance. Table 4 shows the details of each implementation with the measured time for the Shepp-Logan phantom. Our acceleration techniques achieve 6.5 times higher backprojection performance than the naive method in Table 4. This improvement is mainly achieved by off-chip memory access reduction that reduces backprojection time to 5.6 seconds with a speedup of 4.8. As compared with the naive method, method #2 has 66% less access to off-chip memory, leading to 44% reduction of device memory access in total.

The loop unrolling technique further reduces the time from 5.6 to 4.2 seconds. As we mentioned in Section 5.2, this technique intends to hide the memory latency with computation, so that we investigate the assembly code to explain this reduction. Since we use $J = 2$ for the proposed method, we think that memory accesses for $j = 0$ can be overlapped with computation for $j = 1$ (line 31 in Fig. 5). We then find that such overlapping computation takes approximately 1.3 seconds if it is not overlapped with memory accesses. This explains why the time is reduced by 1.4 seconds.

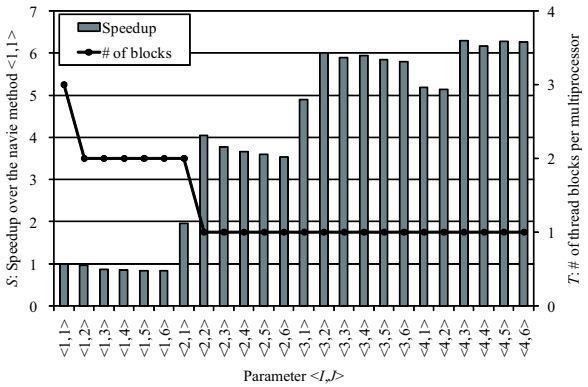
We next investigate the performance with different parameters $\langle I, J \rangle$. Figure 9 shows the number T of TBs per MP and the speedup S over the method with $\langle I, J \rangle = \langle 1, 1 \rangle$. As we mentioned in Section 5.2, increasing I or J requires more registers to run the kernel. Due to this consumption, T decreases with the increase of I or J. For example, the proposed method uses 30 registers per thread when $\langle I, J \rangle = \langle 2, 3 \rangle$. In our kernel, a TB consists of 256 threads, so that it requires 7680 registers per TB while 8192 registers are available for an MP in a GeForce 8800 GTX card. Thus, we have $T = 1$ for $\langle I, J \rangle = \langle 2, 3 \rangle$.

An interesting behavior in Fig. 9 is that we observe $S > 4$ if and only if $T = 1$. This behavior is not consistent with the highly-threaded GPU architecture, which is designed to concurrently run multiple TBs on every MP in order to hide the memory latency. One possible reason for this is that the cache mechanism might be not optimized well to deal with such concurrent TBs. For example, the cache data could be cleaned every time when activating the background TB, because each TB takes the responsibility for different volume region. Actually, there is no cache mechanism for global memory, which is designed to hide the memory latency by switching the current TB to another TB when an MP has to wait for data fetch. Thus, this behavior might explain why global memory does not have a cache mechanism.

The impact of increasing I can be seen at $\langle I, J \rangle = \langle 2, 1 \rangle$. As compared with any



(a)



(b)

Figure 9: Relationship between the number of thread blocks per multiprocessor and the relative backprojection performance with different parameters $\langle I, J \rangle$. (a) Results on GeForce 8800 GTX and (b) those on Tesla C870. The texture bandwidth is maximized if a single thread block is assigned to every multiprocessor.

condition with $I = 1$, this condition has 50% less data access to global memory, so that we achieve $S = 1.93$. However, we cannot see this reduction effect when $I > 2$. In these cases, the data amount does not dominate the performance due to the cache mechanism. Actually, the speedup S sharply increases at $\langle I, J \rangle = \langle 2, 2 \rangle$ though the kernel fetches the same data amount as when $\langle I, J \rangle = \langle 2, 1 \rangle$.

Finally, the performance at $\langle I, J \rangle = \langle 2, 2 \rangle$ is almost the same as that presented in [3]. Therefore, we think that reducing the number of registers is not always a good optimization strategy for memory-intensive applications that use textures for hardware interpolation. Instead, it is important to allow every TB to use a dedicated cache with achieving a higher locality of data access. That is, using more registers will lead to higher performance in texture-based methods. Thus, our texture-based method takes exactly the opposite strategy from the previous method [3], which we mentioned in Section 2.

Table 5: Breakdown of execution time on dual Tesla C870 cards. Since high-resolution data requires more space than the memory capacity, it requires projection readback after the backprojection stage.

Breakdown	Low-resolution data ($N = U = V = 512$)			High-resolution data ($N = U = V = 1024$)		
	Single GPU	Projection parallelism	Voxel parallelism	Single GPU	Projection parallelism	Voxel parallelism
Initialization	0.11	0.14	0.12	0.71	0.75	0.43
Projection download	0.14	0.07	0.14	6.67	4.10	4.15
Filtering	0.80	0.40	0.80	3.28	1.64	3.28
Backprojection	4.20	2.09	2.10	31.69	15.76	15.84
Projection readback	—	—	—	0.89	0.56	1.29
Volume readback	0.37	0.67	0.34	2.42	3.29	2.48
Post-processing	—	0.34	0.00	—	2.09	0.00
Total	5.62	3.71	3.50	45.66	28.19	27.47

6.3. Performance on Multi-GPU Environment

Table 5 shows the execution time of the single-GPU implementation, the projection parallelism scheme, and the voxel parallelism scheme on the dual Tesla C870 configuration. We use two datasets to understand the scalability of parallelization schemes: low-resolution data and high-resolution data. The high-resolution data requires at least 4GB of video memory to store the entire volume. To deal with such a large-scale dataset, the projection parallelism scheme divides the volume into P portions and then processes each subvolume in a serial manner. We use $P = 8$ for this experiment. In contrast, this volume division is naturally implemented by the voxel parallelism scheme. We also perform data reuse at the filtering stage in both schemes. That is, the filtered projections are computed only for the first subvolume and are temporally pushed to main memory after backprojection. However, they are loaded again to skip the filtering stage for the remaining $P - 1$ subvolumes.

By comparing performance results between low-resolution data and high-resolution data in Table 5, we find that the analytical model in Section 5.3 successfully characterizes the complexity of filtering and backprojection stages. For example, both parallelization schemes reduce the backprojection time of the single-GPU configuration approximately into a half. Since we use two Tesla cards ($C = 2$), this behavior is clearly represented by the model in Table 5.

Table 5 also shows a trade-off relationship between the download time and the readback time for low-resolution data. That is, the projection parallelism scheme reduces the download time while the voxel parallelism scheme reduces the readback time. However, this relationship is not observed when processing large-scale data. For example, both schemes take approximately 4.1 seconds to download high-resolution projections. This similar behavior can be explained as follows. Since the projection parallelism scheme iterates the download stage P times, it requires $O(PKUV)$ time to download projections (see also Table 5). In contrast, the voxel parallelism scheme iterates the download stage P/C times, so that requires the same $O(PKUV)$ time. In other words, the projection parallelism scheme loses its advantage when dealing with such a large-scale data.

According to the extended model mentioned above, the high-resolution data requires at least 32 times more data downloads than the low-resolution data. Thus, the

download/readback time reveals as a performance bottleneck of high-resolution reconstruction. For example, the voxel parallelism scheme spends 29% of the entire time for data transfer between main memory and video memory. Since such data transfer is currently processed in a serial manner, it takes at least 7.9 seconds to reconstruct high-resolution data though we minimize the filtering and backprojection time using more graphics cards. Thus, as Amdahl's law [33] points out, we think that a parallel data transfer mechanism is essential to increase the overall performance with the number C of graphics cards.

Finally, the projection parallelism scheme effectively sends the high-resolution volume to main memory. This is due to the increased data size, because the effective bandwidth generally increases with the data amount. In fact, the effective bandwidth between main memory and video memory increases from 1.3 GB/s to 1.6 GB/s in this case.

7. Conclusion

We have presented a fast method for CB reconstruction on CUDA-enabled GPUs. The proposed method is based on the FDK algorithm accelerated using three techniques: off-chip memory access reduction; loop unrolling; and multithreading. We have described how these techniques can be incorporated into CUDA code.

The experimental results show that the proposed method takes 5.6 seconds to reconstruct a 512^3-voxel volume from 360 512^2-pixel projections. This execution time is at least 41% faster than previous methods, allowing us to obtain the entire volume immediately after a scan rotation of the flat panel detector. We also find that the filtering and backprojection performances are limited by the instruction issue rate and the memory bandwidth, respectively. With respect to acceleration techniques, off-chip memory access reduction is essential to run the GPU as an accelerator for the CPU. We also demonstrate out-of-core reconstruction for large-scale datasets, up to 1024^3-voxel volume.

Future work includes the support of streaming mechanism [13]. Such a mechanism is useful to overlap kernel execution with data transfer between main memory and video memory.

Acknowledgments

This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(2) (20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.

References

[1] M. Kachelrieß, M. Knaup, O. Bockenbach, Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware, Medical Physics 34 (4) (2007) 1474–1486.

[2] F. Xu, K. Mueller, Real-time 3D computed tomographic reconstruction using commodity graphics hardware, Physics in Medicine and Biology 52 (12) (2007) 3405–3419.

[3] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, Fast GPU-based CT reconstruction using the common unified device architecture (CUDA), in: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'07), 2007, pp. 4464–4466.

[4] D. Riabkov, X. Xue, D. Tubbs, A. Cheryauka, Accelerated cone-beam backprojection using GPU-CPU hardware, in: Proc. 9th Int'l Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D '07), 2007, pp. 68–71.

[5] X. Zhao, J. Bian, E. Y. Sidky, S. Cho, P. Zhang, X. Pan, GPU-based 3D cone-beam CT image reconstruction: application to micro CT, in: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'07), 2007, pp. 3922–3925.

[6] T. Schiwietz, S. Bose, J. Maltz, R. Westermann, A fast and high-quality cone beam reconstruction pipeline using the GPU, in: Proc. SPIE Medical Imaging (MI 2007), 2007, pp. 1279–1290.

[7] P. B. Noël, A. M. Walczak, K. R. Hoffmann, J. Xu, J. J. Corso, S. Schafer, Clinical evaluation of GPU-based cone beam computed tomography, in: Proc. High-Performance Medical Image Computing and Computer Aided Intervention (HP-MICCAI'08), 2008.

[8] N. Gac, S. Mancini, M. Desvignes, Hardware/software 2D-3D backprojection on a SoPC platform, in: Proc. 21st ACM Symp. Applied Computing (SAC'06), 2006, pp. 222–228.

[9] D. Luebke, G. Humphreys, How GPUs work, Computer 40 (2) (2007) 96–100.

[10] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, Synergistic processing in cell's multicore architecture, IEEE Micro 26 (2) (2006) 10–24.

[11] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.

[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.

[13] nVIDIA Corporation, CUDA Programming Guide Version 1.1 (Nov. 2007).
URL `http://developer.nvidia.com/cuda/`

[14] F. Ino, S. Yoshida, K. Hagihara, RGBA packing for fast cone beam reconstruction on the GPU, in: Proc. SPIE Medical Imaging (MI 2009), 2009, 8 pages (CD-ROM).

[15] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, 5th Edition, Addison-Wesley, Reading, MA, 2005.

[16] Y. Okitsu, F. Ino, K. Hagihara, Fast cone beam reconstruction using the CUDA-enabled GPU, in: Proc. 15th Int'l Conf. High Performance Computing (HiPC'08), 2008, pp. 108–119.

[17] L. A. Feldkamp, L. C. Davis, J. W. Kress, Practical cone-beam algorithm, J. Optical Society of America 1 (6) (1984) 612–619.

[18] H. Yang, M. Li, K. Koizumi, H. Kudo, Accelerating backprojections via CUDA architecture, in: Proc. 9th Int'l Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D '07), 2007, pp. 52–55.

[19] D. W. Shattuck, J. Rapela, E. Asma, A. Chatzioannou, J. Qi, R. M. Leahy, Internet2-based 3D PET image reconstruction using a PC cluster, Physics in Medicine and Biology 47 (15) (2002) 2785–2795.

[20] Y. Kawasaki, F. Ino, Y. Mizutani, N. Fujimoto, T. Sasama, Y. Sato, N. Sugano, S. Tamura, K. Hagihara, High-performance computing service over the Internet for intraoperative image processing, IEEE Trans. Information and Technology in Biomedicine 8 (1) (2004) 36–46.

[21] F. Ino, K. Ooyama, K. Hagihara, A data distributed parallel algorithm for nonrigid image registration, Parallel Computing 31 (1) (2005) 19–43.

[22] B. Cabral, N. Cam, J. Foran, Accelerated volume rendering and tomographic reconstruction using texture mapping hardware, in: Proc. 1st Symp. Volume Visualization (VVS'94), 1994, pp. 91–98.

[23] K. Mueller, R. Yagel, Rapid 3-D cone-beam reconstruction with the simulatneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware, IEEE Trans. Medical Imaging 19 (12) (2000) 1227–1237.

[24] F. Xu, K. Mueller, Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware, IEEE Trans. Nuclear Science 52 (3) (2005) 654–663.

[25] H. M. Hudson, R. S. Larkin, Accelerated image reconstruction using ordered subsets of projection data, IEEE Trans. Medical Imaging 13 (4) (1994) 601–609.

[26] A. H. Anderson, A. C. Kak, Simultaneous algebraic reconstruction technique (SART): A superior implementation of the ART algorithm, Ultrasonic Imaging 6 (1984) 81–94.

[27] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: A system for programming graphics hardware in a C-like language, ACM Trans. Graphics 22 (3) (2003) 896–897.

[28] T. Ikeda, F. Ino, K. Hagihara, A code motion technique for accelerating general-purpose computation on the GPU, in: Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06), 2006, 10 pages (CD-ROM).

[29] M. Grass, T. Köhler, R. Proksa, 3D cone-beam CT reconstruction for circular trajectories, Physics in Medicine and Biology 45 (2) (2000) 329–347.

[30] H. Turbell, Cone-beam reconstruction using filtered backprojection, Ph.D. thesis, Linköpings Universitet, Linköping, Sweden (Jan. 2001).

[31] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, 2nd Edition, Addison-Wesley, Reading, MA, 2003.

[32] L. A. Shepp, B. F. Logan, The fourier reconstruction of a head section, IEEE Trans. Nuclear Science 21 (3) (1974) 21–43.

[33] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proc. the AFIPS Conf., 1967, pp. 483–485.