Accelerating Cone Beam Reconstruction Using the CUDA-enabled GPU[⋆]

Yusuke Okitsu, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{y-okitu,ino}@ist.osaka-u.ac.jp

Abstract. Compute unified device architecture (CUDA) is a software development platform that enables us to write and run general-purpose applications on the graphics processing unit (GPU). This paper presents a fast method for cone beam reconstruction using the CUDA-enabled GPU. The proposed method is accelerated by two techniques: (1) off-chip memory access reduction; and (2) memory latency hiding. We describe how these techniques can be incorporated into CUDA code. Experimental results show that the proposed method runs at 82% of the peak memory bandwidth, taking 5.6 seconds to reconstruct a 512^3-voxel volume from 360 512^2-pixel projections. This performance is 18% faster than the prior method. Some detailed analyses are also presented to understand how effectively the acceleration techniques increase the reconstruction performance of a naive method.

1 Introduction

Cone beam (CB) reconstruction is an imaging process for producing a three-dimensional (3-D) volume from a sequence of 2-D projections obtained by a CB computed tomography (CT) scan. This reconstruction technique is integrated into many mobile C-arm CT systems in order to assist the operator during image-guided surgery. In general, a CB reconstruction task should be completed within ten seconds because the operator has to stop the surgical procedure until obtaining the intraoperative volume. However, it takes 3.21 minutes to obtain a 512^3-voxel volume on a single 3.06 GHz Xeon processor [1]. Accordingly, many projects are trying to accelerate CB reconstruction using various accelerators, such as the graphics processing unit (GPU) [2–6], Cell [1], and FPGA [7].

To the best of our knowledge, Xu et al. [2] show the fastest method using the GPU, namely a commodity chip designed for acceleration of graphics tasks. Their method is implemented using the OpenGL library in order to take an advantage of graphics techniques such as early fragment kill (EFK). It takes 8.3 seconds to reconstruct a 512^3-voxel volume from 360 projections. In contrast to this graphics-based implementation strategy, a non-graphics implementation strategy is proposed by Scherl et al. [3]. They use compute unified device architecture (CUDA) [8] to implement CB reconstruction

⋆ 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.

on the GPU. The reconstruction of a 512^3-voxel volume from 414 projections takes 12.02 seconds, which is slightly longer than the graphics-based result [2]. However, it is still not clear whether the CUDA-based strategy will outperform the graphics-based strategy, because their implementation is not presented in detail. In particular, optimization techniques for CUDA programs are of great interest to the high-performance computing community.

In this paper, we propose a CUDA-based method capable of accelerating CB reconstruction on the CUDA-enabled GPU. Our method is based on the Feldkamp, Davis, and Kress (FDK) reconstruction algorithm [9], which is used in many prior projects [1–5, 7, 10]. We optimize the method using two acceleration techniques: (1) one is for reducing the number and amount of off-chip memory accesses; and (2) another for hiding the memory latency with independent computation. We also show how effectively these techniques contribute to higher performance, making it clear that the memory bandwidth and the instruction issue rate limit the performance of the proposed method.

2 Related Work

Xu et al. [2] propose an OpenGL-based method accelerated using the graphics pipeline in the GPU. They realize a load balancing scheme by moving instructions from fragment processors to vertex processors, each composing the pipeline. This code motion technique also contributes to reduce the computational complexity [11]. Furthermore, their method uses the EFK technique to restrict computation to voxels within the region of interest (ROI). Although this fragment culling technique leads to acceleration, we cannot obtain the correct data outside the ROI. In contrast, our goal is to achieve higher reconstruction performance for the entire volume.

Scherl et al. [3] show a CUDA-based method with a comparison to a Cell-based method. They claim that their method reduces the number of instructions and the usage of registers. In contrast, our acceleration techniques focus on reducing the number and amount of off-chip memory accesses and hiding the memory latency with computation. Such memory optimization is important to improve the performance of the FDK algorithm, which can be classified into a memory-intensive problem.

Another acceleration strategy is to perform optimization at the algorithm level. For example, the rebinning strategy [12] rearranges and interpolates CB projections to convert them into parallel beam projections. This geometry conversion simplifies the backprojection operation needed for the FDK reconstruction. One drawback of the rebinning strategy is that it creates artifacts in the final volume. Using this rebinning strategy, Li et al. [10] develop a fast backprojection method for CB reconstruction. Their method is implemented using CUDA and takes 3.6 seconds to perform backprojection of 360 512^3-pixel projections. In contrast, our method accelerates the entire FDK algorithm for CB projections without rebinning.

3 Overview of CUDA

Figure 1 illustrates an overview of the CUDA-enabled GPU. The GPU consists of multiprocessors (MPs), each having multiple stream processors (SPs). Each MP has small

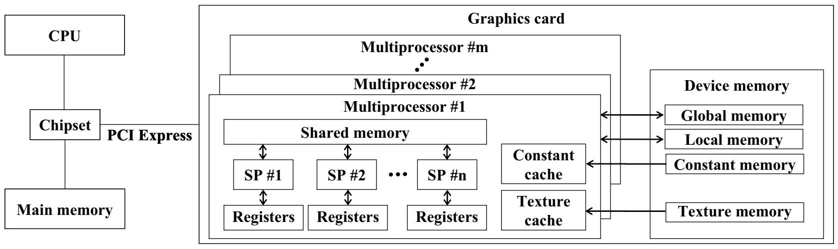

Fig. 1. Architecture of CUDA-enabled GPU. SP denotes a stream processor.

on-chip memory, called shared memory, which can be accessed from internal SPs as fast as registers. However, it is not shared between different MPs. Due to this constraint, threads are classified into groups and each group is called as a block, which is the minimum allocation unit assigned to an MP. That is, developers have to write their code such that there is no dependence between threads in different blocks. On the other hand, threads in the same block are allowed to have dependence because they can communicate each other by shared memory.

CUDA also exposes the memory hierarchy to developers, allowing them to maximize application performance by optimizing data access. As shown in Fig. 1, there is off-chip memory, called device memory, containing 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. 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 [8]. Local memory cannot be explicitly used by developers. This memory space is implicitly used by the CUDA compiler in order to avoid resource consumption. For example, an array will be allocated to such space if it is too large for register space. Local memory cannot be accessed in a coalesced manner, so that it is better to eliminate such inefficient accesses hidden in the kernel code.

4 Feldkamp Reconstruction

The FDK algorithm [9] consists of the filtering stage and the backprojection stage. Suppose that a series of $U \times V$-pixel projections $P_1, P_2, \dots P_K$ are obtained by a scan rotation of a detector in order to create an N^3-voxel volume F. The algorithm then applies the Shepp-Logan filter [13] to each projection, which gives a smoothing effect to minimize noise propagation at the backprojection stage. At this filtering stage, the pixel value $P_n(u,v)$ at point (u,v) is converted to value $Q_n(u,v)$ such that:

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

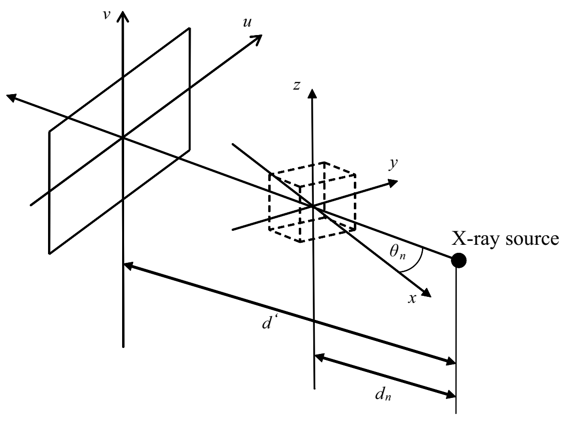

Fig. 2. Coordinate system for backprojection. The xyz space represents the volume while the uv plane represents a projection that is to be backprojected to the volume.

where S represents the filter size and $W_1(s, v)$ represents the weight value given by $W_1(s, v) = d'/\sqrt{d'^2 + s^2 + v^2}$, 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 projection P_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 is 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:

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

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

$$W_2(x, y, n) = \left(\frac{d_n}{d_n - xcos\theta_n - ysin\theta_n}\right)^2, \tag{3}$$

$$u(x, y, n) = \frac{d'(-xsin\theta_n + ycos\theta_n)}{d_n - xcos\theta_n - ysin\theta_n}, \tag{4}$$

$$v(x, y, z, n) = \frac{d'z}{d_n - xcos\theta_n - ysin\theta_n}. \tag{5}$$

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

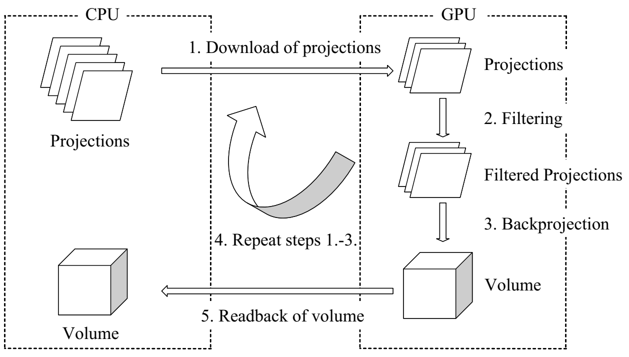

Fig. 3. Overview of the proposed method. Projections are serially sent to the GPU in order to accumulate their pixels into the volume in video memory.

5 Proposed Method

To make the description easier to understand, we first show a naive method and then the proposed method with acceleration techniques.

5.1 Parallelization Strategy

Since a 512^3-voxel volume requires at least 512 MB of memory space, it is not easy for commodity GPUs to store both the entire volume and the projections in device memory. To deal with this memory capacity problem, we have decided to store the entire volume in device memory because earlier projections can be processed and removed before the end of a scan rotation. In other words, this decision allows us to structure the reconstruction procedure into a pipeline. Figure 3 shows an overview of our reconstruction method. In the naive method, the first projection P_1 is transferred to global memory, which is then filtered and backprojected to the volume F in global memory. This operation is iteratively applied to the remaining projections to obtain the final accumulated volume F. See also Fig. 4 for the pseudocode of the naive method.

In Fig. 4, the filtering stage is parallelized in the following way. Eq. (1) means 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 column u refer partly the same pixels in projection P_n. Therefore, it is better to use shared memory to save the memory bandwidth. Thus, we have decided to write the filtering kernel such that a thread block is responsible for applying the filtering operation to pixels in a column. On the other hand, a thread is responsible for computing a pixel value $Q_n(u, v)$. As shown in Fig. 4, threads in the same block cooperatively copy a column u to shared memory at line 13, which are then accessed instead of the original data in global memory at line 15.

Similarly, there is no constraint at the backprojection stage in terms of parallelism. That is, any voxel can be processed at the same time. However, it is better to use 1-D or 2-D thread blocks rather than 3-D thread blocks in order to reduce the computational

```
Input: Projections $P_1 \dots P_K$, filter size $S$ and
       parameters $d', d_1 \dots d_K, \theta_1 \dots \theta_K$
Output: Volume $F$

Algorithm NaiveReconstruction()
 1: Initialize volume F
 2: for $n = 1$ to $K$ do
 3:     Transfer projection $P_n$ to global memory
 4:     $Q \leftarrow$ FilteringKernel($P_n, S$)
 5:     Bind filtered projection $Q$ as a texture
 6:     $F \leftarrow$ BackprojectionKernel($Q, d', d_n, \theta_n, n$)
 7: end for
 8: Transfer volume $F$ to main memory

Function FilteringKernel($P$, $S$)
 9: __shared__ float $array[U]$        // $U$: projection width
10: $u \leftarrow \mathrm{index(threadID)}$     // returns responsible $u$
11: $v \leftarrow \mathrm{index(blockID)}$
12: Initialize $Q(u, v)$
13: $array[u] \leftarrow W_1(u, v) * P(u, v) * 2/\pi^2$
14: for $s = -S$ to $S$ do
15:     $Q(u, v) \leftarrow Q(u, v) + array[u + s]/(1 - 4s^2)$
16: end for

Function BackprojectionKernel($Q, d', d_n, \theta_n, n$)
17: $x \leftarrow \mathrm{index(blockID)}$
18: $y \leftarrow \mathrm{index(threadID)}$
19: $u \leftarrow u(x, y, n)$              // Eq. (4)
20: $v \leftarrow v(x, y, 0, n)$           // Eq. (5)
21: $v' \leftarrow v'(x, y, n)$            // Eq. (6)
22: for $z = 0$ to $N - 1$ do
23:     $F(x, y, z) \leftarrow F(x, y, z) + W_2(x, y, n) * Q(u, v)$
24:     $v \leftarrow v + v'$
25: end for
```

Fig. 4. Pseudocode of the native method. This code is a simplified version.

complexity by data reuse. This data reuse technique can be explained by Eqs. (3) and (4), which indicate that $W_2(x, y, n)$ and $u(x, y, n)$ do not depend on z. Therefore, these two values can be reused for voxels in a straight line along the z-axis: line $(X, Y, 0) - (X, Y, N - 1)$, where X and Y are constant values in the range $[0, N - 1]$. To perform this data reuse, our naive method employs 1-D thread blocks (but 2-D blocks after optimization shown later in Section 5.2) that assign such voxels to the same thread. In summary, a thread is responsible for a line while a thread block is responsible for a set of lines: plane $x = X$ for thread block X, where $0 \leq X \leq N - 1$.

The data reuse can be further applied to reduce the complexity of Eq. (5). Although $v(x, y, z, n)$ depends on z, it can be rewritten as $v(x, y, z, n) = v'(x, y, n)z$, where

$$v'(x, y, n) = \frac{d'}{d_n - xcos\theta_n - ysin\theta_n}. \tag{6}$$

```
Function OptimizedBackprojectionKernel($Q[I*J], d', d_n[I*J], \theta_n[I*J], n$)
 1: var $u[I], v[I], v'[I], w[I]$
 2: $x \leftarrow \mathrm{index}(\mathrm{blockID}, \mathrm{threadID})$
 3: $y \leftarrow \mathrm{index}(\mathrm{blockID}, \mathrm{threadID})$
 4: for $j = 0$ to $J - 1$ do        // unrolled
 5:   for $i = 0$ to $I - 1$ do
 6:     $w[i] \leftarrow W_2(x, y, 3j + i + n)$        // Eq. (3)
 7:     $u[i] \leftarrow u(x, y, 3j + i + n)$          // Eq. (4)
 8:     $v[i] \leftarrow v(x, y, 0, 3j + i + n)$       // Eq. (5)
 9:     $v'[i] \leftarrow v'(x, y, 3j + i + n)$        // Eq. (6)
10:   end for
11:   for $z = 0$ to $N - 1$ do
12:     $F(x, y, z) \leftarrow F(x, y, z) + w[0] * Q[3j](u[0], v[0])$
                          $+ w[1] * Q[3j + 1](u[1], v[1])$
                          $\cdots$
                          $+ w[I - 1] * Q[3j + (I - 1)](u[I - 1], v[I - 1])$
13:     for $k = 0$ to $I - 1$ do
14:       $v[k] \leftarrow v[k] + v'[k]$
15:     end for
16:   end for
17: end for
```

Fig. 5. Pseudocode of the proposed method. The actual code is optimized by loop unrolling, for example.

Therefore, we can precompute $v'(x, y, n)$ for any z (line 21), in order to incrementally compute Eq. (5) at line 24.

Note that the filtered projection data is accessed as a texture. 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

The acceleration techniques we propose in this paper optimize the backprojection kernel of the naive method. These techniques are motivated to maximize the effective memory bandwidth because the backprojection stage is a memory-intensive operation. We maximize the effective bandwidth by two techniques which we mentioned in Section 1. The naive method presented in Fig. 4 is modified to the optimized code shown in Fig. 5 by the following five steps.

1. Memory access coalescing [8]. This technique is important to achieve a full utilization of the wide memory bus available in the GPU. We store the volume data in global memory so that the memory accesses can be coalesced into a single contiguous, aligned memory access. This can be realized by employing 2-D thread blocks instead of 1-D blocks. It also improves the locality of texture access, which leads to a higher utilization of the texture cache.

2. Global memory access reduction. We modify the kernel to perform backprojection of I projections at a time, where I represents the number of projections processed by a single kernel invocation. This modification reduces the number of global memory accesses to $1/I$ because it allows us to write temporal voxel values to local memory before writing the final values to global memory. We cannot use registers because a large array of size N is needed to store the temporal values for all z (line 23 in Fig. 4). Note that the increase of I results in more consumption of resources such as registers. We currently use $I = 3$, which is experimentally determined for the target GPU.
3. Local memory access reduction. The technique mentioned above decreases accesses to global memory but increases those to local memory. In order to reduce them, we pack I successive assignments into a single assignment. This modification is useful if the assignments have the same destination variable placed in local memory.
4. Local memory access elimination. We now have a single assignment for accumulation, so that we can write the accumulated values directly to global memory, as shown at line 12 in Fig. 5.
5. Memory latency hiding. 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 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. As we did for I, we have experimentally decided to use $J = 2$.

6 Experimental Results

In order to evaluate the performance of the proposed method, we now show some experimental results including a breakdown analysis of execution time and a comparison with prior methods: the OpenGL-based method [2]; the prior CUDA-based method [3]; the Cell-based method [1]; and the CPU-based method [1]. For experiments, we use a desktop PC equipped with a Core 2 Duo E6850 CPU, 4GB main memory, and an nVIDIA GeForce 8800 GTX GPU with 768MB video memory. Our implementation runs on Windows XP with CUDA 1.1 and ForceWare graphics driver 169.21. Figure 6 shows the Shepp-Logan phantom [13], namely a standard phantom widely used for evaluation. The data size is given by $U = V = N = 512$, $K = 360$, and $S = 256$.

6.1 Performance Comparison

Table 1 shows the execution time needed for reconstruction of the Shepp-Logan phantom. Since the number K of projections differs from prior results [1, 3], we have normalized them to the same condition ($K = 360$ and $U = V = 512$) as prior work [1, 2] did in the paper. The proposed method achieves the fastest time of 5.6 seconds, which is 29% and 18% faster than the OpenGL-based method [2] and the prior CUDA-based method [3], respectively. This performance is equivalent to 64.3 projections per second (pps), which represents the throughput in terms of input projections. On the other hand,

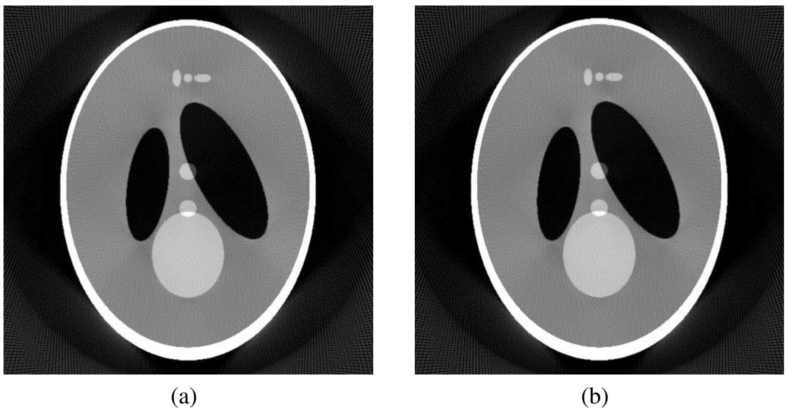

Fig. 6. Sectional views of the Shepp-Logan phantom [13] reconstructed (a) by the GPU and (b) by the CPU.

Table 1. Performance comparison with prior 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
Cell [1]	Cell Broadband Engine	9.6	37.6
OpenGL [2]	GeForce 8800 GTX	8.9	40.5
Prior CUDA [3]	GeForce 8800 GTX	7.9	45.5
OpenGL w/ EFK [2]	GeForce 8800 GTX	6.8	52.9
Proposed method	GeForce 8800 GTX	5.6	64.3

the image acquisition speed in recent CT scans ranges from 30 to 50 pps [2]. Therefore, the performance achieved by the proposed method is sufficient enough to produce the entire volume immediately after a scan rotation. Note here that the OpenGL-based method is also faster than the image acquisition speed if it is accelerated by the EFK technique. However, as we mentioned in Section 2, this technique does not reconstruct the volume area outside the ROI. In contrast, the proposed method reconstructs the entire volume within a shorter time.

Table 2 shows a breakdown of execution time comparing our method with the prior CUDA-based method. We can see that the acceleration is mainly achieved at the back-projection stage. As compared with the prior method, our method processes multiple projections at a kernel invocation. Therefore, we can reduce the number and amount of global memory accesses by packing I assignments into a single assignment, as shown at line 12 in Fig. 5. This reduction technique cannot be applied to the prior method, which processes a single projection at a time. Since we use $I = 3$, the proposed method achieves 67% less data transfer between MPs and global memory. With respect to the filtering stage, our method achieves the same performance as the prior method, which uses the nVIDIA CUFFT library. In this sense, we think that our filtering kernel achieves performance competitive to the vendor library. We also can see that the proposed method

Table 2. Breakdown of execution time.

Breakdown item	Proposed method (s)	Prior CUDA [3] (s)
Initialization	0.1	N/A
Projection download	0.2	0.2
Filtering	0.7	0.7
Backprojection	4.3	6.1
Volume readback	0.3	0.9
Total	5.6	7.9

Table 3. Effective floating point performance and memory bandwidth of our kernels. We assume that the GPU issues a single instruction per clock cycle and a stream processor executes two floating point (multiply-add) arithmetics per clock cycle. The effective memory bandwidth can be higher than the theoretical value due to cache effects.

Performance measure		Measured value		Theoretical value
		Filtering	Backprojection	
Instruction issue (MIPS)		1391	980	1350
Floating point (GFLOPS)	Processing units	105.8	38.3	345.6
	Texture units	—	124.3	172.8
	Total	105.8	162.6	518.4
Memory bandwidth (GB/s)		130.5	71.0	86.4

transfers the volume three times faster than the prior method. We think that this is due to the machine employed for the prior results, because the transfer rate is mainly determined by the chipset in the machine. Actually, there is no significant difference between the download rate and the readback rate in our method.

Table 3 shows the measured performance with the theoretical peak performance. We count the number of instructions in assembly code to obtain the measured values. This table indicates that the instruction issue rate limits the performance of the filtering kernel. Due to this bottleneck, the floating point performance results in 105.8 GFLOPS, which is equivalent to 20% of the peak performance. On the other hand, the effective memory bandwidth reaches 130.5 GB/s, which is higher than the theoretical value. This is due to the cache mechanism working for constant memory. The filtering kernel accesses 130 times more constant data, as compared with the variable data in global memory.

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, the backprojection kernel achieves higher floating point performance because it exploits texture units for linear interpolation. The effective performance reaches 162.6 GFLOPS including 124.3 GFLOPS observed at texture units. Exploiting this hardware interpolation is important (1) to reduce the amount of data accesses between device memory and SPs and (2) to offload workloads from SPs to texture units. For example, SPs must fetch four times more texel data if we perform linear interpolation on them.

Table 4. Backprojection performance with different acceleration techniques.

Technique	Method					
	Naive	#1	#2	#3	#4	Proposed
1. Memory access coalescing	×	○	○	○	○	○
2. Global memory access reduction	×	×	○	○	○	○
3. Local memory access reduction	×	×	×	○	○	○
4. Local memory access elimination	×	×	×	×	○	○
5. Memory latency hiding	×	×	×	×	×	○
Backprojection time (s)	436.7	27.0	15.6	13.5	5.7	4.3

6.2 Breakdown Analysis

In order to clarify how each acceleration technique contributes to higher performance, we develop five subset implementations and measure their performance. Table 4 shows the details of each implementation with the measured time needed for backprojection of the Shepp-Logan phantom. Although the naive method is slower than the CPU-based method, the acceleration techniques reduce the backprojection time to approximately 1/102. This improvement is mainly achieved by memory access coalescing that reduces backprojection time to 27.0 seconds with a speedup of 16.2. In the naive method, every thread simultaneously accesses voxels located on the same coordinate x. This access pattern is the worst case, where 16 accesses can be coalesced into a single access [8], explaining why memory access coalescing gives such a speedup. Thus, the coalescing technique is essential to run the GPU as an accelerator for the CPU.

Reducing off-chip memory accesses further accelerates the backprojection kernel. As compared with method #1 in Table 4, method #4 has 66% less access to local memory and global memory, leading to 44% reduction of device memory access. On the other hand, the backprojection time is reduced to 5.7 seconds with a speedup of 4.7, whereas the speedup estimated from the reduction ratio of 44% becomes approximately 1.8. Thus, there is a gap between the measured speedup and the estimated speedup. We think that this gap can be explained by cache effects.

The last optimization technique, namely memory latency hiding, reduces the time by 25%. We analyze 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 4 in Fig. 5). We find that such overlapping computation takes approximately 1.3 seconds under the optimal condition, where MPs execute an instruction on each clock cycle. This probably explains why the time is reduced from 5.7 to 4.3 seconds.

7 Conclusion

We have presented a fast method for CB reconstruction on the CUDA-enabled GPU. The proposed method is based on the FDK algorithm accelerated using two techniques: off-chip memory access reduction; and memory latency hiding. 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 projection images. This execution time is at least 18% faster than the prior methods [2, 3], 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, memory access coalescing is essential to run the GPU as an accelerator for the CPU.

References

1. Kachelrieß, M., Knaup, M., Bockenbach, O.: Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware. Medical Physics **34**(4) (April 2007) 1474–1486
2. Xu, F., Mueller, K.: Real-time 3D computed tomographic reconstruction using commodity graphics hardware. Physics in Medicine and Biology **52**(12) (June 2007) 3405–3419
3. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast GPU-based CT reconstruction using the common unified device architecture (CUDA). In: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'07). (October 2007) 4464–4466
4. Riabkov, D., Xue, X., Tubbs, D., Cheryauka, A.: 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). (July 2007) 68–71
5. Zhao, X., Bian, J., Sidky, E.Y., Cho, S., Zhang, P., Pan, X.: GPU-based 3D cone-beam CT image reconstruction: application to micro CT. In: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'07). (October 2007) 3922–3925
6. Schiwietz, T., Bose, S., Maltz, J., Westermann, R.: A fast and high-quality cone beam reconstruction pipeline using the GPU. In: Proc. SPIE Medical Imaging 2007. (February 2007) 1279–1290
7. Gac, N., Mancini, S., Desvignes, M.: Hardware/software 2D-3D backprojection on a SoPC platform. In: Proc. 21st ACM Symp. Applied Computing (SAC'06). (April 2006) 222–228
8. nVIDIA Corporation: CUDA Programming Guide Version 1.1 (November 2007) `http://developer.nvidia.com/cuda/`.
9. Feldkamp, L.A., Davis, L.C., Kress, J.W.: Practical cone-beam algorithm. J. Optical Society of America **1**(6) (June 1984) 612–619
10. Li, M., Yang, H., Koizumi, K., Kudo, H.: Fast cone-beam CT reconstruction using CUDA architecture. Medical Imaging Technology **25**(4) (September 2007) 243–250 (In Japanese).
11. Ikeda, T., Ino, F., Hagihara, K.: 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). (April 2006) 10 pages (CD-ROM).
12. Grass, M., Köhler, T., Proksa, R.: 3D cone-beam CT reconstruction for circular trajectories. Physics in Medicine and Biology **45**(2) (February 2000) 329–347
13. Shepp, L.A., Logan, B.F.: The fourier reconstruction of a head section. IEEE Trans. Nuclear Science **21**(3) (June 1974) 21–43