PAPER

Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volumes

Yuechao LU[†], ***Nonmember***, **Fumihiko INO**[†a)], ***and*** **Kenichi HAGIHARA**[†], ***Members***

SUMMARY This paper proposes a cache-aware optimization method to accelerate out-of-core cone beam computed tomography reconstruction on a graphics processing unit (GPU) device. Our proposed method extends a previous method by increasing the cache hit rate so as to speed up the reconstruction of high-resolution volumes that exceed the capacity of device memory. More specifically, our approach accelerates the well-known Feldkamp-Davis-Kress algorithm by utilizing the following three strategies: (1) a loop organization strategy that identifies the best tradeoff point between the cache hit rate and the number of off-chip memory accesses; (2) a data structure that exploits high locality within a layered texture; and (3) a fully pipelined strategy for hiding file input/output (I/O) time with GPU execution and data transfer times. We implement our proposed method on NVIDIA's latest Maxwell architecture and provide tuning guidelines for adjusting the execution parameters, which include the granularity and shape of thread blocks as well as the granularity of I/O data to be streamed through the pipeline, which maximizes reconstruction performance. Our experimental results show that it took less than three minutes to reconstruct a 2048^3-voxel volume from 1200 2048^2-pixel projection images on a single GPU; this translates to a speedup of approximately 1.47 as compared to the previous method.

key words: *cone beam reconstruction, GPU, CUDA, cache optimization*

1. Introduction

Cone beam computed tomography (CT) reconstruction is a radiology imaging technology that converts two-dimensional X-ray projection images generated by a rotary CT scanner into a three-dimensional volume, such that CT data can be viewed using three-dimensional visualization software. The Feldkamp-Davis-Kress (FDK) algorithm [1] is the de facto standard for cone beam CT reconstruction and is widely adopted in medical and industrial applications [2]–[4]. Because reconstruction time is critical, especially for real-time medical applications such as image-guided surgery [5], [6], research activities on accelerating the FDK algorithm have been ongoing ever since its advent in 1984.

With the development of parallel computing and computer graphics technologies, efforts to parallelize the FDK algorithm have included various computing devices, including the graphics processing unit (GPU) [7], [8], a cell broadband engine (CBE) [9], a field-programmable gate array (FPGA) [10], and a Xeon Phi coprocessor [11]. In particular, utilizing the compute unified device architecture (CUDA) compatible GPUs [12], to parallelize FDK computation has gained popularity due to its high-performance and low-cost implementation as compared to other devices [2], [13], [14].

In general, using implementations based on CUDA offloads the performance bottleneck of a CPU-based sequential code. Such offloaded workloads can be implemented as *kernel functions*, which can be parallelized via millions of GPU threads for acceleration. Given this parallelization technique, the performance bottleneck of the FDK algorithm lies in its back-projection of projection images in which interpolated pixel values are accumulated back to form voxel values, which then compose the three-dimensional volume. Therefore, typical implementations store projection images in *textures* to take advantage of hardware-accelerated interpolation available on the GPU.

In addition to this fundamental implementation scheme, Okitsu *et al.* [15] presented a multiplication method that back-projects multiple projections with a single kernel invocation. This multiplication method accelerates the back-projection procedure by reducing the number of off-chip memory accesses. In [15], Okitsu *et al.* concluded that device memory bandwidth (i.e., the memory bandwidth between streaming multiprocessors (SMs) [12] and off-chip memory) determines reconstruction throughput on a GeForce 8800 GTX GPU; however, cache optimization issues were not considered in detail because the deployed G80 architecture [16] did not have a sophisticated cache as compared to state-of-the-art GPU architectures.

Therefore, in this paper, we propose a cache-aware optimization method to accelerate the FDK algorithm for handling a large data on a GPU. Our proposed method extends Okitsu's method [15] by increasing the cache hit rate, thereby improving out-of-core cone beam reconstruction. Note that the term "out-of-core" here means that the input/output (I/O) data exceed the capacity of device memory. Our proposed method consists of the following three key strategies: (1) a loop organization strategy which identifies the best tradeoff point between the cache hit rate and the number of off-chip memory accesses; (2) a data structure that exploits high locality within a layered texture; and (3) a fully pipelined strategy for hiding file I/O times with GPU execution and data transfer times. We analyze the underlying mechanism of these strategies and provide tuning guidelines for adjusting the execution parameters, which include the granularity and shape of thread blocks [12] and the granularity of I/O data to be streamed through the pipeline, the

Manuscript received April 26, 2016.
Manuscript publicized September 5, 2016.
[†]The authors are with the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: ino@ist.osaka-u.ac.jp
DOI: 10.1587/transinf.2016EDP7174

Table 1 Comparison of the performance of our work with relevant recent works. Note that back-projection throughput ρ, which is measured in GUPS, is given by $\rho = NXYZ/T$, where N is the number of projections, $X \times Y \times Z$ is the volume size in voxels, and T is the back-projection time, which includes data transfer times between CPU and GPU. Further, U and V are the horizontal and vertical sizes of a projection, respectively. Finally, efficiency E is given by $E = 4\rho/B$, where B is the total memory bandwidth of the deployed machine; efficient cache utilization achieves an efficiency of more than 100%.

Work	Platform	Memory spec.		Data spec.		Performance	
		Bandwidth (GB/s)	Capacity (GB)	$NUV \to XYZ$	Size (GB)	ρ (GUPS)	E (%)
Scherl [14]	8800 GTX	86.4	0.8	$414 \times 1024 \times 1024 \to 512^3$	$1.6 \to 0.5$	6.2	29
Okitsu [15]	2×Tesla C870	76.8	1.5	$1024 \times 1024 \times 1024 \to 1024^3$	$4.0 \to 4.0$	48.9	127
Ino [17]	4×Tesla S1070	102.0	4.0	$2048 \times 2048 \times 2048 \to 2048^3$	$32.0 \to 32.0$	105.7	104
Noël [18]	GTX 280	141.7	1.0	$106 \times 1024 \times 1024 \to 512^3$	$0.4 \to 0.5$	2.3	6
Zheng [19]	GTX 480	177.4	1.5	$364 \times 1024 \times 768 \to 512^3$	$1.1 \to 0.5$	12.0	27
Zhang [20]	2×GTS 450	57.7	1.0	$360 \times 1024 \times 1024 \to 512^3$	$1.4 \to 0.5$	11.1	38
Treibig [21]	4×Xeon E7-4870	34.2	N/A	$496 \times 1248 \times 960 \to 1024^3$	$2.2 \to 4.0$	12.0	35
Papenhausen [22]	GTX 680	192.3	2.0	$496 \times 1248 \times 960 \to 512^3$	$2.2 \to 0.5$	72.3	150
Zinßer [23]	GTX 680	192.3	2.0	$496 \times 1248 \times 960 \to 1024^3$	$2.2 \to 4.0$	88.2	183
Blas [24]	2×GTX 680	192.3	2.0	$720 \times 1024 \times 1024 \to 1024^3$	$2.8 \to 4.0$	117.5	122
Serrano [25]	2×GTX 680	192.3	2.0	$360 \times 512 \times 512 \to 512^3$	$0.4 \to 0.5$	27.1	28
	2×Xeon Phi 7120P	352.0	16.0	$360 \times 512 \times 512 \to 512^3$	$0.4 \to 0.5$	26.7	15
This work	GTX 980	224.0	4.0	$1200 \times 512 \times 512 \to 512^3$	$1.2 \to 0.5$	116.7	208
				$1200 \times 1024 \times 1024 \to 1024^3$	$4.7 \to 4.0$	128.5	229
				$1200 \times 2048 \times 2048 \to 2048^3$	$18.8 \to 32.0$	92.9	166

latter maximizing reconstruction performance on NVIDIA's latest Maxwell architecture [26].

The rest of this paper is organized as follows. Section 2 introduces related studies regarding the acceleration of cone beam CT reconstruction. Section 3 summarizes the FDK algorithm and its previous GPU-based implementation [15]. Section 4 describes our proposed method, and then Sect. 5 presents our experimental results along with a discussion on tuning for the Maxwell architecture. Finally, Sect. 6 concludes our paper with future work.

2. Related Work

Table 1 shows a comparison of our present work with recent studies, showing the advantages of our proposed method. In the table, the back-projection throughput ρ is presented in giga voxel updates per second (GUPS), where giga denotes 10^9. In summary, our present work employs a single-GPU machine to achieve high back-projection throughput for large amounts of data that exceed not only device memory but also host memory. Further, we incorporate an optimization method based on NVIDIA's latest Maxwell architecture [26].

To our knowledge, Scherl *et al.* [14] first proposed the use of a CUDA-based GPU to accelerate the FDK algorithm, claiming that reducing register file usage raised GPU occupancy [12] so as to accelerate reconstruction. They demonstrated that a GeForce 8800 GTX GPU achieved two times higher reconstruction performance as compared to a CBE. A similar CUDA-based approach with similar results was presented by Noël *et al.* [18].

As for kernel optimization, Okitsu *et al.* [15] extended Xu's method [7], who first proposed back-projecting multiple projections in a single kernel invocation. These multiplication schemes reduced the number of off-chip memory accesses and that of kernel invocations. They concluded that device memory bandwidth determines reconstruction performance; thus, the back-projection kernel should process more projections at a time. A similar scheme was presented by Papenhausen *et al.* [22], who processed 64 projections with a single kernel execution. In contrast to the above studies, we show that excessive projections result in a lower texture cache hit rate on the latest Maxwell architecture [26]. Consequently, it is important to find the best tradeoff point between texture cache hit rate and the number of off-chip memory accesses.

Based on Okitsu's multiplication method [15], Zinßer *et al.* [23] modified the nested loop structure proposed in [15] such that threads that share the same instruction can simultaneously access a single projection. Zinßer *et al.* claimed that their loop organization not only increased the texture cache hit rate but also reduced the number of off-chip memory accesses by processing 32 projections with a single kernel invocation. A key drawback of their loop organization is that it consumes more registers than the original organization. Consequently, only four *xy*-slices of the volume were produced by a kernel invocation, whereas the original organization produced 512 *xy*-slices at a time. This consumption issue must be resolved for large amounts of data, which we focus on in the present study, because 128 times more kernel invocations are required to produce the entire volume. In our work, we present a data structure capable of achieving an L1/texture cache hit rate of more than 95%, even with the original loop organization.

In contrast to the input-related optimization mentioned above, Zheng *et al.* [19] presented a cache-aware method capable of maximizing write throughput for the output volume. Their method rearranges volume data according to the back-projection angle such that a series of memory transactions can be coalesced into a single transaction. Since this data rearrangement incurs overhead, they allocated another copy of the volume to avoid rearrangement overhead;

however, such duplicated data must be eliminated to handle large amounts of data on limited device memory. Our method realizes memory access coalescence by adopting a workload distribution scheme in which threads are responsible for angle-independent regions of the volume.

With respect to out-of-core reconstruction in which I/O data flows exceed device memory, several studies have explored a multi-GPU machine to achieve further acceleration [17], [20], [24]. Existing multi-GPU implementations adopt a pipelined approach to overlap kernel execution with data transfer between CPU and GPU; however, except for Blas *et al.* [24], file I/O overhead has not been considered in detail. Blas *et al.* [24] did indeed consider file I/O overhead, thus realizing on-the-fly reconstruction, which produces the volume immediately after image acquisition; however, cache optimization issues were not addressed. Our out-of-core pipelined strategy yields a fully pipelined cache-aware solution for processing large amounts of data on a single-GPU system. Further, although we evaluated the advantages of our method on a single-GPU machine, our method can be expanded to support a multi-GPU environment in a straightforward manner.

On the other hand, Serrano *et al.* [25] proposed using a directive-based programming approach [27] to parallelize the FDK algorithm on GPUs and Intel Xeon Phi coprocessors. Compared with CUDA, this directive-based approach provides an easy programming scheme in which parallelization is achieved by adding compiler directives to sequential code; however, using such a high-level programming style degrades the performance. Due to the same performance related reason, we prefer CUDA rather than OpenCL [28].

Finally, Treibig *et al.* [21] explored optimizing a single-instruction multiple data (SIMD) instruction set called Advanced Vector eXtensions (AVX) [29]. They indicated that GPU-based solutions degrade reconstruction throughput for large amounts of data, because limited device memory requires data transfers between CPU and GPU. Our pipelined solution overlaps these required data transfers with GPU computation, thereby achieving higher out-of-core reconstruction performance as compared to CPU-based approaches (Table 1).

3. Preliminaries

Let F be the three-dimensional volume of size $X \times Y \times Z$ to be reconstructed. To estimate each voxel value $F(x, y, z)$ in F, where $0 \leq x < X$, $0 \leq y < Y$, and $0 \leq z < Z$, the FDK algorithm back-projects set $\mathcal{P}$ of two-dimensional projection images onto the target volume F, where $\mathcal{P} = \{P_0, P_1, \ldots, P_{N-1}\}$ and N is the number of projection images. As shown in Fig. 1, P_n represents a projection obtained with rotational angle θ_n, where $0 \leq n < N$. In the following subsections, we assume that all projection images comprise $U \times V$ pixels.

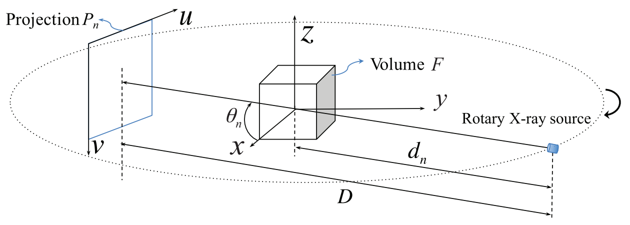

Fig. 1 Geometry for back-projection of the n-th filtered projection, where $0 \leq n < N$.

3.1 FDK Reconstruction Algorithm

The FDK algorithm is composed of two processing stages, i.e., ramp filtering and back-projection. The ramp filtering stage performs one-dimensional convolution along the horizontal direction (i.e., the u-axis). Given raw projection P_n, where $0 \leq n < N$, the pixel value $Q_n(u, v)$ of filtered projection Q_n, where $0 \leq u < U$ and $0 \leq v < V$, is given by

$$Q_n(u, v) = \sum_{r=-R}^{R} \frac{2}{\pi^2(1-4r^2)} \frac{D}{\sqrt{D^2+r^2+v^2}} P_n(r, v), \quad (1)$$

where R denotes the ramp filter radius and D denotes the distance between the X-ray source and the projection panel, as shown in Fig. 1.

Next, the back-projection stage back-projects filtered projections $Q_0, Q_1, \ldots, Q_{N-1}$ into three-dimensional volume F. Voxel value $F(x, y, z)$ at point (x, y, z) is given by

$$F(x, y, z) = \frac{1}{2\pi N} \sum_{n=0}^{N-1} W(x, y, n) \, Q_n(u(x, y, n), v(x, y, z, n)), \quad (2)$$

where weight $W(x, y, n)$ and coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are calculated separately by

$$W(x, y, n) = \left(\frac{d_n}{d_n - x\cos\theta_n - y\sin\theta_n}\right)^2, \quad (3)$$

$$u(x, y, n) = \frac{D(-x\sin\theta_n + y\cos\theta_n)}{d_n - x\cos\theta_n - y\sin\theta_n}, \quad (4)$$

$$v(x, y, z, n) = \frac{Dz}{d_n - x\cos\theta_n - y\sin\theta_n}, \quad (5)$$

where d_n denotes the distance between the X-ray source and the origin of the xyz coordinates.

According to Eq. (2), the time complexity of the back-projection stage is $O(NXYZ)$, which represents the performance bottleneck of the FDK algorithm. In particular, three-dimensional data accesses required to read $Q_n(u(x, y, n), v(x, y, z, n))$ and write $F(x, y, z)$ determine total execution time; as such, the back-projection stage constitutes more than half of the total execution time [15], [19]. Note that Eq. (5) can be efficiently computed via the following recurrence relation:

$$v(x, y, z + 1, n) = v(x, y, z, n) + v(x, y, 1, n). \quad (6)$$

Because the second term $v(x, y, 1, n)$ can be precomputed for all n, this relation is useful for reducing computational cost; more specifically, only an addition is needed to compute $v(x, y, z, n)$ within the z loop.

3.2 Maxwell GPU Architecture

Figure 2 provides an overview of the Maxwell GPU architecture [26]. Similar to other CUDA-compatible GPUs, this architecture has an array of SMs to process millions of tasks in parallel. Each SM has 128 CUDA cores with on-chip memory, including registers, a shared memory, and an L1/texture cache. Registers provide the shortest access latency, analogous to CPU registers, but different in number; here there are 64K 32-bit registers. The shared memory is a software-managed cache that allows CUDA cores to more efficiently share data inside an SM. Finally, the L1/texture cache acts as a read-only cache and coalescing buffer for off-chip memory access [30].

Outside of the SMs, there is an off-chip memory called device memory. Although device memory provides a large storage of up to 12 GB, its latency of several hundreds of clock cycles is much longer than that of the on-chip memory. Device memory can be used as both texture and global memory. Here, texture memory stores read-only data that can be accessed using hardware interpolation, while global memory stores readable/writable data for CUDA cores. As shown in the figure, the L2 cache is an on-chip cache located between the SMs and global memory. Note that the L1/texture cache is bypassed to access data stored in global memory [30].

During kernel execution, threads are cyclically assigned to SMs in the unit of a thread block [12], i.e., a group of threads organized by CUDA programmers. Different thread blocks must be independent of each other with respect to data dependencies; otherwise parallelization cannot be correctly achieved. Resident thread blocks that have been assigned consume SM resources, including registers and shared memory, such that there are limitations on the maximum number of resident threads and that of thread blocks. This assignment process is repeated until all thread blocks finish execution.

Each resident thread block is further divided into groups of 32 consecutive threads, called a *warp*. Threads in a warp, which share the same instruction at each clock cycle, are processed on an SM in parallel, which is called single-instruction multiple-thread (SIMT) [12] in vendor terminology. While threads in the same warp access global memory, memory access coalescence is critical for maximizing effective memory bandwidth [12]. In most cases, assigning multiple thread blocks to an SM is an effective approach for overlapping memory accesses with computation, because the SM has more data-independent resident warps to switch while waiting for data to be fetched from off-chip memory. Thus, if we maximize the occupancy, or ratio of the number of resident thread blocks to the maximum number of resident thread blocks, we can hide memory access latencies with computation. In other words, fewer resident thread blocks expose memory latency.

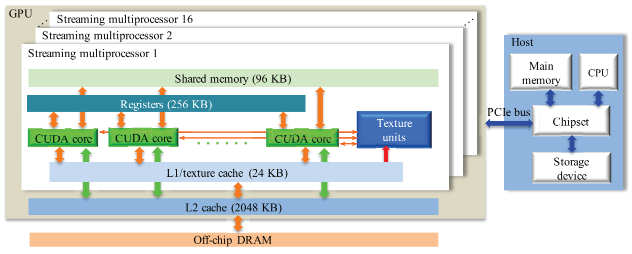

Fig. 2 Maxwell GPU architecture [26]. A GPU consists of an array of SMs, each including hundreds of CUDA cores, an L1/texture cache, and registers. In addition to this on-chip memory, an off-chip memory is available on the graphics board.

3.3 GPU Implementation of the FDK Algorithm

As presented in Table 1, Okitsu's method [15] was one of the most efficient comparative methods in terms of memory bandwidth efficiency. Figure 3 (a) illustrates the reconstruction pipeline realized in their implementation. This pipeline has four major stages, i.e., projection input, ramp filtering, back-projection, and volume output.

In the projection input stage, raw projections P_0, $P_1, \ldots, P_{N-1}$ are loaded from a storage device to host memory (i.e., main memory), and then sequentially transferred to device memory. Next, the ramp filtering stage filters each projection P_n into Q_n, where $0 \leq n < N$, and transfers these filtered projections back to host memory. The back-projection stage then handles both filtered projections and volume via a divide-and-conquer approach to

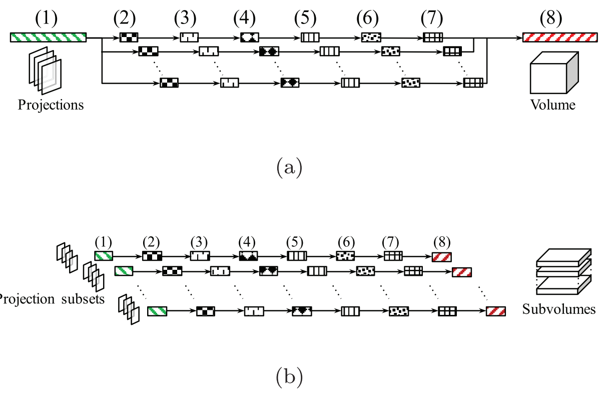

Fig. 3 Reconstruction pipelines of (a) Okitsu's previous method [15] and (b) the proposed method. Both pipelines consist of the following eight steps, with the first and last steps not pipelined in the previous method: (1) raw projections are loaded from a storage device into host memory, (2) projections in host memory are then transferred to device memory; (3) ramp filtering is applied to produce filtered projections, (4) filtered projections are transferred back to host memory if necessary, (5) filtered projections are transferred from host memory to device memory if necessary; (6) back-projection is performed to produce a subvolume; (7) the subvolume is transferred to host memory, and (8) the subvolume in host memory is written to the storage device.

overcome limited device memory, as illustrated in Fig. 4. From the figure, volume F is partitioned along the z-axis into subvolumes $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{Z/Z'-1}$, each with Z' xy-slices, whereas the filtered projections are partitioned into subsets $Q_0, Q_1, \ldots, Q_{N/N'-1}$ with N' projections each, thus indicating that the back-projection kernel is launched N/N' times for each subvolume $\mathcal{F}_k$, where $0 \le k < Z/Z' - 1$. Finally, the produced subvolumes are transferred back to host memory, and then written to the storage device in the volume output stage.

Algorithm 1 shows the pseudocode of the back-projection kernel, which generates subvolume F_k from subset Q_m of projections, where $0 \le k < Z/Z' - 1$ and $0 \le m < N/N' - 1$. This kernel assumes that each thread block is in charge of reconstructing a slab along the z-axis, where a thread with global index (x, y) computes $F(x, y, z)$ for all $kZ' \le z < (k + 1)Z'$ (i.e., Z' voxels along the z-axis); this volume is stored in writable global memory, whereas projections are stored in textures to take advantage of hardware-accelerated interpolation. Note that this workload distribution scheme always achieves memory access coalescing when writing voxel values, because threads in the same warp access consecutive voxels on the same xy-plane, i.e., threads are responsible for an angle-independent region of the volume. Similar to [22], atomic instructions are deployed to maximize the collision-free write throughput of the volume on line 14 of Algorithm 1.

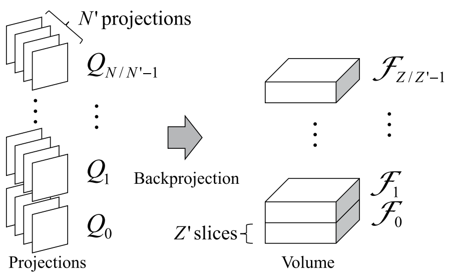

Fig. 4 Data decomposition scheme in which the back-projection kernel is invoked for each pair $\langle Q_m, \mathcal{F}_k \rangle$ of subset Q_m of projections and subvolume $\mathcal{F}_k$, where $0 \le m < N/N' - 1$ and $0 \le k < Z/Z' - 1$.

Algorithm 1: Back-projection kernel

Input : subset Q_m of N' filtered projections, projection subset index m, and subvolume index k, where $0 \le m < N/N' - 1$ and $0 \le k < Z/Z' - 1$
Output: subvolume $\mathcal{F}_k = \{F_k(x, y, z) \mid 0 \le x < X,\ 0 \le y < Y,\ kZ' \le z < (k + 1)Z'\}$ of Z' slices

```
1  calculate responsible voxel coordinate $(x, y)$ from thread index
   and thread block index;
2  $z \leftarrow k \times Z'$ ;                // first z coordinate in $\mathcal{F}_k$
3  $n \leftarrow m \times N'$ ;           // first projection index in $Q_m$
4  for $i \leftarrow 0$ to $N' - 1$ do              // for each projection
5      $w_i \leftarrow W(x, y, n + i)$; $u_i \leftarrow u(x, y, n + i)$;
       $v_i \leftarrow v(x, y, z, n + i)$ ;                  // Eqs. (3)-(5)
6  end
7  for $j \leftarrow 0$ to $Z' - 1$ do                // for each z-slice
8      $t \leftarrow 0$;
9      for $i \leftarrow 0$ to $N' - 1$ do          // for each projection
10         $v_i \leftarrow v(x, y, z + j, n + i)$ ;               // Eq. (6)
11         $r \leftarrow Q_{n+i}(u_i, v_i)$ ;      // texture memory access
12         $t \leftarrow t + w_i \times r$ ;                      // Eq. (2)
13     end
14     $F_k(x, y, z + j) \leftarrow F_k(x, y, z + j) + t$ ;   // atomic write
       to global memory
15 end
```

4. Proposed Method

The main approach to optimizing GPU performance is to locate the performance bottleneck, or resource constraint, and then attempt to exchange it for the use of another resource. Applying this to the FDK algorithm, lines 11 and 14 of Algorithm 1 determine the performance of the back-projection kernel, because an arithmetic instruction generally takes several clock cycles, whereas an off-chip memory access takes hundreds of clock cycles. Therefore, reducing or hiding memory access latency is pivotal to maximizing the reconstruction performance on a GPU. Our solution here is twofold: (1) maximize cache utilization to reduce memory access latency on line 11 of the algorithm; and (2) back-project multiple projections to reduce the number of off-chip memory access on line 14 [15]. Therefore, in the subsections below, we present the following three strategies to systematically optimize the reconstruction procedure:

1. Cache-aware loop organization, which identifies the best tradeoff point between the cache hit rate and the number of off-chip memory accesses (Sect. 4.1)
2. A cache-aware data structure with a layered texture (Sect. 4.2)
3. A pipelined strategy that includes I/O (Sect. 4.3)

4.1 Cache-Aware Loop Organization

As shown in Fig. 5, Eqs. (4) and (5) indicate that coordinates $(u(x, y, n), v(x, y, z, n))$ are similar in that they are calculated

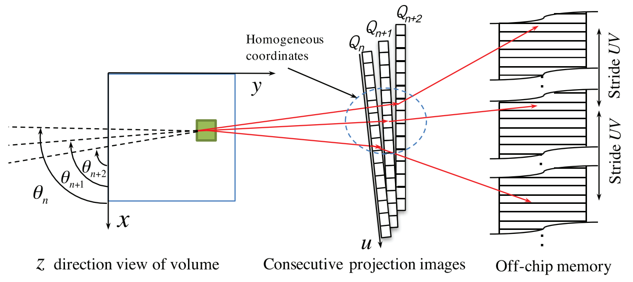

Fig. 5 Schematic illustrating texture access. In this example, a thread block is responsible for producing the enclosed region in the volume. Given filtered projections Q_n, Q_{n+1}, and Q_{n+2}, warps in the given thread block access homogeneous texture coordinates (u_n, v_n), (u_{n+1}, v_{n+1}) and (u_{n+2}, v_{n+2}) on the filtered projections, respectively (see Algorithm 1). This locality of two-dimensional coordinates cannot extend to the locality of off-chip memory because successive projections are stored with a stride of UV, where U and V are the horizontal and vertical resolutions of projections, respectively.

```
for $l \leftarrow 0$ to $L-1$ do {for each layer}
  for $v \leftarrow 0$ to $V-1$ do
    for $u \leftarrow 0$ to $U-1$ do
      fetch texel at coordinate $(u,v)$ of layer $l$;
    end for
  end for
end for
```

(a)

```
for $v \leftarrow 0$ to $V-1$ do
  for $u \leftarrow 0$ to $U-1$ do
    for $l \leftarrow 0$ to $L-1$ do {for each layer}
      fetch texel at coordinate $(u,v)$ of layer $l$;
    end for
  end for
end for
```

(b)

Fig. 6 Micro-benchmarks for evaluating texture access performance. Here, two access patterns were examined with $U = V = 1024$ and $1 \leq L \leq 64$; the access patterns are (a) intra-layer-first and (b) inter-layer-first patterns. A single thread was used to fetch all texels. The innermost loop of both patterns was unrolled for optimization.

from consecutive projection angles. In other words, a series of pixels (u_n, v_n), (u_{n+1}, v_{n+1}), $\ldots, (u_{n+N'-1}, v_{n+N'-1})$ are intensively fetched from a small area of projections on line 11 of Algorithm 1; however, this high locality of coordinates may not be extended to that of texture pixels (i.e., texels) because successive projections are stored with a stride of UV (Fig. 5). To handle this issue, Zinßer *et al.* [23] reversed the ji-loop of Algorithm 1 to form an ij-loop and synchronized threads before proceeding to the next projection, j; further, they changed the inter-projection first loop to an intra-projection first loop. As mentioned in Sect. 2, this method improved the texture cache hit rate by creating threads to simultaneously access the same projection within the inner j-loop, but threads consume more registers to store their responsible slabs, i.e., voxel values along the z-axis. In more detail, variable t in Algorithm 1 must be extended as variables $t_0, t_1, \ldots, t_{Z'-1}$ because they cannot be flushed within the inner j-loop. Without this extension, the amount of global memory access cannot be minimized. In this study, we therefore adopt the original loop organization of the previous method [15] and present a data structure that is more tolerant to the intra-projection first loop: three-dimensional access spreading over different projections.

As mentioned above, the previous method [15] partitions input projections into N/N' subsets to back-project each subset with a single kernel invocation. This multiplication method reduces both the number of kernel invocations and the number of global memory writes by a factor of N', because N' successive kernel executions are packed into a single execution. More specifically, the previous method [15] improves reconstruction performance by maximizing N', which leads to efficient use of registers; however, increasing N' consumes more registers for variables w_i, u_i and v_i, which decreases the occupancy on the SM.

Large N' also implies that threads can simultaneously fetch pixels from different projections because they are executed in an SIMT manner in which different warps are allowed to simultaneously process different lines in the kernel. Therefore, excessive N' decreases the L1/texture cache hit rate, particularly for high-resolution images, because stride UV between successive projections increases with image resolution $U \times V$. Note that synchronization is useful to enforce warps keeping pace with other warps, which prevents warps from accessing different projections; however, CUDA prohibits inter-block synchronization during kernel execution; thus, synchronization is not a perfect solution for this issue.

In addition to the loop organization described above, our method optimizes cache behavior by choosing the best granularity and shape of thread blocks according to the characteristics of memory access patterns. Because CUDA organizes threads in a three-level hierarchy composed of elementary threads, warps, and thread blocks, cache optimization can be achieved at each of these three levels accordingly. Sugimoto *et al.* [31] concluded that, with respect to volume rendering applications, the most important level is the warp level, because memory access transactions are issued on a per-warp basis. Similarly, Okitsu *et al.* [15] concluded that square-shaped thread blocks are suitable for the back-projection kernel because warps are organized such that back-projection performance is averaged for arbitrary rotational angles. Refer to [15] for details.

4.2 Cache-Aware Data Structure with Layered Texture

Our proposed data structure is realized via a layered texture [12], which packs equally-sized textures of the same type into a single object. Texels in a layered texture can be accessed using floating-point coordinate (x, y) and a layer specified by integer index l. Intra-layer interpolation can be applied to the xy-plane, but inter-layer interpolation is not available. Layered textures are ideal for processing multiple textures of the same size and format in that they reduce the overhead of texture access. Because its three-dimensional locality has not been explicitly stated [12], we design a suite of micro-benchmarks to analyze its performance advantages with the memory access patterns of the FDK algorithm.

Figure 6 shows the pseudocode for the micro-benchmarks, which run a single GPU thread to determine whether a layered texture has been optimized for three-dimensional locality; here, two access patterns are investigated to compare the performance of a layered texture and non-layered (i.e., naive two-dimensional) textures. The first micro-benchmark, i.e., Fig. 6 (a), examines an intra-layer-first pattern in which a thread finishes fetching all texels from the current layer before accessing the next layer. Conversely, the other micro-benchmark, i.e., Fig. 6 (b), examines an inter-layer-first pattern in which a thread fetches tex-

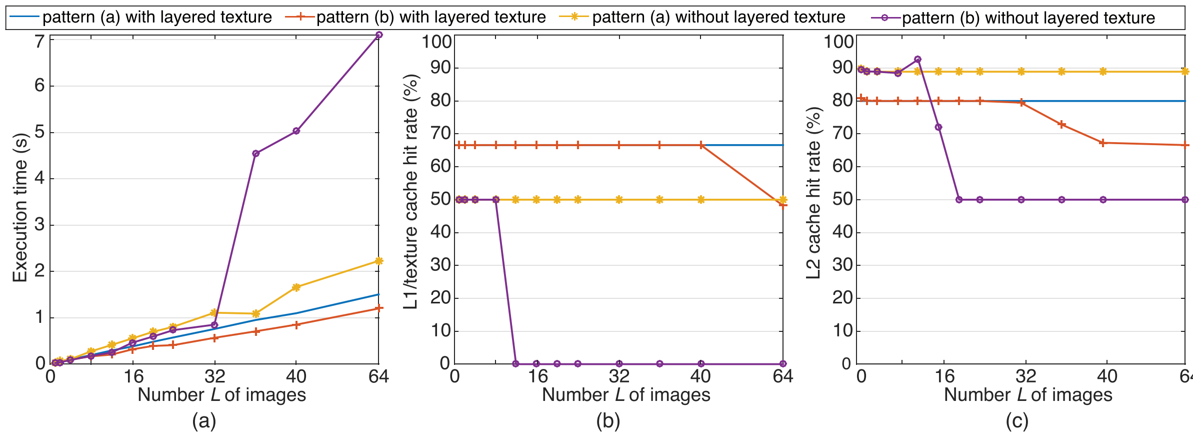

Fig. 7 Benchmark and profiling results for different loop organizations and data structures, showing (a) execution time, (b) L1/texture cache hit rate, and (c) L2 cache hit rate. Here, intra-layer-first and inter-layer-first patterns were investigated with a layered texture and non-layered (i.e., naive two-dimensional) textures.

els from all layers at the same coordinate (u, v) before going to the next coordinate.

Figure 7 shows the timing and profiling results obtained using the micro-benchmarks. As shown in the figure, the inter-layer-first pattern on a layered texture achieved the best performance with a higher L1/texture cache hit rate. Consequently, we conclude that layered textures have three-dimensional locality and are thereby optimized for three-dimensional texture access. This performance characteristic agrees with the memory access pattern of the back-projection kernel, which is an inter-layer-first pattern as shown on lines 7–15 of Algorithm 1.

According to our benchmark results, we decided to use a layered texture with the inter-layer-first loop. We implemented the layered texture with the texture object application programming interface (API) introduced in the Kepler architecture [32]. Compared to the legacy texture reference API, the texture object API simplifies the resulting programming style and eliminates several restrictions [12]. For example, the texture object API does not require manual binding and unbinding of texture references to memory addresses. Therefore, texture references can be used in a more dynamic manner, whereas the texture reference API requires texture references to be declared as static global variables.

4.3 Pipelined Strategy That Includes I/O

Similar to Blas *et al.* [24], our out-of-core pipelined strategy decomposes not only computation steps but also file I/O steps, namely steps (1) and (8) in Fig. 3 (b), thus overlapping them with other steps. Such an I/O-included pipelining strategy is extremely important for large amounts of data that exceed not only device memory but also host memory. Without this strategy, the entire reconstruction throughput is bounded by file I/O time, even though the back-projection procedure is significantly accelerated on the GPU.

Algorithm 2 presents pseudocode for our pipelined FDK algorithm. As with the previous method [15], this algorithm divides the input projections and output volume into

Algorithm 2: Fully pipelined FDK reconstruction

```
Input  : set $\mathcal{P} = \{P_0, P_1, \ldots, P_{N-1}\}$ of raw projections
Output: volume $F = \bigcup_{k=0}^{Z/Z'-1} \mathcal{F}_k$
1  for $k \leftarrow 0$ to $Z/Z' - 1$ do in parallel // for each subvolume
2      for $j \leftarrow 0$ to $N/N' - 1$ do   // for each projection subset
3          if $k == 0$ then
4              for $i \leftarrow 0$ to $N' - 1$ do in parallel // for each projection in projection subset
5                  load raw projection $P_{N'j+i}$ from the storage device;
6                  transfer $P_{N'j+i}$ from host to device asynchronously;
7                  $Q_{N'j+i} \leftarrow$ RampFilteringKernel($P_{N'j+i}$) ; // Eq. (1)
8                  transfer $Q_{N'j+i}$ from device to host asynchronously;
9              end
10             set $Q_j = \{Q_{N'j}, Q_{N'j+1}, \ldots, Q_{N'j+N'-1}\}$;
11         else
12             transfer $Q_j = \{Q_{N'j}, Q_{N'j+1}, \ldots, Q_{N'j+N'-1}\}$ from host to device asynchronously ;
               // $Q_j$ is double buffered on device
13         end
14         $\mathcal{F}_k \leftarrow$ back-projectionKernel($Q_j, j, k$) ; // Algorithm 1
15     end
16     transfer $\mathcal{F}_k$ from device to host ;    // $\mathcal{F}_k$ is double buffered on host
17     store $\mathcal{F}_k$ to the storage device asynchronously;
18 end
```

N/N' subsets and Z/Z' subvolumes, respectively. The filtering and back-projection procedures are then carried out for each pair $\langle Q_m, \mathcal{F}_k \rangle$ of projection subset Q_m and subvolume $\mathcal{F}_k$, where $0 \le m < N/N' - 1$ and $0 \le k < Z/Z' - 1$. Given the limited capacity of device memory, filtered projections are pushed back to host memory to save device memory consumption for the subvolume to be generated (i.e., line 8 of Algorithm 2). Further, filtered projections are reused to al-

Table 2 Specifications of our experimental machine.

Item	Specification
CPU	Intel Core i7-4770K
Main memory capacity	32 GB
GPU	NVIDIA GeForce GTX 980
Clock speed	1126 MHz (base), 1216 MHz (boost)
Texture fill rate	144.1 Gtexel/s (base), 155.6 Gtexel/s (boost)
Device memory capacity	4 GB
Device memory bandwidth	224.3 GB/s
Bus interface	PCIe 3.0 ×16 bus
Storage	Samsung 850 EVO SSD 500 GB
OS	Fedora 22 64-bit
Compiler	CUDA 7.5 and gcc 5.11
Compiler option	`-O3 -arch=compute_52 -code=compute_52 -Xcompiler -fopenmp -lgomp`
Driver	352.55

Fig. 8 Shepp-Logan phantom [34] reconstructed by our experimental machine.

low us to skip the filtering step for succeeding subvolumes (i.e., line 12). Note that the buffers to be used for the back-projection kernel are doubled to enable overlaps with other steps (i.e., lines 12 and 16). Without these double-buffers, overlaps cannot be achieved due to the data dependence that exists between succeeding steps.

5. Experimental Results

We compared our proposed method with the previous method [15] in terms of reconstruction time. All timing results were measured using the NVIDIA Visual Profiler [33]. Table 2 lists the specifications of our experimental machine.

In our experiments, we used three sets of the Shepp-Logan phantom [34] at different resolutions: a small dataset with $U = V = X = Y = Z = 512$, a medium dataset with $U = V = X = Y = Z = 1024$, and a large dataset with $U = V = X = Y = Z = 2048$, each with $N = 1200$ projections. The middle slice of the reconstructed volume is shown in Fig. 8. Pixels in the projections and voxels in the volumes consist of four bytes; thus, the small, medium, and large datasets consumed 1.7 GB, 8.7 GB, and 50.8 GB of memory, respectively. Therefore, the medium and large datasets could not be entirely stored in host memory or device memory (Table 2).

5.1 Parameter Configuration

We conducted preliminary experiments to identify execution parameters that achieve the highest reconstruction performance; these parameters included (1) the granularity and shape of thread blocks, (2) projection subset size N', and (3) subvolume size Z'.

The appropriate shape of the thread blocks was firstly investigated for the back-projection kernel. Figure 9 shows the back-projection times, L1/texture cache hit rates, and L2 cache hit rates for all rotational angles with different thread block shapes. As expected (see Sect. 4.1), square blocks of 16×16 threads achieved the highest performance with approximately 95% L1/texture cache hit rates. In contrast, the lowest performance was obtained with a shape of 256×1 threads, which yielded a 50% L1/texture cache hit rate, observed around rotational angles of 90 and 270 degrees. With these rotational angles, resident warps in such non-squared thread blocks accessed wide rows of projections, thereby dropping the L1/texture cache hit rate.

Next, as shown in Fig. 10, we investigated back-projection performance with different projection subset sizes N' and thread block sizes. As shown in Fig. 10 (a), setting $N' = 20$ and using blocks of 256 threads yielded the best performance for the medium dataset. As for projection subset size N', Figs. 10 (b) and 10 (c) show that there was a tradeoff between the L1/texture cache hit rate and the number of global memory accesses, as stated in Sect. 3.3. The number of global memory accesses here is given by $8XYZN/N'$ in bytes, because four-byte voxels are loaded and stored once per kernel invocation.

Conversely, the previous method [15] improved reconstruction performance by maximizing N' on the G80 architecture [16]; thus, this previous idea must to be adapted to the new Maxwell architecture accordingly. In other words, the previous results partially agreed with our results when $N' < 20$, where the number of global memory accesses decreased with N', but back-projection time increased slightly with N' when $N' > 20$. As shown in Fig. 10 (b), excessive N' increased memory access strides and degraded L1/texture hit rates, which outweighed the performance gain contributed by fewer kernel invocations and fewer write accesses.

As noted earlier, global memory access bypasses the L1 caches; therefore, the L1/texture cache hit rate primarily corresponds to texture memory performance (i.e., the read throughput of the back-projection kernel). In summary, an appropriate N' can be selected according to the tradeoff mentioned above. Similarly, we found that $N' = 20$ and $N' = 16$ were the best sizes (i.e., the tradeoff point) for the

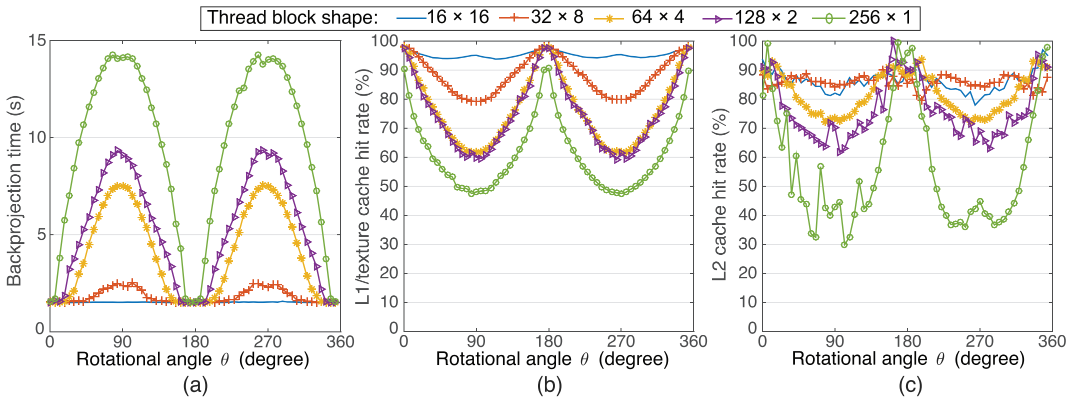

Fig. 9 Back-projection performance and profiling results with different shapes of thread blocks and rotational angles: (a) back-projection time, (b) L1/texture cache hit rate, and (c) L2 cache hit rate. These results were obtained with a thread block size of 256 for the medium dataset, with $N' = 20$ and $Z' = 512$.

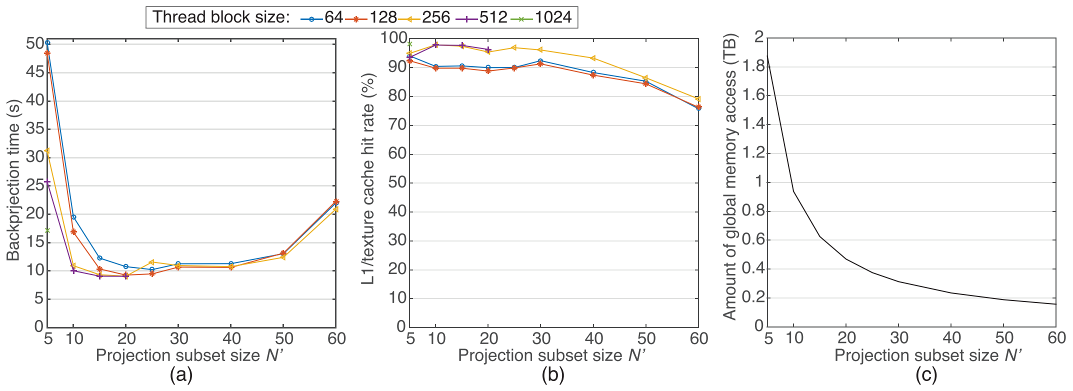

Fig. 10 Back-projection performance and profiling results with different projection subset sizes N' and thread block sizes: (a) back-projection times, (b) L1/texture cache hit rates, and (c) number of global memory accesses. These results were obtained with a medium dataset and $Z' = 512$.

small and large datasets, respectively.

Next, we discuss thread block size. As we decrease the thread block size, more thread blocks can be dispatched to each SM, which leads to higher occupancy; however, a thread block size of 64 was too small for assigning a square region to the SMs. In this case, we found that eight thread blocks were resident on each SM. Although each thread block was responsible for a square 8×8 region, these eight squares appeared in a rectangular region because of the cyclic assignment described in Sect. 3.2. Such a rectangular region cannot be efficiently back-projected from an unfavorable angle, as we presented in Fig. 9, for the shape of thread blocks. In contrast, the number of resident thread blocks decreases as we increase the thread block size; however, a thread block size of 512 was too large to maximize the projection subset size N', because such a large thread block consumes more registers. Execution for $N' > 8$ failed when using a thread block size of 1024. In summary, our solution is to maximize the thread block size such that its shape is kept as a square (i.e., a block of 16×16 threads).

Finally, we minimized subvolume size Z' such that (1) the off-chip memory could hold both a subvolume and a projection subset and (2) at least two subvolumes were generated for overlapping file I/O time ($Z/Z' \geq 2$). Note that for all subvolumes, filtered projections must be transferred to device memory. Consequently, the amount of data transfer between CPU and GPU increases with Z/Z'. According to this guideline, we used $Z' = 256$, 512, and 128 for the small, medium, and large datasets, respectively.

5.2 Breakdown Analysis

Using the best parameters identified above, we investigated the impact of our data structure and that of our I/O-included pipeline using different data sizes; results are summarized in Table 3. Here, the layered texture efficiently improved back-projection performance for the small and medium datasets, achieving speedups of at least a factor of 1.44 over the given baseline; however, speedup decreases to only a factor of 1.14 for the large dataset, which fetches texels from projections that are four times as large. In this case, data size $4UV$ of a projection reaches 16 MB, which immediately exceeds the L1/texture and L2 caches. For such large datasets, our pipeline increased speedup from a factor of 1.14 to a factor of 1.47 by realizing overlapped file I/O steps (1) and (8), which consumed 31% of the overall time before such

Table 3 Comparing the performance of our proposed method and previous method using different data sizes. The baseline corresponds to Okitsu's method [15] with the best parameters tuned for the Maxwell architecture. Further, "Both strategies" corresponds to our proposed method.

Data size	Step	Baseline [15]	Layered texture		I/O pipeline		Both strategies	
		Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Small	(6) Backproj.	1.7	1.1	1.55	1.7	1.00	1.1	1.55
	Total	5.4	4.6	1.17	4.3	1.26	3.9	1.38
Medium	(6) Backproj.	13.1	9.1	1.44	13.1	1.00	9.1	1.44
	Total	31.3	25.9	1.20	25.0	1.25	21.8	1.43
Large	(6) Backproj.	112.9	98.4	1.14	113.0	1.00	98.3	1.14
	Total	234.3	211.8	1.11	166.1	1.41	159.6	1.47

Table 4 Breakdown analysis of execution times for the large dataset. Here, "No pipeline" means that all steps were processed sequentially with synchronous APIs, whereas the "Previous pipeline" and "Proposed pipeline" were processed asynchronously. These results were obtained with the large dataset, with $N' = 16$ and $Z' = 256$.

Step	No pipeline	Previous pipeline [15]	Proposed pipeline
(1) T_1: Storage → host	36.9	—	—
(2) T_2: Host → device	3.2	—	—
(3) T_3: Ramp filtering	16.3	—	—
(4) T_4: Device → host	3.2	—	—
(5) T_5: Host → device	55.1	—	—
(6) T_6: Back-projection	98.4	—	—
(7) T_7: Device → host	5.4	—	—
(8) T_8: Host → storage	46.5	—	—
(1)–(4)	59.6	58.1	38.5
(5)–(8)	205.4	153.7	121.1
Total	265.5	211.8	159.6

overlapping was achieved. Thus, our I/O-included pipeline complements cache-aware back-projection, thereby demonstrating large speedups for both small and large datasets.

Next, we investigated the breakdown of reconstruction time for the large dataset, with our results summarized in Table 4. We evaluated the impact of our pipelined strategy; thus, all comparative methods used the same cache-aware kernel during measurements. In addition, a non-pipelined version deployed synchronous APIs, so that the sum of the breakdowns never equaled the execution times of the pipelined versions, which deployed asynchronous APIs. The previous method reduced the execution time from 265.5 s to 211.8 s, and our proposed method further reduced the execution time to 159.6 s, achieving speedups of 1.66 and 1.33 times the non-pipelined method and the previous method, respectively. Thus, pipelining must be applied not only to the filtering and back-projection steps (2)–(7) but also to file I/O steps (1) and (8).

With respect to the filtering stage, i.e., steps (1)–(4), the previous method had little advantage over the non-pipelined implementation in that only data transfer stages (2) and (3) were partially overlapped with filtering stage (3). In contrast, our proposed method realized a full overlap, including file I/O stage (1), such that the execution time was reduced from 59.6 s to 38.5 s.

With respect to the back-projection stage, i.e., steps (5)–(8), the previous method overlapped step (5) with step (6) such that the corresponding execution time was reduced from 205.4 s to 153.7 s. Our method further realized an overlap of steps (7) and (8), thereby reducing the execution time to 121.1 s. Note that step (8) for the last subvolume cannot be overlapped with other steps. Similarly, step (1) for the first subvolume cannot be overlapped with other steps.

5.3 Efficiency Analysis

To analyze the performance bottleneck of our method, we measured arithmetic performance, L1/texture cache throughput, L2 cache texture load throughput, and texture fill rate using the NVIDIA Visual Profiler; results are shown in Fig. 11. According to Eq. (2), each voxel requires one pixel per projection; thus, the effective texture fill rate can be given by $NXYZ/T_6$, where T_6 is the back-projection time. The peak texture fill rate was derived according to a boosted clock speed because the clock speed was boosted during back-projection.

Our results indicate three key findings. First, Fig. 11 (d) implies that the texture fill rate determined reconstruction performance for the small and medium datasets. The Maxwell architecture has eight texture units devoted to each SM; thus, 16 CUDA cores share a single texture unit. Given these limited resources, the effective texture fill rates were limited to approximately 141.2 Gtexel/s, which is only 9.3% lower than the peak (boosted) value of 155.6 Gtexel/s. These effective values were close to the peak (base) value of 144.1 Gtexel/s; thus, we conclude that our back-projection kernel is highly optimized for the Maxwell architecture. As for the large dataset, we determined that the average L2 texture read cache hit rate decreased to 55.8% due to the increased projection size, which decreased the effective texture fill rate to 104.9 Gtexel/s.

Second, our cache-aware method and the rich caching mechanism of the Maxwell architecture moved the performance bottleneck from the off-chip memory bandwidth to the texture fill rate. As shown in Fig. 11 (c), the effective throughput was approximately one-quarter of the theoretical

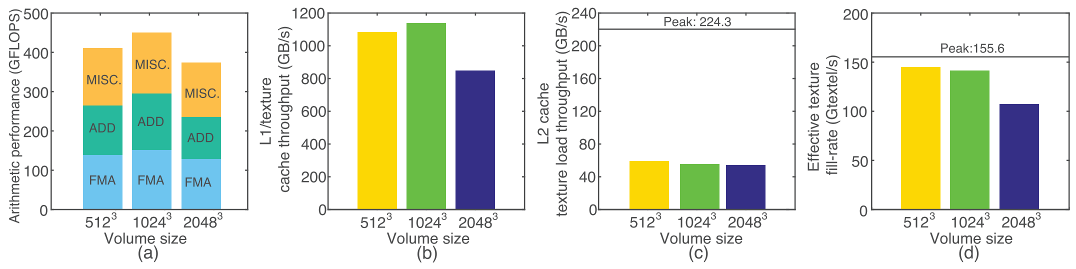

Fig. 11 Profiling results for different data sizes: (a) arithmetic performance in GFLOPS, (b) L1/texture cache throughput, (c) L2 cache texture load throughput, and (d) effective texture fill rate. FMA, ADD, and MISC in (a) refer to fused multiply-add [12], addition, and other instructions, respectively. The horizontal lines in (c) and (d) are peak memory bandwidth and peak (boosted) texture fill rate, respectively, with the latter derived according to the boosted clock speed presented in Table 2.

peak value, which indicates that off-chip memory accesses do not limit back-projection performance on the Maxwell architecture. These results were not observed in the previous study [15], which concluded that off-chip memory access was the performance bottleneck of the back-projection kernel on the G80 architecture.

Third, reconstruction performance for the large dataset was sacrificed due to limited device memory. As compared to the medium dataset, the large dataset consisted of four times larger *xy*-slices, which decreased the subvolume size Z' from 512 to 128. This decrease in Z' led to more kernel executions and reduced the amount of memory accesses per warp, i.e., $32N'Z'$ in bytes, from 1.25 MB to 0.25 MB. An alternative solution for increasing Z' is to partition the volume along the *x*- or *y*-plane instead of the *z*-plane; however, this solution requires recombining the volume after reconstruction. Such a post-processing task will likely slow the I/O-included pipeline.

6. Conclusions

In this paper, we presented a cache-aware optimization method to accelerate out-of-core cone beam reconstruction on a GPU. Our proposed method extended the previous method described in [15] and accelerated the FDK algorithm via three key strategies, i.e., an improved loop organization strategy, an improved data structure, and an I/O-included pipeline. We also presented tuning guidelines for determining the best configuration for the granularity and shape of thread blocks, as well as the projection subset size and subvolume size, i.e., the granularity of I/O data to be streamed through the pipeline.

Our experimental results showed a tradeoff between the texture cache hit rate and the number of off-chip memory accesses. We also found that it took 159.6 s on a GeForce GTX 980 to reconstruct a 2048^3-voxel volume from 1200 2048^2-pixel projections, consuming 50.8 GB of memory. This reconstruction performance is approximately 1.47 times higher than that achieved by the previous method [15]. Concerning GPU optimization, we found that it is not necessarily more efficient to compact as many tasks as possible into kernel execution to decrease kernel executions. Instead, proper tuning is required to identify the optimum number of tasks that will minimize the overall time required. With the aid of texture interpolation and cache-aware strategies, our presented GPU implementation achieved performance advantages over other computing platforms.

Our future work includes enabling our method to run on a multi-GPU machine to achieve further acceleration.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008, 15H01687, and 16H02801, and the Japan Science and Technology Agency CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] L.A. Feldkamp, L.C. Davis, and J.W. Kress, "Practical cone-beam algorithm," J. Optical Society of America, vol.1, no.6, pp.612–619, June 1984.

[2] A. Eklund, P. Dufort, D. Forsberg, and S.M. LaConte, "Medical image processing on the GPU — past, present and future," Medical Image Analysis, vol.17, no.8, pp.1073–1094, Dec. 2013.

[3] K. Machin and S. Webb, "Cone-beam X-ray microtomography of small specimens," Physics in Medicine and Biology, vol.39, no.10, pp.1639–1657, Oct. 1994.

[4] J. Kastner, B. Harrer, G. Requena, and O. Brunke, "A comparative study of high resolution cone beam X-ray tomography and synchrotron tomography applied to Fe- and Al-alloys," NDT & E Int'l, vol.43, no.7, pp.599–605, Oct. 2010.

[5] W.C. Scarfe and A.G. Farman, "What is cone-beam CT and how does it work?," Dental Clinics of North America, vol.52, no.4, pp.707–730, Oct. 2008.

[6] M.J. Wallace, M.D. Kuo, C. Glaiberman, C.A. Binkert, R.C. Orth, and G. Soulez, "Three-dimensional C-arm cone-beam CT: Applications in the interventional suite," J. Vascular and Interventional Radiology, vol.19, no.6, pp.799–813, June 2008.

[7] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," IEEE Trans. Nucl. Sci., vol.52, no.3, pp.654–663, June 2005.

[8] H. Yang, M. Li, K. Koizumi, and H. Kudo, "Accelerating backprojections via CUDA architecture," Proc. 9th Int'l Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear

Medicine (Fully 3D '07), pp.52–55, July 2007.
[9] M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware," Medical Physics, vol.34, no.4, pp.1474–1486, April 2007.
[10] N. Gac, S. Mancini, and M. Desvignes, "Hardware/software 2D-3D backprojection on a SoPC platform," Proc. 21st ACM Symp. Applied Computing (SAC'06), pp.222–228, April 2006.
[11] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Performance engineering for a medical imaging application on the Intel Xeon Phi accelerator," Proc. 27th Int'l Conf. Architecture fo Computating Systems (ARCS'14), pp.222–228, Feb. 2014.
[12] NVIDIA Corporation, "CUDA C Programming Guide Version 7.5," Sept. 2015.
[13] F. Ino, S. Yoshida, and K. Hagihara, "RGBA packing for fast cone beam reconstruction on the GPU," Proc. SPIE Medical Imaging (MI 2009), Feb. 2009. 8 pages (CD-ROM).
[14] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (CUDA)," Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'07), pp.4464–4466, Oct. 2007.
[15] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," Parallel Computing, vol.36, no.2/3, pp.129–141, Feb. 2010.
[16] NVIDIA Corporation, "NVIDIA GeForce 8800 GPU architecture overview," Tech. Rep. TB-02787-001_v01, NVIDIA Corporation, Nov. 2006.
[17] F. Ino, Y. Okitsu, T. Kishi, S. Ohnishi, and K. Hagihara, "Out-of-core cone beam reconstruction using multiple GPUs," Proc. 7th IEEE Int'l Symp. Biomedical Imaging (ISBI'10), pp.792–795, April 2010.
[18] P.B. Noël, A.M. Walczak, J. Xu, J.J. Corso, K.R. Hoffmann, and S. Schafer, "GPU-based cone beam computed tomography," Computer Methods and Programs in Biomedicine, vol.98, no.3, pp.271–277, June 2010.
[19] Z. Zheng and K. Mueller, "Cache-aware GPU memory scheduling scheme for CT back-projection," Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'10), pp.2248–2251, Nov. 2010.
[20] H. Zhang, B. Yan, L. Lu, L. Li, and Y. Liu, "High performance parallel backprojection on multi-GPU," Proc. 9th Int'l Conf. Fuzzy Systems and Knowledge Discovery (FSKD'12), pp.2693–2696, May 2012.
[21] J. Treibig, G. Hager, H.G. Hofmann, J. Hornegger, and G. Wellein, "Pushing the limits for medical image reconstruction on recent standard multicore processors," Int'l J. High Performance Computing Applications, vol.27, no.2, pp.162–177, May 2013.
[22] E. Papenhausen and K. Mueller, "Rapid Rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction," Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'13), pp.1–2, Oct. 2013. 2 pages.
[23] T. Zinßer and B. Keck, "Systematic performance optimization of cone-beam back-projection on the Kepler architecture," Proc. 12th Int'l Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D '13), pp.225–228, June 2013.
[24] J.G. Blas, M. Abella, F. Isalia, J. Carretero, and M. Desco, "Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray reconstruction algorithm," The J. Systems and Software, vol.95, pp.166–175, Sept. 2014.
[25] E. Serrano, G. Bermejo, J.G. Blas, and J. Carretero, "High-performance X-ray tomography reconstruction algorithm based on heterogeneous accelerated computing systems," Proc. 16th IEEE Int'l Conf. Cluster Computing (CLUSTER'14), pp.331–338, Sept. 2014.
[26] NVIDIA Corporation, "NVIDIA GeForce GTX 980," Nov. 2014.
[27] OpenACC-Standard.org, "The OpenACC application programming interface, version 2.0," Aug. 2013.
[28] M. Leeser, S. Mukherjee, and J. Brock, "Fast reconstruction of 3D volumes form 2D CT projection data with GPUs," BMC Research Notes, vol.7, no.582, June 2014. 8 pages.
[29] Intel Corporation, "Intel architecture instruction set extensions programming reference," Dec. 2013.
[30] NVIDIA Corporation, "Tuning CUDA applications for Maxwell," Sept. 2015.
[31] Y. Sugimoto, F. Ino, and K. Hagihara, "Improving cache locality for GPU-based volume rendering," Parallel Computing, vol.40, no.5/6, pp.59–69, May 2014.
[32] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," May 2012.
[33] NVIDIA Corporation, "Profiler User's Guide Version 7.5," Sept. 2015. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf.
[34] L.A. Shepp and B.F. Logan, "The Fourier reconstruction of a head section," IEEE Trans. Nucl. Sci., vol.21, no.3, pp.21–43, June 1974.

Yuechao Lu received the M.E. degree in Vehicle Engineering from Tongji University, Shanghai, China, in 2011. He is currently working toward the Ph.D. degree in computer science at Osaka University. His current research interests include computer graphics and high performance computing. Ph.D. degree in computer science at Osaka University.

Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Associate Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.

Kenichi Hagihara received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. From 1992 to 1993, he was a Visiting Researcher at the University of Maryland. His research interests include the fundamentals and practical application of parallel processing.