IEICE
TRANSACTIONS

on Information and Systems

DOI:10.1587/transinf.2016EDP7178

Publicized:2016/12/12

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Cache-aware, In-place Rotation Method for Texture-based Volume Rendering

Yuji MISAKI[†], ***Nonmember***, **Fumihiko INO**[†], ***and*** **Kenichi HAGIHARA**[†], ***Members***

SUMMARY We propose a cache-aware method to accelerate texture-based volume rendering on a graphics processing unit (GPU) that is compatible with the compute unified device architecture. The proposed method extends a previous method such that it can maximize the average rendering performance while rotating the viewing direction around a volume. To realize this, the proposed method performs in-place rotation of volume data, which rearranges the order of voxels to allow consecutive threads (warps) to refer to voxels with the minimum access strides. Experiments indicate that the proposed method replaces the worst texture cache (TC) hit rate of 42% with the best TC hit rate of 93% for a 1024^3-voxel volume. Thus, the average frame rate increases by a factor of 1.6 in the proposed method compared with that in the previous method. Although the overhead of in-place rotation slightly decreases the frame rate from 2.0 frames per second (fps) to 1.9 fps, this slowdown occurs only with a few viewing directions.

key words: *Cache optimization, volume rendering, in-place algorithm, GPU, CUDA.*

1. Introduction

Volume rendering [1] is a visualization technique that can generate a 2-D projection of 3-D volume data with an arbitrary viewing direction. For example, volume rendering visually supports the understanding of time-varying fluid simulation [2–4] and computer-aided diagnosis in clinical fields [5, 6]. A typical visualization technique is ray casting [7], which propagates rays from the viewpoint to every pixel on the screen, as shown in Fig. 1. A pixel value can be computed by accumulating the values of penetrated voxels on the ray at regular intervals. Ray casting can be easily parallelized because each pixel computation is independent in terms of data dependence. However, ray casting is a memory-bound application because of the low locality of references; penetrated voxels can only be reused between neighboring rays.

Therefore, many volume renderers [8] are accelerated by a graphics processing unit (GPU) [9], which provides not only one magnitude higher memory bandwidth over the CPU but also thousands of processing cores within a chip. For example, NVIDIA's Maxwell architecture [9] provides 336 GB/s of memory bandwidth with 3072 processing cores. These rich resources use millions of GPU threads to independently compute pixel values. Furthermore, the GPU provides hardware-accelerated trilinear interpolation. This acceleration is realized by texture units [10] that are separated from the processing cores. Thus, interpolation for every sampling point on rays can be offloaded from the processing cores to the texture units. To take advantage of this technique, existing volume renderers [11–14] typically store volume data in texture memory, which is then accessed by the processing cores to rapidly fetch interpolated texels through a texture cache (TC). Consequently, maximizing the TC hit rate is the key to achieving high performance for memory-bound volume rendering on a GPU.

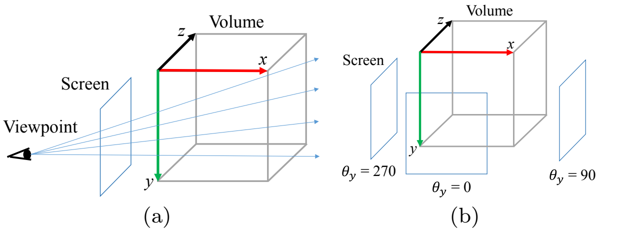


Fig. 1 Geometric relations between the volume axes and the screen: (a) rays are propagated from the viewpoint, which can be rotated around the volume; (b) θ_y represents the viewing angle around the y-axis. Angles $\theta_y = 90$ and 270 are unfavorable whereas angle $\theta_y = 0$ is favorable.

Previous studies have accelerated volume rendering by maximizing the TC hit rate. Sugimoto *et al.* [13] presented a cache-aware volume rendering method that dynamically selects the best shape (i.e., width and height) of thread blocks [10], according to the viewing direction. Their dynamic method attempts to minimize the memory access stride for warps [10], i.e., a series of consecutive threads that execute the same instruction every clock cycle. To achieve this, they analyzed the memory access stride, which varies according to the direction of the neighboring voxels, i.e., the x-, y-, and z-directions, as shown in Fig. 2. For example, their method selects a horizontally long thread block (i.e., horizontally long warps) when the x-axis of the volume, which has the smallest stride between neighboring voxels, is rendered horizontally on the screen. With an NVIDIA GeForce 580 GTX GPU, this minimization increases the worst frame rates by a factor of

[†] The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871, Japan.

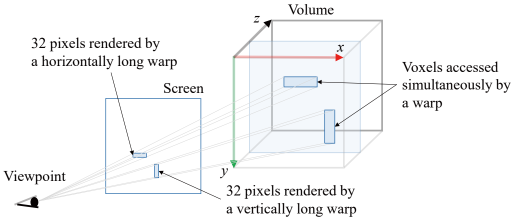


Fig. 2 Principles of thread block shaping [13]. Thirty-two threads in a warp are assumed to simultaneously access voxels on a facing place (the xy-plane, in this example). The shape of thread blocks is determined such that the memory access stride between simultaneously accessed voxels is minimized. In this example, a horizontally long shape rather than a vertically long shape is selected because neighboring voxels along the x-axis have a smaller stride than those along the y-axis.

2.2 when projecting a 1024^3-voxel volume onto a 1024^2-pixel screen. The main drawback of this method is that the minimization is insufficient for *unfavorable viewing directions* where the x-axis is parallel to the penetrating rays as shown in Fig. 1(b). In general, threads in the same warp simultaneously access voxels on the yz-plane in this case, and this access pattern causes relatively large strides because the simultaneously accessed voxels are not oriented along the x-axis. In fact, the best frame rate was 2.7 times higher than the worst frame rate.

In this study, we present a cache-aware method to accelerate texture-based volume rendering on a GPU. The proposed method extends a previous method [13] such that it can increase the average frame rate obtained while rotating the volume using an in-place rotation algorithm that dynamically rearranges the order of voxels to allow unfavorable viewing directions to be interpreted as favorable. Here, an in-place algorithm is defined as an algorithm that requires $\mathcal{O}(1)$ memory space (except the original volume data). The in-place procedure is activated when the viewing direction enters from a favorable (unfavorable) region to an unfavorable (favorable) region. This data rearrangement takes place at run-time; therefore, its overhead can drop the worst rendering performance. However, after changing the order of voxels, unfavorable directions can be processed as favorable directions rapidly; thus, the average performance can be significantly increased. Our renderer was implemented on a GPU that is compatible with the compute unified device architecture (CUDA) [10].

Our in-place implementation is based on a tile-based matrix transpose method [15] that utilizes (on-chip) shared memory [10] for optimization of off-chip memory access. Because the volume rotation cannot be realized with matrix transpose, we extend this previous method for volume rotation. We show that (1) a tile-based approach is useful for in-place rotation and (2) 3-D tiles rather than 2-D tiles realize efficient rotation with reduced (off-chip) memory access.

The remainder of this paper is organized as follows. Related studies are discussed in Section 2. Section 3 summarizes the previous method [13], which is the basis of the proposed cache-aware method. Section 4 describes the proposed method, and Section 5 gives experimental results. Section 6 concludes this paper and provides suggestions for future studies.

2. Related Work

Weiskopf *et al.* [16] presented a data structure that splits a volume into smaller subvolumes called bricks. Each brick comprises 4^3 voxels, and the bricks are oriented in different directions. Consequently, for any viewing direction, half the bricks are oriented along the viewing direction and the remaining bricks are oriented in the perpendicular direction. Thus, the performance variance for different viewpoints is averaged. In contrast to this averaging strategy, the proposed method accepts some overheads at specific viewing directions to obtain the maximum performance for other viewing directions.

A similar averaging strategy was presented by Wang *et al.* [14, 17], who presented a sampling strategy called warp marching to realize constant rendering performance while rotating the volume by exploiting the data parallelism not only between rays but also on each ray. In other words, a warp is responsible for a ray so that threads in the warp can simultaneously access voxels along the ray. This is useful for unfavorable viewing directions where the x-axis is parallel to the rays. With an NVIDIA GeForce GTX TITAN GPU, their method increased the worst frame rates from 15 frames per second (fps) to 30 fps when projecting a 1024^3-voxel volume onto a 512^2-pixel screen. However, the best frame rate dropped from 60 fps to 45 fps because the best performance was obtained while facing the xy-plane. In other words, the warps responsible for the same ray can suffer from large strides when accessing voxels neighboring along the z-axis.

Jönsson *et al.* [18] presented a data structure that exploits inter-ray coherence by sharing cached data called marching cache between neighboring rays. Their method stores a marching cache in shared memory [10], which can be rapidly accessed as software managed cache. However, one drawback of shared memory is a lack of hardware-accelerated interpolation. Compared with a texture-based implementation, the rendering performance decreased by roughly 71% using trilinear interpolation. Consequently, marching caches are useful for high-quality volume rendering that uses a complex filter, such as the Catmull-Rom spline [19] or a B-spline [20]. A similar approach was presented by Mensmann *et al.* [21], who stored the subvolume in shared memory.

In general, cache-aware methods are useful for accelerating memory-bound applications. Zheng *et*

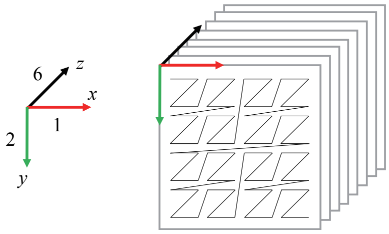


Fig. 3 Memory access strides between neighboring voxels in a 3-D texture. The access stride ratio of the x-, y-, and z-axis is 1:2:6 [13] if the texture is stored in the Morton order [25].

al. [22] presented a cache-aware memory-scheduling scheme for cone beam backprojection [23, 24], which is an inverse volume rendering problem. In other words, a series of 2-D projections are backprojected into 3-D space to reconstruct volume data. To increase the cache hit rate during backprojection, their method rotates the volume such that warps can access a small region on the projections. This idea is similar to our data rearrangement strategy, but we employ an in-place rotation algorithm for an inverse problem wherein warps primarily access a small region in the volume (3-D texture) rather than the projections (2-D textures).

3. Principles of Warp-Level Cache Optimization

The basic idea of the previous method [13] is the warp-level cache optimization for volume rendering. This warp-level strategy is effective on the GPU because threads in the same warp issue the same memory transaction simultaneously. To achieve this warp-level optimization, Sugimoto *et al.* [13] analyzed the memory access strides along the volume axes. They focused on the Morton order [25] and showed that the memory stride ratio along the x-, y-, and z-axis of a 3-D texture is around 1:2:6, as shown in Fig. 3. This non-uniform ratio indicates that the rendering performance can vary according to the viewing direction. In fact, rays can propagate along the x-, y-, or z-axis. We assume that a thread is responsible for computing a pixel value on the screen. Here, let θ_y be the viewing angle around the y-axis, as shown in Fig. 1.

Voxels are sampled at regular intervals from the viewpoint; thus, a warp simultaneously accesses voxels on the surface of a sphere. For simplicity, the previous method assumes that this spherical surface can be approximated with a plane. Under this approximation, a warp simultaneously accesses voxels on a plane that are parallel to the screen (Fig. 2). For such a facing plane, there are three cases, i.e., the xy-, yz-, and xz-plane. Moreover, there are two variations for each case, i.e., the vertical (horizontal) axis of the plane has smaller access strides than the horizontal (vertical) axis. Thus, any viewing direction can be classified into these 3×2 groups.

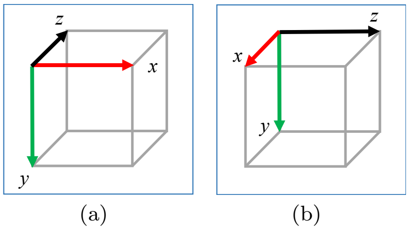


Fig. 4 Favorable and unfavorable viewing directions: (a) favorable viewing direction $\theta_y = 0$; (b) unfavorable viewing direction $\theta_y = 90$ (the facing plane for the former is the xy-plane and that for the latter is the yz-plane, which does not include the x-axis).

Given a viewing direction, the previous method first identifies the facing plane and selects an axis from the two axes that constitute the facing plane such that the selected axis has smaller access strides. For example, the facing plane in Fig. 4(a), where $\theta_y = 0$, is the xy-plane and the plane axis with smaller access strides is the x-axis, which can be seen horizontally on the screen. In this case, the previous method selects horizontally long thread blocks (i.e., horizontally long warps) to allow warps to access voxels with the minimum access strides. In contrast, for the viewing direction presented in Fig. 4(b), where $\theta_y = 90$, the method selects vertically long thread blocks because the vertical axis has smaller access strides than the horizontal axis (i.e., the y-axis rather than the z-axis).

Although the previous method selects the best thread block shape for the viewing direction, this selection is insufficient when the facing plane comprises the y- and z-axis. Such an unfavorable viewing direction is shown in Fig. 4(b). In this example, the x-axis is parallel to the viewing direction; thus, warps fail to access voxels with the minimum access strides. To address this issue, a ray must be assigned to a warp (i.e., multiple threads), as proposed by Wang *et al.* [14]. An alternative solution is to rearrange the order of voxels, which we present in the next section.

4. Proposed Method

The basic idea of the proposed method is to rearrange the order of voxels to allow unfavorable viewing directions to be interpreted as favorable. For example, the viewing direction in Fig. 4(b) can be considered as that in Fig. 4(a) if the volume is rotated 90° around the y-axis prior to rendering. After this data rearrangement, the facing plane changes from the yz-plane to the xy-plane, which can be rapidly accessed by warps with the minimum access strides.

Note that this rotation is intended to change the order of voxels in the texture rather than update the view matrix. Consequently, a naive method might distinguish the output volume from the input volume during rotation. However, this method consumes twice the video memory, which reduces the maximum volume size

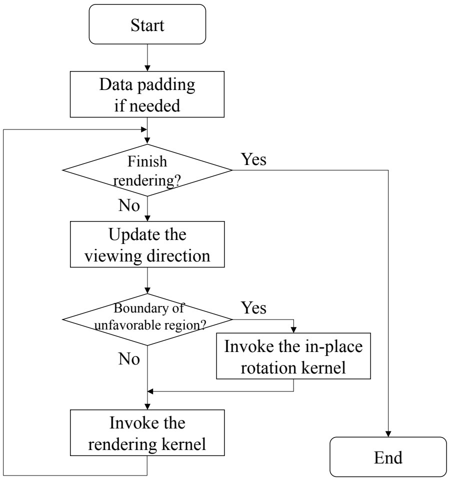

Start
Data padding if needed
Finish rendering?
Yes
No
Update the viewing direction
Boundary of unfavorable region?
Yes
No
Invoke the in-place rotation kernel
Invoke the rendering kernel
End

Fig. 5 Flowchart of the proposed volume rendering method. The "Start" and "End" states are associated with a CPU rendering process.

that can be rapidly rendered without swapping data from the video to the main memory. Therefore, the proposed method realizes in-place rotation of the volume, which integrates the input volume with the output volume to limit memory consumption.

Figure 5 shows an overview of the proposed cache-aware volume rendering method. The proposed method first identifies whether the viewing direction is on a boundary of an unfavorable region. If a viewing angle θ_y is within the range of $[45, 135)$ or $[225, 315)$, the viewing direction is unfavorable; otherwise, it is favorable. After this identification process, the proposed method invokes an in-place rotation kernel if the viewing direction enters from a favorable (unfavorable) region to an unfavorable (favorable) region. Finally, the method invokes a rendering kernel to produce the final image for the viewing direction. This rendering kernel is invoked with an appropriate shape of the thread blocks to perform warp-level cache optimization [13]. Note that the in-place rotation kernel must be separated from the rendering kernel because barrier synchronization is required between the two kernels.

4.1 In-place rotation

Similar to the matrix transpose method [15], the proposed method is based on a tile-based approach that facilitates in-place rotation of the volume. We assume that xz-slices of the volume are isometric, i.e., the volume size N_x along the x-axis is the same as N_z along the z-axis. For non-isometric data, dummy transparent voxels are padded to make the data isometric prior to rendering. This padding operation can be processed as an initialization phase, which occurs only once after the dataset to be rendered has been loaded. The proposed method avoids rendering such dummy voxels using an empty space skipping technique [26].

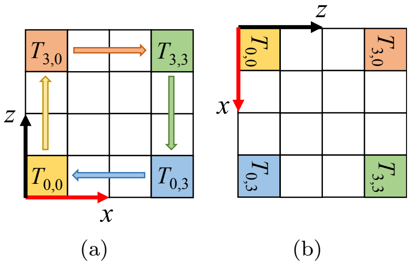


Fig. 6 Tiled approach for rotating an xz-slice: (a) before rotation and (b) after rotation. Four data-dependent tiles $T_{0,0}$, $T_{0,3}$, $T_{3,3}$, and $T_{3,0}$, each containing $t \times t$ voxels, are rotated with a cyclic permutation processed by a thread block. Reads and writes to a tile are parallelized by $t \times t$ threads within the thread block.

Here, let N $(= N_x = N_z)$ be the volume size. Let $t \times t$ be the thread block size of the in-place rotation kernel. Figure 6 shows a data arrangement example that rotates an xz-slice 90° around the y-axis. As shown, the proposed method partitions the xz-slice into square tiles of $t \times t$ voxels. Note that, for simplicity, we describe our method using 2-D tiles of $t \times t$ voxels; however, our implementation uses 3-D tiles of $t \times t \times k$ voxels, where k (> 0) represents the tile size along the y-axis. Here, let $T_{i,j}$ be a tile of the i-th row and the j-th column, where $0 \leq i, j < N/t$. We assume that N is a multiple of t for simplicity.

A series of data-dependent tiles is then assigned to a thread block. For example, the series of tiles $T_{0,0}$, $T_{0,3}$, $T_{3,3}$, and $T_{3,0}$ in Fig. 6 must be processed by the same thread block because the i-th tile moves to the $((i+1) \bmod 4)$-th tile, where $0 \leq i < 4$. In other words, the rotation of these four tiles can be considered a cyclic permutation, which must be processed in sequence. However, voxels in a tile can be moved in parallel so that tile movement can be parallelized by a thread block. More formally, a series of data-dependent tiles can be given by $T_{i,j}$, $T_{N/t-j-1,i}$, $T_{N/t-i-1,N/t-j-1}$ and $T_{j,N/t-i-1}$, where $0 \leq i, j < N/t$.

Algorithm 1 shows the pseudocode of the in-place rotation kernel, which is processed by GPU threads in a single-instruction multiple-thread manner [10]. We assume that the local index of a thread is given by (o_1, o_2), where $0 \leq o_1, o_2 < t$. Given the original volume V and the global coordinates (x, y, z) of the local origin of the first tile $T_{i,j}$, the GPU threads output the rotated volume using shared memory. To do this, each thread iterates the following steps four times:

1. Tile load (lines 1 and 7). Each thread accesses video memory to load a voxel of the current tile in a register. Threads in the same warp simultaneously access contiguous memory addresses during this step.
2. Tile rotation (line 9). Loaded voxels are written into shared memory such that the shared memory holds the rotated tile.

Algorithm 1 In-place rotation kernel

```
Input: Original volume $V$
Input: The global coordinates $(x, y, z)$ of the local origin of the first tile $T_{i,j}$
Input: The local index $(o_1, o_2)$ of a thread, where $0 \leq o_1, o_2 < t$
Output: Rotated volume $V$
 1: $r_1 \leftarrow V[x + o_1][y][z + o_2]$;  ▷ load a voxel of the first tile in a register
 2: for $i = 0$ to 3 do  ▷ for each data-dependent tile
 3:     $tmp \leftarrow z$;  ▷ compute the coordinates of the next tile
 4:     $z \leftarrow x$;
 5:     $x \leftarrow N/t - tmp - 1$;
 6:     if $i \neq 3$ then
 7:         $r_2 \leftarrow V[x + o_1][y][z + o_2]$;  ▷ load a voxel of the next tile in a register
 8:     end if
 9:     $shared[o_2][t - o_1 - 1] \leftarrow r_1$;  ▷ store the rotated tile in the shared memory
10:     $\_\_syncthreads()$;
11:     $V[x + o_1][y][z + o_2] \leftarrow shared[o_1][o_2]$;  ▷ store the rotated tile in the video memory
12:     $r_1 \leftarrow r_2$;  ▷ prepare for the next iteration
13:     if $i \neq 3$ then
14:         $\_\_syncthreads()$;
15:     end if
16: end for
```

3. Tile store (line 11). Each thread stores the rotated tile from the shared memory to the video memory. Similar to the first step, threads in the same warp simultaneously access contiguous memory addresses during this step.

Note that t-strided memory accesses are required to rotate a tile by threads. Such noncontiguous memory accesses cannot be handled efficiently with (off-chip) video memory because memory transactions are performed per warp [10]. Therefore, our kernel uses shared memory (i.e., on-chip memory) to avoid strided accesses to the video memory. However, synchronization is required prior to loading shared data, which can be stored asynchronously by other threads.

As mentioned above, our method can be regarded as an extension of the matrix transpose method [15]. However, this method cannot be directly used for volume rotation, which requires cyclic permutations instead of simple swap operations. Each permutation consists of a sequence of shift operations applied to four tiles as illustrated in Fig. 6. Consequently, our method assigns these four tiles to a thread block whereas the matrix transpose method assigns two tiles to a thread block.

4.2 Issues with writable textures

As mentioned in Section 1, a volume is typically stored in a 3-D texture to take advantage of hardware-accelerated trilinear interpolation. However, the CUDA currently prohibits GPU threads from writing to textures [10]. Consequently, the texture data cannot be directly rotated by GPU threads.

Table 1 Experimental environment.

Item	Specification
CPU	Intel Core i7-3770K 3.5 GHz
Main memory capacity	16 GB
GPU	NVIDIA GeForce GTX TITAN X
Video memory capacity	12 GB
OS	Windows 7
CUDA version	7.0
Driver version	353.30

To address this issue, we adopt a two-way approach to access the volume data. This approach uses texture memory and surface memory [10], which is writable from GPU threads but does not provide hardware-accelerated interpolation. In other words, the CUDA array [10] allocated for the volume is bound to a texture object and a surface object. The in-place rotation kernel, which never requires interpolated values, then writes the volume data through the surface object. In addition, the rendering kernel accesses the volume data through the texture object to take advantage of hardware-based interpolation.

Notice here that only a single CUDA array is required to store the entire volume with padded area. The presented two-way approach never requires a copy of the volume data. Consequently, the overall memory usage can be approximated by the number of voxels in the original volume and padded region.

5. Experimental Results

We evaluated the proposed cache-aware method by comparing it to a previous method [13] and a naive method distributed as CUDA sample code [10]. Comparisons were performed in terms of the TC hit rate and the frame rate. Similar to the previous method [13], a color map table, which associates color and opacity values with each voxel, was stored in shared memory to avoid perturbation of TC behavior. All methods activated early ray termination [26] to skip accumulation that did not significantly contribute to the pixel.

According to [27], at least four over-samplings are required to reconstruct the ray integral with sufficient accuracy. Consequently, the sampling distance along the rays was set to $\max(N_x, N_y, N_z)/4$, so that the renderer used at most $4 \times \max(N_x, N_y, N_z)$ samples per ray, starting from the first penetrated voxel in the volume. We determined experimentally that $t = 32$ achieves the highest rendering performance on the deployed GPU. Details of the experimental environment are listed in Table 1.

We used four volume datasets: head magnetic resonance imaging (MRI), Hazelnut, Flower, and Porsche, as shown in Fig. 7. The head MRI and Porsche datasets are non-isometric; the Hazelnut and Flower datasets are isometric. The isometric datasets differ with respect to volume size. All voxels had an 8-bit scalar value, and the volume was stored in a 3-D texture in `unsigned`

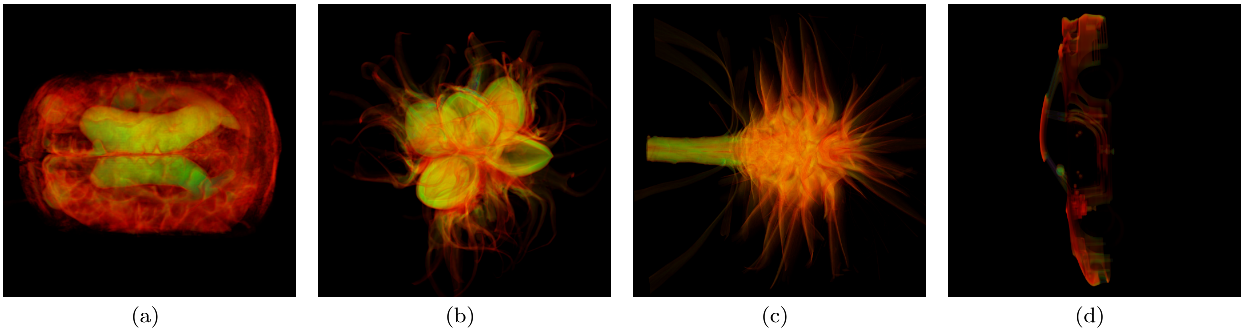


Fig. 7 Experimental datasets rendered on a 1024 × 1024-pixel screen: (a) head MRI (256 × 256 × 124 voxel); (b) Hazelnut (512 × 512 × 512 voxel); (c) Flower (1024 × 1024 × 1024 voxel); and (d) Porsche (559 × 1023 × 347 voxel). Head MRI and Porsche datasets were obtained from http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/new.html and http://www9.informatik.uni-erlangen.de/External/vollib/, respectively. Hazelnut and Flower datasets were obtained from http://www.ifi.uzh.ch/vmml/research/datasets.html.

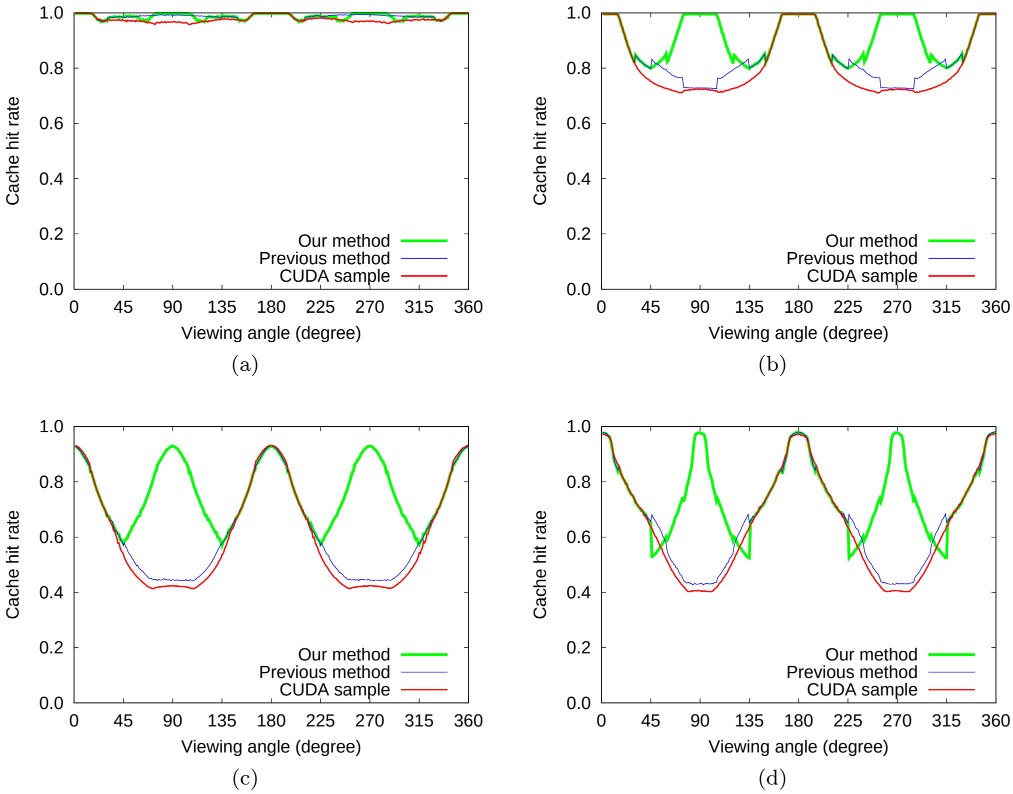


Fig. 8 TC hit rates of the proposed method, previous method [13], and CUDA sample code [10] for (a) head MRI, (b) Hazelnut, (c) Flower, and (d) Porsche datasets.

`char` format. The screen size was set to 1024^2 pixels during the experiments. A pair of movie files capturing real-time visualization of the Flower dataset is available from `http://www-hagi.ist.osaka-u.ac.jp/research/movie/misaki-{proposed,previous}.mp4`.

5.1 Rendering performance

Figure 8 shows the TC hit rates of the rendering kernel measured while rotating the viewing direction around the y-axis, which significantly varies the rendering performance. As for the rotation around the x-axis or z-axis, where the volume is always rendered from favorable viewing directions, the rendering performance was kept high [13]. Consequently, these results are not presented here. The TC hit rates were obtained using the CUDA toolkit nvprof profiling tool [28].

As shown in Fig. 8(a), there was no significant gap between the TC hit rate of the proposed method and that of the previous method when the smallest head

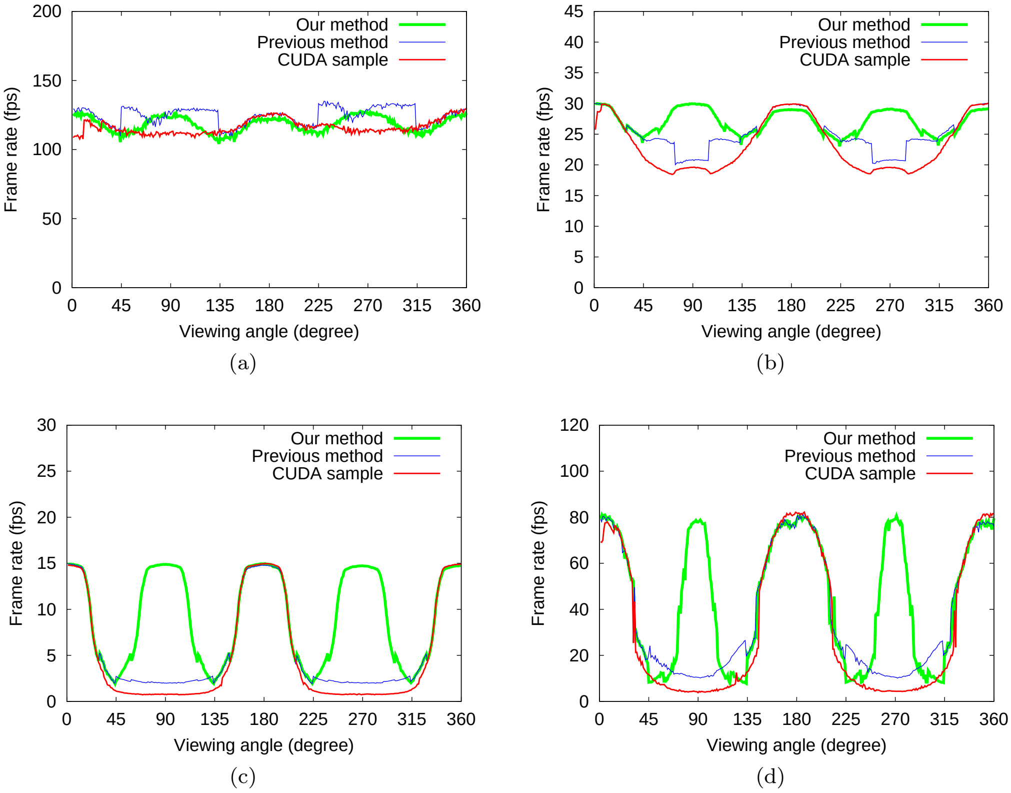


Fig. 9 Frame rates of the proposed method, previous method [13], and CUDA sample code [10] for (a) head MRI, (b) Hazelnut, (c) Flower, and (d) Porsche datasets.

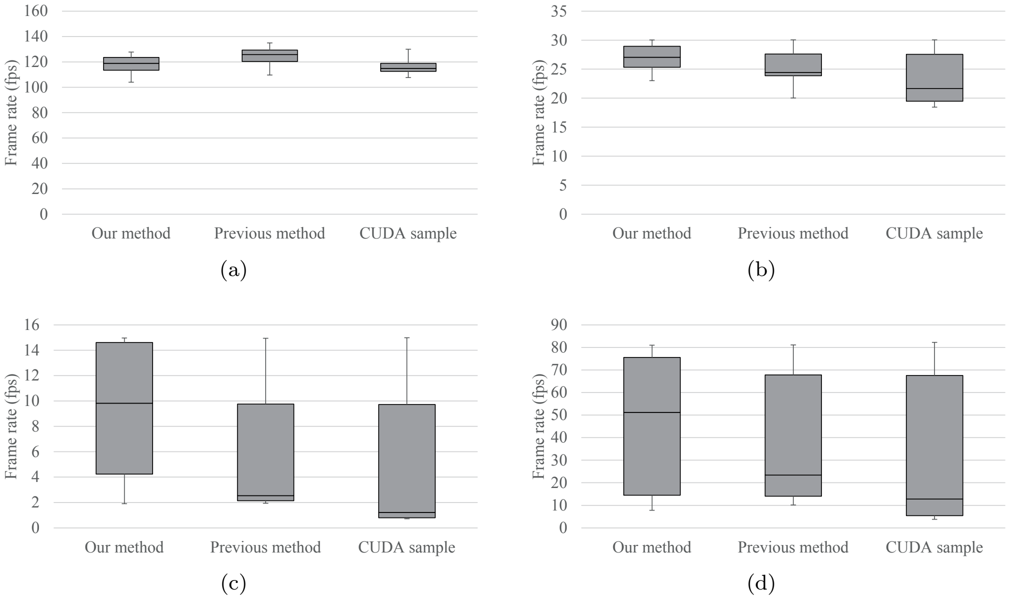
MRI dataset was rendered. Obviously, such a small dataset can be rendered efficiently with many cache hits because the largest stride between neighboring voxels is given by $N_x \times N_y = 65,536$. Consequently, 64 KB of memory is sufficient to store an xy-slice of the volume. On the other hand, the deployed GPU has 1152 KB of L1 texture cache; the GeForce GTX TITAN X has 24 streaming multiprocessors, each having 2 × 24 KB of L1 texture cache [29, 30].

However, as shown in Figs. 8(b) and 8(c), the gap between the proposed method and the previous method increased for larger datasets that comprised at least 512^3 voxels. Both large datasets show similar behavior in terms of the TC hit rate. The proposed method significantly increased the TC hit rates for unfavorable viewing directions within the ranges of $[45, 135)$ and $[225, 315)$. For example, with the Flower dataset (Fig. 8(c)), the worst TC hit rates at $\theta_y = 90$ and 270 increased from 42% to 93%. Moreover, this increased TC hit rate was almost equal to the best hit rate achieved by the previous method, i.e., 93% at $\theta_y = 0$ and 180. In other words, the proposed method successfully translated unfavorable viewing directions into favorable viewing directions.

A similar periodical behavior was observed with the non-isometric Porsche dataset. Note that the proposed method decreased the TC hit rate at four viewing regions: $[46, 59)$, $[126, 136)$, $[226, 236)$, and $[302, 316)$ (Fig. 8(d)). This side effect was due to our rotation approach, which exchanges N_x with N_z. In other words, this non-isometric dataset satisfies $N_x > N_z$ $(559 > 347)$ before rotation; however, rotation changes this to $N_x < N_z$. Thus, the number of non-dummy voxels neighboring along the x-axis was reduced after rotation. Therefore, the TC hit rate decreased within the abovementioned four viewing regions. For the same reason, the peaks at $\theta_y = 90$ and 270 are more pronounced with the proposed method than those of the previous method and the CUDA sample code at $\theta_y = 0$ and 180.

Next, we analyzed the rendering performance, which includes the in-place rotation kernel run-time overhead. Figure 9 shows the frame rates observed while rotating the viewing direction around the y-axis. With the smallest dataset, i.e., head MRI, there was no significant gap between the frame rates of the proposed method and those of the previous method (Fig. 9(a)). However, similar to the TC hit rate behavior, the proposed method successfully increased frame rates for large datasets, as shown in Figs. 9(b) and 9(c). Thus, maximizing the TC hit rate is critical to maximizing rendering performance for large datasets.

Compared with the previous method, the worst frame rate for the Flower dataset was reduced from



(a)

(b)

(c)

(d)

Fig. 10 Box plots of frame rates of the proposed method, previous method [13], and CUDA sample code [10] for (a) head MRI, (b) Hazelnut, (c) Flower, and (d) Porsche datasets. Maximum, 75% percentile, 50% percentile (median), 25% percentile, and minimum rates are presented.

approximately 2.0 fps to 1.9 fps with the proposed method. These dropped frame rates were observed when $\theta_y = 45$, 135, 225, or 315, where the in-place rotation kernel was invoked to rearrange the order of voxels. Similarly, the overhead of the rotation kernel was ignorable for the Hazelnut dataset as shown at $\theta_y = 45$, 135, 225, and 315 in Fig. 10(b). In contrast, the overhead slightly decreased the frame rate for non-isometric datasets: the head MRI and Porsche datasets as can be seen in Figs. 10(a) and 10(d), respectively. This was due to padded voxels. Although these transparent voxels were skipped during rendering, the frame rate slightly dropped at these angles. However, the invocation overhead allowed unfavorable regions $[45, 135)$ and $[225, 315)$ to be processed as rapidly as the remaining favorable regions. Consequently, compared to the previous method, the execution times spent from $\theta_y = 0$ to $\theta_y = 359$ were reduced by 13%, 38%, and 18% for the Hazelnut, Flower, and Porsche datasets, respectively; as for the smallest MRI Head dataset, the execution time increased by 5%. Thus, the rendering performance was primarily dominated by texture access.

Box plots of the measured frame rates are shown in Fig. 10. As volume size increased, the performance gain of the proposed method was evident. The most significant improvement can be seen in the median of the measured performance, which increased from 2.5 fps to 9.8 fps for the Flower dataset (Fig. 10(c)). The average frame rate also increased from 5.7 fps to 9.3 fps. A similar significant increase was observed with the non-isometric Porsche dataset. Consequently, the proposed method can increase the average rendering performance at the cost of four viewing directions, where in-place rotation occurs ($\theta_y = 45$, 135, 225, and 315).

Note that with the previous method, there was a relatively large gap between the median and average frame rates; the gap was 3.2 ($= 5.7 - 2.5$) fps with the previous method, whereas it was 0.5 ($= 9.8 - 9.3$) fps with the proposed method. This large gap implies that the previous method suffers from many low frame rates below the median. Such low frame rates were primarily observed when the volume was rendered with unfavorable viewing directions. Similar behavior was observed with the CUDA sample code.

As presented above, our method increased the frame rates for many viewing directions at the expense of few directions (slowdowns). Because volume rendering is designed for visualization with an arbitrary viewing direction, we think that these slowdowns are acceptable if the rendering performance increases for many viewing directions. In fact, when rotating the Flower dataset, we faced little (4%) slowdowns only for 1.1% of viewing directions, whereas the frame rates increased by ×3.5 for 50% of viewing directions. No significant performance improvement (>2%) was observed for the remaining (49%) directions. In this case, the influence of the worst frame rate was limited, and thus, demonstrating the practicality of our method.

5.2 Efficiency of in-place rotation

We also analyzed the efficiency of the in-place rotation kernel because rendering performance can be limited by its run-time overhead. As shown in Fig. 9, the worst frame rates at $\theta_y = 45$, 135, 225, and 315 decreased slightly with the proposed method.

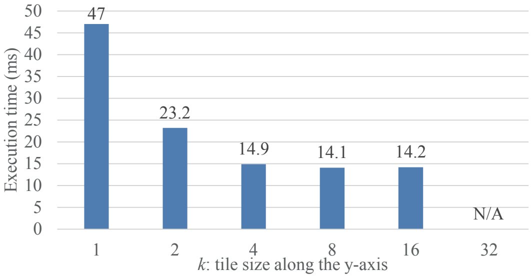


Fig. 11 Execution time of the in-place rotation kernel with different tile sizes $t \times t \times k$, where $t = 32$ for Flower dataset. Due to the limited size of thread blocks (i.e., the maximum number of threads in a thread block), execution with $k = 32$ resulted in a failure.

Figure 11 shows the execution times of the in-place rotation kernel with different tile sizes $t \times t \times k$, where $t = 32$ and $1 \leq k \leq 32$. The execution time was minimized when $k = 16$. With $k = 16$, the effective memory throughput reached 231 GB/s, which was 93% of the effective off-chip memory bandwidth (248 GB/s) measured by the CUDA software development kit bandwidthTest program. Because the performance of in-place rotation is limited by off-chip memory access, we believe that the achieved performance is the best result with the deployed GPU.

The execution times at $k = 1$ and $k = 2$ were, respectively, 4.1 and 2.2 times longer than the shortest execution time ($k = 16$). According to the nvprof profiling tool, execution with $k = 1$ ($k = 2$) incurred four (two) times more off-chip memory accesses than with $k = 4$. In contrast, off-chip memory access was nearly constant when $k \geq 4$. This behavior indicates the internal structure of surface memory, which uses byte addressing [10]. Owing to this byte addressing, surface memory might be optimized for 3-D surface objects of depth of at least four. Thus, 3-D tiles ($k > 1$) rather than 2-D tiles ($k = 1$) realize efficient rotation with reduced off-chip memory access.

Finally, the execution time of in-place rotation was much shorter than that of data transfer between the CPU and GPU. We found that it took at least 270 ms to swap out the Flower dataset from the video memory to the main memory, whereas the in-place rotation took 9.3 ms, as shown in Fig. 11. Therefore, without the proposed in-place algorithm, the worst rendering performance dropped from 1.9 fps to 1.3 fps due to the data transfer required to swap out the volume data from the video memory to the main memory. In contrast, our in-place algorithm avoided dropping the worst rendering performance because it took 9.3 ms to complete the rotation kernel; the rotation kernel can be processed at 108 (=1000/9.3) fps even though it is invoked after every frame.

6. Conclusion

In this study, we have presented a cache-aware method for accelerating texture-based volume rendering on a GPU. The proposed method maximizes the TC hit rate by changing the order of voxels at specific viewing directions. This data rearrangement is performed by an in-place algorithm, which requires $\mathcal{O}(1)$ memory space (except the volume data). Thus, a large volume can be rapidly rendered without swapping data from the video memory to main memory. We integrated the proposed method into a previous cache-aware method [13] that is capable of selecting the best thread organization according to the viewing direction.

We compared the proposed method with the previous method and the CUDA sample code. The experimental results show that the proposed method successfully increased the worst TC hit rate (42%) to 93%. Accordingly, the average frame rate increased from 5.7 fps to 9.3 fps, achieving a 1.6× speedup over the previous method. We also found that the performance of the proposed in-place rotation kernel was limited by off-chip memory bandwidth.

In the future, we plan to develop a sophisticated mechanism to dynamically activate the in-place rotation kernel according to the movement history of the viewing directions.

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] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," Computer Graphics (Proc. SIGGRAPH'88), vol.22, no.3, pp.65–74, Aug. 1988.

[2] H.W. Shen, L.J. Chiang, and K.L. Ma, "A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree," Proc. 10th IEEE Visualization Conf. (VIS'99), pp.371–377, Oct. 1999.

[3] E.B. Lum, K.L. Ma, and J. Clyne, "Texture hardware assisted rendering of time-varying volume data," Proc. 12th IEEE Visualization Conf. (VIS'01), pp.263–270, Oct. 2001.

[4] D. Nagayasu, F. Ino, and K. Hagihara, "A decompression pipeline for accelerating out-of-core volume rendering of time-varying data," Computers and Graphics, vol.32, no.3, pp.350–362, June 2008.

[5] P.S. Calhoun, B.S. Kuszyk, D.G. Heath, J.C. Carley, and M.E.K. Fishman, "Three-dimensional volume rendering of spiral ct data: Theory and method," Radiographics, vol.19, no.3, pp.745–764, May 1999.

[6] A. Takeuchi, F. Ino, and K. Hagihara, "An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors," Parallel Computing, vol.29, no.11/12, pp.1745–1762, Nov. 2003.

[7] M. Levoy, "Display of surfaces from volume data," IEEE Computer Graphics and Applications, vol.8, no.3, pp.29–37, May 1988.

[8] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in GPU-based large-scale volume visualization," Computer Graphics Forum, vol.xx, no.xx, pp.xx–xx, Sept. 2015.

[9] NVIDIA Corporation, "NVIDIA GeForce GTX 980," Nov. 2014.

[10] NVIDIA Corporation, "CUDA C Programming Guide Version 7.0," March 2015.

[11] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution volume visualization with a texture-based octree," The Visual Computer, vol.17, no.3, pp.185–197, May 2001.

[12] J. Krüger and R. Westermann, "Acceleration techniques for GPU-based volume rendering," Proc. 14th IEEE Visualization Conf. (VIS'03), pp.287–292, Oct. 2003.

[13] 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.

[14] J. Wang, F. Yang, and Y. Cao, "Cache-aware sampling strategies for texture-based ray casting on GPU," Proc. 4th IEEE Symp. Large Data Analysis and Visualization (LDAV'14), pp.19–26, Nov. 2014.

[15] M. Harris, "Optimizing CUDA," SC'07 Tutorial, Nov. 2007. `http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf`.

[16] D. Weiskopf, M. Weiler, and T. Ertl, "Maintaining constant frame rates in 3D texture-based volume rendering," Proc. 21st Computer Graphics Int'l (CGI'04), pp.604–607, June 2004.

[17] J. Wang, F. Yang, and Y. Cao, "Cache-aware sampling strategies for texture-based ray casting on GPU," Proc. 8th IEEE Pacific Visualization Symp. (PacificVis'15), April 2015.

[18] D. Jönsson, P. Ganestam, A. Ynnerman, M. Doggett, and T. Ropinski, "Explicit cache management for volume raycasting on parallel architectures," Proc. 12th Eurographics Symp. Parallel Graphics and Visualization (EGPGV'12), pp.31–40, May 2012.

[19] E. Catmull and R. Rom, A Class of Local Interpolating Splines, pp.317–326, Academic Press, New York, NY, 1974.

[20] S. Lee, G. Wolberg, and S.Y. Shin, "Scattered data interpolation with multilevel B-splines," IEEE Trans. Visualization and Computer Graphics, vol.3, no.3, pp.228–244, July 1997.

[21] J. Mensmann, T. Ropinski, and K. Hinrichs, "An advanced volume raycasting technique using GPU stream processing," Proc. 5th Int'l Conf. Computer Graphics Theory and Applications (GRAPP'10), pp.190–198, May 2010.

[22] 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.

[23] 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.

[24] Y. Lu, F. Ino, and K. Hagihara, "Cache-aware GPU optimization for out-of-core cone beam CT reconstruction of high-resolution volumes," IEICE Trans. Information and Systems, vol.E99-D, no.12, pp.xx–xx, Dec. 2016.

[25] G.M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," tech. rep., IBM Ltd, Ottawa, Ontario, Aug. 1966.

[26] M. Levoy, "Efficient ray tracing of volume data," ACM Trans. Graphics, vol.9, no.3, pp.245–261, July 1990.

[27] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, "Smart hardware-accelerated volume rendering," Proc. 5th Eurographics/IEEE TCVG Symp. Visualization (VisSym'03), pp.231–238, May 2003.

[28] NVIDIA Corporation, "Profiler User's Guide Version 7.0," March 2015. `http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf`.

[29] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern GPUs," Proc. 11th IFIP Int'l Conf. Network and Parallel Computing (NPC'14), pp.144–156, Sept. 2014.

[30] X. Mei, K. Zhao, C. Liu, and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," IEEE Trans. Parallel and Distributed Systems, `http://arxiv.org/pdf/1509.02308.pdf`.

Yuji MISAKI received the B.E. degree in information and computer sciences from Osaka University, Osaka, Japan, in 2015. He is currently working toward the M.E. degree at Osaka University. His current research interests include computer graphics and high performance computing.

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.