An Acceleration Method for GPU-Based Volume Rendering by Localizing Texture Memory Reference

Yuki Sugimoto[1] Fumihiko Ino[1] Kenichi Hagihara[1]

Abstract: This paper presents a cache-aware method for accelerating texture-based volume rendering on the graphics processing unit (GPU). Because GPUs have a hierarchical architecture in terms of processing and memory units, cache optimization is important to maximize their effective performance for this kind of memory-intensive applications. To accomplish this, our method localizes texture memory reference according to the location of the viewpoint. The key idea for this localization is to dynamically select the width and height of thread blocks (TBs) such that each warp, which is a series of 32 threads simultaneously processed on the GPU, can minimize the stride of memory access. We also incorporate transposed indexing of threads to perform TB-level cache optimization for specific viewpoints. Furthermore, we maximize the TB size so that the spatial locality can be exploited with less active TBs. For relatively large stride, we synchronize threads of the same TB at regular intervals to realize synchronous ray propagation. In experiments using a GeForce GTX 580 card, we find that our cache-aware method doubles the worst rendering performance, as compared with the original implementation provided by the CUDA and OpenCL software development kits (SDKs).

Keywords: volume rendering, CUDA, cache optimization, texture memory

1. Introduction

Volume rendering [1] is a visualization technique for intuitive understanding of three-dimensional (3-D) objects. For example, this technique helps us in performing clinical diagnosis from computed tomography (CT) images [2], [3] and in understanding large-scale simulation results related to computational fluid dynamics [4], [5].

In order to generate such helpful visualization, voxel values of the target volume are accumulated into pixel values on the screen. In more detail, a ray is generated from the viewpoint to each pixel, and then values of penetrated voxels are sampled at regular intervals along the ray for accumulation. Thus, the accumulation is accomplished from 3-D space to 2-D space. Volume rendering is a memory-intensive application rather than a compute-intensive application, because voxel values can be reused only within neighboring rays. Consequently, achieving efficient memory access is essential to attain high rendering performance.

To deal with this large amount of memory access, many renderers [4], [6], [7] were implemented using the graphics processing unit (GPU) [8], which is an accelerator for graphics applications. The memory bandwidth of the GPU is an order of magnitude higher than that of the CPU: the bandwidth reaches 192.4 GB/s on a GeForce GTX 680 card, whereas it remains 25.6 GB/s on a Core i7 3770K processor. Furthermore, GPU architecture is capable of running thousands of lightweight threads in parallel, which are useful to hide memory latency with data-independent computation. Using this accelerator, the accumulation procedure can be easily parallelized because it does not have data dependence between different rays (i.e., different pixels). The volume data is typically loaded as a 3-D texture to interpolate voxel values by taking advantage of texture mapping hardware of the GPU [9]. This special hardware has a cache mechanism to reduce the latency of data access for acceleration. Consequently, the rendering performance can be increased by maximizing the locality of reference.

In this paper, we present a cache-aware method for increasing the frame rate of texture-based volume rendering. To achieve this, our method maximizes the locality of reference by dynamically selecting the width w and height h of *thread blocks* (TBs) so that a group of threads called *warp* [10] can access data with a small stride. Because threads in the same warp are simultaneously processed on the GPU, such parallel threads have to maximize the locality of reference. The selection of the TB shape $w \times h$ can be determined according to the geometrical relationship between the viewpoint and the volume axes, because the physical stride between two adjacent voxels depends on the volume axis they are parallel. In addition to this warp-level optimization, our method performs TB-level optimization for specific viewpoints. Our method currently works with the compute unified device architecture (CUDA) [10] and OpenCL [11].

2. Related Work

Krüger *et al.* [6] presented the impact of optimization techniques such as early ray termination and empty space skipping [12] on the GPU. Using these techniques, the rendering performance is increased by a factor of 3. A similar technique is presented by Rijters *et al.* [7], who employ an octree data structure

[1] Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565-0871, Japan

on the GPU. These techniques intend to reduce the amount of data access while our optimization strategy reduces the latency of data access.

An optimization strategy is presented by Ryoo *et al.* [13] for CUDA applications. Their strategy investigates the number of resident TBs to evaluate resource utilization. The TB size wh is optimized using this metric but the TB shape $w \times h$ is not investigated for further optimization.

Liu *et al.* [14] presented an optimization framework capable of empirically searching for the best optimizations for GPU applications. Using their framework, we can easily find the best shape of TBs in terms of the performance. In contrast to this empirical approach, our approach gives insight into the relationship between the data locality and the memory access pattern. According to our insight, we can prune the search space in terms of the TB shape, which contributes to reduce the overhead of run-time optimization.

3. GPU-based Volume Rendering

In this section, we present an overview of CUDA and a well-known ray casting algorithm [15]. We then explain how this algorithm can be typically implemented using texture mapping hardware.

3.1 Compute Unified Device Architecture (CUDA)

A CUDA-compatible GPU [10] consists of hundreds of CUDA cores structured in a hierarchy. This hardware has tens of streaming multiprocessors (SMs), each containing 8 or 32 CUDA cores depending on its generation. Using these cores, thousands of threads are executed in a single-instruction, multiple-thread (SIMT) fashion [10].

Threads are classified into data independent groups, namely TBs, which are then assigned to a SM in a cyclic manner until they exhaust available resources such as register files. Such data independent TBs contribute to have more flexibility for efficient scheduling of threads, so that more TBs should be resided and processed together on the SM to achieve efficient overlap of memory operations and arithmetic instructions. Each resident TB is further broken into groups of 32 consecutive threads called warps. A warp is the minimum scheduling unit managed by the SM. The execution order of warps is dynamically determined by the warp scheduler, which cannot be controlled by the program.

Threads usually form a 2-D TB, where they can be identified with a 2-D index. The TB shape $w \times h$ can be specified by an argument to the kernel function, which runs on the GPU for acceleration. In contrast, the warp shape $p \times q$ cannot be directly specified in the program, where p and q represent the width and the height of warps, respectively. However, the warp shape is automatically determined by its enclosing TB, because threads in a warp belong to the same TB and have consecutive indexes.

This implies that the warp shape $p \times q$ can be specified indirectly through the TB shape $w \times h$. Table 1 shows the correspondence between the TB shape and the warp shape when $wh = 256$. This table indicates that horizontal warps (i.e., $p > q$) are generated if $w \geq 8$. Otherwise, vertical warps (i.e., $p < q$) are generated.

Table 1 Relationship between TB shape $w \times h$ and warp shape $p \times q$. Values are presented for the TB size wh of 256. Horizontal warps (i.e., $p > q$) are generated if $w \geq 8$. Otherwise, vertical warps are generated.

TB shape $w \times h$	Warp shape $p \times q$	Aspect ratio of warp
1×256	1×32	1 : 32
2×128	2×16	1 : 8
4×64	4×8	1 : 2
8×32	8×4	2 : 1
16×16	16×2	8 : 1
32×8	32×1	32 : 1
64×4	32×1	32 : 1
128×2	32×1	32 : 1
256×1	32×1	32 : 1

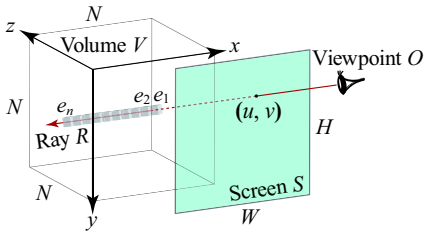


Fig. 1 Geometry of ray casting. Pixel values are computed by accumulating color and opacity values of voxels penetrated by a ray from the viewpoint.

3.2 Ray Casting

Figure 1 illustrates the geometry used for ray casting [15]. Let V be the volume to be rendered from the viewpoint O. We consider a cubic volume of $N \times N \times N$ voxels, where N represents the volume size. We assume that each voxel has a scalar data associated with color and opacity values. Let x, y, and z be elements of the voxel coordinates.

The ray casting technique casts a ray from the viewpoint O to every pixel (u, v) on the screen S, where $1 \leq u \leq W$ and $1 \leq v \leq H$. W and H here represent the width and the height of the screen S. The value $S(u, v)$ of pixel (u, v) is then computed by accumulating color and opacity values of penetrated voxels. This accumulation is done at regular intervals along the ray in front-to-back order:

$$S(u, v) = \sum_{i=1}^{n} \left(\alpha(e_i) c(e_i) \prod_{j=0}^{i-1} \left(1 - \alpha(e_j)\right) \right), \quad (1)$$

where e_i represents the i-th voxel penetrated by the ray, n represents the number of penetrated voxels, $c(e_i)$ and $\alpha(e_i)$ represent the color and the opacity of voxel e_i, respectively, and $\alpha(e_0) = 0$.

3.3 Texture-based Rendering with CUDA

Eq. (1) indicates that different pixel values can be computed in parallel because there is no data dependence between them. Consequently, the computation of a pixel is assigned to a thread in typical renderers. A screen of $W \times H$ pixels can then be rendered by WH threads, which compose $\lceil W/w \rceil \times \lceil H/h \rceil$ TBs. Using this parallel scheme, voxels are accessed in front-to-back order.

Since rays do not always penetrate the center of voxels, voxel values must be interpolated before accumulation. To accelerate this interpolation, many implementations employ texture-based

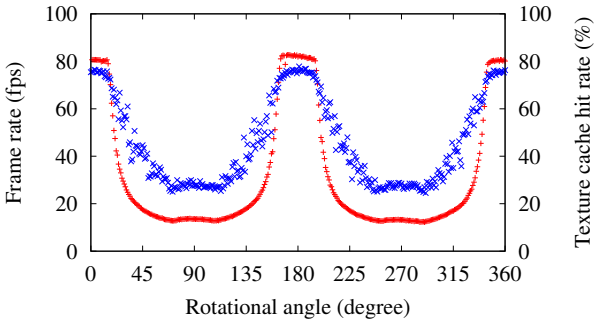


Fig. 2 Relationship between the frame rate and the hit rate of texture cache. These results were obtained with a fullerene dataset. The horizontal axis represents the rotational angle around the y-axis. Red and blue lines show the frame rate and the hit rate, respectively.

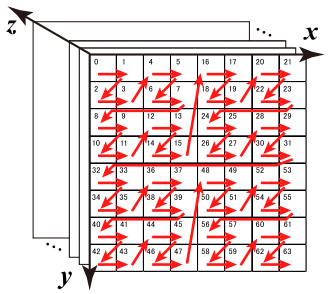


Fig. 3 Organization of a 3-D texture in CUDA. A 3-D texture consists of a bunch of 2-D slices optimized for 2-D spatial locality via Morton curve [18]. A series of red arrows represents the sequence of physical memory address in a 2-D slice. The physical address is shown in each texel's upper left corner.

rendering [9], which performs interpolation using texture mapping hardware of the GPU. Thus, the volume is accessed via a 3-D texture to take advantage of hardware accelerated interpolation. Viewpoint movement is usually realized by rotating the 3-D texture with a fixed viewpoint. Let θ_x, θ_y, and θ_z be the rotational angles around the x-, y-, and z-axes, respectively. An arbitrary viewpoint then can be specified by $(\theta_x, \theta_y, \theta_z)$, where $0 \le \theta_x, \theta_y, \theta_z < 360$.

Figure 2 shows our preliminary evaluation results obtained with a CUDA software development kit (SDK) renderer [10]. The cache hit rate was measured by CUDA Visual Profiler, which provides profiling results on a SM. As shown in this figure, the hit rate of texture cache mainly determines the rendering frame rate. These results motivated us to tackle the issue of cache optimization for fast visualization.

4. Texture Memory Organization

Figure 3 shows how a logical address space is mapped onto a physical memory space in a 3-D texture [16], [17]. As shown in this figure, a 3-D texture consists of a bunch of 2-D slices. Each slice is further optimized for 2-D spatial locality via Morton's z-order curve [18], as illustrated in a sequence of red arrows in Fig. 3. Morton curve has a recursive hierarchy, so that a z-ordered block at the l-th level of hierarchy contains a 2-D slice of $2^l \times 2^l$ texels, where $1 \le l \le \lceil \log N \rceil$ (see Fig. 4). For simplicity, we assume N being a power of two (i.e., $N = 2^l$) in the following discussion.

Although Morton curve is optimized for 2-D spatial locality,

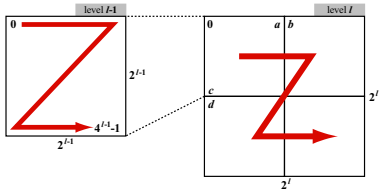


Fig. 4 Hierarchical structure of Morton curve. A block at the l-th level contains four internal blocks of the $(l-1)$-th level. The maximum stride appears between these internal blocks: between texels a and b along the horizontal axis, and between texels c and d along the vertical axis. The physical index of texels b and d are 4^{l-1} and $2 \cdot 4^{l-1}$, respectively. The physical index of texels a and c are $\sum_{k=0}^{l-2} 4^k$ and $2 \cdot \sum_{k=0}^{l-2} 4^k$, respectively.

the physical stride between two adjacent voxels is not uniform in this data structure. Here after we investigate the physical stride in more detail. Suppose that two adjacent voxels e_i and e_{i+1} are accessed during rendering. The stride between these voxels can then be classified into two groups depending on their coordinates:

(1) The adjacent voxels e_i and e_{i+1} have different z. In this case, e_i and e_{i+1} exist on two adjacent 2-D slices. These voxels can be accessed with a stride of N^2, because they have the same x and y.

(2) The adjacent voxels have different x or y. In these cases, voxels e_i and e_{i+1} exist on the same slice. The stride between them varies according to the axis they are parallel. For example, the strides in Fig. 3 range from 1 to 11 if e_i and e_{i+1} are parallel to the x-axis. However, the maximum stride at the l-th level of hierarchy appears between adjacent blocks of the $(l-1)$-th level, as shown in Fig. 4. The maximum stride along the x-axis can be given by $(2 \cdot 4^{l-1} + 1)/6$ while that along the y-axis can be given by $(2 \cdot 4^{l-1} + 1)/3$. Since $N = 2^l$, voxels along the x-axis and the y-axis can be accessed with a stride of $(N^2 + 2)/6$ and that of $(N^2 + 2)/3$, respectively.

In summary, the x-axis, y-axis, and z-axis have a different stride between adjacent voxels, and their ratio can be approximated by $1 : 2 : 6$. Therefore, it is better to access voxels along the x-axis in order to achieve more cache hits.

5. Proposed Method

Our cache-aware method, which primarily minimizes the stride of memory access for warps, consists of four strategies as follows:

(1) Dynamic selection of the TB shape.
(2) Transposed indexing of threads.
(3) TB size maximization.
(4) Synchronous ray propagation.

The first strategy aims at performing warp-level optimization while the remaining strategies are responsible for TB-level optimization. These strategies are dynamically activated according to the location of the viewpoint (i.e., the rotational angles). Table 2 summarizes the relationship between the activated strategies and the rotational angles. Note that this table shows the relationship for $0 \le \theta_x, \theta_y, \theta_z \le 90$, but it can be extended to other rotational angles by using the symmetricity of geometry.

Table 2 Relationship between the activated strategies and rotational angles. An empty cell corresponds to switched off.

Rotational axis	Strategy	$\theta_x, \theta_y, \theta_z$: rotational angle (degree)					
		0–15	15–30	30–45	45–60	60–75	75–90
x	(1)	32×1	32×1	32×1	32×1	32×1	32×1
	(2)						
	(3)						
	(4)						
y	(1)	32×1	16×2	8×4	4×8	2×16	1×32
	(2)				On	On	On
	(3)				On	On	On
	(4)						
z	(1)	32×1	16×2	8×4	4×8	2×16	1×32
	(2)				On	On	On
	(3)						
	(4)	On	On	On	On	On	On

5.1 Dynamic Selection of TB shape

To maximize the locality of reference, our method focuses on three points as follows:

(1) The SM processes threads in the same warp at the same time.
(2) Each volume axis has a different stride between adjacent voxels, as we analyzed in Section 4.
(3) The warp shape $p \times q$ is determined by the TB shape $w \times h$, as we mentioned in Section 3.1.

The first point motivates us to optimize the memory access pattern of a warp rather than that of a thread. This point is a unique feature owing to highly-threaded GPU architecture. On earlier acceleration systems such as cluster systems [3], [19], optimization is successfully done for a single process (i.e., a single ray), because they are based on distributed memory architecture. In contrast, we emphasize optimization of a warp (i.e., a ray frustum) as a key acceleration strategy for the GPU. Thus, we must investigate the memory access pattern that can be caused by a warp. Because voxels are sampled at regular intervals from the viewpoint, a warp accesses voxels on the surface of a sphere. For simplicity, we assume that this spherical surface can be approximated with a plane. Under this approximation, a warp accesses voxels on a plane that are parallel to the screen.

With respect to the second point, voxels should be always accessed along the x-axis, which has the smallest stride among the volume axes. However, this is not a practical solution because the volume can be rendered from an arbitrary viewpoint, as shown in Fig. 5. Consequently, the volume axes have different appearance on the screen, and thus, the x-axis can be parallel to one of the horizontal, vertical, and depth directions. Therefore, we determined to give priority to the volume axes: voxels should be accessed in the order of x, y, and z to have smaller strides. To realize this prioritization for warps, we optimize the warp shape.

The third point plays the key role in realizing the prioritized access mentioned above. The warp size $p \times q$ must be selected such that each warp can access voxels in the order of x, y, and z. For example, horizontal warps are better than vertical warps if the primary axis with a smaller stride appears as a horizontal line on the screen, as shown in Fig. 5(a) and 5(b). In these cases, horizontal warps are allowed to access voxels with smaller strides than vertical warps.

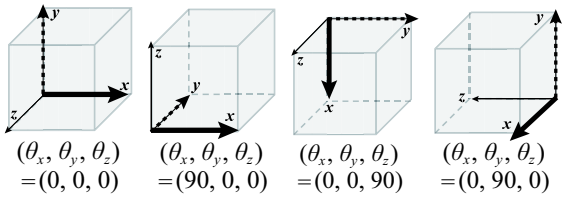


Fig. 5 Geometrical relationship between the viewpoint and the volume axes. For simplicity, we consider here four representative viewpoints (a)–(d), which make two of the volume axes parallel to the screen axes. The x-axis and the z-axis have the smallest stride and the largest stride among the volume axes, respectively.

According to the three points mentioned above, we determined the TB shape $w \times h$ for an arbitrary rotational angle $(\theta_x, \theta_y, \theta_z)$ (Table 2). Given $(\theta_x, \theta_y, \theta_z)$, the TB shape $w \times h$ is selected in the following three steps:

(1) Parallel plane detection. Our method detects the plane most parallel to the screen. For example, the xy-plane is such a parallel plane in Figs. 5(a) and 5(c).
(2) Primary axis detection. The primary axis with a smaller stride is selected from two axes that compose the parallel plane. For example, the parallel plane in Fig. 5(d) is the yz-plane, and thus, we select the y-axis as the primary axis.
(3) TB shape selection. The TB shape is selected according to the direction of the primary axis rendered on the screen. Vertical and horizontal warps are selected if the primary axis is rendered in a vertical line and in a horizontal line on the screen, respectively. For example, the TB shape of 1×256 is selected for viewpoints in Figs. 5(c) and 5(d), because the primary axis is rendered in a vertical line from these viewpoints. For other oblique lines, we use a hybrid of vertical and horizontal warps. To achieve this, we classify the rotational domain $0 \leq \theta_x, \theta_y, \theta_z \leq 90$ into six groups (Table 2), because the warp shape $p \times q$ can be one of the six configurations (Table 1) and a vertical line turns to be a horizontal line after 90 degree rotation.

With respect to the warp shape of 32×1, there are four candidates for the TB shape in Table 1: $w \times h = 32 \times 8$, 64×4, 128×2, and 256×1. Among these candidates, we decided to use $w \times h = 32 \times 8$, according to our preliminary evaluation results. The highest performance was obtained for this shape, because textures are optimized for 2-D spatial locality, as we mentioned in Section 4. From this point of view, horizontal warps in a TB should be placed vertically rather than horizontally.

5.2 Transposed Indexing of Threads

In contrast to the dynamic selection of the TB shape, which performs warp-level optimization, the remaining strategies perform TB-level optimization. The transposed indexing of threads is activated when the primary axis is parallel to the vertical direction (Table 2). This strategy intends to place a series of TBs as vertical as possible, and thus, SMs are allowed to access voxel along the primary axis with a smaller stride.

Figure 6 shows an example of transposed indexing. As shown in Fig. 6(b), our method chooses vertical TBs if the primary axis appear in vertical. However, a series of vertical TBs is placed hor-

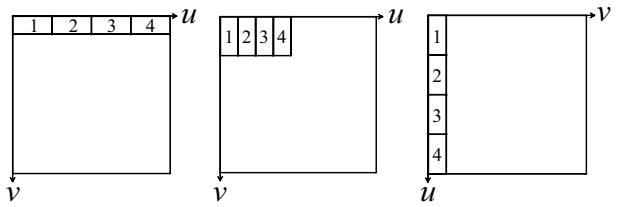


Fig. 6 Transposed indexing of threads. (a) Horizontal TBs, (b) vertical TBs without transpose, and (c) vertical TBs with transpose. Because TB assignment is done along the u-axis in a cyclic manner, (b) a naive assignment of vertical TBs is not symmetric to (a) that of horizontal TBs. In contrast, our transposed indexing exchanges the u- and v-axes so that its assignment is symmetric to (a).

izontally in the original indexing, which are then assigned to SMs in a cyclic manner. Consequently, SMs access different columns simultaneously, failing to exploit the data locality along the primary axis.

To solve this problem, we apply a transpose operation to thread indexing by exchanging the (u, v) coordinate with the (v, u) coordinate. Owing to this transposed geometry, a series of vertical TBs is placed vertically as shown in Fig. 6(c). As a result, this execution configuration is symmetric to Fig. 6(a), which is selected if the primary axis is parallel to the horizontal direction.

5.3 Thread Block Size Maximization

As we mentioned in Section 3.1, each SM usually executes several TBs concurrently. Concurrent TBs are useful to maximize the effect of memory access hiding if they have high locality with high cache hit rate. However, this does not apply if concurrent TBs access a wide range of memory addresses with poor locality. In this case, it is better to maximize the TB size wh to reduce the number of concurrent TBs. In addition, the TB size wh must be a multiple of the warp size (i.e., 32) to avoid CUDA cores from being idle during SIMT execution.

With respect to our renderer, we found that the latter case appears when the x-axis is parallel to the depth direction. Such a situation forces threads to access along the y-axis, so that warps are required to tolerate relatively large strides. To make the matter worse, concurrent TBs are usually responsible for separated region on the screen, mainly due to the cyclic task distribution mentioned in Section 3.1. Consequently, we decided to use a TB size wh of 512, which is the largest configuration that runs on our experimental GPU.

5.4 Synchronous Ray Propagation

Owing the nature of SIMT execution, threads can execute different lines of the GPU code. This implies that rays proceed independently though they are assigned to the same TB. Synchronous ray propagation intends to minimize the gap in the depth direction between propagating rays. To achieve this, we simply perform synchronization by calling `__syncthreads()` at every loop iteration (i.e., at every sampling step).

Because this strategy incurs synchronization overhead, our method activates the strategy when the xy-plane is parallel to the screen. In this case, rays are parallel to the z-axis, which has the largest stride. Consequently, the overhead can be ignored as compared with the benefit of cache utilization.

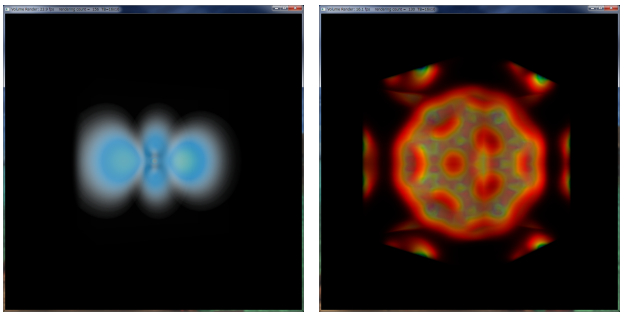


Fig. 7 Experimental datasets. (a) The spatial probability distribution of the electron in an hydrogen atom. (b) a fullerence called buckyball.

6. Experimental Results

To evaluate our cache-aware method in terms of the rendering performance, we applied our method to renderers included in CUDA and OpenCL SDKs [20], [21]. The original renderers use a 1-D texture to store a color map table, which associates color and opacity values with each voxel. Because reference to this table results in perturbation of cache behavior. Therefore, we modified the code to store the table in shared memory. Thus, the modified code uses textures only for the volume data. We also modified the OpenCL-based renderer because it employs a back-to-front algorithm. We not only replaced the algorithm with a front-to-back algorithm but also adopted early ray termination [6] for acceleration. The original versions use a fixed TB shape $w \times h = 16 \times 16$ (i.e., $p \times q = 16 \times 2$) for arbitrary viewpoints.

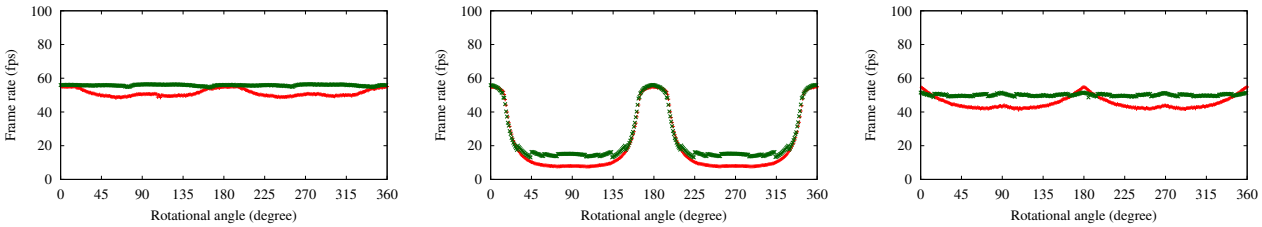
We used two datasets, fullerene and atom, shown in Fig. 7. The atom dataset can be regarded as a transparent dataset, whereas the fullerene dataset can be regarded as an opaque dataset. Both datasets consist of 8-bit data and are resized to $N = 1024$. These datasets were rendered on a screen of size $W = H = 1024$.

For experiments, we used a desktop PC equipped with a GeForce 580 GTX card. Our machine runs with Windows 7, CUDA 4.2, OpenCL 1.1, and graphics driver 306.97.

Figure 8 shows the frame rates of the atom dataset with our dynamic method and the original static method. During measurement, we rotated the volume around the x-, y-, then z-axes. Here, the width p of warps to select for viewpoints is shown in Fig. 9.

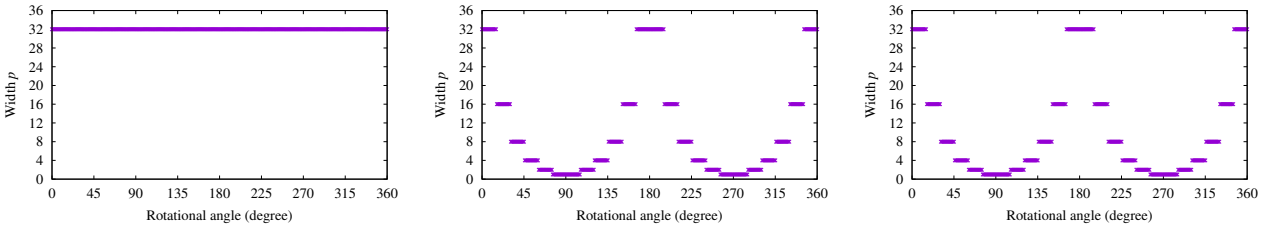
Firstly, for the x-axis rotation, the parallel plane can be the xy-plane or the xz-plane (Figs. 5(a) and 5(b)). As shown in Fig. 8(a), the static method show relatively high frame rates, but slightly decreases the frame rate for rotational angles except around $\theta_x = 0$ and 180. In contrast, our method successfully eliminates such performance degradation by using horizontal warps for all θ_x (Fig. 9(a)). As shown in Figs. 5(a) and 5(b), the performance degradation is due to the z-axis, which becomes parallel to the vertical direction at $\theta_x = 90$ and 270. Because the z-axis has the largest stride, it is better to use horizontal 32×1 warps for this rotation.

For the y-axis rotation, on the other hand, the parallel plane can be the xy-plane or the yz-plane (Figs. 5(a) and 5(d)). The frame rates in Fig. 8(b) show a quite different behavior as compared with



(a) Rotation around the x-axis (b) Rotation around the y-axis (c) Rotation around the z-axis

Fig. 8 Frame rates of our dynamic method and the original static method. The atom dataset was used with CUDA. Green and red plots correspond to our method and the original method, respectively.



(a) Rotation around the x-axis (b) Rotation around the y-axis (c) Rotation around the z-axis

Fig. 9 The width p of warps selected for viewpoints. The height is given by $q = 32/p$. The original static method uses $p = 16$ for arbitrary viewpoints.

those in Fig. 8(a). Both the static method and our method drops the frame rate when $30 \leq \theta_y \leq 150$ and $210 \leq \theta_y \leq 330$, but our method increases the worst frame rate from 7.9 fps to 15.2 fps, owing to the switch to vertical warps (Fig. 9(b)). These significant drops are due to the z-axis again, which appears in vertical when $\theta_y = 90$ and 270 (Fig. 5(d)). At these rotational angles, the x-axis is parallel to the depth direction. Consequently, warps cannot exploit the highest locality, even though they change their shape. When $\theta_y = 90$, our dynamic selection strategy increases the frame rate from 7.9 fps to 11.0 fps, which is then increased to 12.6 fps by our transposed indexing strategy.

Finally, for the z-axis rotation, the parallel plane can be the xy-plane (Figs. 5(a) and 5(c)). The frame rates in Fig. 8(c) are similar to those in Fig. 8(a), but our method failed to outperform the static method at $\theta_z = 0$ and 180. This is due to the overhead of thread synchronization. As we mentioned in Section 5.4, we activate this strategy when the z-axis is parallel to the depth direction. If we switched off this strategy at these angles, the frame rate was increased from 51.6 fps to 54.3 fps, which is close to that of the static method, namely 54.8 fps. However, he frame rates at other angles dropped in contrast to $\theta_z = 0$ and 180. Consequently, further investigation must be needed to use this strategy.

We next analyzed the frame rates for the fullerence dataset, which can be regarded as an opaque dataset as compared with the atom dataset. Figure 10 shows the measured results for the x-, y-. and z-axis rotations.

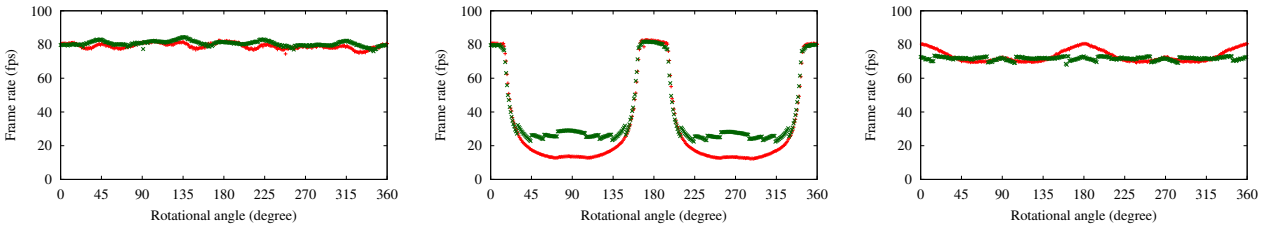
In comparison with Fig. 8, the frame rates of the static method were increased by roughly 20 fps, owing to early ray termination. That is, this opaque dataset allows rays to be terminated earlier than the transparent atom dataset. However, this increase cannot be clearly seen when $30 \leq \theta_y \leq 150$ and $210 \leq \theta_y \leq 330$ in Fig. 10(b), where the frame rates are less than 20 fps. In contrast, our dynamic method successfully increases the frame rates around 30 fps in Fig. 10(b). Especially, our method improves degraded frame rate from 13.0 fps to 29.0 fps when $\theta_y = 90$ and gains the best speedup of a factor of 2.2. This indicates that the impact of cache optimization is larger than that of visibility-based optimization such as early ray termination. Furthermore, the achieved frame rates around 30 fps are higher than those obtained with the opaque dataset (i.e., approximately 20 fps in Fig. 8(b)). Consequently, we think that our cache-aware method cooperates well with the visibility-based method, leading to better utilization of texture cache.

In Fig. 10(c), we again observed performance decreases due to the synchronization overhead. By contrast to the results of the transparent dataset in Fig. 8(c), the decreases can be found in many rotational angles around $\theta_z = 0$ and 180. This is due to shortly terminated rays, because it contributes to reduce the gap in the depth direction between propagating rays. The reduced gap means that rays are naturally synchronized each other, and thus, the benefit of explicit synchronization is minimized when using the visibility-based optimization.

Finally, we evaluated our method using an OpenCL-based renderer. Figure 11 show the frame rates for the atom dataset. As compared with the CUDA-based results in Fig. 8, the frame rates were reduced by roughly 33%. This lower performance is probably due to the driver, which is not optimized well for OpenCL. Despite this lower performance, the performance characteristics in Fig. 11 is similar to those in Fig. 8. Actually, our method successfully increased the frame rates for the y-axis rotation. However, the synchronization overhead again reduced the frame rates for the z-axis rotation.

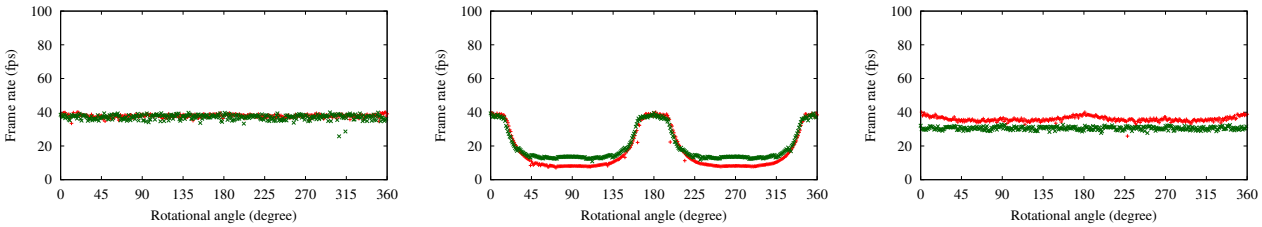
7. Conclusion

In this paper, we presented an acceleration method for texture-based volume rendering on the GPU. Our method increases the hit rate of texture cache by choosing the shape of TBs during rendering. This dynamic selection focuses on the geometrical rela-



(a) Rotation around the x-axis (b) Rotation around the y-axis (c) Rotation around the z-axis

Fig. 10 Frame rates of our dynamic method and the original static method. The fullerence dataset was used with CUDA. Green and red plots correspond to our method and the original method, respectively.



(a) Rotation around the x-axis (b) Rotation around the y-axis (c) Rotation around the z-axis

Fig. 11 Frame rates of our dynamic method and the original static method. The atom dataset was used with OpenCL. Green and red plots correspond to our method and the original method, respectively.

tionship between the viewpoint and the volume axes. Our method determines the TB shape such that threads in the same warp can have a small stride of memory access. Such a small stride can be obtained if each warp accesses consecutive voxels along the x-axis. In addition to this warp-level optimization, our method activates TB-level optimization for some viewpoints.

In experiments, we compared our method with a naive method that uses a fixed shape of TBs. We found that our dynamic method increases the frame rates for the y-axis rotation, where the rendering performance is relatively lower than other viewpoints. Moreover, our cache-related method efficiently works with a visibility-based method, demonstrating an efficient utilization of texture cache particularly for an opaque dataset. Owing to our cache utilization, the worst frame rates were increased from 13.0 fps to 29.0 fps, achieving the best speedup of a factor of 2.2.

Future work includes abstraction of our cache-aware method for other memory-intensive applications.

Acknowledgments This study was partly supported by the JSPS KAKENHI Grant Number 23300007, 23700057, and the JST CREST program "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." The atom dataset is available at http://volvis.org/ and is courtesy of SFB 382 of the German Research Council.

References

[1] Drebin, R. A., Carpenter, L. and Hanrahan, P.: Volume Rendering, *Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 3, pp. 65–74 (1988).
[2] Ney, D. R., Fishman, E. K., Magid, D. and Drebin, R. A.: Volumetric Rendering of Computed Tomography Data: Principles and Techniques, *IEEE Computer Graphics and Applications*, Vol. 10, No. 2, pp. 24–32 (1990).
[3] Takeuchi, A., Ino, F. and Hagihara, K.: An Improved Binary-Swap Compositing for Sort-Last Parallel Rendering on Distributed Memory Multiprocessors, *Parallel Computing*, Vol. 29, No. 11/12, pp. 1745–1762 (2003).
[4] Nagayasu, D., Ino, F. and Hagihara, K.: A Decompression Pipeline for Accelerating Out-of-Core Volume Rendering of Time-Varying Data, *Computers and Graphics*, Vol. 32, No. 3, pp. 350–362 (2008).
[5] Uselton, S. P.: Volume Rendering for Computational Fluid Dynamics: Initial Results, Technical Report RNR-91-026, Nasa Ames Research Center (1991).
[6] Krüger, J. and Westermann, R.: Acceleration Techniques for GPU-based Volume Rendering, *Proc. 14th IEEE Visualization Conf. (VIS'03)*, pp. 287–292 (2003).
[7] Rijters, D. and Vilanova, A.: Optimizing GPU Volume Rendering, *J. WSCG*, Vol. 14, No. 1/3, pp. 9–16 (2006).
[8] Luebke, D. and Humphreys, G.: How GPUs Work, *Computer*, Vol. 40, No. 2, pp. 96–100 (2007).
[9] Hibbard, W. and Santek, D.: Interactivity is the Key, *Proc. Chapel Hill Workshop Volume Visualization (VVS '89)*, pp. 39–43 (1989).
[10] NVIDIA Corporation: CUDA Programming Guide Version 4.2 (2012).
[11] Khronos OpenCL Working Group: The OpenCL Specification Version 1.1 (2011). `http://www.khronos.org/registry/cl/`.
[12] Levoy, M.: Efficient Ray Tracing of Volume Data, *ACM Trans. Graphics*, Vol. 9, No. 3, pp. 245–261 (1990).
[13] Ryoo, S., Rodrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, S.-Z., Baghsorkhi, S. S. and mei W. Hwu, W.: Program optimization carving for GPU computing, *J. Parallel and Distributed Computing*, Vol. 68, No. 10, pp. 1389–1401 (2008).
[14] Liu, Y., Zhang, E. Z. and Shen, X.: A Cross-Input Adaptive Framework for GPU Program Optimizations, *Proc. 23th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'09)* (2009). 10 pages (CD-ROM).
[15] Levoy, M.: Display of Surfaces from Volume Data, *IEEE Computer Graphics and Applications*, Vol. 8, No. 3, pp. 29–37 (1988).
[16] Montrym, J. and Moreton, H.: The GeForce 6800, *IEEE Micro*, Vol. 25, No. 2, pp. 41–51 (2005).
[17] Pharr, M. and Fernando, R.(eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA (2005).
[18] Morton, G. M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, Technical report, IBM Ltd, Ottawa, Ontario (1966).
[19] Matsui, M., Ino, F. and Hagihara, K.: Parallel Volume Rendering with Early Ray Termination for Visualizing Large-Scale Datasets, *Proc. 2nd Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'04)*, pp. 245–256 (2004).
[20] NVIDIA Corporation: GPU Computing SDK (2012).
[21] NVIDIA Corporation: OpenCL SDK (2012).