Parallel Computing 40 (2014) 59–69

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Improving cache locality for GPU-based volume rendering

Yuki Sugimoto [a], Fumihiko Ino [b,*], Kenichi Hagihara [b]

[a] Nippon Telegraph and Telephone East Corporation, 19-2, Nishi-shinjuku 3-chome, Shinjuku, Tokyo 163-8019, Japan
[b] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamada-oka, Suita, Osaka 565-0871, Japan

ARTICLE INFO

Article history:
Received 2 August 2013
Received in revised form 14 February 2014
Accepted 24 March 2014
Available online 1 April 2014

Keywords:
Volume rendering
CUDA
Cache optimization
Texture memory

ABSTRACT

We present a cache-aware method for accelerating texture-based volume rendering on a graphics processing unit (GPU). Because a GPU has hierarchical architecture in terms of processing and memory units, cache optimization is important to maximize performance for memory-intensive applications. Our method localizes texture memory reference according to the location of the viewpoint and dynamically selects the width and height of thread blocks (TBs) so that each warp, which is a series of 32 threads processed simultaneously, can minimize memory access strides. We also incorporate transposed indexing of threads to perform TB-level cache optimization for specific viewpoints. Furthermore, we maximize TB size to exploit spatial locality with fewer resident TBs. For viewpoints with relatively large strides, we synchronize threads of the same TB at regular intervals to realize synchronous ray propagation. Experimental results indicate that our cache-aware method doubles the worst rendering performance compared to those provided by the CUDA and OpenCL software development kits.

1. Introduction

Volume rendering [1] is a visualization technique for the intuitive assessment of three-dimensional (3D) objects. For example, volume rendering facilitates the advanced clinical diagnosis of computed tomography (CT) images [2,3] and the improved analysis of large-scale fluid dynamics simulations [4–6].

To generate helpful visualizations, voxel values of the target volume are accumulated into pixel values on the screen. A ray is generated from the viewpoint to each pixel, and then values of the penetrated voxels are sampled at regular intervals along the ray for accumulation. Thus, accumulation is accomplished from 3D space to 2D space. However, volume rendering is memory intensive rather than compute intensive, because voxel values can only be reused within neighboring rays. Consequently, rendering performance is highly dependent on efficient memory access.

To deal with such significant amounts of memory access, many renderers [6–8] have been implemented using a graphics processing unit (GPU) [9], which is an accelerator for graphics applications. The memory bandwidth of a GPU is an order of magnitude higher than that of a CPU; bandwidth can reach 192.4 GB/s on a GeForce GTX 580 card and that for a Core i7 3770K processor can reach 25.6 GB/s. Furthermore, the GPU is capable of running thousands of lightweight threads in parallel, which is useful to compensate memory latency in data-independent computation. By using such an accelerator, the accumulation procedure can be parallelized easily, because it does not have data dependence between different rays (i.e., different pixels). Volume data is typically loaded as 3D texture to interpolate voxel values by taking advantage of the

* Corresponding author. Tel.: +81 6 6879 4353; fax: +81 6 6879 4354.
E-mail address: ino@ist.osaka-u.ac.jp (F. Ino).

http://dx.doi.org/10.1016/j.parco.2014.03.013
0167-8191/

GPU's texture mapping hardware [10]. This special hardware has a cache mechanism that reduces data access latency. Consequently, rendering performance can be increased by maximizing the locality of reference.

We present a cache-aware method to increase the frame rate of texture-based volume rendering. The proposed 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* [11] can access data with a small stride. Because threads in the same warp are simultaneously processed on the GPU, we believe that such parallel threads must 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 volume axes, because the physical stride between two adjacent voxels depends on the volume axis to which 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) [11] and OpenCL [12], which is supported by most GPU architectures. We have extended our preliminary study [13] by (1) realizing TB-level optimization that improves the worst frame rates and (2) collecting additional evaluation results using an OpenCL-based renderer.

The remainder of the paper is organized as follows. Section 2 presents related studies in the area of GPU application optimization. Section 3 describes a typical implementation of texture-based volume rendering, and Section 4 gives an architectural overview of textures. Section 5 describes our run-time method that selects the TB shape for acceleration. Section 6 presents experimental results, and Section 7 provides conclusions and suggestions for future studies.

2. Related work

Many research studies have examined the automatic tuning of TB size wh for specific applications such as fast Fourier transform (FFT) [14], sparse matrix–vector multiplication [15], and stencil computation [16], where w and h represent the width and height of TBs, respectively. Ryoo et al. [17] presented a tuning strategy that optimizes the TB size wh. Their strategy evaluates resource utilization to investigate the number of resident TBs that run concurrently on the GPU. Although the TB size wh was optimized using their strategy, the TB shape $w \times h$ was not investigated for optimization. We primarily focus on the TB shape $w \times h$ to improve cache locality for volume rendering.

Torres et al. [18] presented performance analysis results that are useful for selecting the TB size wh and shape $w \times h$. They studied how global memory access performance varies depending on the TB size and shape. According to this study, they concluded that some performance criteria such as memory access pattern and total workload per thread helps to choose a TB geometry. They also further developed a collection of micro-benchmarks (uBench) [19], which provide useful performance information relative to TB size and shape for different architectures and applications. On the other hand, we clarify the critical factors that determine the best TB shape for texture-based volume renderers. According to our findings, we further realize an auto-tuning technique that dynamically tunes the TB shape for a given viewpoint.

Liu et al. [20] presented an optimization framework capable of empirically searching for the best optimizations for GPU applications. Their framework allows the easy determination of the best TB shape $w \times h$ in terms of performance. Similar auto-tuning strategies have been independently presented by Du et al. [21] and by Brodtkorb et al. [22]. In contrast to these empirical approaches, our approach yields insight into the relationship between data locality and memory access pattern. According to this insight, we can prune search space in terms of TB shape, and thus reduce run-time optimization overhead.

Zheng and Muller [23] investigated a cache-aware scheme for CT reconstruction, in which they analyze the area of regions where values are written by the GPU. According to this analysis, their scheme switches the data format from row- to column-major order. However, format conversion is not a practical solution for volume renderers, because the volume can be rendered from arbitrary viewpoints. A similar cache-aware scheme was also presented by Okitsu et al. [24].

Govindaraju and Manocha [25] presented cache-efficient algorithms for sorting, FFT, and matrix multiplication problems. These algorithms were implemented using the OpenGL graphics library. Compared with CUDA, this graphics-oriented programing style is not sufficiently flexible for general-purpose computation. In fact, the TB shape is hidden in the underlying graphics layer.

Krüger and Westermann [7] presented the impact of visibility-based GPU acceleration techniques such as early ray termination and empty space skipping [26]. Using these acceleration techniques, rendering performance was increased by a factor of 3. In addition, a similar technique was presented by Rijters and Vilanova [8], who employ an octree data structure on the GPU. These visibility-based techniques reduce data access, while our optimization strategy reduces data access latency by effective cache utilization.

3. GPU-based volume rendering

Here we present a well-known ray-casting algorithm [27] and explain a typical implementation of the algorithm on a GPU.

3.1. Ray casting

Fig. 1 illustrates the geometry used for ray casting [27]. Let V be the volume to be rendered from the viewpoint. We consider a cubic volume of $N \times N \times N$ voxels, where N is the volume size. We assume that each voxel has scalar data associated with color and opacity values. Let x, y, and z be elements of the voxel coordinates.

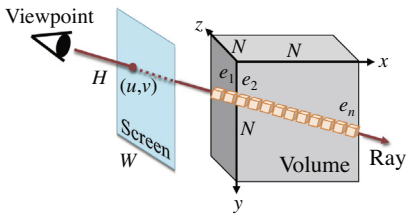

Fig. 1. Geometry of ray casting. Pixel values are computed by accumulating color and opacity values of voxels penetrated by a ray from the viewpoint.

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

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

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

3.2. Compute unified device architecture

A CUDA-compatible GPU [11] consists of at least hundreds of CUDA cores structured in a hierarchy. This hardware has tens of streaming multiprocessors (SMs), each containing multiple CUDA cores depending on its generation. Using these rich cores, thousands of threads can be executed in a single-instruction, multiple-thread (SIMT) fashion [11].

Threads are classified into data independent groups, i.e., TBs, which are then assigned to an SM in a cyclic manner until they exhaust available resources such as register files. Such data-independent TBs contribute to the flexible and efficient scheduling of threads. This allows more TBs to be resided and processed together on an SM and achieves the efficient overlap of memory operations and arithmetic instructions. Each resident and concurrent TB is further divided 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 from applications.

Threads usually form a 2D TB and can be identified with a 2D 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 specified in the application directly. Here p and q represent the width and height of warps, respectively. However, 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 indirectly specified through the TB shape $w \times h$. Table 1 shows the correspondence between TB shape and warp shape when $wh = 256$. This table shows that horizontal warps (i.e., $p > q$) are generated if $w \geqslant 8$. Otherwise, vertical warps (i.e., $p < q$) are generated.

3.3. Texture-based rendering with CUDA

Eq. (1) indicates that different rays can be processed in parallel, because there is no data dependence between them. Consequently, typical renderers exploit data parallelism by assigning the accumulation of a ray to a thread. 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 the front-to-back order.

Table 1
Relationship between TB shape $w \times h$ and warp shape $p \times q$.

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

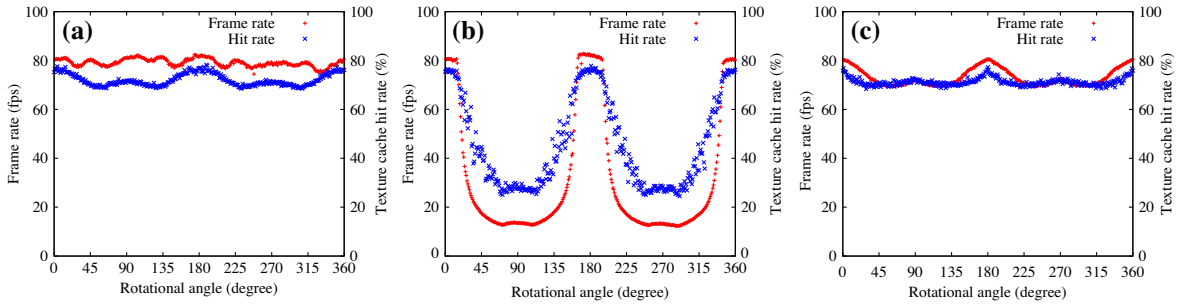

Fig. 2. Relationship between frame rate and hit rate of texture cache: rotations around (a) the *x*-axis, (b) the *y*-axis, and (c) the *z*-axis. These results were obtained with a fullerene dataset (see Section 6).

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 rendering [10], which performs interpolation using GPU texture mapping hardware. Thus, the volume is accessed via 3D texture to take advantage of hardware-accelerated interpolation. Before accessing the volume, the 3D texture must be rotated against the fixed viewpoint to reflect the location of the viewpoint. Let θ_x, θ_y, and θ_z be the rotational angles around the *x*-, *y*-, and *z*-axes, respectively. We assume that these rotational transformations, each corresponding to one of the three axes, are applied to the volume in the order of the *x*-, *y*-, and *z*-axes. The location of an arbitrary viewpoint can then be specified by angle $(\theta_x, \theta_y, \theta_z)$, where $0 \leqslant \theta_x, \theta_y, \theta_z < 360$.

Fig. 2 shows the results of a preliminary evaluation obtained with a CUDA software development kit (SDK) renderer [11]. The cache hit rate was measured by CUDA Visual Profiler, which provides profiling results on an SM. As shown in this figure, the hit rate of the texture cache primarily determines the rendering frame rate. These results motivated us to tackle the issue of cache optimization for fast rendering.

4. Texture memory organization

Fig. 3 shows how a logical address space is mapped to a physical memory space in 3D texture [28,29]. As shown in this figure, 3D texture consists of many 2D slices. Each slice is further optimized for 2D spatial locality via Morton's z-order curve [30], as illustrated by the sequence of red arrows in Fig. 3. For simplicity, we assume that volume size N is a power of two in the following discussion: $N = 2^L$, where L is a natural number. As shown in Fig. 4, a Morton curve has a recursive hierarchy. Thus, a z-ordered block at the lth level of the hierarchy contains a 2D slice of $2^l \times 2^l$ texels, where $1 \leqslant l \leqslant L$.

Although the Morton curve is optimized for 2D spatial locality, the physical stride between two adjacent voxels is not uniform in this data structure. Here we investigate the physical stride in more detail. Suppose that two adjacent voxels e_i

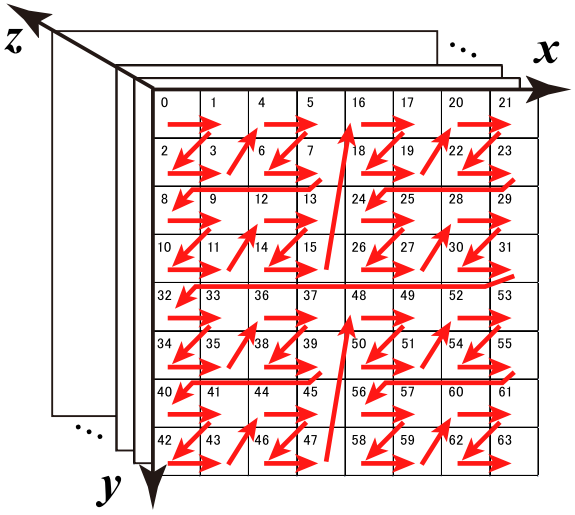

Fig. 3. Organization of 3D texture in CUDA. The series of red arrows represents the sequence of physical memory address in a 2D slice. The physical address is shown in the top left corner of each texel. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

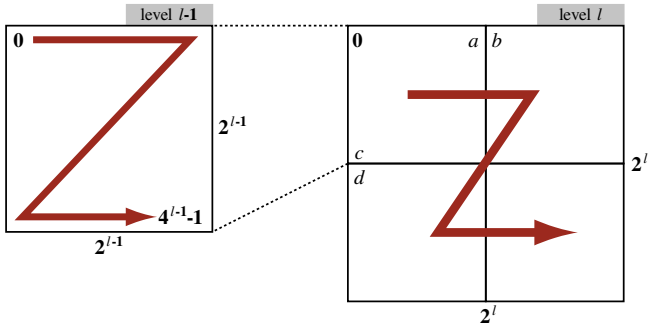

Fig. 4. Hierarchical structure of a Morton curve. A block at the lth level contains four internal blocks of the $(l-1)$th level. The maximum stride appears between these internal blocks; between texels a and b along the horizontal axis and between texels c and d along the vertical axis, where a, b, c, and d represent the top-right, top-left, bottom-left, and top-left texels of the internal blocks, respectively.

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 2D 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 to which 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, as shown in Fig. 4, the maximum stride at the lth level of the hierarchy appears between adjacent blocks of the internal $(l-1)$th level, i.e., between texels a and b along the horizontal axis and between texels c and d along the vertical axis (Fig. 4). The physical indexes of texels b and d are 4^{l-1} and $2 \cdot 4^{l-1}$, respectively. On the other hand, the physical indexes of texels a and c are $\sum_{k=0}^{l-2} 4^k$ and $2 \cdot \sum_{k=0}^{l-2} 4^k$, respectively. Consequently, the maximum stride along the x-axis can be given by $4^{l-1} - \sum_{k=0}^{l-2} 4^k = (2 \cdot 4^{l-1} + 1)/3$, and the maximum stride along the y-axis can be given by $2 \cdot 4^{l-1} - 2 \cdot \sum_{k=0}^{l-2} 4^k = 2 \cdot (2 \cdot 4^{l-1} + 1)/3$. Because $N = 2^L$ and $1 \leqslant l \leqslant L$, voxels along the x-axis and y-axis can be accessed with maximum strides of $(N^2 + 2)/6$ and $(N^2 + 2)/3$, respectively.

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

5. Proposed method

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

1. Dynamic selection of the TB shape (Section 5.1).
2. Transposed indexing of threads (Section 5.2).
3. TB size maximization (Section 5.3).
4. Synchronous ray propagation (Section 5.4).

The first strategy performs warp-level optimization, and 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 angle $(\theta_x, \theta_y, \theta_z)$). Table 2 summarizes the relationship between the activated strategies and typical rotations. Note that this table shows the relationship for $0 \leqslant \theta_x, \theta_y, \theta_z \leqslant 90$; however, it can be extended to other rotational angles $90 < \theta_x, \theta_y, \theta_z \leqslant 360$ using the symmetricity of geometry.

5.1. Dynamic selection of TB shape

To maximize the locality of reference, the proposed method focuses on the following three points:

1. The SM simultaneously processes threads in the same warp.
2. Each volume axis has a different stride between adjacent voxels, as discussed in Section 4.
3. The warp shape $p \times q$ is determined by the TB shape $w \times h$, as mentioned in Section 3.2.

Table 2
Relationship between the activated strategies and typical rotations. An empty cell corresponds to an inactive strategy.

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

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 [3,31], optimization was achieved for a single process (i.e., a single ray). Such an optimization scheme is sufficient for earlier systems, where each processing core processes a single ray at a time. In contrast, we emphasize the optimization of a warp (i.e., a ray frustum) as a key GPU acceleration strategy, in which each SM simultaneously processes 32 threads (i.e., a warp). Thus, we must investigate the memory access pattern 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. Note that this approximation is used only for TB shape optimization. Our ray sampling procedure computes the exact ray length, because such an approximation can cause ring artifacts on the final image [32].

With respect to the second point, voxels should always be 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 appearances on the screen; thus, the x-axis can be parallel to one of the horizontal, vertical, and depth directions. Therefore, we give priority to the volume axes, i.e., voxels should be accessed in the order of x, y, and z to produce smaller strides. We optimize the warp shape to realize this warp prioritization.

The third point plays a key role in realizing prioritized access. The warp size $p \times q$ must be selected so 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 (b). In such cases, horizontal warps are permitted to access voxels with smaller strides than vertical warps.

According to the three points mentioned above, we determine 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 determined by the following three steps:

1. Parallel plane detection. From the xy-, yz-, and xz-planes, our method detects the plane most parallel to the screen. For example, the xy-plane is such a parallel plane in Fig. 5(a) and (c). This parallel plane can be easily detected by choosing a plane that has the maximum inner product of its perpendicular line and the viewing direction.
2. Primary axis detection. The primary axis with a smaller stride is selected from the two axes that compose the parallel plane. For example, the parallel plane in Fig. 5(d) is the yz-plane; 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 warps are selected if the primary axis is rendered in vertical lines on the screen, and horizontal warps are selected if the primary axis is rendered in horizontal lines on the screen. For example, the TB shape of 1×256 is selected for viewpoints

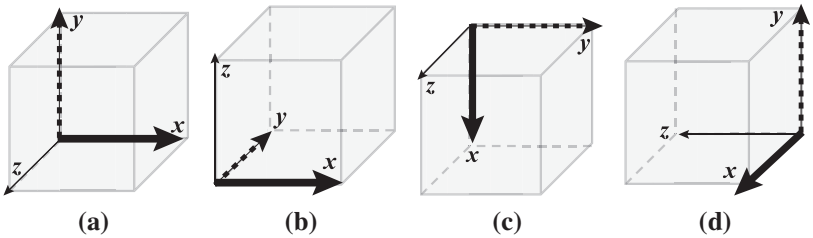

Fig. 5. Geometrical relationship between the viewpoint and the volume axes: (a) $(\theta_x, \theta_y, \theta_z) = (0, 0, 0)$, (b) $(\theta_x, \theta_y, \theta_z) = (90, 0, 0)$, (c) $(\theta_x, \theta_y, \theta_z) = (0, 0, 90)$, and (d) $(\theta_x, \theta_y, \theta_z) = (0, 90, 0)$. For simplicity, we consider four representative viewpoints (a)–(d), which make two of the volume axes parallel to the screen axes. The x-axis and z-axis have the smallest stride and the largest stride among the volume axes, respectively.

in Fig. 5(c) and (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 $0 \leqslant \theta_x, \theta_y, \theta_z \leqslant 90$ into six groups (Table 2), because the warp shape $p \times q$ can be one of the six configurations (Table 1 vertical line becomes a horizontal line after 90 degree rotation.

With respect to the 32×1 warp shape, there are four candidates for the TB shape in Table 1: $w \times h = 32 \times 8, 64 \times 4, 128 \times 2$, and 256×1. Among these candidates, we use $w \times h = 32 \times 8$, according to our preliminary evaluation results. This shape achieves the highest performance among the candidates, because textures are optimized for 2D spatial locality, as mentioned in Section 4. From this viewpoint, 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 position a series of TBs as vertical as possible; thus, SMs are allowed to access voxels along the primary axis with a smaller stride.

Fig. 6 shows an example of transposed indexing. As shown in Fig. 6(a), our method chooses vertical TBs if the primary axis is vertical. However, a series of vertical TBs is placed horizontally in the original indexing, which are then assigned to SMs in a cyclic manner. Consequently, SMs simultaneously access different columns, failing to exploit the data locality along the primary axis.

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

5.3. Thread block size maximization

As mentioned in Section 3.2, the SM is expected to concurrently execute several resident TBs. Having more resident TBs is useful to maximize the effect of memory access hiding if they have high locality with a high cache hit rate. However, this does not apply if resident 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 resident TBs. In addition, the TB size *wh* must be a multiple of the warp size (i.e., 32) to avoid CUDA cores idling 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 voxels along the *y*-axis; thus, the warps are required to tolerate relatively large strides. To make the matter worse, resident TBs are usually responsible for separate regions on the screen due to the cyclic task distribution mentioned in Section 3.2. Consequently, we use a TB size *wh* of 512, which is the largest configuration that can run on the GPU used in our experiments.

5.4. Synchronous ray propagation

Owing to the nature of SIMT execution, resident threads can concurrently execute different lines of the GPU code. This implies that rays proceed independently even though they are assigned to the same TB. Our 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 (Fig. 5(a) and (c)). In this case, rays are parallel to the *z*-axis, which has the largest stride. Consequently, overhead may be negligible compared with the benefit provided by cache utilization.

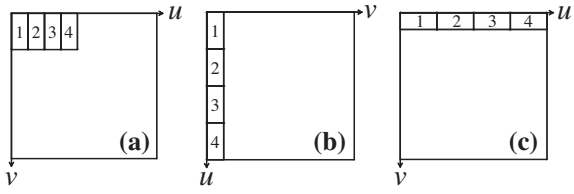

Fig. 6. Transposed indexing of threads. (a) Vertical TBs without transpose, (b) vertical TBs with transpose, and (c) horizontal TBs without transpose. A rectangle with a number represents a TB. Because TB assignment is performed along the *u*-axis in a cyclic manner, (a) a naive assignment of vertical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

6. Experimental results

To evaluate our cache-aware method in terms of rendering performance, we applied the method to renderers included in both CUDA and OpenCL SDKs [33,34]. The original renderers use a 1D texture to store a color map table, which associates color and opacity values with each voxel. Because reference to this table results in the perturbation of cache behavior, we modified the code to store the table in fast shared memory [11]. Thus, the modified code uses textures only for the volume data. We also modified the OpenCL-based renderer so that it can behave the same as the CUDA-based renderer; we replaced its back-to-front rendering algorithm with a front-to-back algorithm and adopted early ray termination [7] for acceleration. Both 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, which are 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 were resized to $N = 1024$, each with 8-bit data, and were rendered on a screen of size $W \times H = 1024 \times 1024$. We used a desktop PC equipped with an NVIDIA GeForce 580 GTX card and an Intel Core i7 930. Our machine ran with Windows 7, CUDA 4.2, OpenCL 1.1, and graphics driver version 306.97.

Fig. 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*-, and then the *z*-axes. Fig. 9 shows the width p of warps selected for viewpoints.

First, for the *x*-axis rotation, the parallel plane can be the *xy*-plane or the *xz*-plane (Fig. 5(a) and (b)). As shown in Fig. 8(a), the static method shows relatively high frame rates; however, the frame rate decreases slightly for rotational angles near $\theta_x = 90$ and 270. In contrast, the proposed method eliminates such performance degradation successfully by utilizing horizontal warps for all θ_x (Fig. 9(a)). As shown in Fig. 5(a) and (b), this 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 such rotational angles.

For the *y*-axis rotation, the parallel plane can be the *xy*-plane or the *yz*-plane (Fig. 5(a) and (d)). The frame rates in Fig. 8(b) show significantly different behavior compared with those in Fig. 8(a). The frame rate decreases in both the static method and our method when $30 \leqslant \theta_y \leqslant 150$ and $210 \leqslant \theta_y \leqslant 330$; however, the worst frame rate increases from 7.9 fps to 15.2 fps by our proposed method. This increase of worst frame rate is due to the switch to vertical warps and transposed indexing of threads (Fig. 9(b)). These significant decreases are also due to the *z*-axis, which appears 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 we optimize their shape. When $\theta_y = 90$, our dynamic selection strategy increased the frame rate from 7.9 fps to 11.0 fps, which was then increased to 12.6 fps by our transposed indexing strategy.

In-place rotation of the volume dataset may be useful for this worst case, because this transformation can significantly increase the frame rate by swapping the *z*-axis with the y-axis. However, current CUDA prohibits writing to textures (i.e., CUDA arrays) [11]. Consequently, this idea will be useful in the future architectures that provide writable CUDA arrays.

Finally, for *z*-axis rotation, the parallel plane can be the *xy*-plane (Fig. 5(a) and (d)). The frame rates in Fig. 8(c) are similar to those of Fig. 8(a); however, our method fails to outperform the static method at $\theta_z = 0 (= 360)$ and 180. This is due to the overhead of thread synchronization. As mentioned in Section 5.4, we activate our synchronous ray propagation strategy when the *z*-axis is parallel to the depth direction. Switching this strategy off at $\theta_z = 0 (= 360)$ and 180 increased the frame rate from 51.6 fps to 54.3 fps, which is close to that of the static method (54.8 fps). However, asynchronous ray propagation dropped the frame rates at other angles.

Note that the fundamentals of our optimization results are similar to those observed by Torres et al. [18,19], because both change the TB shape to find the best memory access pattern of a warp. However, we found that vertical warps maximize rendering performance for some viewpoints, whereas such execution configurations resulted in a poor performance in [18,19]. We think that the different performance behaviors are due to the memory access pattern inherent in the target problem (i.e., volume rendering), where voxels can be accessed from arbitrary directions. Our work extends the previous work by realizing an auto-tuning technique that considers the real memory access pattern in a specific application.

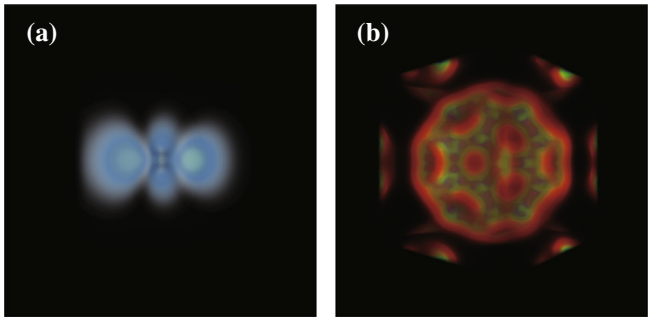

Fig. 7. Experimental datasets. (a) Spatial probability distribution of the electron in a hydrogen atom; (b) a buckyball fullerene.

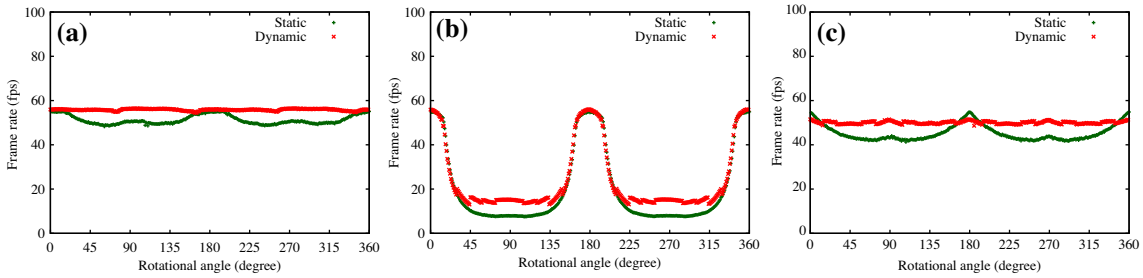

Fig. 8. Frame rates of our dynamic method and the original static method: rotations around (a) the x-axis, (b) the y-axis, and (c) the z-axis. The atom dataset was used with CUDA.

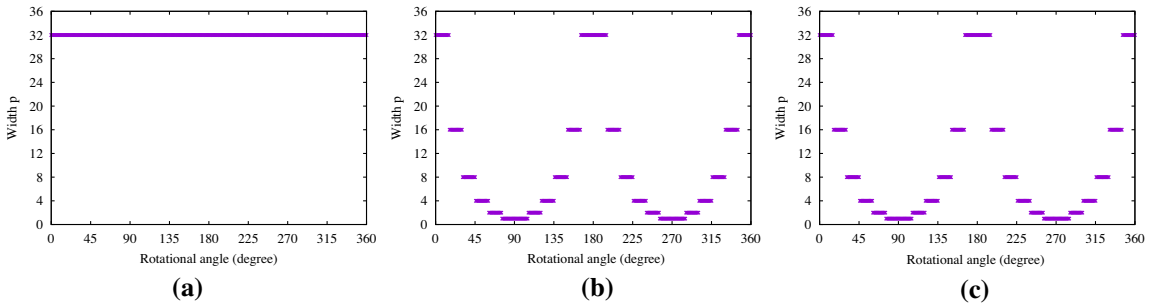

Fig. 9. The width p of warps selected for viewpoints: rotations around (a) the x-axis, (b) the y-axis, and (c) the z-axis. The height is given by $q = 32/p$. The original static method uses $p = 16$ for arbitrary viewpoints.

We then analyzed the frame rates for the opaque fullerene dataset. This opaque dataset allows rays to be terminated earlier than the transparent atom dataset. For example, the mean and deviation of sampling counts per ray are 320 and 250 for y-axis rotation of the opaque dataset, respectively. Without early ray termination, these values increase to 499 and 417, respectively, for both datasets. For the transparent dataset, the mean and deviation are 480 and 401, respectively, which are slight decrease compared to the original values. Note that we consider here rays penetrating at least one voxel, because rays that propagate outside the volume region are not generated during rendering.

Fig. 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 increase by roughly 20 fps because of early ray termination. However, this increase cannot be clearly seen when $30 \leqslant \theta_y \leqslant 150$ and $210 \leqslant \theta_y \leqslant 330$, as shown in Fig. 10(b), where the frame rates are less than 20 fps. In contrast, our dynamic method successfully increases the frame rates to approximately 30 fps, as shown in Fig. 10(b). The proposed method improves the degraded frame rate from 13.0 fps to 29.0 fps when $\theta_y = 90$, obtaining a speedup factor of 2.2, which was the best result. This indicates that, for opaque datasets, the impact of cache optimization is larger than that of visibility-based optimization, such as early ray termination. Furthermore, the achieved frame rates ($\sim$30 fps) are higher than those obtained with the transparent dataset ($\sim$20 fps, Fig. 8(b)), which has longer rays than the opaque dataset. As compared to long rays, short rays prevent large gaps between progress in ray propagation. This means that, for opaque datasets, warps can exploit locality along the depth direction. Actually, the best speedup of 2.2 is higher than that for the transparent dataset, and it is obtained when rays proceed along the best x-axis ($\theta_y = 90$). We believe that our cache-aware method cooperates

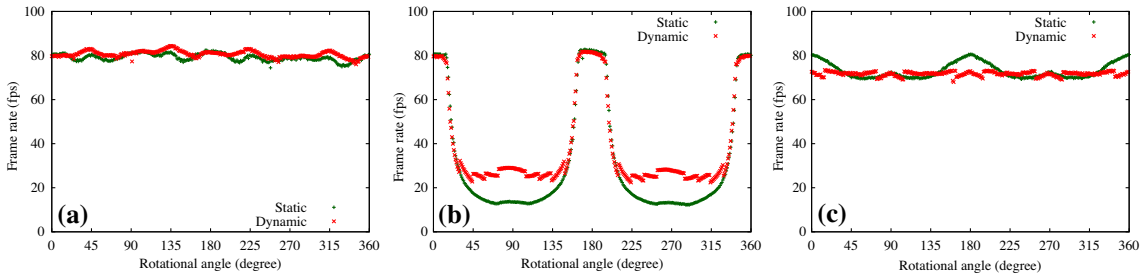

Fig. 10. Frame rates of our dynamic method and the original static method: rotations around (a) the x-axis, (b) the y-axis, and (c) the z-axis. The fullerene dataset was used with CUDA.

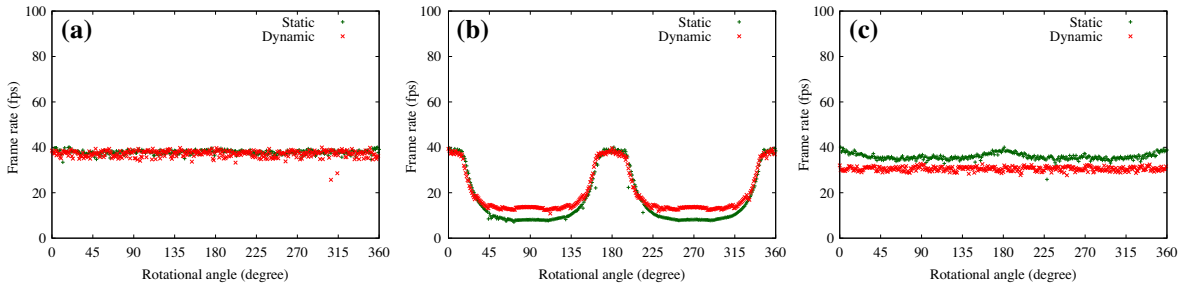

Fig. 11. Frame rates of our dynamic method and the original static method: rotations around (a) the x-axis, (b) the y-axis, and (c) the z-axis. The atom dataset was used with OpenCL.

well with visibility-based methods, because both methods focus on different performance criteria. That is, our method reduces the latency of a texture fetch while early ray termination reduces the number of texture fetches.

As shown in Fig. 10(c), we again observed performance decrease due to the synchronization overhead. In contrast to the transparent dataset results shown in Fig. 8(c), the decrease can be found in many rotational angles around $\theta_z = 0(= 360)$ and 180. This is due to early ray termination, which shortly terminates rays on the object surface. Because of this early termination, the penetration length in the depth direction is reduced between propagating rays. Consequently, such short rays can be more naturally synchronized compared with long rays propagating transparent objects. Thus, the static method, which does not synchronize threads at every iteration, propagates naturally synchronized rays for the opaque dataset. For such short rays, the synchronization cost outweighs the benefit (i.e., better cache utilization).

Finally, we evaluated our method using an OpenCL-based renderer. Fig. 11 shows the frame rates for the atom dataset. Compared with the CUDA-based results shown in Fig. 8, the frame rates decrease by roughly 33%. This lower performance is most likely due to the driver, which is not well optimized for OpenCL. Despite this lower performance, the performance characteristics shown in Fig. 11 are similar to those of Fig. 8. The proposed method increased the frame rates for the y-axis rotation successfully. However, the synchronization overhead again reduced frame rates for the z-axis rotation. Consequently, a detailed overhead investigation is required before using our synchronous ray propagation strategy.

7. Conclusion

In this study, we presented an acceleration method for texture-based volume rendering on a GPU. The proposed method increases the hit rate of texture cache by selecting the shape of TBs at run time. This dynamic selection of TB shape focuses on the geometrical relationship between the viewpoint and volume axes and determines TB shape so that threads in the same warp can have a small memory access stride. 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 also activates TB-level optimization for some specific viewpoints.

We compared our method with a static method that uses a fixed TB shape. We found that our dynamic method increased frame rates for y-axis rotation; however, rendering performance was relatively lower than other viewpoints. Moreover, our cache-aware method worked efficiently with a visibility-based method, demonstrating particular efficient utilization of texture cache for an opaque dataset. Owing to our cache utilization, the worst frame rates were increased from 13.0 fps to 29.0 fps, achieving a speedup factor of 2.2.

Future work includes further investigation of the synchronous ray propagation strategy. Although we found that this strategy increased the frame rates for some viewpoints, its effectiveness depends on the implementation platform and data content.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Nos. 23300007 and 23700057 and the Japan Science and Technology Agency 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/ courtesy of SFB 382 of the German Research Council. We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] R.A. Drebin, L. Carpenter, P. Hanrahan, Volume rendering, Comput. Graphics (Proc. SIGGRAPH'88) 22 (3) (1988) 65–74.
[2] D.R. Ney, E.K. Fishman, D. Magid, R.A. Drebin, Volumetric rendering of computed tomography data: principles and techniques, IEEE Comput. Graphics Appl. 10 (2) (1990) 24–32.

[3] A. Takeuchi, F. Ino, K. Hagihara, An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors, Parallel Comput. 29 (11/12) (2003) 1745–1762.
[4] S.P. Uselton, Volume Rendering for Computational Fluid Dynamics: Initial Results, Tech. Rep. RNR-91-026, Nasa Ames Research Center, 1991.
[5] D.S. Ebert, R. Yagel, J. Scott, Y. Kurzion, Volume rendering methods for computational fluid dynamics visualization, in: Proc. 5th IEEE Visualization Conf. (VIS'94), 1994, pp. 355–361.
[6] D. Nagayasu, F. Ino, K. Hagihara, A decompression pipeline for accelerating out-of-core volume rendering of time-varying data, Comput. Graphics 32 (3) (2008) 350–362.
[7] J. Krüger, R. Westermann, Acceleration techniques for GPU-based volume rendering, in: Proc. 14th IEEE Visualization Conf. (VIS'03), 2003, pp. 232–239.
[8] D. Rijters, A. Vilanova, Optimizing GPU volume rendering, J. WSCG 14 (1/3) (2006) 9–16.
[9] D. Luebke, G. Humphreys, How GPUs work, Computer 40 (2) (2007) 96–100.
[10] W. Hibbard, D. Santek, Interactivity is the key, in: Proc. Chapel Hill Workshop Volume Visualization (VVS '89), 1989, pp. 39–43.
[11] NVIDIA Corporation, CUDA Programming Guide Version 4.2. <http://developer.nvidia.com/cuda/>, 2012.
[12] Khronos OpenCL Working Group, The OpenCL specification version 1.1. <http://www.khronos.org/registry/cl/>, 2011.
[13] Y. Sugimoto, F. Ino, K. Hagihara, Improving cache locality for ray casting with CUDA, in: Proc. 25th Int'l Conf. Architecture of Computing Systems Workshops (ARCS Workshops '12), 2012, pp. 339–350.
[14] A. Nukada, S. Matsuoka, Auto-tuning 3-d FFT library for CUDA GPUs, in: Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'09), 2009, p. 10 (CD-ROM).
[15] P. Guo, L. Wang, Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs, in: Proc. Int'l Conf. Computational and Information Sciences (ICCIS'10), 2010, pp. 1154–1157.
[16] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'10), 2010, p. 12 (CD-ROM).
[17] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, S.-Z. Ueng, S.S. Baghsorkhi, W. mei, W. Hwu, Program optimization carving for GPU computing, J. Parallel Distrib. Comput. 68 (10) (2008) 1389–1401.
[18] Y. Torres, A. Gonzalez-Escribano, D.R. Llanos, Using Fermi architecture knowledge to speed up CUDA and OpenCL programs, in: Proc. 10th Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'12), 2012, pp. 617–624.
[19] Y. Torres, A. Gonzalez-Escribano, D.R. Llanos, uBench: exposing the impact of CUDA block geometry in terms of performance, J. Supercomput. 65 (3) (2013) 1150–1163.
[20] Y. Liu, E.Z. Zhang, X. Shen, A cross-input adaptive framework for GPU program optimizations, in: Proc. 23th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'09), 2009, p. 10 (CD-ROM).
[21] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming, Parallel Comput. 38 (8) (2012) 391–407.
[22] A.R. Brodtkorb, T.R. Hagen, M.L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, J. Parallel Distrib. Comput. 73 (1) (2013) 4–13.
[23] Z. Zheng, K. Mueller, Cache-aware GPU memory scheduling scheme for CT back-projection, in: Proc. Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC'10), 2010, pp. 2248–2251.
[24] Y. Okitsu, F. Ino, K. Hagihara, High-performance cone beam reconstruction using CUDA compatible GPUs, Parallel Comput. 36 (2/3) (2010) 129–141.
[25] N.K. Govindaraju, D. Manocha, Cache-efficient numerical algorithms using graphics hardware, Parallel Comput. 33 (10/11) (2007) 663–684.
[26] M. Levoy, Efficient ray tracing of volume data, ACM Trans. Graphics 9 (3) (1990) 245–261.
[27] M. Levoy, Display of surfaces from volume data, IEEE Comput. Graphics Appl. 8 (3) (1988) 29–37.
[28] J. Montrym, H. Moreton, The GeForce 6800, IEEE Micro 25 (2) (2005) 41–51.
[29] M. Pharr, R. Fernando (Eds.), GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, Reading, MA, 2005.
[30] G.M. Morton, A Computer Oriented Geodetic Data Base and A New Technique in File Sequencing, Tech. rep., IBM Ltd, Ottawa, Ontario, 1966.
[31] M. Matsui, F. Ino, K. Hagihara, Parallel volume rendering with early ray termination for visualizing large-scale datasets, in: Proc. 2nd Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'04), 2004, pp. 245–256.
[32] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, W. Strasser, Smart hardware-accelerated volume rendering, in: Proc. 5th Eurographics-IEEE TCVG Symp. Visualization (VisSym'03), 2003, pp. 231–238.
[33] NVIDIA Corporation, GPU Computing SDK. <http://developer.nvidia.com/gpu-computing-sdk/>, 2012.
[34] NVIDIA Corporation, OpenCL SDK. <http://developer.nvidia.com/opencl/>, 2012.