Improving Cache Locality for Ray Casting with CUDA*

Yuki Sugimoto, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{y-sugimt,ino}@ist.osaka-u.ac.jp

Abstract: In this paper, we present an acceleration method for texture-based ray casting on the compute unified device architecture (CUDA) compatible graphics processing unit (GPU). Since ray casting is a memory-intensive application, our method increases the hit rate of the texture cache during rendering. To achieve this, our method dynamically selects 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 achieve high data locality for specific viewpoints. The objective of this selection is to allow every warp rather than every thread to access data with a small stride, because the GPU executes multiple threads at the same time. In experiments using a GeForce GTX 480 card (i.e., the latest Fermi architecture), we find that the speedup of our method ranges from a factor of 1.0 to that of 4.0, depending on viewpoints. We think that optimizing the shape of TBs is important to achieve more cache hits in the highly-threaded CUDA hardware.

1 Introduction

Ray casting [Lev88] is a visualization technique for intuitive understanding of three-dimensional (3-D) objects. For example, this technique is useful to analyze not only computed tomography (CT) images in medical area [TIH03] but also simulation results in computational fluid dynamics [NIH08]. In ray casting, the voxel values of the volume are accumulated into pixel values on the screen. To do this, 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. During this accumulation procedure, voxel values can be reused only within neighboring region. Therefore, ray casting is a memory-intensive application rather than a compute-intensive application.

To deal with this large amount of memory access, many renderers [NIH08, KW03, RV06] were implemented using the graphics processing unit (GPU) [MM05], which is an ac-

*This work was partly supported by JSPS Grant-in-Aid for Scientific Research (B)(23300007) and Young Researchers (B)(23700057)

celerator for graphics applications. The memory bandwidth of the GPU is an order of magnitude higher than that of the CPU. Furthermore, this 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 there is no data dependence between different rays (i.e., different pixels). The volume data is typically loaded as a 3-D texture to interpolate voxel values using texture mapping hardware of the GPU. This 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 references.

In this paper, we present a view-dependent method for increasing the hit rate of the texture cache, aiming at accelerating texture-based ray casting [HS89]. To achieve this, our method maximizes the data locality by dynamically selecting the width and height of TBs according to the geometrical relationship between the viewpoint and the volume axes. The shape of TBs is selected such that a group of threads called warp [NVI10] can access data with a small stride. Since threads in the same warp are simultaneously processed on the GPU, such parallel threads have to maximize the locality of references. Our method currently works with the compute unified device architecture (CUDA) [NVI10], which is a programming framework for the NVIDIA GPU.

2 GPU-based Volume Rendering

2.1 Compute Unified Device Architecture (CUDA)

The CUDA-compatible hardware [NVI10] consists of hundreds of CUDA cores structured in a hierarchy. The hardware has tens of streaming multiprocessors (SMs), each containing 8 or 32 CUDA cores depending on the generation. Using these cores, thousands of threads are executed in a single-instruction, multiple-thread (SIMT) fashion [NVI10]. This highly-threaded architecture is designed to overlap memory latency with computation.

To achieve efficient overlap, threads are classified into data independent groups, namely *thread blocks* (TBs). Therefore, more TBs should be resided and processed together on the SM. Since there is no data dependence between TBs, such concurrent TBs contribute to have more flexibility for efficient scheduling of threads. 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.

Threads can be identified using a 2-D index, forming a 2-D TB. The TB shape $w \times h$, where w and h be the width and the height of TBs, respectively, can be specified by an argument to the kernel function, which runs on the GPU. On the other hand, the warp shape $p \times q$ cannot be directly specified by the program, where p and q represent the width and the height of warps, respectively. Since threads in a warp have consecutive indexes, the warp shape is automatically determined by the TB shape. The execution order of warps is dynamically determined by the warp scheduler, which cannot be controlled by the program.

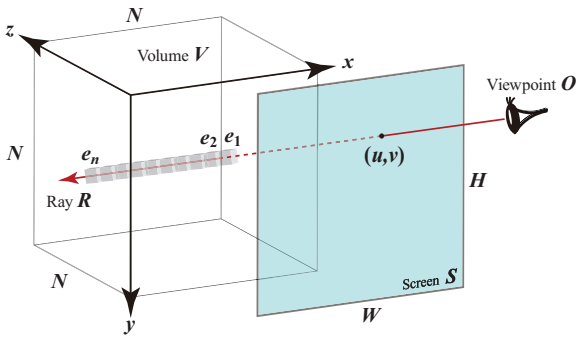


Figure 1: Geometry of ray casting. Pixel values are computed by accumulating color and opacity values of voxels penetrated by a ray from the viewpoint.

2.2 Ray Casting

Figure 1 illustrates the geometry used for ray casting [Lev88]. 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 R 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. Ray R penetrates voxels in the volume, so that the value $S(u, v)$ of pixel (u, v) is computed by accumulating color and opacity values of penetrated voxels in front-to-back order. This accumulation is done at regular intervals along ray R as follows:

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

where e_i represents the i-th voxel penetrated by ray R, 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 $\alpha(e_0) = 0$.

2.3 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 interpo-

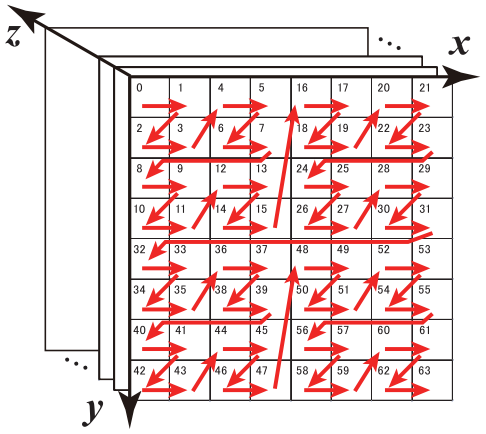


Figure 2: Organization of a 3-D texture in CUDA. A 3-D texture consists of a bunch of 2-D slices optimized for 2-D spatial locality via a z-order curve [Mor66]. A series of red arrows represents the sequence of physical memory address in a 2-D slice. The physical address is shown in each texel's upper left corner.

lated before accumulation. To accelerate this interpolation, many implementations employ texture-based rendering [HS89], 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.

3 Texture Memory Organization

Figure 2 shows how the GPU maps a logical address space onto a physical memory space in a 3-D texture [MM05, PF05]. 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 a z-order curve [Mor66], as illustrated in a sequence of red arrows in Fig. 2. The z-order 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 $1 \leq l \leq \lceil \log N \rceil$ (see Fig. 3). For simplicity, we assume N being a power of two (i.e., $N = 2^l$) in the following discussion.

Although the z-order curve is optimized for 2-D spatial locality, the physical stride between two adjacent voxels is not uniform in this data structure. To investigate this issue in more detail, let us consider accessing two adjacent voxels e_i and e_{i+1}. The stride between these voxels can then be classified into two groups depending on their coordinates:

1. The adjacent voxels have different z. In this case, voxels e_i and e_{i+1} exist on two adjacent 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 y or x. In these cases, voxels e_i and e_{i+1} exist

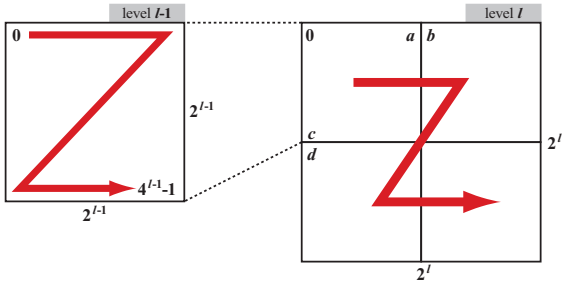


Figure 3: Hierarchical structure of a z-order curve. A block at the l-th level contains four internal blocks of the $(l-1)$-th level. The maximum stride appears between these internal blocks: between texels a and b along the horizontal axis, and between texels c and d along the vertical axis. The physical 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.

on the same slice. The stride between them varies according to their location on the slice. For example, the strides along the x-axis range from 1 to 11 in Fig. 2. 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. 3. 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.

4 Proposed Method

In this section, we describe our acceleration method that selects the TB shape $w \times h$ during rendering. We first explain our acceleration strategy, and then present how our method selects the TB shape according to the strategy.

4.1 Acceleration Strategy

As we mentioned in Section 1, our method maximizes the locality of references. To achieve this, we focus on four 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.

3. The TB shape $w \times h$ determines the warp shape $p \times q$. For details, see [NVI10].

4. The number of resident TBs should be maximized to take advantage of the highly-threaded GPU architecture.

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 the highly-threaded GPU architecture. On earlier acceleration systems such as cluster systems [TIH03, MIH04], optimization is successfully done for a single process (i.e., a single ray). 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. Since 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. For example, the x-axis can be rendered as a vertical line on the screen, but it can appear as a horizontal line with a different viewpoint. Thus, the volume axes have different appearance on the screen, depending on the location of the viewpoint (see Fig. 4). Therefore, we have determined to give priority to the volume axes: voxels should be accessed in the order of x, y, and z to have smaller strides. Clearly, this priority should be implemented as per-warp order instead of per-thread order. Therefore, we optimize the warp shape to realize the prioritization.

The third point plays the key role in realizing the prioritized access mentioned above. As we mentioned in Section 2.1, the warp shape $p \times q$ is determined by the TB shape $w \times h$. Table 1 shows their relationship when using the TB size wh of 128. 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. 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, vertical warps are better than horizontal warps if the x-axis appears as a vertical line on the screen. In this case, vertical warps are allowed to access voxels with smaller strides than horizontal warps.

The last point determines the size wh of TBs. As we mentioned in Section 2.1, the number of resident TBs should be maximized to efficiently hide memory latency with computation. Due to the limitation of available resources, up to 8 TBs can be processed on the SM at a time [NVI10]. Similarly, up to 1536 threads can be resident on the SM. Therefore, the TB size wh must be $wh \leq 192$ to maximize resident TBs on the SM. In addition, the TB size wh must be a multiple of the warp size (i.e., 32) to avoid cores from being idle during SIMT execution. Since the number of resident TB depends on the amount of resource consumption, we compiled our rendering kernel with $wh = 192$, 160, and so on. We then found that $wh = 128$ is the maximum TB size that can run 8 TBs on the SM. Thus, our kernel runs with $wh = 128$.

Table 1: Relationship between TB shape $w \times h$ and warp shape $p \times q$. Values are presented for the TB size wh of 128. 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
1×128	1×32	$1 : 32$
2×64	2×16	$1 : 8$
4×32	4×8	$1 : 2$
8×16	8×4	$2 : 1$
16×8	16×2	$8 : 1$
32×4	32×1	$32 : 1$
64×2	32×1	$32 : 1$
128×1	32×1	$32 : 1$

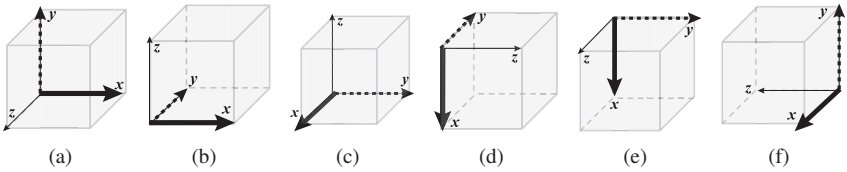


Figure 4: Geometrical relationship between the viewpoint and the volume axes. Each of subfigures (a)–(f) corresponds to one of six representative viewpoints. In these viewpoints, the x-axis can be parallel to one of the horizontal, the vertical, and the depth directions. The x-axis and the z-axis have the smallest stride and the largest stride among the volume axes, respectively.

4.2 Selection of Thread Block Shape

Our method selects the TB shape $w \times h$ according to the geometrical relationship between the viewpoint and the volume axes. For simplicity, we consider here six representative viewpoints, which make two of the volume axes parallel to the screen axes. Figure 4 shows six images rendered with the representative viewpoints. Given the viewpoint O, the TB shape is selected in the following three steps:

1. Plane detection. Our method detects the plane most parallel to the screen. For example, the xy-plane is such a parallel plane in Figs. 4(a) and 4(e). According to our approximation, voxels on a parallel plane are simultaneously accessed by a warp.

2. Primary axis detection. The primary axis with a smaller stride is selected from two axes that create the parallel plane. For example, the yz-plane is the parallel plane in Figs. 4(c) and 4(f), so that the y-axis is selected 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. Our method currently uses the TB shape of $w \times h = 32 \times 4$ for horizontal warps and that of 1×128 for vertical warps. For example, the TB shape of 32×4 is selected for viewpoints in Figs. 4(a), 4(b), and 4(c), because the primary axis is rendered in a horizontal line from these viewpoints.

5 Experimental Results

To evaluate our method in terms of the rendering performance, we measured the frame rate and the hit rate of the texture cache. For experiments, we used a desktop PC equipped with a GeForce 480 GTX card. Our machine runs with Windows 7, CUDA 3.2 [NVI10], and graphics driver 260.61. The cache hit rate was measured by CUDA Visual Profiler.

As a rendering implementation, we used a sample code distributed with CUDA SDK. This code originally uses a texture to refer a color map table, which associates color and opacity values with each voxel. Since references to this table results in perturbation of cache behavior, we modified the code such that it stores the table in shared memory. Thus, the modified code uses textures only for the volume data. We used $N = W = H = 1024$ for performance evaluation. The volume consists of 8-bit data.

The volume data was rendered using six viewpoints (a)–(f) presented in Fig. 4. Note that these viewpoints are in symmetric positions. We used symmetric viewpoints, because threads are allowed to have the same workload despite the difference of viewpoints. That is, the same number of voxels is accessed for each viewpoint, but with a different order. This is necessary to avoid misunderstandings caused by asymmetric viewpoints, which assign different workloads to threads. Due to the same reason, optimization techniques such as early ray termination and empty space skipping [Lev90] are not used during experiments.

Figure 5 shows the frame rates of our dynamic method and a static method. The static method uses a fixed shape $w \times h = 16 \times 16$ (i.e., $p \times q = 16 \times 2$) for arbitrary viewpoints. For viewpoints (b), (d), (e), and (f), our method achieves higher frame rates than the static method. The speedup over the static method reaches 1.1, 4.0, 1.3, and 1.5 for viewpoints (b), (d), (e), and (f), respectively. In particular, the frame rate for viewpoint (d) increases from 11.6 to 46.6 fps, with increasing the cache hit rate from 51.2% to 73.9%. On the contrary, there is no significant difference for viewpoints (a) and (c). This is due to the warp shape used in the static method. For these viewpoints, the static method generates horizontal warps, as our method does. Therefore, our method cannot increase the cache hit rate for these viewpoints.

Figure 6 explains how the cache hit rate determines the frame rate. We measured both rates using eight different TB shapes, ranging from $w \times h = 1 \times 128$ to 128×1. Each frame rate is the average of ten trials. As shown in Fig. 6, higher frame rates are obtained when higher cache hit rates are achieved. For instance, the highest frame rate of 50.8 fps is observed when the cache hit rate reaches 73.8%. In contrast, the lowest frame rate of 5.2 fps results in a cache hit rate of 4.2%. Thus, the frame rate is mainly determined by

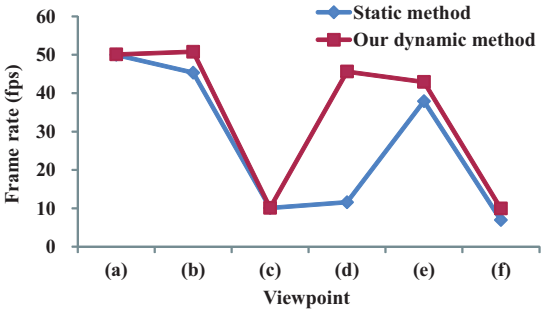


Figure 5: Comparison of frame rates between our dynamic method and the static method. The former uses the TB shape $w \times h = 32 \times 4$ for viewpoints (a)–(c) and $w \times h = 1 \times 128$ for viewpoints (d)–(f). The latter uses the default shape $w \times h = 16 \times 16$ for arbitrary viewpoints.

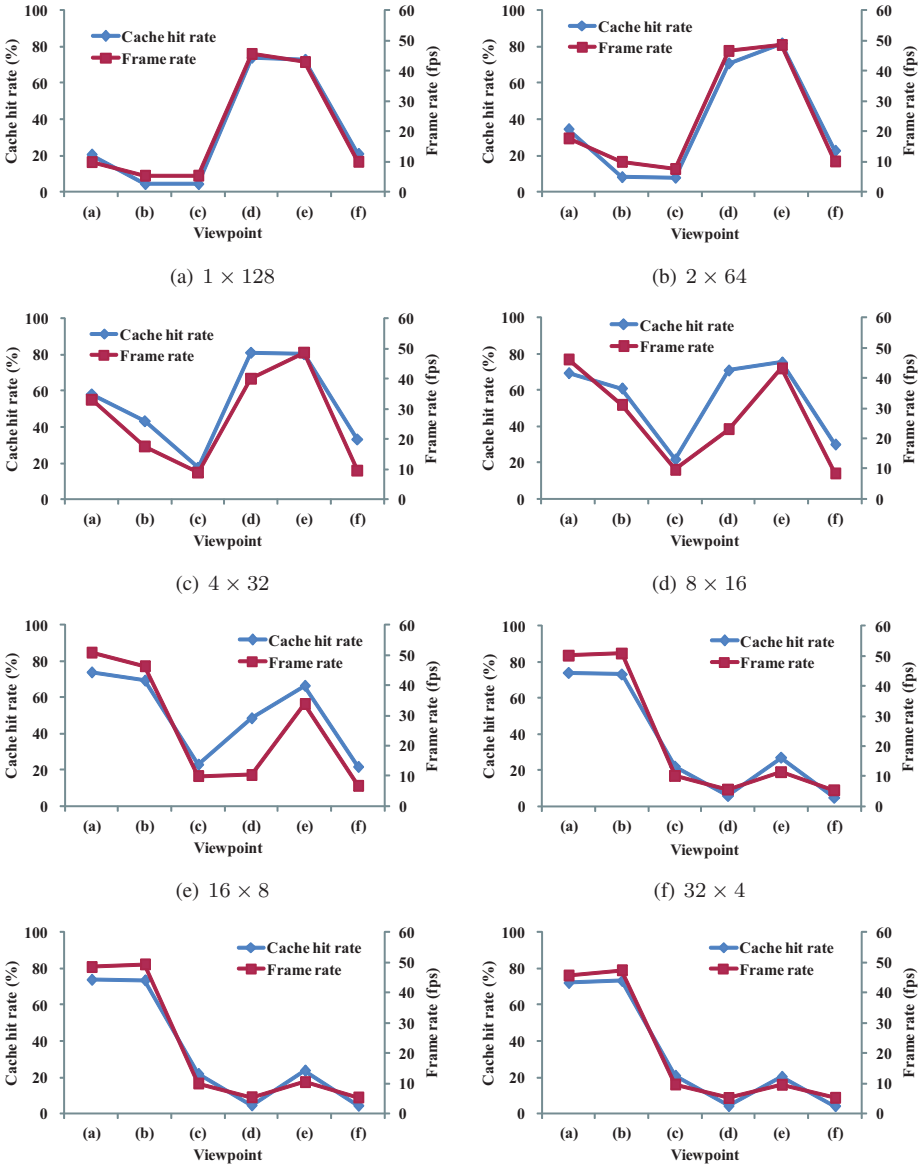
the cache hit rate.

Another important behavior in Fig. 6 is that all of the eight TB shapes have a wide range of cache hit rates ranging approximately from 5% to 80%. This behavior indicates that a single shape of TBs is not sufficient to obtain higher cache hit rates for all viewpoints. Therefore, it is better to change the TB shape according to the location of the viewpoint, as we do in our method.

We next investigate the relationship between the TB shape and the frame rate. Figure 7 shows frame rates with different TB shapes, ranging from $w \times h = 1 \times 128$ to 128×1. The results are classified into two groups: (1) viewpoints (a)–(c), which have smaller strides for horizontal warps; and (2) viewpoints (d)–(f), which have smaller strides for vertical warps. As we mentioned in Section 4.1, the warp size varies from $p \times q = 1 \times 32$ to 32×1, according to the TB shape $w \times h$ (see Table 1). Recall that horizontal warps are generated if $w \geq 8$.

Figure 7(a) indicates that horizontal warps rather than vertical warps yield high frame rates. In contrast, Fig. 7(b) shows that vertical warps achieve higher frame rates than horizontal warps. Actually, these figures are in a symmetric relation. A vertical symmetry axis exists between $w \times h = 4 \times 32$ and 8×16 (i.e., between $p \times q = 4 \times 8$ and 8×4). For example, viewpoints (b) and (d) have a cross point on the vertical symmetry axis, and both have the xz-plane as a parallel plane. Therefore, it is better to change the TB shape at the symmetry axis, which determines the warp shape to be vertical or horizontal. This dynamic optimization is exactly what our method implements.

Although our method optimizes the TB shape, the frame rates for viewpoints (c) and (f) result in lower values. When these viewpoints are used, the x-axis is parallel to the depth direction, as shown in Figs. 4(c) and 4(f). Therefore, voxels are always accessed with a large stride, which decreases the cache hit rate to at most 33.1%, as shown in Fig. 6. These results also imply that the capacity of the texture cache is not large enough to deal with $N = 1024$. Actually, the cache hit rate ranges from 33.2% to 69.2% if a smaller volume of $N = 512$ is rendered with viewpoints (c) and (f).



(a) 1×128 (b) 2×64

(c) 4×32 (d) 8×16

(e) 16×8 (f) 32×4

(g) 64×2 (h) 128×1

Figure 6: The frame rate and the cache hit rate with different TB shapes ranging from $w \times h = 1 \times 128$ to 128×1. Each subfigure contains results for six viewpoints shown in Fig. 4(a)–(f).

6 Related Work

Krüger *et al.* [KW03] presented the impact of optimization techniques such as early ray termination and empty space skipping [Lev90] on the GPU. Using these techniques, the

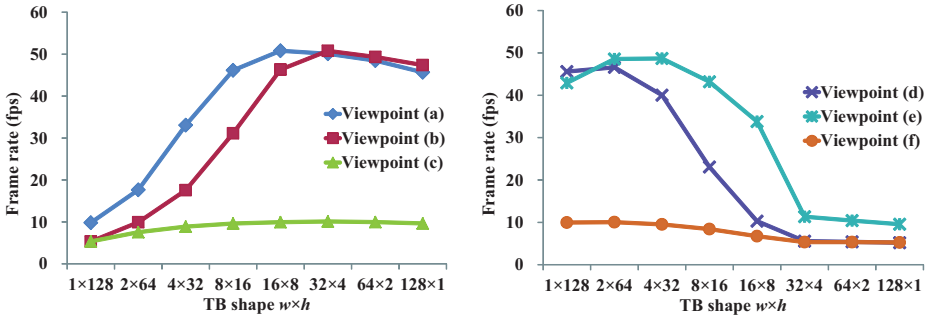


Figure 7: Frame rates with different shapes of TBs. (a) Results for three viewpoints (a)–(c), which are efficient with horizontal warps. (b) Results for the remaining viewpoints (d)–(f), which are efficient with vertical warps.

rendering performance is increased by a factor of 3. These techniques intend to reduce the amount of data access while our optimization strategy reduces the latency of data access. A similar technique is presented by Rijters *et al.* [RV06], who employ an octree data structure on the GPU.

An optimization strategy is presented by Ryoo *et al.* [RRS+08] 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.* [LZS09] 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.

7 Conclusion

In this paper, we presented an acceleration method for texture-based ray casting on the CUDA-enabled GPU. Our method increases the hit rate of the texture cache by selecting the shape of TBs during rendering. This selection focuses on the geometrical relationship 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 experiments, we investigated the cache hit rate and the frame rate using six viewpoints. We found that our method increases the cache hit rate by approximately 20%. This higher locality achieves a frame rate of 46.6 fps, which is four times higher than that of a naive method

that uses TBs of a fixed shape. Future work includes further evaluation using other GPU architectures that are compatible with OpenCL.

References

[HS89] William Hibbard and David Santek. Interactivity is the Key. In *Proc. Chapel Hill Workshop Volume Visualization (VVS '89)*, pages 39–43, May 1989.

[KW03] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proc. 14th IEEE Visualization Conf. (VIS'03)*, pages 287–292, October 2003.

[Lev88] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lev90] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Graphics*, 9(3):245–261, July 1990.

[LZS09] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A Cross-Input Adaptive Framework for GPU Program Optimizations. In *Proc. 23th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'09)*, May 2009. 10 pages (CD-ROM).

[MIH04] Manabu Matsui, Fumihiko Ino, and Kenichi 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)*, pages 245–256, December 2004.

[MM05] John Montrym and Henry Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, March 2005.

[Mor66] G. M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. Technical report, IBM Ltd, Ottawa, Ontario, August 1966.

[NIH08] Daisuke Nagayasu, Fumihiko Ino, and Kenichi Hagihara. A Decompression Pipeline for Accelerating Out-of-Core Volume Rendering of Time-Varying Data. *Computers and Graphics*, 32(3):350–362, June 2008.

[NVI10] NVIDIA Corporation. CUDA Programming Guide Version 3.2, September 2010.

[PF05] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, Reading, MA, March 2005.

[RRS+08] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen mei W. Hwu. Program optimization carving for GPU computing. *J. Parallel and Distributed Computing*, 68(10):1389–1401, October 2008.

[RV06] Daniel Rijters and Anna Vilanova. Optimizing GPU Volume Rendering. *J. WSCG*, 14(1/3):9–16, January 2006.

[TIH03] Akira Takeuchi, Fumihiko Ino, and Kenichi Hagihara. An Improved Binary-Swap Compositing for Sort-Last Parallel Rendering on Distributed Memory Multiprocessors. *Parallel Computing*, 29(11/12):1745–1762, November 2003.