Cache-Aware Sampling Strategies for Texture-Based Ray Casting on GPU

Junpeng Wang* Virginia Tech
Fei Yang† Chinese Academy of Sciences
Yong Cao‡ Virginia Tech

ABSTRACT

As a major component of volume rendering, the ray casting algorithm is memory-intensive. However, most existing texture-based volume rendering methods blindly map computational resources to texture memory and result in an incoherent access pattern, causing low cache hit rates in certain cases. The distance between samples taken by threads of the same scheduling unit (e.g. a warp of 32 threads in CUDA), of the GPU is a major factor that affects the texture cache hit rate. Based on this fact, we present a new sampling strategy, i.e. warp marching, which displays a novel computation-to-core mapping. In addition, a double buffer approach is introduced and special GPU operations are leveraged to improve the efficiency of parallel executions. To keep a roughly constant rendering performance when rotating the volume, we change our warp marching algorithm, so that samples can be taken along different directions of the volume. As a result, varying texture cache hit rates in different viewing directions are averaged out. Through a series of micro-benchmarking and real-life data experiments, we rigorously analyze our sampling strategies, and demonstrate significant performance enhancements over existing sampling methods.

1 INTRODUCTION

Scientific simulation often generates large scale volumetric datasets, which are represented by discrete samples and rendered with the ray casting algorithm. Interpolation among these discrete samples is one of the most frequent operations. Since the texture memory of GPU provides highly optimized tri-linear interpolation, it is the obvious choice for data storage in volume rendering. Numerous rays with a large amount of samples make the texture memory access a bottleneck of the ray casting, especially when dealing with large volumes. Cache hierarchy for the texture memory is built to alleviate this performance issue. In modern GPUs, such as NVIDIA's Kepler series, a cache-hit can be hundreds of times faster than a cache-miss. In this paper, we focus on optimizing texture cache performance, especially in a parallel computing environment.

On the parallel architecture of GPU, massive threads are executed concurrently. These threads are divided into atomic groups when mapped to the GPU hardware. For example, in CUDA [11], 32 threads form a warp and it is mapped to one stream multiprocessor of the GPU. Threads of a group process the same instructions and synchronize each step. For texture-based ray casting, the distance between samples taken by such a group of threads has significant consequences on the texture cache hit rate.

In this paper, we try to minimize the memory stride accessed by an atomic group of threads rather than every single thread of the GPU. To achieve this, we propose a new sampling strategy that introduces a novel computation-to-core mapping. Our strategy provides cache-friendly texture memory access patterns and significantly improves the cache performance, in certain viewing directions. To enhance the GPUs' resource utilization, a double buffer accumulation approach for the new sampling strategy is proposed. Also, the power of the newly introduced warp-level operations for the recent NVIDIA Kepler architecture is leveraged. We further average out the performance variations from different viewing directions by concurrently taking samples along different directions of the volume. In summary, three major contributions have been introduced in this paper:

1. We analyze the influence of cache coherence on the ray casting algorithm through a set of benchmarks, and introduce a new sampling strategy to improve the texture cache hit rate in certain viewing directions.

2. Two hardware-oriented optimizations, *double buffer accumulation* and *warp-level operations*, are presented to efficiently utilize the computing resources of GPUs.

3. Meanwhile, an approach that maintains roughly constant rendering frame rates regardless of the viewing direction is also proposed.

2 BACKGROUND AND RELATED WORK

Due to the fast hardware-accelerated linear interpolation, texture memory becomes the best choice to store volume data on GPU. The volume mapped to the 3D texture is organized as many 2D slices and inside each slice, z-curves [10, 19] are used to optimize the 2D spatial locality (Figure 1).

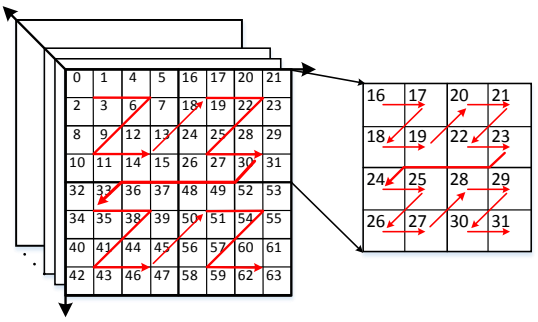

Figure 1: Recursive z-curve hierarchy of the texture memory.

We focus on the CUDA-capable NVIDIA GPU to illustrate our new sampling strategy in this paper. The massive threads of the GPU work in a single instruction multiple data (SIMD) mode. They are organized as grids of blocks. Thread blocks are further divided into warps of 32 threads when mapped to the GPU hardware. A warp of threads is an atomic scheduling unit. If such a group of concurrent threads access the texture memory at close memory locations, high texture cache hit rates are expected [11]. However, when accessing across many slices or covering a big memory stride, the texture cache hit rate will drop dramatically. Our new sampling strategy tries to minimize the memory strides inside a warp of threads.

*e-mail: junpeng@vt.edu
†e-mail: yangfei09@mails.gucas.ac.cn
‡e-mail: yongcao@vt.edu

Ray casting [5] is a major component of the direct volume rendering. The 3D texture-based implementation of this algorithm was proposed in the early 1990s. Examples of such work include that by Cullip and Neumann [2] and Cabral et al. [1]. Existing studies [3, 17] have demonstrated that view-aligned approaches with 3D textures [18] are superior to axis-aligned approaches with 2D textures [14] in several aspects. However, view-aligned approaches suffer from the penalty of texture cache. We focus on view-aligned approaches and propose new sampling strategies to alleviate this cache penalty.

The basic principle of volume rendering can be given by the volume rendering integral. A description of it can be found in [3], which also gives a good overview of the basic methods used in volume ray casting. The volume rendering integral can be approximated with discrete algorithms that accumulate the color of a ray by samples or by segments. In the latter case, pre-integration methods are frequently used, such as [4, 6]. No matter which method is used, the basic operation on the fly is alpha blending [13]. The blending operation is associative [8]. Ma et al. [9] take advantage of this property and propose a binary-swap compositing algorithm to effectively utilize the computing nodes of distributed systems. Yu et al. [20] extend this work by proposing the 2-3 swap image compositing. In this paper, however, we also use the associative property to composite samples along rays. Our composition is inside a warp of threads. To keep more computing units (GPU threads) active, a double buffer approach is proposed. This approach feeds more work to each thread of the GPU and works in a pipeline manner.

Sugimoto et al. [16] analyze the z-curve pattern and prove the memory stride ratio along three directions of the 3D texture is 1:2:6. They propose a dynamic approach by adjusting the thread block shape, so that threads of the same warp cover a smaller memory stride. However, due to the sampling limitation, their dynamic approach cannot guarantee a high texture cache hit rate in all viewing directions. Weiskopf et al. [17] design an algorithm that keeps roughly constant frame rates regardless of the viewing direction by reorganizing 3D volumes into bricks with different orientations. This approach has to pre-process the volume data, which may not be feasible in certain scenarios, such as for in-situ visualizations [7]. Also their approach requires the entire volume to be in the view frustum. Choosing a proper resolution for bricks and the overlap between neighboring bricks make the solution complicated. In this paper, we propose an alternate approach, which overcomes these shortcomings.

3 Motivation

Our work is directly motivated by the cache performance of volume rendering. The distance between samples taken from volumes is a key factor that impacts the texture cache hit rate. Here we show two major factors that affect this distance and design a benchmark to demonstrate their effects.

3.1 Sampling Distance

Viewing Direction (Volume Orientation): In the existing GPU implementations of the ray casting algorithm, samples taken by different threads are fitted into a plane, *sampling plane*. The plane goes through the volume along a viewing direction from front to back. Distance between samples on this plane is affected by the viewing direction. When the direction is perpendicular to 2D slices of the volume (Figure 2, *facing XY*), samples are close to each other in the texture memory. However, when the direction is parallel to the image stack (Figure 2, *facing ZY*), samples cover different image slices and are far from each other in the memory space.

Ray Distance (Volume Image Ratio): The ratio between volume dimensions and projected image resolutions also influences the distance between samples on the sampling plane. For example, when

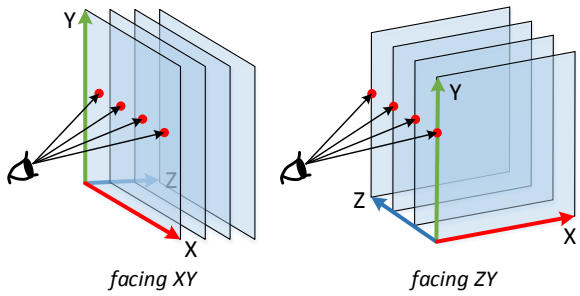

Figure 2: Effects of different viewing directions.

rendering a 1024^3 volume and facing the *XY* plane (shown in Figure 2) of the volume, if the projected image resolution is 512^2, the distance between pixels is the length of 2 voxels in orthographic projection (assume the spacings along three dimensions of the volume are the same). If one ray maps to one pixel, the ray distance is also the length of 2 voxels. Varying projected image resolutions result in different ray distances, as well as different distances between samples taken from these rays.

3.2 Benchmark Design

We design a benchmark to demonstrate the effects of these two factors. The benchmark program is a standard implementation of the semi-transparent volume rendering, such as the CUDA sample *'volumeRender'*. In this benchmark, one ray is mapped to one pixel of the projected image and one GPU thread works on such a ray to accumulate color and opacity values for a pixel. Samples taken by a warp of threads form a line or a plane (depends on the thread block shape) that is always perpendicular to the viewing direction. This warp line or warp plane goes through the volume from front to back to take samples and accumulates color and opacity values for a warp of pixels. We call this sampling strategy the **Standard** sampling strategy throughout the paper and compare it with our new sampling strategies.

To accurately collect the texture cache hit rate, several changes are introduced to the benchmark program. The first one is to disable the early ray termination and have rays march a fixed number of steps, 512 in this case. This guarantees a fixed amount of workload and rules out the effects of different transfer functions. Most of the existing volume rendering implementations put transfer functions into the texture memory, because linear interpolation is needed during the classification [3]. The size of the transfer function is usually small, but it is accessed frequently. In order to reflect the texture cache hit rate from only accessing the volume data, the second change is to put the transfer function into the constant memory (or the shared memory) of the GPU. Third, an orthographic projection is used to keep a constant ray distance during marching. Finally, to eliminate the impacts from different volume data sets, an 8-bit 1024^3 volume with random density values is generated. Sampling distance along rays is always fixed at 0.5 voxel length throughout the paper. The projected image resolution is fixed at 512^2, so all executions trace 512^2 rays. Distance between rays is varied by tracing different percentages of the volume. For example, if the 512^2 rays cover the entire volume, distance between neighboring rays is 2 voxels length. However if they trace a quarter of the volume, distance between rays is 1 voxel length. We use 256 threads per block for all experiments (both benchmark and real-life data experiments) in this paper.

Figure 3 shows results collected from the benchmark. As the distance between rays increases, the texture cache hit rate drops when facing the *ZY* plane, because samples from neighboring rays

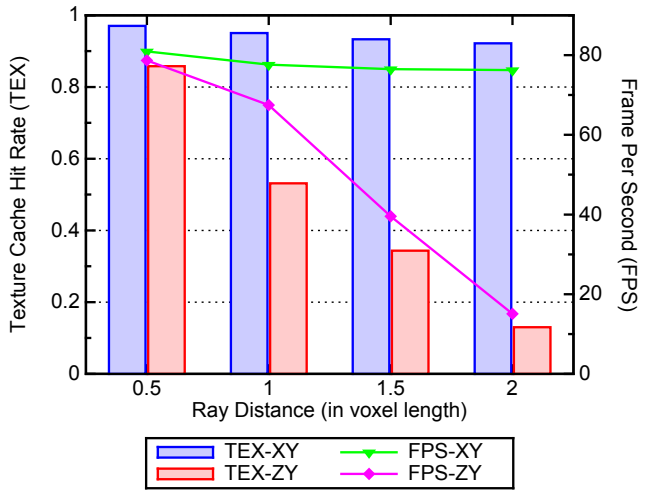

Figure 3: Effects of different viewing directions and different ray distances. The thread block shape is 16×16.

cover more 2D slices of the volume. The decline is insignificant when facing the *XY* plane, because samples are still on the same slice and the memory stride is small. Texture cache hit rates (metric *tex_cache_hit_rate*) and L2 cache hit rates (metric *l2_texture_read_hit_rate*) reported in this paper are collected by the CUDA Profiler Tools Interface (CUPTI) [12] on an NVIDIA GTX GeForce TITAN GPU with 6 GB device memory. The CUPTI introduces some performance overheads, so frame rates are collected from separate executions with the CUPTI disabled.

4 Warp Marching Algorithm

To address the cache related performance issues in the standard sampling mechanism, we introduce a novel parallel sampling strategy called *warp marching*. This approach introduces a unique computation-to-core mapping in the parallel algorithm design. It significantly mitigates the effects of viewing direction and ray distance to the texture cache performance.

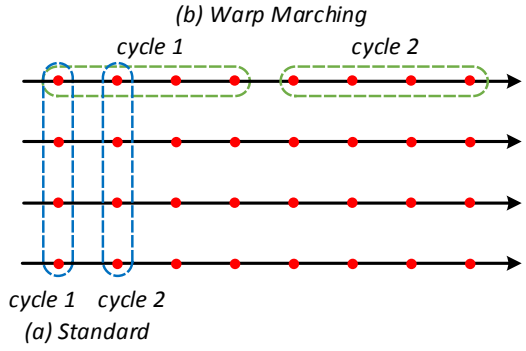

Figure 4: Different sampling strategies: (a) a warp of threads work on a warp of pixels; (b) a warp of threads work on one pixel. The warp size is 4 in this figure.

The new strategy samples the volume along rays. Figure 4 (*b*) demonstrates this approach, in which a warp of threads work on one pixel (or ray). Color and opacity blending for a pixel is done in parallel inside the warp. During each cycle, the new sampling strategy marches a warp of samples along the ray, so we call it warp marching. The algorithm works along the same lines as [8], which breaks a ray into a series of segments and composites these ray segments.

4.1 Algorithm Design

In general, there are two steps in the ray casting algorithm: (1) ray bounding-box intersection tests; (2) marching rays and accumulating color and opacity values. We only focus on the second step when applying our new sampling strategy.

The warp marching algorithm needs to handle two cases: (1) the samples taken by a warp of threads, all have density values of zero; (2) some or all samples have non-zero values. A binary bit is used for each thread to represent the state of a sample and the warp voting function, i.e. `__ballot()`, is used to reflect the state of the entire warp. For case (1), the warp voting function returns zero and the warp of samples are skipped. For case (2), a non-zero value is returned and contributions of the warp of samples are needed to be integrated to the final pixel. The associative property of the front to back color blending has been verified [9]. Based on this property, blending a warp of samples can be done using a reduction method, as illustrated in Figure 5. For simplicity, we assume 8 threads per warp. Step 1 of each warp cycle is to take samples and fill the buffers of threads with color and opacity values. Reduction starts from step 2 and continues till step 4. One extra memory unit is required for a warp to store the reduction value (intermediate result). In our implementation, a `float4` variable is used to store the RGBA value of each sample, so 9 such `float4` (8 for 8 threads of the warp and 1 for the intermediate result) are needed in total, and they are allocated from the shared memory. Since each thread only needs one buffer from the shared memory, we call this approach single buffer shared (SBShared) warp marching.

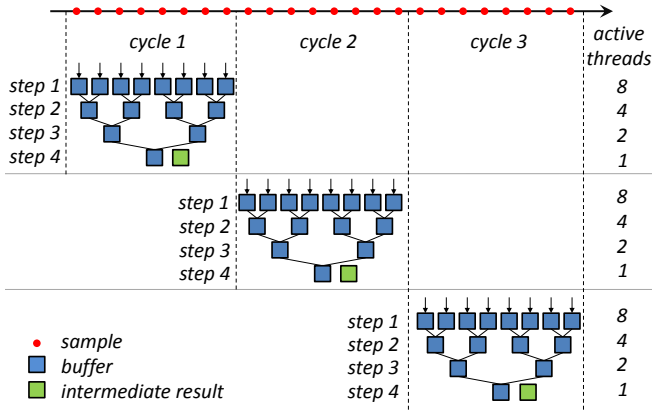

Figure 5: Single buffer approach (8 threads per warp).

4.2 Hardware-Oriented Optimization

Although reductions in the SBShared approach are performed in parallel, the number of active threads reduces by half after each step. To keep more threads active, a double buffer approach is introduced. Furthermore, we consider efficient warp shuffling operations to accelerate the communication among threads of the same warp.

4.2.1 Double Buffer Approach

In the SBShared approach, the number of active threads decreases because the workload is reduced by half after each step. To alleviate this problem, we delay the blending process until a warp receives

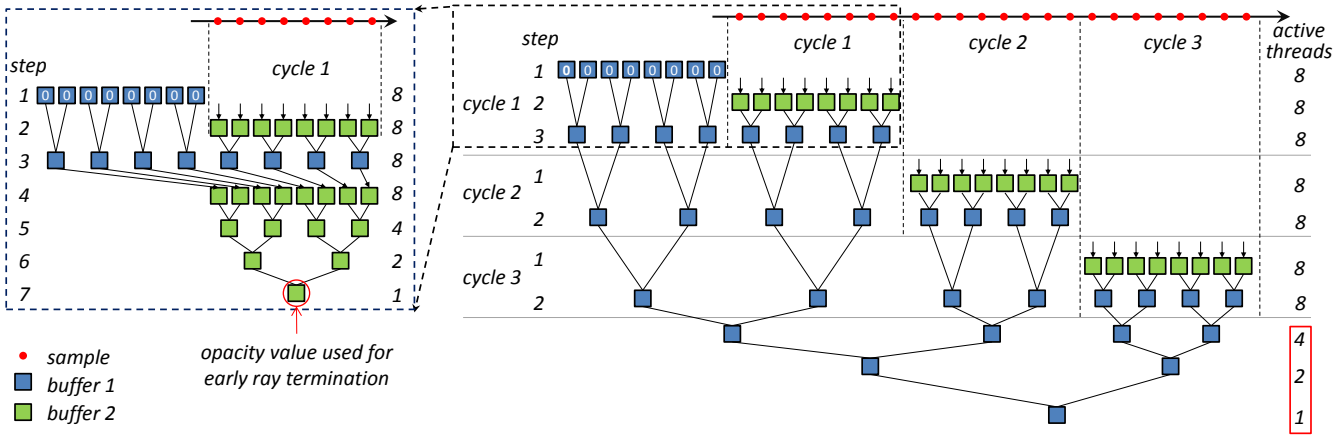

Figure 6: Double buffer approach (8 threads per warp). Sub-figure on the left shows the early ray termination mechanism.

new data. Before the reduction happens at step 2 (Figure 5), the warp of threads takes another warp of samples (Figure 6: cycle 1, step 2). Each thread needs two buffers to store two sample values now. Since these buffers are from the shared memory, we call this approach double buffer shared (DBShared) warp marching.

The double buffer approach is illustrated in Figure 6. In the first cycle, all threads of the warp initialize their first buffer to 0 (cycle1, step 1). Then threads of the warp take samples, derive color and opacity values from the transfer function and load them to their second buffer (cycle1, step 2). Now, every warp has 16 buffer units with values. 8 threads perform one step reduction and write the results back to their first buffer (cycle1, step 3). In these steps, all threads of the warp are active. In the second cycle, threads of the warp load color and opacity values to their second buffer (cycle2, step 1). After that, again, all threads work on one step reduction and write results to their first buffer (cycle2, step 2). All cycles from the third cycle onwards are similar to the second. In order to derive the final integrated value, some threads in the last warp cycle will be inactive (as shown in the red rectangle), but most of the time, the whole warp is active.

One problem for the DBShared is that the early ray termination does not work, because the exact integrated value cannot be derived at each warp cycle. In other words, the DBShared approach delays the termination of a ray by 3 warps ($\log_2 8$). Compared to implementations with the early ray termination, the DBShared performs 24 extra texture fetching operations. Fortunately, the early ray termination only relies on accumulated opacity values, so we can still do the "delayed blending" for color values, though the accumulated opacity values will have to be derived at each warp cycle. The sub-figure on the left of Figure 6 shows this repair. After step 3, opacity values are copied from buffer 1 to buffer 2 and the opacity blending is then performed there. Instead of blending 8 `float4` (in the SB-Shared), the DBShared blends 8 `float`. So it reduces the amount of work that few threads of the warp will work on. Based on the final blended opacity value, the algorithm decides whether the ray should be terminated or not.

4.2.2 Warp-Level Operation

Our warp marching strategy can benefit from efficient warp-level operations. In previous descriptions, we have shown how the warp voting function helps in reflecting the state of the entire warp. Here we demonstrate how shuffling operations help in efficient data communication among threads of the same warp. In these shuffling approaches, data is not read from the shared memory but is shuffled from the registers of other threads.

Single Buffer Shuffling (SBShuffle): Instead of allocating 9 `float4` from the shared memory, the SBShuffle approach allocates a `float4` from the registers of each thread. One thread of the warp needs to create one extra `float4` to store the intermediate result and this result will be broadcast (`__shfl()`) to all other threads of the warp at each iteration. Alternately, all threads can have a copy of the intermediate result. In that case, the broadcasting is avoided, but more registers are required. Algorithm 1 illustrates how the reduction (Figure 5, step 2 to 4) is performed by shuffling operations inside a warp. The shuffling width, i.e. variable *width* in Algorithm 1, is the number of threads in the warp.

Algorithm 1 Blending with shuffling operations.

```
1: for i ⇐ width/2; i≥1; i/ =2 do
2:     // __shfl(variable, id, width)
3:     sample1 ⇐ __shfl(color, threadIdx.x*2, width)
4:     sample2⇐__shfl(color, (threadIdx.x*2)+1, width)
5:     if threadIdx.x<i then
6:         color⇐sample1+(1−sample1.w)*sample2
7:     end if
8: end for
```

Double Buffer Shuffling (DBShuffle): For the double buffer shuffling approach, each thread creates two `float4` from its register space. The accumulated opacity value (for early ray termination) can be stored in the second buffer of the first thread. In the reduction process (Figure 6: cycle 1, step 3), all threads of the warp shuffle their buffer 1 values to the first half warp of threads, and this half warp performs one step reduction there. Then all threads shuffle their buffer 2 values to the second half warp of threads, and this half warp processes another reduction. Divergent branching happens here due to different shuffling sources. Consequently, this step is divided into 2 steps and half warp is active in each.

4.3 Benchmark Result

The warp marching algorithm and its optimizations are integrated to the benchmark we designed in section 3. All experiment settings are the same, unless explicitly mentioned. The performance improvements from two hardware-oriented optimizations are described first. We then establish how the new sampling strategy addresses the two problems mentioned in section 3.

Optimization: The results shown in Figure 7 indicate the shuffling approaches are always better than the shared memory approaches. Double buffer approaches are better than single buffer approaches

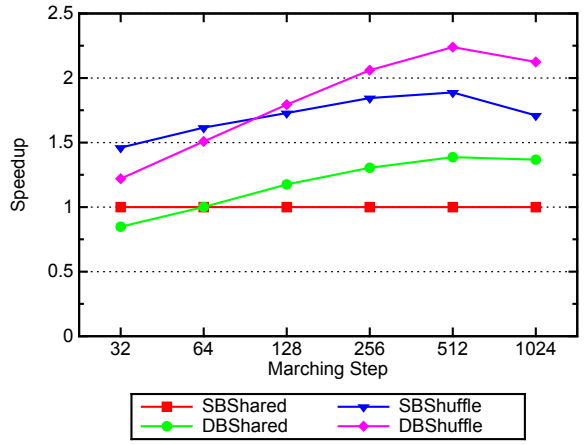

Figure 7: Speedup of different optimizations. Results are collected when facing the *ZY* plane of the volume.

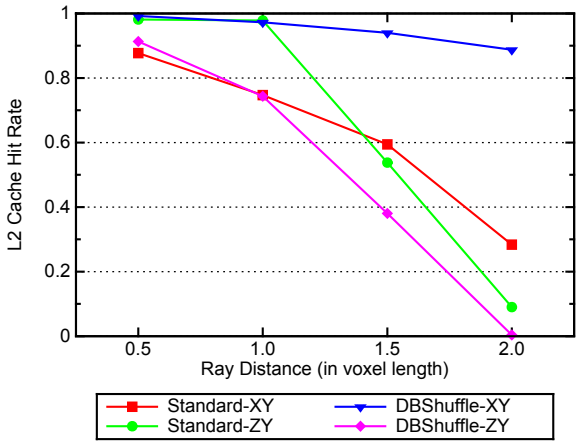

Figure 9: L2 cache performance.

when rays march by taking a larger number of steps. The number of active threads in the final part of the double buffer approach still lessens by half at each step (Figure 6). However, this only happens in the last warp cycle. When rays march more steps, the percentage of this part becomes small. On the contrary, this problem happens in every cycle of the single buffer approach. As rays march more steps, the time that threads are inactive increases, so double buffer approaches become better.

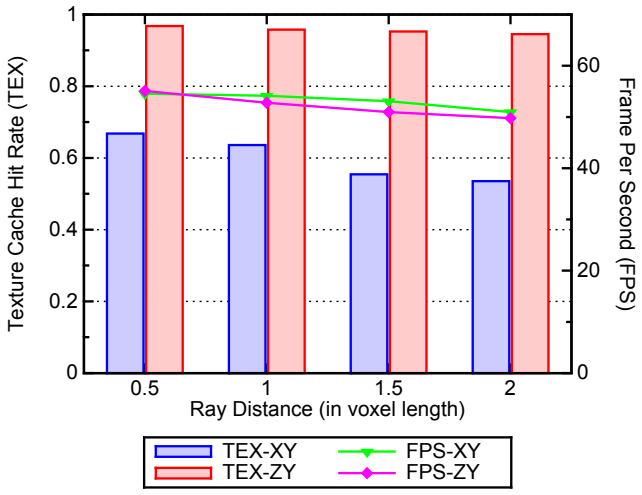

Figure 8: Effects of different viewing directions and different ray distances in the DBShuffle approach.

Viewing Direction and Ray Distance: Figure 8 shows the results of the DBShuffle (Rays march 512 steps, so we use the DBShuffle). The DBShuffle approach is insignificantly affected by the ray distance. However, similar to the standard sampling strategy, it is also affected by the viewing direction. Facing the *ZY* plane is better than facing the *XY* plane of the volume. This is because, when facing *ZY*, samples are on the same slice. Meanwhile, since marching step length along rays is fixed at 0.5 voxel length (usually less than 1 voxel length to satisfy the Nyquist frequency), the texture cache hit rate of facing *XY* case is not dramatically low.

The rendering performance when facing the *ZY* plane of the volume is not significantly better than that when facing the *XY* plane, though the texture cache hit rate is obviously higher. To explain this problem, we have to introduce the performance of L2 cache. Figure 9 shows the L2 cache hit rate of read requests from texture cache (CUPTI metric *l2_texture_read_hit_rate*) of the standard and the DBShuffle approaches. When using DBShuffle and the view is facing *XY*, the L2 cache hit rate is only marginally affected by the ray distance. In Figure 8, although the texture cache hit rate when facing *XY* is not as good as it was when facing *ZY*, facing *XY* case always has a high L2 cache hit rate. When requested data is missing from texture cache, the data can quickly be found from L2 cache, so the performance is still good. On the contrary, for the standard sampling method, when texture cache hit rate drops (Figure 3), L2 cache hit rate also drops, so the performance decreases.

5 3D Warp Marching

We have shown that the texture cache hit rate can be optimized when facing certain viewing directions. However there is not a simple approach that can maintain a good and roughly constant rendering performance when taking all viewing directions into consideration. Weiskopf et al. [17] try to address this problem by reorganizing volumes to small bricks with varying directions. Instead of reorganizing data, we take a different approach by distributing samples taken by a warp of threads along different directions. In this case, warp shape is a 3D volume.

In order to organize the warp shape as a volume, two small modifications are applied to the warp marching (only the shuffling approaches are discussed here, since they perform better than the shared memory approaches). The first modification is to reduce the number of samples taken along rays (depth direction) by a warp of threads. This can be done by changing the shuffling width, i.e. the variable *width* in Algorithm 1. Changing this width breaks a warp of threads into several groups with same number of threads in each. One such group is mapped to one pixel (or ray) in ray casting and the number of groups is the number of pixels that the warp will cover. The second modification is to organize these pixels into a 2D plane, which can be done by carefully organizing the shape of the thread blocks. For example, if the warp size is 32 and the shuffling width is 8, the warp will cover 4 pixels (or rays). 8 threads work on one ray and the number of samples taken along the viewing direction is 8. The thread block size is 256, so one block will cover 32 pixels. When organizing these pixels into 2 columns by 16 rows, one warp will process two rows of them. The warp shape in this case is 2×2×8 (i.e. the number of samples taken along the horizon-

tal, the vertical and the depth direction is 2, 2 and 8), and the block shape is 2×16×8.

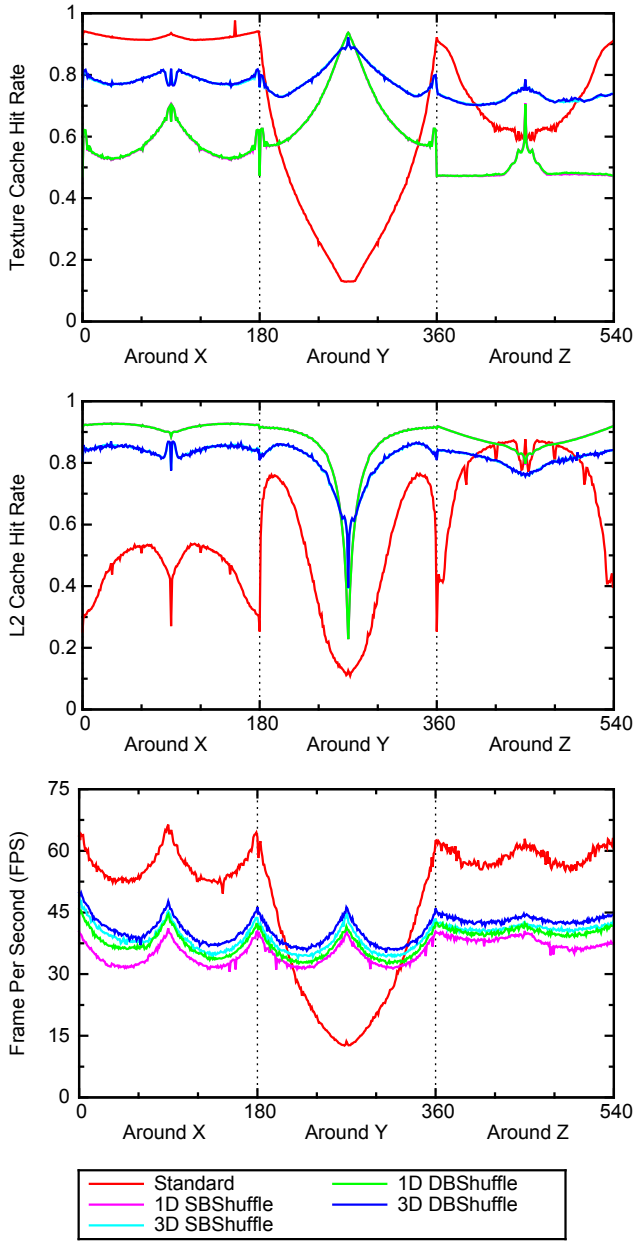

Figure 10: Results of rotating the volume around X, Y and Z 180°. The texture cache hit rate and the L2 cache hit rate from SBShuffle and DBShuffle implementations are the same, so lines are overlapped in the top two figures.

Except the number of samples along different directions, the distance between neighboring samples should also be taken into consideration. In the benchmark settings, the distance between rays is 2 voxels length, and distance between samples along rays is 0.5 voxel length. So the sampling distance along the horizontal, the vertical and the depth direction is the length of 2 voxels, 2 voxels and 0.5 voxel respectively. Multiplying this distance with the number of samples taken along each direction is the number of voxels that one warp will access along that direction. For example, warp shape 4×2×4 covers 8, 4 and 2 voxels along three directions. There are six possible 3D warp shapes: 4×4×2, 4×2×4, 2×4×4, 8×2×2, 2×8×2 and 2×2×8. Among them, shape 2×2×8 covers the same number of voxels (4 voxels) along three directions. So, its variance of texture cache hit rate in different viewing directions should be small. We use the same benchmark to demonstrate the texture cache hit rate, L2 cache hit rate and overall rendering performance of this 3D warp shape, in comparison with the standard method (warp shape 16×2×1) and the warp marching method (warp shape 1×1×32) in Figure 10. Results are collected when rotating the experiment volume around X, Y and Z axis 360° respectively. Due to symmetry, only 180° rotation results are shown. To differentiate the original warp marching from 3D warp marching, we call it 1D warp marching, since the warp shape is 1D.

Figure 10 gives both the SBShuffle and the DBShuffle results for 1D and 3D warp marching. The texture cache hit rate and the L2 cache hit rate of these two types of implementations are exactly the same (lines are overlapped). As we expected, the texture cache hit rate of 3D warp marching shows slight variances during rotation. Consequently, the 3D warp marching maintains a roughly constant frame rate. On the contrary, the standard method demonstrates a big performance variance during rotation. A deep trough appears in the results due to its poor texture cache and L2 cache performance when rotating around the Y axis. The texture cache hit rate of 1D warp marching is not high when rotating around the X or the Z axis, but its L2 cache performs very well in these two rotations. As a result, a roughly constant frame rate is also achieved. The projected image resolution is not always 512^2 during these rotations. When the volume is rotated to 45° or 135°, the projected image resolution becomes the largest. Heavier workloads lead to lower frame rates at these angles.

For the standard method, the texture cache hit rate also drops when rotating around the Z axis. The thread block shape of the standard method is 16×16, so a warp of 32 threads take 16 samples along the X axis and 2 samples along the Y axis when facing the XY plane. After rotating around Z 90°, 16 samples are taken along the Y axis and 2 samples are taken along the X axis by a warp. Memory stride along the Y axis is larger than that along the X axis [16], so texture cache hit rate drops.

6 Application

In this section, we describe how to apply our cache-aware sampling strategies to an existing volume rendering algorithm and demonstrate results with real-life data sets. The early ray termination is enabled and the transfer function is moved to the texture memory. Images are rendered with a resolution of 1024^2. To reflect the ray distance, we list the volume dimensions and the projected resolutions (*active resolutions*) in Table 1. The volume image ratios are also shown. However, since results in this section are collected in perspective projection, these ratios are not exactly the distances between rays.

		XY	ZY
vhf_head	Volume Dimension	2048×1216	1203×1216
	Projected Resolution	885×526	609×616
	Ratio	2.3×2.3	2.0×2.0
vhm_body	Volume Dimension	2048×1024	1878×1024
	Projected Resolution	494×247	1015×183
	Ratio	4.1×4.1	1.9×5.6

Table 1: Volume dimensions, projected image resolutions (active resolutions) and volume image ratios.

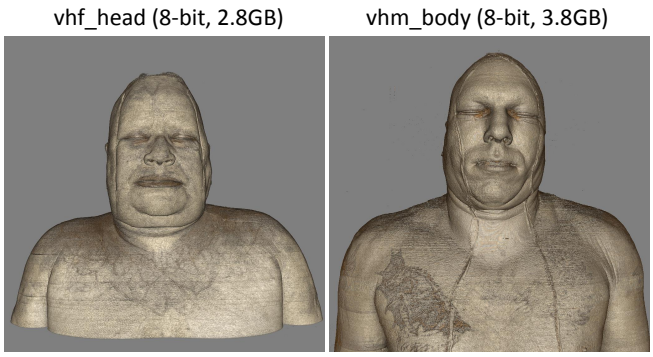

Figure 11: The Visible Human female (head) and Visible Human male (full body) data sets. Their dimensions are 2048×1216×1203 and 2048×1024×1878; spacing ratios along X Y and Z for them are 1:1:1 and 1:1:3 respectively.

Lum et al. [6] propose a high quality pre-integrated volume rendering algorithm with the standard sampling strategy. We replace the sampling strategy with our warp marching methods. There is no specific reason to choose this algorithm. Our sampling strategies can also be applied to other texture-based volume rendering algorithms. In order to do the pre-integrated volume rendering, each thread needs 2 density values (from front and back of the slab [15]). So, every thread of the warp takes one sample and shuffles (`__shfl_down()`) the sample to its neighboring thread. Figure 11 demonstrates the rendering results using the technique of Lum et al. with our 1D warp marching.

	vhf_head		vhm_body	
Facing	XY	ZY	XY	ZY
Standard	13.93	2.66	24.42	4.74
SBShuffle	7.98	10.07	20.60	17.72
DBShuffle	8.49	11.12	23.42	19.91
Speedup	0.61	4.18	0.96	4.20

Table 2: Frame rates (fps) of different sampling methods.

1D Warp Marching: Table 2 shows the results of using different sampling strategies in two viewing directions. When facing the XY plane of the volume, the standard method achieves higher frame rates. Based on our benchmark analysis, this is expected, because texture cache hit rate of the standard method is higher. The performance of warp marching is not as good as the standard one, but it is neither extremely poor. Since sampling distance along rays is fixed at 0.5 voxel length, the texture cache hit rate of warp marching in this viewing direction is not very low. In addition, L2 cache performs well (Figure 9). When facing the ZY plane, warp marching achieves about 4 times the speedup than the standard method. In this direction, warp marching has a high texture cache hit rate, whereas the standard sampling strategy suffers from its poor texture cache and L2 cache performance. For both data sets, the double buffer approach performs better than the single buffer approach (thread block shape of the standard sampling method is 16×16).

3D Warp Marching: Results of the 180° rotations around three axes are shown in Figure 12. We keep the entire volume inside the view frustum during these rotations. Since *vhm_body* has a smaller projected image resolution (the projected resolutions when facing the XY and ZY plane are shown in Table 1), its performance is better than the performance of *vhf_head* in both the standard sampling and the warp marching method. Compared to the standard method, both 1D and 3D warp marching maintain roughly constant frame rates during rotation.

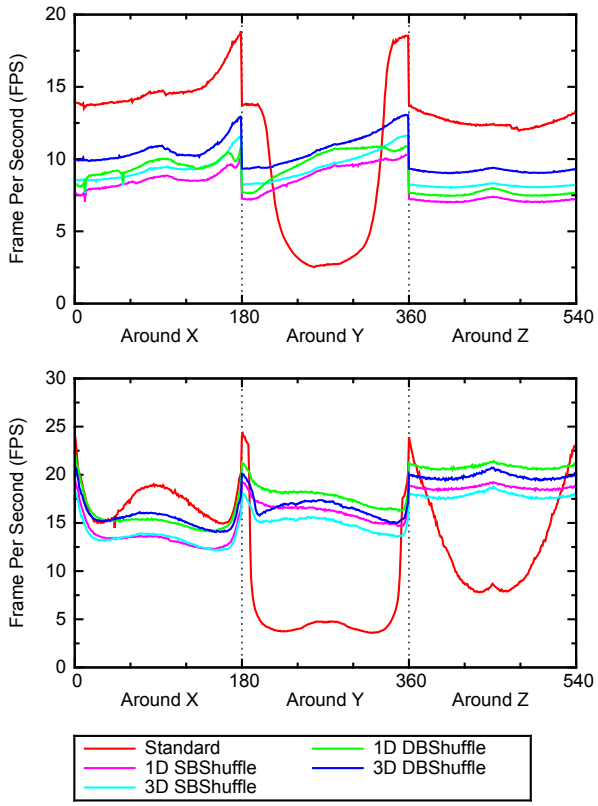

Figure 12: Results from 180° rotations. Top: results from vhf_head; bottom: results from vhm_body.

7 Conclusion and Future Work

In this paper, we demonstrate the significant impact of cache hit rate and design a cache-aware sampling strategy, i.e. warp marching, for the ray casting algorithm. The strategy uses the associative property of color blending in volume rendering and takes advantage of hardware-accelerated warp voting and warp shuffling operations. A double buffer approach is proposed to reduce the number of inactive threads. The 3D warp marching maintains a roughly constant texture cache hit rate regardless of volume orientation. Consequently, it only shows slight performance variances during rotation.

Our 1D and 3D warp marching algorithms are presented on a single-GPU workstation and data sets can be loaded into the device memory. Such a scenario is unlikely to happen when visualizing outputs from large scale scientific simulations, where data sets usually cannot be fitted into the memory of a GPU, and multiple GPUs or multiple nodes are involved in the computation and visualization. In the future, we will explore how the new computation-to-core mapping fashion can benefit these real-life visualization scenarios. Specifically, we would like to extend the warp marching to distributed systems and integrate out-of-core techniques to deal with the large size of volumes. Our discussion in this paper is limited to CUDA-capable NVIDIA GPUs. Extending the work to other types of GPUs, which may have varying number of threads in a warp, is also a good direction for future exploration.

REFERENCES

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, pages 91–98. ACM, 1994.

[2] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. 1993.

[3] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *Proceedings of the conference on SIGGRAPH 2004 course notes - GRAPH '04*, pages 29–es, New York, USA, 2004. ACM Press.

[4] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM New York, NY, USA, 2001.

[5] M. Levoy. Display of surfaces from volume data. *Computer Graphics and Applications, IEEE*, 8(3):29–37, 1988.

[6] E. Lum, B. Wilson, and K. Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proceedings Joint Eurographics-IEEE TVCG Symposium on Visualization*, pages 25–34, 2004.

[7] K.-L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, 29(6):14–19, 2009.

[8] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the 1993 symposium on Parallel rendering*, pages 15–22. ACM, 1993.

[9] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *Computer Graphics and Applications, IEEE*, 14(4):59–68, 1994.

[10] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM, Ottawa, Canada*, 1966.

[11] NVIDIA. Cuda c programming guide. *NVIDIA Corporation, July*, 2013.

[12] NVIDIA. Cupti user's guide. *NVIDIA Corporation, July*, 2013.

[13] T. Porter and T. Duff. Compositing digital images. In *ACM Siggraph Computer Graphics*, volume 18, pages 253–259. ACM, 1984.

[14] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM, 2000.

[15] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the symposium on Data visualisation*, pages 231–238, 2003.

[16] Y. Sugimoto, F. Ino, and K. Hagihara. Improving cache locality for ray casting with cuda. In *ARCS Workshops*, pages 339–350, 2012.

[17] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3D texture-based volume rendering. *Proceedings Computer Graphics International*, pages 604–607, 2004.

[18] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177. ACM, 1998.

[19] N. Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[20] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2–3 swap image compositing. In *High Performance Computing, Networking, Storage and Analysis. SC.*, pages 1–11, 2008.