



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

공학석사학위논문

Improving GPU-Speedup of Optimized Volume Rendering

최적화된 Volume Rendering의
GPU-Speedup 개선기법

2015년 8월

서울대학교 대학원
전기·컴퓨터 공학부
전 상 수

Improving GPU-Speedup of Optimized Volume Rendering

최적화된 Volume Rendering의 GPU-Speedup 개선 기법

지도교수 신영길

이 논문을 공학석사학위논문으로 제출함

2015년 4월

서울대학교 대학원

전기·컴퓨터 공학부

전 상 수

전 상 수의 석사학위논문을 인준함

2015년 6월

위 원 장 _____ 김 명 수 (인)

부 위 원 장 _____ 신 영 길 (인)

위 원 _____ 이 광 근 (인)

Improving GPU-Speedup of Optimized Volume Rendering

Sang-Soo Jun

Department of Electrical Engineering&Computer Science
Seoul National University

Abstract

This paper presents a speedup improvement method for optimized volume rendering in GPU platforms. First, from a set of experiments, we found that the speedup of volume rendering optimized with transparent voxel skipping *decreases* with dependency on the complexity of target images. In order to evaluate the complexity of volume images, we developed a new algorithm, called EVIC. Next, we present another new algorithm, called RBDV, that reduces the branch divergence in transparent voxel skipping by factoring out structurally similar code from branch paths in GPU programs. We empirically proved that this RBDV algorithm *increases* the GPU-speedup of transparent voxel skipping at least by 14%, improving it from x17.5 upto x20.0 or more, on average, for complex target images.

Keywords. image complexity, volume rendering, transparent voxel skipping, speedup, graphics processing unit (GPU)

Contents

Chapter 1. Introduction	7
Chapter 2. Background	9
2.1. Volume ray-casting	9
2.2. Optimization of volume rendering	11
2.3. GPU-based parallelization	13
2.4. Branch divergence	14
Chapter 3. Findings on Image Complexity Dependence	16
3.1. The complexity evaluation algorithm	16
3.2. Experimentation on image complexity	18
3.3. Analysis on image complexity	20
Chapter 4. Reducing Branch Divergence	24
4.1. The branch divergence reduction algorithm	24
4.2. Experimentation on branch divergence	28
4.3. Analysis on branch divergence	30
Chapter 5. Conclusion and Future Work	32
References	34
Abstract (in Korean)	37

List of Figures

Figure 1. Four basic steps of ray casting: (1) Ray Casting (2) Sampling (3) Shading (4) Compositing	10
Figure 2. Volume rendering techniques	12
Figure 3. EVIC algorithm	17
Figure 4. Three samples of original 2D slice target images in horizontal order: (1) Engine Block 512*512*512, (2) Metal Plate 512*512*512, (3) Abdomen 512*512*86	19
Figure 5. Rendered results of three target images with different complexities in vertical order: (1) Engine Block, (2) Metal Plate, (3) Abdomen	20
Figure 6. Speedup based on optimization techniques	21
Figure 7. Speedup based on image complexity	22
Figure 8. Code example of branch distribution	26
Figure 9. Applying branch distribution method for GPU-based volume rendering program	28
Figure 10. The TVS speedup based on image complexity	31

List of Tables

Table 1. Speedups of volume rendering techniques (with CPU/GPU time in sec)	20
Table 2. Speedup after applying branch distribution method	29

Chapter 1.

Introduction

Volume rendering is a visualization method for volumetric images. It is widely used for scientific visualization which requires simulation of realistic 3-dimensional data. The scientific visualization is applied to many widely recognized fields including medical imaging and industrial imaging. These fields requires volume rendering of such images, because it must help the user to better understand the inside and outside conditions of machine parts or patients.

Unfortunately, typical sizes of modern data are very large and will continue to increase in the future due to technological advances in acquisition devices.⁵⁾ Thus, processing these data sets efficiently is important,^{1,7)} so improving GPU-speedup has been a major research goal in volume rendering community for many years. When running ray casting on GPU platforms, the speedup is dependent on the amount of branch divergence incurred by the threads of kernel program.⁵⁾

This paper examines the speedup issues of volume rendering in GPU platforms with regard to its dependency with image complexity. First, from a set of experiments, we found that the speedup of volume

rendering optimized with transparent voxel skipping *decreases* with dependency on the complexity of target images. In order to evaluate the complexity of volume images, we developed a new algorithm, called EVIC. Next, we present another new algorithm that reduces the branch divergence in transparent voxel skipping by factoring out structurally similar code from branch paths in GPU programs.

The remainder of this paper is organized as follows. Chapter 2 provides background for understanding our work. Chapter 3 presents our own findings on the image complexity dependence on the GPU-speedup of optimized volume rendering and our EVIC algorithm that evaluates the complexity of volume images. Chapter 4 presents another new algorithm that reduces branch divergence in GPU-based volume rendering programs. Finally, we conclude our paper and give directions for future work in chapter 5.

Chapter 2.

Background

This paper presents a speedup improvement method for optimized volume rendering in GPU platforms. This chapter briefly explains the basic algorithm, called *ray casting*, of volume rendering. Next, we introduce two approaches that can increase its performance by optimizing the basic algorithm and parallelizing it on GPU platforms. Lastly, we finish this chapter after explaining the concept of branch divergence which is the main problem solved by this paper

2.1. *Volume Ray-Casting*

Volume ray casting (or simply ray casting) is the basic technique for volume rendering.^{12,13,15)} This processes volume data by tracing a path of light rays through pixels in an image plane. In its basic form, this algorithm is composed of *four* steps, as illustrated in Figure 1:^{13,14)}

1. *Ray casting*: For each pixel in the final image, a ray is casted through the volume.
2. *Sampling*: Along the casted ray inside the volume, equi-distant samples are selected. When evaluating the values of these

samples, it is necessary to interpolate them from its surrounding voxels, since the volume is not aligned with the rays most of the time, and samples are usually located in between voxels.

3. *Shading*: Each samples are coloured and lit according to their surface orientation and the light source location.
4. *Compositing*: Final colour value for that pixel is evaluated using front-to-back rendering equation.

However, unlike general surface rendering, volume rendering is used for visualizing 3-dimensional data with large volumes, which makes rendering time slow. So, in order to render images as real-time as possible, research in effective rendering became an important issue.¹⁾ There are two main stream approaches in the research. The first approach aims to avoid rendering empty regions in the images as much as possible;⁴⁾ the second approach is to utilize ever growing GPU technology to volume rendering.^{4,8)}

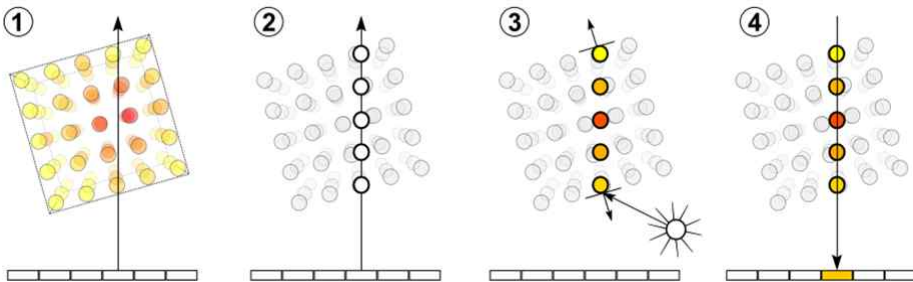


Figure 1 Four basic steps of ray casting: (1) Ray Casting (2) Sampling (3) Shading (4) Compositing.

2.2. Optimization of Volume Rendering

Optimization techniques of volume rendering (or simply optimization techniques) aim to avoid rendering empty regions as much as possible. We use two kinds of volume rendering techniques: one for non-optimized ray-casting and the other for three representative optimization techniques, as shown in Figure 2:^{3,9)}

1. *basic volume rendering* (BVR): This technique, as shown in Figure 2(a), represents the traditional volume rendering that does not apply any type of optimization to the original ray-casting method
2. *early ray termination*⁸⁾ (ERT): This technique, as shown in Figure 2(b), represents an optimized volume rendering that stops progression of each ray whenever an accumulated opacity reaches a high fractional value that is greater than zero and equal to one, say 0.98.
3. *empty space skipping*⁸⁾ (ESS): This technique, as shown in Figure 2(c), represents an optimized volume rendering that performs in the preprocessing stage. ESS divides the volume into sub-volumes, calculates its minimum and maximum colour values of each target sub-volume, and finally decide if each pair of values are within the range of *transfer function*.
4. *transparent voxel skipping*⁸⁾ (TVS): This technique, as shown in

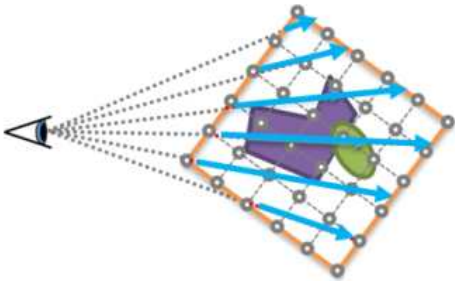
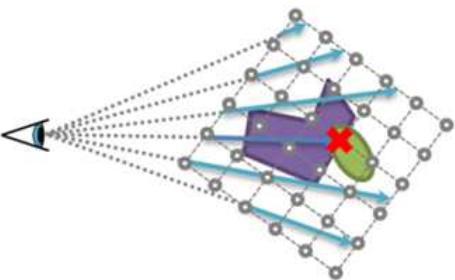
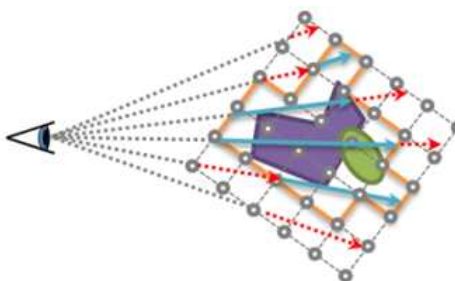
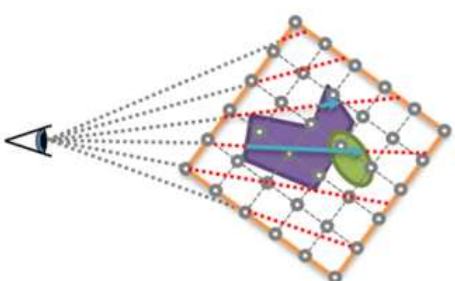
techniques	illustration
BVR	 A diagram illustrating Backward Volume Rendering (BVR). An eye icon on the left emits a series of rays (dotted lines) that pass through a 3D volume. The volume is defined by a grid of points (grey dots) and is divided into cells by orange lines. Blue arrows indicate the direction of light transport from the volume towards the eye. The volume contains two overlapping shapes: a purple one in the foreground and a green one in the background.
ERT	 A diagram illustrating Error Thresholding (ERT). Similar to BVR, it shows an eye and rays passing through a 3D volume grid. However, a red 'X' marks a point where a ray is terminated, indicating that the error threshold has been reached, and further rendering of that ray is skipped. The volume contains purple and green shapes.
ESS	 A diagram illustrating Expected Surface Sampling (ESS). It shows an eye and rays passing through a 3D volume grid. Red arrows indicate the sampling of surface points on the volume's boundary. The volume contains purple and green shapes.
TVS	 A diagram illustrating Truncated Volume Sampling (TVS). It shows an eye and rays passing through a 3D volume grid. Red arrows indicate the sampling of points within the volume. The volume contains purple and green shapes.

Figure 2. Volume rendering techniques

Figure 2(d), represents an optimized volume rendering that skips transparent voxels from rendering because they do not necessarily contribute to the final image.

We compare the performance of these four techniques in regards to their speedup on a GPU platform in the next chapter.

2.3. GPU-based Parallelization

Computational complexities are very high in volume rendering due to such large data. Fortunately, in ray casting, each light rays that goes through pixels are totally independent from one another, making it suitable for parallelization. The GPU-based volume rendering evaluates each pixel in the image plane^{11,17)} in parallel by allocating one core for each ray.

When comparing the performance between programs running on CPU and GPU, we use a measure called *speedup*. Although absolute running time is the ultimate measure of any program's performance, there are some useful relative measure, such as the speedup, that can provide insight into how well a parallel program is exploiting potential parallelism.⁴⁾ The speedup of a parallel program is typically defined as

$$S_p = \frac{T_1}{T_p},$$

where p is the number of processor cores, and T_k is the running time

on k cores. When T_1 is the execution time of a sequential version of the program, the S_p is called the *absolute speedup*. This paper employs this notion of the absolute speedup, and we will refer to it just as speedup for simplicity.

2.4. Branch Divergence

Threads are often bundled into fixed-size warps for executing them on a CUDA core; a set of threads within a warp must follow the same execution path. This means that all threads in a warp must execute the *same* instruction at the same given time. But when different threads within a warp does different things, this causes threads to diverge to different branch paths of executions.⁸⁾ In these cases, the warp serially executes each branch path resulting in overall performance loss, because a warp executes just one instruction at a time. The following code segment is a typical example of branch divergence.

```
1: for (every thread within a warp) {  
2:     thread_value = Array[tid];  
3:     if (thread_value > 10) {  
4:         variable = variable * 2;  
5:     }  
6:     . . . . .  
7: }
```

If a set of threads is supposed to execute the statement in line 4 that assigns a new value to the *variable*, at the next timing, the other set of threads with the boolean condition of false in the same warp must go through without executing the statement. On average, let's say half of the threads within a warp actually execute this branch condition, this means that the utilization of execution units are also just half.

In case of the four techniques discussed in Figure 2, only TVS causes additional branch divergence on parallel environments. This is because each ray has different voxels to skip; therefore each ray may execute one of two different tasks on a branch operation that must be serialized with each other.

Chapter 3.

Findings on Image Complexity Dependence

This chapter presents that the GPU-speedup of volume rendering optimized with three representative optimization techniques decreases with dependency on the complexity of target images from a set of experiments. In order to evaluate the complexity of volume images, we developed a new algorithm, called EVIC.

3.1. The Complexity Evaluation Algorithm

Bahnisch, Stelldinger and Kothe (2009) defined image complexity for 2D images as the number of edges that separates two regions from each other.¹⁾ To the best of our knowledge, so far there has been no completed work on how to define image complexity for volume images.

There are two difficulties for the users to evaluate the complexities of volume images. First, as seen in Figure 4, changes in the intensity of image values so frequently appear in each slice of the image, thus it is not easy for the user to define every discrete part that causes change in regions. Second, most volume images are composed of huge

number of image slices. Due to these reasons, it is impossible for users to manually evaluate the image complexity. Our new algorithm, called EVIC, extends the Bahnisch's definition of image complexity to our context in volume rendering, where EVIC stands for Evaluation of Volume Image Complexity.

This EVIC algorithm works at the sampling stage of ray casting. In its first stage of EVIC, we take all of the values of sampling points from the entire volume image and calculate the average of all sampled values. We set this average value as the threshold to determine if we meet the change of regions. Whenever the difference between two sampling values exceeds this threshold, we add one to the variable that is designated to signify a change between regions. Since target images usually differ in the number of slices, we have to divide the value stored in this variable by the number of slices. Figure 3 shows the pseudo code of this algorithm.

```

for (entire volume) {
    final_intensity = final_intensity + intensity[i];
}
threshold = final_intensity / n;

for (entire volume) {
    inten_diff = intensity[i] - intensity[i-1];
    if (inten_diff > threshold) region_change++;
}
final_complexity = region_change / slice_number;

```

Figure 3. The EVIC Algorithm

3.2. Experimentation on Image Complexity

We found that the GPU-speedup of volume rendering optimized with three representative optimization techniques decreases with dependency on the complexity of target images. To support it, we performed a set of experiments, in which we take three kinds of image complexities using three target images: Engine Block, Metal Plate, and Abdomen, as shown in Figure 4.

These images are taken from two representative applications of volume rendering. Engine Block and Metal Plate are taken from industrial applications, and Abdomen are done from medical applications. Using the EVIC algorithm introduced in the previous section, we evaluated all three target images and concluded that Engine Block has the biggest complexity by 21.3, the next is Abdomen by 16.6, and the last is Metal Plate by 8.2.

We performed the speedup experiments of the implemented code of the EVIC algorithm developed in Visual C++ 2012 using an Intel i7-2600 CPU with an NVIDIA GTX-680 GPU running under the Microsoft Windows 7 operating system . We performed twelve experiments and obtained the corresponding set of rendered images and their speedup data, as shown in Figure 5 and Table 1, respectively. In this table, the measured CPU-time and GPU-time in seconds represent

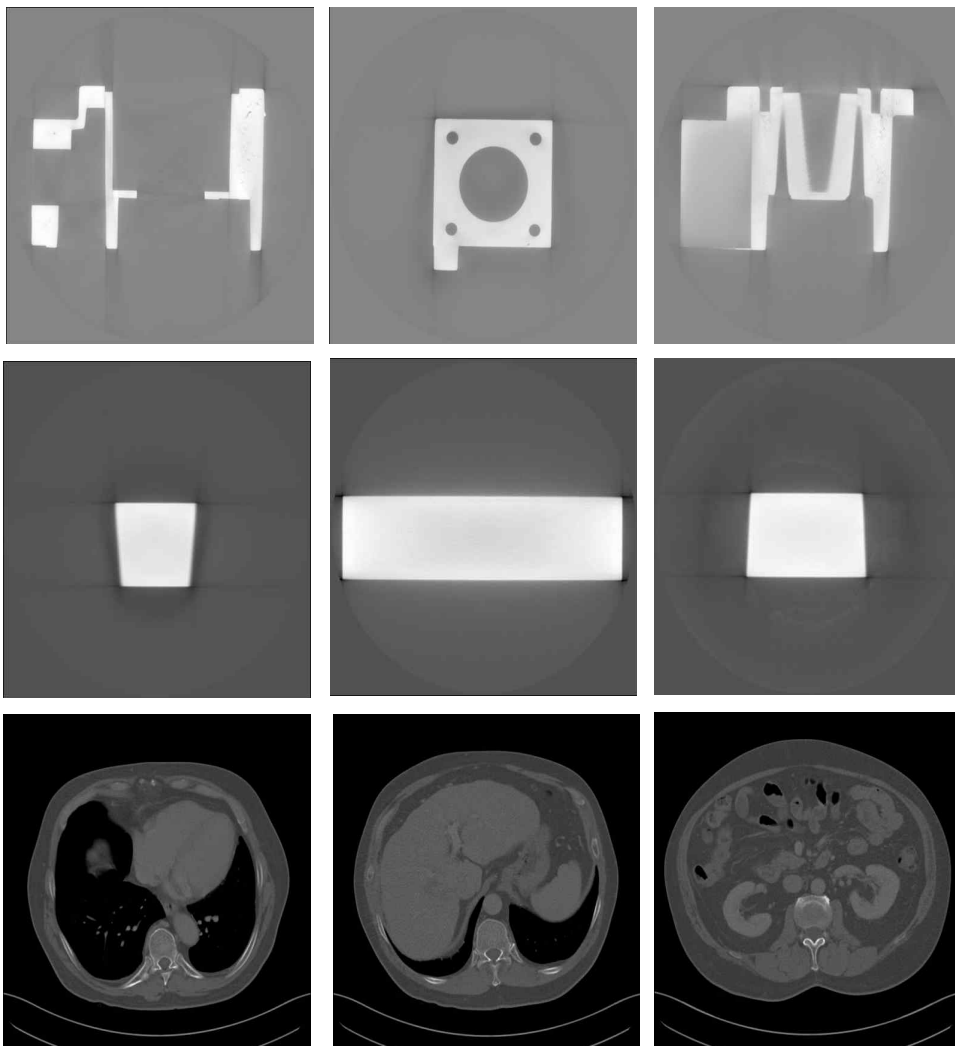


Figure 4. Three Samples of original 2D slice target images in horizontal order:
 (1) Engine Block 512*512*512, (2) Metal Plate 512*512*512, and (3) Abdomen
 512*512*86

the sequential and parallel running times of four employed techniques of volume rendering applied to three target images respectively.

We characterize the running times of volume rendering from two

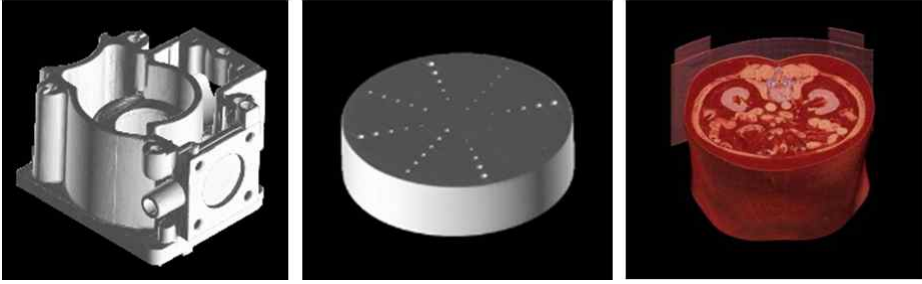


Figure 5 Rendered results of three target images with different complexities in vertical order: (1) Engine Block (2) Metal Plate (3) Abdomen

directions of viewpoints in the table: one viewpoint on every *image* to compare the speedups of four volume rendering techniques, and the other viewpoint on every *techniques* to compare the speedups of three complexities of target images.

3.3. Analysis on Image Complexity

Regarding the viewpoints on an individual image, as shown in Table 1 and represented in Figure 6, we obtained a performance order of

	Engine Block			Metal Plate			Abdomen		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
BVR	8.0	0.19	42.1	7.8	0.18	43.8	3.0	0.07	43.0
ERT	4.4	0.17	25.9	5.3	0.15	35.6	2.8	0.08	35.0
ESS	2.2	0.11	20.0	2.4	0.11	21.8	0.9	0.03	30.0
TVS	2.8	0.16	17.5	3.0	0.12	25.0	1.2	0.05	24.0

Table 1 Speedups resulted from the experiments (with CPU/GPU time in sec)

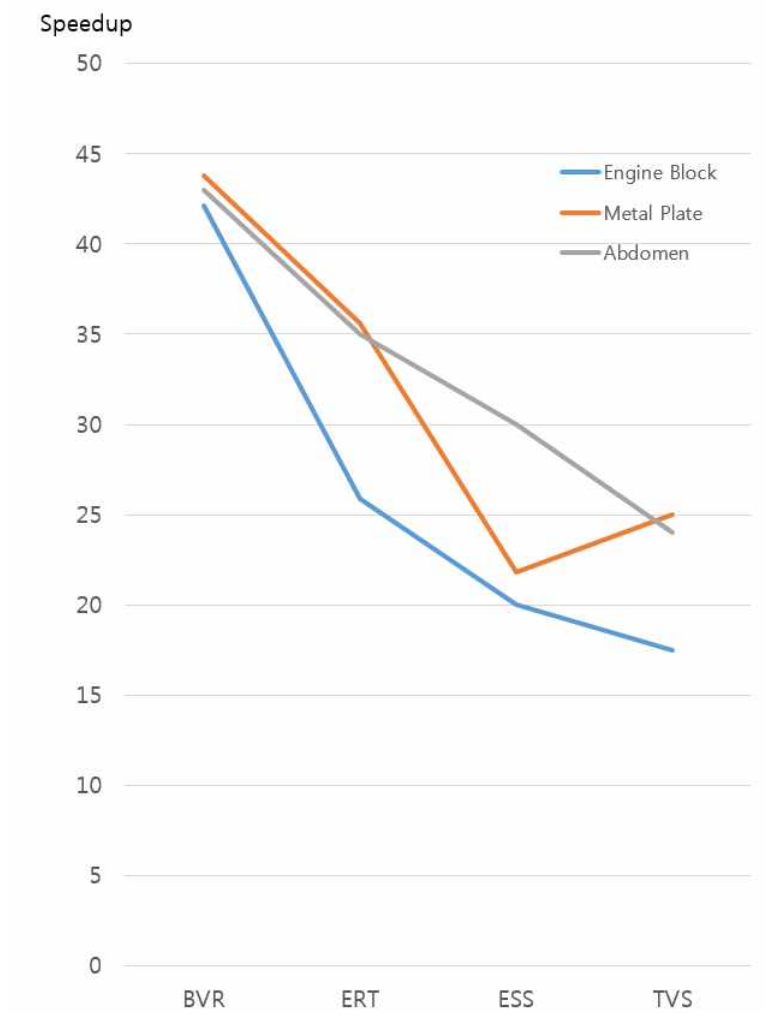


Figure 6. Speedups based on optimization techniques

techniques ($ESS > TVS > ERT > BVR$) across a big range of performance in sequential CPU executions, while they reside within a small range of performance in parallel GPU executions. Due to this performance order in the CPU-time, BVR that is not optimized shows the best speedup practice compared to the other optimization techniques. The other three techniques shows a slight difference in the

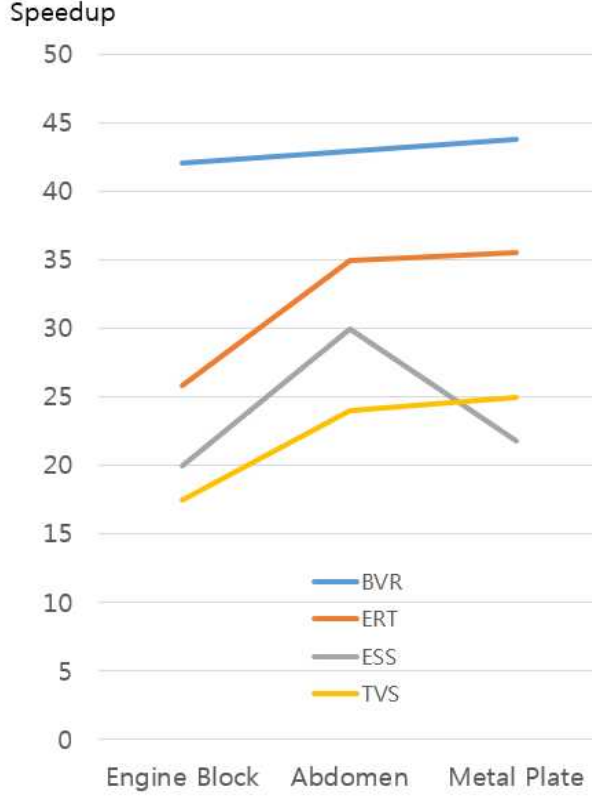


Figure 7. Speedups based on image complexity

order ($BVR > ERT > ESS > TVS$) in which ESS is not always the last one in that order, especially in case of more complex images. To explain this irregularity, we argue that TVS incurs additional branch divergence for rendering optimization and then cannot improve speedup especially for complex target images.

Regarding the viewpoints on each technique, we compare the speedup cases on the three images with different complexities. BVR

shows the lowest speedup for Engine Block that is the highest in image complexity and then becomes the basis of comparisons with the other optimization techniques.

Regarding the viewpoint of optimization techniques, as shown in Table 1 and represented in Figure 7, we can classify them again into two classes: one type of techniques that has no additional branch divergence, such as ERT and ESS, and the other type of techniques that has additional branch divergence, such as TVS. ERT and ESS show the medium-level speedup for Engine Block, because these optimization technique show both good sequential performance and no branch divergence. TVS, however, shows the lowest speedup for Engine Block, because the technique incurs a serious amount of branch divergence. To explain this irregularity, we argue that any optimization technique incurring additional branch divergence cannot improve speedup for complex target images.

Therefore, our results show that the TVS optimization technique cannot improve speedup for complex target images, because TVS incurs additional branch divergence for rendering optimization.

Chapter 4.

Reducing Branch Divergence

In this chapter reduces the branch divergence in TVS by factoring out structurally similar code from branch paths in the GPU programs. For this kind of reduction in general classes of GPU programs, Han and Abdelrahman (2011) present two novel software-based optimizations: *iteration delaying* and *branch distribution*.⁸⁾ Our reduction algorithm is established based on this branch distribution algorithm for volume rendering in GPU platforms.

4.1. *The Branch Divergence Reduction Algorithm*

Iteration delaying is a method that targets a divergent branch enclosed by a loop within a GPU kernel. It improves performance by executing loop iterations that take the same branch direction and delaying those that take the other direction until later iterations.

Two strategies are proposed to decide which direction to take. The first is *majority vote*. In each iteration, all threads in a warp communicate with each other to determine the number of threads that take each path, and then choose the direction that at least half of

threads take. For example, this half of threads in a warp of NVIDIA GPU is sixteen, because the size of the warp is fixed to thirty two. The rationale behind this strategy is to utilize at least half of the execution units. The second decision strategy is *round-robin*. This strategy works by changing the decision for each iteration. This means that branching decision for the i -th iteration is the opposite of that for the $(i-1)$ -th iteration.

For majority vote strategy, the main challenge of implementing iteration delaying is reaching a consensus among the warp threads on which path to take in each loop iteration. This implies that the threshold parameter that defines what “majority” is need to be defined first. And then, for each iteration, decision needs to be made based on this threshold condition. To do this, this strategy uses two vote functions for CUDA warps: `__ballot` and `__popc`. The `__ballot` instruction collects branch conditions for all 32 warp threads into a 32-bit integer and the `__popc` instruction counts the number of bit 1’s in a 32-bit integer.

The round-robin strategy is implemented by inverting the threshold condition periodically. This means that we declare two variables where each of them indicates the two different branches. And finally, the execution of all threads inside a warp is checked by using two CUDA warp vote functions: `__all` and `__any`. The `__all` instruction returns true if all threads conditions are true, and `__any` instruction returns true if at least one thread’s condition is true.

Although iteration delaying improves speedup by x1.12 for real-world applications, it is highly dependent on the functionality of the CUDA platform.²⁾ We take iteration delaying algorithm out of the equation, since our program is implemented using OpenCL that does not support any warp vote functions.⁹⁾

While the iteration delaying relies on a per-thread loop that surrounds the target branch, Han and Abdelrahman (2011) propose the *branch distribution* method. This method “factors out” code from the branch paths that are structurally the same, so that the total number of dynamic instructions are reduced. For example, consider the code fragment shown in Figure 8a. The structures of the two branches are almost identical, and we can produce less divergent code as shown in Figure 8b. Thus, this optimization “distributes” the branch condition evaluation over the two branch bodies, which results in one or smaller branch blocks interleaved with blocks of straight-line code, reducing the

<pre> if (c > 0) { x = x * a1 + b1; y = y * a1 + b1; } else{ x = x* a2 + b2; y= y* a2 + b2; } </pre>	<pre> if (c > 0) { a = a1; b = b1; } else{ a = a2; b = b2; } x = x * a + b; y = y * a + b; </pre>
(a) original code	(b) optimized code

Figure 8. Code example of branch distribution

impact of branch divergence.

We now apply this algorithm to volume rendering with TVS optimization. As mentioned in Section 2.1, the ray casting algorithm is composed of four stages including the sampling stage. The TVS decides in the sampling stage whether it skips a sample point or not based on its sample value. And if that sample point is not zero, it proceeds to the third stage. Otherwise, it does not proceed to the shading stage, but it simply allocates zero to its return value. In the last stage, called compositing stage, this return value is used iteratively to be multiplied to obtain the final sample point value. If the shading's return value is zero, the final colour value for that ray's pixel is zero, which implies an empty space.

The code fragment in Figure 9a is a simple pseudo code of TVS before applying branch distribution. We see that both of the branch paths are structurally similar with each other in two locations of the codes: assigning values to the shading weight, and compositing based on the shading weight. Thus, we can produce less divergent code by distributing the branch condition evaluation, which means factoring out the compositing stage, as shown in Figure 9b. We call this new algorithm as the RBDV algorithm, denoting the reduction of branch divergence in volume-rendering.

float shad_weight;	float shad_weight;
ray_casting();	ray_casting();
sampling();	sampling();
if (sample_value > 0) {	if (sample_value > 0) {
shad_weight = shading();	shad_weight = shading();
compositing(shad_weight);	} else {
} else {	shad_weight = 0;
shad_weight = 0;	}
compositing(shad_weight);	compositing(shad_weight);
}	
(a) original code	(b) optimized code

Figure 9. Applying branch distribution method for GPU-base volume rendering program.

4.2. *Experimentation on Branch Divergence*

We performed a set of experiments to prove that the RBDV algorithm improves the GPU-speedup of TVS for complex images. Although BVR does cause branch divergence, we only experimented with TVS to see the branch distribution's effect with maximum amount of branch divergence.

The environment for these experiments is identical with the experiments which are performed in Chapter 3. We performed three more additional experiments to this prior set of experiments, which apply the RBDV algorithm to see if the TVS algorithm with RBDV actually shows actual effects of reducing branch divergence for

rendering our three target images: Engine Block, Metal Plate, and Abdomen.

Table 2 shows the resulted data from these new experiments, which has three rows on target images described above and six columns on the times, in seconds, monitored in the new experiments. Among these six columns, the first three columns of them rearrange the last row of Table 1 that describes the measured results of TVS in three views: only CPU-time, the GPU-time without the RBDV algorithm, and its speedup. And, the next two columns show the new times and speedups that are measured from the new three experiments for the TVS with RBDV. The final column shows the speedups that are evaluated by dividing the column on the speedup with RBDV by another column on the speedup without RBDV.

After we apply RBDV to its GPU code. as shown in Table 2, we see significant changes of speedups between the GPU-time without and with RBDV. Although images with low complexities shows little

	CPU	GPU-time of TVS without RBDV	Speedup	GPU-time of TVS with RBDV	Speedup	Speedup Ratio
Engine Block	2.8	0.16	17.5	0.14	20.0	1.14
Metal Plate	3.0	0.12	25.0	0.12	25.0	0
Abdomen	1.2	0.05	24.0	0.05	24.0	0

Table 2. Speedup after applying branch distribution method.

improvement, the target image with the highest complexity shows 14% improvement, from x17.5 speedup to x20.0.

4.3. Analysis on Branch Divergence

This phenomena can be explained with the relationship between image complexity and the amount of branch divergence caused by the complexity. Branch divergence are proportional to the degree of image complexity, because frequent changes over various image regions means that the code requires execution instances of **if-then-else** statements as many as the changes.

Figure 10 shows a graphical representation of the GPU-speedups shown in Table 2 based on image complexity. As the complexity of image decreases, so does the difference between GPU speedups with or without the RBDV algorithm. This phenomena proves that the branch reduction algorithm in GPU-based volume rendering is more effective, if the input image is associated with a relatively higher value of complexity that is evaluated by the EVIC algorithm.

Therefore, branch reduction technique such as RBDV shows an significant improvement in case of complex images, because a reasonable amount of branch divergence can be reduced in those kinds of cases. On the other hand, since non-complex images have less frequently changes over image regions and requires the less number of

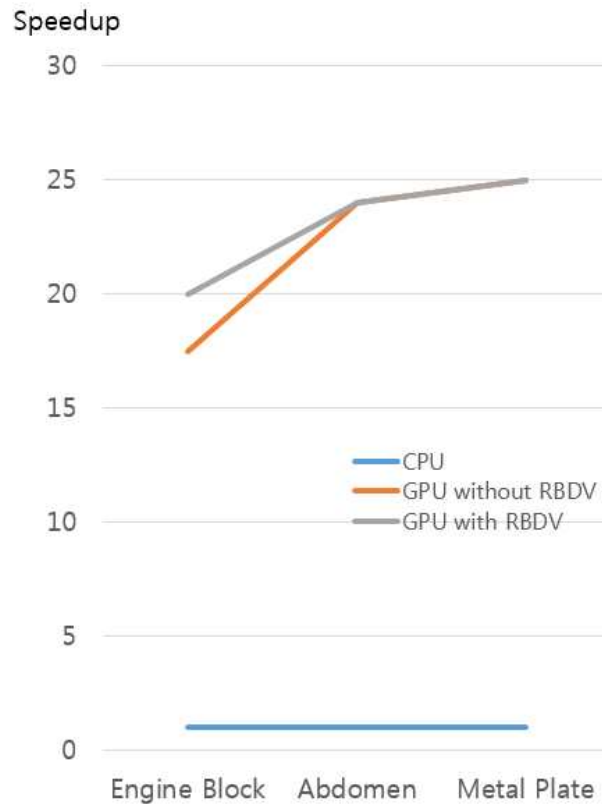


Figure 10. The TVS speedup based on image complexity

executions of conditional statements, such images show little improvement even though we applied the RBDV algorithm.

Chapter 5.

Conclusion and Future Work

This paper presents a speedup improvement method for optimized volume rendering in GPU platforms. First, from a set of experiments, we found that the speedup of volume rendering optimized with transparent voxel skipping *decreases* with dependency on the complexity of target images. In order to evaluate the complexity of volume images, we developed a new algorithm, called EVIC. Next, we present another new algorithm that reduces the branch divergence in transparent voxel skipping by factoring out structurally similar code from branch paths in GPU programs. This algorithm *increases* the GPU-speedup of transparent voxel skipping by 14%, improving it from x17.5 to x20.0 on average for complex target images.

There are several issues for future work. The first issue is to apply two iteration delaying methods discussed by Han and Abdelrahman to GPU-based volume rendering implemented with OpenCL. Although OpenCL does not support the warp vote functions which is provided in CUDA environment, this could achieve decent results. This is because iteration delaying method achieves higher speedups than branch distribution method.⁸⁾ The second issue is to extend our RBDV algorithm. Because the benefit of branch distribution is proportional

with the size of the code factored out,⁸⁾ we expect better speedup if more than two steps of the ray casting algorithm can be factored out.

Reference

- [1] C. Balhnisch, P. Stelldinger, and U. Kothe, "Fast and Accurate 3D Edge Detection for Surface Reconstruction," *31st Annual Symposium of the German Association for Pattern Recognition*, Springer-Verlag, September 2009.
- [2] M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," *International Conference on Compiler Construction (CC)*, pp. 244–263, Springer-Verlag, 2010.
- [3] S. Bruckner, *Efficient Volume Visualization of Large Medical Datasets*, Master's Thesis, Vienna University of Technology, 2004.
- [4] R. E. Bryant, and D. R. O'Hallaron, *Compter Systems – A Programmer's Persepective*, 2nd Edition, Prentice Hall, 2009.
- [5] G. Cox, et al, "Exploring Parallelism in Volume Ray Casting: Understanding the Programming Issues of Multithreaded Accelerators," *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, ACM, February 2012.
- [6] K. Engel, et al, *Real-Time Volume Graphics*, A. K. Peters Publishers, 2006.
- [7] K. Engel, M. Kraus and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading,"

International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH), ACM, 2001.

- [8] T. D. Han, and T. S. Abdelrahman, "Reducing Branch Divergence in GPU Programs," *Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 2011.
- [9] L. Howes, and A. Munshi, *The OpenCL Specification Version 2.0*, Khronos OpenCL Working Group, 2014.
- [10] Joe Kniss, et al., "Interactive Texture-Based Volume Rendering for Large Data Sets," *Computer Graphics and Applications*, Vol. 21, No. 4, IEEE, 2001.
- [11] J. Kruger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," *Visualization (VIS)*, IEEE, 2003.
- [12] B. Lee, J. Yun, J. Seo, B. Shim, Y. Shin, and B. Kim, "Fast High-Quality Volume Ray Casting with Virtual Samplings," *Transactions on Visualization and Computer Graphics*, Vol. 16, No. 6, pp. 1525-1532, IEEE, November 2010.
- [13] L. Marsalek, et al, "High-Speed Volume Ray Casting with CUDA", *Symposium on Interactive Ray Tracing*, IEEE, 2008.
- [14] M. Meißner, H. Pfister, R. Westermann, and C. M. Wittenbrink, "Volume Visualization and Volume Rendering Techniques," *European Association for Computer Graphics (Eurographics)*, Tutorial, 2000.
- [15] H. Song, J. Yun, B. Kim, and J. Seo, "GazeVis: Interactive 3D Gaze Visualization for Contiguous Cross-Sectional Medical Images," *Transactions on Visualization and Computer Graphics*, Vol. 20, No. 5, pp. 726-739, IEEE, May 2014.

- [16] D. Weiskopf, *GPU-Based Interactive Visualization Techniques* (Mathematics and Visualization), Springer-Verlag, 2006
- [17] H. Yu, et al, "I/O Strategies for Parallel Rendering of Large Time-Varying Volume Data," *Eurographics Symposium on Parallel Graphics and Visualization* (EGPGV), pages 31-40, Eurographics/ACM-SIGGRAPH, 2004.

국문 초록

최적화된 Volume Rendering의 GPU-Speedup 개선 기법

서울대학교 전기컴퓨터공학부

전 상 수

이 논문은 volume rendering을 GPU 기반으로 병렬처리 했을 때의 speedup 개선 기법을 소개한다. 첫 번째로, 우리는 transparent voxel skipping을 이용하여 최적화된 volume rendering의 speedup은 영상 복잡도가 높을수록 감소한다는 것을 실험을 통해서 발견하였다. 이러한 볼륨 영상의 복잡도를 계산하기 위해서 EVIC 이라는 새로운 알고리즘을 개발하였다. 그리고 GPU 프로그램에서 구조적으로 비슷한 코드들을 분기제어 경로에서 제외시킴으로서 transparent voxel skipping을 구현한 프로그램의 branch divergence를 감소시키는 새로운 RBDV 알고리즘을 제시한다. 복잡도가 높은 영상에서, 이 알고리즘은 transparent voxel skipping의 GPU-speedup을 평균 17.5배에서 20배까지 14% 이상으로 증가시킬 수 있다는 것을 확인하였다.

주요어. 영상 복잡도, volume rendering, transparent voxel skipping, speedup, graphics processing unit (GPU)