Computation-to-Core Mapping Strategies for Iso-Surface Volume Rendering on GPUs

Junpeng Wang* Virginia Tech
Fei Yang† NVIDIA
Yong Cao‡ Virginia Tech

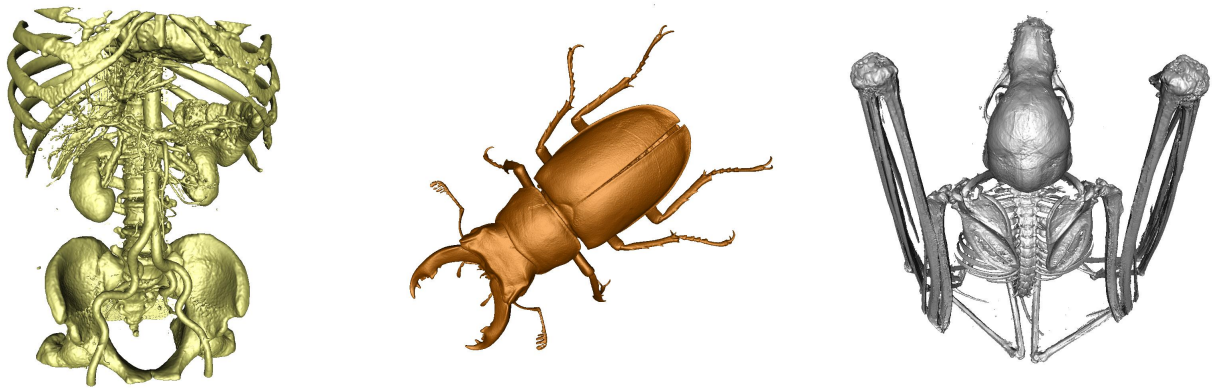

Figure 1: Data sets used for performance tests (from left to right): the Abdominal (340 MB, 16-bit per sample), the Stag Beetle (652 MB, 16-bit per sample) and the Bat data set (1.2 GB, 8-bit per sample). They are rendered with the Warp Marching algorithm.

Abstract

Ray casting algorithm is a major component of the direct volume rendering, which exhibits inherent parallelism, making it suitable for graphics processing units (GPUs). However, blindly mapping the ray casting algorithm on a GPU's complex parallel architecture can result in a magnitude of performance loss. In this paper, a novel computation-to-core mapping strategy, called *Warp Marching*, for the texture-based iso-surface volume rendering is introduced. We evaluate and compare this new strategy with the most commonly used existing mapping strategy. Texture cache performance and load balancing are the two major evaluation factors since they have significant consequences on the overall rendering performance. Through a series of real-life data experiments, we conclude that the texture cache performances of these two computation-to-core mapping strategies are significantly affected by the viewing direction; and the Warp Marching performs better in balancing workloads among threads and concurrent hardware components of a GPU.

1 Introduction

Ray casting based direct volume rendering is one of the most important volumetric data visualization approaches. Its flexibility and embarrassingly parallel nature, in recent years, have made it a dominant over other approaches such as volume segmentation and explicit surface extraction [5]. Ray casting is inherently a parallel algorithm, in which all rays can be calculated concurrently without computational dependency. The algorithm, therefore, is well adopted in implementations targeted at GPUs, where all the rays can be mapped to their massive-thread architectures.

However, careless design and implementation of the ray casting algorithm on GPUs can cause performance issues that result in significant loss in efficiency. Similar to most computing hardware architectures, the GPU's architecture has a series of hardware optimizations to enhance the efficiency of parallel execution. Among them, the two most important and commonly analyzed features/issues are: *texture cache performance* and *load balancing*.

- *Texture Cache Performance:* Since the GPU's 3D texture memory provides hardware-accelerated tri-linear interpolation, it becomes the obvious choice for data storage in GPU-based volume rendering. However, caching strategy for the texture memory is optimized in a 2D fashion to improve the spatial locality. As a result, some parallel access patterns of the 3D texture lead to poor texture cache performance.

- *Load Balancing:* Load balancing is a traditional problem for parallel algorithm design. The early ray termination can cause imbalanced workloads among parallel threads. In this case, some GPU threads and their assigned GPU processors will be idle and used ineffectively. Load imbalance also happens among the concurrent hardware components of a GPU, in which case, some computational resources will be idle.

This paper analyzes these two performance issues of two different computation-to-core mapping strategies for the ray casting algorithm. The first mapping strategy is a traditional one, in which every GPU thread is mapped to one ray. The second mapping strategy, called Warp Marching, is derived from the work of Wang et al. [14]. In the latter, a number (warp) of GPU threads are mapped to one ray in the ray casting algorithm. Cache performances of these two strategies show high dependency on the viewing direction. The Warp Marching increases the granularity of parallelization and shows better performance in balancing workloads among concurrent GPU hardware components.

*e-mail: junpeng@vt.edu
†e-mail: feiy@nvidia.com
‡e-mail: yongcao@vt.edu

2 Background and Related Work

The GPU hardware is built with one or multiple stream multiprocessors (SMs). Each SM consists of numerous computational units, i.e. GPU processors/cores. When working with GPUs, tens of thousands of threads are created and organized as grids of blocks. They are scheduled to SMs group by group. In CUDA, 32 threads form such an atomic group, called warp. Memory stride, inside such a warp of GPU threads, has a significant effect on texture cache performance [8].

Most of the GPU-based volume rendering methods [2, 3, 10, 15, 16] map the volume to the 3D texture memory of the GPU, to take advantage of the hardware-accelerated tri-linear interpolation. The 3D texture is organized with many 2D slices [17], and Morton code [6] is used to optimize the spatial locality inside these slices. However, the locality does not exist between slices. As a result, the viewing direction has an effect on the texture cache performance, as well as the rendering performance [3, 14, 15]. For instance, in Figure 2, when the viewing direction is perpendicular to the 2D slices of the 3D texture (facing XY), samples are close to each other in the memory space. Thus, high texture cache hit rates are expected. However, when the viewing direction is parallel to these 2D slices (facing ZY), large memory gaps between samples result in a poor texture cache performance.

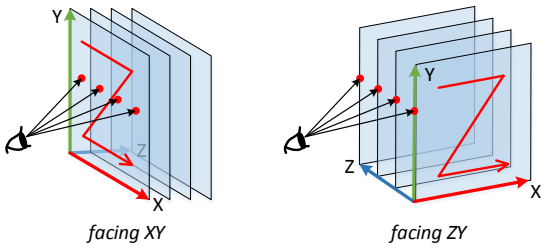

Figure 2: Effects of different viewing directions.

Over the last decade, much research has been done to improve the texture cache performance, such as [4, 11, 15]. Most recently, Wang et al. [14] introduced a novel computation-to-core mapping strategy, called "Warp Marching", for the texture-based volume rendering. However, their approach relies on the associative property of color blending, thus it only works for the semi-transparent volume rendering. In this paper, we first extend their Warp Marching to the iso-surface volume rendering, and then analyze this new computation-to-core mapping strategy by comparing it with the traditional one.

The load balancing problem of GPUs is also an active area of research. Aila and Laine [1] efficiently solved this problem on GPUs with their "persistent threads" mechanism. Tzeng et al. [12] balanced irregular workloads with a dynamic task management strategy. Wald [13] compacted active threads to effectively utilize the GPU hardware. This paper will demonstrate that the Warp Marching performs better than the traditional computation-to-core mapping strategy in balancing workloads among different GPU threads, as well as different GPU SMs.

3 Parameters of Interest

We are interested in two performance factors: texture cache performance and load balancing. In detail, our discussion will focus on the performance of two different mapping strategies in the following four parameters:

Viewing Direction: Similar to [14], the traditional computation-to-core mapping strategy is also called the Standard strategy/method. In this mapping, one ray is mapped to one GPU thread. A warp of threads takes samples on a plane that is always perpendicular to the viewing direction. When the plane is aligned with the 2D slices of texture data (the "facing XY" case in Figure 2), the parallel access of the samples on the plane is efficient and the cache hit rate is high. On the contrary, when the plane is perpendicular to the 2D slices of volumes (the "facing ZY" case in Figure 2), the texture cache hit rate is low and the rendering performance is poor. Thus, the viewing direction has a big impact on the rendering performance.

Projection Method: The perspective projection is more commonly used in 3D volume rendering than the orthographic projection. However, the distance between samples keeps increasing as the rays march further (Figure 3, left) in this projection method. On the contrary, the sampling distance is constant in the orthographic projection. We will demonstrate how seriously these two different projection methods affect the texture cache performance.

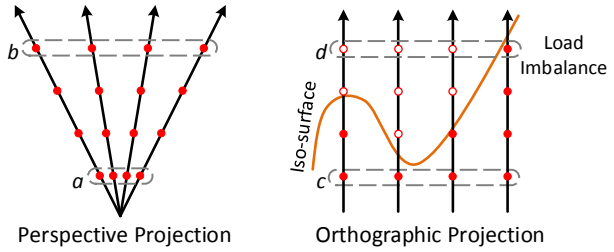

Figure 3: The distance between four samples in c and d is the same; while that between a and b is different.

Load Imbalance Among Threads: In the ray casting algorithm, all rays march along the viewing direction and sample the volume data in parallel. However, the marching process of some rays can be terminated earlier than others, as illustrated in the "Orthographic Projection" in Figure 3. In the iso-surface rendering, this occurs when the rays have already reached the iso-surface. The threads associated with the terminated rays are idling while the other threads are still performing, which will result in imbalanced workloads among threads of the same warp (assume the warp size is 4 in Figure 3).

Load Imbalance Among SMs: Other than the thread level load imbalance, we are also interested in the load imbalance among SMs. Since a thread warp (32 threads in CUDA) is the atomic scheduling unit of GPUs, it can only be mapped to one SM. For the Standard mapping, a thread warp covers 32 rays. The warp will occupy one SM for as long as even a single ray is still marching. The varying life time of warps leads to imbalanced workloads among SMs. During execution, a lengthy warp can only be scheduled to one SM, even though more SMs are available.

4 Warp Marching

This section shows how the Warp Marching algorithm [14] can be adapted to the iso-surface volume rendering. Instead of mapping one GPU thread to one ray, the Warp Marching maps a warp of threads to one ray (Figure 4 *b*, assume the warp size is 4). At any given time, the thread warp samples multiple points along the ray in parallel. Then, it marches along the ray direction in this manner from front to back until it finds the iso-surface.

This approach achieves good cache hit rates when facing the ZY plane of a volume. Since the distance between samples along rays is usually small (to satisfy the Nyquist frequency), it could also achieve good texture cache performance when facing the XY plane. Moreover, the imbalanced load issue is mitigated since one warp only covers one ray/pixel. Load imbalance happens, at most, once for each warp. The Warp Marching creates more warps of threads, but the workload for each warp is reduced. This way, finer work

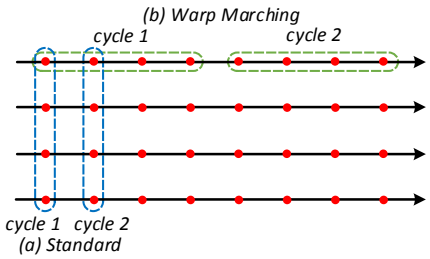

Figure 4: Different computation-to-core mapping strategies.

granularity is successfully achieved and scheduling warps to SMs can be more flexible.

In implementation, each thread of a warp is given a binary bit. If the corresponding sample's density is less than the iso-value, the bit is set to 0, otherwise it is set to 1. Then the warp voting function, i.e. `__ballot()`, can be used to quickly get the state of the whole warp. There are three possible states:

1. All threads of the warp take samples with densities less than the iso-value (Figure 5: A (1), `__ballot()` returns 0x00).
2. All threads of the warp take samples with densities greater than or equal to the iso-value (Figure 5: A (2), `__ballot()` returns 0xff).
3. Some threads of the warp take samples with densities less than the iso-value, while others take samples with densities greater than the iso-value (Figure 5: B, `__ballot()` returns a value between 0x00 and 0xff).

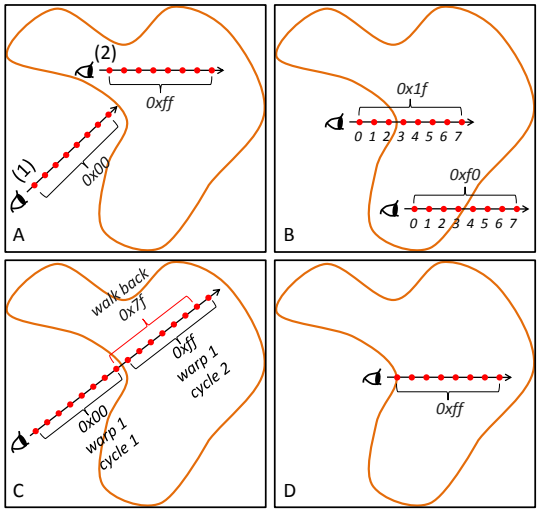

Figure 5: Three general cases (A, B) and two special cases (C, D) of the iso-surface Warp Marching (the warp size is 8 in this figure).

Rays can march through the spaces that do not contain the iso-value by simply skipping the first two cases (Figure 5, A). The third case indicates the iso-surface is detected and it is the only case where load imbalance among threads of a warp may happen. In this case, the warp will identify the thread that takes the sample around the iso-surface (apply the intrinsic function `__clz()` to the integer value returned from `__ballot()`) and have it do a linear interpolation to find the estimated iso-surface position.

The iso-surface may be located inside the gap between two warp cycles (Figure 5, C), in which case the iso-value cannot be detected. Such a condition can be managed by having the warp walk back one sample in each cycle. In addition, if the iso-value is taken by the first thread of the warp (Figure 5, D), Warp Marching cannot detect it either. To address this, Warp Marching tests whether the first sample's density is equal to the iso-value at the first cycle. This problem will not happen at the second cycle and cycles after.

5 Results and Analysis

Results in this section are collected on an NVIDIA GeForce GTX TITAN GPU with 14 SMs and 6 GB device memory. The results of the texture cache hit rate and the load balancing indicator are collected by CUDA profiling tools interface (CUPTI) [9]. Data sets used in our experiments are shown in Figure 1. They are rendered with a resolution of 1024×1024. We also show the actual projected image resolutions of these data sets when facing different viewing directions in Table 1. These resolutions and the volume dimensions together provide indications of the distance between rays. The distance between samples along rays (z direction) is fixed at 0.5 voxel length. Both the Standard and the Warp Marching algorithms use 256 threads per block.

5.1 Texture Cache Performance

Viewing Direction: As shown in Table 1, when the viewing direction is facing the *XY* plane, the Standard method is always better than the Warp Marching method in terms of the texture cache hit rate and the rendering performance. Threads of the same warp in this computation-to-core mapping strategy take samples on the same 2D slice and these samples are close to each other in memory space. The Warp Marching takes samples across slices. However, since the distance along rays is fixed at a small constant value (0.5 voxel length), this method still achieves real-time rendering performance. On the contrary, when facing the *ZY* plane, the Warp Marching is significantly faster than the Standard mapping. In this direction, the texture cache hit rate of the Standard method is very low, consequently resulting in a poor rendering performance, especially when the volume size is large. For the Standard mapping, the memory stride inside a warp is decided by the ratio of the volume dimension and the projected image resolution. In Table 1, the larger data sets have larger ratios in our experiment settings, so texture cache performs worse when testing on them.

Projection Method: In the Standard mapping strategy, when the rays march further, the distance between samples gets larger due to the fencing effect of the perspective projection. The texture cache hit rate drops as the sampling distance increases. Figure 6 *A* shows this trend from orthographic projection to perspective projection with increasing field of view angles. On the contrary, in Warp Marching, as the field of view angle increases, the texture cache results are marginally better (Figure 6 *B*). We believe this is because, the rays cover a larger view range but go across a lesser number of slices in this viewing direction. When facing the *ZY* plane, the texture cache hit rates of both mapping strategies drop, because both of them cover more numbers of slices of the volume.

5.2 Load Balancing

Our discussion of the load balancing is focused on two metrics: *Control Flow Efficiency* and *SM Activity*, which are defined as follows [7, 9]:

$$\textit{Control Flow Efficiency} = \frac{\textit{Thread Instructions Executed}}{\textit{Instructions Executed} \times \textit{Warps Size}}$$

$$\textit{SM Activity} = \frac{\textit{Active Cycles}}{\textit{Elapsed Clocks}}$$

Load Balancing Among Threads (*Control Flow Efficiency*): In section 3, we showed that a thread warp in the Standard mapping

		Facing the *XY* Plane			Facing the *ZY* Plane		
	Data Set	Abdominal	Stag Beetle	Bat	Abdominal	Stag Beetle	Bat
	Dimension	512×512	832×832	906×911	681×512	494×832	1466×911
	Projected Resolution	780×780	871×871	772×776	872×738	603×1014	1023×652
	Ratio	0.66×0.66	0.96×0.96	1.17×1.17	0.78×0.69	0.82×0.82	1.43×1.40
Texture Cache Hit Rate	*Standard*	91.01%	86.64%	87.98%	56.25%	51.62%	29.12%
	Warp Marching	55.44%	66.67%	63.59%	82.87%	88.38%	87.80%
	Speedup	0.61	0.77	0.72	1.47	1.71	3.02
Frame Per Second	*Standard*	135.52	71.71	48.28	49.81	15.24	3.50
	Warp Marching	68.63	47.67	34.44	106.28	62.24	46.01
	Speedup	0.51	0.66	0.71	2.13	4.08	13.15
Control Flow Efficiency	*Standard*	90.09%	94.62%	84.88%	90.42%	91.62%	90.89%
	Warp Marching	92.46%	95.50%	97.09%	92.01%	96.36%	96.75%
	Speedup	1.03	1.01	1.14	1.02	1.05	1.06
SM Activity	*Standard*	99.84%	99.83%	99.90%	99.93%	99.98%	99.98%
	Warp Marching	99.98%	99.99%	99.99%	99.97%	99.99%	99.99%
Standard Deviation of Active Cycles	*Standard*	5352.90	11024.53	12470.70	5931.84	7965.91	25816.52
	Warp Marching	150.30	103.26	157.53	137.19	143.15	177.05
	Speedup	35.61	106.76	79.16	39.30	55.65	145.81

Table 1: Comparison of the Warp Marching and the Standard algorithm. Results are collected in perspective projection with a 45° field of view.

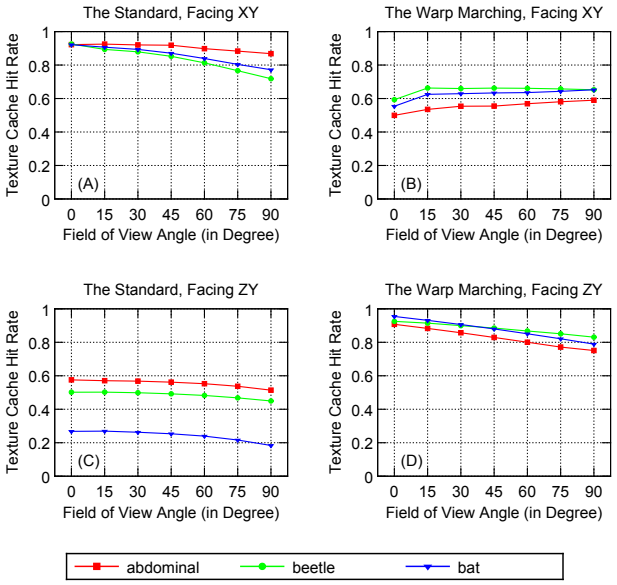

Figure 6: The effect of projection method to the texture cache hit rate. The horizontal axis shows the field of view angles, and 0° indicates an orthographic projection.

covers multiple rays. The early ray termination mechanism causes imbalanced workloads among threads in this mapping strategy. The Warp Marching mitigates this problem since a warp only works on one ray and the parallelism is always along the ray. *Control Flow Efficiency* (also known as CUPTI metric *Warp Execution Efficiency*) reflects the percentage of active threads inside a warp. A lower *Control Flow Efficiency* value means more computational resources are idling. From results shown in Table 1, the Warp Marching always achieves better *Control Flow Efficiencies* than the Standard mapping strategy regardless of data set or viewing direction.

Load Balancing Among SMs (*SM Activity*): The experiment GPU has 14 SMs. We study how different computation-to-core mapping strategies affect the workload distribution among these SMs. Compared to the Standard algorithm, the Warp Marching generates more threads to process the same amount of work. It alleviates the heavy work of a thread by dispersing the work to a warp of threads. The workload variance among warps is also reduced. So the Warp Marching has a finer work granularity, which should decrease the workload variance among different SMs of a GPU.

SM Activity reflects the overall status of all SMs of a GPU. Results of this indicator for the Standard and the Warp Marching methods are shown in Table 1. Both methods have very high *SM Activities*. However, this metric only shows the overview of hardware usage and does not reflect the workload differences among 14 SMs. Since the overall *SM Activity* is derived from 14 instances, it will be beneficial to reveal the differences among them. For each kernel execution, the *Elapsed Clocks* for the 14 SMs are the same, but the *Active Cycles* are different. The standard deviation of the *Active Cycles* among 14 SMs has been shown in Table 1. This metric clearly illustrates that the Warp Marching method is better than the Standard method in balancing workloads among SMs.

6 Conclusion

We analyze two different computation-to-core mapping strategies for the iso-surface volume rendering in two hardware-oriented factors. The texture cache performance of these two methods shows a complementary pattern on different viewing directions. When the viewing direction is perpendicular to the 2D slices of a 3D texture, the Standard mapping strategy achieves higher texture cache hit rates than the Warp Marching. However, when the viewing direction is parallel to the 2D slices, the Warp Marching performs better. Field of view angles of the projection method have effects on both strategies. The performance difference between these two algorithms is more obvious when testing with larger data sets.

Compared to the Standard mapping strategy, the Warp Marching has smaller workload variances among threads as well as warps. It balances workload more evenly than the Standard computation-to-core mapping, regardless of the viewing direction and the volume size. Consequently, the Warp marching could utilize the GPU hardware more effectively.

REFERENCES

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[2] 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.

[3] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[4] D. Jönsson, P. Ganestam, M. Doggett, A. Ynnerman, and T. Ropinski. Explicit cache management for volume ray-casting on parallel architectures. In *Symposium on Parallel Graphics and Visualization*. Eurographics, 2012.

[5] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM, 1987.

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

[7] NVIDIA. Nvidia nsight visual studio edition user guide.

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

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

[10] 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.

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

[12] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[13] I. Wald. Active thread compaction for gpu path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 51–58. ACM, 2011.

[14] J. Wang, F. Yang, and Y. Cao. Cache-aware sampling strategies for texture-based ray casting on gpu. In *Proceedings of the 4th IEEE symposium on Large Data Analysis and Visualization*, pages 19–26, 2014.

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

[16] 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.

[17] N. Wilt. The cuda handbook: A comprehensive guide to gpu programming. 2013.