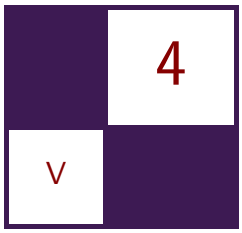


Semi-static Load Balancing for Low Latency Ray Tracing on Heterogeneous Multiple GPUs

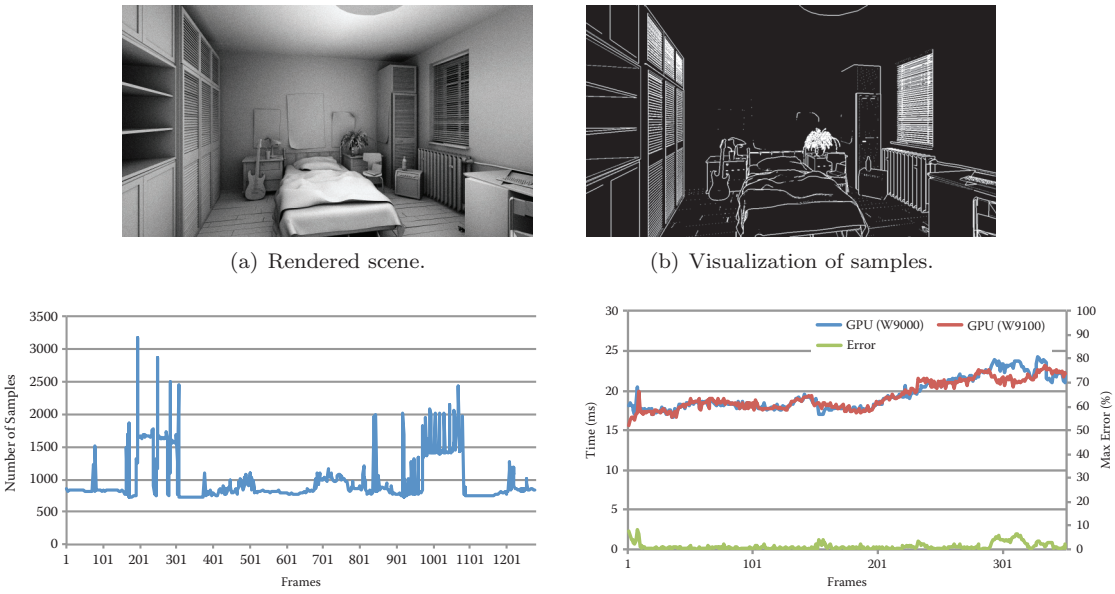
Takahiro Harada

4.1 Introduction

Ray tracing is used to render a realistic image but the drawback is its high computational cost. Although there are studies accelerating ray tracing using the GPU, even with the latest GPU, we cannot get a satisfactory rendering speed. An obvious way to accelerate it further is to use more than one GPU. To exploit the computational power of multiple GPUs, the work has to be distributed in a way so that it minimizes the idle time of GPUs. There are studies on load balancing CPUs, but they are not directly applicable to multiple GPUs because of the difference of the architectures, as discussed in Section 4.2.

If we could restrict the target platform as GPUs with the same compute capability, the problem is simpler. However, there are more and more PCs with multiple GPUs with different compute capabilities (e.g., a PC with an integrated GPU on a CPU and a discrete GPU). Also, when we build a PC with multiple discrete GPUs, it is easier to get different-generation GPUs than GPUs with the same specification, or the same compute capability. Therefore, if we develop a ray tracing system that works well on multiple GPUs with nonuniform compute capabilities, there are more PCs that benefit from the method comparing to a ray tracing system developed only for GPUs with a uniform compute capability.

If we restrict ourselves to a system with multiple GPUs of the same specification, we could use alternate frame rendering [Advanced Micro Devices, Inc. 16]. However, an issue of the method is latency; it does not improve the latency to render a single frame. There are many applications that prefer a low-latency rendering. They include games and other interactive applications. Also, the rise of the head-mounted display is another strong push of a low-latency rendering.



(a) Rendered scene. (b) Visualization of samples.

(c) Sample distribution. (d) Computational time and error.

Figure 4.1. (a) Ray traced scene on AMD FirePro W9000 and W9100 GPUs. (b) Visualization of the number of samples per pixel (black = 1, white = 5). The depth buffer of the scene is first rendered using OpenGL. Then, an edge detection algorithm computes this image, which is an input for a primary ray generation kernel generating more samples at pixels containing geometry edges. (c) Histogram of the number of samples of (b) for each vertical scanline. (d) Computational time on two GPUs and maximum deviation of computational time under a camera motion. Average error is 1.2%.

The goal of this chapter is to develop a low-latency ray tracing system for multiple GPUs with nonuniform compute powers. To realize this goal, we propose a semi-static load balancing method that uses rendering statistics of the previous frame to compute work distribution for the next frame. The proposed method does not assume uniform sampling density on the framebuffer, thus it is applicable for a problem with an irregular sampling pattern as shown in Figure 4.1. The method is not only applicable for the multi-GPU environment, but it can be used to distribute compute work load on GPUs and a CPU as we show in Section 4.4.

4.2 Load Balancing Methods

4.2.1 Frame Distribution

Frame distribution, also known as alternate frame rendering, is often used to utilize multiple GPUs for a raster graphics for interactive application [Advanced

Micro Devices, Inc. 16]. Although it performs well when all the GPUs in a system have the same compute capability, it results in underutilization of GPUs unless we use the same GPUs. When n GPUs are used, a GPU should spend $n \times t$ for computation of a single frame to have zero idle time where t is the time to display a single frame. Therefore, the latency of interaction is high; it takes time to propagate a user input to all the GPUs. Thus, alternate frame rendering is not suited for many GPUs with different compute capabilities.

4.2.2 Work Distribution

Data distribution, also known as sort last rendering, splits input geometry into small chunks each of which is processed on a node (when GPUs are used, a node is a GPU). Although it reduces the rendering time for each GPU, it is not straightforward to use for global illumination in which rays bounce. Moreover, the computation time is view dependent, thus it is difficult to get a uniform computation time for all the nodes. It also requires transferring screen-sized images with depth, which results in large network traffic. Therefore, it is not suited for rendering running at an interactive speed.

Pixel distribution, also known as sort first rendering, splits the screen into cells, and rendering a cell is distributed on nodes as work. If the works are distributed proportional to the compute capability of the nodes, all the nodes remain active and therefore we maximize the computation power of all nodes. This is often the choice to distribute work on multiple CPUs [Heirich and Arvo 98]. We also employ pixel distribution for work distribution, although the preferable work size is different for GPUs than for CPUs.

4.2.3 Work Size

CPUs prefer small work size for pixel distribution because it allows the system to adjust the workload on each node, which results in a uniform computation time on all nodes. However, when GPUs are used for computation, we also need to take the architectural difference into consideration. A GPU prefers a large or wide computation because of its architecture optimized for very wide computation. If a work size is small, it cannot fill the entire GPU, which results in underutilization of the GPU. Thus, we want to make the work as large as possible when GPUs are used as compute nodes. However, load balancing becomes more difficult if we make the work size larger and the number of works smaller, as it easily causes starvation of a GPU. The optimal strategy for our case is to generate m works for m GPUs and to adjust the work size so that computation times on GPUs are exactly the same. This is challenging for ray tracing in which the computation time for a pixel is not uniform. We realize this by collecting GPU performance statistics and adjust the work size for each GPU over the frames.

Cosenza et al. studied a load balancing method utilizing frame coherency, but they assume the same compute capability for processors [Cosenza et al. 08].

The method only splits or merges a work, thus it cannot perform precise load balancing unless using small leaves. Therefore, it is not well suited as a load balancing strategy for multiple compute devices. Another similar work to ours is work by Moloney et al., who studied load balancing on multiple GPUs for volume rendering [Moloney et al. 07]. However, they assume uniform compute capabilities and uniform distribution of samples. They also assume that the computational cost for each ray can be estimated. As none of those applies to ray tracing, their method cannot be used for our purpose.

4.3 Semi-static Load Balancing

A frame rendering starts with a master thread splitting the framebuffer into m areas using the algorithm described below, where m is the number of GPUs. Once the framebuffer assignment is sent to slaves, parallel rendering starts. Each GPU executes the following steps:

1. Generate samples (primary rays) for the assigned area.
2. Ray trace at sample location to compute radiance.
3. Send the framebuffer and work statistics to the master.

Note that master-slave communication is done only twice (send jobs, receive results) in a frame computation.

At the first frame, we do not have any knowledge about workload nor compute capabilities of the GPUs. Thus, an even split is used for the frame. After rendering frame t, compute device i reports the area of processed framebuffer s_i^t, the number of samples processed n_i^t, and the computation time for the work t_i^t. That information is used to compute the optimal framebuffer split for frame $t+1$.

The algorithm first estimates processing speed $p_i^t = n_i^t/t_i^t$ (number of processed samples per second) for each compute device. Then, it computes the ideal time $T = N^t/\sum p_i^t$ to finish the work with the perfect load balancing, where $N^t = \sum n_i^t$ is the total number of samples processed at t. With these values, we can estimate the number of samples we need to assign for compute device i at frame $t+1$ as $n_i'^{t+1} = Tp_i^t$.

If the sample distribution is uniform on the screen, we could assign area $s_i'^{t+1} = Sn_i'^{t+1}/N$ for compute device i, where $S = \sum s_i^t$. However, as we do not assume the uniform distribution over the frame, we need to compute the area of the framebuffer that contains $n_i'^{t+1}$ samples for compute device i. The procedure to compute area s_i^{t+1} is illustrated in Figure 4.2 in which we assume that there are four GPUs. GPU i processed the assigned area s_i^t at frame t and reported that there are n_i^t samples in the area (Figure 4.2(a)). A histogram of sample distribution at frame t is built from these values (Figure 4.2(b)). Samples n_i^t are stacked up, as shown in Figure 4.2(c), to draw lines as shown in Figure 4.2(d).

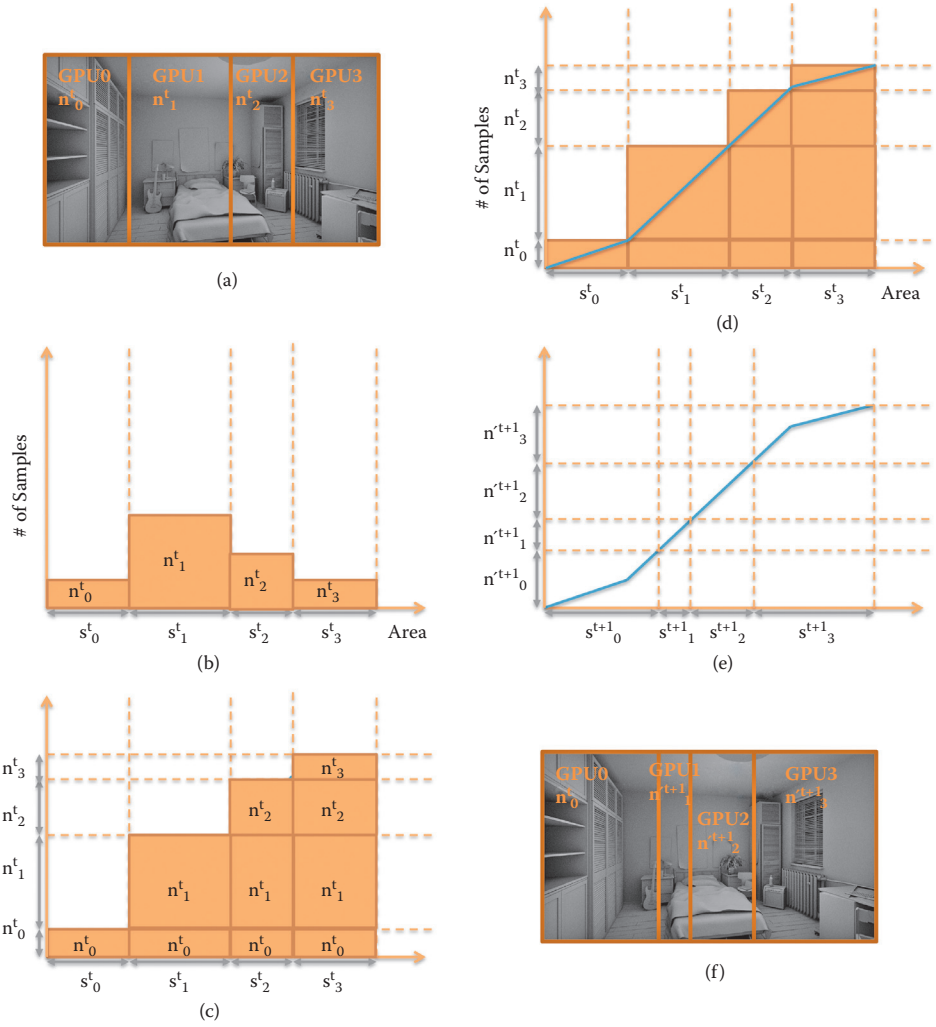


Figure 4.2. Illustration of computation steps for sample distribution at frame $t+1$ (f) using the information reported at frame t (a).

These lines are built to look up the number of samples at a given area. For example, we can find that there are n_0^t samples at s_0^t, and $n_0^t + n_1^t$ samples at $s_0^t + s_1^t$. When building the lines, we ignored the distribution of samples in s_i^t and assumed the uniform distribution. After building them, we search for s_i^{t+1} corresponding to $n_i^{\prime t+1}$ by the binary search.

Since we linearize the sample distribution at area processed at each GPU, there is no guarantee that the computed work distribution is perfect. Therefore,

we gradually move the distribution to the computed distribution by interpolating the split of t and $t+1$ as $n_i''^{t+1} = (1-\alpha)n_i^t + \alpha n_i'^{t+1}$, where α is the only parameter for the proposed method. We set $\alpha = 0.4$ for computation of Figures 4.1 and 4.3 and $\alpha = 0.2$ for Figure 4.4, which has a higher variation in the sample density.

4.4 Results and Discussion

The proposed method is implemented in a OpenCL ray tracer. Experiments are performed using three combinations of compute devices: AMD FirePro W9000 GPU + AMD FirePro W9100 GPU, Intel Core i7-2760QM CPU + AMD Radeon HD 6760m GPU, and four AMD FirePro W9000 GPUs. The framebuffer is split vertically for all the test cases. The method can be used with rendering pipelines with any sampling strategies, but here we show example usages of it with two rendering pipelines.

The first test rendering pipeline is similar to [Mitchell 87] but implemented as a hybrid of rasterization and ray tracing. It first fills the depth buffer using OpenGL, and it is used to compute a sample density map, as shown in Figure 4.1(b). The primary ray generation kernel for ray tracing reads the map and decides the number of samples per pixel. In our test case, we generate five samples for a pixel containing edges of geometry to reduce geometric aliasing, and one for the other pixels. Ambient occlusion is progressively calculated at 1280×720 resolution with two shadow rays per sample per frame. This is a challenging case for the proposed method because it has high variation in the number of samples in the direction of the split axis, as shown in Figure 4.1(b). We interactively control the camera for all the test cases to evaluate the robustness of the method for a dynamic environment. Sample distribution changes as the camera moves. This is the reason why the computational times and work distribution reported in Figures 4.1 and 4.3 have ups and downs. We can see that the method successfully keeps the computational time on different compute devices almost the same. Figures 4.3(d) and (e) show that the analysis of the work load distribution on the framebuffer is good. The same number of pixels would have been assigned for GPUs if we ignored the sample distribution. It however splits the framebuffer into works with different framebuffer area to achieve load balancing. The averages of the maximum deviations of computational time are 1.4, 0.9, 1.8, 2.9, and 2.1% for Figures 4.3(a), (b), (c), (d), and (e), respectively.

The other test rendering pipeline uses a foveated sampling pattern [Guenter et al. 12]. The sampling pattern we prepared in advance has higher sampling density at the center of the screen, and density decreases as the distance of the pixel from the center increases (Figure 4.4(a)). Sample density is less than one per pixel for sparse area. Primary rays are generated according to the pattern, and direct illumination is computed. We can see that the method keeps the computation time on four GPUs almost the same (Figure 4.4).

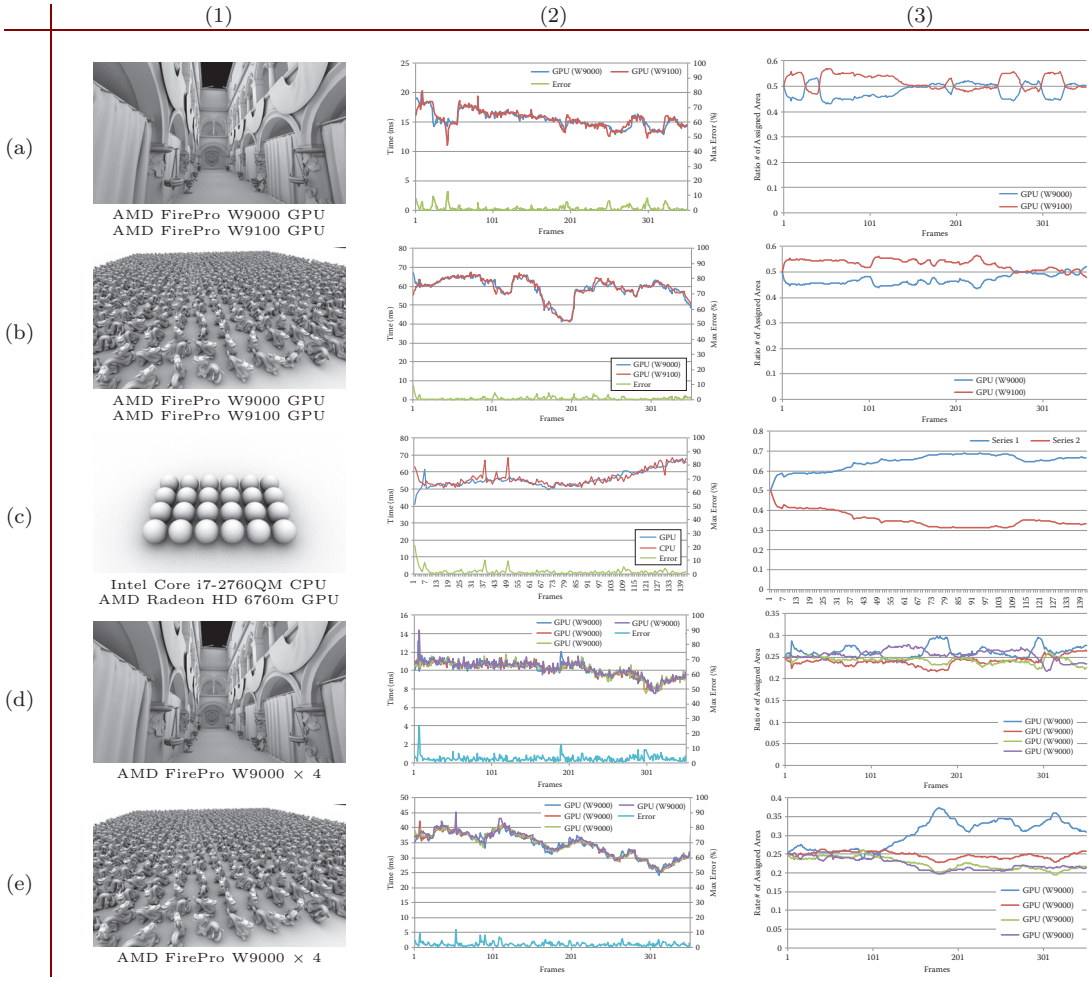
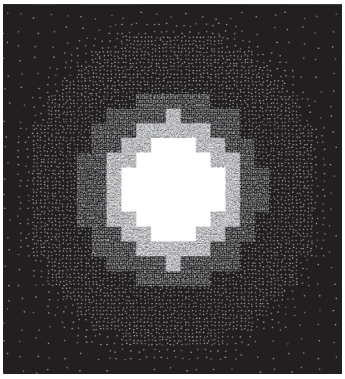


Figure 4.3. (1) Test scene and compute devices used for testing. (2) Computation time over frames. (3) Ratio of the number of processed pixels.

The method is also applicable to load balancing on multiple machines. In the example shown in Figure 4.5, the framebuffer is split into three areas each of which are processed by each machine, and each machine split the area further to distribute the computation on installed GPUs.

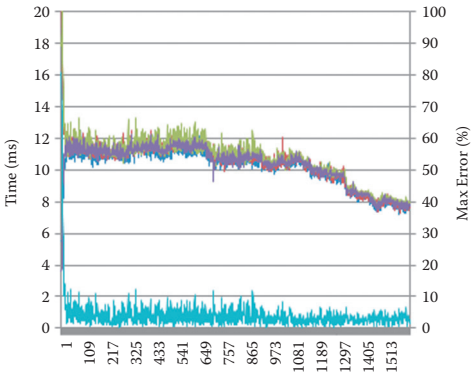
4.5 Acknowledgement

We thank Syoyo Fujita for help in implementing the foveated rendering.



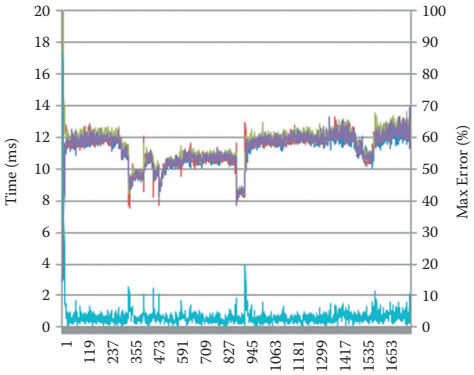
(a) Sample pattern.

(b) Image 1.



(c) Time and error of (b).

(d) Image 2.



(e) Time and error of (d).

Figure 4.4. Foveated rendering on four AMD FirePro W900 GPUs. Samples are only created at white pixels in (a).

Figure 4.5. Bedroom scene rendered using three machines connected via 10-Gb Ethernet. The frame is split horizontally to distribute the work for machines. In each machine, the frame is split vertically on GPUs. We used 4 × AMD Radeon HD 7970, 2 × AMD Radeon HD 6970, and 1 × AMD Radeon HD 6850.

Bibliography

[Advanced Micro Devices, Inc. 16] Advanced Micro Devices, Inc. "AMD Radeon Dual Graphics." http://www.amd.com/en-us/innovations/software-technologies/dual-graphics, 2016.

[Cosenza et al. 08] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. "On Estimating the Effectiveness of Temporal and Spatial Coherence in Parallel Ray Tracing." In *Eurographics Italian Chapter Conference*, pp. 97–104. Aire-la-Ville, Switzerland: Eurographics Association, 2008.

[Guenter et al. 12] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. "Foveated 3D Graphics." *ACM Trans. Graph.* 31:6 (2012), 164:1–164:10.

[Heirich and Arvo 98] Alan Heirich and James Arvo. "A Competitive Analysis of Load Balancing Strategiesfor Parallel Ray Tracing." *J. Supercomput.* 12:1-2 (1998), 57–68.

[Mitchell 87] Don P. Mitchell. "Generating Antialiased Images at Low Sampling Densities." *SIGGRAPH Comput. Graph.* 21:4 (1987), 65–72.

[Moloney et al. 07] Brendan Moloney, Daniel Weiskopf, Torsten Möller, and Magnus Strengert. "Scalable Sort-First Parallel Direct Volume Rendering

with Dynamic Load Balancing." In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*, pp. 45–52. Aire-la-Ville, Switzerland: Eurographics Association, 2007.