CHAPTER 10

A Simple Load-Balancing Scheme with High Scaling Efficiency

Dietger van Antwerpen, Daniel Seibert, and Alexander Keller
NVIDIA

ABSTRACT

This chapter describes an image partitioning scheme that can be used to distribute the work of computing pixel values across multiple processing units. The resulting workload distribution scheme is simple to implement, yet effective.

10.1 INTRODUCTION

A key question in attempts to distribute the rendering of a single image frame over a number of processing units is how to assign pixels to processing units. In the context of this chapter, we will use an abstract notion of a processing unit or *processor*. For example, a processor could be a GPU, a CPU core, or a host in a cluster of networked machines. A number of processors of various types will generally be combined into some form of rendering system or cluster.

This chapter is motivated by the workloads commonly found in path tracing renderers, which simulate the interaction of light with the materials in a scene. Light is often reflected and refracted multiple times before a light transport path is completed. The number of bounces, as well as the cost of evaluating each material, varies dramatically among different areas of a scene.

Consider, for example, a car in an infinite environment dome. Rays that miss all geometry and immediately hit the environment are extremely cheap to compute. In contrast, rays that hit a headlight of the car will incur higher ray tracing costs and will bounce around the reflectors of the headlight dozens of times before reaching the emitters of the headlight or exiting to the environment. Pixels that cover the headlight may thus be orders of magnitude more costly to compute than pixels that show only the environment. Crucially, this cost is not known a priori and thus cannot be taken into account to compute the optimal distribution of work.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_10

10.2 REQUIREMENTS

An effective load balancing scheme yields good scaling over many processors. Computation and communication overhead should be low so as not to negate speedups from a small number of processors. In the interest of simplicity, it is often desirable to assign a fixed subset of the workload to each processor. While the size of the subset may be adapted over time, e.g., based on performance measurements, this re-balancing generally happens between frames. Doing so yields a scheme that is static for each frame, which makes it easier to reduce the amount of communication between the load balancer and the processors. A proper distribution of work is crucial to achieving efficient scaling with a static scheme. Each processor should be assigned a fraction of the work that is proportional to the processor's relative performance. This is a nontrivial task when generating images using ray tracing, especially for physically based path tracing and similar techniques. The situation is further complicated in heterogeneous computing setups, where the processing power of the various processors varies dramatically. This is a common occurrence in consumer machines that combine GPUs from different generations with a CPU and in network clusters.

10.3 LOAD BALANCING

We will now consider a series of partitioning schemes and investigate their suitability for efficient workload distribution in the context we have described. For illustration, we will distribute the work among four processors, e.g., four GPUs on a single machine. Note, however, that the approaches described below apply to any number and type of processors, including combinations of different processor types.

10.3.1 NAIVE TILING

It is not uncommon for multi-GPU rasterization approaches to simply divide the image into large tiles, assigning one tile to each processor as illustrated on the left in Figure 10-1. In our setting, this naive approach has the severe drawback that the cost of pixels is generally not distributed uniformly across the image. On the left in Figure 10-1, the cost of computing the tile on the bottom left will likely dominate the overall render time of the frame due to the expensive simulation of the headlight. All other processors will be idle for a significant portion of the frame time while the processor responsible for the blue tile finishes its work.

Figure 10-1. *Left: uniform tiling with four processors. Right: detail view of scanline-based work distribution.*

Additionally, all tiles are of the same size, which makes this approach even less efficient in heterogeneous setups.

10.3.2 TASK SIZE

Both issues related to naive tiling can be ameliorated by partitioning the image into smaller regions and distributing a number of regions to each processor. In the extreme case, the size of a region would be a single pixel. Common approaches tend to use scanlines or small tiles. The choice of region size is usually the result of a trade-off between finer distribution granularity and better cache efficiency.

If regions are assigned to processors in a round-robin fashion, as illustrated on the right in Figure 10-1, rather than in contiguous blocks, workload distribution is much improved.

10.3.3 TASK DISTRIBUTION

Since the cost of individual pixels is unknown at the time of work distribution, we are forced to assume that all pixels incur the same cost. While this is generally far from true, as described earlier, the assumption becomes reasonable if each set of pixels assigned to a processor is well-distributed across the image [2].

To achieve this distribution, an image of n pixels is partitioned into m regions, where m is significantly larger than the number of available processors, p. Regions are selected to be contiguous strips of s pixels such that the image is divided into $m = 2^b$ regions. The integer b is chosen to maximize m while keeping the number of pixels s per region above a certain lower limit, e.g., 128 pixels. A region size of at least $s = \lceil n/m \rceil$ is needed to cover the entire image. Note that it may be necessary to slightly pad the image size with up to m extra pixels.

The set of region indices $\{0, \ldots, m-1\}$ is partitioned into p contiguous ranges proportional to each processor's relative rendering performance. To ensure a uniform distribution of regions across the image, region indices are then permuted in a specific, deterministic fashion. Each index i is mapped to an image region j by reversing the lowest b bits of i to yield j. For example, index $i = 39 = 00100111_2$ is mapped to $j = 11100100_2 = 228$ for $b = 8$. This effectively applies the radical inverse in base 2 to the index. The chosen permutation distributes the regions of a range more uniformly across the image than a (pseudo-)random permutation would. An example of this is illustrated in Figure 10-2 and in the pseudocode in Listing 10-1, where $\lfloor x \rceil$ means rounding to nearest integer.

Figure 10-2. *Adaptive tiling for four processing units with relative weights of 10% (red), 15% (yellow), 25% (blue), and 50% (green). Note how the headlight pixels are distributed among processing units.*

Listing 10-1. *Pseudocode outlining the distribution scheme.*

```
const unsigned n = image.width() * image.height();
const unsigned m = 1u << b;
const unsigned s = (n + m - 1) / m;
const unsigned bits = (sizeof(unsigned) * CHAR_BIT) - b;

// Assuming a relative speed of w_k, processor k handles
// ⌊w_k m⌉ regions starting at index base = Σ_{i=0}^{k-1} ⌊s_i m⌉.

// On processor k, each pixel index i in the contiguous block
// of s⌊w_k m⌉ pixels is distributed across
// the image by this permutation:
const unsigned f = i / s;
const unsigned p = i % s;
const unsigned j = (reverse (f) >> bits) + p;

// Padding pixels are ignored.
if (j < n)
    image[j] = render(j);
```

The bit reversal function used in the permutation is cheap to compute and does not require any permutation tables to be communicated to the processors. In addition to the straightforward way, bit reversal can be implemented using masks [1], as shown in Listing 10-2. Furthermore, CUDA makes this functionality available in the form of the `__brev` intrinsic.

Listing 10-2. *Bit reversal implementation using masks.*

```
1 unsigned reverse(unsigned x) // Assuming 32 bit integers
2 {
3     x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
4     x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
5     x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
6     x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
7     return (x >> 16)  | (x << 16);
8 }
```

For production scenes, the regions are unlikely to correlate strongly with image features due to differing shapes. As a result, the pixels assigned to each processor are expected to cover a representative portion of the image. This ensures that the cost of a task becomes roughly proportional to the number of pixels in the task, resulting in uniform load balancing.

10.3.4 IMAGE ASSEMBLY

In some specialized contexts, such as network rendering, it is undesirable to allocate and transfer the entire framebuffer from each host to a master host. The approach described in Section 10.3.3 is easily adapted to cater to this by allocating only the necessary number of pixels on each host, i.e., $s\,\lfloor w_k m \rfloor$. Line 18 of Listing 10-1 is simply changed to write to `image[i-base]` instead of `image[j]`.

A display image is assembled from these permuted local framebuffers. First, the contiguous pixel ranges from all processors are concatenated into a single master framebuffer on the master processor. Then, the permutation is reversed, yielding the final image. Note that the bit reversal function is involutory, i.e., its own inverse. This property allows for efficient in-place reconstruction of the framebuffer from the permuted framebuffer, which is shown in Listing 10-3.[1]

[1]Note the use of the same reverse function in both the distribution (Listing 10-1) and the reassembly of the image (Listing 10-3).

Listing 10-3. *Image assembly.*

```
// Map the pixel index i to the permuted pixel index j.
const unsigned f = i / s;
const unsigned p = i % s;
const unsigned j = (reverse(f) >> bits) + p;

// The permutation is involutory:
// pixel j permutates back to pixel i.
// The in-place reverse permutation swaps permutation pairs.
if (j > i)
    swap(image[i],image[j]);
```

10.4 RESULTS

Figure 10-3 illustrates the differences in per-pixel rendering cost of the scene shown in Figure 10-1. The graphs in Figure 10-4 compare the scaling efficiency of contiguous tiles, scanlines, and two types of strip distributions for the same scene. Both strip distributions use the same region size and differ only in their assignment to processors. Uniformly shuffled strips use the distribution approach described in Section 10.3.3.

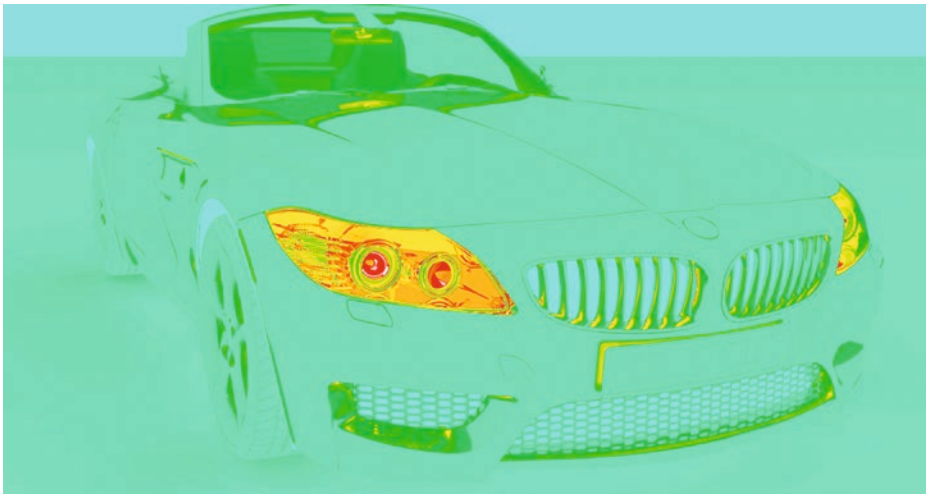

Figure 10-3. *Heat map of the approximate per-pixel cost of the scene shown in Figure 10-1. The palette of the heat map is (from low to high cost) turquoise, green, yellow, orange, red, and white.*

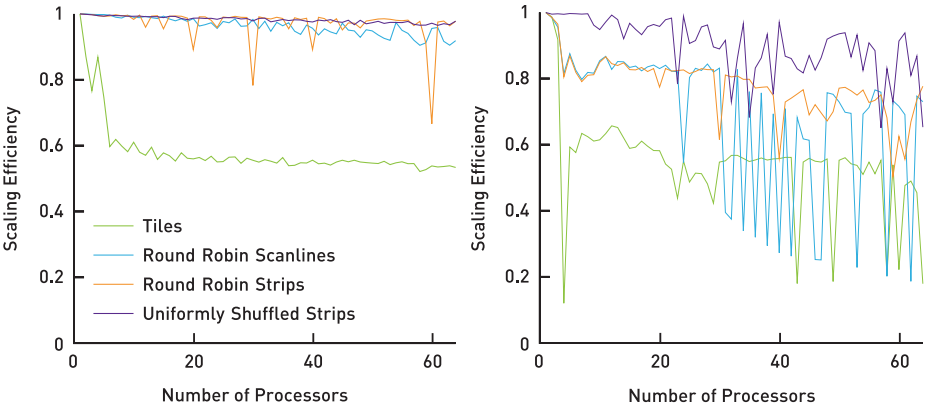

Figure 10-4. *Scaling efficiency of different workload distribution schemes for the scene shown in Figure 10-1. Left: the processors are identical. Right: the processors have different speeds.*

The predominant increase in efficiency shown on the left in Figure 10-4, especially with larger processor counts, is due to finer scheduling granularity. This reduces processor idling due to lack of work. The superior load balancing of the uniformly shuffled strips becomes even more obvious in the common case of heterogeneous computing environments, as illustrated on the right in Figure 10-4.

REFERENCES

[1] Dietz, H. G. The Aggregate Magic Algorithms. Tech. rep., University of Kentucky, 2018.

[2] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, https://arxiv.org/abs/1705.01263, 2017.