COST-BASED WORKLOAD BALANCING FOR RAY TRACING ON A HETEROGENEOUS PLATFORM

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Mario Rincón-Nigro

December 2012

COST-BASED WORKLOAD BALANCING FOR RAY TRACING ON A HETEROGENEOUS PLATFORM

Mario Rincón-Nigro

APPROVED:

Zhigang Deng, Advisor
Dept. of Computer Science, UH

Barbara Chapman
Dept. of Computer Science, UH

Zhu Han
Dept. of Electrical and Computer Engineering, UH

Mark A. Smith
Dean, College of Natural Sciences and Mathematics

Acknowledgments

I would like to thank my family and friends for all their love and support throughout this endeavor: Maria Eugenia, José Gregorio, Eugenia, Gonzalo, Alejandra, Paquita, Teresa, Maigualida, Clarisa, José Manuel, Vitalia, Ana Karina, Glaisa, Alexander, Carlos, Nelson, Andrea, Luz, Sandra, Miche, Sandrita, Michu, Anselmo, Gian Carlo, and Junior.

I would also like to thank everyone at the UH Computer Graphics and Interactive Media Lab, for they have been my other family.

COST-BASED WORKLOAD BALANCING FOR RAY TRACING ON A HETEROGENEOUS PLATFORM

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Mario Rincón-Nigro

December 2012

Abstract

Heterogeneous computational platforms consisting of CPUs and multiple discrete GPUs are becoming common and widely available these days. Such availability has brought up the need to develop techniques that allow the effective use of the computational capability they offer. In this work we investigate efficient strategies for load balancing of ray tracing on heterogeneous (CPU and multi-GPU) workstations. The main difficulty in achieving a high efficiency for ray tracing in this context has to do with the fact that although an embarrassingly parallel problem, ray tracing also exhibits a highly irregular workload. We propose an approach based on fast ray traversal cost estimation to improve the balancing efficiency among GPU and reduce overall rendering times. An accurate and low-overhead cost estimation of ray tracing tasks is performed by means of a reduced traversal of the rays through bounding volume hierarchies. Our estimation exploits the capabilities of modern GPUs to quickly collect information about the number of primitive intersection tests performed by batches of rays. These estimated costs are then used to achieve a more accurate assignation of tasks to processing units. We conduct a comparison between commonly used static and dynamic load balancing strategies, with and without the cost-based enhancement. Our results show that the rendering times achieved by a static cost-based strategy outperforms its static regular counterpart, and both dynamic strategies, on a heterogeneous platform.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays Graphics Processing Units (GPUs) [38] are being extensively used to accelerate graphics applications as well as general purpose scientific computations. It is common for mainstream computing platforms to include a CPU and one or more discrete GPUs available to do computations. Clusters of GPUs are also becoming popular in scientific computing. A well-known example, Tianhe-1A, one of the current fastest supercomputers in the world, makes use of 7,168 NVIDIA Tesla M2050 GPUs and 14,336 Intel Xeon X5670 CPUs to achieve a theoretical peak performance of 4.701 petaflops [53]. The increasing availability of heterogeneous computing platforms at all scales has consequently brought up the need to develop techniques that effectively use all of their available resources, as well as providing efficient ways to map well-known applications to these resources.

Ray tracing is one of the most widely used rendering techniques in computer graphics [62, 14, 32, 48, 23, 63]. It is the base of numerous rendering algorithms for

synthesizing photo-realistic images, such as path tracing [50], ambient occlusion [64], photon mapping [28, 27, 59], metropolis light transport [56], subsurface scattering [45, 29], and volume rendering [31, 37]. Compared to raster approaches, ray tracing makes it much easier to simulate phenomena such as reflection, refraction, shadows, diffuse phenomena, and the effects of indirect illumination in general. Furthermore, the core of the technique also occupies a prominent position in disciplines such as optics and acoustics where it is used to calculate the paths followed by waves [35]. On the other hand, ray tracing is also known to be a computationally intensive application, and much of the research work devoted to it has been targeted to developing efficient techniques to accelerate its computations. One such speed-up trend is doing ray tracing on parallel computing platforms. The main difficulty in achieving a high efficiency for a parallel implementation of ray tracing has to do with the fact that it exhibits a highly irregular workload. That is because the traversal of the hierarchical spatial acceleration data structures that are commonly used to reduce the number of computations (e.g., uniform grids, octrees, KD-trees, BVHs) [6], require different computational effort for different rays.

Traditionally in parallel ray tracing, the irregular load is handled through task systems [24]. The set of rays to be traced is split into a number of tasks which are assigned to the processing units following a predefined scheduling policy. The main challenge for efficiently ray tracing on a parallel platform is thus the balancing of the two goals: (1) maximizing the use of resources through effective task scheduling, and (2) minimizing the communicational and computational overhead associated with the scheduling. Load balancing strategies can be roughly classified into static and

dynamic strategies. Static scheduling strategies (i.e. a priori assignation of tasks to processing units) incurs in low scheduling overhead while exhibiting high load imbalance. On the other hand, dynamic scheduling strategies exhibit much better load balance while incurring in much higher scheduling overhead. The granularity of a task system, refers to the size of the tasks. A fine-grained (i.e. small tasks) system usually achieves better balancing than a coarse-grained system, at the cost of increased scheduling overhead, for smaller tasks translates into a larger number of tasks to be scheduled.

In this thesis we propose a task system approach to load balancing based on the estimation of the computational cost of ray tracing tasks. After, ray generation we perform a reduced traversal of the BVH for a subset of the rays within a task in order to quickly estimate the number of primitive intersection tests involved in the full trace operation of the entire task. The estimated cost of the tasks is then used to do a more accurate assignation of tasks to processors. The approach is meant to improve load balancing, and reduce scheduling overheads and data transfers, at the expense of the computational effort required to get an accurate estimate of the task costs. Our approach is applied to static and dynamic load balancing strategies, specifically a *distributed static task assignation*, and a *centralized queue*.

The main contribution of our work is the proposal of an instrumental low-overhead method for the estimation of task costs in ray tracing, and its application to load balancing in multi-GPU systems. Our cost estimation method makes efficient use of the capabilities offered by modern GPUs, and achieves high accuracy at low computational cost. In addition, we report on a quantitative comparison of static

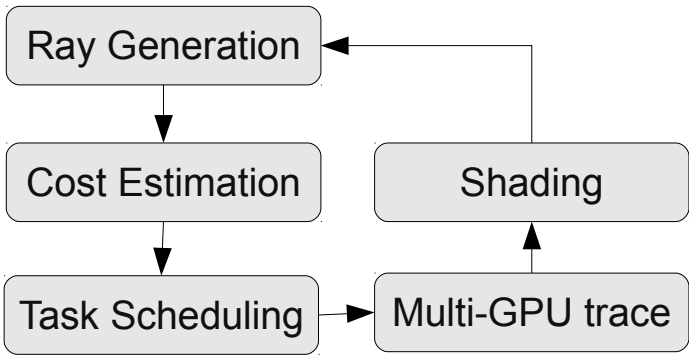

Figure 1.1: Cost-based load balancing approach.

and dynamic load balancing strategies in heterogeneous environment. We evaluate the effect of the proposed cost-based enhancement on the considered load balancing strategies. Our results demonstrate that in a heterogeneous environment a well-initialized static assignation of tasks measurably outperforms a dynamic strategy.

1.1 Overview of the Approach

The time cost of tracing a ray through a spatial acceleration structure is proportional to the number of primitive intersection tests it performs in the process. The basic idea of our approach is to collect information about the number of primitive intersection tests that tracing rays within tasks is going to generate. We then take advantage of such information for the purpose of achieving a better load balance across processing units.

Figure 1.1 shows a schematic view of our approach. The pipeline starts with the creation of a generation of rays (i.e. primary rays, shadow rays, reflection rays, or diffuse rays) that are split into tasks (i.e. fixed size sets of rays). The cost of each

of the task is then estimated by performing a reduced traversal over the BVH for a subset of the rays conforming a task. By reduced traversal we mean that it does not perform an intersection point query against the scene geometric primitives, but rather it results on the number of primitives intersection tests that need to be carried in order to answer such a query (details in section 3.1). The estimated cost is then utilized to perform an informed task scheduling. The goal of the scheduling is to partition tasks across processors, such that each of them is assigned an equivalent amount of work. After tracing a generation of rays, secondary rays are handled in a similar fashion. That is, ray-primitive intersections may create new generations of rays which are handled in the same way as the previous generation.

1.2 Background

This subsection provides a brief review of ray tracing, and bounding volume hierarchies, for this topics are central to our work. A brief introduction to GPU programming with the CUDA framework, its execution model, and the architectural features of modern GPUs is also provided, as these concepts are heavily used throughout the following chapters.

Ray Tracing

Ray tracing at its core consists of finding the intersection points (also known as *hits*) between a set of directed rays and a number of geometric primitives. In order to

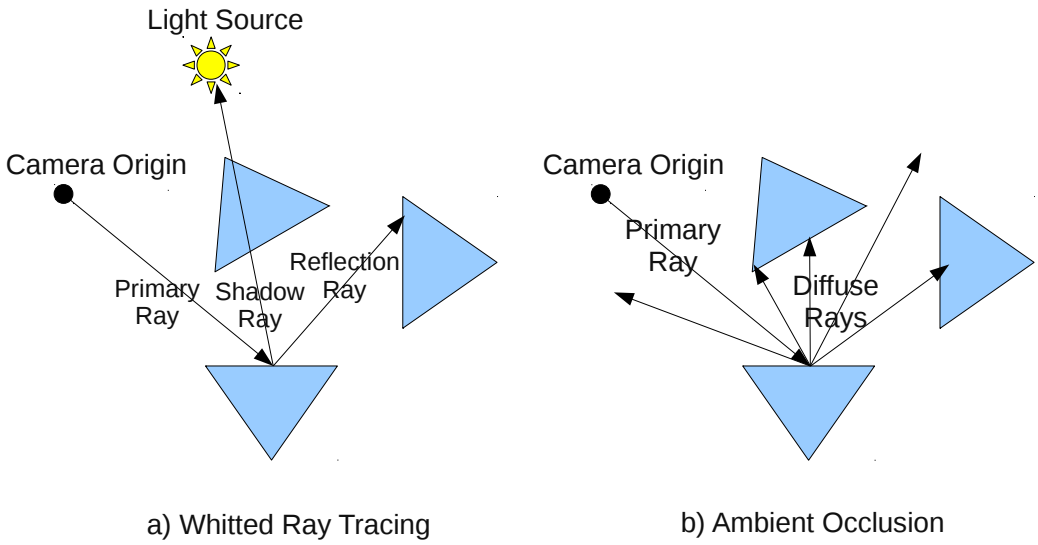

Figure 1.2: Ray tracing general procedure. a) Whitted (or classical) ray tracing. b) Ambient occlusion

render an image, a set of rays is directed from a specified point (called the *camera origin*), through a plane into a region of interest within a scene. The plane represents the image to be rendered, and each of the pixels within the image needs to correspond to at least one of the rays. This initial group of rays is known as *primary rays*. For each primary ray the closest intersection point with the geometric primitives within the scene has to be determined. The color of the pixel associated with the ray is determined by the material properties of the intersected primitive, and by the configuration of the light sources within the scene [62]. The procedure of coloring the pixels according to material and lighting configuration is known as *shading*. The shading model can be classified as local or global, depending on whether it models the effect of direct illumination or also handles the effects of indirect illumination. By indirect illumination we refer to the contribution of rays of light not coming directly from the light source, but rather that are reflected by other objects. Figure

1.2 illustrates the general procedure of classical ray tracing. Examples of renderings obtained through ray tracing are shown in Figure 1.3.

Complex indirect illumination effects, can be simulated by generating *secondary rays* from the intersection points. To create shadowing effects, a secondary ray (known as *shadow ray*) is created with origin at the intersection point of the primary ray, and directed towards the light sources. If the shadow ray intersects any image with the scene it means that the color of the pixel associated to the primary ray has to be attenuated. The effect of reflection is achieved in a similar fashion by bouncing the primary rays on the intersection point. For an in-depth description of these procedures see [60, 18]. This simple technique is the base of many other renderings algorithms for creating photo-realistic artificial imagery [28, 17]. Some of the results in our work are presented in terms of one of those algorithms, namely *ambient occlusion*. It generates the soft-shadowing effect caused by a global pseudo-light source, and it is significantly more computationally demanding than the classic ray tracing algorithm. In ambient occlusion, for each primary ray hit, a number of randomly directed rays (known as *diffuse rays*) are generated for each primary ray hit. All of the diffuse rays are tested for intersection against the scene geometry. The color of the pixel associated to the primary ray is then attenuated by a factor proportional to the number of diffuse rays that intersect some geometric primitive. Ambient occlusion is a heavier algorithm than ray tracing because it requires the processing of a larger number of rays to generate high-quality images (using few samples of diffuse ray generate noisy images). In addition to that, diffuse rays exhibit greater variability than the rays involved in classical ray tracing. Rays with low variability

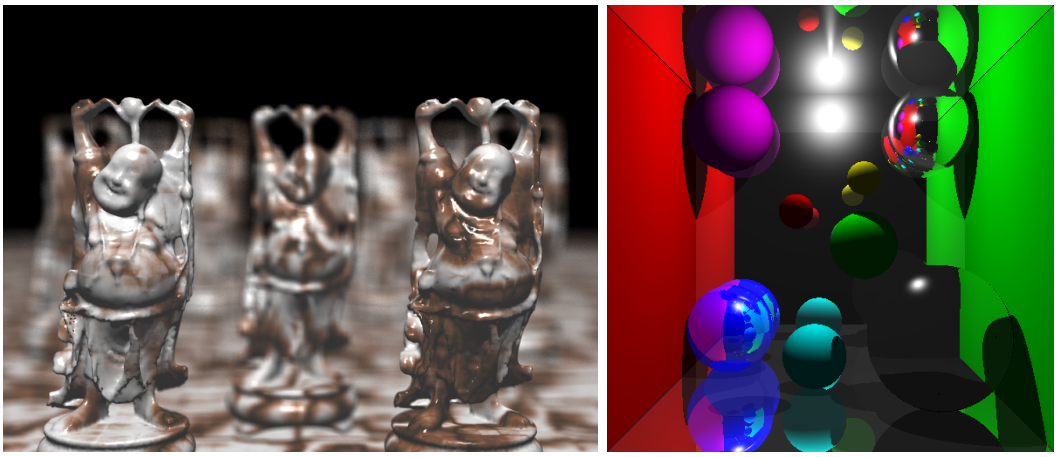

Figure 1.3: Example of synthetic images rendered using ray tracing. Left: depth-of-field effect. Right: sharp shadows, reflection, and refraction effects.

are easier to trace than diffuse rays, for adjacent rays usually follow similar paths on spatial acceleration data structures, making it easy to take advantage of the processor architectural features (e.g. exploiting cache coherency). In the following chapters we refer to the rays involved in classical ray tracing (i.e. primary rays, shadow rays, and reflection rays) as *coherent rays*.

Bounding Volume Hierarchies

To avoid a large bulk of the computations required by ray tracing, without sacrificing precision, it is common to use acceleration spatial data structures. A popular acceleration structure for ray tracing is the Bounding Volume Hierarchy (BVH) [32, 6], since they are inexpensive to compute, and they can be quickly restructured to support scenes that may go through dynamic deformations. It is noted that while the

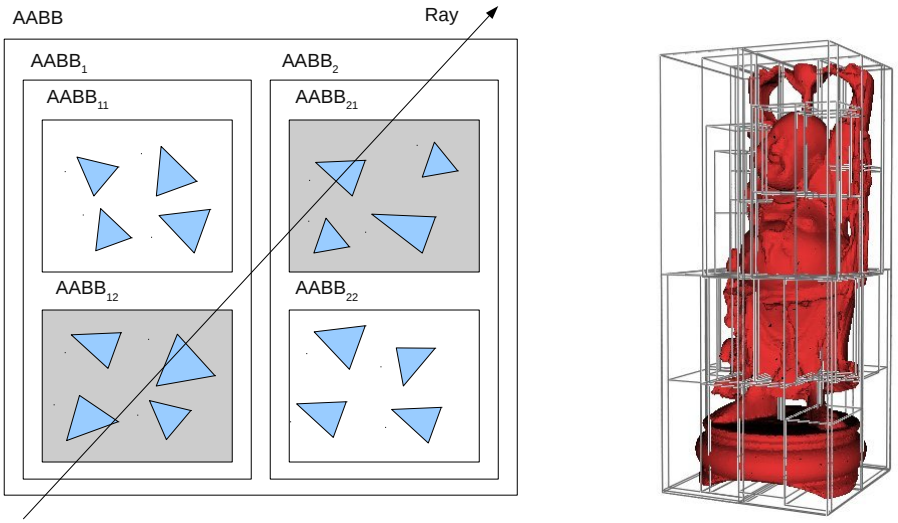

Figure 1.4: Bounding Volume Hierarchies (BVH). Left: 2D example of a BVH. Right: BVH for a complex 3D model, down to a depth of six.

latter is a general case, in this work we only consider static scenes. A BVH is a partition of geometric objects into a hierarchy that can be used to accelerate geometric queries as the ones involved in ray tracing. Formally, BVHs are binary trees, where each node represents a group of objects and the region in space that they occupy. Children nodes are recursive partitions of the group of objects within their parent nodes. Figure 1.4 (left) shows a 2D examples of a BVH, and a BVH for a complex 3D mesh (right). A given scene can be embedded into many different BVHs. By that we mean that there is not a unique way to construct a BVH for a given scene. Common approaches to building BVHs involve recursively splitting the BVH nodes by half [52, 41], or by minimizing the expected number of primitive intersection tests involved in traversing each child node [21, 23].

A geometric query over the BVHs involves a traversal of the data structure. For ray tracing we need to figure out the point where each ray intersects geometric

primitives within the scene. Starting from the root, at each step of the traversal, an intersection test against the children nodes is performed. If a node is intersected, then the traversal must be continued in a recursive manner over the substructure for which the node is root. Otherwise, the whole substructure can be safely ignored, because it is guaranteed that none of the geometric primitives that it holds can be intersected by the ray. Geometric primitives are actually contained within the leaf nodes of the tree structure, so when a leaf nodes is found, each of its geometric primitives must be tested for intersection against the directed ray. The traversal stops when a geometric primitive hit by the ray is found, or when all the geometric primitives within the intersected nodes have been tested. Figure 1.4 illustrates this idea. The BVH traversal starts with a test of the box labeled AABB, and as this is intersected continue with tests of both its children ($AABB_1$ and $AABB_2$). $AABB_1$ is intersected, so both its children must be tested as well. $AABB_{11}$ is not intersected by the path (going further down that node can be safely discarded), but $AABB_{12}$ is, and since it is a leaf node, all the primitives it contains must be tested for intersection. In the case of $AABB_2$, we can see that only the children labeled $AABB_{21}$ must be tested for intersection.

GPU Programming

Data parallel applications such as ray tracing are ideal candidates for implementation on massively parallel processors such as GPUs. Individual rays can be processed in complete independence to other rays within the same generation. To describe our ray tracing implementation, as well as our approach to load balancing on a heterogeneous

platform featuring multiple GPUs, it is instructive to briefly review their operational structure of single GPUs. The specific details of the following description of the operational structure of GPUs is specific to NVIDIA GPUs [40]. However, there exists a direct architectural equivalence with other kinds of GPUs, so our description still holds valid in a more general context.

A GPU consists of a number of processing units, i.e. the streaming multiprocessors (SMs). In the streaming programming paradigm individual threads are executed within SMs and are assigned a portion of the overall computation. In the CUDA framework the computations to be performed by the threads are specified in device code units known as *kernels*, that are written in a C-like programming language. The full set of threads (known as *grid*) is subdivided into groups known as blocks. The dimensions of the grid, and the blocks are specified as parameters to the kernel invocation on host code. Different blocks of threads are assigned for execution to different SM in a *static manner*. i.e. after a block has been assigned to an SM, it cannot be executed on another SM. After assigned, the threads within a block are further subdivided into groups of 32 consecutive threads, the warps. Threads within a warp are executed in parallel within the SM in Single Instruction Multiple Thread (SIMT) mode. In SIMT mode, each thread in a warp executes the same instruction simultaneously (though not necessarily during the same clock cycle). The scheduling of threads is transparent from the point of view of the programmer. The rationale behind this execution model, is to provide a certain degree of intrisic load balancing within the massivelly parallel processor, and hide long latency operations such as global memory accesses.

Modern GPUs have different kinds of memory available for programmers to use [33, 40]. Most GPUs provide: a large device *global memory*, *shared memory*, *texture memory*, *local memory*, and *registers*. Data transfer between the host and the device can only be done through global memory. Thus, data to be processed in the GPU has to be first transferred through the PCI bus to the device. This data transfers can be performed in a synchronous or asynchronous manner. The texture memory is cached, and is meant to be used for exploit spatial coherency in the data. Global and texture memory are visible to all the threads in the grid. On the other hand, shared memory is accessed at the block level (i.e. only threads within the same block can modify the same section of shared memory). Finally, local memory, and registers, are administered at the thread level. The occupancy of the GPU is determined by a combination of the specified number of blocks, size of assigned shared memory, and number of registers used by the threads. The CUDA programming framework provides means to allocate memory both in device memory and in host memory. Host memory is allocated always as non-pageable memory by the CUDA runtime. Host-device data transfers that involve non-pageable host memory, have usually less latency than data transfers with pageable memory. Capabilities for mapping host memory and device global memory are also provided by the framework.

Another important set of features provided by the CUDA framework is atomic operations such as *compare-and-swap*, and *fetch-and-add*. The compare-and-swap operation, compares the values in two memory locations, and modifies the value of a third memory location based on the result of the comparison. On the other hand, fetch-and-add retrieves a value from memory, performs an addition on the

value and stores the result back into memory. Both operations are performed by the hardware as atomic instructions. Thus, they enable synchronization between threads in different blocks [40], and can be used to avoid race conditions between threads for critical operations over data structures in global device memory.

Chapter 2

Related Work

2.1 Ray Tracing on Parallel Platforms

Ray tracing as one of the most popular rendering techniques has been extensively studied. Early influential works simulated effects such as reflection, refraction, depth of field, motion blur, and soft shadows [62, 2, 14, 5, 30]. As the number of geometric primitives grows, ray tracing becomes a very computationally intensive task. A common approach used to speed up the computations is to create hierarchical structures from the geometric primitives in order to reduce the number of intersection tests that must be performed [32]. Among the most popular acceleration structures we have uniform grids [3], KD-trees [46], and BVHs (bounding volume hierarchies) [36]. A fast algorithm for building BVHs based on the surface area heuristic was proposed in [23]. Real time BVH construction methods in the GPU are described in [36, 41, 20]

The problem of estimating costs for ray traversal has been addressed [4, 49] through analytical models for octrees. The estimator proposed by Aronov et al. [4] computes the average cost of tracing a scene embedded in a particular octree. Their approach is not meant to compute the cost of particular sets of rays, so the information it provides is of no use for workload balancing. Reinhard et al. [49] estimate the expected cost of traversing a voxel within the octree and estimate the cost for a set of rays by adding up the contribution of all the voxels that fall within the ray frustum. Although their estimator is meant to be used for improving balance in parallel ray tracing, it is not satisfactory in terms of accuracy for improving the efficiency of parallel ray tracing. As opposed to these approaches we obtain more accurate estimations by using an instrumental approach instead of accurate. We obtain an exact estimation of a subset of rays by performing a reduced traversal that takes advantage of the capabilities of modern GPUs.

Since GPUs became mainstream and their high performance computing capabilities were recognized, the interest in using them to do ray tracing started growing. Carr et al. [10] were among the first to use GPUs to accelerate ray-tracing. Specifically, they implemented ray-triangle intersection in the GPU as a pixel shader. As the programmability of GPUs improved, a ray tracing mapping to the streaming programming paradigm was described in [48]. Later, interactive ray tracing on the GPU using a KD-tree acceleration data structure was proved feasible in [26]. Günther et al. [23] additionally described an efficient implementation of ray tracing based on packet traversal that is targeted at GPU architectures. The computing capabilities of GPUs for accelerating ray tracing were further explored by Zhou et al. [63].

They proposed an algorithm for building high quality KD-trees on the GPU achieving real-time performance. Most recently, Parker et al. [43] presented the OptiX system,a general purpose ray tracing engine whose pipeline can be programmed by means of a shading-like language. The OptiX system features workload balancing capabilities for multiple different GPUs. They employ a centralized queue approach with coarse-grained tasks, using the workload balancing scheme proposed in [1] for multiple GPUs of distinct performances. Efficient implementations on the CPU that achieve interactive rendering rates have also been reported [58].

Ray tracing on parallel and distributed platforms has also been a subject of study. Heirich and Arvo [24] report a quantitative analysis of workload balancing strategies for ray tracing in clusters. They show that a simple static assignation of tasks results in unacceptable workload imbalance, and that a hybrid strategy including a good static initialization and dynamic workload balancing can provide very efficient results. Load balancing for parallel ray tracing on non-heterogeneous systems has been widely studied [42, 57, 16, 61, 47, 34]. This works describe load balance systems that either distribute geometry among processors, or subsets of the rays (i.e. portions of the final image). Parallel ray tracing systems that make efficient use of computational resources to minimize communication overhead, memory limitations and the capabilities offered by heterogeneous resources have been also reported [15, 9]. De Marle et al. [15] address the problem of communication overhead when rendering large scenes on a distributed environment using a distributed page-based memory system and distributed load balancing. An approach for efficient use of hybrid computational resources was proposed by Budge et al. [15, 9]. Different stages of the ray

tracing pipeline are mapped to the most capable computational resources available in the parallel system.

2.2 Workload Balancing on GPUs

Cederman and Tsigas [11] quantitatively compared four different polices for GPU workload balancing on the task of partitioning a set of points in an octree data structure. Their load balancing methods were either lock-free or lock-based. They showed that inside a GPU, mutual exclusion operations are very expensive, and thus lock-free methods provide better performances. It has been widely believed that the low performance of GPUs in relation to CPUs for ray tracing is mainly due to their memory bandwidth limitations. Aila and Laine [1] showed that the main bottleneck is the workload imbalance caused by the irregularity of ray traversal. In NVidia GPUs threads are executed in parallel in sequences of 32 threads known as warps [40]. In their work it is argued that some of the threads within a warp take more time to per- form the traversal than other threads and this is the major cause of the low performance of the GPUs. In order to cope with this issue, they further proposed the use of persistent threads that enable workload balance in GPUs. The main idea of this technique is that a set of threads executing in SIMT mode retrieve batches of rays to be traversed from a centralized queue, and then these threads finish execution when every ray in the queue have been processed. Their work provides performance results for one of the fastest GPU ray tracers presented to date.

A similar idea to persistent threads was described in [25] where a numbers of

irregular graphs algorithms were evaluated. Following this line, workload balancing for a Reyes renderer [13] on the GPU was shown in [54]. In addition to the above centralized queue approach they evaluate the use of distributed queues with task stealing and task donation. The main difference between our work and previous approaches is that our workload balancing policies are based on a fast estimation of the cost of the tasks to be scheduled to achieve a higher utilization of the computational resources.

Chapter 3

Methodology

3.1 Traversal Cost Estimation

A typical *trace* operation results in the answer to a geometric query, such as finding out the closest intersection point between a large set of geometric primitives, or whether the ray hits any primitive at all. It consists of two stages: i) traversal of nodes within the acceleration structure, and ii) scene geometric primitives intersection tests on reached leaves. The cost of ray traversals can be modeled in terms of the number of geometric primitives intersection tests. Axis-aligned bounding box (AABB) tests are carried on node traversal, and triangle test are performed on. In addition to the arithmetic computations incurred by the intersection tests, each stage also incurs in an overhead associated to primitive data fetching from memory. The cost of traversing a ray r through BVH b can thus be modeled as,

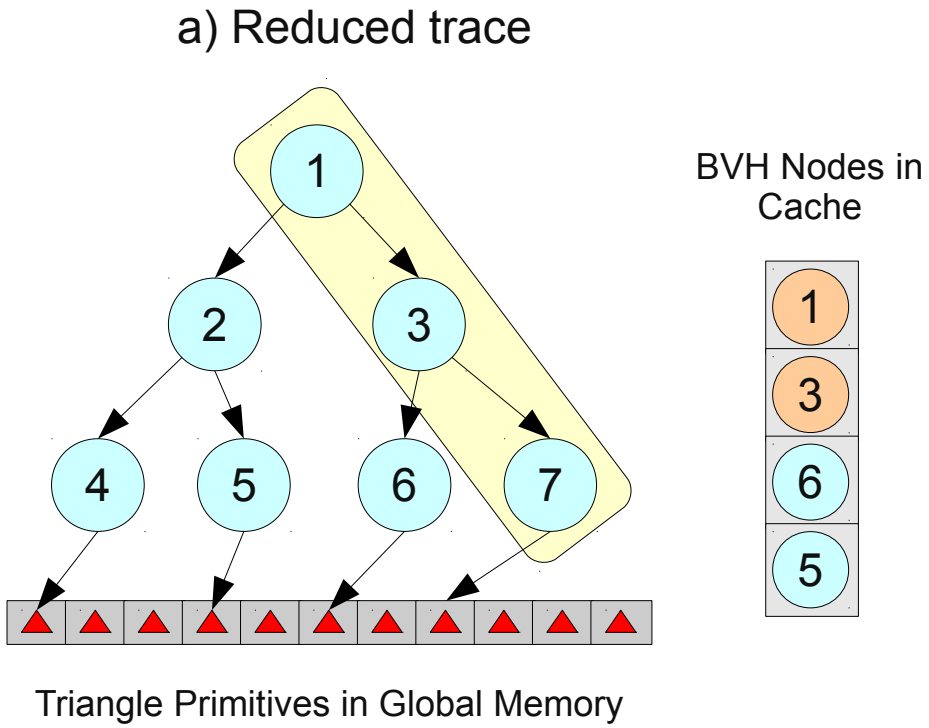

Figure 3.1: Our approach to cost estimation. a) Reduced traversal for individual rays. BVH node caching benefits node traversal. b) Uniform task sampling (in the example 25% of coherent rays are sampled for tasks of size 8 rays).

$$C(r,b) = k_B B(r,b) + k_T T(r,b) \tag{3.1}$$

where k_T and k_B, are constants that account for the cost of intersection tests and memory fetches, and B and T represent the number of traversed nodes and triangle primitives reached by the node. However, in order to be feasible, the estimation must comply with two requirements: first, the estimation should be fast, and the gained speed-up should surpass the estimation overhead; and second, the estimation must be reasonably accurate.

Instead of using an analytical model to obtain B and T, we have opted for an instrumental approach, as none of the existing analytical approaches satisfied our requirements [21, 49, 4]. Figure 3.1 illustrates our instrumental approach. We exploit the capabilities offered by modern GPUs to quickly obtain the exact number of primitive intersection tests that a ray performs. Against all intuitions we compute the number of intersection tests for a particular ray by *performing a reduced traversal of the ray*. A closer look at high performance implementation of ray tracing on GPUs shows why this approach works. High performance implementations (e.g., the work of [1, 43]) rely on texture memory for caching BVH nodes during ray traversals. Triangle primitives on the other hand should not be cached. That is because including the same triangles are not requested nearly as often as the subset of nodes recently traversed. If adjacent rays have similar origin and direction, much of the nodes required for the traversal of the next ray will be found populating the cache. Figure 3.1, shows an example in which the traversal of nodes 1, 3 and 7 incurs in a single cache miss (for node 7) for the given state of the cache. A significantly large

Figure 3.2: Cost image for the Fairy Forest test/benchmark scene, 174117 triangles (left). Ambient occlusion rendering of the Fairy Forest scene (right).

portion of the traversal time on high-performance GPU ray tracing is consequently spent in global memory loads for triangle primitives, which are relatively high latency operations. B and T can thus be collected through the reduced traversal, in a fraction of the time required by the full trace operation, as there is no need to fetch triangles from global memory and the traversal can be accelerated by node caching.

Although it is not possible to get a better estimate of the number of primitive tests than by carrying on the reduced traversal over the full set of rays, it yields a large unwanted overhead as shown in Table 4.1. However, reducing the number of tested rays through uniform sampling provides a good trade-off between the estimation overhead and the estimation error (see Table 4.1). In order to improve similarity among adjacent rays we order coherent rays following a Z-curve or Morton order [51] (Fig. 3.1 b.). In addition to being fast and highly accurate, our approach is

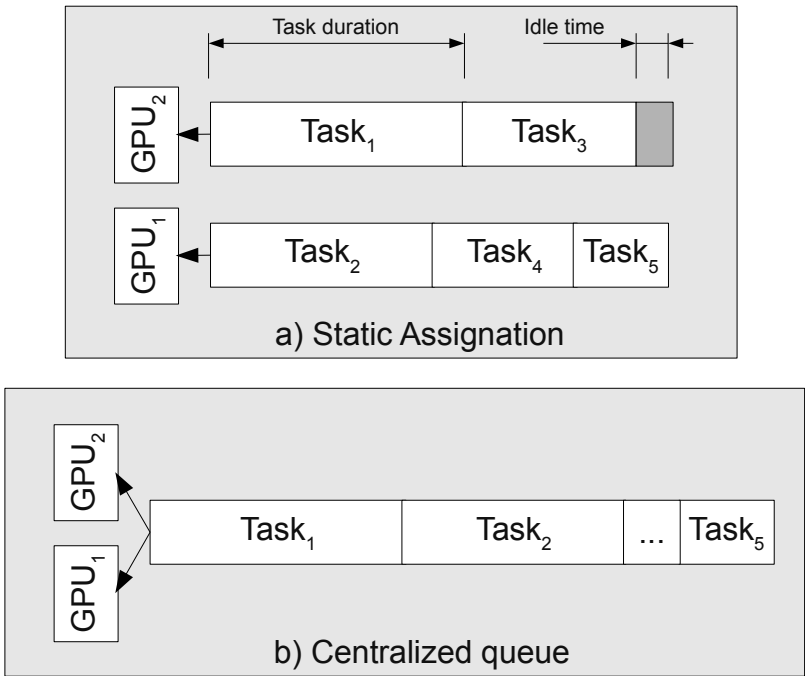

Figure 3.3: Load balancing strategies for a two-GPU configuration.

satisfactory for diffuse rays. Similarity between diffuse rays is increased by generating them through a Halton sequence [7]. We argue that the reason it works for diffuse rays is that since they are generated on the surface of geometric primitives, they start at high density regions in the BVH hierarchy so that their traversal also gets benefits from the cached traversal.

3.2 Cost-based Task Scheduling

We take advantage of obtained information about the cost of tasks to perform a more accurate task scheduling over different load balancing strategies. The goal is to assign each GPU an equivalent amount of work, such that the total rendering time is

reduced. Figure 3.3 illustrate the load balancing strategies considered in this work.

In a *Distributed Static Assignation*, every processing unit has its own task queue. Tasks are distributed prior to the computations and each processing unit is allowed to retrieve tasks only from its own queue. This is the simplest strategy, and the one with the least data transfer overhead. However, in general it leads to low balancing efficiency. The static assignation is prone to large spans of idle times in some of the processing units as shown in Figure 3.3.

In a *Centralized Queue*, every processing unit retrieves tasks from a queue shared by all the processing units. Whenever a processing unit finishes a task, it retrieves another task from the common queue until the queue is emptied. The data transfer as well as synchronization overhead is of course higher than for a static assignation. This more complex strategy leads to better load balance and thus to a lesser make-span time than the static strategy. The centralized queue guarantees that the make-span time is within a factor of 2 from the optimal mapping of tasks to processors. No processing unit is going to be idle as long as there are tasks to process on the centralized queue [55].

Notice than on the ideal case, in which no scheduling/communicational overhead is incurred and all tasks require the same amount of work, both policies are equivalent. On the other hand, if tasks require a different amount of effort, as in ray tracing, the centralized queue is expected to behave better than the static assignation (idle time is guaranteed to be no larger than the last task in the queue). Knowing the exact cost of each of the tasks can be used to improve both balancing strategies.

In order to improve the load balancing properties of both strategies, tasks are first sorted in decreasing order of their estimated costs. Intuitively, we would like to have the longest tasks to be processed at the beginning of the overall computation. For the static task assignation, the assignation of tasks to processors is analogous to the bin-packing problem, and can be solved efficiently through an approximation algorithm [55]. Tasks are sequentially assigned in decreasing order to the processor with the least amount of work. Following the resulting schedule guarantees that the total make-span schedule time is not larger than 4/3 of the optimal time [22].

For a centralized queue, tasks are simply put on the queue in decreasing order. Figure 3.3 shows the resulting configuration for both load balancing strategies when following this schedule. It is interesting to observe that in the ideal case in which no scheduling/communicational overhead is incurred, the schedule resulting from the previously described algorithm make both strategies also equivalent. In a real case scenario both strategies are not expected to behave the equally. That is because the estimated cost of tasks is not exact, and the load balancing strategies have different associated overheads.

3.3 Implementation

3.3.1 GPU Ray Tracer

Since the calculations for each ray are independent from the other rays, the computational task results ideal for implementation on massively parallel processors such

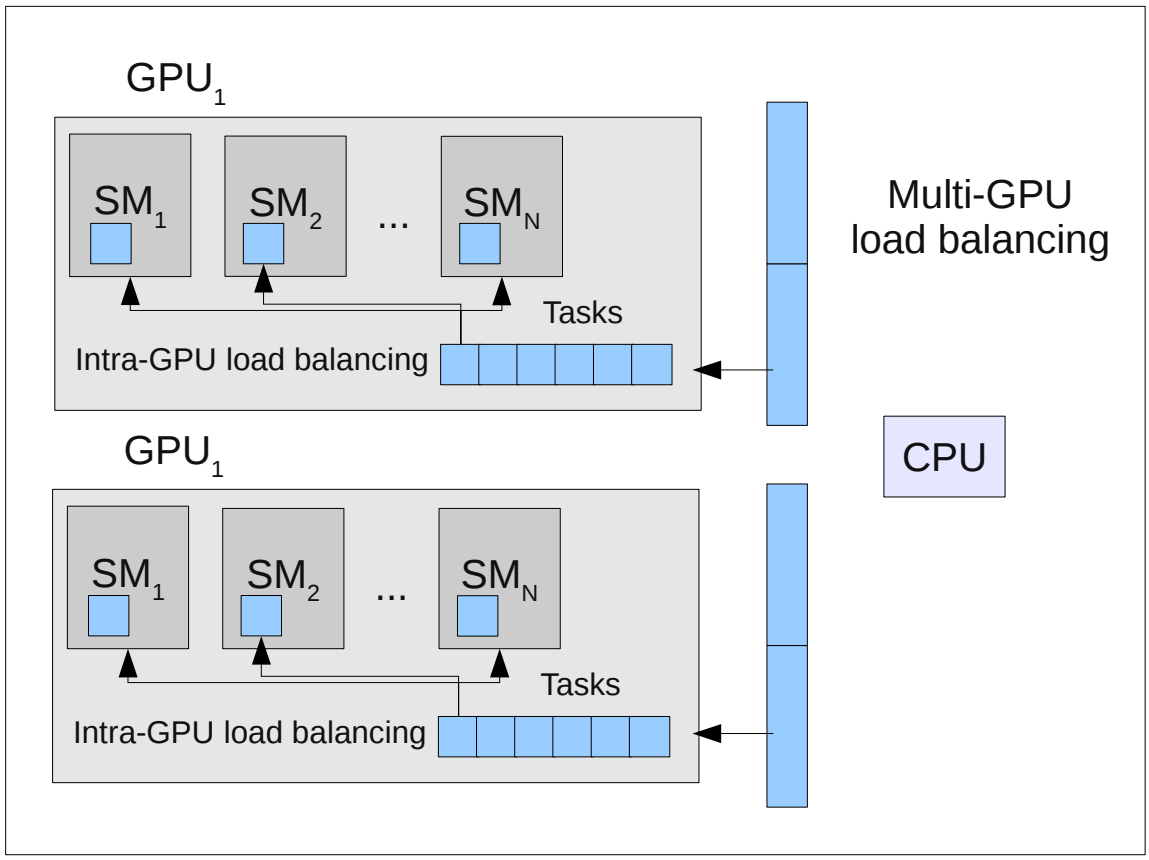

Figure 3.4: Illustration of multi-level balancing. Load is balanced at the multi-GPU level, as well as within GPUs.

as GPUs [38]. For this work we have implemented, using C++ and CUDA, a ray tracer with an integrated load balancer for heterogeneous platforms. It is important to notice that load balancing is performed at two levels in our implementation. Tasks are first distributed between the GPU, and then finer-grained tasks are distributed within the GPU between SMs. Figure 3.4, illustrates this idea. Our ray tracer can render images using the classical ray tracing algorithm, and ambient occlusion. In our proposed approach, the system renders the images by performing the calculations discussed in Section 1.2. In our heterogeneous implementation, all the ray

tracing stages (i.e. ray generation, tracing, reduced traversal for cost estimation, and shading) are performed on the GPUs, and implemented in independent kernels. The host code (that is, code executing on the CPU) on the other hand, manages the GPU contexts, as well as all data transfers, and the multi-GPU load balancing strategy. Among the ray tracing stages, only the tracing operation and the reduced traversal exhibits an irregular workload. By tracing operation we mean the traversal of the ray through the BVH to compute intersection points. The reduced traversal can be performed using any of the regular load balancing strategies, and the tracing operation can use the cost-based approach as well.

Primary rays are generated following a Z-curve (a.k.a. Morton order curve) to further exploit the similarity of coherent rays. A pseudo random Halton-sequence is used to generate the outgoing directions of diffuse rays. The Halton-sequence was used, because it is fast to generate, and provides a good coverage of the space even for few samples. It also enforces similarity between adjacent diffuse rays, which is beneficial for the cost prediction stage. BVHs are built splitting nodes by half, as explained in [52]. The BVH data is replicated in each of the devices, as well as the ray buffers. Ray buffers and hit buffers are kept in device memory. Though access to the buffers is sequential and coalesced within the kernels, it was measured that the global device memory provides a better performance than the host mapped memory. Host counter-parts of ray and hit buffers are kept in non-pageable memory. Ray and hit buffers are stored as structures of arrays.

The cost of rays is estimated following the approach described in section 3.1. Rays are sampled in uniform intervals, and the subset of sampled rays to be traversed is

compacted as explained in [44]. This guarantees coalesced memory reads for the reduced trace operation over the subset. The reduced ray traversal is performed using the persistent threads approach without speculative traversal as explained in [1]. After the cost of each task is estimated rays are not shuffled/sorted according to their cost, as it involves a large unwanted overhead. The task order is rather kept in a buffer and it is transferred to the devices before the tracing kernels are launched.

Block sizes that lead to a maximum occupancy are determined for every kernel by using the CUDA occupancy calculator [39]. The number of registers for each kernel is kept under 20 to achieve a maximum occupancy. Different generations of rays are handled through different kernel launches in our implementation. That is, secondary rays are not handled by the same kernel launch that processes the hits from which they are created. This is mainly due to the fact that the large number of secondary rays created for diffuse rays cannot be hold in global device memory in some cases (e.g., in ambient occlusion). For each generation of rays, costs are estimated by using the static distributed approach, and then they are traced.

Our multi-GPU load balancer works in two modes: *regular mode*, and *cost-based mode*. In both modes it supports the two balancing strategies described in section 3.2. Tasks are determined by: a *start index*, indicating the first ray within the task; a *task size*, indicating the number of rays comprising the task; and a *task cost*.

3.3.2 Multi-GPU Load Balancing

Our load balancing system in a heterogeneous platform first distribute tasks at the multi-GPU level. Tasks are kept in a queuing system in host memory and are distributed between the processing units according to one of the strategies mentioned bellow. In this work we only consider processing units of similar computing capability. However, the extension to devices of different capabilities is straightforward, as it only involve an adjust on the number of assigned tasks (for regular load balance), or an adjust on the estimated cost (for cost-enhanced workload balance).

At the multi-GPU level there is a trade-off between the balance efficiency and the communication overhead with respect to the size of tasks. Coarse-grained tasks for irregular workloads usually leads to workload imbalance between the GPUs. On the other hand fine-grained tasks leads to a high overhead due to data transfer between host memory and device memory through the PCI-E bus, which is a high latency operation.

Multi-GPUs Distributed Static Balancing

Initialization for the static distributed queue strategy is done as follows. In the regular mode the load balancer considers every task to have the same cost and evenly distributes all the tasks among available GPUs. In cost-based mode it follows the schedule explained in section 3.2. A queue is created for each of the available GPUs and tasks are successively assigned one-by-one to the queue with the least workload at each step. As mentioned earlier, ray buffers are not shuffled or sorted according to their costs. For the static balancing strategy they are just copied to the devices,

along with a buffer containing pairs of indexes indicating the start and the end of every their assigned tasks.

Multi-GPUs Centralized Queue

In regular mode tasks are inserted in the queue in the order they appear in the ray buffer. On cost-based mode it is initialized by sorting the tasks according to their estimated costs. As in the static case, ray and hit buffers are copied to the devices once. Processing each of the tasks within our centralized queue involves one kernel launch for each task. This is due to the fact that in this dynamic approach processing units need to synchronize between each other, and CUDA provides no direct means to achieve this without interrupting the kernels. One host thread handles the execution of tasks in each device. Once a kernel ends, the thread retrieves a task from a thread safe queue and launches the trace kernel to process it again. This process repeats until there are no more tasks that need to be processed.

3.3.3 Intra-GPU Load Balancing

Workload is further balanced at the GPU level. In the CUDA programming model threads to be executed within the GPU are organized into a grid a blocks. Blocks are assigned to GPU cores for execution, and these are further subdivided for execution into warps. Warps are scheduled to execute in the GPU cores and their threads run in SIMT (Single Instruction Multiple Thread) mode, similar to SIMD mode but with the flexibility that threads execution paths are allowed to diverge.

In spite of the intrinsic parallelizable nature of the problem, achieving high performance on GPUs is not straightforward. The origin of this is that the computations required for each ray are highly variable, and this irregular workload leads to underutilization of GPU resources. To describe the presented GPU implementation, it is instructive to briefly review its operational structure (see Section 1.2). Though the division of work provided by the CUDA runtime provides workload balancing within the GPU to some extent, it can be improved for applications such as ray tracing. The threads that require the most time for performing traversal held hostage computational resources degrading thus performance.

In the problem of computing ray intersections, a naive scheme of assigning work to processing units in CUDA or OpenCL [8] would correspond to launch the execution of a grid with as many threads as the rays. Therefore, the computations for each thread would be performed in a single thread. Though the GPU execution model provides intrinsic capabilities for load balancing, they are not totally satisfactory for ray tracing due to the fact that thread blocks are statically assigned to SMs. As a result, the high variability of the required computations for each ray will lead to resource underutilization. Practically, the workload assigned to some blocks will need more time than other blocks, eventually leading to a situation where some of the SMs are idle for a portion of the overall computation time, as in the multi-GPU case.

To achieve a high performance, we balance the workload within the GPU using a centralized queue in a similar fashion as in [1]. All the rays to be processed are kept on a single queue implemented as an array in global memory. The head of

the queue is also kept in the global memory, and it is visible to every thread in the grid. Instead of assigning the computations of a ray to a single thread, we make long-running warps retrieve batches of rays (tasks) from the global queue. This intra-GPU tasks are of finer granularity than the tasks used for multi-GPU load balancing. In our implementation we make each intra-GPU task to hold 64 rays. In our work, long-running warps are those that only finish their execution when all the rays in the global queue have been processed. This can be appreciated when the warps process is considered: when they are first launched, the first thread takes the next task by updating the head of the queue through an atomic fetch-and-add operation (i.e. *atomicAdd* in CUDA). It needs to be atomic in order to avoid race conditions between threads in different blocks). Then, each of the threads, within the warp, processes one of the rays in the task. When a warp completes this batch, it takes another task, or finish execution in case the queue is empty. The processing of each ray, first, involves traversing the scene BVH of triangles for determining primitive hits. The costs predicted for balancing at the multi-GPU level do not have a direct effect within the GPU workload balance, and are not used within the GPU for task scheduling. We observed a performance increse of around 40% for the trace kernels by using the described approach.

Chapter 4

Evaluation and Results

In our experiments we recorded the rendering time, the idle time of processing units, and the scheduling overhead for a number of benchmark scenes [1]. For each test scene experiment, we compute the average values of three repetitions for cameras located at 4 different pre-defined positions. Specifically, the cameras are positioned at the maximum values in the Y direction of the scene axis aligned bounding box and pointed toward the center of the box. Our test platform is an AMAX machine running a Xeon E5520 processor, 8GB of RAM memory, and 3 NVIDIA Tesla C1060 GPUs. The Tesla C1060 includes 240 CUDA cores and 4GB of device memory. Each workload balance policy ran under the same conditions for each test scene. In our evaluation we rendered 1024x1024 pixel images with one primary ray per pixel using

[1] The used benchmark scenes can be found in:
http://www.sci.utah.edu/ wald/animrep/
http://www.oyonale.com/modeles.php
http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/
http://code.google.com/p/efficient-sparse-voxel-octrees/

		Sampled rays (%)			
		12.5	25	50	100
Prediction Error (%)	Coherent Ray	3.2	2.5	0.7	0
	Diffuse Ray	4.1	2.7	1.2	0
Overhead Time (%)	Coherent Ray	5.0	12.5	22.5	42.5
	Diffuse Ray	6.2	15.2	23.5	43.5

Table 4.1: Prediction errors and overhead time for varying ray sampling rates for coherent (i.e. primary rays, sharp shadow rays and reflection rays) and diffuse rays by our approach. Results are averaged over our nine benchmark test scenes (see Figures 3.2, 4.3, and 4.4 for number of triangles). The overhead time is shown as the percentage of the full traversal including triangle intersection tests.

Whitted ray tracing and the ambient occlusion algorithms. The experiments for coherent rays were performed for a Whitted shader, with two bounces of reflection rays and shadow ray local contribution. For diffuse rays we compute the attenuation values using 32 samples per primary ray hit. In addition, we use three different task sizes: fine-grained (16384), medium-grained (4096), and coarse-grained (512). For cost-estimation 12.5% percent of rays was empirically determined to provide a satisfactory trade-off between estimation overhead and accuracy (see Table 4.1).

Figure 4.1 shows the actual rendering efficiencies of the cost-based version in terms of *million of rays processed per second* (MRays/sec). Similar tendency in rendering time is observed across all the test scenes. Both of the static strategy versions completely outperformed the dynamic centralized queue in all the cases. The best performance is achieved in all the cases by the static version with cost-based initialization. However, the performance of the centralized queue is improved as task size is increased.

The Hair Ball test scene is a challenging benchmark with a high geometric primitive density, and our test setup yields a near-optimal task assignation even for a simple equal-share balancing strategy. Therefore, the effect of cost-based initialization on the rendering efficiency is almost unnoticeable for the Hair Ball test scene, as illustrated in Figure 4.1. On the other hand, the Fairy Forest test scene is the best case among all the test scenes to demonstrate the improvement of the rendering efficiency by our cost-based initialization, since it has very sparse geometry and primitives of a high variability in size.

As illustrated in Figure 4.1, no significant difference in the centralized queue is observed between the cost-based version and the regular version. An interesting observation, however, is that even in the cases where none or a small performance improvement is observed, the effect of cost-based initialization is almost unnoticeable since the reported rendering time also includes the estimation and initialization overheads.

The general low performance of the centralized queue is due to the high overhead incurred by a kernel launch for each task to be processed. This overhead increases, in proportion to the number of tasks. Additionally, fine-grained tasks in a centralized queue will lead to underutilization inside the GPUs. Tasks within GPUs are balanced using persistent threads. When task sizes are too fine-grained, all the tasks are quickly fetched by a reduced number of warps. This means that only a reduced number of threads is going to be ready for execution at any moment, in turn, which will not efficiently utilize the CUDA architecture since the latency of memory operations cannot be hidden. In this regard, the centralized queue is thus only feasible

for coarse-grained tasks systems, no matter how it is initialized.

Figure 4.2 shows the balance efficiency achieved by the regular and cost-based versions of the static and dynamic policies. In this work, we define the *balance efficiency* as the ratio between the busy time of all the GPUs, and the rendering time multiplied by the number of GPUs. A balance efficiency of 100% means that all GPUs finish processing their last task at the same time. Figure 4.2 shows that the centralized queues achieve better balance efficiencies than the static assignment in all cases. In general, the resultant balance efficiencies by the centralized queue are negligibly sensitive to the size of the task, since the rendering time is dominated by the incurred overhead due to PCI-express bus latency.

Our implementation of the centralized queue can potentially benefit from an approach that avoids interrupting the kernel in order to retrieve tasks from the host queue. This can be achieved through asynchronous memory copies and the host mapped memory, as described in [12]. However, the performance of the cost-based initialized version is competitive with the theoretical performance of an optimized implementation of the centralized queue, as observed by the balance efficiency it achieves.

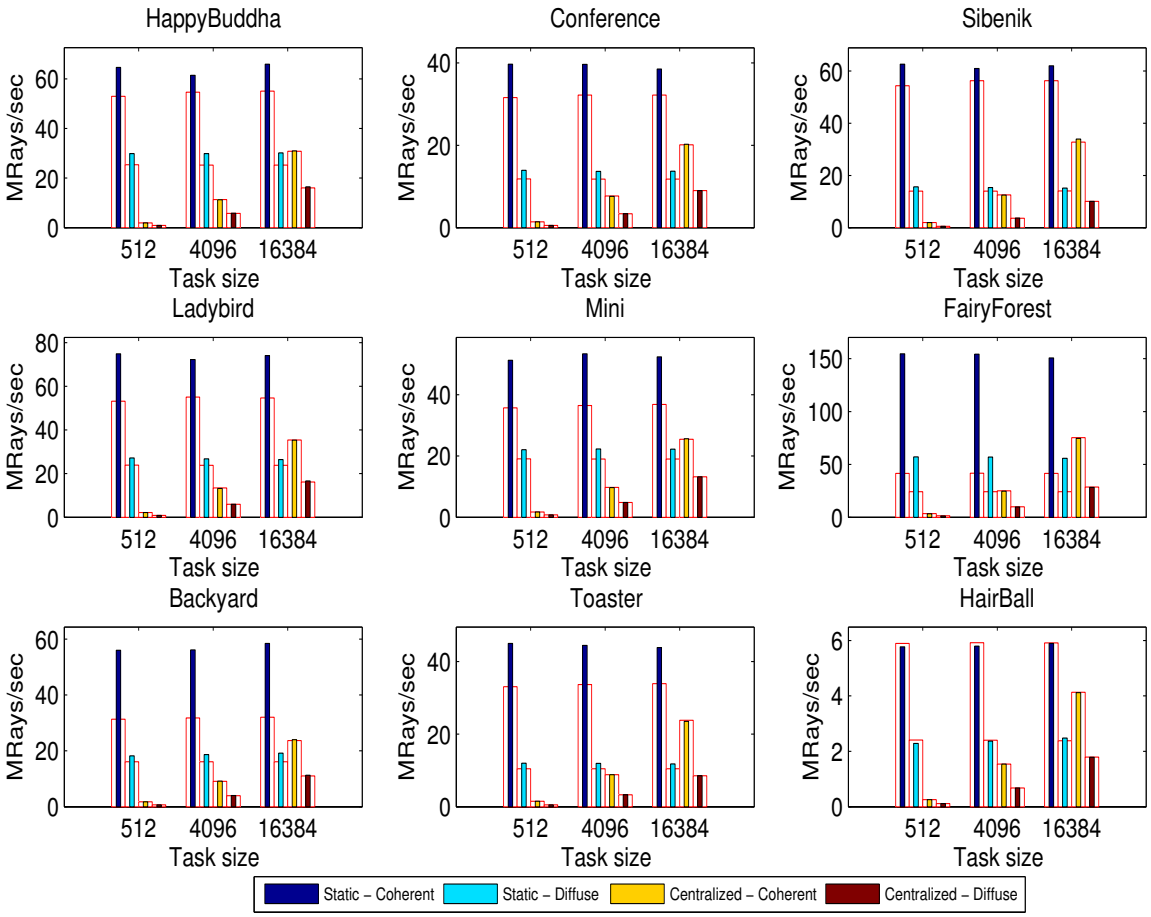

Figure 4.1: Rendering efficiency comparisons between a static equally distributed scheme and a centralized queue scheme over varied task sizes, with and without cost-based initialization. The color bars show the rendering efficiency of the cost-based initialization version (in MRays/sec). The background red-edged boxes show the rendering efficiencies of the corresponding regular versions. The rendering time includes the cost estimation overhead as well as the balancer initialization overhead.

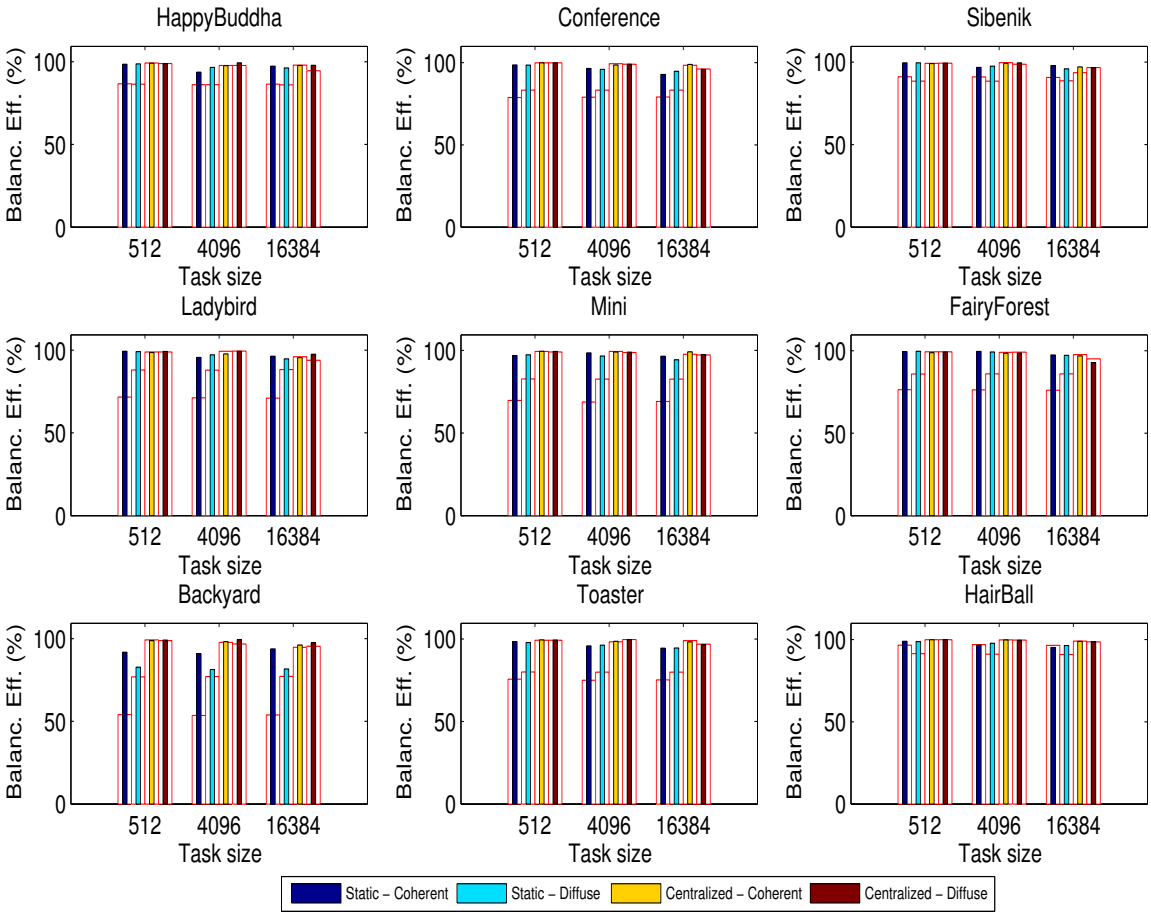

Figure 4.2: Comparison in terms of balancing efficiency between a static equally distributed scheme and a centralized queue scheme over varied task sizes, with and without cost-based initialization. Balancing efficiency is defined here as the ratio between GPU busy time, and rendering time multiplied by the number of GPUs. The color bars show balancing efficiencies, as a percentage, for the cost-based initialized versions. The background red-edged bars show the balancing efficiencies of the corresponding regular versions.

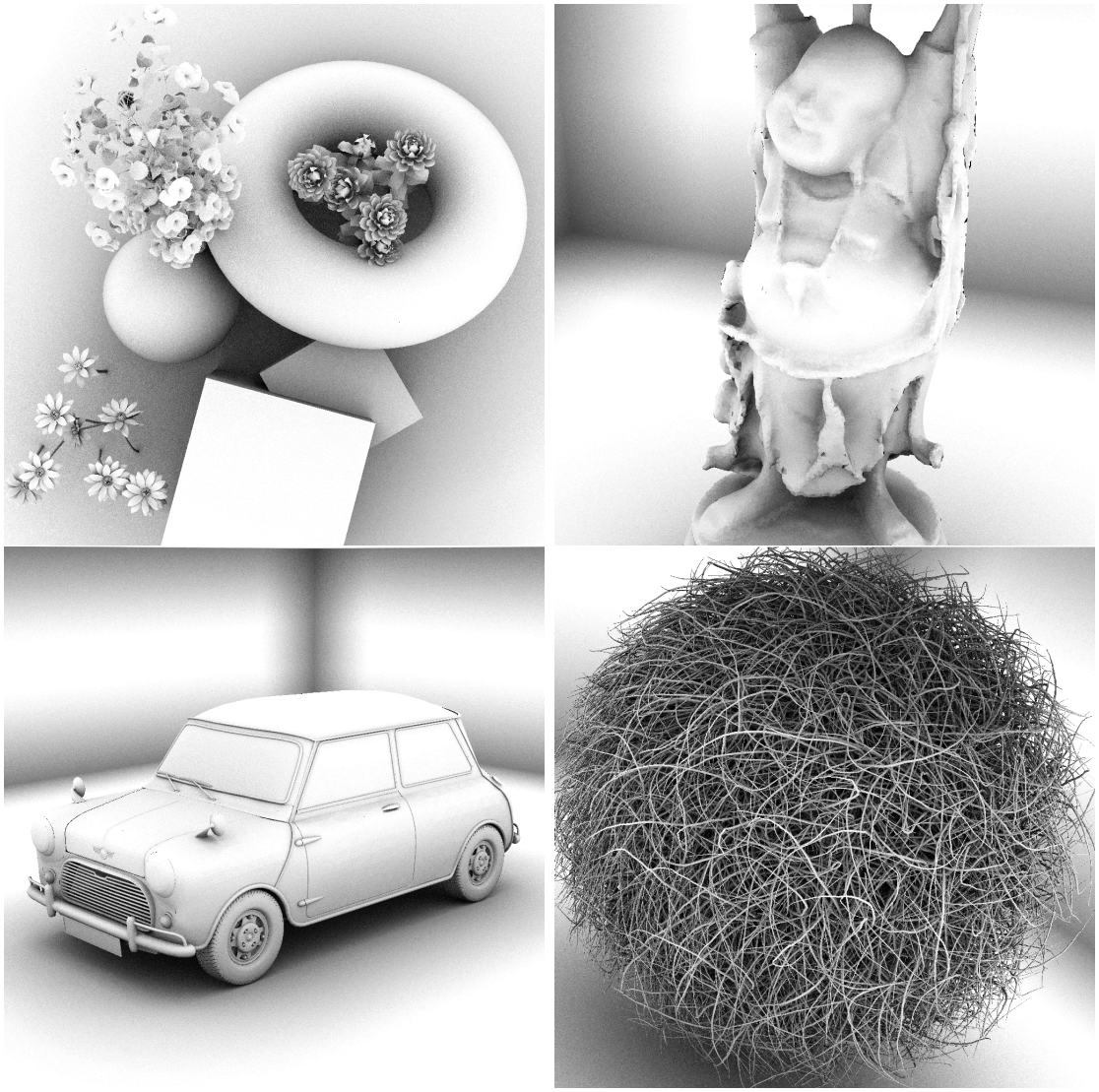

Figure 4.3: Ambient occlusion renderings of (top left to bottom right): the Backyard benchmark/test scene (213802 triangles), the Happy Buddha benchmark/test scene (32336 triangles), the Mini benchmark/test scene (234443 triangles), and the Hairball benchmark/test scene (2880000 triangles).

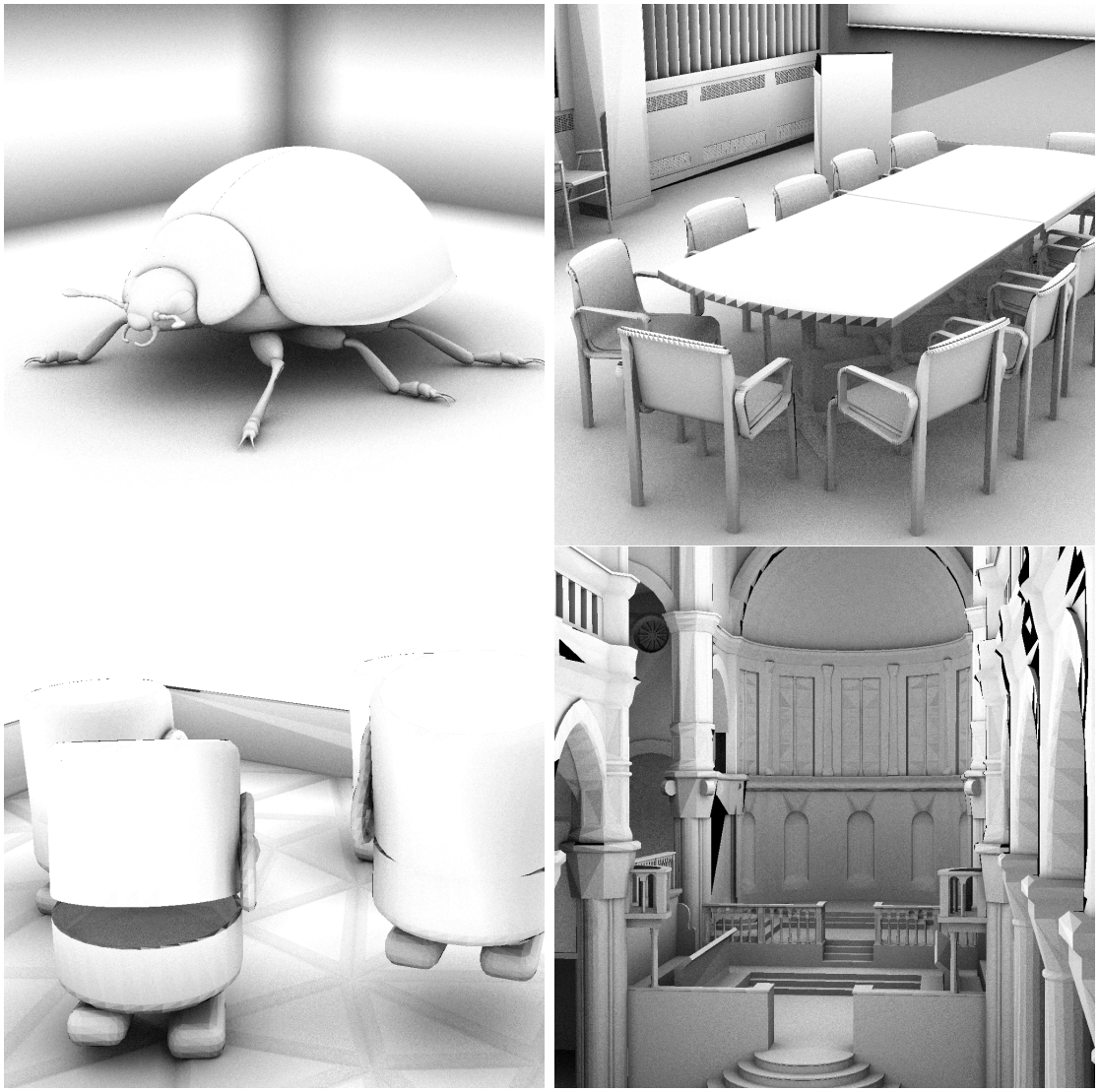

Figure 4.4: Ambient occlusion renderings of (top left to bottom right): the Ladybird benchmark/test scene (47004 triangles), the Conference benchmark/test scene (282755 triangles), the Toaster benchmark/test scene (11141 triangles), and the Sibenik benchmark/test scene (80131 triangles).

Chapter 5

Conclusion

5.1 Discussion and Conclusion

In this work, we propose a cost-based workload balancing scheme for ray tracing on a heterogeneous platform, including extensive comparisons between regular and cost-based versions of commonly used workload balancing strategies. Our approach was evaluated on a machine with one CPU and three GPUs. A quantitative performance comparison between the regular and cost-based balancing strategies was conducted for a number of benchmark/test scenes and sets of rays exhibiting different levels of variability.

Our experiment results demonstrate that our introduced cost-based workload balancing scheme is an attractive and promising option to enhance workload balancing

for GPUs-accelerated ray tracing. We show that the balancing efficiency for ray tracing can be measurably improved using cost-based initialization for the distributed static strategy, and the overhead associated with cost prediction is negligible to enable significant speed-up for heavy workloads. As described in our results (Section 4), our proposed approach provides performance gains for tasks of varying degree of granularity, and it is well suitable to be used with any type of rays in terms of their degree of variability.

The proposed cost-based approach is attractive and useful for multi-GPUs systems, since it enables *a static assignation of tasks as competitive as a centralized queue, dynamic balance strategy*. That is because a static strategy with a sound initialization can have a similar balancing efficiency to a centralized queue, but carries much more less communicational overhead. In this work, we tested the proposed cost-based approach on a shared memory multi-GPU platform, in which the bottleneck for data transfer is the PCI bus bandwidth. The advantages of the proposed cost-based initialization would be even more noticeable for clusters of GPUs as network latencies are longer.

5.2 Limitations and Future Work

A more efficient implementation of the centralized queue is possible with more complex synchronization mechanisms than the one used in our implementation. A reduction on the communicational overhead for such strategies will also translate into benefits from cost based initialization. In an ideal case (perfect cost prediction, and

no communicational overhead), the cost-based static balancing and the centralized queue lead to equivalent schedules. However, in reality, the cost prediction has a small estimation error, and a low-overhead implementation of the centralized queue can make up for that. As future work, low-overhead synchronization mechanisms required for an efficient implementation of the centralized queue can be investigated.

In this work we have evaluated the cost-based approach on a heterogeneous workstation featuring a single CPU, and three multiple discrete GPUs. The cost-based approach can be extended to larger scale systems, featuring a large number of CPU nodes, and GPUs. However, different factors, such as network latencies, should be considered in an efficient implementation of our approach for distributed systems. Also, our evaluation considered only processing units of similar computing capabilities. An interesting venue for future work, would be the evaluation of the effect of cost-based load balancing when processing units of varying computing capabilities are available.

A thorough scalability test involving a larger number of GPUs was not performed in this work. However, the results we have obtained allow us to reasonably extrapolate the behavior when more GPUs are involved. Currently, computational nodes involving more than 8 GPUs are not common. For multi-GPU platforms featuring a number of processing units within that order, we expect to observe a similar performance for the cost-based static share, and a slight performance improvement in the centralized queue as the number of GPUs is increased.

Additionally, the current cost-based approach only affects, in a direct way, workload balancing at the multi-GPU level. Task ordering at a finer granularity for load

balancing within GPUs might provide further performance improvement. Though in the current work we discard the cost-based workload balancing at the intra-GPU level, as future work we will probe the feasibility of efficient mapping of this scheme to GPU architectures. Fast ray sorting in coherent packets, such as the one described in [19], might result useful for this purpose.

Besides ray tracing, many other computations exhibiting irregular workload might also benefit from a similar approach to the one described in this paper. Particularly, we believe that the approach can be straight-forwadly extended to other problems that involve space/object partitioning data structures, in general, such as collision detection, and geometrical queries (e.g. k-nearest points).

Bibliography

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[2] J. Amanatides. Ray tracing with cones. In *Proceedings of Graphics Interface '84*, pages 97–98, May 1984.

[3] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10. North-Holland, Aug. 1987.

[4] B. Aronov, H. Brönnimann, A. Y. Chang, and Y.-J. Chiang. Cost prediction for ray shooting. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, SCG '02, pages 293–302, New York, NY, USA, 2002. ACM.

[5] J. Arvo. Backward ray tracing. In *SIGGRAPH '86 Developments in Ray Tracing course notes*. Aug. 1986. also appeared in SIGGRAPH '89 Radiosity course notes.

[6] J. Arvo and D. Kirk. *A Survey of Ray Tracing Acceleration Techniques*, pages 201–262. Academic Press, 1989.

[7] E. Braaten and G. Weller. An improved low-discrepancy sequence for multidimensional quasi-monte carlo integration. *Journal of Computational Physics*, 33(2):249–258, 1979.

[8] J. Breitbart and C. Fohry. OpenCL - an effective programming model for data parallel computations at the cell broadband engine. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[9] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core data management for path tracing on hybrid resources. *Comput. Graph. Forum*, 28(2):385–396, 2009.

[10] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[11] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[12] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *IPDPS*, pages 1–12. IEEE, 2010.

[13] R. L. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21:95–102, August 1987.

[14] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18:137–145, January 1984.

[15] D. E. DeMarle, C. P. Gribble, and S. G. Parker. Memory-savvy distributed interactive ray tracing. In B. Raffin, D. Bartz, and H.-W. Shen, editors, *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, Grenoble, France, 2004. Eurographics Association.

[16] D. E. DeMarle, S. G. Parker, M. Hartner, C. P. Gribble, and C. D. Hansen. Distributed interactive ray tracing for large volume visualization. In A. H. J. Koning, R. Machiraju, and C. T. Silva, editors, *IEEE Symposium on Parallel and Large-data Visualization and Graphics*, pages 87–94. IEEE, 2003.

[17] P. Dutre, P. Bekaert, and K. Bala. *Advanced Global Illumination*. AK Peters Limited, 2003.

[18] J. D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley Systems Programming Series, 1997.

[19] K. Garanzha and C. T. Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Comput. Graph. Forum*, 29(2):289–298, 2010.

[20] K. Garanzha, J. Pantaleoni, and D. K. McAllister. Simpler and faster HLBVH with work queues. In C. Dachsbacher, W. Mark, and J. Pantaleoni, editors, *High Performance Graphics*, pages 59–64. Eurographics Association, 2011.

[21] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[22] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[23] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society.

[24] A. Heirich and J. Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1-2):57–68, 1998.

[25] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[26] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[27] H. W. Jensen. Global illumination using photon maps. In *7th Eurographics Workshop on Rendering*, pages 22–31, 1996.

[28] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Natick, MA, 2001.

[29] H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 511–518. ACM Press / ACM SIGGRAPH, 2001.

[30] J. T. Kajiya. The rendering equation. In *Proceedings of SIGGRAPH '86*, pages 143–150, 1986.

[31] J. T. Kajiya and B. P. V. Herzen. Ray tracing volume densities. In *Computer Graphics (ACM SIGGRAPH '84 Proceedings)*, volume 18, pages 165–174, July 1984.

[32] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20:269–278, August 1986.

[33] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010.

[34] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for A parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, Oct. 1988.

[35] C. Lauterbach, A. Chandak, and D. Manocha. Interactive sound rendering in complex and dynamic scenes using frustum tracing. *IEEE Trans. Vis. Comput. Graph*, 13(6):1672–1679, 2007.

[36] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[37] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–61, July 1990.

[38] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, March 2008.

[39] NVIDIA. CUDA occupancy calculator http://developer.nvidia.com/cuda-toolkit-32-downloads, Apr. 2011.

[40] NVIDIA. Cuda programming guide 4.0, Mar. 2012.

[41] J. Pantaleoni and D. Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In J. Hensley, P. Slusallek, D. K. McAllister, and C. P. Gribble, editors, *High Performance Graphics*, pages 87–95. ACM, 2010.

[42] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 119–126, New York, NY, USA, 1999. ACM.

[43] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM.

[44] M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005.

[45] M. Pharr and P. Hanrahan. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In *SIGGRAPH*, pages 75–84, 2000.

[46] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless KD-tree traversal for high performance GPU ray tracing. *Comput. Graph. Forum*, 26(3):415–424, 2007.

[47] T. Priol and K. Bouatouch. Static load balancing for A parallel ray tracing on a MIMD hypercube. *The Visual Computer*, 5(1/2):109–119, Mar. 1989.

[48] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21:703–712, July 2002.

[49] E. Reinhard, A. J. F. Kok, and A. Chalmers. Cost distribution prediction for parallel ray tracing. In *Second Eurographics Workshop on Parallel Graphics and Visualisation*, Rennes, France, Sept. 1998.

[50] I. Sadeghi, B. Chen, and H. W. Jensen. Coherent path tracing. *J. Graphics, GPU, & Game Tools*, 14(2):33–43, 2009.

[51] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[52] P. Shirley. *Realistic Ray Tracing*. A K Peters, 2000.

[53] Top500. Top 500 supercomputers site, http://www.top500.org/system/10587, Apr. 2012.

[54] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In M. Doggett, S. Laine, and W. Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[55] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[56] E. Veach and L. J. Guibas. Metropolis light transport. In *Computer Graphics (ACM SIGGRAPH '97 Proceedings)*, volume 31, pages 65–76, 1997.

[57] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 15–24, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[58] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Comput. Graph. Forum*, 20(3), 2001.

[59] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao. An efficient GPU-based approach for interactive global illumination. *ACM Trans. Graph*, 28(3), 2009.

[60] A. Watt. *3D Computer Graphics*. Addison-Wesley, 1993.

[61] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, July 1994.

[62] T. Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, New York, NY, USA, 1979. ACM.

[63] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.

[64] S. Zhukov, A. Inoes, and G. Kronin. An ambient light illumination model. In G. Drettakis and N. Max, editors, *Rendering Techniques '98*, Eurographics, pages 45–56. Springer-Verlag Wien New York, 1998.