Title

Out-of-core Data Management for Path Tracing on Hybrid Resources

Permalink

https://escholarship.org/uc/item/1750k9st

Authors

Budge, Brian C.
Bernardin, Tony
Stuart, Jeff A.
et al.

Publication Date

2009

DOI

10.1111/j.1467-8659.2009.01378.x

Peer reviewed

Out-of-core Data Management for Path Tracing on Hybrid Resources

Brian Budge   Tony Bernardin   Jeff A. Stuart   Shubhabrata Sengupta   Kenneth I. Joy   John D. Owens

Institute for Data Analysis and Visualization,
Department of Computer Science, and
Department of Electrical and Computer Engineering
University of California, Davis

Abstract

We present a software system that enables path-traced rendering of complex scenes. The system consists of two primary components: an application layer that implements the basic rendering algorithm, and an out-of-core scheduling and data-management layer designed to assist the application layer in exploiting hybrid computational resources (e.g., CPUs and GPUs) simultaneously. We describe the basic system architecture, discuss design decisions of the system's data-management layer, and outline an efficient implementation of a path tracer application, where GPUs perform functions such as ray tracing, shadow tracing, importance-driven light sampling, and surface shading. The use of GPUs speeds up the runtime of these components by factors ranging from two to twenty, resulting in a substantial overall increase in rendering speed. The path tracer scales well with respect to CPUs, GPUs and memory per node as well as scaling with the number of nodes. The result is a system that can render large complex scenes with strong performance and scalability.

Categories and Subject Descriptors (according to ACM CCS):   Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1   Introduction

Enhancements in performance and storage in modern computers have enabled the realistic display of ever-larger and more complex datasets. Graphics applications constantly stress these bounds with scenes exceeding the total memory of even the largest systems, leading to *out-of-core* data access. When combined with the workload of global illumination, complex scenes can take days or weeks to render.

Advances in the design of computational systems have produced systems with a variety of general purpose computational engines. Machines can be designed with CPUs and GPUs, allowing substantial computational power even on the desktop. The challenge is to fully utilize the power of these hybrid computational resources for solving problems.

We describe a system architecture that addresses these large-scale complex out-of-core problems on machines that contain both CPUs and GPUs. We build our renderer on top of an out-of-core data-management layer that controls data access and schedules tasks to exploit the hybrid resources available. This data management layer is designed to exploit the coherent operations inherent in rendering. By breaking data and algorithmic elements into modular components, we can queue tasks until we reach a critical mass of work, only then fetching the data necessary to execute the tasks. In this way, our system can handle large-scale geometry, but unlike other methods, it can also work on complex materials with long shaders and large textures in the presence of GPUs. By utilizing our data-management layer, we have developed a path tracing application that scales to systems with many processors and many nodes, and that can use multiple types of computational resources, including scalar processors such as CPUs and data-parallel processors such as GPUs.

In this paper, we discuss key design and implementation details of the system. We present an overview of the system explaining the function of components and how they are con-
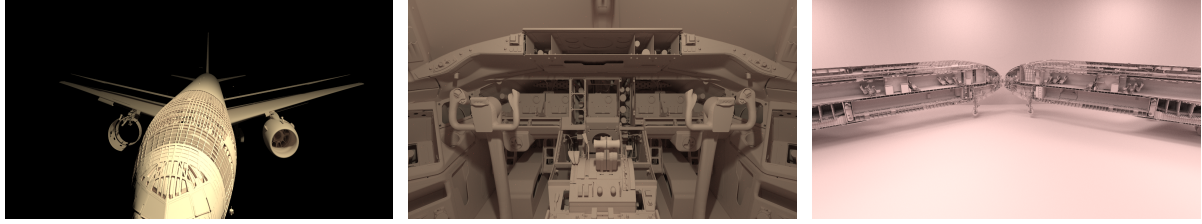
Figure 1: *The 337 million triangle Boeing 777 model. The scene, preprocessed for ray tracing, is several times larger than our machine's system memory. It is extremely expensive to path trace due to out-of-core data access, but our technique helps to alleviate this cost. From left to right: Viewing the Boeing 777 from nose to tail, within the cockpit, and split down the center. The render times ranged from 27 minutes for the leftmost image, to about 17 hours for the rightmost image.*

nected. We also give details of the most crucial components, and discuss the path tracer and how it was designed to utilize the data-management layer, allowing for fast, scalable, out-of-core ray and shadow tracing, shading and lighting.

2 Related Work

Teller et al. described an out-of-core radiosity system [TFFH94] which calculated intermediate results in core, storing results for later, memory-coherent use. Wald et al. [WDS04] proposed an out-of-core method that utilizes Linux low-level memory and I/O functions. The method tries to pre-fetch all memory pages. When page faults occur, the method can decide to replace the ray by a shading proxy to approximate the ray's color value thus achieving fast render times, at the cost of approximation. Demarle et al. [DGP04] designed a distributed virtual memory system for rendering where page faults resulted in a request being sent to gather the data from another node's memory. Their work did not handle scenes that do not fit into the memory of the cluster. Fradin et al. [FMH05] allow out-of-core photon mapping in large buildings, but are limited to in-core geometry at render time. In contrast, our system allows the estimation of the full rendering equation [Kaj86] with out-of-core scenes.

Lefer [Lef93] ray traced large scenes by distributing the scene amongst the system nodes. Task parallelism was combined with data parallelism to achieve good scaling. Reinhard et al. [RCJ99a, RCJ99b] presented a hybrid approach for rendering large scenes that combined demand-driven parallelism with data parallelism. They send bundles of coherent rays to processors to request work and push incoherent work from processor to processor to find the required data. Ward [War94] bundled primary and shadow rays so both would be run as demanded tasks and executed secondary rays in a data-driven way. Kato and Saito [KS02] distributed geometry across all nodes in the cluster semi-randomly, tracing each ray through all nodes. They require all data to fit in the memory of the cluster. In our path tracing application, rays and other temporary data all remain on the node where they originate, and we are able to handle scenes larger than the memory of our combined render nodes.

To eliminate the need for going out-of-core at all, Christensen et al. [CLF*03] cache tessellations of higher order surfaces, which helps them avoid re-tessellation and allows them to stay in-core. They extended this technique to allow out-of-core photon maps [CB04]. Unfortunately these approaches cannot be used for large polygonal models. Yoon et al. explored ray tracing of level-of-detail models in a method called R-LOD [YLM06]. Techniques that might reduce data size include using smaller ray acceleration structures, such as Bounding Interval Hierarchies [WK06], and reduction of mesh sizes via lossless compression [LI06].

Pharr et al. [PKGH97] reorder and cache rays to ensure that rays are traced only against objects in memory. The scheme pre-processes data so bundles of geometry with spatial locality remain close together in memory. Navratil et al. [NFLM07] proposed a finer-grained ray scheduling algorithm than Pharr et al., improving cache utilization with fewer rays in flight. Gribble and Ramani [GR08] stream-filter ray tracing data to coherently process it on a SIMD machine. We use a similar approach for our path tracing, but our filtering occurs at a more granular level to aid out-of-core access. Gribble et al. and Navratil et al. concentrate on in-core data.

We extend Pharr et al.'s approach of caching rays by saving *all* algorithmic computation until a time when sufficient coherence is achieved. This allows our system to not only handle large geometry, but also large textures and complex materials on GPUs. The absence of virtual memory for GPUs makes this a non-trivial extension of Pharr et al.'s approach.

GPGPU We leverage the general-purpose capabilities of the modern GPU to accelerate complex, non-real-time rendering tasks. Related work includes ray tracing, photon mapping, radiosity, and hybrid rendering [OLG*07, WGER05]. The hybrid CPU-GPU approach advocated in this paper is also characteristic of systems like Pixar's Lpics [PVL*05], which achieves near-real-time performance on complex scenes by evaluating light shaders on GPUs, and Lightspeed [RKKS*07], which evaluates complex GPU shaders for fast relighting.

Scheduling Arguably the most performance-critical and complex component of our system is the scheduling of tasks. Path tracing is a Monte Carlo algorithm, which disqualifies scheduling techniques that require predictable tasks [Ant06, CDM97, Nos98], however, recent work has investigated scheduling less predictable tasks. Boyer and Hura schedule dependent tasks by using a random ordering technique combined with re-evaluation of estimated task execution times [BH04]. To keep the load balanced, they allow tasks to migrate to different processors if dependency criteria are met. The need to migrate associated data with the task makes this unsuitable for a high-throughput, out-of-core system. Mehta et al. [MSS*06] have also investigated unpredictable scheduling in heterogeneous computing environments, but their work requires independent tasks. A previous paper describes a visualization of our scheduling process, which allowed debugging and fine-tuning of our scheduler [BBH08]. In Section 3.3 we describe our hybrid data-dependent scheduling algorithm, which tries to maximize throughput at the expense of individual task latency and temporal fairness.

3 System Architecture

The hybrid data-management system (Figure 2) forms the backbone of out-of-core applications, and was designed with a specific subset of problems in mind. It understands algorithms that can map to three key concepts: *kernels* that encapsulate the processing logic to complete a task (e.g. ray intersection); *static data* that provides unchanging persistent application data (e.g. scene geometry); and *transient data* that describes the actively manipulated workload (e.g. rays). Components in the base system are agnostic to the specific data or kernel functions of an application.

The execution of an application in our system is driven by two concurrent modes: task management, which is predominately handled via *queueing* threads, and task execution, which is performed by execution threads. The queueing threads are triggered to run by the availability of jobs in the job buffer. Each job is destined for execution by a specific kernel, and is further divided amongst the queues of that kernel by the kernel's queueing filter as described in Section 3.2.

Task execution is triggered when an execution thread becomes idle. That thread starts the scheduling process which determines an appropriate task for the requesting processor as detailed in Section 3.3. The static and transient data is moved to the required layer in the memory hierarchy and the kernel executed. The output transient data is finally routed as a new job to the job buffer and the cycle repeats.

The goals of the system are to manage memory allocation and data migration, leverage coherence, and use processing resources in the most efficient manner, all while maintaining a generic interface through which applications can be imple-

mented. Without these features, performance and extensibility of out-of-core applications are lost. The remainder of this section will explain how our components were designed to match these goals.
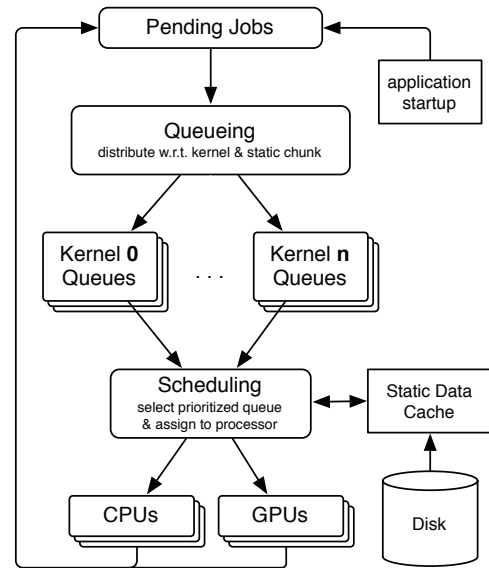
Figure 2: *Transient data flow: Pending jobs are redirected via the queueing filter to the appropriate kernel queues. As processors become idle, the scheduler is asked to select the most appropriate queue to run on the resource. The static and transient data are made available to the processing unit, the kernel is executed, and the kernel's output is filtered by destination kernel and placed back in the job buffer for routing.*

3.1 Memory Allocation and Migration

NVIDIA's CUDA [NVI06], our GPU computing platform of choice, cannot directly use data resident in main memory and it must be explicitly loaded into video memory before processing. We designed the data-management layer to hide such details from kernel programmers: memory is allocated where it is needed and migrated according to kernel needs. Upon initialization, the system partitions the video RAM into several regions: one region becomes part of a static-data cache, the remainder is reserved for input and output of transient data, as well as scratch space for filtering and compaction of output jobs. Allocations are organized in pools in system RAM, which are configured at start-up according to the needs of the application's kernels. Pools allow for fast, thread-safe allocations with low fragmentation.

As a task is assigned to a processor, the data-management layer gathers information about the task requirements, in particular the need for static data and the type of the processor. With this information, static data is potentially read

from disk and migrated to system RAM in the case of a CPU or video RAM for a GPU. The transient data region of the GPU is repartitioned to accommodate the task and input data is transferred for execution, and output data is copied back after execution. CPU kernels directly use the pool buffers.

We also designed a hybrid static-data cache that spans the main and GPU RAM. The cache is used to keep data close to the preferred execution unit, minimizing data transfers and improving scalability. For efficiency reasons, we avoid duplication of static data chunks in the cache (see Section 3.3). This is especially important since CUDA does not allow direct transfer of data between GPUs.

3.2 Leveraging Coherence

A single kernel can potentially access many different static data chunks. For example, the geometry of a scene could be partitioned into several chunks, such that a ray intersector processes them depending on rays' starting locations and their paths through the scene. We use a transient data queueing scheme to optimize for this behavior. In it, each kernel exposes several queues, and each queue is associated with a specific static chunk. When a kernel receives transient work from the queueing thread, the work is divided based on the static data necessary for operating on that work, and it is queued for later processing. This *queueing-filter* is invoked by the queueing thread, and the functionality is integrated into the data management layer through a generic interface implemented by each kernel. We present some examples of queueing filters in Section 4.

The queueing mechanism provides two core features of the data management layer: Buffering related work increases the coherency of processing, and organizing work not only into kernel-specific tasks but also into static data-specific ones provides a basis for data-driven execution. When paired with an appropriate scheduler, both are critical to enabling efficient out-of-core applications.

3.3 Utilizing Processing Resources

We manage execution through a central scheduling component. Its purpose is to assign tasks to processors, moving data in the memory hierarchy as needed. The scheduler aids efficient out-of-core access because of the following design tenets:

- Lazy access to storage: We utilize static data chunks in the fast layers of the memory hierarchy as much as possible. We delay access to storage memory until it is inevitable.

- Maximize coherence: We strive to process tasks that expose the most coherency or those that may aid the coherency of other tasks.

- Maximize processor utilization: We prioritize tasks based on a processor's proficiency for performing a task and its access to the required data.

- Maximize cache resources: Execution directly influences the locality of static data chunks in the memory hierarchy. We avoid chunk replication in the hierarchy when scheduling tasks and strongly favor tasks for which chunks are present.

The efficient, automatic assignment of tasks to hybrid resources without intimate knowledge of the algorithm domain is a key difference between a traditional out-of-core approaches and our hybrid out-of-core scheme.

General Scheduler Design. Initially our scheduling component was designed with a task-queue for each processor. In a typical producer/consumer approach a separate scheduler thread filled task-queues with appropriate tasks and execution threads retrieved them as they became ready. Task-queues were three tasks deep. This design proved inefficient because the execution of a single task could drastically alter the workload, outdating the tasks already committed to the queues. We now use a serial "on-demand" scheme. As a processor becomes available it (1) locks the scheduler; (2) scans all the kernel-queues for tasks and (3) prioritizes them; (4) selects the highest priority task allowed; (5) marks selected kernel-queue as processed; and (6) releases the scheduling lock.

General Prioritization Design. Prioritizing the workload is the most involved step of the scheduling process. We considered a cost-driven model where memory transfers incurred a cost and workload size attributed a benefit. This model was not effective in promoting cooperation between the hybrid processing resources, because each greedily evaluated independent costs. To allow finer tuning of the prioritization we chose an empirically-driven classification scheme. We define several prioritized categories for binning tasks, and further differentiate by workload. To compare tasks to one another, we evaluate and tag them based on the availability of required static data in the memory hierarchy, the processor preference, and the size of the workload. More specifically we tag each task with

N No static data is required for this task.

G Data resides on scheduling GPU or any GPU when scheduling a CPU.

O Data resides on a GPU other than the scheduling GPU.

C Data resides in system RAM.

T Data is being transfered to system RAM.

P This task prefers to run on this kind of processor.

S Like P, but the workload is too small to run efficiently.

The categories, representing an ordered subset of tag combinations, are presented in order of priority in Table 1. Overlined tags denote that a criterion is *not* met. In general, the categories are ordered to favor preferred tasks ($P > S > \overline{P}$),

tasks that require no static data or having the data in the closest memory hierarchy ($N > G > C > T > \overline{GCT}$), and tasks that support cooperative processing. Tasks are binned in a filtering process, where a task begins testing against the highest-priority category and continues through categories until the criteria for a category are met.

Prioritization Adjustments. We hand-tuned the categorizations to satisfy our design tenets. The first two adjustments in the CPU case are PC and SC which are favored over static data independent PN. N-tasks can always be executed whereas C-tasks depend on static data availability, which can be lost when another thread schedules work. We trade-off coherency to avoid losing the cache availability. The S and $\overline{P}$ cases avoid stealing GPU work by filtering G-tasks to the lowest categories $\overline{PG}$ and an implicit final category including SG. This is why $\overline{G}$ shows up in $ST\overline{G}$, $\overline{PCG}$, $S\overline{G}$ and $\overline{PT}\overline{G}$. To improve coherency and delay out-of-core access, S-tasks which are not C fall below non-preferred tasks with resident data, so $S\overline{G}$ is after $\overline{PC}$.

The GPU scheduling case presents similar adjustments. Small tasks are relegated to the bottom of the prioritization to avoid costly transfers to GPU memory with small workload. The exception is $SG\overline{C}$ as the static data is only available in GPU memory and a more costly transfer would be needed to move it to main RAM for CPU access. The categories $SG\overline{C}$, $PC\overline{O}$, $PT\overline{O}$, P and PO are arranged to avoid duplicating static data chunks within the GPU memory layer of the static data cache, opting to fetch new data instead. We avoid executing $\overline{P}$ tasks, but if no other tasks are available we favor running such tasks that most benefit from the current cache: ($\overline{PGC}$ and $\overline{PG}$ over $\overline{PN}$).

Some constraints exist that cannot be handled through prioritization alone. Examples are tasks that have no implementation for the scheduling processor or tasks that need to be executed serially.

3.4 Algorithm Genericity

The system has no understanding of what an algorithm does. It reads configuration files at runtime that specify cache sizes, the number of CPU, GPU, and queueing threads, the location of dynamic libraries that support our kernel interface, and state machines that describe how to transfer data between kernels.

Genericity is maintained by keeping a very simple kernel interface, where each kernel must provide functions to perform initialization, data queueing, execution, and post-execution-filtering. All data passing through the interfaces are associated with a base data location (a pointer and whether it is GPU or CPU) and a size. Kernels are responsible for knowing how to deal with that data once received.
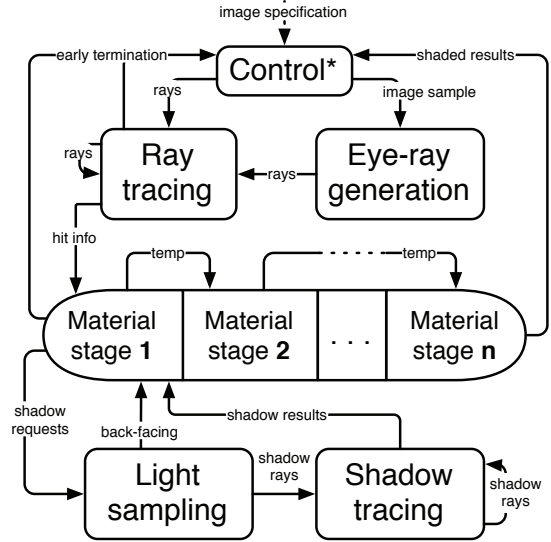
Figure 3: *Modular path tracing algorithm and transient data communication. The asterisk indicates that the Control shader can only run on CPUs. All other kernels are capable of running at least some of their computation on GPUs.*

4 Path Tracer Design

Using the system architecture described in the previous section, we implemented a path tracer that uses both CPUs and GPUs and scales with system resources. Path tracing [PH04, SM03] is a global illumination algorithm that begins by tracing samples from the eye. Eye rays are generated for each sample from the eye and then traced into the scene to determine the first hit location, and rays are reflected or refracted as appropriate. We also trace shadow rays to light samples among all of the light sources and shade at the traced rays' hit locations. This process continues recursively with the reflected ray in place of the eye ray. Path tracers tend to show fairly incoherent behavior with regard to memory access, and require substantial computation for image convergence. Contrast this with ray tracing and polygon projection, where large amounts of natural coherence can be more easily exploited.

In the stream processing model [RDK*98], data is expressed as streams—long arrays of data of the same datatype—and computation is expressed as kernels—programs that operate on streams. Our system is not required to use a stream processing model, but much of our efficiency can be attributed to this kind of design, and our system lends itself naturally to this computational pattern. We constructed our path tracer application from a collection of kernels that plug into our data management system. These kernels form a state diagram like the one in Figure 3 that shows the general flow of data.

cpu	PC	SC	PN	PT	P	SN	$ST\overline{G}$	$\overline{P}N$	$\overline{P}C\overline{G}$	$\overline{P}C$	$S\overline{G}$	$\overline{P}T\overline{G}$	$\overline{P}$	$\overline{P}G$
gpu	PN	PG	$SG\overline{C}$	$PC\overline{O}$	$PT\overline{O}$	P	PO	$\overline{P}G\overline{C}$	$\overline{P}G$	$\overline{P}N$	$\overline{P}C$	$\overline{P}T$	$\overline{P}$	S

Table 1: *Binned classification of tasks prioritized from left to right by the task properties and the type of scheduling processor.*

The *Control* kernel handles many tasks including making path tracing decisions (e.g. eliminating or continuing paths), communicating with outside components (e.g. master MPI process), and constraining the number of concurrent paths in flight. The kernel receives requests for image samples and sends compact sub-image information to the *EyeRayGeneration* kernel. The rays are intersected in the *RayTracing* kernel, potentially many times, as the ray traverses the scene. Rays can intersect many different materials. The material kernels run in several stages depending on static-data dependencies. The *RayTracing* kernel forwards hits to the first stage of the appropriate material kernel. The first stage determines if a ray is terminated (eliminating light source double counting), a shadow ray is generated (diffuse bounce) via stochastic light-source sampling in the *LightSampling* kernel, or the point can be directly shaded and the ray reflected or refracted (specular). Finally, shade values and new rays are looped to the *Control* kernel for further decision making.

In our path tracer, the *Control* kernel is the source and sink of all transient data, while all kernels (*Control* kernel included) *convert* transient data from one type to another. For example, the *RayTracing* kernel takes rays as input and outputs rays, early termination, or hit points, and the *LightSampling* kernel receives sample requests, and outputs shadow rays or occlusion (due to back-facing geometry). We strived to design simple kernels to perform straightforward tasks rather than complex kernels combining several types of work, with only the *Control* kernel carrying out multiple tasks; the *Control* kernel handles the generation of new eye-ray requests, path propagation and termination, and image generation. Due to light computation and heavy logic, the *Control* kernel is the only kernel in our implementation that runs only on the CPU.

The *RayTracing* kernel is a good example of a fairly general kernel with complex incoming and outgoing connectivity, and so it is ideal for illustrating data flow through our system. The *RayTracing* kernel can receive new incoming rays to trace from either the *EyeRayGeneration* kernel or from the *Control* kernel (due to reflections and refractions). As the *RayTracing* kernel receives incoming ray data, the data must pass through a queueing filter, which redirects this data into queues based on the memory chunk that contains the geometry and the origin of each ray. A large scene may have hundreds to thousands of these chunks and will have a corresponding number of input queues. When the *RayTracing* kernel is scheduled to operate on a particular queue, the system removes rays from the appropriate queue, ensures that the static data for this queue is in the scheduled processing resource's memory, and then executes the kernel.
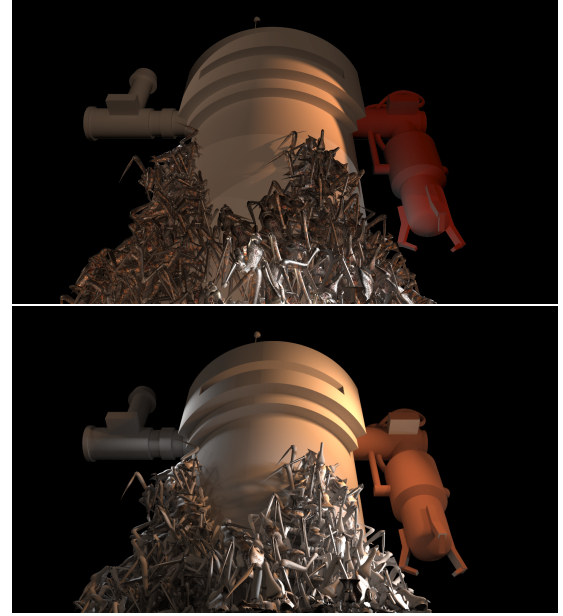
Figure 4: *The Starship Troopers dataset from Tippett Studio. Top: RenderMan. Bottom: Our path traced image. Most differences between the two images are because we use area light sources, while RenderMan is using spotlights. RenderMan tessellates to the same level as our geometry (~243 M triangles), and uses 64 stochastic hemisphere samples per shade point for single-bounce global illumination, requiring several days to render on 2 CPUs. We use 1024 paths per pixel with full path tracing, taking just over two hours using 1 CPU and 1 GPU for execution. We use 2 CPU threads for filtering data into queues, which requires less than 9% of the runtime.*

When the *RayTracing* kernel runs, it traces the rays in one of its input queues and outputs hit information, ray termination, or copies of the original rays (advanced to the next kd-tree cell). These data are all destined for different kernels, and they are grouped together by type in the post-schedule filtering process. Hit information is forwarded to the appropriate material kernel, early termination notification is sent to the *Control* kernel, and ray copies are filtered back into the *RayTracing* kernel's input queues.

The geometry chunks associated with the *RayTracing* kernel are each part of the higher level of a two-level kd-tree. The high level kd-tree is based on a medial split build heuristic. Each geometry chunk contains a kd-tree that has been highly

optimized for ray tracing. We use the typical node layout from the thesis of Wald [Wal04], adding slight enhancements for triangle list traversal from leaf nodes. Our triangle format is also the 48-byte triangle format from Wald's thesis; however, we also utilize the unused bytes to store texture coordinate and material indices. Because the median split kd-tree is highly inefficient for geometry culling, we try to keep the top-level tree shallow. This typically results in geometry chunks of ~200 MB. In the future, a surface area heuristic approach should be used at both levels. This would have two effects: Less disk reads would be required to traverse empty space, and chunks could be smaller (~10 MB) to allow fine-grained cache usage.

Some material kernels may require reads from multiple large textures, and due to memory restrictions, only a subset of the textures can be read in a single pass. In such cases, we split the kernel during preprocessing into several incremental kernels. Each stage will perform texture reads on some portion of the texture data, and will implement a queueing-filter that uses incoming texture coordinates to decide which chunks of texture (and corresponding queues) a transient data element belongs to. This process can be seen in Figure 3. We compute each subpart kernel and output the result into a queue of the next subpart for further computation. The intermediate values are used to complete future stages. The final stage of the material will generate a shaded point paired with a reflection/refraction ray direction and send shaded points back to the *Control* kernel for framebuffer accumulation and ray propagation.

5 Results

In this section we present experiments that characterize our system's ability to scale with the scene size in geometry and texture, the resources present, the amount of work to be computed, and the scene's global illumination complexity.

Our rendering nodes each have two dual-core AMD Opterons running at 2.4 GHz and 4 GB of main memory, two NVIDIA QuadroFX 5600s with 1.5 GB of video RAM, and four disk drives in a RAID-0 configuration. The measured bandwidth to our RAID is 220 MB/s. The master node consists of four dual-core AMD Opterons running at 2.4 GHz, with 16 GB of main memory, and an eight-disk RAID-0.

Our system can run on any single node, or on multiple nodes networked together via MPI (message passing interface). The system is running Gentoo Linux with kernel version 2.6.24. Except in the case of the resource-scaling experiment, all experiments were run on a single node. Additionally, we have designed the system so that filtering into queues occurs in separate threads. All experiments were run with two filtering threads, and the processing time of these threads is generally a few percent of the overall runtime for the Troopers scene, and 10 to 15% for the Boeing scenes.

Figure 4 shows our first test case. This scene is from the

Scene	Triangles	Light Sources	Materials	Texture	Total Size
Starship Troopers	4,8 M 19.6 M 66.8 M 134.6 M 242.6 M	13	Lambertian, Ashikhmin/ Shirley+ PRMan tweaks	108 MB 4.4 GB	393 MB 1.3 GB 4.2 GB 8.3 GB 15.1 GB 4.6 GB 5.6 GB 8.5 GB 12.6 GB 19.5 GB
Boeing Nose Boeing Cockpit	337.1 M	2	Lambertian	none	21 GB
Boeing Split	340 M	56			19 GB

**Table 2:** *The configurations used for our experiments.*

movie Starship Troopers from Tippett Studio. The geometry and scene data was provided in a RIB file as NURBS surface patches and RenderMan shaders utilizing a set of 35 textures (~50 MB total). We preprocessed the scene to generate tessellated versions that contained roughly 5, 19, 67, 134 and 243 million triangles. We also created two sets of texture from the original: The first set is stitched into a single texture of 108 MB (with fragmentation), and the second set is scaled by 8x8 or by 16x16 to give 4.4 GB of data in 35 distinct textures. We attempted to model the materials to be as close as possible to the original materials. The specular surfaces were modeled with an Ashikhmin-Shirley BRDF with parameters generated via code from the original RenderMan shader. The 13 provided spotlights were also converted into triangular area lights with normals pointing in the spotlight directions. These light differences and the small changes to the materials account for the majority of visual differences in the images.

Figure 1 shows our second scene comprised mainly of the 337-million-triangle Boeing 777 dataset from the Boeing Corporation. The Boeing scene configurations were created to expose different levels of global illumination complexity. The outside view allows many rays to exit the scene. The cockpit keeps rays contained and increases path lengths. The split scene is also mostly contained, but increases visibility of all geometry and light sources. Each of the first two setups are lit with one simple area light source. For the split Boeing scene, we split the model about a plane and placed the result into a box with an open front face, and we light the scene with 56 area light sources (Figure 1). Because of the exposure of geometry and the use of more light sources, this is our most difficult scene. We believe this is the first time that the highly-complex Boeing 777 model has been rendered with full global illumination.

5.1 Resource Utilization

Figure 5 shows how the resources were used for different tasks in the path tracer for the Split Boeing scene and
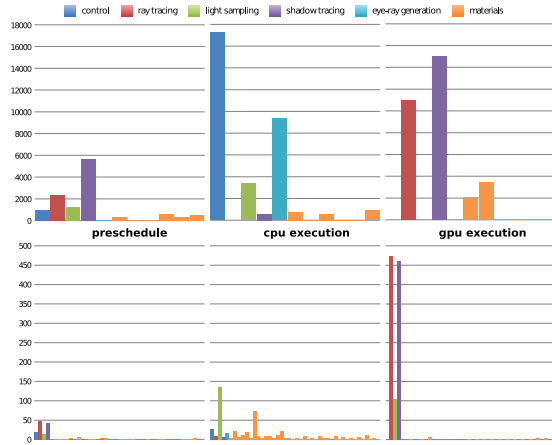
Figure 5: *The top pair of graphs shows the breakdown of execution times on a per-kernel basis of our split-Boeing 777 scene, with a total runtime of 62,668 seconds. The bottom pair of graphs show execution times for the Starship Troopers scene with 243 million triangles and large partitioned textures with a total runtime of 2966 seconds.*

the Troopers scene tessellated to 243 million triangles with large textures. Both the CPU and the GPU perform sizable amounts of computation; the Boeing scene uses the CPU primarily for running the *LightSampling* kernel, which was specified to prefer running on the CPU, and the *Control* kernel. The GPUs primarily traced rays and shadow rays. The Troopers scene exhibits slightly different utilization: The CPUs are used heavily for material shading, especially the first phases of shading, where typically data is read from texture and stored in transient data for processing at a later phase. GPU utilization for the Troopers scene is similar to the Boeing scene.

Figure 6 provides resource usage characterization plots. When rendering the Boeing scene, the CPU resources are well utilized, and because of virtual memory, it is easy to switch tasks and not have to explicitly wait for data before processing. The GPUs are also fairly utilized since we allow them to work on other tasks while they wait for disk reads, but they still exhibit more stalling than the CPUs. This shows that one of our main bottlenecks is still waiting for disk, with waiting for data transfer across the PCI-Express bus a secondary bottleneck. We have good cache utilization since only a few thousand out-of-core accesses occur. However, these out-of-core accesses account for a large percentage of the runtime, and this gives us a feeling of how poorly the path tracer could perform *without* the caching mechanisms in our data-management layer. The cache hit rate is low in the case of the Troopers scene, due to the highly fragmented nature of the material textures. The CPU and GPU utilization is much worse than the Boeing scene, because we serialize disk reads, and there is not enough work in the system, but

the overall runtime is still quite good at 2966 seconds for 512 samples per pixel (2551 seconds are spent reading static data).

5.2 Scaling with Scene Size and Texture
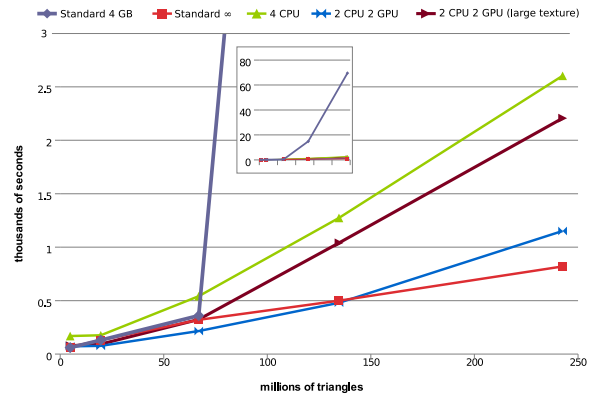
Figure 7: *Scaling with scene size. The images of the 134 million triangle Troopers scene were generated at 1280x720 resolution with 64 samples per pixel. The inset is a zoom-out of the graph to show full scale. The (large texture) configuration shows how scaling changes when 35 textures comprising 4.4 GB data replace a single texture of 108 MB.*

In order to demonstrate scalability with scene size, we additionally implemented a simple multi-threaded in-core CPU path tracer built on a state-of-the-art ray tracer [WH06] as a separate piece of software. This standard path tracer can consume our pre-processed scenes as input with only minor data modifications (our system uses a two-level kd-tree, but the standard path tracer uses a traditional single-level tree; the second level of the two-level tree is built exactly as the single-level tree is). We emulated unlimited memory for our in-core path tracer by running it on our master node, which has enough main memory to hold each of the Troopers scenes without shading data. The scenes are mapped from disk using the Linux mmap function call in order to allow scenes larger than main memory, and it is left to the operating system to swap the data in and out of core. Both applications solve essentially the same problem, the primary difference being the overhead of the data management layer of our system, the storage of transient data, and more expensive shading. This standard path tracer performs only monochrome shading using Lambertian materials, and so it should be noted that a "production-level" path tracer of this type is likely to be much slower, particularly when large textures are incorporated. Despite these disadvantages to our system, we perform well when compared to the standard path tracer.

Figure 7 shows the trends as scene size increases. The standard path tracer, unfettered by memory limitations, shows sub-linear scaling with scene size, which is common in ray
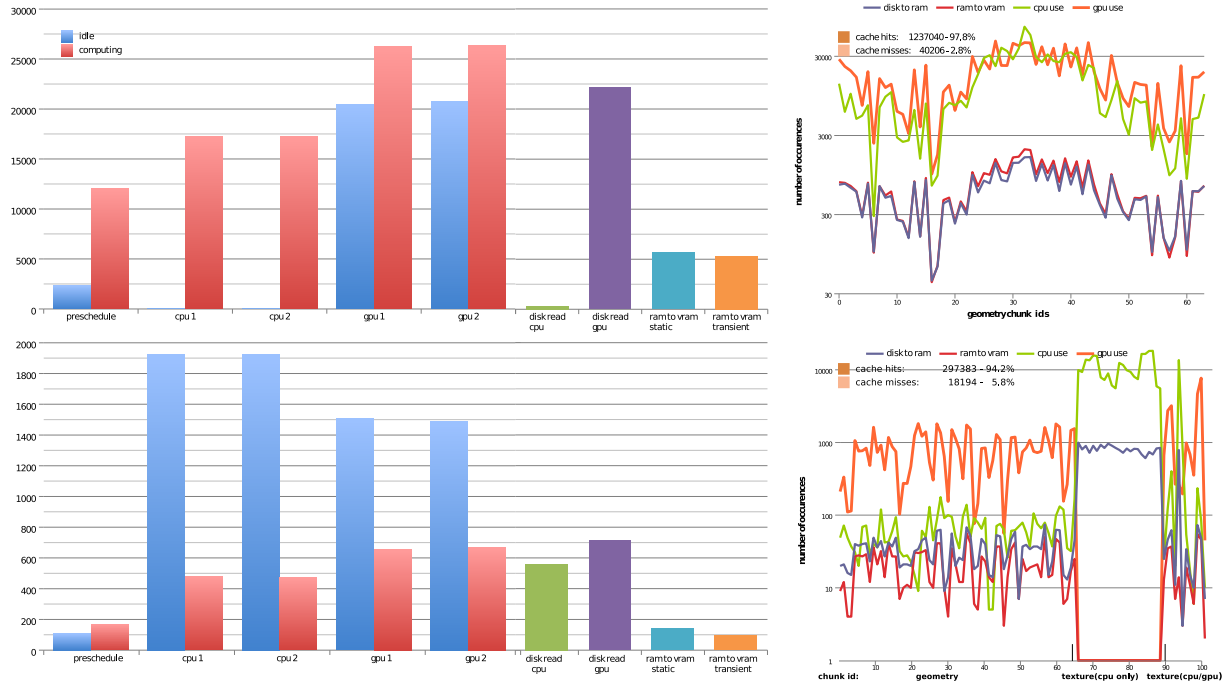
Figure 6: *Resource characterization. Top row: split-Boeing scene. Bottom row: 243 M Troopers scene with partitioned textures.*

tracing applications. Under more hostile memory conditions, however, the standard path tracer scales very poorly with scene size, taking roughly 100 times as long to compute the image on the largest scene. Our solution, when run only on CPUs presents a good middle ground. The solution is robust to out-of-core scene sizes, and while it does not scale sub-linearly, the super-linear scaling is more graceful. Our solution with GPUs is faster even than the unlimited standard solution for medium sized scenes, and is always twice as fast as our CPU-only configuration. Because this is true even for the smaller in-core scenes, this implies that while the extra GPU memory for caching may be beneficial, the compute resources of the GPU are also helpful. Our system clearly alleviates much of the disk bandwidth bottleneck.

As a second scaling characterization, we compared the 134-million-triangle Troopers scene with one versus multiple textures. In the small texture case, all of the texture can be packed into a single out-of-core chunk, while the large partitioned textures are each represented by a chunk. This trend can be seen in Figure 7. The trend to out-of-core begins slightly sooner with the larger set of textures. It is important to show scalability with texture size, and with the number of textures in order to ensure that larger and more complex scenes will continue to perform well with our system design. The scalability shown is similar to adding more geometry (for example, the 134-million-triangle Troopers scene with large texture is roughly the same size, and takes roughly the

same amount of time as the 243-million-triangle Troopers scene with small textures).
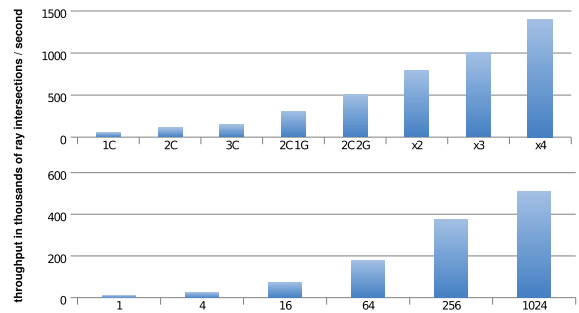
Figure 8: *Top graph: System scaling with increasing computing resources. Here 'C' means CPU and 'G' means GPU. 'xN' denotes the number of nodes run with two CPUs and two GPUs. Images were run with 1024 samples per pixel. Bottom graph: System scaling with increasing samples per pixel, clearly showing that our efficiency increases with samples. Both graphs use the 134 million triangle Troopers scene, and the rays/second values are obtained by counting the total number of rays and shadow rays which are spawned, and dividing by the runtime of the program. All images were rendered at 1280x720.*

5.3 Scaling with Resources and Work

The top of Figure 8 shows how we scale with additional resources. Scalability with a single node begins to slow at three CPUs; however, the addition of GPUs again increases scalability due to the superior processing capabilities, and also due to the extension of the static data cache. Up to four nodes, we show nearly linear scaling. This is unlikely to scale beyond many more nodes because we rely on garnering natural coherency, and because of naive load balancing. However, we have found that if more work (such as samples per pixel or larger image sizes) is added to the system, scalability will get better, as shown in Figure 8. As we request more samples per pixel, the throughput of the system increases dramatically. Taking 16 samples vs. four samples is three times as efficient, while taking 1024 samples vs. 256 samples is about 1.4 times as efficient.

5.4 Scene Layout Complexity

Table 3 shows the rendering times for six configurations of the Boeing 777 scene. We rendered the three scenes, each viewed with both simple ray tracing and with full unbiased global illumination via path tracing. As one would expect, the view of the Nose of the Boeing from outside in space is the cheapest rendering configuration. There are two reasons: several geometry chunks are never seen from any path; and most paths have a very short depth, as statistically it is very easy for bounced rays to traverse into space. Only a few regions, inside the turbine for example, show global illumination for this scene. For the same reasons, ray tracing is not much cheaper for this scene. Additionally, because of the simple lighting, few samples are needed for convergence.

The Cockpit view of the Boeing 777 is the next highest in scene complexity. Although the eye can only see a few geometry chunks directly, because of global illumination, nearly every chunk is visited during simulation. Global illumination is highly visible in this scene (see Figure 9). Simple ray tracing of this scene is relatively inexpensive because only a small percentage of the geometry chunks are touched during rendering; very few chunks comprise the cockpit, and the light source is also inside the cockpit.

The split view of the Boeing 777 is the most complex scene of the three. Through global illumination, every geometry chunk in the scene can see every other geometry chunk in the scene through some number of indirections. The scene is made somewhat worse because of our choice to use a medial splitting kd-tree as our high-level scene subdivision. Because of this, although the scene from our eye to the plane is empty, we are still traversing through many geometry chunks, each of which is required to be in-core during ray tracing. This, coupled with the 56 light sources, is also why the ray tracing is still so expensive for this incarnation of the Boeing 777.

Scene	Samples	Ray Traced	Path Traced
Boeing Nose	512	1139	1600
Boeing Cockpit	1024	2544	23461
Boeing Split	1536	22160	62688

Table 3: *Timing in seconds for 1280x720 rendering the various Boeing 777 configurations.*

6 Discussion

Implementation for Hybrid Resources Our GPU kernels are written in NVIDIA's CUDA framework for general-purpose computation [NVI06]. CUDA exposes a programming model of parallel threads grouped into thread blocks that run kernel programs on input data. CUDA threads have access to both global GPU memory as well as a local per-block shared data cache.

CUDA contexts are associated with a CPU thread and cannot be accessed from other CPU threads, hence in our system we control each GPU context from a distinct CPU thread. This limits our data management capabilities, because although CUDA can allocate special main memory buffers that allow faster transfer speeds to and from GPU, as well as asynchronous copy operation, we are unable to make use of them between contexts. Our graphics hardware is not capable of concurrently copying data and running kernels. Newer hardware providing this functionality should allow for much more efficient engineering of the data flow to and from GPUs. We estimate at least a 25% speedup on the rendering of some scenes from data transfer alone with next-generation Tesla GPUs if the thread-data sharing restrictions are relaxed.

Certain tasks will run better than others on the GPU. Generally tasks with limited intermediate data, that perform a large amount of computation, and that are mostly independent have the most to gain with having a GPU implementation. As an example, the performance of our *LightSampling* kernel suffers since it computes a large amount of intermediate data to be used for importance sampling. On the other hand, ray tracing requires very little intermediate computation to be stored for reuse, and the performance is very good.

Takeaway Points Our system scales well with resources, and efficiency increases as more work is introduced to the renderer. The renderer also scales gracefully as scene geometry and texture increase. Based on our results, at some level it may be necessary to introduce more work to continue scaling. At 1024 samples per pixel, our system scales quite well to four nodes with two CPUs and two GPUs each.

The basis for our scalability is our static data cache combined with our scheduler, which, like Pharr et al. [PKGH97], allows coherent computation and reduces disk access. Our extension of this queueing and caching to data other than
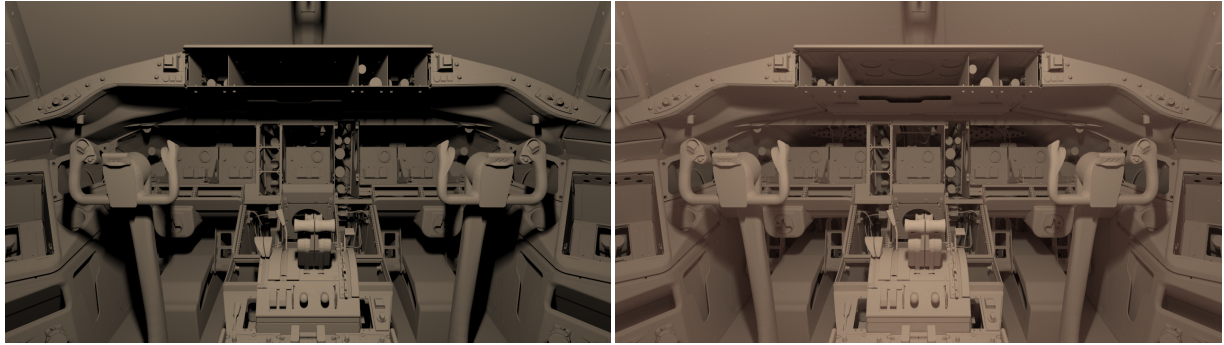
Figure 9: *A cockpit view of the Boeing 777. Left: Ray tracing. The dark areas correspond to geometry that cannot see the light source. Right: Path tracing. Those same areas are illuminated due to indirect lighting.*

rays allows us to use GPUs for general computation, drastically increasing computing power and further extending the static data cache for even fewer disk reads.

Despite good performance, the system is quite complex and if possible, an in-core implementation is desirable. The hybrid resources are clearly beneficial whether the target is in-core or out-of-core rendering. Even on our in-core scenes, the GPU configurations were twice as fast as those without GPUs. If data transfer costs can be hidden or alleviated, GPUs could provide an even larger boost. Tests on our stand-alone *RayTracing* kernel show that GPU ray casting is roughly 6 to 12 times faster than CPU ray casting.

7 Conclusions and Future Work

We have presented a system that enables the rendering of globally illuminated images of large, complex scenes. We achieved this goal by developing an efficient out-of-core data-management layer, and coupling this with an application layer containing a path tracer. The system effectively exploits hybrid resources to accelerate rendering and optimize memory utilization, resulting in robust scalability when compared to related systems.

Although we are mostly interested in leveraging our data management framework for graphics applications, the framework should work well in situations where there is a large amount of static data used, and where the application can benefit from GPU acceleration. The only requirement for algorithms is that they map to the concepts of transient and static data and kernels.

The system is currently implemented in such a way that all transient data is assumed to be in-core. This assumption can fail, and in this case we rely on the operating system to handle swapping for us. As a consequence, we limit the amount of concurrent work generated in our *Control* kernel, in order to keep the queues in-core. In the future, efforts should gen-

erate out-of-core queues, which would allow running more tasks concurrently.

Currently each node's cache is independent of one another, and each node has a copy of all data in its local disk. If a high-speed interconnect is employed, sharing caches between nodes may be faster than reading from disk. Future work could investigate this extension of the static data cache.

Our application scales well with scene size, and also shows reasonable scalability with single node resources as well as scalability with additional nodes. However, as with all frameworks of this type, scheduling improvements, such as disk-load-aware pre-fetching, coupled with intelligent assistance of cache eviction policy are important for increasing both the scalability and performance of the system, and we will continue to pursue these improvements.

8 Acknowledgments

References

[Ant06] ANTICONA M. T.: A GRASP algorithm to solve the problem of dependent tasks scheduling in different machines. In *IFIP AI* (2006).

[BBH08] BERNARDIN T., BUDGE B. C., HAMANN B.: Stacked-

widget visualization of scheduling-based algorithms. In *SoftVis '08* (2008), pp. 165–174.

[BH04]   BOYER W. F., HURA G.: Dynamic scheduling in distributed heterogeneous systems with dependent tasks and imprecise execution time estimates. In *Parallel and Distrib. Comp. and Sys.* (2004).

[CB04]   CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. In *Eurographics Symposium on Rendering* (June 2004), pp. 133–142.

[CDM97]   CHAN Y.-N., DANDAMUDI S. P., MAJUMDAR S.: Performance comparison of processor scheduling strategies in a distributed-memory multicomputer system. In *International Parallel Processing Symposium* (Apr. 1997).

[CLF*03]   CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum 22*, 3 (Sept. 2003), 543–552.

[DGP04]   DEMARLE D. E., GRIBBLE C. P., PARKER S. G.: Memory-savvy distributed interactive ray tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (June 2004), pp. 93–100.

[FMH05]   FRADIN D., MENEVEAUX D., HORNA S.: Out of core photon-mapping for large buildings. In *Eurographics Symposium on Rendering* (June 2005), pp. 65–72.

[GR08]   GRIBBLE C., RAMANI K.: Coherent ray tracing via stream filtering. In *Interactive Ray Tracing* (Aug. 2008), no. 3, pp. 59–66.

[Kaj86]   KAJIYA J. T.: The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)* (Aug. 1986), pp. 143–150.

[KS02]   KATO T., SAITO J.: "Kilauea"—parallel global illumination renderer. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Sept. 2002), pp. 7–16.

[Lef93]   LEFER W.: An efficient parallel ray tracing scheme for distributed memory parallel computers. In *ACM SIGGRAPH Symposium on Parallel Rendering* (Nov. 1993), pp. 77–80.

[LI06]   LINDSTROM P., ISENBURG M.: Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (Sept./Oct. 2006), 1245–1250.

[MSS*06]   MEHTA A. M., SMITH J., SIEGEL H. J., MACIEJEWSKI A. A., JAYASEELAN A., YE B.: Dynamic resource management heuristics for minimizing makespan while maintaining an acceptable level of robustness in an uncertain environment. *ICPADS 2006* (2006).

[NFLM07]   NAVRATIL P., FUSSELL D., LIN C., MARK W.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing* (Sept. 2007), pp. 95–104.

[Nos98]   NOSSAL R.: An evolutionary approach to multiprocessor scheduling of dependent tasks. *Future Gener. Comput. Syst. 14*, 5-6 (1998).

[NVI06]   NVIDIA CORPORATION: NVIDIA CUDA compute unified device architecture programming guide. http://developer.nvidia.com/cuda, 2006.

[OLG*07]   OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (Mar. 2007), 80–113.

[PH04]   PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[PKGH97]   PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P. M.: Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH 97* (Aug. 1997), Computer Graphics Proceedings, Annual Conference Series, pp. 101–108.

[PVL*05]   PELLACINI F., VIDIMČE K., LEFOHN A., MOHR A., LEONE M., WARREN J.: Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics 24*, 3 (Aug. 2005), 464–470.

[RCJ99a]   REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *Symposium on Parallel Visualization and Graphics* (Oct. 1999), pp. 21–28.

[RCJ99b]   REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for realistic image synthesis. In *Proceedings of the International Conference on Parallel Computing, Fundamentals and Applications* (Aug. 1999), Imperial College Press, pp. 21–28.

[RDK*98]   RIXNER S., DALLY W. J., KAPASI U. J., KHAILANY B., LOPEZ-LAGUNAS A., MATTSON P., OWENS J. D.: A bandwidth-efficient architecture for media processing. In *Inter. Symp. on Microarch.* (Dec. 1998).

[RKKS*07]   RAGAN-KELLEY J., KILPATRICK C., SMITH B. W., EPPS D., GREEN P., HERY C., DURAND F.: The light-speed automatic interactive lighting preview system. *ACM Transactions on Graphics 26*, 3 (July 2007), 25:1–25:11.

[SM03]   SHIRLEY P., MORLEY R. K.: *Realistic Ray Tracing*. A. K. Peters, Ltd., 2003.

[TFFH94]   TELLER S., FOWLER C., FUNKHOUSER T., HANRAHAN P.: Partitioning and ordering large radiosity computations. In *Proceedings of SIGGRAPH 94* (July 1994), Computer Graphics Proceedings, Annual Conference Series, pp. 443–450.

[Wal04]   WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, CG Group, Saarland University, 2004. http://www.mpi-sb.mpg.de/~wald/PhD/.

[War94]   WARD G. J.: The RADIANCE lighting simulation and rendering system. In *Proceedings of SIGGRAPH 94* (July 1994), Computer Graphics Proceedings, Annual Conference Series, pp. 459–472.

[WDS04]   WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *Eurographics Symposium on Rendering* (June 2004), pp. 81–92.

[WGER05]   WEXLER D., GRITZ L., ENDERTON E., RICE J.: Gpu-accelerated high-quality hidden surface removal. In *Graphics Hardware 2005* (July 2005), pp. 7–14.

[WH06]   WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 61–69.

[WK06]   WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierachy. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering* (June 2006), pp. 139–150.

[YLM06]   YOON S.-E., LAUTERBACH C., MANOCHA D.: R-lods: fast lod-based ray tracing of massive models. *The Visual Computer 22*, 9-10 (2006), 772–874.