PARALLEL IRRADIANCE CACHING ON THE GPU

Nathaniel L. Jones
Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

18.337/6.338 Parallel Computing
Final Report
December 16, 2013

ABSTRACT

While ray tracing is highly parallelizable in concept, the Radiance suite of programs for architectural global illumination simulation was written for serial execution and makes use of certain heuristic techniques that are not easily performed in parallel environments. It uses irradiance caching to store and reuse the results of expensive indirect irradiation computations. The irradiance cache has unpredictable size, and Radiance alternately reads and adds to it during a simulation, making it unfriendly to parallelization.

The irradiance caching method proposed in this paper uses the OptiX™ engine for GPU ray tracing. Irradiance records are stored in buffers in GPU memory which can as necessary be used to create a bounding volume hierarchy (BVH) of known irradiance values based on their valid ranges. Queries into the irradiance cache are performed by casting a short ray into the BVH. Because the CPU maintains a handle to the irradiance cache, it is possible to read irradiance records from a file prior to simulation and write them back afterward. Therefore, this proposed method maintains consistency with the expected use and output of Radiance while performing calculations at higher speeds.

INTRODUCTION

Lighting of interior spaces is an important yet poorly understood component of architecture. While many countries have standards for acceptable indoor illumination levels, few people have the ability to accurately and quantitatively gage, let alone predict, illuminance levels. Because building designers must meet indoor illuminance criteria while balancing the desire for daylight with the problems of heat loss through windows and productivity loss caused by glare, physically-based global illuminance simulation is needed during building design.

The Radiance suite of programs (Larson and Shakespeare, 1998) has become the gold standard for global illumination calculation used by architects and lighting designers. This can be attributed to Radiance's flexibility, open source availability, and extensive validation through comparison to physical measurements (see Ochoa *et al*. (2012) for list of references). Radiance is used as a simulation engine by widely-used building performance simulation tools such as IES<VE>, Ecotect®, OpenStudio, DAYSIM, and DIVA for Rhino. However, Radiance simulations tend to be slow, especially in scenes with complex geometry. As a result, global illumination simulation using Radiance tends to take place late in the design process, after most design decisions are made, or else simplified models are used for simulation that may not accurately predict conditions in the physical building. Faster simulations are necessary in order to better predict and design interior lighting.

In this paper, we propose a solution to speed up Radiance calculations by tracing multiple primary rays in parallel on a graphics processing unit (GPU). First, we briefly describe a framework built on Nvidia®'s OptiX™ ray tracing engine that allows our port of the Radiance source code to run on the GPU. Next, we introduce irradiance caching as a method commonly used in Radiance to speed up serial calculations and describe our method for reading an irradiance cache (IC) on the GPU by mapping it to a bounding volume hierarchy (BVH). Then, we describe how create and adaptively vary the size of an IC on the GPU using a two-stage method. Finally, we propose a multi-stage method that promises to produce more accurate irradiance values by tracing more ray reflections without an exponential increase in the number of rays cast.

BACKGROUND

Backward Ray Tracing

The Radiance package includes a number of executable programs built around a bespoke backward ray tracing engine. In backward ray tracing, primary rays are emitted by from an origin point (a virtual camera or illuminance sensor) to sample the environment. Wherever a ray intersects a surface, it recursively spawns one or more new rays, depending on the surface material, and gathers their results into a single value that is returned as the parent ray's result (Figure 1). Typically, a small number of spawned rays are required for direct and specular reflections, and a much larger number of rays are spawned to sample the indirect irradiance due to ambient lighting at the intersection point. In Radiance, each ray returns red, green, and blue values with units of radiance ($W \bullet sr^{-1} \bullet m^{-2}$). The array of values returned from the primary rays can be displayed as an image; however, because the range of returned values typically varies by several orders of magnitude, these images are more easily interpreted using false colors (Figure 2). Additionally, each ray carries with it a weight corresponding to the maximum contribution it may make to its parent ray's value. Rays weighted below a minimum threshold may be terminated without spawning new rays in order to prevent exponential growth in the number of rays traced by the simulation.

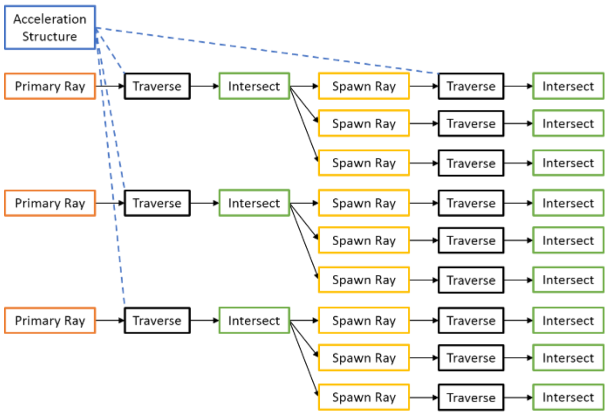

Figure 1 In backward ray tracing, each primary ray traverses an acceleration structure of scene geometry until it intersects a surface. The intersection usually spawns additional rays.

Irradiance Caching

The number of rays to traverse can be further reduced by caching and reusing indirect irradiance values. While direct and specular reflections change abruptly over spatial dimensions, ambient lighting due to indirect irradiance is less variable, so a single value may be applied to all ray intersections within a calculated radius of the point where it was measured. For instance, given two cached irradiance values at points *E1* and *E2* in Figure 3, the irradiance at point *A* may be found by interpolation, and the irradiance at point *B* may be found by extrapolation. Only when a ray intersection is not contained within the validity radius of any IC record (such as at point *C*) must a new record be calculated and added to the IC. This strategy reduces overall ray tracing time by an order of magnitude (Larson and Shakespeare, 1998), but it also eliminates the possibility of straightforward parallelization of the Radiance source code because the final value of each ray depends on the IC records created by previous rays (Figure 4).

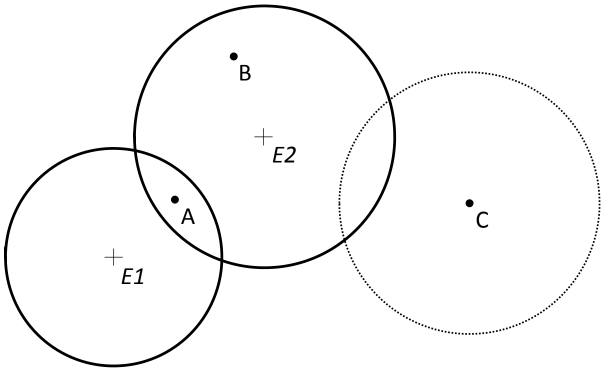

Figure 3 IC records may be applied to all points within valid radii (Larson and Shakespeare, 1998).

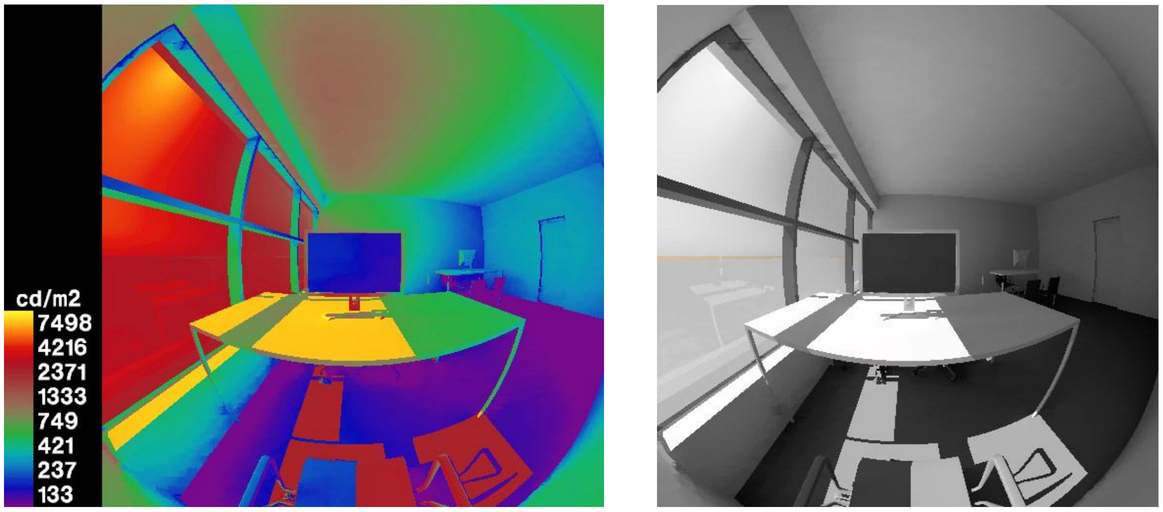

Figure 2 Radiance image tone-mapped with false colors (left) and low dynamic range photorealistic colors (right).

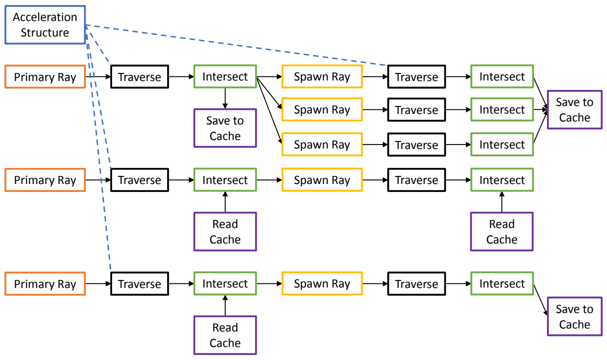

Figure 4 In single-threaded ray tracing, the program may alternately write to and read from the IC.

Various methods have been proposed for creating an IC using parallel threads. On CPU clusters, most strategies involve occasional synchronization of separate local ICs for each thread. This may occasionally result in duplicate IC records created simultaneously by more than one CPU. On UNIX systems, multiple instances of Radiance may share a single irradiance cache using network file locks (Larson and Shakespeare, 1998). Synchronization can also be performed using MPI (Koholka *et al.*, 1999; Debattista *et al.*, 2006). Dubla *et al.* (2009) propose a multi-threaded approach that allows wait-free synchronization of local ICs.

Using GPUs, others have implemented IC creation as a pre-process carried out prior to ray tracing-based image creation. Generally, these approaches are geared toward creation of visually plausible images rather that attempting to achieve physical accuracy. Wang *et al.* (2009) use photon mapping and k-means clustering to select points in the scene for use as seeds for IC records. Křivánek and Gautron (2009) use splatting to store irradiance values in a two-dimensional cache that may be projected onto the scene from the camera's vantage point. Using pure ray tracing, Frolov *et al.* (2012) create an irradiance cache in 20-30 passes, where each pass involves both addition of IC entries visible to the camera and elsewhere in the scene for secondary rays.

Ray Tracing on the GPU

While GPUs are primarily designed for raster rendering, the development of GPU-based ray tracers has closely paralleled the development of programmable GPU raster pipelines. As early as 2002, the GPU was used for geometry traversal, ray intersection testing, and ray result shading (Purcell *et al.*, 2002). Dietrich *et al.* (2003) developed a ray tracing library, OpenRT, that imitates OpenGL® in structure, including programmable shaders to execute at ray intersections; however, their specification was not widely implemented by hardware manufacturers. Eventually, general purpose GPU (GPGPU) language extensions such as Nvidia®'s Compute Unified Device Architecture (CUDA™) made it possible to implement all components of a ray tracing engine on GPU shader processors (Aila and Laine, 2009; Wang *et al.*, 2009). In 2010, Nvidia® released the OptiX™ ray tracing engine, which uses CUDA™ to perform both ray traversal and shading on the GPU (Parker *et al.*, 2010).

The OptiX™ library is intended to be easily inserted into existing CPU-based code to replace a serial ray tracing engine. OptiX™ provides built-in BVH creation and ray traversal algorithms to detect ray-surface intersections. The programmer is required to re-implement ray generation, intersection testing, closest hit, any hit, and miss algorithms as CUDA™ programs. OptiX™ compiles these programs into virtual machine code to be further optimized by a just-in-time compiler to create device-specific instructions at runtime.

PARALLEL RAY TRACING WITHOUT IRRADIANCE CACHING

When the IC is disabled, Radiance's backward ray tracing algorithm is "embarrassingly parallel," and aside from the task of porting the C code to CUDA™, its parallelization is trivial (Figure 1). After the scene geometry is created and used to generate the BVH acceleration structure for ray traversal, a thread is created on the GPU for each primary ray, and all rays spawned by the primary ray or its descendants are traversed on that thread.

Conceptually, it is easy to think of all primary rays as being traversed simultaneously, but in reality, this is not the case. The tests we present were carried out on a workstation equipped with a 3.4 GHz Intel® Core™ i7-4770 processor and an Nvidia® Quadro® k4000 graphics card. While the images we show are each 512 × 512 pixels (that is, 262,144 primary rays), the GPU has only 768 CUDA™ cores. OptiX™ groups threads into warps of 32, so the GPU can process at most 24 warps simultaneously, while images of that size are divided into 8192 warps of adjacent rays. As a result, simulation time with OptiX™ generally varies linearly with the number of primary rays, despite the fact that primary rays are traced in parallel.

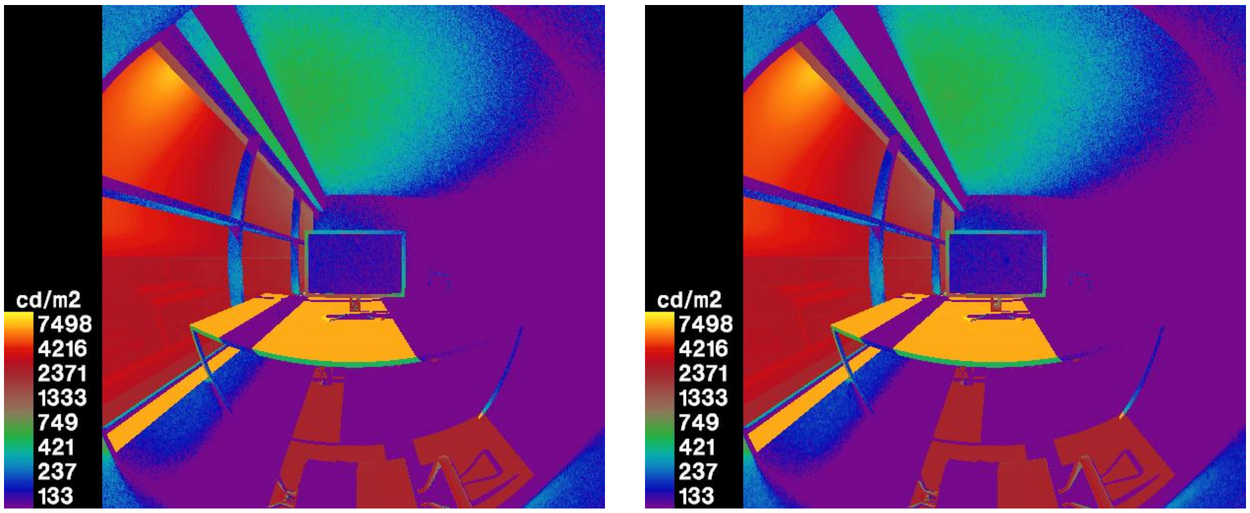

Figure 5 Results from our OptiX™ implementation (left) and Radiance (right) without an IC.

The values obtained from our OptiX™ implementation closely match those from Radiance without an IC (Figure 5). Furthermore, the OptiX™ implementation produces results in 27.6 seconds, while Radiance required 243 seconds with the same settings, not including the time required to read the input file. This represents an 8.8-fold speed improvement.

Our goal, however, is to match the image in Figure 2, which shows higher radiance values due to its IC but requires 4838 seconds to create in Radiance. We theorize that the higher values occur because the cached irradiance levels are themselves created by summing the contributions of rays that sample other cached values. Because the earlier created IC records started with a higher ray weight, this has the effect of artificially lowering the minimum ray weight threshold. While it is not yet practical to run tests with very low minimum ray weight thresholds[1], we can test our hypothesis on the GPU by reducing other accuracy parameters in order to reduce simulation time. Figure 6 shows the OptiX™ kernel and total simulation times (not including input file reading) for reduced-accuracy simulations with a typical minimum ray weight threshold of 0.002 and a low threshold of 0.0002. As the number of bounces allowed in gathering indirect irradiance increases, the total number of rays created and the total simulation time are expected to increase exponentially. However, the actual times level off quickly, suggesting that the minimum ray weight threshold is indeed a limiting factor. Furthermore, the lower threshold produces an image closer to the target than the normal threshold (Figure 7).

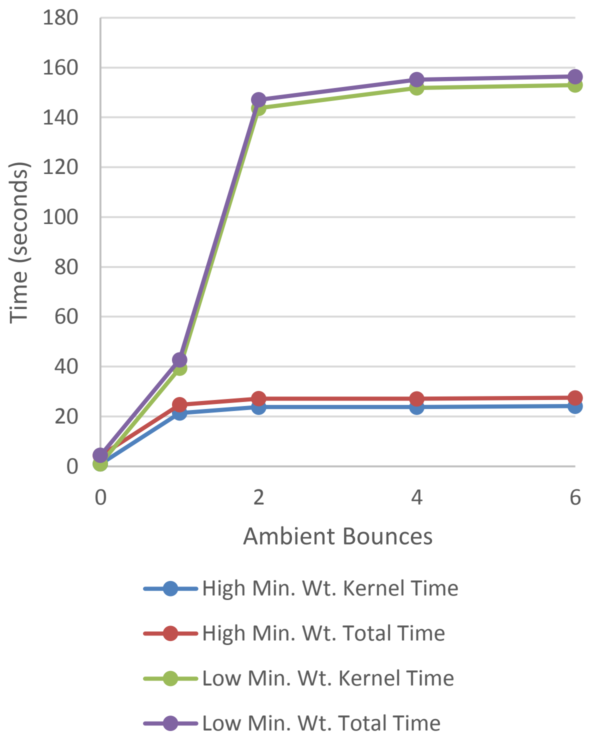

Figure 6 Simulation time dependence on number of ambient bounces.

[1] Simulations with low minimum ray weight thresholds can take days to run on the CPU. On the GPU, they result in high processor loads which can cause the GPU to become unresponsive and in turn cause Windows' Timeout Detection and Recovery (TDR) system to reboot the GPU, terminating any running simulation. By default, TDR occurs after 2 seconds without response, during which the screen appears to be frozen. For our tests, we have set the TDR delay to 10 seconds, which allows our GPU threads to remain active longer at the expense of user interaction. A better solution would be to use NVIDIA®'s Tesla® GPUs, which are intended solely for GPGPU applications and do not timeout.

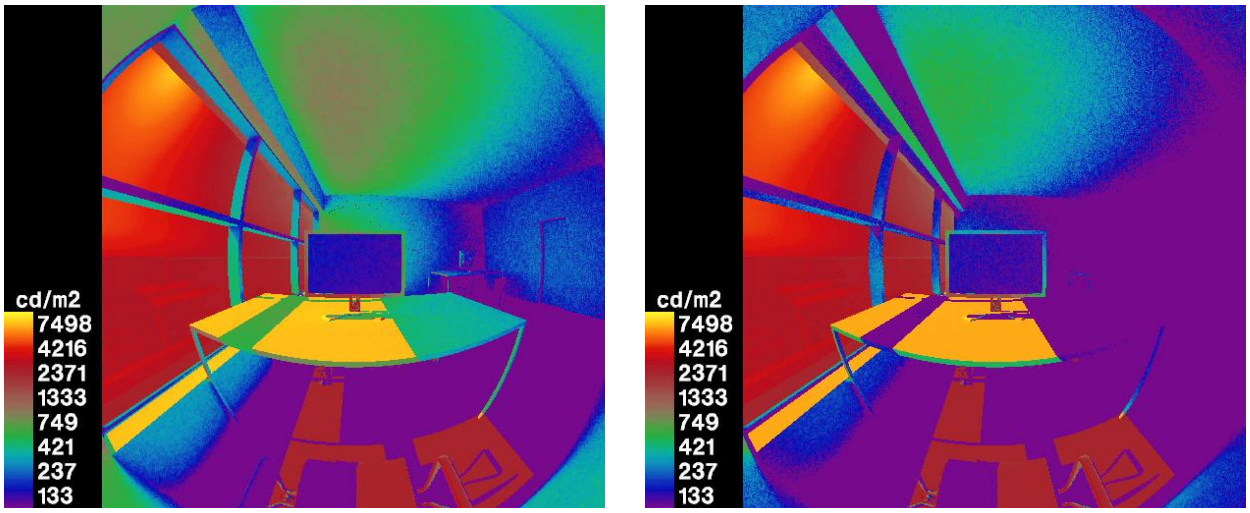

Figure 7 Results from our OptiX™ implementation with a low minimum ray weight (left) and normal minimum ray weight (right) using a reduced number of ambient divisions for indirect irradiance sampling.

READING FROM AN IRRADIANCE CACHE IN PARALLEL

Because Radiance runs on a single thread, it is trivial to read from and write to the same IC during a simulation. Radiance is also able to save its IC as a binary file to later use to enable multiple simulations of the same space. On the GPU we must read from and write to the IC at separate times. Here, we discuss our method for reading from the IC, using IC files created by Radiance as input.

Each IC record represents a disc over which a given indirect irradiance value is valid, along with directional vectors indicating the disc's orientation in space and gradients in the plane of the disc. Our first step is to enter all available IC records into a BVH acceleration structure. While OptiX™ generates the BVH automatically, we must specify a bounding volume for each disc. We define an axis-aligned bounding box (*AABB*) for each entry as

$$AABB = P \pm r\sqrt{1 - D^2} \qquad (1)$$

where P is the center point of the disc, r is its radius, D is the normal vector of the plane containing the disc, and all operations are element-wise (Figure 8). Because CUDA™ treats tuples as primitives, this is a rare case in which the source code closely resembles the mathematical expression. The *AABB*s of all IC record are independent and can be computed in parallel on the GPU, although their insertion into the BVH tree is a serial operation. Once the BVH containing IC values is created, our OptiX™ implementation can determine indirect irradiance values at each intersection by spawning a single ray into the IC BVH acceleration structure, rather than by spawning thousands of rays into the scene (Figure 9).

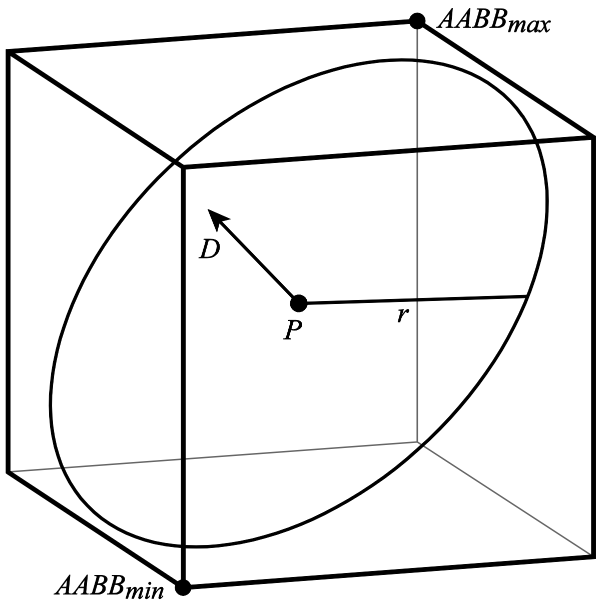

Figure 8 An axis-aligned bounding box for a disc.

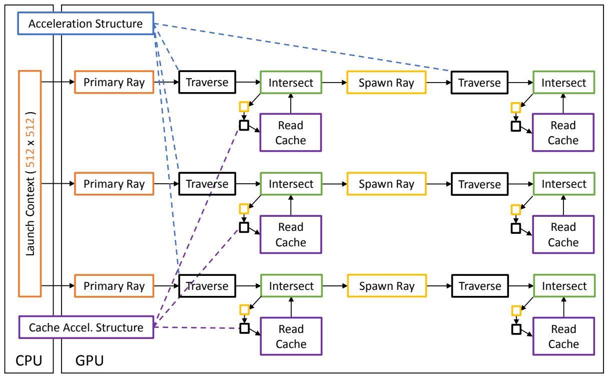

Figure 9 Our OptiX™ implementation uses a single ray at each intersection to find relevant IC records.

Figure 10 shows the time required to generate the scene from Figure 2 depending on the number of IC entries included in the BVH. The OptiX™ kernel time varies linearly with IC size, and the total simulation time (not including file reading) demonstrates a constant 3.6-second overhead. While our OptiX™ implementation requires 56.8 seconds to traverse the full 109,998 IC entries created by Radiance in our example scene, we note that Radiance itself is able to produce the same result in 6 seconds given the same IC file. (The file itself took well over an hour to create with Radiance.) Furthermore, the vast majority of these IC values will not be useful to final processing as they represent intermediate steps in creating other IC values (through multiple bounces). Clearly, a better approach would be to adaptively vary the size of the IC and copy only the necessary IC records to the GPU.

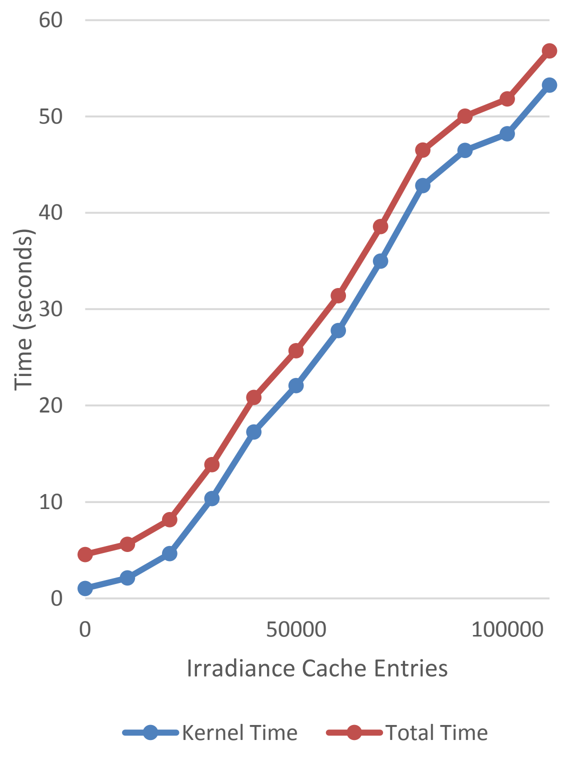

Figure 10 Simulation time dependence on IC size.

CREATION OF AN IRRADIANCE CACHE IN PARALLEL

A simple approach to creating an IC on the GPU is to generate IC records as a pre-processing step prior to generating the image. We use a second OptiX™ kernel for this purpose. In our ambient OptiX™ kernel, each primary ray in our grid returns a 72-byte IC record as its payload rather than a 12-byte RGB radiance value, as in our first OptiX™ kernel (Figure 11). The rules for spawning additional rays are unchanged between the two kernels. We launch our ambient kernel on a smaller number of threads, sending out one primary irradiance ray and returning one IC record (or fewer should the ray not intersect any surface) per thread. The number of returned IC records thus varies linearly with the number of primary rays (Figure 12). However, total simulation time also increases with the size of the IC produced (Figure 13). While large IC sizes result in slow simulations, small cache sizes can leave areas of the scene uncovered by any IC entry (Figure 14).

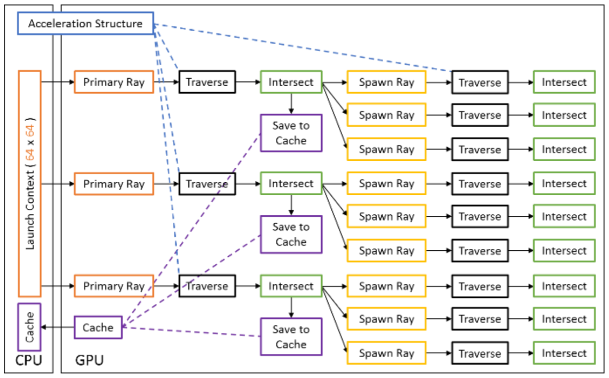

Figure 11 Our ambient kernel produces an IC record at from each primary ray.

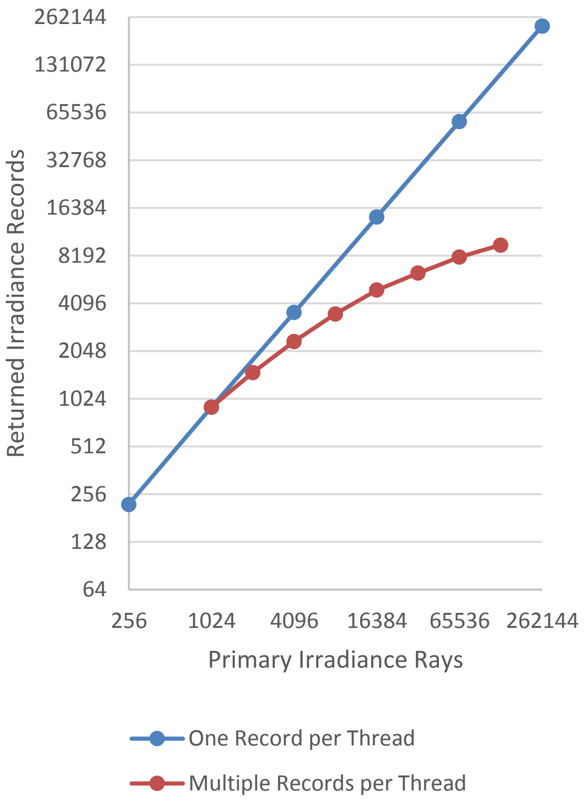

Figure 12 IC size dependence on number of primary rays in ambient kernel.

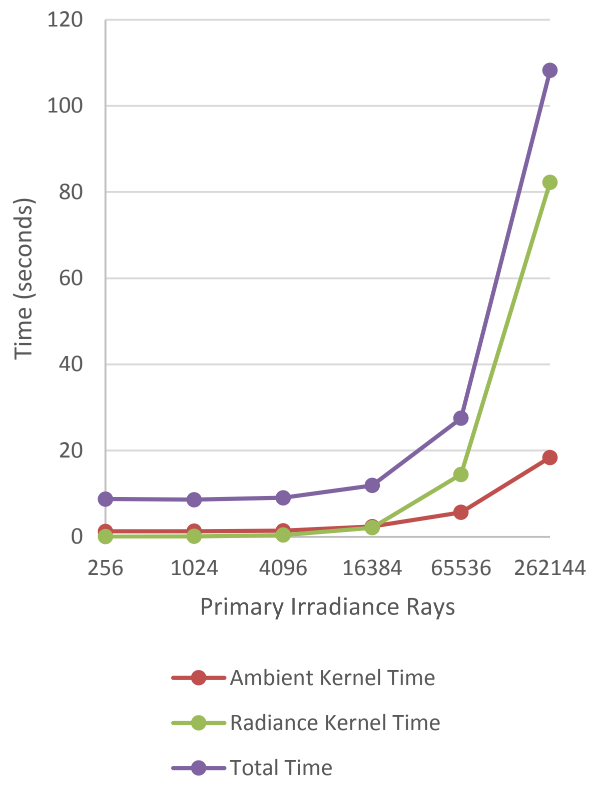

Figure 13 Simulation time dependence on number of primary rays in ambient kernel with one ray per thread.

Our solution is to allow the number of primary rays produced by each ambient kernel GPU thread to vary. The ray generation program loops until it reaches a user-defined limit, and generates a new primary ray with each loop in a direction defined by the following code and illustrated in Figure 15.

```
static __device__
float2 get_offset( unsigned int index ) {
    float2 offset = make_float2( 0.5f );
    float delta = 0.5f;

    for ( ; index > 0u; index >>= 2 ) {
        unsigned int x = index & 1u;
        unsigned int y = (index >> 1 ) & 1u;
        y = x ^ y;
        if ( x ) offset.x += delta;
        if ( y ) offset.y += delta;
        delta *= -0.5f;
    }

    return offset;
}
```

Because no BVH exists to search for overlapping IC records yet, the primary ray can only test for intersection with IC records generated on its own thread. As a shortcut, each primary ray other than the first is assigned as a parent IC record the closest record previously calculated by its thread. This is the record returned by the ray whose index is the current index less the greatest power of two less than or equal to the current index (Figure 16). If the parent's disc overlaps the new primary ray's intersection with the scene, then

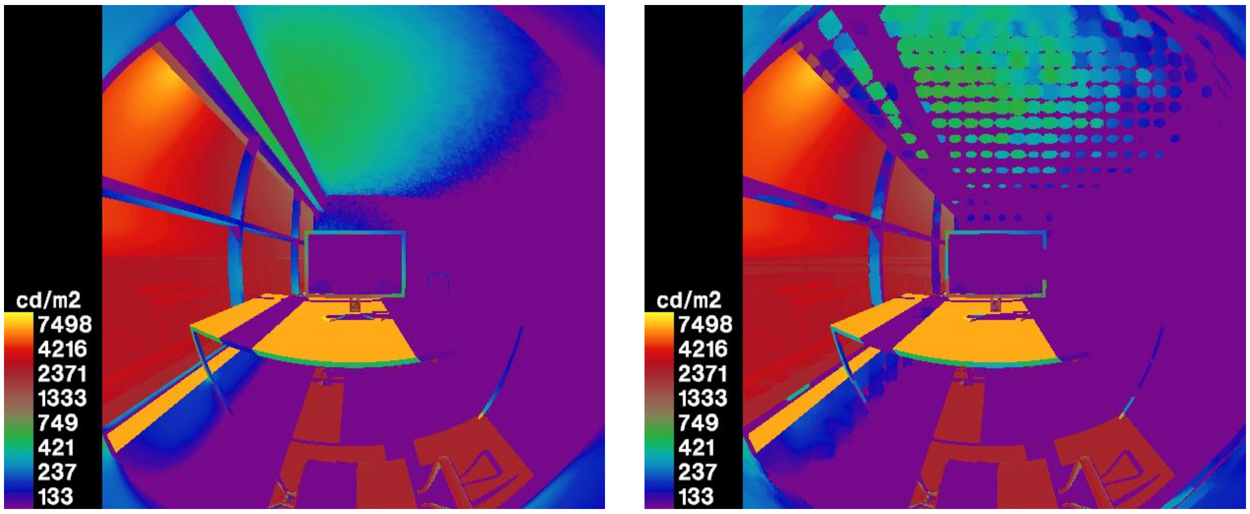

Figure 14 Results from our OptiX™ implementation with a large IC creating smoother irradiance gradients (left) and small IC providing insufficient coverage (right).

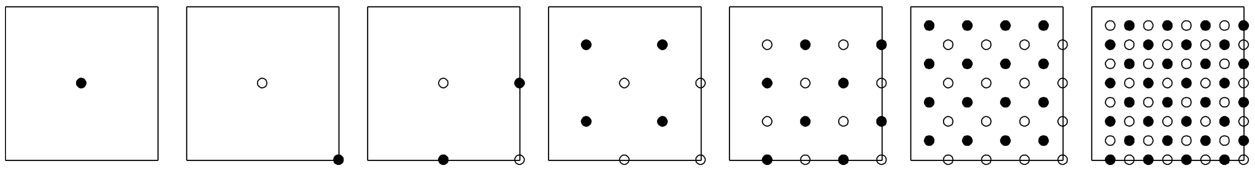

Figure 15 Order of primary rays sent by one thread. Filled circles indicate locations of new rays at each level.

no new IC record is generated for this ray or any of its future closest neighbors. This dramatically lowers the number of IC records created (Figure 12) and the overall simulation time (Figure 17) while providing good coverage for the scene (Figure 18).

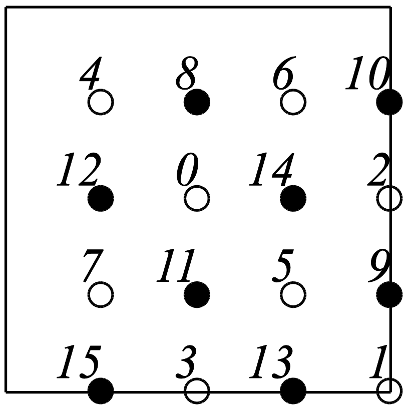

Figure 16 Parent IC records for each primary ray are the result of the ray whose index is the current index minus the largest possible power of two.

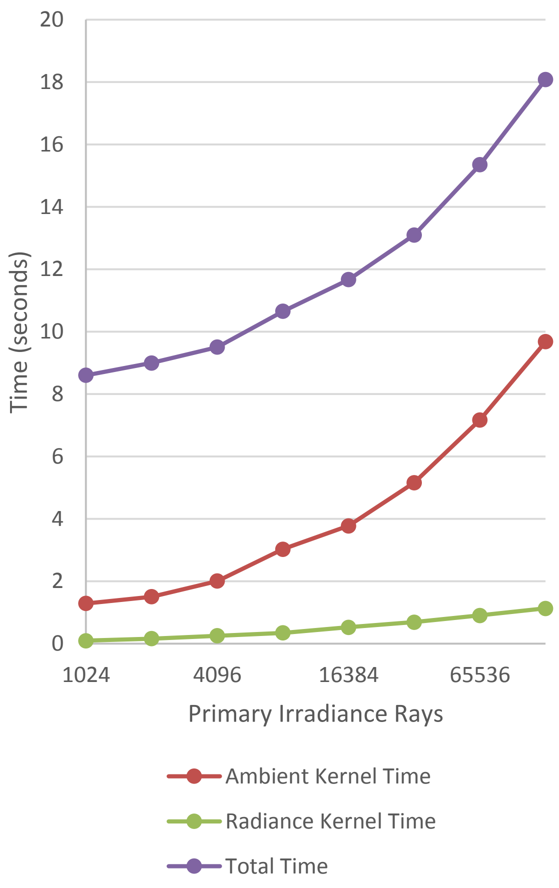

Figure 17 Simulation time dependence on number of primary rays in ambient kernel with multiple rays per thread.

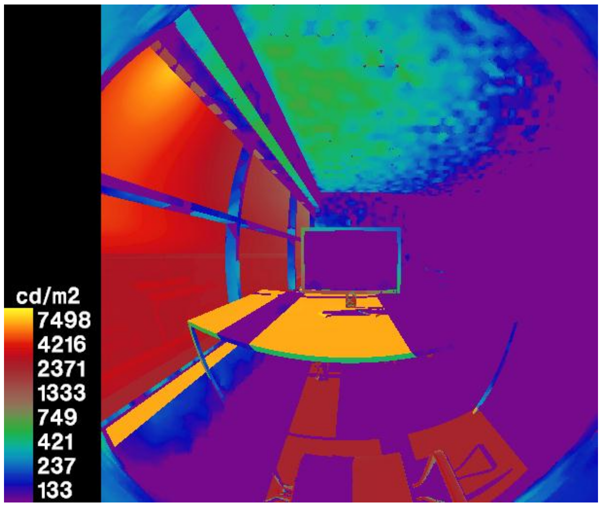

Figure 18 IC created with multiple primary rays per thread in our OptiX™ implementation.

ITERATIVE IRRADIANCE CACHE CREATION IN PARALLEL

We have theorized that the difference in radiance levels between Figure 2 and Figure 5 is caused by the presence of IC records that are created by summing rays that sampled other IC records. Because the GPU methods that we have explored so far do not allow this to happen, our simulated images tend to resemble the lower radiance values of Figure 5. We expect that we can account for this missing radiance by iterating our previous step. By running the ambient kernel multiple times, each time using as input an IC BVH created from the previous run's output, we can accumulate this missing radiance in linear time with the number of iterations, rather than in exponential time by changing the minimum ray weight threshold.

At present, our implementation of iterative calls to the ambient kernel is unreliable due to timeout issues[2]. However, preliminary tests using small ICs and small numbers of spawned rays show that this method can produce radiance values much closer to the target values. The left image in Figure 19 was rendered using seven calls to the ambient kernel, simulating six ambient bounces, in 32.1 seconds, of which 8.4 seconds represent GPU kernel activity and the rest is CPU overhead. A corresponding Radiance simulation with the same low accuracy settings takes 32.0 seconds (Figure 19 right). We hope that with continued work, our implementation's simulation time will improve and will scale better than Radiance to simulations with greater numbers of rays.

[2] See previous footnote.

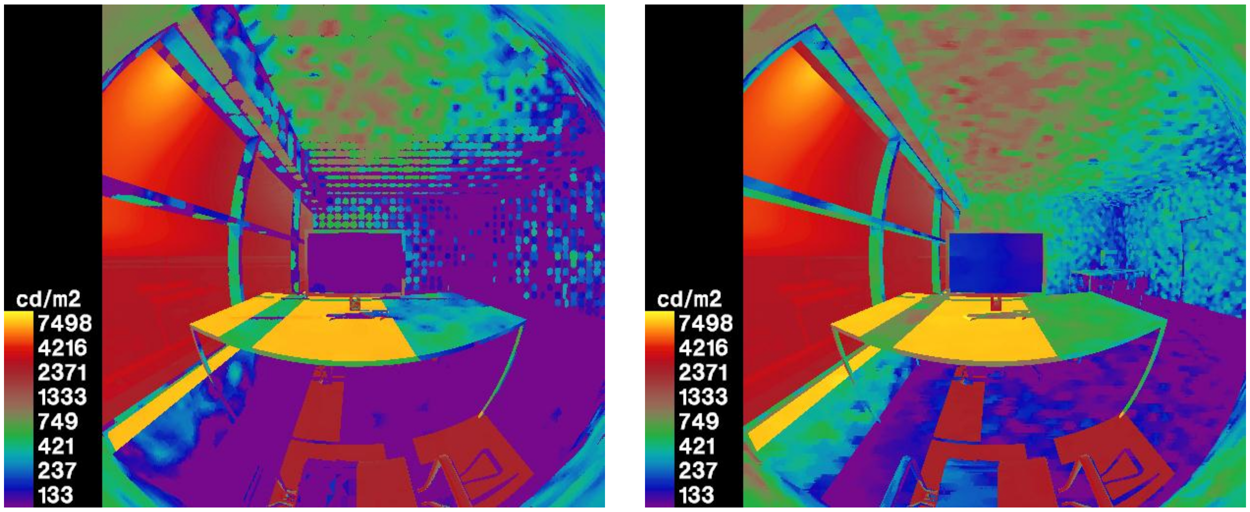

Figure 19 IC created with multiple passes and low accuracy settings in our OptiX™ implementation (left) and with the same settings in Radiance (right).

CONCLUSION

ICs on the GPU have potential to speed up parallel ray tracing for global illumination calculation, but they need to be used wisely. In this paper, we have demonstrated that core algorithms from Radiance can be implemented in OptiX™ to achieve an order of magnitude speed increase in basic backward ray tracing. We have shown further that small ICs that provide good scene coverage can be generated on the GPU in less time than is required to render the scene with a large IC created by Radiance. However, these small ICs create results with less radiance because a minimum ray weight threshold prevents them from accumulating indirect irradiance through many ambient bounces. We speculate that a multi-pass approach to generating IC records will account for this missing radiance, and preliminary testing of this approach shows that it has promise.

Continued work is necessary in a number of areas. First, a significant amount of code optimization can be done to improve the performance of our OptiX™ implementation. In particular, many branching methods ported from Radiance can be broken up into separate CUDA™ programs that can be called individually without occupying the GPU with instructions that will not be carried out. This will also be necessary in order to implement additional material types and light transport methods that are not handled by the current implementation. Second, additional work is necessary to determine appropriate placement of IC record seeds. We currently place all seeds in view of the virtual camera and use a fisheye lens to provide good scene coverage. However, while the IC records used to create the final image should all be in view of the camera, IC records from earlier iteration steps should be more evenly distributed about the space, including on the back sides of surfaces. Finally, additional work is necessary stabilize and reduce the overhead on the iterative approach to IC creation. This could involve reducing memory transfer between the GPU and CPU memory and varying accuracy parameters between steps.

There are many potential benefits to the architecture profession if Radiance algorithms can be parallelized on the GPU. Faster simulation results can be produced more frequently as an aid to design, and their sooner availability makes it less likely that the design will change during the simulation, rendering the simulation results useless. Accurate simulation results make it easier for architect to correctly size windows and provide adequate artificial lighting without consuming unneeded electricity. They also reduce the likelihood of glare, which can reduce productivity in a work environment. Faster ray tracing will also make annual simulations more practical, as these take much longer than point-in-time simulations. Thus, we believe that successful creation of ICs on the GPU will be of great benefit to future architects.

ACKNOWLEDGEMENTS

We wish to thank Professor Alan Edelman and Jeff Bezanson for their comments in carrying out this project, and our advisor Christoph Reinhart for his input. Frédo Durand and Jaroslav Křivánek offered useful insight into previous work on irradiance caching. Andrew McNeil of Lawrence Berkley National Laboratory provided suggestions for solving the missing radiance problem. Thanks also to J. Alstan Jakubiec for providing the model and Radiance input files used for timing and illustration.

REFERENCES

Aila, Timo and Samuli Laine, 2009. Understanding the efficiency of ray traversal on GPUs. *Proceedings of High-Performance Graphics 2009*, 145-149.

Debattista, Kurt, Luís Paulo Santos and Alan Chalmers, 2006. Accelerating the irradiance cache through parallel component-based rendering. *Proceedings of the 6th Eurographics conference on Parallel Graphics and Visualization*, 27-35.

Deitrich, Andreas, Ingo Wald, Carsten Benthin and Philipp Slusallek, 2003. The OpenRT application programming interface - towards a common API for interactive ray tracing. *Proceedings of the 2003 OpenSG Symposium*.

Dubla, Piotr, Kurt Debattista, Luís Paulo Santos and Alan Chalmers, 2009. Wait-Free Shared-Memory Irradiance Cache. *Proceedings of the 9th Eurographics Symposium on Parallel Graphics and Visualization*, 57-64.

Frolov, Vladimir, Konstantin Vostryakov, Alexander Kharlamov and Vladimir Galaktionov, 2013. Implementing irradiance cache in a GPU photorealistic renderer. *Transactions on Computational Science XIX*. Marina L. Gavrilova, C.J. Kenneth Tan and Anton Konushin, eds. Springer Berlin Heidelberg, 17-32.

Koholka, Roland, Heinz Mayer and Alois Goller, 1999. MPI-parallelized Radiance on SGI CoW and SMP. *Proceedings of the 4th International ACPC Conference Including Special Tracks on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia: Parallel Computation*, 549-558

Křivánek, Jaroslav and Pascal Gautron, 2009. Practical global illumination with irradiance caching. *Synthesis Lectures on Computer Graphics and Animation*, 4 (1), 1-148.

Larson, Gregory Ward and Robert Shakespeare, 1998. *Rendering with Radiance*. San Francisco: Morgan Kaufmann Publishers, Inc.

Ochoa, Carlos E., Myriam B.C. Aries and Jan L.M. Hensen, 2012. State of the art in lighting simulation for building science: A literature review. *Journal of Building Performance Simulation*, 5 (4), 209-233.

Parker, Steven G., Austin Robison, Martin Stich, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire and Keith Morley, 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29 (4).

Purcell, Timothy J., Ian Buck, William R. Mark and Pat Hanrahan, 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2002*, 21 (3), 703-712.

Wang, Rui, Kun Zhou, Minghao Pan and Hujun Bao, 2009. An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2009*, 28 (3).