Physically Based Global Illumination Calculation Using Graphics Hardware

Nathaniel L Jones[1] and Christoph F Reinhart[1]
[1] Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

This paper examines the feasibility of implementing core algorithms from Radiance using a new type ray tracing engine optimized for highly parallel graphics hardware environments. It presents solutions to a number of implementation challenges. First, the Radiance data format is reinterpreted as a set of buffered data arrays compatible with graphics processing unit (GPU) memory. Second, the ray tracing core of the Radiance RPICT and RTRACE programs for global illumination calculations of scenes and discrete sensors is broken up into a number of small GPU programs that execute in parallel. Third, command-line user settings are declared as variables on the GPU with scopes appropriate to their functions. As a proof of concept, the paper presents a reference implementation using the OptiX™ ray tracing engine that produces images indistinguishable from Radiance up to twenty times faster for scenes with a palette of common materials.

1 Introduction

For the past twenty years, Radiance has stood out as the best-in-class simulation software for physically based rendering and irradiance calculation in building design (Larson & Shakespeare 1998). However, designers and practitioners often cannot realize the full benefits of physically based lighting simulation due to the amount of time these simulations require. In the context of architectural building design, time-consuming processes such as lighting and energy performance simulation tend to occur after-the-fact for use in presentation or for validation against building codes. In contrast, fast simulations, which execute at a pace keeping up with the designer's train of thought, are integrated into the design process. For instance, non-photorealistic rendering of three-dimensional CAD models in real time is now so ubiquitous that architects constantly alter designs in reaction to their appearance in those synthetic images. In order for physically based lighting data from Radiance to have the same effect, building designers need access to it through a real-time ray tracing engine.

Real-time ray tracing can provide designers with data that is useful in early design phases. Daylight autonomy simulation, which predicts the extent of the daylit area within a space, is useful in early design to specify appropriate window-to-wall ratios, affecting both the appearance of a building and its heating and cooling loads (IES Daylight Metrics Committee 2012). Glare analysis, which predicts occupant comfort and usage frequency of movable blinds within a space, has potential in early design to inform decisions on building orientation and the design of shades (Reinhart & Wienold 2011). Ray tracing simulations performed early in the design process may also be useful in designing for outdoor environmental comfort and on-site photovoltaic energy production. In the context of early design decisions, it is not sufficient to request and wait for simulation results, or even to batch multiple simulations to run overnight; data that requires excessive time to obtain is likely to be overlooked. Rather, simulation results need to be produced in real time so that the design process can continue uninterrupted.

It is important here to define real-time simulation. Although the phenomenon of global illumination occurs at the speed of light, duplication of this speed is not possible *in silico*. Instead, the goal in real-time simulation is to provide results at interactive rates such that the time required for computation is similar to the time taken to request it. In interactive computer graphics, where animated frames must flow seamlessly together and each frame's content may be influenced by the user's actions during the previous frames, real-time feedback necessitates rendering times under one sixtieth of a second (Akenine-Möller et al. 2008). For global illumination simulation, the goal is to answer a question posed by the user, so a slower rate of interaction is tolerable. The simulation result must simply be returned quickly enough that the question is still in the user's mind and the user can pose additional questions based on the result. Simulation times on the order of magnitude of one second serve as the goal in this project.

With this ambitious goal in mind, a 2005 estimate proposed that ray tracing speeds must increase by two orders of magnitude over central processing unit (CPU) speeds in order to achieve real-time performance (Woop et al. 2005). CPU speeds have increased since then, but the actual magnitude of the required performance increase may be higher still due to the complexity of models and level of physical accuracy required for global illuminance simulation. The Radiance image in Figure 1 of a typical office space with moderate geometric complexity requires 45 minutes to render on a 3.4 GHz Intel® Core™ i7-4770 processor. Assuming that a one-second delay in returning simulation results is acceptable real-time performance, over three orders of magnitude of ray tracing speed improvement are necessary to make real-time design feedback a reality.

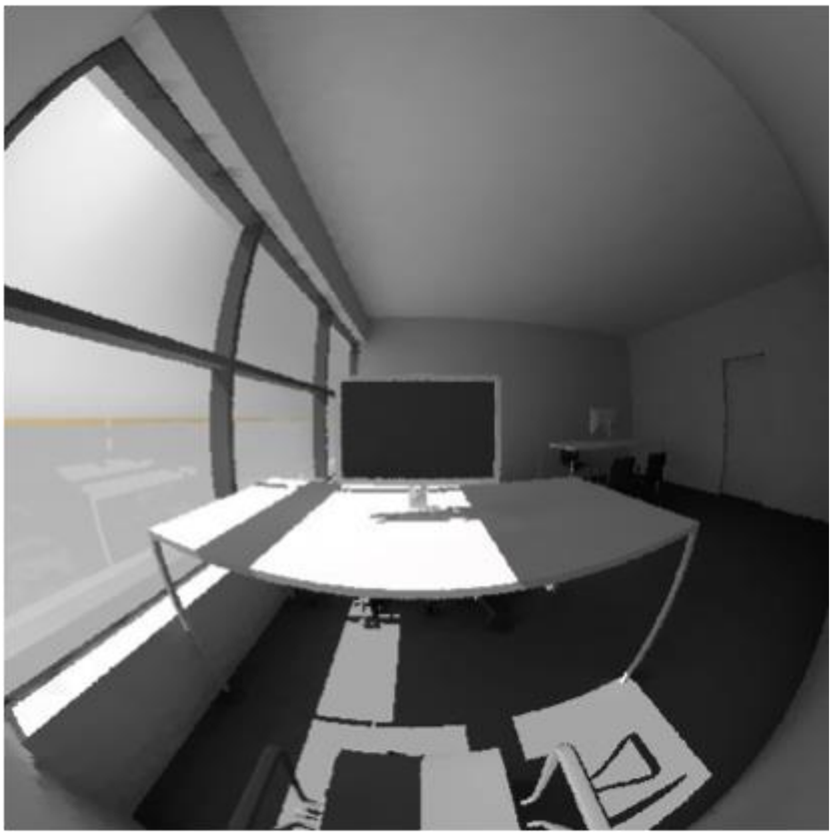

Figure 1: A typical office space rendered with Radiance

The past decade has witnessed remarkable speed increases in scientific computing due to the development of general purpose GPU (GPGPU) architectures for highly parallel tasks.

Within the field of building performance simulation, GPUs have been exploited for computational fluid dynamics for indoor airflow (Zuo & Chen 2010, Wang et al. 2011), room acoustics (Mehra et al. 2012), and incident solar radiation (Jones et al. 2011, Zuo et al. 2014). Ray tracing performed through GPGPU techniques has been implemented for acoustics (Jedrzejewski & Marasek 2006), urban heat islands (Clark 2012, Halverson 2012), and visualization of global illumination results (Andersen et al. 2013). Ray tracing for global illumination is parallel to such an “embarrassing” degree that it should benefit greatly from the parallelism of GPU architectures (Woop et al. 2005). In light of this, it is not without irony that the graphics processor is not used for architectural simulation of global illumination, given that the two share a common origin in computer graphics.

This paper outlines a number of technical challenges facing programmers who attempt to parallelize Radiance. It demonstrates the feasibility of potential solutions using OptiX™, a free GPGPU ray tracing engine created by NVIDIA® (Parker et al. 2010). It shows that the core algorithms of Radiance can be duplicated in OptiX™ 3.0.1 on the GPU and accessed through a modification of the source code for Radiance. This transition from the CPU to GPU provides up to a twenty-fold performance increase, even without additional code optimization. Finally, this paper speculates on how faster global illumination simulation will benefit the design community.

2 Previous Work

Radiance is a set of open source command-line executables that perform ray tracing and related operations (Larson & Shakespeare 1998). Two of the included programs, RPICT and RTRACE, carry out global illumination calculations of scenes and discrete sensors, respectively. These programs date back to the 1980’s, and little about them has changed since they were first described (Ward & Rubinstein 1988). Radiance has been validated by comparison to physical architectural spaces (Grynberg 1989, Ng et al. 2001, Galasiu & Atif 2002) and in controlled environments for modeling daylight illuminance distributions (Mardaljevic 1995, Reinhart & Herkel 2000, Mardaljevic 2001, Reinhart & Walkenhorst 2001) and material properties (Reinhart & Andersen 2006). This extensive testing has reinforced its use as a back-end by building performance simulation tools such as IES<VE>, Ecotect®, OpenStudio, Adeline, Desktop Radiance, DAYSIM, and DIVA for Rhino.

The ray tracing algorithm used by Radiance is similar to that of all backward ray tracing engines. Primary rays originating from a common point (the camera or sensor) are traced until they intersect a surface contained in a geometric hierarchical structure (e.g. an octree in Radiance). At each intersection, the surface material determines an action to take, which generally involves spawning more rays to sample transmitted and reflected direct and diffuse light intensities. Upon completion of these actions, a result value (usually a color) is assigned to the initial ray and returned as the result of the command that spawned that ray. The collection or sum of the primary ray results produces the image or sensor reading, respectively. Conceptually, the computation for each primary ray will be similar to but entirely independent of adjacent primary rays, so the algorithm should be easily parallelizable (Woop et al. 2005).

While the programs in Radiance are written primarily for serial execution, they have a limited ability to run in parallel on UNIX-based systems. A single RTRACE instance can use multiple CPU cores if given certain command-line arguments, and multiple instances of RPICT and RTRACE on separate machines may share access to the same scene data and irradiance cache using network file locks (Larson & Shakespeare 1998). While these core programs have not been made available in GPU versions, a Radiance utility program for three-phase method calculation of complex fenestration properties has been implemented on the GPU (Zuo et al. 2014).

Although the building performance community continues to rely on Radiance and its bespoke ray tracing engine, much work has been done within the computer graphics community to develop GPU-based ray tracing engines for general use. These advances have closely paralleled developments in the programmable rasterization pipeline used for non-photorealistic rendering. 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 parallels OpenGL® in structure, including user-programmable shaders to execute at ray intersections; however, their specification was not widely implemented by hardware manufacturers. Because early programmable GPUs lacked the capacity for complex control flow involved in branching and recursion, many instead looked to the development of independent ray processing unit (RPU) hardware to speed up ray tracing (Woop et al. 2005). Such hardware was closely modeled on existing GPUs but added special processing units for ray traversal. Ultimately though, separate hardware became unnecessary as the GPU's ability was expanded to allow the use of GPGPU language extensions such as Compute Unified Device Architecture (CUDA™) developed by NVIDIA® (Aila & Laine 2009, Wang et al. 2009). The OptiX™ ray tracing engine uses CUDA™ to perform both ray traversal and shading on the GPU.

Ray tracing engines can take advantage of two methods of parallelism built into GPU hardware. The more basic method is to use built-in vector types. These data structures allow the GPU to store and manipulate sets of two, three, or four integer or floating point numbers. Manipulation of these vectors is done with a single command as opposed to a loop over the vector elements. In ray tracing, points, directions, and colors can all be stored as `float3` vectors. At a higher level, parallelism is also introduced through the GPU's single-instruction, multiple-thread (SIMT) architecture. Threads are organized into collections of 32 threads called warps. Multiple warps may execute simultaneously and independently of one another, but each warp executes a single instruction at a time on all of its active threads. While the SIMT model allows for divergent execution among threads within a warp, better performance is achievable when threads do not diverge (NVIDIA 2012). For optimal performance in a ray tracer, the rays traced by each warp should be coherent, intersecting similar materials and spawning similar numbers of rays. These forms of parallelism allow today's GPUs to perform ray tracing at faster speeds than would be possible on CPUs.

3 Design Decisions

OptiX™ is a ray tracing engine in the sense that it provides a mechanism for traversing rays to detect intersections with surface geometry. The definition of the geometry, the actions to take upon intersecting any material, and the result data structure to be returned as the payload of a ray are all design decisions left up to the programmer. With this flexibility, OptiX™ may be used as a replacement for another ray tracing engine in existing source codes. The programmer must implement several alterations to the existing program in order to accomplish this (Parker et al. 2010).

- The scene geometry and materials, which would normally be stored in a hierarchical acceleration structure (e.g. an octree), must be copied to GPU memory. Similarly, the results from the primary rays (e.g. their high dynamic range (HDR) RGB values) must be copied from GPU memory back into the program's memory.
- The portions of the program responsible for detecting and reacting to ray intersections must be rewritten as shader programs in CUDA™. Ideally, these portions should be broken up so as to maximize coherence between threads executed as part of the same warp. OptiX™ provides eight types of programs that may be implemented, of which the relevant program types are described below.

- Finally, parameters that affect the behavior of the program when intersections are detected must be transferred to the GPU. In Radiance, numerous command-line arguments are used to establish a trade-off between simulation speed and accuracy. These parameters change the behavior of the shader programs and necessarily cause inefficiency as their values are not known until runtime.

Data Preparation

Radiance uses a unique internal data structure for both geometric and material data that closely mirrors its text-based input file format. All the elements making up a scene are stored together in a single octree, which may be saved as a binary file. However, OpitX™ does not use octrees and instead creates a bounding volume hierarchy (BVH) internally to facilitate faster ray traversal. As a result, a two-pass reading of the internal octree is necessary to transfer geometry and materials to the GPU. The first pass counts the number of each type of object (surface, material, light, etc.) in order to determine the size of the buffers necessary to store each one on the GPU. Because the GPU prefers geometry to be defined as triangles, this pass must also count the number of vertices belonging to each surface and adjust the size of the geometry buffers accordingly. The current implementation requires that all polygons be convex, but future implementations could perform ear clipping to handle concave polygons.

Once the first pass is complete and the buffers are created, a second pass copies each object from the octree into the appropriate buffer or buffers based on its type. Surfaces are used to populate the contents of vertex, normal, and texture coordinate buffers, and the index of the associated material for each surface is placed into a material index buffer. Material objects are treated as instances of material shaders. Light sources, including the sun and sky, are copied into specialized data structures. While Radiance allows numerous user-defined functions to be written, at present only two of these, “skybright” for the CIE sky model (Commission Internationale de l'Eclairage 1973, Matsuura & Iwata 1990) and “perezlum” for the Perez All-Weather Sky Model (Perez et al. 1993), are implemented as data structures that may be buffered to the GPU because they appear regularly in many Radiance scenes and are important to daylighting calculations. For RTRACE, one additional buffer is created containing the input ray origins and directions.

After the completion of the second pass, the OptiX™ kernel is compiled and launched. This prompts construction of the BVH on the GPU followed by a call to the ray generation program, which populates an output buffer. For RPICT, this output contains floating point RGB values scaled as metric radiance values. These values are copied back into the Radiance data structure so that they may be output as a HDR image or a sensor value. For RTRACE, the output buffer contains radiance values and other ray data and metadata that may be requested by the “-o” command-line argument.

Several inefficiencies exist within this prototype implementation. Because the octree structure is never used and its creation is time-consuming, a more efficient approach might be to read scene data in directly without ever creating the octree. This step could be combined with the first pass over the geometry so as to significantly reduce the work involved in data preparation. Similarly, the RPICT output buffer could be used directly to create an HDR image without first being copied into the Radiance data structure from which it is read out of order. However, since the present work is involved only in replacing Radiance’s bespoke ray tracing engine with OptiX™, these optimizations have not been attempted.

Shaders

This section describes the parallels between the OptiX™ program types and components of Radiance. These parallels enable decision making about how to effectively distribute Radiance functionality between OptiX™ programs. For details on the programs themselves, see Parker et al. (2010).

- **Bounding Box Program:** Before any ray tracing occurs, the creation of a BVH requires that each geometric primitive (i.e. triangle) be assigned a conservative bounding volume, i.e., a three-dimensional box guaranteed to enclose the primitive. This operation can be performed in parallel for all primitives by finding the minimum and maximum *x*-, *y*- and *z*-coordinates of each one. Creation of the BVH itself may or may not happen in parallel, depending on the method used. Acceleration structures that allow faster traversal typically take longer to build (Parker et al. 2010). While this program is similar in purpose to Radiance's octree creation, its operation is quite different and is mostly automated by OptiX™, so it does not borrow any code from Radiance.
- **Ray Generation Program:** This program is called once for the generation of each primary ray, and may be run in parallel in up to as many instances as there are GPU cores and available GPU thread memory. In RPICT, it duplicates the `viewray()` method to define the origin and direction of a single primary ray, then spawns that ray, and upon completion of the ray's processing, copies the color from the ray's payload to the output buffer. In RTRACE, it responds to a single origin and direction input pair, and may spawn multiple rays before summing their results if used to calculate irradiance rather than radiance.
- **Intersection Program:** During ray traversal, this program is called each time the ray intersects a surface until the ray encounters a surface that terminates it or until no more surfaces are found in its path. In Radiance, rays are generally terminated by the first surface they hit. While customization of the program can allow it to work with non-planar objects, modified behavior is unnecessary for Radiance. Instead, the job of this program is mainly to determine the type of material that was hit by referring to the geometric primitive's material index and call the corresponding closest hit program. The current implementation also determines the normal direction and texture coordinates of the surface at the intersection point. Allowing these to vary provides for future implementation of bump maps (called texdatas in Radiance) and texture maps (called colordatas and brightdatas in Radiance).
- **Closest Hit Program:** This program defines the action to take when a ray intersects a surface, including the spawning of new rays and the creation of a payload for the incident ray. The program can be defined multiple times within an OptiX™ context, once for each combination of material type and ray type. Furthermore, multiple instances of each definition may be created to allow multiple materials of the same type. In Radiance, this program type is equivalent to the methods `m_normal()` (for intersections with plastic, metal, and trans materials), `m_glass()` (for intersections with glass materials), `m_light()` (for intersections with lights), and other functions that follow the naming convention `m_<type>()`. In order to guarantee that the results produced by OptiX™ are as similar as possible to those produced by Radiance, the current implementation follows the original source code of these functions as closely as possible with little optimization for the GPU other than the use of built-in vector types. Currently, only plastic, metal, translucent, glass, and light materials are supported.
- **Miss Program:** When a ray does not intersect any surface, this program is called to assign it a payload. In typical rendering, a value is provided from a

background image, but Radiance defines a unique object type, "source," which is located infinitely far from the ray origin and is defined by a solid angle rather than by geometric coordinates. The current implementation uses a miss program to add the effects of three types of sources: those of uniform brightness (e.g. the sun), those defined by a "skybright" function, and those defined by a "perezlum" function.

- **Exception Program:** Radiance makes the user aware of errors by printing messages to the standard error stream. However, this text stream which usually appears in the command line is not directly accessible from the GPU, so an alternate method is necessary to make the user aware of errors. When an issue arises that prevents normal execution of the ray tracing kernel, the exception program inserts a color-coded error value into the output buffer for that thread. In RPICT, the error appears in the HDR image as a saturated pixel in the position of the corresponding primary ray. An advantage of this method for error reporting is that even when several rays fail, the majority of the HDR image is still likely to be produced correctly.

Command-Line Arguments

Because Radiance is distributed as a set of command-line executables, it depends heavily on arguments passed to it through the command line to determine its behavior. This means that the exact behavior of the ray tracing kernel is not determined until the program starts. For instance, by providing the "-i" argument to either PRICT or RTRACE, the user can instruct the programs to calculate irradiance rather than radiance. OptiX™ uses a just-in-time compiler to refine its assembly-language code for available hardware before launching the kernel, so the governing program could use the input arguments to choose between multiple versions of each OptiX™ program. While this method would result in more compact shader programs with less potential for divergence between parallel threads, it also requires significant duplication of code which is not practical during prototype development. Instead, the current implementation creates GPU variables to hold the values of command-line arguments.

In OptiX™, the scope of an argument can be global or limited to certain programs. RPICT camera arguments (those starting with "-v" or "-p") are visible only to the ray generation program, while sampling parameters (those starting with "-d", "-a", "-l", or "-s"), which are used by many material programs, are globally visible. Material parameters, which derive from data in the octree rather than from command-line arguments, are also passed to the GPU as variables but are visible only from within the relevant closest hit program.

4 Results

The performance improvement achieved using the GPU can be measured by comparing the computation times of the standard RPICT and RTRACE programs with those of modified code using the OptiX™ ray tracing engine. These tests were carried out on the moderate complexity scene shown in Figure 1, which contains 278,695 triangles and 11 distinct materials. Identical input arguments were used for both the standard and modified programs, and both were compiled from source code with identical compiler settings. Tests were run on two workstations. The first, with a 3.4 GHz Intel® Core™ i7-4770 processor and an NVIDIA® Quadro® K4000 graphics card with 768 CUDA™ cores, is representative of mid-range workstations. The second, with a 2.27 GHz Intel® Xeon® E5520 processor and an NVIDIA® Tesla® K40 graphics accelerator with 2880 CUDA™ cores, is representative of high-end workstations and servers. The standard CPU implementations of RPICT and RTRACE were tested only on the first workstation with the faster processor.

While the standard versions run on single CPU threads, the OptiX™ implementations split their time between the CPU and GPU. Therefore, two times are reported for the OptiX™ results – one representing only time spent in parallel computation on the GPU, and the other including the overhead of CPU operations such as data buffer creation. The time required for initial setup operations such loading the octree file was a constant overhead for both the standard and OptiX™ implementations and is not included in the timings.

RPICT

Figure 2 shows the scene from Figure 1 rendered with Radiance's RPICT command and with the current implementation that uses the OptiX™ ray tracing engine. For this single 512 × 512 pixel image, the OptiX™ implementation performs more than four times faster than standard RPICT on the Quadro® K4000 and seven times faster on the Tesla® K40 (Table 1).

Table 1: RPICT Rendering Times for a Single Image

Version	GPU Time (seconds)	Total Time (seconds)	Improvement (percent)
Standard		91.8	
OptiX™ on Quadro® K4000	18.7	19.9	78.3
OptiX™ on Tesla® K40	10.3	12.4	86.5

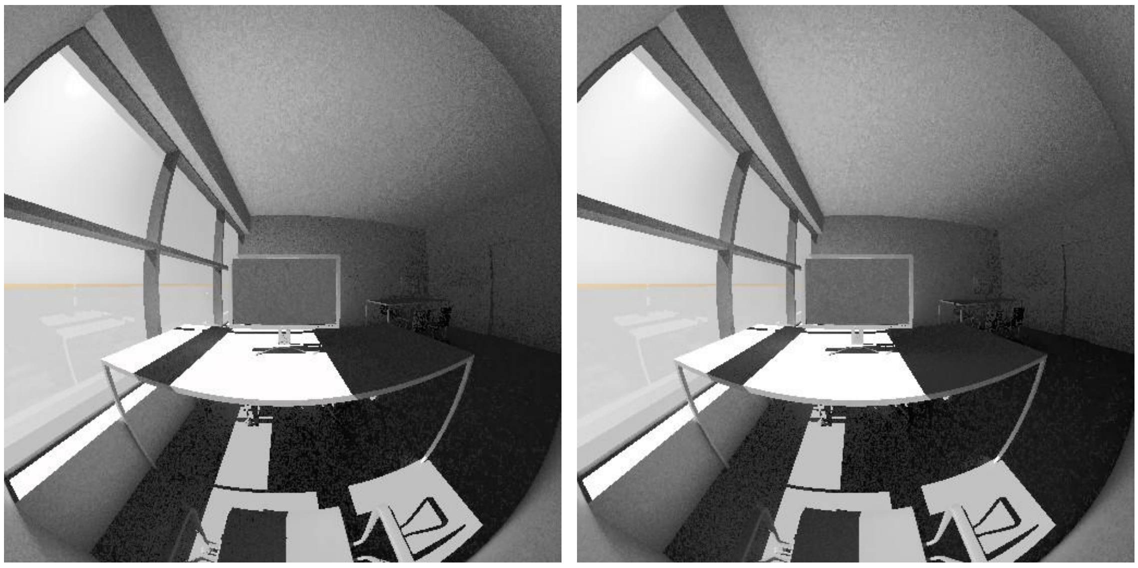

Figure 2: The scene rendered with OptiX™ (left) and standard RPICT (right)

A single OptiX™ context, once defined, can be used repeatedly for ray tracing while the program is running. Geometry need only be copied to GPU memory once and can be altered if necessary between launches of the OptiX™ kernel. This is useful for glare analysis, where multiple camera positions and directions are used within the same scene. Following the model of Jakubiec & Reinhart (2012), a view file was created instructing RPICT to render 360° of rotational views of the office from Figure 1 at 3° increments (Figure 3). While standard RPICT takes nearly four hours to complete this task, OptiX™ implementation is over five times faster on the Quadro® K4000 and seventeen times faster on the Tesla® K40 (Table 2). Furthermore, the marginal CPU overhead for additional images is minimal; the CPU utilization to render 120 images was only six times that for a single image.

Table 2: RPICT Rendering Times for 120 Images

Version	GPU Time (seconds)	Total Time (seconds)	Improvement (percent)
Standard		13,492	
OptiX™ on Quadro® K4000	2388	2395	82.2
OptiX™ on Tesla® K40	779	790	94.1

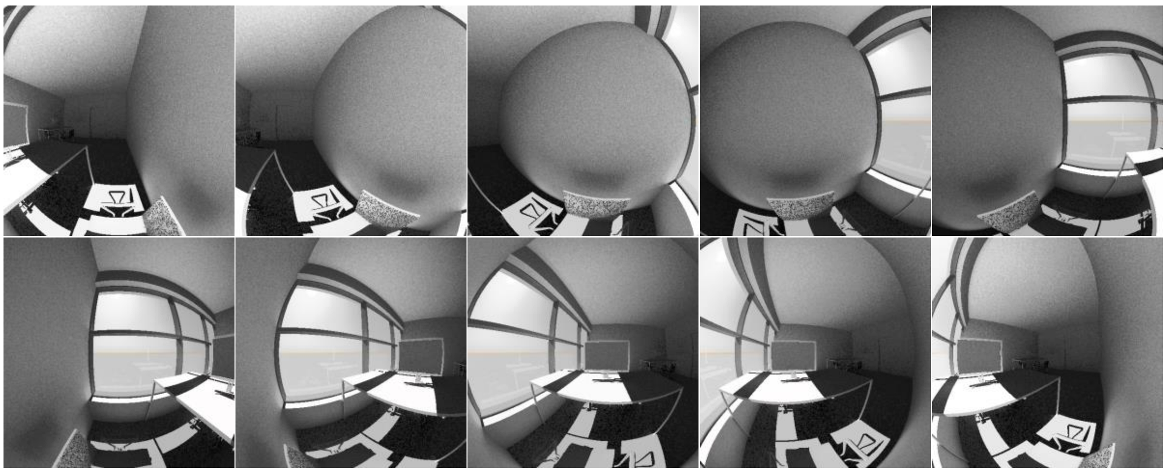

Figure 3: Select rotational views rendered with OptiX™

RTRACE

While RPICT demonstrates the coarse level of scalability achievable through rendering image sets, RTRACE shows finer scalability achievable by tracing large numbers of rays. The numeric output from RTRACE also demonstrates the fidelity with which the OptiX™ implantation reproduces Radiance results.

In this test, the scene from Figure 1 was again used, and rays were cast from the camera position toward the ceiling to sample radiance values. In order to increase coherence within warps, the same origin and direction were used for all samples, though stochastic processes within the Radiance algorithms produce unique results for each sample. While standard RTRACE timings increase linearly with the number of primary rays at a rate of 1.1×10^{-3} seconds per sample, the OptiX™ implementation's computation time is relatively constant for low numbers of rays (Figure 4). The OptiX™ implantation outperforms standard RTRACE for simulations that take over 4 seconds on the CPU. For large numbers of rays, the OptiX™ implementation approaches a rate of 1.3×10^{-4} seconds per sample on the Quadro® K4000, which represents an 89% marginal speed improvement over standard RTRACE. The Tesla® K40 performs even faster at 4.8×10^{-5} seconds per sample, giving it a 96% marginal speed improvement over standard RTRACE. While this scenario is admittedly constructed to achieve greater-than-usual coherence within warps by repeatedly tracing the same ray, it demonstrates that the OptiX™ implementation can run more than twenty times faster than Radiance under certain conditions.

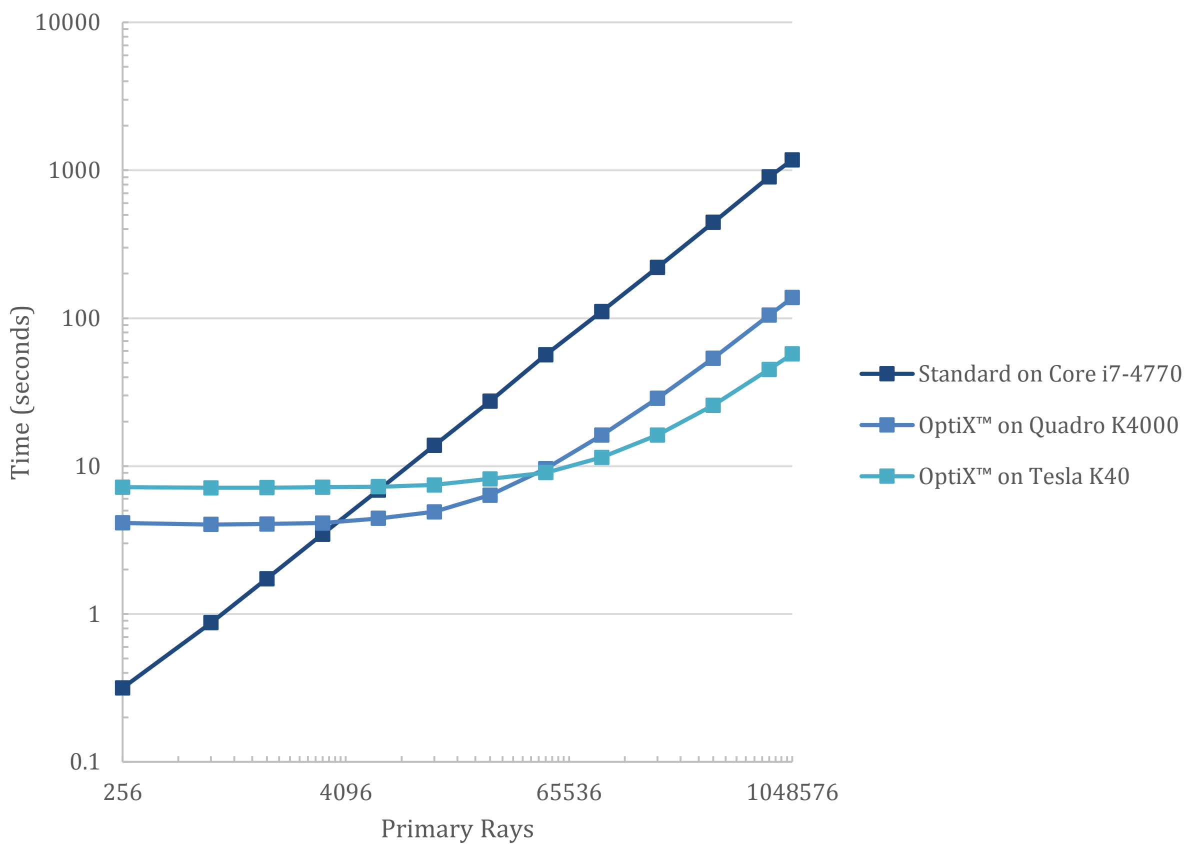

Figure 4: RTRACE Computation Times

The timings achieved by maximizing coherence within warps should approach the theoretical maximum speed of the ray tracing engine. This speed limit is dependent both on the scene and the hardware used. However, random jitter applied to secondary rays causes divergence within the warps, which both reduces efficiency and improves sample or image quality. The level of randomness produced in the OptiX™ implementation using the cuRAND library from NVIDIA® closely resembles that from standard RTRACE (Table 3). In the largest tests, which traced over a million primary rays, the averaged results of the two versions differ by less than one percent, and the population standard deviations of the two sets of output are nearly identical. This indicates that the OptiX™ implementation produces results with the same accuracy and reproducibility as Radiance.

Table 3: RTRACE Results from 1,048,576 Samples

Version	Average (W•sr^{-1}•m^{-2})	Range (W•sr^{-1}•m^{-2})	Std. Dev. (W•sr^{-1}•m^{-2})
OptiX™	2.569	1.31 – 3.90	0.276
Standard	2.576	1.35 – 3.95	0.276

5 Future Work

The preliminary results presented above demonstrate that certain basic Radiance simulations can be duplicated using OptiX™. However, significant work remains before OptiX™ can be used for accurate architectural lighting simulation. Additionally, while a twenty-fold speed increase is a step toward the goal of real-time simulation feedback, significant code optimization remains possible to improve the performance of Radiance algorithms implemented on the GPU.

The highest priority is the implementation of irradiance caching on the GPU. This strategy allows further speed-up of the Radiance algorithms by selectively reusing ambient values from previous calculations. Unfortunately, the serial approach used by Radiance, in which the irradiance cache is both written to and read from by a single thread, is simply not practical in multithreaded environments (Wang et al. 2009). Various solutions to this problem have been proposed, including multi-CPU approaches (Dubla et al. 2009) and GPU preprocessing methods such as photon mapping (Wang et al. 2009), irradiance splatting (Křivánek & Gautron 2009), and multi-pass construction (Frolov et al. 2013).

The second priority, code optimization, could result in significant performance improvements for the OptiX™ implementations. While programs on the CPU tend not to be limited by memory availability, GPU programs create hundreds or thousands of threads which each require dedicated memory for local variables. The complex control structure and branching of the Radiance code is unfriendly to SIMT architecture and requires a large amount of stack space per GPU thread. By dividing large methods into smaller functions and replacing complex shaders with simple shaders for specific material types and command-line parameters, further significant increases in calculation speed may be achieved.

6 Conclusion

Physically based global illumination simulation is an important but underutilized tool in architectural design. GPUs have great potential to speed up this type of simulation, making it easy to inform design decisions with global illumination data. Initial OptiX™ implementations of the Radiance RPICT and RTRACE programs are five to twenty times faster than the standard CPU implementations, and further speed improvements are likely through code optimization. These speed improvements are scalable, especially benefiting simulations that produce large numbers of rays or large sets of images. This has obvious benefits for annual simulations, where geometry and camera positions are static and only the sun and sky change, as well as for glare analysis, where multiple camera directions are used within the same scene (Jakubiec & Reinhart 2012). While the demonstrated performance improvements are immediately useful, a good deal of work remains in order to reach the three-order-of-magnitude speed increase necessary to provide accurate global illumination results to building designers in real time.

Instantly-available global illumination simulation results could have major effects on architectural design practice. First, when simulation results can be produced quickly, they should be requested earlier and more often by designers. This will also result in an increased need for architects to be trained in understanding the relationship between numerical or pictorial illumination data and the experiential qualities of light in physical spaces. Second, faster simulations will make users more likely to render full scenes with RPICT in addition to simulating sensors with RTRACE. Rendered images will make users aware of modeling mistakes and inaccuracies that generally go unnoticed in numerical output. This visual feedback, combined with a reduced time cost for repeating simulations, should spur users to put more effort into creating geometrically accurate simulation models with physically accurate material characteristics. Faster simulations will also reduce the additional time necessary to render scenes with more complex geometry and materials. Third, when simulation feedback is instantaneous and minimal effort is necessary to start a simulation, the results can be generated by designers concurrently with the designs. Designers can then react to simulation results and base design changes on them when otherwise problems might go unnoticed and unsolved. In the long term, simulation packages similar to Radiance might be integrated directly with design software much the way that non-photorealistic rendering has been in the past. Designers may then take for granted the feedback that today is still relatively difficult to retrieve.

7 Acknowledgements

This research was funded through the Kuwait-MIT Center for Natural Resources and the Environment by the Kuwait Foundation for the Advancement of Sciences. The Tesla K40 used for this research was donated by the NVIDIA Corporation. J. Alstan Jakubiec provided the model and Radiance input files used for timing and illustration. The authors wish to thank Frédo Durand for his consultation on GPU programming and Philip Thompson for providing technical support.

8 References

Aila, T. & Laine, S., 2009. Understanding the efficientcy of ray traversal on GPUs. *Proceedings of High-Performance Graphics 2009*, pp. 145-149.

Akenine-Möller, T., Haines, E. & Hoffman, N., 2008. *Real-Time Rendering*. 3rd ed. Natick: A K Peters, Ltd.

Andersen, M., Guillemin, A., Amundadottir, M. L. & Rockcastle, S., 2013. Beyond illumination: An interactive simulation framework for non-visual and perceptual aspects of daylighting performance. *Proceedings of BS2013: 13th Conference of International Building Performance Simulation Association, Chambéry, France, August 26-28*, pp. 2749-2756.

Clark, J. G., 2012. *A Fast and Efficient Simulation Framework for Modeling Heat Transport.* Master's Thesis. University of Minnesota.

Commission Internationale de l'Eclairage, 1973. *CIE 22-1973 Standardisation of Luminance Distribution on Clear Skies*, Paris: Bureau Central CIE.

Deitrich, A., Wald, I., Benthin, C. & Slusallek, P., 2003. The OpenRT application programming interface - towards a common API for interactive ray tracing. *Proceedings of the 2003 OpenSG Symposium*, pp. 23-31.

Dubla, P., Debattista, K., Santos, L. P. & Chalmers, A., 2009. Wait-free shared-memory irradiance cache. *Proceedings of the 9th Eurographics Symposium on Parallel Graphics and Visualization*, pp. 57-64.

Frolov, V., Vostryakov, K., Kharlamov, A. & Galaktionov, V., 2013. Implementing irradiance cache in a GPU photorealistic renderer. In: M. L. Gavrilova, C. K. Tan & A. Konushin, eds. *Transactions on Computational Science XIX*. Berlin: Springer, pp. 17-32.

Galasiu, A. D. & Atif, M. R., 2002. Applicability of daylighting computer modeling in real case studies: comparison between measured and simulated daylight availability and lighting consumption. *Building and Environment*, 37(4), pp. 363-377.

Grynberg, A., 1989. *Validation of Radiance*, Berkeley, CA: Lawrence Berkeley Laboratories. Document ID 1575.

Halverson, S., 2012. *Energy Transfer Ray Tracing with OptiX*. Master's Thesis. University of Minnesota.

IES Daylight Metrics Committee, 2012. *IES LM-83-12: Approved Method: IES Spatial Daylight Autonomy (sDA) and Annual Sunlight Exposure (ASE)*, New York: Illuminating Engineering Society of North America.

Jakubiec, J. A. & Reinhart, C. F., 2012. The 'adaptive zone' - A concept for assessing discomfort glare throughout daylit spaces. *Lighting Research and Technology,* 44(2), pp. 149-170.

Jedrzejewski, M. & Marasek, K., 2006. Computation of room acoustics sing programmable video hardware. In: K. Wojciechowski, et al. eds. *Computer Vision and Graphics: International Conference, ICCVG 2004, Warsaw, Poland, September 2004, Proceedings.* Dordrecht: Springer, pp. 587-592.

Jones, N. L., Greenberg, D. P. & Pratt, K. B., 2011. Fast computer graphics techniques for calculating direct solar radiation on complex building surfaces. *Journal of Building Performance Simulation,* 5(5), pp. 300-312.

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

Larson, G. W. & Shakespeare, R., 1998. *Rendering with Radiance: The Art and Science of Lighting Visualization.* San Francisco: Morgan Kaufmann Publishers, Inc.

Mardaljevic, J., 1995. Validation of a lighting simulation program under real sky conditions. *Lighting Research and Technology,* 27(4), pp. 181-188.

Mardaljevic, J., 2001. The BRE-IDMP dataset: a new benchmark for the validation of illuminance prediction techniques. *Lighting Research and Technology,* 33(2), pp. 117-136.

Matsuura, K. & Iwata, T., 1990. A model of daylight source for the daylight illuminance calculations on the all weather conditions. In: A. Spiridonov, ed. *Proceedings of Third International Daylighting Conference, Moscow.* Moscow: NIISF.

Mehra, R. et al., 2012. An efficient GPU-based time domain solver for the acoustic wave equation. *Applied Acoustics,* 73(2), pp. 83-94.

Ng, E. Y.-Y., Poh, L. K., Wei, W. & Nagakura, T., 2001. Advanced lighting simulation in architectural design in the tropics. *Automation in Construction,* 10(3), pp. 365-379.

NVIDIA, 2012. *CUDA C Programming Guide,* PG-02829-001_v5.0, October 2012.

Parker, S. G. et al., 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2010,* 29(4).

Perez, R., Seals, R. & Michalsky, J., 1993. All-weather model for sky luminance distribution—Preliminary configuration and validation. *Solar Energy,* 50(3), pp. 235-245.

Purcell, T. J., Buck, I., Mark, W. R. & Hanrahan, P., 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2002,* 21(3), pp. 703-712.

Reinhart, C. F. & Andersen, M., 2006. Development and validation of a radiance model for a translucent panel. *Energy and Buildings,* 38(7), pp. 890-904.

Reinhart, C. F. & Herkel, S., 2000. The simulation of annual daylight illuminance distributions - a state-of-the-art comparison of six RADIANCE-based methods. *Energy and Buildings,* 32(2), pp. 167-187.

Reinhart, C. F. & Walkenhorst, O., 2001. Validation of dynamic RADIANCE-based daylight simulations for a test office with external blinds. *Energy and Buildings,* 33(7), pp. 683-697.

Reinhart, C. F. & Wienold, J., 2011. The daylighting dashboard – A simulation-based design analysis for daylit spaces. *Building and Environment,* 46(2), pp. 386-396.

Wang, R., Zhou, K., Pan, M. & Bao, H., 2009. An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2009,* 28(3).

Wang, Y., Malkawi, A. & Yi, Y., 2011. Implementing CFD (computational fluid dynamics) in OpenCL for building simulation. *Proceedings of Building Simulation 2011: 12th Conference of International Building Performance Simulation Association, Sydney, 14-16 November,* pp. 1430-1437.

Ward, G. J. & Rubinstein, F. M., 1988. A new technique for computer simulation of illuminated spaces. *Journal of the Illuminating Engineering Society,* 17(1), pp. 80-91.

Woop, S., Schmittler, J. & Slusallek, P., 2005. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2005,* 24(3), pp. 434-444.

Zuo, W. & Chen, Q., 2010. Fast and informative flow simulations in a building by using fast fluid dynamics model on graphics processing unit. *Building and Environment,* 45(3), pp. 747-757.

Zuo, W., McNeil, A., Wetter, M. & Lee, E. S., 2014. Acceleration of the matrix multiplication of Radiance three phase daylighting simulations with parallel computing on heterogeneous hardware of personal computer. *Journal of Building Performance Simulation,* 7(2), pp. 152-163.