Optical Engineering

SPIEDigitalLibrary.org/oe

Open-source graphics processing unit–accelerated ray tracer for optical simulation

Florian Mauch
Marc Gronle
Wolfram Lyda
Wolfgang Osten

SPIE

Open-source graphics processing unit–accelerated ray tracer for optical simulation

Florian Mauch
Marc Gronle
Wolfram Lyda
Wolfgang Osten
University Stuttgart
Institut fürTechnische Optik
Stuttgart Research Centre of Photonic Engineering
Pfaffenwaldring 9, 70569 Stuttgart, Germany
E-mail: mauch@ito.uni-stuttgart.de

Abstract. Ray tracing still is the workhorse in optical design and simulation. Its basic principle, propagating light as a set of mutually independent rays, implies a linear dependency of the computational effort and the number of rays involved in the problem. At the same time, the mutual independence of the light rays bears a huge potential for parallelization of the computational load. This potential has recently been recognized in the visualization community, where graphics processing unit (GPU)-accelerated ray tracing is used to render photorealistic images. However, precision requirements in optical simulation are substantially higher than in visualization, and therefore performance results known from visualization cannot be expected to transfer to optical simulation one-to-one. In this contribution, we present an open-source implementation of a GPU-accelerated ray tracer, based on nVidias acceleration engine OptiX, that traces in double precision and exploits the massively parallel architecture of modern graphics cards. We compare its performance to a CPU-based tracer that has been developed in parallel. [DOI: 10.1117/1.OE.52.5.053004]

Subject terms: ray tracing; graphics processing unit-accelerated computing; optical simulation.

Paper 130271 received Feb. 19, 2013; revised manuscript received Apr. 8, 2013; accepted for publication Apr. 18, 2013; published online May 9, 2013.

1 Introduction

Ray tracing has been the most important tool to optical designers ever since Ernst Abbe and Carl Zeiss started to systematically design optical microscopes. It is ultimately based on the Eikonal equation as a short-wavelength approximation to the scalar Helmholtz equation.[1] In piecewise homogeneous media, it results in a repetitive calculation of intersections of lines (i.e., rays) with the surfaces separating the volumes of homogeneous refractive index (i.e., the optical elements) as well as evaluation of Snell's law. Even though these calculations initially were done manually, soon enough computer programs were developed that sped up the process substantially.[2] Today a vast number of highly developed software packages for ray tracing are commercially available (see Ref. 3, for example, for a recent and certainly incomplete listing). Despite the dramatic increase in computer performance over recent years, ray tracing remains a time-consuming task for optical designers and engineers. Especially, stray light analysis of complex optical systems as well as design and evaluation of illumination systems rely on Monte Carlo techniques, whose signal-to-noise ratio is inversely proportional to the square root of the number of rays reaching the detector. These applications are still limited by computation speed, even on today's machines.[4] Furthermore, innovative applications of ray tracing in current research topics often use a huge number of rays (e.g., Refs. 5 and 6). Such applications typically need additional data, e.g., optical pathlengths or local wavefront curvatures, to be carried by the rays. Such extensions to the classic ray tracing procedure demand access to the basic algorithms of the tracing engine, which is usually not available for commercial software. Therefore, these applications are often realized with custom-built tracing engines written in script languages (e.g., Refs. 5 and 7), which generally do not perform well for iterative procedures such as ray tracing.

At the same time, ray tracing attracted a lot of attention in the visualization community, where it is used to render photorealistic images for video games and special effects in movies. It was mainly this community that realized the prospects of massively parallel computing architectures of modern graphics processing units (GPU) for accelerating ray tracing.[8] It is presumably thanks to the marketing prospects in this area that nVidia introduced a ray-tracing application acceleration engine called OptiX[9] at SIGGRAPH 2009 in New Orleans. It provides an intuitive C-style access to the parallel computing architecture of nVidia's latest graphics cards, which is specifically designed to enable quick development of ray-tracing applications. It is therefore the intent of this contribution to present an openly accessible, GPU-accelerated ray-tracing software based on OptiX that is dedicated to optical simulation and design, and to compare its performance to CPU ray tracing.

2 GPU-Accelerated Ray Tracer

While a central processing unit (CPU) is optimized for fast execution of wildly branching programs, a GPU is especially designed for so-called data parallel computations, i.e., problems in which the same instructions are executed on many data elements. Therefore, in a GPU, many arithmetic logic units (ALU) are grouped together in a single instruction, multiple data (SIMD) architecture. All of these ALUs perform the same operation as decoded by the shared control logic on its locally loaded data elements simultaneously,

i.e., in parallel. Details about GPU architectures and the implications on general purpose programming can be found in Ref. 10, for example. For the actual application of ray tracing, it has been pointed out that its basic principle, i.e., propagating a set of mutually independent rays through an optical scene, fits perfectly with this kind of parallel architecture.[8]

One problem of GPU computing is that the performance of algorithms strongly depends on the actual hardware at hand. Therefore, programmers have to break down the computation task into smaller parts that can be efficiently distributed among the computational units available on the GPU at hand. To have a chance of automatic performance scaling among different models of current GPUs as well as with upcoming GPU hardware, we chose to use the OptiX acceleration engine of nVidia to implement our ray tracer. OptiX is not a ray tracer itself, providing no algorithms for calculations of intersections or similar things at all. Instead, it is a simple abstract model of a ray tracer, providing a framework for building a ray-tracing application and relieving the programmer from dealing with the details of the actual hardware. In fact OptiX roughly defines a ray tracer via a set of three functions:[11] (1) a so-called ray-generation function serves as an entry point into the ray tracing application and generates rays, starts the actual tracing, and processes the rays after the trace; (2) the intersection function calculates the distance of a given ray to a given surface; and (3) the hit function encapsulates the interaction of a ray with a surface that is hit, i.e., roughly the functionality of a material in classic optical-design ray tracers. For example, the hit function of a surface enclosing a simple refracting material will calculate Snell's law and change the direction of an incoming ray accordingly.

OptiX merely defines these functions in its abstract model. The actual implementation has to be provided entirely by the developer in compliance with CUDA-C,[10] i.e., nVidia's extension to the C-standard. Additionally, OptiX defines a hierarchy to organize the functions representing the scene that is to be simulated and, most importantly, a data structure representing the ray that is to be traced through the scene. As OptiX has been designed for the visualization community, the variables defining the origin and the direction of these rays are 32-bit single-precision floating-point variables. This precision is commonly not sufficient for scientific optical simulations, especially if wavefront properties are to be modeled. Fortunately OptiX also offers the possibility to attach a user-defined data structure to each ray. Therefore, we use this structure to pass around 64-bit double-precision floating-point variables defining the origin and direction of the ray. This way, we can trace the rays with an accuracy adequate for optical simulation and stay within the OptiX framework. To be able to compare the principle performance of the GPU acceleration as objectively as possible, we defined the algorithms of the intersect and hit functions, i.e., the main workload of the tracing, as ANSI-C-style inline functions. This way, we can compile exactly the same algorithms into the GPU code and a CPU code that was developed in parallel.

The sources of the ray tracer are released under a GPL license.[12] Even though the code should be platform independent in principle, we did develop it in Microsoft Visual Studio 2010 for 64-bit Microsoft Windows. The software can be used as a stand-alone command line program or in conjunction with a graphical user interface (GUI). In the latter case, the tracer is statically linked to a plugin for another software package of our institute, called "itom."[13] The source code of the GUI plugin as well as a Windows installer of the base program are also provided at Ref. 12.

3 Performance Results

Ray tracing is a method routinely applied to a broad range of problems. In an attempt to create an overview of the applicability of GPU-accelerated computing for optical-simulation ray tracing in general, we chose three different optical systems that are meant to cover the most typical applications of ray tracing in optical simulation. All computations presented in this contribution were done on a Windows 7 system that uses 20 GB of DDR3 RAM and runs on an Intel i7 980 CPU at 3.33 GHz. A nVidia GeForce GTX580 graphics board was used to do the GPU-accelerated computations and for the system viewport simultaneously. The measurement of the computation times presented in Figs. 1 and 2 was done with standard timing routines inside the code and includes transmission of all data of all rays to the GPU and back. No postprocessing of the rays, e.g., calculation of irradiance distributions, is included in the time measurements, and the tracing times of the CPU were all recorded utilizing a single thread.

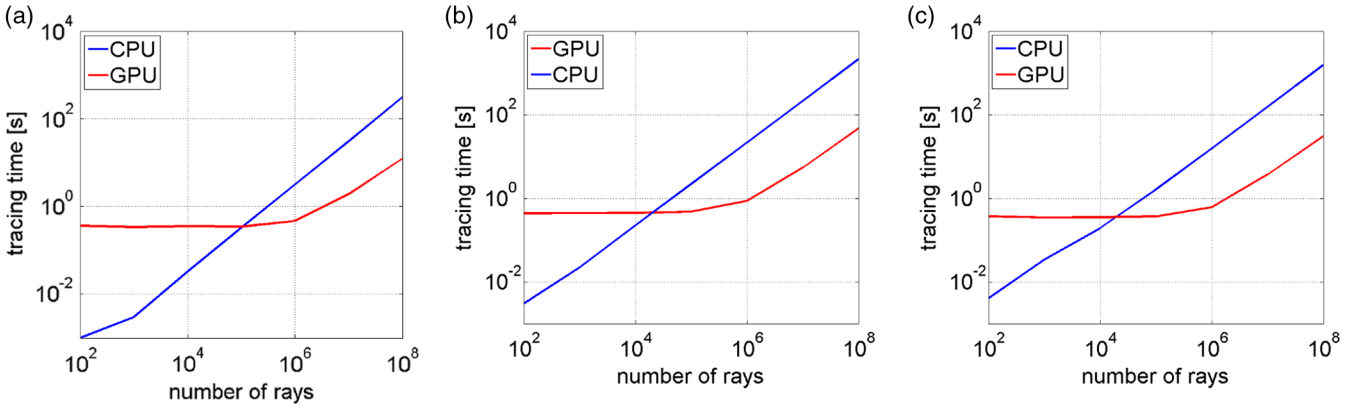

Fig. 1 Comparison of computation times depending on the number of traced rays. Blue curves show the computation times using GPU acceleration, and red lines show the results of the conventional CPU mode for the simple sequential system (a), the complex sequential system (b), and the nonsequential stray light system (c).

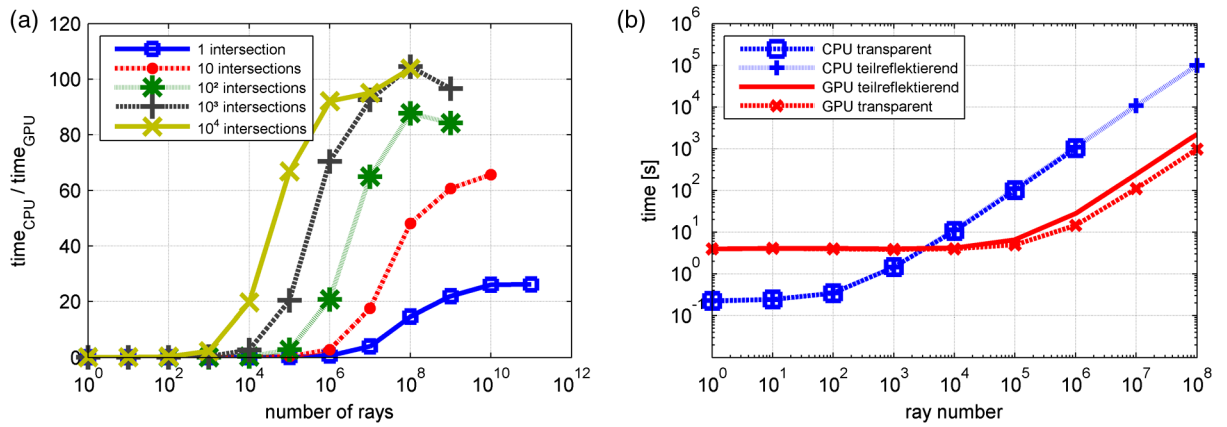

Fig. 2 (a) speedup of the GPU implementation depending on number of rays and number of intersections per ray. The benchmark system was equipped with an nVidia Tesla C2070 GPU and Intel i7 3.2 GHz CPU. (b) Computation times depending on uniformity of ray paths allowing 1000 intersections.

The first test system represents the class of applications that deals with the sequential analysis of comparably simple optical systems. In this case, we chose the double-Gauss objective example that is distributed as a sample system with copies of the commercial ray-tracing software Zemax (see Fig. 3).

Figure 1(a) shows the computation times depending on the number of rays that are traced through the system for the GPU-accelerated tracer and the CPU-based tracer in double logarithmic scale. The computation time of the CPU-based code shows the expected, nearly perfect linear dependence to the number of rays. The curve representing the computation time for the GPU-accelerated code shows two different regimes. For low numbers of rays, the computation time is independent of the actual number of rays and substantially higher than the computation time of the CPU code. In this regime, the computation time is dominated by the time that is needed to transfer data to the GPU and back. For high ray numbers, the curve approaches a linear form that is roughly parallel to the curve of the CPU code, indicating that the GPU is fully occupied by the calculations and issuing more computation commands to the GPU results in sequential execution. For the optical system of Fig. 3, the GPU code starts to be faster than the CPU code at a ray number of roughly 100,000. According to the timing data that is summarized in Tables 1 and 2, the performance advantage of the GPU code saturates at a factor of 25 for ray numbers >100 million.

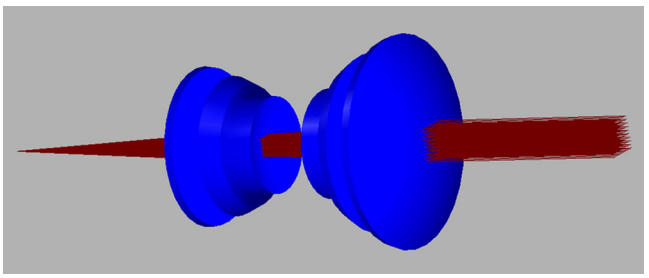

Fig. 3 Rendered view of the double-Gauss objective used to represent simple sequential applications. This picture is directly exported from our ray-tracing program.

The second test system is depicted in Fig. 4 and was chosen to represent the class of complex sequential applications. It comprises a lithography objective according to a US patent.[14] It consists of 38 spherical lenses, i.e., 76 surfaces, for a sequential analysis that ignores the side faces of the lenses.

An examination of the computation times for this optical system as depicted in Fig. 2(b) reveals the same principle of behavior as for the simple sequential system; however, the GPU code starts to be faster than the CPU code for 20,000 rays compared to 100,000 rays for the simple sequential system. Additionally, the acceleration factor approaching ray numbers >100 million reaches a value of approximately 45.

Figure 5 depicts a spectrometer system that represents the class of nonsequential stray light applications. It comprises 16 surfaces, including a diffraction grating with complex reflectance functions, that is highlighted in green in Fig. 5. It was investigated in detail in an earlier publication[4] that described the first application of an early version of the GPU-accelerated ray tracer presented here.

Table 1 Computation times of CPU-based tracing in seconds.

	100 rays	1000 rays	10,000 rays	100,000 rays	1 million rays	10 million rays	100 million rays
Double-Gauss objective	0001	0003	0033	0327	3,203	31,240	313,125
Lithography objective	0003	0022	0222	2193	21,383	212,739	2,127,471
Spectrometer system	0004	0034	0194	1621	15,431	154,310	1,540,818

Table 2 Computation times of GPU-based tracing in seconds.

	100 rays	1000 rays	10,000 rays	100,000 rays	1 million rays	10 million rays	100 million rays
Double-Gauss objective	0367	0338	0359	0347	0465	1925	12,450
Lithography objective	0436	0444	0451	0482	0882	5404	47,179
Spectrometer system	0372	0349	0355	0374	0611	3703	30,767

An examination of the computation times for this nonsequential optical systems again shows the typical behavior already observed for the other systems. As for the complex sequential system, GPU code starts to be faster for 20,000 rays and traces rays approximately 50 times faster than a CPU for ray numbers >100 million.

4 Systematization of Performance Results

The performance results of Sec. 3 show that GPU-accelerated ray tracing provides a significant speedup compared to conventional CPU-based tracing. However, the actual speedup depends strongly on both the number of rays involved in the simulation and the complexity of the optical system. For ray numbers below a certain threshold, performance of the GPU-accelerated code drops below the performance of the CPU code. Additionally, the advantage of the GPU acceleration increases when the optical system consists of more surfaces. To systematize the results of these application-oriented test systems, we present a more generic system that might help to illustrate the effects that affect the performance of the GPU-accelerated code. Figure 6 shows a schematic view of this test scene.It consists of a set of four plane surfaces. The outer two plane surfaces are made perfectly reflecting, whereas the inner two surfaces confine an area filled with a refracting material whose refracting index equals the background refracting index. To control the number of circulations in this resonator, an extra variable counting the number of intersections that a ray encounters has been added to each ray. The scene is traced nonsequentially, and each ray encounters six intersections during one circulation. Snell's law has to be evaluated four times upon refraction at the two inner surfaces, and reflection has to be calculated two times. By restricting the numbers of intersections for each ray to a low number, we mimic a simple scene, and by increasing the number of allowed intersections, we increase the complexity of the scene. Figure 2(a) compares the computation times of the GPU and CPU implementations depending on the number of rays that are traced.

As the transfer of data to the GPU and back is included, it can be seen that the CPU implementation is faster for low numbers of rays, resulting in a quotient of the CPU time and the GPU time being close to zero. Furthermore, it can be seen that the speedup of the GPU implementation can reach values up to 100 for complex simulations involving more than a thousand intersections of more than 10 million rays.

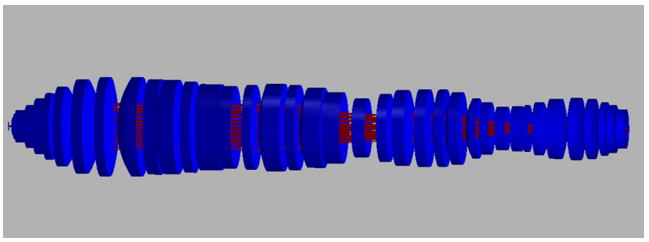

Fig. 4 Rendered view of lithography objective adapted from Ref. 12, used to represent complex sequential applications. This picture is directly exported from our ray-tracing program.

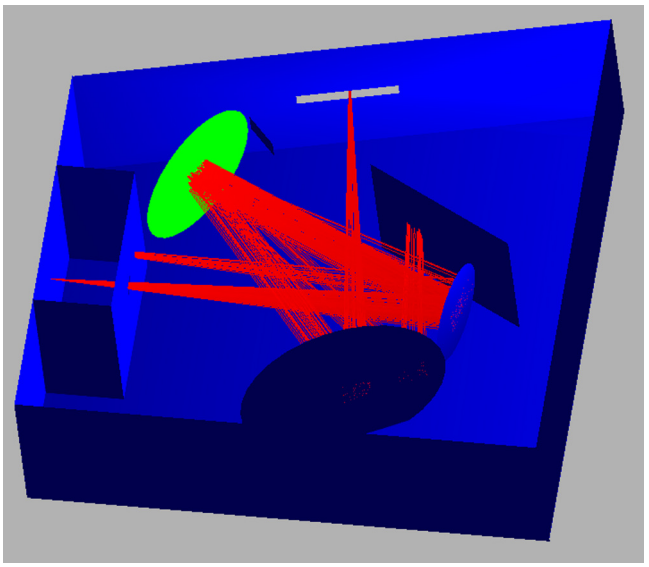

Fig. 5 Rendered view of spectrometer system that was investigated in detail in Ref. 4, used to represent nonsequential stray light analysis. This picture is directly exported from our ray-tracing program.

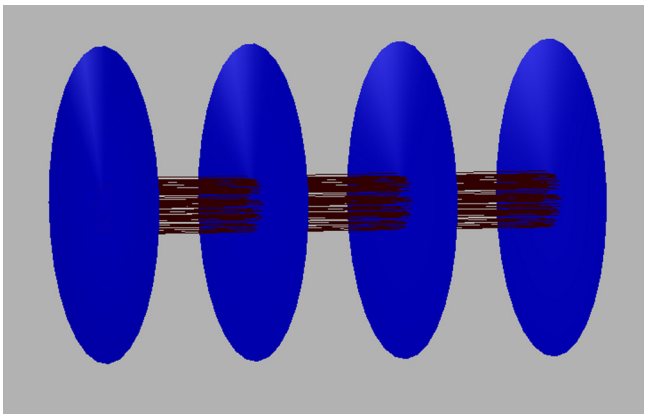

Fig. 6 Schematic view of the generic test scene proposed for benchmarking GPU ray tracing against CPU ray tracing.

Figure 2(b) illustrates another interesting effect. Making the inner two surfaces reflecting 50% of the rays causes the rays to propagate along different paths through the resonator of Fig. 6. The blue lines indicating the computation time of the CPU implementation are not distinguishable. However, running this simulation on a GPU shows that computation time of the semi-reflecting scene approximately doubles. This effect is due to the SIMD architecture of the GPU cores. As the rays propagate along different paths through the scene, it gets harder to load rays to the cores that are processed using the same instructions, and the so-called thread coherence[10] is violated. Differing instructions are in fact processed sequentially on the GPU, thereby increasing computation time.[10]

5 Conclusions

The results of the application-oriented examples of Sec. 3 show that the performance of the GPU-accelerated ray tracing strongly depends on the actual problem at hand. Therefore Sec. 4 presents a generic test system that attempts to systematize this application dependency. It is apparent that the GPU acceleration becomes more effective, the more rays are traced in the simulation and the more complex the optical system is. Acceleration factors of up to 100 compared to the CPU-based code can be observed in Fig. 2. In general, nonsequential ray tracing increases the computational load per ray, and therefore GPU acceleration becomes significant for lower ray numbers compared with sequential simulations. However, as rays may travel along a large number of different paths through a complex scene in nonsequential simulations, thread-coherence issues start to impact GPU performance. By giving full access to the source code of this software to the open-source community, we hope that this high-speed ray-tracing software will prove helpful for the development of new innovative ray tracing-based applications.

Acknowledgments

The financial support of the Bundesministerium für Bildung und Foschung (BMBF) under the Grant 13N10386 is gratefully acknowledged.

References

1. M. Born and E. Wolf, *Principles of Optics*, 7th Ed., Cambridge University Press, Cambridge, United Kingdom (2006).
2. D. P. Feder, "Optical calculations with automatic computing machinery," *J. Opt. Soc. Am.* **41**(9), 630–635 (1951).
3. The Scott Partnership, "Optical simulation software," *Nat. Photon.* **4**(4), 256–257 (2010).
4. F. Mauch et al., "Combining rigorous diffraction calculation and GPU accelerated nonsequential raytracing for high precision simulation of a linear grating spectrometer," *Proc. SPIE* **8083**, 80830F (2011).
5. F. Riechert et al., "Ray-based simulation of the propagation of light with different degrees of coherence through complex optical systems," *Appl. Opt.* **48**(8), 1527–1534 (2009).
6. M. A. Alonso and G. W. Forbes, "Stable aggregates of flexible elements give a stronger link between rays and waves," *Opt. Express* **10**(16), 728–739 (2002).
7. J. C. Halimeh et al., "Photorealistic images of carpet cloaks," *Opt. Express* **17**(22), 19328–19336 (2009).
8. T. J. Purcell et al., "Ray tracing on programmable graphics hardware," *ACM Trans. Graph.* **21**(3), 703–712 (2003).
9. nVidia, http://www.nvidia.com/object/optix.html (accessed 23 July 2012).
10. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, Elsevier, New York (2010).
11. nVidia Corp., "OptiX Programming Guide," version 2.5 (2008).
12. See http://bitbucket.org/itom/macrosim for current source code.
13. The source code of itom will be realeased under LGPL at https://bitbucket.org/itom/itom.
14. K.-H. Schuster, "Projection Objective," US Patent 6522484B1 (2003).

Biographies and photographs of the authors not available.