EPJ Web of Conferences **214**, 02027 (2019) https://doi.org/10.1051/epjconf/201921402027
CHEP 2018

Opticks : GPU Optical Photon Simulation for Particle Physics using NVIDIA® OptiX™

Simon Blyth[1,*]

[1]Institute of High Energy Physics, CAS, Beijing, China.

Abstract. Opticks is an open source project that integrates the NVIDIA OptiX GPU ray tracing engine with Geant4 toolkit based simulations. Massive parallelism brings drastic performance improvements with optical photon simulation speedup expected to exceed 1000 times Geant4 with workstation GPUs.
Optical physics processes of scattering, absorption, scintillator reemission and boundary processes are implemented as CUDA OptiX programs based on the Geant4 implementations. Wavelength-dependent material and surface properties as well as inverse cumulative distribution functions for reemission are interleaved into GPU textures providing fast interpolated property lookup or wavelength generation. OptiX handles the creation and application of a choice of acceleration structures such as boundary volume hierarchies and the transparent use of multiple GPUs. A major recent advance is the implementation of GPU ray tracing of complex constructive solid geometry shapes, enabling automated translation of Geant4 geometries to the GPU without approximation. Using common initial photons and random number sequences allows the Opticks and Geant4 simulations to be run point-by-point aligned. Aligned running has reached near perfect equivalence with test geometries.

1 Introduction

Opticks[1,2] enables Geant4[3-5]-based optical photon simulations to benefit from the high performance massively parallel GPU ray tracing provided by NVIDIA® OptiX™[6-8]. With the recently introduced NVIDIA Turing architecture GPUs OptiX performance is further enhanced by dedicated ray-tracing processors called RT cores[9].

Monte Carlo simulations of optical photon propagation are used to develop models of diverse photon transport systems ranging from human tissue within medical imaging scanners to large volumes of scintillators instrumented with photon detectors in neutrino and dark matter search experiments. These simulations are the primary technique used to design, optimize and analyse complex detection systems. However the versatility of the Monte Carlo technique comes with computational and memory costs that become extreme when propagating large numbers of photons using traditional sequential processing. Opticks aims to provide a parallel solution to this optical photon simulation problem. Drastically improved optical photon simulation performance can be transformative to the design, operation and understanding of diverse optical systems.

*e-mail: simon.c.blyth@gmail.com

These proceedings focus on recent major changes to the geometry workflow of Opticks and to the validation approach. Prior proceedings[10] provide further details on NVIDIA OptiX, use of GPU textures and the CUDA port of Geant4 photon generation and optical physics.

1.1 Importance of Optical Photon Simulation to Neutrino Detectors

Cosmic muon induced processes are crucial backgrounds for neutrino detectors such as Daya Bay [11] and JUNO[12], necessitating underground sites, water shields and muon veto systems. Minimizing the dead time and dead volume that results from applying a veto requires an understanding of the detector response to a muon. Large simulated samples of muon events are crucial in order to develop such an understanding. The number of optical photons estimated to be produced by a muon of typical energy 100 GeV crossing the JUNO scintillator is at the level of tens of millions. Profiling the Geant4-toolkit-based simulation shows that the optical photon propagation consumes more than 99% of CPU time, and imposes severe memory constraints that have forced the use of event splitting. As optical photons in neutrino detectors can be considered to be produced by only the scintillation and Cerenkov processes and yield only hits on photomultiplier tubes it is straightforward to integrate an external optical photon simulation with a Geant4 simulation of all other particles.

1.2 Throughput Oriented Processing

Graphics Processing Units (GPUs) originally designed for rendering are now increasingly used for computing tasks with parallel workloads. Efficient use of GPUs requires problems which can be divided into many thousands of mostly independent parallel tasks, which are each executed within separate threads. To maximize the number of simultaneous running threads the use of resources such as registers, shared memory and synchronization between threads must be minimized. Optical photon simulation is well matched to the requirements for efficient GPU usage, with sufficient parallelism from the large numbers of photons, low register usage from the simplicity of the optical physics, and decoupled nature of the photons avoiding synchronization.

The most computationally demanding aspect of optical photon simulation is the calculation, at each step of the propagation, of intersection positions of rays representing photons with the geometry of the system. This ray tracing limitation of optical photon simulation is shared with the synthesis of realistic images in computer graphics. Due to the many applications of ray tracing in the advertising, design, games and film industries the computer graphics community has continuously improved ray tracing techniques. The Turing GPU architecture introduced by NVIDIA in 2018 is marketed as the worlds first Ray-tracing GPU, with hardware "RT Cores" in every streaming multiprocessor (SM) dedicated to the acceleration of ray geometry intersection. NVIDIA claims performance of more than 10 billion ray intersections per second, which is a factor 10 more than possible with its Pascal architecture GPUs which performs the intersection acceleration in software.

GPUs are throughput-oriented[13], optimizing the total amount of work completed per unit time, by using many simple processing units. This contrasts with CPUs which are latency-oriented, minimizing the time elapsed between initiation and completion of a single task by the use of complex caching systems to avoid the latency of memory access and other complicated techniques such as branch prediction and speculative execution. GPUs generally do not use the complicated techniques common in CPUs, allowing more chip area to be dedicated to parallel compute, which results in greater computational throughput across all threads at the expense of slower single-thread execution. Threads blocked while waiting

to access memory are tolerated by using hardware multithreading to resume other unblocked threads, allowing latencies to be hidden if there are sufficient parallel threads in flight. GPUs evolved to meet the needs of real-time computer graphics, rendering images of millions of pixels from geometries composed of millions of triangles, they are designed to execute literally billions of small "shader" programs per second. When porting CPU code to run on the GPU a total reorganization of data and computation is required in order to make effective use of the totally different processor architecture.

1.3 NVIDIA® OptiX™ Ray Tracing Engine

OptiX is a general-purpose ray tracing engine designed for NVIDIA GPUs that exposes an accessible single ray programming model. The core of OptiX is a domain-specific just-in-time compiler that constructs ray tracing pipelines combining code for acceleration structure creation and traversal together with user provided CUDA code for ray generation, object intersection and closest hit handling. Spatial index data structures, such as the boundary volume hierarchy (BVH), are the principal technique for accelerating ray geometry intersection. The OptiX API[8] provides only the acceleration of ray geometry intersection, not the intersection itself thus affording users full flexibility to implement intersections with any form of geometry. In the recently introduced Turing architecture GPUs some of the BVH traversal workload has been moved into hardware "RT cores"[9]. The OptiX acceleration structure supports instancing, allowing them to be shared between multiple instances of the same geometry such as the photomultiplier tubes in the JUNO geometry.

1.4 Thrust High Level C++ interface to CUDA

The CUDA Thrust [14] C++ template library provides a high level interface to CUDA, greatly simplifying GPU development. Opticks uses Thrust for several tasks:

- associating photons with their "gensteps" entirely on the GPU
- downloading photons that hit sensors to the CPU using the stream compaction technique
- sorting photons by step sequence histories, flags for up to 16 steps are stored

Gensteps are the parameters of optical photon producing steps of scintillation or Cerenkov processes, including the number of photons and the line segment along which to generate them. For a detailed description see the prior proceedings [10].

2 Geometry Translation and Upload to GPU

Implementing an efficient GPU optical photon simulation equivalent to the Geant4 simulation requires that all aspects of the Geant4 context relevant to optical photon propagation are translated into an appropriate form and serialized for upload to the GPU.

2.1 Modelling Complex Shapes using Constructive Solid Geometry

Opticks provides ray intersection for ten primitive shapes including sphere, hyperboloid and torus. Ray primitive intersection uses the parametric ray equation together with implicit equations of the primitives to yield a polynomial in t, the distance along the ray from its origin position. Intersections are found from the roots of the polynomial with $t > t_{min}$ and surface normals are obtained at intersects using the derivative of the implicit equation or by inspection. Arbitrarily complex solids are described using constructive solid geometry

(CSG) modelling, which builds shapes from the combination of primitive constituent shapes by boolean set operations: union, intersection and difference. A binary tree data structure with primitives at the leaves of the tree and operators at the internal nodes is used to represent the solids. Any node can have an associated local transform, represented by a 4x4 transformation matrix, which is combined with other local transforms to yield global transforms in the frame of the root node of the tree.

2.2 Ray Intersection with CSG Shapes

Intersecting rays with general CSG shapes requires the appropriate primitive intersect to be selected depending on the origin and direction of the ray and the current t_{min}. Traditional implementations of CSG intersection first calculate ray intervals with each primitive and then combine these intervals using the boolean operators to determine intersects. Efficient use of GPUs requires many thousands of simultaneously operational threads which disfavors the traditional approach due to the requirement to store intervals for all constituent primitives. A quite different approach described by Andrew Kensler[15] avoids interval storage by instead selecting between the two candidate intersects at each level of the binary tree, which allows a recursive algorithm to be developed. The two candidate intersects at each level are classified as "Enter", "Exit" or "Miss" using the angle between the ray direction and surface normal. Six decision tables corresponding to which side is closer and to the three boolean operators are used to determine an action from the classifications such as returning an intersect or advancing t_{min} and intersecting again. Recursive function calls are a natural way to process self similar structures such as CSG trees, however recursion is a memory expensive technique which makes it inappropriate for GPU usage. Although NVIDIA OptiX supports recursive ray tracing in does not support recursion within intersect programs. The Opticks "evaluative" CSG implementation was inspired by the realization that CSG node tree intersection directly parallels binary expression tree evaluation and that techniques to simplify expression tree evaluation such as using postorder traversals could be applied. Binary expression trees are used to represent and evaluate mathematical expressions. A postorder traversal of a node tree visits every node in sequence such that child nodes are visited before their parents. Factoring out the postorder sequence allowed an iterative solution to be developed for a recursive problem.

The CSG implementation relies on selection of the closer of two intersects at each level of the node tree. When the faces of constituent shapes coincide the ambiguity regarding which is closer can cause spurious intersects. Modifying some constituents to prevent coincident faces avoids the issue without changing the intended geometry. As such coincidences are rather common Opticks includes detection and automated fixing for some common situations.

2.3 Solid Serialization and Translation from Geant4 Solids

Each primitive or operator node is serialized into an array of up to 16 32-bit elements. These elements include float parameters of the primitives and integer index references into a separate global transforms buffer. For the convex polyhedron primitive which is defined by a list of surface planes, the primitive contains an integer index referencing into a separate plane buffer together with the number of planes. A complete binary tree serialization, illustrated in Figure 1, with array indices matching level order tree indices and zeros at missing nodes is used for the serialization of the CSG trees. This structure allows tree navigation directly from bitwise manipulations of the serialized array index.

Complete binary tree serialization is simple and effective for small trees but very inefficient for unbalanced trees necessitating tree balancing for shapes with many constituent primitives to reduce the tree height. A two-stage procedure to balance input trees was developed,

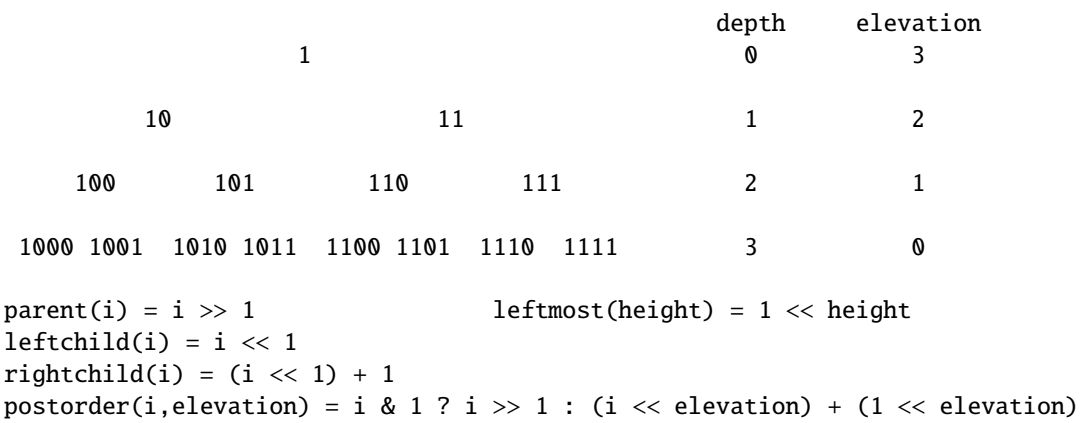

Figure 1. Height 3 complete binary tree, with nodes labelled with 1-based level order indices *i* in binary. Postorder traversal of a binary tree visits every node in sequence such that child nodes are visited before their parents. The simplicity of the complete binary tree data structure allows tree navigation from the bitwise manipulations of level indices. Total nodes in a complete binary tree of height h is $2^{h+1} - 1$.

first converting to a positive form tree by recursive application of De Morgan's laws replacing non-commutative differences with intersections with complemented sub-trees, $A - B = A \cap !B$. The end result is a tree with only union and intersection operators and some complemented leaves. Intersections with complemented primitives, which correspond to "inside out" solids was implemented by flipping normals and reclassifying "Miss" as "Exit", because it is not possible to miss the unbounded "other side" of a complemented primitive. Subsequently the commutative nature of intersect and union operators enables the tree to be reconstructed in a more balanced form.

Opticks provides translations in both directions between Geant4 solids and Opticks primitives. Depending on parameter values such as inner radii or phi segments there is not always a one-to-one correspondence between the two models with some Geant4 solids being represented as Opticks node trees. The Opticks approach of relying more on the CSG implementation was adopted to minimize duplicated code in the primitives.

2.4 Geometry Structure Model and Direct Translation

A major recent advance made possible by the general CSG solid intersection described above is the implementation of automated direct geometry translation allowing the Geant4 geometry model in memory to be directly translated and uploaded to the GPU. The former approach to geometry translation described in [10] required the geometry to be exported and imported via GDML and G4DAE files.

The Opticks geometry model is based upon the observation that many elements of a detector geometry are repeated demanding the use of instancing for efficient representation. Geometry instancing is a technique used in computer graphics libraries including OpenGL and NVIDIA OptiX that avoids duplication of information on the GPU by storing repeated elements only once together with 4x4 transform matrices that specify the locations and orientations of each instance. The Geant4 geometry model comprises a hierarchy of volumes with associated transforms.

The first step in the automated translation from Geant4 to the Opticks geometry is to traverse the Geant4 volume tree converting materials, surfaces, solids, volumes and sensors into Opticks equivalents. The converted solids contain both analytic CSG node trees and triangulated meshes. Each solid is assigned a boundary index uniquely identifying the combination of four indices representing outer and inner materials and outer and inner surfaces. Outer/inner surfaces handle inwards/outwards going photons allowing the Geant4 border and skin surface functionality to be translated.

A geometry digest string for every structure node is formed from the transforms and shape indices of the progeny nodes descended from it in the geometry tree. Subsequently repeated groups of volumes and their placement transforms are identified using the geometry digests, after disqualifying repeats that are contained within other repeats. All structure nodes that pass instancing criteria regarding the number of vertices and number of repeats are assigned an instance index with the remainder forming the global non-instanced group. These groups of volumes are then used for the creation of the NVIDIA OptiX analytic geometry instances, and OpenGL mesh geometry instances.

3 Interface between Geant4 and Opticks

A single class, `G4Opticks`, is used to provide a minimal interface between Geant4 user code and the Opticks package. At initialization the Geant4 top volume pointer is passed to Opticks which translates the geometry and constructs the OptiX GPU context.

As Geant4 has no "genstep" interface it is necessary to modify the classes representing scintillation and Cerenkov processes. Instead of generating photon secondary tracks in a loop the relevant "genstep" parameters, such as the number of photons to generate and the line segment along which to generate them, are collected. Collecting and copying gensteps rather than photons avoids allocation of CPU memory for the photons, only photons that reach sensors requiring CPU memory allocation. The generation loops of the scintillation and Cerenkov processes are ported to CUDA which effectively splits the photon generation implementation between the CPU and GPU. Reemission is implemented by a fraction of photons absorbed within scintillators being reborn with a wavelength generated using an appropriate inverse cumulative distribution function which is accessed via a GPU texture.

At the end of Geant4 event processing gensteps are uploaded to the GPU and an OptiX kernel is launched to generate and propagate the photons. Further details on the CUDA port of the optical photon simulation can be found in reference [10].

4 Random Number Aligned Comparison of Opticks and Geant4

Validation comparisons use a single executable that performs both the Geant4 and hybrid Opticks simulations and writes two events in a format which includes highly compressed positions, times, wavelengths and polarizations at up to 16 steps of the optical photon propagations. Opticks uses the cuRAND[16] library for concurrent generation of millions of reproducible sequences of pseudorandom numbers. Copying cuRAND sequences to the CPU and configuring the Geant4 random engine to use them makes it possible to align the consumption of random numbers between the two simulations, resulting in nearly perfectly matched results with every scatter, absorption and reflection happening with the same positions, times, wavelengths and polarizations. Direct comparison of the aligned simulation results allows any discrepancies to be identified immediately. The primary cause of differences are spurious intersects resulting from coincidences between the surfaces of CSG constituents as described in section 2.2 which require geometry fixes.

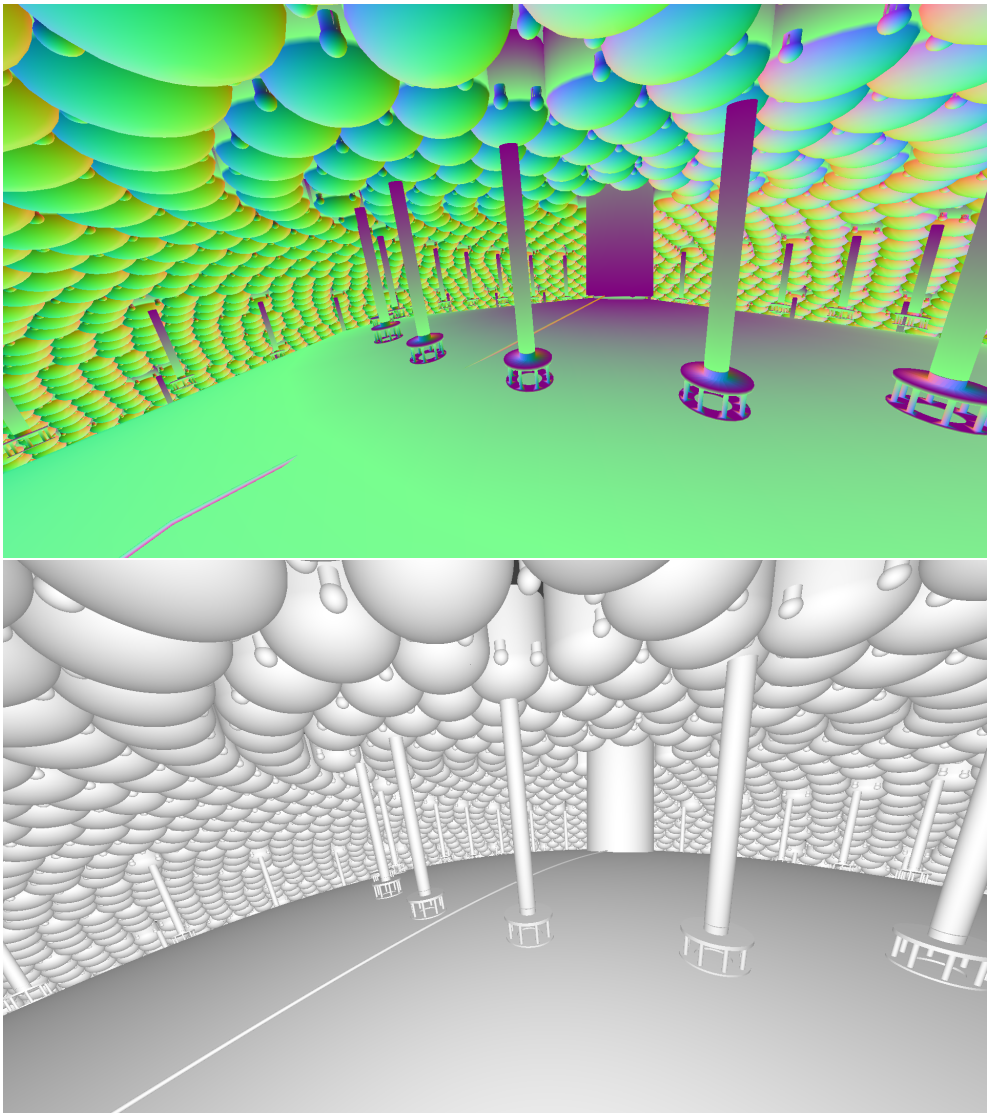

Figure 2. Renders of the chimney region of JUNO detector showing photomultiplier tubes, acrylic sphere, supports and calibration guide tube torus. The geometry was directly converted from Geant4 into an Opticks geometry including analytic CSG as well as approximate mesh representations of all solids, and persisted into a geometry cache of NumPy[17] binary files. The upper render is an OpenGL rasterization of the mesh geometry. Effects of the approximation are visible with the guide tube torus appearing to impinge into the acrylic sphere. The lower render is a ray-traced image of the analytic CSG geometry which is the same geometry used in the simulation.

5 Summary

Opticks enables Geant4-based simulations to benefit from effectively zero time and zero CPU memory optical photon simulation, due to the massively parallel GPU processing made accessible by NVIDIA OptiX. Recent Opticks developments enable automated translation of Geant4 geometries without approximation. This together with the adoption of modern CMake [18] configuration techniques [19] and development of a minimal interface between Geant4 user code and embedded Opticks make getting started with a new geometry drastically simpler than before. Opticks can benefit any simulation limited by optical photons, the more limited the greater the benefit.

Acknowledgements

The Daya Bay and JUNO collaborations are acknowledged for the use of detector geometries and simulation software. Dr. Tao Lin is acknowledged for his assistance with the JUNO Offline software. This work is funded by Chinese Academy of Sciences President's International Fellowship Initiative, Grant No. 2018VMB0002.

References

[1] Opticks Repository, `https://bitbucket.org/simoncblyth/opticks/`

[2] Opticks Group, `https://groups.io/g/opticks`

[3] Agostinelli S, Allison J, Amako K, Apostolakis J, Araujo H, Arce P, et al. 2003 Geant4–a simulation toolkit *Nucl Instrum Methods Phys Res* A **506** pp 250-–303

[4] Allison J, Amako K, Apostolakis J, Araujo H, Dubois P, Asai M, et al. 2006 Geant4 developments and applications *IEEE Trans Nucl Sci* **53** pp 270–8

[5] Allison J, Amako K, Apostolakis J, Arce P, Asai M, Aso T, et al. 2016 Recent developments in Geant4 *Nucl Instrum Methods Phys Res* A **835** pp 186–225

[6] Parker S, Bigler J, Dietrich A, Friedrich H, Hoberock J, et al. 2010 OptiX: a general purpose ray tracing engine *ACM Trans. Graph. : Conf. Series* **29** p 66

[7] OptiX introduction, `https://developer.nvidia.com/optix`

[8] OptiX API, `http://raytracing-docs.nvidia.com/optix/index.html`

[9] NVIDIA RTX, `https://developer.nvidia.com/rtx`

[10] Blyth Simon C 2017 J. Phys.: Conf. Ser. **898** 042001

[11] An F, et al. 2016 The detector system of the Daya Bay reactor neutrino experiment *Nucl Instrum Methods* A **811** pp 133–161

[12] An F et al. 2016 Neutrino physics with JUNO *J Phys G* **43** 030401

[13] M. Garland and D. B. Kirk, Understanding Throughput Oriented Architectures, *COMMUN ACM* Volume 53 Issue 11, November 2010, pp 58-66

[14] Bell N and Hoberock J 2011 *Thrust: a Productivity-Oriented Library for CUDA* (GPU Computing Gems Jade Edition) ed W W Hwu, Chapter 26

[15] A. Kensler, Ray Tracing CSG Objects Using Single Hit Intersections (2006), with corrections by author of XRT Raytracer `http://xrt.wikidot.com/doc:csg`

[16] cuRAND, `http://docs.nvidia.com/cuda/curand/index.html`

[17] Van der Walt S, Colbert S, Varoquaux G 2011 The NumPy array: a structure for efficient numerical computation *Comput. Sci. Eng.* **13** pp 22–30

[18] CMake website, `https://cmake.org`

[19] Fork of Boost CMake Modules, `https://github.com/simoncblyth/bcm`