DOI:10.1145/2447976.2447997

GPU Ray Tracing

By Steven G. Parker, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys, Morgan McGuire, and Martin Stich

Abstract

The NVIDIA® OptiX™ ray tracing engine is a programmable system designed for NVIDIA GPUs and other highly parallel architectures. The OptiX engine builds on the key observation that most ray tracing algorithms can be implemented using a small set of programmable operations. Consequently, the core of OptiX is a domain-specific just-in-time compiler that generates custom ray tracing kernels by combining user-supplied programs for ray generation, material shading, object intersection, and scene traversal. This enables the implementation of a highly diverse set of ray tracing-based algorithms and applications, including interactive rendering, offline rendering, collision detection systems, artificial intelligence queries, and scientific simulations such as sound propagation. OptiX achieves high performance through a compact object model and application of several ray tracing-specific compiler optimizations. For ease of use it exposes a single-ray programming model with full support for recursion and a dynamic dispatch mechanism similar to virtual function calls.

1. INTRODUCTION

Many CS undergraduates have taken a computer graphics course where they wrote a simple ray tracer. With a few simple concepts on the physics of light transport, students can achieve high quality images with reflections, refraction, shadows, and camera effects such as depth of field—all of which present challenges on contemporary real-time graphics pipelines. Unfortunately, the computational burden of ray tracing makes it impractical in many settings, especially where interactivity is important. Researchers have invented many techniques for improving the performance of ray tracing,[13] especially when mapped to high-performance architectural features such as explicit SIMD instructions[12] and Single-Instruction Multiple-Thread (SIMT)-based[6] GPUs.[1] Unfortunately most such techniques muddy the simplicity and conceptual purity that make ray tracing attractive. Nor have industry standards emerged to hide these complexities, as Direct3D and OpenGL do for rasterization.

To address these problems, we introduce OptiX, a general purpose ray tracing engine. A general programming interface enables the implementation of a variety of ray tracing-based algorithms in graphics and non-graphics domains, such as rendering, sound propagation, collision detection, and artificial intelligence. This interface is conceptually simple yet enables high performance on modern GPU architectures and is competitive with hand-coded approaches.

In this paper, we discuss the design goals of the OptiX engine as well as an implementation for NVIDIA GPUs. In our implementation, we compose domain-specific compilation with a flexible set of controls over scene hierarchy, acceleration structure creation and traversal, on-the-fly scene update, and a dynamically load-balanced GPU execution model. Although OptiX primarily targets highly parallel GPU architectures, it is applicable to a wide range of special- and general-purpose hardware, including modern CPUs.

1.1. Ray tracing, rasterization, and GPUs

Computer graphics algorithms for *rendering*, or image synthesis, take one of two complementary approaches. One family of algorithms loop over the pixels in the image, computing for each pixel, the first object visible at that pixel; this approach is called *ray tracing* because it solves the geometric problem of intersecting a ray from the pixel into the objects. A second family of algorithms loops over the objects in the scene, computing for each object the pixels covered by that object. Because the resulting per-object pixels (called *fragments*) are formatted for a raster display, this approach is called *rasterization*. The central data structure of ray tracing is a spatial index called an *acceleration structure*, used to avoid testing each ray against all objects. The central data structure of rasterization is the *depth buffer*, which stores the distance of the closest object seen at each pixel and discards fragments from invisible objects. While both approaches have been generalized and optimized greatly beyond this simplistic description, the basic distinction remains: ray tracing iterates over rays while rasterization iterates over objects. High-performance ray tracing and rasterization, both focus on rendering the simplest of objects: triangles.

Historically, ray tracing has been considered slow and rasterization fast. The simple, regular structure of depth-buffer rasterization lends itself to highly parallel hardware implementations: each object moves through several stages of computation (the so-called *graphics pipeline*), with each stage performing similar computations in data-parallel fashion on the many objects, fragments, and pixels in flight throughout the pipeline. As graphics hardware has grown more parallel it has also grown more general, evolving from specialized fixed-function circuitry implementing the various stages of the graphics pipeline into fully programmable processors that virtualize those stages onto hundreds or even thousands of small general-purpose cores. Today's graphics processing units, or GPUs, are massively parallel processors capable of performing trillions of floating-point math operations and rendering billions of triangles each second. The computational horsepower and power efficiency of modern GPUs has made them attractive for high-performance

The original version of this paper is entitled "OptiX: A General Purpose Ray Tracing Engine" and was published in *ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH*, July 2010, ACM

computing, from many of the fastest supercomputers in the world to science, math, and engineering codes on the desktop. All of which raises the question: can ray tracing be implemented efficiently and flexibly on GPUs?

1.2. Contributions and design goals

To create a high-performance system for a broad range of ray tracing tasks, several trade-offs and design decisions led to the following contributions:

- *A general, low level ray tracing engine*. OptiX is not a renderer. It focuses exclusively on the fundamental computations required for ray tracing and avoids embedding, rendering-specific constructs such as lights, shadows, and reflectance.
- *A programmable ray tracing pipeline*. OptiX shows that most ray tracing algorithms can be implemented using a small set of lightweight programmable operations. It defines an abstract ray tracing execution model as a sequence of user-specified programs, analogous to the traditional rasterization-based graphics pipeline.
- *A simple programming model*. OptiX avoids burdening the user with the machinery of high-performance ray tracing algorithms. It exposes a familiar recursive, single-ray programming model rather than ray packets or explicit vector constructs, and abstracts any batching or reordering of rays.
- *A domain-specific compiler*. The OptiX engine combines just-in-time compilation techniques with ray tracing-specific knowledge to implement its programming model efficiently. The engine abstraction permits the compiler to tune the execution model for available system hardware.

2. RELATED WORK

While numerous high-level ray tracing libraries, engines, and APIs have been proposed,[13] efforts to date have been focused on specific applications or classes of rendering algorithms, making them difficult to adapt to other domains or architectures. On the other hand, several researchers have shown how to map ray tracing algorithms efficiently to GPUs and the NVIDIA® CUDA™ architecture,[1, 3, 11] but these systems have focused on performance rather than flexibility.

Further discussion of related systems and research can be found in the original paper.[10]

3. A PROGRAMMABLE RAY TRACING PIPELINE

The core idea behind the OptiX engine is that most ray tracing algorithms can be implemented using combinations of a small set of programmable operations. This is directly analogous to the programmable rasterization pipelines employed by OpenGL and Direct3D. At a high level, these systems expose an abstract rasterizer containing lightweight callbacks for vertex shading, geometry processing, tessellation, and pixel shading operations. An ensemble of these program types, often used in multiple passes, can be used to implement a broad variety of rasterization-based algorithms.

We have identified a corresponding programmable ray tracing execution model along with lightweight operations that can be customized to implement a wide variety of ray tracing-based algorithms.[9] These user-provided operations, which we simply call *programs*, can be combined with a user-defined data structure (*payload*) associated with each ray. The ensemble of programs together implement a particular client application's algorithm.

3.1 Programs

OptiX includes seven different types of these programs, each of which conceptually operates on a single ray at a time. In addition, a bounding box program operates on geometry to determine primitive bounds for acceleration structure construction. The combination of user programs and hardcoded OptiX kernel code forms the *ray tracing pipeline*, which is outlined in Figure 2. Unlike a feed-forward rasterization pipeline, it is more natural to think of the ray tracing pipeline as a call graph. The core operation, *rtTrace*, alternates between locating an intersection (*Traverse*) and responding to that intersection (*Shade*). By reading and writing data in user-defined ray payloads and in global device-memory arrays called *buffers*, these operations are combined to perform arbitrary computation during ray tracing.

Ray generation programs are the entry into the ray tracing pipeline. A single invocation of *rtContextLaunch* from the host will create many instantiations of these programs. A typical ray generation program will create a ray using a camera model for a single sample within a pixel, start a trace operation, and store the resulting color in an output buffer. But by distinguishing ray generation from pixels in an image, OptiX enables other operations such as creating photon maps, precomputing lighting texture maps (also known as baking), processing ray requests passed from OpenGL, shooting multiple rays for super-sampling, or implementing different camera models.

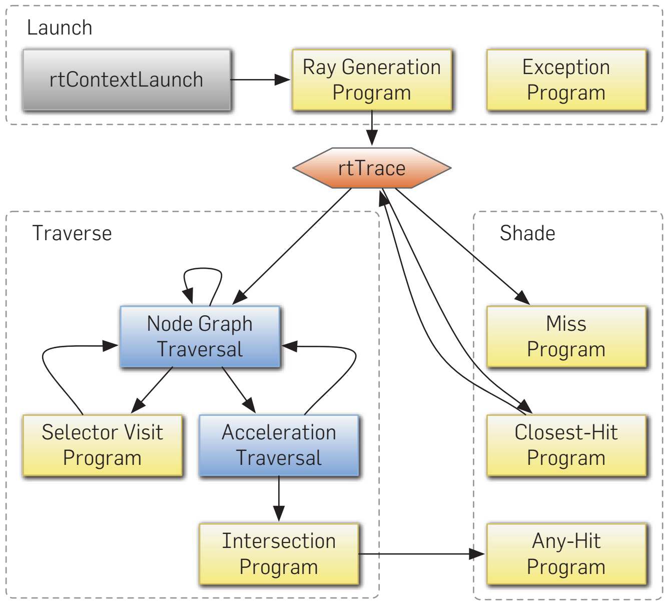
Intersection programs implement ray-geometry intersection tests. As the acceleration structures are traversed, the system will invoke intersection programs to perform geometric queries. The program determines if and where the ray touches the object and may compute normals, texture coordinates, or other attributes based on the hit position. An arbitrary number of attributes may be associated with each intersection. Intersection programs enable support for arbitrary surfaces beyond polygons and triangles, such as displacement maps, spheres, cylinders, high-order surfaces, or even fractal geometries like the Julia set in Figure 1. A programmable intersection operation is useful even in a triangle-only system because it facilitates direct access to native mesh formats.

Closest-hit programs are invoked once traversal has found the nearest intersection of a ray with the scene geometry. This program type resembles surface shaders in classical rendering systems. Typically, a closest-hit program will perform computations like shading, potentially casting new rays in the process, and store resulting data in the ray payload.

Any-hit programs are called during traversal for every ray-object intersection that is found. The any-hit program allows the material to participate in object intersection decisions while keeping the shading operations separate from the geometry operations. It may optionally terminate the ray

Figure 1. Images from various applications built with OptiX. *Top:* Physically based light transport through path tracing. *Bottom:* Ray tracing of a procedural Julia set, photon mapping, large-scale line of sight and collision detection, Whitted-style ray tracing of dynamic geometry, and ray traced ambient occlusion. All applications are interactive.

Figure 2. A call graph showing the control flow through the ray tracing pipeline. The yellow boxes represent user-specified programs and the blue boxes are algorithms internal to OptiX. Execution is initiated by the API call *rtContextLaunch*. A built-in function, *rtTrace*, can be employed by the ray generation program to cast rays into the scene. This function may also be called recursively by the closest-hit program for shadow and secondary rays. The exception program is executed when the execution of a particular ray is terminated by an error such as excessive memory consumption.



using the built-in function *rtTerminateRay*, which will stop all traversal and unwind the call stack to the most recent invocation of *rtTrace*. This is a lightweight exception mechanism that can be used to implement early ray termination for shadow rays and ambient occlusion. Alternatively, the any-hit program may ignore the intersection using *rtIgnoreIntersection*, allowing traversal to continue looking for other geometric objects. For instance, a program may choose to ignore an interaction based on a texture channel lookup to implement efficient alpha-mapped transparency without restarting traversal. Another use case for the any-hit program can be found in Section 6.1, where the application performs visibility attenuation for partial shadows cast by glass objects. Note that intersections may be presented out of order. The default any-hit program is a no-op, which is often the desired operation.

Miss programs are executed when the ray does not intersect any geometry in the interval provided. They can be used to implement a background color or environment map lookup.

Exception programs are executed when the system encounters an exceptional condition, for example, when the recursion stack exceeds the amount of memory available for each thread, or when a buffer access index is out of range. OptiX also supports user-defined exceptions that can be thrown from any program. The exception program can react, for example, by printing diagnostic messages or visualizing the condition by writing special color values to an output pixel buffer.

Selector visit programs expose programmability for coarse-level node graph traversal. For example, an application may choose to vary the level of geometric detail for parts of the scene on a per-ray basis.

3.2 Scene representation

An explicit goal of OptiX was to minimize the overhead of scene representation, rather than forcing a heavyweight scene graph onto users. The OptiX engine employs a simple but flexible structure for representing scene information and associated programmable operations, collected in a container object called the *context*. This representation is also the mechanism for binding programmable shaders to the object-specific data that they require.

Hierarchy nodes. A scene is represented as a lightweight graph that controls the traversal of rays through the scene. It can also be used to implement instancing two-level hierarchies for animations of rigid objects, or other common scene structures. To support instancing and sharing of common data, the nodes can have multiple parents.

Four main node types can be used to provide the scene representation using a directed graph. Any node can be used as the root of scene traversal. This allows, for example, different representations to be used for different ray types.

Group nodes contain zero or more (but usually two or more) children of any node type. A group node has an acceleration structure associated with it and can be used to provide the top level of a two-level traversal structure.

Geometry Group nodes are the leaves of the graph and contain the primitive and material objects described below. This node type also has an acceleration structure associated with it. Any non-empty scene will contain at least one geometry group.

Transform nodes have a single child of any node type, plus an associated 4×3 matrix that is used to perform an affine transformation of the underlying geometry.

Selector nodes have zero or more children of any node type, plus a single visit program that is executed to select among the available children.

Geometry and material objects. The bulk of the scene data is stored in the geometry nodes at the leaves of the graph. These contain objects that define geometry and shading operations. They may also have multiple parents, allowing material and geometry information to be shared at multiple points in the graph. As an example, consider Figure 3. The graph on the right shows a complete OptiX context for a simple scene with a pin-hole camera, two objects, and shadows. The ray generation program implements the camera, while a miss program implements a constant white background. A single geometry group contains two geometry instances with a single geometric index—in this case a bounding-volume hierarchy (BVH)—built over all underlying geometry in the triangle mesh and ground plane. Two types of geometry are implemented, a triangle mesh and a parallelogram, each with its own set of intersection and bounding box programs. The two geometry instances share a single material that implements a diffuse lighting model and fully attenuates shadow rays via closest-hit and any-hit programs, respectively.

The diagram on the left of Figure 3 illustrates how these programs are invoked for 3 rays that traverse through the scene: 1. The ray generation program creates rays and traces them against the geometry group. This initiates the Traverse stage shown in Figure 2, executing parallelogram and triangle-mesh intersection until an intersection is found (2 and 3). If the ray intersects with geometry, the closest-hit program will be called whether the intersection was found on the ground plane or on the triangle mesh. The material will recursively generate show rays to determine if the light source is unobstructed. 4. When any intersection along the shadow ray is found, the any-hit program will terminate ray traversal and return to the calling program with shadow occlusion information. 5. If a ray does not intersect with any scene geometry, the miss program will be invoked.

Geometry Instance objects bind a *geometry object* to a set of *material objects*. This is a common structure used by scene graphs to keep geometric and shading information orthogonal.

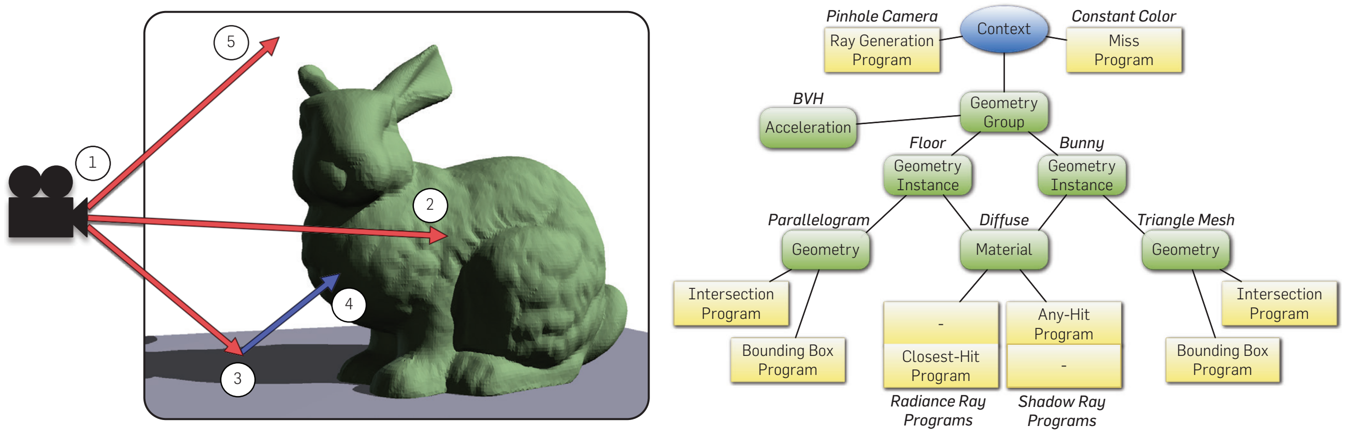
Geometry objects contain a list of geometric primitives. Each geometry object is associated with a bounding box program and an intersection program, both of which are shared among the geometry object's primitives.

Material objects hold information about shading operations, including the any-hit and closest-hit programs described in Section 3.1.

3.3. System overview

The OptiX engine consists of two distinct APIs. The *host API* is a set of C functions that the client application calls to create and configure a context, assemble a node graph, and launch ray tracing kernels. It also provides calls to manage GPU devices. The *program API* is the functionality exposed to user programs. This includes function calls for tracing rays, reporting intersections, and accessing data. In addition, several semantic variables encode state specific to ray tracing,

Figure 3. Example OptiX scene construction and execution.



for example, the current distance to the closest intersection. Printing and exception handling facilities are also available for debugging.

After using OptiX host, API functions to provide scene data such as geometry, materials, acceleration structures, hierarchical relationships, and programs, the application will then launch ray tracing with the *rtContextLaunch* API function that passes control to OptiX. If required, a new ray tracing kernel is compiled from the given user programs, acceleration structures are built (or updated) and data is synchronized between host and device memory, and finally, the ray tracing kernel is executed, invoking the various user programs as described above.

After execution of the ray tracing kernel has completed, its resulting data can be used by the application. Typically, this involves reading from output buffers filled by one of the user programs or displaying such a buffer directly, for example, via OpenGL. An interactive or multi-pass application then repeats the process starting at context setup, where arbitrary changes to the context can be made, and the kernel is launched again.

4. DOMAIN-SPECIFIC COMPILATION

The core of the OptiX host runtime is a just-in-time (JIT) compiler that serves several important functions. First, the JIT stage combines all of the user-provided shader programs into one or more kernels. Second, it analyzes the node graph to identify data-dependent optimizations. Finally, the resulting kernel is executed on the GPU using the CUDA driver API.

Generating and optimizing code for massively parallel architectures provide some challenges. One challenge is that code size and live state per computation must be minimized for maximum performance. Another challenge is structuring the code to reduce divergence. Our experience with OptiX highlights the interesting tensions between these sometimes conflicting requirements.

4.1. OptiX programs

The user-specified programs described in Section 3.1 are provided to the OptiX host API in the form of Parallel Thread Execution (PTX) functions.[8] PTX is a virtual machine assembly language for NVIDIA's CUDA architecture, similar in many ways to the popular open source Low-Level Virtual Machine (LLVM) intermediate representation.[5] Like LLVM, PTX defines a set of simple instructions that provide basic operations for arithmetic, control flow and memory access. PTX also provides several higher-level operations such as texture access and transcendental operations. Also similar to LLVM, PTX assumes an infinite register file and abstracts many real machine instructions. A JIT compiler in the CUDA runtime will perform register allocation, instruction scheduling, dead-code elimination, and numerous other late optimizations as it produces machine code targeting a particular GPU architecture.

PTX is written from the perspective of a single thread and thus does not require explicit lane mask manipulation operations. This makes it straightforward to lower PTX from a high-level shading language, while giving the OptiX runtime the ability to manipulate and optimize the resulting code.

NVIDIA's CUDA C/C++ compiler, *nvcc*, emits PTX and is currently the preferred mechanism for programming OptiX. Programs are compiled offline using *nvcc* and submitted to the OptiX API as a PTX string. By leveraging the CUDA C++ compiler, OptiX shader programs have a rich set of programming language constructs available, including pointers, templates, and overloading that come automatically by using C++ as the input language. A set of header files is provided that support the necessary variable annotations and pseudo-instructions for tracing rays and other OptiX operations. These operations are lowered to PTX in the form of a *call* instruction that gets further processed by the OptiX runtime.

4.2. PTX to PTX compilation

Given the set of PTX functions for a particular scene, the OptiX compiler rewrites the PTX using multiple PTX to PTX transformation passes, which are similar to the compiler passes that have proven successful in the LLVM infrastructure. In this manner, OptiX uses PTX as an intermediate representation rather than a traditional instruction set. This process implements a number of domain-specific operations including an ABI (calling sequence), link-time optimizations, and data-dependent optimizations. The fact that most data structures in a typical ray tracer are read-only, provides a substantial opportunity for optimizations that would not be considered safe in a more general environment.

One of the primary steps is transforming the set of mutually recursive programs into a non-recursive state machine. Although this was originally done to allow execution on a device that does not support recursion, we found benefits in scheduling coherent operations on the SIMT device and now employ this transformation even on newer devices that have direct support for recursion. The main step in the transformation is the introduction of a *continuation*, which is the minimal set of data necessary to resume a suspended function.

The set of PTX registers to be saved in the continuation is determined using a backward dataflow analysis pass that determines which registers are live when a recursive call (e.g., *rtTrace*) is encountered. A live register is one that is used as an argument for some subsequent instruction in the dataflow graph. We reserve slots on a per-thread stack array for each of these variables, store them on the stack before the call and restore them after the call. This is similar to a caller-save ABI that a traditional compiler would implement for a CPU-based programming language. In preparation for introducing continuations, we perform a loop-hoisting pass and a copy-propagation pass on each function to help minimize the state saved in each continuation.

Finally, the call is replaced with a branch to return execution to the state machine described below, and a label that can be used to eventually return control flow to this function. Further detail on this transformation can be found in the original paper.

4.3. Optimization

The OptiX compiler infrastructure provides a set of domain-specific and data-dependent optimizations that would be challenging to implement in a statically compiled environment. These include:

- Elide transformation operations for node graphs that do not utilize a transformation node.
- Eliminate printing and exception related code if these options are not enabled in the current execution.
- Reduce continuation size by regenerating constants and intermediates after a restore. Since the OptiX execution model guarantees that object-specific variables are read-only, this local optimization does not require an interprocedural pass.
- Specialize traversal based on tree characteristics such as existence of degenerate leaves, degenerate trees, shared acceleration structure data, or mixed primitive types.
- Move small read-only data to constant memory or textures if there is available space.

Furthermore, the rewrite passes are allowed to introduce substantial modifications to the code, which can be cleaned up by additional standard optimization passes such as dead-code elimination, constant propagation, loop-hoisting, and copy-propagation.

5. EXECUTION MODEL

Fundamentally, ray tracing is a highly parallel MIMD operation. In any interesting rendering algorithm, rays will rapidly diverge even if they begin together in the camera model. At first blush, this is a challenge for GPUs that rely on SIMT execution for efficiency. However, it should be observed that execution divergence is only temporary; a ray that hits a glass material temporarily diverges from one that hits a painted surface, yet they both quickly return to the core operation of tracing rays - a refraction or reflection in the former case and a shadow ray in the latter.

Consequently, the state machine described in Section 4 provides an opportunity to reconverge after temporary divergence. To accomplish this, we link all of the transformed programs into a monolithic kernel, or *megakernel*, an approach that has proven successful on modern GPUs.[1] This approach minimizes kernel launch overhead but potentially reduces processor utilization as register requirements grow to the maximum across constituent kernels. OptiX implements a megakernel by linking together a set of individual user programs and traversing the state machine induced by execution flow between them at runtime.

5.1. Megakernel execution

A straightforward approach to megakernel execution is simple iteration over a switch-case construct. Inside each case, a user program is executed and the result of this computation is the case, or state, to select on the next iteration. Within such a state machine mechanism, OptiX may implement function calls, recursion, and exceptions.

Figure 4 illustrates a simple state machine. The program states are simply inserted into the body of the switch statement. The state index, which we call a *virtual program counter (VPC)*, selects the program snippet that will be executed next. Function calls are implemented by setting the VPC directly, virtual function calls are implemented by setting it from a table, and function returns simply restore the state to the continuation associated with a previously active function (the virtual return address). Furthermore, special control flows such as exceptions manipulate the VPC directly, creating the desired state transition in a manner similar to a lightweight version of the *setjmp/longjmp* functionality provided by C.

5.2. Fine-grained scheduling

While the straightforward approach to megakernel execution is functionally correct, it suffers serialization penalties when the state diverges within a single SIMT unit.[6] To mitigate the effects of execution divergence, the OptiX runtime uses a fine-grained scheduling scheme to reclaim divergent threads that would otherwise lay dormant. Instead of allowing the SIMT hardware to automatically serialize a divergent switch's execution, OptiX explicitly selects a single state for an entire SIMT unit to execute using a scheduling heuristic. Threads within the SIMT unit that do not require the state simply idle that iteration. The mechanism is outlined in Figure 5.

We have experimented with a variety of fine-grained scheduling heuristics. One simple scheme that works well determines a schedule by assigning a static prioritization over states. By scheduling threads with like states during execution, OptiX reduces the number of total state transitions made by a SIMT unit, which can substantially decrease execution time over the automatic schedule induced by the serialization hardware. Figure 6 shows an example of such a reduction.

As GPUs evolve, different execution models may become practical. For example, a streaming execution model[2] may be useful on some architectures. Other architectures may provide hardware support for acceleration structure traversal or other common operations. Since the OptiX engine does not

Figure 4. Pseudo-code for a simple state machine approach to megakernel execution. The state to be selected next is chosen by a switch statement. The switch is executed repeatedly until the *state* variable contains a special value that indicates termination.

```
state = initialState;
while( state != DONE )
  switch(state) {
    case 1:      state = program1();   break;
    case 2:      state = program2();   break;
    ...
    case N:      state = programN();   break;
  }
```

Figure 5. Pseudo-code for megakernel execution through a state machine with fine-grained scheduling.

```
state = initialState;
while( state != DONE ) {
  next_state = scheduler();
  if(state == next_state)
    switch(state) {
      // Insert cases here as before
    }
}
```

prescribe an execution order between the roots of the ray trees, these alternatives could be targeted with a rewrite pass similar to the one we presently use to generate a megakernel.

6. APPLICATION CASE STUDIES

This section presents some example use cases of OptiX by discussing the basic ideas behind a number of different applications. More examples can be found in Parker et al.[10]

6.1. Whitted-style ray tracing

The OptiX SDK contains several example ray tracing applications. One of these is an updated re-creation of Whitted's original sphere scene (Figure 7).[14] This scene is simple, yet demonstrates important features of the OptiX engine.

The sample's ray generation program implements a basic pinhole camera model. The camera position, orientation, and viewing frustum are specified by a set of program variables that can be modified interactively. The ray generation program begins the shading process by shooting a single ray per pixel or, optionally, performing adaptive antialiasing via supersampling. The material *closest-hit* programs are then responsible for recursively casting rays and computing a shaded sample color. After returning from the recursion, the ray generation program accumulates the sample color, stored in the ray payload, into an output buffer.

The application defines three separate pairs of intersection and bounding box programs, each implementing a different geometric primitive: a parallelogram for the floor, a sphere for the metal ball, and a thin-shell sphere for the hollow glass ball. The glass ball could have been modeled with two instances of the plain sphere primitive, but the flexibility of the OptiX program model gives us the freedom to implement a more efficient specialized version for this case. Each intersection program sets several attribute variables: a geometric normal, a shading normal, and, if appropriate, a texture coordinate. The attributes are utilized by material programs to perform shading computations.

The ray type mechanism is employed to differentiate radiance from shadow rays. The application attaches to the materials' *any-hit* slots for shadow rays, a trivial program that immediately terminates a ray. This early ray termination yields high efficiency for mutual visibility tests between a shading point and the light source. The glass material is an exception, however: here, the any-hit program is used to attenuate a visibility factor stored in the ray payload. As a result, the glass sphere casts a subtler shadow than the metal sphere.

Figure 6. The benefit of fine-grained scheduling with prioritization, as achieved when rendering 7. Bars represent the number of state executions per pixel. A substantial reduction can be seen by scheduling the state transitions with a fixed priority, as described in Section 5.2.

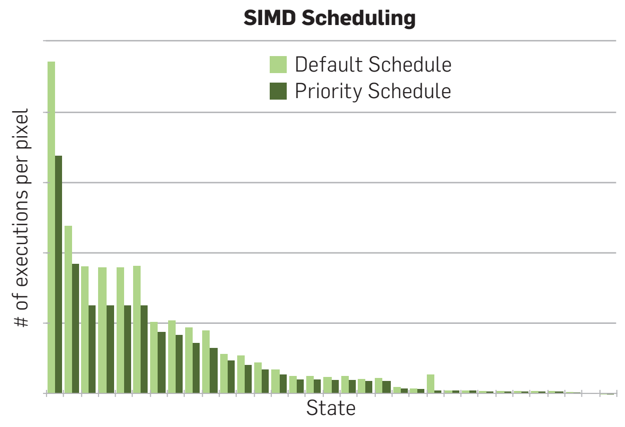
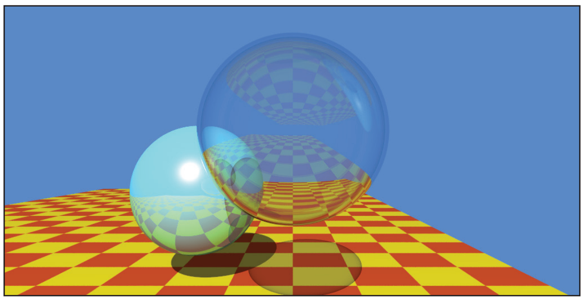


Figure 7. Re-creation of Whitted's sphere scene with user-specified programs: sphere and rectangle intersection; glass, procedural checker, and metal hit programs; sky miss program; and pinhole camera with adaptive anti-aliasing ray generation. Runs at over 100 fps on a GeForce GTX680 at 1k by 1k resolution.



6.2. NVIDIA design garage

NVIDIA Design Garage is a sophisticated interactive rendering demonstration intended for public distribution. The top image of Figure 2 was rendered using this software. The core of Design Garage is a physically-based Monte Carlo path tracing system[4] that continuously samples light paths and refines an image estimate by integrating new samples over time. The user may interactively view and edit a scene as an initial noisy image converges to the final solution.

To control stack utilization, Design Garage implements path tracing using iteration within the ray generation program rather than recursively invoking *rtTrace*. The pseudocode of Figure 8 summarizes.

In Design Garage, each material employs a closest-hit program to determine the next ray to be traced, and passes that back up using a specific field in the ray payload. The closest-hit program also calculates the throughput of the current light bounce, which is used by the ray generation to maintain the cumulative product of throughput over the complete light path. Multiplying the color of the light source hit by the last ray in the path yields the final sample contribution.

OptiX's support for C++ in ray programs allow materials to share a generic closest-hit implementation that implements

Figure 8. Pseudo-code for iterative path tracing in Design Garage.

```
float3 throughput     = make_float3( 1, 1, 1 );
payload.nextRay      = camera.getPrimaryRay();
payload.shootNextRay = true;

while( payload.shootNextRay == true ) {
  rtTrace( payload.nextRay, payload );
  throughput *= payload.throughput;
}
sampleContribution = payload.lightColor * throughput;
```

light-loops and other core lighting operations, while a specific Bidirectional Scattering Distribution Function (BSDF) model implements importance sampling and probability density evaluation. Design Garage implements a number of different physically-based materials, including metal and automotive paint. Some of these shaders support normal and specular maps.

While OptiX implements all ray tracing functionality of Design Garage, an OpenGL pipeline implements final image reconstruction and display. This pipeline performs various post processing stages such as tone mapping, glare, and filtering using standard rasterization-based techniques.

6.3. Image space photon mapping

Image space photon mapping (ISPM)[7] is a real-time rendering algorithm that combines ray tracing and rasterization strategies (Figure 9). We ported the published implementation to the OptiX engine. That process gives insight into the differences between a traditional vectorized serial ray tracer and OptiX.

The ISPM algorithm computes the first segment of photon paths from the light by rasterizing a "bounce map" from the light's reference frame. It then propagates photons by recursively ray tracing until the last scattering event before the eye. At each scattering event, the photon is deposited into an array that is the "photon map." Indirect illumination is then gathered in image space by rasterizing a small volume around each photon from the eye's viewpoint. Direct illumination is computed by shadow maps and rasterization.

Consider the structure of a CPU-ISPM photon tracer. It launches one persistent thread per core. These threads process photon paths from a global atomic work queue. ISPM photon mapping generates incoherent rays, so traditional packet strategies for vectorizing ray traversal do not help with this process. For each path, the processing thread enters a while-loop, depositing one photon in a global photon array per iteration. The loop terminates upon photon absorption.

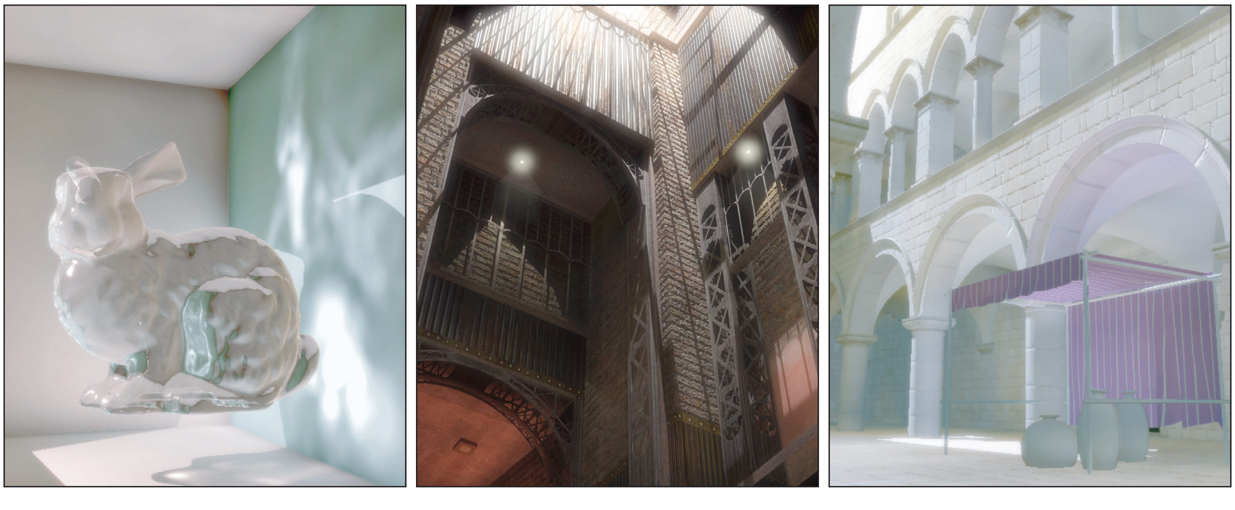
Trace performance increases with the success of fine-grain scheduling of programs into coherent units and decreases with the size of state communicated between programs. Mimicking a traditional CPU-style of software architecture would be inefficient under OptiX because it would require passing all material parameters between the ray generation and hit programs and a variable iteration while-loop in the closest-hit program. OptiX-ISPM therefore follows an alternative design that treats all propagation iterations as co-routines. It contains a single ray generation program with one thread per photon path. A recursive closest-hit program implements the propagate-and-deposit iterations. This allows threads to yield between iterations so that the fine-grained scheduler can regroup them.

7. SUMMARY AND FUTURE WORK

The OptiX system provides a general-purpose and high performance ray tracing API. OptiX balances ease of use with performance by presenting a simple programming model, based on a programmable ray tracing pipeline for single-ray user programs that can be compiled into an efficient self-scheduling megakernel. Thus the heart of OptiX is a JIT compiler that processes *programs*, snippets of user-specified code in the PTX language. OptiX associates these programs with nodes in a graph that defines the geometric configuration and acceleration data structures against which rays are traced. Our contributions include a low-level ray tracing API and associated programming model, the concept of a programmable ray tracing pipeline and the associated set of program types, a domain-specific JIT compiler that performs the megakernel transformations and implements several domain-specific optimizations, and a lightweight scene representation that lends itself to high-performance ray tracing and supports, but does not restrict, the structure of the application scene graph. The OptiX ray tracing engine is a shipping product and already supports a wide range of applications. We illustrate the broad applicability of OptiX with multiple examples ranging from simplistic to fairly complex.

While OptiX already contains a rich set of features and is suitable for many use cases, it will continue to be refined and improved. For example, we (or a third party developer) can add support for further high level input languages, i.e. languages that produce PTX code to be consumed by OptiX. In

Figure 9. ISPM real-time global illumination. A recursive closest-hit program in OptiX implements the photon trace.



addition to language frontends, we are planning support for further backends. Because PTX serves only as an intermediate representation, it is possible to translate and execute compiled megakernels on machines other than NVIDIA GPUs. OptiX has a CPU fallback path that employs this approach.

One downside of OptiX, like any compiler, is that performance of the compiled kernel does not always match a hand-tuned kernel for a specific use-case. We continue to explore optimization techniques to close that gap. The original paper discusses performance in more detail.

We have also discovered tradeoffs in the compile-time specialization of kernels that achieve high performance, but result in small delays when assumptions are violated and a kernel must be regenerated. In the future, the system may choose to fall back to a generalized kernel to maintain slightly degraded interactivity while a new specialized kernel is compiled.

Acknowledgments

The car, frog, and engine model in Figure 1 are courtesy of TurboSquid. The bunny model in Figures 3 and 9 is courtesy of the Stanford University Graphics Lab. Phil Miller was instrumental in keeping the effort on track. The authors benefited greatly from groundwork and numerous conversations on ray tracing with members of NVIDIA Research and the SceniX team. C

References

1. Aila, T., Laine, S. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics 2009* (2009), 145–149.
2. Gribble, C.P., Ramani, K. Coherent ray tracing via stream filtering. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2008), 59–66.
3. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (2007), ACM, New York, NY, USA, 167–174.
4. Kajiya, J.T. The rendering equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH)* (1986), 143–150.
5. Lattner, C., Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the 2004 International Symposium on Code Generation and Optimization* (2004).
6. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28* (2008), 39–55.
7. McGuire, M., Luebke, D. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics* (2009).
8. NVIDIA. PTX: Parallel Thread Execution ISA Version 2.3 (2011). http://developer.download.nvidia.com/-compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf.
9. NVIDIA. NVIDIA OptiX Ray Tracing Engine Programming Guide Version 2.5 (2012). http://www.nvidia.com/object/optix.html.
10. Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., Stich, M. OptiX: A general purpose ray tracing engine. In *ACM Transactions on Graphics (TOG) – Proceedings of ACM SIGGRAPH* (2010).
11. Popov, S., Günther, J., Seidel, H.P., Slusallek, P. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum, (Proceedings of Eurographics)*, vol. 26, no. 3 (Sept. 2007), 415–424.
12. Wald, I., Benthin, C., Wagner, M., Slusallek, P. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of Eurographics 2001)*, vol. 20, (2001)
13. Wald, I., Mark, W.R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S.G., Shirley, P. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007* (2007), 89–116.
14. Whitted, T. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

Steven G. Parker, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys, and Martin Stich ({sparker, hfriedrich, dluebke, kmorley, jbigler, jhoberock, davemc, arobison, adietrich, ghumphreys, mstich}@nvidia.com), NVIDIA, Santa Clara, CA.

Morgan McGuire (morgan@cs.williams.edu), NVIDIA and Williams College.