DOI:10.1145/2447976.2447997

GPU 光线跟踪

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

摘要

NVIDIA® OptiX™ 光线跟踪引擎是针对 NVIDIA GPU 及其他高度并行的体系结构设计的可编程系统。OptiX 引擎基于一个重要发现而构建，即大多数光线跟踪算法都可以使用一小组可编程运算来实现。因此，OptiX 的核心就是领域特定的即时编译器，该编译器可以将用户提供的程序（用于光线生成、材料着色、对象求交以及场景遍历）组合起来生成自定义的光线跟踪内核。这样就可以实现高度多样化的基于光线跟踪的算法和应用，包括交互式渲染、离线渲染、碰撞检测系统、人工智能查询以及科学模拟（如声传播）。OptiX 通过精简的对象模型和对光线跟踪特定的多种编译器优化技术的应用来实现高性能。为便于使用，它提供了一个全面支持递归的单光线编程模型和一个类似于虚拟函数调用的动态调度机制。

1. 简介

许多 CS 大学生都上过计算机图形课程，并编写过简单的光线跟踪器。只要了解有关光传输物理性质的一些简单概念，学生们就可以使用反射、折射、阴影和相机特效（如景深）来完成高品质图像，而所有这些都是当代实时图形流水线所面临的挑战。不幸的是，光线跟踪的计算负担使其在许多领域都成为了空谈，尤其在注重交互性的领域更是如此。研究人员发明了许多技术来改善光线跟踪的性能，[13]尤其在显式 SIMD 指令[12]和基于单指令多线程 (SIMT) 的[6] GPU 等涉及到高性能结构化特性的领域，这种现象表现得更为明显。[1]不幸的是，大多数这类技术都破坏了简单性和概念纯度，而这些正是光线跟踪的魅力所在。也没有出现行业标准来隐藏这些复杂性，就像 Direct3D 和 OpenGL 之于光栅化一样。

为了解决这些问题，我们引进了 OptiX，一个通用的光线跟踪引擎。通用的编程接口使图形和非图形领域（如渲染、声传播、碰撞检测和人工智能）的各种基于光线跟踪的算法能够得以实现。该接口在概念上很简单，但却能为现代 GPU 体系结构提供高性能，并且可以与手工编码方法媲美。

在本文中，我们将讨论 OptiX 引擎的设计目标以及一种针对 NVIDIA GPU 的实现。在我们的实现中，我们通过对场景层次结构、加速结构创建和遍历、实时场景更新进行灵活控制，使用动态负载平衡的 GPU 执行模型进行领域特定的编译。虽然 OptiX 主要针对高度并行的 GPU 体系结构，但它同样适用于各种专用和通用的硬件，包括现代 CPU。

1.1. 光线跟踪、光栅化和 GPU

适用于渲染或图像合成的计算机图形算法采用两种互补的方法之一。其中一类算法会遍历图像中的每个像素，以计算各像素处的第一个可见对象；这种方法就叫光线跟踪，因为它解决了对从像素至对象的光线进行求交的几何问题。另一类算法则是遍历场景中的每个对象，以计算各对象所覆盖的像素。由于每个对象的结果像素（称为片元）会针对光栅显示屏进行格式化，因此这种方法被称为光栅化。光线跟踪的中心数据结构是一个空间索引，称为加速结构，用于避免针对所有对象测试每条光线。光栅化的中心数据结构是深度缓存，其存储了每个像素最近可见对象的距离，并丢弃不可见对象的片元。虽然这两种方法都已得到了极力推广和优化（远非本文简短的描述所能涵盖），但仍存在基本的区别：光线跟踪遍历光线，而光栅化遍历对象。高性能的光线跟踪和光栅化都注重渲染最简单的对象：三角形。

一直以来，光线跟踪都被认为速度慢，而光栅化速度快。深度缓冲光栅化简单、规则的结构适用于高度并行的硬件实现：每个对象都会经过多个阶段的计算（即所谓的图形流水线），而流水线的每个阶段都会以数据并行的方式对许多对象、片元和像素实时地执行类似的计算。随着图形硬件变得更加并行化，它也变得更加普遍，正在从实现图形流水线各个阶段的专用型固定功能电路，演变成将这些阶段虚拟化为成百上千小型通用核心的完全可编程处理器。今天的图形处理单元 (GPU) 是大规模并行处理器，每秒能够执行数万亿次浮点数学运算和渲染数十亿个三角形。现代 GPU 的计算能力和功耗效率使其受到高性能计算的青睐，其中包括众多世界上最快的超级计算机、科学、数学和桌面工程代码等。所有这些领域都提出了一个问题：GPU 上能够高效且灵活地实现光线跟踪吗？

本文的原始版本名为“OptiX:A General Purpose Ray Tracing Engine”（OptiX：通用的光线跟踪引擎），于 2010 年 7 发表在 ACM 的 *ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH* 上。

1.2. 要求和设计目标

为了为各种光线跟踪任务创建高性能系统，本文通过提出一些权衡和设计决策作出了如下贡献：

- *通用的低级光线跟踪引擎*。OptiX 不是一个渲染器。它只专注于光线跟踪所需的基本计算，避免嵌入、渲染特定的构造，如照明、阴影和反射。
- *可编程的光线跟踪流水线*。OptiX 告诉我们，大多数光线跟踪算法都可以使用一小组轻量级可编程运算来实现。它定义了一个抽象的光线跟踪执行模型，以作为用户程序的执行序列，类似于传统的基于光栅化的图形流水线。
- *简单的编程模型*。OptiX 不会用高性能光线跟踪算法给用户增加负担。它提供了一个用户熟悉的单光线递归编程模型，而不是光线包或显式向量构造，并对光线的批处理或重排进行抽象化。
- *领域特定的编译器*。OptiX 引擎结合了即时编译技术和光线跟踪特定的知识，可高效地实现其编程模型。引擎的抽象化允许编译器针对可用的系统硬件调整执行模型。

2. 相关工作

虽然已经提出了众多高级的光线跟踪库、引擎和 API，[13]但迄今为止的努力一直专注于特定的应用程序或各类渲染算法，使得它们难以适应其他领域或体系结构。另一方面，一些研究人员已经提出了如何将光线跟踪算法有效地映射到 GPU和 NVIDIA® CUDA™ 体系结构[1, 3, 11]的方法，但这些系统注重的是性能而非灵活性。

原始论文对相关系统和研究做了进一步讨论。[10]

3. 可编程的光线跟踪流水线

OptiX 引擎背后的核心思想是：大多数光线跟踪算法都可以使用一小组可编程运算来实现。这直接类似于 OpenGL 和 Direct3D 使用的可编程光栅化流水线。在较高级别上，这些系统提供了一个抽象的光栅器，包含适用于顶点着色、几何处理、曲面细分以及像素着色运算的轻量级回调机制。这些程序类型（通常用在多个阶段中）组合在一起可以实现各种基于光栅化的算法。

我们已经确定了相应的可编程光线跟踪执行模型和可通过定制实现各种基于光线跟踪的算法的轻量级运算。[9]用户提供的这些运算（简称程序）可以与用户定义的与各光线关联的数据结构（负载）进行组合。将程序组合起来可以实现特定客户端应用程序的算法。

3.1 程序

OptiX 包括七种不同类型的程序，其中每种类型理论上每次只对一条光线进行运算。此外，包围盒程序会进行几何运算，以确定加速结构构造的原始边界。用户程序与硬编码的 OptiX 内核代码的组合就形成了光线跟踪流水线，如图 2 所示。与前馈型光栅化流水线不同，我们可以更自然地将光线跟踪流水线视为调用图。核心运算 rtTrace 与交点定位（遍历）和交点响应（着色）相互交替。通过在用户定义的负载和全局设备内存阵列（称为缓存）中读写数据，这些运算可在光线跟踪期间组合起来执行任意计算。

*光线生成*程序是光线跟踪流水线的入口。从主机调用一次 *rtContextLaunch* 将创建这些程序的多个实例。一个典型的光线生成程序将使用摄像机模型为像素中的单个样本创建一条光线，并启动追踪运算，然后将结果颜色存储在输出缓存中。但通过将光线生成与图像中的像素进行区分，OptiX 还启用了其他运算，如创建光子贴图、预先计算照明纹理贴图（也称烘焙）、处理从 OpenGL 传递的光线请求、为超采样发射多条光线或实现不同的摄像机模型。

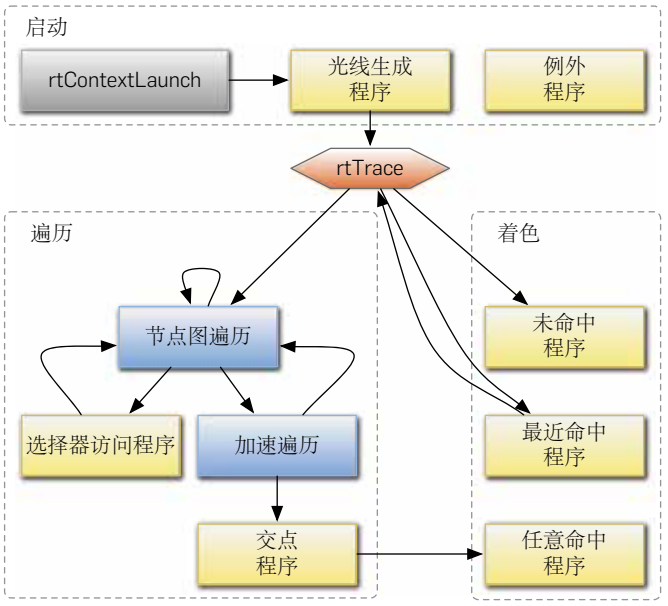
*求交*程序实现光线几何求交测试。遍历加速结构时，系统将调用求交程序执行几何查询。该程序确定光线是否以及在何处触及对象，并可根据命中位置计算法线、纹理坐标或其他属性。每个交点可关联任意数量的属性。除多边形和三角形外，求交程序还支持任意曲面，如置换贴图、球体、圆柱体、高阶曲面，甚至分形几何形状，如图 1 所示的 Julia 集。可编程的求交运算即使在纯三角形系统中也是非常有用的，因为它支持直接访问原生网格格式。

遍历运算找到光线与场景几何体的最近交点后将调用最近命中程序。该程序类型类似于经典渲染系统的表面着色器。通常情况下，“最近命中”程序将执行着色等运算（可能会在这个过程中投射新的光线），并将结果数据存储在光线负载中。

在遍历过程中，每当找到光线–对象交点时都将调用任意命中程序。“任意命中”程序允许材料参与对象求交决策，同时使着色运算与几何运算分开进行。它可使用内建函数 *rtTerminateRay* 选择终止光线，这将停止所有遍历并释放最近一次 *rtTrace* 调用的调用堆栈。这是一个轻量级的异常机制，可用于实现阴影光线和环境光遮蔽的早期光线终止。另外，“任意命中”程序可使用 *rtIgnoreIntersection* 忽略交点，从而允许遍历运算继续查找其他几何对象。例如，程序可以选择根据纹理通道查找来忽略交点，以实现高效的 alpha 映射透明，而无需重新启动遍历运算。第 6.1 节中介绍了“任意命中”程序的另一个用例，其中应用程序为玻璃对象投射的部分阴影执行能见度衰减。请注意，交点可

图 1来自使用 OptiX 构建的各种应用程序的图像。顶部：通过路径追踪的基于物理的光传输。底部：过程 Julia 集的光线跟踪、光子映射、大规模视线和碰撞检测、动态几何的 Whitted 式光线跟踪和光线跟踪的环境光遮蔽。所有应用程序均为交互式应用程序。

图 2显示贯穿整个光线跟踪流水线的控制流的调用图。黄色方框表示用户指定的程序，蓝色方框表示 OptiX 的内部算法。执行由 API 调用 rtContextLaunch 启动。光线生成程序可使用内建函数 rtTrace 将光线投射到场景中。“最近命中”程序也可以递归方式调用该函数来投射阴影和二级光线。当特定光线的执行因错误（例如内存消耗过大）被终止时，将执行异常程序。



能会显得杂乱无章。默认的“任意命中”程序是个空运算，而这往往就是理想的运算。

如果在所提供的间隔内，光线未与任何几何体相交，将执行未命中程序。这些程序可用于实现背景颜色或环境贴图查找。

当系统遇到异常状况时（例如递归堆栈超出每个线程可用的内存量，或者缓存访问索引超出范围），将执行异常程序。OptiX 还允许用户自定义任何程序可抛出的异常。例如，“异常”程序可通过打印诊断消息或可视化异常状况（通过将特殊的颜色值写入到输出像素缓存）来进行响应。

选择器访问程序为粗粒度节点图遍历提供可编程性。例如，对于每条光线，应用程序可以选择将场景中部分几何体的细节粒度多样化。

3.2 场景呈现

OptiX 一个明确的目标就是将场景表示的开销降到最低，而不是将重量级的场景图强加给用户。OptiX 引擎使用一个简单而灵活的结构来表示场景信息和相关的可编程运算，并将它们收集在一个称为环境的容器对象中。这种表示方法也是可编程着色器与其所需对象特定数据的绑定机制。

层次结构节点。 我们用控制场景光线遍历的轻量级图形来表示场景。它也可用于实现刚性对象动画的两层结构或其他常见场景结构的实例化。为了支持公用数据的实例化和共享，节点可以有多个父节点。

可使用四个主要的节点类型和有向图来表示场景。任何节点都可用作场景遍历的根节点。这样就可以针对不同的光线类型使用不同的表示方法

组节点包含零个或多个（但通常是两个或两个以上）任意节点类型的子节点。组节点关联了一个加速结构，并可用于提供两层遍历结构的顶层。

几何组节点是图形的叶节点，包含如下所述的图元对象和材料对象。该节点类型也有一个与之

关联的加速结构。任何非空场景将包含至少一个几何组。

变换节点具有一个任意类型的子节点以及相关联的4×3矩阵，该矩阵用于执行基本几何体的仿射变换。

选择器节点有零个或多个任意节点类型的子节点和一个用于选择可用子节点的访问程序。

几何对象和材料对象。 大容量场景数据存储在图形叶节点处的几何节点中。它们包含定义几何运算和着色运算的对象。它们也可以具有多个父节点，允许在图形中的多个点共享材料和几何信息。请参见图 3 中的示例。图中显示了一个简单场景的完整 OptiX 环境，场景包含一个针孔摄像机，两个对象和阴影。光线生成程序实现了摄像机，而“未命中”程序实现了恒定的白色背景。一个几何组包含两个几何实例和一个几何索引 – 本例中为包围体层次结构 (BVH) – 构建于三角形网格和地平面的所有基本几何体上。这个例子实现了两个类型的几何体，即三角形网格和平行四边形，它们都具有各自的一组求交程序和包围盒程序。这两个几何实例共享了一个材料，该材料通过“最近命中”程序实现了漫射照明模型，并通过“任意命中”程序使阴影光线完全衰减。

左侧图 3 中的图表显示了如何为遍历场景的 3 条光线调用这些程序。1.光线生成程序创建光线并根据几何组进行追踪。这将启动图 2 中所示的“遍历”阶段，并执行平行四边形和三角形网格求交，直到找到交点（2 和 3 ）。如果光线与几何体相交，则无论在地平面还是三角形网格中找到交点，都将调用“最近命中”程序。材料将以递归方式生成显示光线，以确定光源是否没有被遮挡。4.如果在阴影光线上发现交点，则“任意命中”程序将终止光线遍历，并带着阴影遮蔽信息返回到调用程序。5.如果光线未与任何场景几何体相交，将调用“未命中”程序。

*几何实例*对象将几何对象绑定到一组*材料*对象。这是一种常见的结构，场景图用它来使几何信息与着色信息保持正交。

几何对象包含一系列几何图元。每个几何对象都关联了一个包围盒程序和一个求交程序，它们都在几何对象的图元中共享。

材料对象包含有关着色运算的信息，包括第 3.1 节中所述的“任意命中”和“最近命中”程序。

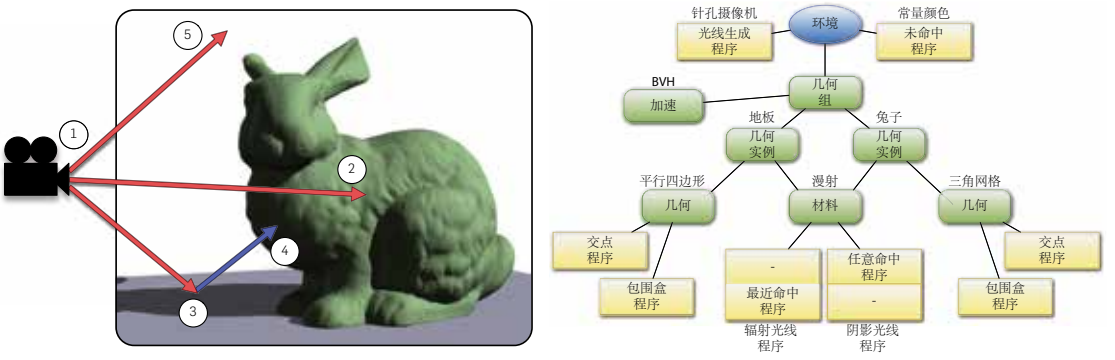
3.3. 系统概述

OptiX 引擎有两套不同的 API。*主机 API* 是一个函数集，供客户端应用程序调用来创建和配置环境、组装节点图和启动光线跟踪内核。它还提供用于管理 GPU 设备的调用。*程序 API* 是暴露给用户程序的功能。它包括用于追踪光线、报告交点和访问数据的函数调用。此外，多个语义变量会对特定于光线跟踪的状态（例如，到最近交点的当前距离）进行编码。还提供了用于调试的打印和异常处理机制。

使用 OptiX 主机和 API 函数提供场景数据（如几何体、材料、加速结构、层次结构关系和程序）后，应用程序将使用将控制传递给 OptiX 的 *rtContextLaunch* API 函数来启动光线跟踪。如果需要，将从指定的用户程序编译新的光线跟踪内核，并构建（或更新）加速结构，然后在主机与设备内存之间同步数据，最后执行该光线跟踪内核，以调用如上所述的各种用户程序。

执行完光线跟踪内核后，应用程序可使用结果数据。通常，这涉及从其中一个用户程序填充的输出缓存读取数据，或直接显示此类缓存（例如，通过 OpenGL ）。然后，交互式或多阶段应用程序从环境设置开始重复该过程，其间可以对环境进行任意更改，然后再次启动内核。

图 3 OptiX 场景构造和执行示例。



4. 领域特定的编译

OptiX 主机运行时的核心是提供许多重要功能的即时 (JIT) 编译器。首先，JIT 阶段会将用户提供的所有着色器程序整合到一个或多个内核中。其次，它会分析节点图以确定依赖于数据的优化方法。最后，使用 CUDA 驱动程序 API 在 GPU 上执行结果内核。

大规模并行体系结构的代码生成和优化带来了一些挑战。其中一个挑战是，每次计算的代码大小和活动状态必须最小化，以获得最高性能。另一个挑战是结构化代码以减少分支结构。我们使用 OptiX 的经验更突出了这些有时相互冲突的要求之间的有趣关系。

4.1. OptiX 程序

第 3.1 节中所述的用户程序以并行线程执行 (PTX) 函数的形式提供给 OptiX 主机 API。[8]PTX 是 NVIDIA CUDA 体系结构的虚拟机汇编语言，在许多方面都类似于流行的开源低级虚拟机 (LLVM) 的中间语言。[5] 与 LLVM 一样，PTX 也定义了一套简单的指令，为算术、控制流和内存访问提供基本运算。PTX 还提供了许多高级运算，例如纹理访问和超越运算。同样类似于 LLVM，PTX 也假定了一个无穷大的寄存器文件并将许多真实机器指令抽象化。CUDA 运行时中的 JIT 编译器将执行寄存器分配、指令调度、死代码消除以及许多其他后期优化，因为它是面向特定的 GPU 体系结构生成机器码。

PTX 由于是从单线程的角度编写，因此不需要显式通道掩码操作运算。这使得 PTX 直接成为有别于高级着色语言的低级语言，同时使 OptiX 运行时能够操作和优化结果代码。

NVIDIA 的 CUDA C/C++ 编译器 *nvcc* 可生成 PTX，是目前 OptiX 编程的首选机制。程序将使用 *nvcc* 进行脱机编译，并作为 PTX 字符串提交至 OptiX API。通过使用 CUDA C++ 编译器，OptiX 着色器程序有了一套丰富的编程语言构造，包括指针、模板以及重载，通过使用 C++ 作为输入语言，将自动获得这些功能。提供了一组头文件，支持光线跟踪和其他 Optix 运算所需的变量注释和伪指令。这些运算以调用指令的形式被降级为 PTX，该指令将由 OptiX 运行时做进一步处理。

4.2. PTX 到 PTX 的编译

鉴于 PTX 函数集是针对特定场景，OptiX 编译器将使用多个 PTX 到 PTX 变换阶段重写 PTX，这些阶段类似于 LLVM 基础设施中已证明成功的编译器阶段。在这种方式中，OptiX 将 PTX 用作中间语言，而不是传统的指令集。这个过程实现了许多领域特定的运算，包括 ABI（调用序列）、链接时优化和依赖于数据的优化。在典型的光线跟踪器中，大多数数据结构都是只读的，这就提供了大量的优化机会，而在更为通用的环境中，这些优化被认为是不安全的。

其中一个主要步骤是将相互递归的程序集变换为非递归的状态机。虽然这原本是为了允许在不支持递归的设备上执行，但我们发现了在 SIMT 设备上调度相干运算的好处，所以现在就算是直接支持递归的新设备，我们也会对其使用这种变换。变换的主要步骤是引入*延续性结构*，这是恢复已挂起函数所需的最少数据。

将保存在延续性结构中的 PTX 寄存器通过向后数据流分析阶段来确定，该阶段确定遇到递归调用（例如 *rtTrace*）时哪些寄存器是活动的。活动寄存器是指作为数据流图中一些后续指令的参数使用的寄存器。我们在每线程堆栈数组中为每个变量都保留了数组槽，在调用之前将它们存储在堆栈中，并在调用后还原。这类似于调用方保存的 ABI，传统编译器为基于 CPU 的编程语言对其进行实现。在准备引入延续性结构的过程中，我们对每个函数执行循环外提阶段和复制传播阶段，以帮助最小化各延续性结构中保存的状态。

最后，调用将被一个分支结构替代，以将执行权交还给如下所述的状态机，同时可使用标签将控制流最终交还给该函数。原始论文对该变换进行了进一步的详细介绍。

4.3. 优化

OptiX 编译器基础设施提供了一系列领域特定和依赖于数据的优化方法，这些方法在静态编译环境中实现起来有些难度。这些方法包括：

- 为不使用变换节点的节点图忽略变换运算。
- 如果当前执行中未启用这些选项，删除与打印和异常相关的代码。
- 通过在还原后重新生成常量和中间代码来减少延续性结构的大小。由于 OptiX 执行模型可确保对象特定的变量为只读，所以该局部优化不需要过程间阶段。
- 根据树的特征（例如，是否存在退化的叶节点、退化的树、共享的加速结构数据，或混合图元类型）专用化遍历运算。
- 如果有可用空间，将少量的只读数据移动到常量内存或纹理中。

此外，重写阶段可对代码进行重大修改，并且可通过额外的标准优化阶段（例如死代码消除、常量传播、循环外提和复制传播）对代码进行清理。

5. 执行模型

从根本上讲，光线跟踪是一种高度并行的 MIMD 运算。在任何相关的渲染算法中，光线都将迅速发散，即使它们都始于摄像机模型。乍看之下，对于依赖 SIMT 执行来获得效率的 GPU 而言，这是一个挑战。但应该注意到的是，执行发散只是暂时的；触及玻璃材料的光线会从触及着色表面的光线发生暂时性的发散，但很快它们都会返回到光线跟踪的核心运算 – 折射或反射（前一种情况），阴影光线（后一种情况）。

因此，第 4 节中所描述的状态机提供了一个暂时发散后再次聚合的机会。为做到这一点，我们将所有已变换的程序链入一个单片内核或巨型内核}，这是一种在现代 GPU 上被证明成功的方法。[1]这种方法可以将内核启动开销降到最低，但当成员内核中寄存器需求增长至最大时，可能会降低处理器的利用率。OptiX 通过将一系列独立的用户程序链接起来并遍历运行期间它们之间的执行流所引发的状态机来实现了一个巨型内核。

5.1. 巨型内核执行

巨型内核执行的直接方法就是多条件分支语句 (switch-case) 的简单循环。在每个条件内，执行一个用户程序，且计算结果就是下一个循环要选择的条件或状态。在这样一个状态机机制中，OptiX 可以实现函数调用、递归和异常。

图 4 显示了一个简单状态机。将程序状态插入到 switch 语句的正文即可。我们称为*虚拟程序计数器* (*VPC*) 的状态索引选择下一步要执行的程序片段。函数调用通过直接设置 VPC 来实现，而虚拟函数调用通过从表中设置 VPC 来实现，函数返回值将状态返回给与前一个活动函数相关联的延续性结构（虚拟返回地址）。此外，特殊控制流（如异常）直接操作 VPC，并以类似于 C 提供的 *setjmp/longjmp* 功能轻量级版本的方式创建所需的状态转换。

5.2. 细粒度调度

虽然巨型内核执行的直接方法在功能上是正确的，但当在 SIMT 单元中状态发生发散时，它会遇到序列化带来的问题。[6]为了减轻执行发散的影响，OptiX 运行时使用了细粒度的调度方案来回收可能会休止的发散线程。OptiX 不允许 SIMT 硬件自动序列化发散 switch 语句的执行，而是为整个 SIMT 单元明确地选择一个状态来用调度启发式算法执行。SIMT 单元内的线程不需要状态闲置该循环。图 5 中对机制进行了描述。

我们已经尝试了多种细粒度的调度启发式算法。一个行之有效的简单方案通过指定静态的状态优先级机制来确定调度。通过在执行期间使用类似的状态调度线程，OptiX 减少了 SIMT 单元执行的状态转换总次数，从而能够大幅缩短序列化硬件引发的自动调度的执行时间。图 6 显示了这种减少的示例。

随着 GPU 的发展，不同的执行模型可能成为现实。例如，流执行模型[2]在某些体系结构上可能很有用。其他体系结构可为加速结构遍历或其他常见的运算提供硬件支持。由于 OptiX 引擎在光线树的根节点之间并没有规定执行顺序，因此这些替代方案应具有类似于我们目前用来生成巨型内核的重写阶段。

6. 应用案例分析

本节介绍了一些 OptiX 使用案例，并讨论了几个不同应用背后的基本思想。更多示例请参阅由 Parker 等人编著的论文。[10]

6.1. Whitted 式光线跟踪

OptiX SDK 包含多个光线跟踪应用程序的示例。其中一个示例就是更新和重建 Whitted 的原始球体场景（图 7）。[14]该场景虽然简单，但却展示了 OptiX 的重要功能。

该示例的光线生成程序实现了基本的针孔摄像机模型。摄像机位置、方向和视体由一组可以交互方式修改的程序变量指定。光线生成程序在开始着色过程时，首先会针对每个像素发射一条光线，或可选择地通过超级采样执行自适应抗锯齿。然后，材料的最近命中程序负责以递归方式

图 4用于执行巨型内核的简单状态机方法的伪代码。下一个被选择的状态通过一个 switch 语句来选择。switch 语句将被重复执行，直到 state 变量包含表示终止的值。

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

图 5用于通过状态机以细粒度调度方式执行巨型内核的伪代码。

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

投射光线并计算已着色的样品颜色。从递归返回后，光线生成程序将光线负载中存储的样品颜色收集到输出缓存中。

应用程序定义了三对独立的求交程序和包围盒程序，每对都实现了不同的几何图元：平行四边形（地板）、球体（金属球）以及薄壳球体（空心玻璃球）。玻璃球本可以用纯球体图元的两个实例来建模，但在本例中，OptiX 程序模型的灵活性使我们可以自由地实现更为高效的专用版本。每个求交程序都设置了多个属性变量：几何法线、着色法线以及纹理坐标（如果适用）。材料程序使用这些属性来执行着色计算。

光线类型机制被用来区分辐射光线和阴影光线。针对阴影光线，应用程序将可立即终止光线的小程序附加到材料的任意命中数组槽中。该早期光线终止可为着色点与光源之间的相互能见度测试产生高效率。但玻璃材料是个例外：在这里，“任意命中”程序用于衰减光线负载中存储的能见度因子。其结果是，玻璃球体投射出比金属球体更加微弱的阴影。

图 6具有优先级机制的细粒度调度的好处（如渲染 7 所达到的效果）。竖条表示每个像素的状态执行次数。可以看到，通过使用固定的优先级来调度状态转换，执行次数大大减少，如第 5.2 节所述。

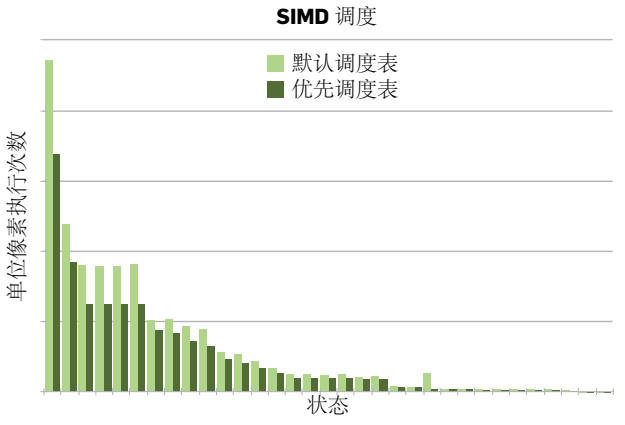
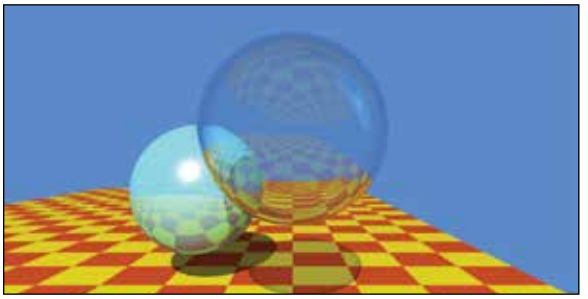


图 7使用用户指定的程序重新创建 Whitted 的球体场景：球体和矩形求交；玻璃、过程检查及金属命中程序；“天空未命中”程序；以及带自适应抗锯齿光线生成功能的针孔摄像机。在 GeForce GTX680 上以 100 fps 的帧率运行，分辨率为 1000 X 1000。



6.2. NVIDIA Design Garage

拟向公众发行的 NVIDIA Design Garage 是交互式渲染技术的先进代表。图 2 顶部的图像就是使用该软件渲染而成。Design Garage 的核心是基于物理的蒙特卡罗路径追踪系统[4]，该系统对光线路径进行连续采样，并通过整合随时间产生的样本精化图像估计值。在初始带噪图像演变成最终面貌的过程中，用户可以交互方式查看和编辑场景。

为了控制堆栈利用率，Design Garage 通过在光线生成程序中使用循环（而不是以递归方式调用 *rtTrace*）来实现路径追踪。图 8 中的伪码进行了概述。

在 Design Garage 中，每种材料都使用“最近命中”程序来确定要追踪的下一条光线和阶段，这些阶段将在光线负载中使用特定字段进行备份。“最近命中”程序还会计算当前光反弹的吞吐量，光线生成程序用它来计算整个光路中的累计吞吐量。将被命中的光源的颜色与路径中的上一条光线相乘即可得到最终的样本贡献。

由于 OptiX 在光线程序中支持 C++，这使得材料可以共享通用的“最近命中”实现，以此来实现光循环及其他核心光照运算，同时特定的双向散射分布函数（BSDF）模型实现了重要性采样和概率密度评估。Design Garage 实现了许多基于物理的不同材料，包括金属和汽车漆。其中一些着色器支持普通贴图和镜面贴图。

OptiX 实现了 Design Garage 的所有光线跟踪功能，而 OpenGL 流水线实现了最终的图像重建和显示。该流水线执行各种后期处理阶段，例如使用基于标准光栅化的技术完成色调映射、眩光以及过滤。

6.3. 图像空间光子映射

图像空间光子映射 (ISPM)[7] 是一种将光线跟踪与光栅化策略进行组合的实时渲染算法（图 9）。我们将已公布的实现移植到了 OptiX 引擎中。在这个过程中，我们深入了解了传统的向量化串行光线跟踪与 OptiX 的区别。

ISPM 算法通过光栅化光参照系的“反弹贴图”来计算光中光子路径的第一个片元。然后，它会通过

图 8Design Garage 中循环路径追踪的伪代码。

```
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;
```

递归的光线跟踪传播光子，直到光子进入视点之前的最后一个散射事件。在每个散射事件中，光子都会被存入到一个数组中，即“光子贴图”。然后，通过从视点方向对每个光子光栅化一个小的包围体来在图像空间收集间接光照。直接照明通过阴影贴图和光栅化来计算。

考虑 CPU-ISPM 光子追踪器的结构。它会为每个核心启动一个持续的线程。这些线程会处理全局原子工作队列中的光子路径。由于 ISPM 光子映射生成不相干的光线，所以用于向量化光线遍历的传统包策略对该过程没什么用。对于每个路径，处理线程都会进入一个 while 循环，每次循环都会将一个光子存入到一个全局光子数组中。光子吸收后循环终止。

追踪性能与相干单元中程序的细粒度调度的成功程度成正比，与程序之间传递的状态大小成反比。在 OptiX 中，模仿传统 CPU 风格的软件体系结构是低效的，因为它需要在光线生成和命中程序与“最近命中”程序中的变量循环（while 循环）之间传递所有材料参数。因此，OptiX-ISPM 采用了将所有传播循环视为协同程序的替代设计。它包含了一个光线生成程序，每个光子路径一个线程。支持递归的“最近命中”程序实现了“传播和保存”循环。这允许线程在各循环之间暂停，以便细粒度调度程序可以对它们重新分组。

7. 总结和展望

OptiX 系统提供了通用的高性能光线跟踪 API。OptiX 基于适用于单光线用户程序（可编译成高效的自调度巨型内核）的可编程光线跟踪流水线，提供了简单的编程模型，在易用性与性能之间找到了最佳平衡点。因此，OptiX 的核心就是处理程序（用户指定的 PTX 代码片段）的 JIT 编译器OptiX 将这些程序与一个图形中的节点进行关联，该图形根据所追踪的光线定义了几何配置和加速数据结构。我们的贡献包括：低级光线跟踪 API 和相关的编程模型、可编程光线跟踪流水线的概念和相关的程序类型集、执行巨型内核变换并实现多个领域特定优化的领域特定 JIT 编译器、适用于高性能光线跟踪并支持（但不限制）应用程序场景图结构的轻量级场景表示技术。OptiX 光线跟踪引擎是一款已经发行的产品，并且已经支持多种应用程序。我们使用从简单到相当复杂的多个示例，说明了 OptiX 广泛的适用性。

虽然 OptiX 已经包含了一系列丰富的功能，并且适用于许多使用情况，但它还将继续改进和完善。例如，我们（或第三方开发商）可以添加对更高级的输入语言（即生成由 OptiX 使用的 PTX 代码的语言）的支持。除了前端语言，我们还计划支持更多后端语言。由于 PTX 只是一门中间语言，用户可能想要在 NVIDIA GPU 以外的其他机器上解释和执行已编译的巨型内核。OptiX 有一个使用该方法的 CPU 回退路径。

与其他编译器一样，OptiX 的一个缺点是，在某些特定情况下，已编译内核的性能并不总能与手工调试的内核相媲美。我们将继续探索优化技术，以弥补这个短板。原始论文对性能进行了更详尽的讨论。

我们也在内核的编译时专用化中发现了折衷方案，该方案可以实现高性能，但当不符合假定的条件时可能会导致较小的延迟，而且必须重新生成内核。在未来，当编译新的专用内核时，系统可能会选择回退到通用化的内核，而牺牲一点交互性。

致谢

图 1 中的车、青蛙和引擎模型图由 TurboSquid 友情提供。图 3 和图 9 中的兔子模型由斯坦福大学图形实

图 9ISPM 实时全局光明。OptiX 中支持递归的“最近命中”程序实现了光子追踪。

验室 (Stanford University Graphics Lab) 友情提供。感谢 Phil Miller 的卓越贡献。作者从与 NVIDIA 研究 (NVIDIA Research) 和 SceniX 团队成员的多次有关光线跟踪的交谈中受益匪浅。

参考文献

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, 和 Martin Stich ({sparker, hfriedrich, dluebke, kmorley, jbigler, jhoberock, davemc, arobison, adietrich, ghumphreys, mstich}@nvidia.com), NVIDIA, 加利福尼亚州圣克拉拉。

Morgan McGuire (morgan@cs.williams.edu), NVIDIA 和威廉姆斯学院。