Journal of System Simulation

Volume 29 | Issue 2 Article 6

6-1-2020

Reconfigurable RTGPU's Architecture and Simulation

Xiaopeng Cao
1. School of Computer Science and Technology, Xidian University, Xi'an 710061, China; ;2. Department of computer, Xi'an University of Posts and Telecommunications, Xi'an 710121, China;

Jungang Han
2. Department of computer, Xi'an University of Posts and Telecommunications, Xi'an 710121, China;

Follow this and additional works at: https://dc-china-simulation.researchcommons.org/journal

Part of the Artificial Intelligence and Robotics Commons, Computer Engineering Commons, Numerical Analysis and Scientific Computing Commons, Operations Research, Systems Engineering and Industrial Engineering Commons, and the Systems Science Commons

This Paper is brought to you for free and open access by Journal of System Simulation. It has been accepted for inclusion in Journal of System Simulation by an authorized editor of Journal of System Simulation.

Reconfigurable RTGPU's Architecture and Simulation

Abstract

Abstract: Although GPU has developed rapidly, the core still uses rasterization and it is a hard problem for the complex scene. The ray tracing can simulate the geometrical optics and get the better effect, but it needs the amounts of computing power. Thus, the characteristic of the ray tracing was studied and *the new pipeline was designed. A dynamic reconfigurable hardware architecture* (*RTGPU*) was *put forward mapping the pipeline to it.* A simulation platform was designed. According to results of experiments, more than 11x on RTGPU could be gained achieving the better performance than GPU.

Keywords

ray tracing, reconfigurable, pipeline, hardware architecture, simulation

Recommended Citation

Cao Xiaopeng, *Han Jungang*. Reconfigurable RTGPU's Architecture and Simulation[J]. Journal of System Simulation, 2017, 29(2): 273-281.

This paper is available in Journal of System Simulation: https://dc-china-simulation.researchcommons.org/journal/vol29/iss2/6

第 29 卷第 2 期　　系统仿真学报©　　Vol. 29 No. 2
2017 年 2 月　　Journal of System Simulation　　Feb., 2017

可重构的光线追踪处理器架构模型及仿真

曹小鹏 [1,2]，韩俊刚 [2]
（1. 西安电子科技大学计算机学院，陕西 西安 710061；2. 西安邮电大学计算机学院，陕西 西安 710121）

摘要：虽然图形处理器发展迅速，但核心染色算法仍然沿用传统光栅化方法，对复杂光照的精确模拟效果较差。光线追踪算法模拟几何光学，能够实现较好的视觉效果，但运算量大。研究光线追踪算法的特点，对算法中各部分的计算量进行了定量分析，*设计了新的流水线。提出了一种动态可重构的光线追踪图形处理器(RTGPU)架构模型，设计了将流水线映射到该架构的算法。*开发了仿真平台，对流水线、硬件架构模型及映射算法进行了仿真。结果表明，在算法执行效率上，与 CPU 相比，RTGPU 执行加速比达到了 11，与 GPU 相比，执行效率显著提高。

关键词：光线追踪；可重构；流水线；硬件架构；仿真
中图分类号：TP391.9　　文献标识码：A　　文章编号：1004-731X (2017) 02-0273-09
DOI: 10.16182/j.issn1004731x.joss.201702006

Reconfigurable RTGPU's Architecture and Simulation

Cao Xiaopeng[1,2], *Han Jungang*[2]
(1. School of Computer Science and Technology, Xidian University, Xi'an 710061, China;
2. Department of computer, Xi'an University of Posts and Telecommunications, Xi'an 710121, China)

Abstract: Although GPU has developed rapidly, the core still uses rasterization and it is a hard problem for the complex scene. The ray tracing can simulate the geometrical optics and get the better effect, but it needs the amounts of computing power. Thus, the characteristic of the ray tracing was studied and *the new pipeline was designed. A dynamic reconfigurable hardware architecture* (*RTGPU*) was *put forward mapping the pipeline to it.* A simulation platform was designed. According to results of experiments, more than 11x on RTGPU could be gained achieving the better performance than GPU.

Keywords: ray tracing; reconfigurable; pipeline; hardware architecture; simulation

引言

随着图形处理技术，集成电路技术的进步，图形处理器(GPU)也在日新月异的发展，GPU 已经从基于 OpenGL 的流水线架构逐渐转变为可编程的统一渲染器架构，但其核心的 3D 渲染技术还一直沿用传统的光栅化方法。光栅化方法是将计算机图形渲染过程抽象为图形处理流水线，包括顶点染色、投影变换、视窗变换、扫描转换、像素染色等，最终转化成屏幕上的像素点进行渲染来产生图像，计算量少、速度快，能够达到每秒至少 27 帧的实时渲染要求。但是对于现实场景中复杂光照效果的模拟，特别是对“全局光照”，比如反射和折射的精确表现，光栅化方法难于实现。

光线追踪算法模拟几何光学，通过追踪与场景中的物体表面发生交互作用的光线，得到光线经过路径的模型，从而计算视觉效果的一种方

收稿日期：2015-05-05　　修回日期：2015-06-28；
基金项目：国家自然科学基金(61136002)，陕西省工业公关项目(2014k06-36)，西安市科技计划项目(CX12188(7))，陕西省教育厅科技计划项目(2013JK1128)；
作者简介：曹小鹏(1976-)，男，陕西，硕士，副教授，研究方向为计算机图形学。

法。它能够得到光栅化方法很难获得的光学特效，但是光线追踪算法运算量大，实时渲染难于实现。用光线追踪算法实时渲染游戏画面，在 1 024×768 的分辨率下，共有 786 432 个像素，乘以每像素 30 束光线(分别用来计算反射，折射光线、阴影光线、环境光照跟其他各种特效)及 60 帧/秒，需要每秒运算 141.5 亿束光线的计算能力。因此，这种技术多年来一直只在离线渲染领域被广泛使用，如电影特效的处理中。对于 3D 游戏等需要实时渲染场合，使用现有的硬件体系结构和算法，绘制一帧简单的图像也需要耗费较长的渲染时间，达不到实时渲染的标准。现在很多学者研究基于 GPU 的通用计算模式来进行光线追踪计算，对于相互邻近的一次光线有相同的原点和基本一致的方向，会以相似的路径穿过场景，集中碰撞到相同的物体，避免了邻近线程执行不同的指令。然而，二级光线在原点位置和光线方向上会有很大的不同，相邻线程因为分支语句以及随机访问内存而破坏并行性能。而且，不是每一条光线都会产生二级光线，大多数光线可能不与任何物体相交或者只产生一条二级光线，这种情况下就会有大量的线程处于空闲状态，GPU 线程块中只要有一个线程没有执行完毕，其它线程也不会提前终止，不会释放资源，造成了运算资源的浪费。因此，研究新的，适合于光线追踪的硬件体系结构与算法，能够从根本上推动 GPU 的发展，具有重要科学与工程意义。

本文研究光线追踪算法的特点，定量分析算法中每个部分的计算量，根据运算的负载均衡，提出光线追踪的处理流水线。同时提出一种适合于光线追踪，可重构的硬件体系结构模型(RTGPU)，它由多个计算核心(RTCore)组成，每个 RTCore 是由多个处理单元(PE)通过短线互联所构成的阵列处理器结构，PE 之间还可以使用片上路由器实现远程递归调用。设计了流水线映射到硬件体系结构的算法，PE 可以根据算法中各部分运算量进行动态重构。RTGPU 能够实现 2 048 条流水线的同时并行，提高了运算效率。同时建立了基于精确时钟的仿真平台，经过验证，RTGPU 能够较好实现光线追踪算法，与 CPU 和 GPU 对比，效率较高，效果较好。

1 相关研究

1980 年 Whitted[1]提出光线追踪算法。算法建立了第一个全局光照模型，实现了反射光线，折射光线，多重阴影等效果。如果完全模拟现实光源发出光线进行追踪，计算量极为庞大，计算机难于实现，同时很多光线没有进入人眼，对于人的视觉没有任何影响，浪费了宝贵的运算资源。因此，James Arvo 等[2]人提出了逆向光线追踪算法，算法跟踪从人眼发出的光线，穿过计算机屏幕上的像素点，碰撞到场景中的物体上，计算颜色，再多次跟踪计算反射，折射等光线分量，通过颜色混合，获得像素点颜色。算法的计算量大大降低，使得计算机实现成为可能。

场景中的不规则物体是以三角形图元形式给出，图元数量众多，光线与图元进行碰撞检测，如果不采用加速结构，对于具有 N 个图元的场景，绘制一幅具有 M 个像素的图像时，光线追踪算法的复杂度为 O(MN)，超过 95%的时间用于光线与场景内物体的碰撞检测，是计算最为耗时部分。很多学者研究场景内的物体的组织即：加速结构，提高计算效率。主要的加速结构有 KDTree，BVH，均匀网格等。Kang Y 等[3]提出了基于群的并行优化 KDTree 算法：gkDtree，试验证明算法建立树的速度比 KDTree 快 166 倍，但是查找树的速度没有显著提高。Novák J 等[4]提出了基于光栅的层次包围盒算法：RBVH，该算法将与光线碰撞的三角形用光线与面的交点所取代，能够获得更高的性能，但是缺少并行化的处理。Kalojanov J 等[5]在 GPU 上实现了两层的网格结构，能够实现了动态场景的光线追踪过程，但是效率还是较低。

德国 Saarland 大学的计算机图形小组开发了

https://dc-china-simulation.researchcommons.org/journal/vol29/iss2/6
DOI: 10.16182/j.issn1004731x.joss.201702006

OpenRT 库，将光线追踪技术应用于游戏 Quake3，使整个游戏的光影效果焕然一新。他们还设计开发了实时光线追踪的硬件架构——SaarCOR[6]，SaarCOR 可以实时生成 3D 场景中高仿真度的画面，产生实时特效，不过 SaarCOR 不能单独工作，需要一定的硬件资源(主要是处理器)和部分内存带宽。Nery A S 等[7]提出了一种能够处理光线追踪的硬件并行体系结构：GridRT，采用了均匀网格的加速结构，8 个处理器单元，渲染时间减少了 80%。

2008 年 NVIDIA 公司公布了光线追踪程序，它通过 4 颗 GPU 芯片(G92)组成的显卡阵列，实现了一个拥有两百万多边形的复杂度场景的实时渲染，2009 年发布了一个光线追踪引擎 Optix，它提供了光线追踪 API，编程人员只需简单调用相应的函数，隐藏了光线追踪算法的技术细节，但是其效率仍然较低[8]。CUDA 出现后，很多学者都在研究基于 GPU 的光线追踪算法[9]，随着云计算模式的出现，Chochlik M 提出了基于 GPU 云的光线追踪模型[10]。

我国在光线追踪研究方面，浙江大学的周昆等在 GPU 上实现了 KDTree，并且已经申请了美国专利[11-12]。邹华等在 GPU 上采用 CG 语言实现了 BVH 层次包围盒，并实现了光线投射算法[13]。卢贺齐等采用 OpenCL 在 GPU 上实现了 KDTree，并且实现了光线追踪算法[14]。

2 流水线设计

本文研究光线追踪算法原理与特点，采用可重构的并行架构对光线追踪算法进行加速，为了做到每个计算核心的高效运行，对光线追踪中每个部分的计算量进行定量分析，在各个部分计算量负载基本均衡的前提下，得到光线追踪流水线。

逆向光线追踪算法如下：

Step 1：人眼作为原点向计算机屏幕上的每一个像素投射光线(光线要进行起始点坐标系变换，光线方向正规化等)，由于在屏幕上是以像素点为显示单位，将连续信号变为离散信号，不可避免的会出现走样现象，表现为锯齿状，需要采用反走样算法处理，常用的反走样算法有：超采样反锯齿，抖动反锯齿等，基本原理是将像素进行进一步的细分，产生至少 4 倍的光线，再进行混合，从而获得更加精细的效果，但是大大增加了计算量；

Step 2：光线与场景内物体进行碰撞检测；

Step 3：判断是否达到了光线追踪的终止条件，通常光线追踪终止条件有：1) 光线与场景中的景物没有交点。2) 追踪光线对像素点颜色的贡献小于某一设定值。3) 追踪深度超过设定的最大追踪深度，一般设定为 4。如果达到终止条件，则结束，否则继续。根据物体表面的材质情况，又投射出三种光线，即反射光线(reflection ray)、折射光线(refraction ray)和阴影光线(shadow ray)。反射光线渲染物体表面镜面反射作用，折射光线渲染物体表面透射作用，阴影光线则投向光源，通过是否被遮挡来判断物体交点是否应该渲染；

Step 4：计算光线与最近的物体交点，计算该点纹理，碰撞点反射，折射光颜色分量和阴影分量；

Step 5：进行颜色混合。

2.1 产生逆射光线

逆射光线是指从人眼发出穿过计算机屏幕上每个像素点的光线。为了反锯齿，本文采用多采样抖动反锯齿算法，每个像素点平均划分成四个部分，每个部分内采用随机方式，产生采样点。总共产生四条原始逆射光线，称为：一次光线。一次光线与物体碰撞后产生的反射光线，折射光线等，称为：二次光线。二次光线进入光线队列。为了提高计算效率，降低颜色混合时的存储消耗，首先判断二次光线队列是否为空，如不为空时，计算二次光线，如为空，才产生一次光线。计算一次光线时，首先要定义一个基于视点的新坐标轴，三个轴分别用向量 $\boldsymbol{u}$，$\boldsymbol{v}$ 和 $\boldsymbol{w}$ 表示，计算如公式(1)。

$$\boldsymbol{u}=\frac{\boldsymbol{up}\times\boldsymbol{w}}{\|\boldsymbol{up}\times\boldsymbol{w}\|},\quad \boldsymbol{v}=\frac{\boldsymbol{w}\times\boldsymbol{u}}{\|\boldsymbol{w}\times\boldsymbol{u}\|},\quad \boldsymbol{w}=\frac{e-l}{\|e-l\|} \tag{1}$$

式中：e 为观察者的世界坐标；l 为观察方向的中心线上某个点；$\boldsymbol{up}$ 为摄像机视点向上的向量。

用公式(2)来计算光线方向。

$$rD=X_w\boldsymbol{u}+Y_w\boldsymbol{v}-d\boldsymbol{w} \tag{2}$$

式中：rD 为光线方向；Xw，Yw 分别为视椎体截平面上 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 方向上的网格点位置；d 为眼睛到投影面的距离。

2.2 碰撞检测

光线与场景中的物体是否相交是光线追踪算法中最常用到的运算。它不仅在确定光线与物体的交点位置时使用，而且在计算阴影时，用来判断阴影光线是否与物体相交。本文的测试场景中主要包括两类碰撞检测：光线和球体，光线和三角形片元。

在计算光线和球体碰撞时，球体可以用 $\|P-c\|=r$ 来表示，P 表示球面上任一点，c 表示球心，r 表示球半径。光线可以表示为：$P(t)=p_0+tu$，p_0 表示光线的起始点，u 表示方向。联立两式，可得 $At^2+Bt+C=0$，其中：$A=u^2$，$B=2p_0u-2cu$，$C=(p_0-c)^2-r^2$。交点用公式(3)计算。

$$t_0=\frac{-B+\sqrt{B^2-4AC}}{2A},\quad t_1=\frac{-B-\sqrt{B^2-4AC}}{2A} \tag{3}$$

光线和三角形片元的求交计算，Möller 等提出了较好的算法[15]，基于该算法，使用了矩阵运算加速器，可以得到用矩阵形式表示的相交点的笛卡尔坐标 t 和在三角形中的重心坐标 u,v，如公式(4)所示：

$$\begin{bmatrix} t\\ u\\ v \end{bmatrix}=\frac{1}{(D\times E_2)\cdot E}\begin{bmatrix}(T\times E_1)\cdot E_1\\ (D\times E_2)\cdot T\\ (T\times E_1)\cdot D\end{bmatrix} \tag{4}$$

式中：$E_1=V_1-V_0, E_2=V_2-V_0, T=\mathrm{O}-V_0$，$V_0,V_1,V_2$ 分别是三角形片元的三个顶点；D 为光线方向；O 为光线起点。

首先需要判断光线与场景中所有物体是否相交，比较与视点最近的碰撞点，该点即为光线与物体的交点。光线与场景中的物体求交可分两种情况：(1) 无碰撞，以背景色代替。(2) 有碰撞，首先产生反射光线，折射光线，进行递归运算，然后计算像素点的反射光线分量，折射光线分量，阴影，纹理等。

2.3 产生二次光线

本文用 I_{tran} 表示折射光分量，I_{amd} 表示环境光分量，I_{tran} 和 I_{amd} 需要采用多次递归运算，追踪计算折射光线和环境光线。计算 I_{tran} 和 I_{amd}，主要确定光线的起点和方向，起点即为光线碰撞点，用公式(5)计算方向。

$$\theta_t=\arcsin\sqrt{(\frac{n_1}{n_2})^2(1-\cos^2\theta_e)} \tag{5}$$

式中：n_1，n_2 分别是入射与折射光线处的介质折射率；θ_{e} 为入射角；θ_{t} 为折射角。

2.4 染色计算

在光线追踪算法中，I_{dtff} 表示相交点的漫反射分量、I_{spec} 表示镜面反射分量。I_{dtff} 和 I_{spec} 是光源对交点的直接光照影响，直接计算即可得到。根据 Lambert 定律，可得：

$$I_{\mathrm{diff}}=K_d I_i\cos\theta \tag{6}$$

式中：K_d 为物体表面的漫反射率；I_i 是光源所发出的入射光亮度；θ 为入射光与表面法向量之间的夹角。根据 Phong 镜面反射光照模型，可得：

$$I_{\mathrm{spec}}=I_i K_s(\cos\alpha)^n \tag{7}$$

式中：K_s 称为镜面高光指数；α 为视线与镜面反射光线之间的夹角；n 是表面粗糙系数。

https://dc-china-simulation.researchcommons.org/journal/vol29/iss2/6
DOI: 10.16182/j.issn1004731x.joss.201702006

2.5 颜色混合计算

光线追踪中每个采样点的颜色，可以通过公式(8)计算。

$$I_{\text{total}} = I_{\text{diff}} + I_{\text{spec}} + I_{\text{tran}} + I_{\text{amd}} \tag{8}$$

因为有一次光线和该光线所产生的二次光线匹配问题，本文采用对每条光线进行编号的方法，前 15 位表示光线编号，第 16 位表示几次光线。把所有光线合成结果存入到一个光线信息表中，进行匹配查询。

2.6 定量分析

假定一条光线与场景中物体相交，产生反射、折射和阴影三条二次光线，共迭代四次，最后进行颜色混合，统计光线追踪中各个部分的计算量，结果如表 1 所示。

表 1 光线追踪中各个部分的计算量

Tab.1 The computation of each part in ray tracing

名称	产生逆射光线	碰撞检测	计算纹理	计算阴影	计算漫反射	计算镜面反射	产生二次光线	颜色混合
计算量所占百分比	2	26	10	15	16	17	12	2

从表 1 中可知，光线追踪中各个部分的运算量差别较大。产生逆射光线和颜色混合对于一条光线只进行一次，所以计算量较少。碰撞检测与场景中物体的个数有关，当物体个数较多时，大量计算都消耗到碰撞检测中。计算漫反射分量和镜面反射分量需要的计算量也较大，但是实际场景复杂，不是每条光线都会产生二次光线，二次光线也不是都迭代四次，所以计算量变化较大。因为场景不同，场景中物体个数也不同，碰撞检测和像素点颜色计算部分运算量变化较大，如果对这些部分采用动态可重构方式，能够提高计算效率。根据对各部分运算量分析，得到的逆向光线追踪的流水线如图 1 所示。

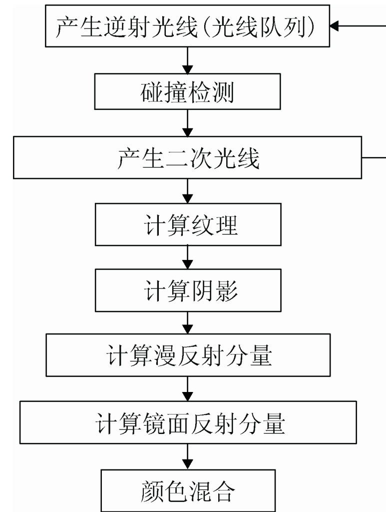


图 1 光线追踪流水线

Fig.1 The pipeline of ray tracing

3 硬件结构模型

3.1 RTGPU 结构

针对光线追踪算法，本文设计了可重构的处理器阵列结构模型—RTGPU，结构如图 2 所示。它由 256 个浮点运算簇即：RTCore 构成，所有 RTCore 通过总线连接，每个 RTCore 都包含矩阵运算加速结构。存储采用三级模式：系统存储，簇存储，Cache。RTGPU 作为协处理器，通过前端处理器(FEP)将场景结构信息存放于系统存储器中，然后发布到各个 RTCore 中。

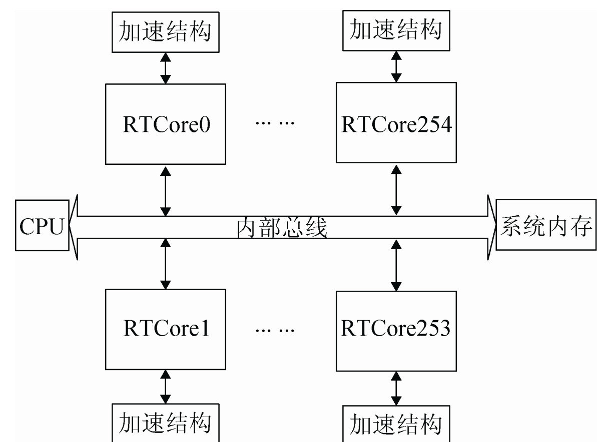


图 2 RTGPU 体系架构

Fig.2 The architecture of RTGPU

Published by Journal of System Simulation, 2017

3.2 RTCore 结构

光线追踪的运算主要在 RTCore 中进行。RTCore 结构如图 3 所示。每个 RTCore 由 16×16 个处理单元(PE)构成，PE 之间采用短线互联的结构，降低访问延时，提高效率，还可以通过路由器(RU)进行远程 PE 的访问。簇控制器(CLC)是整个 RTCore 的核心处理器，主要下发数据和指令、缓存原始数据、回收结果数据、动态重构 PE、控制 RTCore 运行。行控制器(RC)主要控制初始化指令的下发、控制指令下发、存储指令索引、指挥一行工作。列控制器(CC)主要完成存储共享数据、收集 PE 的结果数据和接收路由数据传输请求。近邻通信线程管理器(TM)主要存储 16 个线程的实时状态，管理 16 个线程的超时、近邻、路由、Halt 切换。路由器(RU)主要负责远程数据传输、远程函数调用。

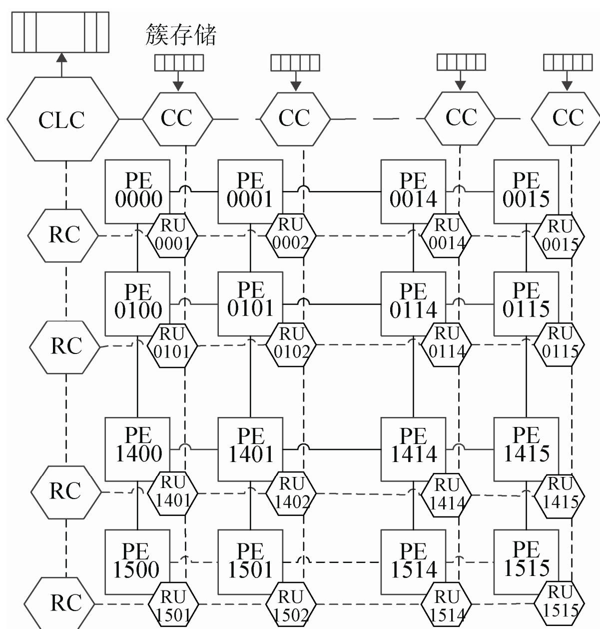


图 3　RTCore 结构

Fig.3　The architecture of RTCore

3.2 PE 结构

每个 PE 由一个 ALU、一个控制器、一个路由器(RU)、4 个邻接共享存储(RF)、数据存储(D-mem)和指令存储(I-mem)组成，结构如图 4 所示。PE 的特点是没有寄存器文件(Register File)，ALU 直接从存储器中读取指令和数据。ALU 平均每个时钟周期可以发出两条以内的指令，执行来自于外部(行、列、簇)控制器的指令序列，数据来自于本地存储或者邻接存储，指令通过控制器(ICTL)执行。路由器 RU 负责将数据传送到远程 PE。相邻的两个 PE 之间通过存储器直接相连，即采用短线互联方式，邻接共享存储分为四个部分：Me(东)、Mw(西)、Ms(南)和 Mn(北)，每部分用于一个方向的通信。这四个部分在逻辑上都是处理单元数据存储的一部分，采用直接寻址方式，大部分指令都可以使用邻接共享存储。这部分存储也可以被相邻处理单元存取。共享存储器的存取有两种模式。

①阻塞模式：每个共享存储地址都有一位数据有效位。当读取数据时，如果数据无效，则当前线程进入等待状态；如果数据有效，则读取数据，并将其置于无效。当写入数据时，若数据无效则直接写入，数据有效则等待。

②非阻塞模式：不管数据是否有效，直接读取数据。写入数据时也不管目标地址数据是否有效。

阻塞模式效率低，但是可靠，非阻塞模式效率高，但可靠性差。

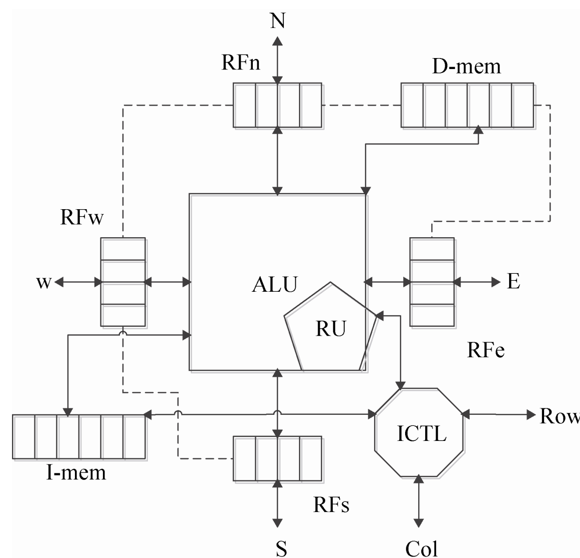


图 4　PE 结构

Fig.4　The architecture of PE

https://dc-china-simulation.researchcommons.org/journal/vol29/iss2/6
DOI: 10.16182/j.issn1004731x.joss.201702006

4　映射算法

本文将光线追踪流水线映射到RTGPU中，首先将计算机屏幕划分为 256 块，每一块对应一个RTCore，在 RTCore 中，一条流水线映射到一列PE 上，流水线中的每一部分映射到一行或几行PE。根据每一部分运算量的大小，动态配置PE的行数。每个 RTCore 同时有 16 条流水线并行运行。RTGPU 同时有256×16=2 048 个流水线在进行线程并行和数据并行。

算法过程：

Step 1：指令序列下载。流水线中不同处理指令序列，通过 FEP 和内部总线从 CPU 下载到每个RTCore 中。先由簇控制器根据场景复杂情况，判断每一部分的计算量，动态分配 PE 行数，再将不同的指令序列通过行控制器分别下载到每个 PE 的I_mem 中。

Step 2：场景数据下载。场景中物体数据同样通过 FEP 和总线，从 CPU 下载到簇控制器，再由列控制器分发到各个 PE 的 D_mem 中，

Step 3：启动所有 PE。采用流水线方式，上下行之间 PE 之间采用非阻塞方式，每一行 PE 执行相同的指令序列。

第 1 部分：产生逆射光线，对应 1 行 PE。首先检查二级光线 FIFO 中是否有尚未处理的光线，如有，处理二级光线。如无，根据抖动反锯齿算法，随机产生一条一级光线，对光线进行编号，并计算起点和方向。然后，将光线信息传送到下一部分。

第 2 部分：碰撞检测，对应 3~10 行 PE。光线和场景内的物体进行碰撞检测，本文没有采用加速结构，所以需要进行场景中逐个物体的检测。该部分的 PE 又分为发送 PE(SPE)、计算 PE(CPE)和回收 PE(RPE)。

SPE：每一列对应一个 SPE。首先接受光线信息，然后通过 RU 判断邻近 CPE 状态，如果不忙，则将光线信息发送到该 CPE，并启动 CPE，如忙，则轮寻下个 CPE。

CPE：每一列对应多个 CPE，可以是本列CPE，也可以是邻近列的 CPE。CPE 主要进行光线和物体相交计算，计算完毕后，将碰撞点信息通过 RU 发送到 RPE。

RPE：每一列对应一个 RPE。主要判断该光线与物体是否相交，如相交还要判断该条光线与视点的最近碰撞点，最近碰撞点即为交点。将该点信息发送到下一部分。如果无碰撞，则直接通过 RU 将该像素点背景颜色传送到颜色合成部分。

第 3 部分：产生二次光线，对应 1 行 PE。判断是否已经达到了光线追踪终止条件，如达到，停止产生二级光线，并产生最终光线信息，过 RU 传送到颜色混合部分。如没有，计算二级光线的方向和起始点，并对光线进行标号。将二级光线信息通过RU 传送到第一部分的二级光线 FIFO 中。

第 4 部分：颜色计算，对应 2-6 行 PE。分别计算纹理，阴影，漫反射分量，镜面反射分量，可根据计算量情况动态调整计算对应的 PE 行数。计算结束后，将结果传入颜色混合部分。

第 5 部分：颜色混合，对应 2 行 PE。

第 1 行 PE 主要完成采样点颜色混合。获得一个采样点颜色信息后，判断是否为最终光线，如是，则在光线信息表中，查找一次光线和与之对应的所有二次光线的颜色信息，进行颜色混合，混合完成后，清除该点所有一次、二次光线信息。如不是最终光线，则将该点颜色信息存入光线信息表中。

第 2 行 PE 主要完成像素点颜色混合。由于采用多采样抖动反锯齿算法，根据光线编号，将四个采样点信息合成为一个像素点信息。

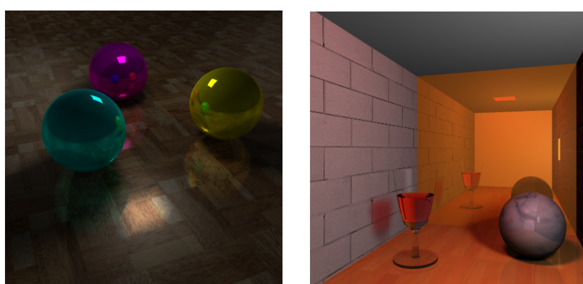
Step 4：回收。通过列控制器，簇控制器，总线和 FEP，将每个像素点颜色信息传送到 CPU，进行显示。

5　仿真

论文设计并开发了基于 C++的软件功能性仿

真平台，采用了基于精确时钟控制方式[16]，每个RTCore 的主频设置为 1GHz。光线追踪程序采用 OpenCL 编写，OpenCL 编写的并行程序，能够覆盖了多种处理器芯片，可以在 CPU、GPU 或其他并行结构上运行，具有良好的通用性和跨平台性[17]。针对 RTGPU，采用 OpenCL_E(扩展的 OpneCL)编写，由于 PE 之间存在邻接和远程通信，不需等待依赖任务完成之后再去执行，新添加两个关键字__in 和__out，用来表明任务之间的通信变量。同时对 OpenCL 编译器进行了扩充，设计开发了 RTGPU 的编译器，对程序进行编译。

仿真使用软件环境为：Windows 7，Visual Studio2008，OpenCL Drivers。硬件环境为：GPU：Nvidia Geforce GTX750，2G 显存，512 个 SP，CPU：Intel i5-4460，主频：3.2 GHz(4 核)，内存：4 GB。将用 OpenCL 写的光线追踪程序，分别在 CPU，GPU 和 RTGPU 三种环境下进行编译，并分别在三种平台上运行。RTGPU 的仿真结果如图 5 所示。



(a) 球间映射 (b) 球与酒杯

图 5 仿真结果
Fig.5 The results of simulation

6 结论

当前，基于 GPU 的并行运算成为研究热点，很多不同应用领域的学者采用 GPU 对算法进行加速。在光线追踪算法的加速上，很多国内外很多学者利用 GPU 的并行性，建立加速结构，进行加速处理。研究主要集中在基于 NVIDIA 公司推出的统一计算设备架构(CUDA)的加速处理。CUDA 是一个比较封闭的架构，主要针对 NVIDIA 的 GPU 产品，对于编译器等的扩充较为困难。本文主要研究光线追踪算法的原理与特点，研究适合于该算法的硬件体系结构，建立算法流水线，研究将流水线映射到硬件体系结构的算法，针对本文所提出的硬件体系结构(RTGPU)的特殊点，如通过路由器，进行远程数据的获取等，需要增加特殊的指令，CUDA 不适合，所以本文选择了开放式的并行计算架构——OpenCL。基于 OpenCL 开发的程序，能够比较容易的映射到不同的硬件体系结构上，如：CPU，GPU 和本文所提出的 RTGPU。同时，基于相同计算架构设计的程序，分别运行到不同的硬件平台上，进行效率分析，具有可比性。

本文设计了基于 OpenCL 的光线追踪算法，采用单线程方式运行在 CPU 上，获得单处理器串行的运行效率。将算法并行化，并映射到 GPU 上，获得主流 GPU 的运行效率。因为 GPU 在内部编译，并运行核程序，对于开发者来说是一个黑盒子，不清楚各个计算部分是如何分配硬件资源，不能做到精确的分析与规划，所以效率较低。对于 RTGPU，本文根据光线追踪算法中各个部分的运算量，动态分配处理单元的个数，做到动态的负载均衡，同时各个流水线独立并行运行，能够获得更好的效率。

实验结果如表 2 所示。通过比较可以看出，多处理器流水线并行的 RTGPU 和单处理器串行的 CPU 相比，效率较高。图 5(a)中，模型个数和片元个数都较少，RTGPU 在碰撞检测部分所分配的 PE 个数较少，在计算颜色分量部分分配 PE 个数较多，计算速度较快。图 5(b)中因为增加了场景的复杂程度，特别是玻璃酒杯，片元个数明显增加，在碰撞检测部分使用了大量的并行 PE，整个渲染速度大幅提升，加速比达到了 11，加速效果明显。RTGPU 与当前主流的GPU 相比，在光线追踪方面，效果较好，能够实现实时光线追踪。

https://dc-china-simulation.researchcommons.org/journal/vol29/iss2/6
DOI: 10.16182/j.issn1004731x.joss.201702006 8

表 2　实验结果
Tab.2　The results of experiment

场景名称	片元个数	GPU		RTGPU		CPU		加速
		时间/ms	帧/秒	时间/ms	帧/秒	时间/ms	帧/秒	比
球间映射	13	15.3	65	14.5	69	116.8	9	8
球与酒杯	45 000	37.3	27	34.5	29	397.3	3	11

RTGPU 是一个可扩充的结构，通过增加 RTCore 的个数，效率会获得进一步的提高，但是随着 PE 个数的增加，采用集成电路实现时的功耗，红砖墙等问题，尤为突出，需要研究。同时在试验中未采用加速结构，所以效率仍然较低，下一步增加场景管理，添加加速结构，减少碰撞检测所需时间及所要使用的 PE 个数，增加颜色计算部分的运算能力，效率会得到进一步提高。

参考文献:

[1] Whitted T. An improved illumination model for shaded display [J]. Communications of the ACM (S0001-0782) 1980, 23(6): 343-349.

[2] Arvo J, Kirk D. Fast ray tracing by ray classification [J]. ACM Siggraph Computer Graphics (S0097-8930), 1987, 21(4): 55-64.

[3] Kang Y, Nah J, Park W, et al. gkDtree: A group-based parallel update kd-tree for interactive ray tracing [J]. Journal of Systems Architecture (S1383-7621), 2013, 59(3): 166-175.

[4] Novák J, Dachsbacher C. Rasterized Bounding Volume Hierarchies [J]. Computer Graphics Forum (S1467-8659), 2012, 31(2): 403-412.

[5] Kalojanov J, Billeter M, Slusallek P. Two-Level Grids for Ray Tracing on GPUs [J]. Computer Graphics Forum (S1467-8659), 2011, 30(2): 307-314.

[6] Schmittler J, Wald I, Slusallek P. SaarCOR: a hardware architecture for ray tracing [C]// Proceedings of Eurographics Workshop on Graphics Hardware. Switzerland: Eurographics Association Aire-la-Ville, 2002: 27-36.

[7] Nery A S, Nedjah N, França F M G. An efficient parallel architecture for ray-tracing [J]. Analog Integrated Circuits & Signal Processing (S0925-1030), 2012, 70(2): 189-202.

[8] Parker S G, Bigler J, Dietrich A, et al. OptiX: A General Purpose Ray Tracing Engine [J]. ACM Transactions on Graphics (S0730-0301), 2010, 29(4): 157-166.

[9] Hu W, Huang Y, Zhang F, et al. Ray tracing via GPU rasterization [J]. Visual Computer (S0178-2789), 2014, 30(6/8): 697-706.

[10] Chochlík M. Scalable multi-GPU cloud raytracing with OpenGL [C]// International Conference on Digital Technologies. USA: IEEE, 2014: 87-95.

[11] Zhou K, Hou Q, Wang R, et al. Real-time kd-tree construction on graphics hardware [J]. ACM Transactions on Graphics (S0730-0301), 2013, 27(5): 126-138.

[12] Zhou K, K Z, Zhou K, et al. Ray tracing on graphics hardware using kd-trees [P]. USA: US20100079451 A1, 2008.

[13] 邹华, 高新波, 吕新荣. 层次包围盒与 GPU 实现相结合的光线投射算法 [J]. 计算机辅助设计与图形学学报(S1003-9775), 2009, 21(2): 172-178. (Zou Hua, Gao Xin-bo, Lü Xin-rong. A Ray Casting Algorithm Based on Hierarchical Bounding Volumes and GPU [J]. Journal of Computer-Aided Design and Computer Graphics (S1003-9775), 2009, 21(2): 172-178.)

[14] 卢贺齐，鲍鹏，冯结青. 基于 OpenCL 的实时 KD—Tree 与动态场景光线跟踪 [J]. 计算机辅助设计与图形学学报 (S1003-9775), 2013, 25(7): 963-973. (Lu He-qi, Bao Peng, Feng Jieqing. OpenCL Based Real-Time KD-Tree and Ray tracing for Dynamic Scene [J]. Journal of Computer-Aided Design and Computer Graphics (S1003-9775), 2013, 25(7): 963-973.)

[15] Möller T, Trumbore B. Fast, minimum storage ray triangle intersection [J]. Journal of Graphics Tools (S2165-347X), 2005, 2(1): 21-28.

[16] Cao X, Huang H，Li T. A cycle accurate simulation platform for GPU performance study [J]. International Journal of Advancements in Computing Technology (S2005-8039)，2012, 4(20): 684-691.

[17] 龚兴全，李康，孔凡敏. 基于 OpenCL 的图形处理器 FDTD 算法仿真研究 [J]. 系统仿真学报 (S1004-731X), 2014, 26(8): 1639-1643. (Gong Xingquan, Li Kang, Kong Fanmin. FDTD Simulation Using Graphic Processing Units Based on OpenCL [J]. Journal of System Simulation (S1004-731X), 2014, 26(8): 1639-1643.)