光子映射在 CUDA 中的研究与实现①

林其选　王毅刚 (杭州电子科技大学 图形图像研究所 浙江 杭州 310018)

摘　要： 通过修改光子映射算法的实现过程，使得该算法能够通过 CUDA 完全运行在最新的 GPU 上，从而能够充分利用 GPU 强大的并行计算能力，加速光子映射的实现。光子映射在 CUDA 中的实现主要通过两个方面来完成：构建光子图和估计辐射能。同时为了提高对光子图中的光子信息的查找速度，采用了 kd-tree 结构来存储光子信息，使得可以通过 KNN（K-Nearest Neighbor）快速搜索光子图。在所测试环境中，渲染速度是 CPU 中的近 10 倍。

关键词： 光子映射；GPU；光子图；kd-tree；KNN

Research and Implementation of Photon Mapping in CUDA

LIN Qi-Xuan, WANG Yi-Gang

(Hangzhou Dianzi University, Hangzhou 310018, China)

Abstract: This paper makes photon mapping algorithm capable of entirely running on latest GPUs by modifying its process in CUDA and harnesses the massive parallel computing power of GPU to accelerate the implementation of photon mapping. The implementation of photon mapping in CUDA includes the construction of the photon maps and estimation of the radiance. To accelerate the search of photons in photon map, this paper uses kd-tree to store photons and search them by KNN. The rendering speed is nearly 10 times by CPU under the test environment.

Keywords: photon mapping; GPU; photon map; kd-tree; KNN

光子映射算法(Photon Mapping)是由 Henrik Wann Jensen 提出并发展起来的一种全局光照算法[1]，现已广泛地应用于全局光照渲染，是当前模拟全局光照最快的算法之一。

光子映射算法分两步实现：第一步发射和跟踪光子，构建光子图；第二步利用光子图估计辐射能，渲染场景。具体地讲，首先光子从光源出发，根据蒙特卡洛(Monte Carlo)随机方向向场景发射，并跟踪这些光子在场景中的运动情况，然后将与表面碰撞的光子信息记录在光子图中。然后，在渲染过程中，通过 KNN (K-Nearest Neighbor)搜索光子图，获取表面点附近光子信息，估计该点的光照辐射能量，渲染场景。

1　前言

全局光照(Global Illumination)渲染是一种物理模拟过程，它包括直接光照(direct lighting)和间接光照(indirect lighting)。在计算机图形学方面，全局光照算法有着悠久的历史，早期大多数的模拟方法都是利用辐射度算法和光线跟踪算法，或者两者的结合实现的，近期出现了光子映射算法，该算法是目前应用最广的算法之一[2]。

光子映射技术与传统的光照渲染方法不同，它从光的物理属性出发，研究光子在场景中的运动，使得对于光照的模拟真正达到了从物理属性的角度来考虑，从而对全局光照的渲染达到了物理模拟的效果。并且该方法能够更有效地模拟一些特殊的光学现象，例如：焦散(Caustics)、辉映(Bleeding)、中间介质(Participating Media)等。

光子映射是一种功能强大的渲染算法，但由于其计算强度大，渲染速度慢，因而无法应用于交互渲染

① 基金项目:浙江省科技计划面上项目(2008C24014)
收稿时间:2009-09-10;收到修改稿时间:2009-10-24

领域。但随着图形硬件的发展，特别是可编程图形处理器(GPU)的快速发展，可以利用其强大的并行计算能力，使得光子映射可以在 GPU 上得到加速渲染，并达到了实时渲染的性能[3,4]。本文在 CUDA 中实现了能够完全运行在 GPU 上的光子映射算法。

2 相关工作

2003 年 Timothy J. Purcell 等在 GPU 上实现了光子映射算法[5]。它采用广度优先的方法在 GPU 上跟踪光子，并将光子直接存储在 GPU 上基于网格的结构中。但由于该方法使用了均匀网格而不是 kd-tree 来存储光子，这样大大降低了最近领域光子搜索的效率，使得光子映射的渲染速度不快。

2005 年 Szabolcs Czuczor 等实现了将光子存储在图形硬件的纹理结构中[5]，并采用纹理滤波的方式搜索最近领域的光照信息。该方法可以一步实现对所有顶点辐射能信息的获取，然后通过单步纹理查找获取某顶点辐射能的平均值。该方法对光子的查找速度相对普通的 kd-tree 结构有所提高，但它的存储结构和搜索过程比较复杂，不便于实现。

2008 年周昆等首次提出并实现了在 GPU 上实时构建 kd-tree 的算法[3]。该算法通过广度优先方式创建树节点，而不是传统的并行算法，并且充分利用图形硬件的优越并行计算能力，实现了 kd-tree 的实时构建，为在 GPU 上实现光子映射提供了良好的基础。

CUDA(Compute Unified Device Architecture)是 NVIDIA 推出的一个并行计算架构[6]。这个架构包含有一个 ISA(指令集架构)以及并行计算的硬件引擎。CUDA 包括了针对 GPU 的 C 语言编译器、纠错器/制模器、专用驱动和标准函数库等。

虽然以前论文曾在 GPU 中实现过光子映射，但大部分都是基于传统的 GPU，不是在类似于 CUDA 这样的软件平台中实现的，使得该算法在 GPU 中的实现有着一定的难度，并且这些论文对算法实现过程细节的说明也不够详细，不便于理解光子映射在 GPU 中实现的具体过程。本文通过对光子映射算法的研究，并对实现过程做了适当的修改，使得该算法可以通过 CUDA 软件平台，完全运行在 GPU 上。同时，本文利用已有的相关文章，对在 CUDA 中实现光子映射的细节进行了详细的说明，并对实现过程中出现的相关难点给出了一些解决方法。

3 将CPU程序移植到GPU上运行的难点

CUDA是目前较为方便和高效的GPU编程架构平台，该架构作为一个专用高性能 GPU 计算解决方案，能够解决以前无法解决的复杂问题。CUDA 的 SDK 中的编译器和开发平台支持 Windows、Linux 等操作系统，并且可以与 Visual Studio 集成在一起，使得 GPU 编程变得比较方便。但相对于 CPU 编程而言，CUDA 编程也存在着一些不便和难点。

由于目前 CUDA 只支持 C 语言，其他语言暂不支持，而 CPU 几乎可以支持所有的编程语言，可以比较容易地应用各种编程技术，特别是面向对象技术，在这一点上 CUDA 编程不如 CPU 编程方便和实用，对某些功能的实现也有一定的困难。

为了减少函数调用的开销，提高 GPU 程序运行效率，目前 CUDA 将普通函数全部采用内联(inline)方式编译，这样导致一个函数无法调用自身函数，即无法递归调用，所以在 CUDA 编程中需要将 CPU 中的递归函数改写为非递归形式，这给 CPU 下的程序移植到 GPU 中运行带来了一些麻烦。

动态空间分配是 CPU 编程中较为常见的技术，该技术可以有效地利用有限的存储空间，可以有效避免空间不足或空间浪费，而目前 CUDA 暂不支持这种技术，所以为了 CUDA 程序能够正常运行，要预先给线程分配固定大小的空间，具体分配的大小要看程序整体空间的需求和显卡存储器的大小而定，并且这种空间的分配方案会直接影响线程的运行效率，所以要多方面考虑情况，存在一定的难度。

每个 Block 中同时运行的线程数目确定问题：由于每个 Multiprocessor 中的线程控制器的个数(如 G80 中的数目为 768)是有限的，使得每个 Block 中不能同时运行超过该数目的线程。并且每个 Multiprocessor 中用于存储所有线程状态的寄存器个数 M(如 G80 中的数目为 8192)也是有限，而每个线程所需要存储的状态数 N 会因任务的不同而有所不同，故同时运行的线程数目最大不应超过 M/N(否则就需要把多余的数据储存到显卡内存中，这样就会降低执行的效率)，所以需要合理地确定每个 Block 中同时运行的线程数目，使得每个线程运行效率较好，且能够充分地利用 GPU 中宝贵的资源，从而使整个任务能够快

速完成。

4 光子映射在CUDA中的实现

通常情况下，光子映射算法主要通过以下两个步骤来实现：

① 发射和跟踪光子，构建光子图。

从光源向场景随机方向发射大量光子，使用类似路径追踪(Light Tracing)的方法跟踪穿过场景的每个光子，并在碰到非镜面时将它们保存起来，以建立光子图(photon map)。由于每次发射光子具有一定的独立性，所以可以在 CUDA 中并行地发射和跟踪光子，从而快速建立光子图(见 4.1)。

② 估计辐射能和渲染场景

使用统计技术从光子图中估计出场景中以表面点为中心的空间(如球体、椭球等)的入射光通量(flux)和反射辐射能，从而确定这一点的光亮度。

本文根据光子跟踪算法的两个基本步骤，修改算法的实现过程，充分利用 CUDA 中的并行处理，实现该算法。

4.1 发送和存储光子，构建光子图。

从光源向场景随机方向发射光子，并跟踪光子在场景中的运动。在 CUDA 中实现时，各个线程(Thread)独立地从光源向场景随机发射光子，并跟踪光子在场景中的运动情况，存储相应的光子信息到光子图中，其中信息包括光子的能量、位置、运动方向等。这样反复循环，直到所有线程发射的光子总数达到预设的光子数目。

由于一般情况下，需要发射的光子数目往往比线程数目大得多，所以每个线程需要发送多个光子，那么每个线程究竟需要发射多少个光子呢？一般有两种方法来决定每个线程所要发射的光子数：

第一种方法：平均分配发射光子数目。假设光子总数为 M，线程数目为 N，(其中 M>N，且 M 是 N 的倍数)，那么每个线程需要发射的光子数为 M/N。这种方法的优点是每个线程在发射光子时，无需关注其他线程发射光子的情况，只需要发射完自己所分配的光子数即可，这样各线程具有良好的独立并发性，有利于提高执行效率；而这种方法的缺点也是明显的：由于每个光子需要跟踪的路径情况、跳跃的次数都有所不同，所以跟踪每个光子所需的时间也不同，所以各线程完成发射和跟踪相同数量的光子所需要的时间也就有所不同，因而就会出现有些线程先完成所分配的任务，而出现空闲等待状态，直到其他所有线程都完成，这样就会出现某些线程资源浪费，从而延长了整个发射过程的时间，影响发射效率。

第二种方法：所有线程协同完成所有光子发射任务。设立一个表示所有线程共同已发射的光子数变量 Count，初始化为 0，当某一线程发射一光子时，将 Count 值增 1。直到 Count 值达到所要发射的光子总数，完成整个发射过程。这种方法的优点是：基本消除了第一种方法的缺点，基本上达到了同时结束发射工作，减少了线程间的空闲等待时间，提高了发射工作效率。而这种方法也有缺点：由于各线程间需要一个中间变量 Count 来进行通信，判断是否已经完成所要发射的光子总数，而决定是否需要继续发射光子，这样就减弱了线程间的独立性；并且由于各个线程是并发执行的，就会出现多个线程同时访问和修改 Count 变量，导致 Count 变量的不一致，所以对 Count 变量进行访问需要有原子操作支持(即对变量的读取－修改－写入三个操作作为一个不可分割的同一整体，一次同时完成操作)，而这种操作需要图形硬件的支持(在 NVIDIA 的图形芯片中，只有满足计算能力在 1.2 或以上的显卡，对 Shared Memory 才支持原子操作)，从而增加了对硬件的依赖性，不利于程序的执行。

纵观以上两种方法，第二种方法的效率要比第一种高些，所以在图形硬件支持的情况下，可以优先考虑第二种方法。本文由于图形硬件不支持原操作，所以采用了第一种方法。同样地，本文由于图形硬件原操作的原因，多个线程也无法同时将自己的光子存储到光子图中，所以需要将光子首先存放在各个线程内部的光子列表中，然后等待所有线程完成光子发射后，再将各个线程中的光子转存到光子图中。

最后需要说明的是：由于从光源发射光子的方向需要随机性，但目前 CUDA 不支持生成随机数操作，所以需要自己实现生成随机数。本文采用线性同余法 LCG(Linear Congruence Generator)生成伪随机数：即选取足够大的正整数 M 和任意自然数 N0，a，b，由递推公式：

$$N_{i+1}=(a*N_i+b)\%M \tag{1}$$

其中 i=0，1，…，M−1，生成伪随机数序列。

到目前为止，所有的光子信息都已存储到光子列表中，但是为了下一步能够利用 KNN(K-Nearest Neighbor)搜索[7,8]快速查找表面点最近领域的光子信息，需要重新组织这些光子，构建平衡 kd-tree。

构建平衡 kd-tree 的基本思想[9]：首先在整个光子列表中，将光子的位置坐标作为关键字，沿着某一坐标轴(一般情况下是沿着这些光子 AABB 包围盒最长的坐标轴)获取一个适当的分割点，将该光子列表分为左右两个子列表，并将该分割点信息(即光子信息)写入 kd-tree 节点中，作为整个 kd-tree 的根节点，然后这样重复操作左右子列表，并将左右子列表的根节点分别作为整个 kd-tree 根节点的左右子节点。其中“获取一个适当的分割点”中的适当标准大致有两个：第一个是直接将中间位置作为分割点，这样做最为简单方便，节省时间，但之后的 KNN 操作效率不高；第二个是根据 VVH(Voxel Volume Heuristic)[10]方法，获取一个适当的分割点，这样做的好处是有利于提高 KNN 的操作效率，但计算和比较各个关键位置的 VVH 需要花费大量的时间，使得构建 kd-tree 的效率不高。在 CUDA 中，可以采用广度优先法，各个线程并行处理各个节点的左右子树，从而加快平衡 kd-tree 的构建。

4.2 估计辐射能和渲染场景

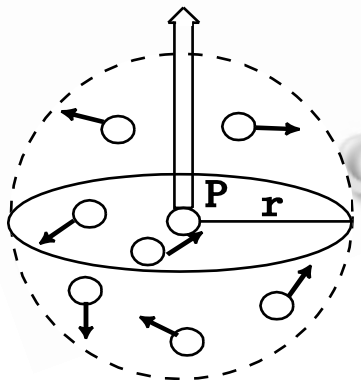


图 1 辐射能估计

利用第一步产生的光子图，根据光线跟踪从照相机位置出发，向场景发射光线，当该光线击中表面点 P 时，根据光子图中的信息，对 P 点进行 KNN 操作，查找以该点为中心的区域(如球体)的入射光通量(flux)等光照信息(见图 1)，然后估计 P 点的辐射能(radiance)信息[11]，最后将该辐射能信息添加到光线跟踪在 P 点的光照信息。

并不是所有在球 S(P,r)内的光子都计入 P 点的辐射能，只有入射方向向量 d 与 P 点法向量 N 的点积大于 0 的光子才有可能被考虑计入 P 点的辐射能，因为若 d·N <= 0，则表示该光子是朝着表面内部运动的，应该忽略该光子。如果在该球体内可以被考虑计入 P 点辐射能的光子数大于规定的最大光子数 max，那么应该减小该球体的半径 r，以减少所包含的光子数，提高估计辐射能的精确度。最后辐射能的计算公式为：

$$\text{Radiance} = \frac{\sum_{i=1}^{m} \text{power}_i * (d_i \bullet N) * \text{factor}}{\pi r^2} \qquad (2)$$

其中 m 为球体 S(P,r)内满足 d·N > 0 的光子数，poweri 为第 i 个光子的能量，factor 为缩放因子，其他的标号见上。

如果 m 值小于某个预设的阈值，那么应该忽略这些光子的辐射能，即该点的光子辐射能计为 0，因为这些稀少的光子会使图像在某些地方出现噪音，变得模糊不清。最后 P 点的光照强度为以上估计所得的辐射能 Radiance 与光线跟踪在该点计算所得的光照强度之和。

由于这样相加所得的结果可能会大于 1，所以最后还得将这些颜色值单位化到[0 1]。

5 实验结果与分析

5.1 实验环境

本文所作的测试是在一台普通的 PC 机上实现的，机器配置如下：

操作系统：Windows XP；

CPU：Intel Pentium 4 530；

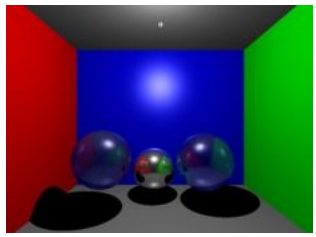
显卡：NVIDIA GeForce 8600 GT；

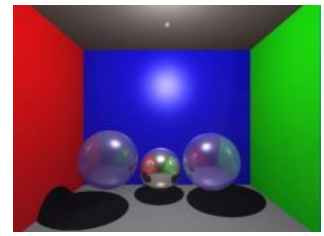
开发平台：MS Visual Studio 2008 + NVIDIA CUDA 2.2；

5.2 实验结果

图 2 为实验渲染的效果图：从左到右分别是无光子图、全局光子图和全局光子与焦散光子图。由于篇幅有限，只给出以下三张图片。

表 1 为光子映射在 CPU 和 GPU 中渲染的时间。


(a) 无光子图


(b) 全局光子图

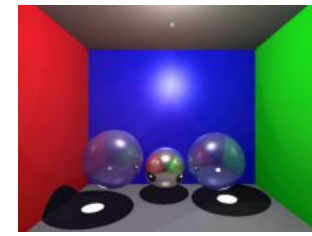

(c) 全局光子与焦散光子图

图 2　渲染效果图

表 1　渲染时间

焦散光子数目	0	20000		50000		100000	
全局光子数目	0	200	1000	200	1000	200	1000
CPU	1.03s	18.34s	39.53s	18.87s	40.11s	19.55s	41.29s
GPU	532ms	1.82s	3.75s	1.98s	3.93s	2.23s	4.16s

观察以上实验结果，可以得出以下结论：从无光子映射的图片和有光子映射的图片对比可以看出，光子映射使场景变得更加明亮，地面的阴影变淡，使场景有辉映效果，更加逼真。添加焦散光子后可以产生焦散效果，更符合实际情况。

在渲染速度方面，在我们的测试环境中，GPU 中渲染的速度是 CPU 中的近 10 倍，速度有明显地提高。但由于受实验硬件等条件的限制，目前还无法达到交互渲染的速度。

6　结论

本文通过适当地修改光子映射算法，使得该算法能够通过 CUDA 完全运行在 GPU 上，从而可以利用 GPU 强大的并行计算能力，加速光子映射实现过程，相对于运行在 CPU 上的速度有明显地提高。同时，本文利用已有的相关文章，对在 CUDA 中实现光子映射的细节进行了详细的说明，并对实现过程中出现的相关难点给出了一些解决方法。

由于本文中的实验程序只对简单模型和场景进行了考虑，而没有对有着复杂纹理材质的模型和无法一次性载入显存的大规模场景的考虑，所以该程序还无法用于渲染这些模型和场景，今后还应该在这些方面进行考虑，并加以完善。还有由于 CUDA 程序还不够完善和图形硬件的限制，目前还无法达到交互渲染的效果，速度方面还有待于进一步提高。

参考文献

1 Jensen HW. Global Illumination using Photon Maps. Rendering Techniques'96. Eds. X. Pueyo and P. Schröder. 1996.

2 陈皓.基于光子映射的虚拟现实真实感渲染算法研究[博士学位论文].合肥:合肥工业大学, 2008.

3 Zhou K, Hou QM, Wang R, Guo BN. Real-Time KD-Tree Construction on Graphics Hardware. SIGGRAPH Asia 2008.

4 李建明.基于 GPU 加速的实时虚拟鱼系统[博士学位论文].大连:大连理工大学, 2007.

5 Szabolcs Czuczor, László Szirmay-Kalos, László Szécsi and László Neumann. Photon Map Gathering on the GPU. EUROGRAPHICS 2005.

6 NVIDIA Inc. NVIDIA 计算统一设备架构编程指南. NVIDIA 官方网站. USA 2008.

7 Purcell TJ, Donner C, Cammarano M, Jensen HW, Hanrahan P. Photon Mapping on Programmable Graphics Hardware. Graphics Hardware (2003).

8 Popov S, Günther J, Seidel HP, Slusallek P. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. EUROGRAPHICS 2007.

9 Jensen HW. Realistic Image Synthesis Using Photon Mapping. ISBN: 1-56881-140-7. AK Peters, July 2001.

10 Fabianowski B, Dingliana J. Interactive Global Photon Mapping. Eurographics Symposium on Rendering 2009.

11 Yu TT, Lowther J, Shene CK. Photon Mapping Made Easy. Technical Symposium on Computer Science Education. USA 2005.