

基于 CUDA 的 Dividing Cubes 算法实现

张文波¹, 曹耀钦¹, 孙 伟², 李海龙¹

(1 第二炮兵工程大学, 陕西 西安 710025; 2 西北综合勘察设计研究院, 陕西 西安 710003)

摘 要: 针对传统 Dividing Cubes 算法中一次性将体元剖分为较小体元而带来大量计算的问题, 提出递进式的体元剖分方式, 以减少剖分过程中的计算量。同时, 将算法移植到图形处理器(GPU)上执行, 在统一计算设备架构(CUDA)下对其进行实现, 并分别对 kernel 函数、线程结构和存储空间进行了设计和分配。实验结果表明, 利用 GPU 的并行加速能力可以达到将近 10 倍的加速比, 有效地提升了等值面的绘制速度。

关键词: 等值面; 剖分立方体算法; 图形处理器; 统一计算设备架构

中图分类号: TP391

文献标识码: A

文章编号: 1000-7180(2013)12-0005-05

Implementation of Dividing Cubes Algorithm Based on CUDA

ZHANG Wen-bo¹, CAO Yao-qin¹, SUN Wei², LI Hai-long¹

(1 The Second Artillery Engineering University, Xi'an 710025, China;

2 Northwest Research Institute of Engineering Investigations and Design, Xi'an 710003, China)

Abstract: In traditional Dividing Cubes algorithm, voxels are divided into many small elements in one time which bring a large number of calculations. Therefore, a progressive subdivision is proposed to reduce calculations. In the meanwhile, the algorithm is transplanted into GPU, and implemented based on CUDA. The kernel, thread organization and memory allocation is also designed. Experimental results show that almost 10 times speedup is reached by using GPU's parallel acceleration capacity.

Key words: isosurface; dividing cubes algorithm; GPU; CUDA

1 引言

等值面在可视化领域中应用非常广泛, 如医学图像处理、几何曲面造型以及科学计算可视化等, 目前, 具有代表性的用于提取等值面的面绘制算法是移动立方体算法^[1] (Marching Cubes, MC 算法)、移动四面体算法^[2] (Marching Tetrahedra, MT 算法) 和剖分立方体算法^[3] (Dividing Cubes)。MT 算法和 Dividing Cubes 算法都是 MC 算法的改进算法, 其中 Dividing Cubes 算法非常适用于采样密集的体数据可视化。等值面提取与绘制算法最大的挑战来自于体数据规模与绘制效率的矛盾^[4]。在体数据实时可视化应用中, 用户希望在等值面阈值调整时能够得到实时的反馈结果, 对数据的实时处理效率提出

了非常高的要求。为此, 许多学者采取了多种方法来优化算法的执行, 如将数据组织成八叉树的结构^[5] 以及利用硬件加速^[6-7] 等。对于 Dividing Cubes 算法, 文献[8]中提出使用插值系数检索表来加速体元剖分过程中的三线性插值计算, 并设计了面点跟踪算法加快检索与等值面相交的小体元; 文献[9]中则利用 CT 数据的特点, 建立体元信息描述表和插值系数表加速体元剖分的计算。虽然这些改进策略都在一定程度上加快了算法的数据处理速度, 但没有充分利用当前图形硬件的高并行计算能力。

Dividing Cubes 算法中计算量最大的是对体元的剖分操作, 包括投影面积判断、小体元顶点值计算和小体元法向计算等。每个体元的剖分操作是独立的, 与其他体元没有依存关系, 非常适合于并行实

收稿日期: 2013-01-20; 修回日期: 2013-03-22

基金项目: 国家自然科学基金重点项目(61132008)

现.因此,将该算法移植到 GPU 中执行,利用统一计算设备架构(Compute Unified Device Architecture, CUDA)强大的并行计算能力加速算法的执行效率.

2 数据的组织及体元剖分方式的改进

三维数据按照行、列、层的规则网格化形式存储,可视化时是以体元为单位进行操作.如图 1 所示,以三维数据中相邻两层的八个角点组成的立方体为一个体元.

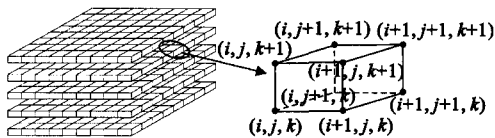


图 1 三维体数据的存储结构

传统的剖分立方体算法在对体元进行剖分时,将体元一次性剖分为在屏幕上投影小于像素的子体元,需要计算相当多子体元的角点值和法线方向,因而存在大量的插值运算.为此,本文提出递进式的体元剖分方式,如果一次剖分后的子体元无等值面通过,则对其不予处理,如一次剖分后的子体元有等值面通过,且在屏幕上的投影依然大于象素时,则对其按照图 2 所示方式进行二次剖分.体元分别在 x 、 y 、 z 方向上作二等分后,体元的每个面都被分为“田”字形的四部分.经过剖分后,一个体元被均分为八个子体元.选取其中一个子体元,再次对其进行检测.

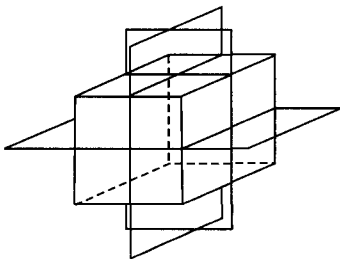


图 2 体元剖分图示

3 基于 CUDA 的 Dividing Cubes 算法设计

在 CUDA 架构中,CPU 和 GPU 之间是协同工作的,因此,算法的执行分为两个部分:CPU 端和 GPU 端^[10].CPU 端也称为主机端(host),执行主机代码,负责处理逻辑性强的事务和串行计算,GPU 端也称为设备端(device),执行设备代码如(kernel 函数),负责并行处理任务.CPU 端和 GPU 端的功能分别如图 3 和图 4 所示.

根据图 3 和图 4 对 CPU 端和 GPU 端功能的描述,Dividing Cubes 算法的执行可以分为三个部

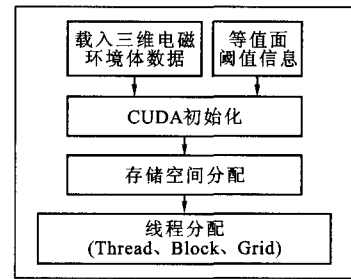


图 3 CPU 端功能

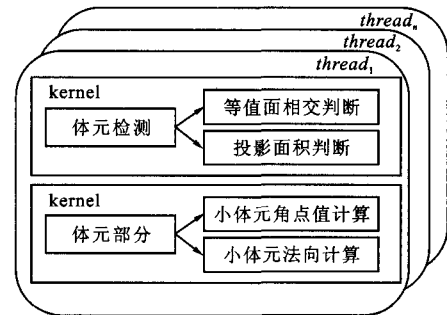


图 4 GPU 端功能

分:kernel 函数设计、存储空间分配和线程结构设计.

3.1 kernel 函数设计

算法中共设计了两个 kernel 函数,其中 Kernel_detectingVoxel 实现体元检测,Kernel_dividingVoxel 实现体元剖分.

三维体数据中有大量的体元是与等值面无交点的,即存在大量的空体元.核函数 Kernel_detectingVoxel 的目的就是对体数据中的体元进行检测,以确定体元是否与等值面相交以及是否需要进一步剖分. Kernel_detectingVoxel 将按照逐行、逐列的顺序对体元进行扫描、检测.首先判断体元的顶点值与给定的等值面阈值的大小关系,如果第一个顶点值大于等值面阈值,则只要再检测到任一顶点值小于或等于等值面阈值时就可以断定此体元与等值面相交,反之亦然.然后以体元投影到屏幕上的面积大小确定是否需要进一步剖分.如果需要剖分,就将此体元 8 个顶点的索引信息(包括坐标值和函数值)按照规定好的顺序写入到 $N \times 8$ 的 CUDA 数组中,这里的 N 表示需要剖分的体元个数.该 CUDA 数组的数据类型为 float4,其每一维的分量分别对应于顶点的 x 、 y 、 z 坐标和函数值.当遍历完最后个体元后,再将此 CUDA 数组绑定到一个二维纹理 tex_dividingCubes 上,由核函数 Kernel_dividingVoxel 对其中的每个体元进行剖分.

核函数 Kernel_dividingVoxel 遍历体元索引纹理 tex_dividingCubes 上对应的各个体元,然后按照

图2的剖分方式逐渐细分,直到满足最终的投影要求为止,并将剖分后的小体元输送至帧缓冲区等待输出绘制。

3.2 线程结构设计

三维体数据以规则网格化存储,设数据的行、列、层数量分别为 N_{row} 、 N_{col} 、 N_{slice} ,以每一行体元的计算归为一个线程块 block 执行,每个线程处理其中的一个体元。由于在算法中设计了两个 kernel 函数,每个 kernel 函数都将创建一个线程网格,因此需要对每个 kernel 函数中的线程结构分别进行设计。

3.2.1 Kernel_detectingVoxel 中的线程结构

CUDA 架构中以 warp 为线程的调度单元。在 NVIDIA GT240 的 GPU 中,每个线程块 block 最多可以分配 512 个线程 thread,每个多核流处理器 (Streaming Multiprocessor, SM) 可以分配 8 个 block,但 SM 中的线程总数不能超过 1024。当一个 block 分配到一个 SM 时,该 block 就被划分为多个 warp,每个 warp 包含 32 个线程,因此,每个 block 中的线程数量应当是 32 的倍数时才能最大化地发挥 CUDA 的并行能力。

block 中的线程一般被组织成二维或三维数组的形式,即 blockDim 可以取 (8×8) 、 (16×16) 、 (32×16) 、 $(16 \times 16 \times 2)$ 等。为了最有效地利用 SM 的计算能力,以及最大化地利用 block 的线程数量,应当根据 N_{row} 的大小选择合适的 blockDim。具体如式(1)所示。

$$\begin{cases} \text{blockDim} : (8 \times 8) & \text{if } N_{row} \leq 64 \\ \text{blockDim} : (16 \times 8) & \text{if } 64 < N_{row} \leq 128 \\ \text{blockDim} : (16 \times 16) & \text{if } 128 < N_{row} \leq 256 \end{cases} \quad (1)$$

当使用式(1)中的 (8×8) 方式时,每个 block 中线程数量为 64,则一个 SM 中最多拥有 $64 \times 8 = 512$ 个线程,没有充分地利用 SM 的计算能力,而且只能得到 16 个 warp,每个 SP 平均可以调度两个 warp,这对于需要执行长延时操作(如访问全局存储器)的线程来说是不利的。因此, (8×8) 的组织方式比较适合于无长延时的操作。在体元的剖分过程中,虽然需要从常数存储器中读取等值面阈值,但这种访问速度非常快,整个过程不需要访问速度很慢的全局存储器,此时可以使用 (8×8) 的组织方式。

当 $N_{row} > 256$ 时,如果 blockDim 取 $(16 \times 16 \times 2)$ 或 (32×16) ,虽然在一个 block 中没有超过 512 个线程,但是在一个 SM 中只能分配两个 block。

GT240 中含有 30 个 SM,那么在一个时刻上只有同时执行 60 个 block。如果一个线程网格 grid 包含大量的 block,此时将会增加总的执行时间。在这种情况下,增加每一个线程的处理量,即由原来一个线程处理一个体元,增加到处理两个体元,这时依然可以选择 (16×16) 的线程组织方式。当 N_{row} 更大时的情况依此类推。

每个线程处理多个体元(设为 n 个)的情况存在两种不同的处理方式。一种是线程处理连续的 n 个体元块,另一种是每隔 m (m 为线程的数量,且 $m \times n$ 等于体元个数)个体元选择一个需要处理的体元,图5中 $m = 3, n = 4$ 。存储器最高效的存取方式是连续存取。方式(a)虽然从逻辑上是按照连续的方式访问每个体元,但是当前线程需要访问存储器或者执行其他长延时操作时,CUDA 就将当前线程切换到下一个线程。而下一个线程处理的第一个体元与上一个线程处理的体元并不是连续存储的,因此形成了跳跃式的存储器存取。方式(b)虽然从逻辑上是跳跃式访问,但在实际执行时,切换线程后,后一个线程将从上一个线程访问的地址后取数据,实现了存储器的连续访问。

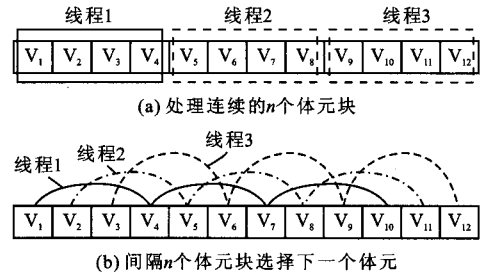


图5 线程处理多个体元的两种不同方式

线程网格的维度 gridDim 限制较小,取为 (N_{col}, N_{slice}) 。

3.2.2 Kernel_dividingVoxel 中的线程结构

Kernel_dividingVoxel 函数主要对 tex_dividingCubes 纹理中的体元进行剖分操作,而 tex_dividingCubes 中体元个数 N 是已知的,因此可以采用式(2)的线程结构分配方案:

$$\begin{cases} \text{blockDim} = (16, 16) \\ \text{gridDim} = \frac{N}{16 \times 16} \\ \text{if } \frac{N}{16 \times 16} \leq 65535 \\ \text{gridDim} = \left(65535, \frac{N}{16 \times 16 \times 65535} \right) \\ \text{if } \frac{N}{16 \times 16} > 65535 \end{cases} \quad (2)$$

3.3 存储空间分配

在目前的 CUDA 架构模式中, GPU 无法直接读取外部存储器和内部存储器中的数据, 需要首先由 CPU 将三维体数据读入内存中, 并用 `cudaChannelFormatDesc` 结构体描述 CUDA 数组的数据类型, 然后通过 `cudaMalloc3DArray()` 功能函数分配一个与体数据相一致的 CUDA 数组, 并利用 `cudaMemcpy3D()` 功能函数将其载入 GPU 内的全局存储器. 由于全局存储器无缓存功能, 访问速度慢, 因此需要将 CUDA 数组绑定到纹理存储器中, 利用纹理存储器的缓存功能加速数据的访问, 并且其具有的滤波功能有利于图像的插值处理. 为此, 需要声明一个 `float` 类型的三维纹理参照系 `texture < float, 3, cudaReadMode Element Type > tex`, 并设置纹理参照系的寻址模式、是否归一化等属性. 为保证纹理的坐标独立于尺寸大小, 纹理参照系采用归一化钳位寻址, 那么, 在纹理缓存中三个维度上的坐标都在 $[0, 0, 1, 0]$ 范围内. 纹理中存储的数据采用 `float1` 类型 (`cudaChannelFormatDesc` 描述 CUDA 数组时指定 `x` 值为 32, 数据类型枚举变量为 `cudaChannelFormatKindFloat` 类型), 体数据中的每一个数据点对应纹理中的一个纹理单元.

等值面阈值信息由主机端指定并传送给设备端中的存储器中. 由于等值面阈值需要经常改变, 而且在 kernel 执行时需要频繁地访问, 因此将其存入常数存储器中, 利用常数存储器的缓存机制加快访问速度, 最快时能够在一个 GPU 时钟周期内访问到所需的数据.

在剖分过程中, 体元是逐步细分的, 每一次剖分后都会在原有体元 8 个顶点的基础上增加 19 个新的顶点, 如图 6 所示. 每一个新顶点的顶点属性值和法向量都是由原体元的顶点值通过三线性插值计算得到. 在每一次的体元检测时都需要将顶点值与等值面阈值作比较, 最好的情况下一个顶点只需要计算一次, 但最坏情况下确需要重复计算 8 次, 而每一次顶点值的计算又都包含大量的插值计算. 为了减少体元检测时的计算量, 在体元一次剖分后将 19 个新顶点的值一次性地计算出来, 并存储在访问速度比较快的存储器中, 以提高后续体元检测的速度. 由于每个新顶点需要 4 个字节的存储空间, 假设每个线程块 block 需要创建 256 个线程, 那么此时就需要 $16 \times 16 \times 19 \times 4 = 19 \text{ kB}$. 以 NVIDIA GT240 为例, 每个 SM 的共享存储器为 16 kB, 小于 19 个新顶

点所需要的存储空间; GT240 中每个 SM 拥有的寄存器数量为 16 384, 平均每个线程可以使用 16 个寄存器, 但寄存器分配的过多会导致整个 SM 中资源不足而使得 SM 减少线程块的数量, 从而减少线程数量和 warp 数量, 造成 SM 的利用率下降. 为此, 将 19 个新增顶点中可能被访问最多的 7 个顶点 (图 6 中标“●”的顶点) 值存储在寄存器中, 而访问次数相对较少的 12 个顶点 (图 6 中标“○”的顶点) 存储在共享存储器中. 这样的存储分配策略既可以将 19 个顶点值存储在较快的存储器中, 又可以充分利用 SM 中的存储资源和计算资源.

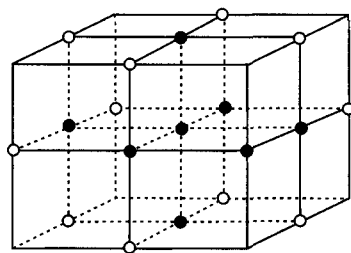


图 6 体元一次剖分后新增加 19 个顶点

4 实验结果与分析

对提出的基于 CUDA 的 Dividing Cubes 算法在 Windows XP 系统上进行了实验. 实验的硬件平台为 Intel i3 2.4 GHz, 内存 2 GB, 图形处理器 NVIDIA GeForce 240, 显存 512 MB; 实验的开发软件平台为 Visual Studio 2010 C++, CUDA toolkit 5.0; 选用了两组实验数据, 大小分别是 $128 \times 128 \times 128$ 和 $256 \times 256 \times 128$.

表 1 给出了 Dividing Cubes 算法分别在 CPU 和 GPU 下实现时所需的时间. 在传统 CPU 计算下, 从较大的体数据中提取等值面所耗费的时间长达 5s 之多, 而利用 GPU 的并行加速能力, 在体数据为 $128 \times 128 \times 128$ 时, 能够具有 7 倍的加速比, 而在体数据为 $256 \times 256 \times 128$ 时, 可以达到将近 10 倍的速度提升. 由此可以看到, 利用图形硬件可以有效地提高等值面的绘制速度.

5 结束语

本文将 Dividing Cubes 算法移植到图形硬件 GPU 上执行, 在 CUDA 架构下实现了该算法. 以 NVIDIA GeForce 240 为硬件实验平台取得了较高的速度提升, 可以预见, 在更高级的 GPU 上还可以达到更高的加速比.

表 1 Dividing Cubes 算法在不同条件下实现时的性能比较

体数据大小	数据点数量	体元数量	剖分体元数		等值面提取时间/ms		加速比
			CPU	GPU	CPU	GPU	
128×128×128	2 097 152	2 048 383	6 145	6 228	2 070	288	7.2
256×256×128	8 388 608	8 258 175	4 772	4 996	5 526	573	9.6

注:由于观察视角不同,在 CPU、GPU 下的剖分体元数量有所不同

参考文献:

[1] Lorensen W E, Cline H E. Marching cubes: a high resolution 3D surfaces construction algorithm [J]. Computer Graphics, 1987,21(4):163-169.

[2] Doi A, Koide A. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells [J]. IE-ICE Transactions on Information and Systems, 1991, E74-D(1):214-224.

[3] Cline H E, Lorensen W E. Two algorithms for three-dimensional reconstruction of tomograms[J]. Medical Physics, 1988, 15(3):320-327.

[4] 张迎平,高国贤,陆一峰,等. 基于区间树硬件加速索引的 Marching Cubes 算法[J]. 计算机辅助设计与图形学学报,2012,24(7):871-878.

[5] Livnat Y, Shen H W, Johnson C R. A near optimal isosurface extraction algorithm using the span space [J]. IEEE Transactions on Visualization and Computer Graphics, 1996, 2(1): 73-84.

[6] Johansson G, Carr H. Acceleration marching cubes with graphics hardware[C]// Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research. New York: ACM Press, 2006: 16-19.

[7] 刘伟峰,杨权一,曹邦功,等. 基于 GPU 的高度并行 Marching Cubes 改进算法[J]. 微电子学与计算机, 2008,25(9):151-154.

[8] 徐美和,唐泽圣,邓俊辉. 用于构造等值面的剖分立方体算法的改进[J]. 计算机辅助设计与图形学学报, 1997,9(3):233-240.

[9] 陈兵,蒙培生. 基于 Dividing Cubes 的算法实现螺旋 CT 数据的三维重建[J]. 计算机应用研究,2003(4):63-64.

[10] David B Kirk, Wen-mei W. Hwu. Programming massively parallel processors: a hands-on approach[M]. San Francisco:Morgan Kaufmann Publishers of Elsevier, 2010.

作者简介:

张文波 男,(1983—),博士研究生.研究方向为计算机网络与分布计算.

曹耀钦 男,(1962—),博士,教授,博士生导师.研究方向为计算机网络与分布计算、软件工程.

孙伟 女,(1980—),博士研究生,工程师.研究方向为岩土工程与科学计算可视化.

李海龙 男,(1978—),讲师.研究方向为电子对抗与网络工程.