山东农业大学学报(自然科学版),2016,47(3):473-476,480 VOL.47 NO.3 2016
Journal of Shandong Agricultural University (Natural Science Edition) doi:10.3969/j.issn.1000-2324.2016.03.030

基于 GPU 通用计算的并行算法和计算框架的实现

朱宇兰
泉州医学高等专科学校, 福建 泉州 362000

摘 要: GPU通用计算是近几年来迅速发展的一个计算领域，以其强大的并行处理能力为密集数据单指令型计算提供了一个绝佳的解决方案，但受限制于芯片的制造工艺，其运算能力遭遇瓶颈。本文从GPU通用计算的基础——图形API开始，分析GPU并行算法特征、运算的过程及特点，并抽象出了一套并行计算框架。通过计算密集行案例，演示了框架的使用方法，并与传统GPU通用计算的实现方法比较，证明了本框架具有代码精简、与图形学无关的特点。

关键词: GPU 通用计算; 并行计算; 计算框架
中图法分类号: TN202 **文献标识码**: A **文章编号**: 1000-2324(2016)03-0473-04

Parallel Algorithm Based on General Purpose Computing on GPU and the Implementation of Calculation Framework

ZHU Yu-lan
Quanzhou Medical College, Quanzhou 362000, *China*

Abstract: GPGPU(General Purpose Computing on Graphics Processing Unit) is a calculation mothed that develops quiet fast in recent years, it provide an optimal solution for the intensive data calculation of a single instruction with a powerful treatment, however it is restricted in CPU making process to lead to entounter the bottleneck of hardware manufacture. This paper started from GPGPU by Graphics API to analyze the featuers, progress and characteristics of GPU parallel algorithm and obtained a set of computing framework to demonstrate it by an intensive line calculation and compared between the traditional GPU and the parallel computing framework to turn out to show that there was a simplified code and had nothing to do with graphics.

Keywords: General Purpose Computing on Graphics Processing Unit (GPGPU); parallel computing; computing framework

GPU 通用计算技术作为一种新兴的计算技术正在处理器架构的领域掀起一场不小的革命。由于不同计算核心之间相互配合的效率问题、散热问题、成本问题等对于 GPU 的计算能力都形成了限制。GPU 通用计算的出现在一定程度上解决了这个问题，不同于 CPU 的单核或多核架构，GPU 的架构是天生众核的，即使是消费级的低端显卡，其中的处理器核心数目也是成千上万的。通常通过使用图形 API 来发挥 GPU 的强大并行计算能力，这种计算需要有极为扎实的图形学功底，因此需要对 OpenGL 或 DirectX 有很深厚的认识。这在很大程度上限制了 GPU 通用计算的发展速度。

1 并行计算框架的设计

1.1 计算过程设计

GPU 通用计算的关键在于通过纹理映射实现科学计算。纹理映射在原有的图形渲染管线中的作用是通过为多边形贴图的方式实现逼真的效果，这是一种不需要增加模型复杂度就能提升渲染真实感的一种搞笑的做法。而 GPU 通用计算正式利用了 GPU 对于纹理映射操作处理高效的特点，使用纹理存储和输出数据，使用纹理映射的过程来实现科学计算[1]。

在 GPU 中，纹理是以缓存的形式存在的，GPU 实现了纹理坐标的差值、转换和采样的过程。纹理的处理是并行计算框架的关键，在本文描述的并行框架中，计算将围绕着纹理映射展开[2]。GPU 通用计算技术的实现凭借的是 GPU 图形流水线强大的大规模并行处理能力，海量的顶点流经相同的流水线成为屏幕上的像素。根据本文的设计，计算框架的计算过程大致可描述为如下 5 个步骤。

1.1.1 *将输入写入到纹理* 并行计算框架最终是通过 CPU 来进行调度的，得到的结果也需要由 CPU 从内存中获得，因此不同于传统图形流水线输出到屏幕，在计算框架中，需要将渲染的结果输出到显存，在这里我们采用纹理作为接受输出的载体，具体的实现技术是通过帧缓存将纹理链接起来[3]。

1.1.2 *设置投影与视图矩阵* 在三维渲染的过程中，投影与视图矩阵用于实现模型的位移、旋转、缩

收稿日期: 2015-03-20 **修回日期**: 2015-04-28
作者简介: 朱宇兰(1979-),女,硕士,讲师,主要研究方向为算法设计与分析、网络管理与安全. E-mail:zhu@163.com

放，以及控制摄像机的属性。其硬件实现的实质是通过这两个矩阵将输入的顶点坐标进行仿射变换，从而达到具有遮挡和近大远小关系的逼真效果。而对并行计算而言，每个像素的值是不具有图像意义的实际数据。是不能对其任意缩放和位移的。因此，我们在这里需要使用正交投影。将投影平面的长宽分别设置为目标纹理的长宽，将视图矩阵设置为单位矩阵。

1.1.3 *视口设置* 视口即在 Open GL 中定义的观察模型的窗口，同时也是投影平面上的可见部分。在三维渲染的过程中，视口的大小可以理解为一个房间窗口的大小，无论窗口是什么样子的，看到的外面的风景都是真实的，但是在通用计算中，视口无论放大还是缩小都会使数据失真。视口必须设置为与纹理图等大小，且必须与投影平面对齐。视口设置的 API 为 gl View Port(0,0,width,height)，该接口的作用是设置与投影平面对齐且等大的视口[4]。

1.1.4 *绘制矩形实现计算* 绘制一个与纹理图等大小的矩形，四个顶点的纹理坐标分别设置为纹理的四个顶 点，即（0，0），（0，1），（1，1），（1，0）。由于矩形的大小与贴图完全相同，这也 就保证了纹理图中的每个像素都得到了覆盖，映射的比例是 1:1，这种映射的方式等同于数据的复制，在接下来的过程中，纹理中的每个像素会被读取到流水线的入口，进入顶点着色器，继而为每个片段着色为对应的像素。在这里我们既可以使用单位化 的纹理坐标。即取值在 0~1 之间的纹理坐标，可以使用非单位化的纹理坐标，根据 GPU 支持情况的不同可以自行选择。

1.1.5 *读回数据* 经过绘制过程之后，渲染的结果就已经存储在了显存的纹理之中，这时可以通过 gl Read Buffer、gl Read Pixel 等函数将数据读回内存。本文描述的计算框架中将会封装读取数据的步骤，如果计算的结果不是最终结果，纹理即可以作为中间变量继续参与下一步的计算，如果纹理已经是最终结果，则可以通过框架的输出函数予以取回。

1.2　GPU 存储接口设计

框架的存储结果是通过对显存的纹理进行封装来实现的。纹理可以理解为一张查找对应颜色的取色板，通过顶点的纹理坐标既可以从纹理中找到对应的像素颜色。纹理在显存中的存储格式与帧缓存是类似的，都是由离散的像素构成。由于纹理坐标都是经过差值计算得到的，因此取到的颜色并非离散，而是通过周围像素差值得到的或者是采用最邻近元素得到的[5]。将纹理看做是连续的数组更为恰当，二维坐标是在实数域内定义的。

纹理坐标分为单位化与未单位化两种。单位化的纹理坐标范围在 0~1 之间，这样即使在渲染的时候不知道纹理的大小也能够正确的指定出正确的纹理坐标。但是对于 GPU 通用计算而言，这种做法却打破了纹理作为数据存储容器本身的便利性。因此，本框架在对存储单元的封装中为用户提供了非单位化的数据存取接口，用户可以直接通过数据的下标访问到纹理中的特定数据。

1.3　框架管理接口设计

除了函数与存储单元这两个计算用单元之外，还会有一些其他的辅助部分用于框架的实现与运行，由于传统的 GPU 通用计算中需要有大量的功过用于图形 API 的初始化，兼容性检查，Open GL 对象的生成、管理与删除等工作，所以框架管理类是非常必要的设计[6]。

表 1 框架管理类接口列表

Table 1 Interface list of framework management class

接口名称 Name	接口作用 Function
初始化框架的功能，铺垫好通用计算环境	包括初始化GL 库函数以及扩展库等
运行环境检测	检测环境是否支持框架的运行
框架清理	用在运行的最后，释放框架管理的资源
生成GPU 函数	管理并生成GPU 函数
生成GPU 数组	管理并生成GPU 数组

2　并行计算框架的实现

2.1　GPU 函数的实现

计算单元的功能类似于 CPU 程序里面的函数，不同的是，这里的计算单元是完全并行计算，并且在 GPU 上运行的。由于其与 Open GL 中 Shader 的相关性。确定 Gpu Function 的输入与输出。由

于图形管线的末端是 FBO，最后的计算结果是作为一张二维纹理贴图的形式从 FBO 上读取出来的，因此，在这里我们限定 Gpu Function 的输出是一个二维数组[7]。由于二维数组和三维数组的实质是纹理贴图，根据 GPU 的限制，使用的最大数组数目是 8 个，因为计算的中间结果作为 GPU 中的存储是不会与内存发生交换而耗费时间的。因此这里应该通过多次的计算来达到相同的目的。为了使 GPU 函数的运算过程更加类似于 CPU，并且减少误操作的概率，Gpu Function 的赋值操作是在实际运算之前发生的。用户可以在计算之前的任何时间为 Gpu Function 添加参数。但实际赋值过程会与运算紧邻发生。这里是通过为 Gpu Function 添加了参数队列实现的。

2.2　存储单元的实现

为了尽可能避免出现图形学中的内容，本框架将存储结构由传统的纹理封装为与 CPU 数组具有相似特性的 GPU 数组，分为 GPU 二维数组（Gpu Array2D）类以及 GPU 三维数组（Gpu Array3D）两个部分，这两个部分都继承自 GPU 数组（Gpu Array）类。Gpu Array 实质是对纹理存储单元的封装。包含一个 GLuint 类型的变量 textureId。该变量是在 Gpu Array 实例化的初期赋值的，其值代表了对应纹理的纹理 Id。

2.3　框架管理器的实现

计算框架的封装是整个实现过程最核心的部分，计算框架管理类是控制整个计算框架的核心，计算框架的管理是通过 Framework Manager 类来实现的。该类的实现借鉴了 Open GL 状态机的实现方式，对其内部的资源进行统一的管理。Framework Manager 类包含一个 Gpu Object 指针类型的数组[8]。Gpu Object 类是所有在该框架中适用的类的基类，在该框架中实现的所有的类都是从 Gpu Object 类派生而来的。该类的结构如下所示：

Class Gpu Object；

public:static const int Gpu Class Serial Id=0；

public:int Gpu Object Serial Number；

该类包含两个主要属性，分别为静态整型常量 Gpu Class Serial Id 以及整数类型属性 Gpu Object Serial Number。

3　体渲染在框架上的实现

体渲染是一种将离散的三维数据集投影到二维平面的绘制技术。选用该算法对框架的实现进行验证，运行的过程中，GPU 充当的是并行数据处理的角色，计算得到的结果通过写入到文件来呈现。典型的三维数据是 CT 或者核磁共振得到的一组二维切面图像，这种三维数据的渲染可以通过提取等值曲面渲染的方式或者是直接渲染体素的方式渲染。其中，Ray-casting 算法是体渲染最简单的实现方法，目标图像的每一个像素都作为一道垂直于平面的射线穿透体素数据。在穿透体素数据的过程中，射线会累积体素的颜色信息，并累加，最终成为该点的颜色信息。如图 1 既是高质量体渲染得到的人颅骨体素信息的渲染效果。

通过使用本文设计实现的框架来实现光线投射算法的体渲染。使用 GPU 三维数组来存储体素数据，使用 GPU 二维数组来存储视线方向以及遍历的步长数据，单独开辟一个 GPU 二维数组用于存储计算结果。在 GPU 上完成体素数据的累积操作。并将最终结果显示出来。

3.1　生成渲染数据

经体渲染 Ray-casting 算法的原理可知，算法需要有两个 Gpu Array 来存储数据，分别为用于存储计算结果的 GPU 二维数组 Gpu Array 2D 指针类型的变量 g Arr Out，以及用于存储计算结果的 GPU 三维数组 Gpu Array3D 指针类型的变量 g Arr VT。

渲染数据存储在二进制的文件“backpack8.raw”中，该文件内部由一组连续的 8 位整数构成的，每个数据代表一个体素的灰度值。体素数据的规格是 512*512*373，可以理解为使用分辨率为 512*512 的扫描设备进行了 373 次分层扫描，得到的数据组成了体素数据，文件大小为 93.25 MB。

体素数据文件的读取是通过标准 C++库进行的，使用二进制方式读取，读取的数据存储在一个 byte 数组里。数组的空间是在文件读取结束之后动态申请的，其长度为文件中体素的个数*4，这是因为最终在存储为 BMP 图像时需要有 RGBA 分量。

3.2 创建并行计算函数

体渲染并行计算的算法大致是这样的，输出数组中的每个元素，根据自己所在的线程的 Id 编号 GPU_FUNCTION_ID_X 与 GPU_FUNCTION_ID_Y 分别到传入的三维数组里面查找 XY 坐标与之对应的元素，并根据 Z 坐标从小到大的顺序，对其灰度根据一定的规则进行叠加，将结果写入输出数组中。

3.3 框架初始化

需要创建一个 EurekaFramework::FrameworkManager 对象 framework_manager 以及一个 EurekaFramework::GpuFunction 指针对象 func。通过调用 framework_manager 的 init()方法实现框架的初始化。对于 GPU 数组类数据需要调用方法 AssignValue 来进行初始化，具体代码如下所示：

framework_manager.init();create_volumetexture();

G Arr Out =framework_manager.Generate Gpu Array 2D(VOLUME_TEX_WIDTH, VOLUME_TEX_HEIGHT); g Arr Out->Assign Value(NULL);

3.4 计算过程

首先使用框架管理器为 func 创建对象，然后通过 func 加载并行计算函数代码，最后对函数进行参数设置并计算，具体代码如下所示：

```
func = framework_manager.GenerateGpuFunction("frag.txt");
func->SetParamArray3D("volume_tex", gArrVT);
func->SetParamInt("frameWidth", VOLUME_TEX_WIDTH);
func->SetParamInt ("frameHeight", VOLUME_TEX_HEIGHT);
func->SetParamInt("frameDepth", VOLUME_TEX_DEPTH); func->exec(gArrOut);
```

3.5 读回数据

计算完成后将 g Arr Out 中的数据读取到内存通过一个图像函数类写入到 BMP 文件中既可以看到渲染结果。渲染结果如图所示。

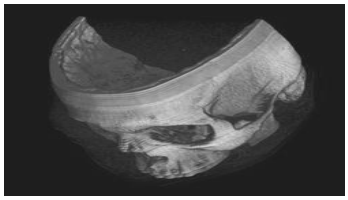

图 1 颅骨体素信息渲染效果

Fig.1 Rendering effects of voxel information in skull

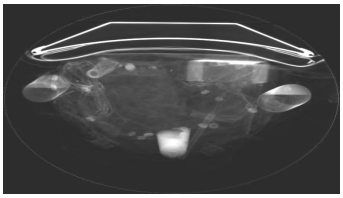

图 2 图像函数在 BMP 文件中渲染结果

Fig.2 Rendering effects of image function in BMP files

3.6 结果分析

对于同一种算法，同样使用图形 API 的方法进行了实现，经过与使用并行框架的代码进行对比可以发现，使用框架的代码量仅仅为图形 API 代码量的一半左右。并且在整段代码中并没有任何关于 OpenGL 的代码出现。这样的设计对于从未使用过图形 API 的用户来讲是一种极大的便利。相对于传统图形 API 的算法，框架的算法更加类似于 CPU 的多线程算法，用户不需要去理解繁琐的图形管线设计，更免去了根据管线特性设计并行计算算法的麻烦。

4 总 结

针对类似于体渲染这类的计算密集行应用，使用 GPU 通用计算技术是良好的解决方案。但是频

（下转第 480 页）