**13 计算着色器**

NVIDIA公司开发的“Fermi”架构最多可支持16个流式多处理器，而每个流式处理器又均含有32个CUDA核心，也就是共512CUDA核心。

我们将GPU用于非图形应用程序的情况称为通用GPU程序设计(GPGPU)。当然，并不是所有的算法都适合由GPU来执行，只有数据并行算法才能发挥出GPU并行架构的优势。

对于GPGPU编程而言，用户通常需要将计算结果返回CPU供其访问，这就需要将数据由显存复制到系统内存，虽说这个过程的速度较慢，但是与GPU在运算时所缩短的时间相比是微不足道的。

针对图形处理任务来说，我们一般将运算结果作为渲染流水线的输入，所以无需再由GPU向CPU传输数据。从本质上来说，计算着色器能够使我们访问GPU来实现数据并行算法，而不必渲染出任何图形。我们可以将计算着色器输出的数据直接绑定到渲染流水线上。

**13.1 线程与线程组**

在GPU编程的过程中，根据程序具体的执行需求，可将线程划分为由线程组构成的网格。一个线程组运行于一个多处理器之上。因此，对于拥有16个多处理器的GPU来说，我们至少应将任务分解为16个线程组，以此令每个多处理器都充分运转起来。但是，要获得更佳的性能，我们还应当令每个多处理器至少拥有两个线程组，使它能够切换到不同的线程组进行处理，以连续不停地工作。

每个线程组中都有一块共享内存，供组内的线程访问。但是，线程并不能访问其他组中的共享内存。同理，同组内的线程间能够进行同步操作，不同组的线程间却不能实现这一点。

一个线程组中含有n个线程。硬件实际上会将这些线程分为多个warp(每个warp中有32个线程)，而且多处理器会以SIMD32的方式(即32个线程同时执行相同的指令序列)来处理warp。每个CUDA核心都可处理一个线程。在Direct3D中，我们能够以非32的倍数值来指定线程组的大小。但是出于性能的原因，我们应当总是将线程组的大小设置为warp尺寸的整数倍。

对于各种型号的图形硬件来说，线程数为256的线程组是一种普遍适于工作的初始设置。我们可以以此值为基础，再根据具体需求尝试将其调整为其他大小。值得注意的是，修改每个线程组中的线程数量也会对线程组的分派(dispatch)次数产生影响。

在Direct3D中可以通过调用下列Dispatch方法来启动线程组。

**13.2 一个简单的计算着色器**

[numthreads(16,16,1)]

void CS(int3 dispatchThreadID : SV\_DispatchThreadID){

}

一个计算着色器由下列要素构成：

1. 通过常量缓冲区访问的全局变量；
2. 输入与输出资源
3. [numthreads(X,Y,Z)]属性，指定3D线程网格中的线程数量
4. 每个线程要执行的着色器指令
5. 线程ID系统值参数

为了开启计算着色器，我们还需使用其特定的“计算流水线状态描述”。使用D3D12\_COMPUTE\_PIPELINE\_STATE\_DESC结构体。

根签名定义了什么参数才是着色器所期望的输入(CBV，SRV等)，而cs字段就是所指定的计算着色器。

**13.3 数据的输入与输出资源**

能与计算着色器绑定的资源类型有缓冲区与纹理两种。

13.3.1 纹理输入

通过给输入纹理创建SRV，再将它们作为参数传入根参数，我们就能令这两个纹理都绑定为着色器的输入资源。这个过程与着色器资源视图绑定到像素着色器的方法相同。

13.3.2 纹理输出与无序访问视图

在前一节的计算着色器代码中，我们定义了一个输出资源：

RWTexture2D<float4> gOutput;

计算着色器处理输出资源的方式比较特殊，它们的类型还有一个特别的前缀”RW”，意为读与写。

输出资源与输入资源的绑定方法是全然不同的。为了绑定在计算着色器中要执行写操作的资源，我们需要将其与称为无序访问视图的新类型视图关联在一起。在代码中，我们用描述符句柄来表示无须访问视图，且通过结构体D3D12\_UNORDERED\_ACCESS\_VIEW\_DESC来对它进行描述。

(创建无序访问视图例子)

13.3.3 利用索引对纹理进行采样

纹理元素可以借助2D索引加以访问。我们可以基于分派的线程ID来索引纹理，而每个线程都要被指定一个唯一的调度ID。

[numthreads(16,16,1)]

void CS(int3 dispatchThreadID : SV\_DispatchThreadID){

gOutput[dispatchThreadID.xy] =

gInputA[dispatchThreadID.xy] + gInputB[dispatchThreadID.xy];

}

计算着色器在通过纹理过滤来对纹理进行采样时候，有两个问题，第一个问题是，我们不能使用Sample方法，而必须采用SampleLevel方法。与Sample相比，SampleLevel需要获取第三个额外的参数，以指定mipmap层级。第二个问题是，当我们对纹理进行采样时，会使用范围为[0,1]2的归一化纹理坐标，而非整数索引。

(例子)

13.3.4 结构化缓冲区资源

struct Data{

float3 v1;

float2 v2;

};

StructuredBuffer<Data> gInputA : register(t0);

StructuredBuffer<Data> gInputB : register(t1);

RWStructuredBuffer<Data> gOutput : register(u0);

结构化缓冲区是一种由相同类型元素所构成的简单缓冲区，其本质上是一种数组。

（例子）

13.3.5 将计算着色器的执行结果复制到系统内存

如何将GPU端显存里的结果回传至系统内存中。首先，应以堆属性D3D12\_HEAP\_TYPE\_READBACK来创建系统内存缓冲区，再通过ID3D12GraphicsCommandList::CopyResource方法将GPU资源复制到系统内存资源之中。其次，系统内存资源必须与待复制的资源有着相同的类型与大小。最后，还需要用映射API函数对系统内存缓冲区进行映射，使CPU可以顺利地读取其中的数据。

(VecAdd程序)

13.4 线程标识的系统值

1，系统会为每个线程组都分配一个ID，这个ID称为线程组ID，其系统值的语义为SV\_GroupID。如果Gx x Gy x Gz为所分派线程组的个数，则组ID的范围为( 0, 0, 0)至(Gx – 1, Gy – 1, Gz - 1)。

2，在线程组中，每个线程都被指定了一个组内的唯一ID。若线程组的规模为X x Y x Z，则组内线程ID的范围实为(0, 0, 0)到(X - 1, Y – 1, Z -1)。组内线程ID系统值的语义为SV\_GroupThreadID。

3，调用一次Dispatch函数便会分派一个线程组网络。调度线程ID是Dispatch调用为线程所生成的唯一标志。换句话说，组内线程ID是线程相对于所在线程组的唯一标识(局部)，而调度线程ID则是Dispatch调用为线程指定的相对于所有线程组中全部线程的唯一识别信息(全局)。今设线程组的规模为ThtreadGroupSize = (X,Y,Z)，那么我们便可以根据线程组ID与组内线程ID，通过以下方法推算出调度线程ID：

dispatchThreadID.xyz = groupID.xyz \* ThreadGroupSize.xyz + groupThreadID.xyz；

调度线程ID的系统值语义为SV\_DispatchThreadID。如果分派了一个大小为3x2的线程组，且其中每个线程组的规模为10x10，则供分发了600个线程，而且所调度线程ID范围为(0,0,0)至(29,19,0)。

4，通过Direct3D的系统值SV\_GroupIndex便可以指定组内线程ID的线性索引，它的换算方法为：

groupIndex = groupThreadID.z \* ThreadGroupSize.x \* ThreadGroupSize.y + groupThreadID.y \* ThreadGroupSize.x + groupThreadID.x；

置于坐标的索引顺序，其第一个坐标指出的是线程在x方向上的位置(或称为列)，第二个坐标则是线程在y方向上的位置(或称为行)。这个顺序与普通矩阵的记法刚好相反。

13.5 追加缓冲区与消费缓冲区

假设我们通过结构体定义了一个存有粒子数据的缓冲区，并且希望基于粒子的速度与恒定加速度在计算着色器中对其位置进行更新。此外，我们还假定不必考虑粒子的更新顺序以及它们被写入输出缓冲区的顺序。消费结构化缓冲区与追加结构化缓冲区便是为这种场景而生的。若使用了这两种缓冲区，我们也就不必再在索引问题上花心思了。

数据元素一旦经过处理(即消费)，其他线程就不能再对它进行任何操作了，事实上也就是从消费缓冲区中移除掉了。而且，一个线程也只能处理一个数据元素。一般来说，某元素位于输入缓冲区的位置于其处理后写入输出缓冲区的位置并不是一一对应的。

13.6 共享内存与线程同步

每个线程组都有一块称为共享内存或线程本地存储器的内存空间。这种内存的访问速度很快，可以与硬件告诉缓冲的速度不相上下。在我们计算着色器的代码中，共享内存的声明如下：groupshared float4 gCache[256]；

数组大小可依用户的需求而定，但是线程组共享内存的上限为32kb。由于共享内存是线程组里的本地内存，所以要通过SV\_GroupThreadID语义对它进行索引。据此，我们可以使组内的每个线程都来访问共享内存中的同一个元素。

使用过多的共享内存会引发性能问题。

共享内存常见的应用场景是存储纹理数据。在特定的算法中，例如像模糊图像这种工作，就需要对同一纹素进行多次拾取。纹理采样实际上是一种速度较慢的GPU操作，因为内存带宽与内存延迟还未能像GPU的计算能力那样得到极大的改善。但是，我们可以将线程组所需要的纹理样本全部预加载至共享内存块，以此来避免密集的纹理拾取操作说带来的性能下滑。

有一个问题随之而来，根源在于我们不发保证线程组内的所有线程都能同时完成任务。这可能会导致线程访问到还未经初始化的共享内存元素，因为负责将这些元素进行初始化的相邻线程也许还没有完成工作。要填上这个坑，就一定要先等待所有的线程都将各自所处理的纹理加载到共享内存之中，而后再令计算着色器继续后面的工作，这时候就轮到同步命令闪亮登场了。

13.7 图像模糊演示程序