The Graphics Processing Unit

第一款消费者级别的图形芯片(graphics chip)(NVIDIA的GeForce256，含有vertex processing)于1999年出现。为了区分GeForce25和前只能做光栅化的芯片，NVIDIA发明了术语*graphics processing unit* (GPU)。接下来，GPU逐渐从一个可配置的 复杂的固定功能的管线 ，演化为一个高度可编程的“白板”，开发者可以实现他们自己的算法。各种可编程着色器(*programmable shaders*)是控制GPU的主要方式。为了效率，渲染管线的某些部分仍然只是可配置的，不可编程。

GPU专注于一小类(a narrow set of)高度可并行化的任务，这使GPU具有极大的运行速度。GPUs有专门的硬件(silicon)用于实现z-buffer，快速访问纹理图及其它buffers，找到被一个三角形覆盖的pixels，等等。

着色器的核心(a shader core)是一个小处理器(a small processor)，它完成一些相对孤立任务(isolated task)，例如：把一个顶点从世间空间变换到屏幕空间，或者计算被一个三角形覆盖的某个像素的颜色。每帧由成千甚至成百万的三角形被送到屏幕上，每秒将有数以亿计的着色器调用 (*shader invocations*，指着色器程序运行的单个实例)。

延迟(*latency*)是所有处理器要面对的问题。访问数据会花费一定量的时间。一个基本的观点是数据信息离处理器(processor，可能是CPU也可能是GPU的处理单元)越远，花费的时间越长。例如，访问存储在内存上的信息花的时间比访问存储在寄存器上的信息长。而等待获取数据意味着处理器处理停顿状态，降低了性能。我们希望处理器一直处于忙碌工作的状态。

3.1 Data-Parallel Architectures (数据并行架构)

不同的处理器使用了各种各样的策略来避免停顿(stalls)。CPUs可以多个处理器(multiple processors)，但是每一个大部分都是以线性方式运行代码，限制了SIMD vector processing。为了最小化延迟带来的影响，许多CPU的芯片都有快速局部缓存，其中填充了下次极可能要访问的数据。为了避免停顿，CPUs也采取了一些聪明的技术：分支预测(branch prediction)，指令排序(instruction reordering)，寄存器重命名(register renaming)，缓存预取(cache prefetching)等。

GPUs采取了不同的方法。GPU芯片的大部分芯片面积(chip area)专用于一大堆处理器(dedicated to a large set of processors)，这种处理器被称为*shader cores*(着色器核心)，通常是数以千计的。GPU是一个流处理器，它依次处理有序的相似数据集(ordered sets of similar data)。由于数据的相似性，GPU可以以一种大规模并行的方式(in a massively parallel fashion)处理这些数据，如：一个顶点或像素的集合。着色器调用之间尽可能相互独立，使它们之间没有信息依赖，也不共享可写的存储位置。有时也可以打破这条规则，代价就是可能有潜在的延迟，毕竟这样做会造成一个processor要等待另一个processor完成了才能工作。

GPU针对吞吐量(*throughput*，数据被处理的最大速率)进行了优化。然而，这种极速处理(rapid processing)是有代价的: 因为较少的芯片面积(chip area)用于缓存(cache memory)和控制逻辑(control logic)，每个shader core的延迟通常会比CPU processor的延迟高得多。

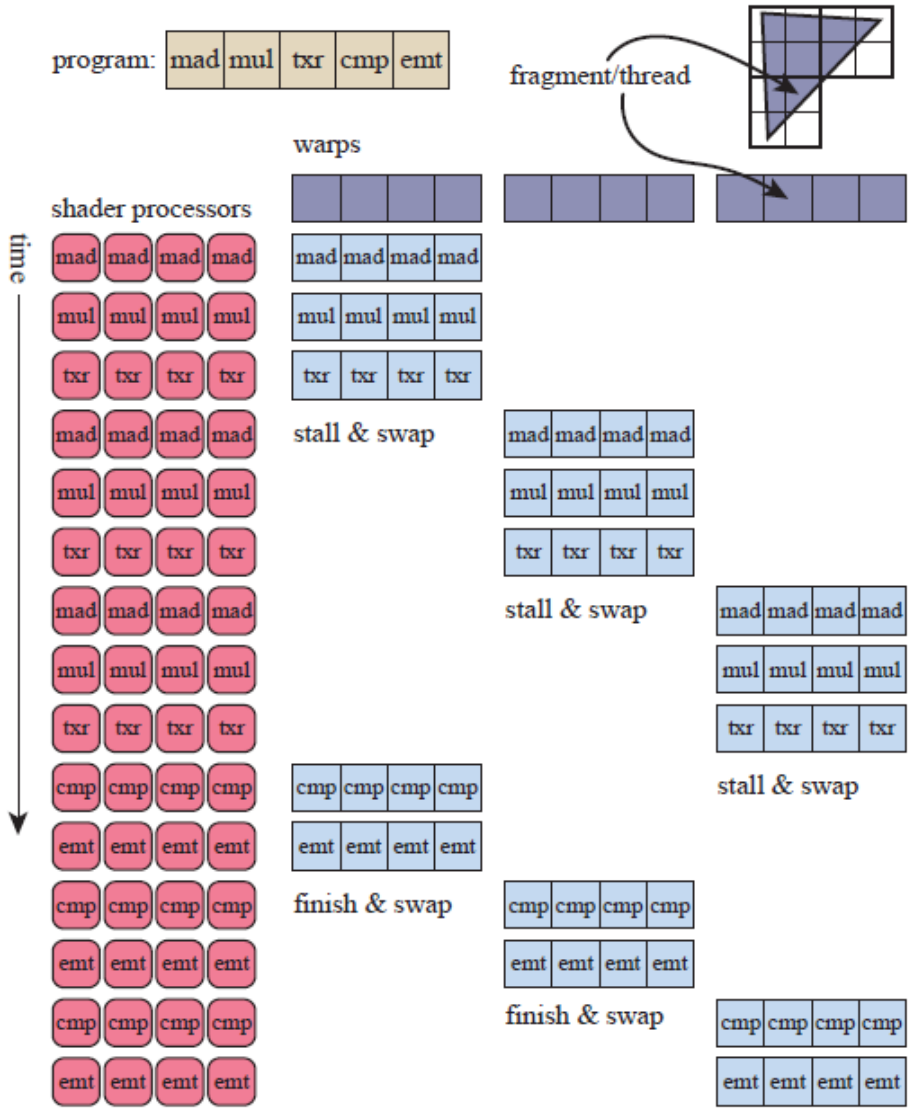
比如说有一个网格被光栅化了，有两千个pixel fragments (一个fragment对应一个像素及其中的数据，如颜色)需要被处理，则一个pixel shader程序要被调用两千次。设想有这么一个GPU(世界上最弱的GPU)，它只有一个着色器处理器(shader processor)。现在，这个GPU开始为第一个fragment(即第一个像素)执行着色器程序(shader program，这里指pixel shader)。着色器处理器对寄存器中的值执行一些算术运算。寄存器是局部的，访问速度极快，没有停顿发生。然后，着色器处理器将要执行一条指令，例如访问纹理(texture access)，对于一个给定的表面位置(surface location)，pixel shader需要知道这个网格(一般一个三维模型由三维网格加上纹理组成)的纹理图像在此位置的像素颜色。一张纹理完全是一个独立的资源，不是pixel shader程序的local memory(局部显/缓存？)的一部分，访问纹理会有点复杂。一次memory fetch (读取内存)可能要使用成百上千个时钟周期(clock cycles)，在这期间GPU processor啥都不干。这个点上，shader processor处于停顿状态，等待纹理颜色的返回。

为了大大改善这个垃圾的GPU的性能，给每个fragment的本地寄存器(local registers)提供一点存储空间。现在，着色器处理器可以切换并执行另一个fragment (2000中的第2个fragment)，而不是在获取纹理时停滞不前。这种切换极快，这两个fragment都不会受到影响，但只需要记住在第一个fragment中执行的是哪条指令。现在，执行的是第二个fragment。和执行第一个fragment一样，先是一些算术运算，然后又需要获取纹理(texture fetch)。此时shader core切换到第三个fragment。最终所有两千个fragments都以这种方式处理了。就在此时，shader processor返回到第一个fragment。这时候，已经取到可用的纹理颜色了，着色器程序可以继续执行。着色器处理器继续以这种方式推进，直到遇到另一条造成执行停顿的指令或程序执行完了。单个fragment的处理时间比shader processor一下处理完这个fragment后再处理别的fragment的方式变长了，但这两千个fragments总的执行时间急剧减少了。

在这种架构中，在遇到等待数据的情况时，GPU就切换到另一个fragment，始终保持忙碌状态，来隐藏延迟(latency)。更进一步地，GPUs把指令执行逻辑和数据分开了。单指令多数据(SIMD)，这种方法一次在固定数量的shader programs上同步(in lock-step)执行一个相同的指令。相比于使用单个逻辑和分发单元来运行每一个program，SIMD的优点是需要专门处理数据和切换(switching，在各个fragments间切换)的硅(silicon，应该指所需要的chip area吧）大大减少。以现代GPU术语，上文提到的两千个fragment例子中，对于一个fragment的每个pixel shader invocation(像素着色器调用)称为一个*thread*。这里的thread和CPU的thread(线程)不是一个东西。它由一些存储空间(a bit of memory, 存储shader的输入值)和任何shader执行所需要的寄存器空间组成。使用同一个shader program(着色器程序)的threads被绑定到一个组中，NVIDIA称这些组为*warps*，AMD称它们为*wavefronts*。一个warp/wavefront由一些GPU shader cores使用SIMD_processing (from 8 to 64)来调度执行。每一个thread被映射为一个*SIMD lane*。

比如说，我们有两千个threads要执行。NVIDIA GUP的一个warp有32个threads。这会生产2000/32=62.5个warps，这意味着要分配63个warps，其中一个warp的一半是空的。一个warp的执行和单个GPU processor类似。一个shader program在所有32个processors中同步执行。当遇到一个memory fetch (获取内存数据) 时，是所有threads同时遇到它(因为它们执行相同的指令)。这个fetch表示包含这些threads的warp将要停顿，每个thread等待对应的(不同的)fetch结果(内存数据)。实际上，与停顿相反的是，这个warp被别一个具有32个threads的warp换出(swapped)去，新的warp会有这个32个cores执行。当一个warp被换进来或换进去时，每一个thread的数据都不会被涉及到(touched)，因此这个交换(swapping)过程和我们的单个processor system一样快。每一个thrad都有它自己的寄存器，每一个warp跟踪在执行的是哪一个指令。在一个新的warp中，swapping仅仅指把一个cores集合指向要执行另一个不同的threads集合，没有别

的开销。Warps要么在执行要么在交换(swap out)，直到所有warps都被完成了。如下图所示:



> *上图是简化的shader执行例子。一个三角形的fragments，被称为threads，它们被分组放进一个个warps中。上图中每个wrap有4个threads，仅是示意，在真实环境下，一个warp可以有32个threads。被执行的shader program (着色器程序)有5个指令长度。首先有4个GPU shader processors为第一个warp中的4个threads执行这个shader program，直到遇到一个停顿条件(stall condition)，即"txr"指令(读取纹理)，它需要时间来获取数据。这时候第二个warp被换进来执行同一个shader program的羊三个指令，直到遇到停顿条件("txt"指令)。然后第三个wrap被换进去、执行、遇到停顿条件，然后把第一个warp换进去继续执行。如果"txr"指令的数据此时还没有返回，执行会真的停顿下来，直到这些数据可用为止。最终每一个warp依次执行完毕。(注意左边的红色圆角方块要从上往下看)*

在我们这个简单的例子(两千个fragment)中，memory fetch的延迟会导致当前warp被(从GPU Cores中)换出去。实际上warps被换出去只花费非常小的延时(delays)，因为swapping的代价很低。尽管也有别的技术用于优化执行，但是warp-swapping是所有GPUs采用的最主要的latency-hiding机制。有几个因素为影响这个过程的执行效率。例如，仅有很少的threads，则仅可创建很少的warps，这会使得latency hiding有问题(problematic)。

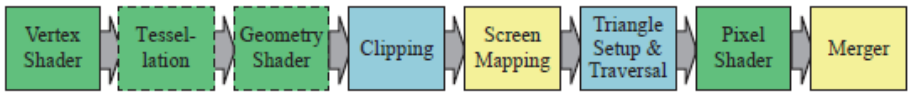
着色器程序(shader program)的结构是影响效率的一个重要特征。每个thread可使用的寄存器的数量是一个重要因素。在我们的例子中，我们假设两千个threads可以同时存在于GPU中。每个thread要执行的shader program需要的寄存器越多，越少的threads，即越少的warps，可以存在于GPU中。Warps的一个缺点可能存在停顿(stall)不能通过swapping减轻的情况。存在GPU中的warps被称为处于"飞行中"的状态(in flight)，处于in flight状态的warps的数量被称为占有率(*occupancy*)。高占有率意味着有许多可用的warps用于processing，很少有空闲的处理器。低占有率往往会导致较差性能。内存获取(memory fetch)的频率，也会影响需要多少latency hiding。

另一个影响整体效率的因素是由`if`语句和循环语句引起的动态分支(dynamic branching)。假如一个shader program遇到一个`if`语句。如果一个warps中所有的threads执行同一个分支(branch)的话，warps可以继续执行，不会care别的分支，也就不是不受影响。然而，如果一些threads选择执行别外的分支，则这个warp必要执行别个分支(为每个thread扔掉不需要的分支执行结果)。这个问题叫作*thread divergence*，即一个warp中的一些threads可能需要执行一个循环迭代或者执行一个`if`语句，而这个warps中的别的threads不需要执行这些循环或`if`，则这些threads这段时间将处于停顿(idle)状态。

所有GPUs实现了这些架构思想，导致系统具有严格的限制但是每瓦特(watt)的计算能力都非常大。接下来，我们要讨论GPU怎样实现渲染管线流程的，可编程shaders是怎么运转的，以及每个GPU阶段(stage)的演变(evolution)和功能(function)。

3.2 GPU Pipeline Overview

GPU实现了第2章中描述的一些概念: geometry processing, rasterization, pixel processing pipeline。这些被分成了几个硬件阶段，它们或是可配置性或是可编程的或是完全固定的。见下图：



> *上图是渲染管线的GPU实现。每个结点的颜色表示操作的用户可控度。绿色的stage表示完全可编程的。虚线框表示可选的阶段。黄色的stages是可配置但不可编程的，例如在merge stage有各种混合模式(blend modes)。蓝色的stages在功能是完全固定的。*

这里讨论GPU的逻辑模型(*logical model*)，以API的形式暴露给程序员。而这个逻辑管线或者物理模型的具体实现取决于硬件提供商(hardware vendor)。这个逻辑模型可以帮助你推理什么会影响性能。

Vertex shader是完全可编程的stage，用于实现geometry processing stage。Geometry shader是完全可编程的stage，在primitive的vertices上进行操作。它可以用于完成: per-primitive的shading操作，销毁primitives，创建新的primitives。Tessellation stage和geometry shader都是可选的，并不是所有GPUs都支持他们，特别是手机设备。

Clipping, triangle setup和triangle traversal stages由固定功能的硬件实现。屏幕映射(screen mapping)受窗口和视口设置影响，内部形成一个简 单的缩放和重定位。Pixel shader stage是完全可编程的。尽管merger stage是不可编程的，但是可高度配置的，可以完成各种各样的操作。它实现了“合并”("merging")功能阶

段，负责修改color, z-buffer, blend, stencil, 及别的输出相关的缓存(buffers)。Pixel shader和merger stage一起构成了第2章中的pixel processing stage。

3.3 The Programmable Shader Stage