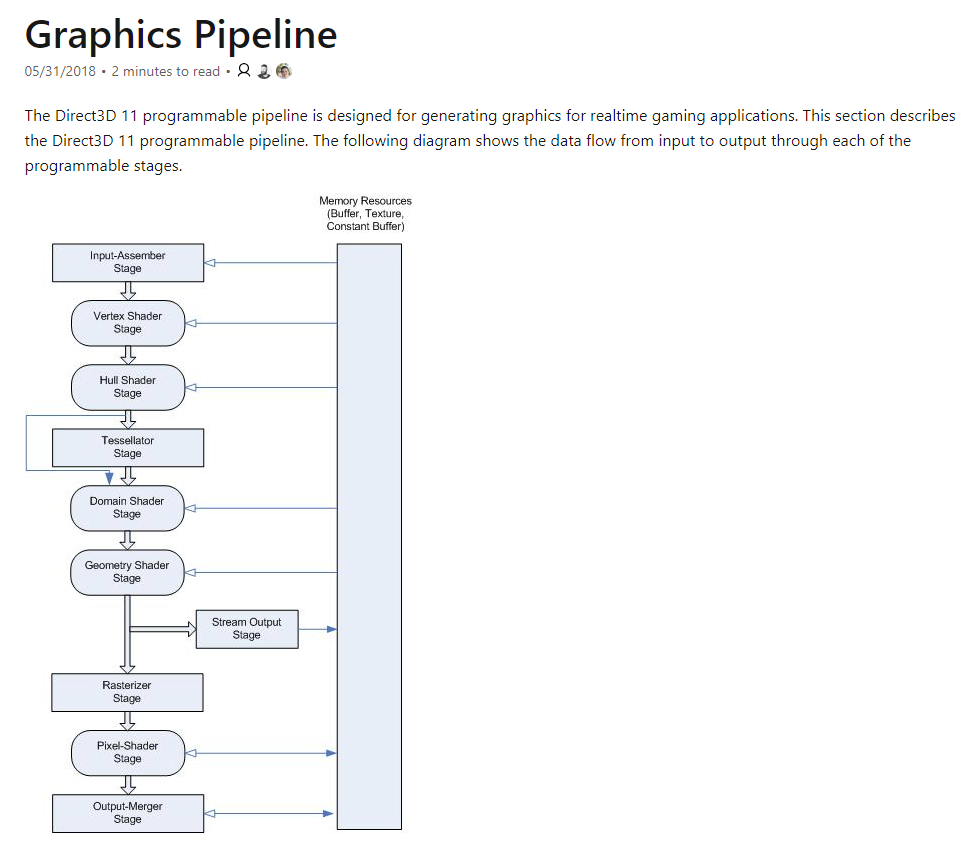
Compute Shader

CS主要是干什么的？为什么我们需要CS？其实，早期的着色器，只有VS跟PS。在更古老的年代，连VS跟PS都没有，就是固定渲染管线。这个就等于直接调接口，传参数，流程，算法，全是固定的，你不能自己弄。

后来，着色器越来越多，增加了GS，CS，还有DS，HS。详细的可以看看下去，下图直接从MS上截取下来的，这种事我很少干，不过这里好像找不到更好的描述方式了。



有兴趣的，可以自己直接上MS官网看详细的。

<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline>

这里贴一下链接算了。

这里我觉得需要强调一下。我基本上只搞Windows开发，主要用DX11/12跟C++，所以这里的管线跟一些默认的术语，都是有局限的。例如渲染管线，其他图形API很可能都有自己的，不大可能都是一样的，毕竟分属于不同的组织。甚至说不定其他管线压根没有什么CS/HS/DS，也可能换了完全不同的名字或者概念。So，不要跟我抬杠说这个架构跟\*\*\*完全不同，是不是搞错了，或者说这个架构有什么毛病，不好用诸如此类的问题。

为什么需要那么多着色器？其实还是跟需求有关。例如GS，主要是GPU粒子的优化用，而DS，HS这类，主要是那个Tessellation的时候用到。Tessellation是个什么东西？我其实有计划有单独的章节。这里稍稍粗略解释一下。首先，我们抛出来一条结论：

一切曲线可以分解成连续的折线/线段！

这个很好理解。同理，我们可以知道：

一切的模型，可以分解成N个三角形。

这些我应该讲过，没讲过再讲一次，这都理解不了的，洗洗睡吧，搞图形学绝对没前途。

那么，三角形越多，自然也就是效果越好，这就是通常说的精模/高模。但是，三角形当然不能无限多，太多的话，计算量太大，显卡负载不了。而且太多的话，由于光栅化的渲染原理，当模型很远的时候，容易造成重面闪烁，也就是所谓的z-fighting，反而效果不好，所以，我们需要做的是LOD，越远的话，模型越粗，越近越精细。Tessellation其实就是传统LOD的延申，传统LOD（Level Of Detail），有很多的局限性，这个以后我写优化章节再仔细讲。反正结论就是：Tessellation是补充了LOD的不足，让近处模型更精细，效果更好。所以新增加了DS，HS。

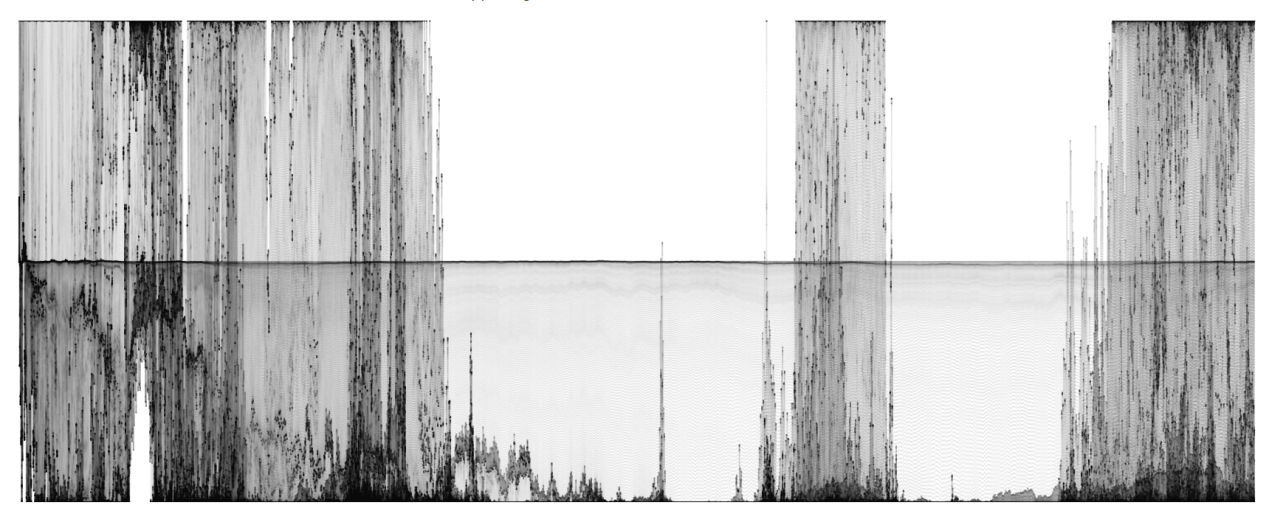
回到正题，CS有什么用呢？我们为什么需要CS？我认为，满足两个条件，那么你可以使用CS。

1. 当你有大量相似/相同的计算的时候。例如，你现在需要做1000000个矩阵乘法。
2. 当大量相似/相同的计算频率非常高的时候。例如，你每帧需要做一次1000000个矩阵乘法。

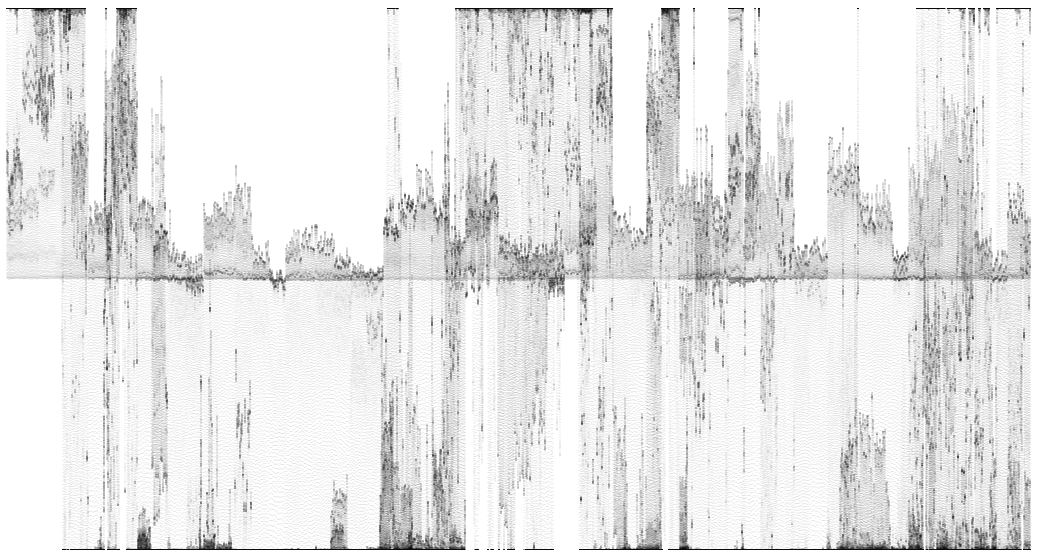
通俗易懂的说法就是：高频的重复计算。假设你有需要做1000000次矩阵乘法，但是你只需要算一次，那么你其实完全不需要用到CS，毕竟只算一次，游戏/程序启动的时候算了就完事了，谁也不会太在乎加载慢了那么一点点。反过来，如果你每次只需要做一个矩阵乘法，但是每帧算一次，那么你也没必要用CS，CPU就够了。

所以，满足以上两个需求的不会太多，我看貌似MS提供的CS的sample里，好像是用CS做了一个排序。这其实完全就是sample了。

最常见的CS用处，就是forward+渲染。老实说，渲染架构从forward到deferred到forward+，有点让人眼花缭乱。不过老实说，还是有本身的意义的。等我讲到deferred的时候，再顺带讲讲forward+，这里只是简单讲讲CS的应用。我以我自己做过的一个CS例子作为案例，介绍一下CS。



这是网上看的一个泼墨的效果，大佬做的。我一时技痒。也用CS做了一个。



这个是我自己做的，用的DX11跟CS。

老实说，我做的跟大佬做的还有差距，我后来想了一下，觉得需要加入Perline Noise才能有更好的效果。不过代码还没写完，就太忙了，没空搞了。就一直放着了。

首先要理解一点，CS是独立于渲染管线之外。上面有个渲染管线的图，你找不到CS的位置。也就是说，CS是个完全独立的东西，你任何时候，不做渲染的时候，你都可以通过CS来实现大量的计算。当然了，大量并行的计算，用CS不说好不好，GPU有更好的支持，例如说CUDA，例如说OpenCL。所以CS我猜测主要还是在渲染的时候用。

由于CS是大量并行的计算，所以就涉及一个东东：线程，线程组。

废话不多说，直接上代码，然后大概解释一下。

[numthreads(1024, 1, 1)]

void ComputeVertexBufferHistoryPixel(uint3 Gid : SV\_GroupID,

uint3 DTid : SV\_DispatchThreadID,

uint3 GTid : SV\_GroupThreadID,

uint GI : SV\_GroupIndex)

{

// refresh velocity and position now

if (RefreshVelocity == 1)

{

float2 uv = float2(Data[GI].Position.x, 0.5f);

float y = rand(float2(uv.x, Time));//noise2d(float2(uv.x, Time));

Data[GI].CurrentVelocity += y;

//Data[GI].CurrentVelocity = y;

}

//VertexBuffer[GI \* 2] = Data[GI].Position;

float RealVelocity = Data[GI].CurrentVelocity \* VELOCITY\_SCALE;

Data[GI].Position += float3(0, TimeDelta \* RealVelocity, 0);

Data[GI].Position.y = clamp(Data[GI].Position.y, -FAR\_DISTANCE, FAR\_DISTANCE);

float3 Original = Data[GI].Position;

// VertexBuffer[GI] = Original;

int VertexPos = 6 \* GI;

float3 QuadVertex[4];

float xOffset = 2.0f / 1920.0f;

float yOffset = 2.0f / 1080.0f;

if (RealVelocity < 0)

{

yOffset = -yOffset;

}

QuadVertex[0] = float3(Original.x - xOffset, Original.y, Original.z);

QuadVertex[1] = float3(Original.x - xOffset, Original.y + yOffset, Original.z);

QuadVertex[2] = float3(Original.x + xOffset, Original.y + yOffset, Original.z);

QuadVertex[3] = float3(Original.x + xOffset, Original.y, Original.z);

// 逆时针

VertexBuffer[VertexPos++] = QuadVertex[0];

VertexBuffer[VertexPos++] = QuadVertex[3];

VertexBuffer[VertexPos++] = QuadVertex[2];

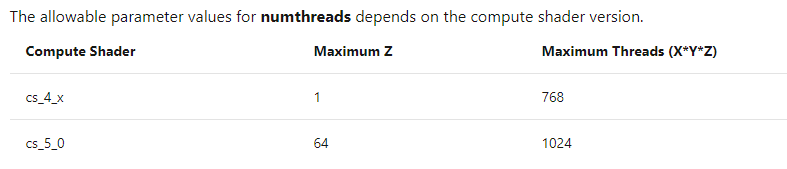
VertexBuffer[VertexPos++] = QuadVertex[0];

VertexBuffer[VertexPos++] = QuadVertex[2];

VertexBuffer[VertexPos++] = QuadVertex[1];

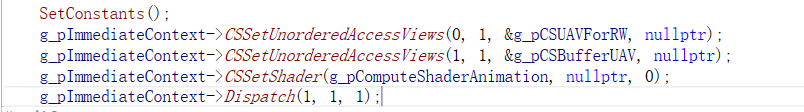
}

第一个，线程数。这里用了一个三维的，定义线程数。这个在CS 5里面，最大是1024.也就是说，一个线程组，最大只支持1024个线程数（CS 5.0），有图为证：



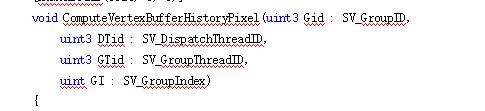
这意味着什么呢？例如，[numthreads(1024, 1, 1)]，这个定义，已经到了极限；当然，你也可以二维定义，例如[numthreads(32, 32, 1)]，这也是到了极限。[numthreads(32, 32, 2)]，这个定义就是错的。什么时候应该一维定义，什么时候应该二维定义？这个，完全取决于你的应用。例如你处理一张二维图片，估计用二维的写法，会简单好理解一点。这就好比写代码，理论上一维数组跟二维数组，都是占那么多的空间，那你什么时候该用一维数组，什么时候该用二维数组呢？难道用一维数组完全替代二维数组，自己算下标不是一样可以的嘛？

Dispatch又是什么用呢？这个，就涉及到大量的线程了。以forward+为例，当你屏幕分割的时候，一个格子大小是16 \* 16，但是你屏幕分辨率是1280 &\* 720，那么，你的线程数就是1280 / 16 = 80，720 / 16 = 45.你现在的线程数是80 \* 45 > 1024。意思是，一个线程组已经不够用了。这个时候，你需要多个线程组。Dispatch的意思，其实就是一次执行多个线程组。下面是我的CPP代码，DX11里面的：



这里面，有一个需要注意的技巧：线程组尽量不要浪费。还是以forward+那个为例。你现在有80 \* 45 = 3600个线程，怎么划分线程组才是最合理的？你当然可以每个线程组用1024个线程，那么你现在需要dispatch的次数是：3600 / 1024 = 3.515625。那么最低就是4.这样一来，其实有接近512个线程浪费掉了。这个应该很好理解吧？浪费太多，不可取。因此，这里其实也是一些细节活。

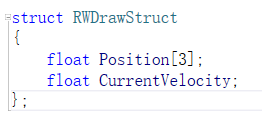
现在理解了线程组，线程，Dispatch的概念，上面几个参数其实很好理解了。你执行一个CS代码的时候，你可能需要知道：当前正在执行的线程组ID，正在执行的线程ID，总的线程分类，线程组分类。所以CS代码里面，这些都是默认就传进来的。也就是这几个参数：



这些其实到处可以查文档的东西，忘记了的时候，随手查查就有了，我也懒得去查一次验证。理解大概的原理基本上写起来都很顺手了。

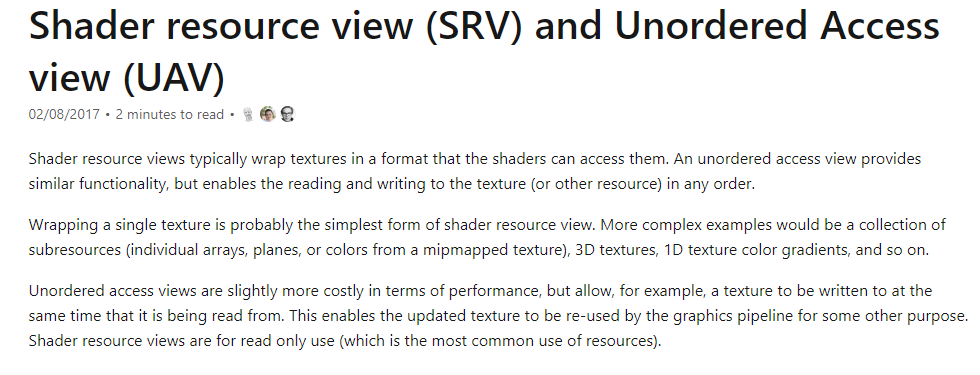
CS还有一个问题，有可能需要把大量的数据，从CPU传入到GPU。这里面，就有一个新概念，叫做UAV（Unordered Access View）。为什么要新增加这个东西？老实说，这东西本质上也就是一块显存，跟传一张图片进去有什么区别吗？还是有区别的。首先，图片的话，就太需要规格了一些，例如一般都是16 \* 16，32 \* 32这样，UAV比较随意一些。你非要抬杠说贴图也可以随意，（首先是贴图真的不能随意，不信自己mapped一下就知道，默认给你对齐。）那么还有另外一个很重要的不同，UAV可以写入自定义的数组数据！

说穿了，你可以自己定义自己的结构体数据，申请显存，传入进去。也可以申请显存，在CS阶段写入，这应该是没有问题的。



这是我定义的结构体。

另外，SRV跟UAV的不同，MS官网也有自己的解释，虽然这个解释我看着不大顺眼，不过确实不该错过。



看不懂的话，可以自己翻译。翻译了实在看不懂，搞图形学其实真的没有什么希望。

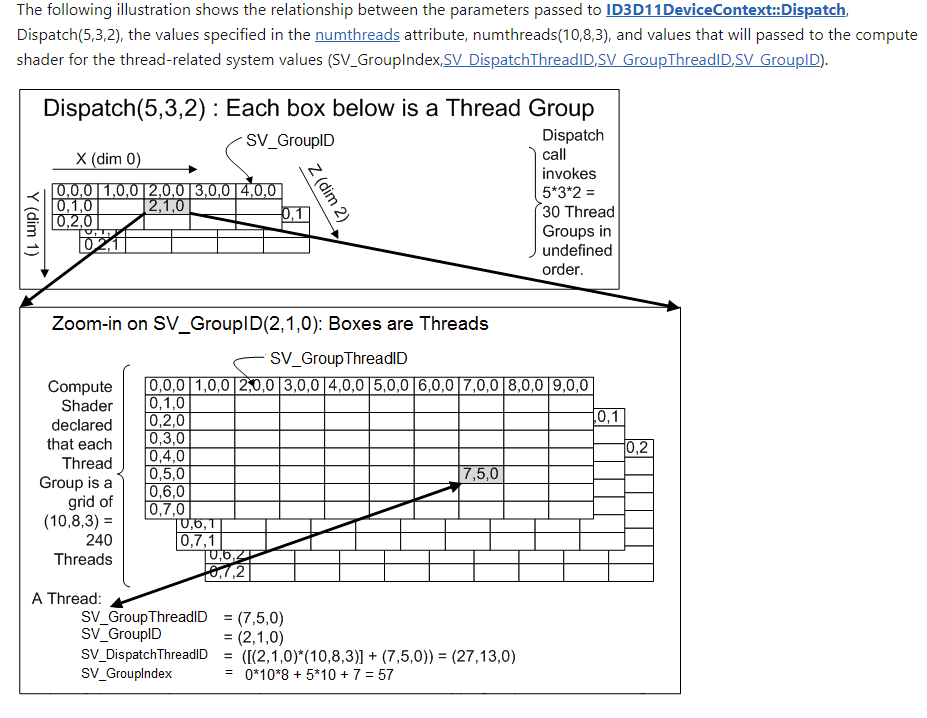
关于UAV，最后讲一点题外的。由于GPU是一个完全独立的硬件，显存的属性相当重要，写过DX12的其实深有体会。这个可以通过CPP程序这样理解，一块独立的CPU，一块独立的内存，操作的时候是很简单速度可以很快的，因为内存只有你可以存取。但是，一旦一块内存/显存，需要考虑到别人也要存取的时候，可以预见必须要做互斥。互斥无论是轮询还是回调，都是一个消耗操作。我记得我在哪个文档看过，强烈建议显存，除非非常必要，不然不要用CPU\_ACESS属性，我深以为然。所以，使用UAV的时候也要注意。

总结一下：

1. CS是用于高频的大量计算的着色器，非高频跟大量计算的，不要用。
2. CS主要还是辅助渲染的，因为非渲染的话，应该有更好的方案，例如OpenCL，CUDA……。反正我一般做B端，用CUDA比较多，而 OpenCL没用过。
3. CS的核心概念，就是线程、线程组、Dispatch，UAV。理解了这些概念，写CS还是很容易的。
4. 线程、线程组的分配尽量科学，减少浪费。

补充：写完之后，上传到github，然后被人喷，说我水了，这章写的太敷衍，看了后没有那种恍然大悟的感觉。我第一反应是：去NMD，哪里有那么多的恍然大悟。你恍然大悟那么多次都还没请我大宝剑呢，现在的读者都这么毒又这么抠门的吗？后来我本着有则改之，无则加勉的原则，回头看了一下，然后我自己也觉得有点水了。好了，基于我自己都觉得水了，现在补充再写详细一些。补充部分其实有大量应该属于其他章节的内容，例如显卡硬件相关的，CUDA相关的。没办法，CS要深刻理解，必然硬件相关，底层相关。

先讲一个我觉得比较容易看懂然后菜鸟可能容易一头雾水的东东，看图：



这图来自MS的官网，传到到处都是，我估计十个有八个写CS的都会有这张图。首先，这张图是必然确定一定以及肯定要看懂的图，看不懂，CS压根没法搞了，洗洗睡吧。

我本来想偷懒不解释这部分，让大家自己看懂，不过想来，还是有不少人不好看懂？自己还是先解释一波，铺垫一波。

先问一个问题：为什么我们需要这些东西？这些什么SV\_GroupThreadID，什么SV\_GroupID，什么SV\_DispatchThreadID，为什么我们需要这些东西？有什么用？

这就回到了所有程序的核心：算法+数据。CS是来写算法的，那么，必然需要数据。当执行到了N个线程组，执行到了线程组里的N个线程的时候，你如何知道你现在要访问的数据是哪些？这些东西，就是让你可以随心所欲的访问Shared Memory的。那么，问题来了，Shared Memory真的可以随心所欲的访问吗？当然没那么简单，这部分涉及到一个专业术语，叫做Thread Group Shared Memory (TGSM)。这里，暂时先不要讲这块，会岔开思路，后面再讲，回到主题。

所以，这些参数的提供，是便于让你为所欲为的访问共享显存的（实际上当然不能为所欲为，只是提供了可能性）。

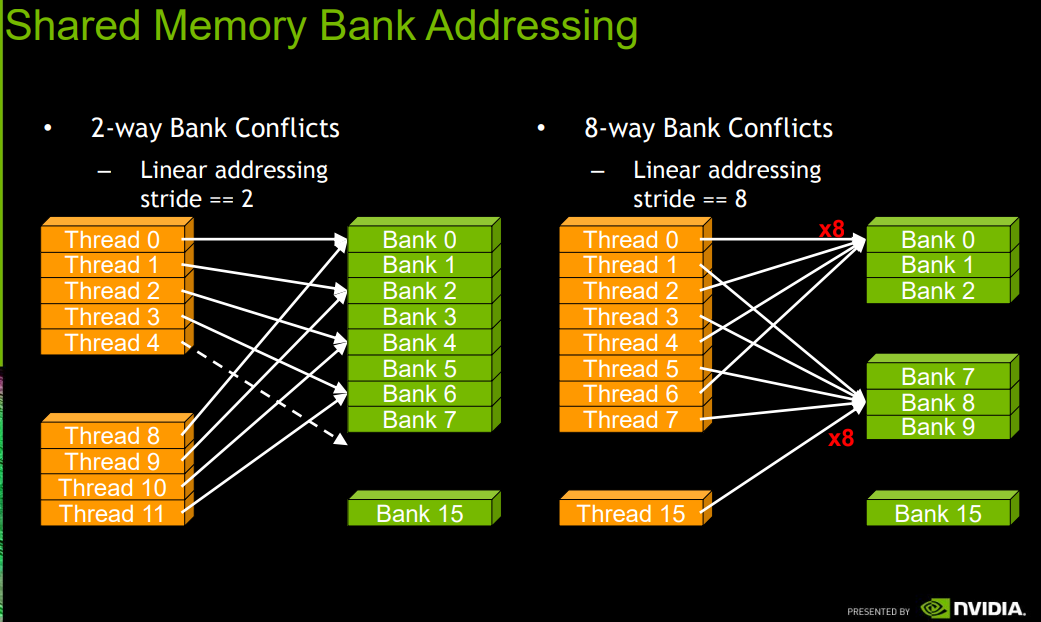
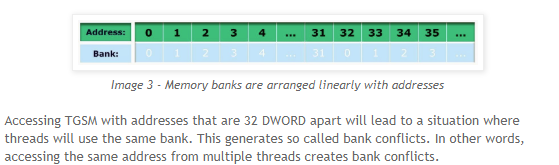
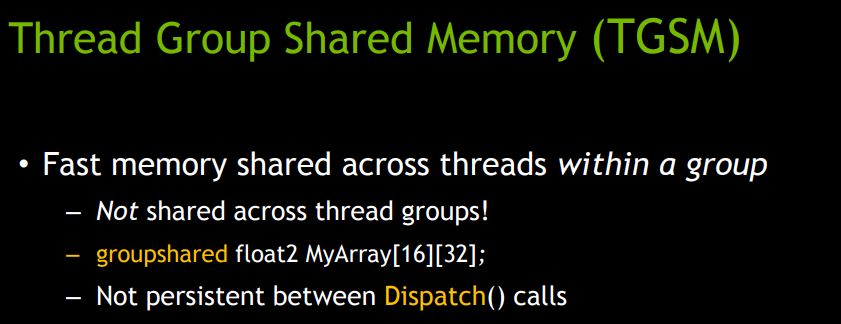
上图，是先做了这样一个假设：假设你定义的线程组是(10, 8, 3)，那么，现在正在执行的线程ID，也就是SV\_GroupThreadID，是(7, 5, 0)。是不是就很好理解了？还是理解不了？那么我放回到一维，而不是三维来解释：假设你现在一个线程组定义了240个线程（10 \* 8 \* 3 = 240），现在多个线程同时执行，你现在这个代码，执行在了Index是57的线程。这么说，好理解吗？为什么是57？上图已经有很清晰的计算了，还看不明白吗？？？那么，回去复习一下多维数组吧，不然我已经没辙了：（。

那个（5，3，2）又是什么意思呢？意思是我们现在需要执行那么多的线程组。一个线程组是240个线程，5 \* 3 \* 2 = 30个线程组，意思是这次执行30 \* 240个线程。

好了，上面的解释应该已经很清楚了，再搞不懂，这块真的只能放弃了。Duang，下面开始上真功夫了。

先来回顾一下CPU下的多线程编程。多线程，如果没有同步，没有共享内存的互斥，那么，写多线程的程序，还是很容易的。你干你的我干我的，互不相干完事了。但是，如果你仅仅是这样设计多线程的方案，不可避免的，非常受限。会给人一种：“这也干不了，那也不行，要你何用？”的即视感。所以，多线程互斥，这个必然是要做的。但是，受限于硬件，这块绝对不能像CPU那样，我给你个指针，你随便来。

此外，首先于架构，GPU也不能像CPU一样，用常用的同步手段，例如信号量，临界区，互斥体这类常见的同步方式。为了方便理解后续的讨论，这里先抛出一些硬件相关的一些结论：

1. 一般的CPU，大概有4个核心或者8个核心或者更多，但是，一个最普通的，垃圾的GPU，集成显卡，都有几百个核心。像NV，AMD主流显卡，一般都有一千个以上的核心。核心太多了，导致了GPU不存在全局同步。开销太大并且容易造成死锁。
2. 一般一个GPU同时执行的线程数，在1000-10000之间。
3. 同一个thread group的线程，必然同时执行。
4. 不同一个thread group的线程，可能同时执行，可能不同时执行。
5. 线程之间同时Access一块数据，通过一个叫做bank layout的东西，如果多个线程同时访问这个同一个地址，就产生了bank冲突，以下是英文的原文解释：因此，可以得出结论：任何时候，尽量减少Access。如果一块shared memory非要多次访问，甚至可以用一个临时变量保存/复制出来，而不是直接访问这个显存。
6. AMD也好，NV也罢，设计上有一个东西，我不知道怎么翻译，暂时翻译成“线程簇”？英文里，NV叫做“Wrap”，而AMD叫做“WaveFront”。这个东西，我的理解是：最小的线程执行单元。现在一般的GPU，NV这个数据是32，而AMD是64。意思是，这个单元里的所有线程，执行的时候，是绝对同步的，不存在不同步的情况，不需要互相等待，执行GroupMemoryBarrierWithGroupSync这个操作。
7. 同个thread group内，共享内存可以在thread之间共同访问。不同的thread group的线程，不能越界访问。这个说法，来自于NV的文档，有图为证：

如何理解这一块？我们以CPU为例。假设CPU写一个程序，调用一个函数，是怎么搞的？

首先，我们需要一个栈内存，没有栈内存，函数内定义一个临时变量，需要用到的内存从何而来？同理，GPU一样的，你函数体内，必然是需要cache的，因此，任何一个线程，都必须需要临时变量，临时内存。在CPU里，默认每开一个线程，栈内存是1M。GPU由于线程太多，shared memory没那么大，我看到的数据，是32k还是64k？还是不同的GPU不同？不同厂商的GPU这块也不同？我记得CUDA Handbook里提到过SM3.0的话，是32k？CUDA Handbook应该是权威靠谱的？这块需要查确切的文档了。

由于受到各种苛刻条件的限制，GPU写程序自由度远远没有CPU的高，如何写好一个CS程序，也是非常考究的。在NV的各大资料里，有一篇流传甚广的，是提到如何优化CS程序的。这篇文章对于新人比较不友好，如果不是对显卡，对GPU有比较深入的了解，看懂这篇文章我认为还是比较有难度的，链接我发一下：<http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>

这里我大概讲解一下，讲解的是我自己的理解，我认为我理解到位了的，但是不能绝对保证。

先来定一个需求，现在要实现一个算法，计算一个超大数组的求和。这类算法，在CPU里简直不要太容易，估计刚学c++几天的小朋友都会写，大概这样：

Int ArraySum()

{

int Sum = 0;

For(int i = 0; i < ArrayCount; i++)

{

Sum += Array[i];

}

Return Sum;

}

这里，先不要考虑数组超大可能导致的溢出，只考虑单纯的算法。这么一个算法，使用GPU做优化到时候，却能写成一篇超长的文章，方方面面的介绍如何写才是最合适的。

在开始介绍GPU优化之前，我先考虑一下这个算法用CPU多线程优化，可以怎么做。假设，我们分为四个线程来算，每个线程计算1/4的数组长度求和，得到四个结果，再求和，这不就可以了吗？说干就干，我们看看这个程序在CPU里怎么实现的，为了方便阅读，这里假设这个数组总长度是4000。那么，最简单最垃圾的写法，我写四个线程函数，一个线程一个，大概这样：

Thread 1:

Void Thread1()

{

For(int i = 0; i < 1000; i++)

{

Result[0] += Array[i];

}

}

Thread 2:

Void Thread1()

{

For(int i = 1000; i < 2000; i++)

{

Result[1] += Array[i];

}

}

另外两个线程也可以大概这么写。

当然了，这是最垃圾的写法，我们一般人，可以写一个更nice一些的写法，一个线程函数即可，通过传入参数来实现，大概如下：

Thread:

Void Thread(int ResultIndex, int ArrayMin, int ArrayMax)

{

For(int i = ArrayMin; i < ArrayMax; i++)

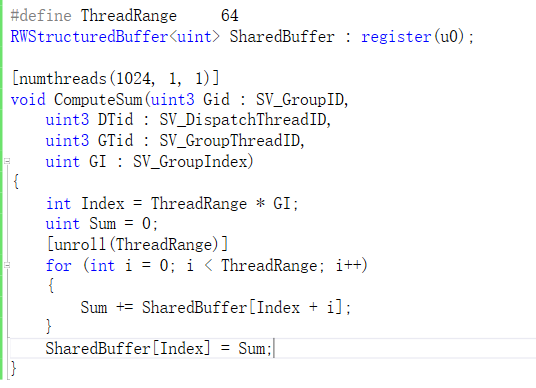
{

Result[ResultIndex] += Array[i];

}

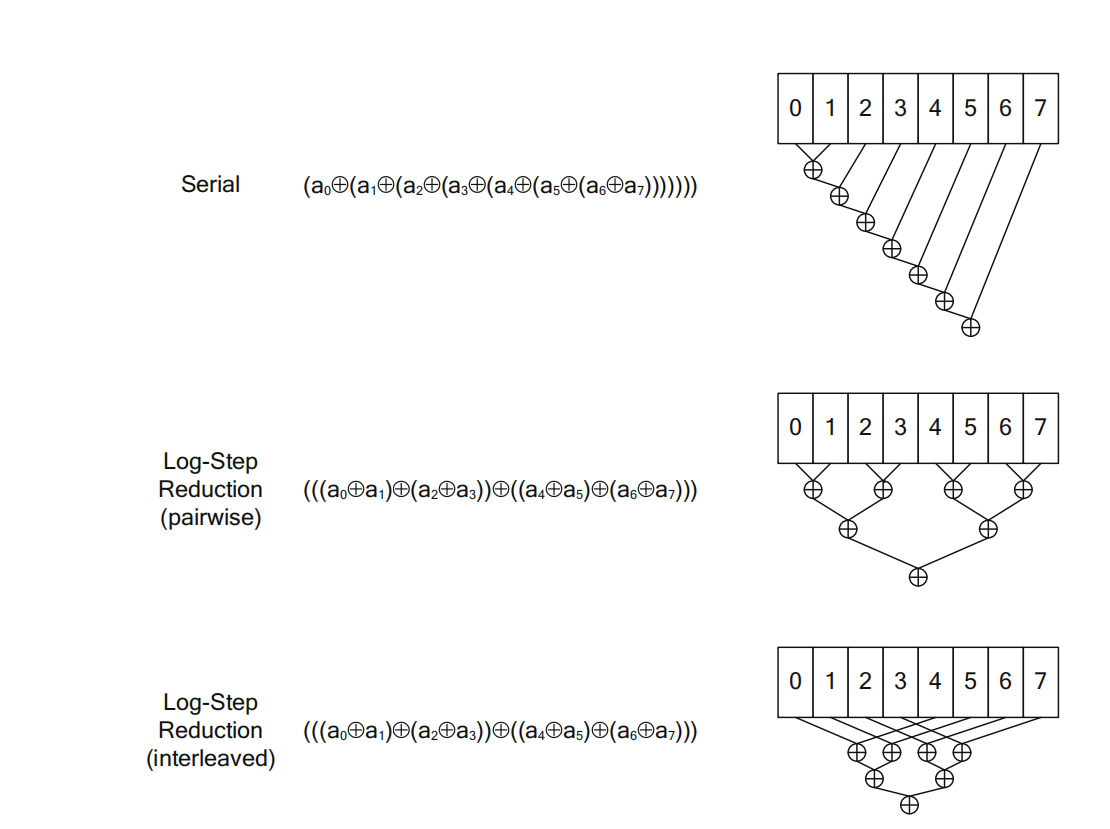
}

以上两种写法，第一种在GPU里面是不可能的，线程太多了。那么，只能考虑第二种。我假定现在同时执行1024个线程，然后，每个线程计算64个整数的相加，这样，我不就可以实现一个牛逼的GPU多线程计算了吗？说干就干，代码如下：



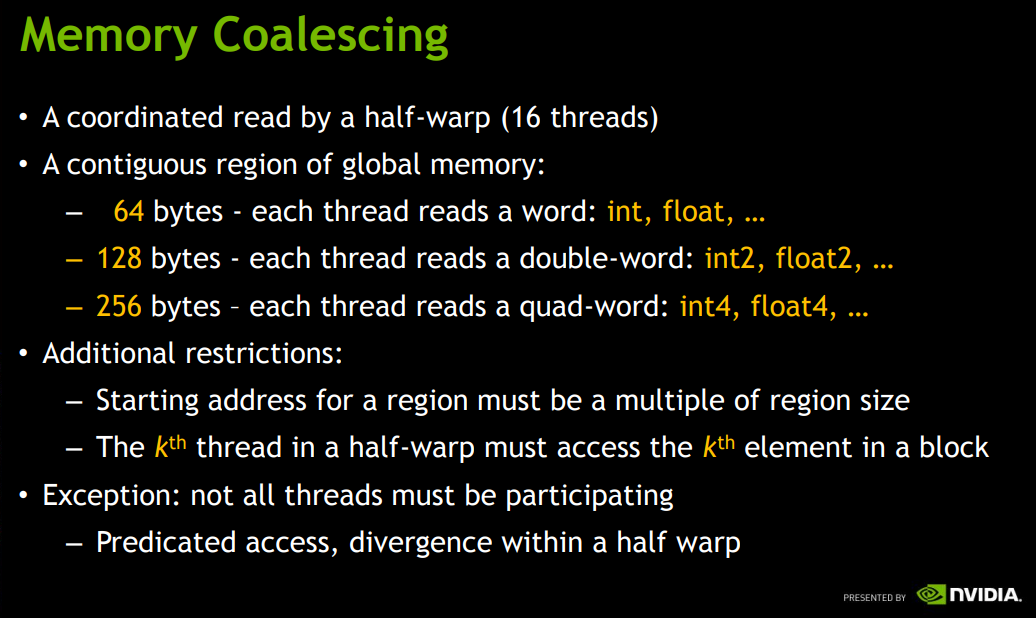
这代码，执行得666，结果正确，代码简洁，可谓潇洒之极，我不禁以为我已经懂得了GPU的真谛，成了图形学之神。

现实却是被各种大佬打脸了。大佬们告诉我，这种写法就是渣！为什么说是渣？我痛定思痛，猛啃一堆资料跟书，应该还是有点收获的。我这里就讲讲我的一些理解，不保证绝对正确。

先来看看这张图：

这图来自于CUDA Handbook，这是三种计算方式，把8个数据做一个相加。如果是CPU程序，毫无疑问第一种是最优雅的，次之是第二种，第三种写起来就别扭。但是GPU上，第三种恰恰是速度最快的，原因何在？

这里，有一个概念，叫做“Memory Coalescing”，由于我看书少，不知道这个中文是怎么翻译的，感觉怎么翻译都不对。我这里，大概解释一下，这个可以认为是内存块处理。可以先从SIMD说起，SIMD其实就是单指令多数据，GPU都是SIMD指令，这个没什么疑问。也就是说，一次处理多个数据。但是，这里又跟传统的SIMD不同，GPU里面，以NV为例，是以Wrap作为单位执行的，这个在上面有提到过，AMD叫做WaveFont。NV里面，单次读取内存，是以一半的Wrap为起点，也就是16个线程一起，并且对齐。可以分别Access 64、128、256Byte的数据，有图为证：

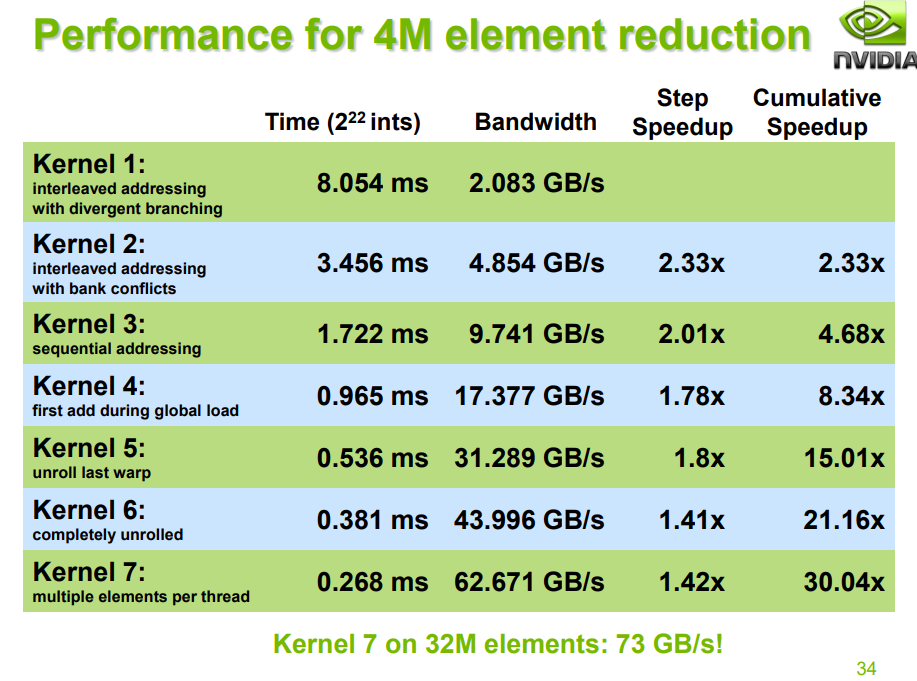
上图来自NV文档。以上的资料，会根据不同的GPU架构、版本之类的，会有所不同，这个数值只能作为参考，但是原理不变。也就是说，同样是读取显存，对齐的，大块的读取，会比分散的，无序的存取，效率低很多。

基于这个原因，导致了上面我写的CS程序显存Access效率低下，这是这个程序渣渣的第一个原因。

第二个原因，还是显存的Access。显存跟内存一样，有各种划分，如果我没有搞错，速度最快的，除了register，就是shared memory了。我的CS程序，全程用的是Global Memory，这个的效率，是比较低的。因此，写CS程序，能用Shared memory就用shared memory，但是，要尽可能的减少access，并且尽量不要产生访问冲突(bank conflicts)。这在上面有说过。

第三个原因，涉及到显卡的负载。习惯性写CPU的程序，我们习惯了大段大段的代码，这跟GPU程序迥然不同。例如我那个渣渣程序，开了1024个线程，dispath一下，就以为很多很厉害了。其实，这是对GPU程序估计严重不足，习惯性用CPU的眼光来看待。举个例子：在CPU里，线程是一个很重要的资源。如果你开了十几个线程一个程序，然后一半的线程不干活，这种写法，估计会被喷死。但是，在GPU里确实可行的。在GPU程序的一个特点是，尽可能的用最多的线程，不干活也没关系，反正闲着也是闲着。

上面发的一个链接，介绍了7种优化方案，一个一个对比执行效率以及显存带宽的开销（band width）。看图：



有没有空间换时间的既视感？关于这个bandwidth，我之前在“认识显卡”章节，有详细讲过。现在比较好的显卡，bandwidth大概300G。看似很大，其实没有那么大的。上面一个优化，就用去了62G，实在是太过恐怖。要知道，这才是执行了一次，要是跟渲染一样，一秒钟执行60帧，那根本不够用，所以，这个bandwidth的消耗，比你想象的大。

所以，尽可能的优化显存access，非常重要。而大块的显存access，能有效理用bandwidth，这个切记。这个优化链接我再发一下，想看的自己看，我在无数的地方看到类似的资料的，NV官网就看过两个，CUDA Handbook里面也有，也在一些AAA游戏的优化文档里面看过，可以说烂大街了。

总体来说，这么多文档，很多都加入了自己的理解，我个人觉得CUDA Handbook里的写的最好，但是，代码也最难懂，毕竟是cuda的代码，而且用了模板。我个人是一个一个都看了，然后互相参照，好不容易才感觉自己弄懂了，太不容易了。链接在这里：<http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>

这块，不打算再讲了，再讲，我觉得也很难讲得比网上各种专业的大佬更好了，一个个都是大佬，细节也挺多的，图文并茂，并且我也抓住了很多细节讲了。

下面，讲讲CS的一些典型应用。

第一个，就是那个典型的forward+，这个我在延迟渲染里面讲过。这个，还有不少的变种，例如分块（tiled）。例如什么（deferred tiled）。还有一个重要应用，例如用CS算粒子的变换，例如风吹草地，这块我打算到了粒子系统再讲。这些就不说了，这里，重点谈一谈用CS优化Post Process效果，也就是中文翻译的后处理效果，典型的例如Bloom、HDR。

第一个问题，我们为什么需要用CS代替PS？

答案是：CS效率更高。当然了，对于这个答案，我是存疑的。我看过无数的论坛，回答，都表示CS比PS效率更高。原因是为什么呢？很简单，用PS，多了个VS，多个光栅化，这块专业术语叫做“ROP”（Render output unit），链接在这里：<https://en.wikipedia.org/wiki/Render_output_unit>。这个过程其实有那么一点点复杂，例如你需要设置顶点，需要设置纹理，需要VS做变换，光栅化，深度缓冲，然后才是PS。大量的开销，其实最终只有一个PS是有效的。

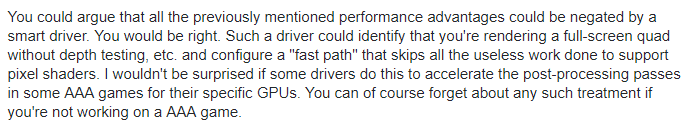
当然了，CS也不是没有缺点，缺点就是庞大的band width开销，这个上面例子里面已经非常明显了。

那么问题来了，为什么网上一边倒都认为CS比PS效率更高呢？我的理解有两个原因，一个，是真的高了，另外一个，回答这个问题的，全是做CS开发的。也就是说，本身就是利益相关。我要是做CS开发，我也会这样回答，因为我如果因为不如PS，那我做来干嘛呢。这个非常能理解。

我的看法是：从原理上看，CS确实是有一定的优势的，确实少了ROP开销，并且增加了bandwidth开销。但是，有一个疑问：为什么不直接在图形API，例如DX，增加一个接口，RenderPostProcess()这样子？

如果增加了这样一个接口，那么肯定可以底层减少了ROP的开销，并且也没有CS那么复杂，需要重新写一套，增加了bandwidth开销，这不是两全其美吗？

这里，我找到一个我认为比较靠谱的解释，看图：



这个大佬认为，确实会存在例如AAA大作，硬件厂家从驱动底层来支持这个事情，并且是可行的。但是我们大部分人没有这个待遇，所以只好老老实实写CS优化。

那么我就想问一句：为什么DX这样的图形接口，不增加这样一个支持呢？按道理NV这样的厂家，做这样的支持其实是非常轻松的事情。

我的看法是：这块的优化收益，实质没有那么大。图形学发展到了今天，全是硬骨头，各种乱七八糟的优化，例如什么early-Z之类的，什么forward+，什么deferred-tiled，什么AO，什么DO，你能想到的，想不到的，全有人做了。只要这个优化是真的巨大收益的，我认为厂家必然是会做的，因此，我个人对HDR，BLOOM之类的后处理用CS优化，能不能起到很好的效果，保留意见，毕竟渲染一个满屏QUAD的ROP开销，能有多大？好点的显卡，渲染个小场景动不动一两千帧，渲染一个QUAD的ROP，会不会只有0.1ms？当然了，我这块算不上专家，也没有自己测试过，没多大发言权，不喜可喷，让我也好学习学习。例如可以有大佬直接晒一张自己做的优化前，优化后的对比，就好比上面那些个NV大佬做的一样，各种细节，让我口服心服。

有大佬会问了，你为什么不做？是啊，其实我真的可以做的。不过快过年了，干啥都提不起精神啊，还是偷偷懒，以后有空再说吧，说不定被大佬喷了我就考虑自己实现一个测试一下了。

好了，关于CS就写到这里。长时间不用，不知道有没有错误，应该没有？有的话欢迎开喷，不介意。