# 计算着色器及应用：模糊

# 概述

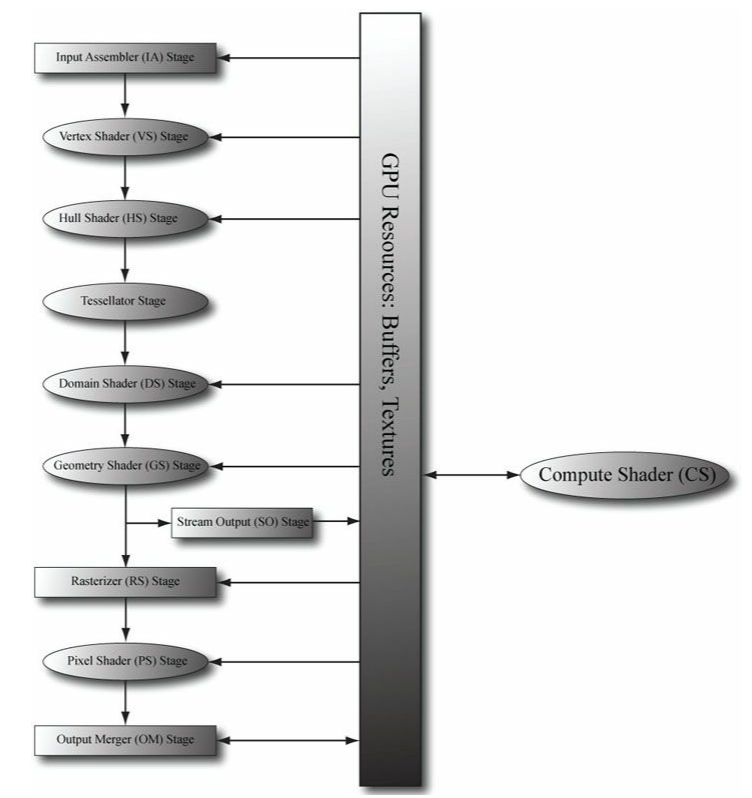
现今的GPU针对连续地址的大量内存处理(streaming operation)进行了优化，这与CPU设计为随机访问内存的方式刚好相反。而且，GPU也采用了大规模并行处理结构，这样可以对顶点和像素进行并行处理。例如NVIDIA的显卡通常有多个流式多处理器(streaming multiprocessor, 简称SM)，每个处理器又包含多个CUDA核心。

当需要大量执行相同操作时，并行处理有着巨大的优势。例如粒子系统，简化的粒子系统里每个粒子是相互无关的，可以并行的处理其物理特征。

虽然GPU架构是专为绘图而设计的，但其强大的计算功能也可用于非图形应用。在这种情况下，我们通常需要将数据传回CPU侧，这时就需要考虑传输速度的瓶颈，这个传输速度远慢于CPU访问RAM和GPU访问VRAM的速度，故应避免频繁地回传数据。

针对图形处理，我们可以将计算结果直接作为渲染管线的输入，无需再传回CPU。

计算着色器既是用于这样的可编程着色器。其独立于渲染管线之外，可以读写GPU资源。



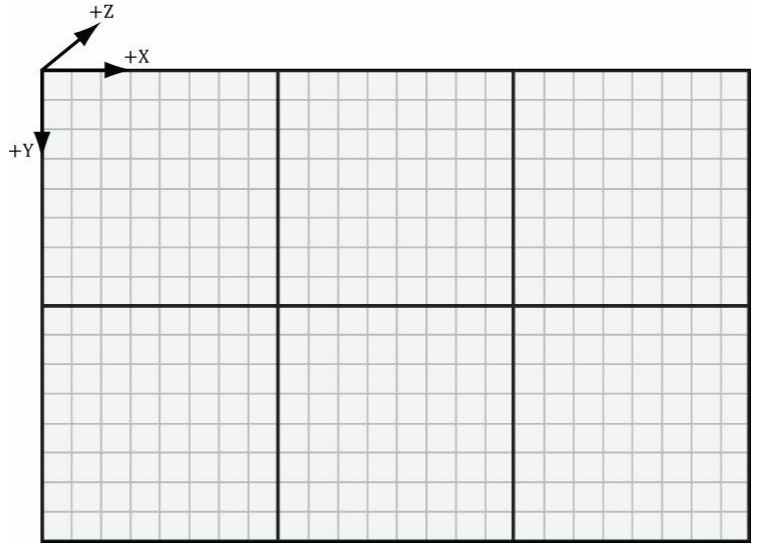
# 线程组(Thread Groups)

在实际使用中，我们可以将线程划分为由线程组(thread group)组成的网格(grid)。一个线程组运行于一个多处理器上。因此，对于拥有16个多处理器的GPU来说，我们应该分配至少16个线程组，以使每个多处理器都能充分运转。而要获得更好的性能，我们应让每个多处理器至少拥有两个线程组，让它能在不同的线程组之间切换，以连续工作，避免等待时的空闲时间。

每个线程组都有一块共享内存，可用于组内不用线程间的同步。不同线程组的共享内存是相互独立的，不能交叉访问。

每个线程组包含多个线程，硬件会将其分为多个wrap来执行，每个warp有32个线程，通过SIMD32（即32个线程同时执行单条指令）的方式来处理。故出于性能考虑，我们指定线程组时的大小应为32的整数倍。我们可以用256来作为初始的设置。

我们通过调用**ID3D11DeviceContext::Dispatch**来启动线程组。此方法有3个参数，可以开启一个3D网格。例如m\_d3dContext->Dispatch(3, 2, 1);将开启一个3x2总共6个线程组的网格：



# 计算着色器(Compute Shader)

下面是一个将两个纹理相加的计算着色器的示例。

|  |
| --- |
| cbuffer cbSettings  {  // Compute shader can access values in constant buffers.  };      // Data sources and outputs.  Texture2D gInputA;  Texture2D gInputB;  RWTexture2D<float4> gOutput;      // The number of threads in the thread group. The threads in a group can  // be arranged in a 1D, 2D, or 3D grid layout.  [numthreads(16, 16, 1)]  **void** CS(int3 dispatchThreadID : SV\_DispatchThreadID) // Thread ID  {  // Sum the xyth texels and store the result in the xyth texel of  // gOutput.  gOutput[dispatchThreadID.xy] = gInputA[dispatchThreadID.xy] + gInputB[dispatchThreadID.xy];  } |

**线程组布局**

我们通过numthreads来定义不同的线程组布局，上面就用了包含16x16个线程的2D线程组

**输入与输出资源**

在上例中，我们使用了两个输入纹理。

我们通过给两个纹理创建对应的SRV(ShaderResourceView， 着色器资源视图)，然后绑定到CS阶段，这与之前的像素着色器相同。注意，SRV都是只读资源。

我们还定义了一个输出资源。它的前缀带有RW，表示可读可写。而且还用模板来表示输出资源的类型与维度。

绑定需要写操作的资源时，我们需要声明UAV(UnorderedAccessView， 无序访问视图)，通过**D3D11\_UNORDERED\_ACCESS\_VIEW\_DESC**来描述。

|  |
| --- |
| DXGI\_FORMAT format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;  CD3D11\_TEXTURE2D\_DESC offscreenDesc(format, m\_outputWidth, m\_outputHeight);  offscreenDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE | D3D11\_BIND\_UNORDERED\_ACCESS;      HRESULT hr = m\_d3dDevice->CreateTexture2D(&offscreenDesc, nullptr, m\_offscreenTextureIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr);    hr = m\_d3dDevice->CreateShaderResourceView(m\_offscreenTextureIntermediate.Get(), nullptr, m\_offscreenSRVIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr);    hr = m\_d3dDevice->CreateUnorderedAccessView(m\_offscreenTextureIntermediate.Get(), nullptr, m\_offscreenUAVIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr); |

如果一个纹理需要与UAV绑定，那么需要使用标志**D3D11\_BIND\_UNORDERED\_ACCESS**。可以看到，我们将纹理同时绑定为SRV和UAV，但是注意这两者不能同时生效。这是一种很常见的手段，因为我们在计算着色器中将该纹理作为输出，之后可能需要将此纹理以SRV绑定到顶点或像素着色器。

与SRV相应的，我们使用**ID3D11DeviceContext::CSSetUnorderedAccessViews**来绑定一个RW资源。

**索引**

纹理元素可以使用2D索引来进行访问，在上面的例子中，我们基于分派线程的ID来进行索引。

# 结构化缓冲（structured buffers）资源

示例

|  |
| --- |
| struct Data  {  float3 v1;  float2 v2;  };      StructuredBuffer<Data> gInputA : register(t0);  StructuredBuffer<Data> gInputB : register(t1);  RWStructuredBuffer<Data> gOutput : register(u0);      [numthreads(32, 1, 1)]  **void** CS(int3 dtid : SV\_DispatchThreadID)  {  gOutput[dtid.x].v1 = gInputA[dtid.x].v1 + gInputB[dtid.x].v1;  gOutput[dtid.x].v2 = gInputA[dtid.x].v2 + gInputB[dtid.x].v2;  } |

结构化缓冲是一种由相同类型元素构成的简单缓冲区，本质上是一种数组。并且其类型可以由用户自定义。

在创建时，我们需要指定相应的flag和元素大小：

|  |
| --- |
| CD3D11\_BUFFER\_DESC outputGpuDataDesc(byteWidth, D3D11\_BIND\_UNORDERED\_ACCESS);  outputGpuDataDesc.Usage = D3D11\_USAGE\_DEFAULT;  outputGpuDataDesc.CPUAccessFlags = 0;  outputGpuDataDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  outputGpuDataDesc.StructureByteStride = sizeof(DataType);      hr = m\_d3dDevice->CreateBuffer(&outputGpuDataDesc, nullptr, m\_outputGpuBuffer.GetAddressOf());  DX::ThrowIfFailed(hr); |

结构化缓冲和纹理一样可以创建SRV或UAV然后绑定到流水线上。

将计算着色器的结果复制到系统内存

我们可以创建一块D3D11\_USAGE\_STAGING的buffer，将其指定为D3D11\_CPU\_ACCESS\_READ。之后我们可以使用**ID3D11DeviceContext::CopyResource**来将其他资源复制到该资源中。注意，该资源与待复制的资源必须要有相同的大小和类型。接着我们就可以使用map/unmap来读取该资源中的数据了。

|  |
| --- |
| CD3D11\_BUFFER\_DESC outputCpuDataDesc(byteWidth, 0);  outputCpuDataDesc.Usage = D3D11\_USAGE\_STAGING;  outputCpuDataDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_READ;  outputCpuDataDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  outputCpuDataDesc.StructureByteStride = sizeof(DataType);      hr = m\_d3dDevice->CreateBuffer(&outputCpuDataDesc, nullptr, m\_outputCpuBuffer.GetAddressOf());  DX::ThrowIfFailed(hr);    ...    m\_d3dContext->CopyResource(m\_outputCpuBuffer.Get(), m\_outputGpuBuffer.Get());      D3D11\_MAPPED\_SUBRESOURCE mappedData;  m\_d3dContext->Map(m\_outputCpuBuffer.Get(), 0, D3D11\_MAP\_READ, 0, &mappedData);  DataType\* outputData = reinterpret\_cast<DataType\*>(mappedData.pData);  ...  m\_d3dContext->Unmap(m\_outputCpuBuffer.Get(), 0); |

# 线程ID系统值

我们有几种ID可以使用:

1.系统会给每个线程组分配一个ID，称为线程组ID（group ID），用SV\_GroupID来表示。

2.在线程组内部，每个线程都有一个组内唯一的ID，称为组内线程ID（group thread ID），用SV\_GroupThreadID来表示。

3.调用dispatch时，会分配一个线程组网格，在所有线程中，每个线程都有一个唯一的标识，称为调度线程ID（dispatch thread ID），用SV\_DispatchThreadID来表示。

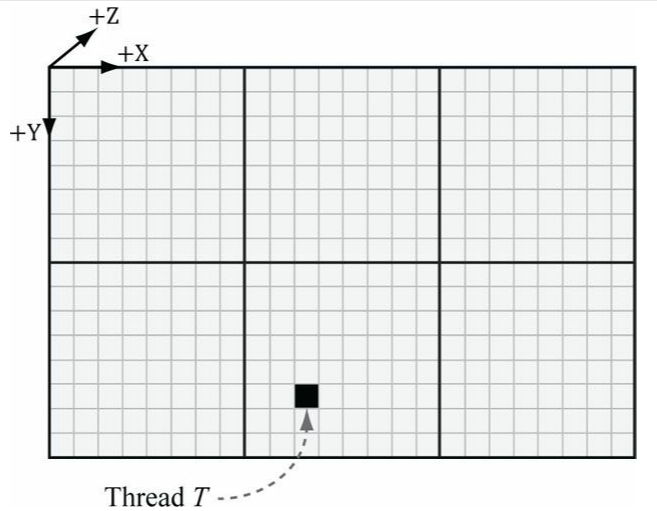
如果线程组规模是XYZ，我们就可以通过线程组ID和组内线程ID推算出调度线程ID：

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

4.组内线程ID也可以用线性索引表示，使用SV\_GroupIndex来表示，其计算方法为

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

通常计算着色器会以若干数据结构作为输入，我们可以用这些线程ID来对数据进行索引并同时处理。



图中T的group ID为(1,1,0)，group thread ID为(2,5,0)，dispatch thread ID为(1,1,0) x (8,8,0) + (2,5,0) = (10,13,0)，group index ID为5 x 8 + 2 = 42。

# 追加缓冲区与消费缓冲区（append and consume buffers）

假设我们需要一个存储粒子的缓冲区，我们希望通过计算着色器更新其数据，并且不需要考虑粒子更新的顺序和他们输出的顺序。此时我们就可以使用消费缓冲区和追加缓冲区了。

|  |
| --- |
| struct Particle  {  float3 Position;  float3 Velocity;  float3 Acceleration;  };      **float** TimeStep = 1.0f / 60.0f;  ConsumeStructuredBuffer<Particle> gInput;  AppendStructuredBuffer<Particle> gOutput;      [numthreads(16, 16, 1)]  **void** CS()  {  // Consume a data element from the input buffer.  Particle p = gInput.Consume();    p.Velocity += p.Acceleration\*TimeStep;  p.Position += p.Velocity\*TimeStep;    // Append normalized vector to output buffer.  gOutput.Append(p);  } |

数据元素一旦经过处理（消费），则其他线程就不能再对其进行操作了。除此之外，我们无法知晓元素消费和追加的顺序，即某元素输入输出对应的索引很有可能是不一样的。

这有点类似stack数据结构，但需要注意，追加结构化缓冲区不是动态增长的，必须要有足够的空间来存放所有要追加的元素。

我们在创建UAV时需要指定**D3D11\_BUFFER\_UAV\_FLAG\_APPEND** flag来声明追加缓冲区。

# 共享内存与线程同步

每个线程组都有一块共享内存，或者叫线程本地存储(thread local storage)。这种内存的访问速度很快，可认为与硬件高速换成速度不相上下。

我们通过如下语句声明共享内存:

groupshared float4 gCache[256];

数组大小可以根据需要自定义，但是线程组共享内存总大小上限为32kb。

处于性能考虑，共享内存不应太大。假如一个计算着色器需要20kb的共享内存，那么每个多处理器将只能处理一个线程组，而不是在两个里切换，因为这需要40kb，超出了最大限制。这样多处理器就无法屏蔽处理过程中的延迟。

共享内存的常见应用场景是存储纹理数据。例如在模糊算法中，我们需要对同一个像素进行多次采样。而采样相比于计算其实是比较慢的GPU操作。如果我们将该线程组需要的纹理样本全部预加载到共享内存中，之后直接读取共享内存，则避免了反复进行采样。

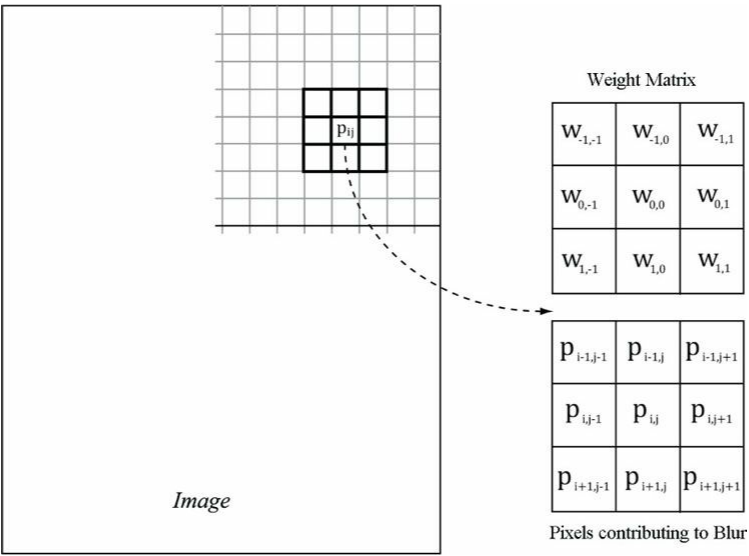
这里还需要注意，由于需要对共享内存进行读写，此时顺序至关重要。如果我们不加以限制，我们在读的时候，其他线程可能还没有完成写操作。

此时就需要使用**GroupMemoryBarrierWithGroupSync()**来同步。确保所有线程都完成了采样并存放到了共享内存里，而后再进行后面的计算。

# 图像模糊示例

# 模糊原理

模糊算法可以这样描述：针对源图像中的每一个像素，取其以之为中心的mxn个像素，计算加权平均值，得到的即是模糊后的结果。



我们采用高斯模糊(Gaussian blur)，即使用高斯函数来计算权值。

高斯模糊以其可分离性而广为人知。根据这条性质，我们可以将其分离为两个1D方向上的模糊过程，例如先横向模糊，再纵向模糊。

假设我们一个9x9的矩阵，原本需要81个样本依次执行2D模糊，但分离为两次1D模糊后，我们只需要18个样本。这样大大减少了采样的数量。

即使有些模糊算法不具有分离性，但只要保证最终的视觉效果足够精准，我们也可以分离模糊来优化性能。

# 渲染到纹理

之前的实例程序在渲染时都是以后台缓冲区作为渲染目标，它其实就是交换链中的一个纹理。其实我们也可以自己创建一个纹理，将其作为渲染目标，绑定到输出合并阶段。这样数据就绘制到了离屏（off-screen）纹理中。这样的操作称为渲染到离屏纹理（render-to-off-screen-texture），简称渲染到纹理（render-to-texture）。

渲染到纹理的常用场景有：

1.阴影贴图(Shadow mapping)

2.屏幕空间环境光遮蔽(Screen Space Ambient Occlusion)

3.动态反射与立方体贴图(Dynamic reflections with cube maps)

我们使用渲染到纹理来实现模糊：先将实际的场景渲染到离屏纹理中，再将该纹理输入至计算着色器执行模糊算法，再将得到的纹理渲染到后台缓冲区。

# 图像模糊的实现概述

我们将模糊分为两个1D模糊，因此需要两个可读写的纹理缓冲区，A和B，并且我们要为其分别创建SRV和UAV。具体过程为

1.给A绑定SRV，作为计算着色器的输入。

2.给B绑定UAV，作为计算着色器的输出。

3.分派线程组执行横向模糊操作。此时，结果存储在B里。

4.给B绑定SRV，作为计算着色器的输入。

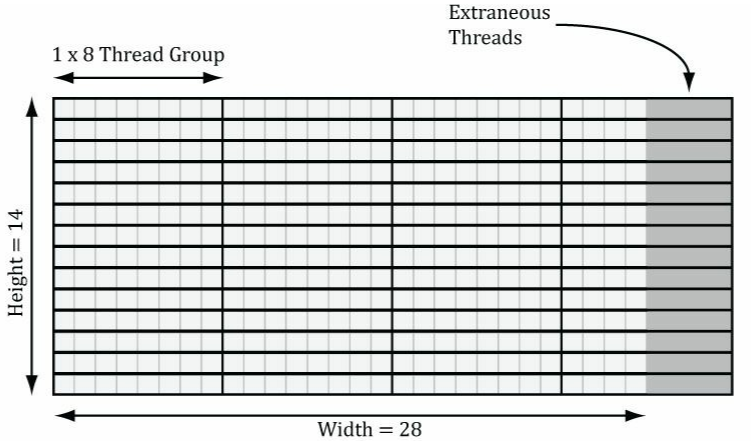
5.给A绑定UAV，作为计算着色器的输出。

6.分派线程组执行纵向模糊操作。此时，我们得到了最终的结果，并保存在A里。

当分派线程时，我们需要计算具体的数量。

设待模糊的图像宽为w，高为h，一个线程组用256个线程，而每个线程负责图像中的一个像素。

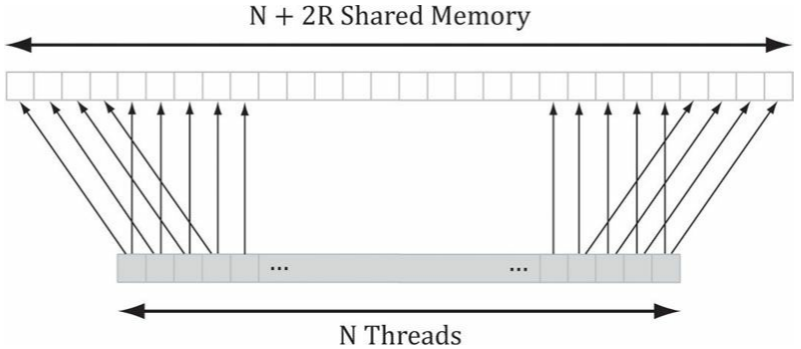
因此，我们需要在x方向上派发(w/256)个线程组,向上取整。如果w不能被256整除，则最后一次分派的线程组会存有多余的线程，对于这种情况，我们无能为力。y方向上的情况也与此类似。



|  |
| --- |
| // We must unbind the resource to be set as shader resource in CS  m\_d3dContext->OMSetRenderTargets(1, m\_renderTargetView.GetAddressOf(), m\_depthStencilView.Get());    // Create an intermediate texture to store the first blur result  ComPtr<ID3D11Texture2D> m\_offscreenTextureIntermediate;  ComPtr<ID3D11ShaderResourceView> m\_offscreenSRVIntermediate;  ComPtr<ID3D11UnorderedAccessView> m\_offscreenUAVIntermediate;    DXGI\_FORMAT format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;  CD3D11\_TEXTURE2D\_DESC offscreenDesc(format, m\_outputWidth, m\_outputHeight);  offscreenDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE | D3D11\_BIND\_UNORDERED\_ACCESS;    HRESULT hr = m\_d3dDevice->CreateTexture2D(&offscreenDesc, nullptr, m\_offscreenTextureIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr);  hr = m\_d3dDevice->CreateShaderResourceView(m\_offscreenTextureIntermediate.Get(), nullptr, m\_offscreenSRVIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr);  hr = m\_d3dDevice->CreateUnorderedAccessView(m\_offscreenTextureIntermediate.Get(), nullptr, m\_offscreenUAVIntermediate.GetAddressOf());  DX::ThrowIfFailed(hr);    ID3D11ShaderResourceView\* nullSRV[1] = { nullptr };  ID3D11UnorderedAccessView\* nullUAV[1] = { nullptr };    m\_d3dContext->CSSetConstantBuffers(0, 1, m\_cbBlurSettings.GetAddressOf());    **for** (**int** i = 0; i < m\_blurCount; ++i)  {  // Horizontal blur  m\_d3dContext->CSSetShader(m\_horzBlurCS.Get(), nullptr, 0);  m\_d3dContext->CSSetShaderResources(0, 1, m\_offscreenSRV.GetAddressOf());  m\_d3dContext->CSSetUnorderedAccessViews(0, 1, m\_offscreenUAVIntermediate.GetAddressOf(), nullptr);  **int** numThreadGroupX = ceil(**float**(m\_outputWidth) / 256);  m\_d3dContext->Dispatch(numThreadGroupX, m\_outputHeight, 1);    m\_d3dContext->CSSetShaderResources(0, 1, nullSRV);  m\_d3dContext->CSSetUnorderedAccessViews(0, 1, nullUAV, nullptr);    // Vertical blur  m\_d3dContext->CSSetShader(m\_vertBlurCS.Get(), nullptr, 0);  m\_d3dContext->CSSetShaderResources(0, 1, m\_offscreenSRVIntermediate.GetAddressOf());  m\_d3dContext->CSSetUnorderedAccessViews(0, 1, m\_offscreenUAV.GetAddressOf(), nullptr);  **int** numThreadGroupY = ceil(**float**(m\_outputHeight) / 256);  m\_d3dContext->Dispatch(m\_outputWidth, numThreadGroupY, 1);    m\_d3dContext->CSSetShaderResources(0, 1, nullSRV);  m\_d3dContext->CSSetUnorderedAccessViews(0, 1, nullUAV, nullptr);  }    // Disable compute shader  m\_d3dContext->CSSetShader(nullptr, nullptr, 0);    // Copy result to back buffer  ComPtr<ID3D11Resource> backBuffer;  m\_swapChain->GetBuffer(0, IID\_PPV\_ARGS(&backBuffer));  m\_d3dContext->CopyResource(backBuffer.Get(), m\_offscreenTexture.Get()); |

# 模糊的计算着色器

当我们执行模糊运算的时候，n个线程实际需要n+2R个数据（R为模糊半径）。我们只需分配n+2R个元素的共享内存，并且采样的时候让2R个线程多采样一次以获得额外的元素。



另一个问题则是越界的情况，最左侧和最右侧的线程在采样时会发送越界，获得的结果为0，即黑色。这会影响到边界的模糊结果。我们希望使用类似clamp寻址模式的效果，在越界时获得边界的数据。

|  |
| --- |
| //  // Fill local thread storage to reduce bandwidth. To blur  // N pixels, we will need to load N + 2\*BlurRadius pixels  // due to the blur radius.  //    // This thread group runs N threads. To get the extra 2\*BlurRadius pixels,  // have 2\*BlurRadius threads sample an extra pixel.  **if**(groupThreadID.x < gBlurRadius)  {  // Clamp out of bound samples that occur at image borders.  **int** x = max(dispatchThreadID.x - gBlurRadius, 0);  gCache[groupThreadID.x] = gInput[int2(x, dispatchThreadID.y)];  }  **if**(groupThreadID.x >= N-gBlurRadius)  {  // Clamp out of bound samples that occur at image borders.  **int** x = min(dispatchThreadID.x + gBlurRadius, gInput.Length.x-1);  gCache[groupThreadID.x+2\*gBlurRadius] = gInput[int2(x, dispatchThreadID.y)];  }    // Clamp out of bound samples that occur at image borders.  gCache[groupThreadID.x+gBlurRadius] = gInput[min(dispatchThreadID.xy, gInput.Length.xy-1)]; |