Week8

Compute Shader

Hooman Salamat

Objectives

1. To learn how to program compute shaders.

2. To obtain a basic high-level understanding of how the hardware processes thread groups, and the threads within them.

3. To discover which Direct3D resources can be set as an input to a compute shader and which Direct3D resources can be set as an output to a compute shader.

4. To understand the various thread IDs and their uses.

5. To learn about shared memory and how it can be used for performance optimizations.

6. To find out where to obtain more detailed information about GPGPU programming.

GPU vs. CPU

CPU designed for random memory accesses.

GPUs have been optimized to process a large amount of memory from a single location or sequential locations (*streaming operation):* a large amount of data elements that will have similar operations performed on them - Graphical operations like shading pixels, particle systems, waves.

GPUs have been architected to be massively parallel; (vertices and pixels can be independently processed)

Using the GPU for non-graphical applications is called *general purpose GPU* (GPGPU) *programming*.

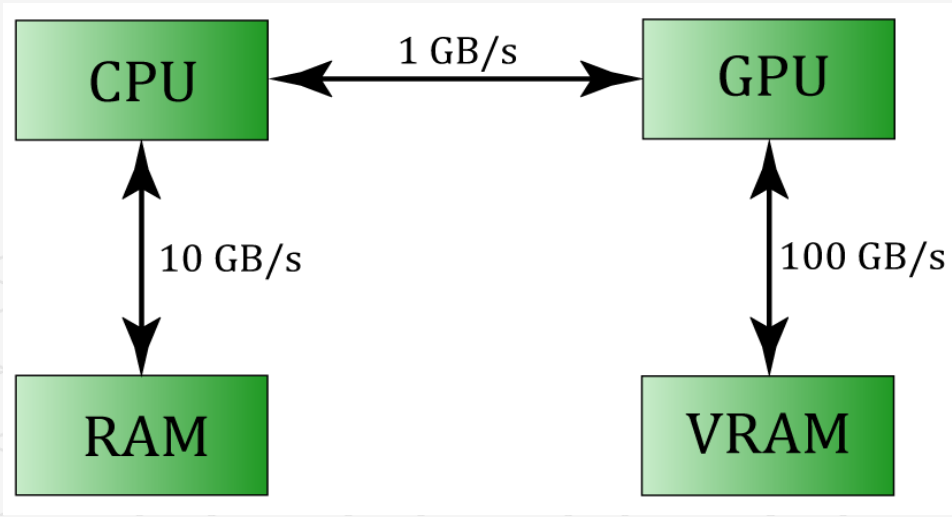
NVIDIA "Fermi" architecture supports up to sixteen streaming multiprocessors of thirty-two CUDA(Compute Unified Device Architecture) cores for a total of 16x32=512 CUDA cores. https://en.wikipedia.org/wiki/Category:Nvidia_microarchitectures.

Latest NVDIA "Turing" microarchitecture (GeForce 20 series: RTX2080) has 4608 CUDA cores, 72 RT(Ray Tracing) Cores, and 576 Tensor Cores! https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

https://docs.nvidia.com/cuda/turing-tuning-guide/index.html

The following figure shows the relative memory bandwidth speeds between CPU and RAM, CPU and GPU, and GPU and VRAM.

These numbers are just illustrative numbers to show the order of magnitude difference between the bandwidths. Observe that transferring memory between CPU and GPU is the bottleneck.



Compute Shader

The Compute Shader is a programmable shader Direct3D exposes that is not directly part of the rendering pipeline. Instead, it sits off to the side and can read from GPU resources and write to GPU resources.

The Compute Shader allows us to access the GPU to implement data-parallel algorithms without drawing anything.

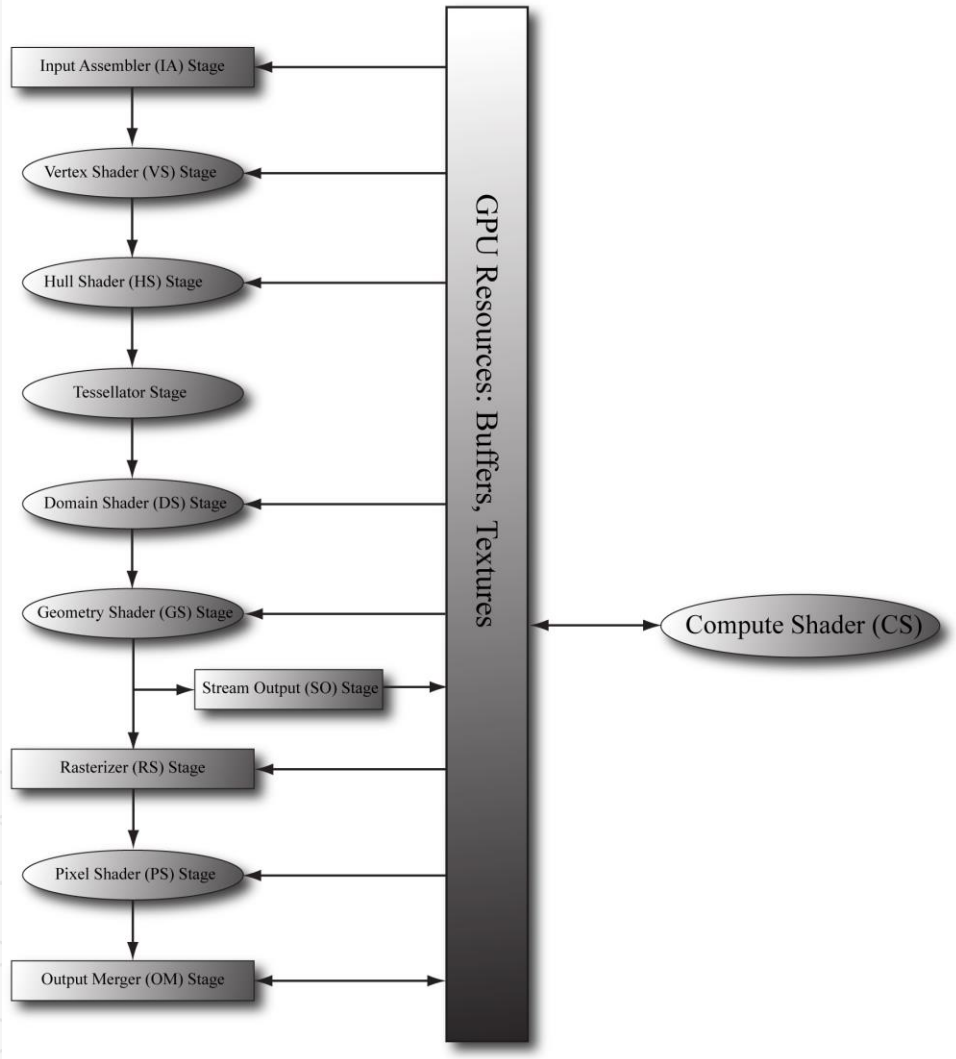
The compute shader can be mixed with graphics rendering, or used alone for GPGPU programming.

Because the Compute Shader is part of Direct3D, it reads from and writes to Direct3D resources, which enables us to bind the output of a compute shader directly to the rendering pipeline.

As an example, if you look at the code for our wave, you will see that in the update step, we perform a calculation on each grid element.

So this, too, is a good candidate for a GPU implementation, as each grid element can be updated in parallel by the GPU.

Particle systems provide yet another example, where the physics of each particle can be computed independently provided we take the simplification that the particles do not interact with each other.



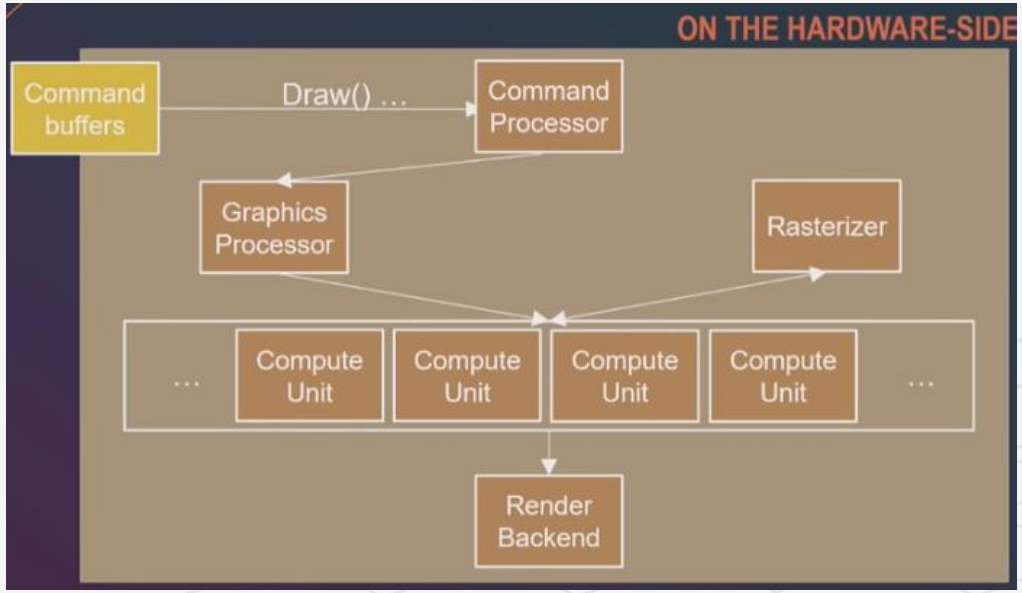
Graphics Pipeline vs. Compute Pipeline on the hardware-side

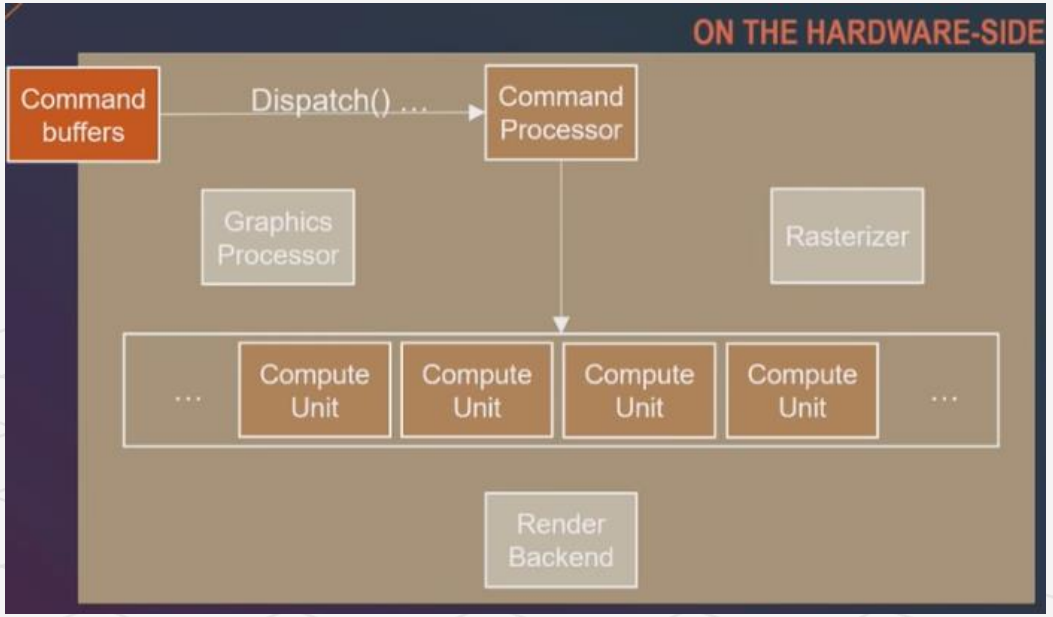
When there is a draw command, we need a graphics processor, therefore the command processor sends the commands to graphics processor, which passes to each compute unit. The compute unit knows how to work with shaders.

When there is a dispatch command, we need a compute shader. So it gets directly routed to compute unit. The compute unit also knows how to work with compute shader.

Input: constants and resources

Output: writable resources (UAVs)





Graphics Pipeline vs. Compute Pipeline on the code-side

Creating the compute pipeline requires less information than creating graphics pipeline.

Compute shaders introduce less hardware overhead compared to the other shader stages.

Graphics pipeline	Compute pipeline
One to several shader stages (VS, HS, DS, GS, PS) Input assembler Tessellation Rasterizer …	CS shader stage

GPU

A GPU is a hardware device which contain multiple small hardware units called *SM*s (Streaming Multiprocessors).

Each *SM* can execute many threads concurrently. But these threads are not exactly the same as the threads run by a CPU.

These GPU threads are grouped physically.

A physical thread group is called a "*warp*" *or "wavefront*", which contains 32 or 64 threads.

Imagine a SM can execute 2048 threads concurrently. So actually this SM can execute 2048/32 = 64 warps concurrently — That's how *warps* fit into the picture.

Different CPU threads can work on different instructions (addition, multiplication) concurrently. But all 32 threads in a warp can execute only same instruction concurrently.

For an example if a warp is performing the operation add, then all 32 threads should perform addition. But each thread can add different numbers in the data set. That is one thread will do 1 + 2 while another thread will do 1 + 3 at the same time (concurrently).

Because of this nature of GPUs, they are said to follow the concept of "*Single Instruction, Multiple Data*" (*SIMD*).

Execution model of shaders

GPUs are designed to process a massive amount of data in parallel.

Need to run the same program (shader) on many vertices and pixels.

The meaning of a compute unit depends on who manufactured a particular GPU but as we know the top two GPU vendors are NVIDIA and AMD: A compute unit is a **stream multiprocessor** in a *NVidia GPU* or a **SIMD engine** in an *AMD GPU*. Each compute unit has several processing elements (ALU/stream processor).

This figure shows an AMD GCN (Graphics Core Next) architecture. For example, for Vega, we have 64 compute units.

The smallest work item for vertex shaders is a vertex.

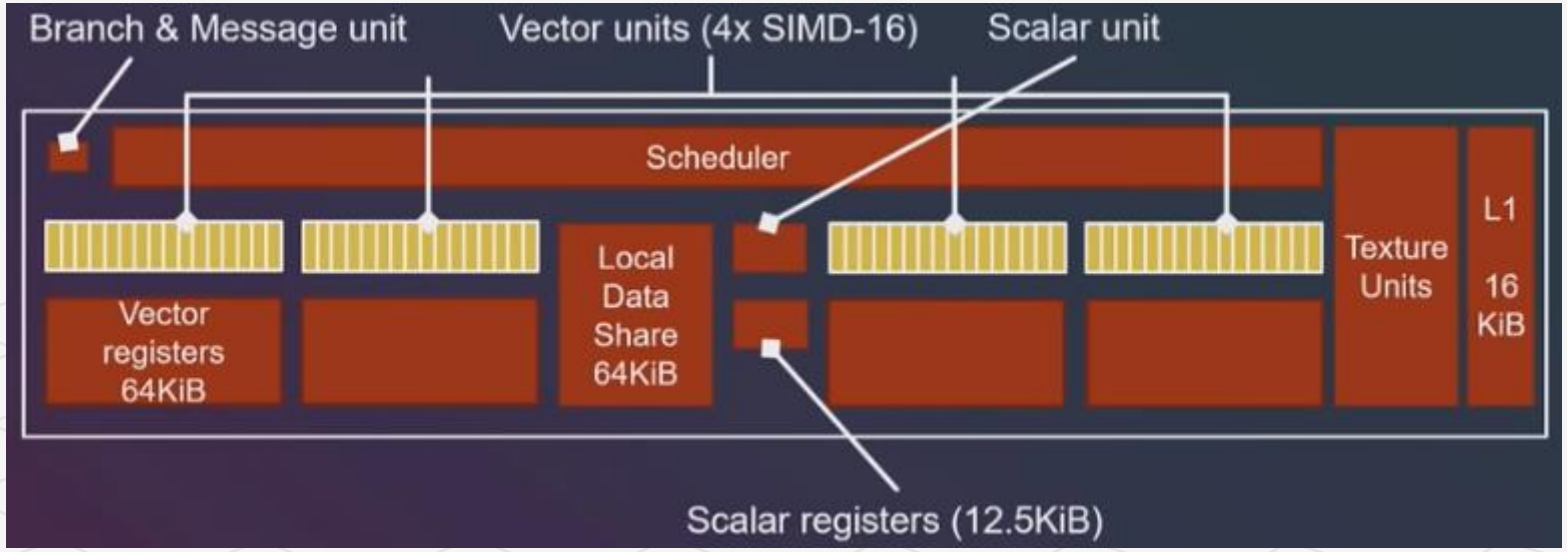
The smallest work item for pixel shaders is a pixel.

The smallest work item for compute shaders is a thread.

In hardware, the parallelism is realized using:

A) SIMD unit, which are grouped on a compute unit (SIMD – Single Instruction Multiple Data)

B) Many compute units



THREADS AND THREAD GROUPS

In GPU programming, the number of threads desired for execution is divided up into a grid of *thread groups*.

A thread group is executed on a single multiprocessor.

A **multiprocessor** is a computer system having two or more processing units (multiple processors) that can execute multiple processes simultaneously where the processors may share "some or all of the system's memory and I/O facilities"

If you had a GPU with sixteen multiprocessors, you would want to break up your problem into at least sixteen thread groups so that each multiprocessor has work to do.

For better performance, you would want at least two thread groups per multiprocessor to handle stalls.

A stall can occur, for example, if a shader needs to wait for a texture operation result before it can continue to the next instruction.

Each thread group gets shared memory that all threads in that group can access.

A thread cannot access shared memory in a different thread group. Thread synchronization operations can "only" take place amongst the threads in the same thread group.

NVDIA CUDA model divides the threads in a thread group into sub-blocks called "warps"

NVIDIA hardware uses warp sizes of thirty-two or sixty-four threads (depending on the architecture: Fermi vs. Turing)

ATI (now AMD) divides the threads in a thread group into sub-blocks called "wavefronts"

AMD uses "wavefront" sizes of sixty-four threads

A warp is processed by the multiprocessor in SIMD32 (i.e., the same instructions are executed for the thirty-two threads simultaneously).

Each CUDA core processes a thread and a "Turing" multiprocessor has sixty-four CUDA cores (so a CUDA core is like a SIMD "lane.")

Thread group sizes of 512 seems to be a good starting point that should work well for various hardware. Changing the number of threads per group will change the number of groups dispatched.

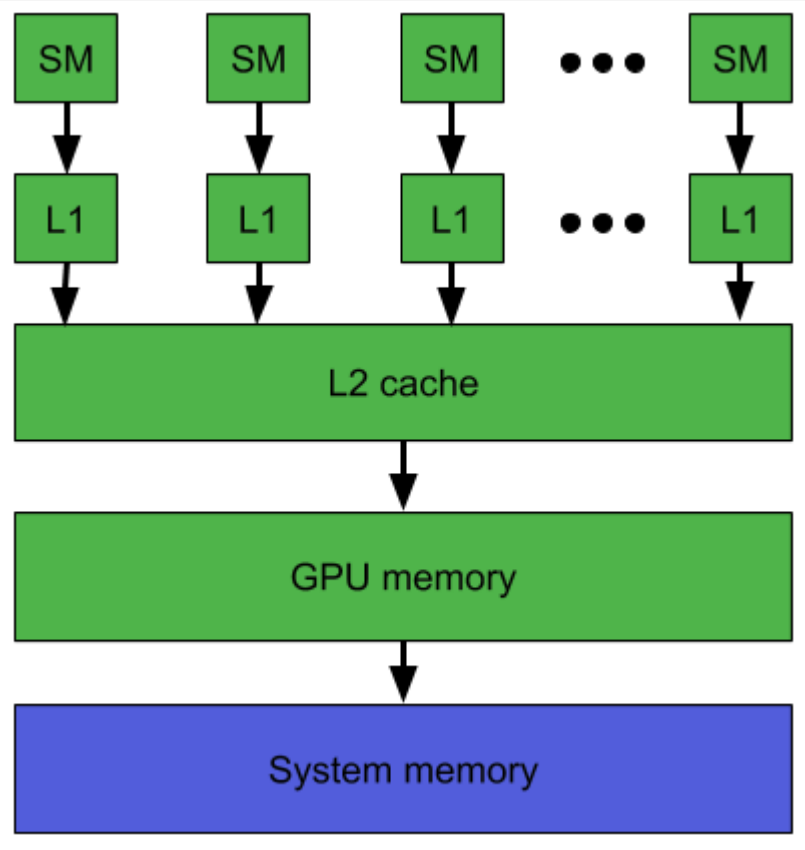
In Direct3D, you can specify a thread group size with dimensions that are not multiples of thirty-two, but for performance reasons, the thread group dimensions should always be multiples of the warp size.

Streaming Multiprocessors

Here the green colored area is the GPU. So notice that a GPU has its own memory on board. This “GPU Memory” can be from 768 megabytes to 6 gigabytes of GDDR5 memory. At present “System Memory” (blue colored one) of computers ranges from 6 gigabytes to 64 gigabytes.

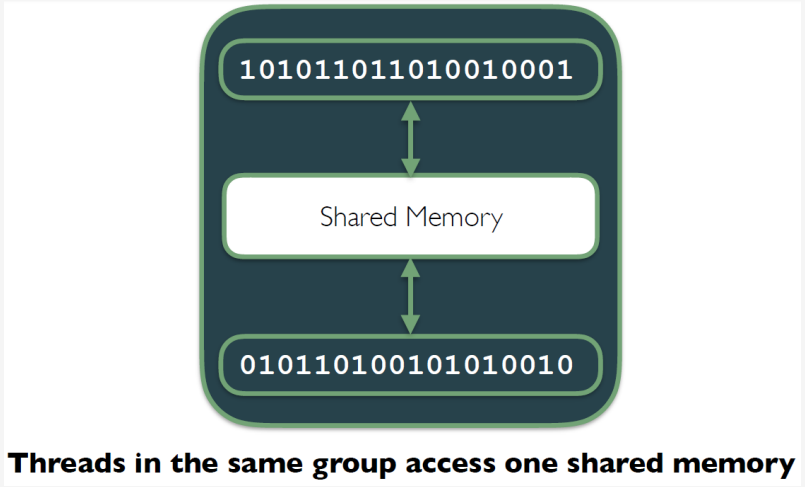
All GPUs have a cache called L2 cache. Within the CPU there is also a cache called L2 cache. The size of L2 cache on GPU is much smaller than size of L2 or L3 cache on CPU. However, bandwidth of L2 cache in GPU is much higher than that of L2 cache on CPU

For each one of these SMs, there is a cache called L1. L1 cache on GPU is much smaller than that of L1 cache on CPU, but bandwidth is much higher.

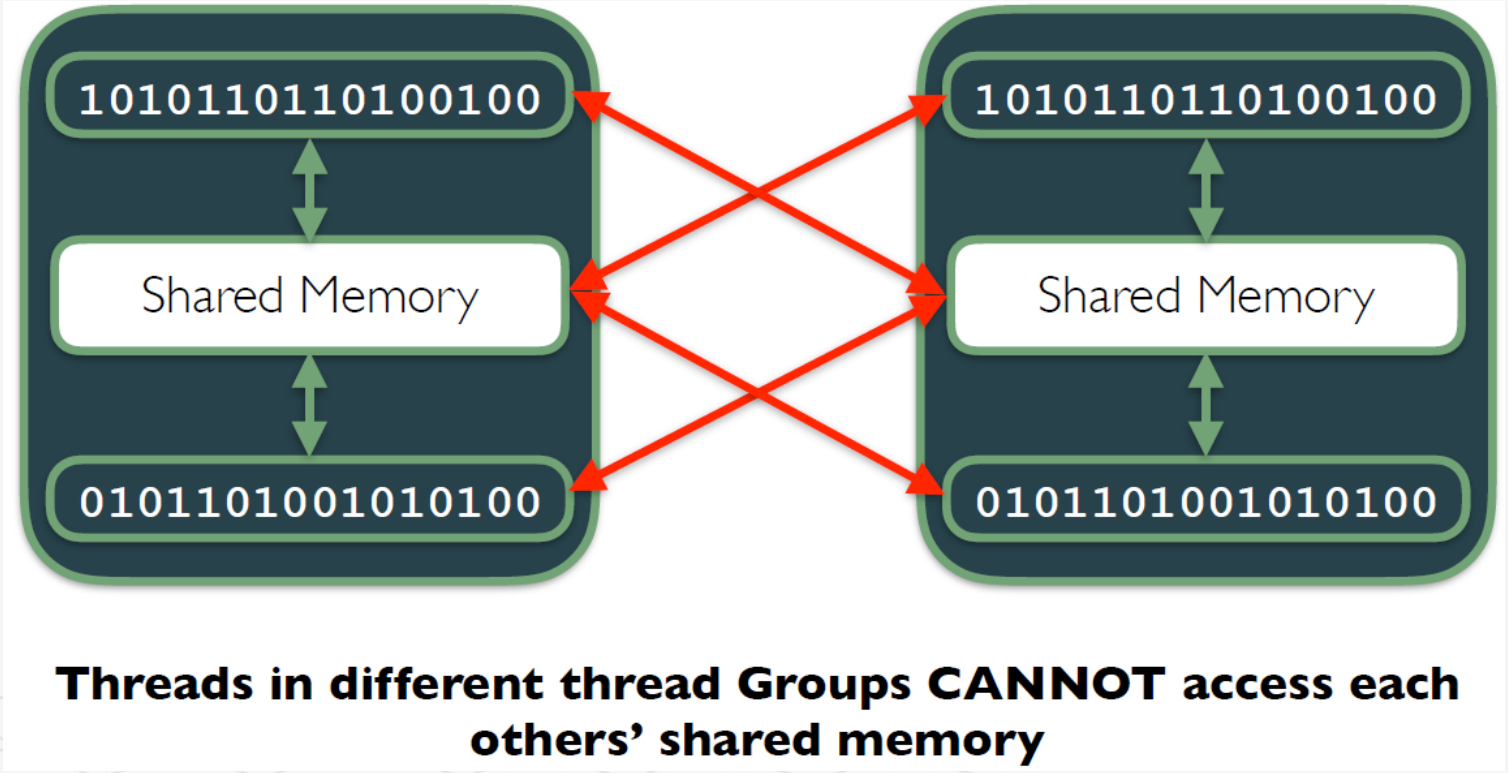


	CPU	GPU
Memory	6 - 64 GB	768 MB - 6 GB
Memory Bandwidth	24 - 32 GB/s	100 - 200 GB/s
L2 Cache	8 - 15 MB	512 - 768 kB
L1 Cache	256 - 512 kB	16 - 48 kB

Threads and Thread groups

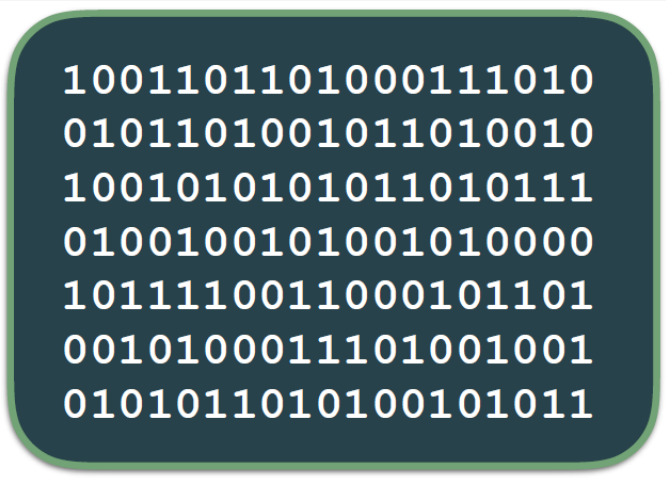


Threads in the same group access one shared memory



Threads in different thread Groups CANNOT access each others' shared memory

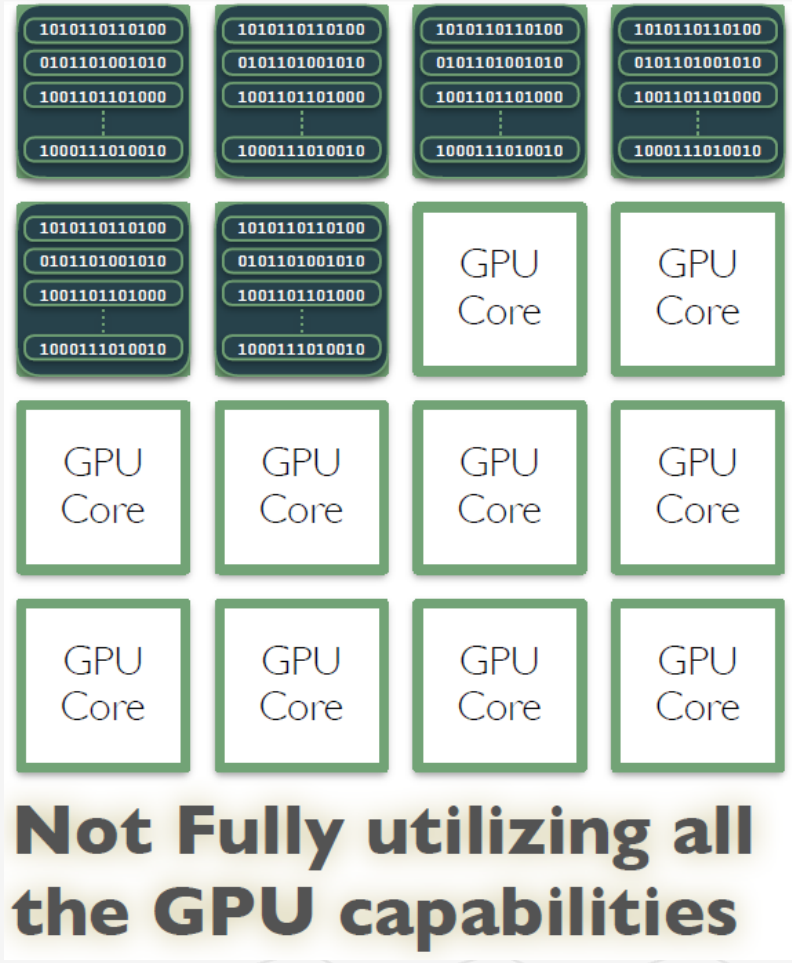
Algorithm

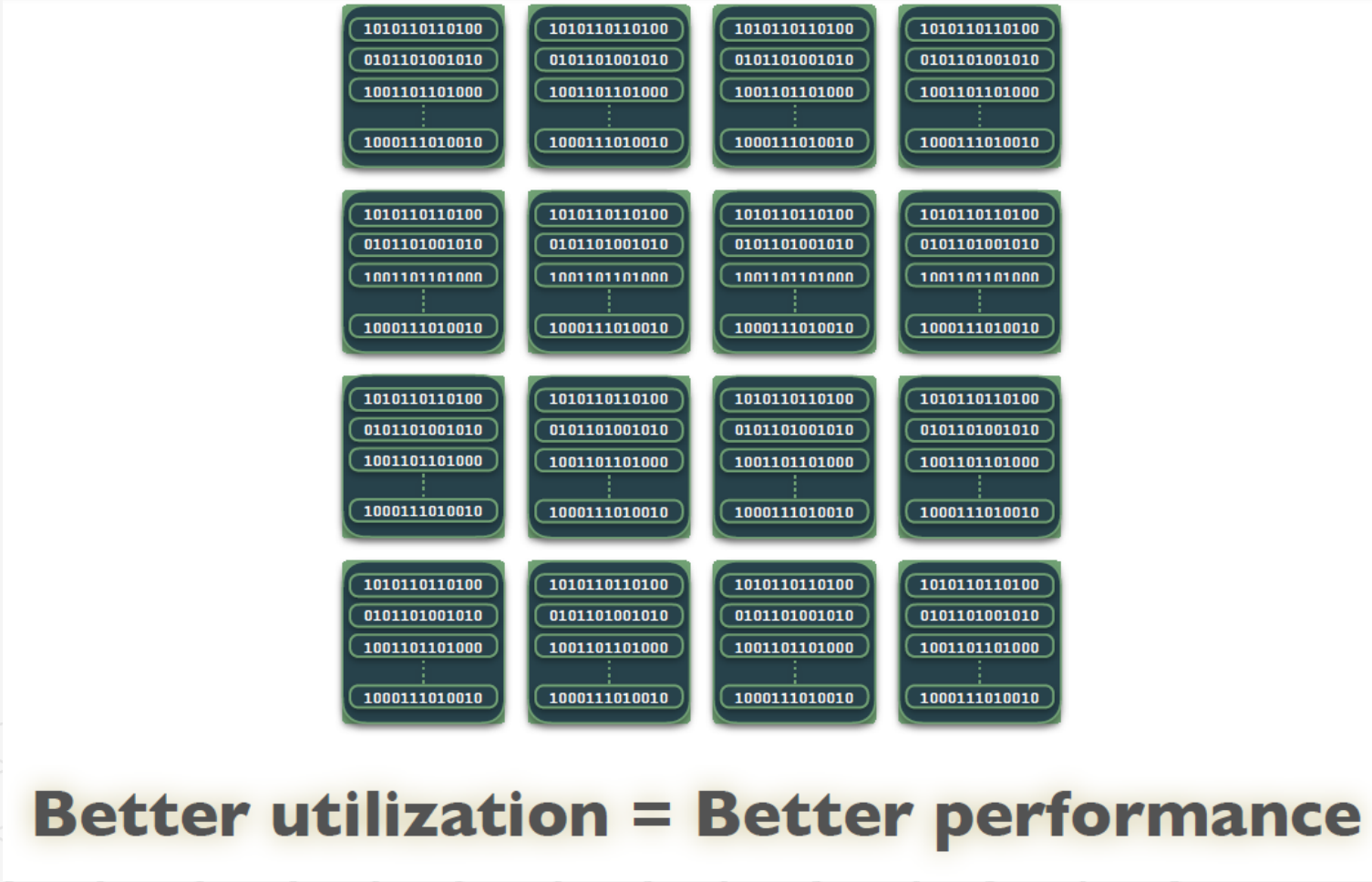


Our algorithm divided into 6 Thread Groups.

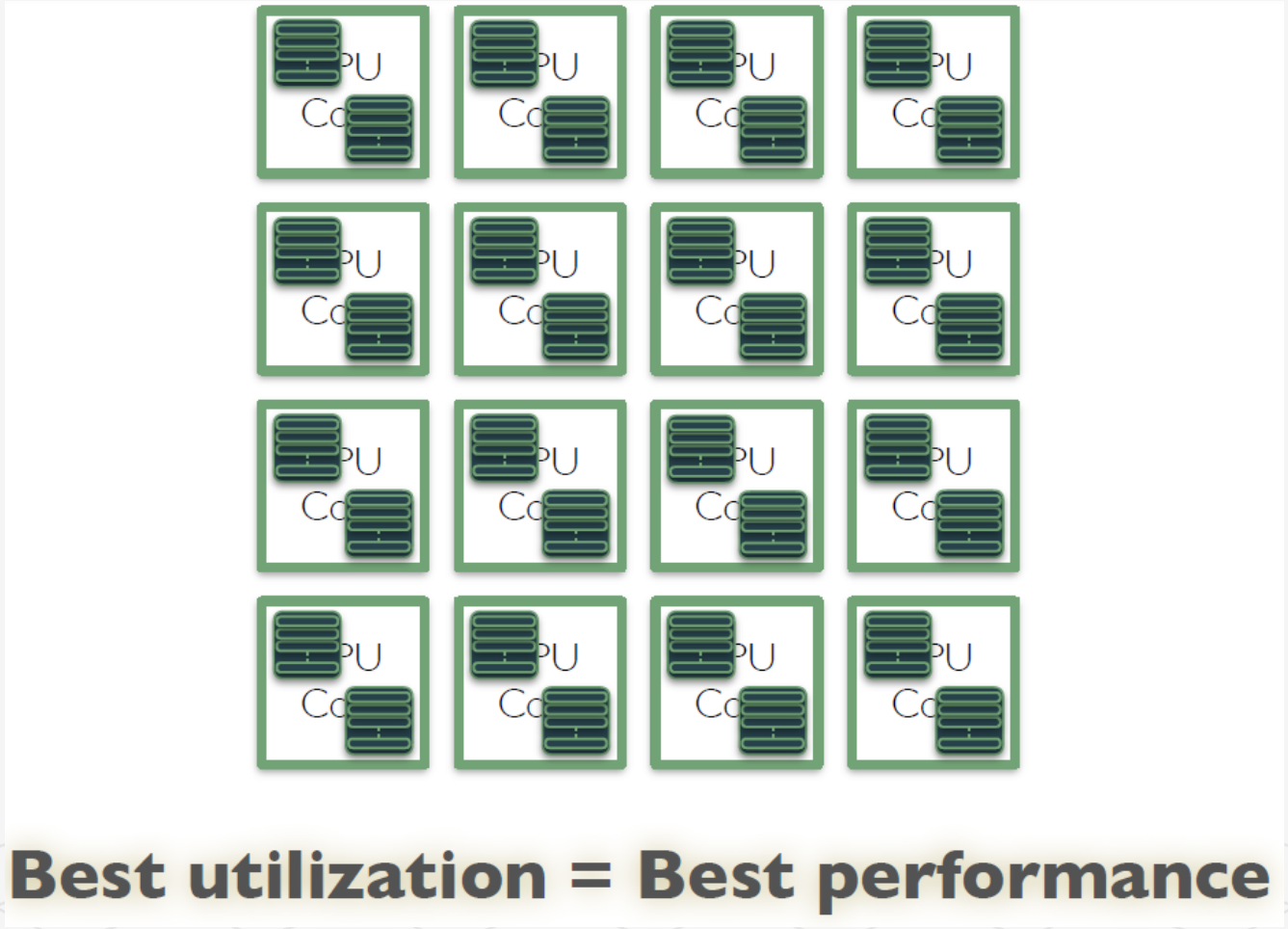
Every thread group run on 1 GPU core (compute unit).

GPU Utilization





Best Utilization

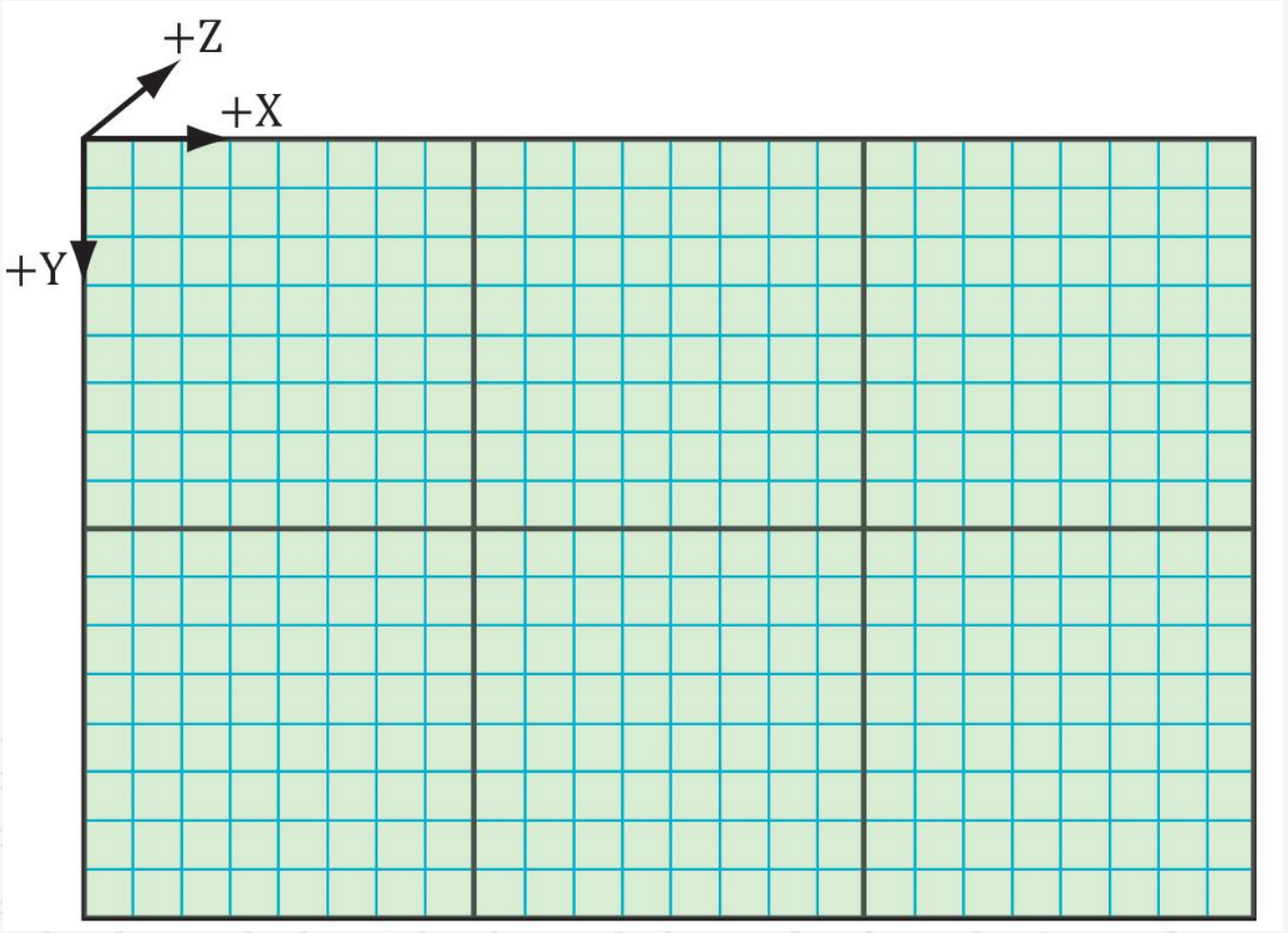


ID3D12GraphicsCommandList::`Dispatch`

In Direct3D, thread groups are launched via the following method call:

```
void ID3D12GraphicsCommandList::Dispatch(
UINT ThreadGroupCountX,
UINT ThreadGroupCountY,
UINT ThreadGroupCountZ)
```

This enables you to launch a 3D grid of thread groups; for now, we will only be concerned with 2D grids of thread groups. The following example call launches three groups in the *x* direction and two groups in the *y* direction for a total of 3 × 2 = 6 thread groups. Each thread group has 8 × 8 threads.



A SIMPLE COMPUTE SHADER

The following is a simple compute shader that sums two textures, assuming all the textures are the same size. Every computer shader needs:

1.Global variable access via constant buffers.

2.Input and output resources.

3.The [numthreads(X, Y, Z)] attribute, which specifies the number of threads in the thread group as a 3D grid of threads.

4.The shader body that has the instructions to execute for each thread.

5.Thread identification system value parameters.

```
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];
}
```

Thread ID System Values

1. Each thread group is assigned an ID by the system; this is called the group ID and has the system value semantic SV_GroupID. If Gx × Gy × Gz are the number of thread groups dispatched, then the group ID ranges from (0, 0, 0) to (Gx – 1, Gy – 1, Gz – 1).

2. Inside a thread group, each thread is given a unique ID relative to its group. If the thread group has size X × Y× Z, then the group thread IDs will range from (0, 0, 0) to (X – 1, Y – 1, Z – 1). The system value semantic for the group thread ID is SV_GroupThreadID.

3. A Dispatch call dispatches a grid of thread groups. The dispatch thread ID uniquely identifies a thread relative to all the threads generated by a Dispatch call. In other words, whereas the group thread ID uniquely identifies a thread relative to its thread group, the dispatch thread ID uniquely identifies a thread relative to the union of all the threads from all the thread groups dispatched by a Dispatch call. Let, ThreadGroupSize = (X,Y,Z) be the thread group size, then the dispatch thread ID can be derived from the group ID and the group thread ID as follows:

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

The dispatch thread ID has the system value semantic SV_DispatchThreadID.

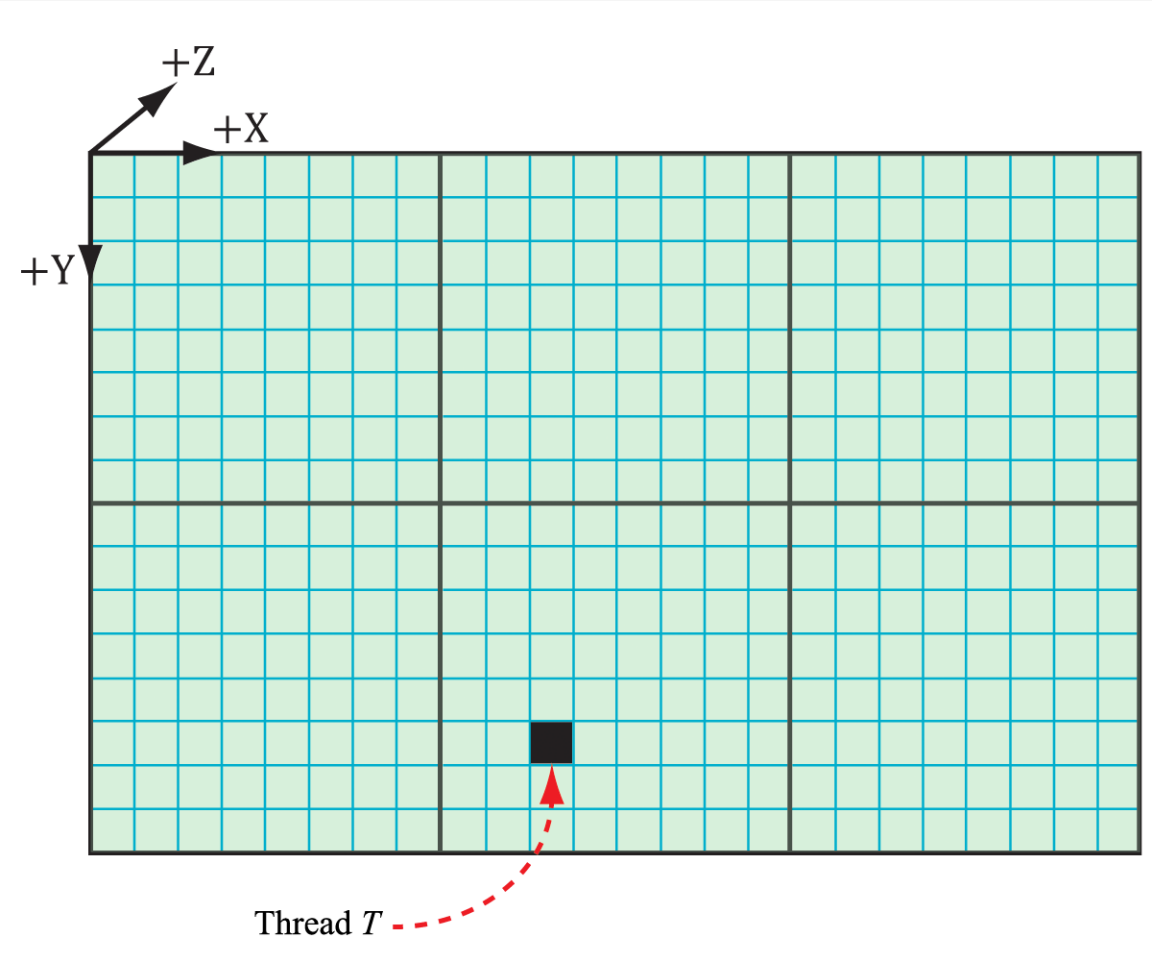
4. A linear index version of the group thread ID is given to us by Direct3D through the SV_GroupIndex system value; it is computed as:

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

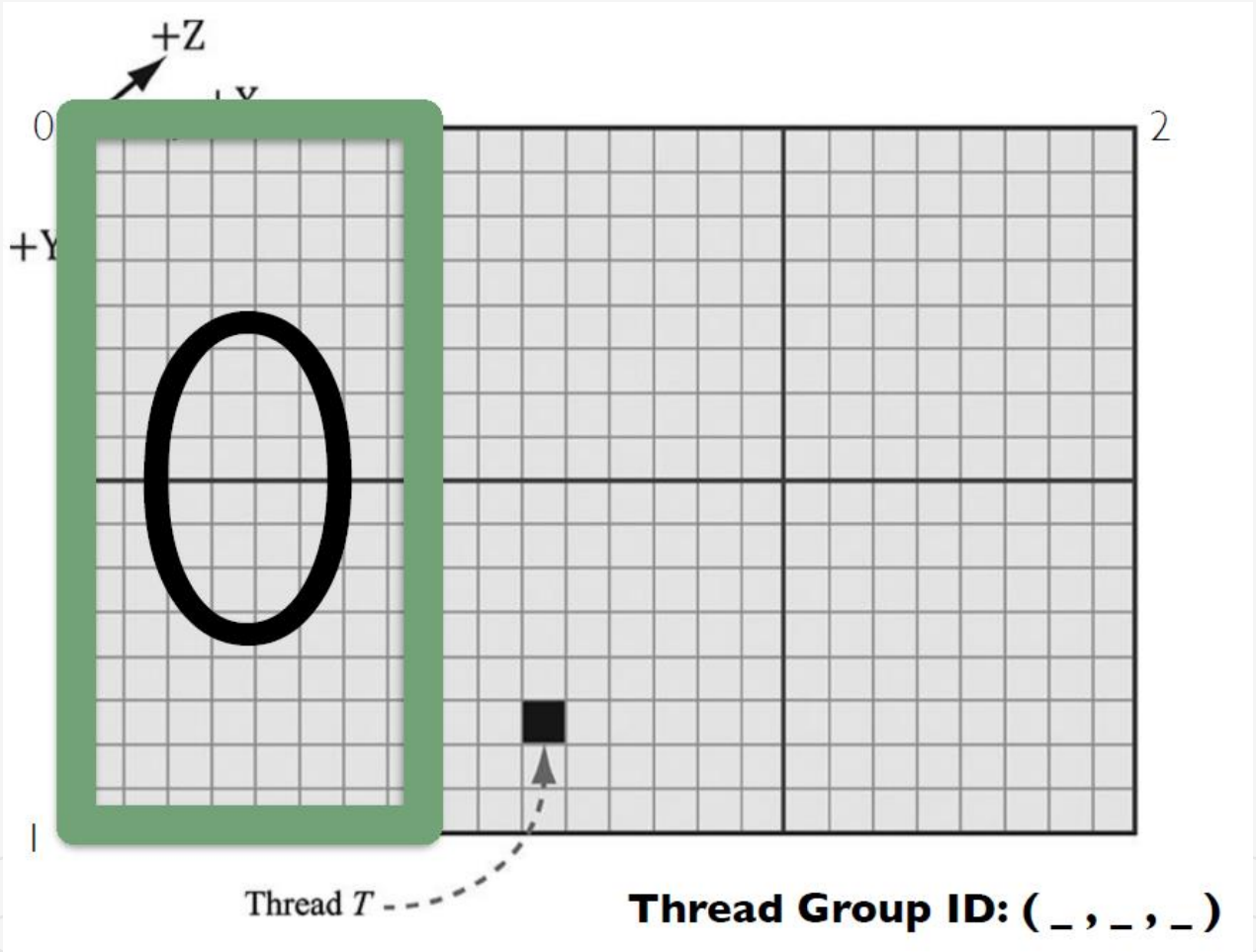
Example: If 3 × 2 thread groups are dispatched, where each thread group is 10 × 10, then a total of 600 threads are dispatched and the dispatch thread IDs will range from (0, 0, 0) to (29, 19, 0).
Thread Group ID: (0 , 0 , 0) to (2 , 1 , 0)
Group Thread ID: (0 , 0 , 0) to (9 , 9 , 0)
Dispatch Thread ID: (groupID.x **,** groupID.y **,** groupID.z **) x (10, 10, 0) + (**groupThreadID.x**,** groupThreadID.y**,** groupThreadID.z**)**

Dispatch Index ID: groupThreadID.z * 10 * 10+ groupThreadID.y * 10 + groupThreadID.x;

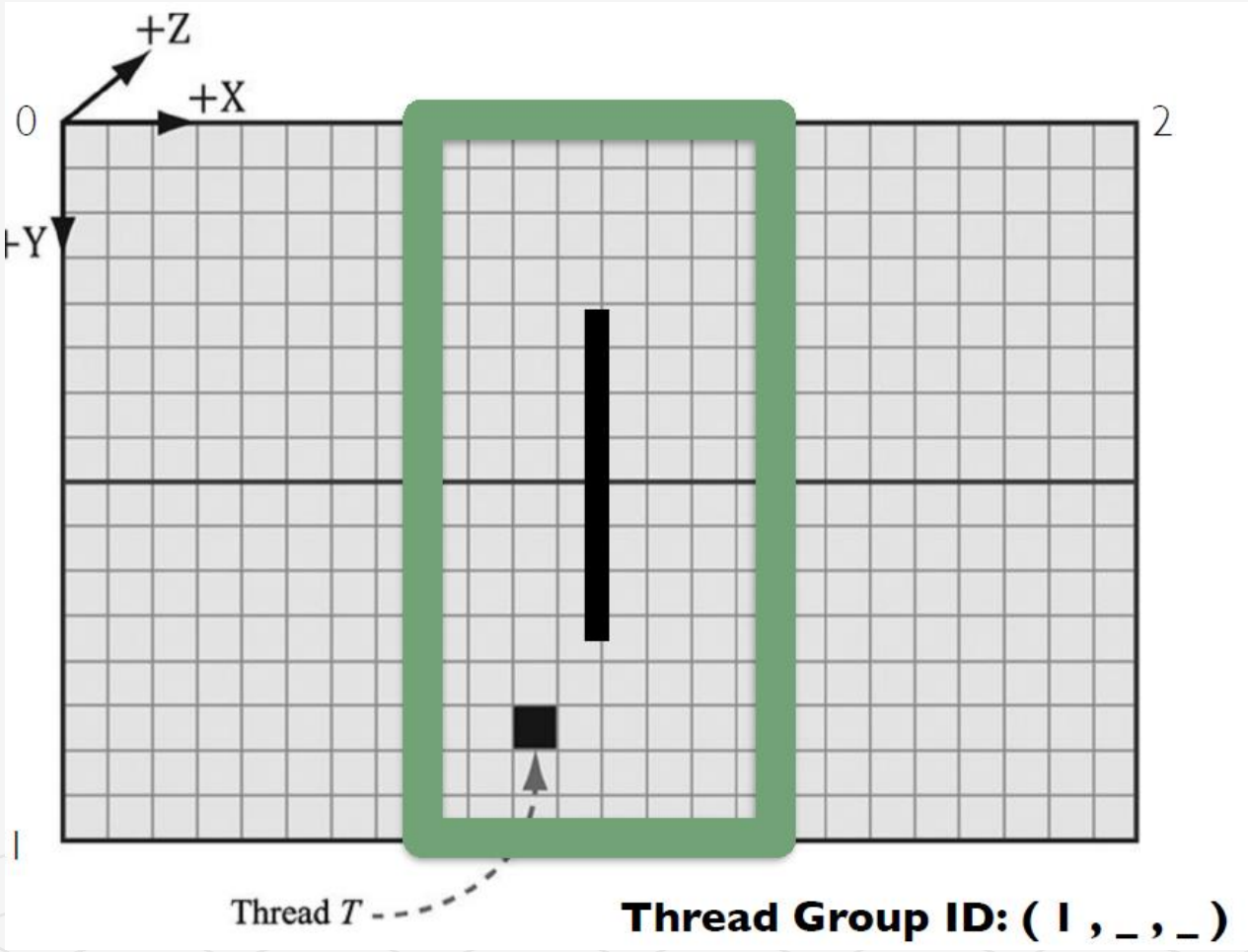
Thread ID System Values



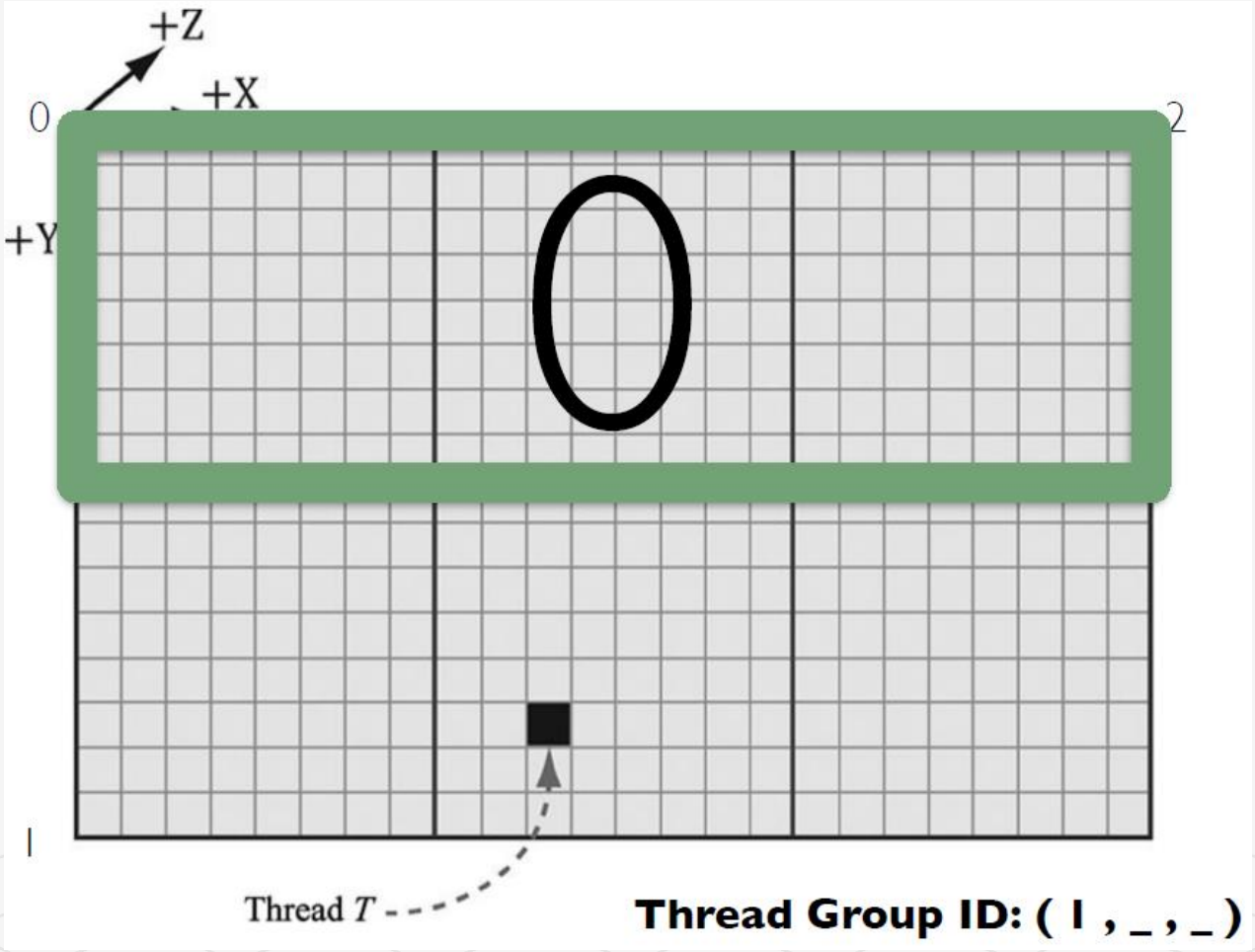
Thread ID System Values



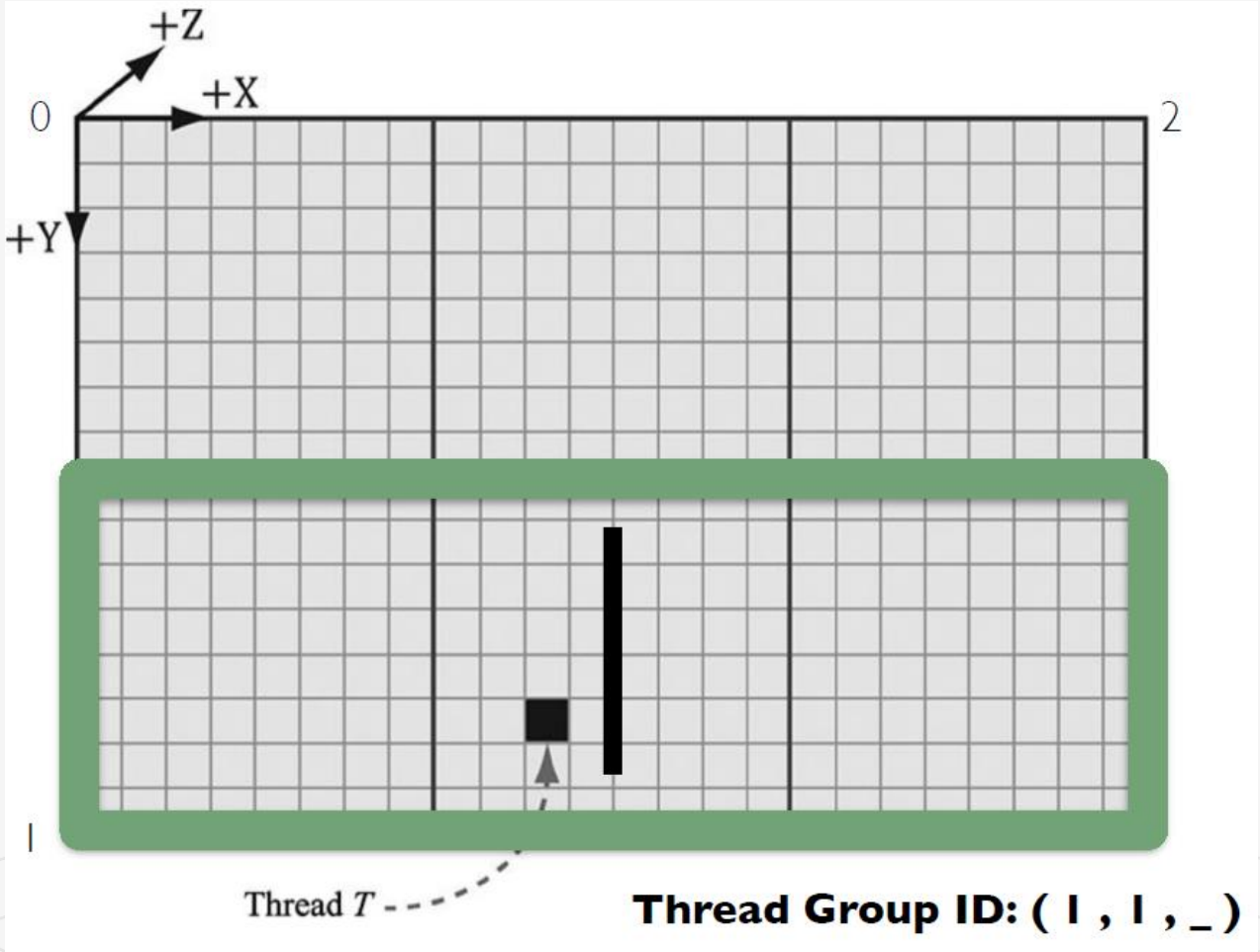
Thread ID System Values



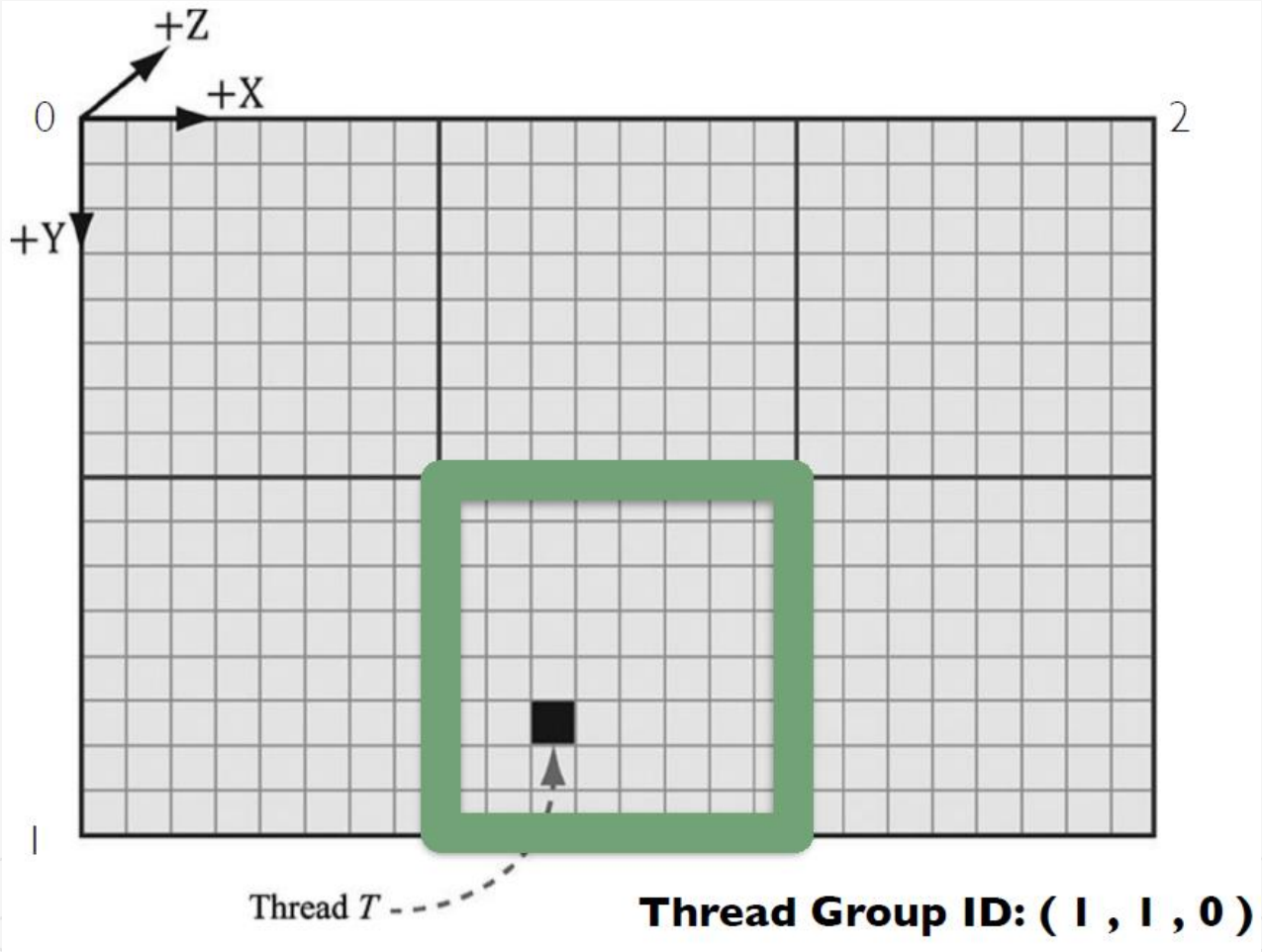
Thread ID System Values



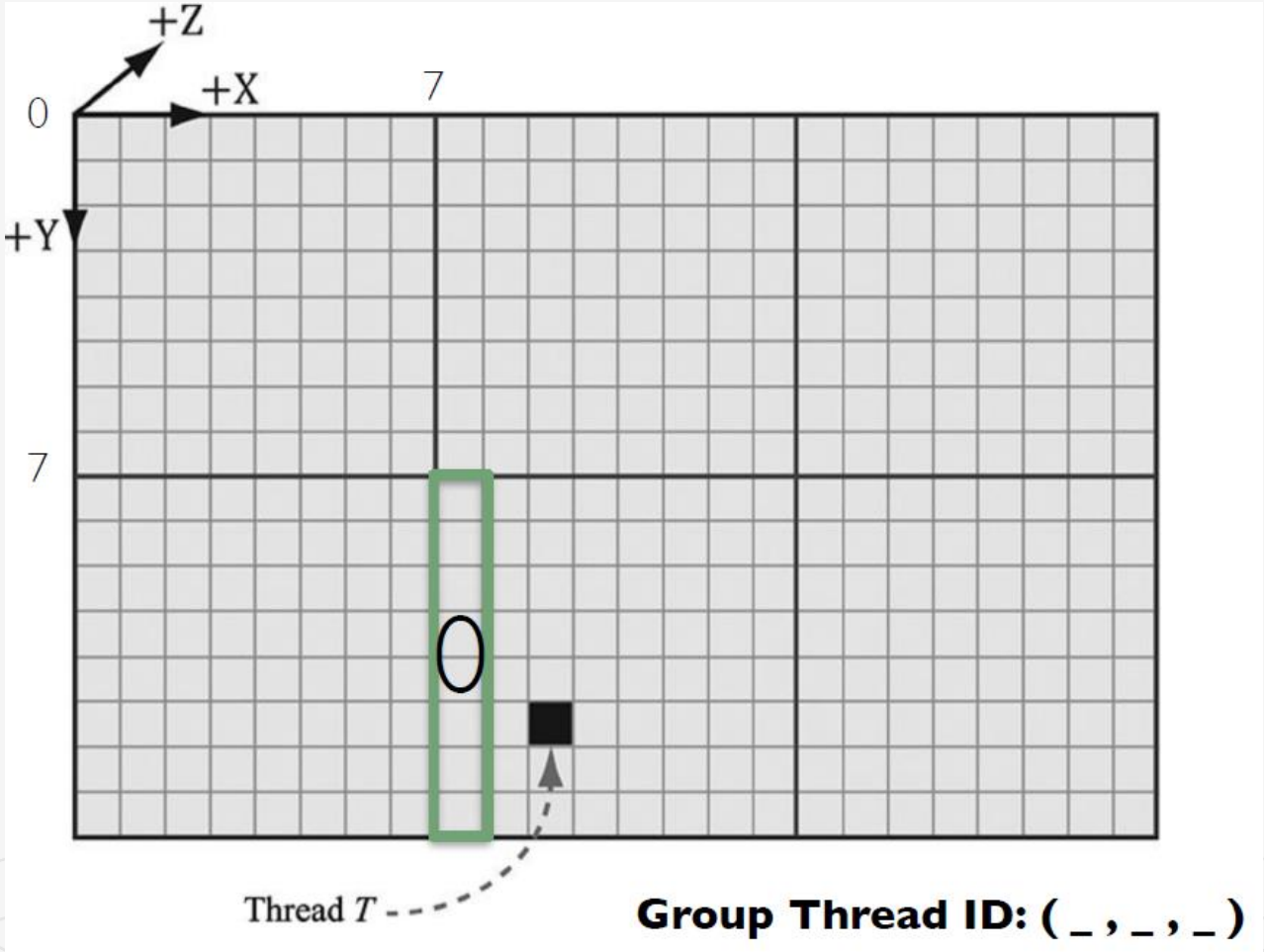
Thread ID System Values



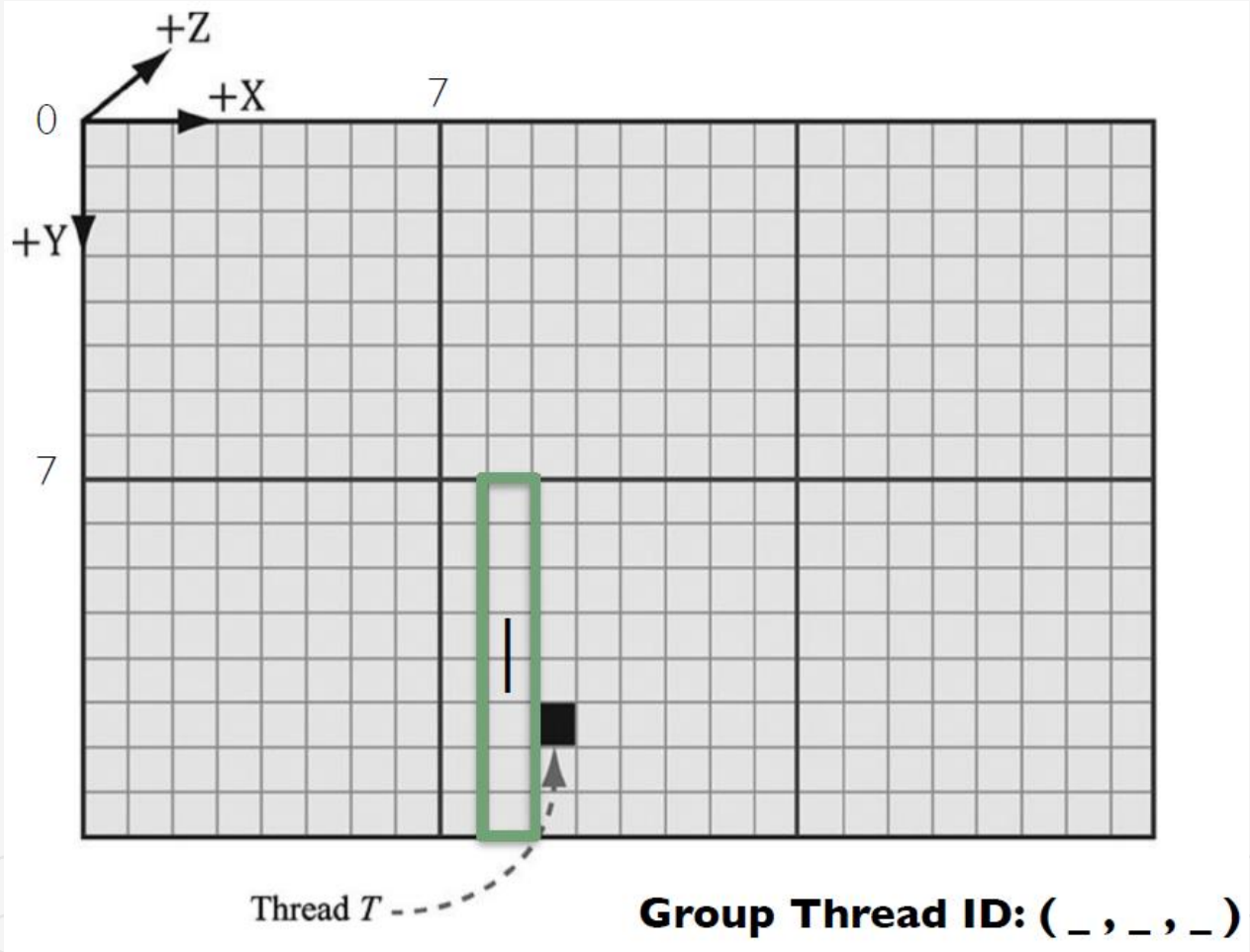
Thread ID System Values



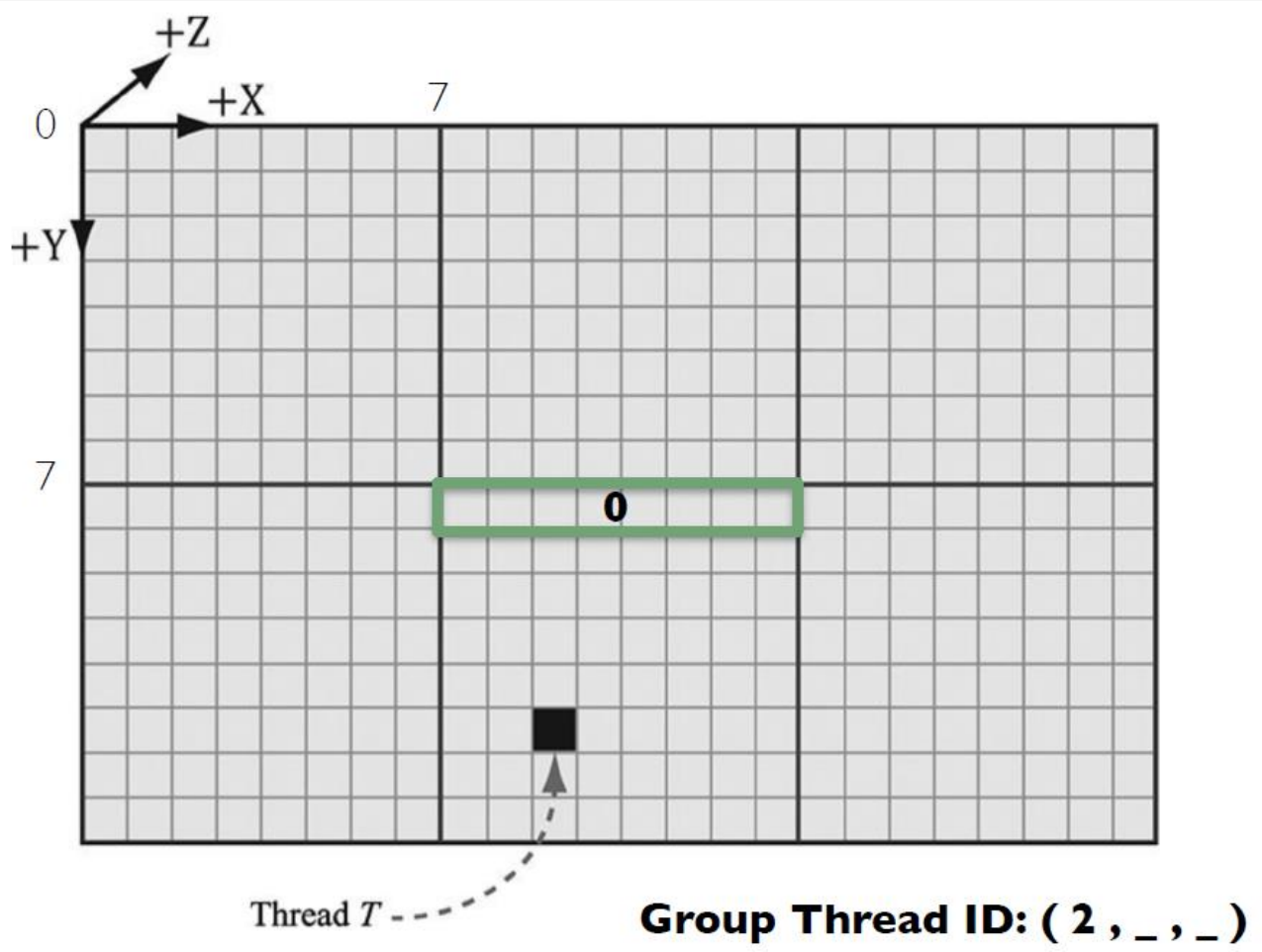
Thread ID System Values



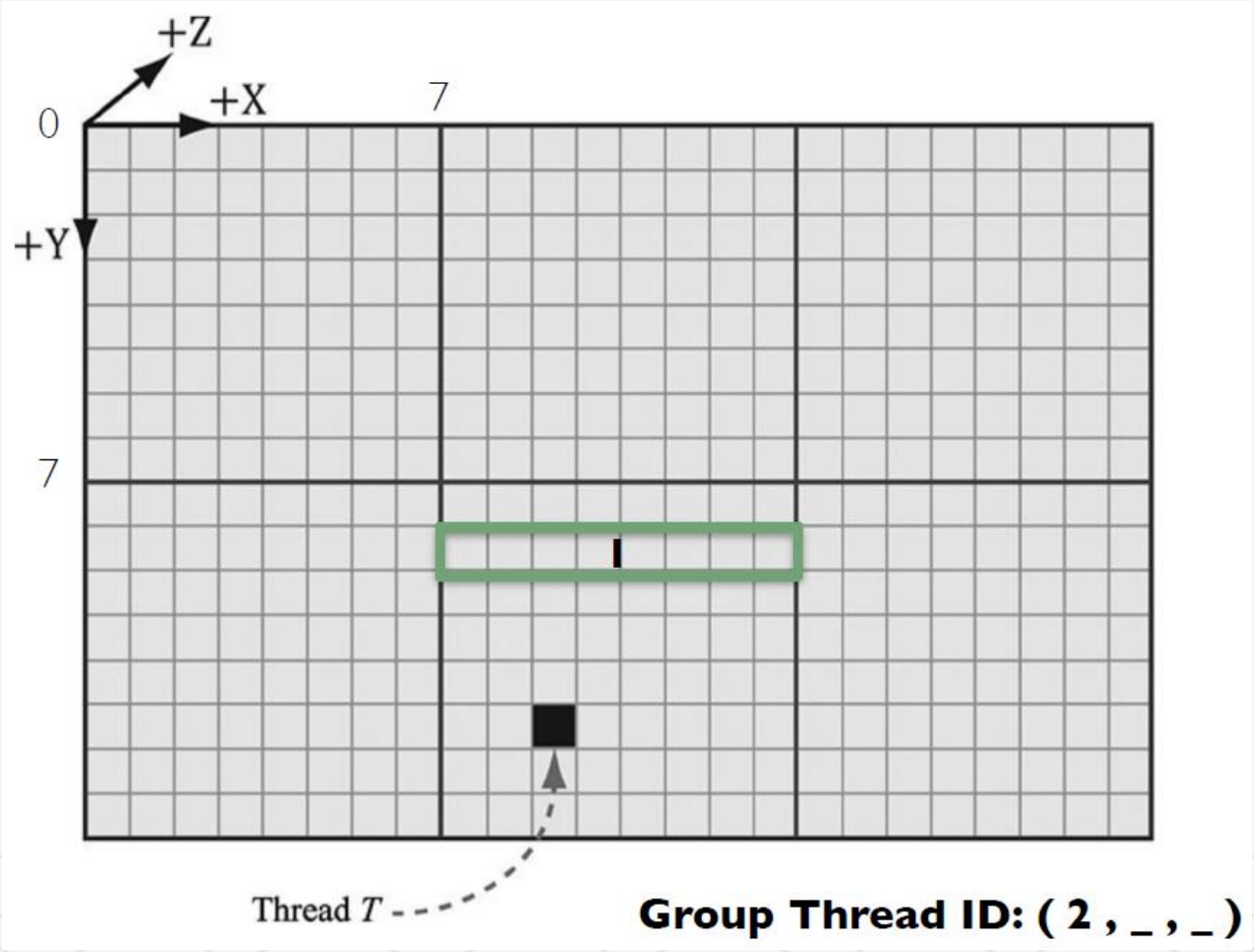
Thread ID System Values



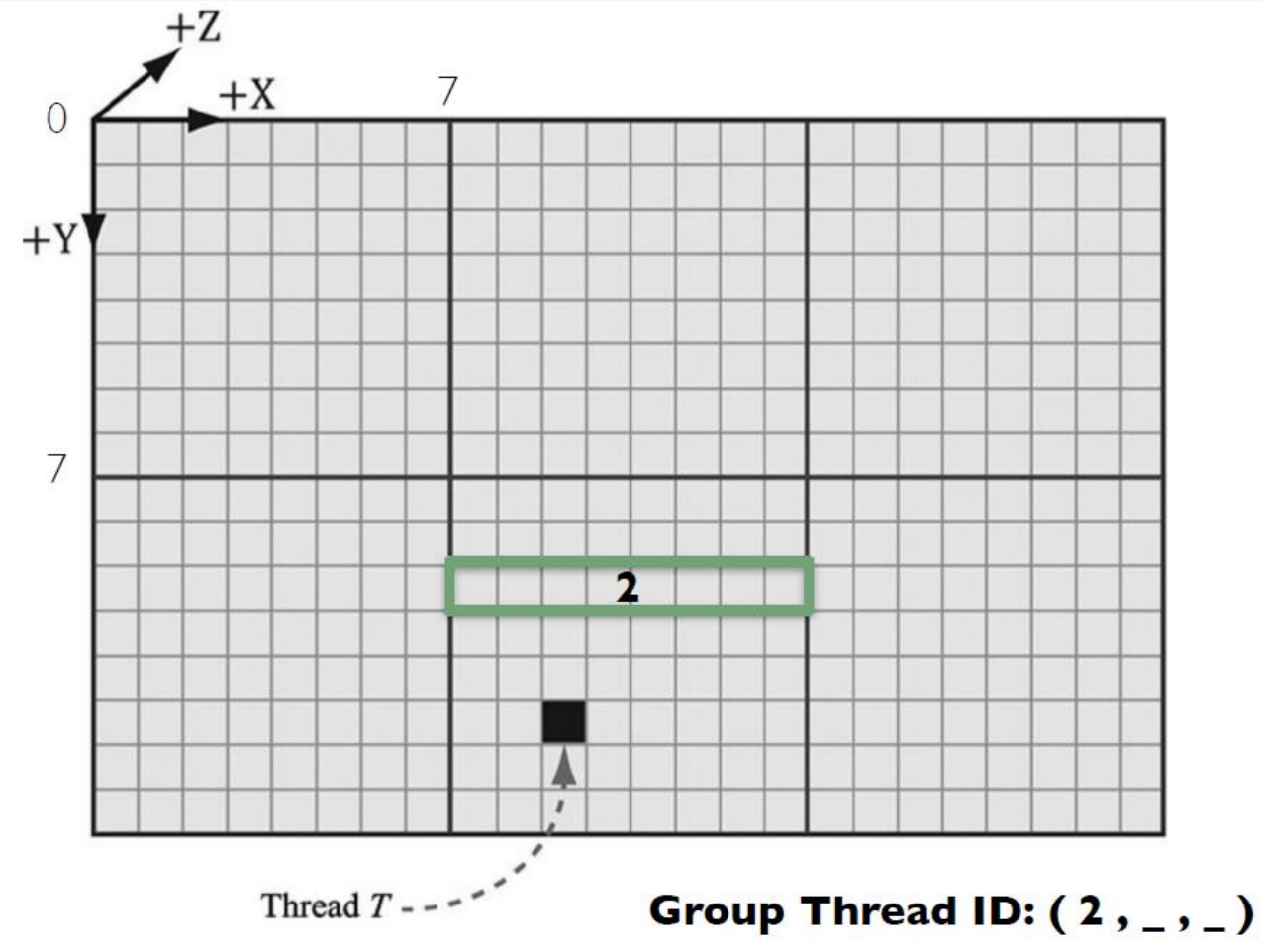
Thread ID System Values



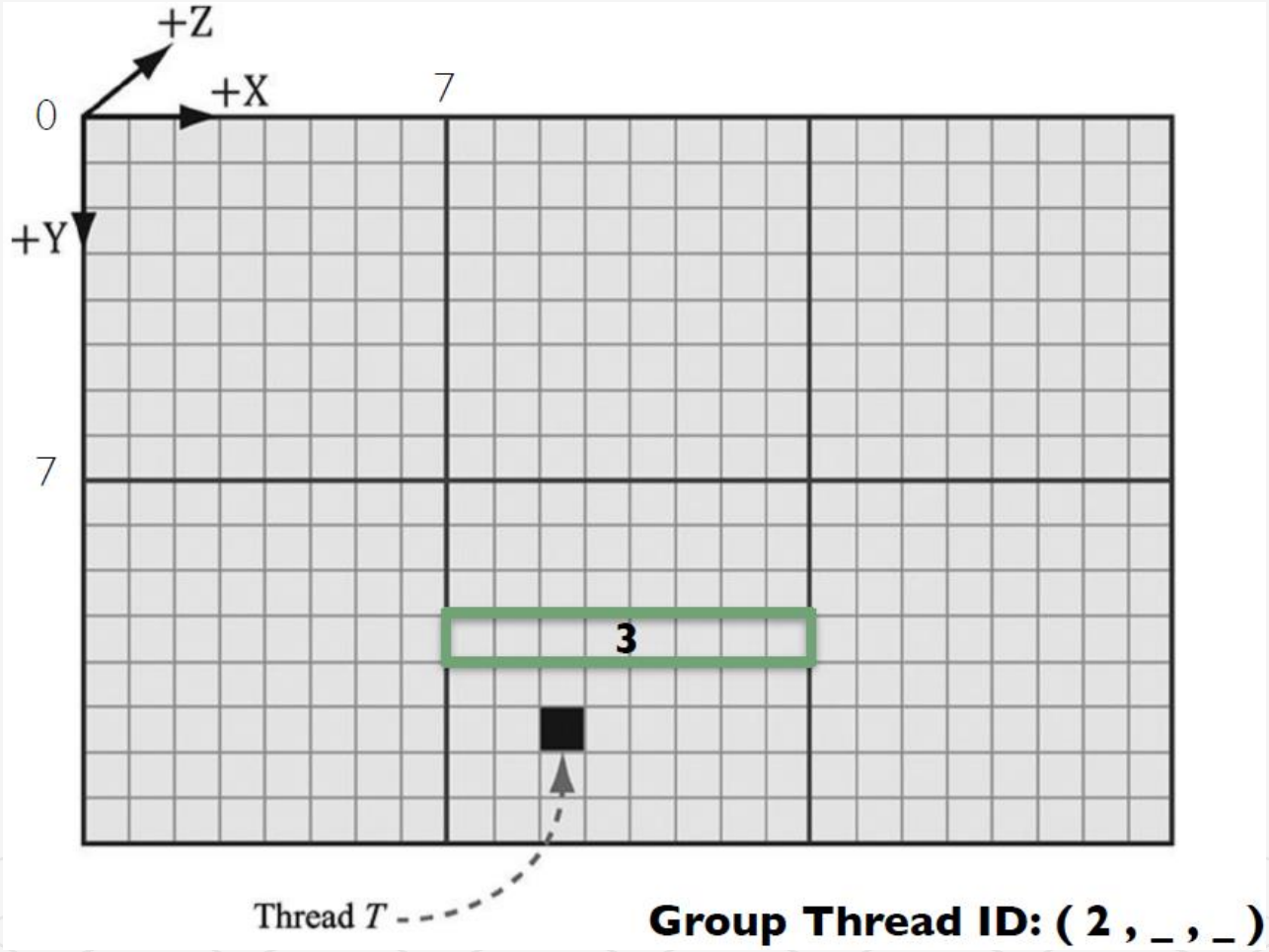
Thread ID System Values



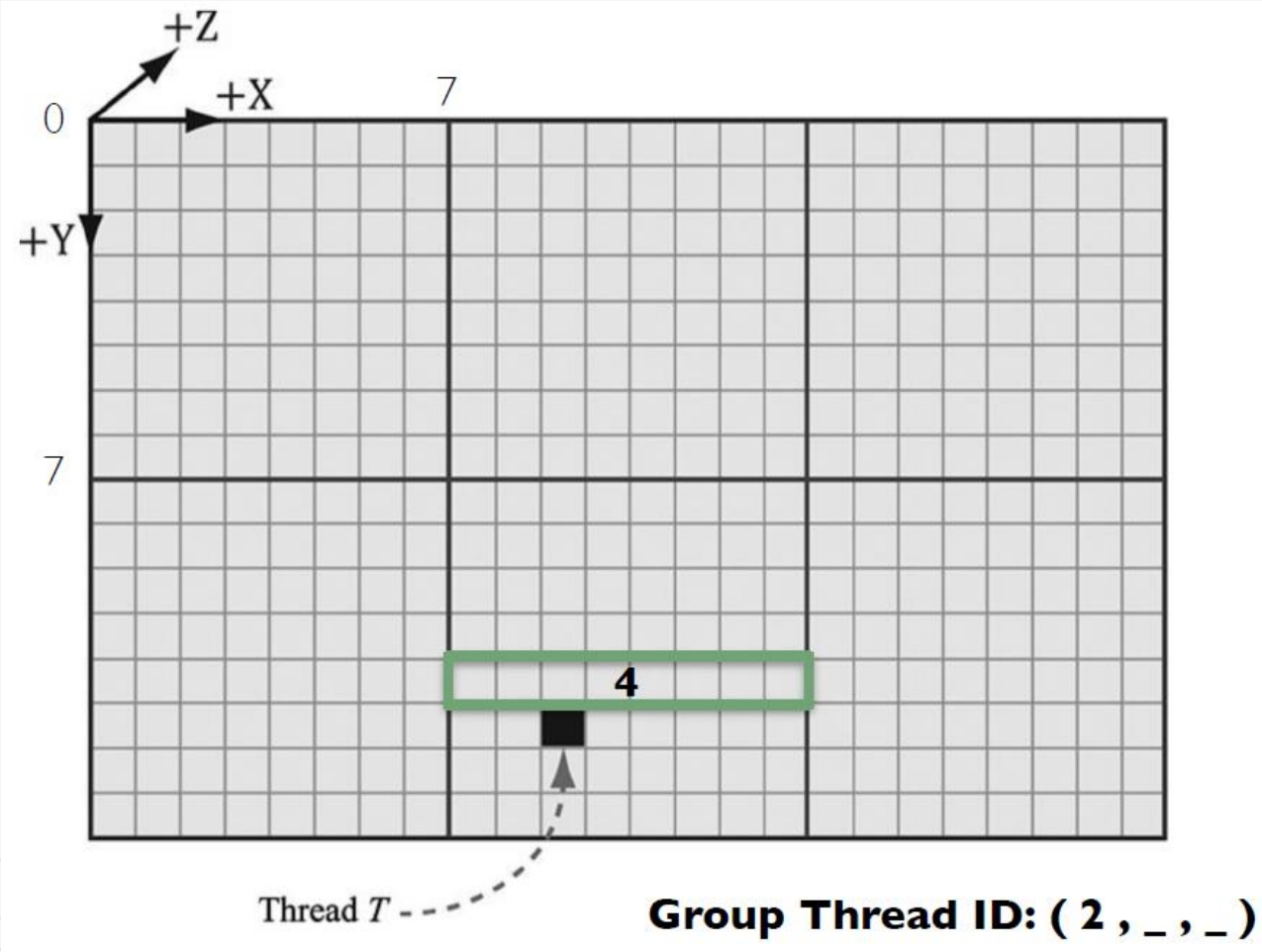
Thread ID System Values



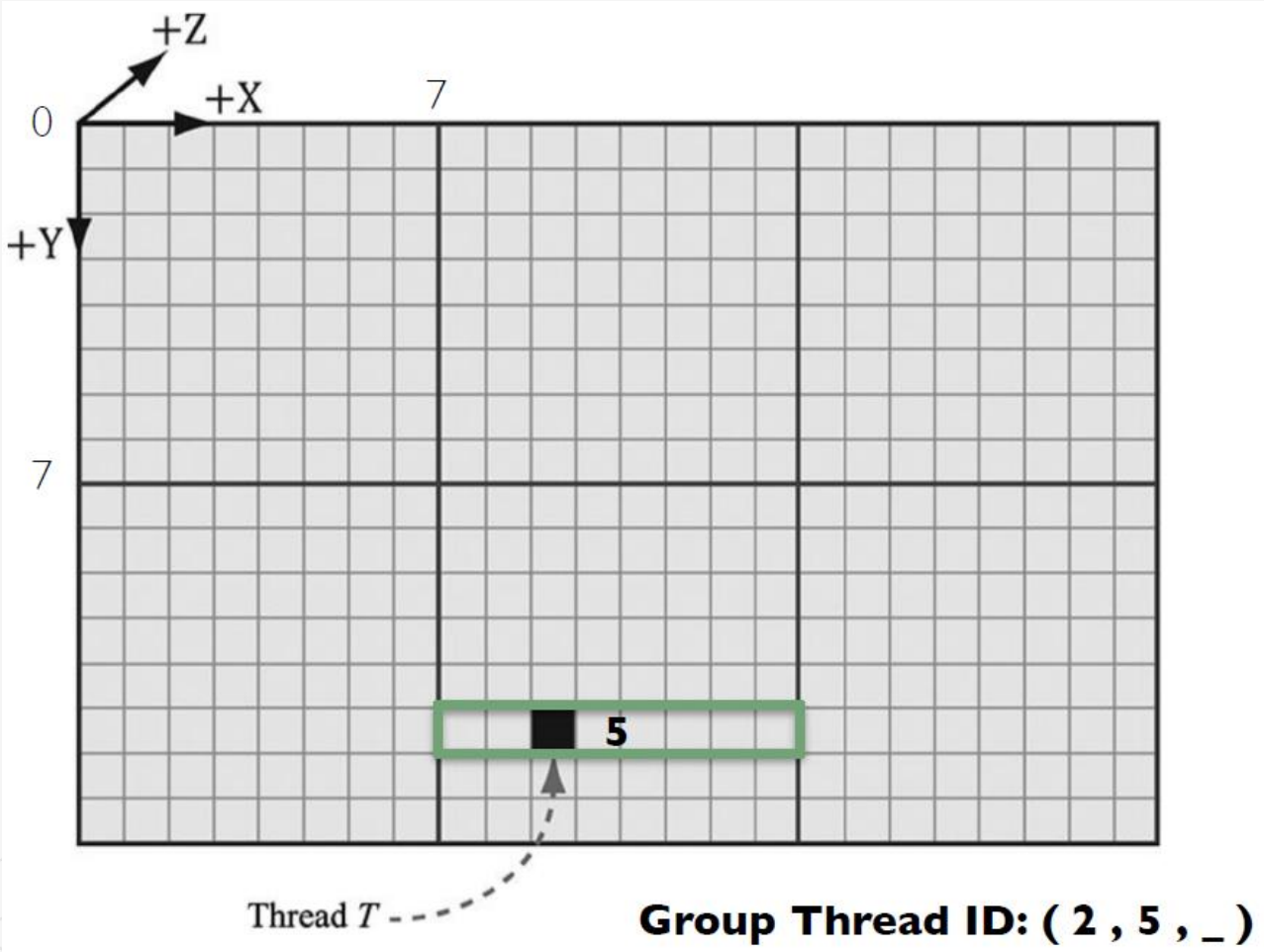
Thread ID System Values



Thread ID System Values



Thread ID System Values



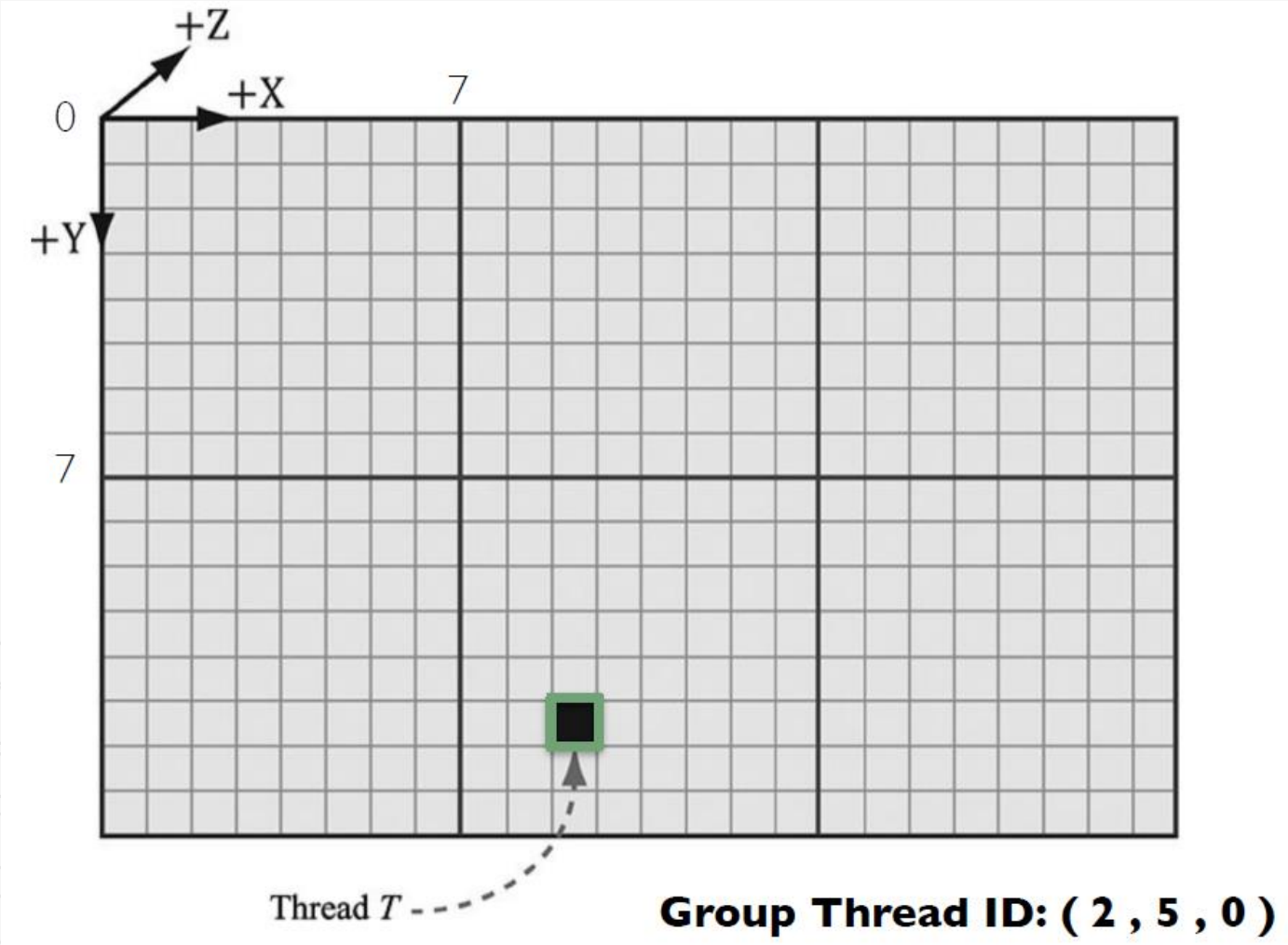
Thread ID System Values

Thread Group ID: (1 , 1 , 0)

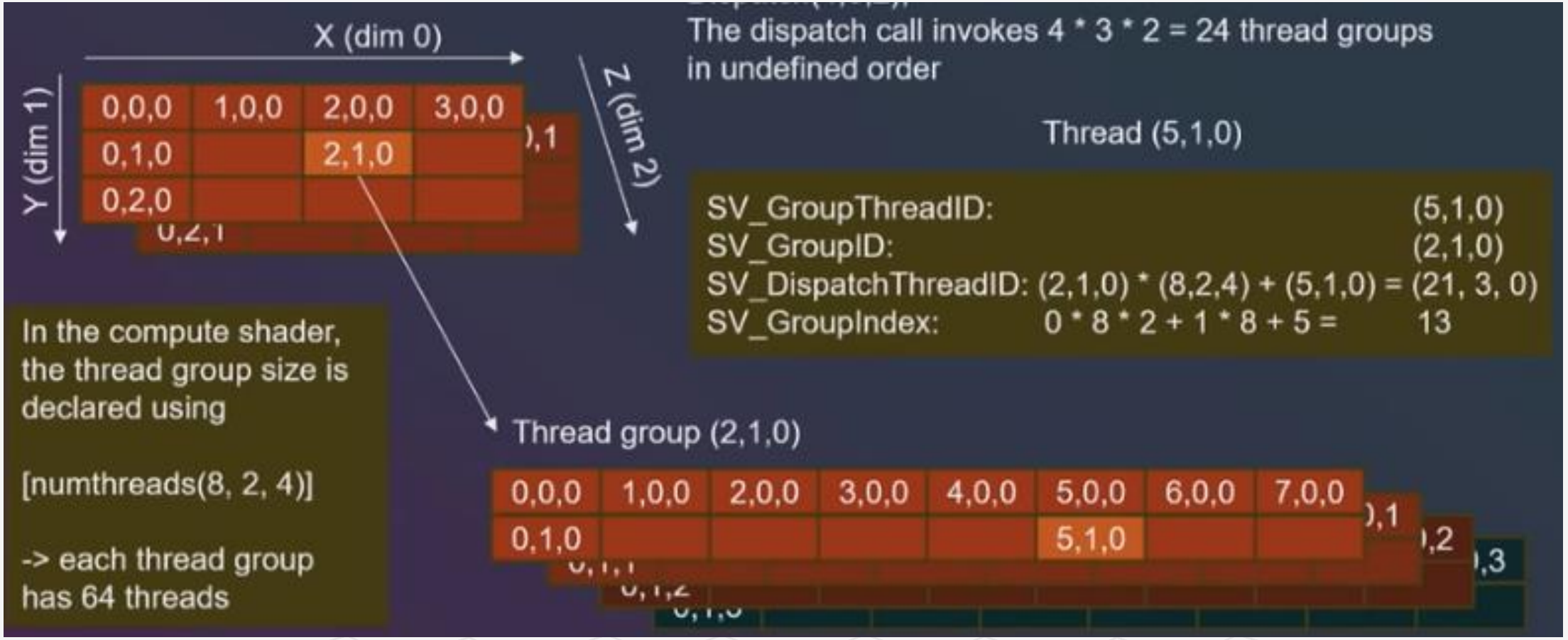
Group Thread ID: (2 , 5 , 0)

Dispatch Thread ID: (1, 1, 0) x (8, 8, 0) + (2, 5, 0) = (10 , 13 , 0)

Dispatch Index ID: 5 x 8 + 2 = 42



Example: Dispatch(4,3,2)



Scheduling Work

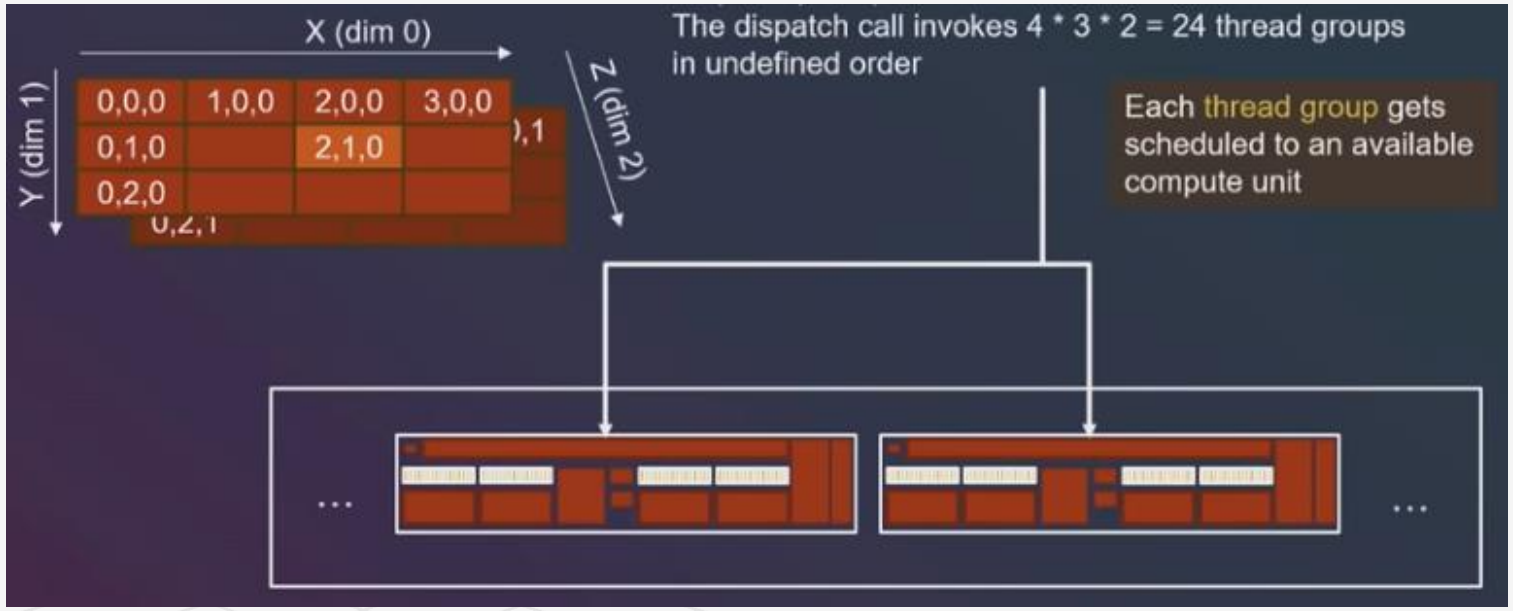
Once the whole thread group gets scheduled to a compute unit (multiprocessor), the thread gets scheduled to SIMDs.

Threads are scheduled to the SIMD units in bundles.

In the case of GCN, the size of such a bundle is 64 and called a wave front.

Per SIMD, several wavefronts can be concurrently in flight. If wavefront a is waiting for something to be complete, wavefront b can use the same SIMD unit.

The whole thread group gets scheduled to a single SIMD unit!



Compute PSO

To enable a compute shader, we use a special "compute pipeline state description."

All the graphics pipeline state does not apply to compute shaders and thus does not need to be set.

The root signature defines what parameters the shader expects as input (CBVs, SRVs, etc.).

The CS field is where we specify the compute shader.

```
mShaders["wavesUpdateCS"] = d3dUtil::CompileShader(L"Shaders\\WaveSim.hlsl", nullptr,
"UpdateWavesCS", "cs_5_0");

// PSO for updating waves

D3D12_COMPUTE_PIPELINE_STATE_DESC wavesUpdatePSO = {};

wavesUpdatePSO.pRootSignature = mWavesRootSignature.Get();

wavesUpdatePSO.CS =

{

reinterpret_cast<BYTE*>(mShaders["wavesUpdateCS"]->GetBufferPointer()),

mShaders["wavesUpdateCS"]->GetBufferSize()

};

wavesUpdatePSO.Flags = D3D12_PIPELINE_STATE_FLAG_NONE;

ThrowIfFailed(md3dDevice->CreateComputePipelineState(&wavesUpdatePSO,
IID_PPV_ARGS(&mPSOs["wavesUpdate"])));
```

DATA INPUT AND OUTPUT RESOURCES

Two types of resources can be bound to a compute shader: buffers and textures.

For example:

Texture2D gInputA;

Texture2D gInputB;

The input textures gInputA and gInputB are bound as inputs to the shader by creating (SRVs) to the textures and passing them as arguments to the root parameters;

The textures gInputA and gInputB are read-only.

```
cmdList->SetComputeRootDescriptorTable(1, mSrvA);

cmdList->SetComputeRootDescriptorTable(2, mSrvB);
```

This is exactly the same way we bind shader resource views to pixel shaders. Note that

SRVs are read-only.

Texture Outputs and Unordered Access Views (UAVs)

Output resources are treated special and have the special prefix to their type "RW," which stands for read-write:

```
RWTexture2D<float4> gOutput;
```

If our output was a 2D integer like DXGI_FORMAT_R8G8_SINT, then we would have instead written:

```
RWTexture2D<int2> gOutput;
```

To bind a resource that we will write to in a compute shader, we need to bind it using a new view type called an *unordered access view* (UAV), which is represented in code by a descriptor handle and described in code by the D3D12_UNORDERED_ACCESS_VIEW_DESC structure.

The texture must be created with the D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS flag.

Note: the texture will be bound as a UAV and as a SRV (but not simultaneously).

```
D3D12_RESOURCE_DESC texDesc;
ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texDesc.Alignment = 0;
texDesc.Width = mWidth;
texDesc.Height = mHeight;
texDesc.DepthOrArraySize = 1;
texDesc.MipLevels = 1;
texDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texDesc.SampleDesc.Count = 1;
texDesc.SampleDesc.Quality = 0;
texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS;
ThrowIfFailed(md3dDevice->CreateCommittedResource( &CD3DX12_HEAP_PROPERTIES( D3D12_HEAP_TYPE_DEFAULT)
,D3D12_HEAP_FLAG_NONE, &texDesc,D3D12_RESOURCE_STATE_COMMON,nullptr,IID_PPV_ARGS(&mBlurMap0)));
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = mFormat;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = 1;
D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};
uavDesc.Format = mFormat;
uavDesc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;
md3dDevice->CreateShaderResourceView(mBlurMap0.Get(),&srvDesc, mBlur0CpuSrv);
md3dDevice -> CreateUnorderedAccessView(mBlurMap0.Get(),nullptr, &uavDesc, mBlur0CpuUav);
```

mPostProcessRootSignature

A descriptor heap of type D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV can mix CBVs, SRVs, and UAVs all in the same heap.

Once they are in a heap, we simply pass the descriptor handles as arguments to the root parameters to bind the resources to the pipeline for a dispatch call.

The root signature defines that the shader expects a constant buffer for root parameter slot 0, an SRV for root parameter slot 1, and a UAV for root parameter slot 2.

```
void BlurApp::BuildPostProcessRootSignature()
{
CD3DX12_DESCRIPTOR_RANGE srvTable;
srvTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);

CD3DX12_DESCRIPTOR_RANGE uavTable;
uavTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_UAV, 1, 0);

// Root parameter can be a table, root descriptor or root constants.
CD3DX12_ROOT_PARAMETER slotRootParameter[3];

// Perfomance TIP: Order from most frequent to least frequent.
slotRootParameter[0].InitAsConstants(12, 0);
slotRootParameter[1].InitAsDescriptorTable(1, &srvTable);
slotRootParameter[2].InitAsDescriptorTable(1, &uavTable);

// A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3, slotRootParameter, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// create a root signature with a single slot which points to a descriptor range consisting
of a single constant buffer
ComPtr<ID3DBlob> serializedRootSig = nullptr;
ComPtr<ID3DBlob> errorBlob = nullptr;
HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc, D3D_ROOT_SIGNATURE_VERSION_1,
serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

if(errorBlob != nullptr){::OutputDebugStringA((char*)errorBlob->GetBufferPointer());}
ThrowIfFailed(hr);

ThrowIfFailed(md3dDevice->CreateRootSignature(0,serializedRootSig->GetBufferPointer(),
serializedRootSig->GetBufferSize(),
IID_PPV_ARGS(mPostProcessRootSignature.GetAddressOf())));}
```

Vertical Blur pass

Before a dispatch invocation, we bind the constants and descriptors to use for this dispatch call:

numGroupsY: How many groups do we need to dispatch to cover a column of pixels, where each group covers 256 pixels (the 256 is defined in the ComputeShader).

```
cmdList->SetPipelineState(vertBlurPSO);

cmdList->SetComputeRootDescriptorTable(1, mBlur1GpuSrv);

cmdList->SetComputeRootDescriptorTable(2, mBlur0GpuUav);


UINT numGroupsY = (UINT)ceilf(mHeight / 256.0f);

cmdList->Dispatch(mWidth, numGroupsY, 1);

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0.Get(),

D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_GENERIC_READ));


cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1.Get(),

D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
```

Indexing Textures

The elements of the textures can be accessed using 2D indices.

We index the texture based on the dispatch thread ID.

Each thread is given a unique dispatch ID.

Assuming that we dispatched enough thread groups to cover the texture (i.e., so there is one thread being executed for one texel), then this code sums the texture images and stores the result in the texture gOutput.

```
[numthreads(16, 16, 1)]

void CS(int3 dispatchThreadID : SV_DispatchThreadID)

{

// Sum the xyth texels and store the result in the xyth texel of gOutput.

gOutput[dispatchThreadID.xy] =

gInputA[dispatchThreadID.xy] +

gInputB[dispatchThreadID.xy];

}
```

Wave equation

The wave equation is a partial differential equation that may constrain some scalar function $u = u(x_1, x_2, \ldots, x_n; t)$ of a time variable t and one or more spatial variables $x_1, x_2, \ldots x_n$.

The quantity u is the displacement, along some specific direction, of the particles of a vibrating water away from their resting positions.

c is a fixed non-negative real coefficient.

The equation is

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \right)$$

Wave constants are calculated in CPU in WaveGPU.cpp file!

dx=spatial step and dt = time step

```
float d = damping*dt + 2.0f;
float e = (speed*speed)*(dt*dt) / (dx*dx);
mK[0] = (damping*dt - 2.0f) / d;
mK[1] = (4.0f - 8.0f*e) / d;
mK[2] = (2.0f*e) / d;
```

WaveSim.hlsl simulates wave equation using two functions:

UpdateWavesCS(): Solves 2D wave equation using the compute shader.

DisturbWavesCS(): Runs one thread to disturb a grid height and its neighbors to generate a wave.

Indexing Textures

Since compute shaders are not used for rendering directly, it does not know how to automatically select a mipmap level.

we must explicitly specify the level with SampleLevel in a compute shader when we sample a texture.

The texture size (*width*, *height*) can be set to a constant buffer variable, and then normalized texture coordinates can be derived from the integer indices (*x*, *y*):

$$u = \frac{x}{width} \qquad v = \frac{y}{height}$$

The following code shows a compute shader using integer indices, and a second equivalent version (next slide) using texture coordinates and SampleLevel, where it is assumed the texture size is 512 × 512 and we only need the top level mip.

```
How many groups do we need to dispatch to
cover the wave grid. Note that mNumRows and
mNumCols should be divisible by 16 so there is
no remainder.

UINT numGroupsX = mNumCols / 16;
UINT numGroupsY = mNumRows / 16;
cmdList->Dispatch(numGroupsX, numGroupsY, 1);
```

The following code solves 2D wave equation using the compute shader.

`gNextSolOutput is the new wave texel. We use two` for ping-ponging the textures.

Ping-pong buffers in preparation for the next update: The previous solution is no longer needed and becomes the target of the next solution in the next update. The current solution becomes the previous solution. The next solution becomes the current solution.

```
// VERSION 1: Using integer indices.

RWTexture2D<float> gPrevSolInput : register(u0);
RWTexture2D<float> gCurrSolInput : register(u1);
RWTexture2D<float> gNextSolOutput : register(u2);
[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
int x = dispatchThreadID.x;
int y = dispatchThreadID.y;
gNextSolOutput[int2(x, y)] =
gWaveConstants0 * gPrevSolInput[int2(x, y)].r +
gWaveConstants1 * gCurrSolInput[int2(x, y)].r +
gWaveConstants2 * (
gCurrSolInput[int2(x, y + 1)].r +
gCurrSolInput[int2(x, y - 1)].r +
gCurrSolInput[int2(x + 1, y)].r +
gCurrSolInput[int2(x - 1, y)].r);
}
```

Sampling Textures: Using SampleLevel and texture coordinates

```
cbuffer cbUpdateSettings
{
float gWaveConstant0;
float gWaveConstant1;
float gWaveConstant2;
float gDisturbMag;
int2 gDisturbIndex;
};
SamplerState samPoint : register(s0);
RWTexture2D<float> gPrevSolInput : register(u0);
RWTexture2D<float> gCurrSolInput : register(u1);
RWTexture2D<float> gNextSolOutput : register(u2);
[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
```

```
// Equivalently using SampleLevel() instead of operator [].
int x = dispatchThreadID.x;
int y = dispatchThreadID.y;
float2 c = float2(x, y) / 512.0f;
float2 t = float2(x, y - 1) / 512.0;
float2 b = float2(x, y + 1) / 512.0;
float2 l = float2(x - 1, y) / 512.0;
float2 r = float2(x + 1, y) / 512.0;
gNextSolOutput[int2(x, y)] =
gWaveConstants0 * gPrevSolInput.SampleLevel(samPoint, c, 0.0f).r +
gWaveConstants1 * gCurrSolInput.SampleLevel(samPoint, c, 0.0f).r +
gWaveConstants2 * (gCurrSolInput.SampleLevel(samPoint, b, 0.0f).r +
gCurrSolInput.SampleLevel(samPoint, t, 0.0f).r +
gCurrSolInput.SampleLevel(samPoint, r, 0.0f).r +
gCurrSolInput.SampleLevel(samPoint, l, 0.0f).r);
}
```

SampleLevel

SampleLevel samples any texture object type: Texture1D, Texture2D, Texture3D, Texture1DArray, Texture2DArray, TextureCube, TextureCubeArray

This function is similar to Sample except that it uses the LOD level (in the last component of the location parameter).

LOD is a number that specifies the mipmap level. If the value is = 0, the zero'th (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

For example, a 2D texture uses the first two components for uv coordinates and the third component for the mipmap level.

```
// Object Declarations Texture1D g_txRandom;

 SamplerState g_samPoint

{

Filter = MIN_MAG_MIP_POINT;

 AddressU = Wrap;

AddressV = Wrap;

 };

// Shader body calling the intrinsic function

float3 RandomDir(float fOffset)

{

float tCoord = (fOffset) / 300.0;

 return g_txRandom.SampleLevel( g_samPoint, tCoord, 0 ); ...
```

Structured Buffer Resources

A structured buffer is simply a buffer of elements of the same type—essentially an array. Structured buffers are defined in the HLSL:

```
struct Data
{
float3 v1;
float2 v2;
};
StructuredBuffer<Data> gInputA : register(t0);
StructuredBuffer<Data> gInputB : register(t1);
RWStructuredBuffer<Data> gOutput : register(u0);
```

A structured buffer used as an SRV can be created just like we have been creating our vertex and index buffers. A structured buffer used as a UAV is almost created the same way, except that we must specify the flag D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS

```
struct Data
{
XMFLOAT3 v1;
XMFLOAT2 v2;
};
// Generate some data to fill the SRV buffers with.
std::vector<Data> dataA(NumDataElements);
std::vector<Data> dataB(NumDataElements);
for (int i = 0; i < NumDataElements; ++i)
{
dataA[i].v1 = XMFLOAT3(i, i, i);
dataA[i].v2 = XMFLOAT2(i, 0);
dataB[i].v1 = XMFLOAT3(-i, i, 0.0f);
dataB[i].v2 = XMFLOAT2(0, -i);
}
UINT64 byteSize = dataA.size() * sizeof(Data);
// Create some buffers to be used as SRVs.
mInputBufferA = d3dUtil::CreateDefaultBuffer( md3dDevice.Get(),
mCommandList.Get(), dataA.data(), byteSize, mInputUploadBufferA);
mInputBufferB = d3dUtil::CreateDefaultBuffer( md3dDevice.Get(),
mCommandList.Get(), dataB.data(), byteSize, mInputUploadBufferB);
// Create the buffer that will be a UAV.
ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE, &CD3DX12_RESOURCE_DESC::Buffer(byteSize,
D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS),
D3D12_RESOURCE_STATE_UNORDERED_ACCESS, nullptr,
IID_PPV_ARGS(&mOutputBuffer)));
```

Structured Buffer Resources

Structured buffers are bound to the pipeline just like textures. We create SRVs or UAV descriptors to them and pass them as arguments to root parameters that take descriptor tables.

Alternatively, we can define the root signature to take root descriptors so that wecan pass the virtual address of resources directly as root arguments without the need to go through a descriptor heap (this only works for SRVs and UAVs to buffer resource, not textures).

Then we can bind our buffers like so to be used for a dispatch call:

```
// Root parameter can be a table, root descriptor or root constants.

CD3DX12_ROOT_PARAMETER slotRootParameter[3];

// Perfomance TIP: Order from most frequent to least frequent.

slotRootParameter[0].InitAsShaderResourceView(0);

slotRootParameter[1].InitAsShaderResourceView(1);

slotRootParameter[2].InitAsUnorderedAccessView(0);

// A root signature is an array of root parameters.

CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3,slotRootParameter,0,
nullptr,D3D12_ROOT_SIGNATURE_FLAG_NONE);

mCommandList->SetComputeRootSignature(mRootSignature.Get());

mCommandList->SetComputeRootShaderResourceView(0,mInputBufferA->GetGPUVirtualAddress());

mCommandList->SetComputeRootShaderResourceView(1, mInputBufferB->GetGPUVirtualAddress());

mCommandList->SetComputeRootUnorderedAccessView(2, mOutputBuffer->GetGPUVirtualAddress());

mCommandList->Dispatch(1, 1, 1);
```

Copying CS Results to System Memory

With structured buffer calculations, and GPGPU computing in general, we might not display our results at all.

How do we get our results from GPU memory back to system memory?

The required way is to create system memory buffer with heap properties D3D12_HEAP_TYPE_READBACK.

Then we can use the ID3D12GraphicsCommandList::CopyResource method to copy the GPU resource to the system memory resource.

Finally, we can map the system memory buffer with the mapping API to read it on the CPU.

We have included a structured buffer demo called "VecAdd," which simply sums the corresponding vector components stored in two structured buffers.

```
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;
}
```

For simplicity, the structured buffers only contain thirty-two elements; therefore, we only have to dispatch one thread group.

VecAdd Demo (void VecAddCSApp::DoComputeWork())

After the compute shader completes its work for all threads in this demo, we copy the results to system memory and save them to file. The following code shows how to create the system memory buffer and how to copy the GPU results to CPU memory:

```
// Create a system memory version of the buffer to read the results back from.
ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_READBACK),
D3D12_HEAP_FLAG_NONE,
&CD3DX12_RESOURCE_DESC::Buffer(byteSize),
D3D12_RESOURCE_STATE_COPY_DEST,
nullptr,
IID_PPV_ARGS(&mReadBackBuffer)));
```

```
// Schedule to copy the data to the default buffer to the readback buffer.
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mOutputBuffer.Get(),
D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_COPY_SOURCE));

//We need to read the data back from the GPU after computer shader does its calculation
//mCommandList->CopyResource(Destination: CPUData, Source: GPUData);
mCommandList->CopyResource(mReadBackBuffer.Get(), mOutputBuffer.Get());

mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mOutputBuffer.Get(),
D3D12_RESOURCE_STATE_COPY_SOURCE, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
// Done recording commands.
ThrowIfFailed(mCommandList->Close());
// Add the command list to the queue for execution.
ID3D12CommandList* cmdsLists[] = { mCommandList.Get()
};
mCommandQueue -> ExecuteCommandLists(_countof(cmdsLists), cmdsLists);
// Wait for the work to finish.
FlushCommandQueue();
// Map the data so we can read it on CPU.
Data* mappedData = nullptr;
ThrowIfFailed(mReadBackBuffer->Map(0, nullptr,
reinterpret_cast<void**>(&mappedData)));
std::ofstream fout("results.txt");
for (int i = 0; i < NumDataElements; ++i)
{
fout << "(" << mappedData[i].v1.x << ", " <<
mappedData[i].v1.y << ", " <<
mappedData[i].v1.z << ", " <<
mappedData[i].v2.x << ", " <<
mappedData[i].v2.y << ")" << std::endl;
}
mReadBackBuffer->Unmap(0, nullptr);
```

Result

The resulting text file contains the following data, which confirms that the compute shader is working as expected.

(0, 0, 0, 0, 0)

(0, 2, 1, 1, -1)

(0, 4, 2, 2, -2)

(0, 6, 3, 3, -3)

(0, 8, 4, 4, -4)

(0, 10, 5, 5, -5)

(0, 12, 6, 6, -6)

(0, 14, 7, 7, -7)

(0, 16, 8, 8, -8)

(0, 18, 9, 9, -9)

(0, 20, 10, 10, -10)

(0, 22, 11, 11, -11)
(0, 24, 12, 12, -12)
(0, 26, 13, 13, -13)
(0, 28, 14, 14, -14)
(0, 30, 15, 15, -15)
(0, 32, 16, 16, -16)
(0, 34, 17, 17, -17)
(0, 36, 18, 18, -18)
(0, 38, 19, 19, -19)
(0, 40, 20, 20, -20)
(0, 42, 21, 21, -21)
(0, 44, 22, 22, -22)
(0, 46, 23, 23, -23)
(0, 48, 24, 24, -24)
(0, 50, 25, 25, -25)
(0, 52, 26, 26, -26)
(0, 54, 27, 27, -27)
(0, 56, 28, 28, -28)
(0, 58, 29, 29, -29)
(0, 60, 30, 30, -30)
(0, 62, 31, 31, -31)

SV_GroupThreadID & SV_DispatchThreadID

A compute shader generally takes some input data structure and outputs to some data structure.

We can use the thread ID values as indexes into these data structures:

The SV_GroupThreadID and `SV_DispatchThreadID` are useful for indexing into thread local storage memory.

```
Texture2D gInputA;

Texture2D gInputB;

RWTexture2D<float4> gOutput;

[numthreads(16, 16, 1)]

void CS(int3 dispatchThreadID : SV_DispatchThreadID)

{

// Use dispatch thread ID to index into output and

input textures.

gOutput[dispatchThreadID.xy] =

gInputA[dispatchThreadID.xy] +

gInputB[dispatchThreadID.xy];

}
```

APPEND AND CONSUME BUFFERS

We want to update the particle positions based on their constant acceleration and velocity in the compute shader.

We do not care about the order the particles are updated nor the order they are written to the output buffer.

Consume and append structured buffers provide the convenience that we do not have to worry about indexing:

Once a data element is consumed, it cannot be consumed again by a different thread; one thread will consume exactly one data element.

```
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);
}
```

SHARED MEMORY

In the compute shader code, shared memory (local data share in the compute unit) can be used to share local data between different threads in the same thread group for the same compute unit.

groupshared float4 gCache[256];

The array size can be whatever you want, but the maximum size of group shared memory is 64kb.

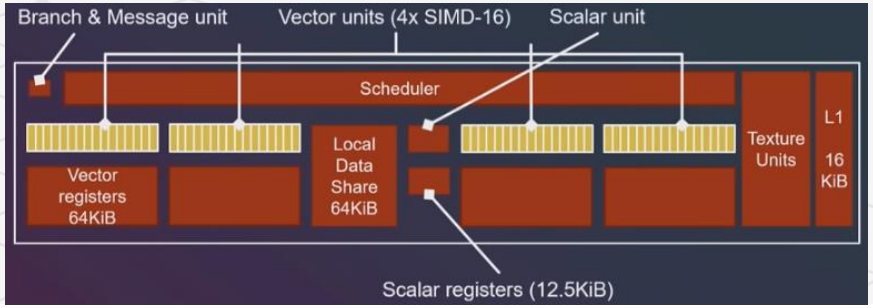
You can write into the shared memory, read from it from any threads within the same thread group.

If you want to give each thread in the group access to one slot in the shared memory, the group shared memory is indexed with the SV_ThreadGroupID.

Suppose a multiprocessor supports 32kb of shared memory, and your compute shader requires 20kb of shared memory. This means that only one thread group will fit on the multiprocessor (20kb + 20kb = 40kb > 32kb)

A common application of shared memory is to store texture values in it.

```
Texture2D gInput;
RWTexture2D<float4> gOutput;
groupshared float4 gCache[256];
[numthreads(256, 1, 1)]
void CS(int3 groupThreadID : SV_GroupThreadID,
int3 dispatchThreadID : SV_DispatchThreadID)
{
// Each thread samples the texture and stores the
// value in shared memory.
gCache[groupThreadID.x] = gInput[dispatchThreadID.xy];
// Do computation work: Access elements in shared memory
// that other threads stored:
// BAD!!! Left and right neighbor threads might not have
// finished sampling the texture and storing it in shared memory.
float4 left = gCache[groupThreadID.x - 1];
float4 right = gCache[groupThreadID.x + 1];
…
}
```



SYNCHRONIZATION

We have no guarantee that all the threads in the thread group finish at the same time.

A thread could go to access a shared memory element that is not yet initialized because the neighboring threads responsible for initializing those elements have not finished yet.

To fix this problem, before the compute shader can continue, it must wait until all the threads have done their texture loading into shared memory.

This is accomplished by a synchronization command:
`GroupMemoryBarrierWithGroupSync();`

`Note: groupshared memory is even faster than L1 cache!`

```
Texture2D gInput;
RWTexture2D<float4> gOutput;
groupshared float4 gCache[256];
[numthreads(256, 1, 1)]
void CS(int3 groupThreadID : SV_GroupThreadID,
int3 dispatchThreadID : SV_DispatchThreadID)
{
// Each thread samples the texture and stores the value in shared memory.
gCache[groupThreadID.x] = gInput[dispatchThreadID.xy];
// Wait for all threads in group to finish.
GroupMemoryBarrierWithGroupSync();
// Safe now to read any element in the shared memory and do computation work.
float4 left = gCache[groupThreadID.x - 1];
float4 right = gCache[groupThreadID.x + 1];
…
}
```

Programming

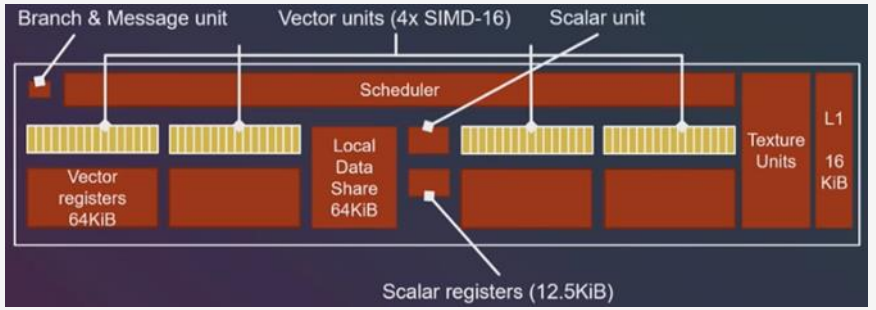
How are instructions executed?

Let's look at an example:

Uniform variables are stored in scalar registers and require 1 X size of uniform variable.

Non-uniform variables are stored in vector registers and require 64 x size of non-uniform variable.

```
groupshared float data[64];

[numthreads(8,8,1)]

void CS(uint index:SV_GroupIndex,

float3 groupIndex: SV_GroupID)

{

    data[index] = (float) (0);

    GroupMemoryBarrierWithGroupSync();

    ...

    //the result of variable a is different per thread

    float a = index + b;  //a is non-uniform variable since it depends on the thread index!

    //Variable c stores the same value across all threads

    int c = groupIndex.x; //c is a uniform variable

    ...

}
```



Execution Mask

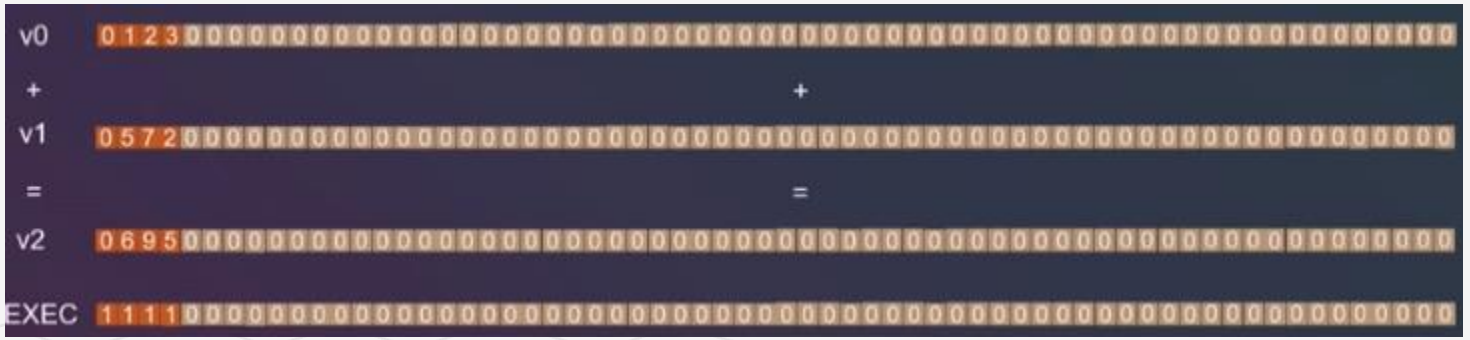
Consider a thread group with 4 threads [numthreads(4,1,1)]

```
void CS(uint index:SV_GroupIndex,

float3 groupIndex: SV_GroupID)

{

    data[index] = (float) (0);

    GroupMemoryBarrierWithGroupSync();

    ...

    //the result of variable a is different per
thread

    float v2 = index + v1;  //a is non-uniform
variable since it depends on the thread index!
```

Instruction: v_add_f32 v2, v0, v1

We still have 64 threads, but we need to activate 4 of them using execution mask.

Remember the SIMD is actually 16-wide. Instruction is executed 4 times with each 6 thread even if EXEC mask is 0!



Blur Postprocessing effect

The blur effect is used for blurring the full screen scene or blurring individual objects in that scene.

In DirectX 12 the real time blur effect is performed by first rendering the scene to a texture, performing the blur on that texture, and then rendering that texture back to the screen.

Whenever we perform 2D image operations on a scene that has been rendered to texture it is called post processing.

To perform any post processing it is generally quite expensive and requires heavy optimization in the shaders.

The Blur Algorithm

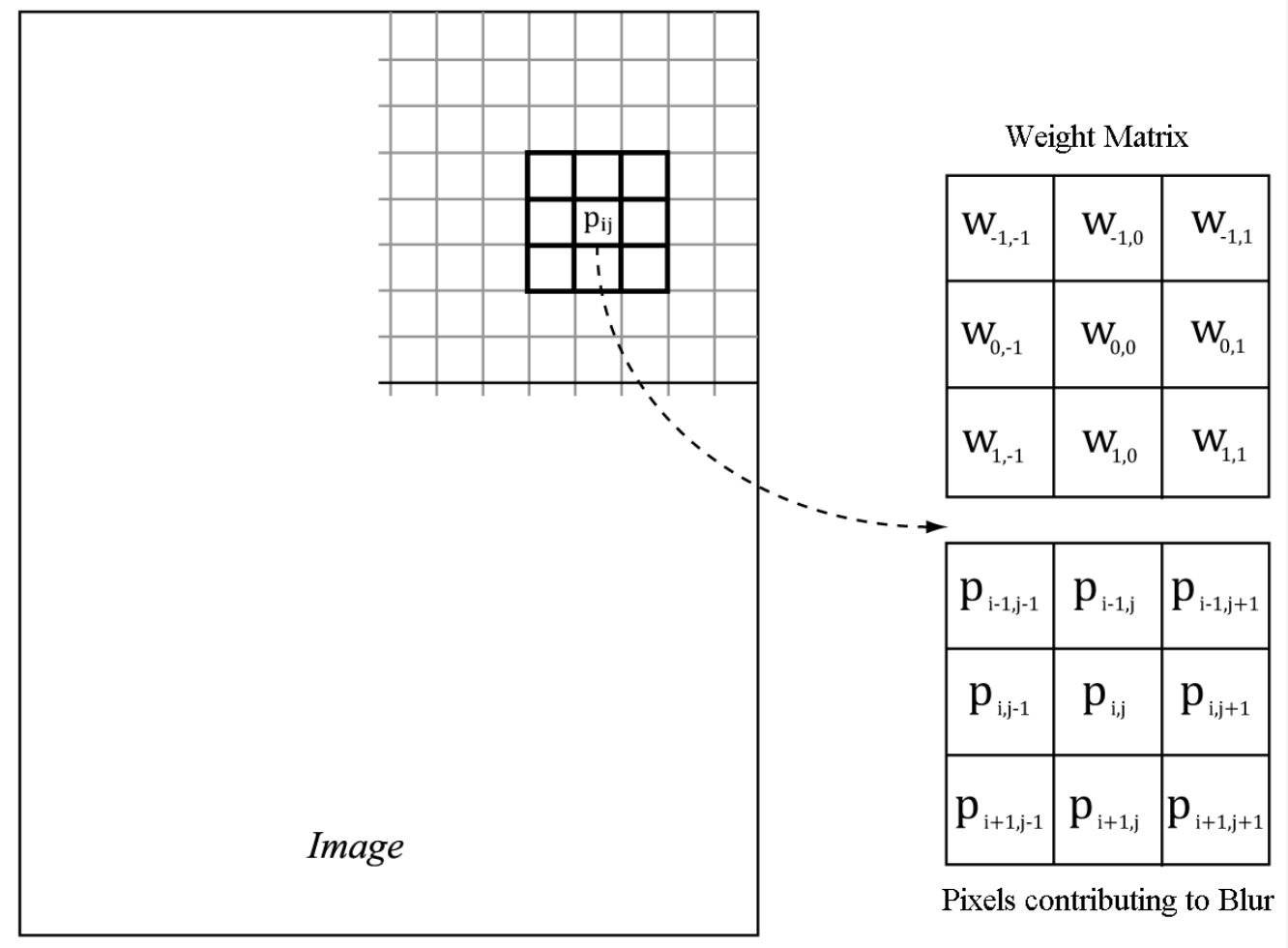
1. Render the scene to texture.
2. Down sample the texture to half of its size or less.
3. Perform a horizontal blur on the down sampled texture.
4. Perform a vertical blur.
5. Up sample the texture back to the original screen size.
6. Render that texture to the screen.

BLUR DEMO

How to implement a blur algorithm on the compute shader?

To blur the pixel $P_{ij,}$ we compute the weighted average of the m × n matrix of pixels centered about the pixel.

In this example, the matrix is a square 3 × 3 matrix, with blur radius $a = b = 1$. Note that the center weight w_{00} aligns with the pixel P_{ij}.



Blurring Theory

The weighted average of the m x n matrix becomes the *ij*th pixel in the blurred image.

We call *a* the vertical blur radius and *b* the horizontal blur radius. If *a* = *b*, then we just refer to the *blur radius* without having to specify the dimension.

m = 2*a* + 1 and *n* = 2*b* + 1.

By forcing *m* and *n* to be odd, we ensure that the *m* × *n* matrix always has a natural "center."

The *m* × *n* matrix of weights is called the *blur kernel*.

Note that the weights must sum to 1.

If the sum of the weights is less than one, the blurred image will appear darker as color has been removed.

If the sum of the weights is greater than one, the blurred image will appear brighter as color has been added.

Mean Blur Kernel		
1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

$$Blur\left(P_{ij}\right) = \sum_{r=-a}^{a}\sum_{c=-b}^{b} w_{rc} P_{i+r,j+c} \quad \text{for} \quad \sum_{r=-a}^{a}\sum_{c=-b}^{b} w_{rc} = 1$$

Image RGB colors per pixel		
255,10,0	20,20,20	10,10,10
255,10,0	20,20,20	50,50,50
0,10,255	200,20,20	10,100,50

Gaussian blur

There are various ways to compute the weights (blur kernel) so long as they sum to 1. What if we want to give more weight to middle pixels!

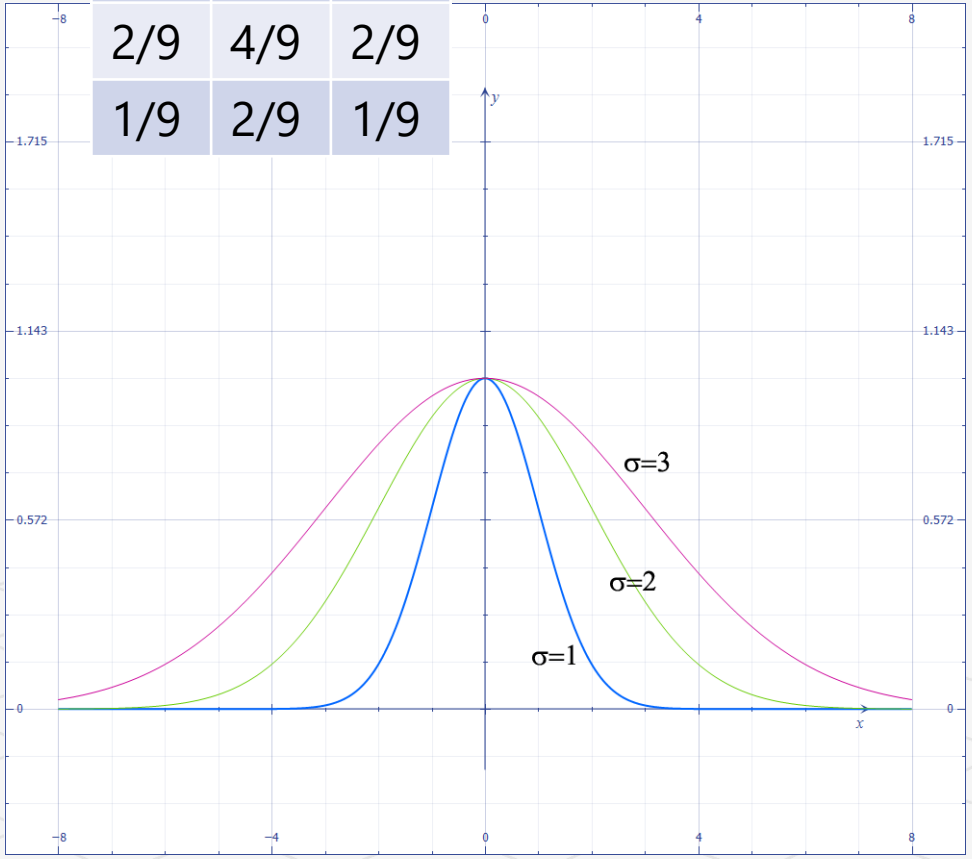
Gaussian Blur Kernel		
1/9	2/9	1/9
2/9	4/9	2/9
1/9	2/9	1/9

In image processing, a **Gaussian blur** (also known as **Gaussian smoothing**) is the result of blurring an image by a Gaussian function (named after mathematician and scientist Carl Friedrich Gauss). It is a widely used effect in graphics software, typically to reduce image noise and reduce detail.

The Gaussian blur obtains its weights from the Gaussian function:

$$G(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

A graph of this function is shown for different σ (standard deviation).



Plot of $G(x)$ for σ = 1, 2, 3. Observe that a larger σ flattens the curve out and gives more weight to the neighboring points.

σ represents the blur radius!

Gaussian blur

Let us suppose we are doing a 1 × 5 Gaussian blur (i.e., a 1D blur in the horizontal direction), and let σ = 1.

Evaluating G(x) for x = −2, −1, 0, 1, 2 we have:

$$G(-2) = \exp\left(-\frac{(-2)^2}{2}\right) = e^{-2}$$

$$G(-1) = \exp\left(-\frac{(-1)^2}{2}\right) = e^{-\frac{1}{2}}$$

$$G(0) = \exp(0) = 1$$

$$G(1) = \exp\left(-\frac{1^2}{2}\right) = e^{-\frac{1}{2}}$$

$$G(2) = \exp\left(-\frac{2^2}{2}\right) = e^{-2}$$

$$\sum_{x=-2}^{x=2} G(x) = G(-2) + G(-1) + G(0) + G(1) + G(2)$$

$$= 1 + 2e^{-\frac{1}{2}} + 2e^{-2}$$

$$\approx 2.48373$$

$$\frac{G(-2) + G(-1) + G(0) + G(1) + G(2)}{\sum_{x=-2}^{x=2} G(x)} = 1$$

Gaussian blur

G(-2), G(-1), G(0), G(1), and G(2) are not the weights because they do not sum to 1. If we normalize them by dividing them by the sum of them, then we obtain weights based on the Gaussian function that sum to 1.

Therefore, the Gaussian blur weights are:

$$w_{-2} = \frac{G(-2)}{\sum_{x=-2}^{x=2} G(x)} = \frac{e^{-2}}{1+2e^{-\frac{1}{2}}+2e^{-2}} \approx 0.0545$$

$$w_{-1} = \frac{G(-1)}{\sum_{x=-2}^{x=2} G(x)} = \frac{e^{-\frac{1}{2}}}{1+2e^{-\frac{1}{2}}+2e^{-2}} \approx 0.2442$$

$$w_{0} = \frac{G(0)}{\sum_{x=-2}^{x=2} G(x)} = \frac{1}{1+2e^{-\frac{1}{2}}+2e^{-2}} \approx 0.4026$$

$$w_{1} = \frac{G(1)}{\sum_{x=-2}^{x=2} G(x)} = \frac{e^{-\frac{1}{2}}}{1+2e^{-\frac{1}{2}}+2e^{-2}} \approx 0.2442$$

$$w_{2} = \frac{G(2)}{\sum_{x=-2}^{x=2} G(x)} = \frac{e^{-2}}{1+2e^{-\frac{1}{2}}+2e^{-2}} \approx 0.0545$$

Two Linear Passes

The method we are going to use for blurring is to take a weighted average all the neighbor pixels around each pixel to determine the value the current pixel should be.

Already you can tell this is going to be fairly expensive to perform but we have a way of reducing the computational complexity by doing it in two linear passes.

We first do one horizontal pass and one then vertical instead of doing a single circular neighborhood pass.

Separating each render to texture also allows you to display the results of each blur pass on the screen for debugging purpose.

To understand the difference in the speed between the two different pass methods take for example just a 100x100 pixel image.

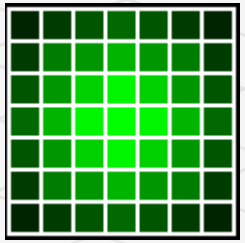
Two linear passes on a 100x100 image requires reading 100 + 100 = 200 pixels. Doing a single circular pass requires reading 100 * 100 = 10,000 pixels.

The first linear pass is going to be a horizontal blur. For example we will take a single pixel, then we will perform a weighted blur of its 3 closest horizontal neighbors to produce something similar to the following for each pixel.

We do this for the entire down sampled texture.

The resulting horizontally blurred image is then rendered to a second render to texture I call the HorizontalBlurTexture.

The vertical works exactly the same way as the horizontal blur except that it goes vertically and uses the horizontal blur render to texture as input instead of the original texture. The vertical blur is also rendered to another new render to texture object I call the VerticalBlurTexture.



Gaussian blur

The Gaussian blur is known to be separable, which means it can be broken up into two 1D blurs:

1. Blur the input image I using a 1D horizontal blur: $I_H = Blur_H(I)$.

2. Blur the output from the previous step using a 1D vertical blur:

$Blur(I) = \text{Blur}_V(I_H)$.

Therefore:

$$\boldsymbol{Blur(I) = \text{Blur}_V(Blur_H(I))}$$

Suppose that the blur kernel is a 9 × 9 matrix:

- We needed a total of 81 samples to do the 2D blur
- By separating the blur into two 1D blurs, we only need 9 + 9 = 18 samples

Typically, we will be blurring textures

Fetching texture samples is expensive, so reducing texture samples by separating a blur is a welcome improvement.

Even if a blur is not separable (some blur operators are not), we can often make the simplification and assume it is for the sake of performance, as long as the final image looks accurate enough.

Render-to-Texture

Remember that the back buffer is just a texture in the swap chain:

```
Microsoft::WRL::ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(mRtvHeap->GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
ThrowIfFailed(mSwapChain->GetBuffer(i, IID_PPV_ARGS(&mSwapChainBuffer[i])));
md3dDevice->CreateRenderTargetView(mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);
rtvHeapHandle.Offset(1, mRtvDescriptorSize);
}
```

We instruct Direct3D to render to the back buffer by binding a render target view of the back buffer to the OM stage of the rendering pipeline:

```
mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true, &DepthStencilView());
```

The contents of the back buffer are eventually displayed on the screen when the back buffer is presented via the IDXGISwapChain::Present method.

A texture that will be used as a render target must be created with the flag D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET.

Create another texture, create a render target view to it, and bind it to the OM stage of the rendering pipeline.

Therefore, we will be drawing to this different “off-screen” texture instead of the back buffer.

This technique is known as *render-to-off-screen-texture* or simply *render-to-texture*. The only difference is that since this texture is not the back buffer, it does not get displayed to the screen during presentation.

Render-to-Texture

So far in our programs, we have been rendering to the back buffer.

The back buffer is just a texture in the swap chain.

We instruct Direct3D to render to the back buffer by binding a render target view of the back buffer to the OM stage of the rendering pipeline.

The contents of the back buffer are eventually displayed on the screen when the back buffer is presented via the IDXGISwapChain::Present method.

```
static const int SwapChainBufferCount = 2;
int mCurrBackBuffer = 0;
Microsoft::WRL::ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];

D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::CurrentBackBufferView()const
{
return CD3DX12_CPU_DESCRIPTOR_HANDLE(
mRtvHeap->GetCPUDescriptorHandleForHeapStart(),
mCurrBackBuffer,
mRtvDescriptorSize);
}

CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(mRtvHeap->GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
ThrowIfFailed(mSwapChain->GetBuffer(i, IID_PPV_ARGS(&mSwapChainBuffer[i])));
md3dDevice->CreateRenderTargetView(mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);
rtvHeapHandle.Offset(1, mRtvDescriptorSize);
}

// Specify the buffers we are going to render to.
    mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true, &DepthStencilView());

 // Swap the back and front buffers

ThrowIfFailed(mSwapChain->Present(0, 0));

mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;
```

Render-to-Texture

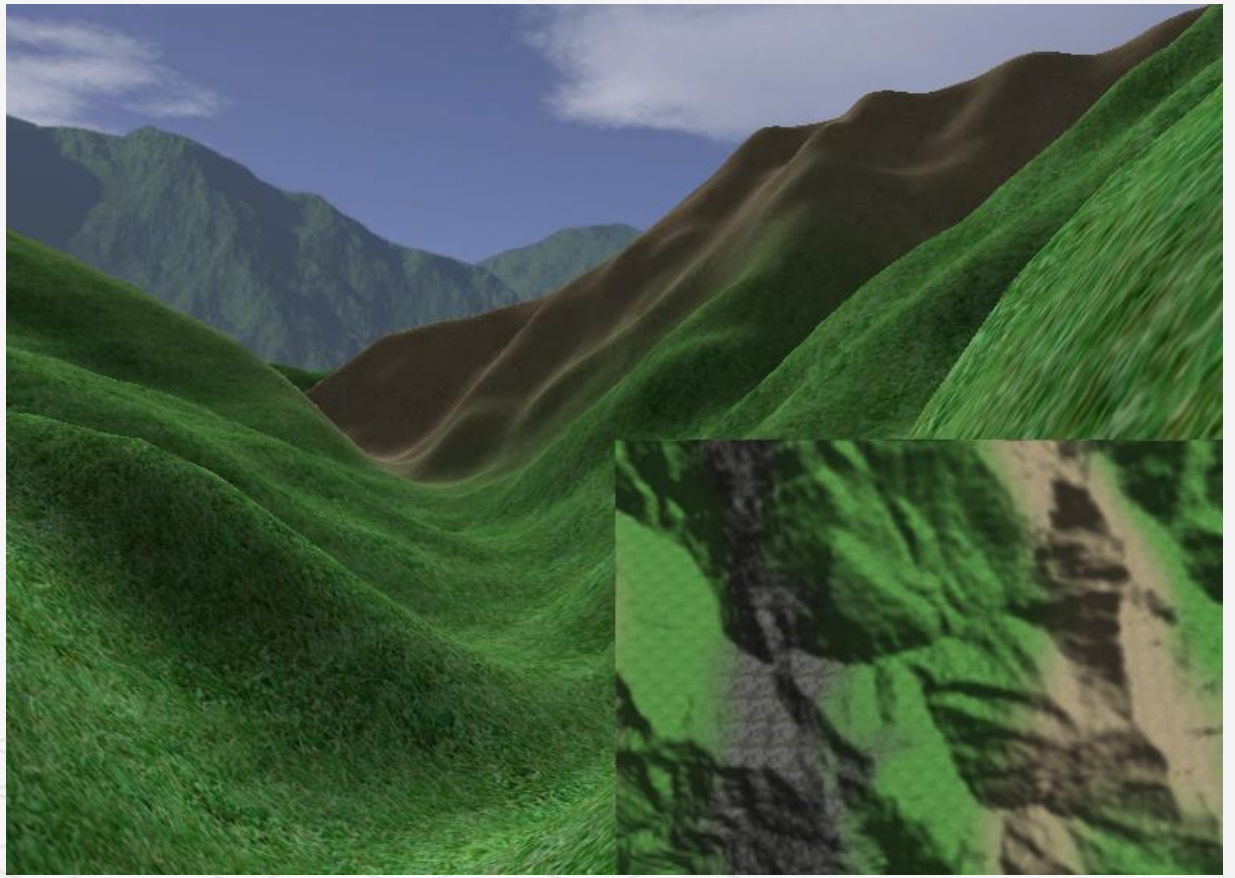
We can create another texture and create a render target view to it, and bind it to the OM stage of the rendering pipeline. Then we will be drawing to this different “off-screen” texture (possible with a different camera) instead of the back buffer.

A texture that will be used as a render target must be created with a flag.

```
D3D12_RESOURCE_DESC texDesc;

texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;
```

Look at SobelFilter example (RenderTarget.cpp)

A camera is placed above the player from a bird’s eye view (an elevated view of an object from above) and renders the scene into an off-screen texture. When we draw the scene from the player’s eye to the back buffer, we map the texture onto a quad in the bottom-right corner of the screen to display the radar map.



Other render-to-texture techniques include:

1. Shadow mapping

2. Screen Space Ambient Occlusion

3. Dynamic reflections with cube maps

Blur implementation using Render to Texture

Using render-to-texture, implementing a blurring algorithm on the GPU would work the following way:

1. Draw scene as usual to an off-screen texture.

2. Blur the off-screen texture using a compute shader program.

3. Restore the back buffer as the render target, and draw a full screen quad with the blurred texture applied.

Assuming that our off-screen textures match the format and size of our back buffer, instead of redirecting rendering to our off-screen texture, we can render to the back buffer as usual, and then do a CopyResource to copy the back-buffer contents to our off-screen texture.

Then we can do our compute work on our off-screen textures, and then draw a full-screen quad to the back buffer with the blurred texture to produce the final screen output.

```
// Copy the input (back-buffer) to BlurMap0 (Texture)

BlurMap0.cmdList->CopyResource(mBlurMap0.Get(), input);
```

The above process requires us:

1. To draw with the usual rendering pipeline,

2. Switch to the compute shader and do compute work,

3. And finally switch back to the usual rendering pipeline.

Blur Implementation Overview

We assume that the blur is separable, so we break the blur down into computing two 1D blurs—a horizontal one and a vertical one.

Implementing this requires two texture buffers where we can read and write to both;

We need a SRV and UAV to both textures.

Let us call one of the textures **A** and the other texture **B**. The blurring algorithm proceeds as follows:

1. Bind the SRV to **A** as an input to the compute shader (this is the input image that will be horizontally blurred).

2. Bind the UAV to **B** as an output to the compute shader (this is the output image that will store the blurred result).

3. Dispatch the thread groups to perform the horizontal blur operation. After this, texture **B** stores the horizontally blurred result $Blur_H(I)$, where I is the image to blur.

4. Bind the SRV to **B** as an input to the compute shader (this is the horizontally blurred image that will next be vertically blurred).

5. Bind the UAV to **A** as an output to the compute shader (this is the output image that will store the final blurred result).

6. Dispatch the thread groups to perform the vertical blur operation. After this, texture **A** stores the final blurred result $Blur(I)$, where I is the image to blur.

OnResize

The texture we render the scene to has the same resolution as the window client area.

We need to rebuild the off-screen texture, as well as the second texture buffer **B** used in the blur algorithm.

We do this on the OnResize method:

```
void BlurApp::OnResize()
{
D3DApp::OnResize();
XMMATRIX P =
XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi,
AspectRatio(), 1.0f, 1000.0f);
    XMStoreFloat4x4(&mProj, P);

if(mBlurFilter != nullptr)
{
mBlurFilter->OnResize(mClientWidth, mClientHeight);
}
}
```

The `mBlurFilter` variable is an instance of a BlurFilter helper class we make.

BlurFilter helper class

1. Encapsulates the texture resources to textures **A** and **B**

2. Encapsulates SRVs and UAVs to the textures,

3. Provides a method that kicks off the actual blur operation on the compute shader.

```
void BlurFilter::OnResize(UINT newWidth, UINT newHeight)
{
if((mWidth != newWidth) || (mHeight != newHeight))
{
mWidth = newWidth;
mHeight = newHeight;

BuildResources();

// New resource, so we need new descriptors (SRV, UAV) to that resource.
BuildDescriptors();
}
}
```

BlurFilter::BuildDescriptors(…)

The BlurFilter class encapsulates texture resources. To bind these resources to the pipeline to use for a draw/dispatch command, we are going to need to create descriptors to these resources.

we will have to allocate extra room in the D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV descriptor heap to store these descriptors.

The BlurFilter::BuildDescriptors method takes descriptor handles to the starting location in the descriptor heap to store the descriptors used by BlurFilter.

The method caches the handles for all the descriptors it needs and then creates the corresponding descriptors.

The reason it caches the handles is so that it can recreate the descriptors when the resources change, which happens when the screen is resized:

```
void BlurFilter::BuildDescriptors(CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuDescriptor,
                                  CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuDescriptor,
                                  UINT descriptorSize)
{
// Save references to the descriptors.
mBlur0CpuSrv = hCpuDescriptor;
mBlur0CpuUav = hCpuDescriptor.Offset(1, descriptorSize);
mBlur1CpuSrv = hCpuDescriptor.Offset(1, descriptorSize);
mBlur1CpuUav = hCpuDescriptor.Offset(1, descriptorSize);

mBlur0GpuSrv = hGpuDescriptor;
mBlur0GpuUav = hGpuDescriptor.Offset(1, descriptorSize);
mBlur1GpuSrv = hGpuDescriptor.Offset(1, descriptorSize);
mBlur1GpuUav = hGpuDescriptor.Offset(1, descriptorSize);

BuildDescriptors();
}
```

BlurFilter::BuildResources

1. We copy the back-buffer to mBlurMap0.
2. Horizontal Blur Pass on mBlurMap0 pixels
3. Result goes to mBlurMap1
4. Vertical Blur Pass on mBlurMap1
5. Result goes to mBlurMap0

```
void BlurFilter::BuildResources()
{
D3D12_RESOURCE_DESC texDesc;
ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texDesc.Alignment = 0;
texDesc.Width = mWidth;
texDesc.Height = mHeight;
texDesc.DepthOrArraySize = 1;
texDesc.MipLevels = 1;
texDesc.Format = mFormat;
texDesc.SampleDesc.Count = 1;
texDesc.SampleDesc.Quality = 0;
texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS;

ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE,
&texDesc,
D3D12_RESOURCE_STATE_COMMON,
nullptr,
IID_PPV_ARGS(&mBlurMap0)));

ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE,
&texDesc,
D3D12_RESOURCE_STATE_COMMON,
nullptr,
IID_PPV_ARGS(&mBlurMap1)));
}
```

BlurFilter::BuildDescriptors()

```
void BlurFilter::BuildDescriptors()
{
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = mFormat;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = 1;

D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};

uavDesc.Format = mFormat;
uavDesc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;

md3dDevice->CreateShaderResourceView(mBlurMap0.Get(), &srvDesc, mBlur0CpuSrv);
md3dDevice->CreateUnorderedAccessView(mBlurMap0.Get(), nullptr, &uavDesc, mBlur0CpuUav);

md3dDevice->CreateShaderResourceView(mBlurMap1.Get(), &srvDesc, mBlur1CpuSrv);
md3dDevice->CreateUnorderedAccessView(mBlurMap1.Get(), nullptr, &uavDesc, mBlur1CpuUav);
}
```

Horizontal vs. Vertical Blur Pass

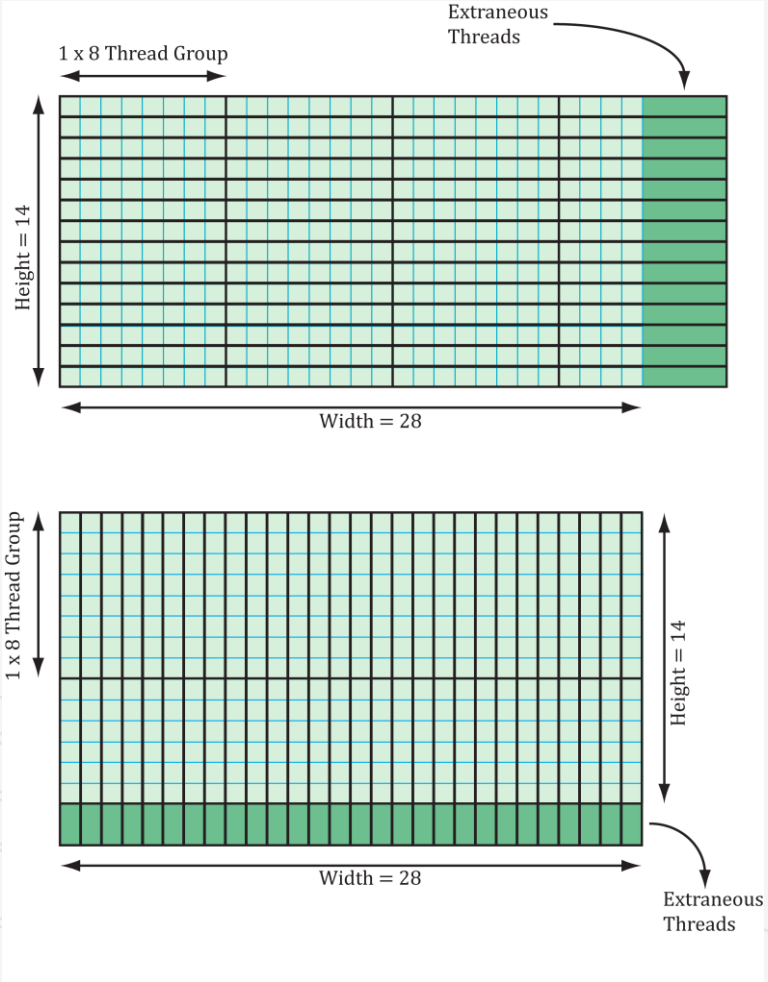
Suppose our image has width *w* and height *h*. Consider a 28 × 14 texture as an example, , where our horizontal thread groups are 8 × 1 and our vertical thread groups are 1 × 8 (*X* × *Y* format).

when we look at the compute shader, for the horizontal 1D blur, our thread group is a horizontal line segment of 8 threads, and each thread is responsible for blurring one pixel in the image.

Therefore, we need to dispatch $ceil\left(\frac{w}{8}\right)$=4 thread groups in the *x*-direction and *h(14)* thread groups in the *y*-direction in order for each pixel in the image to be blurred.

If 8 does not divide evenly into *w*, the last horizontal thread group will have extraneous threads! We take care of out-of-bounds with clamping checks in the shader code.

For the vertical pass, in order to cover all the pixels we need to dispatch $ceil\left(\frac{h}{8}\right)$=2 thread groups in the *y*-direction and *w(28)* thread groups in the *x*-direction in order for each pixel in the image to be blurred.

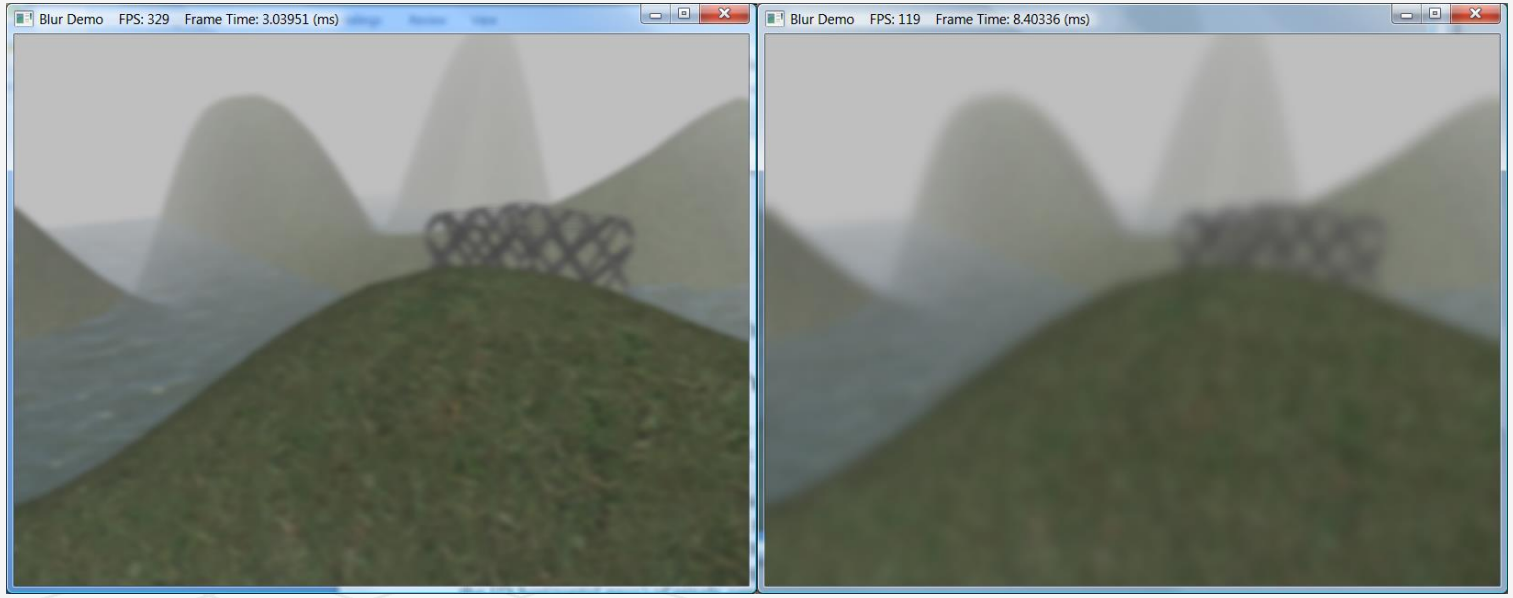


Blue Demo

Left: A screenshot of the “Blur” demo where the image has been blurred two times.

Right: A screen shot of the “Blur” demo where the image has been blurred eight times.

Our thread group is a horizontal line segment of 256 threads, and each thread is responsible for blurring one pixel in the image.



BlurFilter::Execute

The code below figures out how many thread groups to dispatch in each direction, and kicks off the actual blur operation on the compute shader:

```
void BlurFilter::Execute(ID3D12GraphicsCommandList* cmdList,
                         ID3D12RootSignature* rootSig,
                         ID3D12PipelineState* horzBlurPSO,
                         ID3D12PipelineState* vertBlurPSO,
                             ID3D12Resource* input,
 int blurCount)
{
auto weights = CalcGaussWeights(2.5f);
int blurRadius = (int)weights.size() / 2;

cmdList->SetComputeRootSignature(rootSig);

cmdList->SetComputeRoot32BitConstants(0, 1, &blurRadius, 0);
cmdList->SetComputeRoot32BitConstants(0, (UINT)weights.size(), weights.data(), 1);

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(input,
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_COPY_SOURCE));

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_COPY_DEST));

// Copy the input (back-buffer in this example) to BlurMap0.
cmdList->CopyResource(mBlurMap0.Get(), input);

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0.Get(),
D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_GENERIC_READ));

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
```

BlurFilter::Execute

```
for(int i = 0; i < blurCount; ++i)
{
// Horizontal Blur pass.

cmdList->SetPipelineState(horzBlurPSO);

cmdList->SetComputeRootDescriptorTable(1, mBlur0GpuSrv);
cmdList->SetComputeRootDescriptorTable(2, mBlur1GpuUav);

// How many groups do we need to dispatch to cover a row of pixels, where each
// group covers 256 pixels (the 256 is defined in the ComputeShader).
UINT numGroupsX = (UINT)ceilf(mWidth / 256.0f);
cmdList->Dispatch(numGroupsX, mHeight, 1);

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0.Get(),
D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1.Get(),
D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_GENERIC_READ));

// Vertical Blur pass.

cmdList->SetPipelineState(vertBlurPSO);

cmdList->SetComputeRootDescriptorTable(1, mBlur1GpuSrv);
cmdList->SetComputeRootDescriptorTable(2, mBlur0GpuUav);

// How many groups do we need to dispatch to cover a column of pixels, where each
// group covers 256 pixels  (the 256 is defined in the ComputeShader).
UINT numGroupsY = (UINT)ceilf(mHeight / 256.0f);
cmdList->Dispatch(mWidth, numGroupsY, 1);

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap0.Get(),
D3D12_RESOURCE_STATE_UNORDERED_ACCESS, D3D12_RESOURCE_STATE_GENERIC_READ));

cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mBlurMap1.Get(),
D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
}
}
```

Blur algorithm

Our thread group is a horizontal line segment of 256 threads, and each thread is responsible for blurring one pixel in the image.

What happens if we just implement blur algorithm directly?

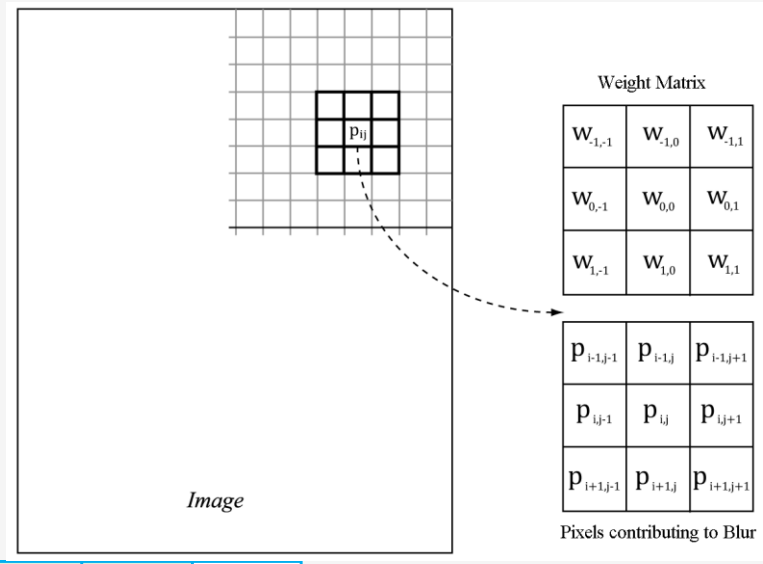
Each thread simply performs the weighted average of the row matrix (row matrix because we are doing the 1D horizontal pass) of pixels centered about the pixel the thread is processing.

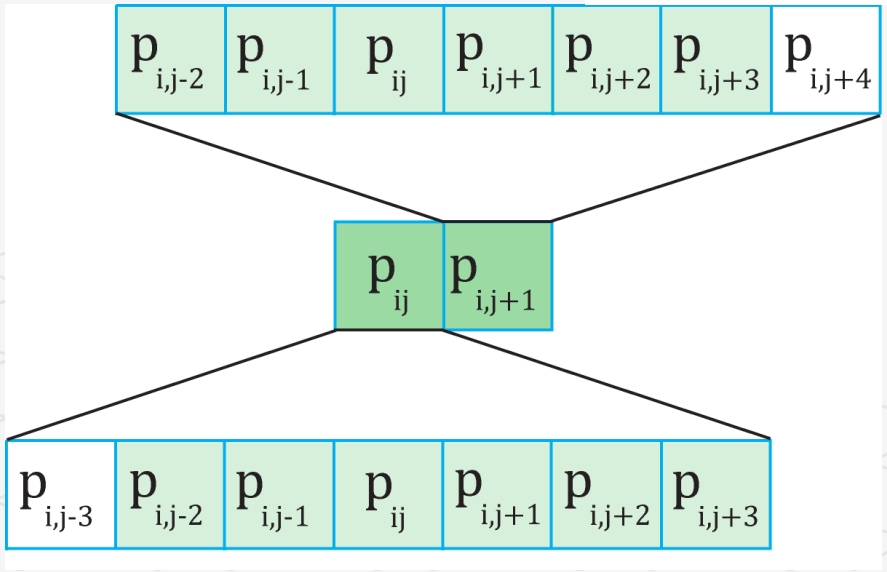
The problem with this approach is that it requires fetching the same texel multiple times.

Consider just two neighboring pixels in the input image, and suppose that the blur kernel is 1 × 7.

Observe that six out of the eight unique pixels are sampled twice—once for each pixel.

We can optimize our blur algorithm by taking advantage of shared memory.





Optimizing Blur Algorithm

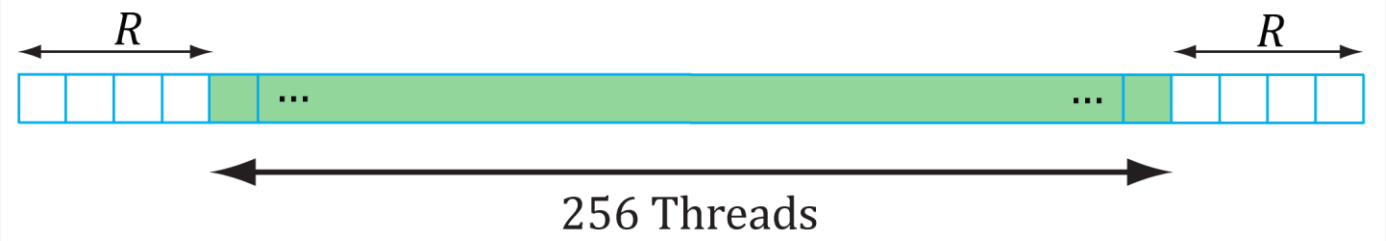
Each thread can read in a texel value and store it in shared memory.

After all the threads are done reading their texel values into shared memory, the threads can proceed to perform the blur by reading the texels from the shared memory, which is fast to access.

The only tricky thing about this is that a thread group of $n = 256$ threads requires $n + 2R$ texels to perform the blur, where R is the blur radius.

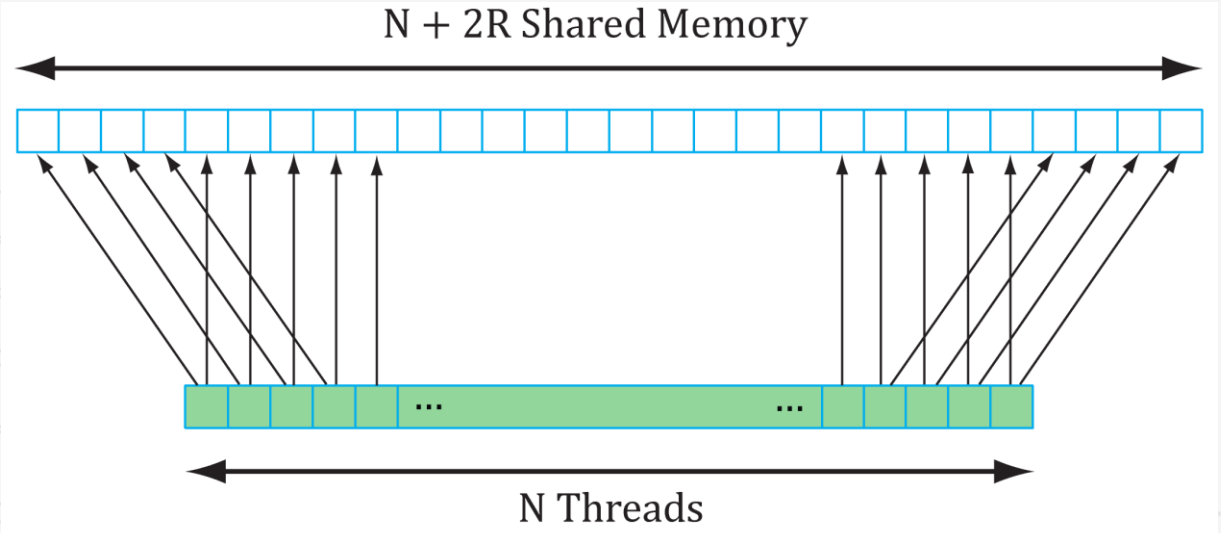
Pixels near the boundaries of the thread group will read pixels outside the thread group due to the blur radius.

The solution is simple; we allocate $n + 2R$ elements of shared memory, and have $2R$ threads lookup two texel values.



In this example, $R = 4$. The four leftmost threads each read two texel values and store them into shared memory. The four rightmost threads each read two texel values and store them into shared memory. Every other thread just reads one texel value and stores it in shared memory. This gives us all the texel values we need to blur N pixels with blur radius R.

The only thing that is tricky about this is that it requires a little more book keeping when indexing into the shared memory; we no longer have the ith group thread ID corresponding to the ith element in the shared memory.



Sampling outside the image

Situations where we can read outside the bounds of the image.

The left-most thread group and the rightmost thread group can index the input image out-of-bounds.

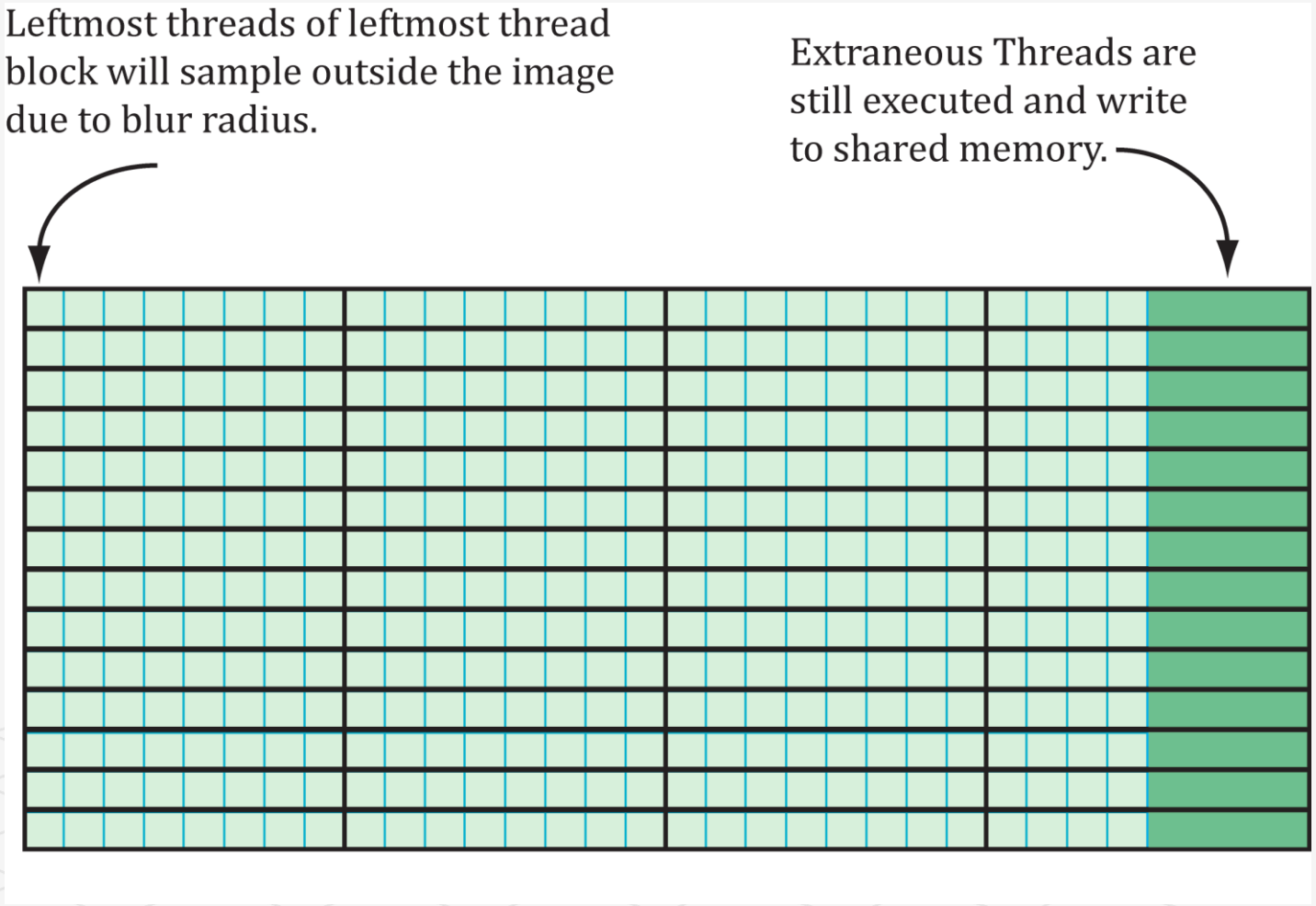
Reading from an out-of-bounds index is not illegal—it is defined to return 0 (and writing to an out-of-bounds index results in a no-op).

However, we do not want to read 0 when we go out-of-bounds, as it means 0 colors (i.e., black) will make their way into the blur at the boundaries.

It is possible in the right-most thread group to have extraneous threads that do not correspond to an element in the output texture.

That is, the dispatchThreadID.xy will be an out-of-bounds index for the output texture.

However, we do not need to worry about handling this case, as an out-of-bound write results in a no-op.



Clamping the out-of-bounds indices

we want to implement something analogous to the *clamp* texture address mode, where if we read an out-of-bounds value, it returns the same value as the boundary texel.

This can be implemented by clamping the indices:

```
// Performs a separable Guassian blur with a blur radius up
to 5 pixels.

cbuffer cbSettings : register b0
{
// We cannot have an array entry in a constant buffer that
gets mapped onto root constants, so list each element.

int gBlurRadius;

// Support up to 11 blur weights.
float w0;
float w1;
float w2;
float w3;
float w4;
float w5;
float w6;
float w7;
float w8;
float w9;
float w10;
};
```

```
static const int gMaxBlurRadius = 5;

Texture2D gInput            : register t0;
RWTexture2D<float4> gOutput : register u0;

#define N 256
#define CacheSize  N + 2*gMaxBlurRadius
groupshared float4 gCache[CacheSize];

[numthreads(N, 1, 1)]

void HorzBlurCS(int3 groupThreadID : SV_GroupThreadID,

int3 dispatchThreadID : SV_DispatchThreadID)
{
// Put in an array for each indexing.
float weights 11 =  { w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10 };

// 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)];

// Wait for all threads to finish.
GroupMemoryBarrierWithGroupSync();
```

Blur.hlsl

```
// Now blur each pixel.

float4 blurColor = float4(0, 0, 0, 0);

for(int i = -gBlurRadius; i <= gBlurRadius;
++i)
{
int k = groupThreadID.x + gBlurRadius + i;

blurColor += weights[i+gBlurRadius]*gCache[k];
}

gOutput[dispatchThreadID.xy] = blurColor;
}
```

```
[numthreads(1, N, 1)]
void VertBlurCS(int3 groupThreadID : SV_GroupThreadID,
int3 dispatchThreadID : SV_DispatchThreadID)
{
// Put in an array for each indexing.
float weights[11] = { w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10 };

if(groupThreadID.y < gBlurRadius)
{
// Clamp out of bound samples that occur at image borders.
int y = max(dispatchThreadID.y - gBlurRadius, 0);
gCache[groupThreadID.y] = gInput[int2(dispatchThreadID.x, y)];
}
if(groupThreadID.y >= N-gBlurRadius)
{
// Clamp out of bound samples that occur at image borders.
int y = min(dispatchThreadID.y + gBlurRadius, gInput.Length.y-1);
gCache[groupThreadID.y+2*gBlurRadius] = gInput[int2(dispatchThreadID.x, y)];
}

// Clamp out of bound samples that occur at image borders.
gCache[groupThreadID.y+gBlurRadius] = gInput[min(dispatchThreadID.xy, gInput.Length.xy-1)];


// Wait for all threads to finish.
GroupMemoryBarrierWithGroupSync();

//
// Now blur each pixel.
//

float4 blurColor = float4 0  0  0  0);

for(int i = -gBlurRadius; i <= gBlurRadius; ++i)
{
int k = groupThreadID.y + gBlurRadius + i;

blurColor += weights[i+gBlurRadius]*gCache[k];
}

gOutput[dispatchThreadID.xy] = blurColor;
}
```

Sobel Filter

Sobel–Feldman operator or **Sobel filter** creates an image emphasising edges.

It is named after Irwin Sobel and Gary Feldman, colleagues at the Stanford Artificial Intelligence Laboratory (SAIL).

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical.

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel.

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. The matrix operation being performed—convolution—is not traditional matrix multiplication, despite being similarly denoted by "*".

For example, if we have two three-by-three matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and then multiplying locally similar entries and summing. The element at coordinates [2, 2] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

Sobel Operator

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical.

If we define **A** as the source image, and $\mathbf{G}_x$ and $\mathbf{G}_y$ are two images which at each point contain the vertical and horizontal derivative approximations respectively, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

where * here denotes the 2-dimensional signal processing convolution operation.

Since the Sobel kernels can be decomposed as the products of an averaging and a differentiation kernel, they compute the gradient with smoothing. For example, G_x can be written as:

$$\mathbf{G}_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \left(\begin{bmatrix} -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \right) \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \mathbf{A} \right)$$

The *x*-coordinate is defined here as increasing in the "right"-direction, and the *y*-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Sobel.hlsl

```
// Performs edge detection using Sobel
operator.

Texture2D gInput : register(t0);

RWTexture2D<float4> gOutput : register(u0);

// Approximates luminance ("brightness") from
an RGB value.  These weights are derived from
experiment based on eye sensitivity to
different wavelengths of light.

float CalcLuminance(float3 color)

{

return dot(color, float3(0.299f, 0.587f,
0.114f));

}

[numthreads(16, 16, 1)]
void SobelCS(int3 dispatchThreadID : SV_DispatchThreadID)
{
// Sample the pixels in the neighborhood of this pixel.
float4 c[3][3];
for (int i = 0; i < 3; ++i)
{
for (int j = 0; j < 3; ++j)
{
int2 xy = dispatchThreadID.xy + int2(-1 + j, -1 + i);
c[i][j] = gInput[xy];
}
}

// For each color channel, estimate partial x derivative using Sobel scheme.
float4 Gx = -1.0f * c[0][0] - 2.0f * c[1][0] - 1.0f * c[2][0] + 1.0f * c[0][2] + 2.0f *
c[1][2] + 1.0f * c[2][2];

// For each color channel, estimate partial y derivative using Sobel scheme.
float4 Gy = -1.0f * c[2][0] - 2.0f * c[2][1] - 1.0f * c[2][1] + 1.0f * c[0][0] + 2.0f *
c[0][1] + 1.0f * c[0][2];

// Gradient is (Gx, Gy).  For each color channel, compute magnitude to get maximum rate of
change.
float4 mag = sqrt(Gx * Gx + Gy * Gy);

// Make edges black, and nonedges white.
mag = 1.0f - saturate(CalcLuminance(mag.rgb));

// Try this to make edges white, and nonedges black
//mag = saturate(CalcLuminance(mag.rgb));

gOutput[dispatchThreadID.xy] = mag;
}
```

Sobel Filter demo

The Sobel Operator measures edges in an image.

For each pixel, it estimates the magnitude of the gradient.

A pixel with a large gradient magnitude means the color difference between the pixel and its neighbors has high variation, and so that pixel must be on an edge.

A pixel with a small gradient magnitude means the color difference between the pixel and its neighbors has low variation, and so that pixel is not on an edge.

Note that the Sobel Operator does not return a binary result(on edge or not on edge); it returns a grayscale value in the range[0, 1] that denotes an edge “steepness” amount, where 0 denotes no edge(the color is not changing locally about a pixel) and 1 denotes a very steep edge or discontinuity(the color is changing a lot locally about a pixel).

The inverse Sobel image(1 − c) is often more useful, as white denotes no edge and black denotes edges.