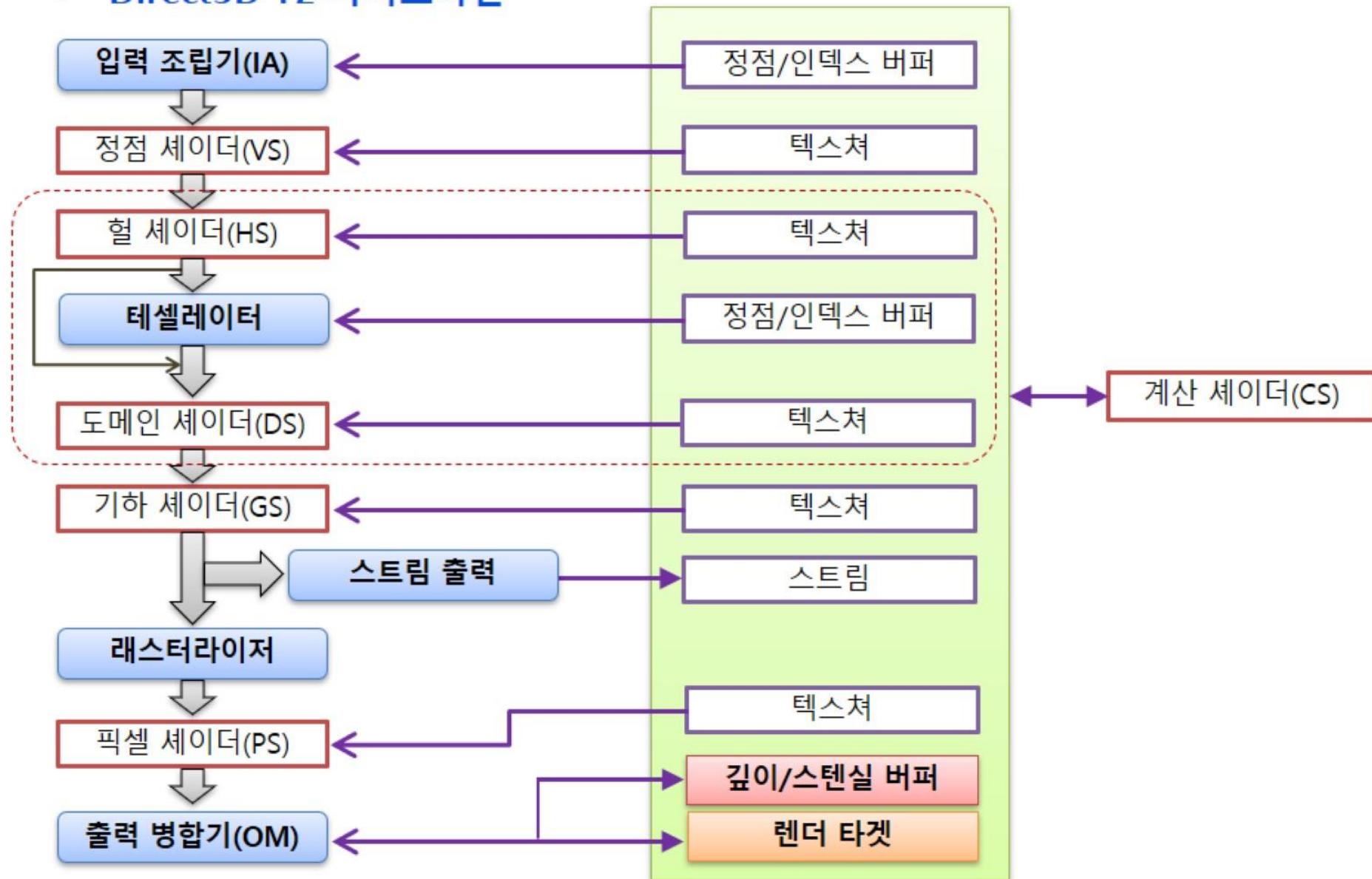


Game Programming with DirectX

Direct3D Graphics Pipeline (Compute Shader)

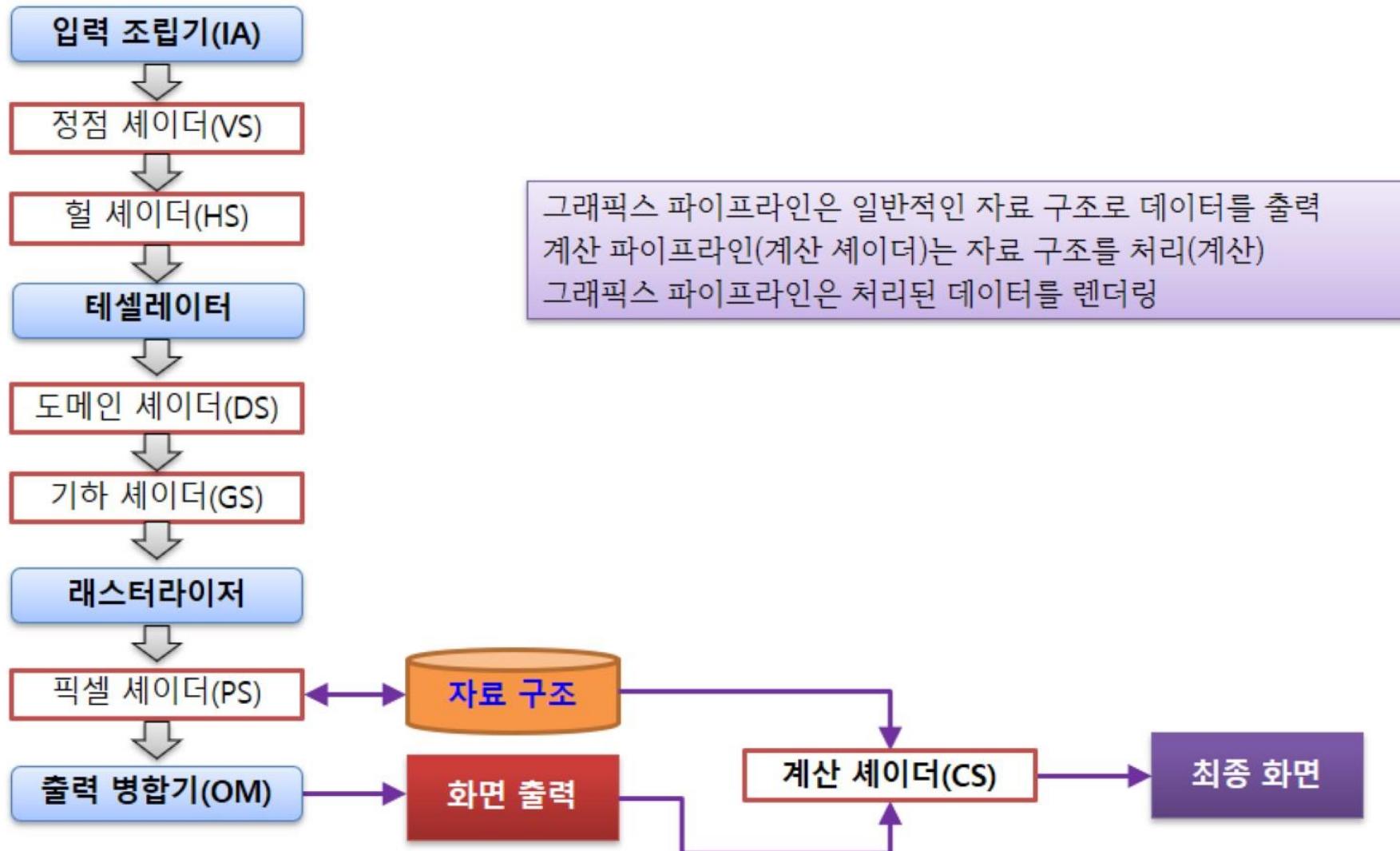
Direct3D 파이프라인

- Direct3D 12 파이프라인



Direct3D 파이프라인

- Direct3D 12 파이프라인



Direct3D 파이프라인

- 간단한 계산 셰이더(Compute Shader)

- 2개의 텍스쳐를 더하여 새로운 텍스쳐를 생성한다고 가정
- 응용 프로그램에서 또는 셰이더 프로그램에서 루프(Loop) 사용(?)

```
ID3D12Resource *pd3dTextureA, *pd3dTextureB, *pd3dTextureC;  
void *pd3dMappedA, *pd3dMappedB, *pd3dMappedC;  
pd3dTextureA->Map(0, NULL, 0, (void **) &d3dMappedA);  
pd3dTextureB->Map(0, NULL, 0, (void **) &d3dMappedB);  
pd3dTextureC->Map(0, NULL, 0, (void **) &d3dMappedC);  
... //텍스쳐 더하기  
pd3dTextureA->Unmap(0, NULL);  
pd3dTextureB->Unmap(0, NULL);  
pd3dTextureC->Unmap(0, NULL);
```

```
Texture2D gtxtInputA: register(t0);  
Texture2D gtxtInputB: register(t1);  
Texture2D gtxtOutput: register(t2); //읽기 전용
```

```
void AddTextures() {  
    for (i = 0; i < gtxHeight; i++)  
        for (j = 0; j < gtxWidth; j++)  
            gtxtOutput[i][j] = gtxtInputA[i][j] + gtxtInputB[i][j];  
}
```

```
Texture2D gtxtInputA : register(t0);  
Texture2D gtxtInputB : register(t1);  
RWTexture2D<float4> gtxtRWOutput : register(u3);
```

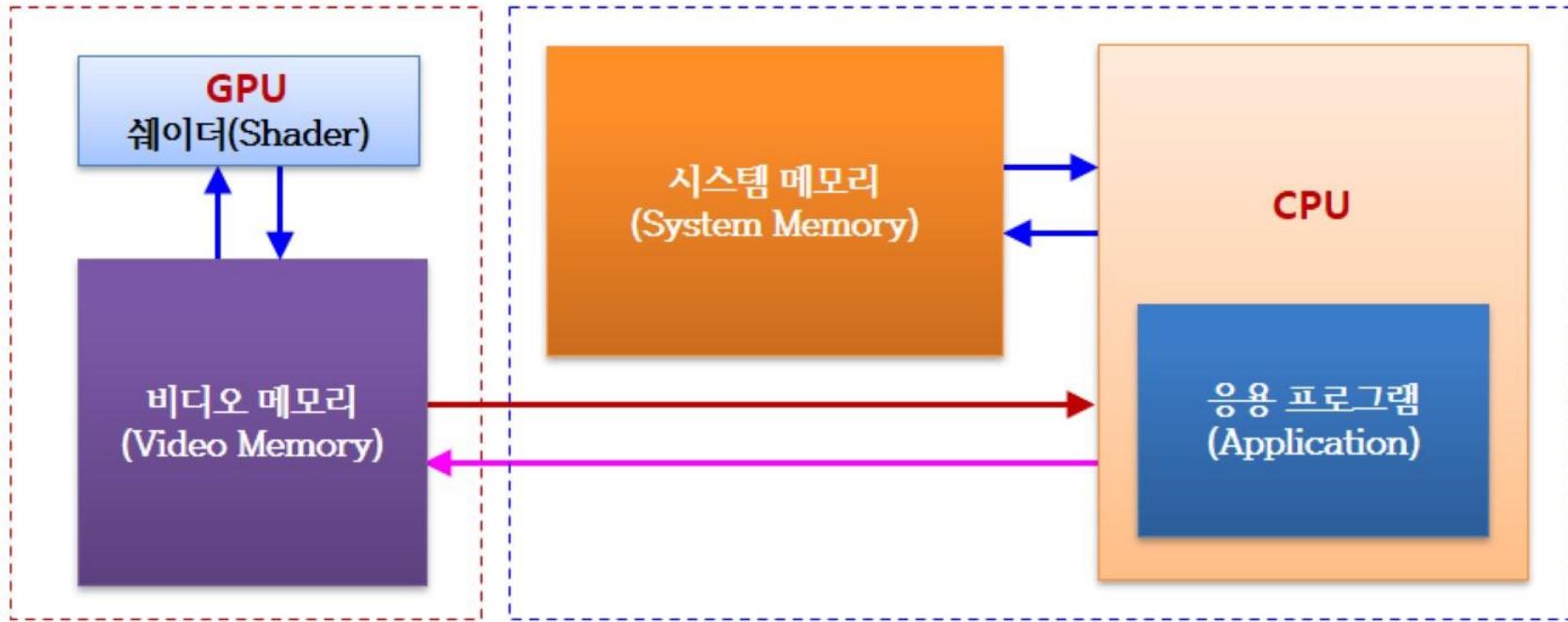
```
[numthreads(16, 16, 1)]  
void CS(int3 dtID : SV_DispatchThreadID)  
{  
    gtxtRWOutput[dtID.xy] = gtxtInputA[dtID.xy] + gtxtInputB[dtID.xy];  
}
```

Direct3D 파이프라인

- 비디오 메모리(Video Memory)

- 응용 프로그램에서 비디오 메모리의 내용을 **읽는** 연산은 엄청 **느리다**.

Graphic Card



  A가 B에서 읽는다

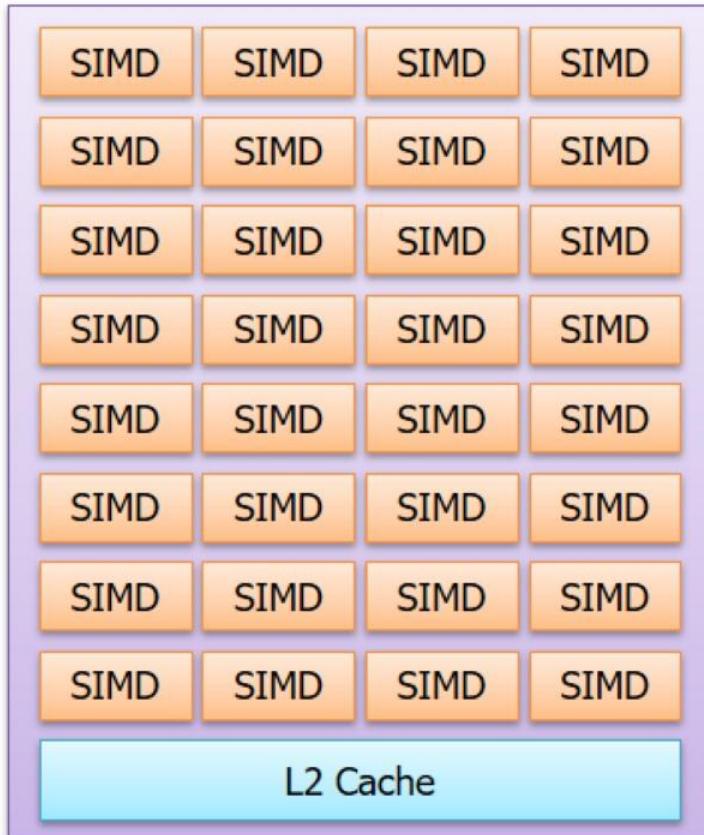
  A가 B에 쓴다

→ 속도가 느리다

→ 속도가 빠르다

Direct3D 파이프라인

- GPU



32 Cores
1GHz
1TeraFlop

NVIDIA GeForce GTX 690
CUDA Core: 3072

CPU

낮은 대기 시간(Latency) 메모리
랜덤 액세스(Random Access)
20GB/s 대역폭
0.1TFlop

GPU

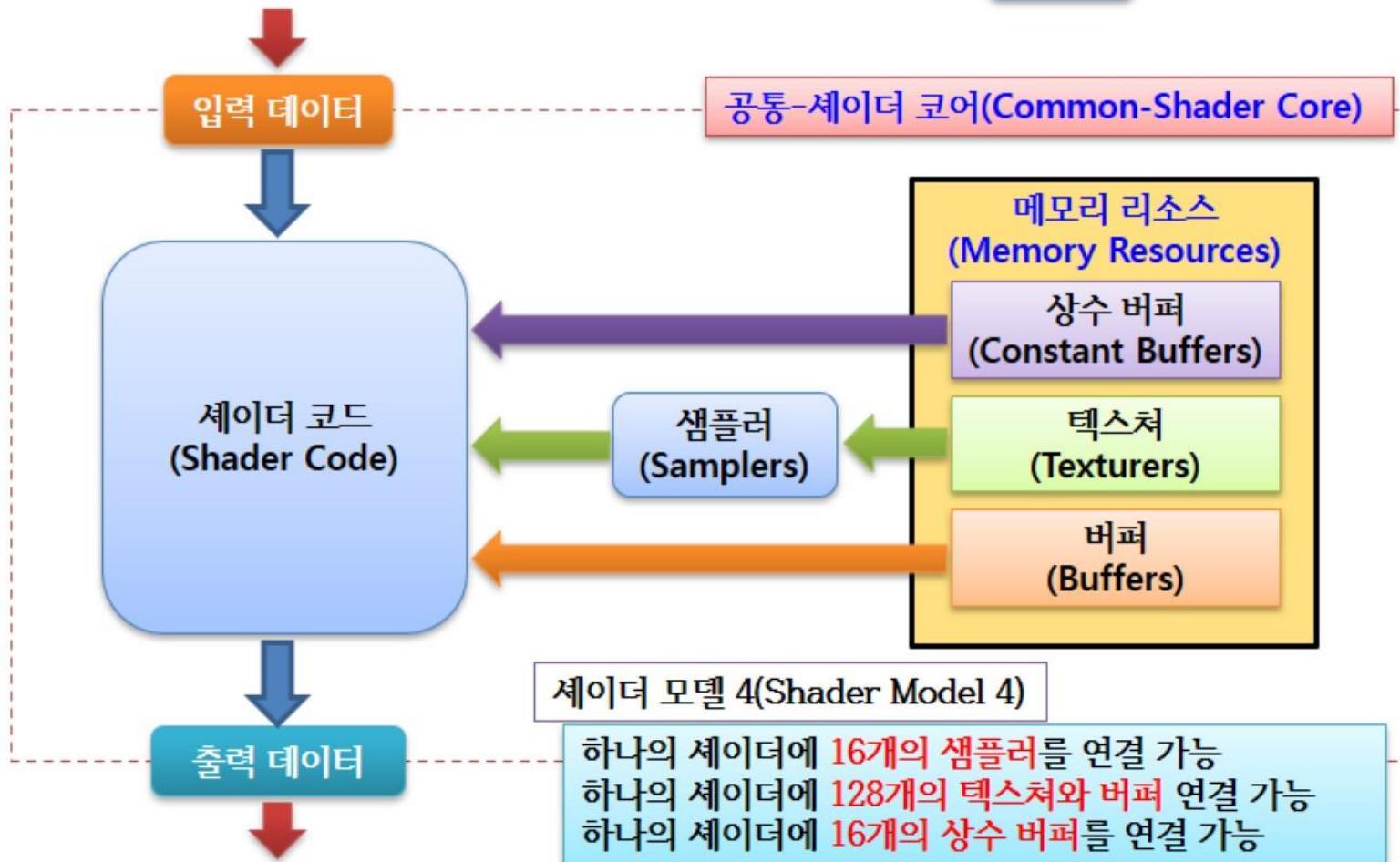
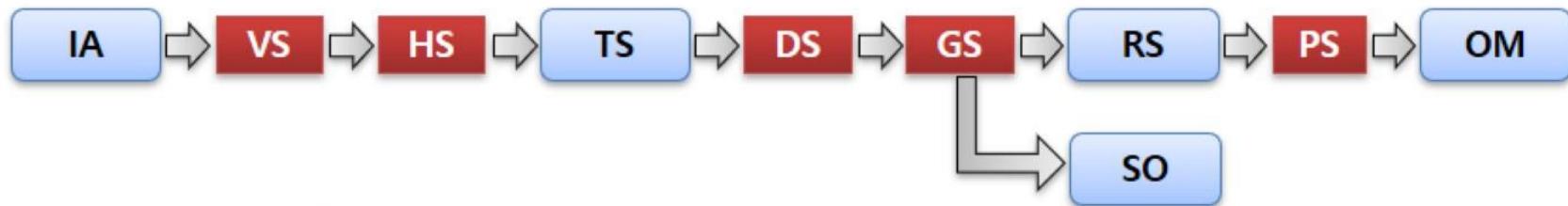
높은 대역폭(Bandwidth) 메모리
순차적 액세스(Sequential Access)
100GB/s 대역폭
1TFlop

GPU는 스트리밍 연산(Streaming Operation)에 최적화되어 있음
하나의 주소 또는 연속된 주소의 다량의 메모리를 처리(10배 빠름)
CPU는 Random Access Memory 접근에 최적화되어 있음
GPU는 기본적으로 병렬처리 구조(NVIDIA “Fermi”: 16x32코어(Core))

CPU는 Random Access Memory 접근에 최적화되어 있음

Direct3D 파이프라인

- Direct3D 셰이더 단계(Shader Stage)

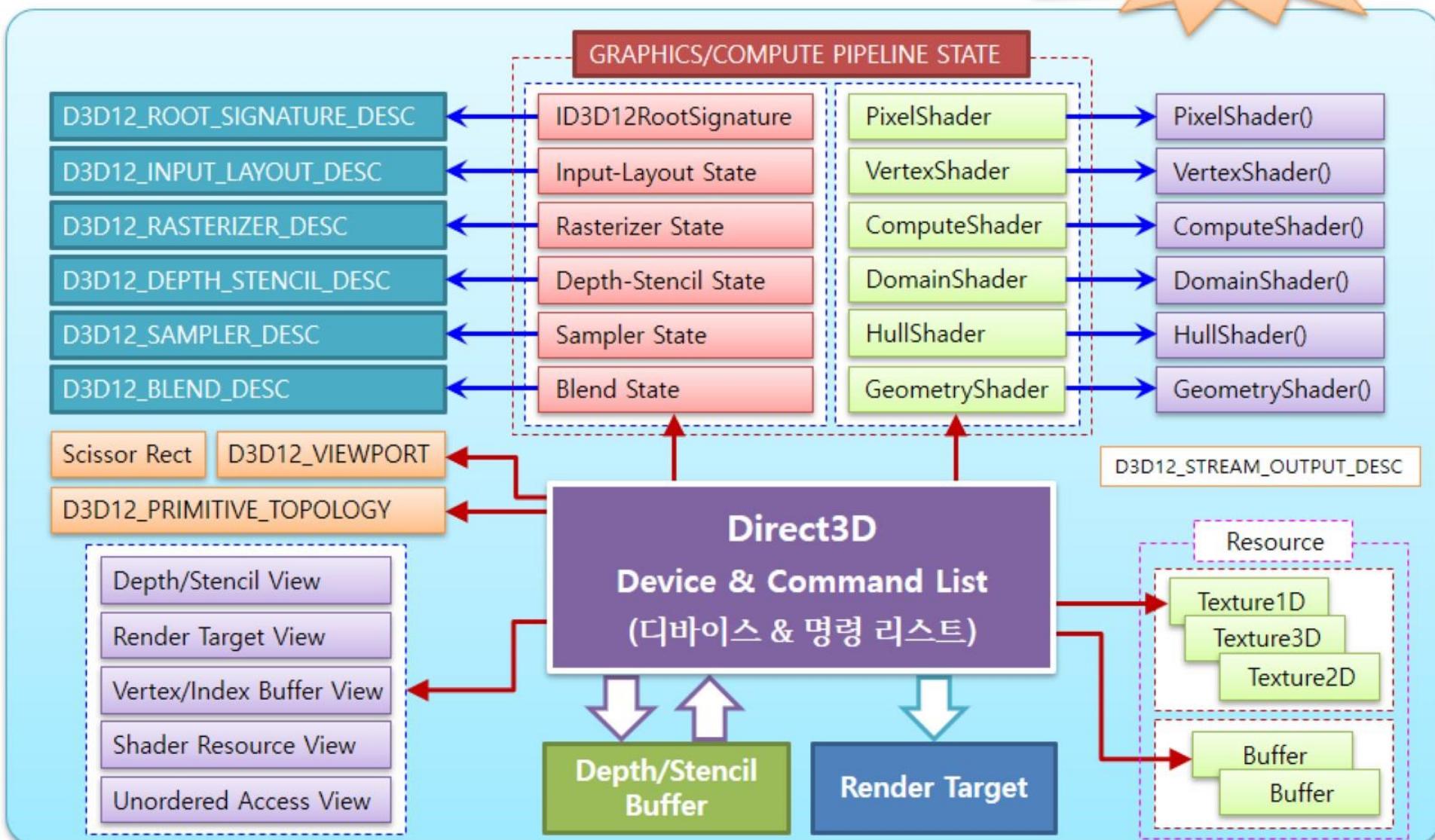


Direct3D 파이프라인

- Direct3D 디바이스

- Direct3D 디바이스는 상태 기계(State Machine)이다.

Set & Draw



Direct3D 12 디바이스

- **ID3D12Device**

- 가상 어댑터(Virtual Adapter)를 나타내는 인터페이스

CheckFeatureSupport

CopyDescriptors

CopyDescriptorsSimple

CreateCommandAllocator

CreateCommandList

CreateCommandQueue

CreateCommandSignature

CreateCommittedResource

CreateComputePipelineState

CreateConstantBufferView

CreateDepthStencilView

CreateDescriptorHeap

CreateFence

CreateGraphicsPipelineState

CreateHeap

CreatePlacedResource

CreateQueryHeap

CreateRenderTargetView

CreateReservedResource

CreateRootSignature

CreateSampler

CreateShaderResourceView

CreateSharedHandle

CreateUnorderedAccessView

Evict

GetAdapterLuid

GetCopyableFootprints

GetCustomHeapProperties

GetDeviceRemovedReason

GetNodeCount

GetDescriptorHandleIncrementSize

GetResourceAllocationInfo

GetResourceTiling

MakeResident

OpenSharedHandle

OpenSharedHandleByName

SetStablePowerState

Direct3D 12 디바이스

- **ID3D12GraphicsCommandList 인터페이스**

BeginEvent	BeginQuery	
ClearDepthStencilView	ClearRenderTargetView	
ClearState	ClearUnorderedAccessViewFloat	ClearUnorderedAccessViewUint
Close		
CopyBufferRegion	CopyResource	CopyTextureRegion
CopyTiles		
DiscardResource		
Dispatch		
DrawIndexedInstanced	DrawInstanced	
EndEvent	EndQuery	
ExecuteBundle	ExecuteIndirect	
IASetIndexBuffer	IASetPrimitiveTopology	IASetVertexBuffers
OMSetBlendFactor	OMSetRenderTargets	OMSetStencilRef
Reset		
ResolveQueryData	ResolveSubresource	
ResourceBarrier		
RSSetScissorRects	RSSetViewports	SetComputeRoot32BitConstant
SetComputeRoot32BitConstants	SetComputeRootConstantBufferView	
SetComputeRootDescriptorTable	SetComputeRootShaderResourceView	
SetComputeRootSignature	SetComputeRootUnorderedAccessView	
SetDescriptorHeaps		
SetGraphicsRoot32BitConstant	SetGraphicsRoot32BitConstants	SetGraphicsRootConstantBufferView
SetGraphicsRootDescriptorTable	SetGraphicsRootShaderResourceView	
SetGraphicsRootSignature	SetGraphicsRootUnorderedAccessView	
SetMarker	SetPipelineState	SetPredication
SOSetTargets		

Direct3D 파이프라인

- 계산-셰이더(Compute-Shader)

```
Texture2D gtxtInputA : register(t0);
Texture2D gtxtInputB : register(t1);
RWTexture2D<float4> gtxtRWOutput : register(u3);
```

[numthreads(32, 32, 1)]

```
void CS(int3 nDispatchID : SV_DispatchThreadID)
{
    gtxtRWOutput[nDispatchID.xy] = gtxtInputA[nDispatchID.xy] + gtxtInputB[nDispatchID.xy];
}
```

[numthreads(16, 16, 1)]

```
void CS(int3 nDispatchID : SV_DispatchThreadID)
{
```

```
    int x = nDispatchID.x;
    int y = nDispatchID.y;
    float tA = gtxtInputA[int2(x, y)].r;
    float tL = gtxtInputB[int2(x-1, y)].r;
    float tR = gtxtInputB[int2(x+1, y)].r;
    float tC = gtxtInputB[int2(x, y)].r;
    float tT = gtxtInputB[int2(x, y-1)].r;
    float tB = gtxtInputB[int2(x, y+1)].r;
    gtxtRWOutput[int2(x, y)] = gfWeights[0] * tA + gfWeights[1] * tC + gfWeights[2] * (tL + tR + tT + tB);
}
```

```
cbuffer cbWeights : register(b0)
{
    float gfWeights[3];
};
```

```
Texture2D gtxtInputA : register(t0);
Texture2D gtxtInputB : register(t1);
RWTexture2D<float> gtxtRWOutput : register(u0);
```

Direct3D 파이프라인

• 계산-셰이더(Compute-Shader)

- GPU의 병렬 프로세서를 사용하는 셰이더 프로그램
- 계산-셰이더(CS)는 GPGPU(General Purpose GPU) 프로그래밍을 제공
- 쓰레드(Thread)는 계산-셰이더의 기본 처리 단위임
- 쓰레드(Thread)들은 쓰레드 그룹(Thread Group)으로 분할됨
쓰레드 그룹의 쓰레드 개수는 계산-셰이더에서 선언
하나의 쓰레드 그룹은 하나의 멀티프로세서(Multiprocessor)에서 실행됨
하나의 계산-셰이더는 하나의 쓰레드 그룹의 각 쓰레드에서 병렬적으로 실행
쓰레드 그룹에서 특정한 쓰레드를 지정하기 위하여 3차원 벡터 (x, y, z)를 사용
- 계산-셰이더는 병렬 프로그래밍을 위해 메모리 공유와 쓰레드 동기화를 제공
- 8개의 무순서화 액세스 뷰(Unordered Access View)를 셰이더에 연결할 수 있음
- RWStructuredBuffer와 RWByteAddressBuffer를 사용할 수 있음
- 하나의 쓰레드는 그룹 공유 메모리의 어떤 영역이라도 읽기/쓰기 가능함
SV_GroupIndex 시멘틱을 사용해야 함
- 아토믹(Atomic) 연산을 제공

쓰레드의 최대 개수	그룹(Group)마다 1024개
numthreads의 Z 차원	64
dispatch의 Z 차원	1 (특성 레벨 10)
그룹 공유 메모리의 양	그룹(Group)마다 16KB
그룹 공유 메모리의 양/쓰레드	256바이트

병렬 알고리즘에 적합
물리/시뮬레이션

이미지 후처리(Image Post Processing)
Image Histogram
Image Reduction
Image FFT
인공지능(AI)
광선추적(Ray-Tracing)

Direct3D 파이프라인

• 계산-셰이더(Compute-Shader)

- 하나의 쓰레드 그룹은 여러 개의 쓰레드로 구성됨
하나의 쓰레드 그룹의 쓰레드들은 하나의 프로세서에서 병렬적으로 실행될 수 있음
하드웨어적으로 쓰레드들은 Warp(32개) 또는 Wavefront(64개) 단위로 분할됨
하나의 Warp 또는 Wavefront는 하나의 SIMD 멀티 프로세서에서 처리됨
Direct3D에서 쓰레드 그룹의 크기는 가급적이면 Wavefront 크기의 배수로 할 것
- 각 쓰레드는 32KB까지의 로컬 메모리(TLS: Thread Local Storage)를 가질 수 있음
로컬 메모리는 쓰레드들 사이에서 공유할 수 있음

→ 계산 셰이더에서 선언해야 함

(0,0,1)	(1,0,1)	(2,0,1)	(3,0,1)	(4,0,1)	
(0)	(0,0,0)	(1,0,0)	(2,0,0)	(3,0,0)	(4,0,0)
(0)	(0,1,0)	(1,1,0)	(2,1,0)	(3,1,0)	(4,1,0)
(0)	(0,2,0)	(1,2,0)	(2,2,0)	(3,2,0)	(4,2,0)
(0)	(0,3,0)	(1,3,0)	(2,3,0)	(3,3,0)	(4,3,0)
	(0,4,0)	(1,4,0)	(2,4,0)	(3,4,0)	(4,4,0)

50개의 쓰레드로 구성된 2개의 쓰레드 그룹
쓰레드 그룹 크기: (5, 5, 2)
하나의 쓰레드는 (x, y, z)로 지정: (4, 1, 0)

```
struct CSDATA
{
    float4 color;
    float factor;
};

groupshared CSDATA gData[5*5*1];

[numthreads(5, 5, 1)] //하나의 쓰레드 그룹의 쓰레드 개수
void CS(uint index : SV_GroupIndex)
{
    gData[index].color = (float4)0;
    gData[index].factor = 2.0f;
    GroupMemoryBarrierWithGroupSync();
    ...
}
```

Direct3D 파이프라인

- **Direct3D 리소스(Resource)**

- 세이더 모델 4

모든 리소스는 읽기 전용(Read Only)임

기본적으로 리소스는 읽기 전용(Read Only)

- Buffer
 - Texture1D, Texture1DArray
 - Texture2D, Texture2DArray
 - Texture3D, Texture3DArray
 - Texture2DMS, Texture2DMSArray
 - TextureCube, TextureCubeArray
 - StructuredBuffer
 - ByteAddressBuffer

모든 리소스 형식은 인덱스로 배열처럼 사용 가능

- 세이더 모델 5

세이더 모델 4의 읽기 전용 리소스를 제공

읽기와 쓰기(Write)가 가능한 리소스를 추가적으로 제공

읽기/쓰기 버퍼(Read/Write Buffers)와 읽기/쓰기 텍스쳐(Read/Write Textures)

- RWBuffer
 - RWTexture1D, RWTexture1DArray
 - RWTexture2D, RWTexture2DArray
 - RWTexture3D
 - RWStructuredBuffer
 - RWByteAddressBuffer

Direct3D 파이프라인

- **Direct3D 리소스(Resource)**

- 리소스는 기본적으로 읽기 전용(Read Only)
- 셰이더 모델 5는 읽기와 쓰기가 가능한 리소스를 제공

	VS	HS	DS	GS	PS	CS
Buffer	○	○	○	○	○	○
Texture1D/2D/3D	○	○	○	○	○	○
StructuredBuffer	○	○	○	○	○	○
ByteAddressBuffer	○	○	○	○	○	○
AppendStructuredBuffer					○	○
ConsumeStructuredBuffer					○	○
InputPatch		○		○		
OutputPatch		○	○			
RWStructuredBuffer					○	○
RWBuffer					○	○
RWByteAddressBuffer					○	○
RWTexture1D/2D/3D					○	○
RWTexture2DArray					○	○

Direct3D 파이프라인

- **HSL 자료형(Data Types): 텍스쳐**

- **Texture2D**

```
<Type> Texture2D.Gather(SamplerState s, float2 location, int2 offset); //텍스쳐를 샘플링하여 4 요소를 반환
<Type> Texture2D.GatherRed(SamplerState s, float2 location, int2 offset);
float4 Texture2D.GatherCmp(SamplerComparisonState s, float2 location, float value, int2 offset);
void Texture2D.GetDimensions(uint mipLevel, out uint width, out uint height, out uint mipLevels);
void Texture2D.GetDimensions(out uint width, out uint height);
<Type> Texture2D.Load(int3 uvw, [int sampleIndex,] int2 offset);
<Type> Texture2D.[](uint2 position); //텍셀 좌표(인덱스)
<Type> Texture2D.mips.Operator[](uint mipSlice, uint2 position);
DXGI_FORMAT Object.Sample(SamplerState S, floatx Location[, intx Offset]);
DXGI_FORMAT Object.Sample(SamplerState S, floatx Location[, intx Offset][, in float ClampLOD]);
DXGI_FORMAT Object.SampleLevel(SamplerState S, floatx Location, float LOD[, intx Offset][, out uint Status]);
float Object.SampleCmp(SamplerComparisonState S, floatx Location, float Value[, intx Offset]);
float Object.SampleCmpLevelZero(SamplerComparisonState S, floatx Location, float Value[, intx Offset]);
DXGI_FORMAT Object.SampleGrad(SamplerState S, floatx Location, floatx DDX, floatx DDY[, intx Offset]);
```

```
Texture2D<float4> gtxtTexture;
float4 color = gtxtTexture.mips[2][xyPos];
float4 color = gtxtTexture[xyPos];
```

- **RWTexture2D**

여러 개의 쓰레드가 동시에 읽기/쓰기를 할 수 있음

같은 무순서 접근 리소스를 두개의 셰이더에 동시에 연결할 수 있음

```
void RWTexture2D.GetDimensions(out uint width, out uint height); //텍스쳐의 크기(텍셀 수)
```

```
<Type> RWTexture2D.Load(int3 uvw); //텍스쳐의 값을 읽음
```

```
<Type> RWTexture2D.[](uint2 position); //텍셀 좌표(인덱스)
```

```
RWTexture2D<float4> gtxtTexture;
uint2 xyPos = { 123, 456 };
gtxtTexture[uint2(3, 2)] = float4(255,0,0,0);
```

Direct3D 파이프라인

• HLSL 자료형(Data Types): 버퍼

Buffer<Type> Name;

RWBuffer<Type> Name;

GetDimensions

버퍼의 길이(바이트)를 반환

Load

버퍼 데이터를 반환(샘플링 또는 필터링을 사용하지 않음)

operator []

읽기/쓰기 리소스 변수를 반환(쓰기는 픽셀 셰이더와 계산 셰이더에서 허용)

Buffer<float4> myBuffer;

float loc;

uint status;

float4 myColor = myBuffer.Load(loc, status);

StructuredBuffer<Type> Name;

RWStructuredBuffer<Type> Name;

void GetDimensions(out uint numStructs, out uint stride);

AppendStructuredBuffer<Type> Name;

ConsumeStructuredBuffer<Type> Name;

void AppendStructuredBuffer::Append(in T value);

T ConsumeStructuredBuffer::Consume();

void GetDimensions(out uint dim);

Type Load(in int Location, out uint Status);

Type Load(in int Location);

Type operator[](in uint pos);

ByteAddressBuffer Name;

RWByteAddressBuffer Name;

uint Load(in int Location);

void Store(in uint address, in uint value);

AppendStructuredBuffer<MyStruct> appendsb : register(u0);
appendsb.Append(myStruct);

ConsumeStructuredBuffer<MyStruct> consumesb : register(u1);
MyStruct s = consumesb.Consume();

struct MyStruct
{
 float4 Color;
 float4 Normal;
};

Direct3D 파이프라인

• 구조화 버퍼(Structured Buffer)

```
RWStructuredBuffer<PARTICLE> ParticlesRW : register(u0);
```

```
StructuredBuffer<PARTICLE> ParticlesRO : register(t0);
```

```
RWStructuredBuffer<PARTICLEFORCES> ParticlesForcesRW : register(u0);
```

```
StructuredBuffer<PARTICLEFORCES> ParticlesForcesRO : register(t2);
```

```
RWStructuredBuffer<unsigned int> GridRW : register(u0);
```

```
StructuredBuffer<unsigned int> GridRO : register(t3);
```

```
RWStructuredBuffer<uint2> GridIndicesRW : register(u0);
```

```
StructuredBuffer<uint2> GridIndicesRO : register(t4);
```

```
float3 vPosition = ParticlesRO[0].position;
```

```
float2 vVelocity = ParticlesRO[1].velocity;
```

```
ParticlesRW[2] = ParticlesRO[3];
```

```
ParticlesRW[0].position = float3(1.0f, 0.0f, 0.0f);
```

```
ParticlesRW[0].velocity = float2(1.0f, 1.0f);
```

```
GridIndicesRW[1] = uint2(0, 0);
```

```
GridIndicesRW[0].x = 0;
```

```
uint2 vIndex = GridIndicesRO[0];
```

```
float3 vPosition = ParticlesRO[0].position;
```

```
GridRW[3] = 10;
```

```
struct PARTICLE
{
    float3 position;
    float2 velocity;
};
```

```
struct PARTICLEFORCES
{
    float2 acceleration;
};
```

```
struct PARTICLE
{
    XMFLOAT3 vPosition;
    XMFLOAT2 vVelocity;
};
```

```
struct PARTICLEFORCES
{
    XMFLOAT2 vAcceleration;
};
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource**

- 리소스와 힙을 동시에 생성(Virtual/Physical Address Space)

```
HRESULT ID3D12Device::CreateCommittedResource(
```

```
    D3D12_HEAP_PROPERTIES *pHeapProperties,
```

```
    D3D12_HEAP_FLAGS HeapFlags,
```

```
    D3D12_RESOURCE_DESC *pResourceDesc,
```

```
    D3D12_RESOURCE_STATES InitialResourceState, //리소스가 사용되는 방법에 따른 리소스의 상태
```

```
    D3D12_CLEAR_VALUE *pOptimizedClearValue, //NULL: D3D12_RESOURCE_DIMENSION_BUFFER
```

```
    REFIID riidResource, //__uuidof(ID3D12Resource)
```

```
    void **ppvResource
```

```
);
```

```
typedef enum D3D12_HE
```

```
    D3D12_HEAP_FLAG_N
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    } D3D12_HEAP_FLAGS;
```

```
    D3D12_TEXTURE_LAYOUT Layout;
```

```
    D3D12_RESOURCE_FLAGS Flags;
```

```
    } D3D12_RESOURCE_DESC;
```

```
    ID3D12Device::CreatePlacedResource(...)
```

```
    ID3D12Device::CreateReservedResource(...)
```

```
typedef struct D3D12_HEAP_PROPERTIES {
```

```
    D3D12_HEAP_TYPE Type;
```

```
    D3D12_CPU_PAGE_PROPERTY CPUPageProperty;
```

```
    D3D12_MEMORY_POOL MemoryPoolPreference;
```

```
    UINT CreationNodeMask;
```

```
    UINT VisibleNodeMask;
```

```
    } D3D12_HEAP_PROPERTIES;
```

```
typedef struct D3D12_CLEAR_VALUE {
```

```
    DXGI_FORMAT Format;
```

```
    union {
```

```
        FLOAT Color[4];
```

```
        D3D12_DEPTH_STENCIL_VALUE DepthStencil;
```

```
    };
```

```
    } D3D12_CLEAR_VALUE;
```

```
    D3D12_RESOURCE_STATE_PREDICATION
```

```
    } D3D12_RESOURCE_STATES;
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef struct D3D12_RESOURCE_DESC {
    D3D12_RESOURCE_DIMENSION Dimension;
    UINT64 Alignment; //정렬: 0(64KB), 4KB, 64KB, 4MB
    UINT64 Width; //리소스의 가로 크기
    UINT Height; //리소스의 세로 크기
    UINT16 DepthOrArraySize; //깊이 또는 배열의 크기
    UINT16 MipLevels; // mip맵 레벨의 개수, 0(자동 계산)
    DXGI_FORMAT Format; //리소스 형식
    DXGI_SAMPLE_DESC SampleDesc; //다중 샘플링
    D3D12_TEXTURE_LAYOUT Layout; //다차원 리소스를 1차원 리소스로 매핑하기 위한 방법
    D3D12_RESOURCE_FLAGS Flags;
} D3D12_RESOURCE_DESC;
```

RWTexture2D
RWStructuredBuffer

```
typedef enum D3D12_RESOURCE_DIMENSION {
    D3D12_RESOURCE_DIMENSION_UNKNOWN,
    D3D12_RESOURCE_DIMENSION_BUFFER,
    D3D12_RESOURCE_DIMENSION_TEXTURE1D,
    D3D12_RESOURCE_DIMENSION_TEXTURE2D,
    D3D12_RESOURCE_DIMENSION_TEXTURE3D
} D3D12_RESOURCE_DIMENSION;
```

```
typedef enum D3D12_RESOURCE_FLAGS {
    D3D12_RESOURCE_FLAG_NONE,
    D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET, //RTV 허용
    D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL, //DSV 허용
    D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS, //UAV 허용
    D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE,
    D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER,
    D3D12_RESOURCE_FLAG_ALLOW_SIMULTANEOUS_ACCESS
} D3D12_RESOURCE_FLAGS;
```

```
typedef enum D3D12_TEXTURE_LAYOUT {
    D3D12_TEXTURE_LAYOUT_UNKNOWN, //어댑터에 의존(최상의 레이아웃을 선택)
    D3D12_TEXTURE_LAYOUT_ROW_MAJOR, //텍스쳐 데이터가 행 우선 순서로 저장됨
    D3D12_TEXTURE_LAYOUT_64KB_UNDEFINED_SWIZZLE, //드라이버에 의존(레이아웃이 64KB 영역으로 배치)
    D3D12_TEXTURE_LAYOUT_64KB_STANDARD_SWIZZLE //레이아웃이 64KB 영역으로 배치(표준적인 방법)
} D3D12_TEXTURE_LAYOUT;
```

Direct3D 파이프라인

- **ID3D12Device::CreateCommittedResource()**

```
typedef enum D3D12_RESOURCE_STATES {  
    D3D12_RESOURCE_STATE_COMMON, //0x00, CPU가 텍스쳐에 접근, COPY 큐의 상태 전이 전  
    D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER, //0x01, 정점 버퍼 또는 상수 버퍼로 사용될 때  
    D3D12_RESOURCE_STATE_INDEX_BUFFER, //0x02, 인덱스 버퍼로 사용될 때  
    D3D12_RESOURCE_STATE_RENDER_TARGET, //0x04, 렌더 타겟으로 사용될 때, ClearRenderTargetView()  
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS, //0x08, 무순서 접근으로 사용될 때  
    D3D12_RESOURCE_STATE_DEPTH_WRITE, //0x10, 깊이 버퍼에 쓸 때, ClearDepthStencilView()  
    D3D12_RESOURCE_STATE_DEPTH_READ, //0x20, 깊이 버퍼를 읽을 때  
    D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE, //0x40, 픽셀 셰이더 이외의 리소스로 사용될 때  
    D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE, //0x80, 픽셀 셰이더의 리소스로 사용될 때  
    D3D12_RESOURCE_STATE_STREAM_OUT, //0x100, 스트림 출력으로 사용될 때  
    D3D12_RESOURCE_STATE_INDIRECT_ARGUMENT, //0x200, ExecuteIndirect()  
    D3D12_RESOURCE_STATE_COPY_DEST, //0x400, 복사 연산의 목표로 사용될 때  
    D3D12_RESOURCE_STATE_COPY_SOURCE, //0x800, 복사 연산의 소스로 사용될 때  
    D3D12_RESOURCE_STATE_RESOLVE_DEST, //0x1000, 리졸브 연산의 목표로 사용될 때  
    D3D12_RESOURCE_STATE_RESOLVE_SOURCE, //0x2000, 리졸브 연산의 소스로 사용될 때  
    D3D12_RESOURCE_STATE_GENERIC_READ, //업로드 힙의 시작 상태(가급적 피할 것)  
    D3D12_RESOURCE_STATE_PRESENT, //D3D12_RESOURCE_STATE_COMMON  
    D3D12_RESOURCE_STATE_PREDICATION //리소스가 예측을 위하여 사용될 때  
} D3D12_RESOURCE_STATES;
```

D3D12_HEAP_TYPE_UPLOAD: 초기 상태는 D3D12_RESOURCE_STATE_GENERIC_READ

D3D12_HEAP_TYPE_READBACK: 초기 상태는 D3D12_RESOURCE_STATE_COPY_DEST

텍스쳐: CPU 접근을 위하여 D3D12_RESOURCE_STATE_COMMON

D3D12_RESOURCE_STATE_GENERIC_READ: (0x01 | 0x02 | 0x40 | 0x80 | 0x200 | 0x800)

텍스쳐(Texture)

- 셰이더 리소스 뷰(Shader Resource View)

```
void ID3D12Device::CreateShaderResourceView(  
    ID3D12Resource *pResource,  
    D3D12_SHADER_RESOURCE_VIEW_DESC *pDesc,  
    D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor  
)
```

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC {  
    DXGI_FORMAT Format;  
    D3D12_SRV_DIMENSION ViewDimension;  
    UINT Shader4ComponentMapping;  
    union {  
        D3D12_BUFFER_SRV E  
        D3D12_TEX1D_SRV Tex  
        D3D12_TEX1D_ARRAY_  
        D3D12_TEX2D_SRV Te  
        D3D12_TEX2D_ARRAY_  
        D3D12_TEX2DMS_SRV  
        D3D12_TEX2DMS_ARR  
        D3D12_TEX3D_SRV Tex  
        D3D12_TEXCUBE_SRV TextureCube;  
        D3D12_TEXCUBE_ARRAY_SRV TextureCu  
    };  
} D3D12_SHADER_RESOURCE_VIEW_DESC;
```

```
typedef enum D3D12_SRV_DIMENSION {  
    D3D12_SRV_DIMENSION_UNKNOWN,  
    D3D12_SRV_DIMENSION_BUFFER,  
    D3D12_SRV_DIMENSION_TEXTURE1D,  
    D3D12_SRV_DIMENSION_TEXTURE1DARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE2D,  
    D3D12_SRV_DIMENSION_TEXTURE2DARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE2DMS,  
    D3D12_SRV_DIMENSION_TEXTURE2DMSARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE3D,  
    D3D12_SRV_DIMENSION_TEXTURECUBE,  
    D3D12_SRV_DIMENSION_TEXTURECUBEARRAY  
} D3D12_SRV_DIMENSION;  
typedef enum D3D12_SHADER_COMPONENT_MAPPING {  
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_0,  
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_1,  
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_2,  
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_3,  
    D3D12_SHADER_COMPONENT_MAPPING_FORCE_VALUE_0,  
    D3D12_SHADER_COMPONENT_MAPPING_FORCE_VALUE_1  
} D3D12_SHADER_COMPONENT_MAPPING;
```

```
typedef struct D3D12_BUFFER_SRV {  
    UINT64 FirstElement;  
    UINT NumElements;  
    UINT StructureByteStride;  
    D3D12_BUFFER_SRV_FLAGS Flags  
} D3D12_BUFFER_SRV;
```

```
typedef struct D3D12_TEX2D_SRV {  
    UINT MostDetailedMip;  
    UINT MipLevels;  
    UINT PlaneSlice;  
    FLOAT ResourceMinLODClamp;  
} D3D12_TEX2D_SRV;
```

텍스쳐(Texture)

- 셰이더 리소스 뷰(Shader Resource View)

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC {  
    DXGI_FORMAT Format; //뷰가 리소스에 접근하는 형식  
    D3D12_SRV_DIMENSION ViewDimension;  
    UINT Shader4ComponentMapping;  
    union {  
        D3D12_BUFFER_SRV Buffer;  
        D3D12_TEX1D_SRV Texture1D;  
        D3D12_TEX1D_ARRAY_SRV Texture1DArray;  
        D3D12_TEX2D_SRV Texture2D;  
        D3D12_TEX2D_ARRAY_SRV Texture2DArray;  
        D3D12_TEX2DMS_SRV Texture2DMS;  
        D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;  
        D3D12_TEX3D_SRV Texture3D;  
        D3D12_TEXCUBE_SRV TextureCube;  
        D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;  
    };  
} D3D12_SHADER_RESOURCE_VIEW_DESC;
```

```
typedef struct D3D12_TEX2D_SRV {  
    UINT MostDetailedMip; //0~MipLevels  
    UINT MipLevels; // mip맵 레벨 수, -1  
    UINT PlaneSlice; //평면 슬라이스 번호(인덱스)  
    FLOAT ResourceMinLODClamp; //최소 mip맵 레벨  
} D3D12_TEX2D_SRV;
```

```
typedef enum D3D12_SRV_DIMENSION {  
    D3D12_SRV_DIMENSION_UNKNOWN,  
    D3D12_SRV_DIMENSION_BUFFER,  
    D3D12_SRV_DIMENSION_TEXTURE1D,  
    D3D12_SRV_DIMENSION_TEXTURE1DARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE2D,  
    D3D12_SRV_DIMENSION_TEXTURE2DARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE2DMS,  
    D3D12_SRV_DIMENSION_TEXTURE2DMSARRAY,  
    D3D12_SRV_DIMENSION_TEXTURE3D,  
    D3D12_SRV_DIMENSION_TEXTURECUBE,  
    D3D12_SRV_DIMENSION_TEXTURECUBEARRAY  
} D3D12_SRV_DIMENSION;  
//뷰의 리소스 유형
```

```
typedef enum D3D12_BUFFER_SRV_FLAGS {  
    D3D12_BUFFER_SRV_FLAG_NONE,  
    D3D12_BUFFER_SRV_FLAG_RAW  
} D3D12_BUFFER_SRV_FLAGS;
```

```
typedef struct D3D12_BUFFER_SRV {  
    UINT64 FirstElement; //첫번째 원소(인덱스)  
    UINT NumElements; //원소의 개수  
    UINT StructureByteStride; //구조체 원소의 크기(바이트)  
    D3D12_BUFFER_SRV_FLAGS Flags;  
} D3D12_BUFFER_SRV;
```

텍스쳐(Texture)

- 셰이더 리소스 뷰(Shader Resource View)

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D12_SRV_DIMENSION ViewDimension;
    UINT Shader4ComponentMapping; //D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING
    union {
        D3D12_BUFFER_SRV Buffer;
        D3D12_ENCODE_SHADER_4_COMPONENT_MAPPING(Src0,Src1,Src2,Src3)
    };
#define D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING D3D12_ENCODE_SHADER_4_COMPONENT_MAPPING(0,1,2,3)
    D3D12_TEX1D_ARRAY_SRV Texture1DArray,
    D3D12_TEX2D_SRV Texture2D;
    D3D12_TEX2D_ARRAY_SRV Texture2DArray;
    D3D12_TEX2DMS_SRV Texture2DMS;
    D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;
    D3D12_TEX3D_SRV Texture3D;
    D3D12_TEXCUBE_SRV TextureCube;
    D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;
};

} D3D12_SHADER_RESOURCE_VIEW_DESC;

typedef enum D3D12_SHADER_COMPONENT_MAPPING {
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_0, //요소 0(Red)
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_1, //요소 1(Green)
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_2, //요소 2(Blue)
    D3D12_SHADER_COMPONENT_MAPPING_FROM_MEMORY_COMPONENT_3, //요소 3(Alpha)
    D3D12_SHADER_COMPONENT_MAPPING_FORCE_VALUE_0, //0
    D3D12_SHADER_COMPONENT_MAPPING_FORCE_VALUE_1 //1
} D3D12_SHADER_COMPONENT_MAPPING;
```

Direct3D 파이프라인

- 무순서 접근 뷰(Unordered Access View) 생성

```
void ID3D12Device::CreateUnorderedAccessView(  
    ID3D12Resource *pResource,  
    ID3D12Resource *pCounterResource, //RWStructuredBuffer의 카운터를 위한 리소스  
    D3D12_UNORDERED_ACCESS_VIEW_DESC *pDesc, //NULL: 전체 리소스를 사용하는 기본 뷰 생성  
    D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor  
);
```

```
typedef struct D3D12_UNORDERED_ACCESS_VIEW_DESC  
{  
    DXGI_FORMAT Format; //구조화 버퍼(UNKNOWN)  
    D3D12_UAV_DIMENSION ViewDimension;  
    union {  
        D3D12_BUFFER_UAV Buffer;  
        D3D12_TEX1D_UAV Texture1D;  
        D3D12_TEX1D_ARRAY_UAV Texture1DArray;  
        D3D12_TEX2D_UAV Texture2D;  
        D3D12_TEX2D_ARRAY_UAV Texture2DArray;  
        D3D12_TEX3D_UAV Texture3D;  
    };  
} D3D12_UNORDERED_ACCESS_VIEW_DESC;
```

```
typedef struct D3D12_TEX2D_UAV {  
    UINT MipSlice; // mip 슬라이스 인덱스  
    UINT PlaneSlice; // 평면 슬라이스 인덱스  
} D3D12_TEX2D_UAV;
```

```
uint RWStructuredBuffer::IncrementCounter();
```

```
typedef enum D3D12_UAV_DIMENSION {  
    D3D12_UAV_DIMENSION_UNKNOWN,  
    D3D12_UAV_DIMENSION_BUFFER,  
    D3D12_UAV_DIMENSION_TEXTURE1D,  
    D3D12_UAV_DIMENSION_TEXTURE1DARRAY,  
    D3D12_UAV_DIMENSION_TEXTURE2D,  
    D3D12_UAV_DIMENSION_TEXTURE2DARRAY,  
    D3D12_UAV_DIMENSION_TEXTURE3D  
} D3D12_UAV_DIMENSION;
```

```
typedef struct D3D12_BUFFER_UAV {  
    UINT FirstElement; // 접근할 첫 번째 원소  
    UINT NumElements; // 리소스 원소의 개수  
    UINT StructureByteStride; // 구조체의 크기  
    UINT64 CounterOffsetInBytes; // 4의 배수  
    UINT Flags; // 리소스에 대한 뷰 선택사항  
} D3D12_BUFFER_UAV;
```

```
typedef enum D3D12_BUFFER_UAV_FLAGS {  
    D3D12_BUFFER_UAV_FLAG_NONE, // 기본 뷰  
    D3D12_BUFFER_UAV_FLAG_RAW // 구조화되지 않음  
} D3D12_BUFFER_UAV_FLAGS;
```

Direct3D 파이프라인

- UAV 카운터(UAV Counter)

```
void ID3D12Device::CreateUnorderedAccessView(  
    ID3D12Resource *pResource,  
    ID3D12Resource *pCounterResource, //RWStructuredBuffer의 카운터를 위한 리소스  
    D3D12_UNORDERED_ACCESS_VIEW_DESC *pDesc, //NULL: 전체 리소스를 사용하는 기본 뷰 생성  
    D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor  
);
```

```
typedef struct D3D12_UNORDERED_ACCESS_VIEW_DESC  
{  
    DXGI_FORMAT Format; //구조화 버퍼(UNKNOWN)  
    D3D12_UAV_DIMENSION ViewDimension;  
    union {  
        D3D12_BUFFER_UAV Buffer;  
        D3D12_TEX1D_UAV Texture1D;  
        D3D12_TEX1D_ARRAY_UAV Texture1DArray;  
        D3D12_TEX2D_UAV Texture2D;  
        D3D12_TEX2D_ARRAY_UAV Texture2DArray;  
        D3D12_TEX3D_UAV Texture3D;  
    };  
} D3D12_UNORDERED_ACCESS_VIEW_DESC;
```

```
typedef struct D3D12_BUFFER_UAV {  
    UINT FirstElement; //접근할 첫 번째 원소  
    UINT NumElements; //리소스 원소의 개수  
    UINT StructureByteStride; //구조체의 크기  
    UINT64 CounterOffsetInBytes; //4의 배수  
    UINT Flags; //리소스에 대한 뷰 선택사항  
} D3D12_BUFFER_UAV;
```

```
uint RWStructuredBuffer.DecrementCounter(); //UAV 카운터를 1 감소시킴  
uint RWStructuredBuffer.IncrementCounter(); //UAV 카운터를 1 증가시킴
```

Direct3D 파이프라인

- 루트 시그너쳐(Root Signature)

- 어떤 리소스(데이터)들이 그래픽스 파이프라인에 연결되는 가를 정의
- 명령 리스트들을 셰이더 리소스에 연결
셰이더(GPU 함수)가 요구하는 데이터 형을 정의
 - 그래픽 루트 시그너쳐(Graphics Root Signature)
 - 계산 루트 시그너쳐(Compute Root Signature)**
- 루트 매개변수(Root Parameters)
하나의 루트 시그너쳐의 크기는 64개의 32-비트(DWORD)까지 허용
많이 사용하는 매개변수를 앞쪽에(빠른 메모리) 배치하는 것이 좋음
 - 루트 상수(Root Constant): 셰이더의 상수 버퍼로 대응, 32-비트(DWORD)
 - 루트 서술자(Root Descriptor): 상수 버퍼(CBV), 버퍼(SRV, UAV), 64-비트
 - 서술자 테이블(Descriptor Table): 리소스 서술자 힙의 영역을 표현, 32-비트
- 루트 인자(Root Argument)
실행시에 셰이더로 전달되는 루트 매개변수의 값
- ID3D12RootSignature 인터페이스

```
D3D12SerializeRootSignature(...);  
ID3D12Device::CreateRootSignature(...);
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC  
ID3D12Device::CreateGraphicsPipelineState(...);  
ID3D12Device::CreateComputePipelineState(...);
```

```
D3D12_COMPUTE_PIPELINE_STATE_DESC {  
    ID3D12RootSignature *pRootSignature;  
    D3D12_SHADER_BYTECODE CS;  
    ...  
} D3D12_COMPUTE_PIPELINE_STATE_DESC;
```

```
ID3D12GraphicsCommandList::SetGraphicsRootSignature(ID3D12RootSignature *pRootSignature);  
ID3D12GraphicsCommandList::SetComputeRootSignature(ID3D12RootSignature *pRootSignature);
```

Direct3D 파이프라인

- 계산 파이프라인 상태(Compute Pipeline State)

```
HRESULT ID3D12Device::CreateComputePipelineState( //계산 파이프라인 상태 객체를 생성
```

```
    D3D12_COMPUTE_PIPELINE_STATE_DESC *pDesc,  
    REFIID riid, //__uuidof(ID3D12PipelineState)  
    void **ppPipelineState //ID3D12PipelineState  
);
```

```
typedef struct D3D12_COMPUTE_PIPELINE_STATE_DESC {  
    ID3D12RootSignature *pRootSignature;  
    D3D12_SHADER_BYTECODE CS;  
    UINT NodeMask; //단일 GPU: 0  
    D3D12_CACHED_PIPELINE_STATE CachedPSO;  
    D3D12_PIPELINE_STATE_FLAGS Flags;  
} D3D12_COMPUTE_PIPELINE_STATE_DESC;
```

```
HRESULT ID3D12PipelineState::GetCachedBlob(  
    ID3DBlob **ppBlob  
);
```

```
void ID3D12GraphicsCommandList::SetPipelineState(  
    ID3D12PipelineState *pPipelineState  
);
```

```
typedef struct D3D12_SHADER_BYTECODE {  
    void *pShaderBytecode;  
    SIZE_T BytecodeLength;  
} D3D12_SHADER_BYTECODE;
```

```
typedef struct D3D12_CACHED_PIPELINE_STATE {  
    void *pCachedBlob;  
    SIZE_T CachedBlobSizeInBytes;  
} D3D12_CACHED_PIPELINE_STATE;
```

```
typedef enum D3D12_PIPELINE_STATE_FLAGS {  
    D3D12_PIPELINE_STATE_FLAG_NONE,  
    D3D12_PIPELINE_STATE_FLAG_TOOL_DEBUG  
} D3D12_PIPELINE_STATE_FLAGS;
```

```
D3D12_COMPUTE_PIPELINE_STATE_DESC d3dComputePSODesc = { };  
d3dComputePSODesc.pRootSignature = m_pd3dComputeRootSignature;  
d3dComputePSODesc.CS.pShaderBytecode = pd3dComputeShaderBlob->GetBufferPointer();  
d3dComputePSODesc.CS.BytecodeLength = pd3dComputeShaderBlob->GetBufferSize();  
m_pd3dDevice->CreateComputePipelineState(&d3dComputePSODesc, IID_PPV_ARGS(&m_pComputeState));
```

"cs_5_1"

Direct3D 파이프라인

- 계산 셰이더(Compute Shader) 데이터 전달

```
void ID3D12GraphicsCommandList::SetComputeRoot32BitConstant(  
    UINT RootParameterIndex,  
    UINT SrcData,  
    UINT DestOffsetIn32BitValues  
);
```

```
void ID3D12GraphicsCommandList::SetComputeRoot32BitConstants(...);
```

```
void ID3D12GraphicsCommandList::SetComputeRootConstantBufferView(  
    UINT RootParameterIndex,  
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation  
);
```

```
void ID3D12GraphicsCommandList::SetComputeRootShaderResourceView(  
    UINT RootParameterIndex,  
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation  
);
```

```
void ID3D12GraphicsCommandList::SetComputeRootUnorderedAccessView(  
    UINT RootParameterIndex,  
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation  
);
```

```
D3D12_GPU_VIRTUAL_ADDRESS ID3D12Resource::GetGPUVirtualAddress();
```

```
void ID3D12GraphicsCommandList::SetComputeRootDescriptorTable(  
    UINT RootParameterIndex,  
    D3D12_GPU_DESCRIPTOR_HANDLE BaseDescriptor  
);
```

```
typedef struct D3D12_GPU_DESCRIPTOR_HANDLE {  
    UINT64 ptr;  
} D3D12_GPU_DESCRIPTOR_HANDLE;
```

Direct3D 파이프라인

• 계산 파이프라인(셰이더)의 실행

```
void ID3D12GraphicsCommandList::Dispatch(
```

 UINT ThreadGroupCountX, //x 방향으로 디스패치될 그룹의 개수, 65,535 보다 작아야 함

 UINT ThreadGroupCountY, //y 방향으로 디스패치될 그룹의 개수, 65,535 보다 작아야 함

 UINT ThreadGroupCountZ //z 방향으로 디스패치될 그룹의 개수, 65,535 보다 작아야 함

);

현재 파이프라인에 연결된 계산 셰이더(Compute Shader)를 병렬적으로 실행하기 위하여 호출
($X \times Y \times Z$)개의 쓰레드 그룹(Thread Group)들을 생성

하나의 쓰레드 그룹에는 여러 개의 쓰레드(Thread)들이 존재할 수 있음

 하나의 쓰레드 그룹의 쓰레드의 개수와 구조는 계산 셰이더에서 표현함

 계산 셰이더에서 처리할 자료구조(Data Structure)에 따라 쓰레드 그룹의 개수와 쓰레드 구조를 결정해야 함

계산 셰이더는 하나의 쓰레드 그룹의 여러 쓰레드에서 병렬적으로 실행될 수 있음

생성된 쓰레드 그룹은 무작위로 디스패치되고 하나의 쓰레드 그룹의 쓰레드들도 무작위로 실행됨

특성 레벨(Feature Level) 10(셰이더 모델 4.x)에서 ThreadGroupCountZ는 1이어야 함

numthreads(X, Y, Z)

하나의 쓰레드 그룹에서 실행될 쓰레드들의 구조와 개수를 정의

$X \times Y \times Z$ 개의 쓰레드가 생성됨 (셰이더 모델 5.0: $1 \leq Z \leq 64$, $1 \leq X \times Y \times Z \leq 1024$)

각 쓰레드가 논리적으로 2D 또는 3D 자료구조를 사용(접근)할 수 있음

계산 셰이더가 4x4 행렬의 합을 계산한다면 (4, 4, 1)로 설정하는 것이 좋음

[numthreads(2, 2, 1)]

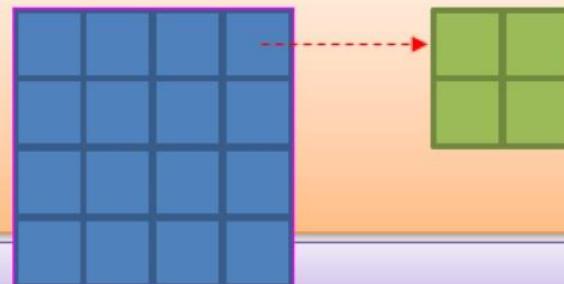
```
void CS(int3 nDispatchID : SV_DispatchThreadID)
```

```
{
```

```
...
```

```
}
```

```
pd3dGraphicsCommandList->Dispatch(4, 4, 1);
```



Direct3D 파이프라인

- 텍스쳐 리소스(Texture Resource) 사용하기

```
[numthreads(16, 16, 1)]
void CS(int3 nDispatchID : SV_DispatchThreadID)
{
    int x = nDispatchID.x, y = nDispatchID.y;
    float tA = gtxtInputA[int2(x, y)].r;
    float tL = gtxtInputB[int2(x-1, y)].r;
    float tR = gtxtInputB[int2(x+1, y)].r;
    float tC = gtxtInputB[int2(x, y)].r;
    float tT = gtxtInputB[int2(x, y-1)].r;
    float tB = gtxtInputB[int2(x, y+1)].r;
    gtxtRWOutput[int2(x, y)] = gfSettings[0] * tA + gfSettings[1] * tC + gfSettings[2] * (tL + tR + tT + tB);
}
```

```
cbuffer cbSettings : register(b3)
{
    float gfSettings[3];
};
```

```
[numthreads(16, 16, 1)]
void CS(int3 nDispatchID : SV_DispatchThreadID)
{
    int x = nDispatchID.x, y = nDispatchID.y;
    float tA = gtxtInputA.SampleLevel(gsSampler, (float2(x, y) / 512.0f), 0.0f).r;
    float tL = gtxtInputB.SampleLevel(gsSampler, (float2(x-1, y) / 512.0f), 0.0f).r;
    float tR = gtxtInputB.SampleLevel(gsSampler, (float2(x+1, y) / 512.0f), 0.0f).r;
    float tC = gtxtInputB.SampleLevel(gsSampler, (float2(x, y) / 512.0f), 0.0f).r;
    float tT = gtxtInputB.SampleLevel(gsSampler, (float2(x, y-1) / 512.0f), 0.0f).r;
    float tB = gtxtInputB.SampleLevel(gsSampler, (float2(x, y+1) / 512.0f), 0.0f).r;
    gtxtRWOutput[int2(x, y)] = gfSettings[0] * tA + gfSettings[1] * tC + gfSettings[2] * (tL + tR + tT + tB);
}
```

```
Texture2D gtxtInputA : register(t0);
Texture2D gtxtInputB : register(t1);
RWTexture2D<float> gtxtRWOutput : register(u0);
```

Direct3D 파이프라인

- 계산-쉐이더(Compute-Shader)

numthreads(X, Y, Z)	<p>하나의 쓰레드 그룹에서 실행될 쓰레드의 개수를 정의 X*Y*Z 개의 쓰레드 ($1 \leq Z \leq 64$, $1 \leq X*Y*Z \leq 1024$) 각 쓰레드가 논리적으로 2D 또는 3D 자료구조를 사용(접근)할 수 있음 계산 세이더가 4x4 행렬의 합을 계산한다면 (4, 4, 1)로 설정하는 것이 좋음</p>
----------------------------	---

numthreads(10, 8, 3)

각 쓰레드 그룹은 $10 \times 8 \times 3 = 240$ 개의 쓰레드로 구성

(16, 1, 1) ?

Dispatch(5,3,2)

SV_GroupID

X					Y	5v_Group
(0,0,0)	(1,0,0)	(2,0,0)	(3,0,0)	(4,0,0)	1)	
(0,1,0)	(1,1,0)	(2,1,0)	(3,1,0)	(4,1,0)	1)	
(0,2,0)	(1,2,0)	(2,2,0)	(3,2,0)	(4,2,0)	1)	
(0,2,1)	(1,2,1)	(2,2,1)	(3,2,1)	(4,2,1)		

쓰레드 그룹

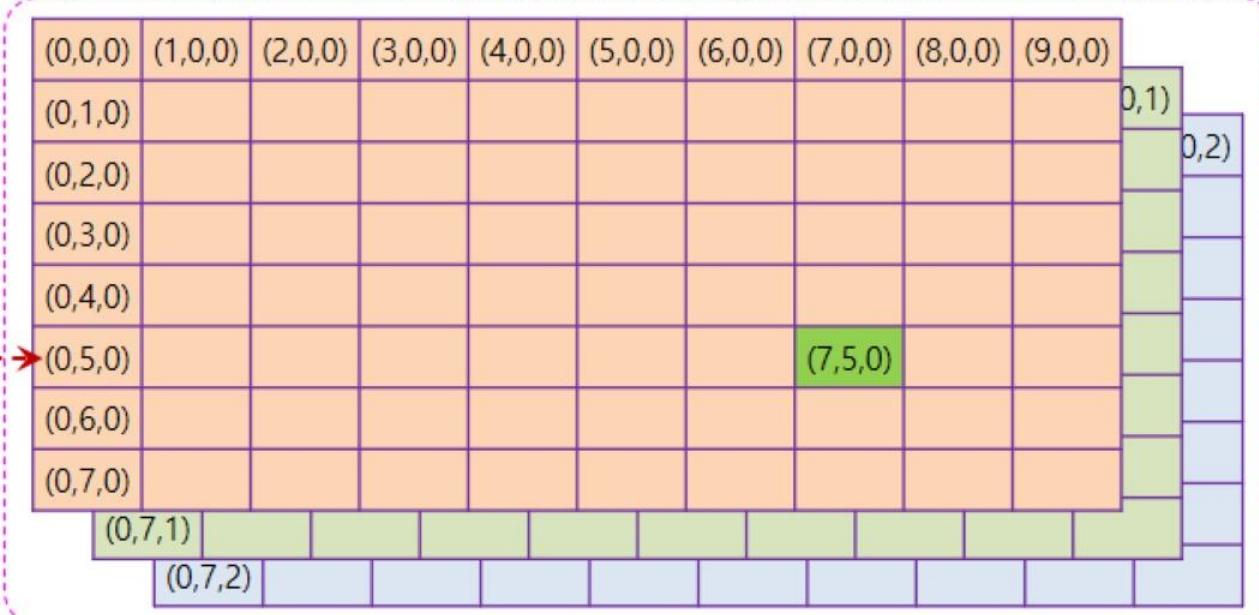
SV_GroupThreadID : (7,5,0)

SV_GroupID : (2,1,0)

$$\text{SV_DispatchThreadID} : (2,1,0) * (10,8,3) + (7,5,0) = (27,13,0)$$

SV_GroupIndex : $0*10^8 + 5*10 + 7 = 57$

SV_GroupThreadID



Direct3D 파이프라인

- 계산-쉐이더(Compute-Shader)

SV_GroupThreadID

쓰레드 그룹에서 계산 셰이더가 실행되고 있는 **쓰레드의 인덱스**(uint3)
numthreads(3,2,1)의 경우 가능한 인덱스는 (0,0,0) ~ (2,1,0) 범위
일반적인 행렬의 인덱스와 다름(첫 번째 인덱스는 열(x), 두 번째 인덱스는 행(y))

numthreads(10, 8, 3)

각 쓰레드 그룹은 $10 \times 8 \times 3 = 240$ 개의 쓰레드로 구성

Dispatch(5,3,2)

SV_GroupID

	X	
		SV_GroupID
(0,0,0)	(1,0,0)	(2,0,0)
(3,0,0)	(4,0,0)	
(0,1,0)	(1,1,0)	(2,1,0)
(3,1,0)	(4,1,0)	1)
(0,2,0)	(1,2,0)	(2,2,0)
(3,2,0)	(4,2,0)	1)
(0,2,1)	(1,2,1)	(2,2,1)
(3,2,1)	(4,2,1)	

SV_GroupThreadID : (7,5,0)

SV_GroupID : (2,1,0)

SV_DispatchThreadID : $(2,1,0) \times (10,8,3) + (7,5,0) = (27,13,0)$

SV_GroupIndex : $0 \times 10 \times 8 + 5 \times 10 + 7 = 57$

쓰레드 그룹

(0,0,0)	(1,0,0)	(2,0,0)	(3,0,0)	(4,0,0)	(5,0,0)	(6,0,0)	(7,0,0)	(8,0,0)	(9,0,0)			
(0,1,0)										0,1)		
(0,2,0)										0,2)		
(0,3,0)												
(0,4,0)												
(0,5,0)									(7,5,0)			
(0,6,0)												
(0,7,0)												
(0,7,1)												
(0,7,2)												

SV_GroupThreadID

Direct3D 파이프라인

- 계산-쉐이더(Compute-Shader)

SV_GroupID	계산 셰이더가 실행하고 있는 쓰레드 그룹의 인덱스 (uint3) Dispatch() 함수 호출의 파라메터로 정해짐 Dispatch(2,1,1)의 경우 가능한 인덱스는 (0,0,0) 또는 (1,0,0)
-------------------	--

numthreads(10, 8, 3)

각 쓰레드 그룹은 $10 \times 8 \times 3 = 240$ 개의 쓰레드로 구성

Dispatch(5,3,2)

SV_GroupID

X					Y	5v_Group
(0,0,0)	(1,0,0)	(2,0,0)	(3,0,0)	(4,0,0)	1)	
(0,1,0)	(1,1,0)	(2,1,0)	(3,1,0)	(4,1,0)		
(0,2,0)	(1,2,0)	(2,2,0)	(3,2,0)	(4,2,0)		
(0,2,1)	(1,2,1)	(2,2,1)	(3,2,1)	(4,2,1)		

SV GroupThreadID : (7,5,0)

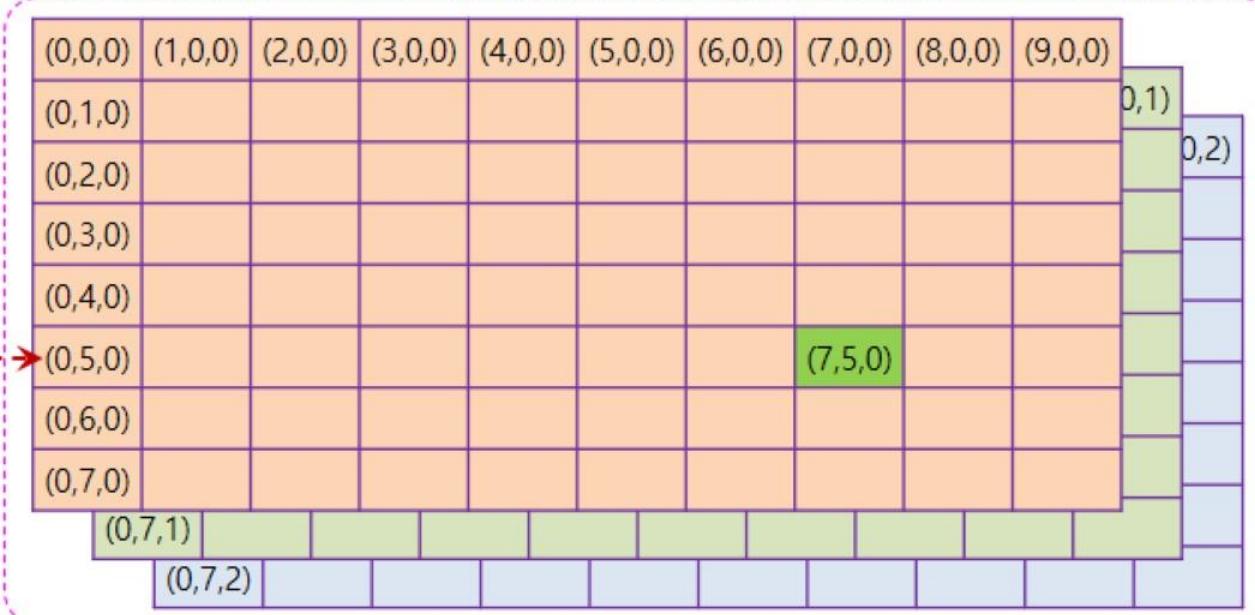
SV_GroupID : (2,1,0)

$$\text{SV_DispatchThreadID} : (2,1,0) * (10,8,3) + (7,5,0) = (27,13,0)$$

SV_GroupIndex : $0*10^8 + 5*10 + 7 = 57$

쓰레드 그룹

SV_GroupThreadId



Direct3D 파이프라인

- 계산-쉐이더(Compute-Shader)

SV_DispatchThreadID	계산 세이더가 실행하고 있는 쓰레드와 쓰레드 그룹이 결합된 인덱스(uint32) Dispatch() 호출로 생성된 모든 쓰레드를 구별할 수 있음 SV_GroupID * numthreads + SV_GroupThreadID
----------------------------	--

numthreads(10, 8, 3)

각 쓰레드 그룹은 $10 \times 8 \times 3 = 240$ 개의 쓰레드로 구성

SV GroupThreadID : (7,5,0)

SV GroupID : (2,1,0)

SV_DispatchThreadID : $(2, 1, 0) * (10, 8, 3) + (7, 5, 0) = (27, 13, 0)$

SV_GroupIndex : $0*10^8 + 5*10 + 7 = 57$

쓰레드 그룹

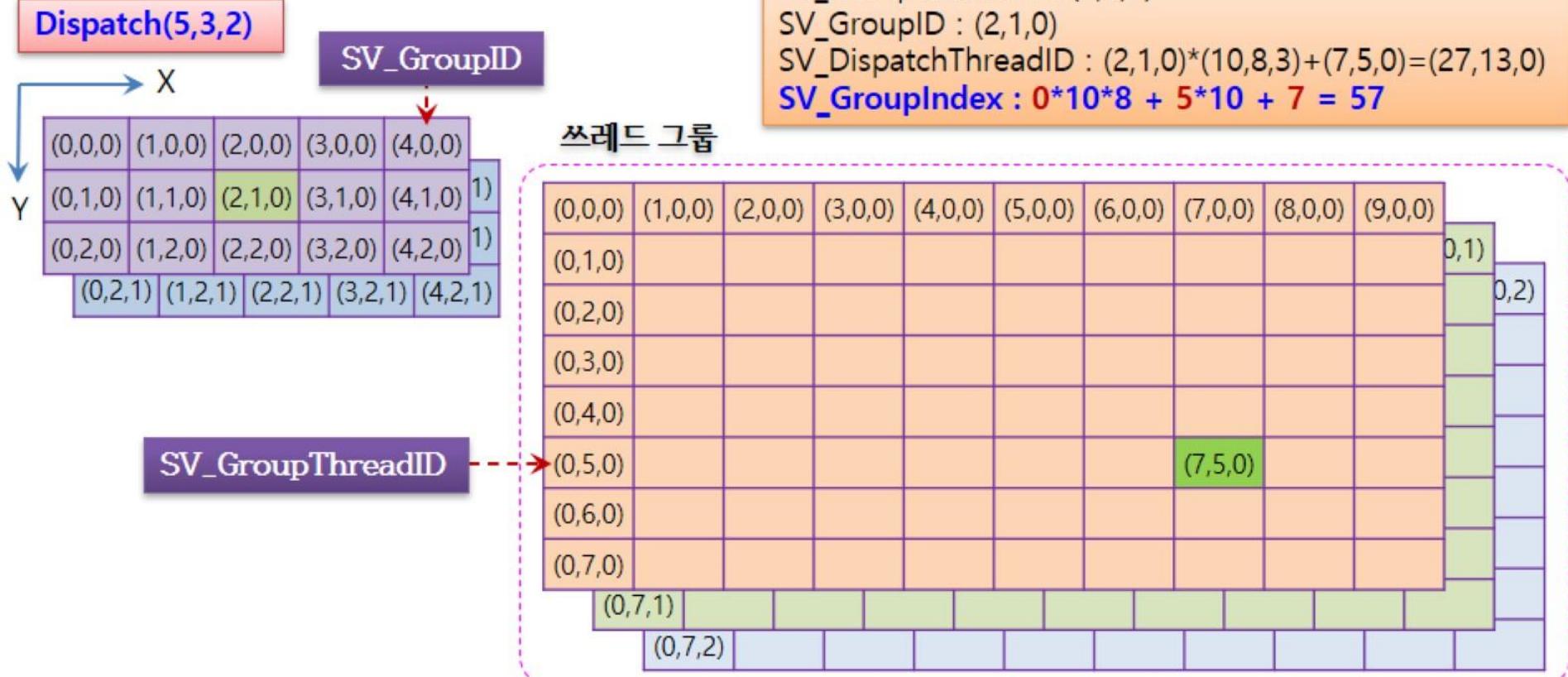
Direct3D 파이프라인

- 계산-쉐이더(Compute-Shader)

SV_GroupIndex	하나의 쓰레드 그룹에서 쓰레드의 인덱스(uint) numthreads(X, Y, Z) $(SV_{GroupThreadID.z} * X * Y) + (SV_{GroupThreadID.y} * X) + SV_{GroupThreadID.x}$
---------------	--

numthreads(10, 8, 3)

각 쓰레드 그룹은 $10 * 8 * 3 = 240$ 개의 쓰레드



Direct3D 파이프라인

• GPU 메모리에서 시스템 메모리로 리소스 복사하기

- 일반적으로 계산 셰이더는 텍스쳐를 처리(계산)하여 화면으로 출력하려고 사용
- 계산 셰이더 실행의 결과는 화면 출력의 목적이 아닌 경우가 있음(예: 파일로 저장)
GPU 계산(계산 셰이더 실행)의 결과는 GPU 메모리에 저장됨
GPU 메모리에서 시스템(CPU) 메모리로 내용을 복사해야 하는 경우가 있음
- GPU 메모리에서 시스템 메모리로 복사하기
 - ① CPU에서 읽을 수 있는 버퍼를 생성
D3D12_HEAP_TYPE_READBACK
 - ② ID3D12GraphicsCommandList::**CopyResource()**로 리소스를 버퍼로 복사
 - ③ 버퍼를 ID3D12Resource::**Map()** 함수로 맵핑하여 읽음

```
typedef struct D3D12_HEAP_PROPERTIES {
    D3D12_HEAP_TYPE Type; //힙의 유형
    D3D12_CPU_PAGE_PROPERTY CPUPageProperty; //힙에 대한 CPU-페이지 속성
    D3D12_MEMORY_POOL MemoryPoolPreference; //힙에 대한 메모리 풀(Pool)
    UINT CreationNodeMask; //0, 1: 단일 GPU 어댑터
    UINT VisibleNodeMask; //다중 어댑터에서 리소스를 볼 수 있는 노드의 집합을 나타내는 비트
} D3D12_HEAP_PROPERTIES;
```

```
typedef enum D3D12_HEAP_TYPE {
    D3D12_HEAP_TYPE_DEFAULT, //GPU는 읽고 쓰기 가능, CPU는 접근할 수 없음, 대부분의 리소스에 적용
    D3D12_HEAP_TYPE_UPLOAD, //업로드를 위한 CPU 접근에 최적화, _RESOURCE_STATE_GENERIC_READ
    D3D12_HEAP_TYPE_READBACK, //읽기 위한 힙, CPU 접근에 최적화, _RESOURCE_STATE_COPY_DEST
    D3D12_HEAP_TYPE_CUSTOM
} D3D12_HEAP_TYPE;
```

Direct3D 파이프라인

- GPU 메모리에서 시스템 메모리로 리소스 복사하기

```
void ID3D12DeviceContext::CopyResource(  
    [in] ID3D12Resource *pDstResource,  
    [in] ID3D12Resource *pSrcResource  
)
```

pDstResource	목표 리소스
pSrcResource	복사할 소스 리소스

GPU가 소스 리소스 **전체**를 목표 리소스로 복사함(GPU memcpy() 처럼 동작)

리소스는 달라야 함, 같은 형식(DXGI 호환 가능해야 함), 같은 크기, 매핑되어 있지 않아야 함
내용을 복사하기만 함(확대/축소, 형식 변환 등이 지원되지 않음)

D3D12_USAGE_IMMUTABLE, 깊이-스텐실 리소스는 목표 리소스가 될 수 없음
다중 샘플링을 할 수 있도록 생성된 리소스는 소스 또는 목표 리소스가 될 수 없음

```
void ID3D12DeviceContext::CopySubresourceRegion(  
    [in] ID3D12Resource *pDstResource,  
    [in] UINT DstSubresource,  
    [in] UINT DstX,  
    [in] UINT DstY,  
    [in] UINT DstZ,  
    [in] ID3D12Resource *pSrcResource,  
    [in] UINT SrcSubresource,  
    [in] const D3D12_BOX *pSrcBox  
)
```

pDstResource	목표 리소스
DstSubresource	목표 서브리소스 인덱스
DstX	목표 영역의 좌측 상단의 x-좌표
DstY	목표 영역의 좌측 상단의 y-좌표, 1D(0)
DstZ	목표 영역의 좌측 상단의 z-좌표, 1D(0), 2D(0)
pSrcResource	복사할 소스 리소스
SrcSubresource	복사할 소스 리소스
pSrcBox	복사할 영역(D3D12_BOX), NULL(전체)

GPU가 소스 리소스의 **일부 영역**을 목표 리소스로 복사함

리소스는 달라야 함, 같은 형식(DXGI 호환 가능해야 함), 같은 크기, 매핑되어 있지 않아야 함
내용을 복사하기만 함(확대/축소, 형식 변환 등이 지원되지 않음)

Direct3D 파이프라인

- **리소스의 갱신(ID3D12Resource::Map() 함수의 사용)**

- 리소스(서브리소스)에 대한 CPU 포인터를 반환(다중-쓰레드 안전)
 - **D3D12_HEAP_TYPE_DEFAULT** 힙
Map() 호출을 할 수 없음(D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE)
 - **D3D12_HEAP_TYPE_UPLOAD** 힙
리소스 포인터에서 읽기를 하지 말 것(성능 문제가 발생)
영구적으로 매핑을 할 수 있음(Unmap()을 호출하지 않을 수 있음)
 - **D3D12_HEAP_TYPE_READBACK**
영구적 매핑을 허용하지 않음(CPU와 GPU 접근 사이에 Unmap을 호출해야 함)

HRESULT ID3D12Resource::Map(

 UINT Subresource, //서브리소스 인덱스, 버퍼(0)

D3D12_RANGE *pReadRange, //CPU가 접근할 메모리 영역, NULL(전체 서브리소스)

 void **ppData //서브리소스에 대한 포인터(서브리소스의 시작 주소), NULL(CPU 가상 주소를 캐싱)

);

void ID3D12Resource::Unmap(

 UINT Subresource, //서브리소스 인덱스, 버퍼(0)

D3D12_RANGE *pWrittenRange //CPU가 수정한 메모리 영역

);

```
typedef struct D3D12_RANGE {
    SIZE_T Begin; //시작 바이트
    SIZE_T End; //종료 바이트(끝+1)
} D3D12_RANGE;
```

UINT D3D12CalcSubresource(UINT MipSlice, UINT ArraySlice, ...); //서브리소스 인덱스 계산

UINT8 *pDataBegin;

m_pd3dReadBackBuffer->Map(0, NULL, reinterpret_cast<void **>(&pDataBegin));

UINT8 *pSystemMemory = ...;

memcpy(pSystemMemory, pDataBegin, sizeof(pSystemMemory));

m_pd3dReadBackBuffer->Unmap(0, NULL);

Direct3D 파이프라인

- **공유 메모리 동기화(Shared Memory Synchronization)**

- 쓰레드 그룹은 공유 메모리 또는 쓰레드 로컬 메모리(TLS)를 가짐
공유 메모리는 쓰레드 그룹의 로컬 메모리(쓰레드 그룹에서 공유)
SV_GroupThreadID로 인덱싱(그룹의 각 쓰레드가 하나의 슬롯을 사용할 수 있음)
하드웨어 캐시(Cache)처럼 빠르게 접근할 수 있음
쓰레드 로컬 메모리의 최대 크기는 32KB(세이더 모델 5.0)
- 세이더 변수를 선언할 때 [Storage_Class]를 **groupshared**로 설정

```
groupshared GROUPSHARED gSharedCache[5*5*1];
```

```
[numthreads(5, 5, 1)]
void CS(uint nGroupIndex : SV_GroupIndex)
{
    gSharedCache[nGroupIndex].color = (float4)0;
    GroupMemoryBarrierWithGroupSync();
    ...
}
```

```
struct GROUPSHARED
{
    float4 color;
    float factor;
}
```

공유 메모리가 너무 크면 병렬성에 제한이 발생(왜?)

텍스쳐 샘플링은 느린 연산이므로 미리 로드하여 사용하면?

```
Texture2D gTexture : register(t3);
groupshared float4 gSharedCache[256];
```

```
[numthreads(256, 1, 1)]
void CS(int3 nGroupThreadID : SV_GroupThreadID, int3 nDispatchThreadID : SV_DispatchThreadID)
{
    gSharedCache[nGroupThreadID.x] = gTexture[nDispatchThreadID.xy];
    float4 cLeft = gSharedCache[nGroupThreadID.x - 1];
    ...
}
```

Direct3D 파이프라인

- 공유 메모리 동기화(Shared Memory Synchronization)

- 쓰레드 그룹의 쓰레드들이 공유 메모리를 사용할 때 동기화 필요

```
void GroupMemoryBarrierWithGroupSync(void);
```

쓰레드 그룹의 모든 그룹 **공유** 접근(쓰기)이 끝날 때까지 그리고 그룹의 모든 쓰레드가 이 함수를 호출할 때까지 각 쓰레드의 실행을 멈추고 기다림

```
void GroupMemoryBarrier(void);
```

모든 그룹 **공유** 접근(쓰기)이 끝날 때까지 하나의 쓰레드 그룹의 모든 쓰레드의 실행을 멈추고 기다림

```
Texture2D gTexture : register(t3);  
groupshared float4 gSharedCache[256];
```

```
[numthreads(256, 1, 1)]
```

```
void CS(int3 nGroupThreadID : SV_GroupThreadId, int3 nDispatchThreadID : SV_DispatchThreadId)  
{
```

```
    gSharedCache[nGroupThreadID.x] = gTexture[nDispatchThreadID.xy];
```

```
GroupMemoryBarrierWithGroupSync();
```

```
    float4 cLeft = gSharedCache[nGroupThreadID.x - 1];
```

```
    float4 cRight = gSharedCache[nGroupThreadID.x + 1];
```

```
...
```

```
}
```

모든 메모리 접근이 끝날 때까지 그룹의 모든 쓰레드 실행을 멈추고 기다림

```
void AllMemoryBarrier(void);
```

```
void AllMemoryBarrierWithGroupSync(void);
```

Direct3D 파이프라인

• 인터락 변수 접근(Interlocked Variable Access)

- 여러 쓰레드가 공유하는 변수에 대한 접근을 동기화해야 함
공유하는 변수에 대한 연산이 부분적으로 이루어지지 말아야 함(Atomically)
공유하는 변수에 대한 읽기와 쓰기 연산에 대한 접근 순서가 동기화되어야 함
- 아토믹 연산(Atomic Operation)
쉐이더 모델 5 이상에서 동기화 함수를 사용할 수 있음
모든 쉐이더 단계에 대하여 사용할 수 있음
공유 메모리의 변수에 대한 읽기와 쓰기는 모든 비트에 대하여 완전하게 이루어짐
리소스 변수에 대한 읽기와 쓰기는 모든 비트에 대하여 완전하게 이루어짐
- 인터락 함수(Interlocked Function)
여러 쓰레드가 공유하는 변수에 대한 접근을 동기화하는 아토믹 함수를 제공
하나의 쓰레드가 공유 변수에 대한 연산이나 접근을 하면 나머지 쓰레드는 대기함

```
void InterlockedAdd(in R dest, in T value, out T original_value); //int, uint
void InterlockedAnd(in R dest, in T value, out T original_value); //int, uint
void InterlockedExchange(in R dest, in T value, out T original_value); //스칼라 자료형
void InterlockedCompareExchange(in R dest, in T compare_value, in T value, out T original_value); //int, uint
void InterlockedCompareStore(in R dest, in T compare_value, in T value); //int, uint
void InterlockedMax(in R dest, in T value, out T original_value); //int, uint
void InterlockedMin(in R dest, in T value, out T original_value); //int, uint
void InterlockedOr(in R dest, in T value, out T original_value); //int, uint
void InterlockedXor(in R dest, in T value, out T original_value); //int, uint
```

계산 쉐이더의 루프문에서 **InterlockedCompareExchange()**를 호출하면 **[allow_uav_condition]**를 사용해야 함