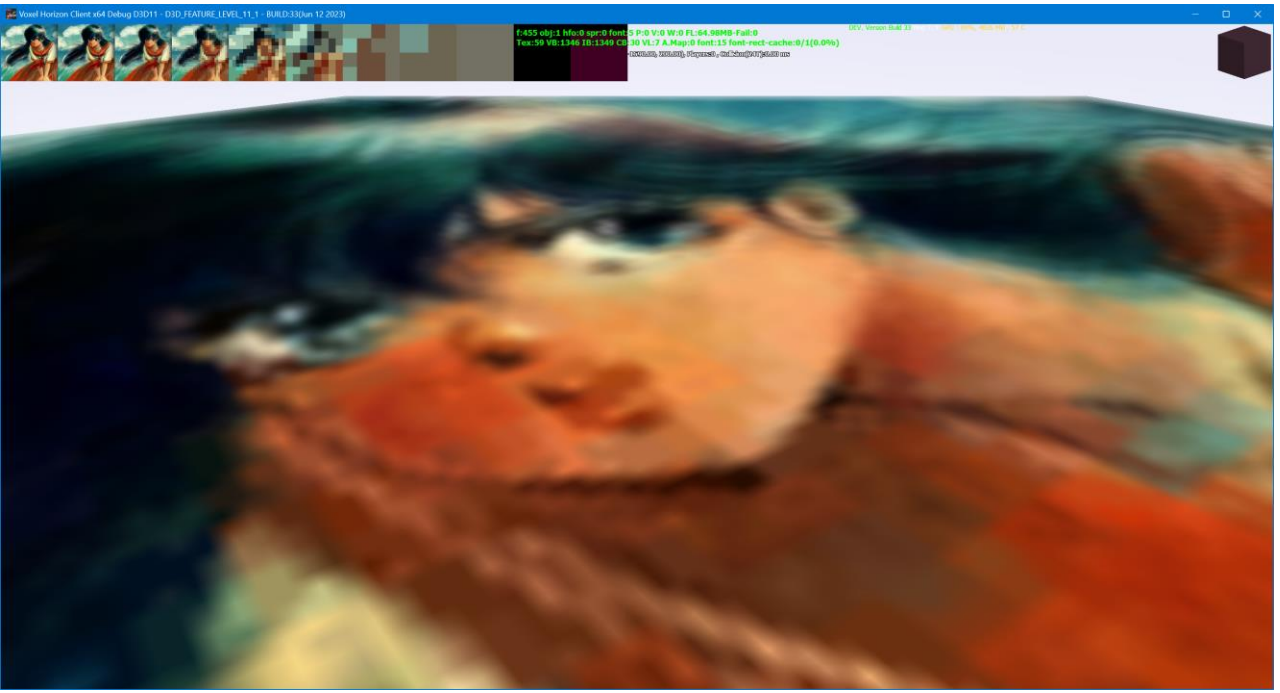
D3D Tiled resources를 이용한 텍스처 스트리밍

https://megayuchi.com
유영천

512 M x 512 M 영역에 맵핑한 텍스처

256x256



16384x16384

- 당연히 넓은 영역에는 큰 텍스처를 맵핑하는 쪽이 화질이 좋다.
- 현실적으로 Height Field구현할 때 10cm x 10cm 면적에 1x1 texel정도는 보장해야 한다.
- 그런데 무작정 큰 텍스처를 사용하면 GPU메모리 사용량도 무작정 증가한다.
  - 512 M x 512 M -> 5120 px x 5120 px = 1GB,
  - 1024 M x 1024 M -> 10240 px x 10240 px = 4GB

거대 텍스처 맵핑의 특성

- 언제나 텍스처의 모든 영역이 한 번에 보여지는 것은 아니다.
- 텍스처의 모든 영역이 한 번에 보여질 경우 최대 정밀도가 필요한 것은 아니다.

텍스처를 ‘필요한만큼의 크기’만, ‘필요한 때’에 사용할 수 없을까?

- 붉은색 영역은 GPU메모리를 점유하지 않음.
- 16384x16384짜리 텍스처를 사용하지만 실제 물리 메모리 사용량은 1/16에도 못미침.

Virtual Texturing

- 현대 CPU + OS의 가상 메모리 시스템과 비슷한 관리체계를 그래픽 프로그래밍에 도입하자.
- 간접주소 ->직접 참조. 기본적으로 2단계 참조
- 이전에는 S/W방식으로 사용. Ex) id soft Rage등…
- 현재는 D3D/Metal등 API에서 지원(당연히 GPU에서도 지원해야 함).

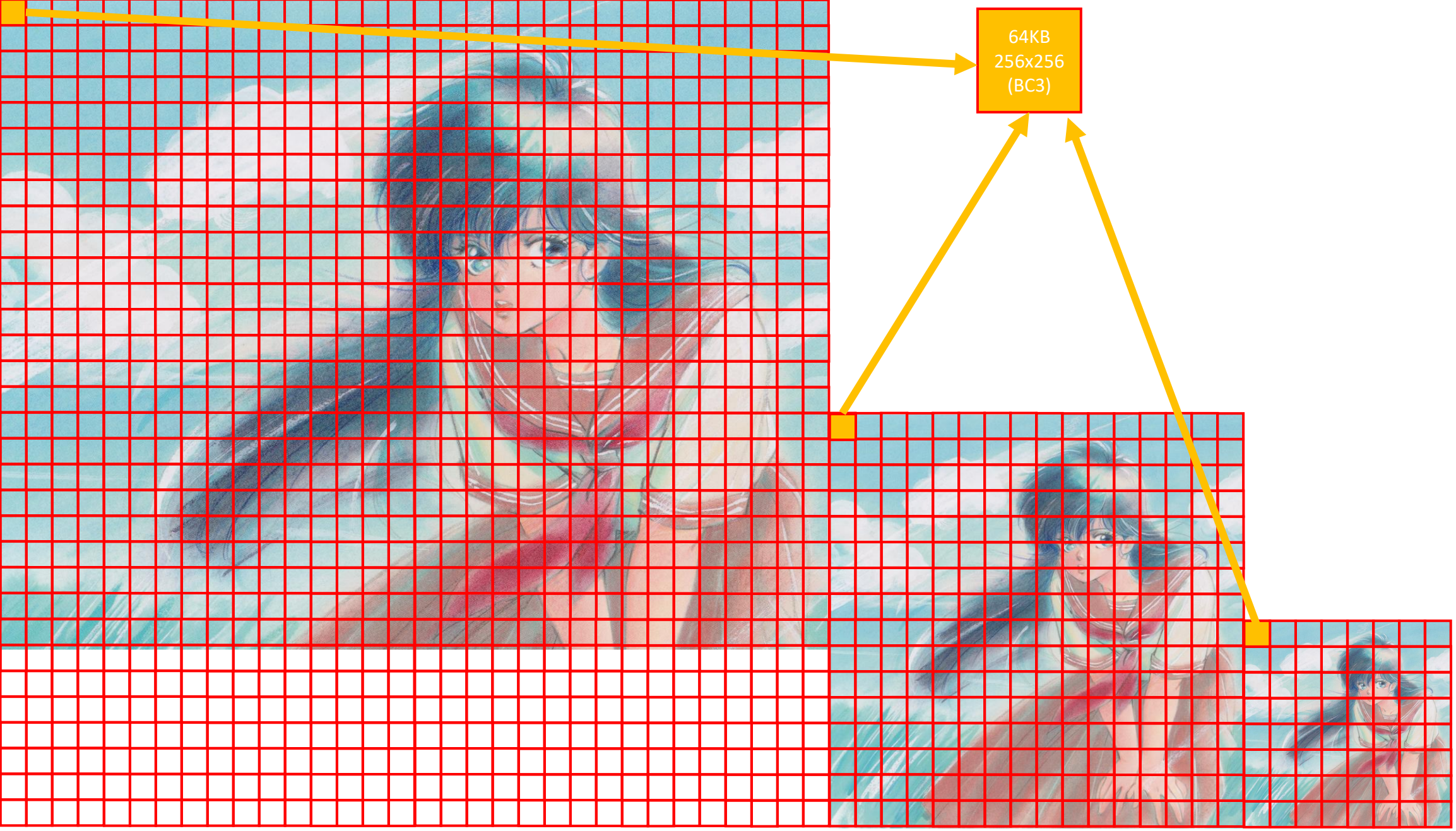
S/W Virtual Texturing

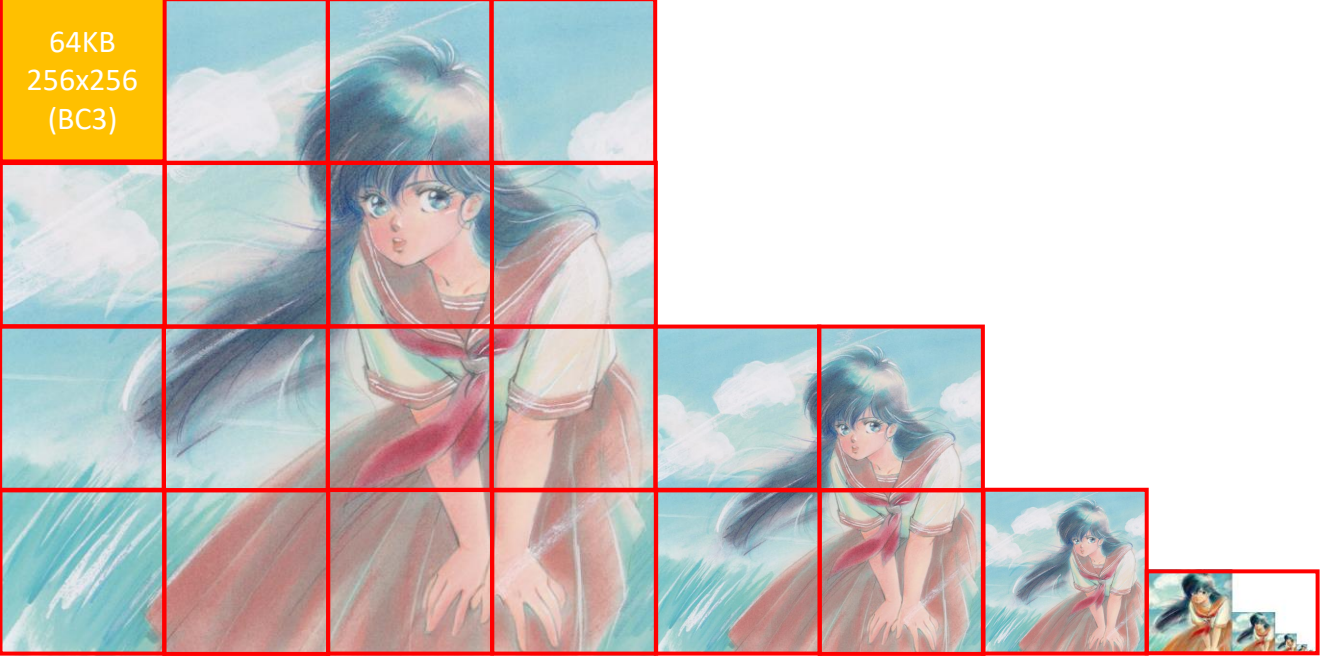
- 1차 texture – Lookup Table, 2차 texture – Tiled Texture
- 어떤 Tiled Texture를 GPU메모리에 올릴지를 결정하기 위해서 실시간으로 프레임 버퍼를 분석해야 함. -> Rage가 CUDA를 사용한 이유
- 인접한 Tile texture는 서로 연관성은 전혀 없다. 이 경우 화면상의 인접한 두 픽셀이 서로 다른 Tile Texture를 사용할 경우 원치 않는 번짐이 발생.
- 반대로 인접한 두 픽셀이 다른 mip단계의 Tiled Texture를 사용할 경우 부드러운 필터링이 쉽지 않음.
- https://www.nvidia.com/content/GTC/posters/37__Hollemeersch_Accelerating_Virtual_Texturing_Using_CUDA.pdf
- https://silverspaceship.com/src/svt/

D3D의 H/W Virtual Texturing

- GPU의 MMU/Page fault처리 기능을 사용
- 주소만 확보해두고 물리 메모리에 맵핑하지 않음.필요에 따라 물리 메모리에 맵핑.
- D3D Tiled Resources가 공식 명칭
- Lookup Table Texture없음.
- 16384x16384 Texture를 Tiled Resource로 만들 경우 플래그 하나만 지정해주면 됨.
- GPU Memory에 commit되지 않은 Tile을 참조하는 경우에도 크래시하지 않는다는 점이 중요.
- Shader에서 Page Fault상태를 알 수 있음.
- 렌더링에 필요한 Tile을 GPU Memory에 commit하고 이미지 데이터를 업데이트하는 것은 프로그래머의 몫.

Tiled Resources


64KB
256x256
(BC3)
8192x8192
32x32(tile)
4096x4096
16x16(tile)
2048x2048
8x8(tile)



1024x1024
4x4(tile)

512x512
2x2(tile)

256x256
1x1(tile)

128x128
64x64
32x32
16x16
8x8
4x4
2x2
1x1
1x1(tile)

Non Packed Mip/Packed Mip

- Mip slice 1개가 1개 이상의 Tile에 맵핑될 경우 이 Mip Slice는 Non Packed Mip.
- BC3(DXT5)포맷 기준 128x128 – 1x1까지는 모든 mip 단계가 하나의 Tile 안에 포함된다.
- 1개의 mip level에 모두 들어가는 mip slice들을 packed mip이라 부름.
- Packed mip들은 모두 같은 Tile을 참조하므로 하나의 mip단계로 간주해서 처리할 수 있음(128x128 – 1x1을 연속된 비트맵스트림으로 처리한다).
- 따라서 가장 큰 16384x16384 텍스처에서 필요한 mip 단계는 0-7까지 3 bits로 표현 가능.
- Packed mip들은 한번에 Tile에 맵핑한다.

기본적인 API사용법

Tiled Resources생성

```
D3D11_TEXTURE2D_DESC    desc;
memset(&desc, 0, sizeof(desc));
desc.Width = Width;
desc.Height = Height;
desc.ArraySize = 1;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.MiscFlags = D3D11_RESOURCE_MISC_TILED;
desc.Format = format;
desc.MipLevels = MipLevels;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;

ID3D11Texture2D*    pTex2D = nullptr;
HRESULT hr = pDevice->CreateTexture2D(&desc, nullptr, &pTex2D);

if (FAILED(hr))
    __debugbreak();
```

Tile Pool 생성

```
ID3D11Buffer* pTilePool = nullptr;

// Create the tile pool.
D3D11_BUFFER_DESC tilePoolDesc;
ZeroMemory(&tilePoolDesc, sizeof(tilePoolDesc));
//tilePoolDesc.ByteWidth = 65536 * m_NumTilesForEntireResource;
tilePoolDesc.ByteWidth = m_dwMaxTileNum * 65536;
tilePoolDesc.Usage = D3D11_USAGE_DEFAULT;
tilePoolDesc.MiscFlags = D3D11_RESOURCE_MISC_TILE_POOL;

if (FAILED(pD3DDevice->CreateBuffer(&tilePoolDesc, nullptr, &pTilePool)))
{
    __debugbreak();
}
```

Tiled Resources의 각 Tile들을 GPU Memory에 commit

```
BOOL CTiledTexture::UpdateTileMappingPerMipWithTilePos(UINT TilePosX, UINT TilePosY, UINT MipLevel)
{
    BOOL bResult = FALSE;
    ID3D11DeviceContext2* pImmContext2 = m_pRenderer->INL_GetImmContext2();

    const INSPECT_MIP_DATA* pMipData = m_pMipLayout->MipData + MipLevel;

    UINT subresourceCount = 1;
    D3D11_TILED_RESOURCE_COORDINATE TRC = {};
    D3D11_TILE_REGION_SIZE  TRS = {};

    TRC.Subresource = MipLevel;

    if (pMipData->Packed)
    {
        // Packed mip
        TRS.bUseBox = FALSE;
        TRS.Width = 0;
        TRS.Height = 0;
        TRS.Depth = 0;
        TRS.NumTiles = 1;
        subresourceCount = m_PackedMipDesc.NumPackedMips;
    }
    else
    {
        // Non Packed mip
        TRS.bUseBox = TRUE;
        TRS.Width = 1;
        TRS.Height = 1;
        TRS.Depth = 1;
        TRS.NumTiles = TRS.Width * TRS.Height * TRS.Depth;
        subresourceCount = 1;
    }

    TRC.X = TilePosX;
    TRC.Y = TilePosY;
    TRC.Z = 0;

    UINT StartOffset = dwIndex;
    //D3D11_TILE_MAPPING_NULL
    if (FAILED(pImmContext2->UpdateTileMappings(pTex, 1, pTRC, pTRS, m_pTilePool, 1, nullptr, &StartOffset, nullptr, D3D11_TILE_MAPPING_NO_OVERWRITE)))
    {
        __debugbreak();
    }
```

Tiled Resources의 각 Tile들에 대해 이미지 데이터 업데이트 (Non packed mip)

```
void CTiledTexture::UpdateTilePerNonPackedMipWithTilePos(UINT TilePosX, UINT TilePosY, UINT MipLevel, const BYTE* pBits, UINT RowPitch, UINT SlicePitch)
{
    ID3D11DeviceContext2* pImmContext2 = m_pRenderer->INL_GetImmContext2();

    const INSPECT_MIP_DATA* pMipData = m_pMipLayout->MipData + MipLevel;

    D3D11_TILED_RESOURCE_COORDINATE TRC = {};
    D3D11_TILE_REGION_SIZE  TRS = {};

    TRC.Subresource = MipLevel;

    // Non packed mip
    TRS.bUseBox = TRUE;
    TRS.Width = 1;
    TRS.Height = 1;
    TRS.Depth = 1;
    TRS.NumTiles = TRS.Width * TRS.Height * TRS.Depth;

    TRC.X = TilePosX;
    TRC.Y = TilePosY;
    TRC.Z = 0;


    // 업데이트 된 타일이 없거나 부분적으로 업데이트 된 경우만 업데이트
    pImmContext2->TiledResourceBarrier(nullptr, nullptr);
    pImmContext2->UpdateTiles(m_pTiledResource, &TRC, &TRS, pBits, D3D11_TILE_COPY_NO_OVERWRITE);
}
```

Tiled Resources의 각 Tile들에 대해 이미지 데이터 업데이트 (Packed mip)

```
void CTiledTexture::UpdateTilePerPackedMip(UINT MipLevel, const BYTE* pBits, UINT RowPitch, UINT SlicePitch)
{
    ID3D11DeviceContext2* pImmContext2 = m_pRenderer->INL_GetImmContext2();
    pImmContext2->UpdateSubresource1(m_pTiledResource, MipLevel, nullptr, pBits, RowPitch, SlicePitch, D3D11_COPY_NO_OVERWRITE);
}
```

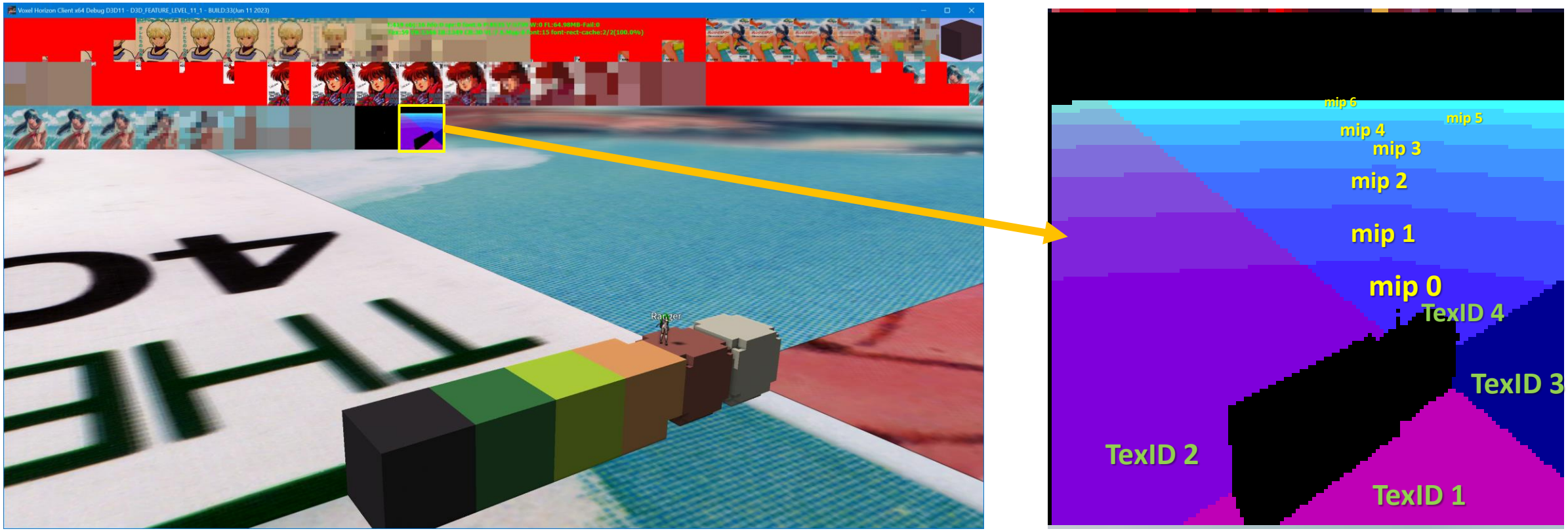
Tiled Resources Status Buffer 구성

Tile 정보 encoding

Tiled Resources Status Buffer 구성

- Deferred Rendering을 위한 Render Target에 1장의 RTV를 추가. 이하 Tiled Resources Status Buffer로 명명.
- Tiled Resource를 사용하는 경우 constant buffer로 전달하는 material정보에 Tex ID를 기입.
- Shader에서 material정보의 Tex ID가 0이 아닐 경우 Tiled Resource Status Buffer에 Tex ID, Tile 좌표, mip 번호, layout 유형을 32 bits 정수로 기록.

Tiled Resource Status Buffer



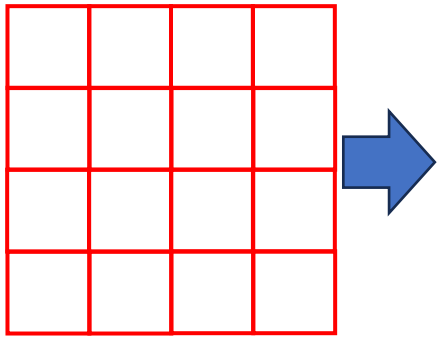
Tile ID, Tile layout 유형, Mip레벨, Tile 좌표를 픽셀당 32 Bits 정수값으로 저장

PageFault(1)	Mip Level(3)	Reserved(1)	Layout Type(3)	Tex ID(12)	Tile X(6)	Tile Y(6)
0 / 1	0-7		0-7	1 - 4095	0 - 63	0 - 63

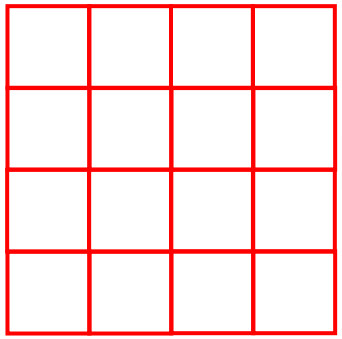
동작 수순

Texture 생성

CreateTexture2D

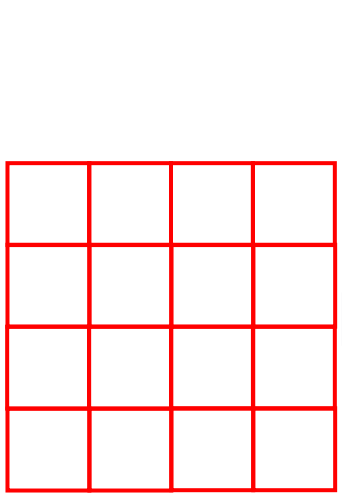


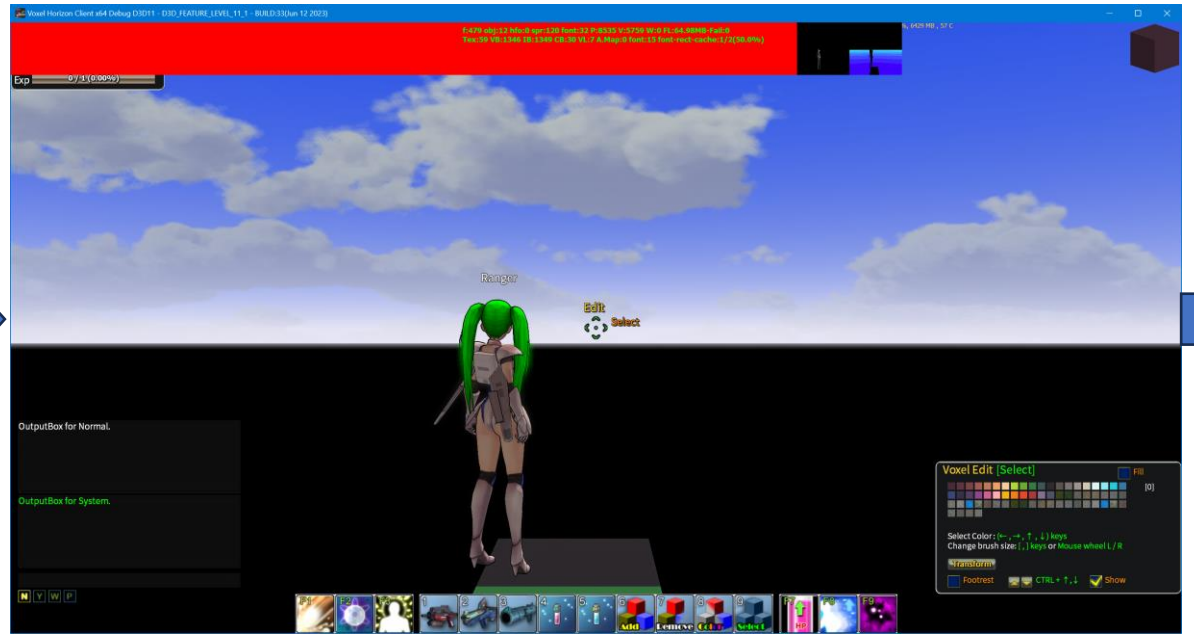
렌더링

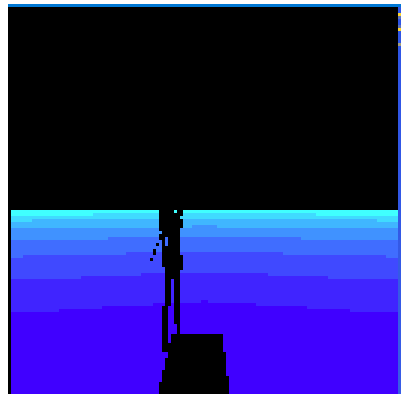



Ranger
Edit
Select
OutputBox for Normal.
OutputBox for System.
Voxel Edit [Select]
Fill
Select Color: (←, →, ↑, ↓) keys
Change brush size: [,] keys or Mouse wheel L / R
Transform
Footrest
CTRL + ↑, ↓
Show

렌더링 이후 Frame Buffer 분석



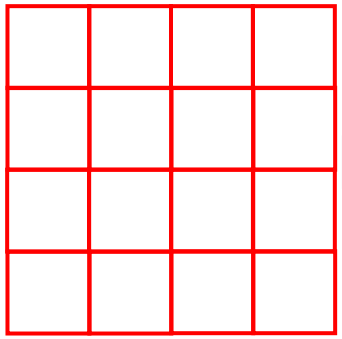


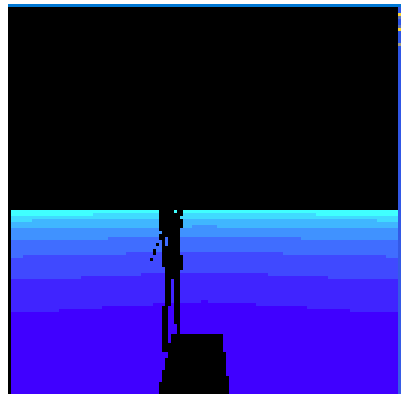


Tiled Resources Status Buffer분석

```
for (DWORD y = 0; y < Height; y++)
{
    for (DWORD x = 0; x < Width; x++)
    {
        UINT Prop = *(UINT*)(pSrc + (x * 4 + y * mappedResource.RowPitch));
        if (Prop)
        {
            UINT PageFault = (Prop & 0x80000000) >> 31;
            UINT MipLevel = (Prop & 0x70000000) >> 28;       // 0 - 7
            UINT TexID = (Prop & 0x00FFF000) >> 12;      // mask = 0b1111111
            if (!TexID)
                __debugbreak();

            UINT Layout = (Prop & 0x07000000) >> 24;

            DWORD dwIndex = TexID - 1;
            CTiledTexture*  pTiledTexture = m_ppTileTextureList[dwIndex];

            UINT TilePosX = (Prop & 0x0000003F);
            UINT TilePosY = (Prop & 0x00000FC0) >> 6;
```

렌더링 이후 분석 – commit/decommit할 Tile 선별





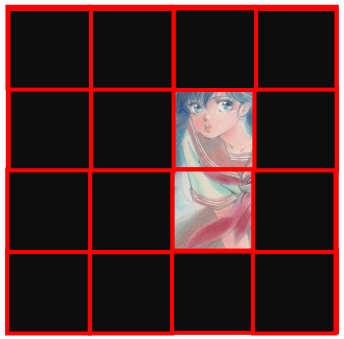
Tiled Resource Status Buffer분석

```
for (DWORD y = 0; y < Height; y++)
{
    for (DWORD x = 0; x < Width; x++)
    {
        UINT Prop = *(UINT*)(pSrc + (x * 4 + y * mappedResource.RowPitch));
        if (Prop)
        {
            UINT PageFault = (Prop & 0x80000000) >> 31;
            UINT MipLevel = (Prop & 0x70000000) >> 28;       // 0 - 7
            UINT TexID = (Prop & 0x00FFF000) >> 12;      // mask = 0b1111111
            if (!TexID)
                __debugbreak();

            UINT Layout = (Prop & 0x07000000) >> 24;

            DWORD dwIndex = TexID - 1;
            CTiledTexture*  pTiledTexture = m_ppTileTextureList[dwIndex];

            UINT TilePosX = (Prop & 0x0000003F);
            UINT TilePosY = (Prop & 0x00000FC0) >> 6;
```

Residency Manager

- TexID:1, mip:0, Tile X:2, Tile Y:1
- TexID:1, mip:0, Tile X:2, Tile Y:2
- TexID:1, mip:1, Tile X:1, Tile Y:0
- TexID:1, mip:1, Tile X:1, Tile Y:1
- TexID:1, mip:2, Tile X:0, Tile Y:0
- TexID:1, mip:3, Tile X:0, Tile Y:0

Tile을 commit 하고 이미지 로딩

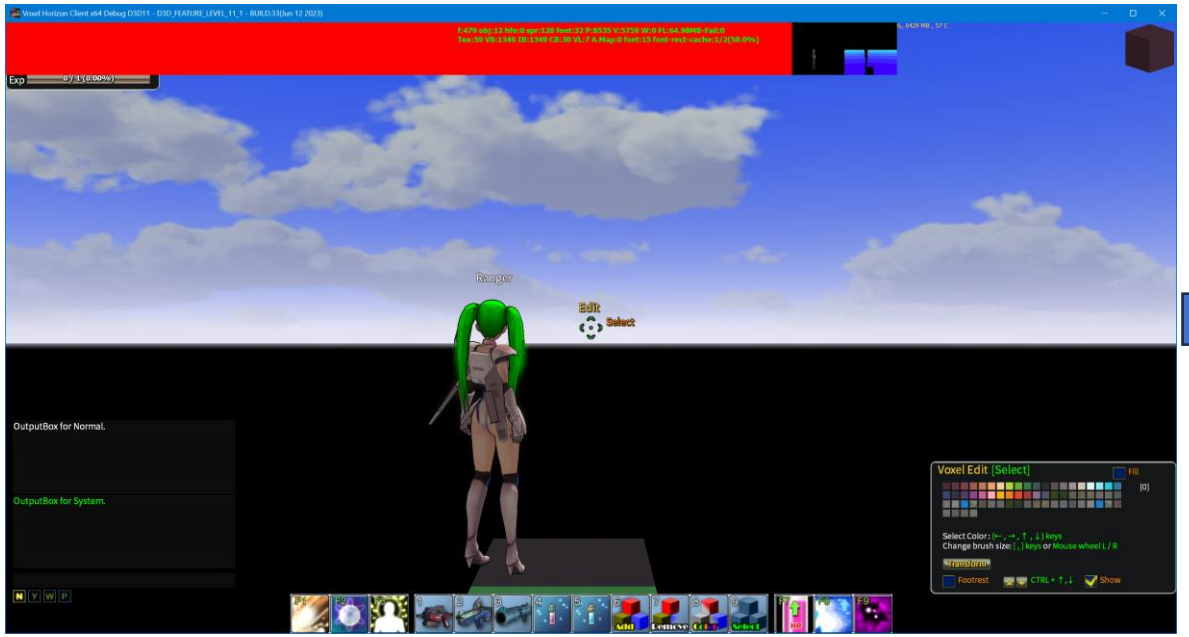


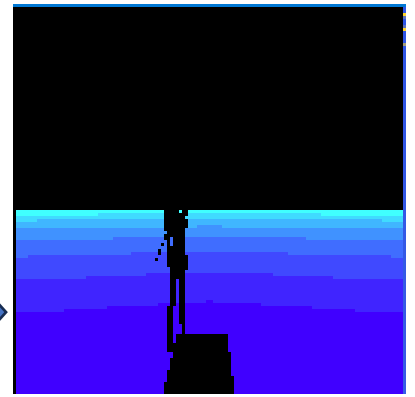
mip 0 (1024x1024)

mip 1 (512x512)

mip 2 (256x256)

mip 3-10 (128x128 – 1x1)





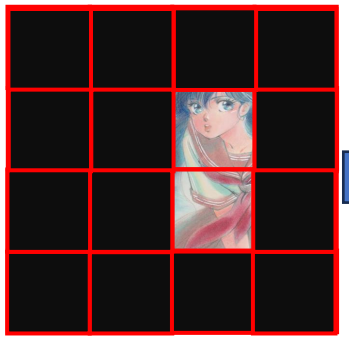
Tiled Resource Status Buffer분석

```
for (DWORD y = 0; y < Height; y++)
{
    for (DWORD x = 0; x < Width; x++)
    {
        UINT Prop = *(UINT*)(pSrc + (x * 4 + y * mappedResource.RowPitch));
        if (Prop)
        {
            UINT PageFault = (Prop & 0x80000000) >> 31;
            UINT MipLevel = (Prop & 0x70000000) >> 28;       // 0 - 7
            UINT TexID = (Prop & 0x00FFF000) >> 12;      // mask = 0b1111111
            if (!TexID)
                __debugbreak();

            UINT Layout = (Prop & 0x07000000) >> 24;

            DWORD dwIndex = TexID - 1;
            CTiledTexture*  pTiledTexture = m_ppTileTextureList[dwIndex];

            UINT TilePosX = (Prop & 0x0000003F);
            UINT TilePosY = (Prop & 0x00000FC0) >> 6;
```

Residency Manager

- TexID:1, mip:0, Tile X:2, Tile Y:1
- TexID:1, mip:0, Tile X:2, Tile Y:2
- TexID:1, mip:1, Tile X:1, Tile Y:0
- TexID:1, mip:1, Tile X:1, Tile Y:1
- TexID:1, mip:2, Tile X:0, Tile Y:0
- TexID:1, mip:3, Tile X:0, Tile Y:0

Tile을 commit 하고 이미지 로딩 후 렌더링

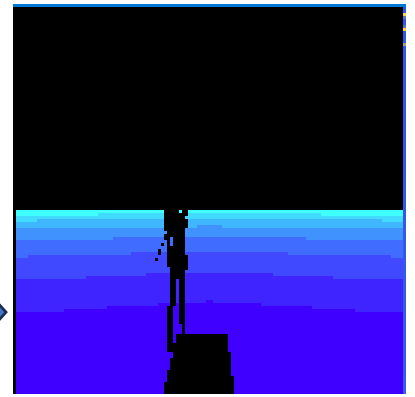


mip 0 (1024x1024)

mip 1 (512x512)

mip 2 (256x256)

mip 3-10 (128x128 – 1x1)



Tiled Resource Status Buffer분석

```
for (DWORD y = 0; y < Height; y++)
{
    for (DWORD x = 0; x < Width; x++)
    {
        UINT Prop = *(UINT*)(pSrc + (x * 4 + y * mappedResource.RowPitch));
        if (Prop)
        {
            UINT PageFault = (Prop & 0x80000000) >> 31;
            UINT MipLevel = (Prop & 0x70000000) >> 28;       // 0 - 7
            UINT TexID = (Prop & 0x00FFF000) >> 12;      // mask = 0b1111111
            if (!TexID)
                __debugbreak();

            UINT Layout = (Prop & 0x07000000) >> 24;

            DWORD dwIndex = TexID - 1;
            CTiledTexture*  pTiledTexture = m_ppTileTextureList[dwIndex];

            UINT TilePosX = (Prop & 0x0000003F);
            UINT TilePosY = (Prop & 0x00000FC0) >> 6;
```

Residency Manager

- TexID:1, mip:0, Tile X:2, Tile Y:1
- TexID:1, mip:0, Tile X:2, Tile Y:2
- TexID:1, mip:1, Tile X:1, Tile Y:0
- TexID:1, mip:1, Tile X:1, Tile Y:1
- TexID:1, mip:2, Tile X:0, Tile Y:0
- TexID:1, mip:3, Tile X:0, Tile Y:0

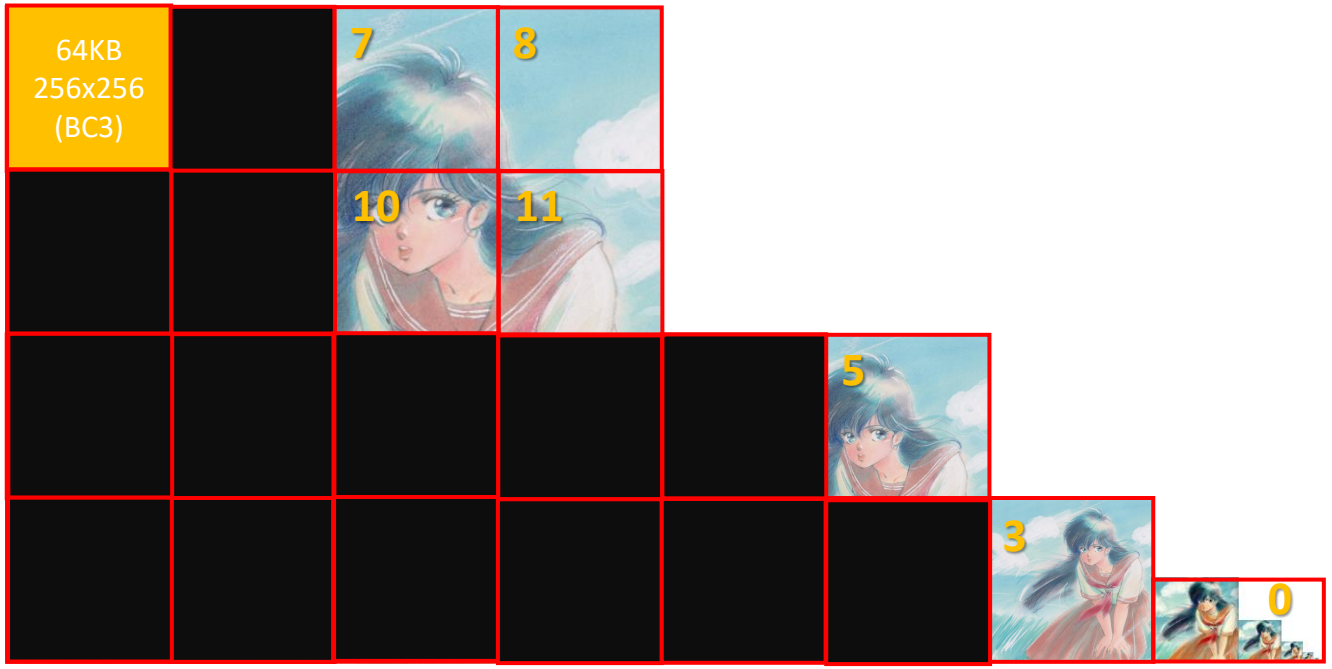
Tiled Resources관리

Tile Pool 관리

- Tile Pool의 각 Tile영역은 start offset과 개수로 map/unmap(commit/decommit)가능.
- 임의의 Index발급기를 만들어서 Tile pool의 start offset으로 사용.
- 타일의 좌표에 발급받은 Index(Tile Pool에서 start offset) 1:1 대응해서 저장.
- 나중에 decommit하기 위해서 Index가 필요함.
- Tiled Resources Status Buffer를 분석한 결과, 보인다고 판정된 Tile에 대해서 최종 억세스 시간을 기록.
- 최종 억세스 시간이 오래된 Tile들에 대해서는 decommit 처리.

Tiled Pool 관리

D3D API아님.
프로그래머가 만든 별도
자료구조



1024x1024
4x4(tile)

512x512
2x2(tile)

256x256
1x1(tile)

128x128
64x64
32x32
16x16
8x8
4x4
2x2
1x1
1x1(tile)

Index Pool	D3D Tile Pool
0	Offset 0, 64KB
1	Offset 1, 64KB
2	Offset 2, 64KB
3	Offset 3, 64KB
4	Offset 4, 64KB
5	Offset 5, 64KB
6	Offset 6, 64KB
7	Offset 7, 64KB
8	Offset 8, 64KB
9	Offset 9, 64KB
10	Offset 10, 64KB
11	Offset 11, 64KB
12	Offset 12, 64KB

Tiled Pool 관리 – 보이지 않는 Tile을 decommit

매 프레임마다 현재 tick을 측정. 현재 tick 카운터가 3000 tick일때

1024x1024
4x4(tile)

512x512
2x2(tile)

256x256
1x1(tile)

128x128
64x64
32x32
16x16
8x8
4x4
2x2
1x1
1x1(tile)

Index Pool	D3D Tile Pool
0	Offset 0, 64KB
1	Offset 1, 64KB
2	Offset 2, 64KB
3	Offset 3, 64KB
4	Offset 4, 64KB
5	Offset 5, 64KB
6	Offset 6, 64KB
7	Offset 7, 64KB
8	Offset 8, 64KB
9	Offset 9, 64KB
10	Offset 10, 64KB
11	Offset 11, 64KB
12	Offset 12, 64KB

Tiled Pool 관리 – 보이지 않는 Tile을 decommit

매 프레임마다 현재 tick을 측정. 현재 tick 카운터가 3000 tick일때

1024x1024
4x4(tile)

512x512
2x2(tile)

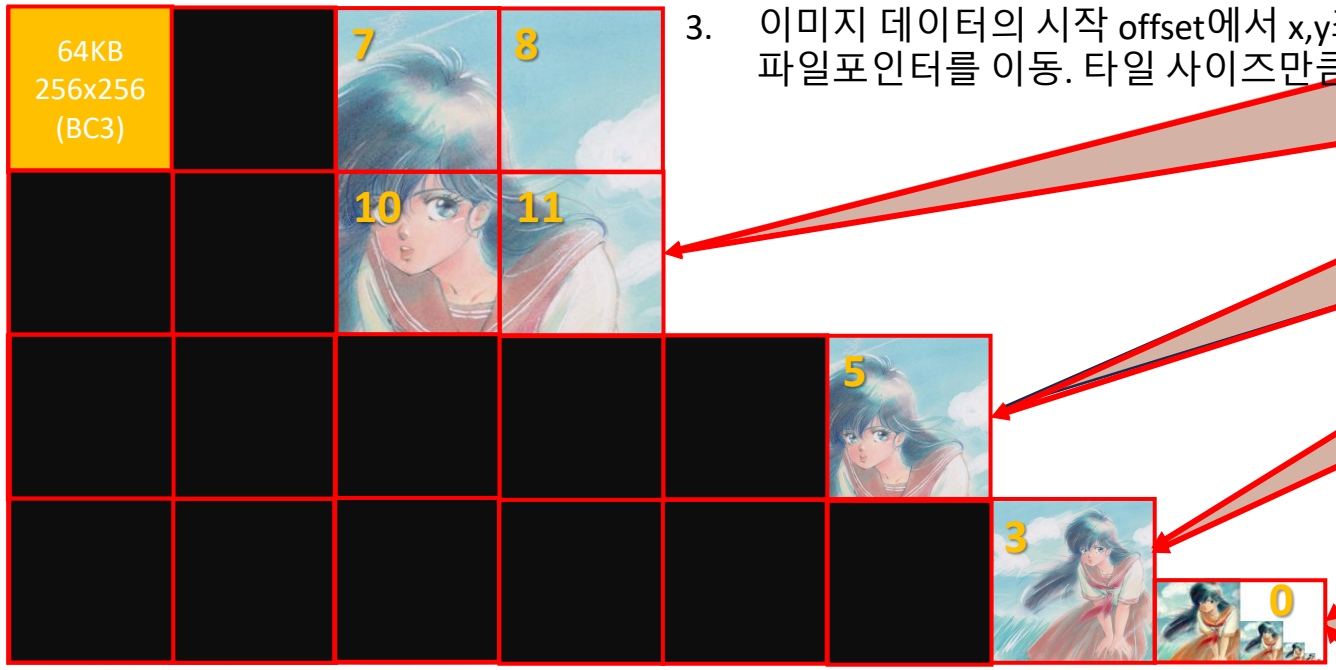
256x256
1x1(tile)

128x128
64x64
32x32
16x16
8x8
4x4
2x2
1x1
1x1(tile)

Index Pool	D3D Tile Pool
0	Offset 0, 64KB
1	Offset 1, 64KB
2	Offset 2, 64KB
3	Offset 3, 64KB
4	Offset 4, 64KB
5	Offset 5, 64KB
6	Offset 6, 64KB
7	Offset 7, 64KB
8	Offset 8, 64KB
9	Offset 9, 64KB
10	Offset 10, 64KB
11	Offset 11, 64KB
12	Offset 12, 64KB

File로부터 이미지 데이터 로드

1. mip 번호에 따라 이미지 데이터의 파일 위치를 한번에 찾을 수 있도록 offset을 파일 헤더에 기록.
2. Packed mip들을 한번에 읽을 수 있도록 이미지 데이터를 연속해서 저장함.
3. 이미지 데이터의 시작 offset에서 x,y좌표로 파일포인터를 이동. 타일 사이즈만큼만 로드.

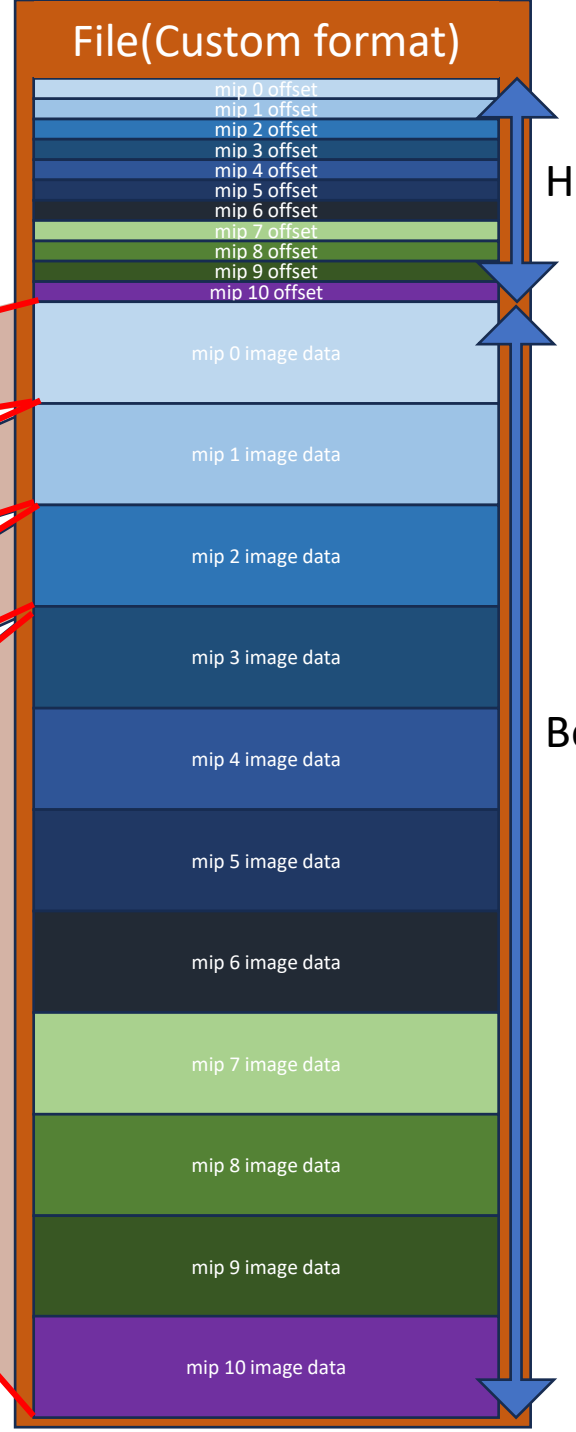


mip 0
1024x1024
4x4(tile)

mip 1
512x512
2x2(tile)

mip 2
256x256
1x1(tile)

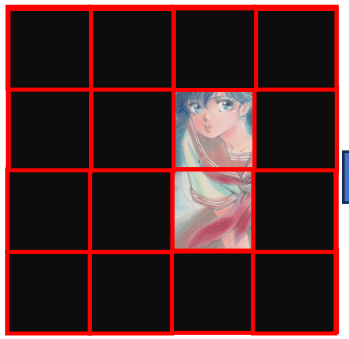
mip 3 – mip 10
128x128
64x64
32x32
16x16
8x8
4x4
2x2
1x1
1x1(tile)



GPU를 이용한 처리

Tiled Resources Status Buffer를 탐색하는 코드를 CPU기반 -> GPU기반으로 변경.

GPU를 이용한 분석

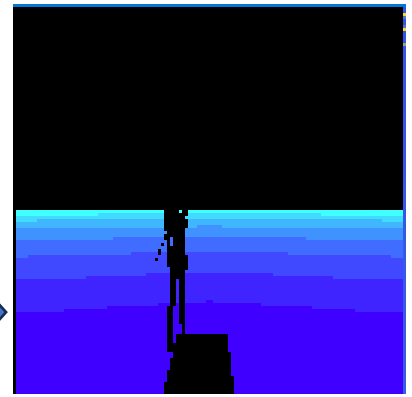


mip 0 (1024x1024)

mip 1 (512x512)

mip 2 (256x256)

mip 3-10 (128x128 – 1x1)



Compute Shader

```
[numthreads(THREAD_WIDTH_PER_GROUP, THREAD_HEIGHT_PER_GROUP, 1)]
void csInspectTiledResourceStatus(uint3 GroupId : SV_GroupID,
    uint3 GroupThreadID : SV_GroupThreadID,
    uint3 DispatchThreadId : SV_DispatchThreadID,
    uint GroupIndex : SV_GroupIndex)
{
    uint sx = DispatchThreadId.x;
    uint sy = DispatchThreadId.y;

#ifdef USE_SHARED_MEMORY_CACHE
    // Clear Shared Memory
    uint IndexInGroup = GroupThreadID.x + GroupThreadID.y * (THREAD_WIDTH_PER_GROUP);
    while (IndexInGroup < GROUP_CACHE_UINT_COUNT)
    {
        groupMemory[IndexInGroup] = 0;
        IndexInGroup += (THREAD_WIDTH_PER_GROUP * THREAD_HEIGHT_PER_GROUP);
    }
    GroupMemoryBarrierWithGroupSync();
#endif

    if (sx < g_SrcTexSize.x && sy < g_SrcTexSize.y)
    {
        // get integer pixel coordinates
        // u, v, array index, mip-level
        uint3 nCoords = uint3(sx, sy, 0);
        uint Prop = TiledResourceBuffer.Load(nCoords);
        // PageFault(1) | Mip Level(3) | Resered(1) | Layout Type(3) |  TexID(12) | Ti
        //      0/1     |      0-7     |     0/1    |      0-7       |   1-4095   |

        uint TexID = (Prop & 0x00FFF000) >> 12; // mask = 0b11111111111111000000000000
        if (TexID)
        {
            uint PageFault = (Prop & 0x80000000) >> 31;

            uint MipLevel = (Prop & 0x70000000) >> 28;  // 0 - 7
            uint LayoutType = (Prop & 0x07000000) >> 24; // 0 - 7 , layout 유형. textur
```

Residency Manager

- TexID:1, mip:0, Tile X:2, Tile Y:1
- TexID:1, mip:0, Tile X:2, Tile Y:2
- TexID:1, mip:1, Tile X:1, Tile Y:0
- TexID:1, mip:1, Tile X:1, Tile Y:1
- TexID:1, mip:2, Tile X:0, Tile Y:0
- TexID:1, mip:3, Tile X:0, Tile Y:0

Bit table

0	0	0	0
0	0	1	0
0	0	1	0
0	0	0	0
0	1		
0	1		
1			
1			

mip 0 (1024x1024) Tile 4x4

mip 1 (512x512) Tile 2x2

mip 2 (256x256) Tile 1x1

mip 3-10 (128x128 – 1x1) Tile 1x1

텍스처 해상도 별 layout정리

- Tiled Resources Status Buffer를 CPU코드만으로 분석할 때는 유형 정리가 필요 없다.
- GPU(Compute Shader)에서 처리할 때 4 bytes 한 단위 안에 많은 정보가 들어가 있어야 한다.
  - Texture size, Tile size, Mip 개수, packing 여부 등…
- 필요한 정보를 4 bytes 한 단위에 다 넣을 수 없으므로 유형을 정리해서 유형에 대한 Index(간접참조)로 표현한다.
- 유형은 8가지이므로 3 bits로 표현할 수 있다.

텍스처 해상도 별 layout 정리

[0] 128x128(비압축 RGBA일 때 Tile 1개 사이즈)
[1] 256x256(BC3압축일 때 Tile 1개 사이즈)
[2] 512x512
[3] 1024x1024
[4] 2048x2048
[5] 4096x4096
[6] 8192x8192
[7] 16384x16384(D3D Tiled Resources API에서 최대크기)

- 총 8가지 – 3 Bits로 표현 가능

Layout 0 128x128

mip	Texel WxH	Tile WxH	Tile Count	Bit Table Size	Bit Table Offset(uint)	Tile Offset	Packed
0	128x128	1x1	1	4	0	0	O
1	64x64	1x1	1	4	0	0	O
2	32x32	1x1	1	4	0	0	O
3	16x16	1x1	1	4	0	0	O
4	8x8	1x1	1	4	0	0	O
5	4x4	1x1	1	4	0	0	O
6	2x2	1x1	1	4	0	0	O
7	1x1	1x1	1	4	0	0	O

4

Layout 1 256x256

mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed
0	256x256	1x1	1	4	0	0	X
1	128x128	1x1	1	4	1	1	O
2	64x64	1x1	1	4	1	1	O
3	32x32	1x1	1	4	1	1	O
4	16x16	1x1	1	4	1	1	O
5	8x8	1x1	1	4	1	1	O
6	4x4	1x1	1	4	1	1	O
7	2x2	1x1	1	4	1	1	O
8	1x1	1x1	1	4	1	1	O

8

Layout 2 512x512

mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed
0	512x512	2x2	4	4	0	0	X
1	256x256	1x1	1	4	1	4	X
2	128x128	1x1	1	4	2	5	O
3	64x64	1x1	1	4	2	5	O
4	32x32	1x1	1	4	2	5	O
5	16x16	1x1	1	4	2	5	O
6	8x8	1x1	1	4	2	5	O
7	4x4	1x1	1	4	2	5	O
8	2x2	1x1	1	4	2	5	O
9	1x1	1x1	1	4	2	5	O

12

Layout 3 1024x1024

mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed
0	1024x1024	4x4	16	4	0	0	X
1	512x512	2x2	4	4	0	16	X
2	256x256	1x1	1	4	1	20	X
3	128x128	1x1	1	4	2	21	O
4	64x64	1x1	1	4	3	21	O
5	32x32	1x1	1	4	3	21	O
6	16x16	1x1	1	4	3	21	O
7	8x8	1x1	1	4	3	21	O
8	4x4	1x1	1	4	3	21	O
9	2x2	1x1	1	4	3	21	O
10	1x1	1x1	1	4	3	21	O

16

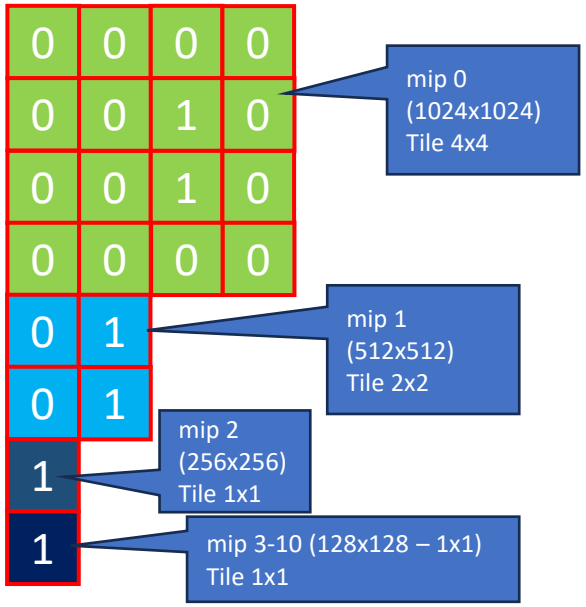
Layout 4	2048x2048							
mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed	
0	2048x2048	8x8	64	8	0	0	X	
1	1024x1024	4x4	16	4	2	64	X	
2	512x512	2x2	4	4	3	80	X	
3	256x256	1x1	1	4	4	84	X	
4	128x128	1x1	1	4	5	85	O	
5	64x64	1x1	1	4	5	85	O	
6	32x32	1x1	1	4	5	85	O	
7	16x16	1x1	1	4	5	85	O	
8	8x8	1x1	1	4	5	85	O	
9	4x4	1x1	1	4	5	85	O	
10	2x2	1x1	1	4	5	85	O	
11	1x1	1x1	1	4	5	85	O	
								24
Layout 5	4096x4096							
mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed	
0	4096x4096	16x16	256	32	0	0	X	
1	2048x2048	8x8	64	8	8	256	X	
2	1024x1024	4x4	16	4	10	320	X	
3	512x512	2x2	4	4	11	336	X	
4	256x256	1x1	1	4	12	340	X	
5	128x128	1x1	1	4	13	341	O	
6	64x64	1x1	1	4	13	341	O	
7	32x32	1x1	1	4	13	341	O	
8	16x16	1x1	1	4	13	341	O	
9	8x8	1x1	1	4	13	341	O	
10	4x4	1x1	1	4	13	341	O	
11	2x2	1x1	1	4	13	341	O	
12	1x1	1x1	1	4	13	341	O	
								56
Layout 6	8192x8192							
mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed	
0	8192x8192	32x32	1024	128	0	0	X	
1	4096x4096	16x16	256	32	32	1024	X	
2	2048x2048	8x8	64	8	40	1280	X	
3	1024x1024	4x4	16	4	42	1344	X	
4	512x512	2x2	4	4	43	1360	X	
5	256x256	1x1	1	4	44	1364	X	
6	128x128	1x1	1	4	45	1365	O	
7	64x64	1x1	1	4	45	1365	O	
8	32x32	1x1	1	4	45	1365	O	
9	16x16	1x1	1	4	45	1365	O	
10	8x8	1x1	1	4	45	1365	O	
11	4x4	1x1	1	4	45	1365	O	
12	2x2	1x1	1	4	45	1365	O	
13	1x1	1x1	1	4	45	1365	O	
								184

Layout 7	16384x16384							
mip	Texel WxH	Tile WxH	Tile Count	BitTable Size	Bit Table Offset(uint)	Tile Offset	Packed	
0	16384x16384	64x64	4096	512	0	0	X	
1	8192x8192	32x32	1024	128	128	4096	X	
2	4096x4096	16x16	256	32	160	5120	X	
3	2048x2048	8x8	64	8	168	5376	X	
4	1024x1024	4x4	16	4	170	5440	X	
5	512x512	2x2	4	4	171	5456	X	
6	256x256	1x1	1	4	172	5460	X	
7	128x128	1x1	1	4	173	5461	O	
8	64x64	1x1	1	4	173	5461	O	
9	32x32	1x1	1	4	173	5461	O	
10	16x16	1x1	1	4	173	5461	O	
11	8x8	1x1	1	4	173	5461	O	
12	4x4	1x1	1	4	173	5461	O	
13	2x2	1x1	1	4	173	5461	O	
14	1x1	1x1	1	4	173	5461	O	
								696

보여지는 Tile결과 버퍼 - Bit단위로 저장

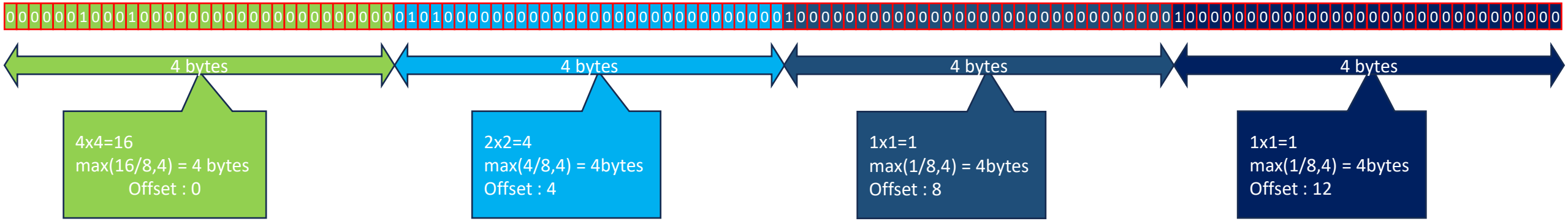
1. Tiled texture가 최대 4096개일 때, 결과 버퍼 크기
   - 타일 한 칸 4 bytes -> ((4096 + 1024 + 256 + 64 + 16 + 4 +1 + 1) x 4) x 4096 = 87.392MB
   - 타일 한 칸 1 bits -> (512 + 128 + 32 + 8 + 4 + 4 + 4 + 4) x 4096 = 2.784MB
- GPU메모리 -> System 메모리로 카피해야 하므로 작은 사이즈 의 메모리를 사용하는 편이 좋다.
- GPU에서 Shared memory 사용을 고려하면 되도록 작은 사이즈의 메모리를 사용하는 편이 좋다.
- 어차피 GPU에서 돌려야 하므로 CPU코드를 GPU에 맞춘다.
- 다만 Bit table대신 4 bytes 변수를 한 단위로 사용할 경우 atomic + 논리 연산에 의한 성능 저하는 오히려 적을 수도 있음(테스트 필요).

Bit Table (Memory map)

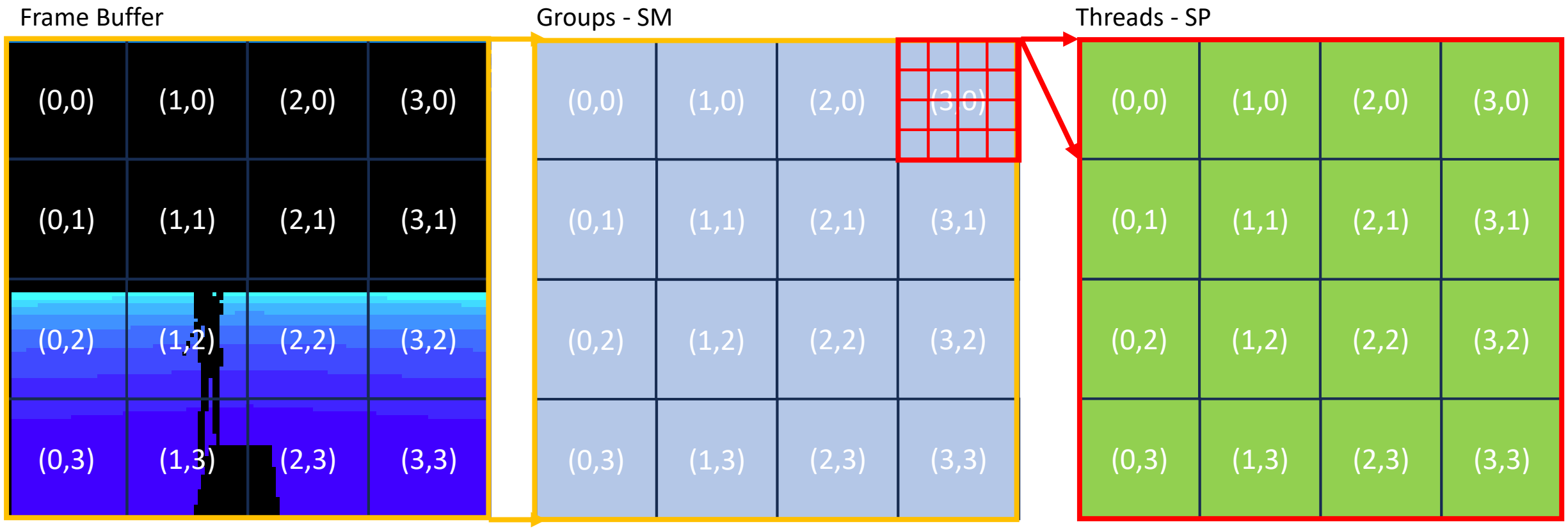


1. Tile이 중복해서 commit되는 상황 방지
2. 적은 메모리만 소비하기 위한 bit단위 억세스
3. GPU에서의 분석 및 결과저장을 위한 자료구조
4. GPU에 맞춰서 CPU에서도 동일 구조를 사용.
5. 16384x16484기준으로 Tiled Texture 한 장당 696 bytes 필요.
6. CPU/GPU측 양쪽 모두 최대 텍스처 개수x 696 Bytes 필요.
7. 타일 한 칸 4 bytes -> ((4096 + 1024 + 256 + 64 + 16 + 4 +1 + 1) x 4) x 4096 = 87.392MB
8. 타일 한 칸 1 bits -> (512 + 128 + 32 + 8 + 4 + 4 + 4 + 4) x 4096 = 2.784MB

Ex) layout 3, 1024 x 1024 , bit table size = 16 bytes



Compute Shader를 이용한 분석

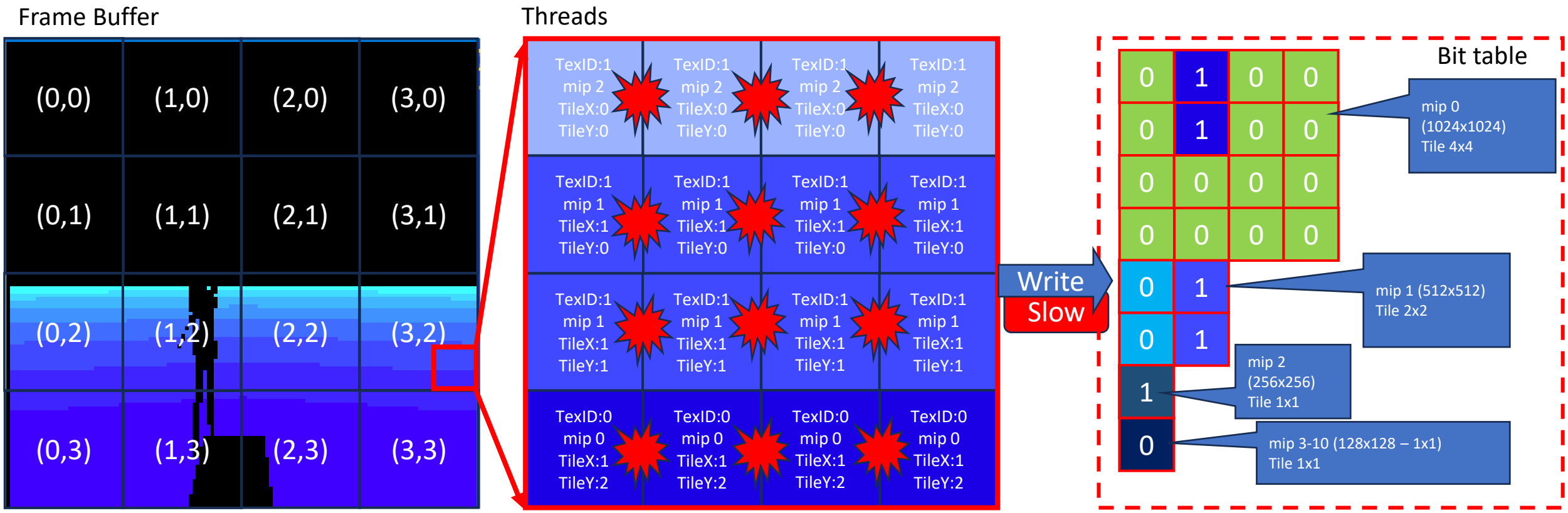


1. Total threads = Threads per group x groups
2. (X ,Y in frame buffer) = (X,Y in Group) x Threads in Group
3. 1개의 Group만으로도 처리는 가능.
4. 가능하다면 Group 내의 여러 스레드가 협력해서 처리하는 방법을 권장함.
5. 일반적으로 Group당 32x32=1024 스레드 사용

Compute Shader를 이용한 분석

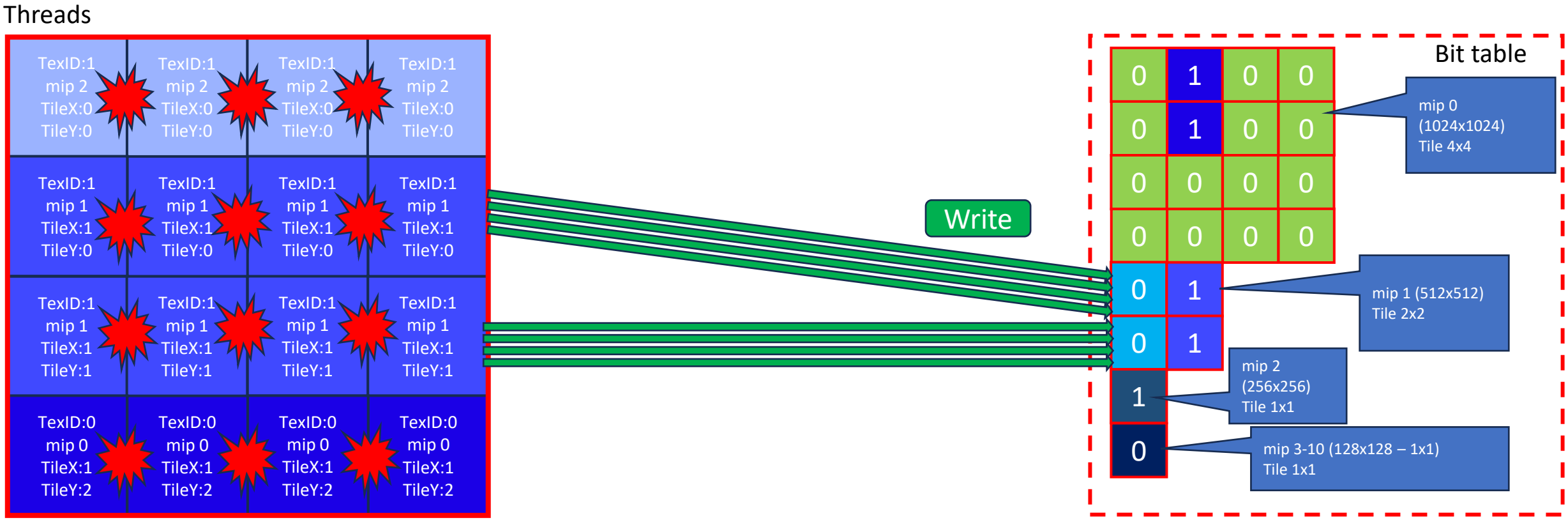
1. 각 스레드는 자신에게 맵핑된 픽셀을 디코딩해서 Tex ID, Tile 좌표, mip 번호, 텍스처 해상도, 타일 해상도, bit table offset을 얻는다.
2. 디코딩한 정보로 UAV로 전달된 버퍼(bit table)에 화면에 보여지는 tile을 기록한다.
3. 보여지는 타일의 위치가 여러 스레드 사이에서 중복될 수 있으므로 UAV에 써 넣을 때 atomic 연산이 필요.

Compute Shader를 이용한 분석—UAV 중복억세스에 따른 성능저하



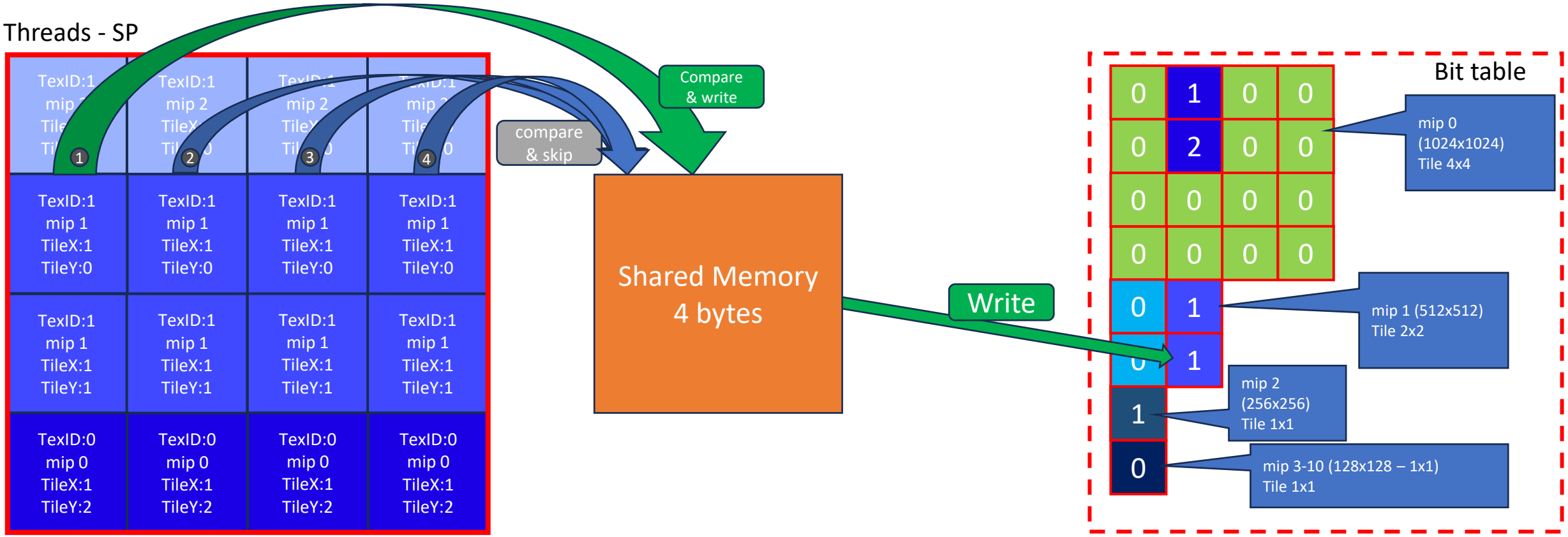
1. 각 스레드는 자신에게 맵핑된 픽셀을 디코딩해서 Tex ID, Tile 좌표, mip 번호, 텍스처 해상도, 타일 해상도, bit table offset을 얻는다.
2. 디코딩한 정보로 UAV로 전달된 버퍼(bit table)에 화면에 보여지는 tile을 기록한다.
3. 보여지는 타일의 위치가 여러 스레드 사이에서 중복될 수 있으므로 UAV에 써 넣을때 atomic 연산이 필요.
4. UAV(GPU의 Global Memory)는 억세스가 느리다.
5. atomic연산으로 더더욱 느려진다.

Compute Shader를 이용한 분석—UAV 중복억세스에 따른 성능저하



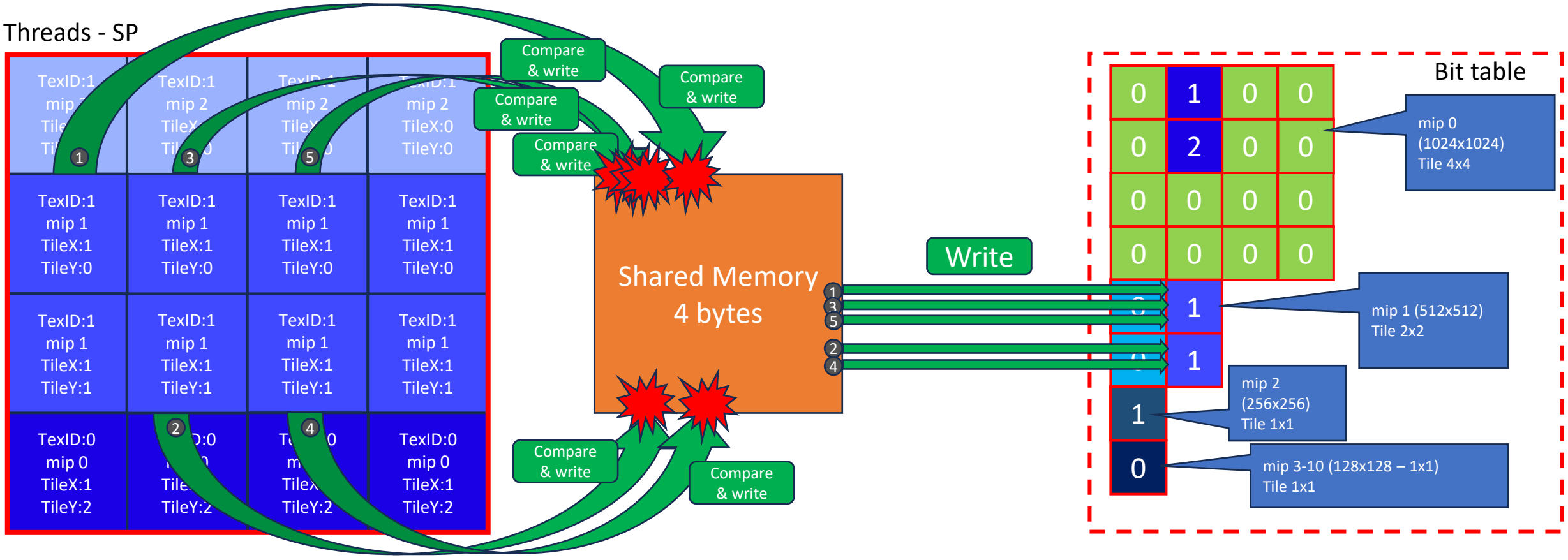
1. 각 스레드는 자신에게 맵핑된 픽셀을 디코딩해서 Tex ID, Tile 좌표, mip 번호, 텍스처 해상도, 타일 해상도, bit table offset을 얻는다.
2. 디코딩한 정보로 UAV로 전달된 버퍼(bit table)에 화면에 보여지는 tile을 기록한다.
3. 보여지는 타일의 위치가 여러 스레드 사이에서 중복될 수 있으므로 UAV에 쓸때 atomic 연산이 필요.
4. UAV(GPU의 Global Memory)는 억세스가 느리다.
5. atomic연산으로 더더욱 느려진다.

Compute Shader를 이용한 분석—cache를 이용한 최적화



1. 현재 스레드에서 다른 스레드에서 디코딩 결과 동일한 값을 얻었고 이미 UAV에 기록했다는 사실을 알 수 있다면 중복 억세스를 피할 수 있다.
2. 프레임 버퍼에서 읽은 값은 4 bytes uint값-> atomic 연산이 가능한 uint 한 단위->디코딩 결과를 대표할 수 있는 key값으로 사용 가능.
3. UAV에 써넣을 때 마다 스레드간에 공유할 수 있는 임의의 메모리에 key값을 써 놓으면 중복 방지 가능.
4. Shared memory는 Global Memory보다 압도적으로 빠르다.

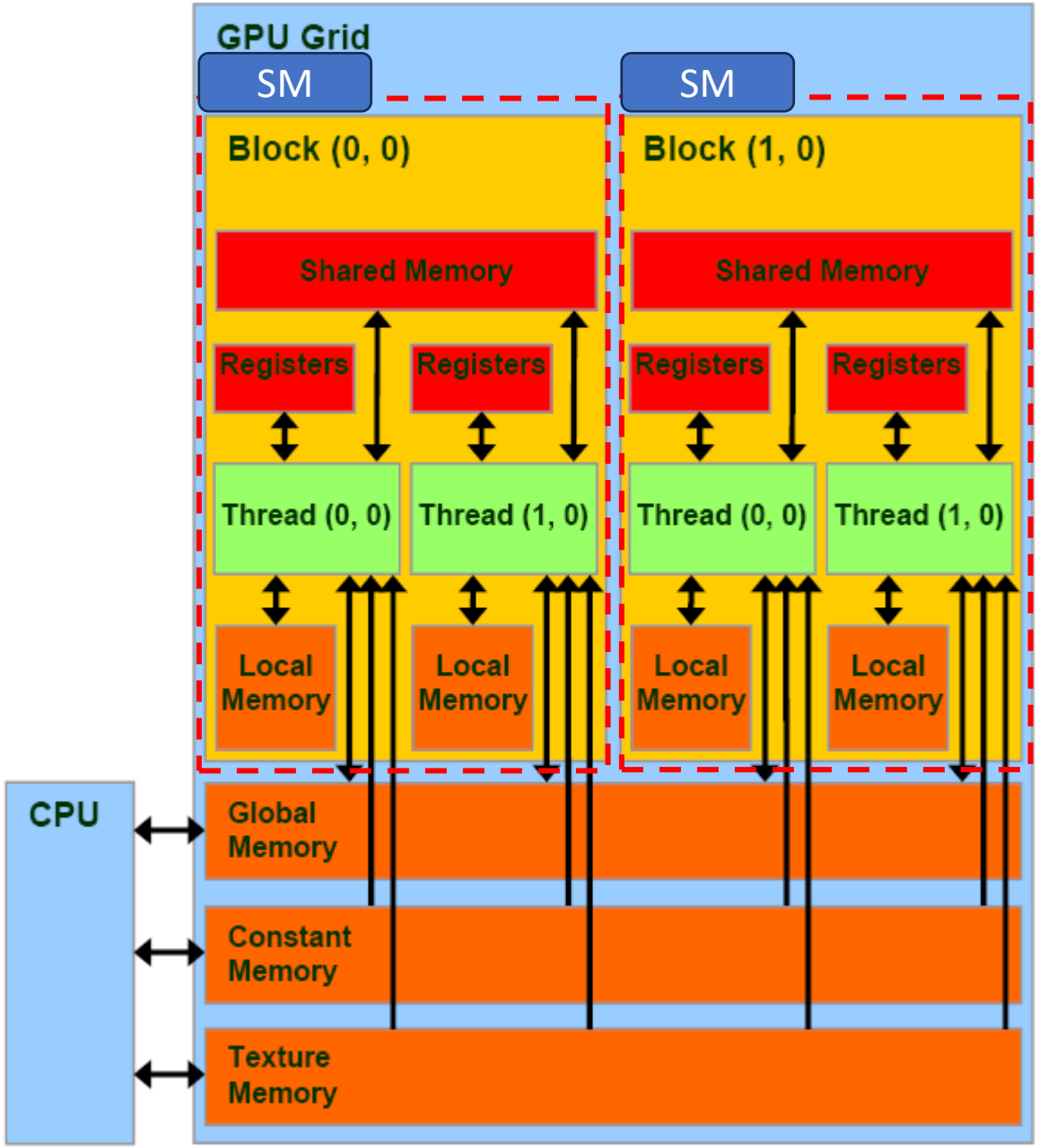
cache를 이용한 최적화- cache 충돌



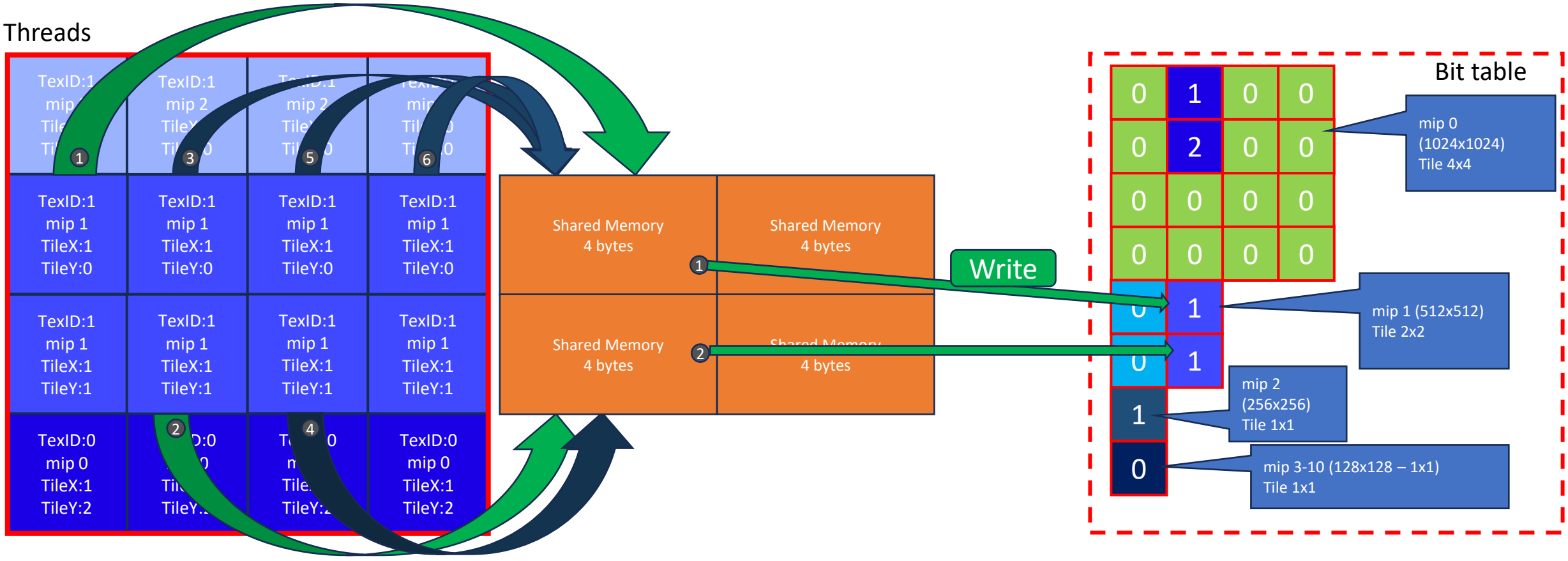
1. 단일 4 bytes 변수 하나만으로는 스레드 진입 순서에 따라서 아주 빈번하게 충돌이 발생할 수 있다.
2. Tex ID/mip번호/ 타일 위치의 모든 경우의 수에 대응하는 bit table을 만들면 충돌을 0%로 낮출 수 있는데?
3. 그것이 불가능하다…

Shared memory 제한

1. CUDA에서의 Block == Compute Shader에서 Group.
2. Block/Group은 SM에 맵핑됨.
3. Register 개수/Shared Memory의 총량이 허용하는 한도 내에서 여러 개의 group/block(내부적으로 warp)를 시분할로 처리.
4. Shared Memory는 물리적으로 L1 cache memory.
5. CUDA에서는 16KB L1 Cache + 48 KB Shared Memory로사용 가능.
6. Compute Shader에서는 32 KB Shared Memory만 사용 가능.
7. Bit table 최대 사이즈(696 bytes) x 텍스처 최대 개수(4096) = 2.784MB. Shared Memory에 할당 불가능.
8. SM당 다수의 Group을 동시 스케쥴링 가능하지만 Group당 사용하는 shared memory 용량의 총합이 32KB이내여야 함. 1개의 Group이 32KB만 사용해도 SM 1개당 1개의 Group만 스케쥴링 됨.



cache를 이용한 최적화- Group당 Shared Memory 1KB 할당.



1. 어느 정도의 cache 충돌은 감수하되 충돌을 줄일 수 있도록 cache memory사이즈를 늘린다.
2. Tile좌표, mip번호, Tex ID를 원래의 경우의 수보다 작은 범위로 맵핑

```
uint cache_tex_id = TexID % CACHE_SLOT_COUNT_TEX_ID;            // TexID들 0-3 사이로 맵핑
uint cache_mip_level = MipLevel % CACHE_SLOT_COUNT_MIP_LEVEL;   // MipLevel을 0-3 사이로 맵핑
uint2 cache_tile_pos = TilePos.xy % CACHE_SLOT_COUNT_TILE_POS; // 타일 좌표x,y를 각각 0-3 사이로 맵핑

uint cache_index = cache_tile_pos.x + (cache_tile_pos.y * CACHE_SLOT_COUNT_TILE_POS) +
                   (cache_mip_level * CACHE_SLOT_COUNT_TILE_POS * CACHE_SLOT_COUNT_TILE_POS) +
                   (cache_tex_id * CACHE_SLOT_COUNT_MIP_LEVEL * CACHE_SLOT_COUNT_TILE_POS * CACHE_SLOT_COUNT_TILE_POS);

uint CacheValue = Prop | 0x80000000; // cache사용할때 uint 4bytes항목에서 Page Fault는 무조건 0이나 1로 설정할것. cache초기값이 0이므로 0을 대
uint PrvCacheValue = 0;

InterlockedExchange(groupMemory[cache_index], CacheValue, PrvCacheValue);
```

데모

이후 작업

- Tiled Resources 기능을 D3D12 렌더러에 포팅.
- D3D12에서의 Tiled Resources는 Direct Storage와 접목해서 사용 가능.

참고자료 – D3D Tiled Resources

- https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm?fbclid=IwAR3btDYV6-qm-_Q0YWqwz1lakeRVZCZEH3_B7vFmk5uQvpmFzu2uDGIbecU#5.9.1%20Overview
- https://youtu.be/OPfr2WkVmQA
- https://www.gdcvault.com/browse/gdc-14/play/1020621

샘플 코드 – D3D Tiled Resources

- https://github.com/microsoftarchive/msdn-code-gallery-microsoft
  - Msdn-code-gallery-Microsoft/Official Windows Platform Sample/Windows 8.1 Store app samples/[C++]-Windows 8.1 Store app samples/Direct3D tiled resources sample/**C++**/
  - Windows 8.1 Store App 환경/Direct3D 11.3
  - VS2015필요. Modern C++과 WinRT과 만나 최악의 가독성.
- https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12ReservedResources
  - Windows 10(win32 Desktop)/Direct3D 12
  - VS2019/VS2022
  - 코드는 비교적 알아보기 쉬움. Tiled Resources의 생성 및 업데이트, 특히 Packed Mip다루는 방법을 알 수 있음.

참고자료 – SW Virtual Texturing

- https://www.nvidia.com/content/GTC/posters/37__Hollemeersch_Accelerating_Virtual_Texturing_Using_CUDA.pdf
- https://silverspaceship.com/src/svt/
- GPU Pro(번역서)
  - 가상 텍스처 매핑 입문
  - CUDA를 이용한 가상 텍스처 가속