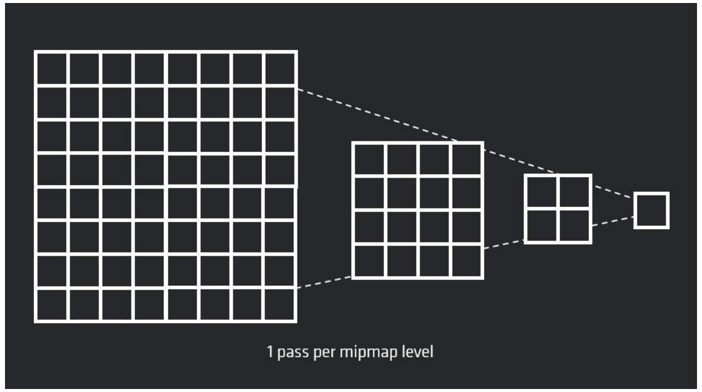
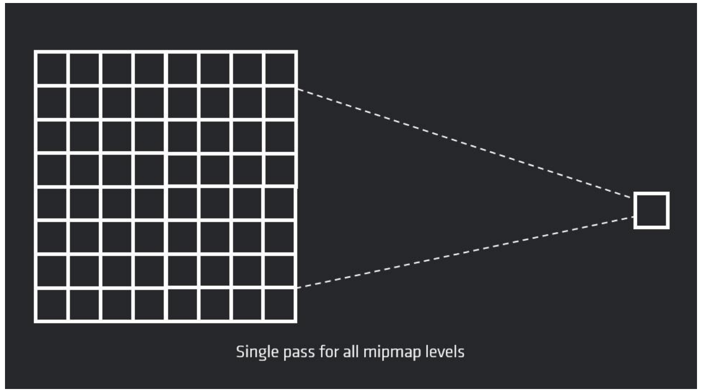
FIDELITY FX – SPD

LOU KRAMER, AMD

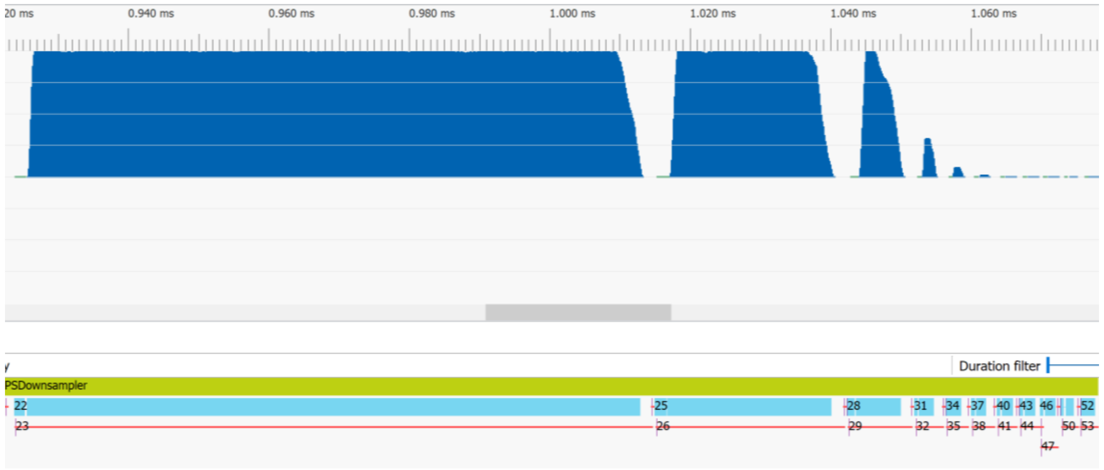
FIDELITY FX
SINGLE PASS DOWNSAMPLER (SPD)

GPUOpen's FidelityFX Single Pass Downsampler (SPD) library provides a single-pass compute shader RDNA-optimized solution to generate up to 12 mip levels of a given texture





MOTIVATION



A common approach to generate the mipmap levels is using a **pixel shader**, **one pass per mip**

Limitations and bottlenecks of a pixel shader approach:

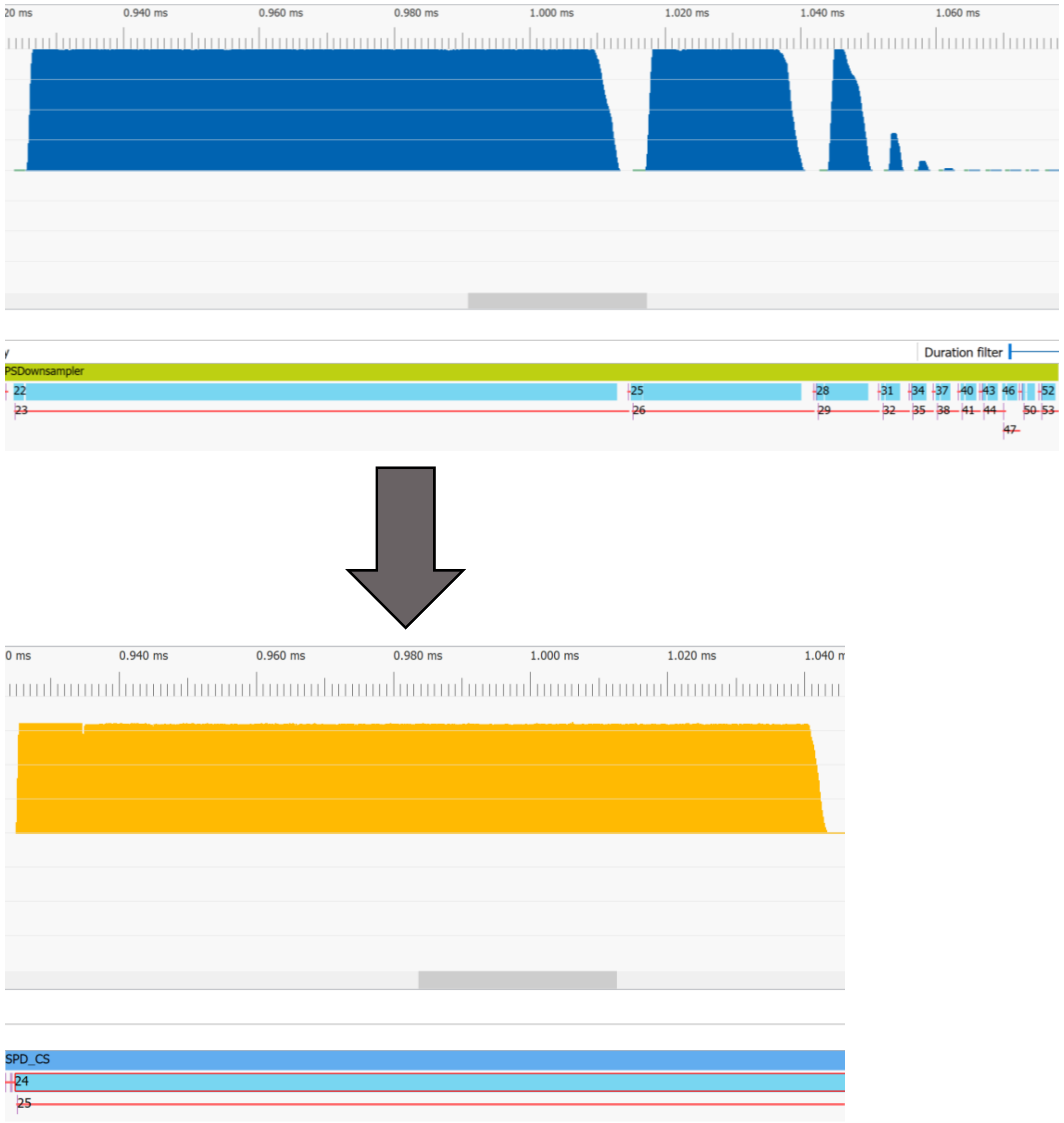
- **Barriers** between the mips
- Few working threads for about ~1/6th of the whole downsampling pass
- Data exchange between the mips via **global memory**

MOTIVATION

SPD uses only a single pass compute shader for all mips

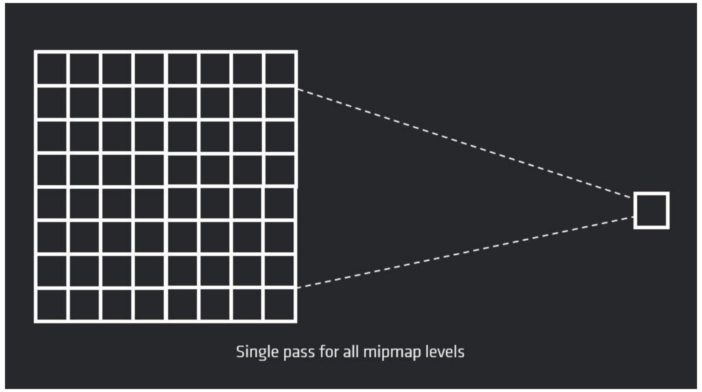
Advantages:

- **No intermediate barriers**
- Few working threads for only ~2% of the pass
- Data exchange between the mips via LDS or DPP except for mip 6
- Can **overlap work** with other dispatches/draw calls due to no barriers between the mip generation



CONCEPT OF SPD

CONCEPT



Basic concept of SPD:

- Threadgroup of 256 threads downsamples a tile of 64x64 down to 1x1

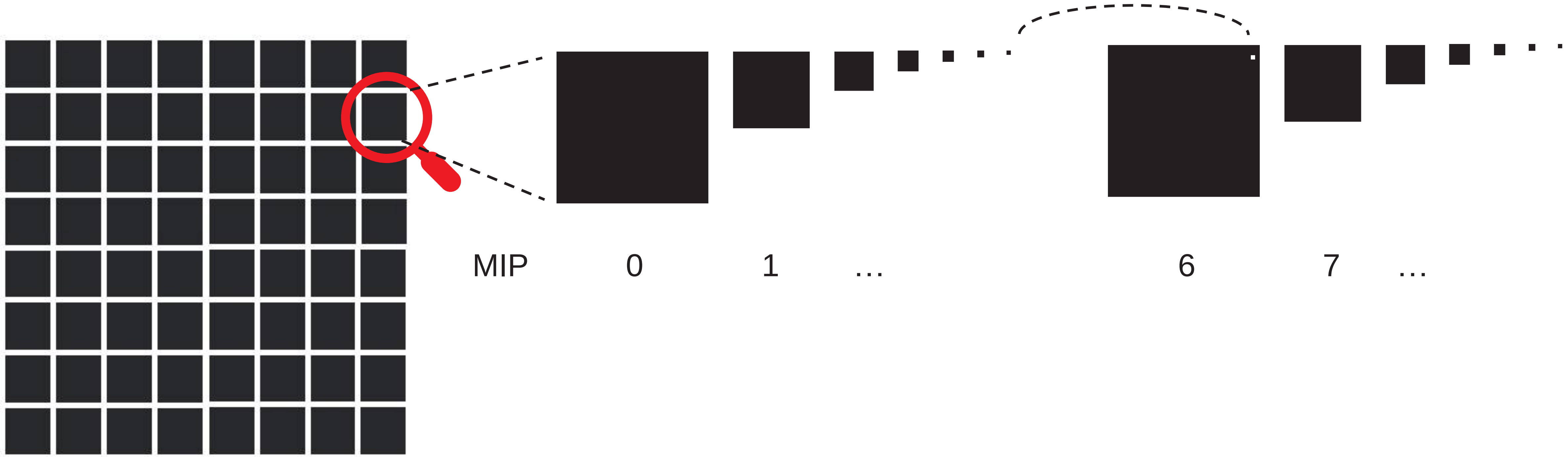
→ Each threadgroup works independently from the other thread groups

- Last active threadgroup computes the remaining mips

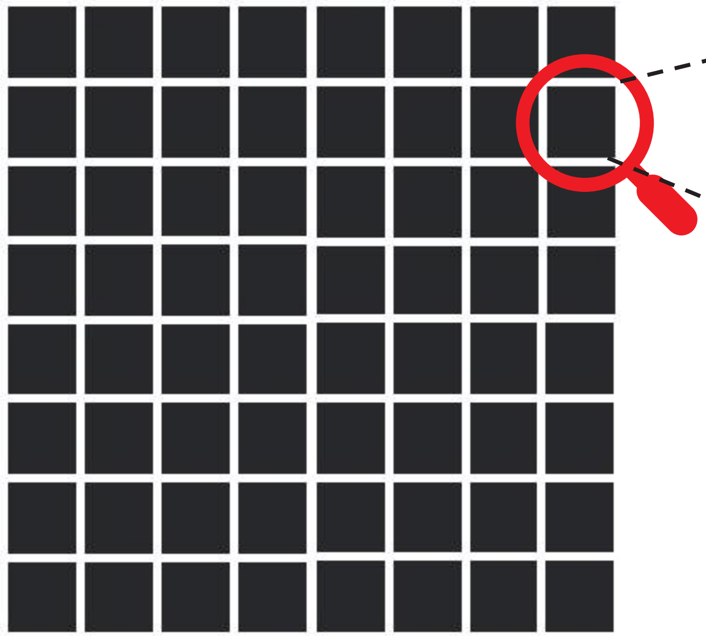
→ **One** synchronization point between all thread groups is required

→ Can downsample a texture of size 4096x4096 to 1x1 (12 MIPs)

CONCEPT



CONCEPT

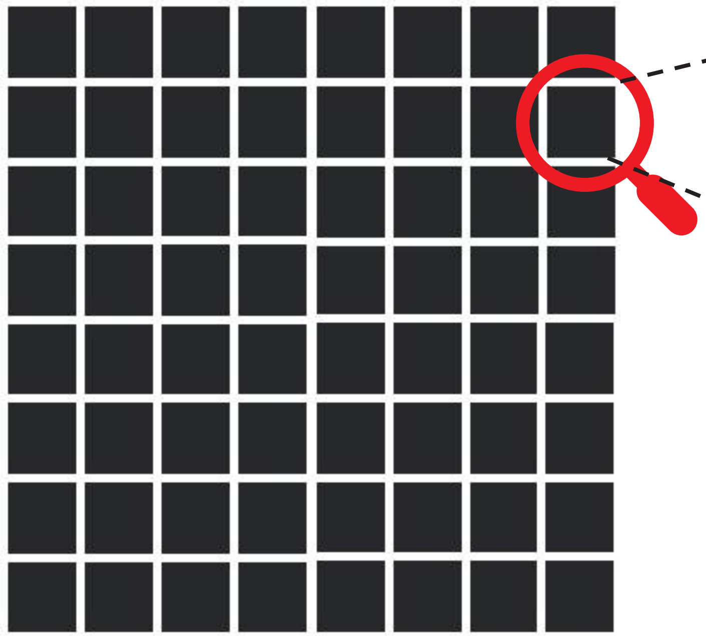


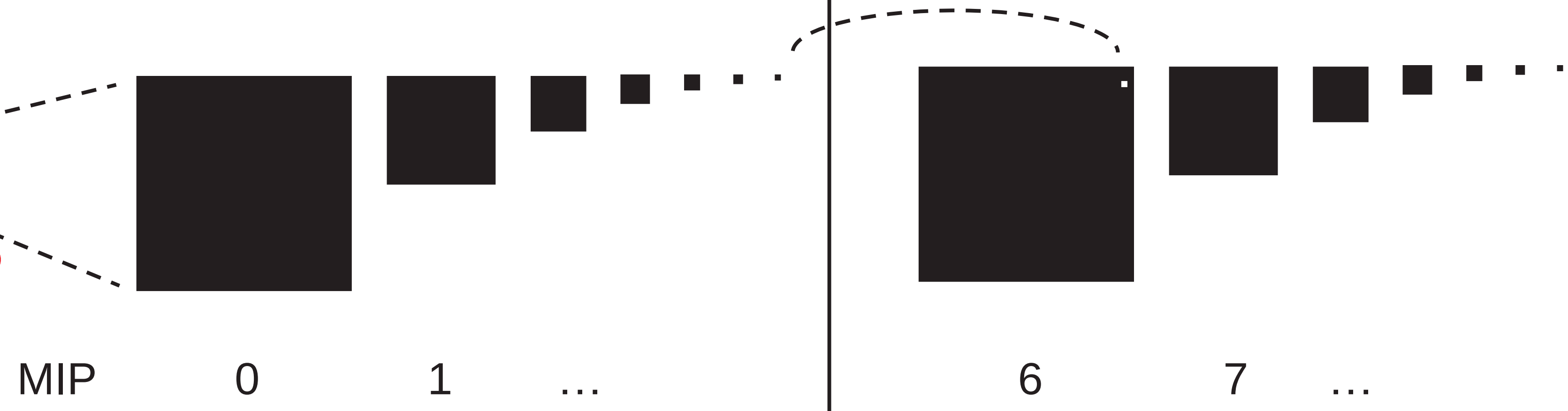
Each threadgroup works on a 64x64 tile

Only last active thread group computes MIP 7 to 12

CONCEPT

Global synchronization point across all thread groups



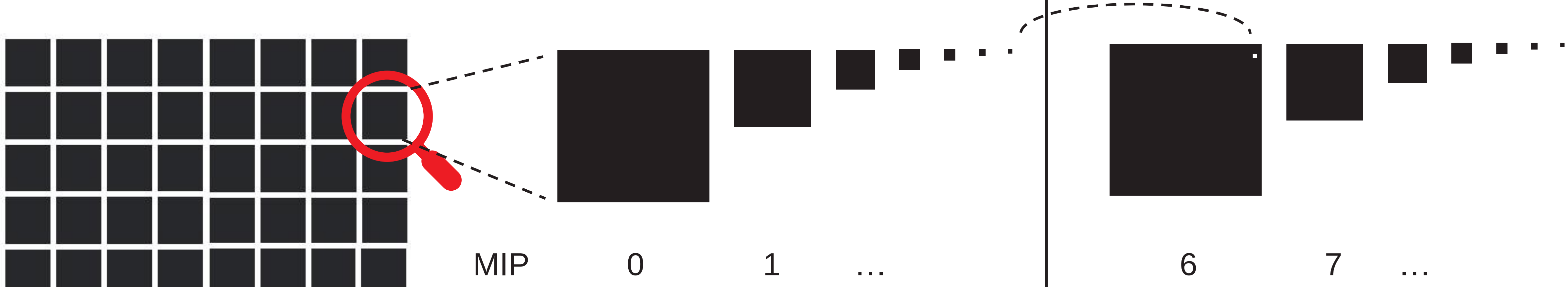


Each threadgroup works on a 64x64 tile

Only last active thread group computes MIP 7 to 12

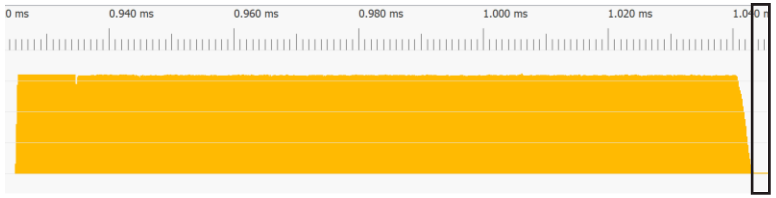
CONCEPT

Global synchronization point across all thread groups



Each threadgroup works on a 64x64 tile

Only last active thread group computes MIP 7 to 12



Takes only ~2% of whole mipmap computation

IMPLEMENTATION

IMPLEMENTATION

Conceptual implementation:

```
GenerateMIP1();

GenerateMIP2();

GenerateMIP3();

GenerateMIP4();

GenerateMIP5();

GenerateMIP6();

IncreaseAtomicCounter();

If (atomicCounter == numberOfThreadGroups)
{

   // Repeat above with offset 0

}
```

Offset for tile from MIP 0: dispatchID.xy * 64;

Offset for tile from MIP 1: dispatchID.xy * 32;

Offset for tile from MIP 2: dispatchID.xy * 16;

…

IMPLEMENTATION

Conceptual implementation:

```
GenerateMIP1();

GenerateMIP2();

GenerateMIP3();

GenerateMIP4();

GenerateMIP5();

GenerateMIP6();

IncreaseAtomicCounter();

If (atomicCounter == numberOfThreadGroups)
{

   // Repeat above with offset 0

}
```

Threadgroup size is <256,1,1>

MIP 0 to MIP 1:

- Each thread **loads 16** values from the source texture
- Each thread **computes 4** values for MIP 1

MIP 1 to MIP 2:

- Each thread **loads 4** values from LDS or uses quad swizzle
- Each thread **computes 1** value for MIP 2

MIP 2 to MIP 3:

- Only every **4th** thread is needed at this point

MIP 3 to MIP 4:

- Only every **16th** thread is needed at this point

…

IMPLEMENTATION

Conceptual implementation:

```
GenerateMIP1();

GenerateMIP2();

GenerateMIP3();

GenerateMIP4();

GenerateMIP5();

GenerateMIP6();

IncreaseAtomicCounter();

If (atomicCounter == numberOfThreadGroups)
{

   // Repeat above with offset 0

}
```

MIP 0 to MIP 1:

- Each thread **loads 16** values from the source texture

This is very time consuming, especially for high resolution source textures

Load pattern of source texture is critical

TEXTURE ACCESS

0	1	8	9	16	17	24	25	64	65	72	73	80	81	88	89
2	3	10	11	18	19	26	27	66	67	74	75	82	83	90	91

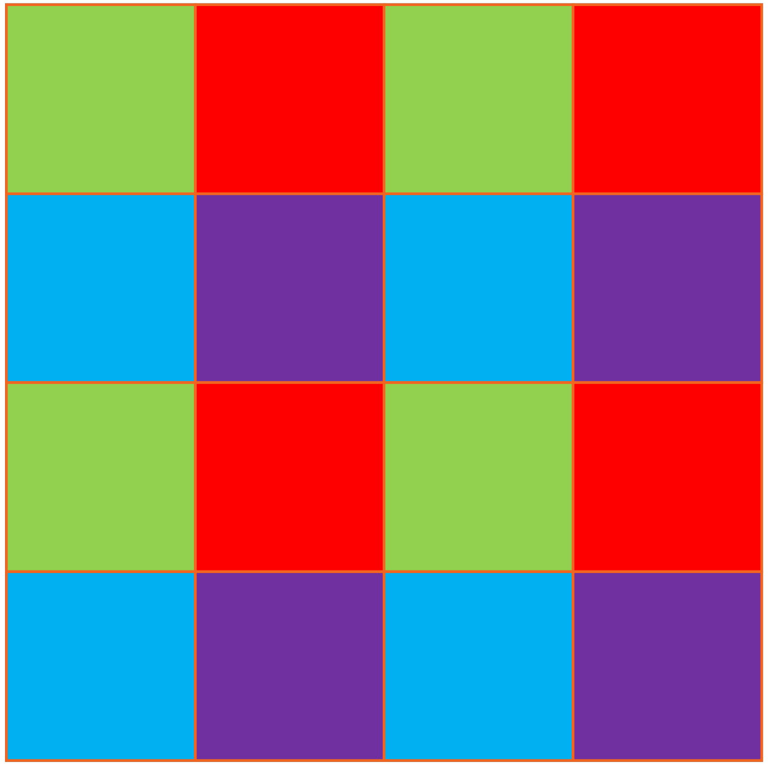
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1	12,1	13,1	14,1	15.1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2	12,2	13,2	14,2	15,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3	12,3	13,3	14,3	15,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4	11,4	12,4	13,4	14,4	15,4

Use morton ordering to rearrange the thread indices in a 2x2 swizzle

- Matches the standard texture layout
- Neighboring pixels are laid out in memory close by

```
x = (((index >> 2) & 0x0007) & 0xFFFE) | index & 0x0001
y = ((index >> 1) & 0x0003) | (((index >> 3) & 0x0007) & 0xFFFC
```

TEXTURE ACCESS



Texel index **(0,0), (1,0), (0,1), (1,1)** loaded by thread with index **0**
Texel index (2,0), (3,0), (2,1), (3,1) loaded by thread with index 1
…

0,0	1,0	2,0	3,0	4,0	5,0
0	0	1	1	8	8
0	0	1	1	8	8
2	2	3	3	10	10
2	2	3	3	10	10

…

32,0	33,0	34,0	35,0	36,0	37,0
0	0	1	1	8	8
0	0	1	1	8	8
2	2	3	3	10	10
2	2	3	3	10	10

IMPLEMENTATION

For exchanging the data between the MIPs LDS and optionally wave operations are used.

If wave operations are used

→ Reduced VGPRs

For bits per pixel (bpp) <= 16, we can also use FP16

→ Reduced VGPRs

→ Reduced LDS

Less number of VGPRs and LDS can be especially beneficially for overlapping FFX SPD with other work in parallel or when used on the async compute queue

→ Potentially less work in flight limited

INTEGRATION

INTEGRATION - CPU

Provide as constants:

- number of MIP levels to be computed (maximum is 12)
- number of total thread groups: ((widthInPixels+63)>>6) * ((heightInPixels+63)>>6)

Bind the resources ☺

→ source texture + optionally sampler

→ output MIPs (can be same resource as source texture or different resource)

Initialize your global atomic counter to 0

Dispatch the shader

```
vkCmdDispatch(cmdBuf,(widthInPixels+63)>>6,(heightInPixels+63)>>6,1);
```

INTEGRATION - GPU

Resources:

- Source image
- Destination images [# of output MIPs]
- Global atomic counter → a single unsigned integer, read & write access
- Constants
- [optional] Sampler

If the 2x2 -> 1 reduction function is computing the average

→ sample from the source image using a bilinear filter

INTEGRATION - GPU

Setup pre-portability-header defines (sets up GLSL/HLSL path, etc.)

```
#define A_GPU 1
#define A_HLSL 1 // or // # define A_GLSL 1
```

→ All following code samples use HLSL

for PACKED version

```
#define A_HALF
```

Include the portability header

```
#include "ffx_a.h"
```

INTEGRATION - GPU

Define LDS variables

`groupshared AU1 spd_counter;` → store current global atomic counter for all threads within the thread group

`groupshared AF4 spd_intermediate[16][16];` → intermediate data storage for inter-mip exchange

PACKED version

```
groupshared AH4 spd_intermediate[16][16];
```

Separating the channels is also possible – we recommend trying out both and measuring performance ☺ it can vary from format and number of channels

```
groupshared AF1 spd_intermediateR[16][16];
groupshared AF1 spd_intermediateG[16][16];
groupshared AF1 spd_intermediateB[16][16];
groupshared AF1 spd_intermediateA[16][16];
```

or for PACKED version:

```
groupshared AH2 spd_intermediateRG[16][16];
groupshared AH2 spd_intermediateBA[16][16];
```

INTEGRATION - GPU

Define SPD interface functions

```
AF4 SpdLoadSourceImage(ASU2 p){ return imgSrc[p]; }
AF4 SpdLoad(ASU2 p){ return imgDst[5][p]; } // load from output MIP 5
void SpdStore(ASU2 p, AF4 value, AU1 mip){ imgDst[mip][p] = value; }
```

If you use sRGB or UNORM, you need to transform your values to linear color space and back:

```
AF4 SpdLoadSourceImage(ASU2 p){ return imgSrc[p] * imgSrc[p]; }
AF4 SpdLoad(ASU2 p){ return imgDst[5][p] * imgDst[5][p]; }
void SpdStore(ASU2 p, AF4 value, AU1 mip){imgDst[mip][p] = sqrt(value);}
```

Add boundary checks if texture resolution is not a power of 2

LOAD FROM SOURCE IMAGE

Standard, default solution:

```
AF4 SpdLoadSourceImage(ASU2 p){return imgSrc[p];}
```

If your reduction function is just computing the average, we recommend you use a bilinear sampler:

```
AF4 SpdLoadSourceImage(ASU2 p) {
      //invInputSize is additionally passed as constant
      AF2 textureCoord = p * invInputSize + invInputSize;
      return imgSrc.SampleLevel(srcSampler, textureCoord, 0); }
```

INTEGRATION - GPU

Define SPD interface functions

```
void SpdIncreaseAtomicCounter(){
      InterlockedAdd(globalAtomic[0].counter, 1, spd_counter); }
AU1 SpdGetAtomicCounter() { return spd_counter; }



AF4 SpdLoadIntermediate(AU1 x, AU1 y){ … }
void SpdStoreIntermediate(AU1 x, AU1 y, AF4 value){ … }
```

LOAD AND STORE TO LDS

```
AF4 SpdLoadIntermediate(AU1 x, AU1 y){ return spd_intermediate[x][y]; }

void SpdStoreIntermediate(AU1 x, AU1 y, AF4 value){
      spd_intermediate[x][y] = value; }
```

You need to adapt above functions to your LDS setup, e.g. if you only have one channel use:

```
groupshared AF1 spd_intermediate[16][16];

AF4 SpdLoadIntermediate(AU1 x, AU1 y){

      return AF4_x(spd_intermediate[x][y].x); }

void SpdStoreIntermediate(AU1 x, AU1 y, AF4 value){

      spd_intermediate[x][y] = value.x; }
```

CUSTOM REDUCTION FUNCTION

Define your reduction function. Input are the 2x2 quad values, output is one single value.

For example you can compute the average of all 4 values:

```
AF4 SpdReduce4(AF4 v0, AF4 v1, AF4 v2, AF4 v3) {
      return (v0+v1+v2+v3)*0.25; }
```

INTEGRATION – GPU - PACKED

If you use the packed version of FFX SPD, every function has the suffix H and uses the packed types:

```
AH4 SpdLoadSourceImageH(ASU2 p){ … }
AH4 SpdLoadH(ASU2 p){return AH4(imgDst[5][p]);}
void SpdStoreH(ASU2 p, AH4 value, AU1 mip){imgDst[mip][p] = AF4(value);}
```

INTEGRATION - GPU

Setup FFX SPD defines

If you only use the PACKED version of FFX SPD

```
#define SPD_PACKED_ONLY
```

If you use a bilinear sampler to load the source texture (recommended!)

```
#define SPD_LINEAR_SAMPLER
```

If subgroup operations are **not** supported / if you can‘t use SM6

```
#define SPD_NO_WAVE_OPERATIONS
```

Include the FFX SPD header file

```
#include „ffx_spd.h"
```

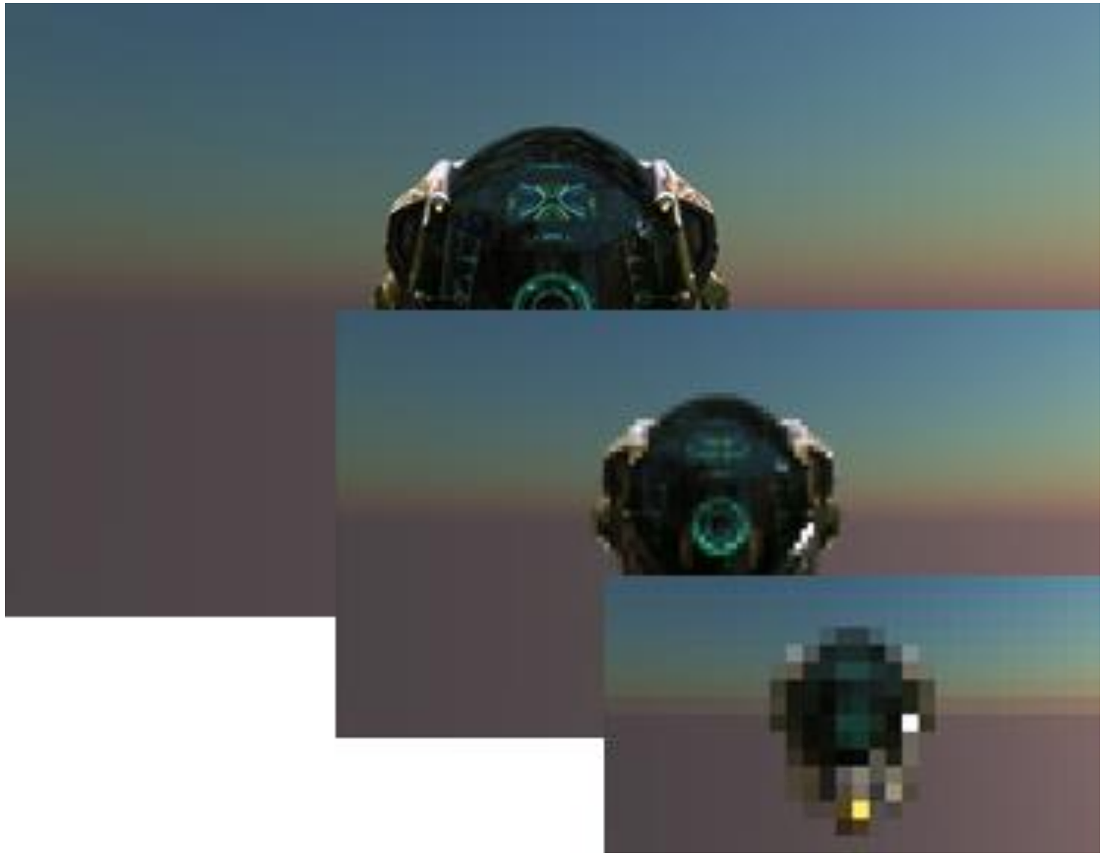
INTEGRATION - GPU

Call the FFX SPD function:

```
[numthreads(256,1,1)]
void main( uint3 WorkGroupId : SV_GroupID, uint LocalThreadIndex : SV_GroupIndex) {

SpdDownsample( AU2(WorkGroupId.xy), AU1(LocalThreadIndex),
               AU1(mips), AU1(numWorkGroups)); //

// PACKED
SpdDownsampleH( AU2(WorkGroupId.xy), AU1(LocalThreadIndex),
                AU1(mips), AU1(numWorkGroups));
};
```



AMD GPUOpen

EPYC RADEON RYZEN AMD

DISCLAIMER & ATTRIBUTION

DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION