A BLEND OF GCN OPTIMIZATION AND COLOR PROCESSING

JORDAN LOGAN & TIMOTHY LOTTES

GDC 2019

JORDAN LOGAN

STORE CACHING

IN SEPARABLE FILTERS

STORE CACHING

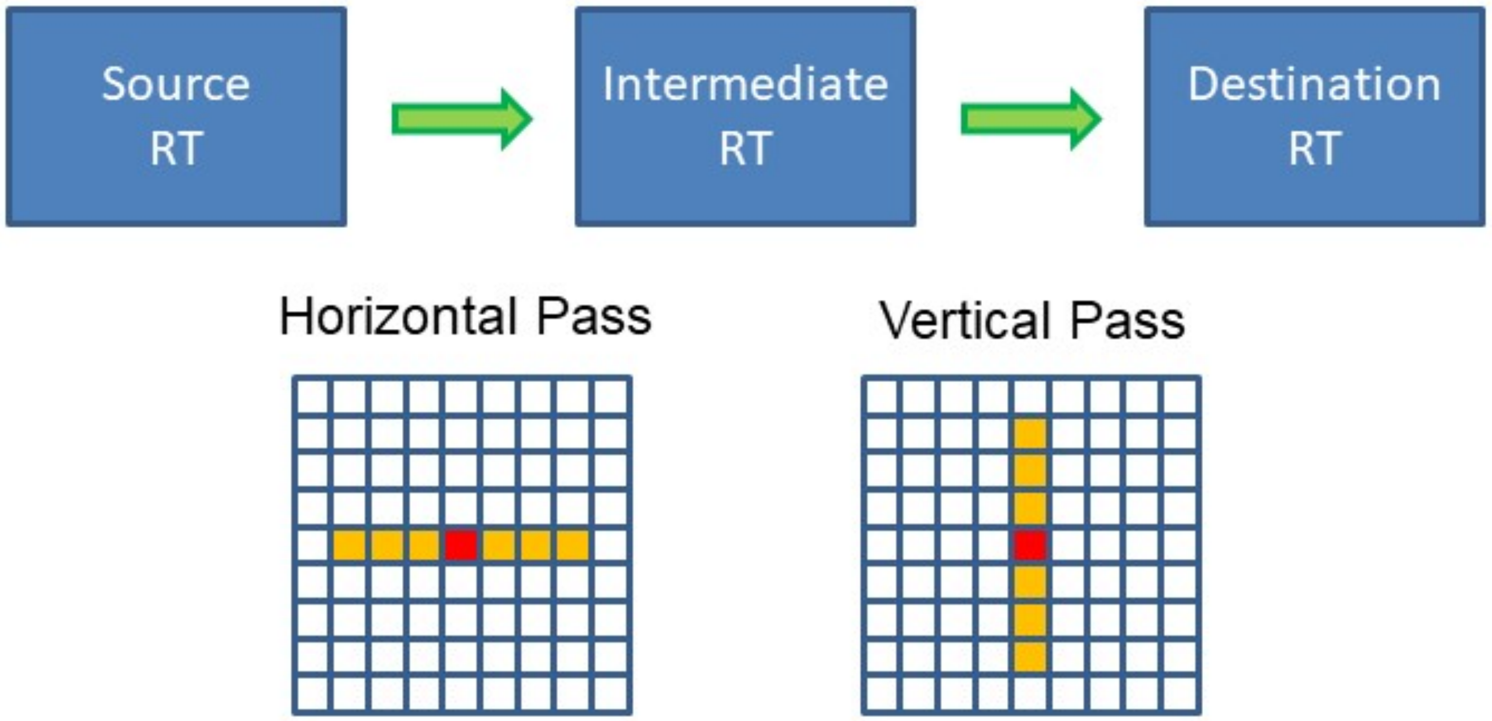
- Follow-up to GDC 2011
  - *"Direct Compute Accelerated Separable Filtering"*

- Shows using group shared memory to cache loads for Separable Filters

Separable Filters

- Much faster than executing a box filter
- Classically performed by the Pixel Shader
- Consists of a horizontal and vertical pass
- Source image over-sampling increases with kernel size
  - Shader is usually TEX instruction limited

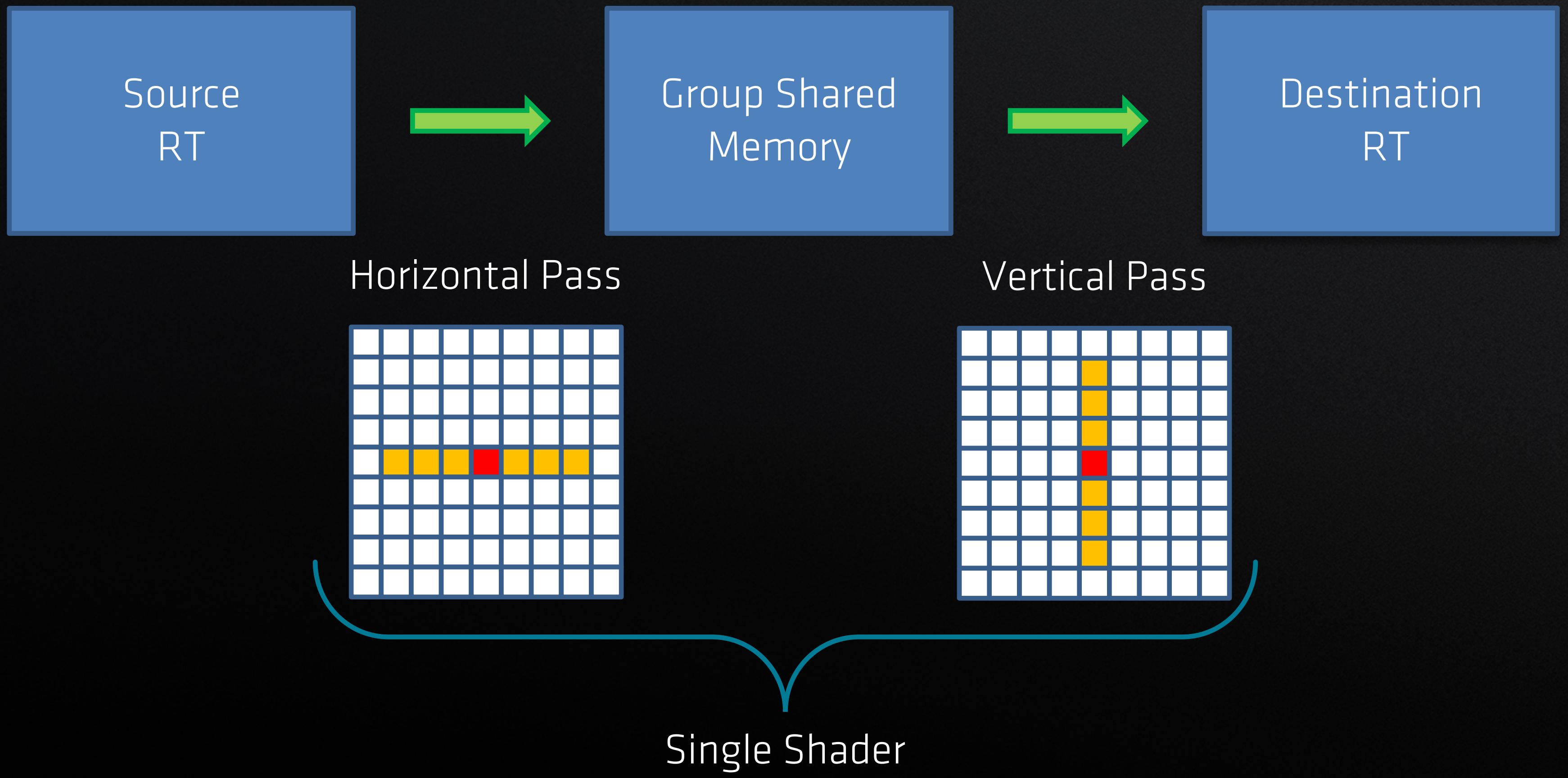
Typical Pipeline Steps



STORE CACHING

- Writing out the intermediate values to a Render Target uses a lot of memory bandwidth
- The data is already on chip so why not keep it there
  - Cache the write in Group Shared Memory
  - Use Group Shared Memory as the source for the second pass

PIPELINE STEPS WITH STORE CACHING



WORKGROUP SIZE

- AMD GPUs run in waves of 64 threads

- Work in 2D to maximize data locality
  - GPUs expect texture accesses to be local in 2D

- Running the waves in 8x8 tiles maximizes locality

MEMORY

- A full column of values won't fit into group shared memory
  - For example a 1080p image would require ~101 KBs

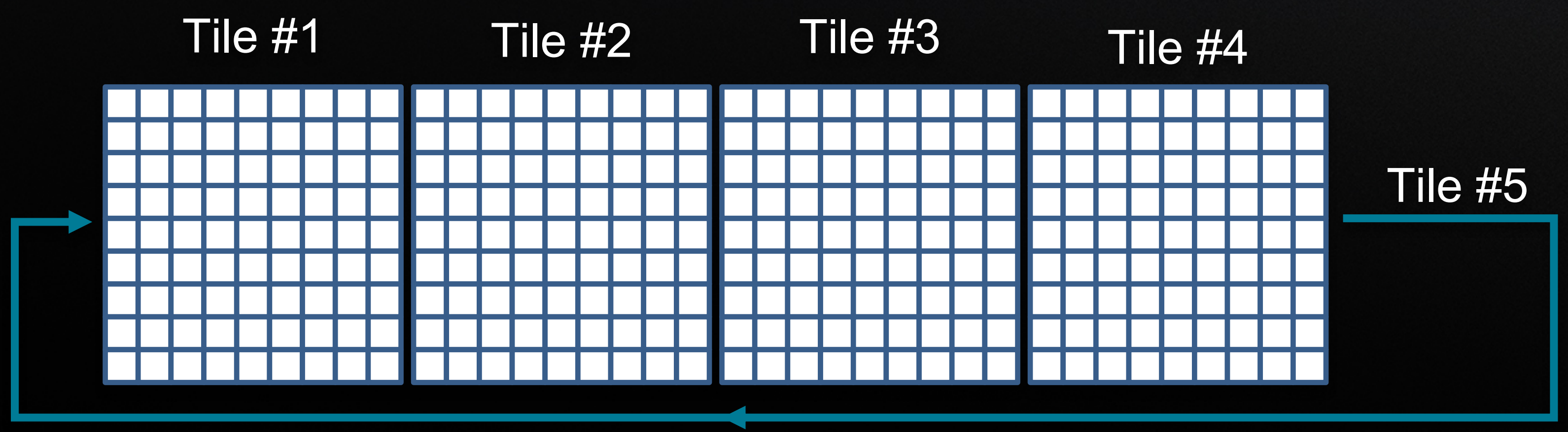
$$\frac{1080\ pixels}{column} * \frac{3\ floats}{pixel} * \frac{4\ bytes}{float} * \frac{8\ columns}{wave} = \frac{103680\ bytes}{column}$$

- The full column should not be needed for every pixel
  - Allows interleaving the 2 passes

- Old data can be discarded once used

RING BUFFER

- A ring buffer can be used for this
  - Min Tiles needed = Ceil(Half Kernel / Tile size) * 2 + 1
- Use a power of 2 to minimize complexity of indexing
  - Allows use of fast bitwise operators
  - Optimal tiles needed = Ceil(Half Kernel / Tile size) * 4



AMD RYZEN RADEON

SCHEDULING FOR STORE CACHING

- A ring buffer requires work to be scheduled in the shader
- Semi-persistent waves can be used to schedule the work manually
  - See the "Engine Optimization Hot Lap" 2018 GDC talk for more about semi-persistent waves

AMD RYZEN RADEON

OCCUPANCY

- Need a lot of waves to fill a GPU
  - 1920 / 8 = 240 waves
  - 64CUs * 4 SIMD/CU = 256 waves in flight
  - <1 wave occupancy = ☹

OCCUPANCY

- Need a lot of waves to fill a GPU
  - 1920 / 8 = 240 waves
  - 64CUs * 4 SIMD/CU = 256 waves in flight
  - <1 wave occupancy = ☹
- Naive Solution
  - Change workgroup size to 4x16, 2x32, or 1x64
  - Reduces cache hit rate

OCCUPANCY

- Need a lot of waves to fill a GPU
  - 1920 / 8 = 240 waves
  - 64CUs * 4 SIMD/CU = 256 waves in flight
  - <1 wave occupancy = ☹
- Naive Solution
  - Change workgroup size to 4x16, 2x32, or 1x64
  - Reduces cache hit rate
- Better Solution
  - Change workgroup size to 8x16, 8x32, 8x64
  - 8x32 is a local maximum for performance
  - Be careful of running out of Group Shared Memory

EDGE CASES

- Image edges require some extra consideration
- An if statement used when reading from store cache can generate unwanted branches
- A fast approach is to just fill the cache with the border color at image edges

AMD RYZEN RADEON

IMPLEMENTATION DETAILS

- Step 1:
  - Pre fill the Store cache
  - Fill the rest of the cache with border value
  - Sync all waves in group

IMPLEMENTATION DETAILS

- Step 2:
  - Loop over column
    - Load new tile of data into the cache for tile n + 1
    - Horizontal pass
    - Sync all waves
    - Vertical pass using values in cache for tile n
    - Save output to texture
  - Sync all waves in group

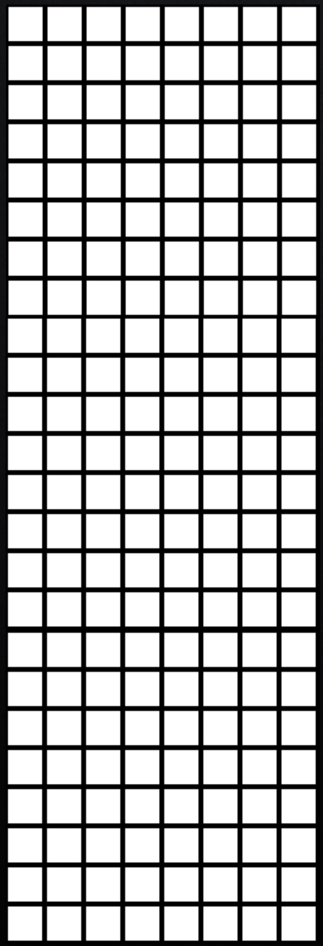
IMPLEMENTATION DETAILS

- Step 3:
    - No more pixels to read but still have some tiles to write out
  - Loop for remaining number of tiles
    - Load border color into cache
    - Sync all waves in group
    - Vertical Pass using values in cache
    - Save output to texture

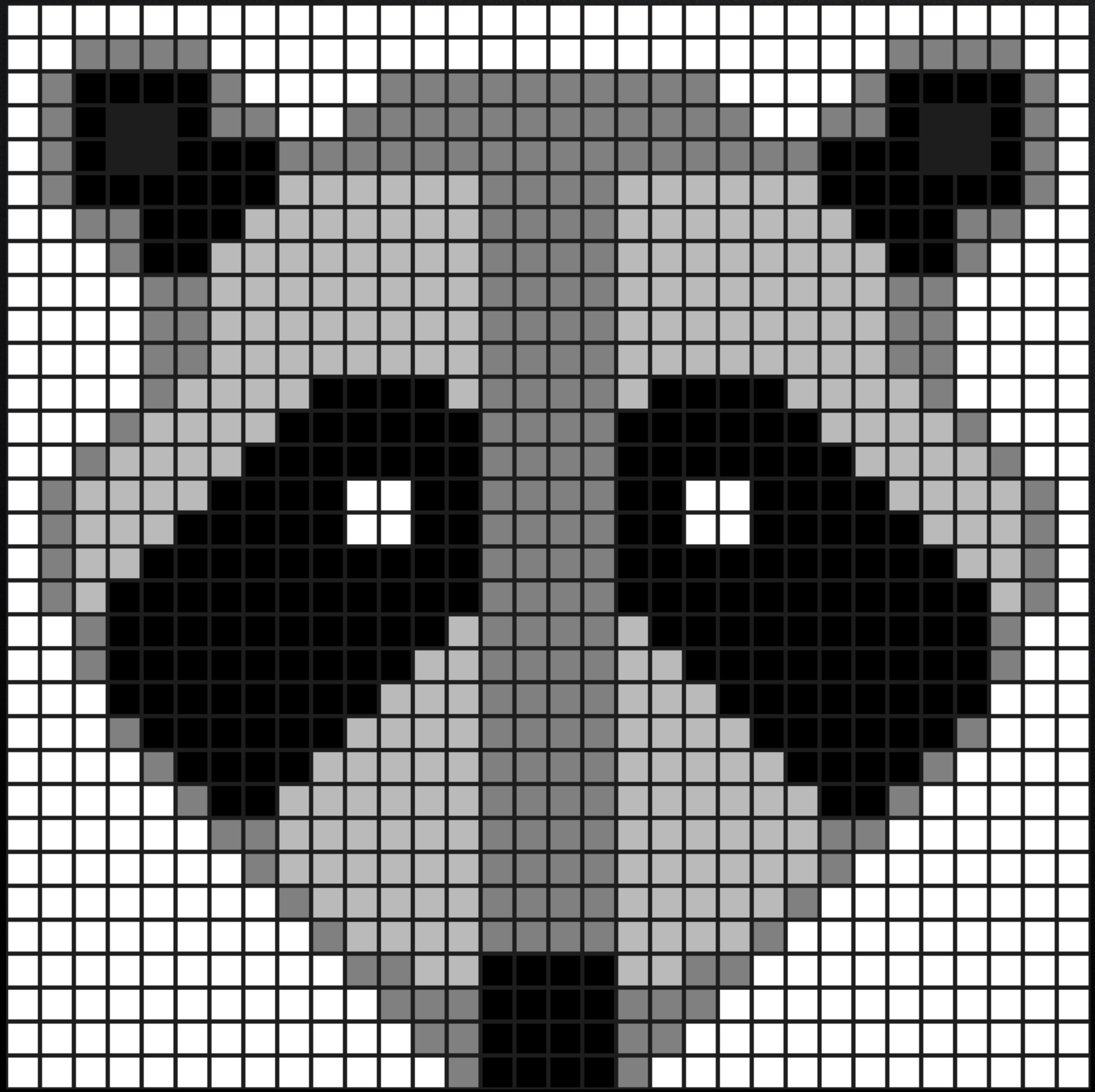
Start with
the store cache
filled with
border color

Red = Read
Green = Write

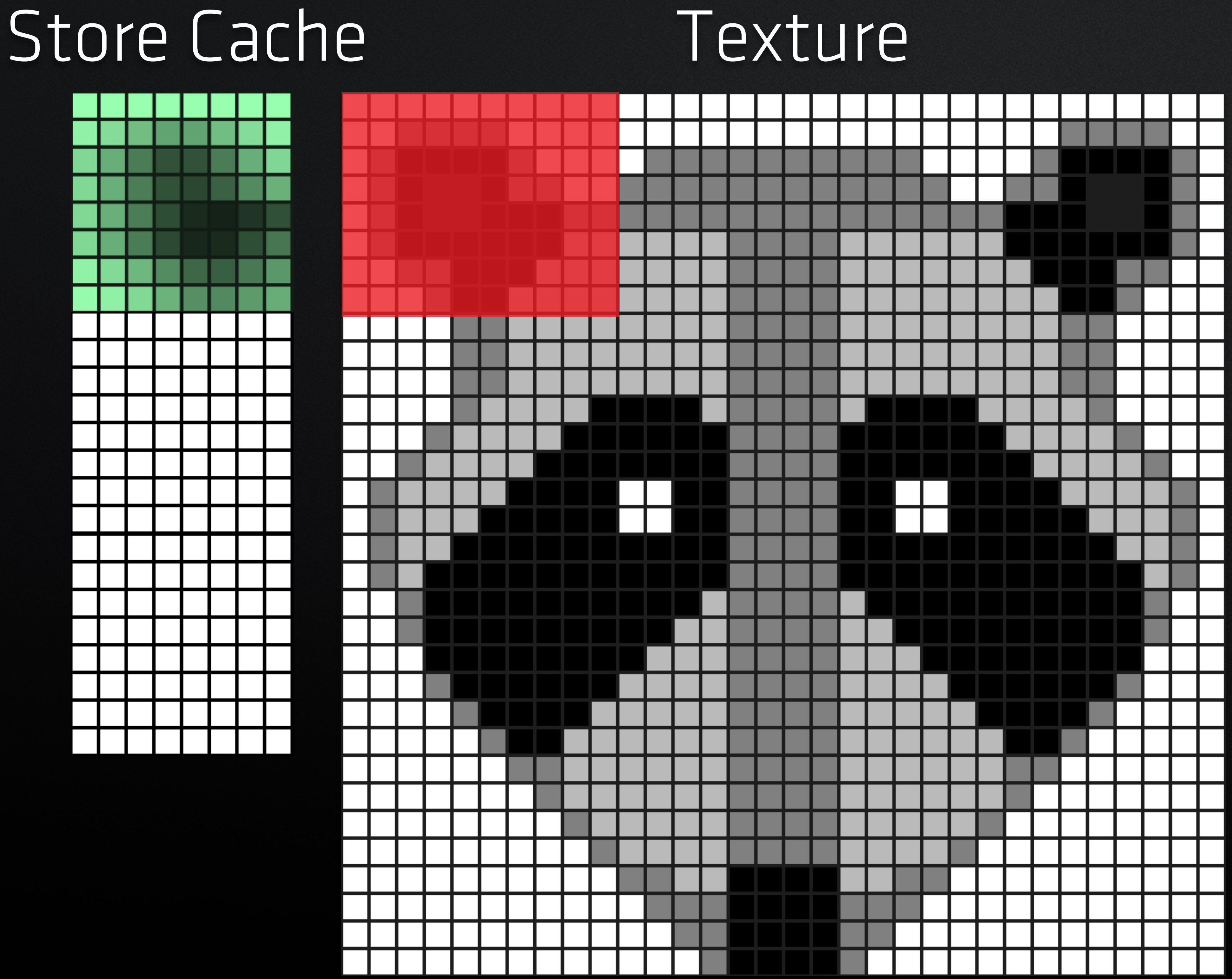
Store Cache



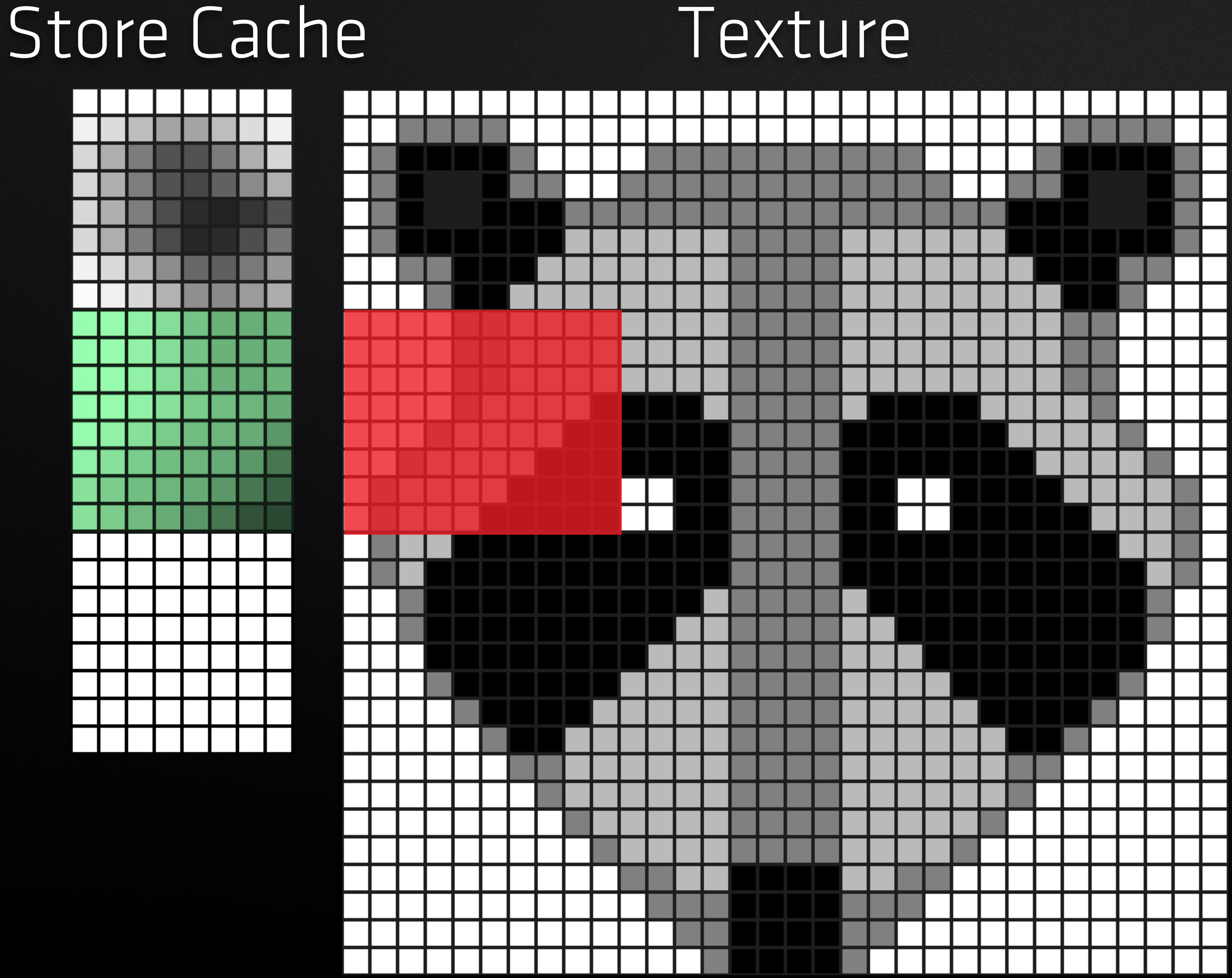
Texture



Read in first tile of data



Red = Read
Green = Write



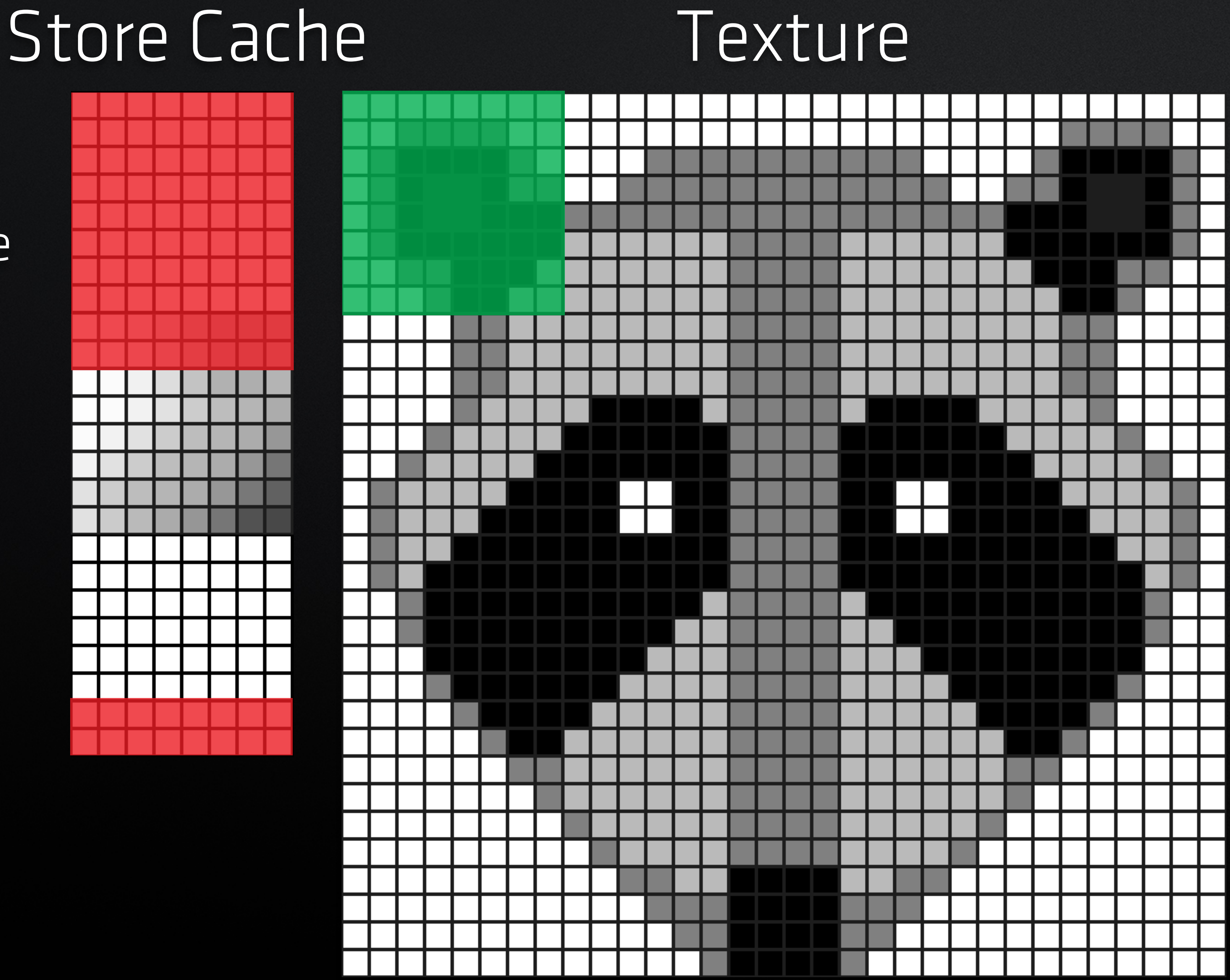
Read in next tile of data

Red = Read

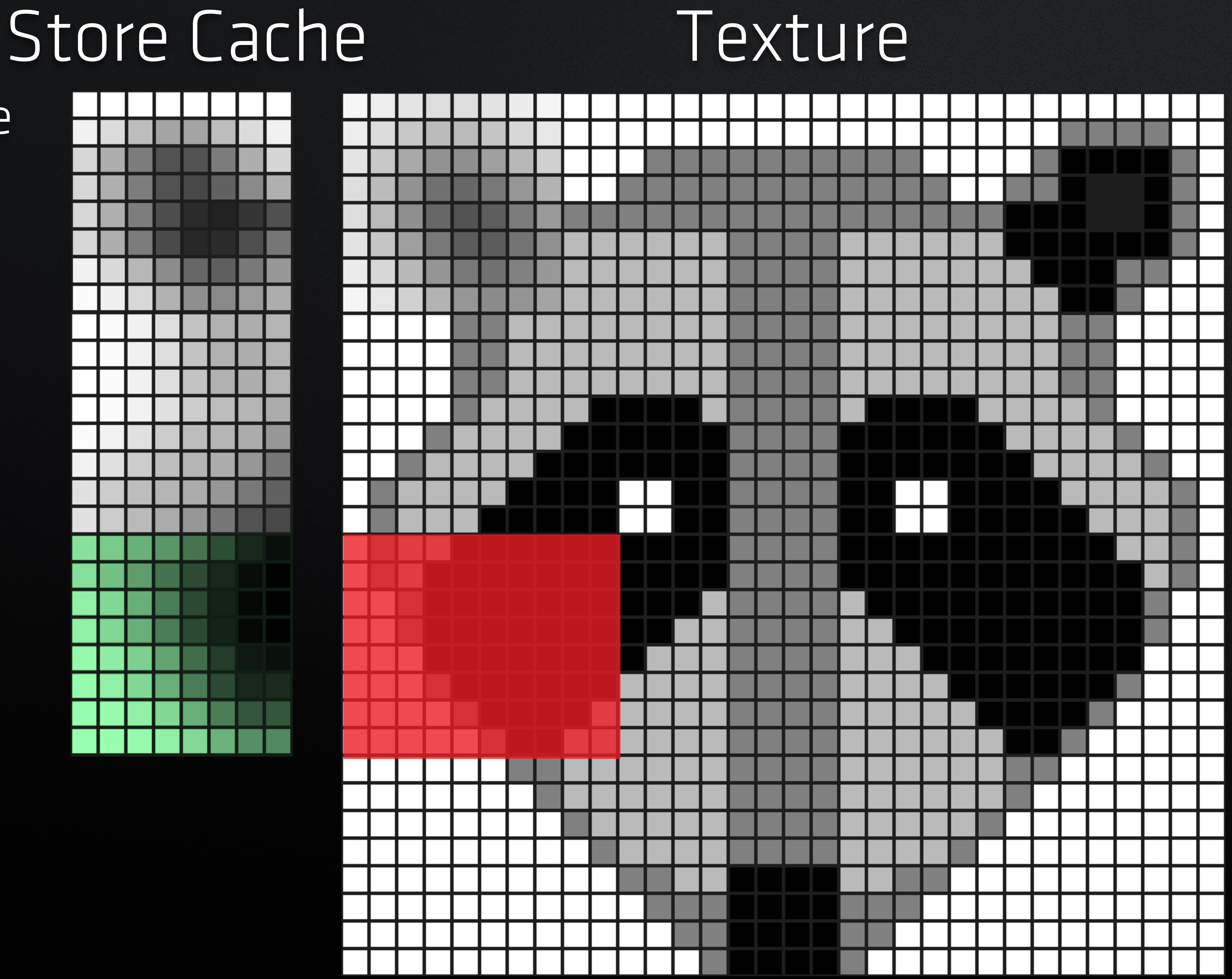
Green = Write

Read values
from cache and
write out to texture

Red = Read
Green = Write



Read in another tile of data

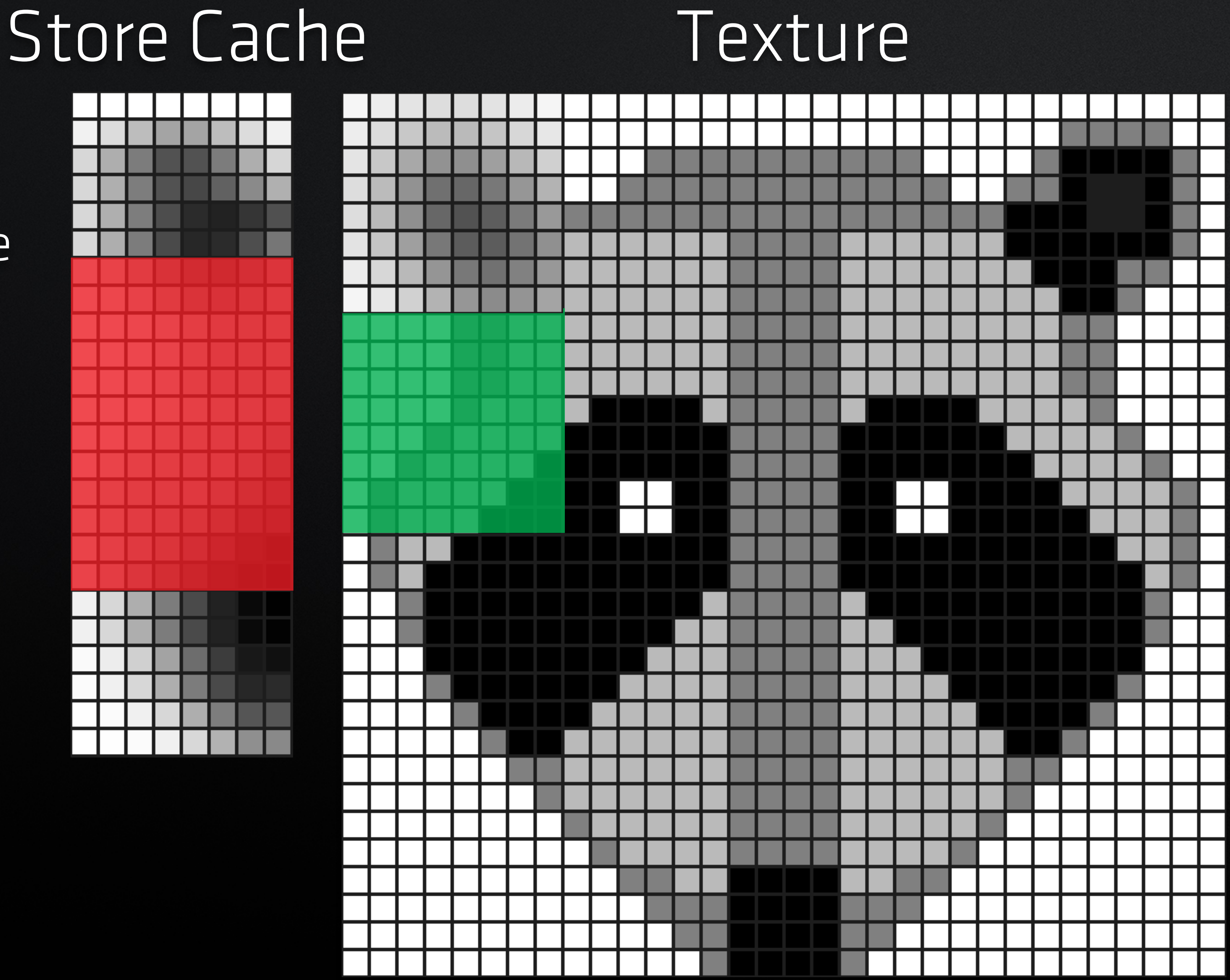


Red = Read

Green = Write

Read values
from cache and
write out to texture

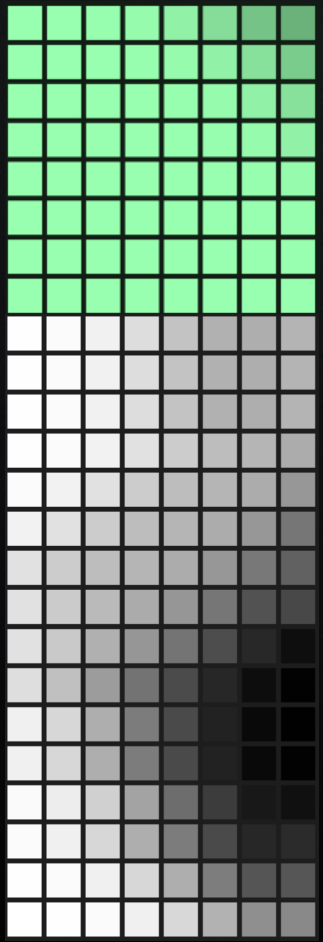
Red = Read
Green = Write

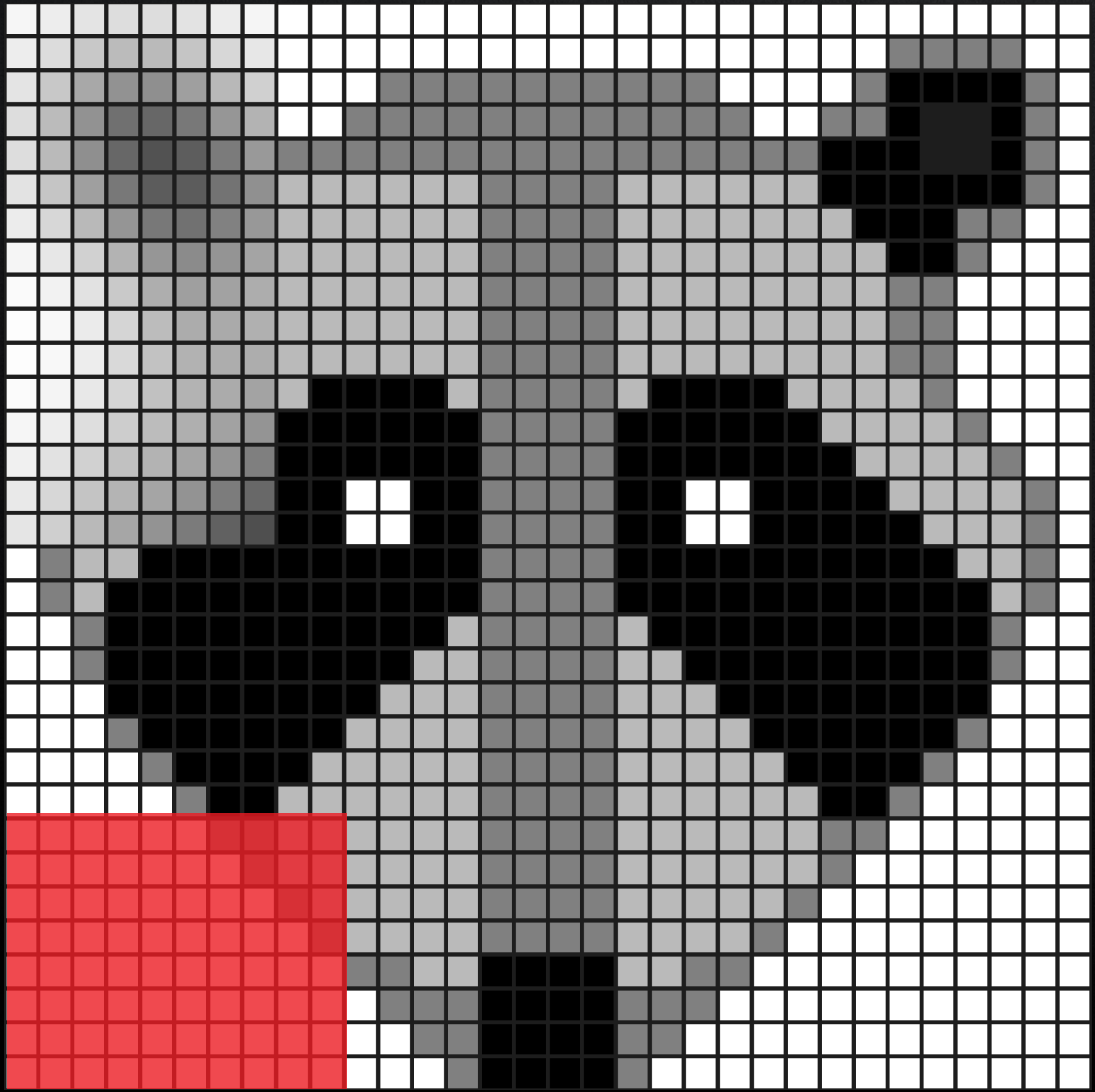


Store Cache

Texture

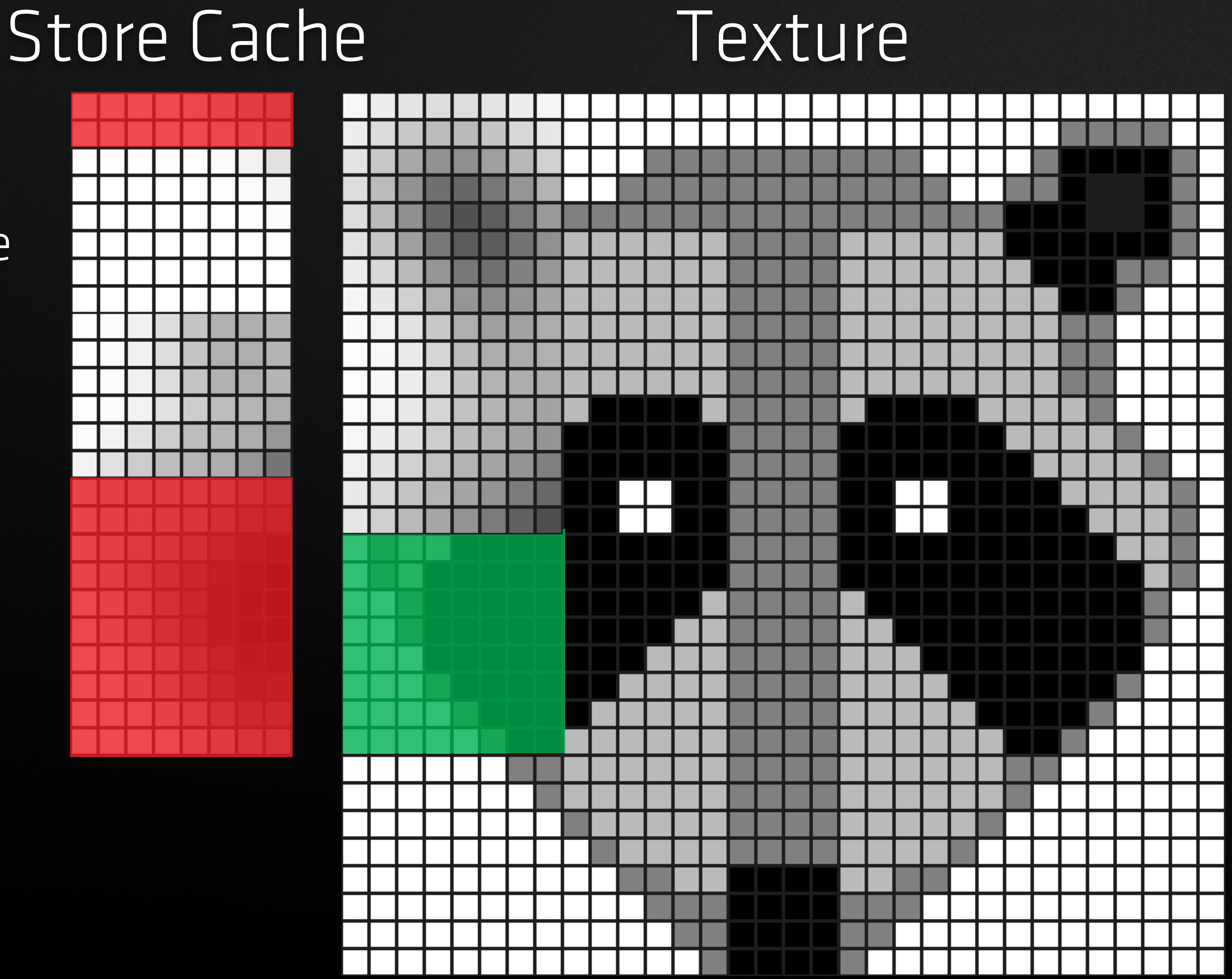
Read in tile of data





Red = Read

Green = Write



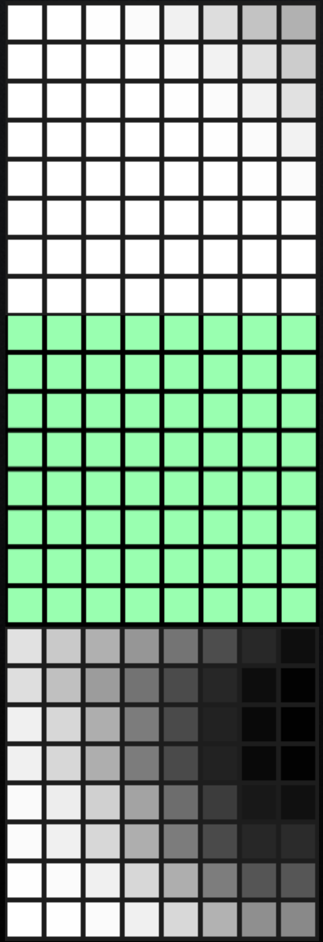
Read values
from cache and
write out to texture

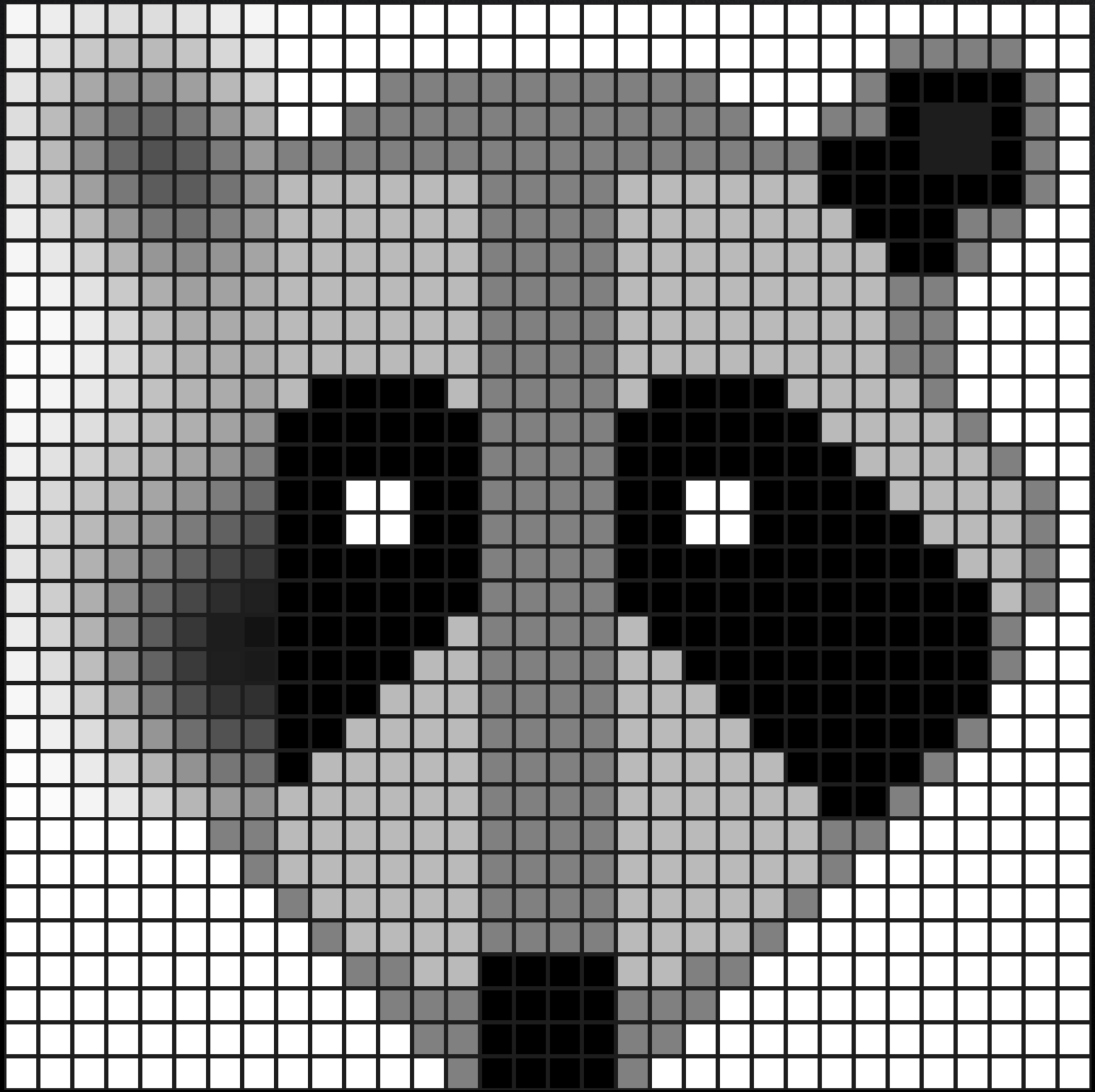
Red = Read
Green = Write

Store Cache

Texture

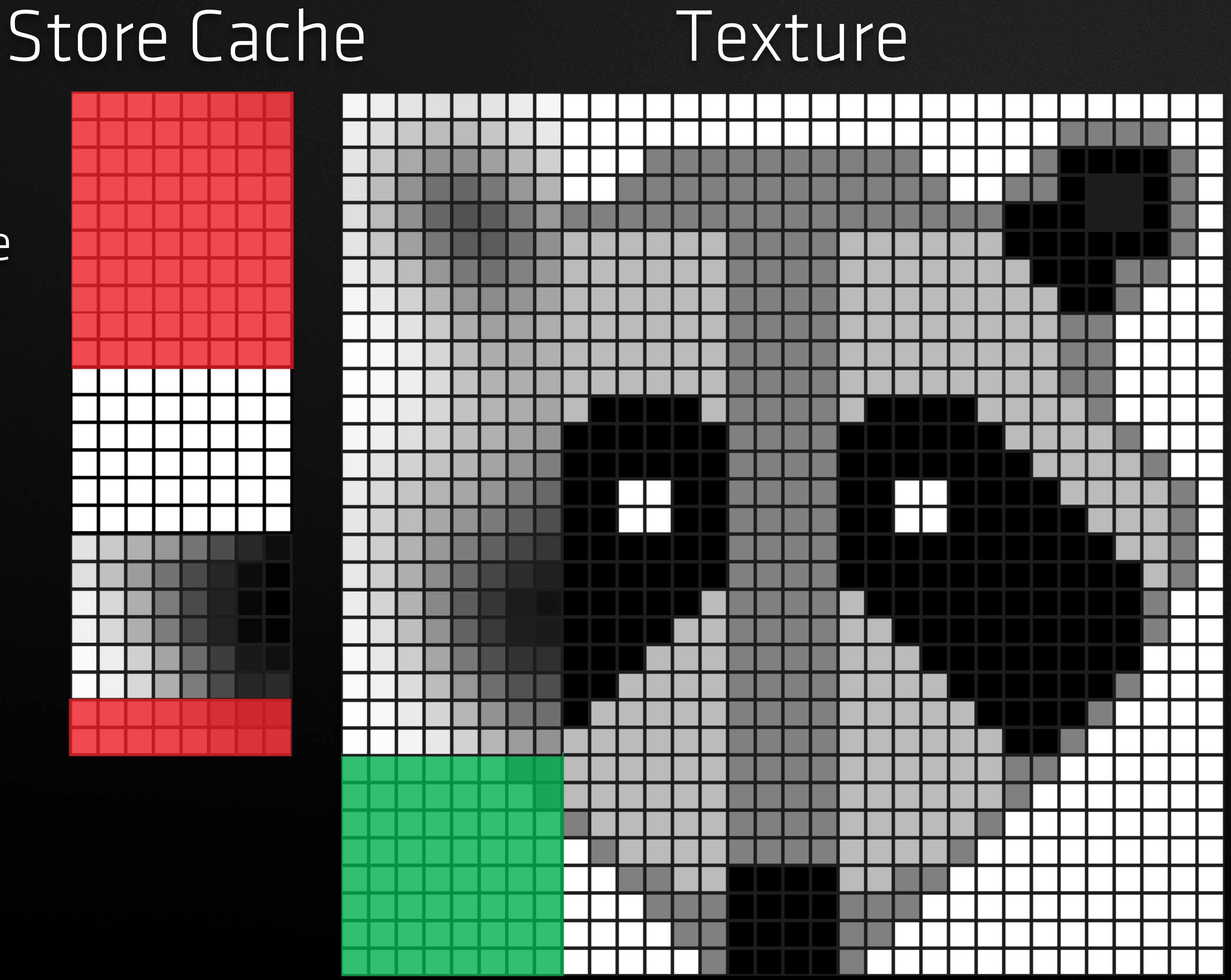
Fill next tile
with border color





Red = Read
Green = Write

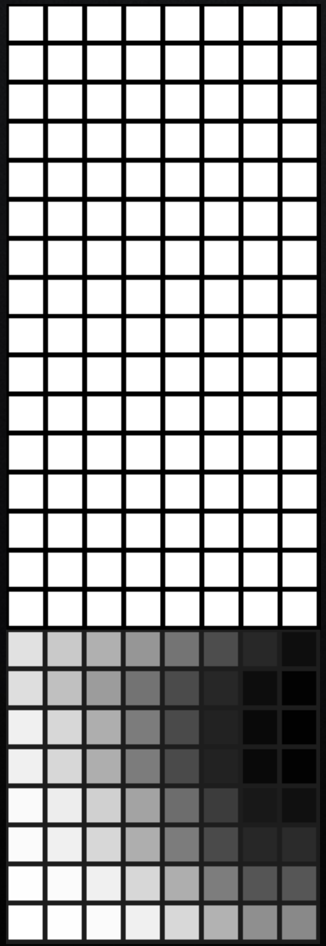
Read values from cache and write out to texture



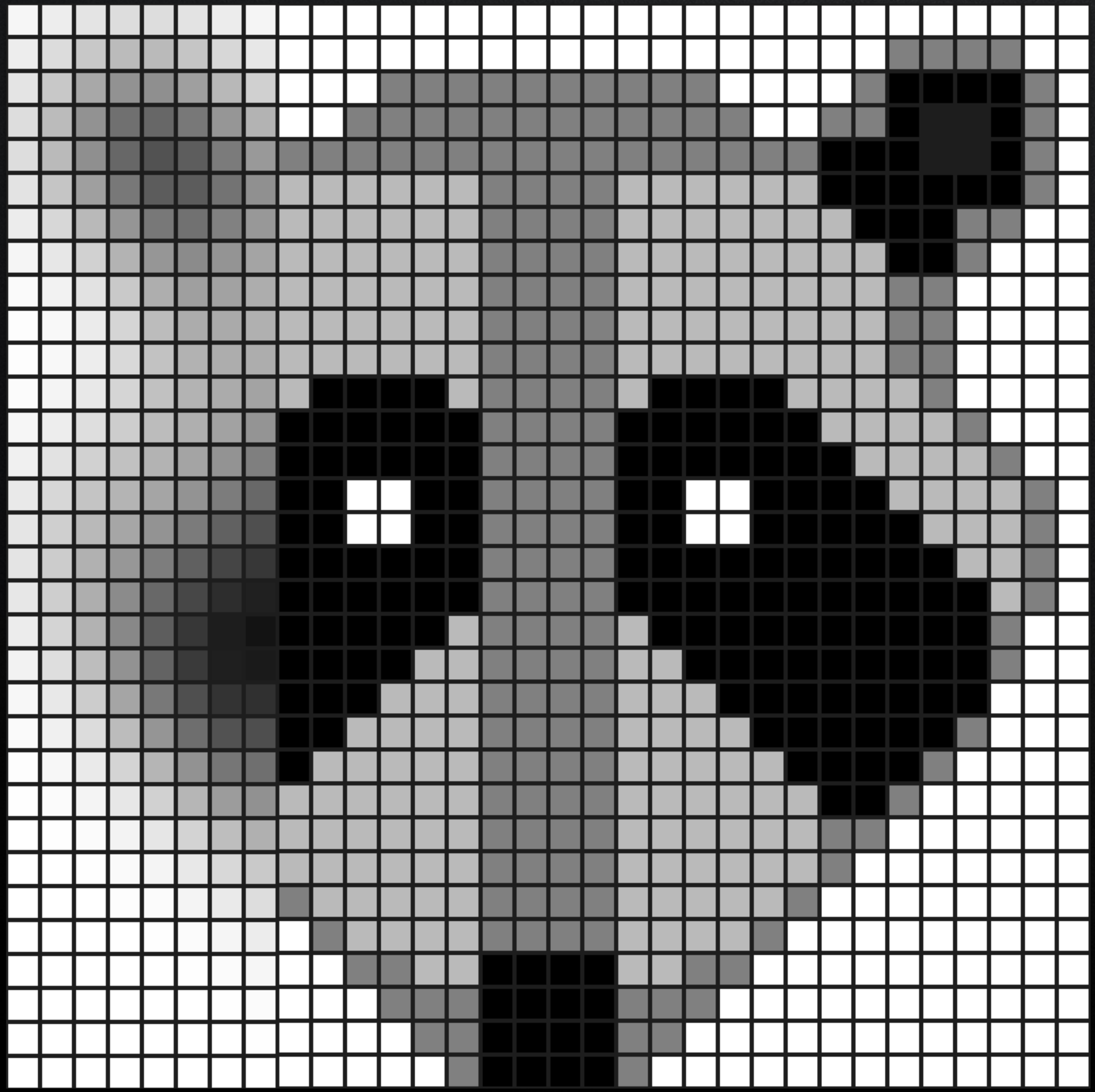
Red = Read

Green = Write

Store Cache



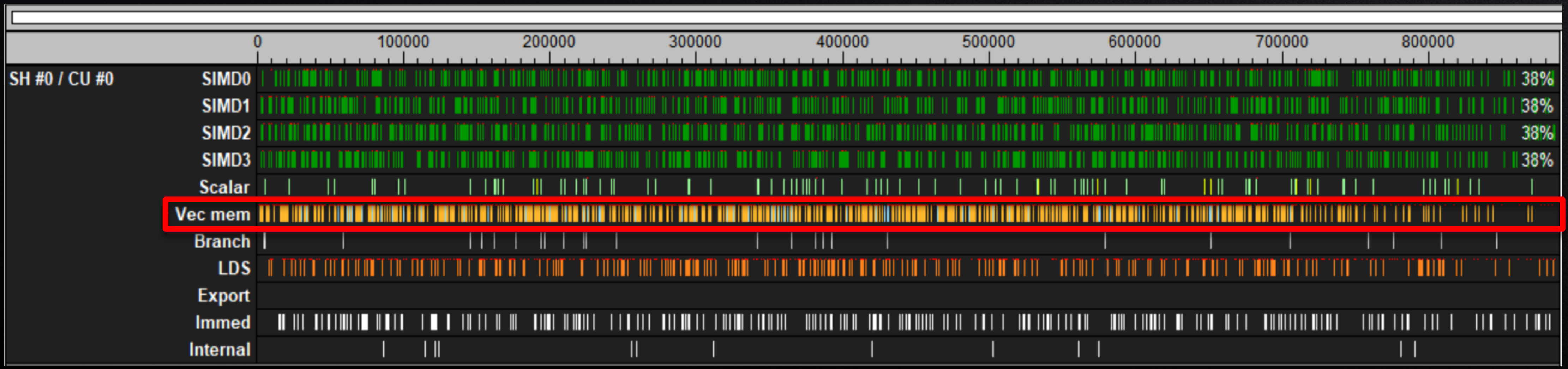
Texture

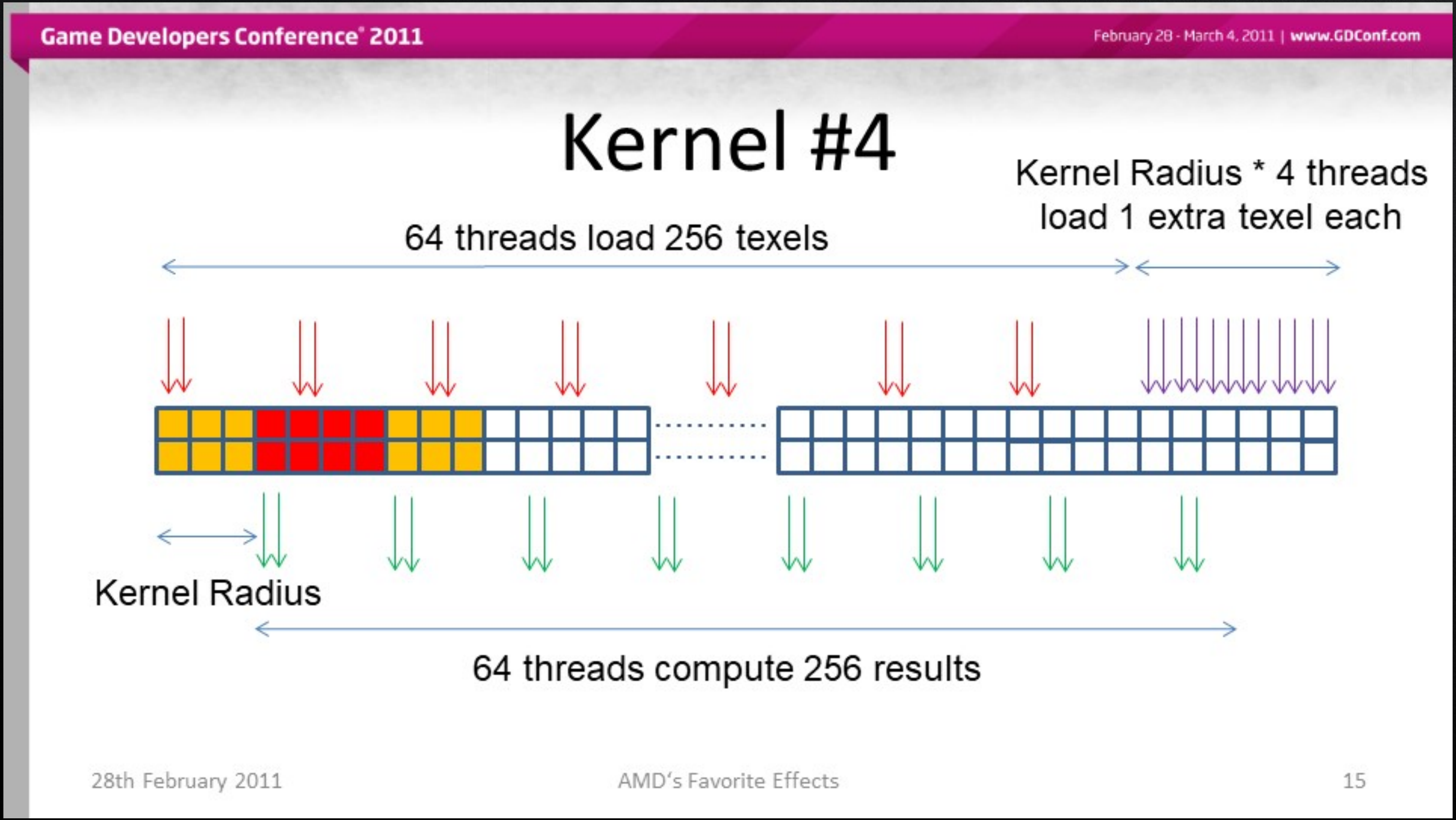


OPTIMIZING

BOTTLENECK

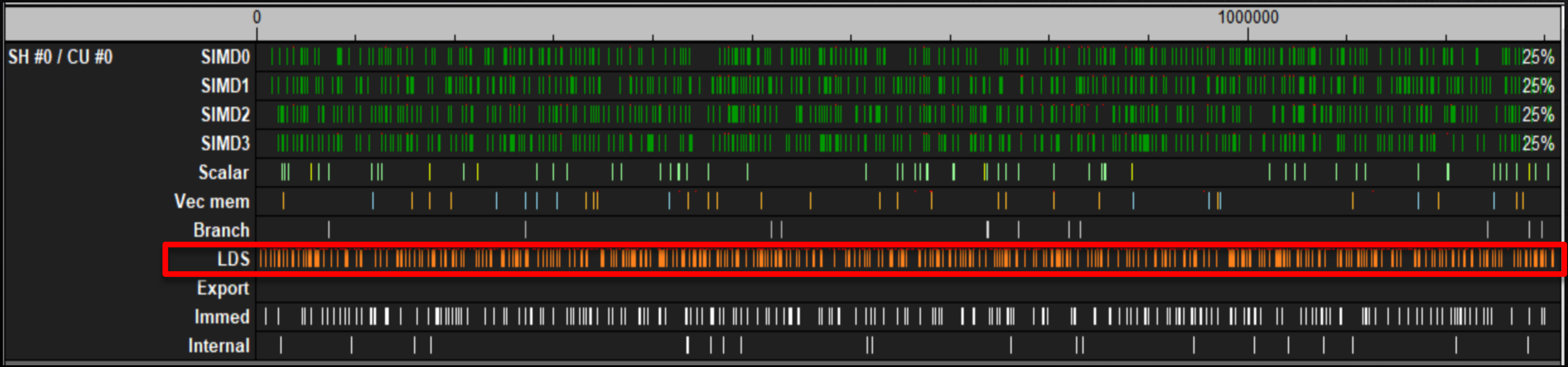
- This implementation was bandwidth bound
- High number of texture loads per pixel
- Load caching can be used to reduce number of texture loads




Game Developers Conference® 2011
February 28 - March 4, 2011 | www.GDConf.com
Kernel #4
Kernel Radius * 4 threads
load 1 extra texel each
64 threads load 256 texels
Kernel Radius
64 threads compute 256 results
28th February 2011
AMD's Favorite Effects
15

BOTTLENECK

- Load caching moved the bottleneck to LDS
- It is also running slower than before



LDS

- Thread group shared memory maps to LDS (Local Data Share)
- LDS memory is banked on GCN
  - It's spread across 32 banks
  - Each bank is 32bits (1 dword)
- Bank conflicts increases latency of instruction
  - Can take up to 64 clocks

LDS

- Use Structure of Arrays (SoA) over Array of Structure (AoS) to reduce potential conflicts
  - Can reduces stride of reads and writes
  - Mileage depends on how data is accessed
- GCN design supports multi dword accesses to LDS
  - Keep the array data type 128bits or less
  - Keep it 64bits or less for older generation support
- Note: Float3 will be padded to 128 bits
  - Deinterleaving float3s can be used to save memory

LDS

Example:

```
groupshared float4 LDS_Cache[64]; // Array of structs
void Store(int index, float4 value)
{
  LDS_Cache[index].xyzw = value; // will unroll to 4 reads
}
```

LDS BANKING

Array of Structs

x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w

LDS BANKING

Array of Structs

x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w

8 bank conflicts 

LDS

Example:

```
groupshared float LDS_Cache[64 * 4]; // Struct of Arrays
void Store(int index, float4 value)
{
  LDS_Cache[index  + X_OFFSET] = value.x;
  LDS_Cache[index  + Y_OFFSET] = value.y;
  LDS_Cache[index  + Z_OFFSET] = value.z;
  LDS_Cache[index  + W_OFFSET] = value.w;
}
```

LDS BANKING

Struct of Array

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

LDS BANKING

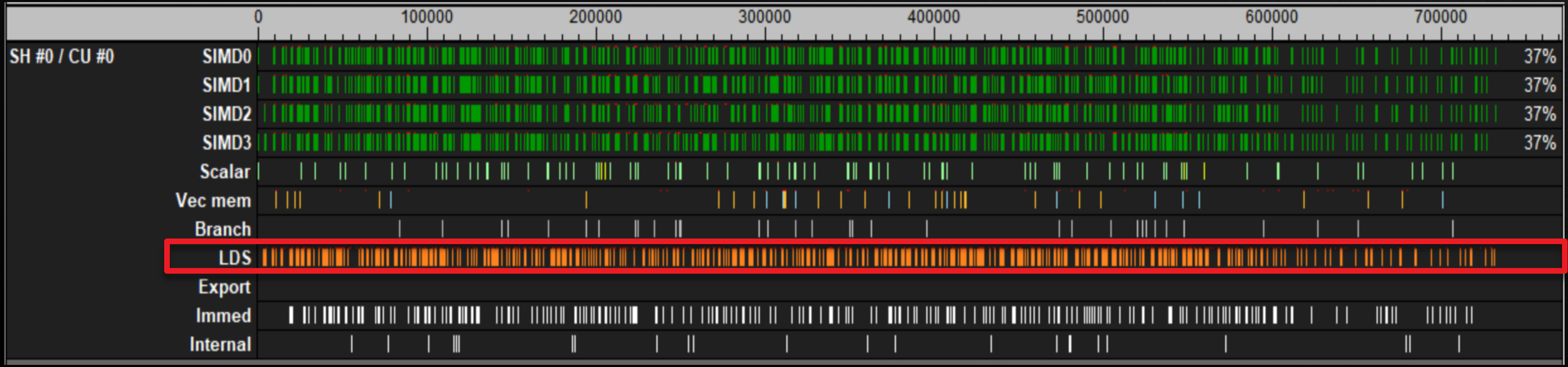
Struct of Array

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

2 bank conflicts

BOTTLENECK

- Back to expected speeds from using load caching
- Less time spent in LDS
  - It can be reduced farther by packing data



Use Packing in TGSM

- Use packing to reduce storage space required in TGSM
  - Only have 32k per SIMD
- Reduces reads/writes from TGSM
- Often a uint is sufficient for color filtering
- Use SM5.0 instructions f32tof16(), f16tof32()

AMD RYZEN RADEON

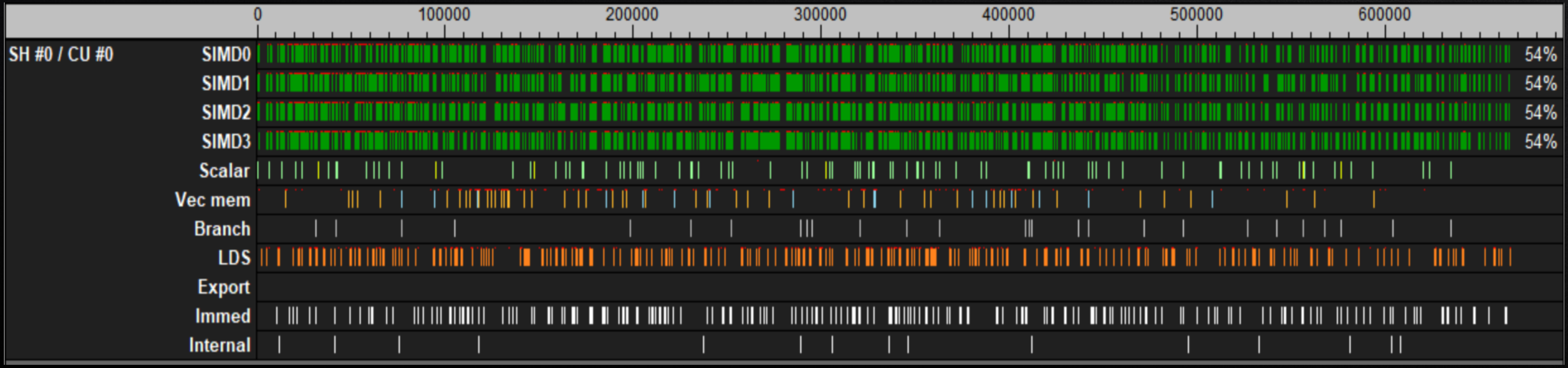
PACKING

- Float3 packing
  - Store x and y into a uint using fp16
  - Keep z in a float
  - If using luminance based color spaces,
    the luminance can be stored into the 32 bit float for the extra precision

- Float4 packing
  - Store x and y into a uint using fp16
  - Store z and w into a uint using fp16

BOTTLENECK

- Time spent in LDS is down
- Bottleneck moved towards ALU



NUMBERS

Kernel Size	Separated passes	Store caching	Store caching Load caching	Store caching Load caching SOA	Store caching Load caching SOA Packed
5	780 us	460 us	810 us	580 us	470 us
9	950 us	600 us	1020 us	580 us	470 us
17	1250 us	910 us	1670 us	730 us	720 us

Testing done by Jordan Logan using a sample framework running on a 4K image on January 14, 2019 with the following system. PC manufacturers may vary configurations yielding different results. Results may vary based on driver versions used. Test configuration: AMD Ryzen™ 7 1800x Processor, 2x16GB DDR4-2666, Vega64 Frontier Edition (driver 19.3.1), ASUS Prime X370-PRO Socket AM4 motherboard, WD Blue 250GB M.2 SSD, Windows 10 x64 Pro (RS4).

PROS / CONS

- Pros
  - Requires one barrier per blur
  - Reduced bandwidth
  - Reduced memory requirements
  - FASTER!
- Cons
  - Large kernels can put heavy pressure on LDS

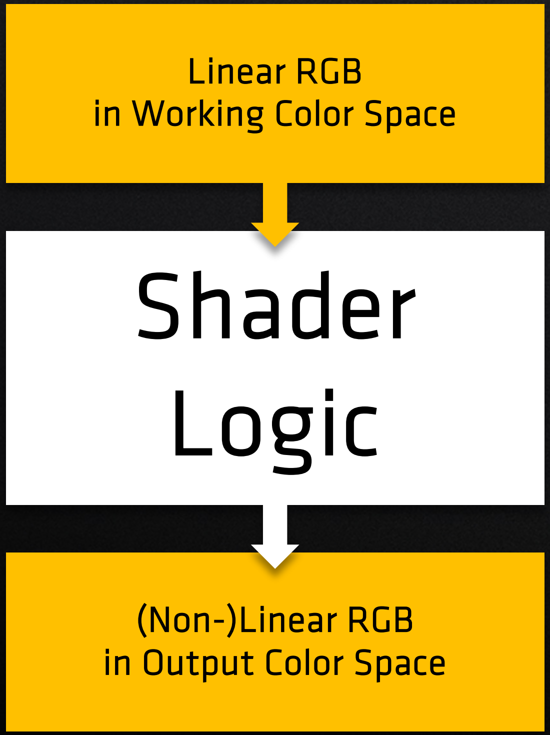
REFERENCES

- [Engine Optimization Hot Lap]
- [DirectCompute Accelerated Separable Filters]

TIMOTHY LOTTES

GENERALIZED

TONE-MAPPING



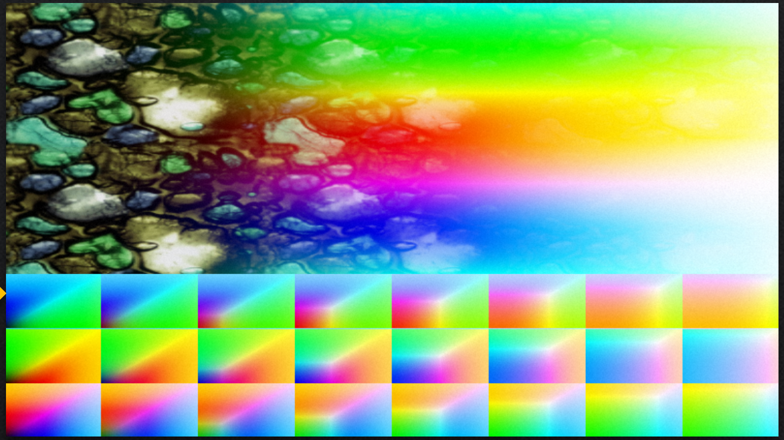
A NEW "GENERALIZED TONE-MAPPER (GTM)"

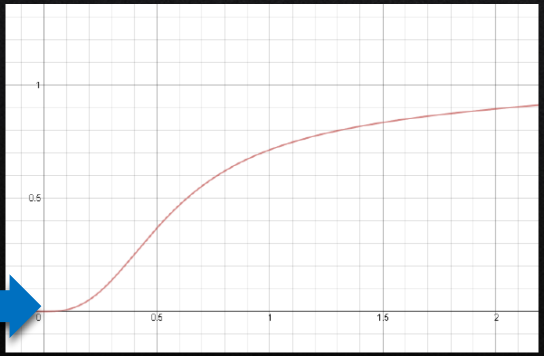
- This is temporary naming just for these slides
- Look for a related GPUOpen release
  - https://gpuopen.com/games-cgi/
- Portable Shader Header
  - #defines to select options and configure between HLSL/GLSL/C
- Follow-up to GDC 2016
  - *"Advanced Techniques and Optimization of VDR Color Pipelines"*
  - https://gpuopen.com/gdc16-wrapup-presentations/

AMD RYZEN RADEON

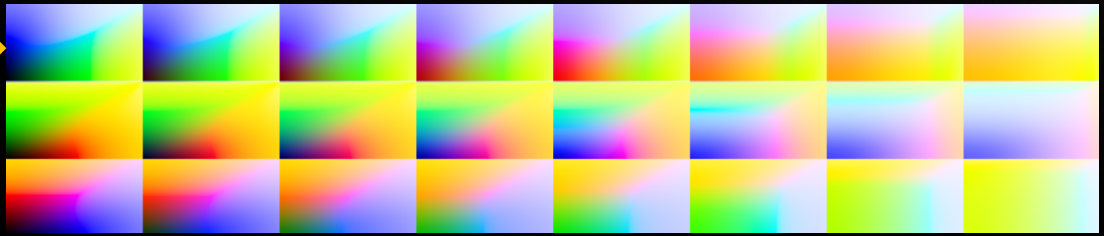
THE PRIOR VERSION

- Incorporated into a sample here
  - https://www.shadertoy.com/view/XljBRK
- GTM expands on prior version
  - Uses the prior tone-mapping curve but applies it to luma instead
  - Adds gamut-mapping
  - Simplifies over-exposure color shaping
  - Targets luma preservation
    - tonemap(luma(RGB)) is similar to luma(tonemap(RGB))





Cleaner Over-Exposure



COLOR GOALS – ONE SIMPLE COLOR PIPELINE

- Master content once and target any display
  - Same color pipeline

- Any positive linear RGB color-space input
  - sRGB, DCI-P3, Rec.2020, or custom primaries

- To any RGB color-space output
  - CRT, Rec.709, sRGB, HDR10, HLG, FreeSync2, etc

AMD RYZEN RADEON

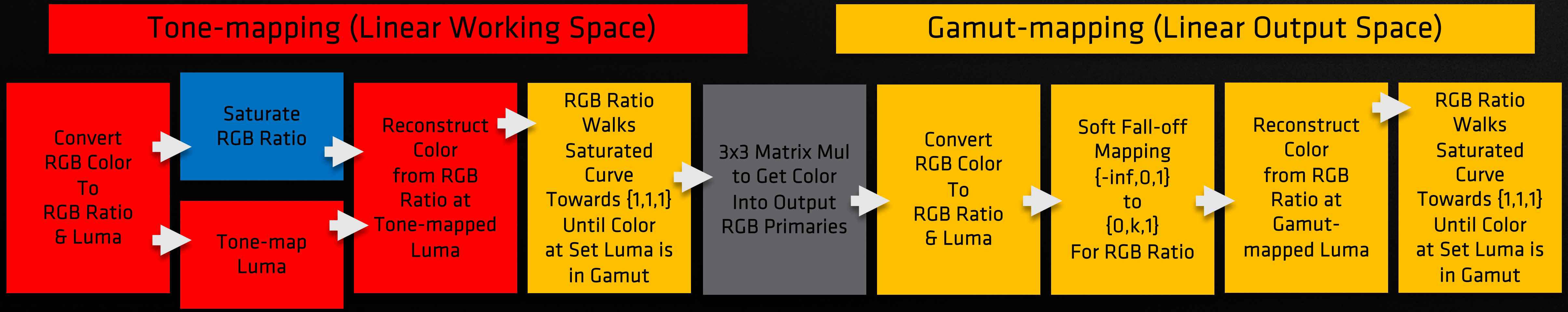
THEORY FOR KEEPING COLOR SIMPLE

- Have both tone-mapping and gamut-mapping not re-grade the image when exposure changes
  - Avoid problems caused by tone-mapping color channels separately
  - sRGB and HDR10 outputs require vastly different exposure
    - A shader does the full tone-mapping for sRGB
    - A shader does only tone-mapping to 10000 nits for HDR10 (display does the rest)

- Exception when over-exposed color must be brought in-gamut
  - Use output-specific shaping of color

ALGORITHM – USED TO GENERATE THE LUT

- Maintains RGB ratio, RGB/max3(R,G,B), when in gamut
  - To avoid re-grading the image when possible
- Maintains tone-mapped luma when gamut-mapping color
  - Designed for smooth fall-off on over-exposure and over-gamut mapping



GAMUT MAPPING COMPONENTS

- Adjusting RGB ratio on over-exposure
  - Done twice in algorithm
  - *"Walking Back in Gamut"* slides

RGB Ratio Walks Saturated Curve Towards {1,1,1} Until Color at Set Luma is in Gamut

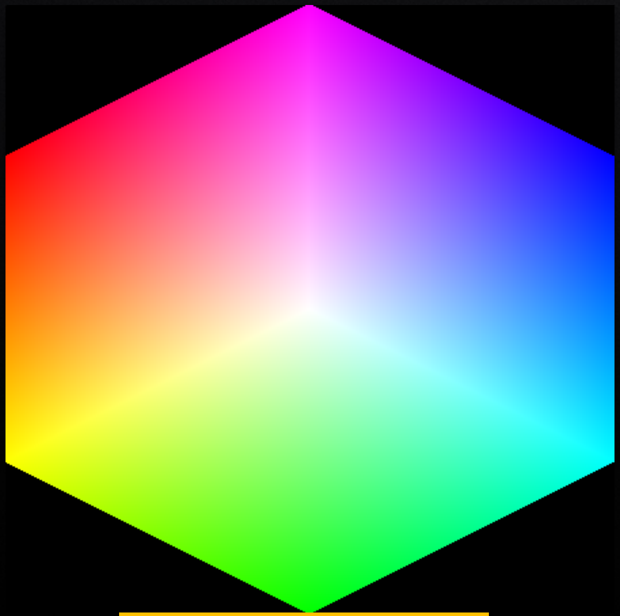
- Map RGB working space to smaller RGB output space
  - Done once to make all RGB values positive
  - *"Soft Fall-off Mapping"* slides

Soft Fall-off Mapping {-inf,0,1} to {0,k,1} For RGB Ratio

WALKING BACK IN GAMUT – RGB RATIO AND LUMA

- White {1,1,1} has peak luma (luma=1.0)
  - Other ratios of RGB primaries have luma<1.0

RGB Ratio Walks Saturated Curve Towards {1,1,1} Until Color at Set Luma is in Gamut

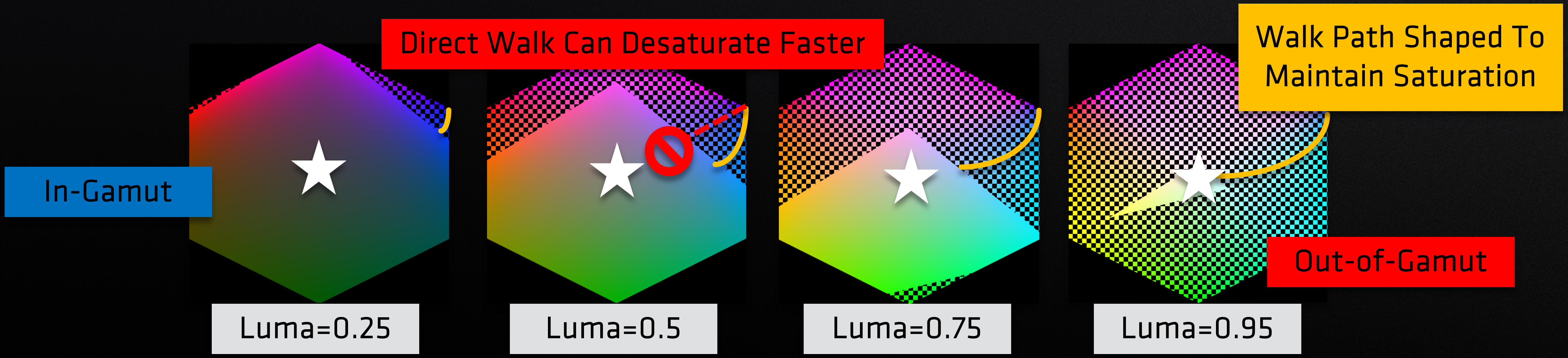


RGB Ratio

Associated sRGB Luma

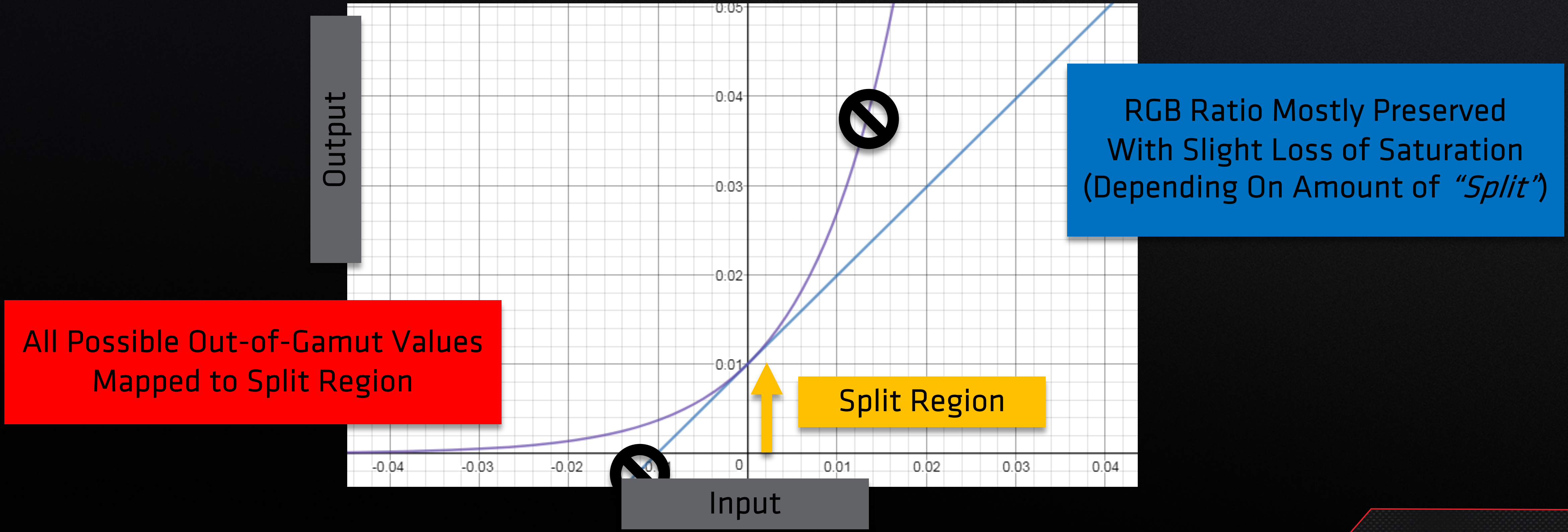
WALKING BACK IN GAMUT – PRESERVING LUMA

- Tone-mapper output {0 to 1} luma regardless of color
  - For a target luma, some RGB ratios will be out-of-gamut
    - Not possible to reproduce luma=1 of pure blue RGB ratio={0,0,1}
  - Algorithm walks RGB ratio towards {1,1,1} until in-gamut



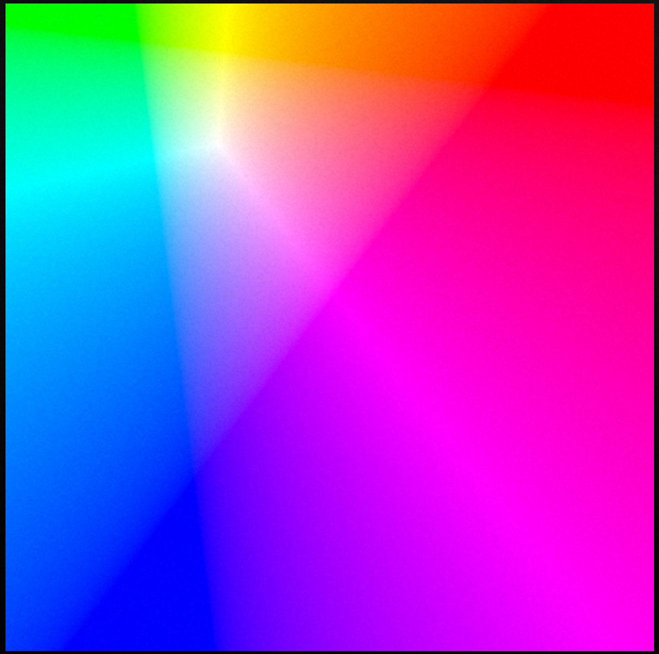
SOFT FALL-OFF MAPPING – 2-PIECE CURVE

- Map {-inf,0,1} RGB ratio component values
  - To {0,split,1} where *"split"* sets amount of gamut for feather

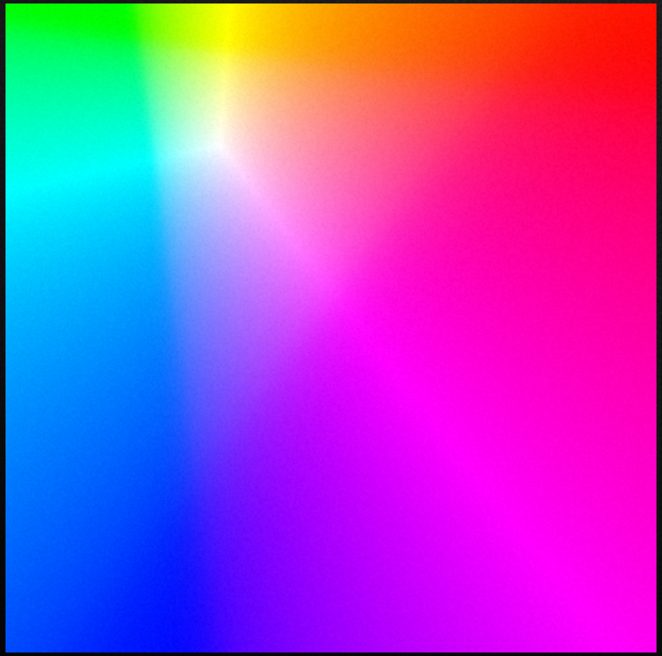


SOFT FALL-OFF MAPPING – VISUALIZED

- CIE1976 visualization of mapping to sRGB

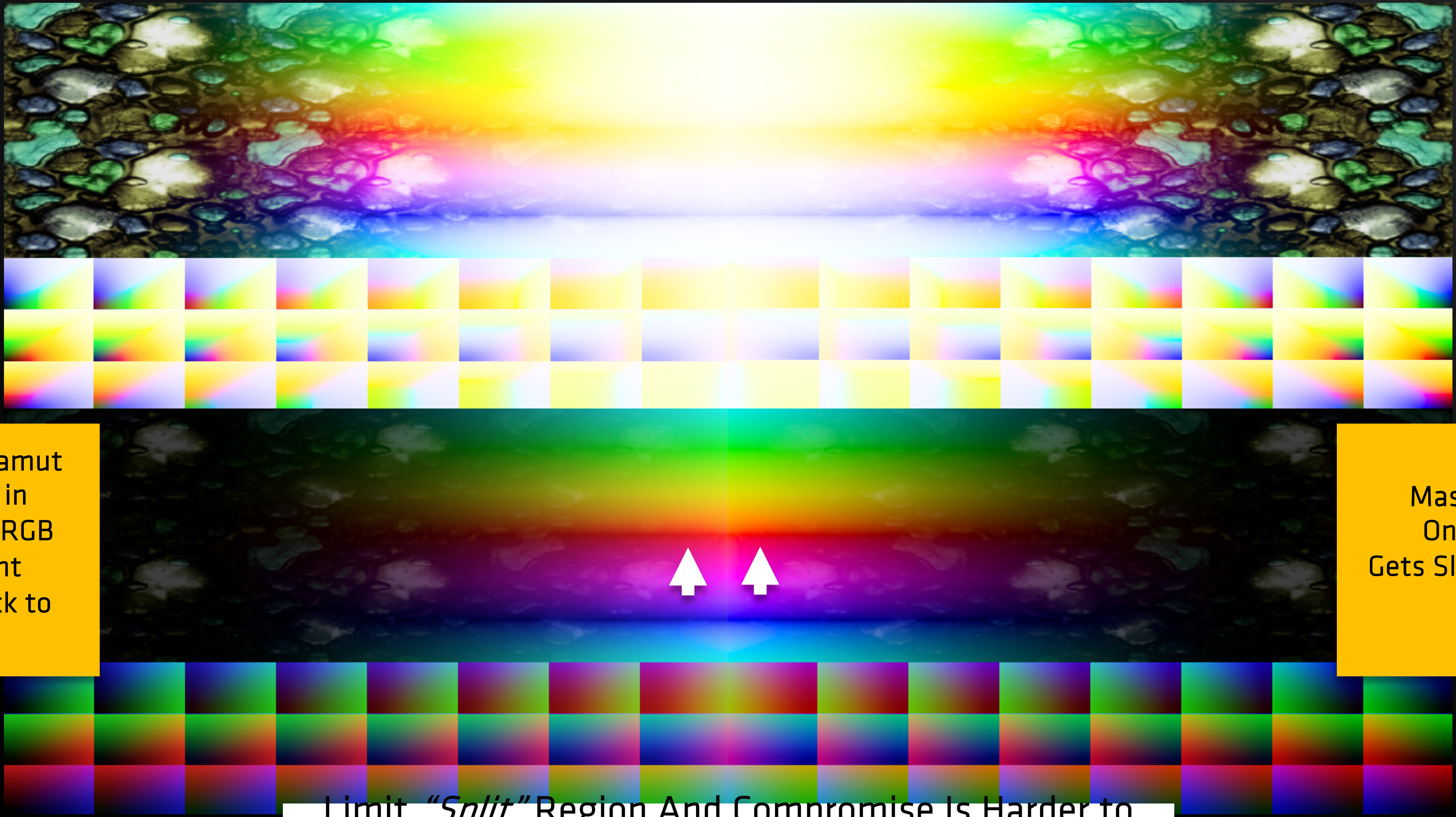


Clipping (No Soft Fall-off)



Soft Fall-off “Split” = 1/32

SOFT FALL-OFF MAPPING – SATURATION COMPROMISE?



Supporting Wide Gamut Content Results in Desaturation For sRGB Range of Content When Mapped Back to sRGB Display

Mastering in sRGB On sRGB Display Gets Slightly Better Peak Saturation

Limit *"Split"* Region And Compromise Is Harder to See

WORKING SPACE GAMUT OPTIONS

- sRGB primaries (good)
  - Wide-gamut displays can cover the full sRGB gamut
- DCI-P3 primaries (good if have wide-gamut content)
  - Shares a blue primary with Rec709 and sRGB
    - Primaries are closer to actual PC HDR hardware than Rec.2020
  - Slight desaturation of LDR range data when mapping back to LDR
- Rec.2020 primaries
  - Primaries are quite different from real display primaries
  - Also slight desaturation of LDR range data when mapping back to LDR

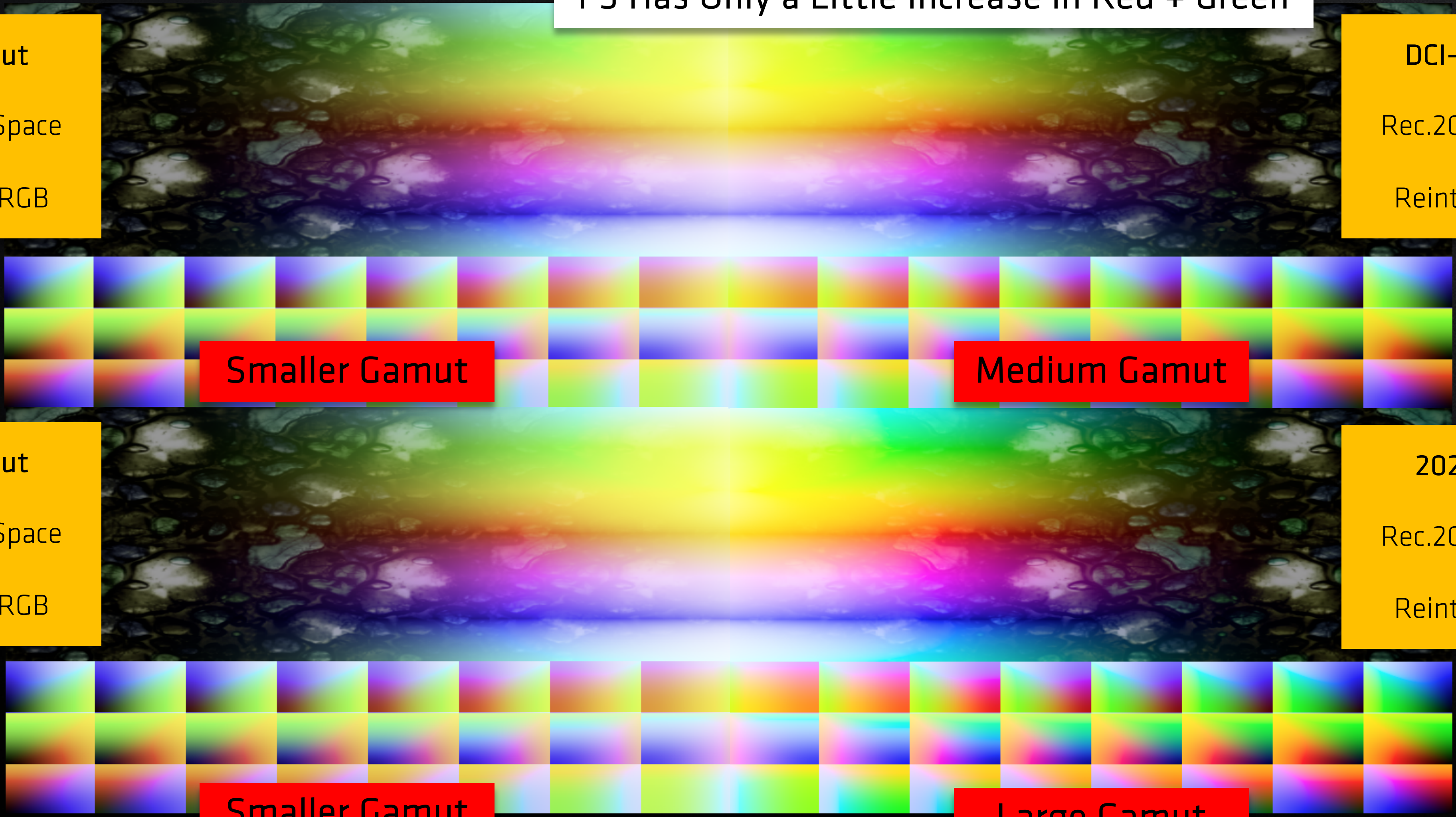
AMD RYZEN RADEON

GAMUT SIZE – VISUALIZED ON SRGB PROJECTOR

P3 Has Only a Little Increase in Red + Green

sRGB Gamut Input
Output In
Rec.2020 Working Space
Then
Reinterpreted as sRGB

DCI-P3 Gamut Input
Output In
Rec.2020 Working Space
Then
Reinterpreted as sRGB



sRGB Gamut Input
Output In
Rec.2020 Working Space
Then
Reinterpreted as sRGB

2020 Gamut Input
Output In
Rec.2020 Working Space
Then
Reinterpreted as sRGB

SWITCHING FROM ALGORITHM TO OPTIMIZATION

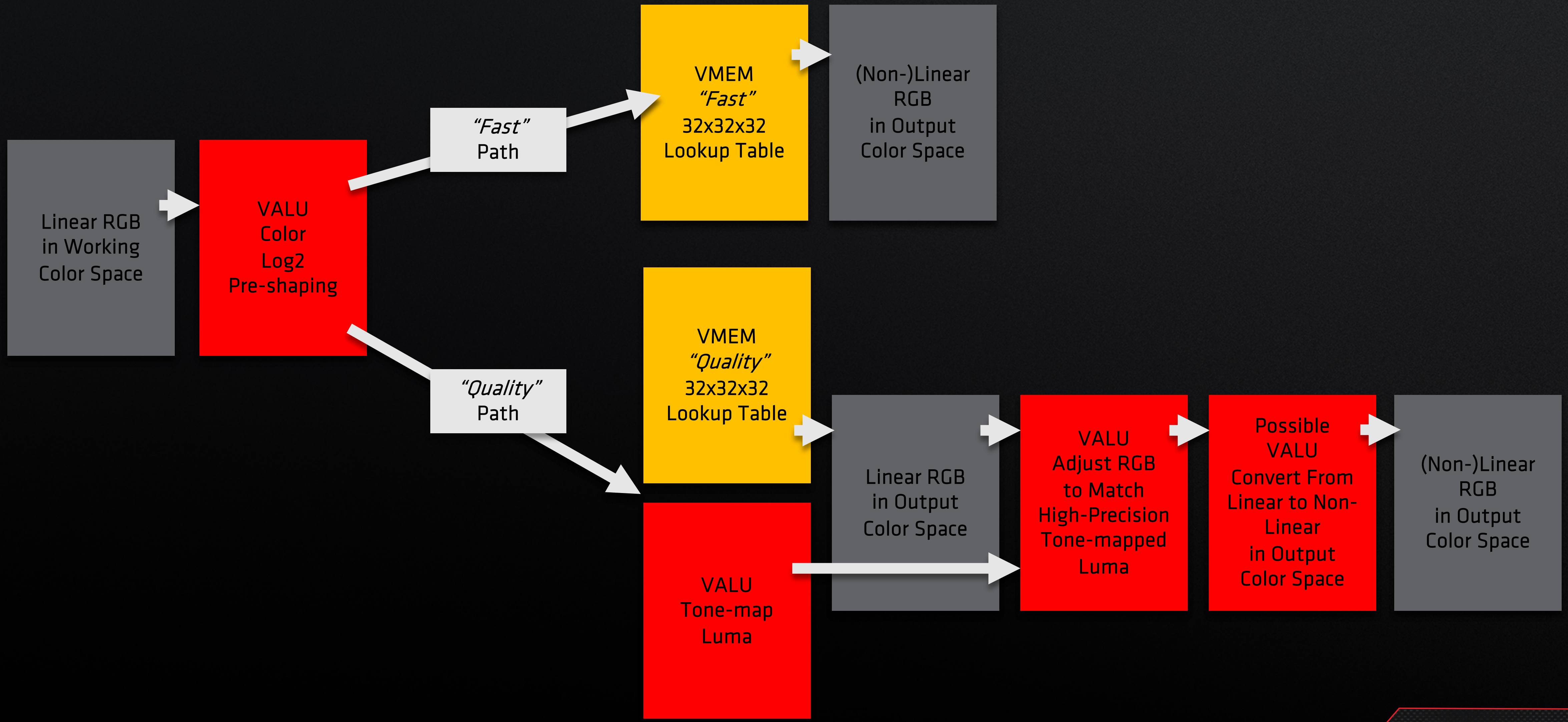
- The majority of the algorithm gets factored out into a LUT
  - What remains is to provide options for

- Precision – Higher Accuracy (aka *"Quality"*)

- Performance – Lower Runtime (aka *"Fast"*)

AMD RYZEN RADEON

OPTIMIZED PIPELINE TWO PATHS



LUT RECOMMENDATIONS

- Maintain typical standard 32x32x32 3D texture
  - Easy to integrate into existing engines
  - Easy to apply existing color grading 3D textures

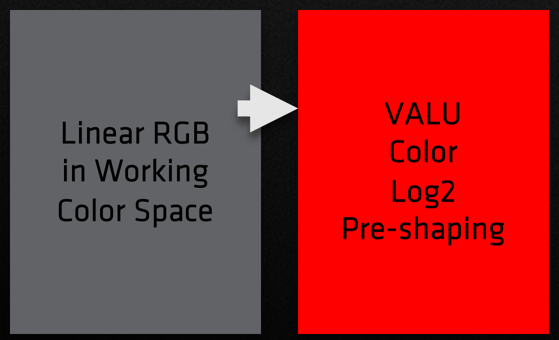
- Formats
  - Use at minimum 10:10:10:2 unorm for non-linear *"Fast"* outputs
  - Use a float based format for linear *"Fast"* or *"Quality"* outputs

COLOR LOG2 PRE-SHAPING BEFORE LUT

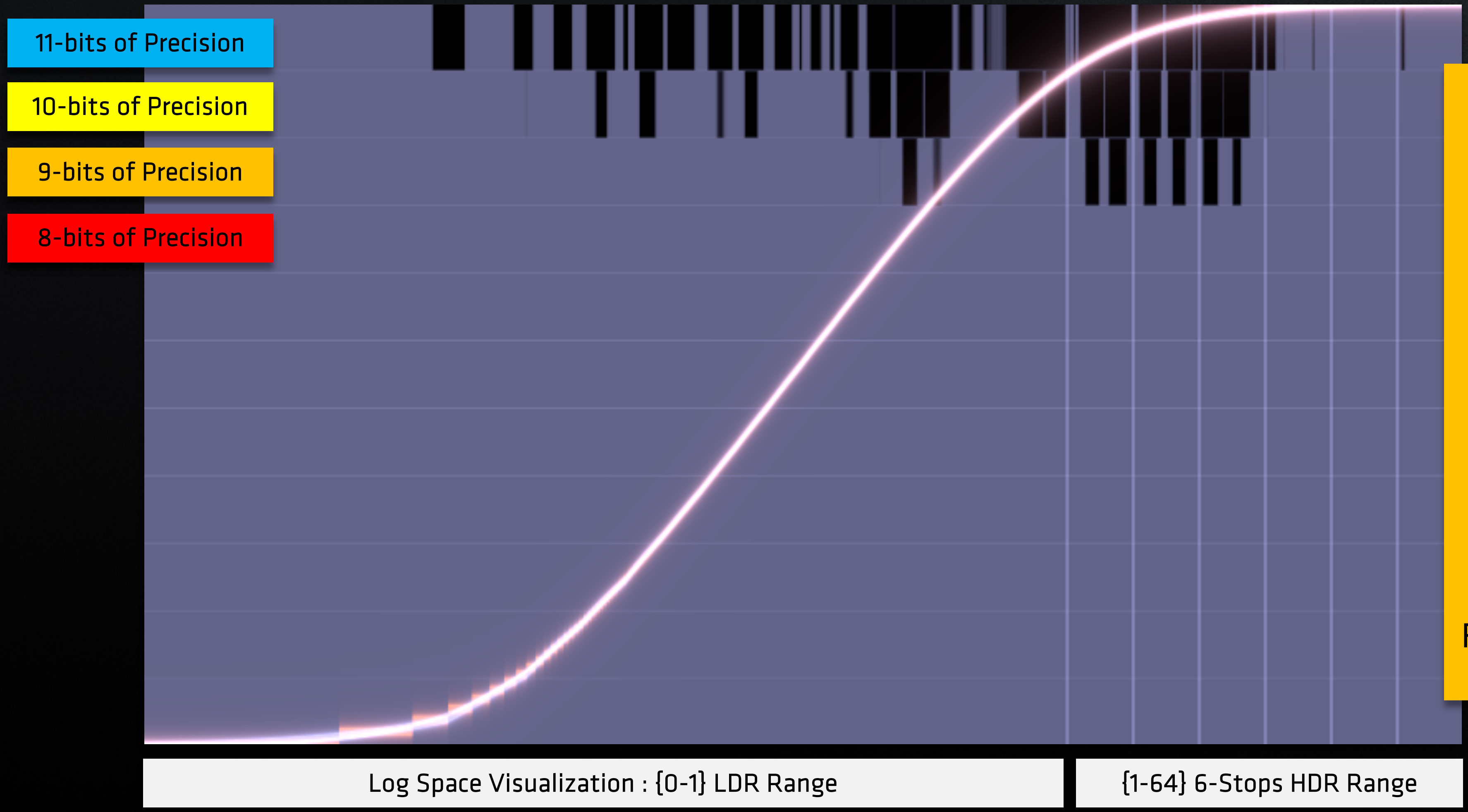
- RGB color input {0 to 1} which maps to {0 to max-HDR}

- Pre-shaping
  - shapedColor = log2(color * scale + 1.0) * (1.0 / log2(scale + 1.0))
  - 3 LOG2, 3 MADs, 3 MUL

- Adapt pre-shaping dynamically
  - Given tone-mapping parameters and output color space
  - Adapt scale value to allocate precision to desired areas



32^3 10:10:10:2-BIT LUT CAN LIMIT OUTPUT PRECISION



Example *"Fast"* Tuned LUT
To sRGB Output Color Space

Unable to Sustain Target of 10-bits of Precision
Across The Full Curve

However
"Good Enough"
For 8-bit/Channel Outputs

THE "QUALITY" PATH FOR INCREASED PRECISION

- One constraint if mixing LUT with color-grading LUT
  - Color grading must preserve luma if using the *"Quality"* path
- Duplicate luma tone-map in VALU
  - ```
    luma = dot(color, colorToLumaWorkingSpace);
    luma = pow(luma, contrast);
    luma = luma / (luma * k0 + k1); // faster version (no shoulder)
    ```
  - 2 EXP2, 2 LOG2, 1 RCP, 3 MAD, 4 MUL
- Re-luma-ize after LUT for increased precision
  - `color *= (luma / dot(color, colorToLumaOutputSpace));`
  - 2 MAD, 5 MUL, 1 RCP
    ```

"QUALITY" LINEAR TO NON-LINEAR TRANSFORM

- When hardware CS stores lack sRGB support

- Recommend a *"branch-free"* linear to sRGB conversion
  - Can be better for the compiler
  - max(min(c*12.92, 0.0031308),1.055*pow(c,0.41666)-0.055);
  - 3 MAX, 3 MIN, 3 EXP2, 3 LOG2, 6 MUL, 3 MAD

COSTS ARE LOW & WILL VARY BY INTEGRATION

- GTM typically added to last CS post-processing pass
  - So for timing below, added GTM to an example up-sampler
  - Running on Radeon™ RX Vega 64 at 2560x1440
  - Timing: {timestamp A, dispatch 16 times (pipelined), timestamp B}
  - Timing is average run-time: (B-A)/16
  - Expect some amount of run-time to be hidden by the up-sampler
- Timing
  - 0.16 ms/frame – Up-sampler alone
  - 0.19 ms/frame – Up-sampler + GTM *"Fast"* (+0.03 ms/frame)
  - 0.20 ms/frame – Up-sampler + GTM *"Quality"* (+0.04 ms/frame)

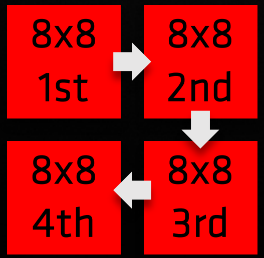
AMD RYZEN RADEON

POST AND SEMI-PERSISTENT WAVES (AKA UNROLLING)

- GTM represents the last part of the post-processing chain
- Recommend trying Semi-Persistent Waves for post
  - Launch a {64,1,1} wave-sized workgroup, then Remap8x8()
  - uint2 Remap8x8(x){return uint2(BFE(x,1,3),BFI(BFE(x,3,3),x,1));
  - Unroll across four 8x8 tiles for block of 16x16 texels

```
// Remap {64,1,1} workgroup to 8x8 setup for 4x unroll of 16x16
uint2 gxy = Remap8x8(gl_LocalInvocationID.x);
gxy += uint2(gl_WorkGroupID.x<<4u, gl_WorkGroupID.y<<4u);

// Simple unroll
float4 c;
Post(c,gxy, . . .); imageStore(img[0], int2(gxy), c); gxy.x += 8u;
Post(c,gxy, . . .); imageStore(img[0], int2(gxy), c); gxy.y += 8u;
Post(c,gxy, . . .); imageStore(img[0], int2(gxy), c); gxy.x -= 8u;
Post(c,gxy, . . .); imageStore(img[0], int2(gxy), c);
```



00	02	04	06	08	0a	0c	0e
01	03	05	07	09	0b	0d	0f
10	12	14	16	18	1a	1c	1e
11	13	15	17	19	1b	1d	1f
20	22	24	26	28	2a	2c	2e
21	23	25	27	29	2b	2d	2f
30	32	34	36	38	3a	3c	3e
31	33	35	37	39	3b	3d	3f

GTM AND AMD FREESYNC™ 2

- Look for related Vulkan® and DirectX® posts on GPUOpen
  - https://gpuopen.com/games-cgi/

- AMD FreeSync 2 enables full control of color mapping
  - Provides ability to query display characteristics
  - Provides a local diming toggle
  - Provides a raw 10-bit output

  - Enables the content author to display content as mastered!

  - GTM is a great option for mapping to AMD FreeSync 2 displays

OUT 3C8H,AL

- Special thanks to
  - Jordan Logan for co-authoring this talk
  - Meith Jhaveri & Ihor Szlachtycz for FreeSync 2 integration guide
  - AMD driver teams for HDR support
  - AMD display team for making FreeSync 2 happen
  - And all the many people providing the inspiration which drives us!

- Post-talk follow-up
  - Jordan.Logan@amd.com
  - Timothy.Lottes@amd.com

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

©2019 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, FreeSync, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

RYZEN RADEON