Lecture 6:

Texturing Part II: Texture Compression and GPU Latency Hiding Mechanisms

Visual Computing Systems
CMU 15-869, Fall 2014

Review: mechanisms to reduce aliasing in the graphics pipeline

- **When sampling visibility?**
  - **Supersampling: sample coverage signal densely (multiple times per pixel)**

- **When sampling shading? (appearance)**
  - **Prefiltering: remove high frequencies from texture signal prior to shading**

Review: operations in a texture fetch

For each texture fetch in a shader program:

1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment
2. Compute mip-map level: d (for tri-linear filtering)
3. Convert normalized texture coordinate uv to texel coordinates: tu,tv
4. Compute required texels **
5. If texture data in filter footprint (eight texels for trilinear filtering) is not in cache:
   - Load compressed texel data from DRAM
   - Decompress texel data
6. Perform tri-linear interpolation according to (tu,tv,d) to get filtered texture sample

A texture fetch involves both data access and also significant amounts of computation: all modern GPUs have dedicated fixed-function hardware support for texture sampling and texture decompression.

** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

Texture data access characteristics

- **Key metric: unique texel-to-fragment ratio**
  - Number of unique texels accessed when rendering a scene divided by number of fragments processed [see Igeny reading for stats: often less than < 1]
  - What is the worst-case ratio? (assuming trilinear filtering)
  - How can incorrect computation of texture miplevel (*d*) affect this?

- **In practice, caching behavior is good, but not CPU workload good**
  - [Montrym & Moreton 95] design for 90% hits
  - Why? (only so much spatial locality)

- **Implications**
  - GPU must provide high memory bandwidth for texture data access
  - GPU must have solution for hiding memory access latency
  - GPU must reduce its bandwidth requirements using caching and texture compression

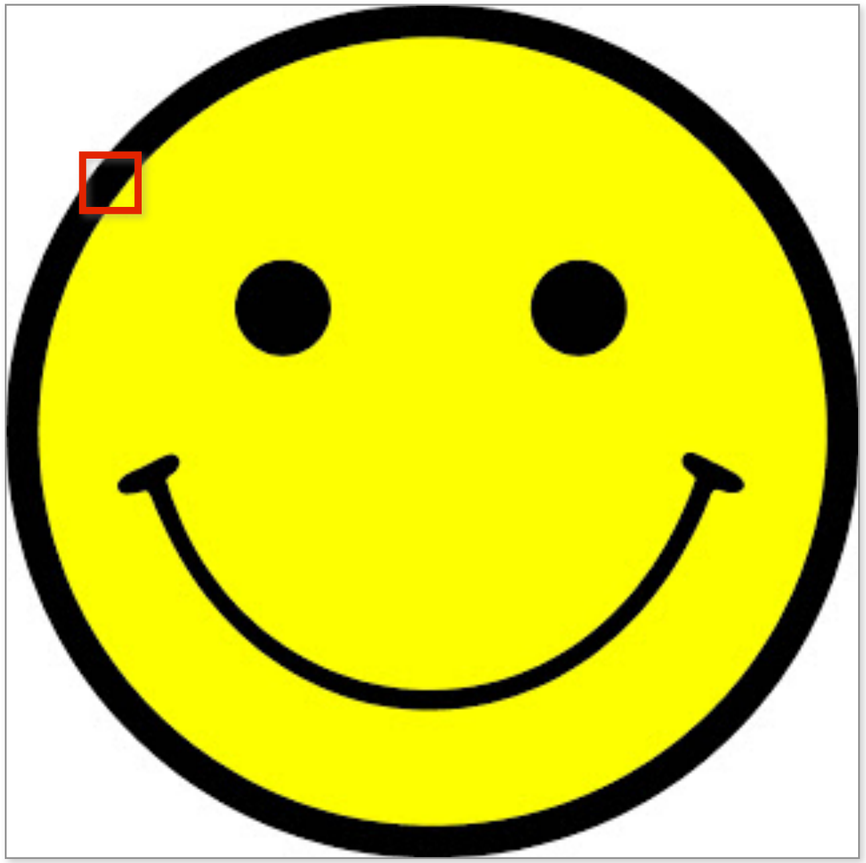
Texture compression

Texture compression

- Goal: reduce bandwidth requirements of texture access
- Texture is read-only data
  - Compression can be performed off-line, so compression algorithms can take significantly longer than decompression (decompression must be fast!)
  - Lossy compression schemes are permissible
- Design requirements
  - Support random texel access (constant time access to any texel)
  - High-performance decompression
  - Simple algorithms (low-cost hardware implementation)
  - High compression ratio
  - High visual quality (lossy is okay, but cannot lose too much!)

Simple scheme: color palette (indexed color)

- Lossless (if image contains a small number of unique colors)



Color palette (eight colors)

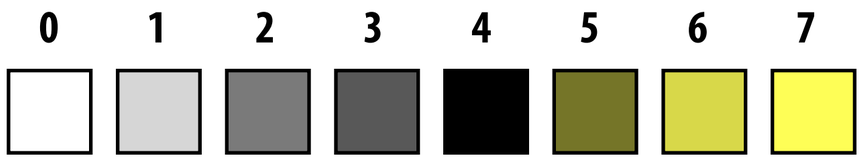
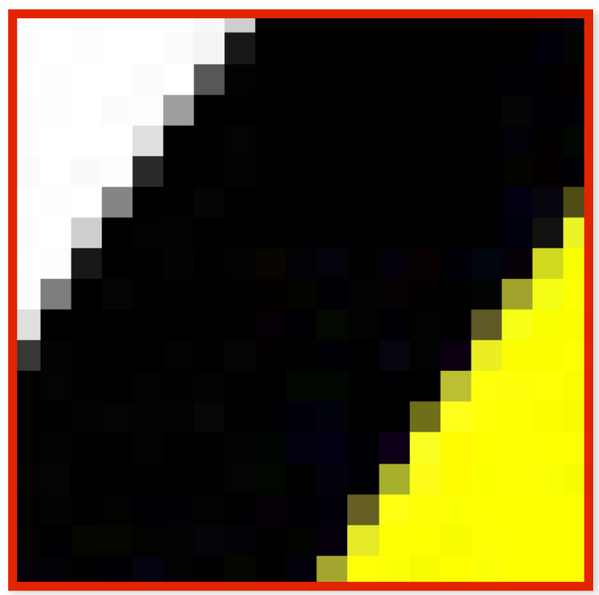


Image encoding in this example:

3 bits per texel + eight RGB values in palette (8x24 bits)

0	1	3	6
0	2	6	7
1	4	6	7
4	5	6	7

What is the compression ratio?



Per-block palette

- Block-based compression scheme on 4x4 texel blocks
  - Idea: there might be many unique colors across an entire image, but can approximate all values in any 4x4 texel region using only a few unique colors

- Per-block palette (e.g., four colors in palette)
  - 12 bytes for palette (assume 24 bits per RGB color: 8-8-8)
  - 2 bits per texel (4 bytes for per-texel indices)
  - 16 bytes (3x compression on original data: 16x3=48 bytes)

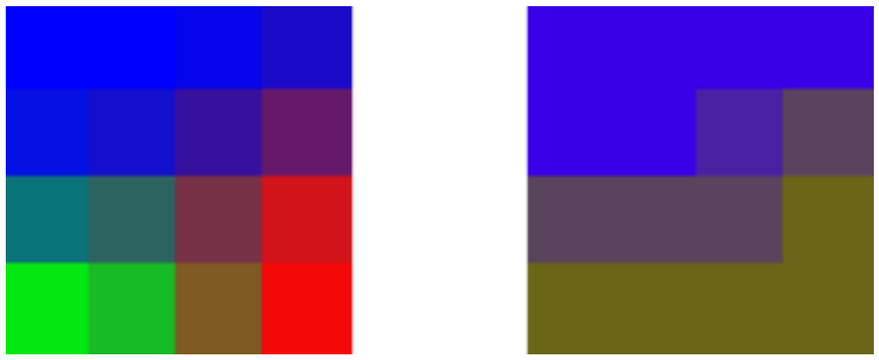
- Can we do better?

S3TC

(Called BC1 or DXTC by Direct3D)

- **Palette of four colors encoded in four bytes:**
  - Two low-precision base colors: C_0 and C_1 (2 bytes each: RGB 5-6-5 format)
  - Other two colors computed from base values
    - $^1/_3C_0 + {}^2/_3C_1$
    - $^2/_3C_0 + {}^1/_3C_1$
- **Total footprint of 4x4 texel block: 8 bytes**
  - 4 bytes for palette, 4 bytes of color ids (16 texels, 2 bits per texel)
  - 4 bpp effective rate, 6:1 compression ratio (fixed ratio: independent of data values)
- **S3TC assumption:**
  - All texels in a 4x4 block lie on a line in RGB color space
- **Additional mode:**
  - If C0 < C1, then third color is $^1/_2C_0 + {}^1/_2C_1$ and fourth color is transparent black

S3TC artifacts



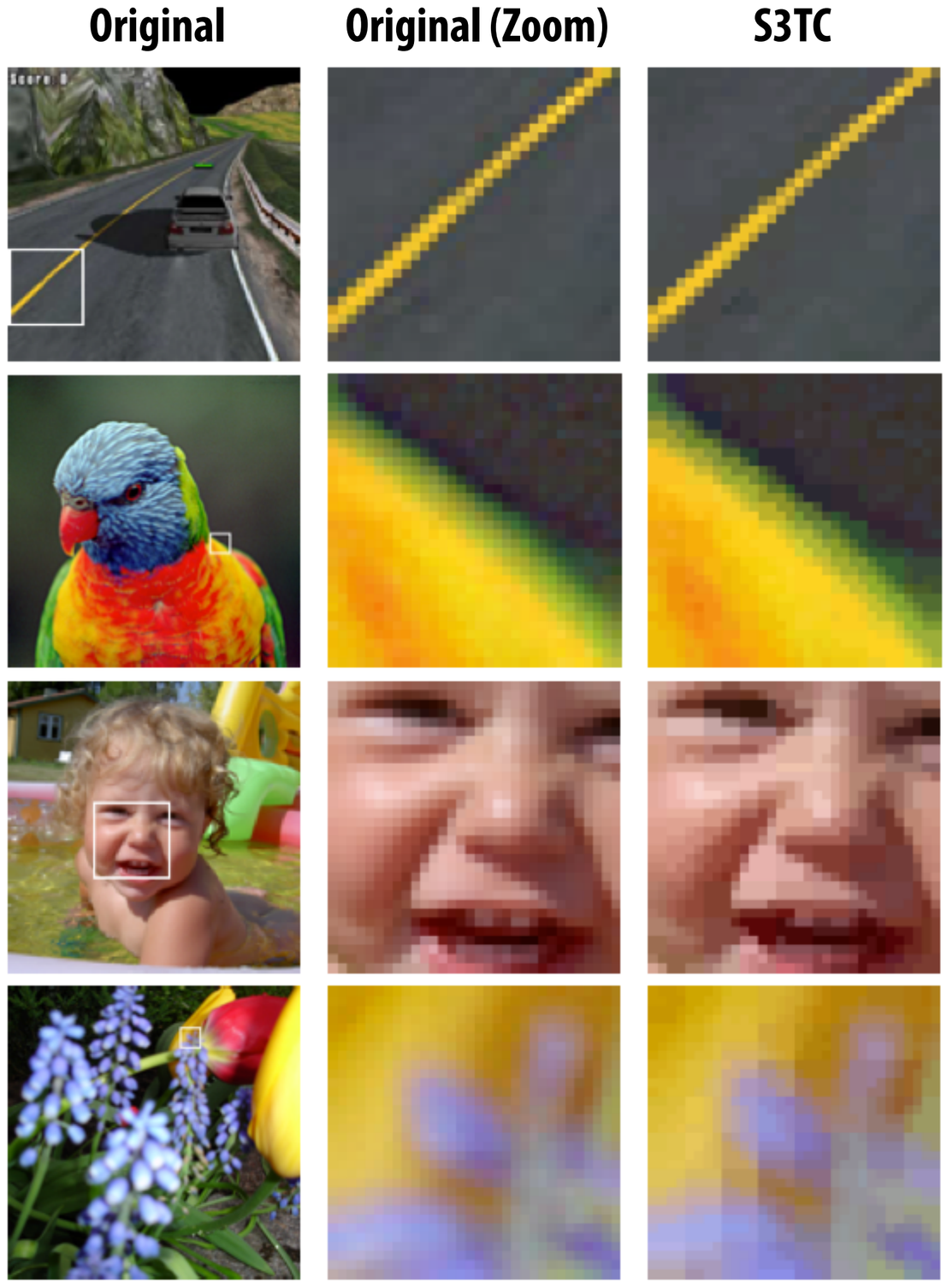
Original data

Compressed result

Cannot interpolate red and blue to get green (here compressor chose blue and yellow as base colors to minimize overall error)

But scheme works well in practice on "real-world" images. (see images at right)

Image credit:
http://renderingpipeline.com/2012/07/texture-compression/



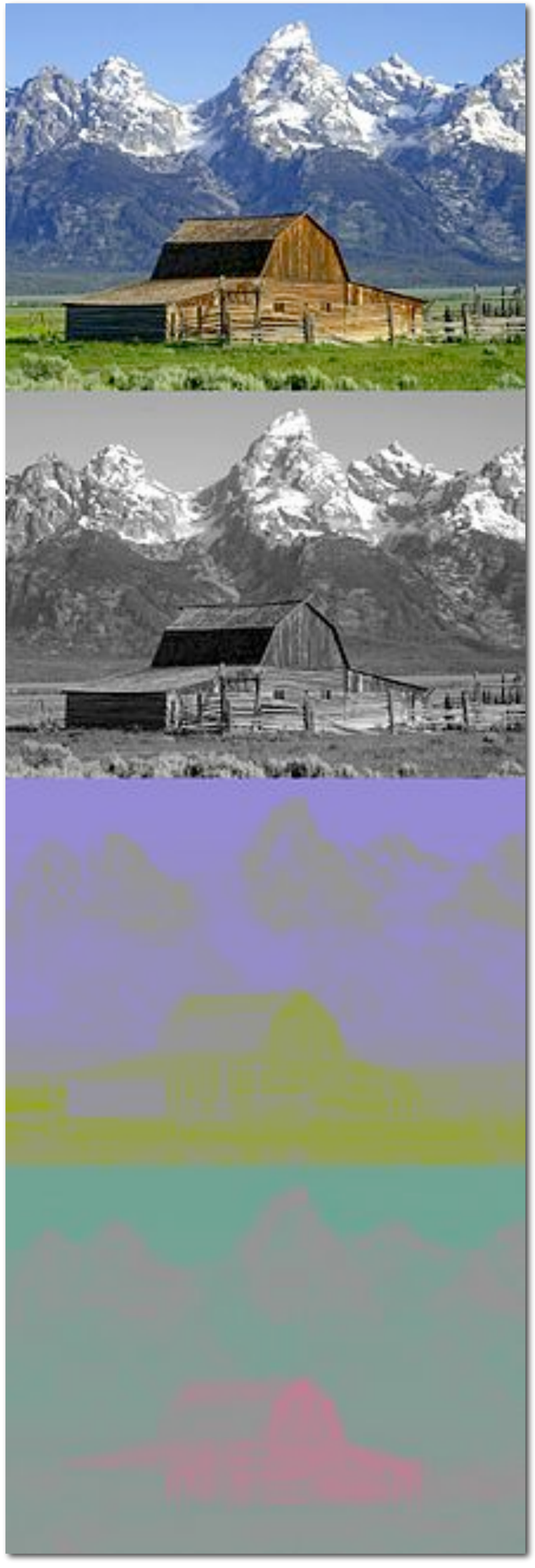
[Strom et al. 2007]

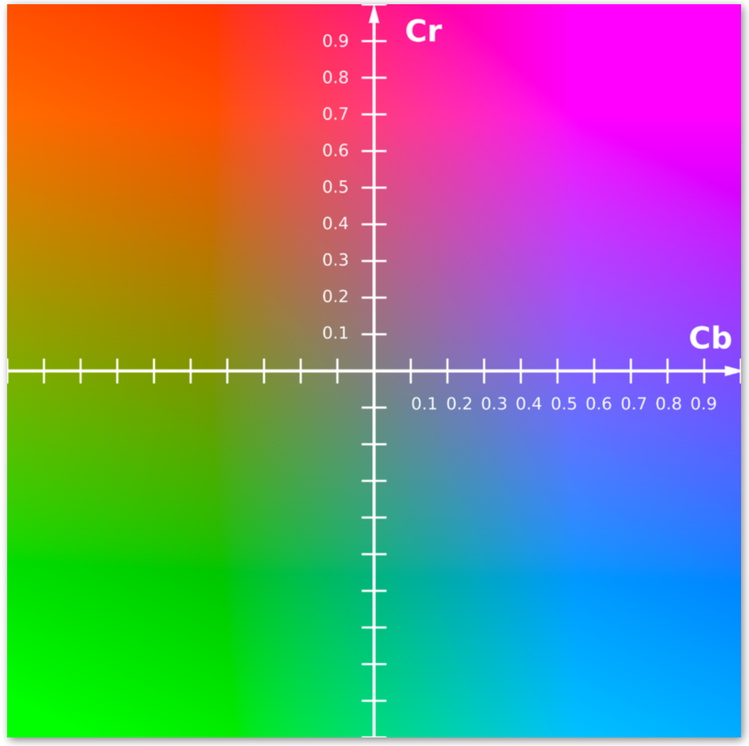
Y′CbCr color space

Y′ = luma: perceived (gamma corrected) luminance

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray





Conversion from R′G′B′ to Y′CbCr:

$$
\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}
\end{aligned}
$$

Image credit: Wikipedia

Demo

Original picture of Kayvon

Demo

Color channels downsampled by a factor of 20 in each dimension (400x reduction in number of samples)

Demo

Full resolution sampling of luminance

Demo

Reconstructed result

Chroma subsampling

Y′CbCr is an efficient representation for storage (and transmission) because Y′ can be stored at higher resolution than CbCr without significant loss in perceived visual quality

Y'_{00} Cb_{00} Cr_{00}	Y'_{10}	Y'_{20} Cb_{20} Cr_{20}	Y'_{30}
Y'_{01} Cb_{01} Cr_{01}	Y'_{11}	Y'_{21} Cb_{21} Cr_{21}	Y'_{31}

4:2:2 representation:

Store Y′ at full resolution
Store Cb, Cr at full vertical resolution, but only half horizontal resolution

Y'_{00} Cb_{00} Cr_{00}	Y'_{10}	Y'_{20} Cb_{20} Cr_{20}	Y'_{30}
Y'_{01}	Y'_{11}	Y'_{21}	Y'_{31}

4:2:0 representation:

Store Y′ at full resolution
Store Cb, Cr at half resolution in both dimensions.

PACKMAN

[Strom et al. 2004]

- **Block-based compression on 2x4 texel blocks**
  - Idea: vary luminance per texel, but specify color per block
- **Each block encoded as:**
  - A single base color per block (12 bits: RGB 4-4-4)
  - 4-bit index identifying one of 16 predefined luminance modulation tables
  - Per-texel 2-bit index into luminance modulation table (8x2=16 bits)
  - Total block size = 12 + 4 + 16 = 32 bits (6:1 compression ratio)
- **Decompression:**
  - `texel[i] =  base_color + table[table_id][table_index[i]];`

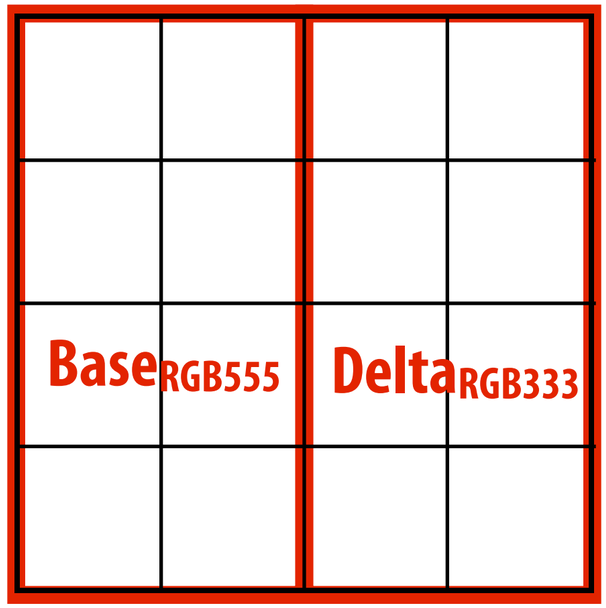
table codeword	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

Example codebook for modulation tables (8 of 16 tables shown)

iPackman (ETC)

[Strom et al. 2005]

- Improves on problems of heavily quantized and sparsely represented chrominance in PACKMAN
  - Higher resolution base color + differential color represents color more accurately
- Operates on 4x4 texel blocks
  - Optionally represent 4x4 block as two eight-texel subblocks with differentials (else use PACKMAN for two subblocks)
    - 1 bit designates whether differential scheme is in use
  - Base color for first block (RGB 5-5-5: 15 bits)
  - Color differential for second block (RGB 3-3-3: 9 bits)
  - 1 bit designating if subblocks are 4x2 or 2x4
  - 3-bit index identifying modulation table per subblock (2x3 bits)
  - Per-texel modulation table index (2x16 bits)
  - Total compressed block size: 1 + 15 + 9 + 1 + 6 + 32 = 64 bits (6:1 ratio)



PACKMAN vs. iPACKMAN quality comparison

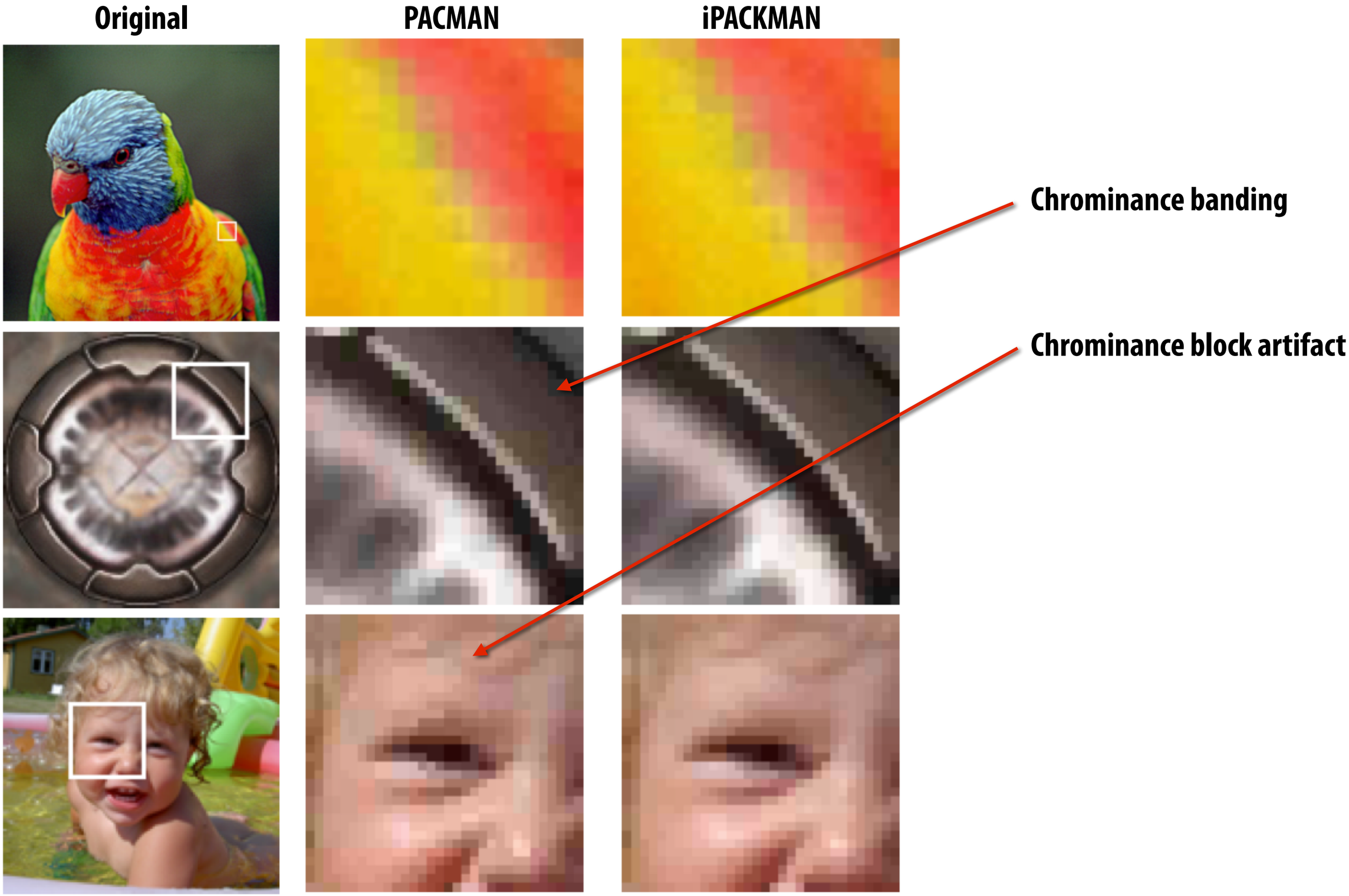
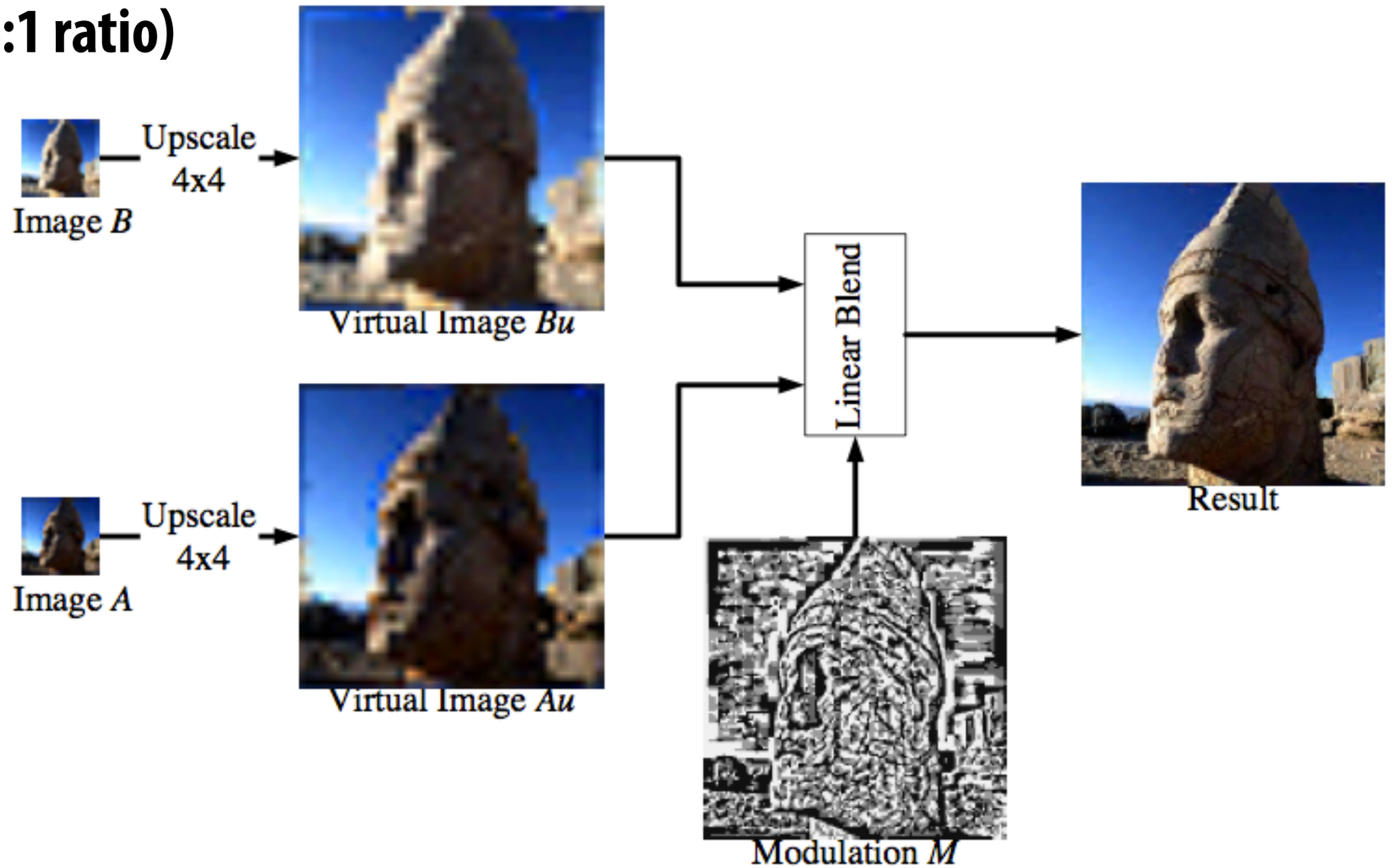


Image credit: Strom et al. 2005

PVRTC (Power VR texture compression)

[Fenney et al. 2003]

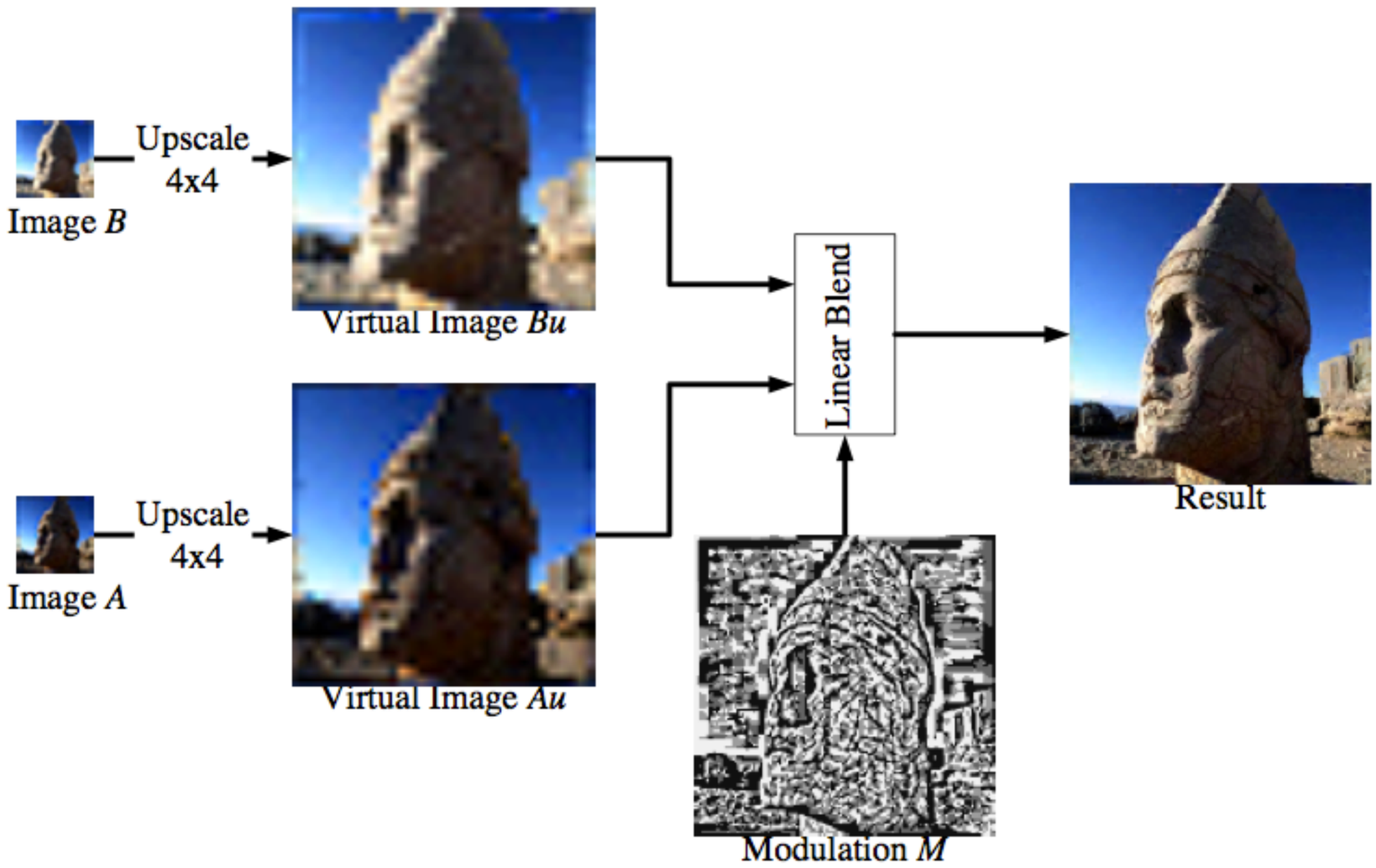
- Not a block-based format
  - Used in Imagination PowerVR GPUs
- Store low-frequency base images A and B
  - Base images downsampled by factor of 4 in each dimension ($^1/_{16}$ fewer texels)
  - Store base image pixels in RGB 5:5:5 format (+ 1 bit alpha)
- Store 2-bit modulation factor per texel
- Total footprint: 4 bpp (6:1 ratio)



PVRTC

[Fenney et al. 2003]

- **Decompression algorithm:**
  - Bilinear interpolate samples from A and B (upsample) to get value at desired texel
  - Interpolate upsampled values according to 2-bit modulation factor

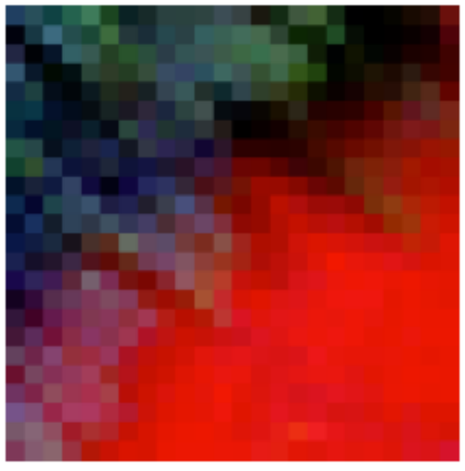


PVRTC avoids blocking artifacts

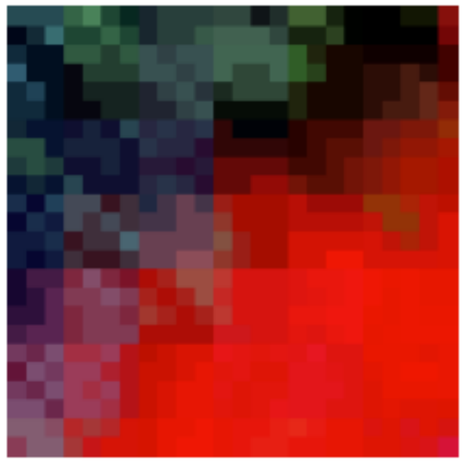
Because it is not block-based!

Recall: decompression algorithm involves bilinear upsampling of low-resolution base images

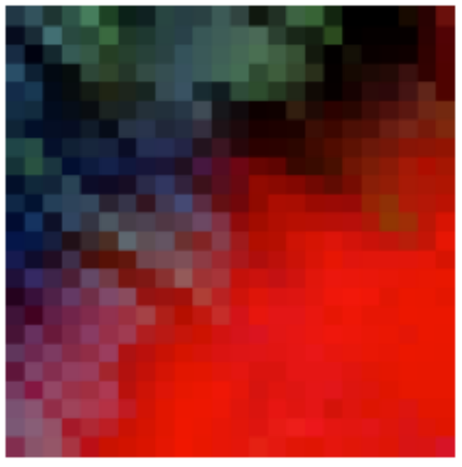
(Followed by a weighted combination of the two images)



Original



S3TC



4bpp PVRTC

Image credit: Fenney et al. 2003

Summary: texture compression

- Many schemes target 6:1 fixed compression ratio (4 bpp)
  - Predictable performance
  - 8 bytes per 4x4-texel block is desirable for memory transfers

- Lossy compression techniques
  - Exploit characteristics of the human visual system to minimize perceived error
  - Texture data is read only, so "drift" due to multiple reads/writes is not a concern

- Block-based vs. not-block based
  - Block-based: S3TC/DXTC/BC1, iPACKMAN/ETC/ETC2, ASTC (not discussed today)
  - Not-block-based: PVRTC

- We only discussed decompression today:
  - Compression can be performed off-line (except when textures are generated at runtime… e.g., reflectance maps)

Hiding the latency of texture sampling and texel data access

Texture sampling is a high-latency operation

For each texture fetch in a shader program:

1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment
2. Compute mip-map level: d (for tri-linear filtering)
3. Convert normalized texture coordinate uv to texel coordinates: (tu,tv)
4. Compute required texel addresses
5. If texture data in filter footprint is not in cache (recall: GPUs miss cache often)
   - Fetch texel data from DRAM
   - Decompress texel data for storage in texture cache
6. Perform tri-linear interpolation according to (tu,tv,d) to get filtered texture sample

Latency of texture fetch involves time to perform math for texel address computation, decompression, and filtering (not just latency of fetching data from memory)

Addressing texture sampling latency

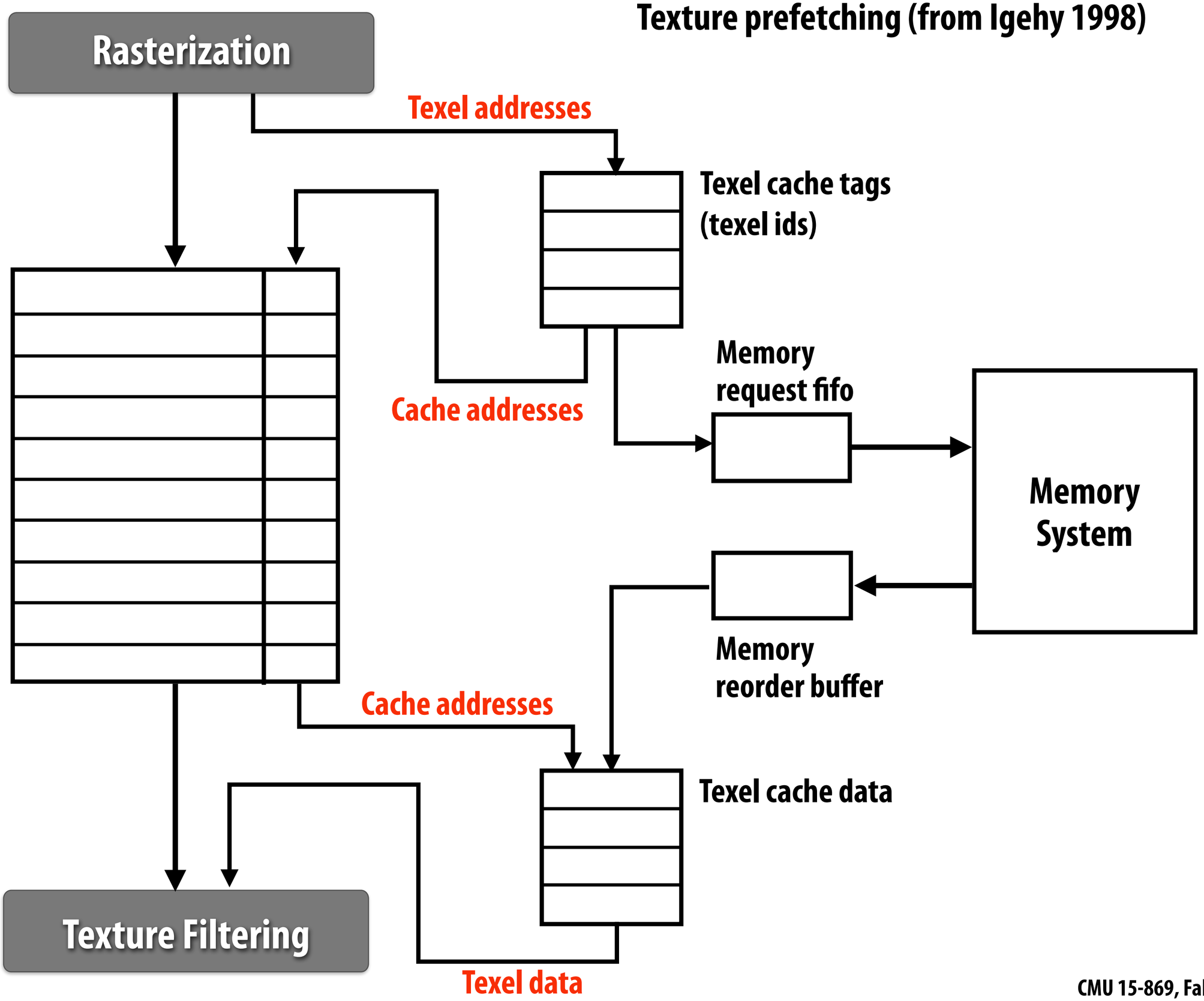
- Processor requests filtered texture data → processor waits hundreds of cycles (significant loss of performance)

- Solution prior to programmable GPU cores: texture data prefetching
  - Today's reading: Igehy et al. *Prefetching in a Texture Cache Architecture*

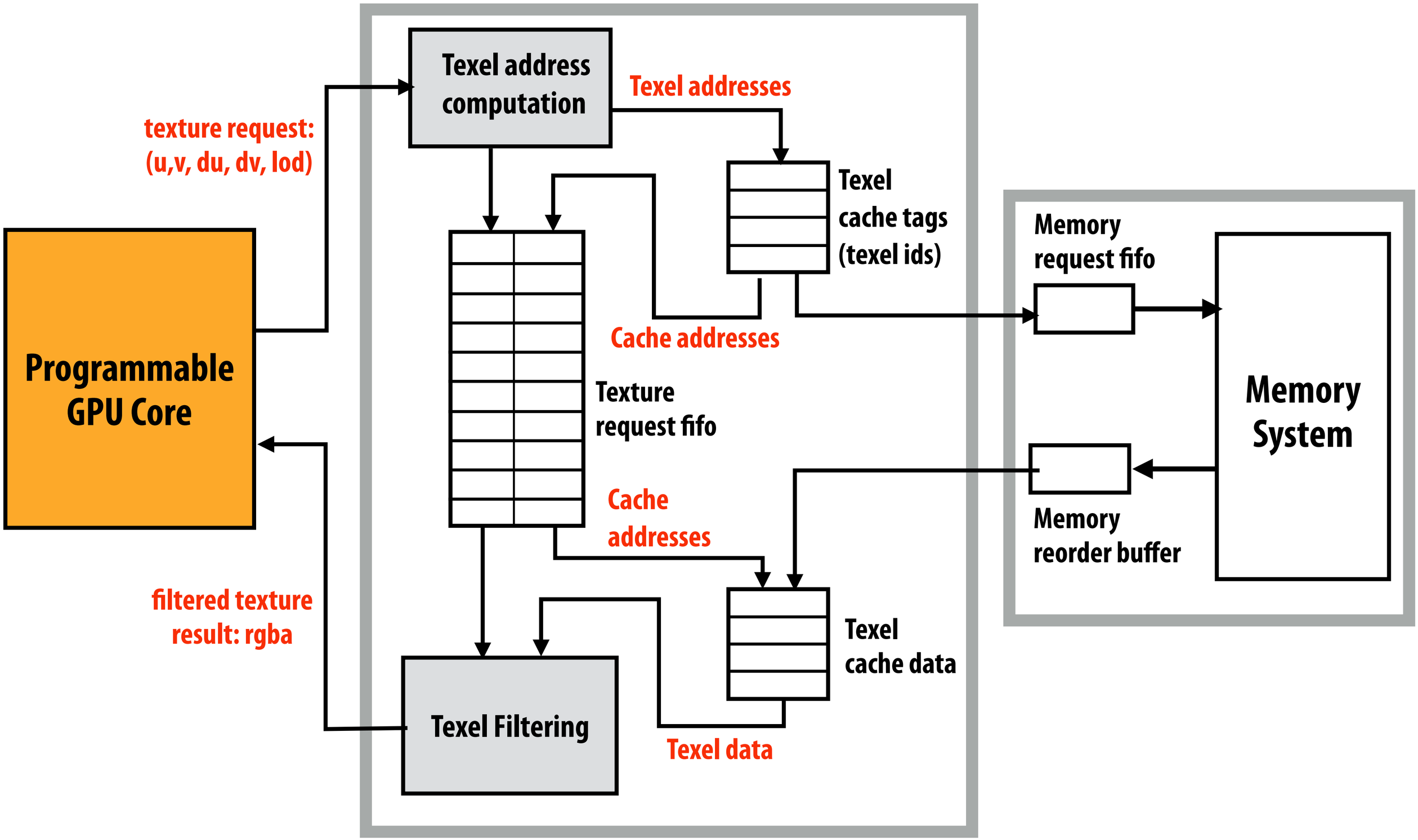
- Solution in all modern GPUs: multi-threaded processor cores
  - Will omit today, but will discuss in detail in a future lecture

Prefetching example: large fragment FIFOs

Texture prefetching (from Igehy 1998)

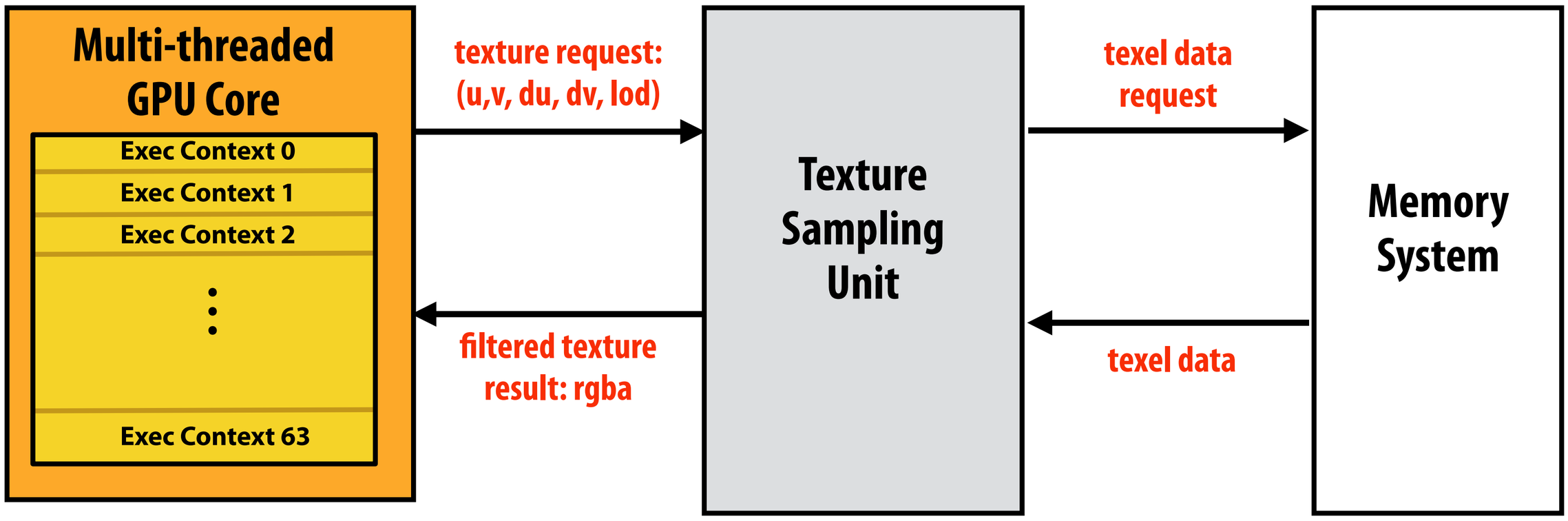


A more modern design



Texture Sampling Unit

Modern GPUs: texture latency is hidden via hardware multi-threading



GPU executes instructions from runnable fragments when other fragments are waiting on texture sampling responses.

Fragment FIFO from Igehy prefetching design is now represented by live fragment state in the programmable core.

Texture system summary

- A texture lookup is a lot more than a 2D array access
  - Significant computational and bandwidth expense
  - Implemented in specialized fixed-function hardware

- Bandwidth reduction mechanism: GPU texture caches
  - Primarily serve to amplify limited DRAM bandwidth, not reduce latency to off-chip memory
  - Small capacity compared to CPU caches, but high BW (need eight texels at once)
  - Tiled rasterization order + tiled texture layout optimizations increase cache hits

- Bandwidth reduction mechanism: texture compression
  - Lossy compression schemes
  - Fixed-compression ratio encodings (e.g, 6:1 ratio, 4 bpp is common for RGB data)
  - Schemes permit random access into compressed representation

- Latency avoidance/hiding mechanisms:
  - Prefetching (in the old days)
  - Multi-threading (in modern GPUs)