

**Lecture 19:**

# **Parallelizing and Optimizing Rasterization**

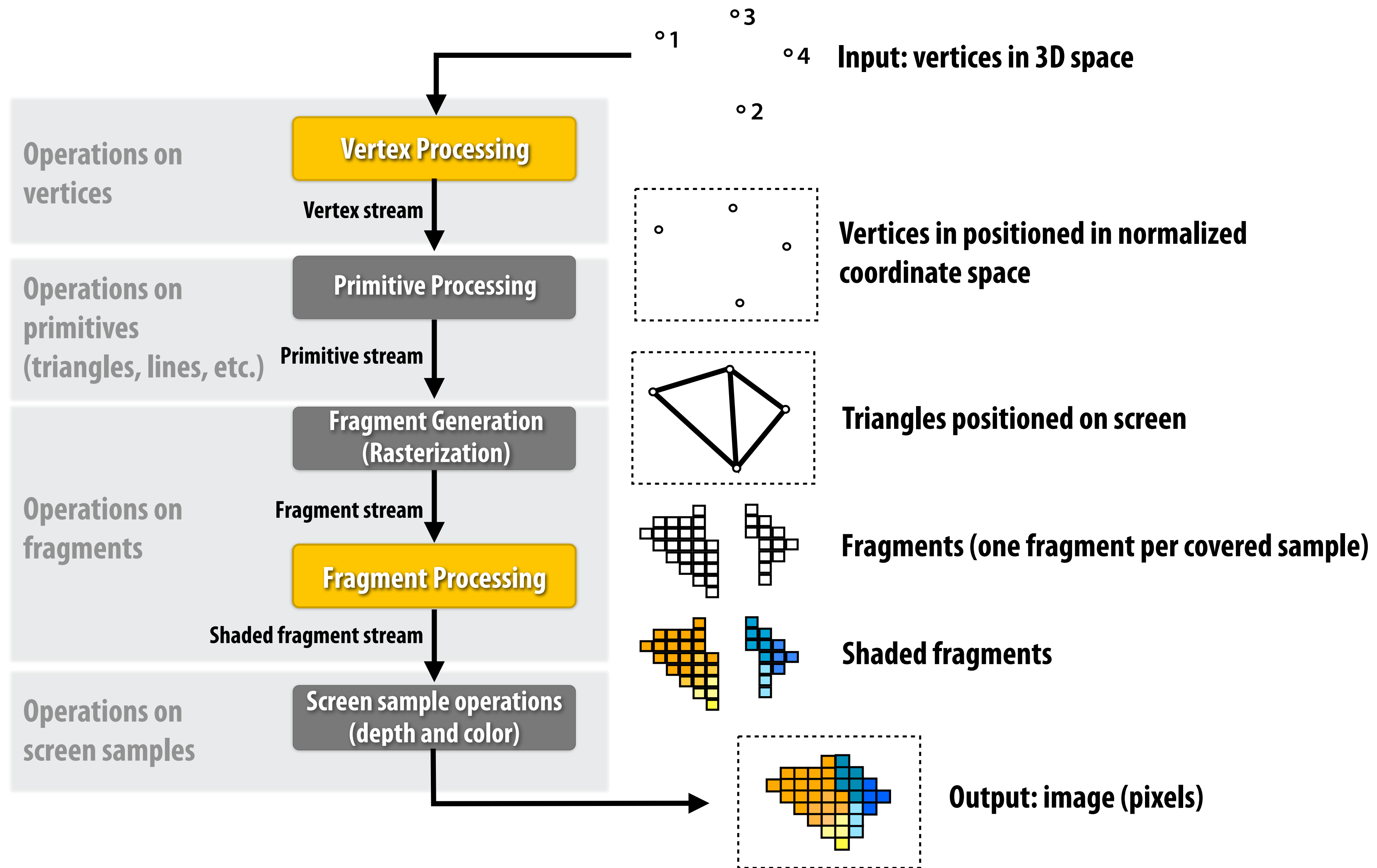
---

**Computer Graphics: Rendering, Geometry, and Image Manipulation**  
**Stanford CS248A, Winter 2023**

# Today's topic

- **A peek under the hood at how modern GPUs implement rasterization-based graphics pipelines (with a focus on mobile GPUs)**
- **Parallelization of work**
- **Saving power by: reducing work, transferring less data**
- **Reducing data transfer using data compression**

# Simple OpenGL/Direct3D graphics pipeline \*



\* Several stages of the modern OpenGL pipeline are omitted











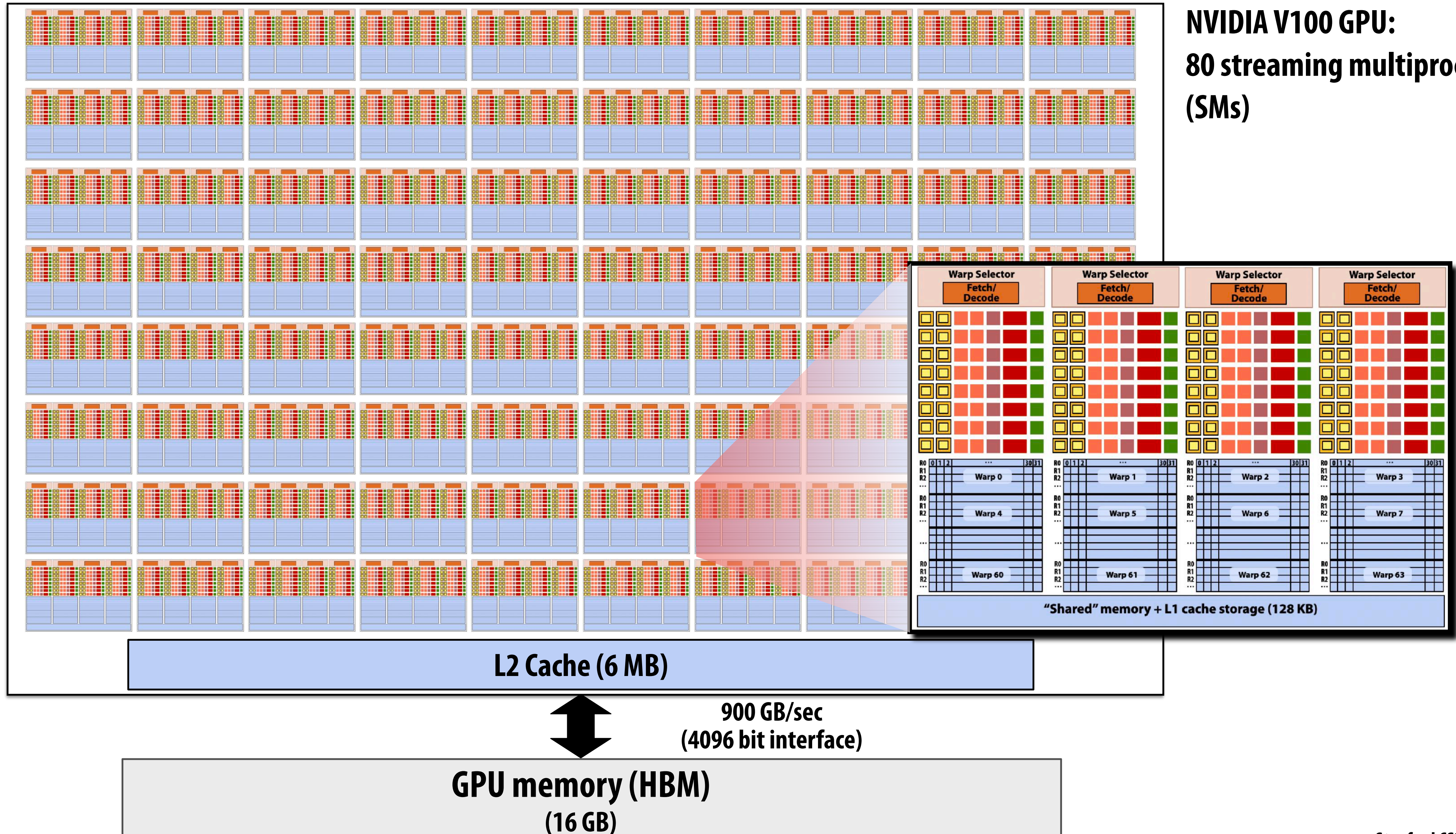
# FORZA 7 MOTORSPORT





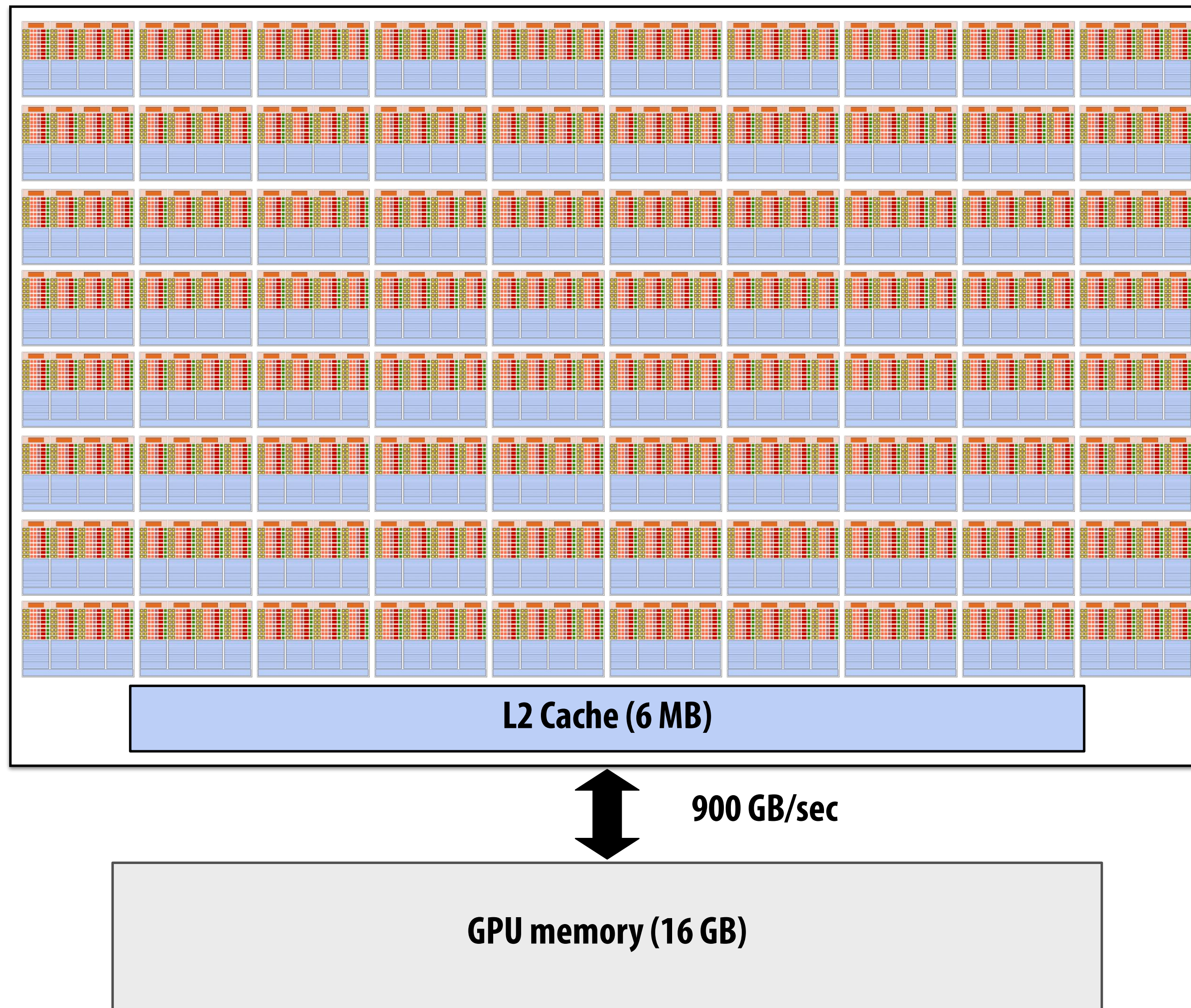
# Lots of parallel processing capability

**NVIDIA V100 GPU:**  
**80 streaming multiprocessors**  
**(SMs)**





# V100 GPU parallelism



**1.245 GHz clock**

**80 SM processor cores per chip**

**64 parallel multiple-add units per SM**

**$80 \times 64 = 5,120$  fp32 mul-add ALUs  
= 12.7 TFLOPs \***

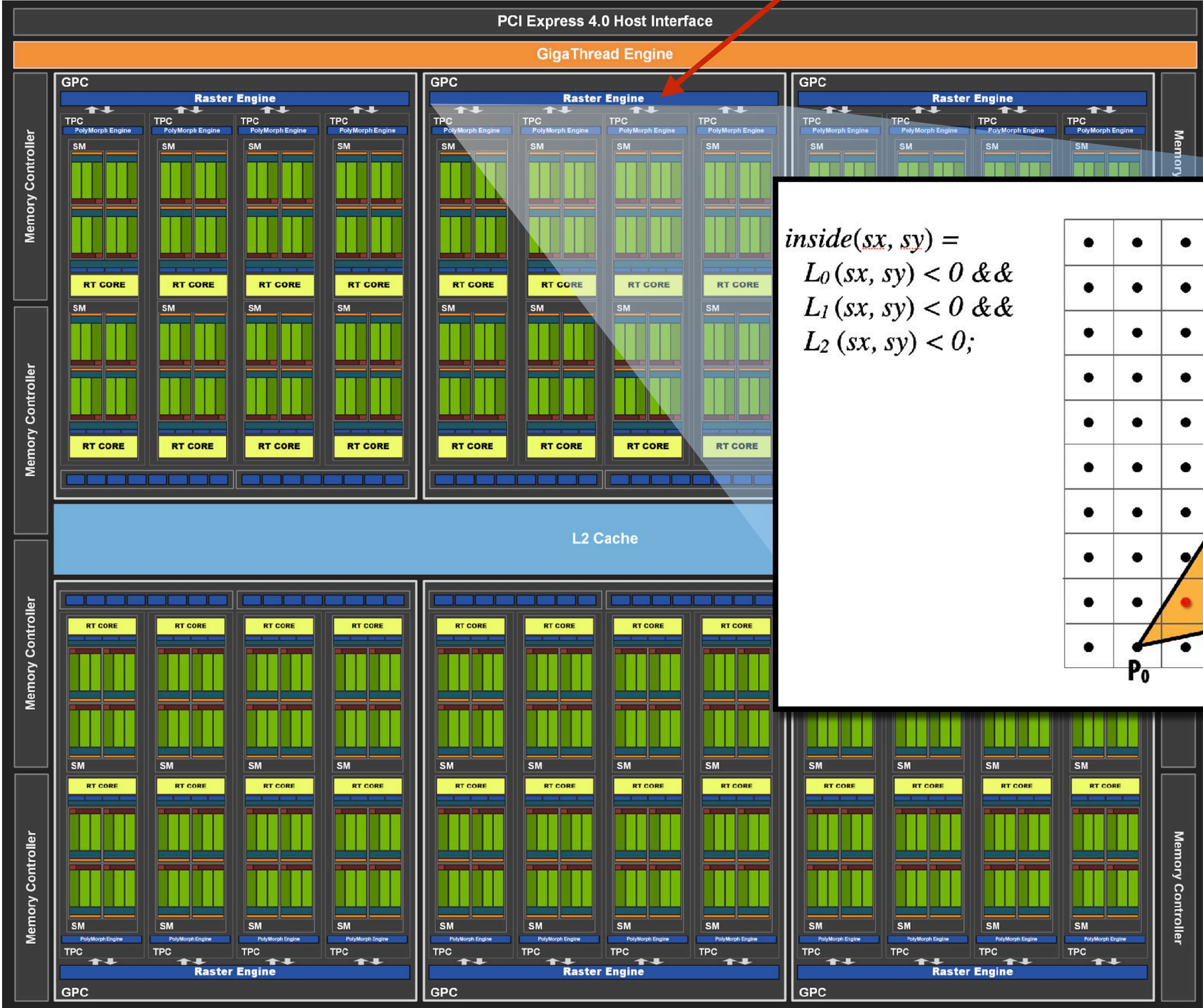
**Up to 163,840 fragments being  
processed at a time on the chip!**

\* mul-add counted as 2 flops:



# RTX 3090 GPU

Hardware units for rasterization

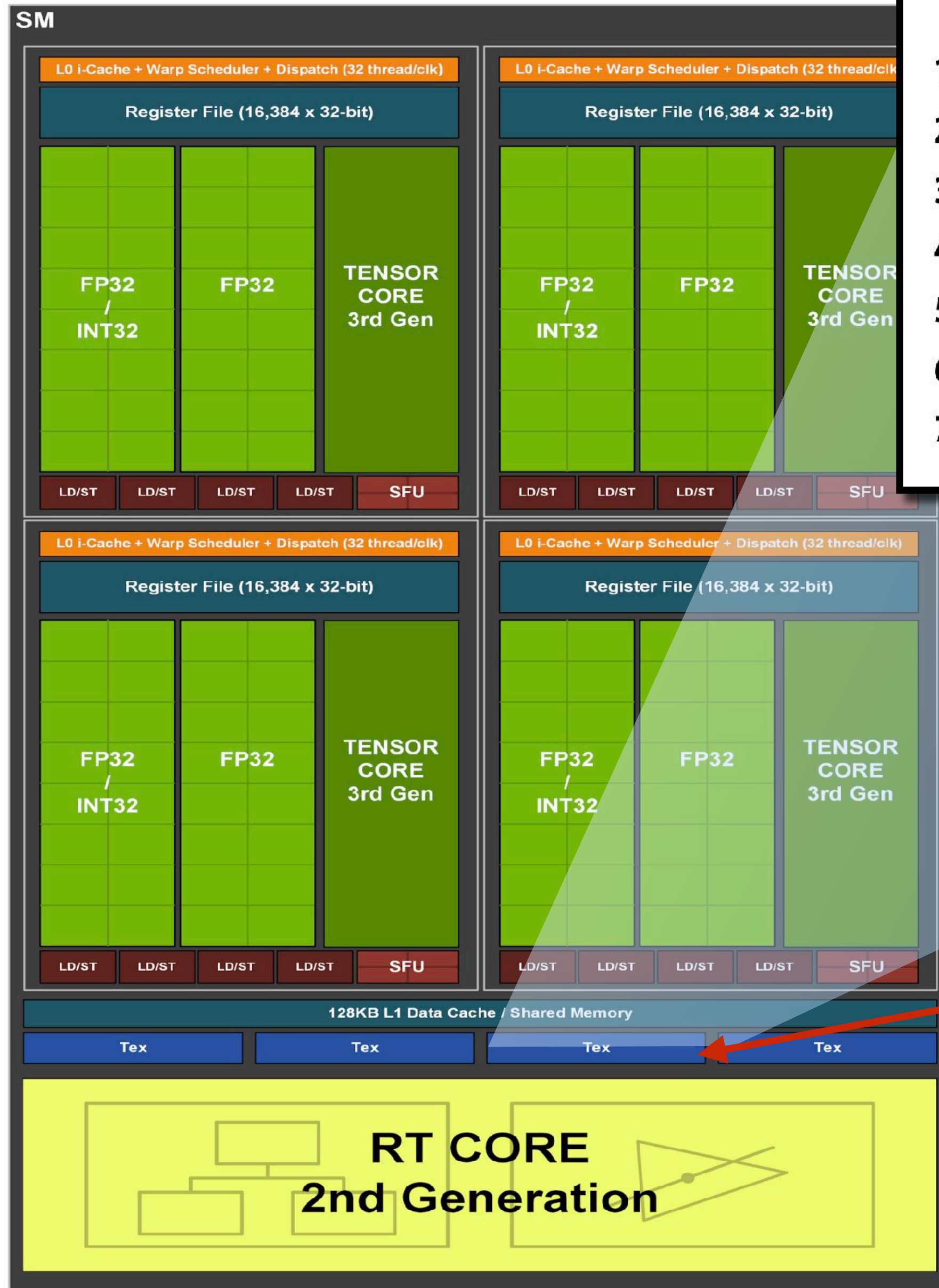


inside( $sx, sy$ ) =  
 $L_0(sx, sy) < 0 \ \&\&$   
 $L_1(sx, sy) < 0 \ \&\&$   
 $L_2(sx, sy) < 0;$

The diagram shows a 10x10 grid of pixels. A triangle is overlaid on the grid, with vertices labeled  $P_0$ ,  $P_1$ , and  $P_2$ . The interior of the triangle is shaded orange and contains red dots, representing the pixels being rasterized. The equation above defines the 'inside' test for a pixel  $(sx, sy)$  based on three linear functions  $L_0$ ,  $L_1$ , and  $L_2$ .

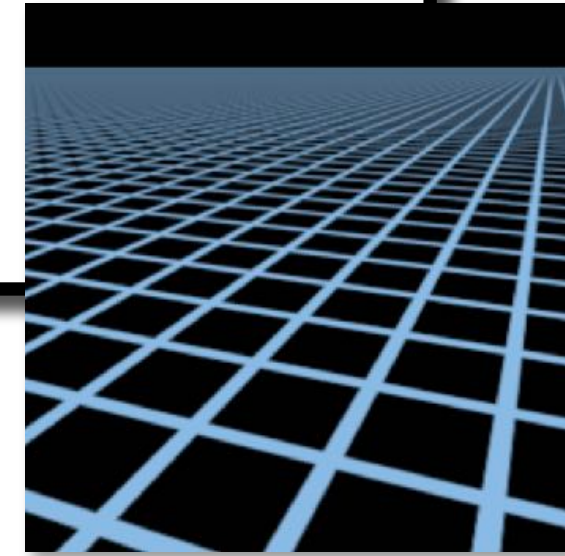


# RTX 3090 GPU



## Texture sampling operation

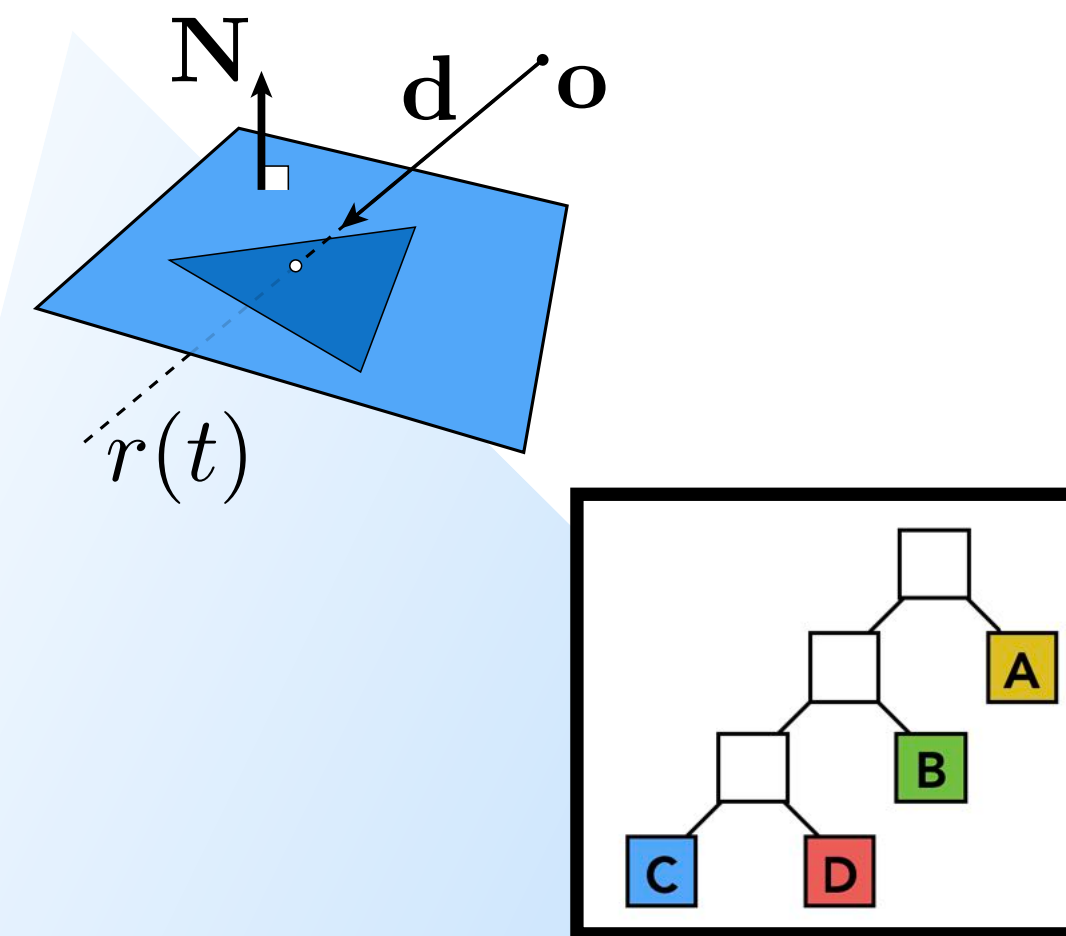
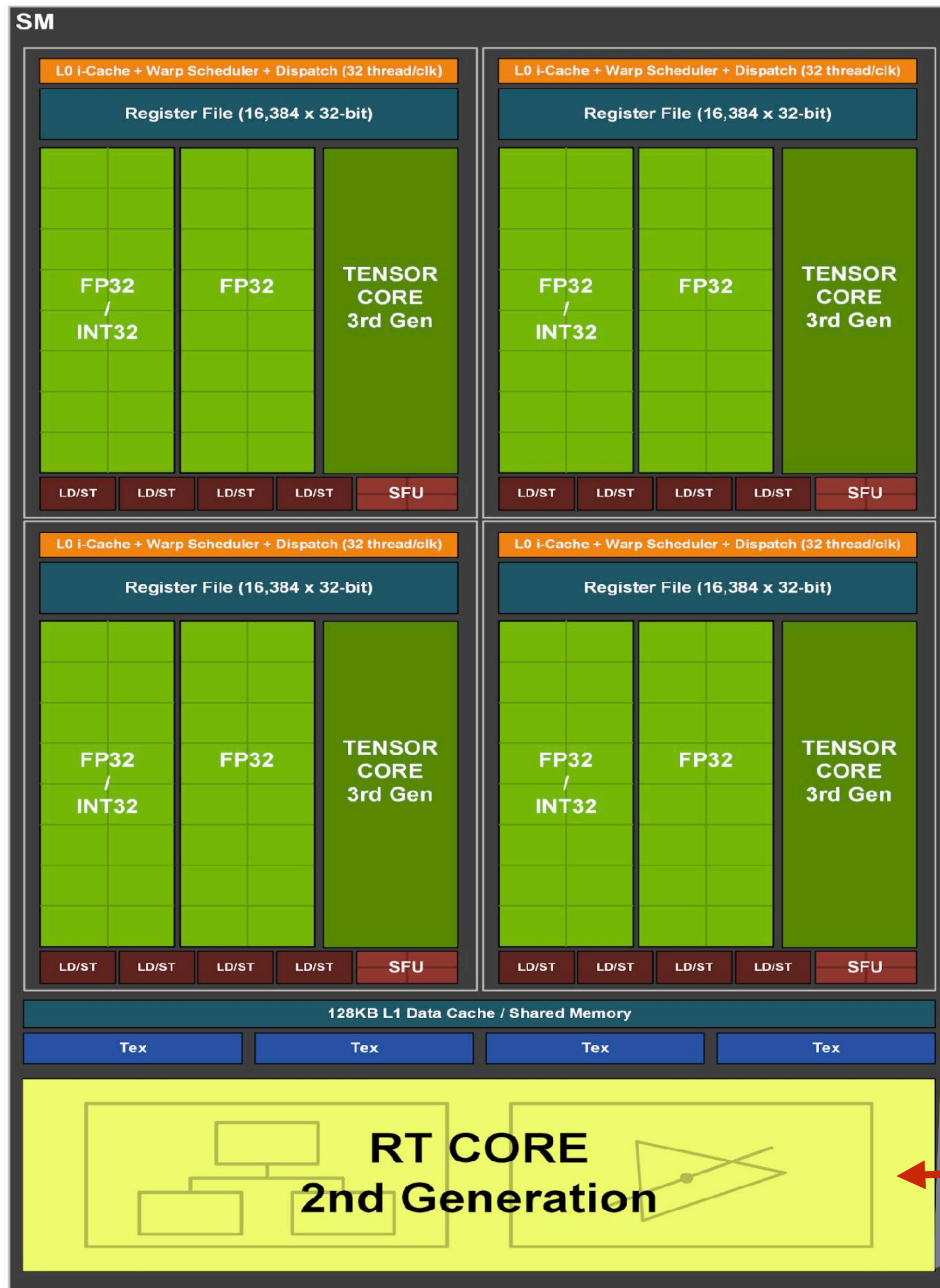
1. Compute  $u$  and  $v$  from screen sample  $x, y$  (via evaluation of attribute equations)
2. Compute  $du/dx, du/dy, dv/dx, dv/dy$  differentials from screen-adjacent samples.
3. Compute mip map level  $d$
4. Convert normalized  $[0,1]$  texture coordinate  $(u,v)$  to texture coordinates  $U,V$  in  $[W,H]$
5. Compute required texels in window of filter
6. Load required texels from memory (need eight texels for trilinear)
7. Perform tri-linear interpolation according to  $(U, V, d)$



Hardware units  
for texture mapping



# RTX 3090 GPU



Hardware units for ray tracing



**For the rest of the lecture, I'm going to focus on mapping  
rasterization workloads to modern mobile GPUs**



**all**

**Q. What is a big concern in ~~mobile~~ computing?**



# A. Power



# Two reasons to save power

Run at *higher performance*  
for a *fixed* amount of time.



Power = heat  
If a chip gets too hot, it must be  
clocked down to cool off

Run at *sufficient performance*  
for a *longer* amount of time.



Power = battery  
Long battery life is a desirable  
feature in mobile devices



# Mobile phone example

Apple iPhone 12

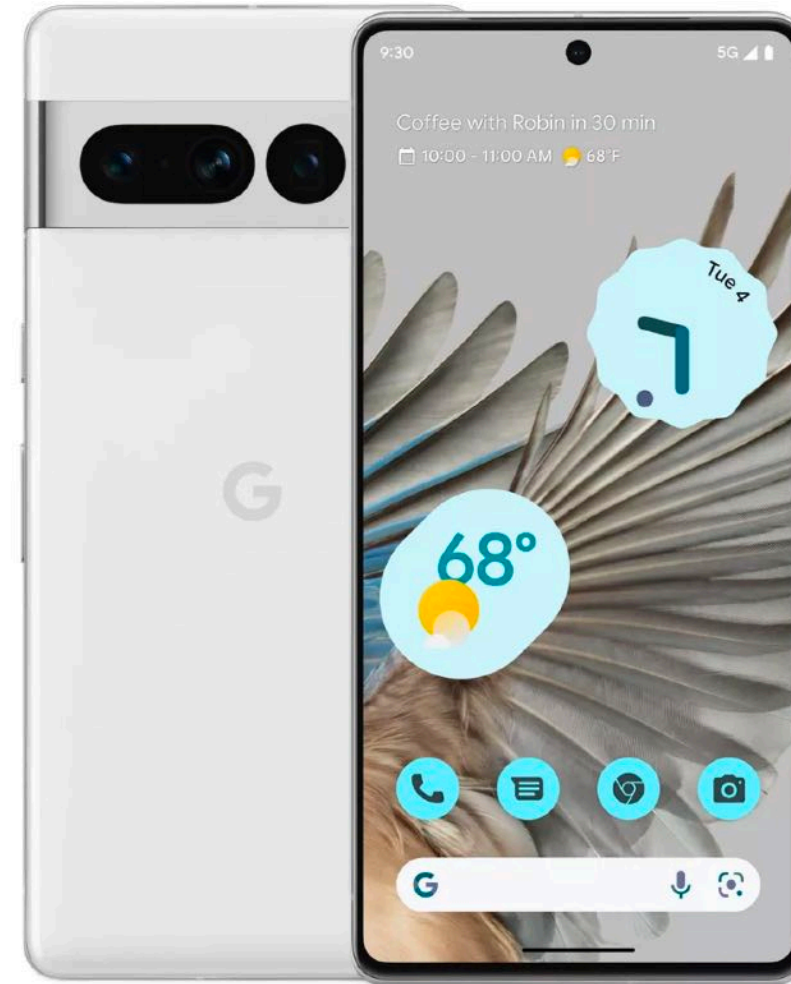


**2,815 mAmp hours  
(10.7 Watt hours)**

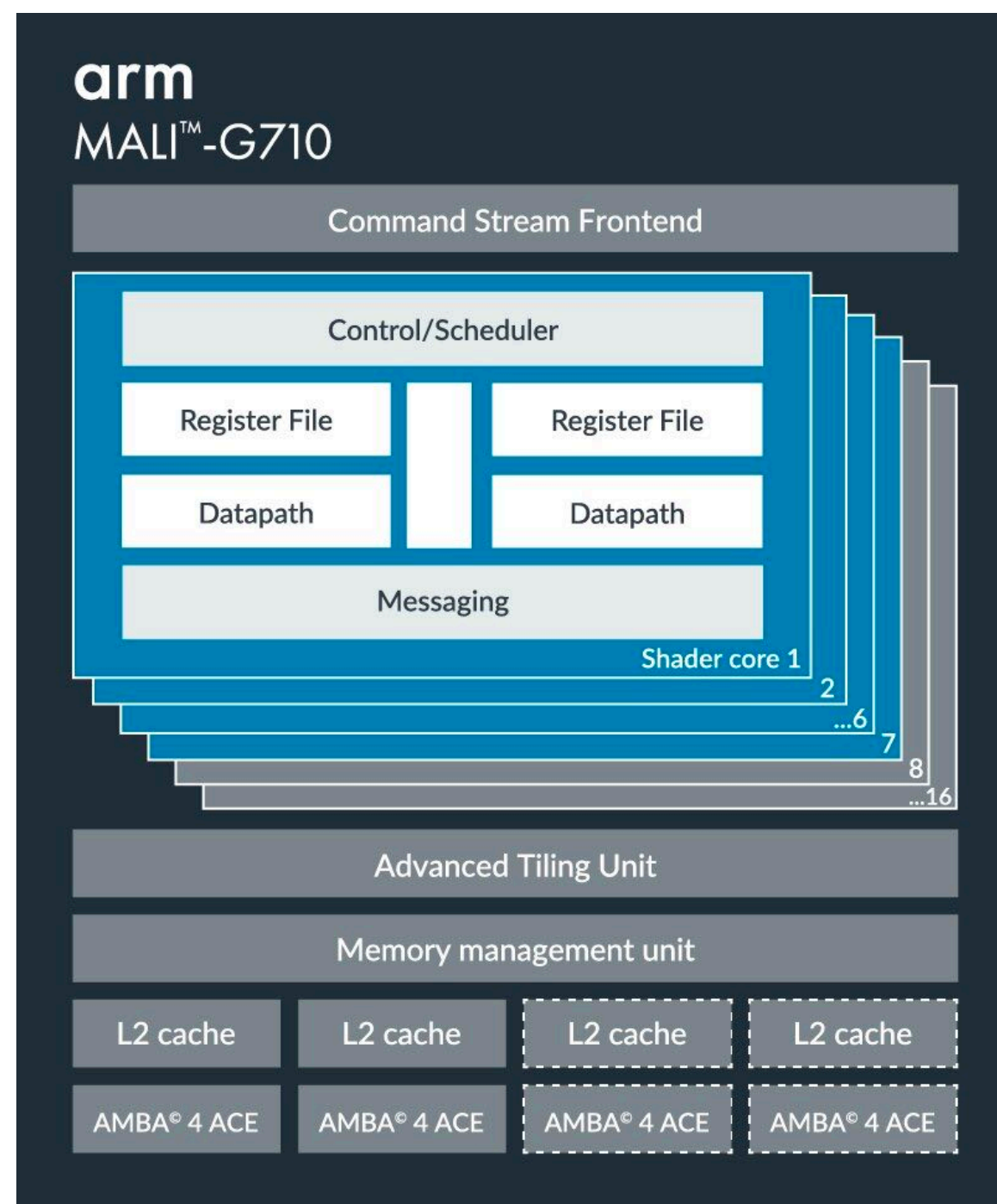


# Graphics processors (GPUs) in these mobile phones

Google Pixel 7



Apple iPhone 12



← **ARM Mali G710 GPU**

**Custom Apple GPU in A14 Bionic Processor**

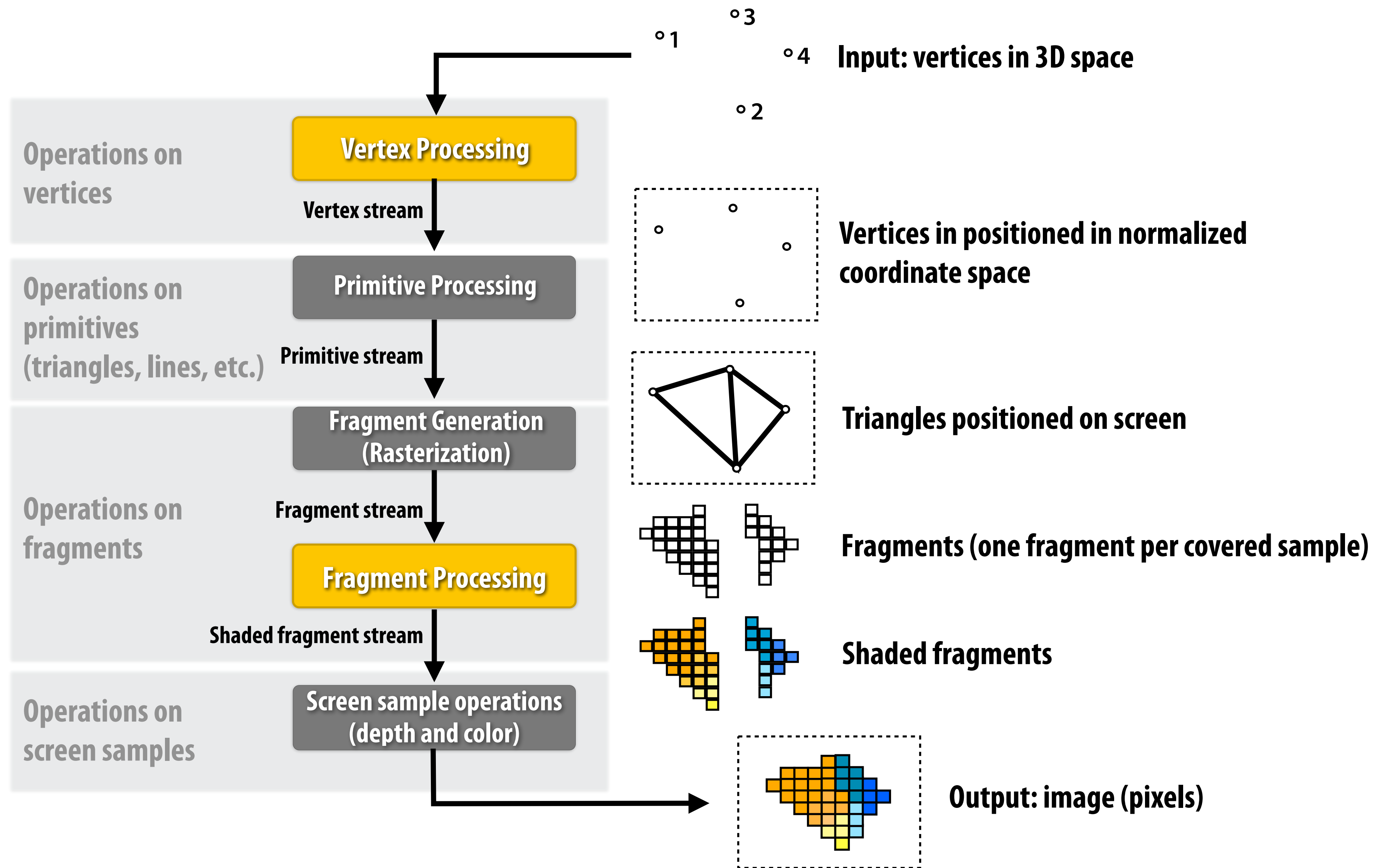


# Ways to conserve power

- **Compute less**
  - **Reduce the amount of work required to render a picture**
  - **Less computation = less power**
  
- **Read less data**
  - **Data movement has high energy cost**



# Simple OpenGL/Direct3D graphics pipeline \*



\* Several stages of the modern OpenGL pipeline are omitted

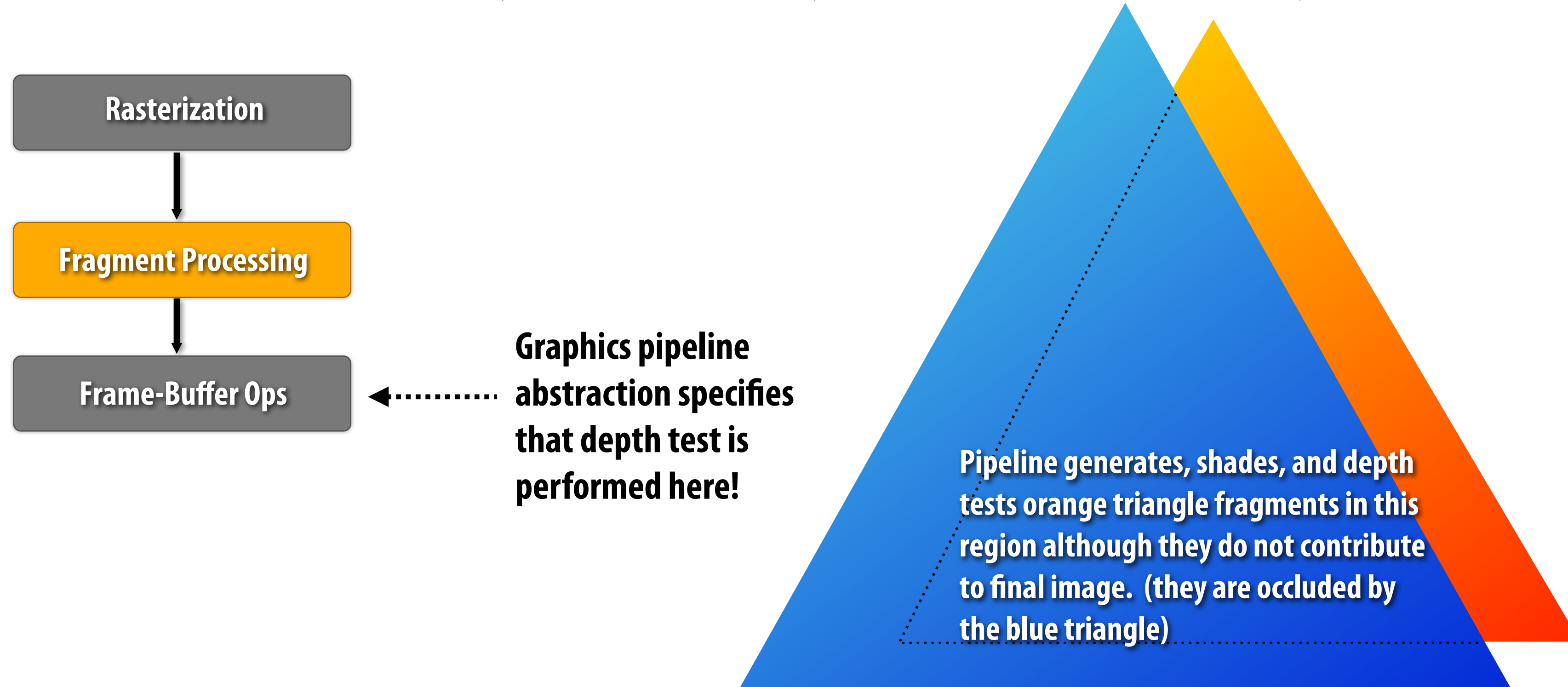


# Early depth culling (“Early Z”)



# Depth testing as we've described it

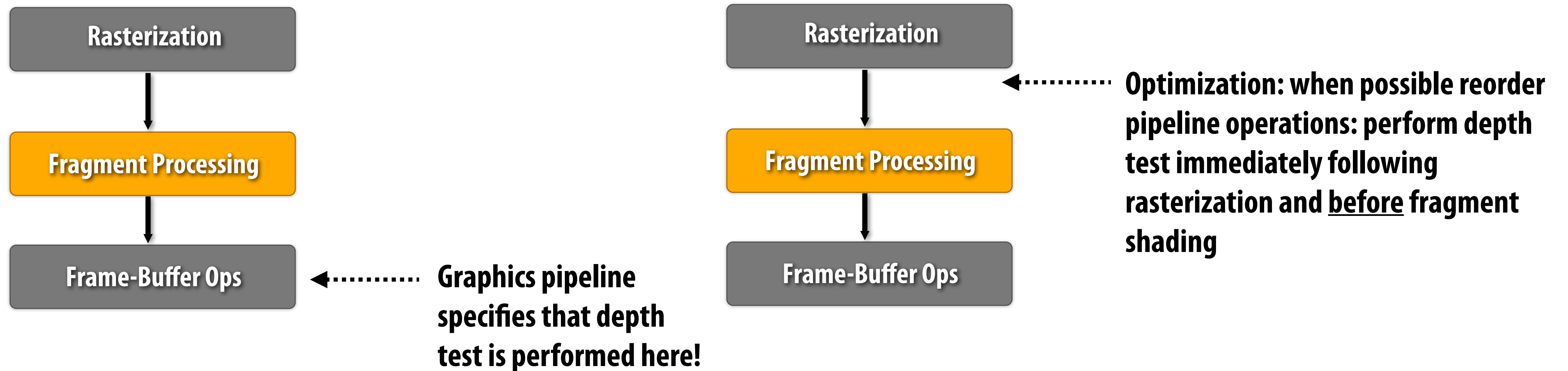
- Implemented by all modern GPUs, not just mobile GPUs
- Application needs to sort geometry to make early Z most effective. *Why?*





# Early Z culling

- Implemented by all modern GPUs, not just mobile GPUs
- Application needs to sort geometry to make early Z most effective. *Why?*



**Key assumption: occlusion results do not depend on fragment shading**

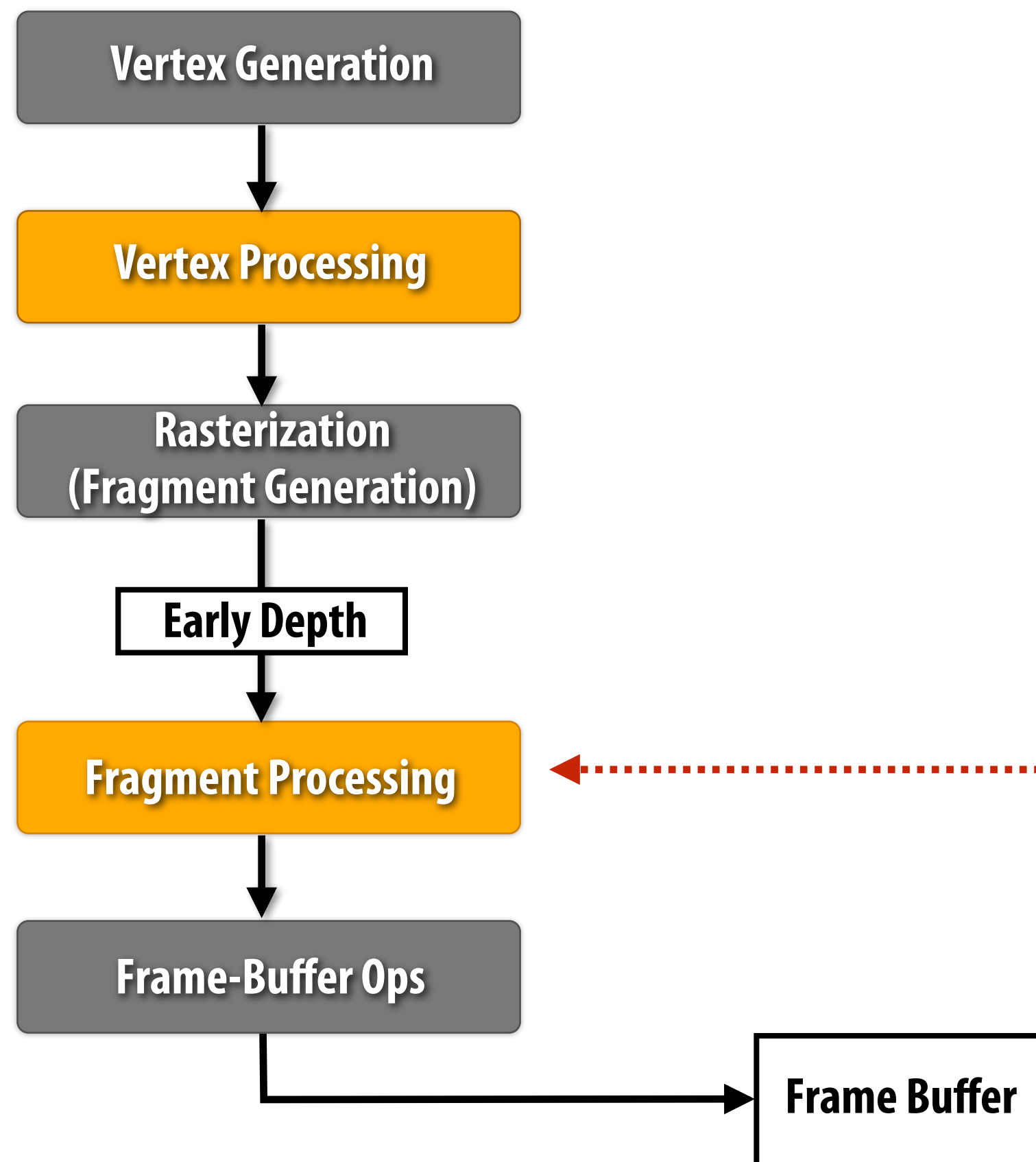
- Example operations that prevent use of this early Z optimization: enabling alpha test, fragment shader modifies fragment's Z value



# Deferred shading



# The graphics pipeline



## “Forward” rendering

**Typical use of fragment processing stage:  
evaluate application-defined function from  
surface inputs to surface color (reflectance)**

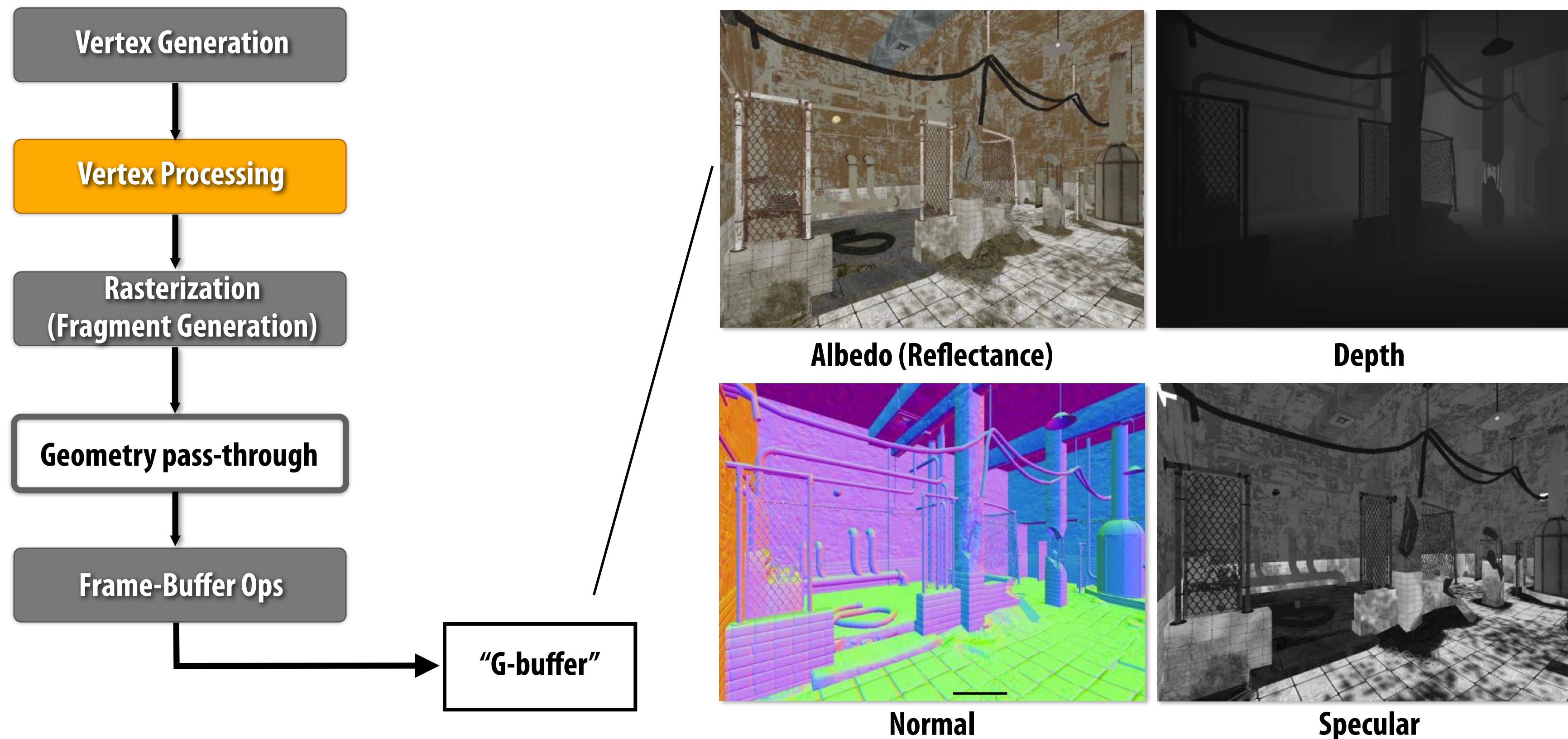


# Deferred shading: two steps

**Step 1: Do not use traditional pipeline to generate RGB image**

Fragment shader now outputs surface properties (future shading inputs)  
(e.g., position, normal, material diffuse color, specular color)

Rendering output is a screen-size 2D buffer representing information about the surface geometry visible at each pixel  
(called a “g-buffer”, for geometry buffer)





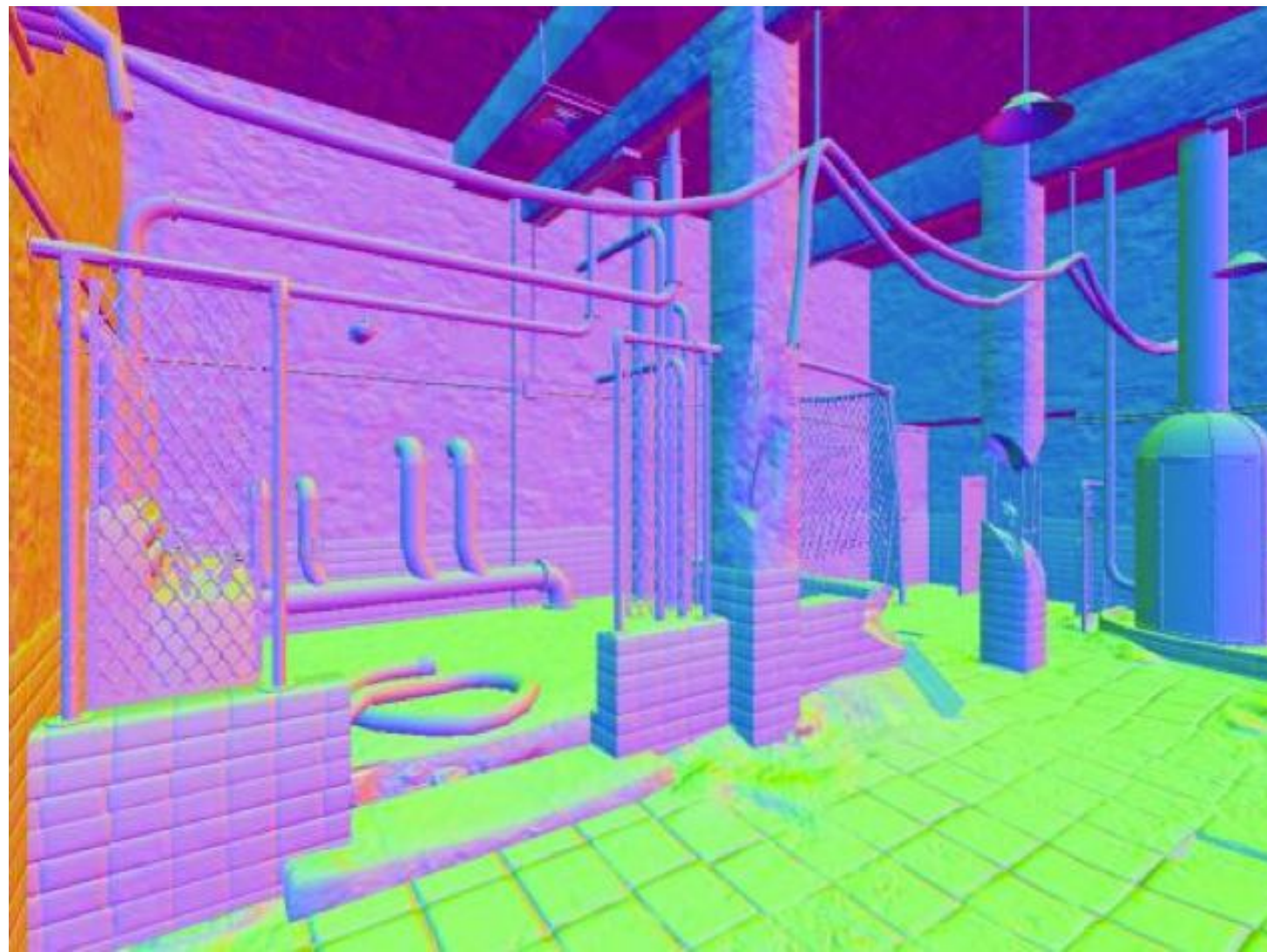
# G-buffer = “geometry” buffer



Albedo (Reflectance)



Depth



Normal



Specular



# Example G-buffer layout

Graphics pipeline configured to render to four RGBA output buffers + depth  
(32-bits per pixel, per buffer)

R8	G8	B8	A8	
	Depth 24bpp		Stencil	DS
	Lighting Accumulation RGB		Intensity	RT0
Normal X (FP16)		Normal Y (FP16)		RT1
Motion Vectors XY		Spec-Power	Spec-Intensity	RT2
	Diffuse Albedo RGB		Sun-Occlusion	RT3

Source: W. Engel, "Light-Prepass Renderer Mark III" SIGGRAPH 2009 Talks

Intuitive to consider G-buffer as one big render target with "fat" pixels

In the example above:  $32 \times 5 = 160$  bits = 20 bytes per pixel

Up to 96-160 bits per pixel in some games.



# Compressed G-buffer layout

- Material information is compressed using indirection
  - Store material ID in G-buffer
  - Material parameters other than albedo (specular shape/roughness/color) stored in table indexed by material ID

## G-buffer layout in Bungie's Destiny (2014)

8	8	8	8
Albedo Color RGB			Ambient Occlusion
Normal XYZ * (Biased Specular Smoothness)			Material ID
Depth			Stencil

RT0  
RT1  
DS



Example material ID visualization



# Two-pass deferred shading algorithm

## ■ Pass 1: G-buffer generation pass

- Render complete scene geometry using traditional pipeline
- Write visible geometry information to G-buffer

After all geometry processing is done...

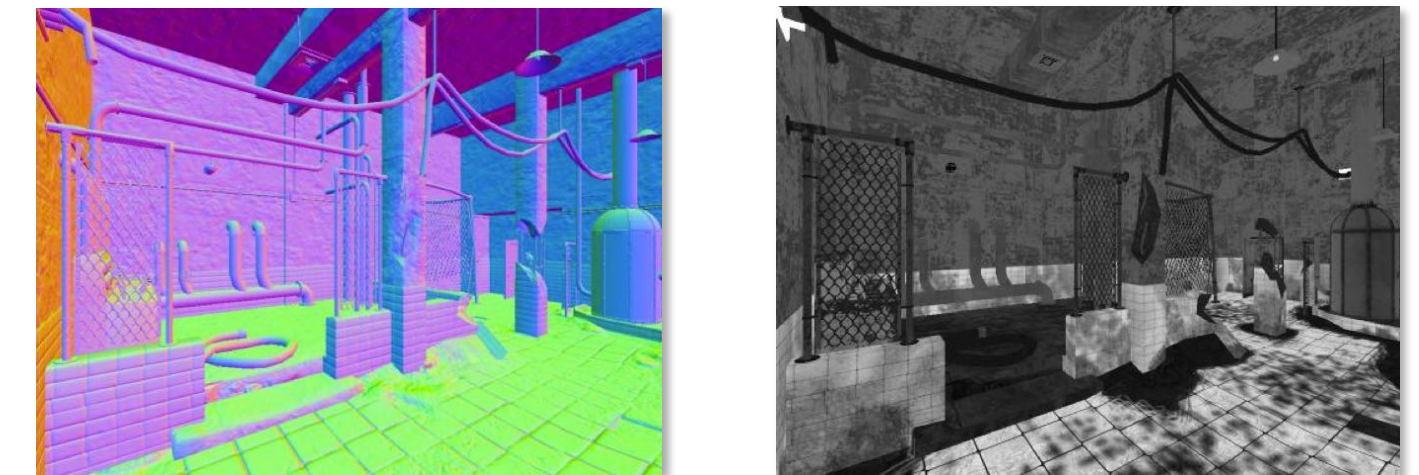
## ■ Pass 2: shading/lighting pass

For each G-buffer sample  $(x,y)$ :

- Read G-buffer data for current sample  $(x,y)$
- Compute shading by accumulating contribution to reflectance of all lights
- Output final surface color for sample  $(x,y)$

Shading/lighting computations are “deferred” until all geometry processing is complete...

G-buffer Inputs



Final Image



**Read data less often**



# Reading less data conserves power

- **Goal: redesign algorithms to make good use of on-chip memory or processor caches**
  - **And therefore transfer less data from memory**
- **A fact you might not have heard:**
  - It is *far more* costly (in energy) to load/store data from memory, than it is to perform an arithmetic operation

**“Ballpark” numbers** [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

- Integer op: ~ 1 pJ \*
- Floating point op: ~20 pJ \*
- Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
- Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

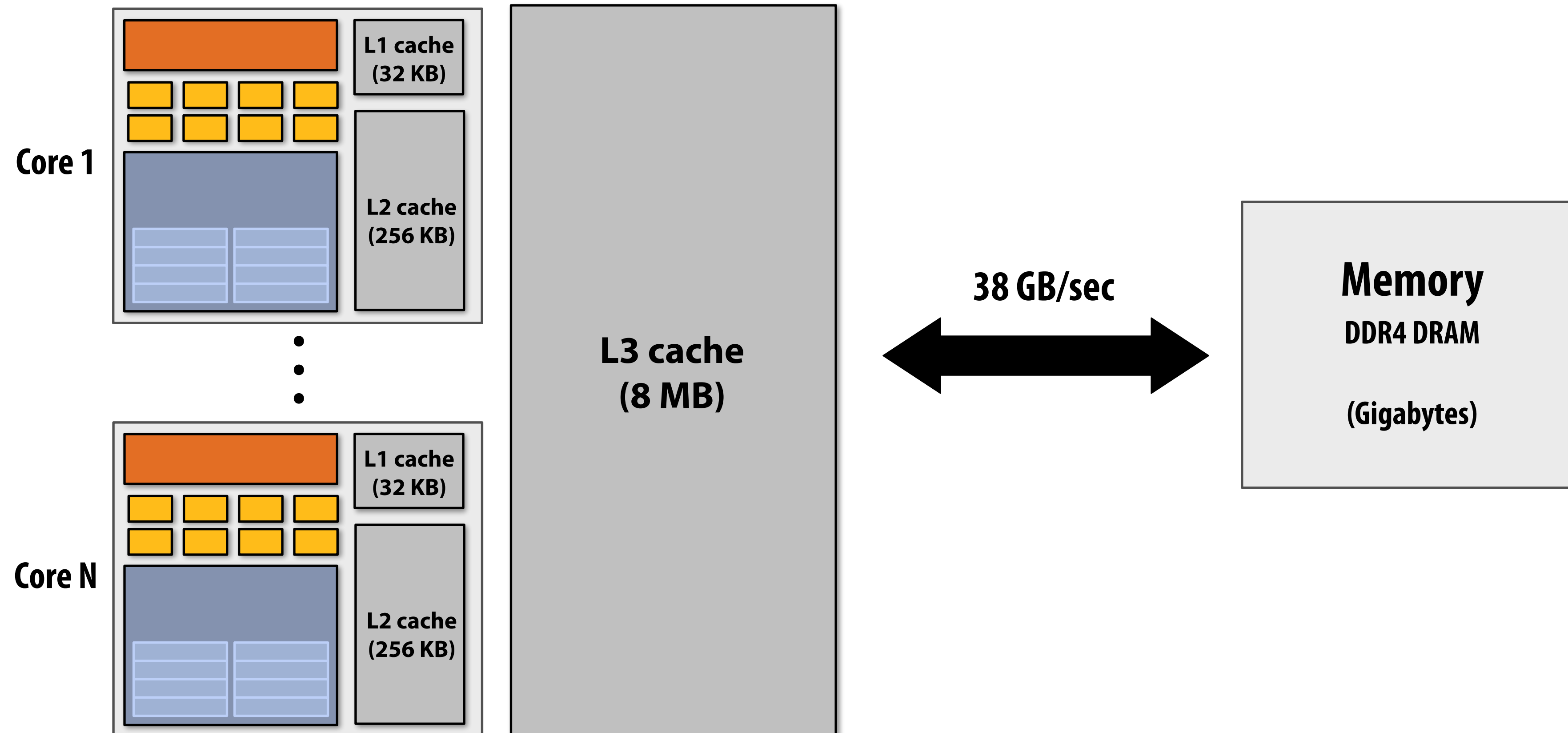
## Implications

- Reading 10 GB/sec from memory: ~1.6 watts

\* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.



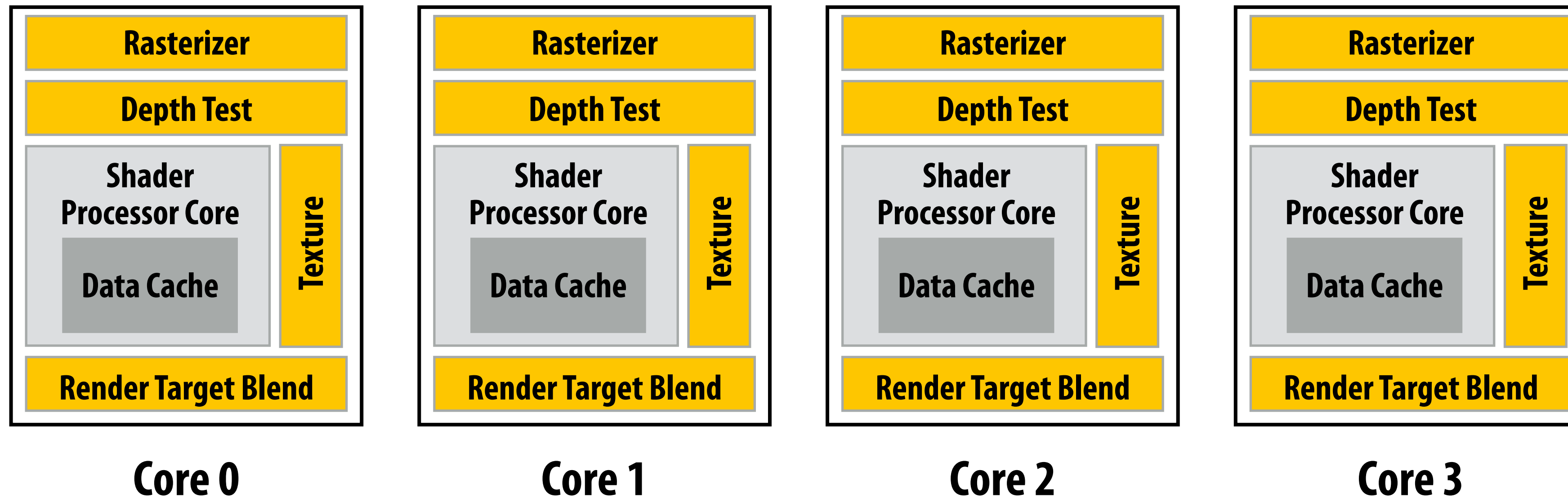
# What does a data cache do in a processor?





# Today: a simple mobile GPU

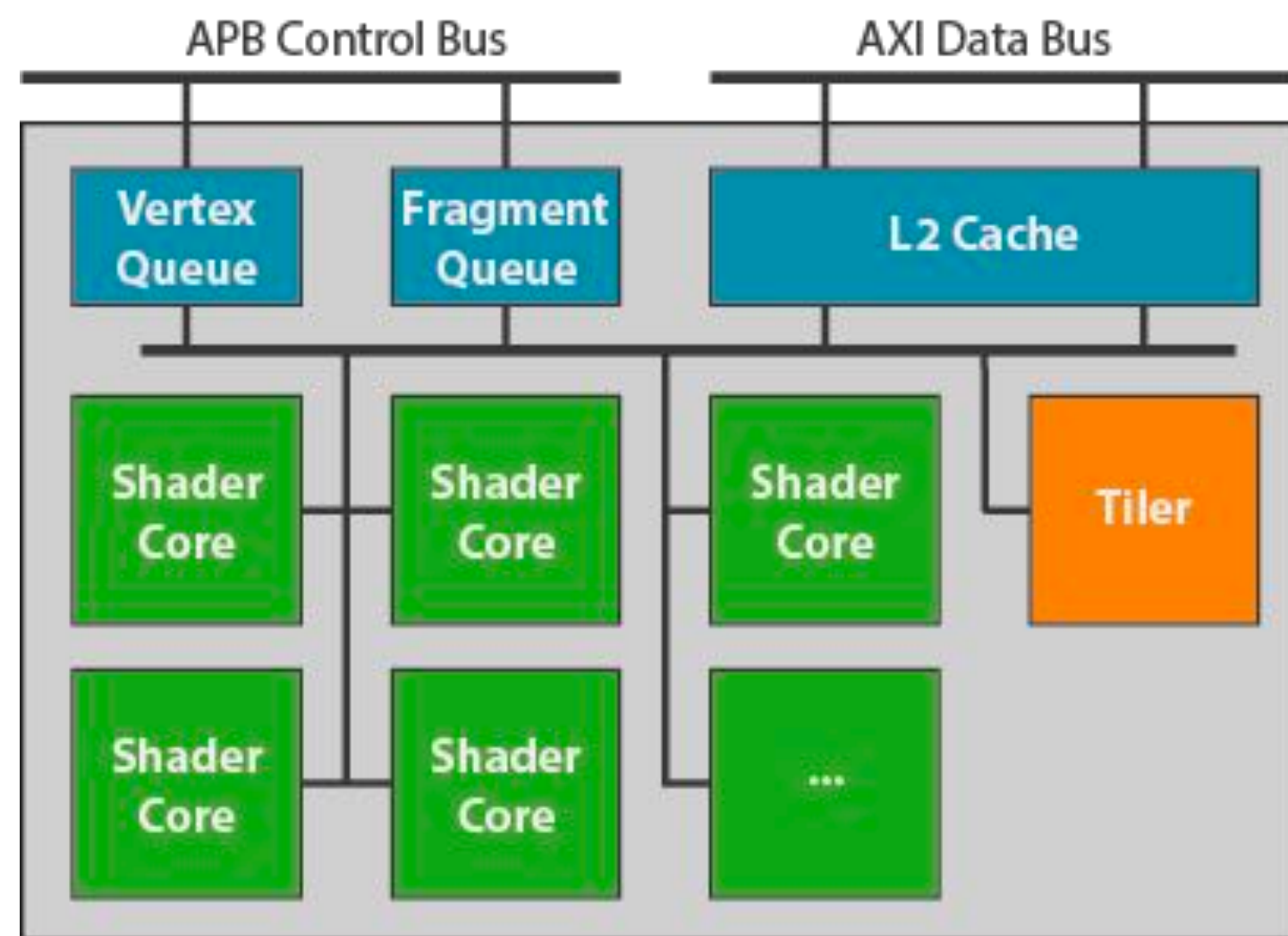
- A set of programmable cores (run vertex and fragment shader programs)
- Hardware for rasterization, texture mapping, and frame-buffer access



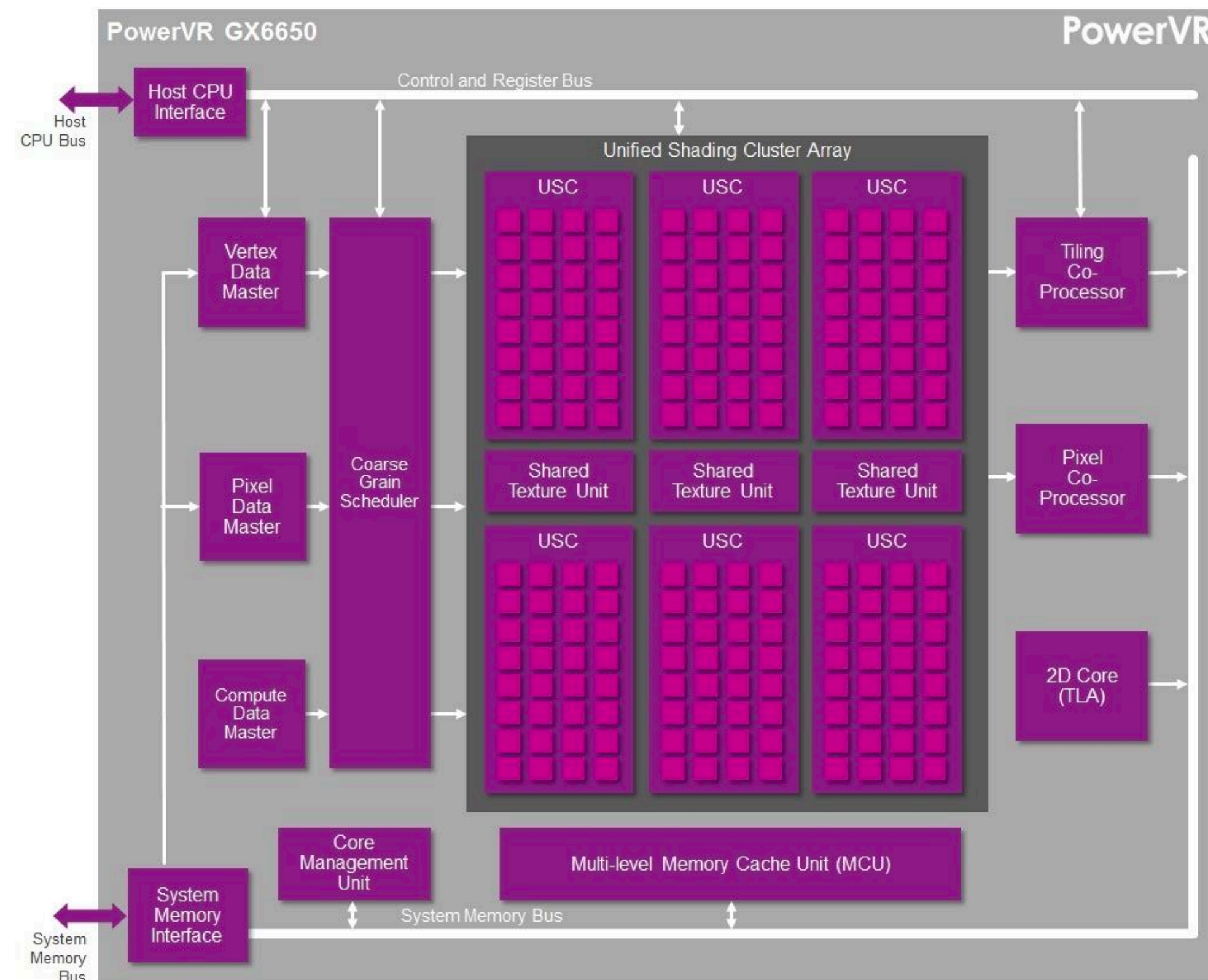
# Block diagrams from vendors

## ARM Mali G72MP18

Mali GPU Block Model



## Imagination PowerVR (in earlier iPhones)





# Let's consider different workloads

## Average triangle size



Image credit:

<https://www.theverge.com/2013/11/29/5155726/next-gen-supplementary-piece>

<http://www.mobymgames.com/game/android/ghostbusters-slime-city/screenshots/gameShotId,852293/>

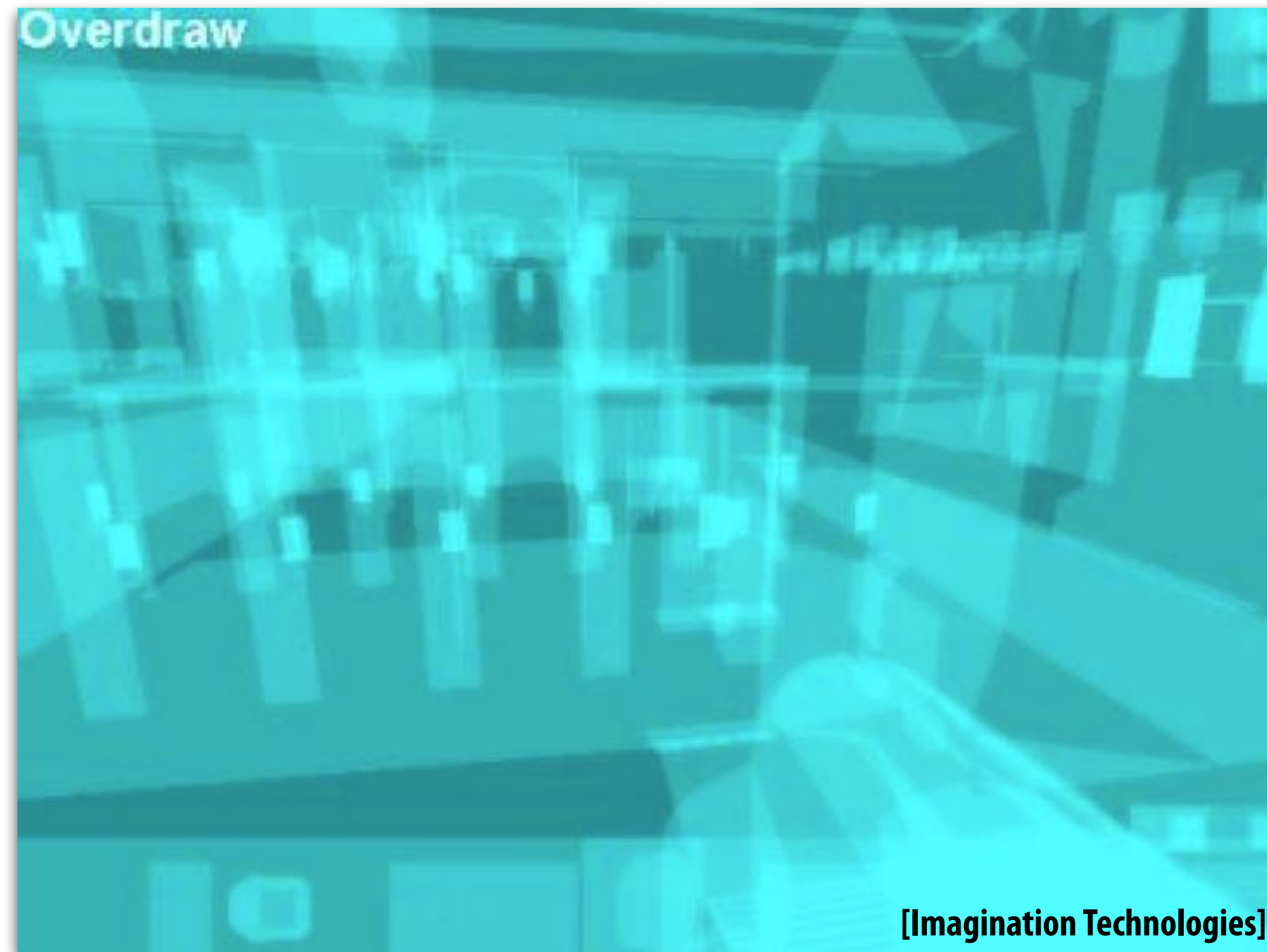




# Let's consider different workloads

Scene depth complexity

Average number of overlapping triangles per pixel



**In this visualization: bright colors = more overlap**



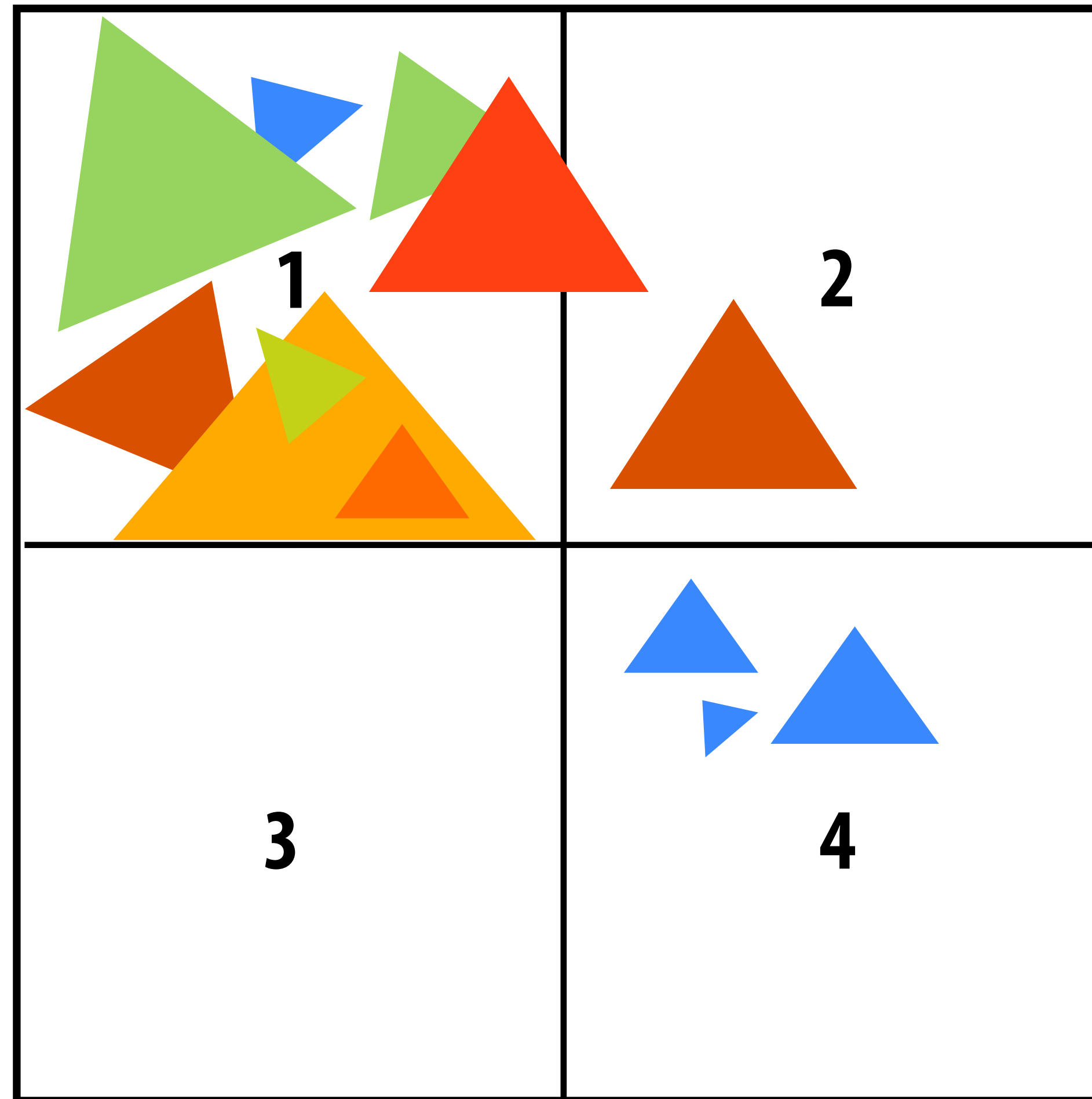
# One very simple solution

- Let's assume four GPU cores
- Divide screen into four quadrants, each processor processes all triangles, but only renders triangles that overlap quadrant
- *Problems?*



# Problem: unequal work partitioning

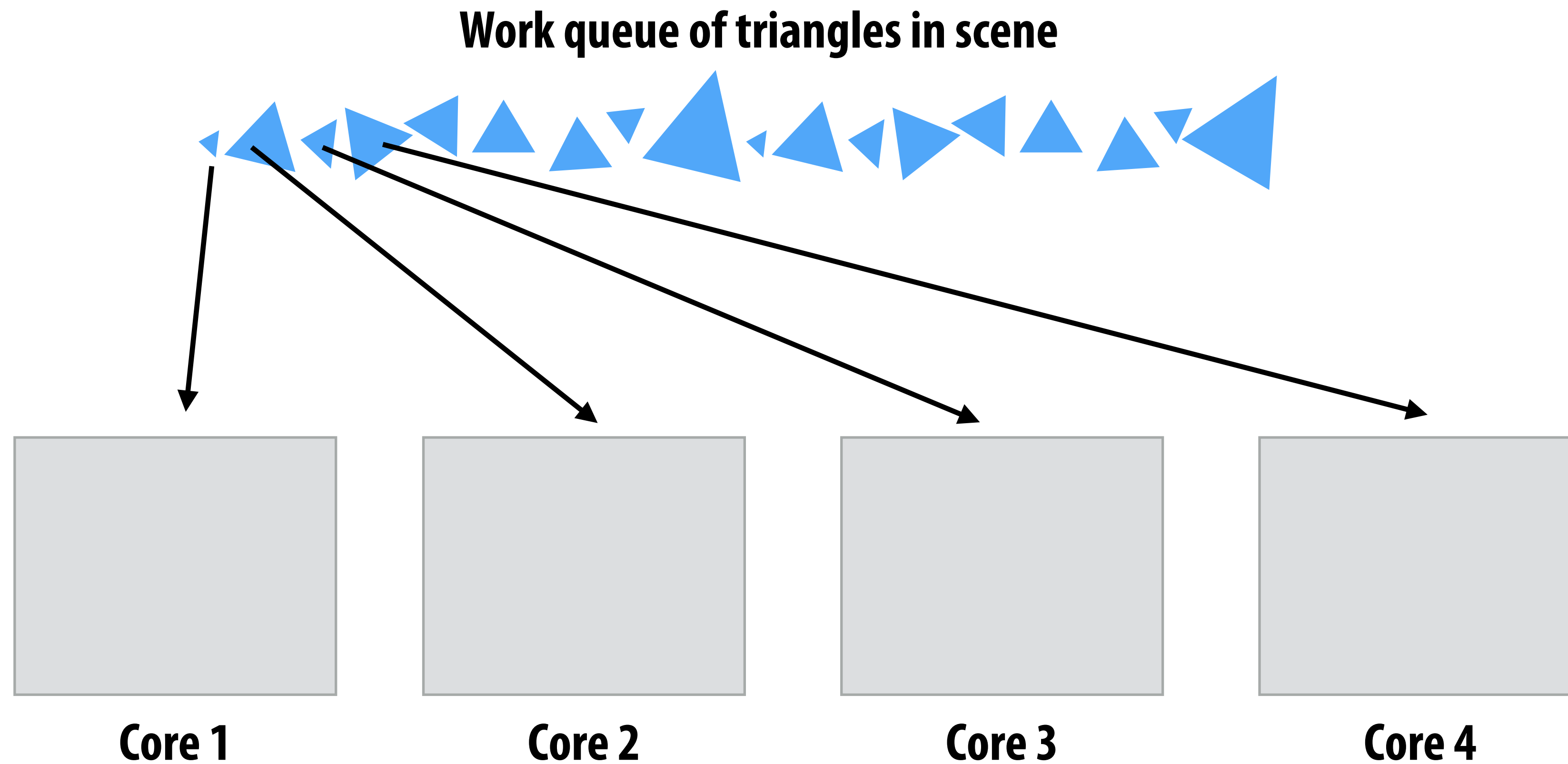
(partition the primitives to parallel units based on screen overlap)





# Step 1: parallel geometry processing

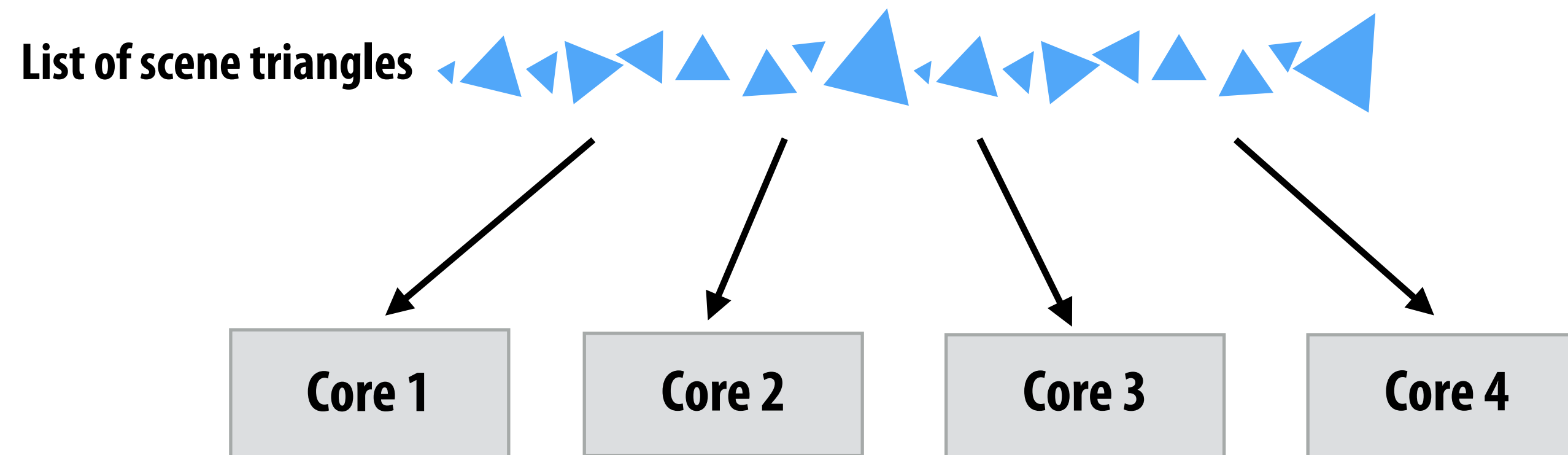
- Distribute triangles to the four processors (e.g., round robin)
- In parallel, processors perform vertex processing





# Step 2: sort triangles into per-tile lists





- Divide screen into tiles, one triangle list per “tile” of screen (called a “bin”)
- Core runs vertex processing, computes 2D triangle/screen-tile overlap, inserts triangle into appropriate bin(s)



After processing first five triangles:

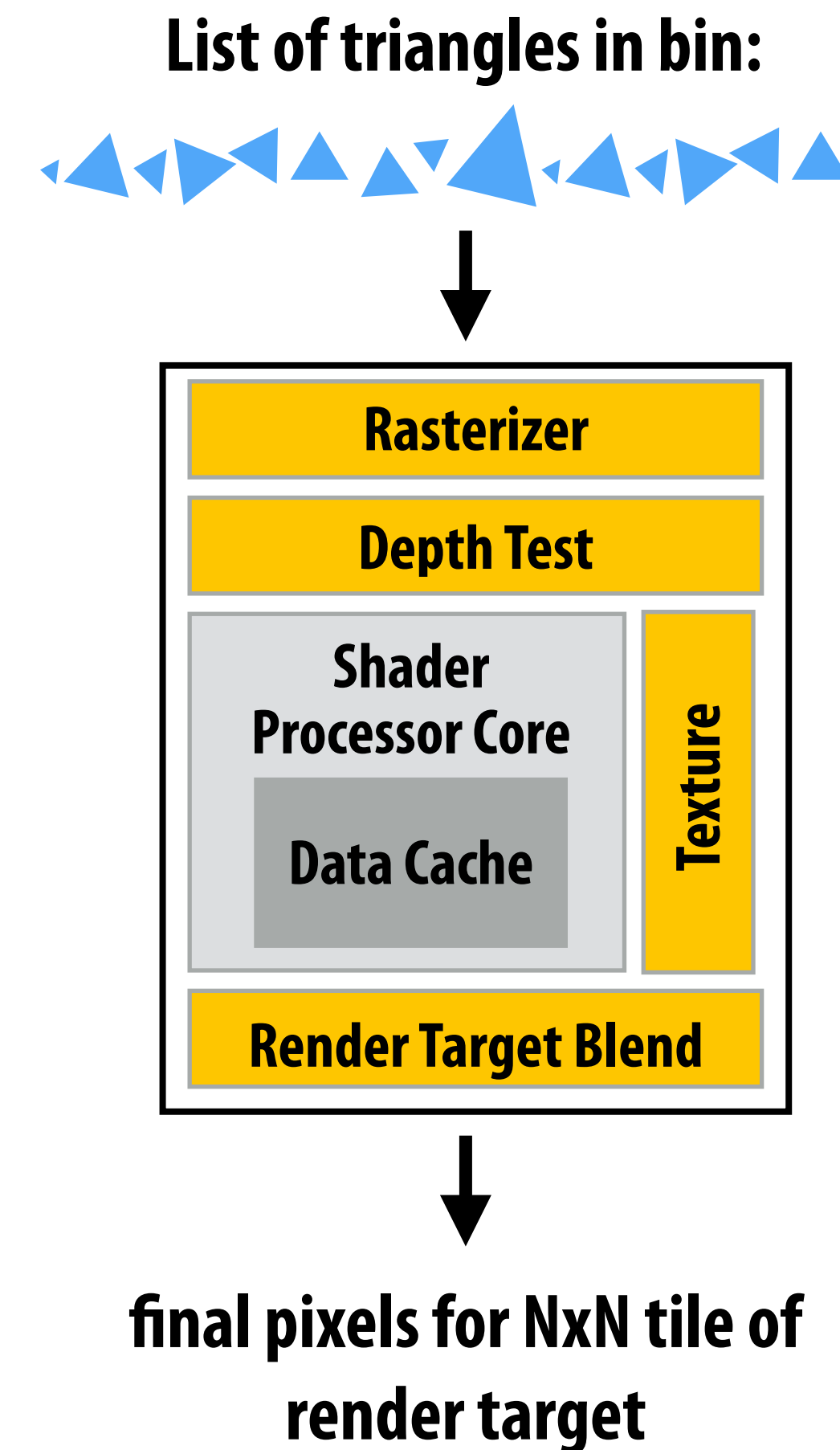
Bin 1 list: 1,2,3,4

Bin 2 list: 4,5

 Bin 1	   Bin 2	Bin 3	Bin 4
Bin 5	Bin 6	Bin 7	Bin 8
Bin 9	Bin 10	Bin 11	Bin 12

# Step 3: per-tile processing

- In parallel, the cores process the bins: performing rasterization, fragment shading, and frame buffer update
- While (more bins left to process):
  - Assign bin to available core
  - For all triangles in bin:
    - Rasterize
    - Fragment shade
    - Depth test
    - Render target blend

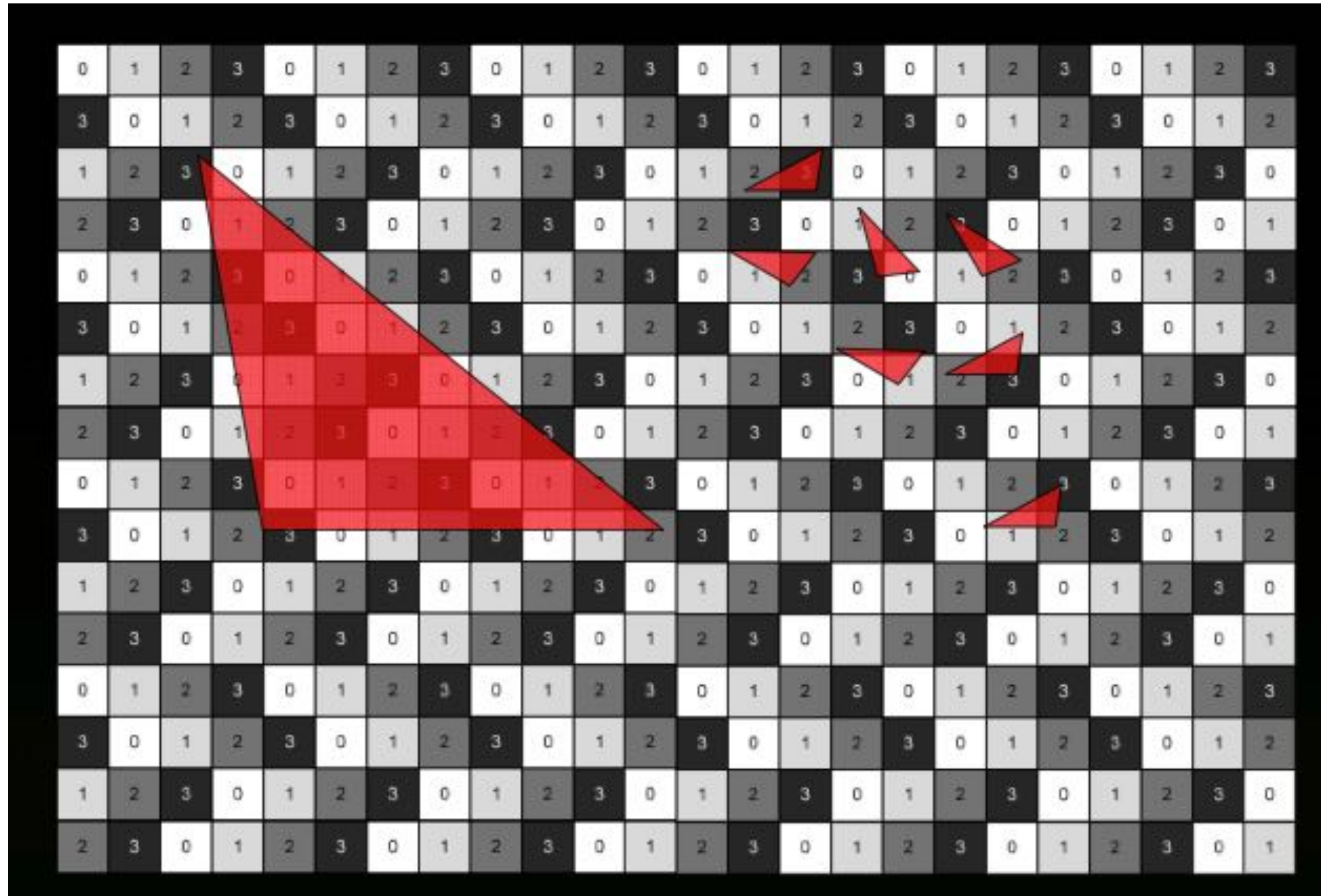




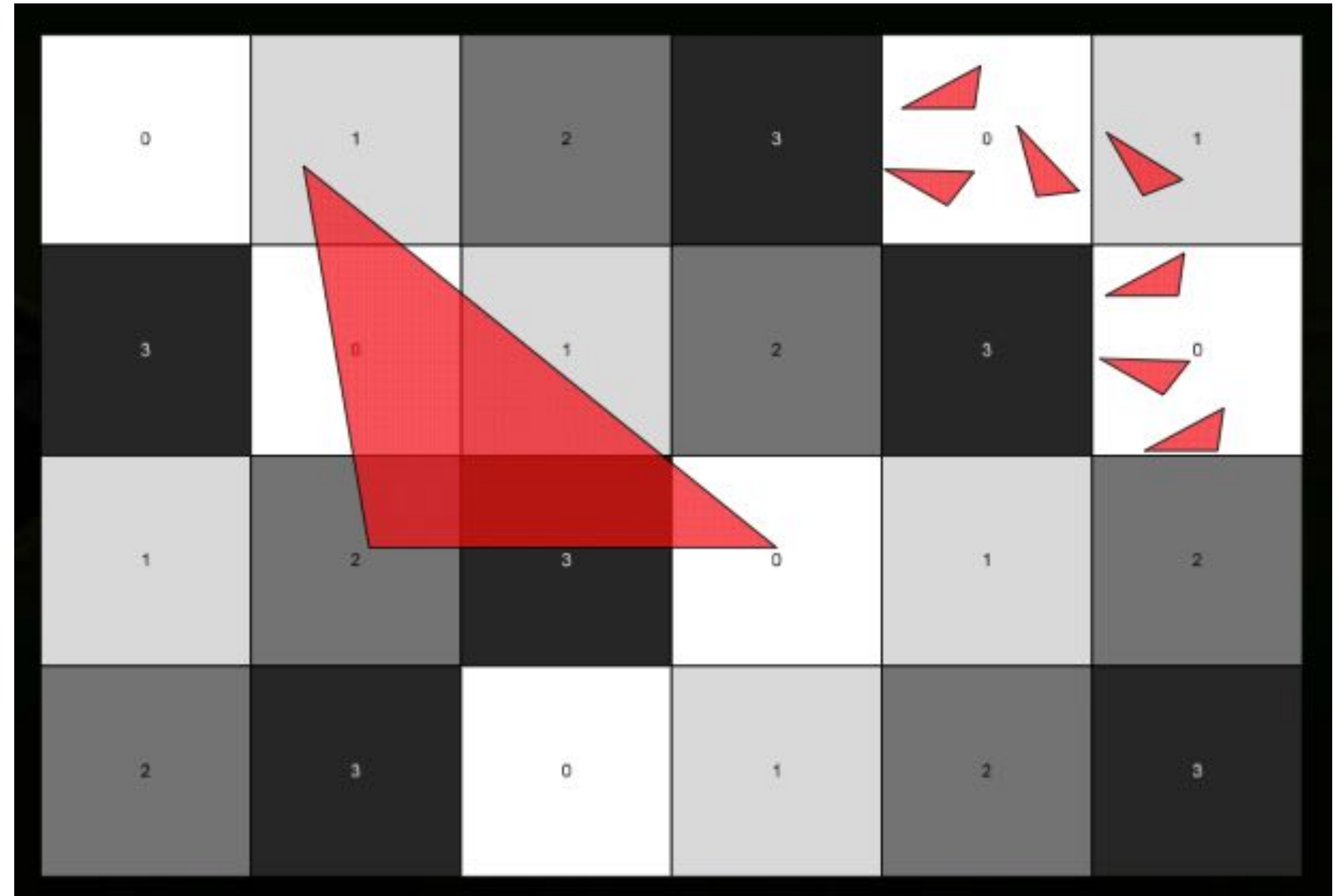
**What should the size of tiles be?**

# What should the size of the bins be?

Fine granularity



Coarse granularity





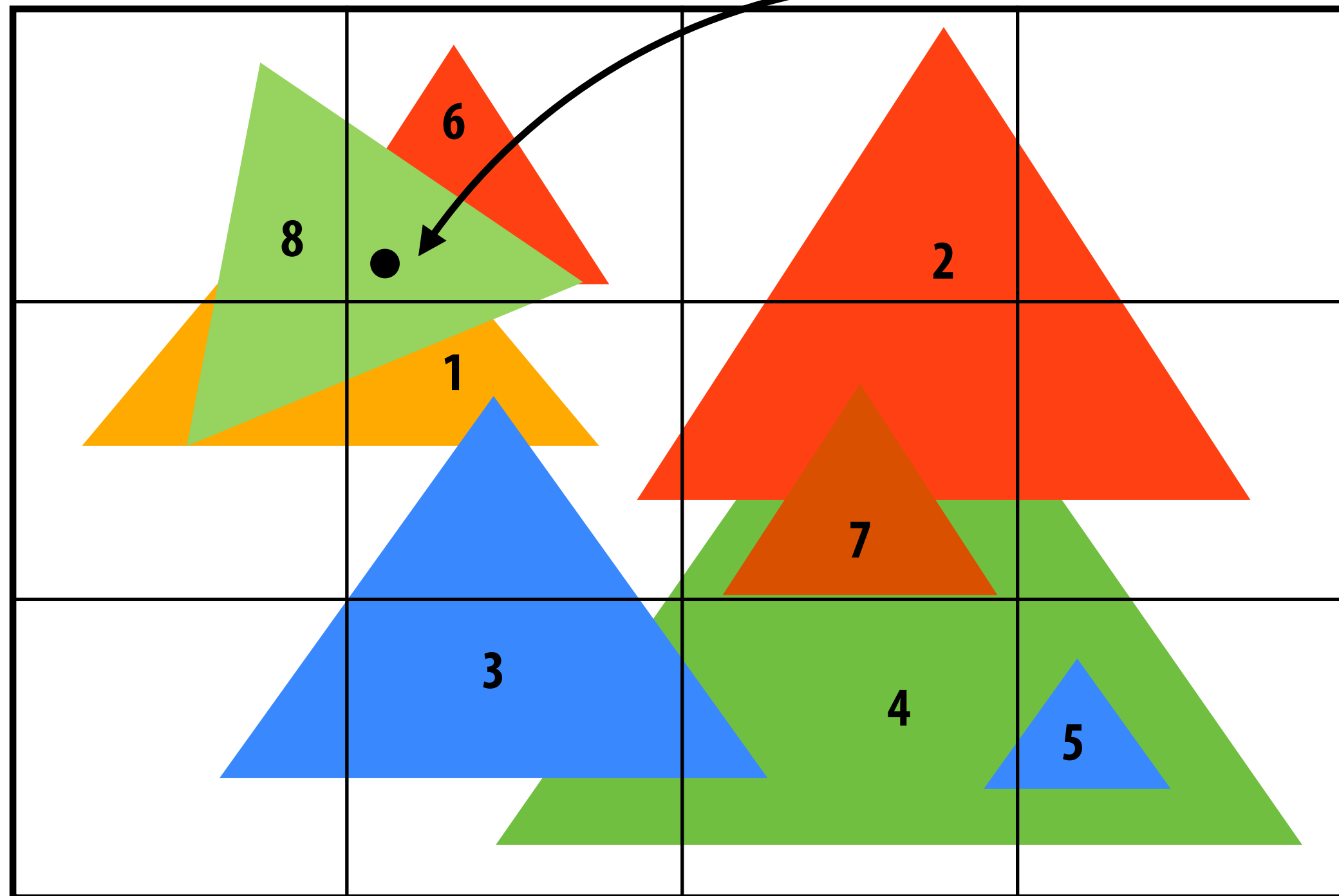
# What size should the tiles be?

- Small enough for a tile of the color buffer and depth buffer (potentially supersampled) to fit in a shader processor core's on-chip storage (i.e., cache)
- Tile sizes in range 16x16 to 64x64 pixels are common
- ARM Mali GPU: commonly uses 16x16 pixel tiles



# Tiled rendering “sorts” the scene in 2D space to enable efficient color/depth buffer access

Consider rendering without a sort:  
(process triangles in order given by application)



This sample is updated three times during rendering, but it may have fallen out of cache in between accesses

Now consider step 3 of a tiled renderer:

```
Initialize Z and color buffer for tile
for all triangles in tile:
  for all each fragment:
    shade fragment
    update depth/color
write color tile to final image buffer
```

Q. Why doesn't the renderer need to read color or depth buffer from memory?

Q. Why doesn't the renderer need to write depth buffer in memory? \*

\* Assuming application does not need depth buffer for other purposes.



# Recall: deferred shading using a G-buffer

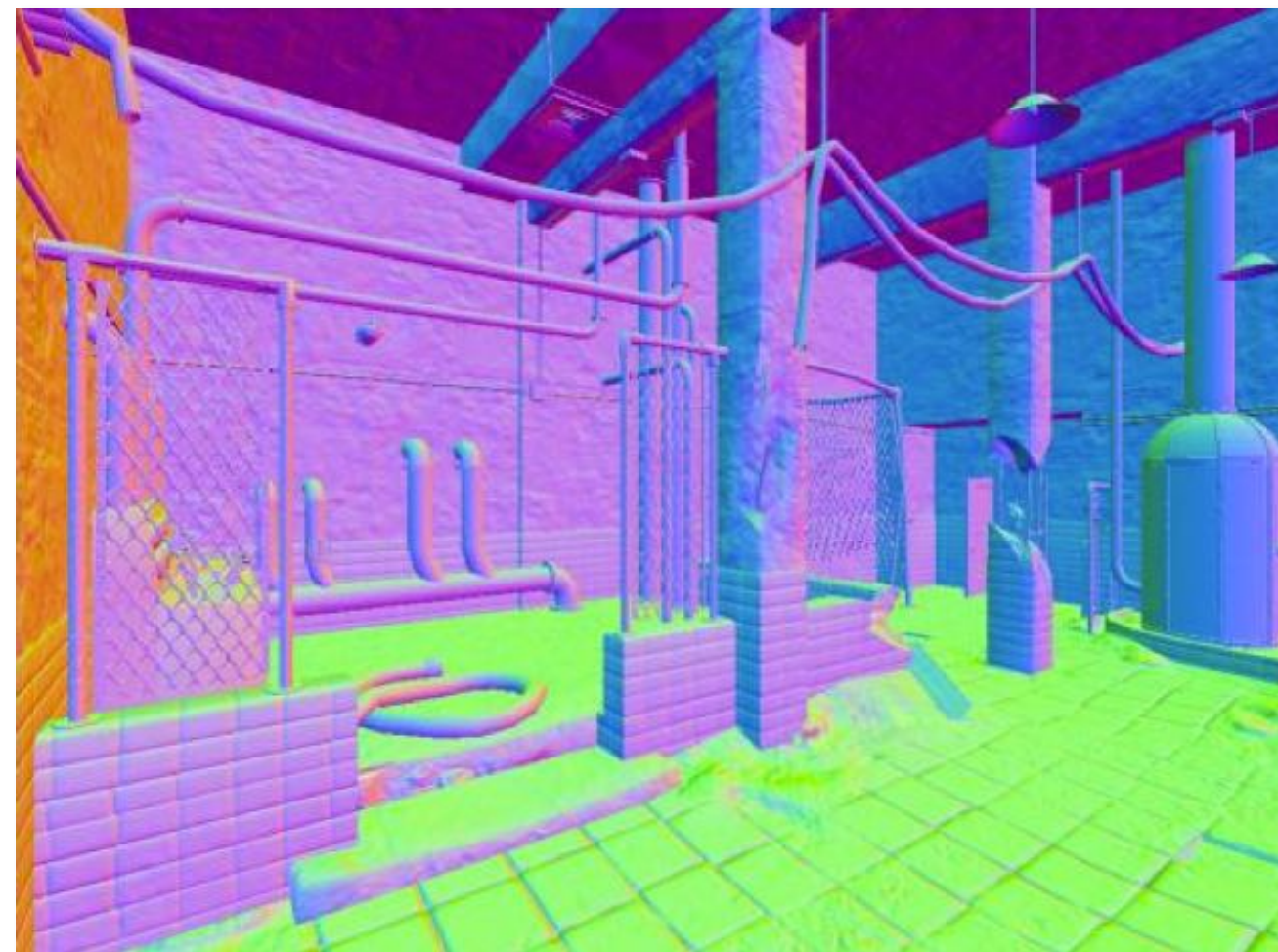
Key benefit: shade each sample *exactly* once.



Albedo (Reflectance)



Depth



Normal

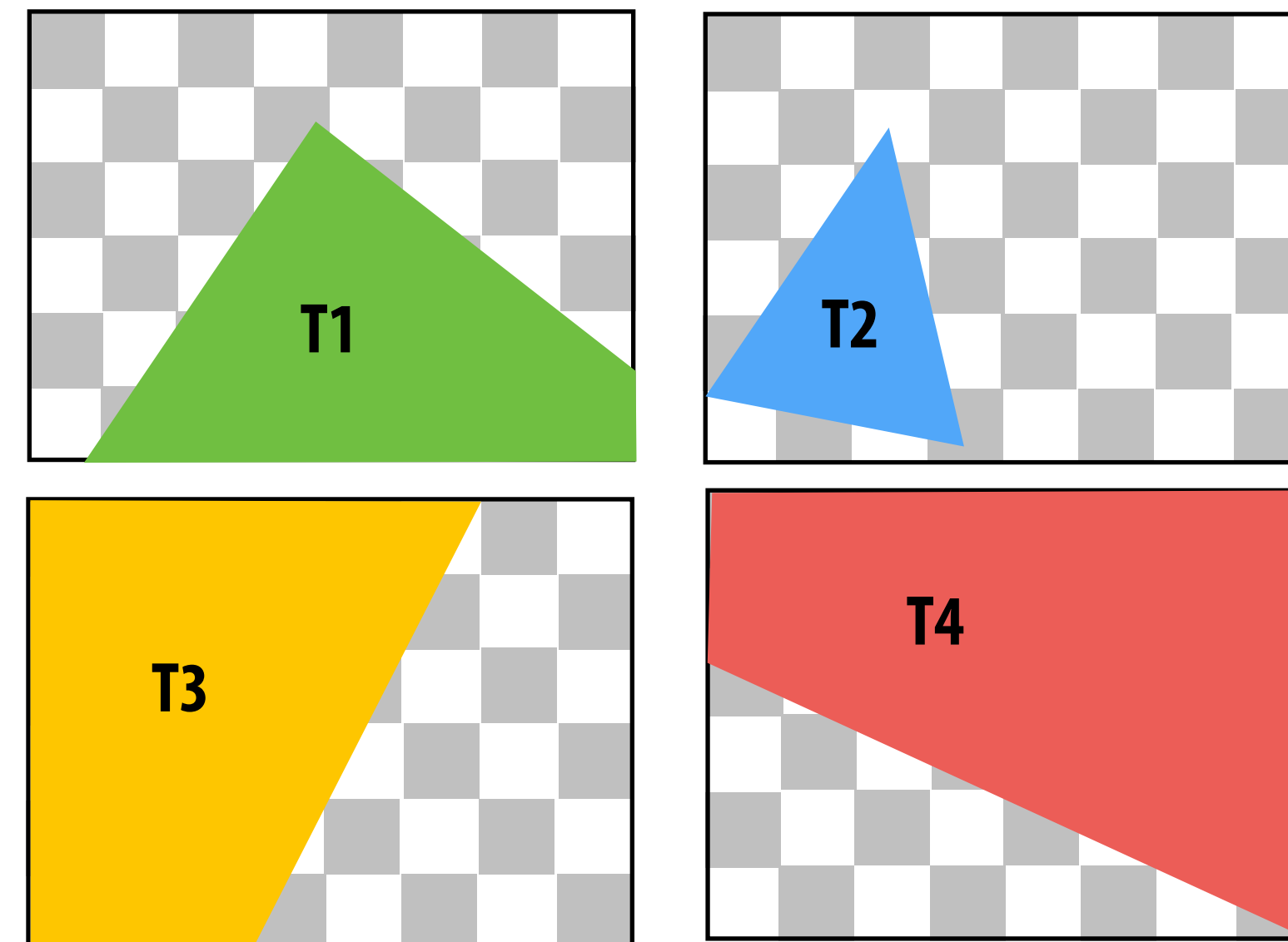
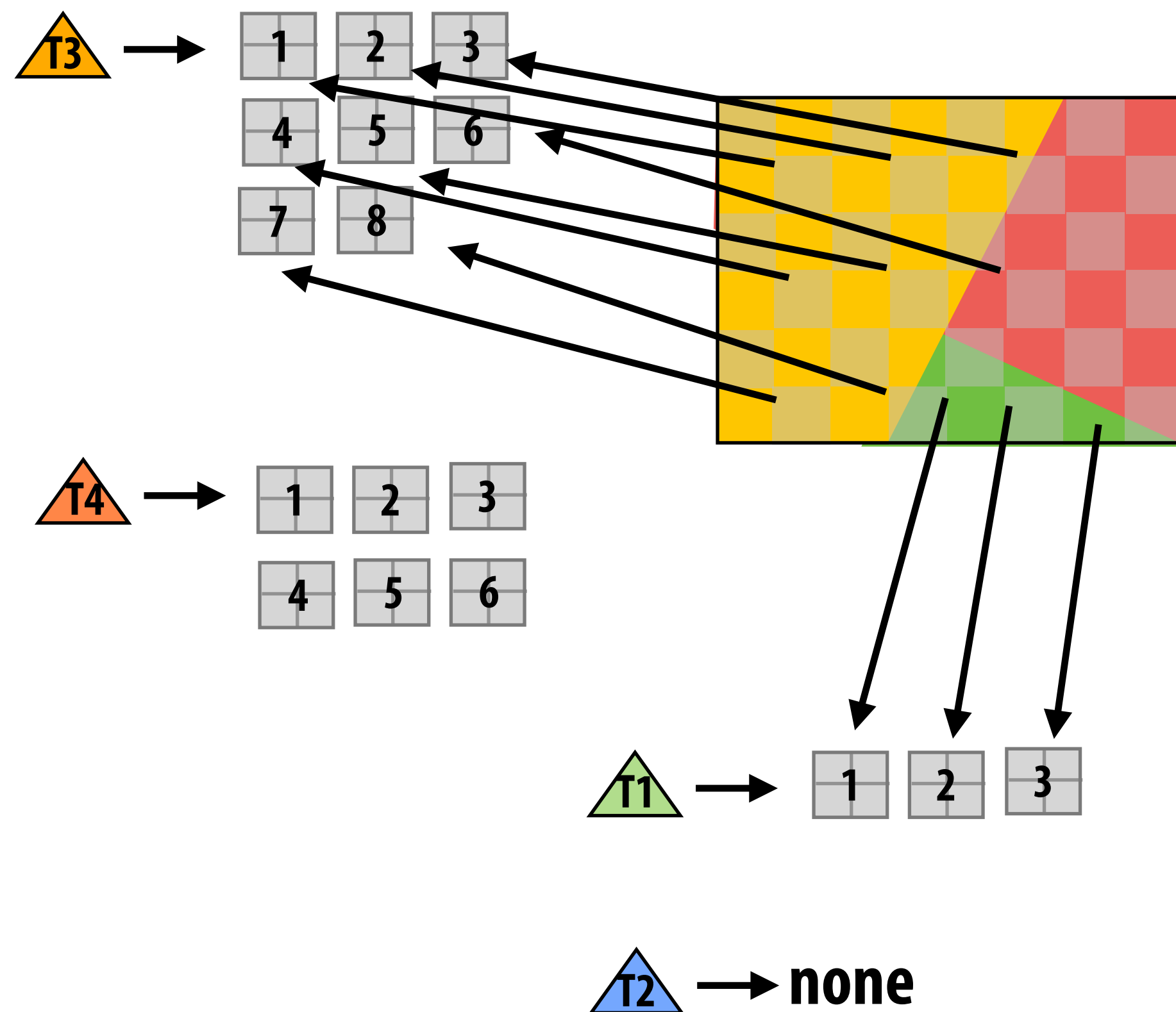


Specular



# Tile-based deferred rendering (TBDR)

- Many mobile GPUs implement deferred shading in the hardware!
- Divide step 3 of tiled pipeline into two phases:
- Phase 1: compute what triangle/quad fragment is visible at every sample
- Phase 2: perform shading of only the visible quad fragments





# The story so far

- **Computation-saving optimizations (shade less)**
  - early Z cull
  - tile-based deferred shading
- **Bandwidth-saving optimizations**
  - tile-based rendering
  - **many more...**

# **Texture compression (reducing bandwidth cost)**



# Recall: a texture sampling operation

1. Compute  $u$  and  $v$  from screen sample  $x, y$  (via evaluation of attribute equations)
2. Compute  $du/dx, du/dy, dv/dx, dv/dy$  differentials from quad-fragment samples
3. Compute mipmap level  $L$
4. Convert normalized texture coordinate  $(u, v)$  to texture coordinates  $texel\_u, texel\_v$
5. Compute required texels in window of filter \*\*
6. **If texture data in filter footprint (eight texels for trilinear filtering) is not in cache:**
  - **Load required texels (in compressed form) from memory**
  - **Decompress texture data**
7. Perform tri-linear interpolation according to  $(texel\_u, texel\_v, L)$

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

# 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 into texture map (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)

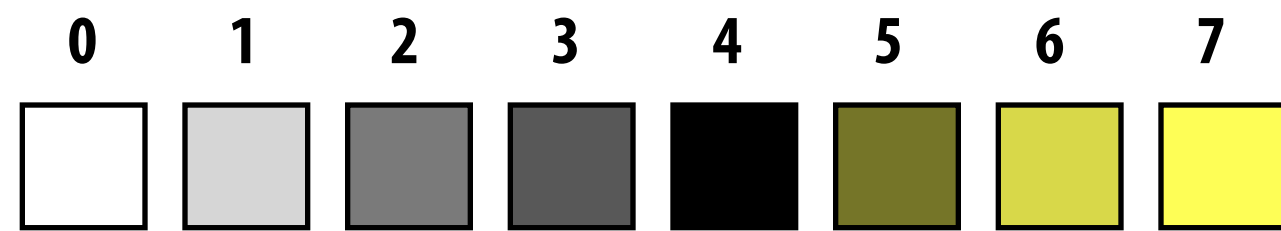
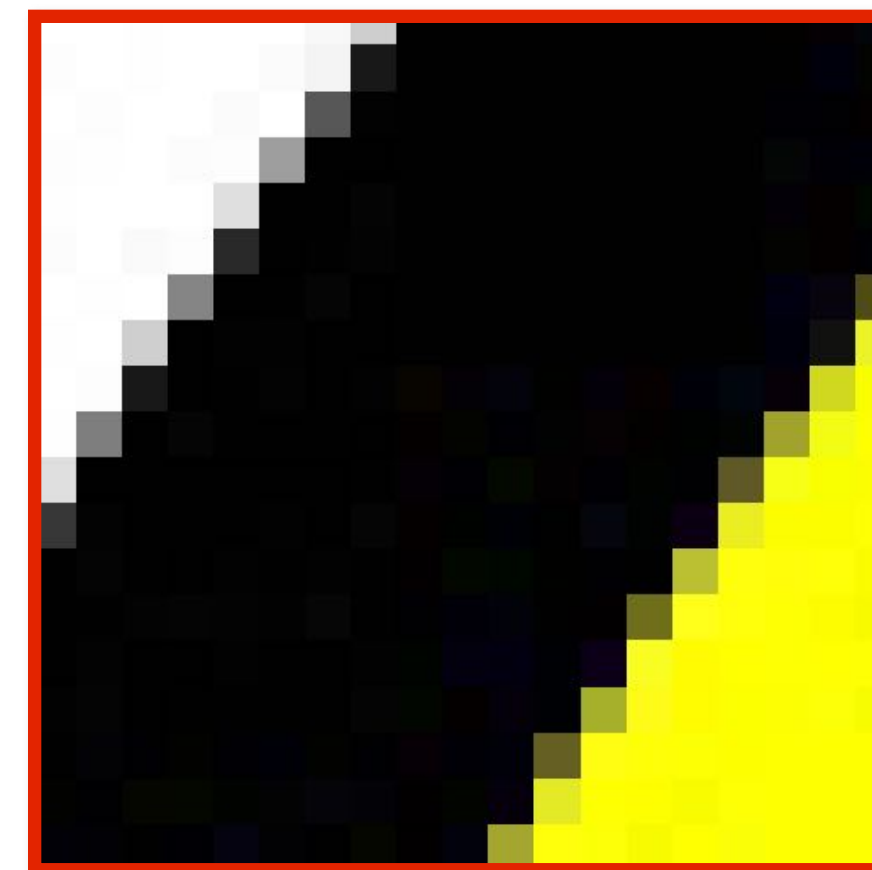
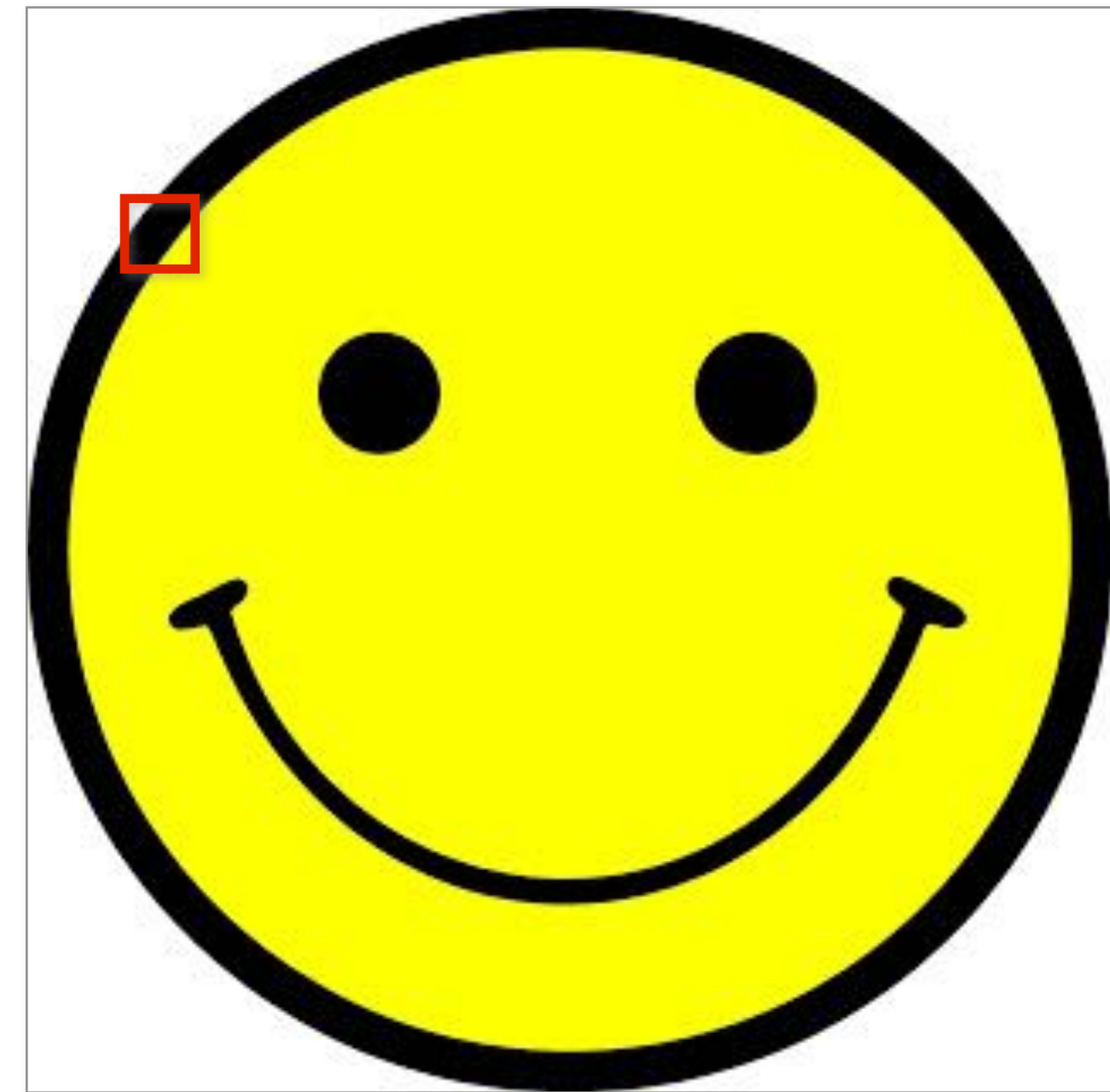


Image encoding in this example:

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:  $16 \times 3 = 48$  bytes)**
- **Can we do better?**



# S3TC (also 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
  - $\frac{1}{3}C_0 + \frac{2}{3}C_1$
  - $\frac{2}{3}C_0 + \frac{1}{3}C_1$

## ■ 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)

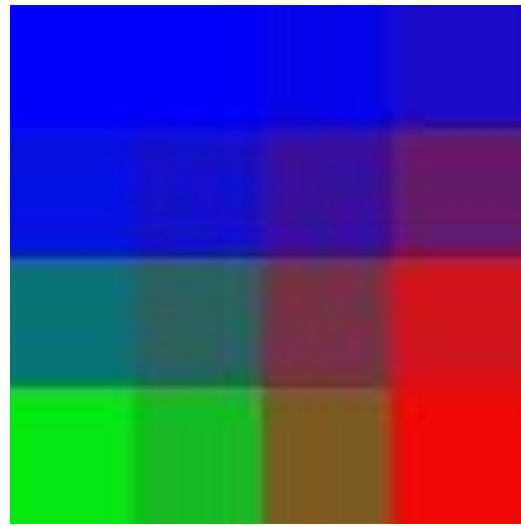
## ■ S3TC assumption:

- All texels in a 4x4 block lie on a line in RGB color space

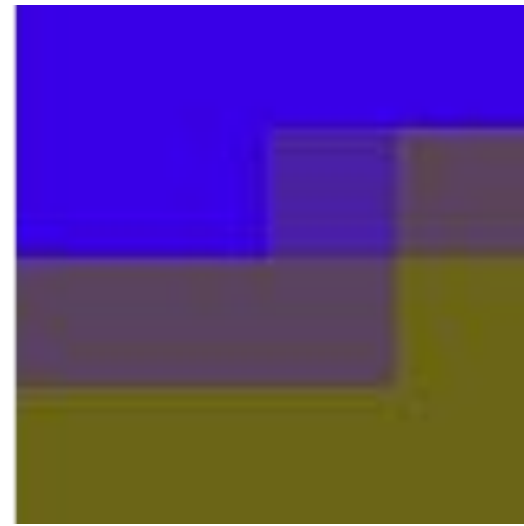
## ■ Additional mode:

- If  $C_0 < C_1$ , then third color is  $\frac{1}{2}C_0 + \frac{1}{2}C_1$  and fourth color is transparent black

# S3TC artifacts



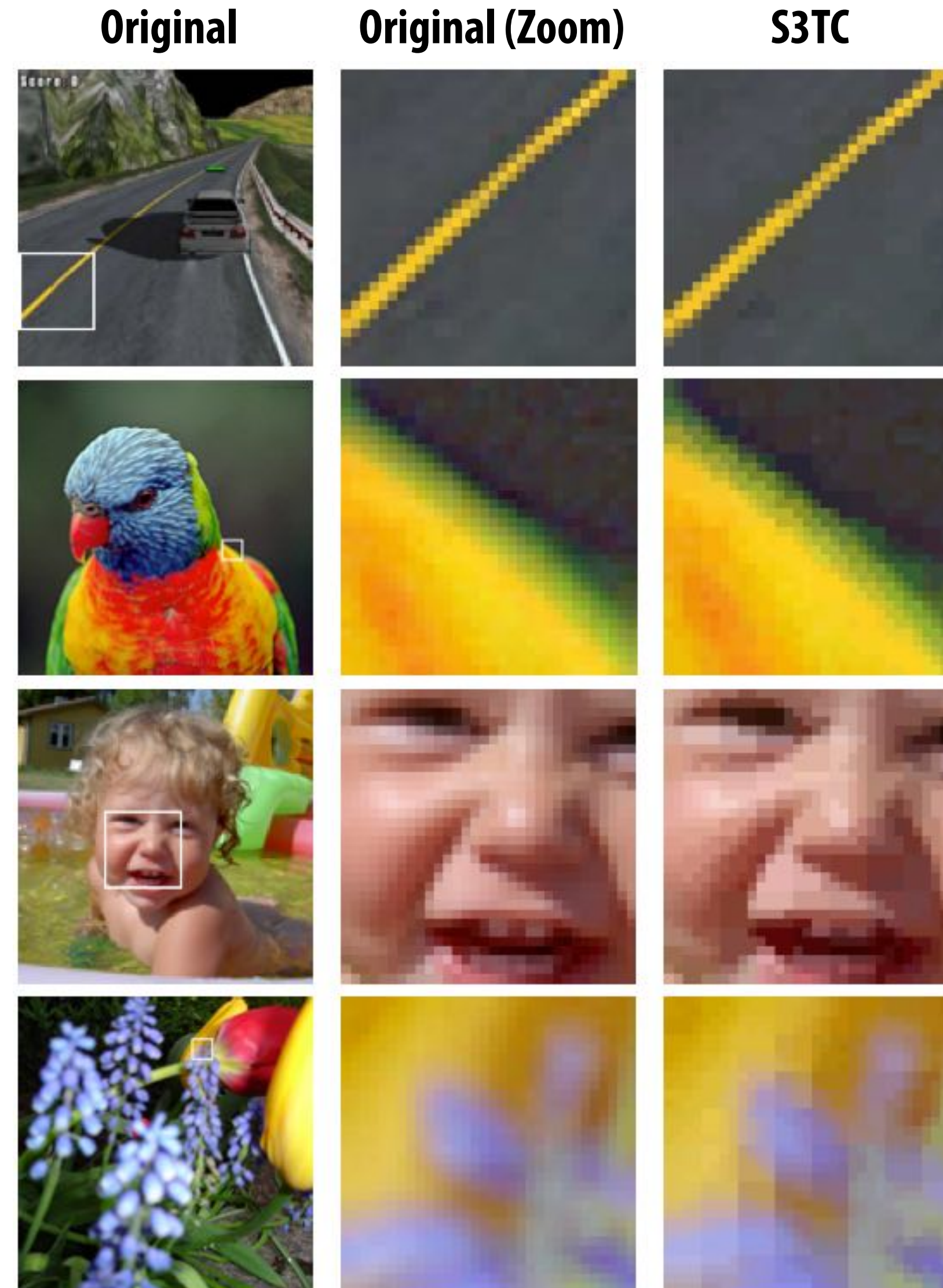
Original data



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)

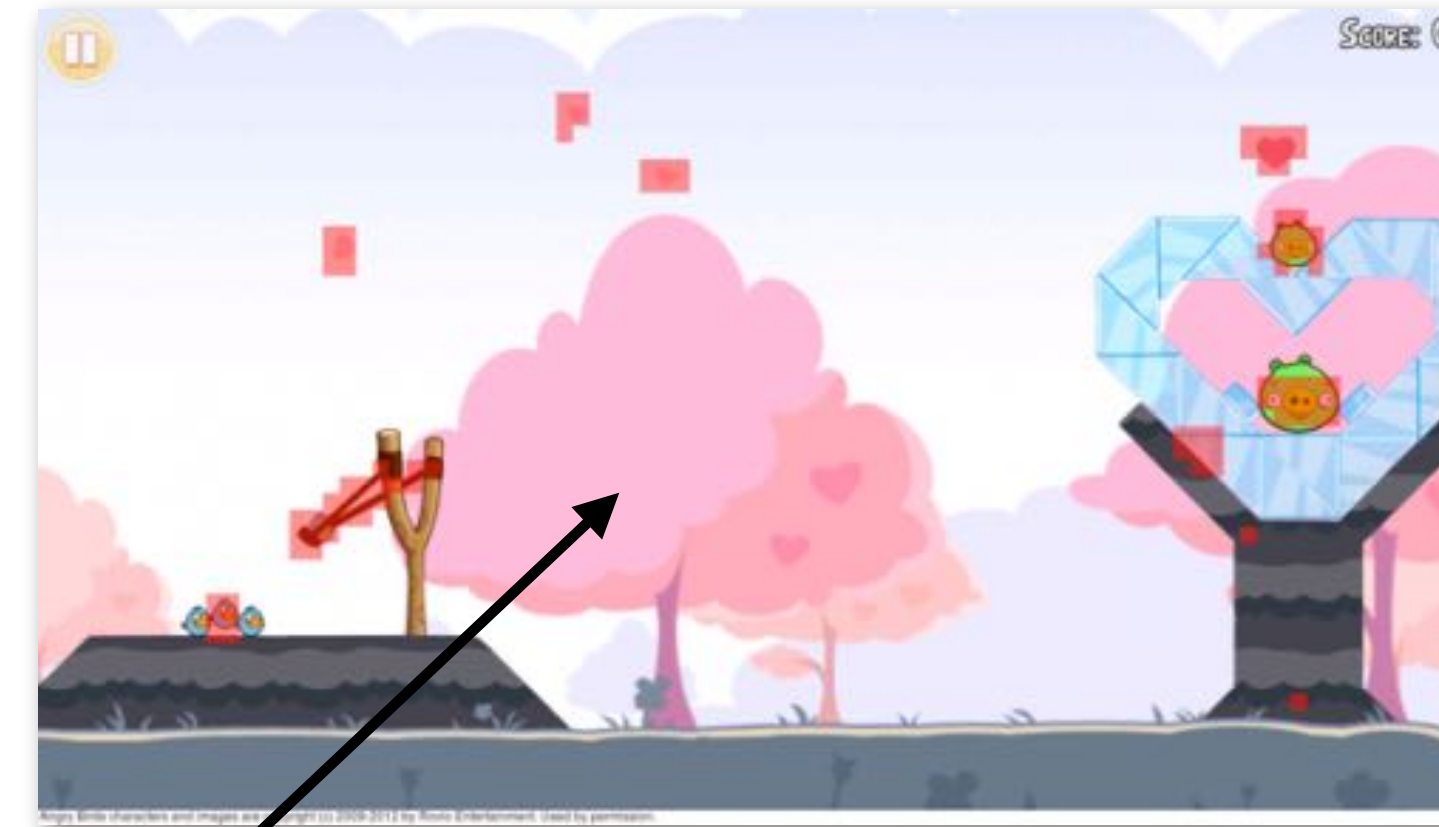


[Strom et al. 2007]



# Mobile GPU architects go to many steps to reduce bandwidth to save power

- Compress texture data
- Compress frame buffer
- Eliminate unnecessary memory writes!
  - Frame 1:
    - Render frame as normal
    - Compute hash of pixels in each tile on screen
  - Frame 2:
    - Render frame tile at a time
    - Before storing pixel values for tile to memory, compute hash and see if tile's contents are the same as in the last frame
      - If yes, skip memory write



Slow camera motion: 96% of writes avoided  
Fast camera motion: ~50% of writes avoided  
(red tile = required a memory write)

# Summary

- **3D graphics implementations are highly optimized for power efficiency**
  - **Tiled rendering for bandwidth efficiency \***
  - **Deferred rendering to reduce shading costs**
  - **Many additional optimizations such as buffer compression, eliminating unnecessary memory ops, etc.**
  
- **If you enjoy these topics, consider CS348K (Visual Computing Systems)**

\* Not all mobile GPUs use tiled rendering as described in this lecture.