GPU Architecture

cwallez@google.com

Who is this guy >_>

- Interned at the OpenGL driver team at NVIDIA
- Joined Google to work on Vulkan before it was called Vulkan
- Continued on WebGL and ANGLE
- Now WebGPU group chair and Chromium WebGPU lead

Structure of the presentation

- Where GPUs sit in the system
- GPU execution units and comparison to CPUs
- GPU memory particularities
- Immediate-mode vs. tile-based GPUs

Warnings

- This will describe an idealized GPU architecture
  - Mobile GPUs can be very different
  - Doesn’t match any specific desktop GPU
- Won’t talk about NVIDIA’s RTX or ML acceleration

Where GPUs fit in the system

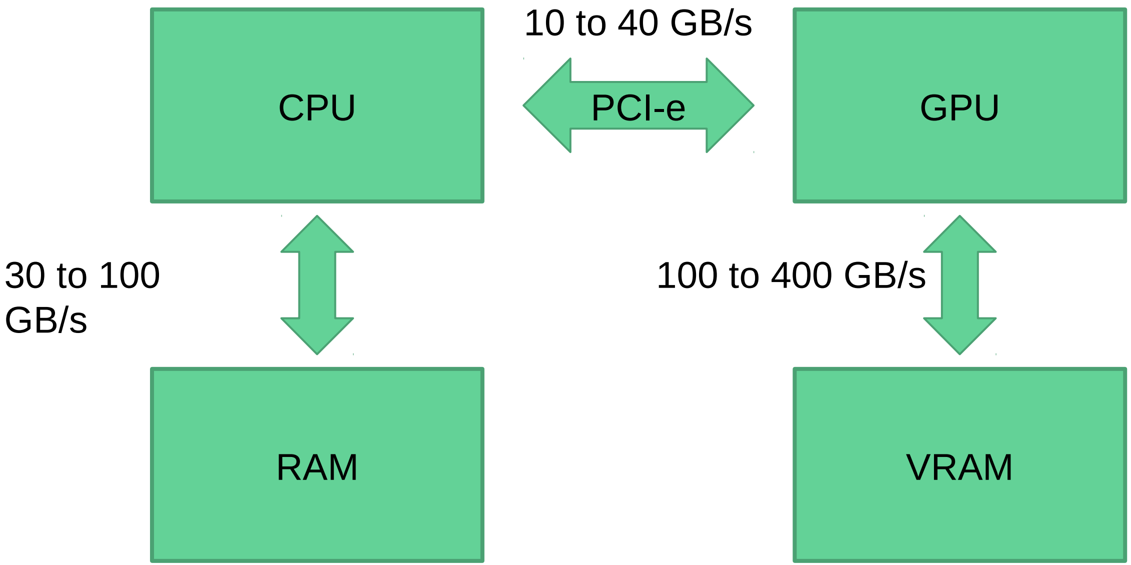
What’s a GPU?

- Graphics Processing Unit
- Asynchronous accelerator for graphics
- Great at very parallel problems, like scientific computing
- Contains a bunch of other graphics related functionality

Where can we find a GPU?

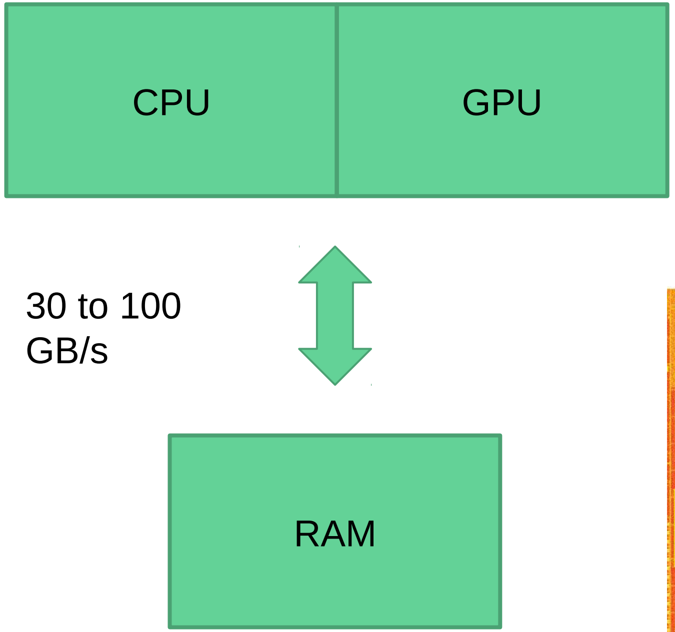
- Almost every consumer device with a screen
  - Desktop and laptop computers
  - Phones, tablets, smartwatches
  - Fridges, ...
- Server farms and supercomputers

Block diagram of computer with discrete GPU

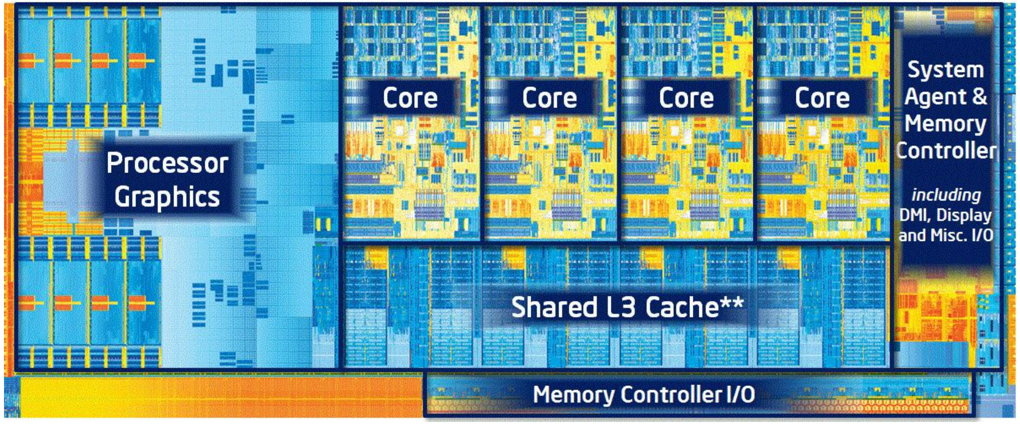


- Two separate chips
- Can see each other's memory through PCI-e
- Different RAM tradeoffs (throughput vs. latency)

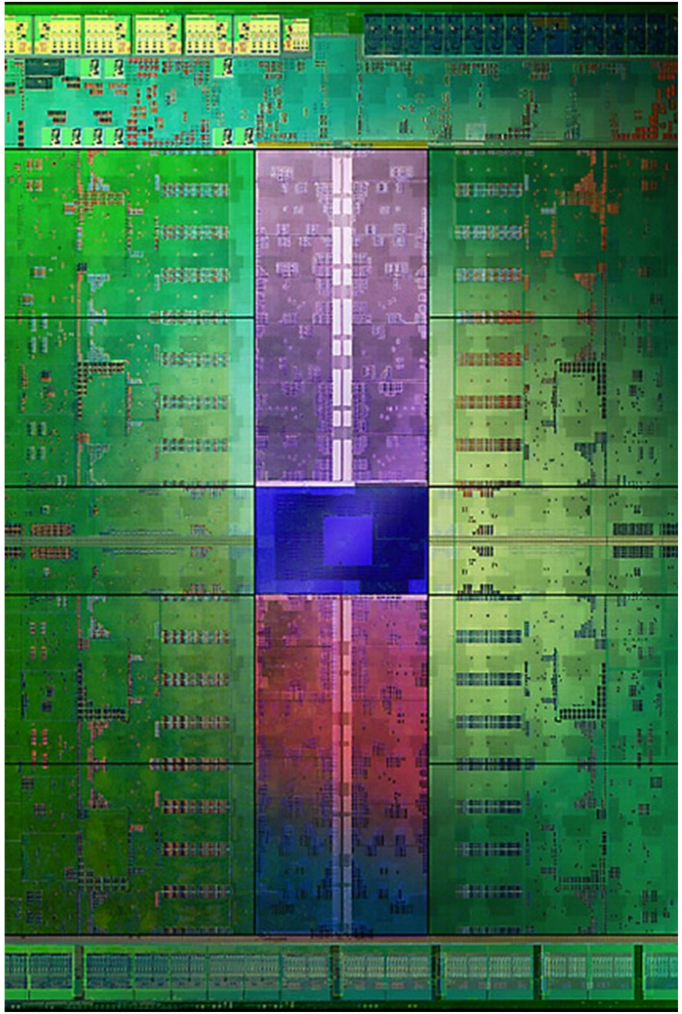
Block diagram of computer with integrated GPU



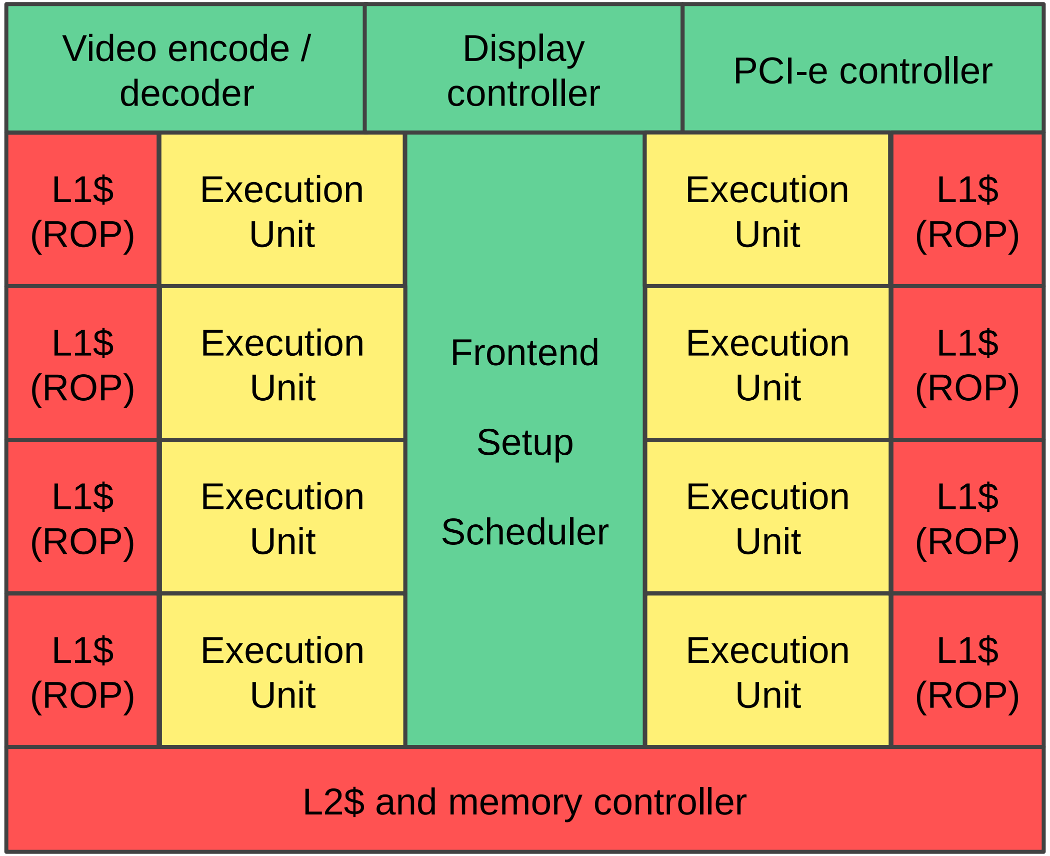
- Both on the same chip
- Usually share the last-level cache
- Example for Ivy Bridge:



Inside a GPU



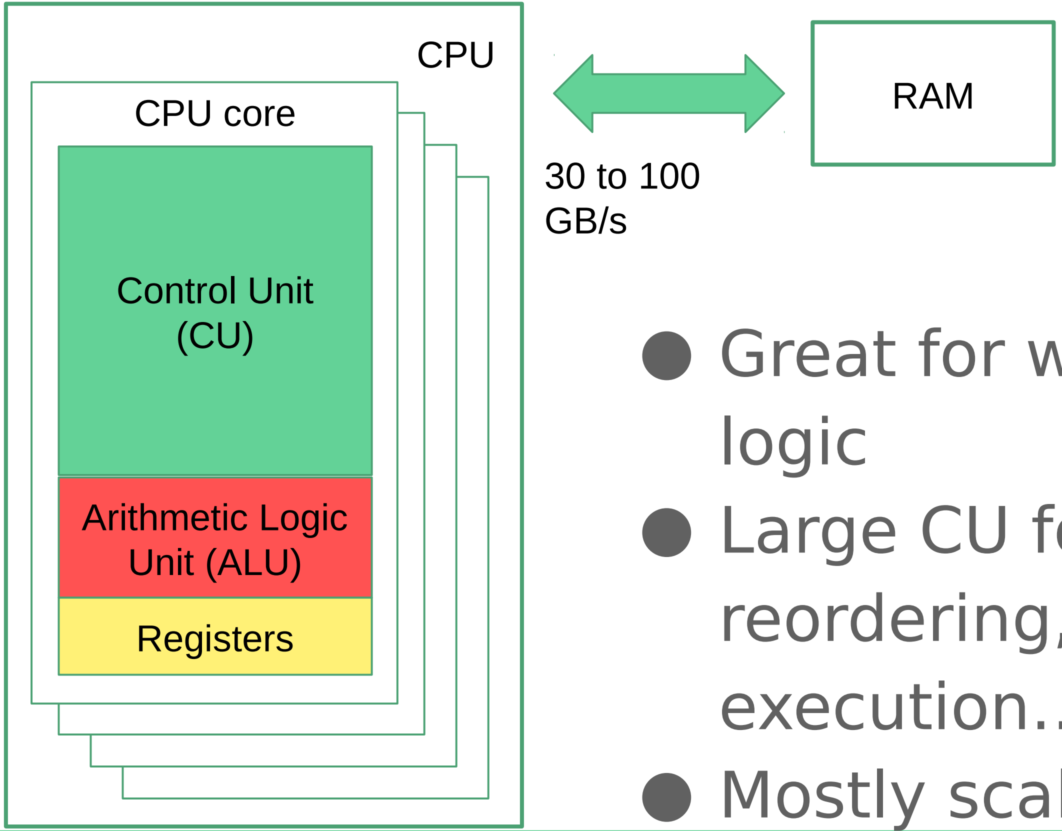
Inside a GPU



- Will later focus on this part:

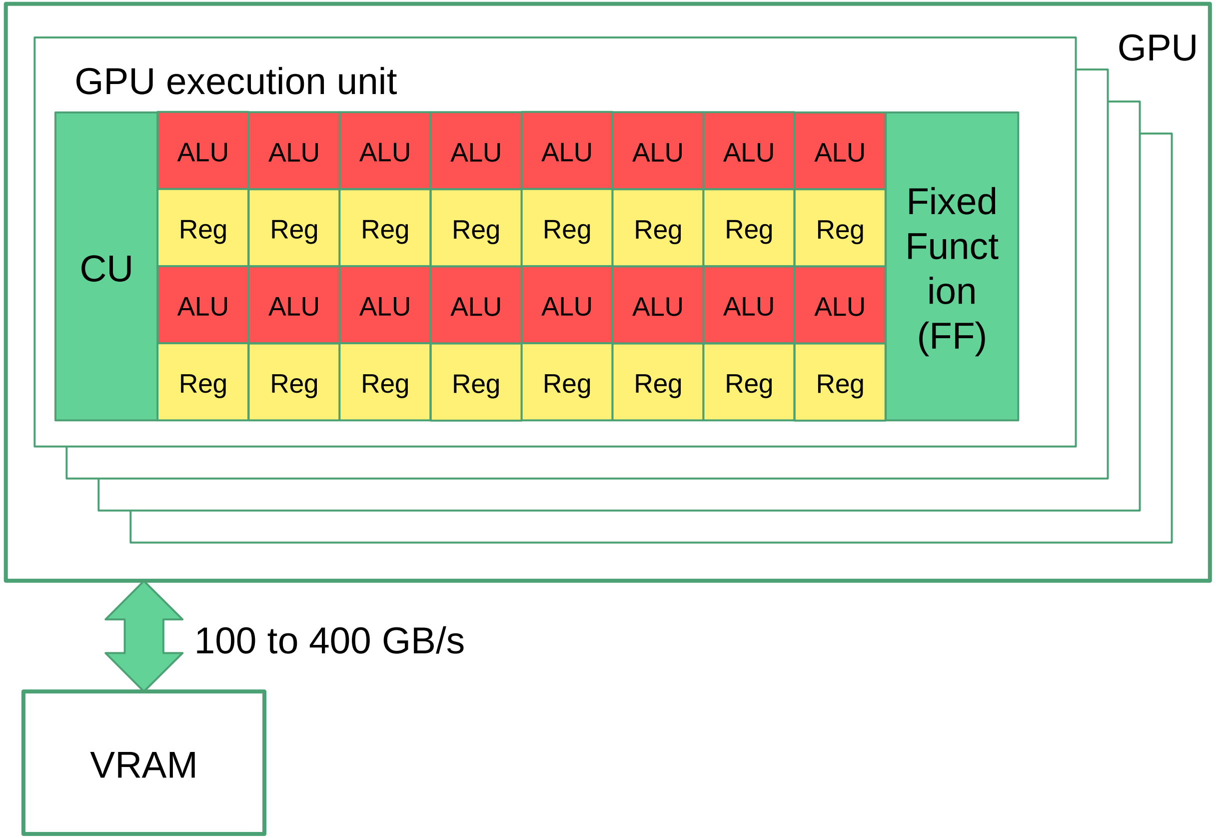
GPU execution units

CPU architecture



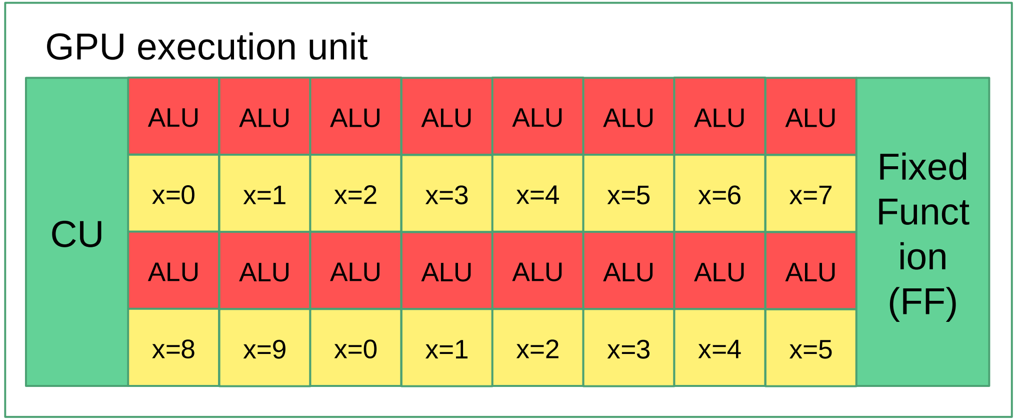
- Great for winding business logic
- Large CU for branch prediction, reordering, speculative execution...
- Mostly scalar ALUs and

GPU architecture



- Programmable chip
- ALUs share a CU
- They all perform the same instruction on different data (SIMD)
- Large memory throughput but larger latency.
- “Fixed function” units for functions expensive to run on ALUs (e.g.

How GPUs do conditions



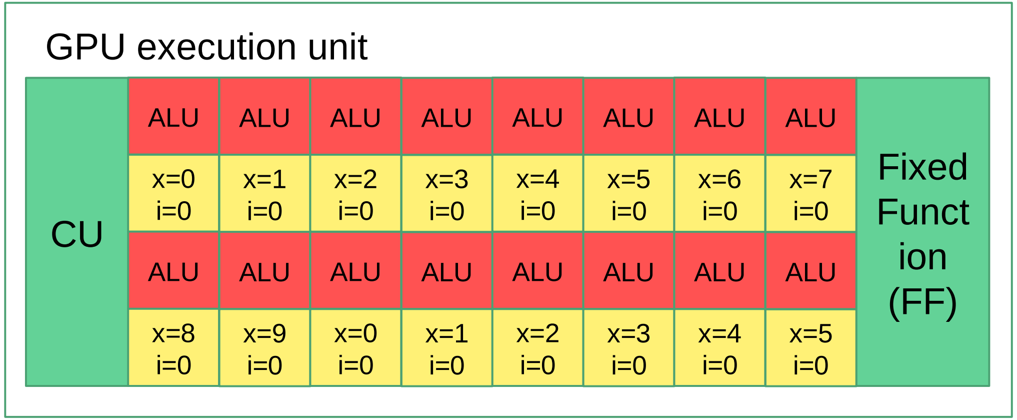

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



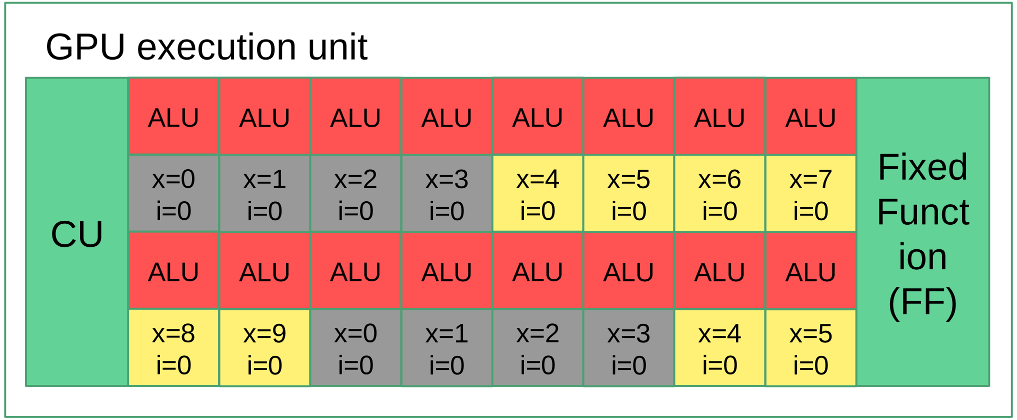

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



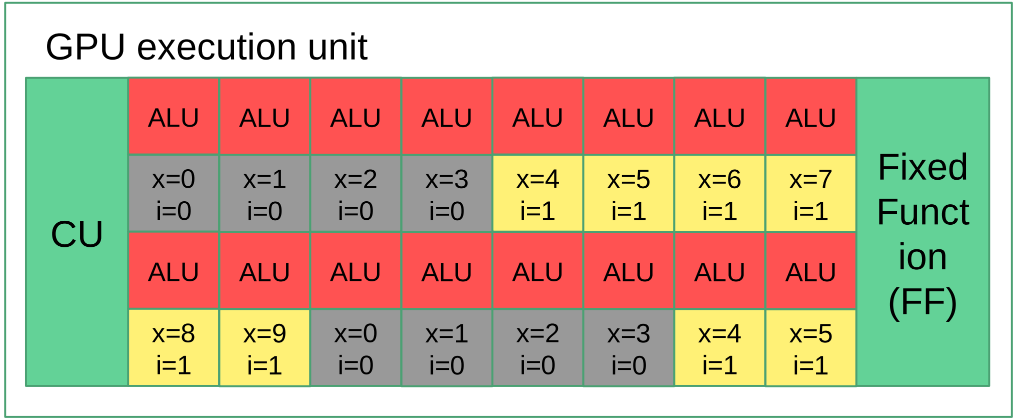

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



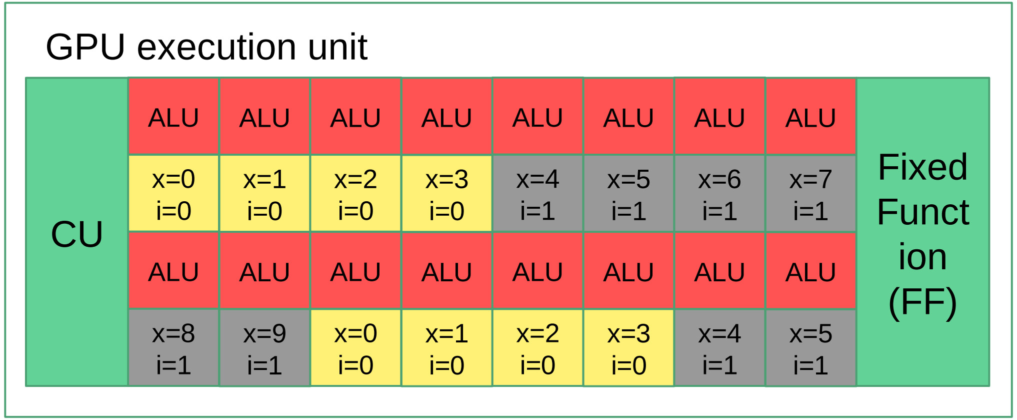

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



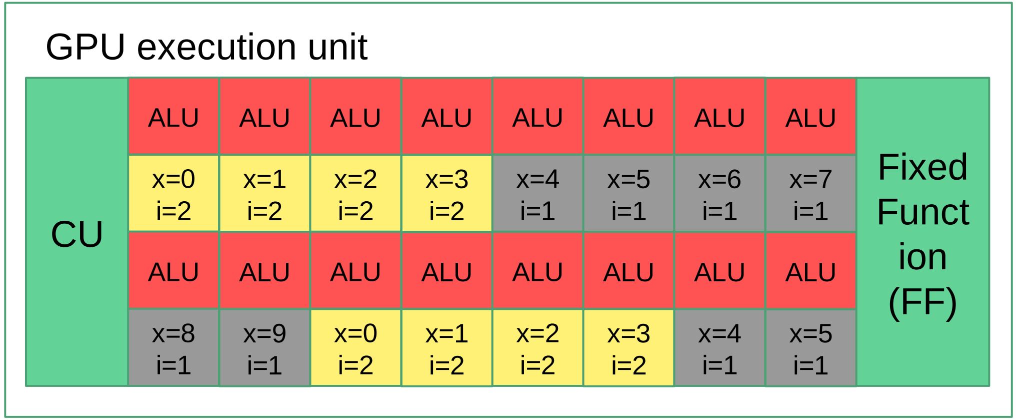

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



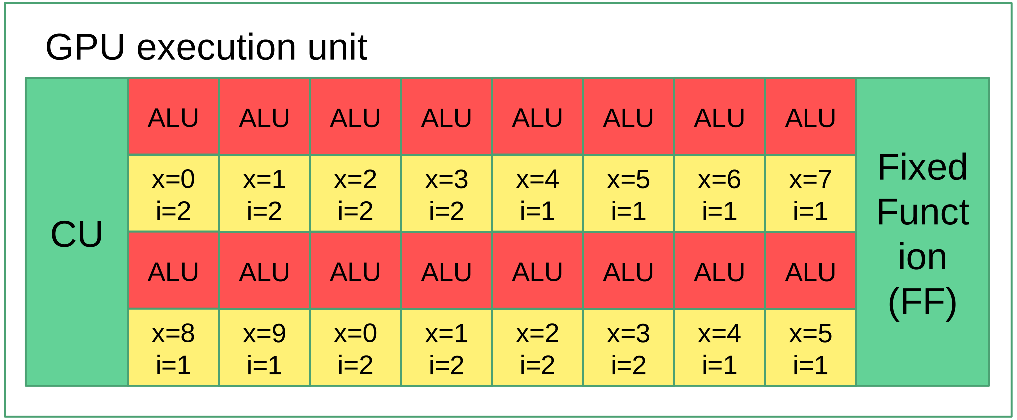

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do conditions



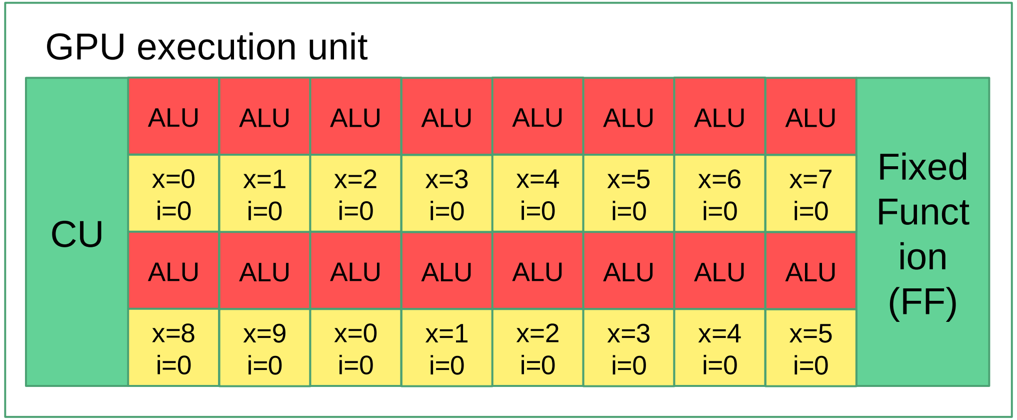

```
varying int x;

void main() {
    int i = 0;

    if (x > 3) {
        i = 1;
    } else {
        i = 2;
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



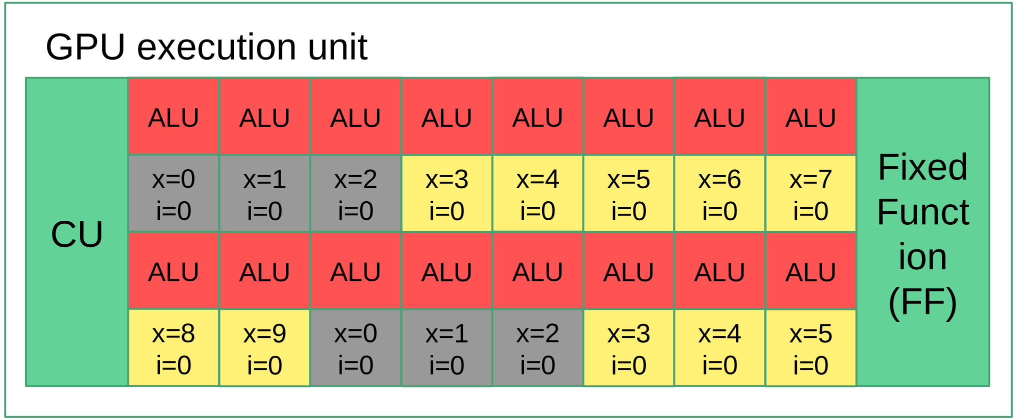

```
varying int x;

void main() {
    int i = 0;

    while (x > 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



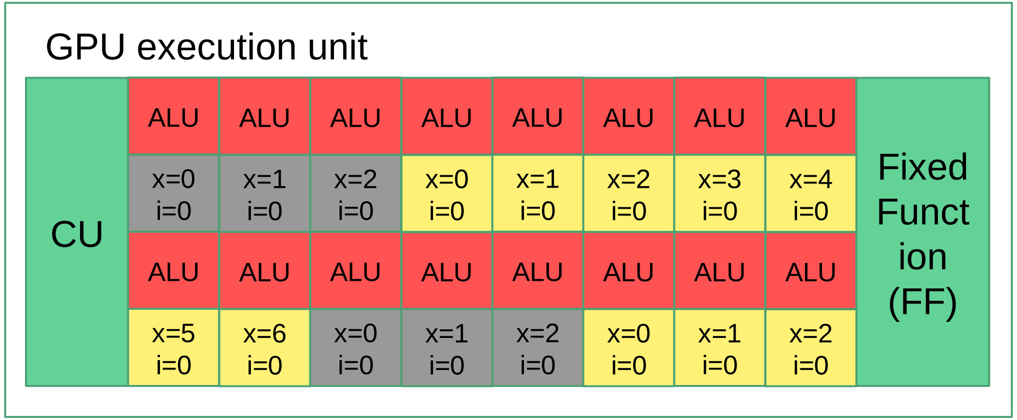

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



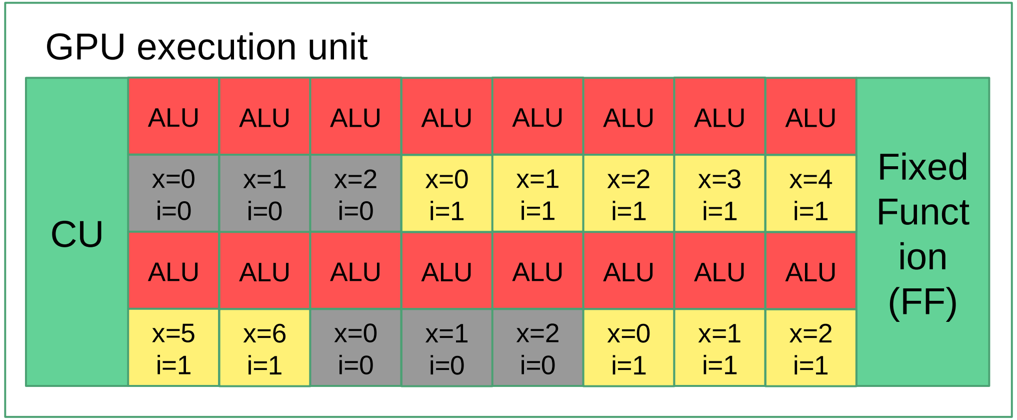

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



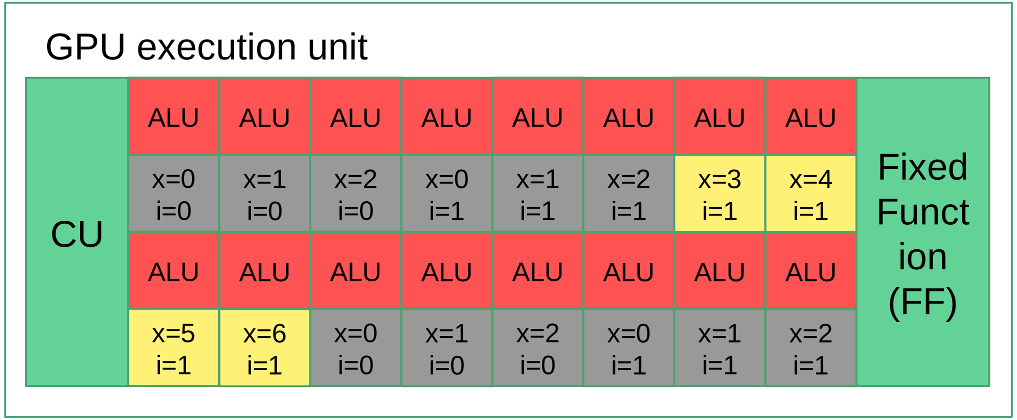

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



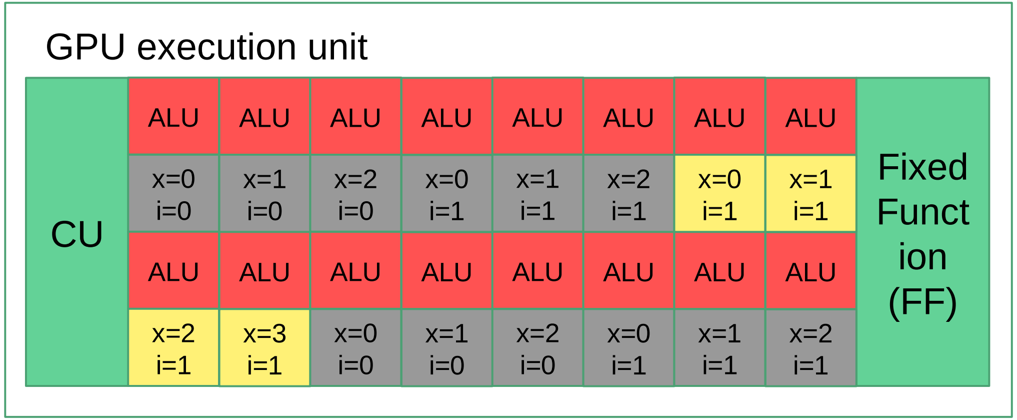

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



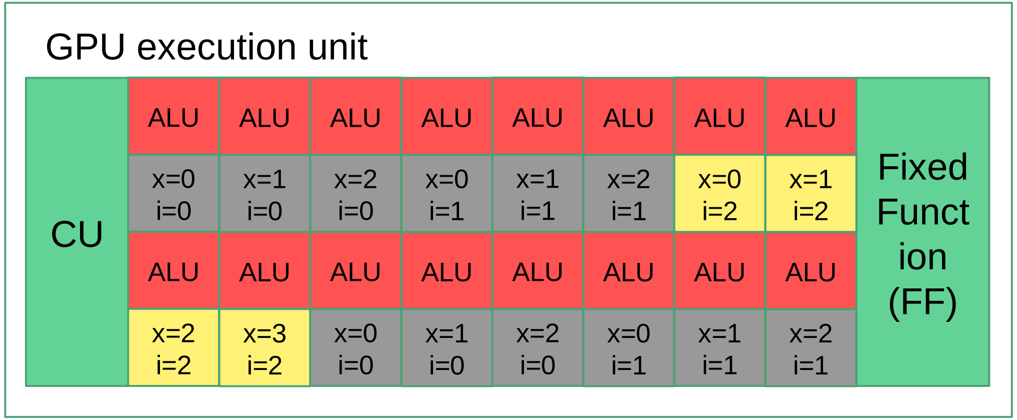

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



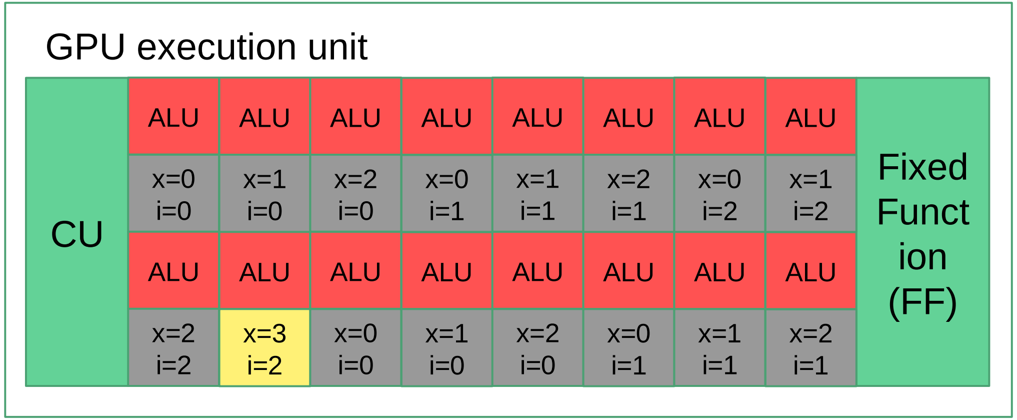

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



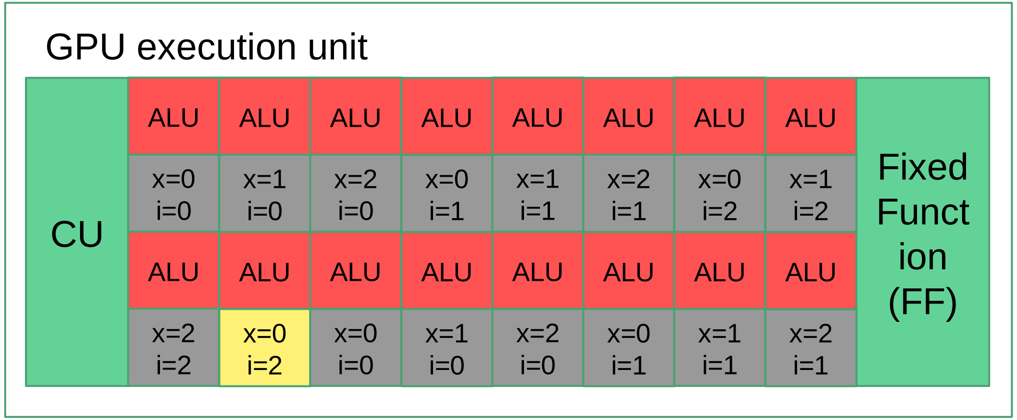

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



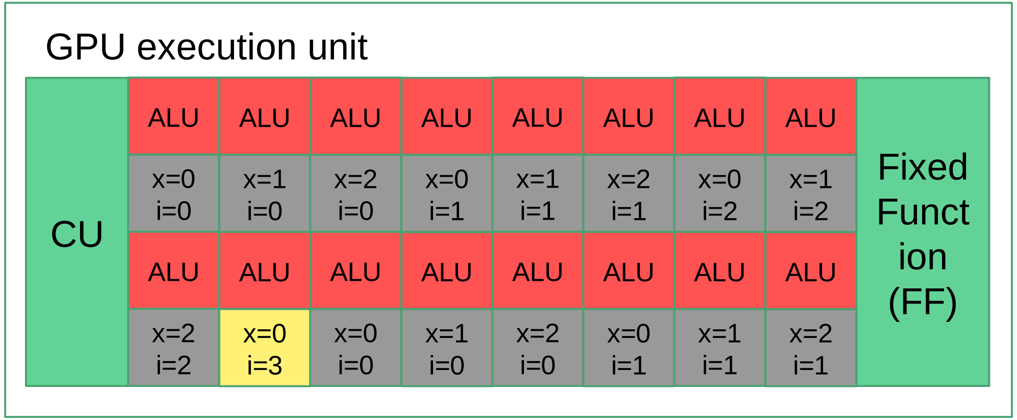

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



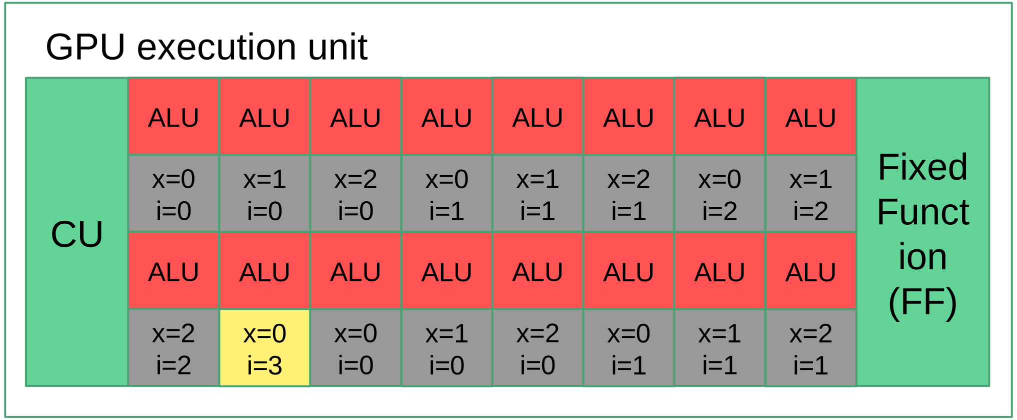

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops



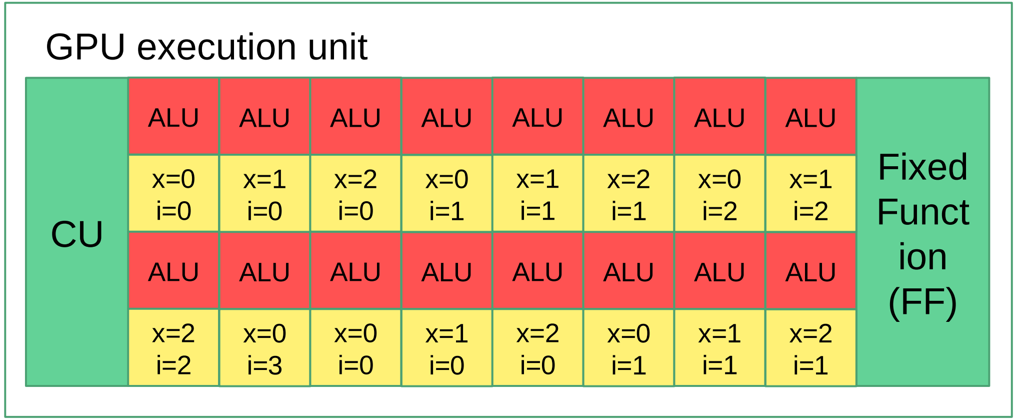

```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

How GPUs do loops




```
varying int x;

void main() {
    int i = 0;

    while (x >= 3) {
        x = max(0, x -
3);
        i++
    }


gl_FragColor(colors[i]);
}
```

Loop/branch takeaways

- If a single ALU take a branch, all ALUs will take it
- Branch coherency is important for performance
- Similar to AVX512 masked operations
    - They were designed for the Larrabee “GPU”

Memory latency hiding

- GPU trade memory latency to gain throughput
- An uncached access can take hundreds of cycles
- GPUs mitigate this with adaptive super "hyperthreading"

Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

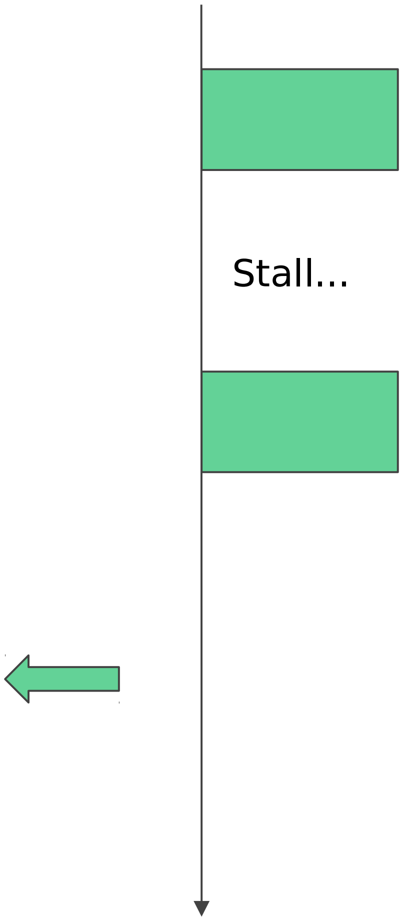
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



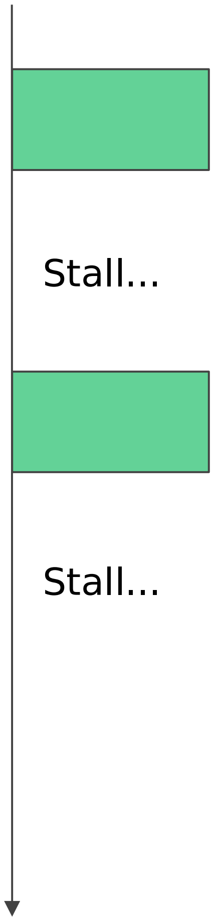
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



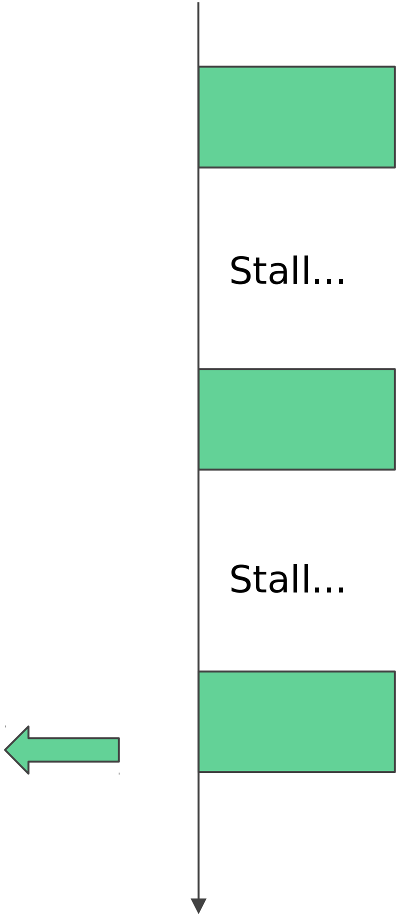
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



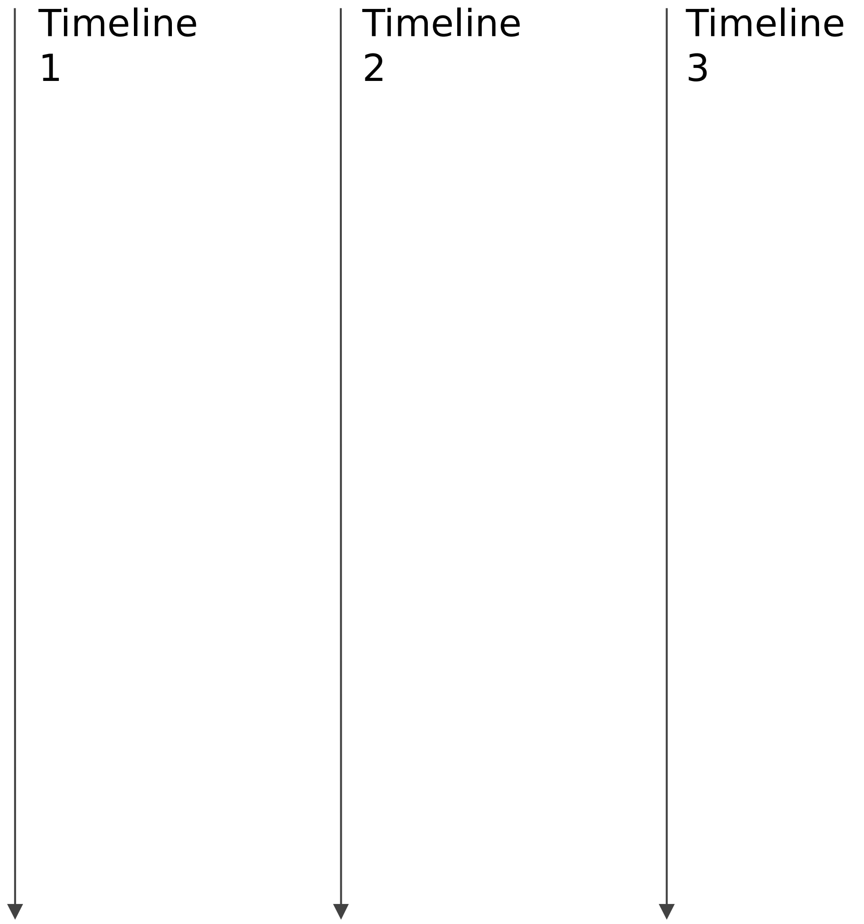
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



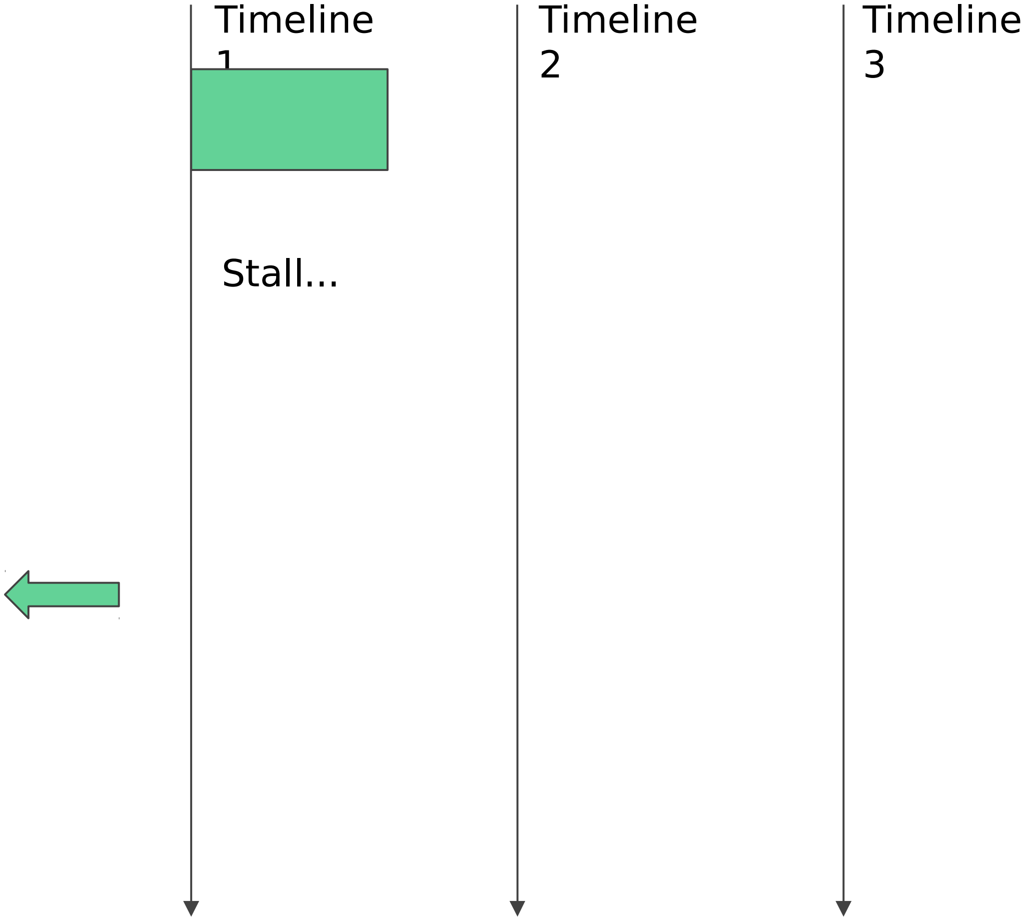
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



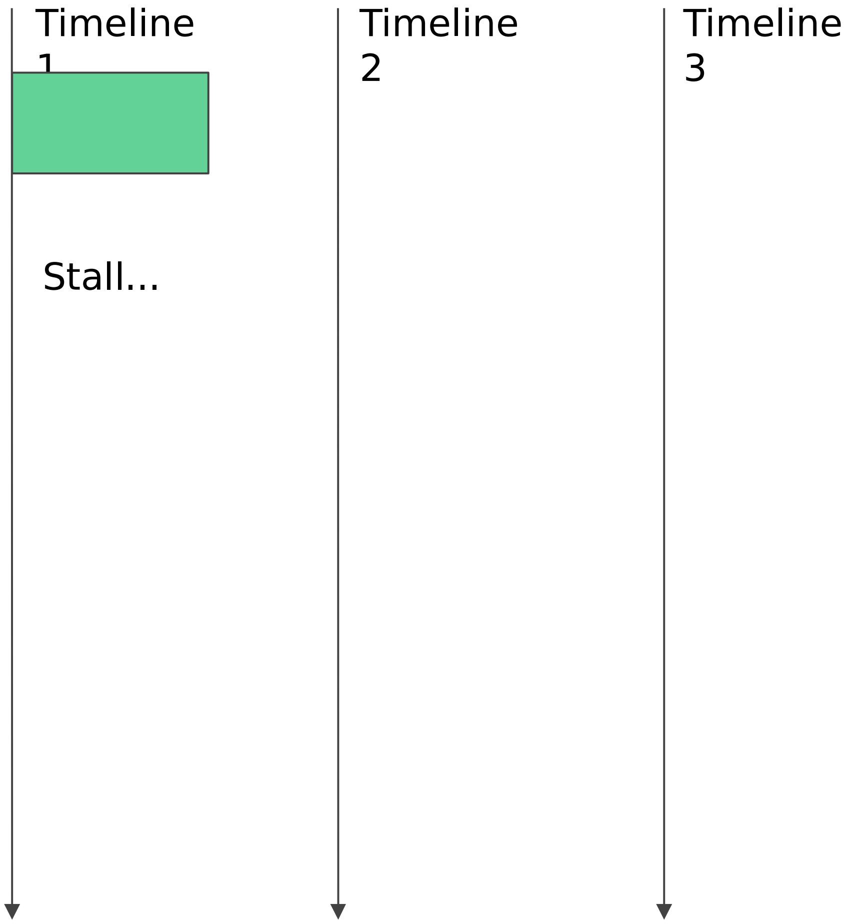
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



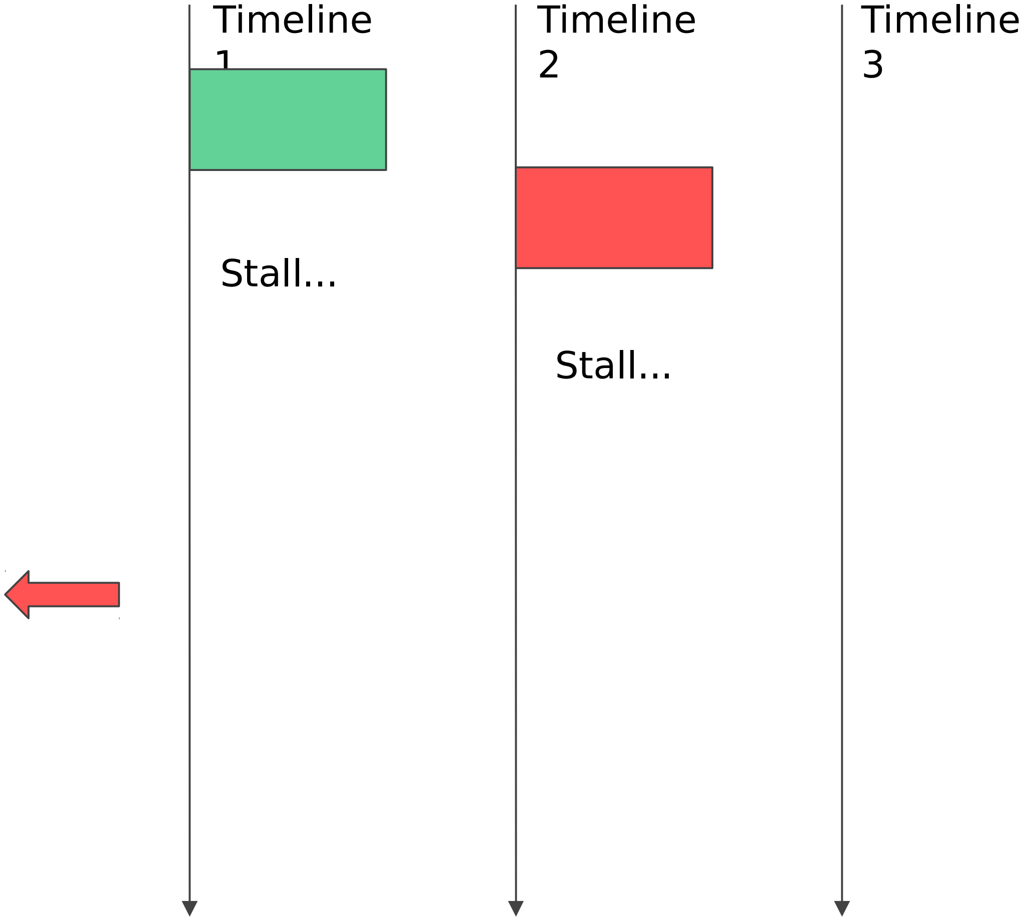
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



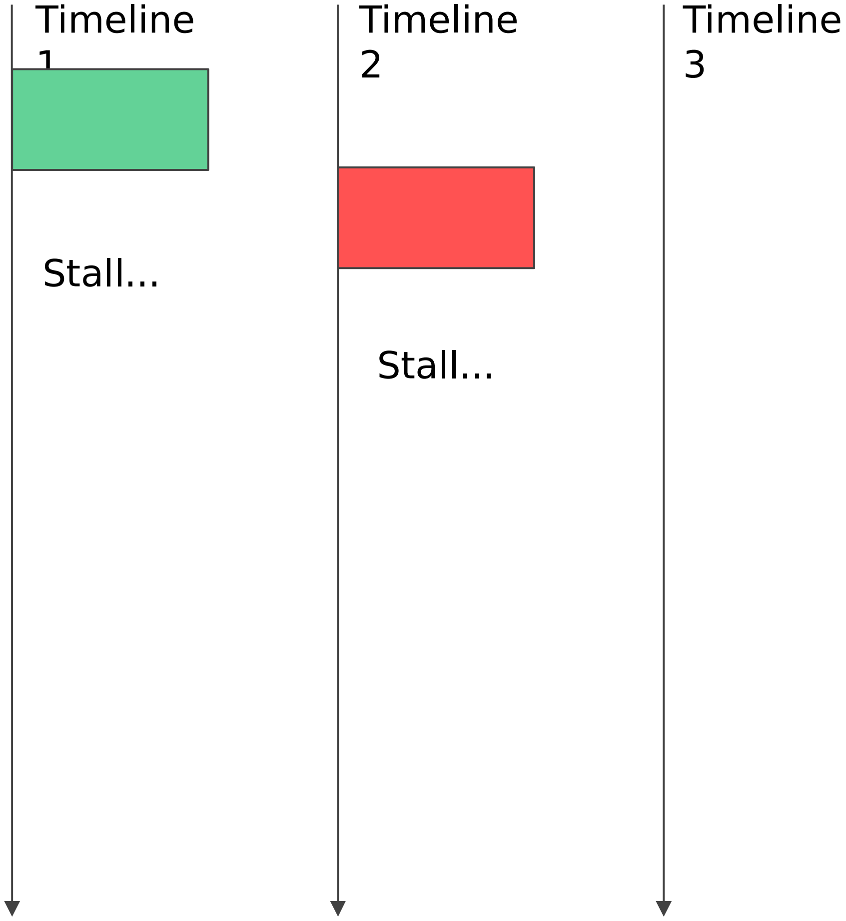
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



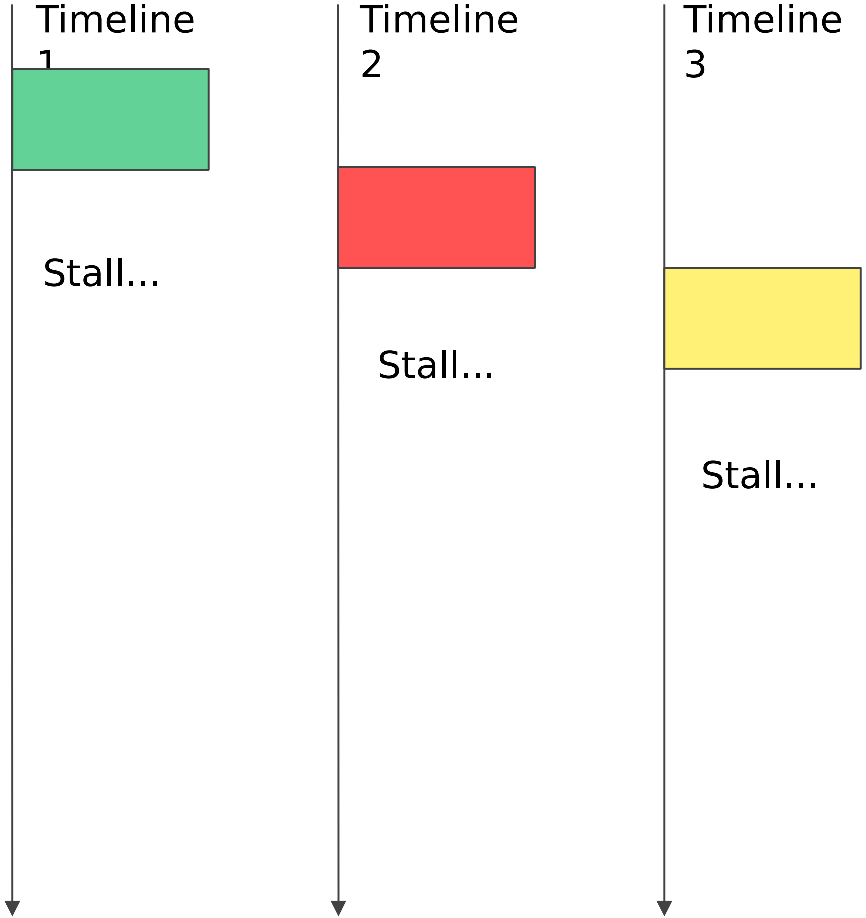
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



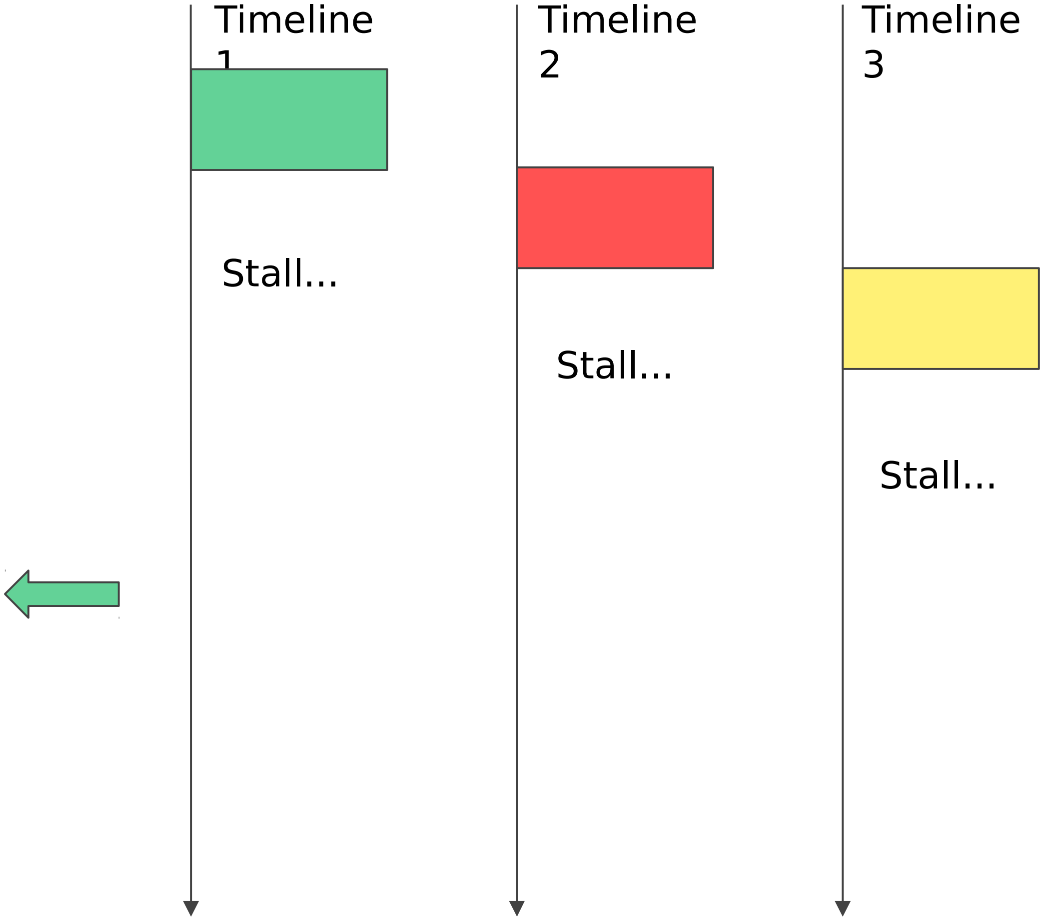
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

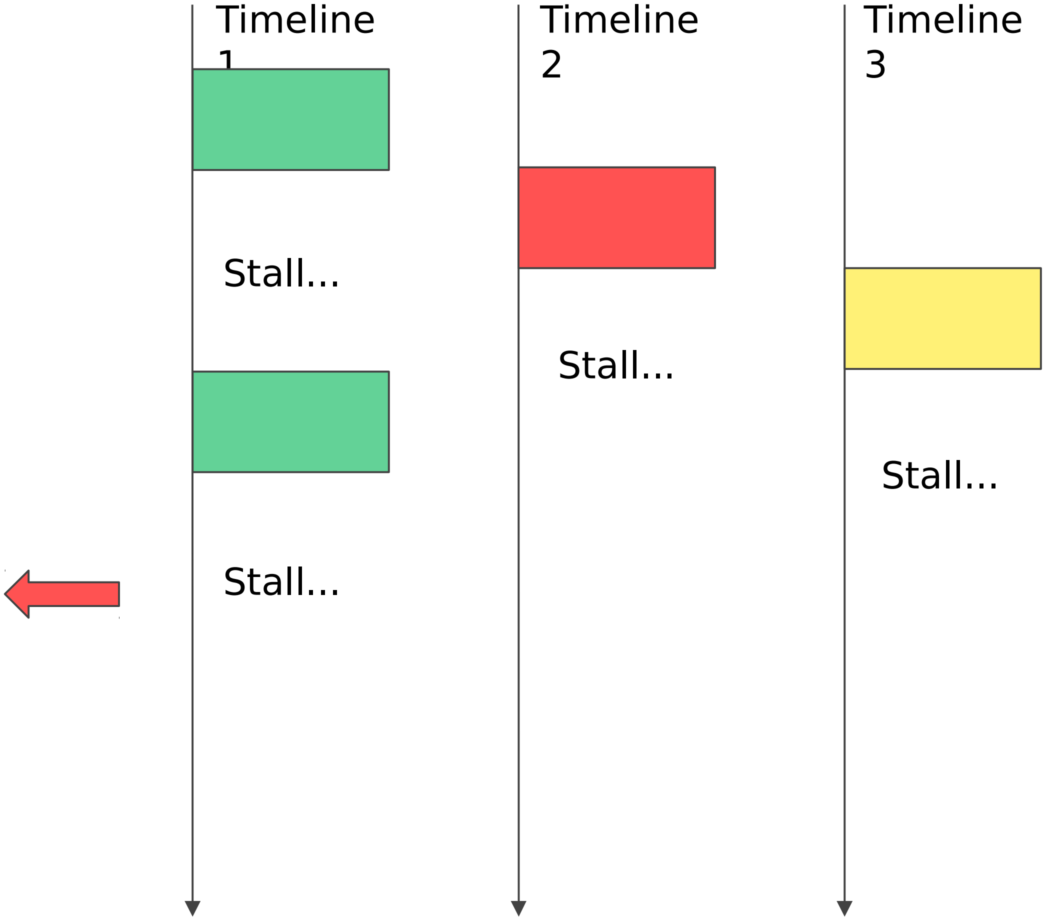
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

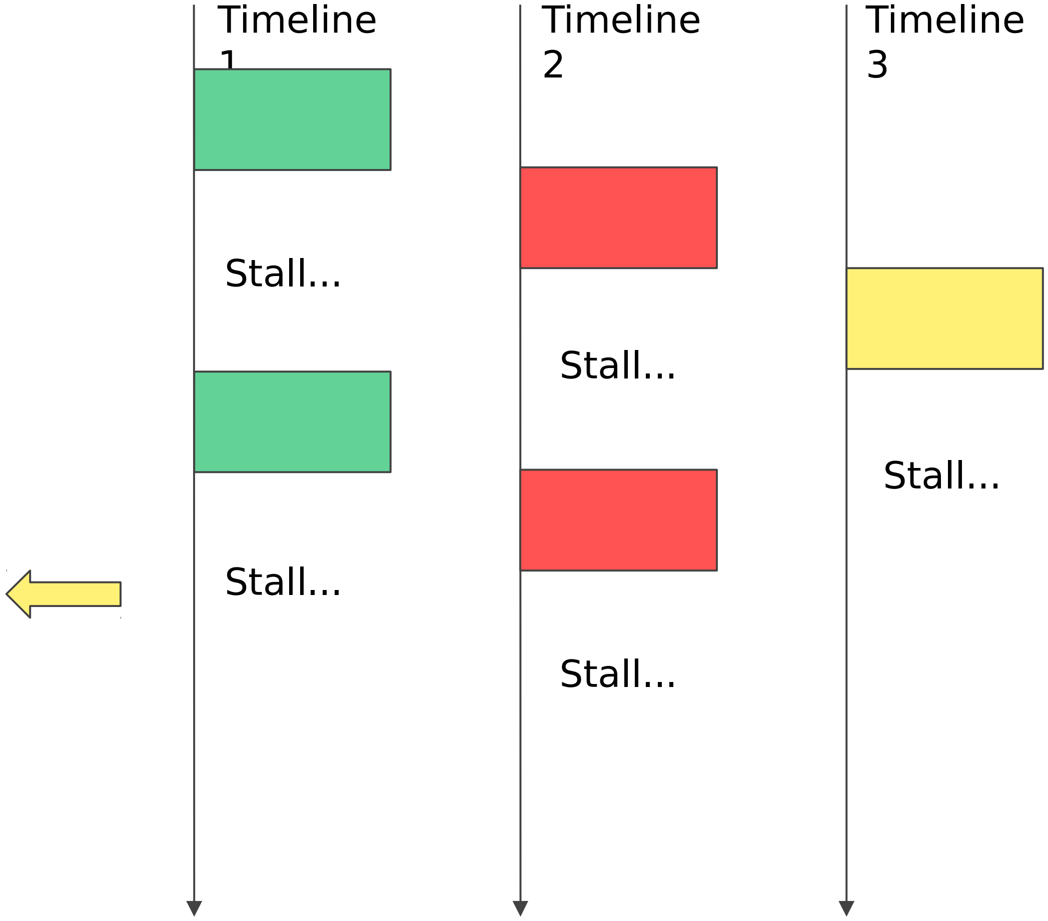
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

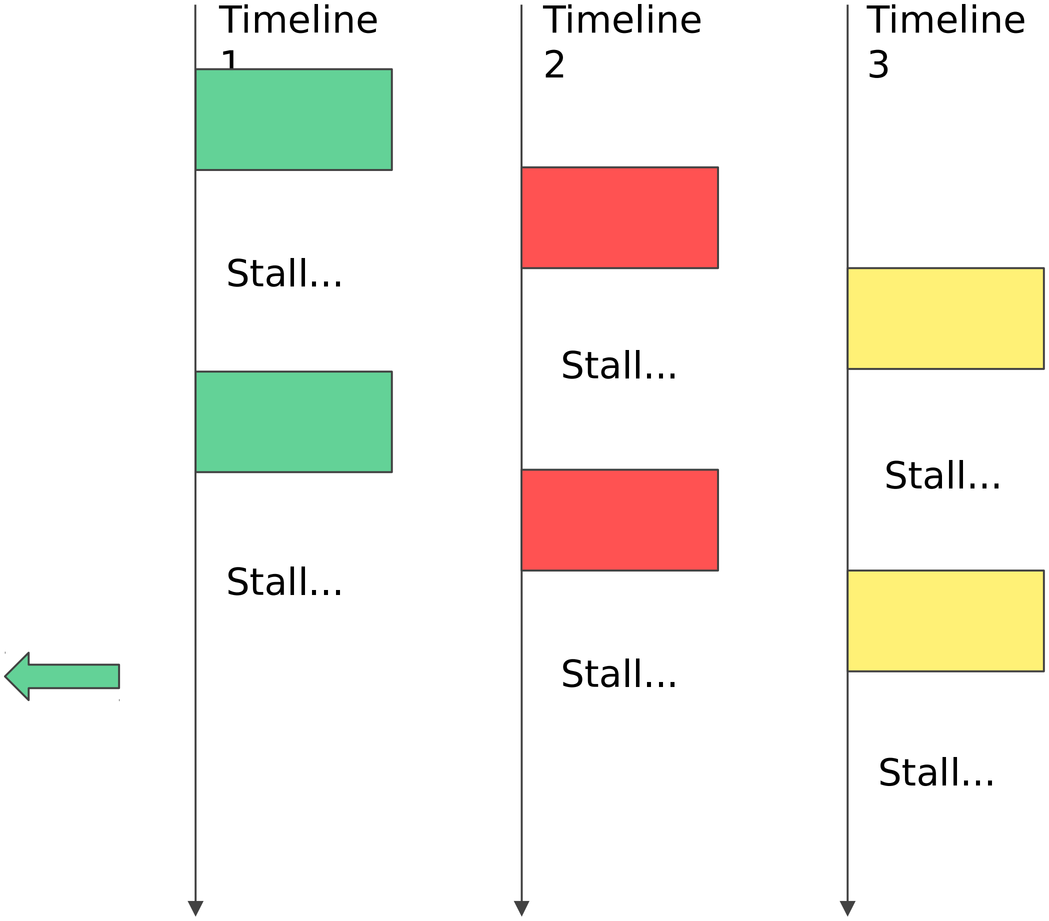
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



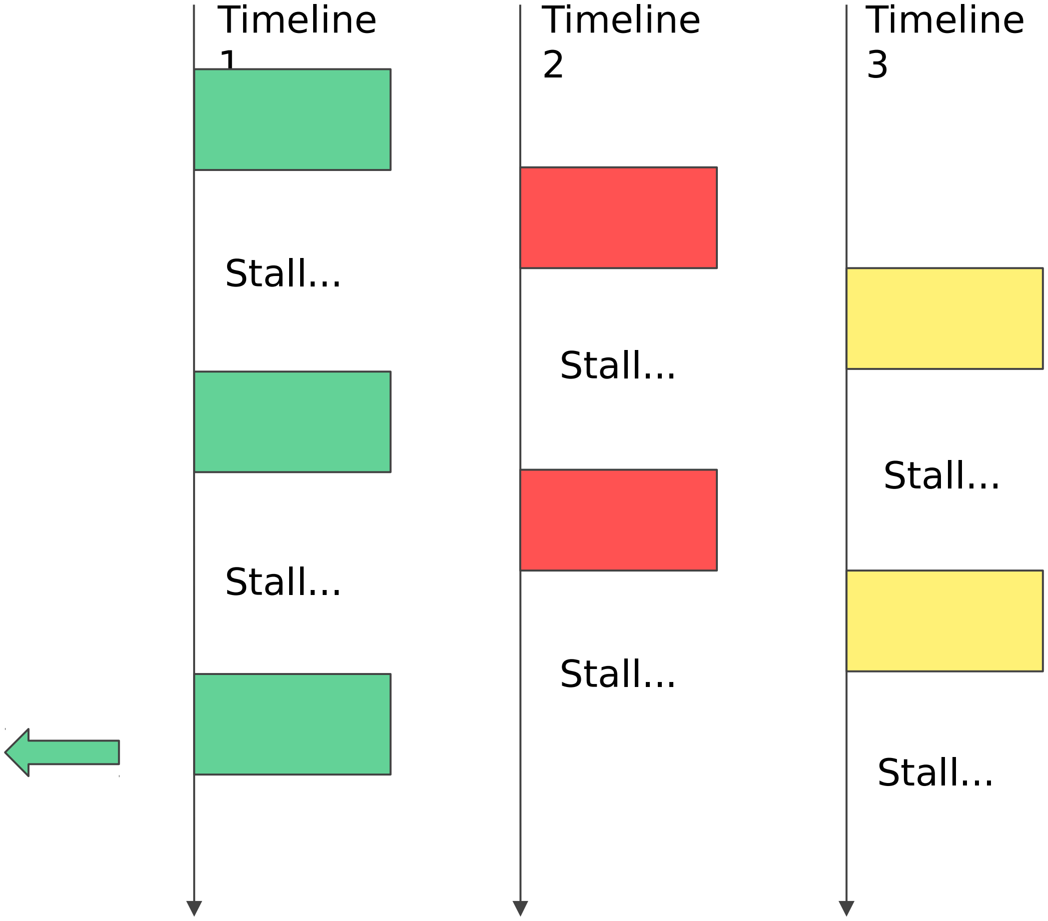
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



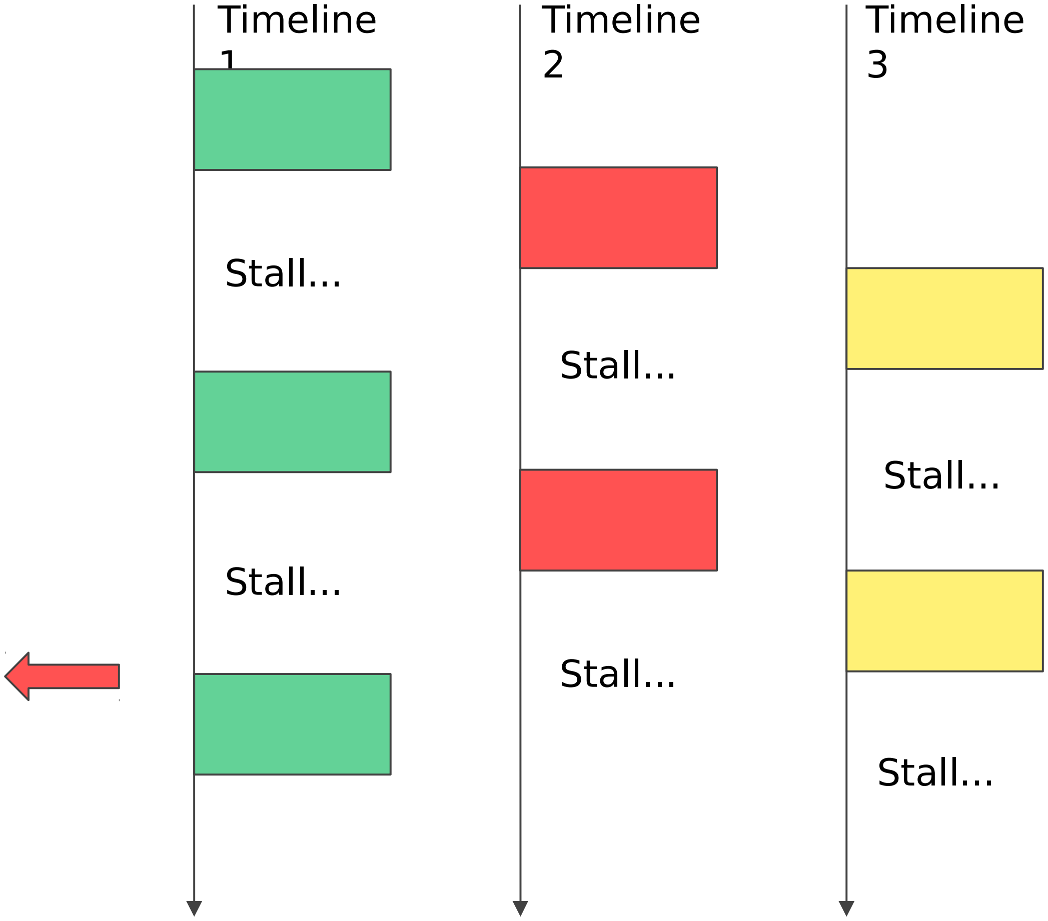
Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```



Memory latency hiding

```
varying vec2 coord;
uniform sampler2D color;
uniform sampler3D colorGradingMap;

void main() {
  vec3 albedo = texture(color,
coord);

  vec3 gradedColor =
    texture(colorGradingMap,
albedo);

  gl_FragColor.xyz = gradedColor;
}
```

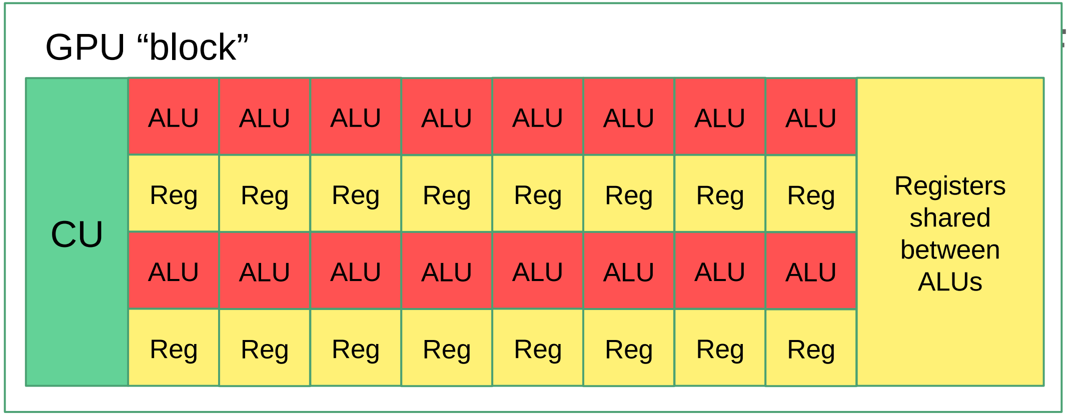
Memory latency hiding takeaways

- Register space is shared between multiple invocations
- The least register used, the more parallel invocations
- Example: AMD GCN has 1024 registers per ALU.
  - 32 registers used => 32 invocations, 100% utilization

Even more register powers ???!

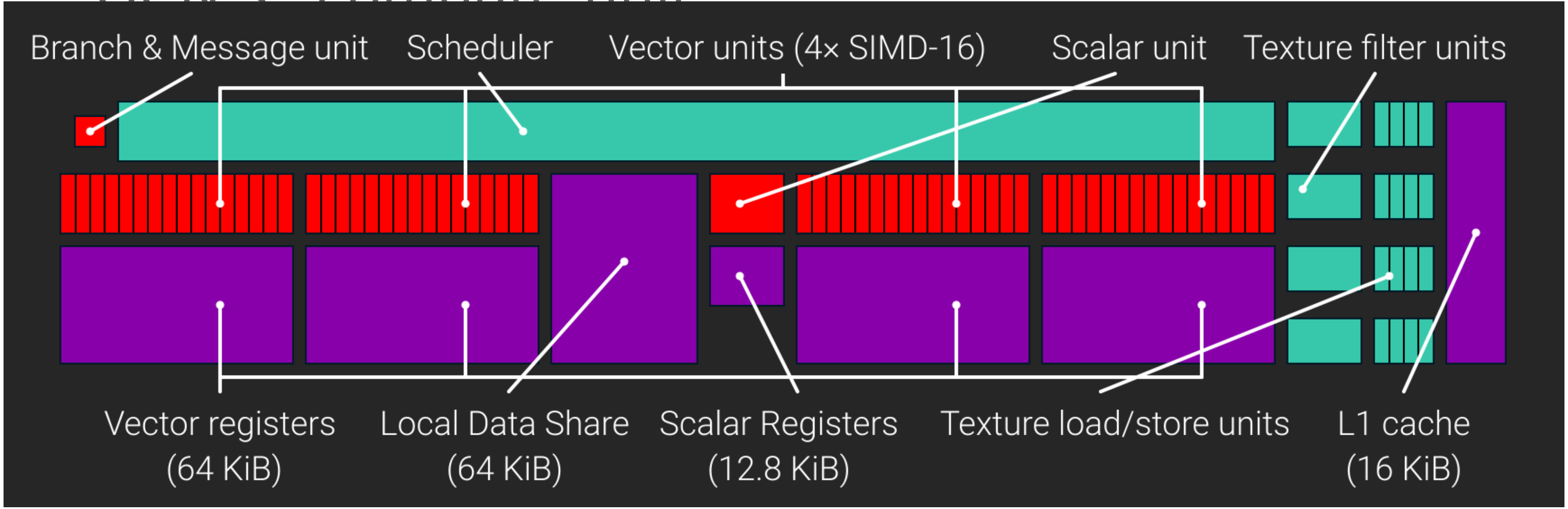
- Some registers are shared between ALUs
- Allows sharing memory loads and partial computations

“compute shaders”



Recap

All that was presented above is very close to AMD GCN's "compute unit"

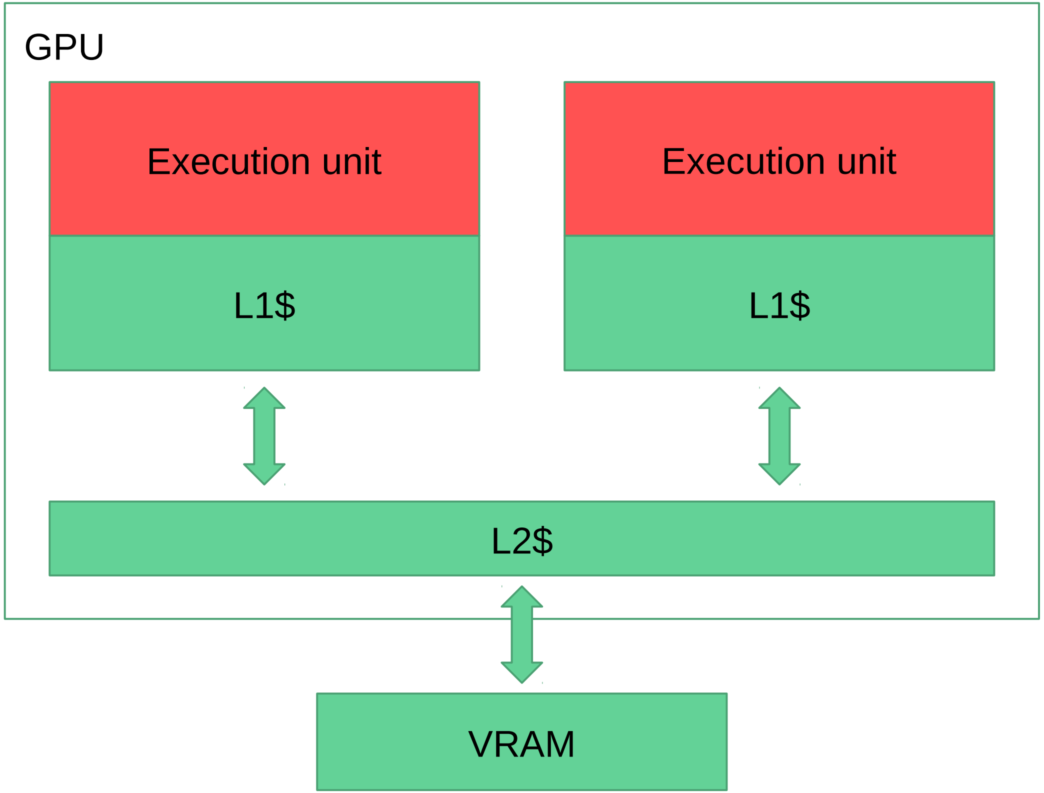


GPU memory particularities

GPUs are more diverse than CPUs

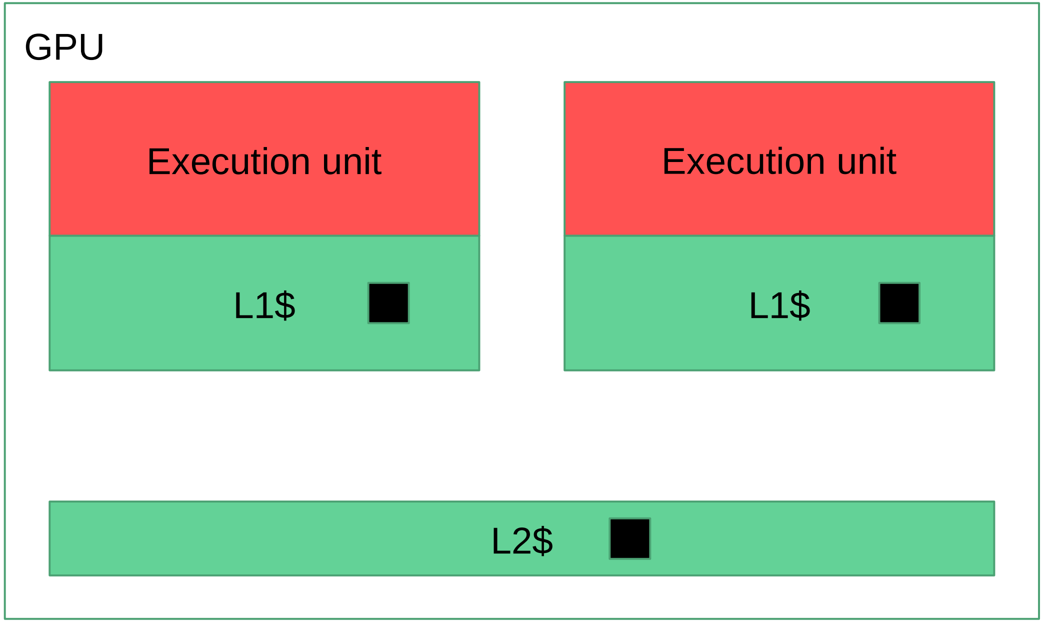
- GPUs are always accessed through a driver API
- This means that hardware details cannot be accessed directly and there is more variability in:
  - Instruction set
  - ALU geometry
  - Memory layout

GPUs have no cache coherency protocol



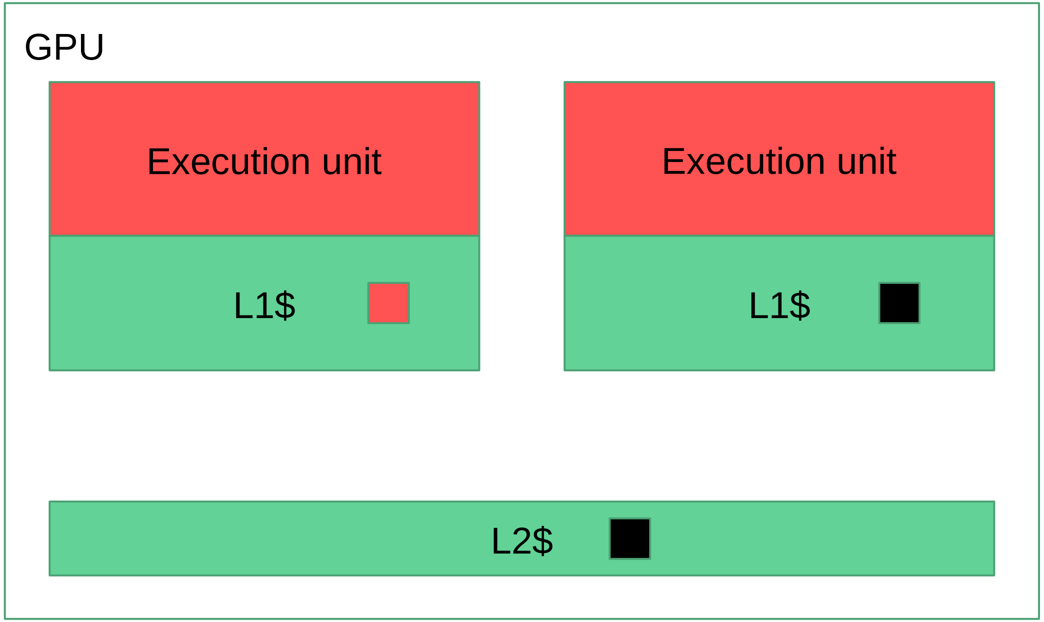
- Execution units don't talk directly to each other
- Always indirectly through L2$ / VRAM
- Cache coherency protocols are a waste of transistor / power.

GPUs have no cache coherency protocol



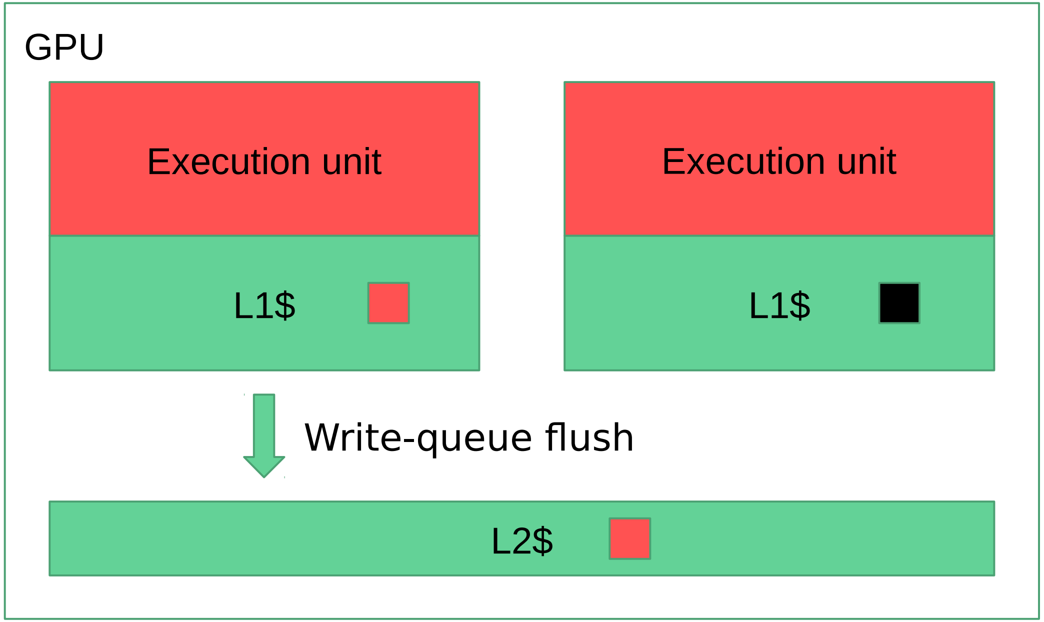
- Cache control is done by the driver "manually"
- Mix of flushing the write-queue and invalidating caches

GPUs have no cache coherency protocol



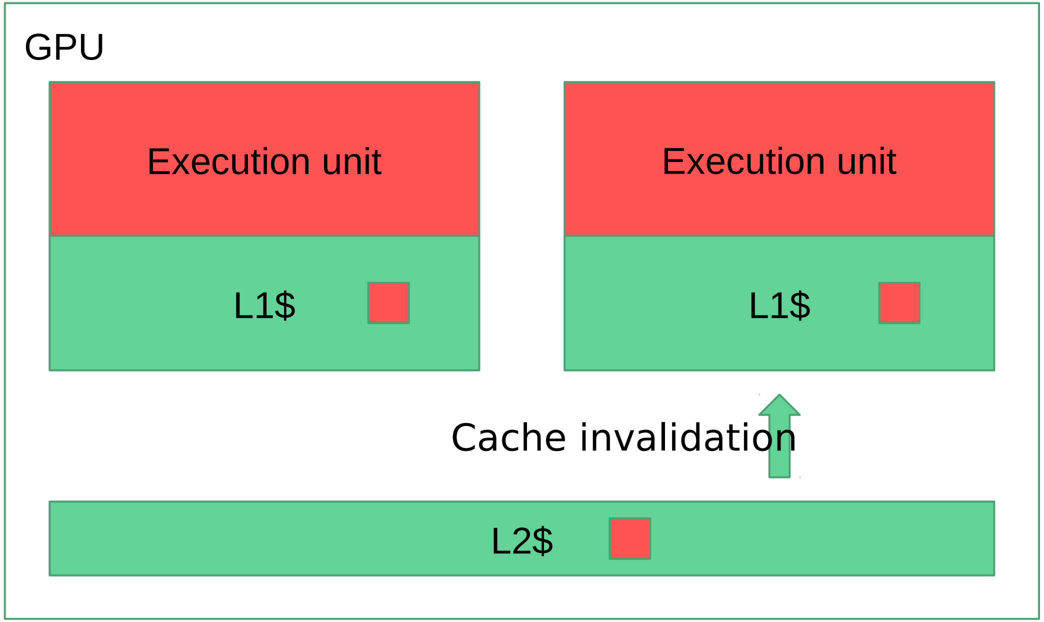
- Cache control is done by the driver "manually"
- Mix of flushing the write-queue and invalidating caches

GPUs have no cache coherency protocol



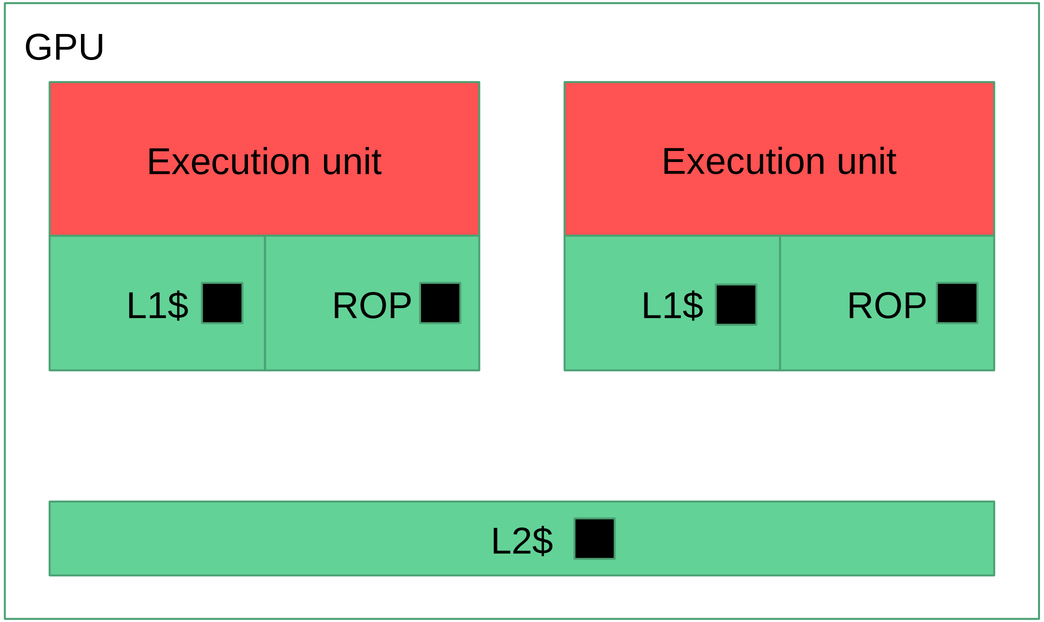
- Cache control is done by the driver “manually”
- Mix of flushing the write-queue and invalidating caches

GPUs have no cache coherency protocol



- Cache control is done by the driver “manually”
- Mix of flushing the write-queue and invalidating caches

GPUs can have multiple types of cache



GPUs can have a mix of caches among:

- Constant cache
- R/W cache
- Texture cache
- ROP (render output) cache

Other synchronization issues

- GPUs have no implicit synchronization between EUs
- GPUs don’t have pre-emption (too many registers to store)
- Again, everything is done by the driver manually

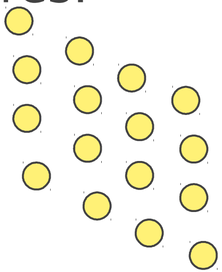
Layout of textures

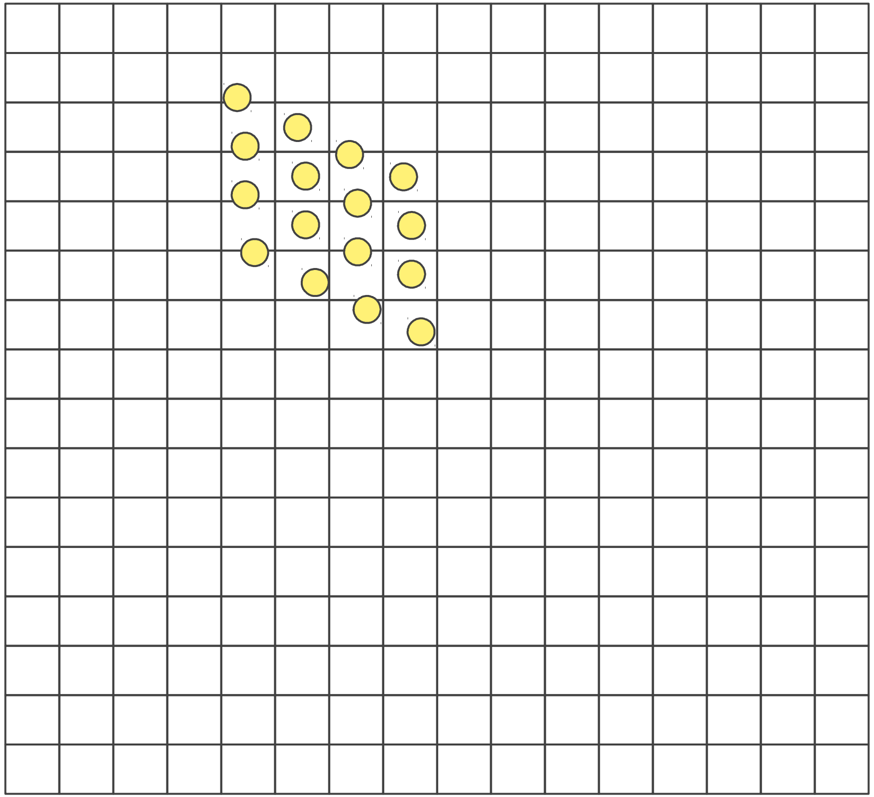
- When rendering triangles GPUs will usually render small squares at a time (for example 4x4 square for 16 ALUs)
- Pixel closeby on the screen will usually look at closeby pixels on textures

Layout of textures

- 1 pixel is 4 bytes
- Cache lines are 64 bytes
- 16 pixels per cache line

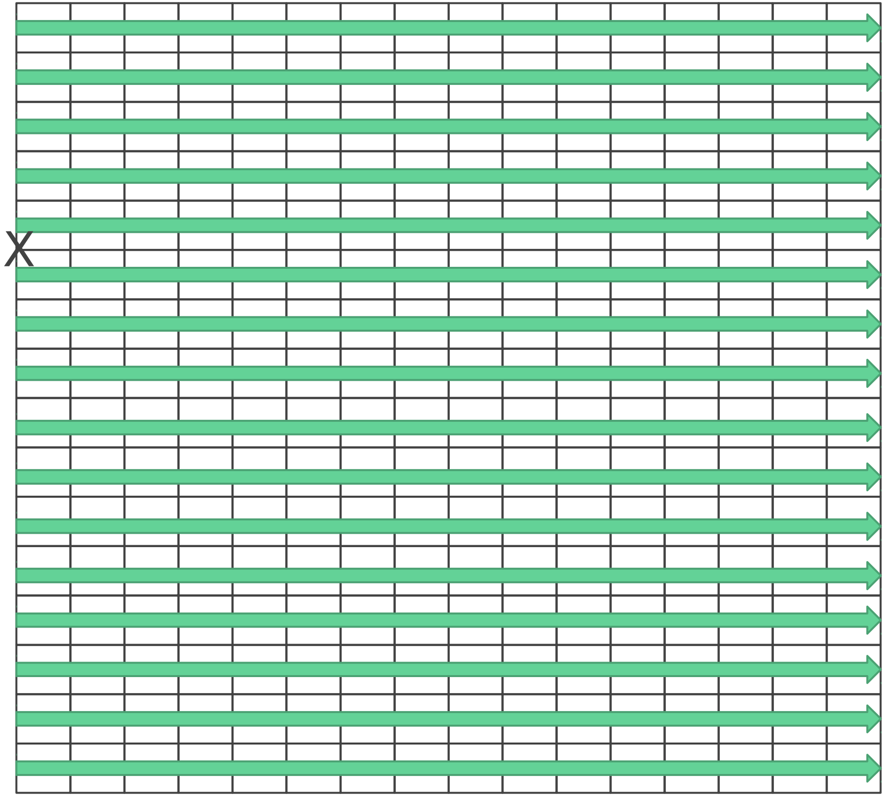
- ALUs render 4x4 pixels
- They all lookup a pixel in the textures:





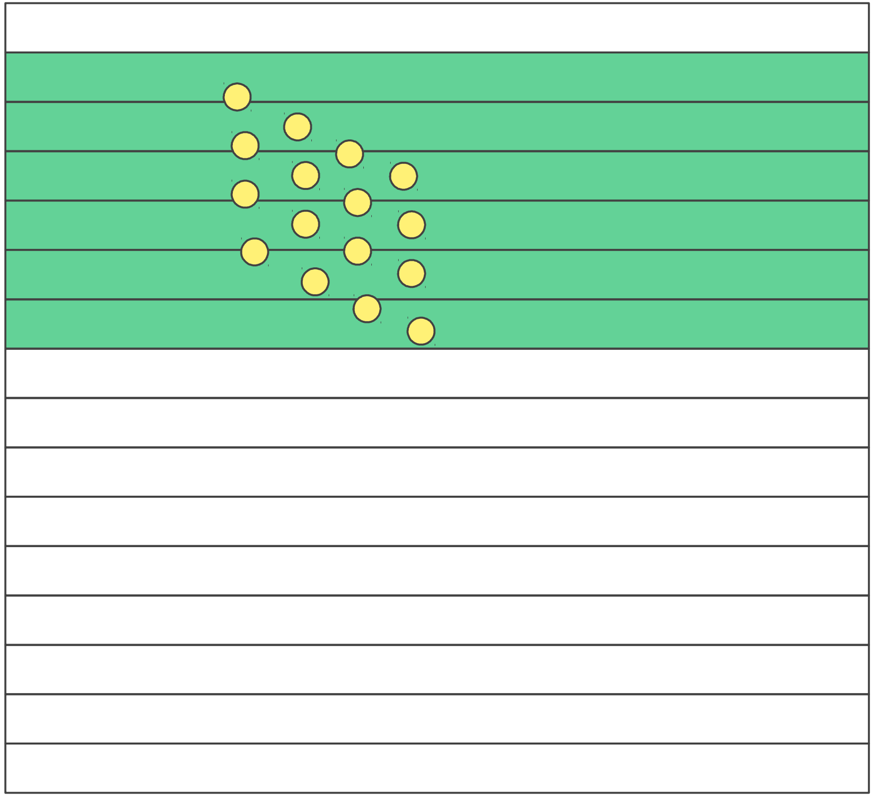
Layout of textures

- Linear layout is the intuitive order. Increasing X then increasing Y.
- A cache line is a 16x1 block.



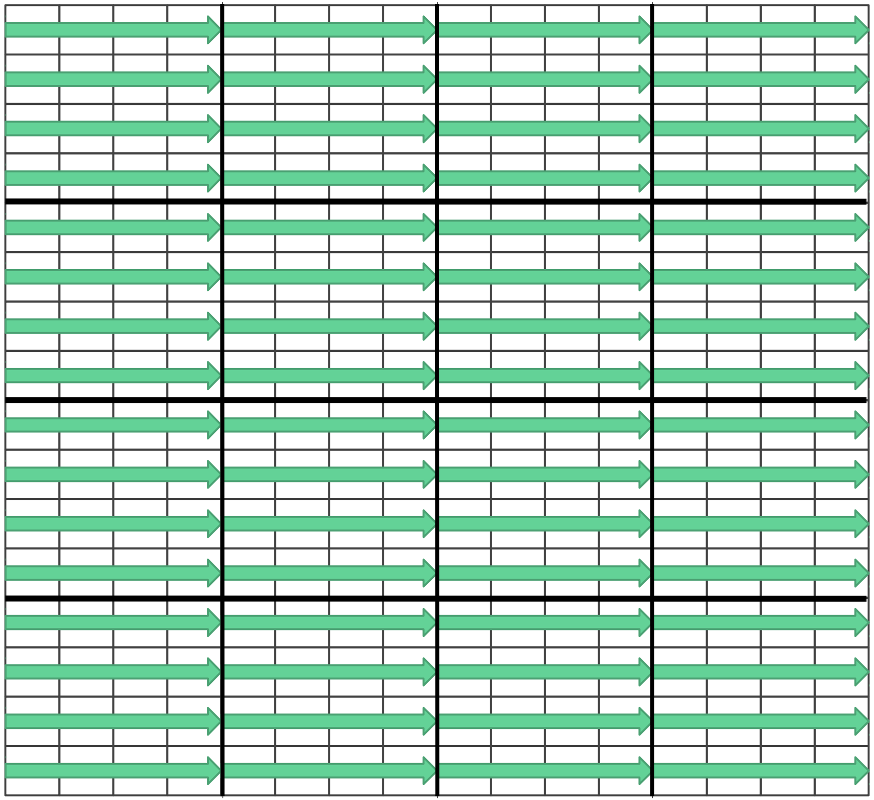
Layout of textures

- Linear layout is the intuitive order. Increasing X then increasing Y.
- A cache line is a 16x1 block.
- Our sampled pattern loads 6 cache lines.



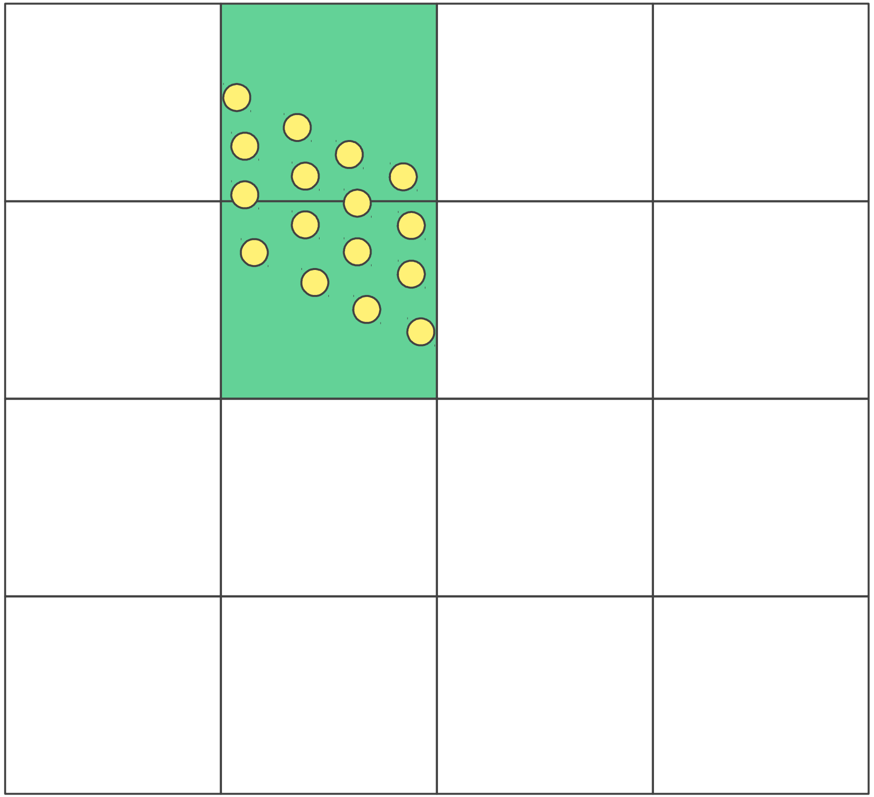
Layout of textures

- “Swizzled” layout uses smaller blocks (and linear layout in the blocks).
- A cache line is a 4x4 block.



Layout of textures

- “Swizzled” layout uses smaller blocks (and linear layout in the blocks).
- A cache line is a 4x4 block.
- Our sampled pattern loads only 2 cache lines!



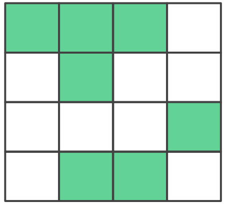
Texture compression

- Texture data is not random and can be compressed as long as it’s lossless compression
- For example a lot of textures contain constant color or even pure black color
- Lots of innovation in hardware for compression

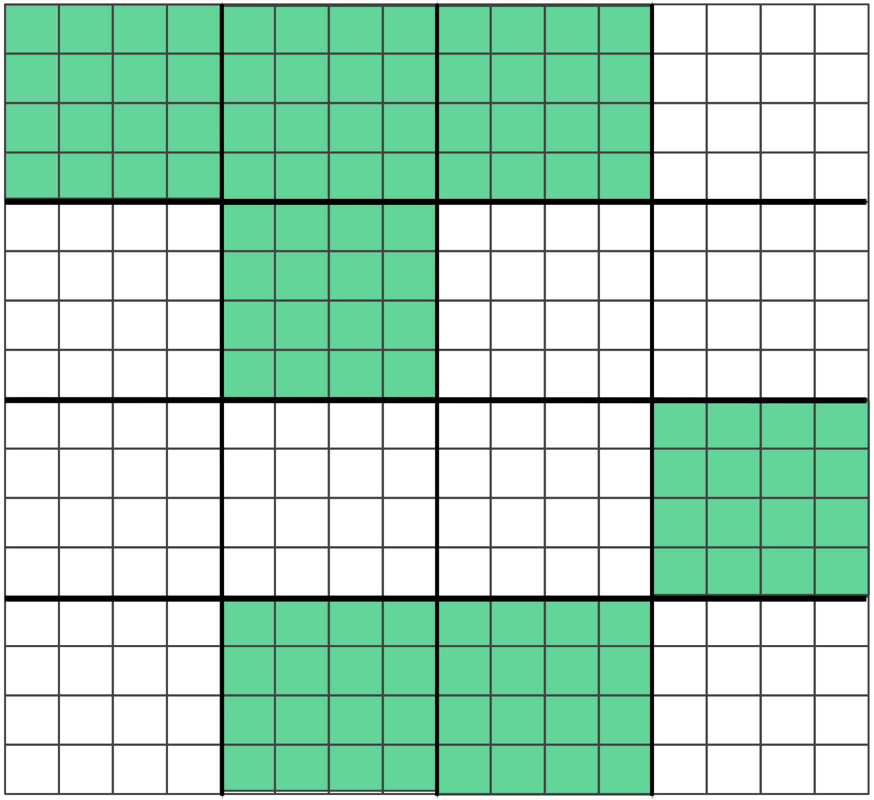
Constant Color Compression example

Clear color metadata:

Cleared tile Metadata:



Metadata size: 4 bytes + 1 bit per 16 pixels
960 pixels in a cache line!



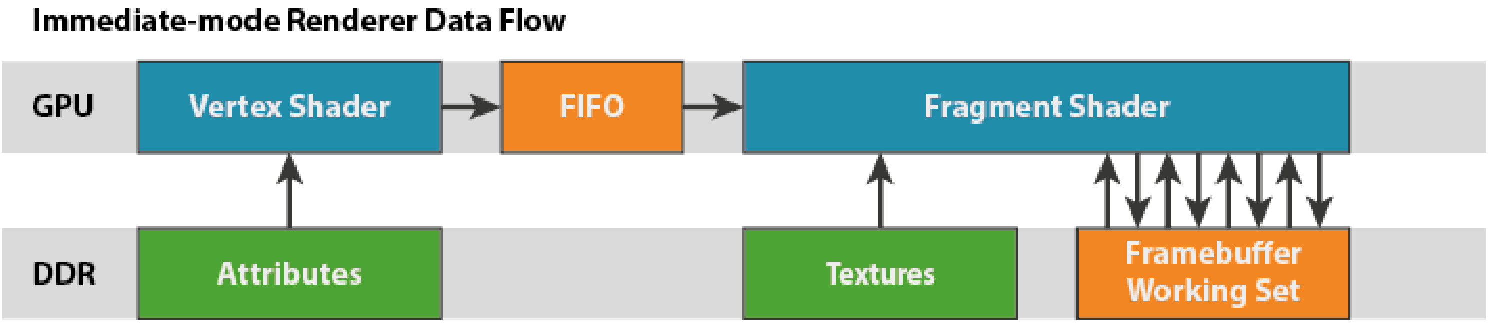
Texture compression in Intel hardware

Taken from the Vulkan driver, exactly what they do is ???

- CCS_D: Constant color compression
- CCS_E (Gen 9+): More general color compression
- HiZ: Hierarchical depth compression
- MCS: Multisample color compression

Immediate-mode vs. tiled-based GPUs

Immediate mode rendering




```
foreach(triangle)
    foreach(fragment)
        load FBO data (color, depth, ...)
        call fragment shader
        store new FBO data
```

Problems of immediate mode rendering

```
foreach(triangle)
    foreach(fragment in triangle)
        load FBO data (color, depth, ...)
        call fragment shader
        store new FBO data
```

- Random accesses thrash the caches
- Loading and storing the same fragment multiple times costs power, especially on mobile

Tile based rendering

```
foreach(fragment)
    load FBO data (color, depth, ...)
    foreach(triangle)
        call fragment shader
    store new FBO data
```

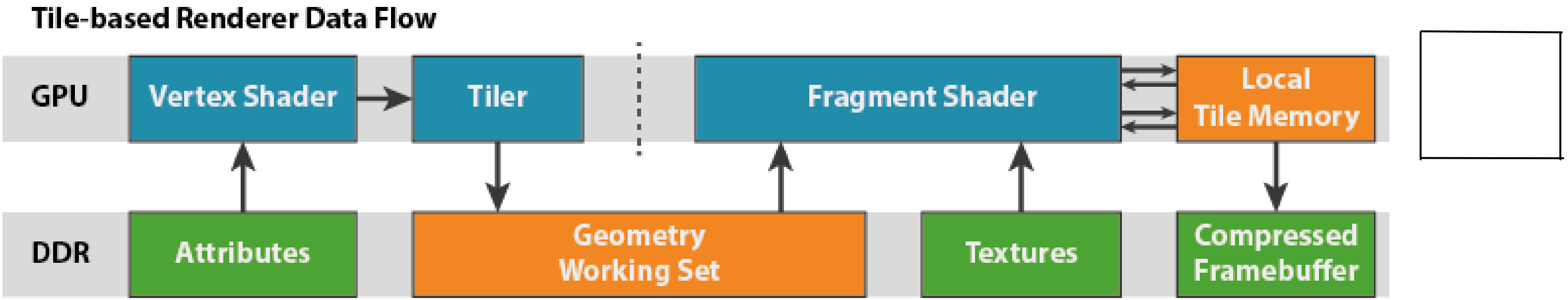
- Idea: switch the loops
- Problem: storing triangles per pixel is too expensive

Tile based rendering

```
foreach(tile)
    load tile FBO data (color, depth, ...)
    foreach(triangle in tile)
        foreach(fragment in triangle in tile)
            call fragment shader
    store new tile FBO data
```

- Idea: split FBO in tiles, store triangles per tile
  - A tile can for example be a 16x16 square
- Helps with cache coherency, FBO stored as array of tiles
- One load and one store per pixel

Tile-based rendering



Per-tile triangle lists

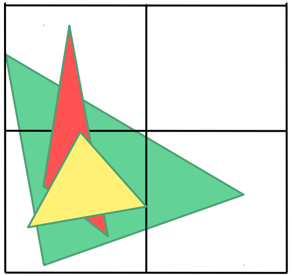
1

2

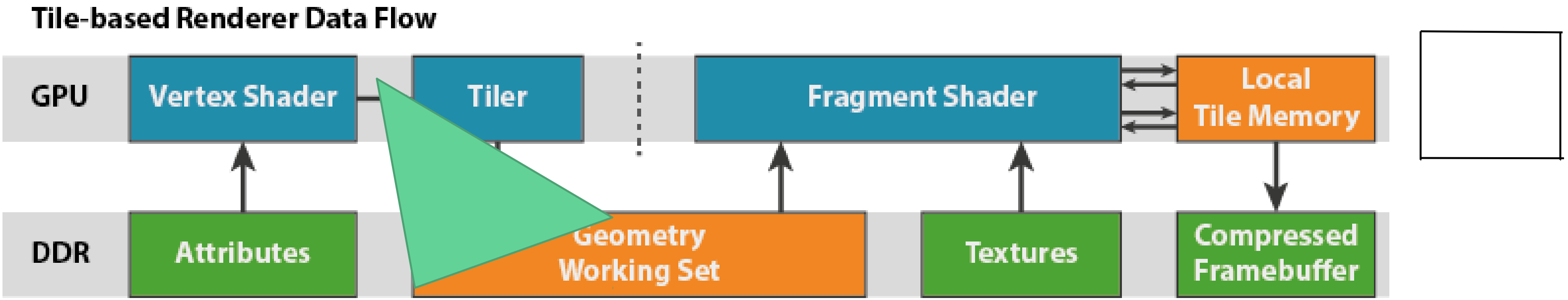
3

4

Final state:



Tile-based rendering



1

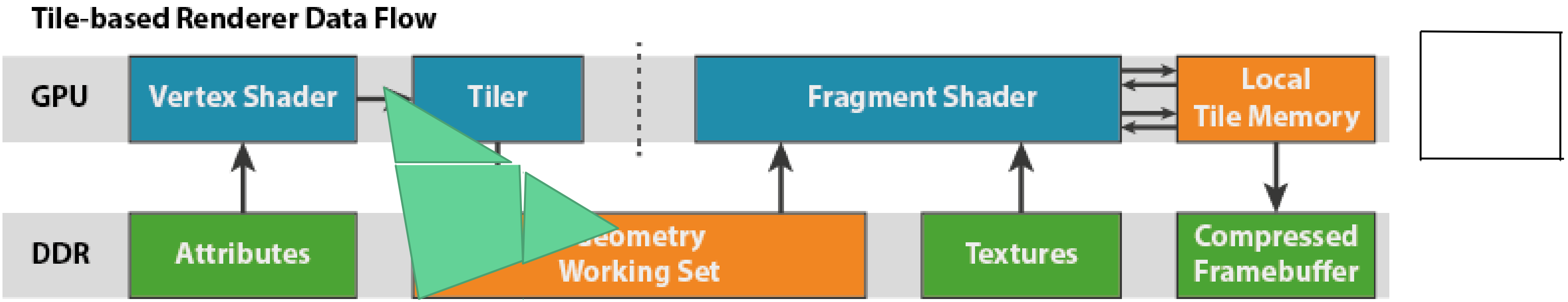
Per-tile triangle lists

2

3

4

Tile-based rendering



1

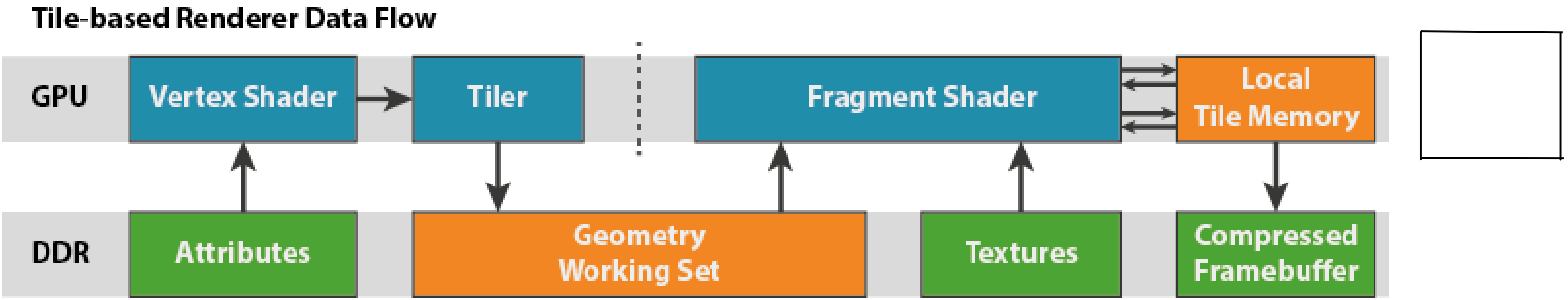
Per-tile triangle lists

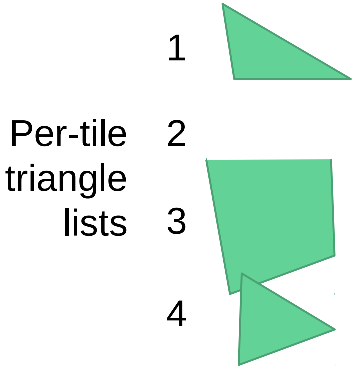
2

3

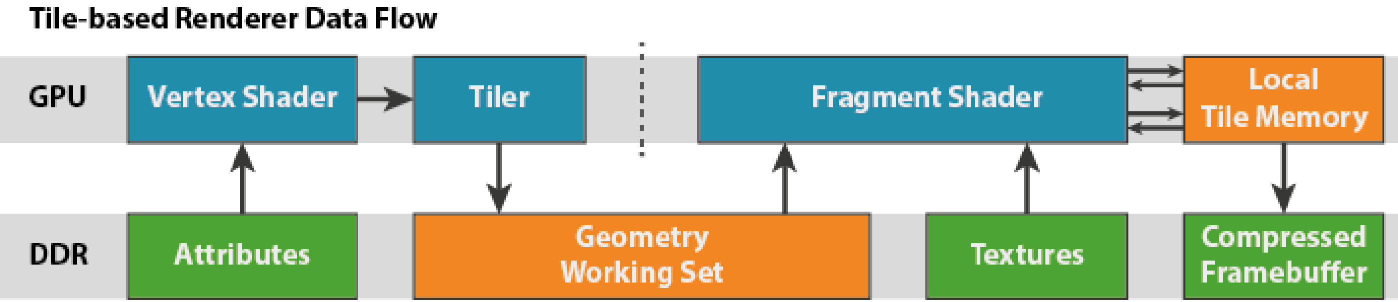
4

Tile-based rendering





Tile-based rendering



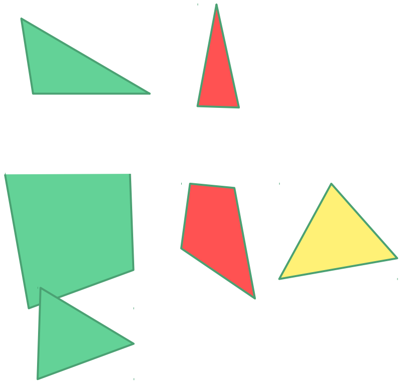
Per-tile triangle lists

1

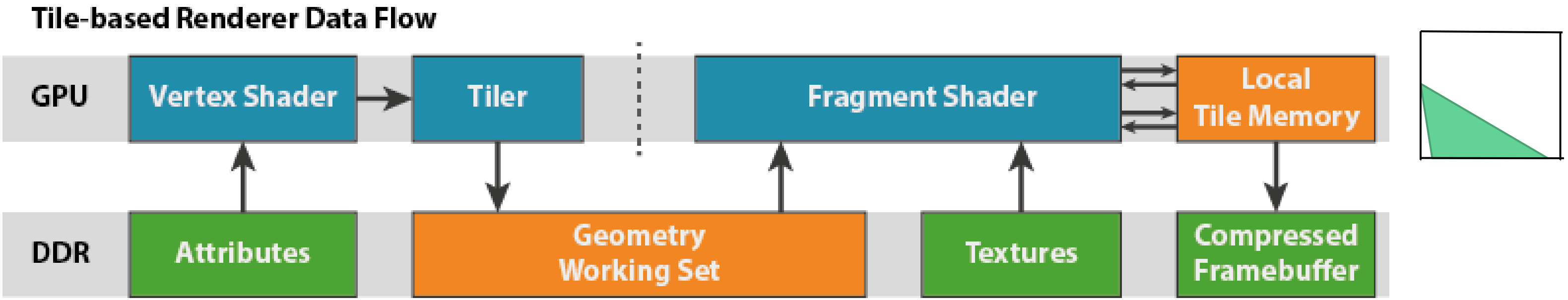
2

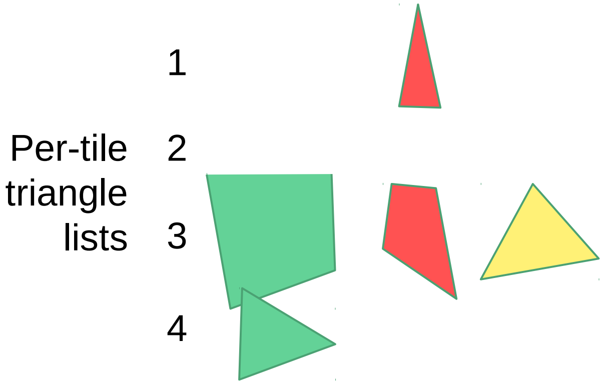
3

4

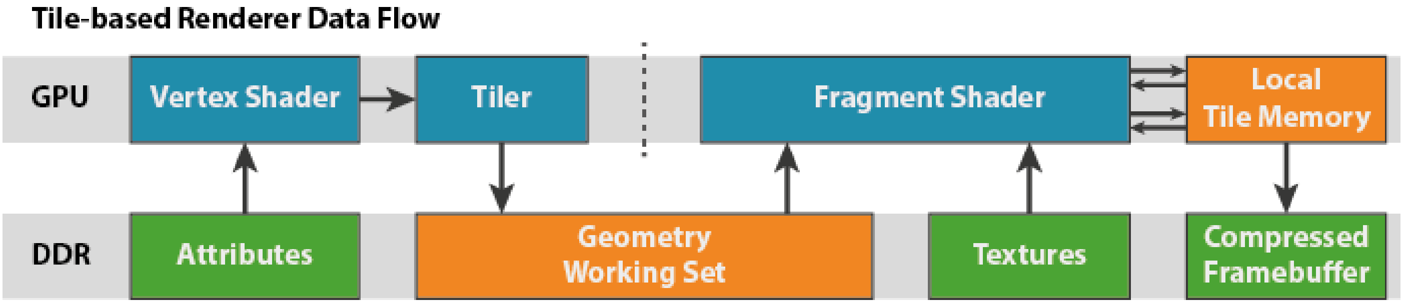


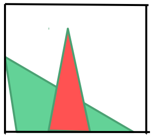
Tile-based rendering





Tile-based rendering





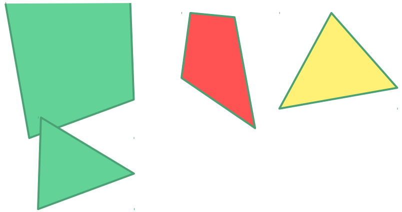
1

Per-tile triangle lists

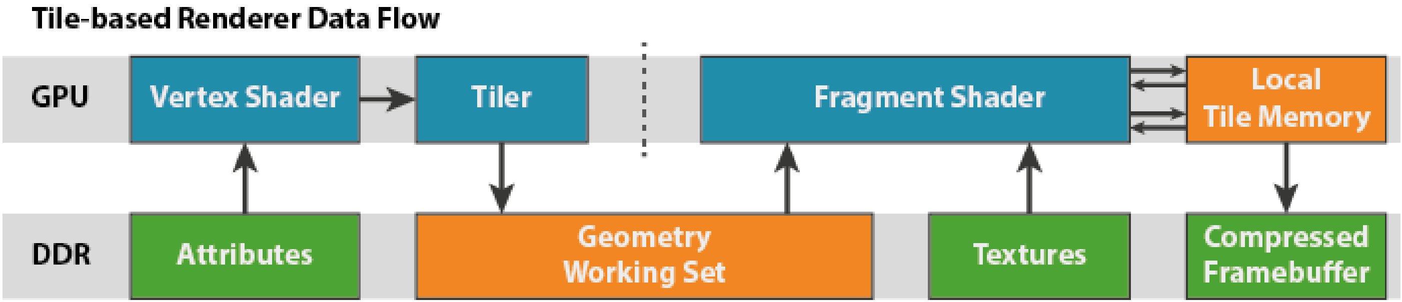
2

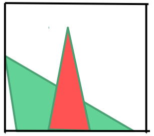
3

4



Tile-based rendering





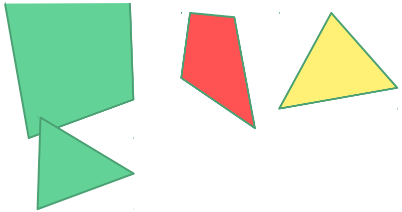
Per-tile triangle lists

1

2

3

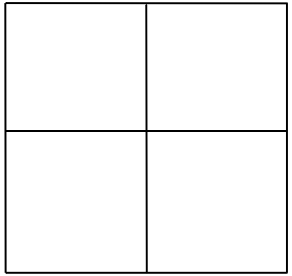
4



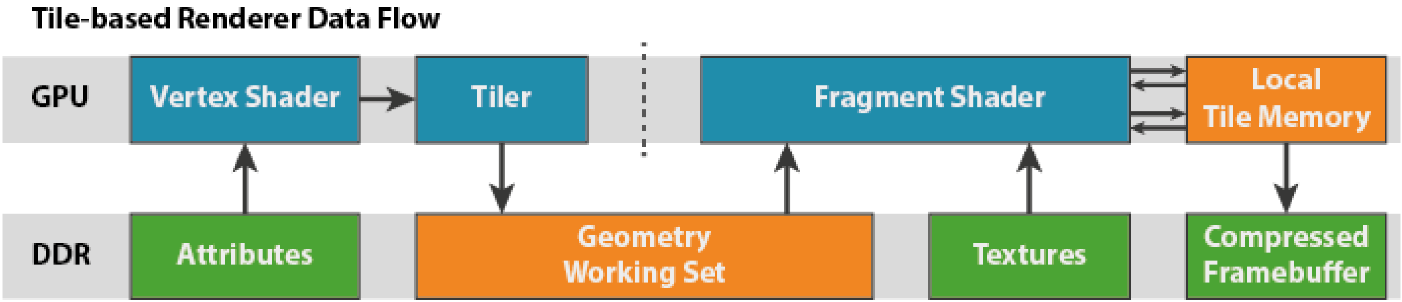
Rendertarget:

1 2

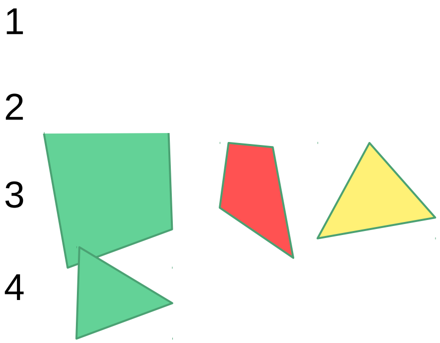
3 4



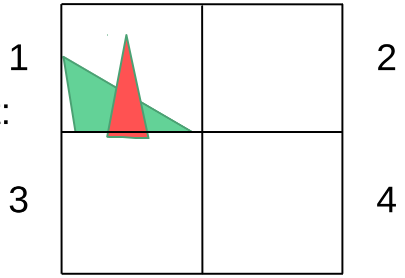
Tile-based rendering



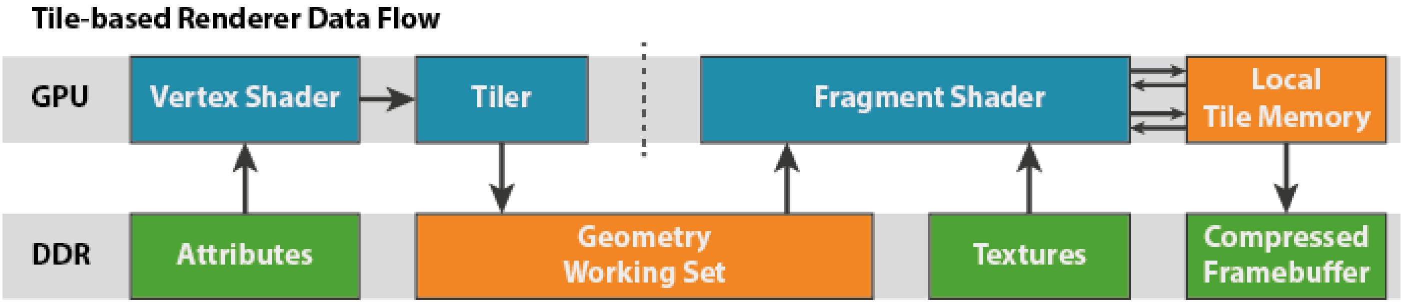
Per-tile triangle lists



Rendertarget:



Tile-based rendering



Per-tile triangle lists

1

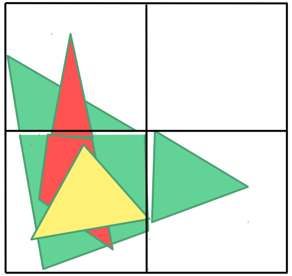
2

3

4

Rendertarget:

1 2



3 4

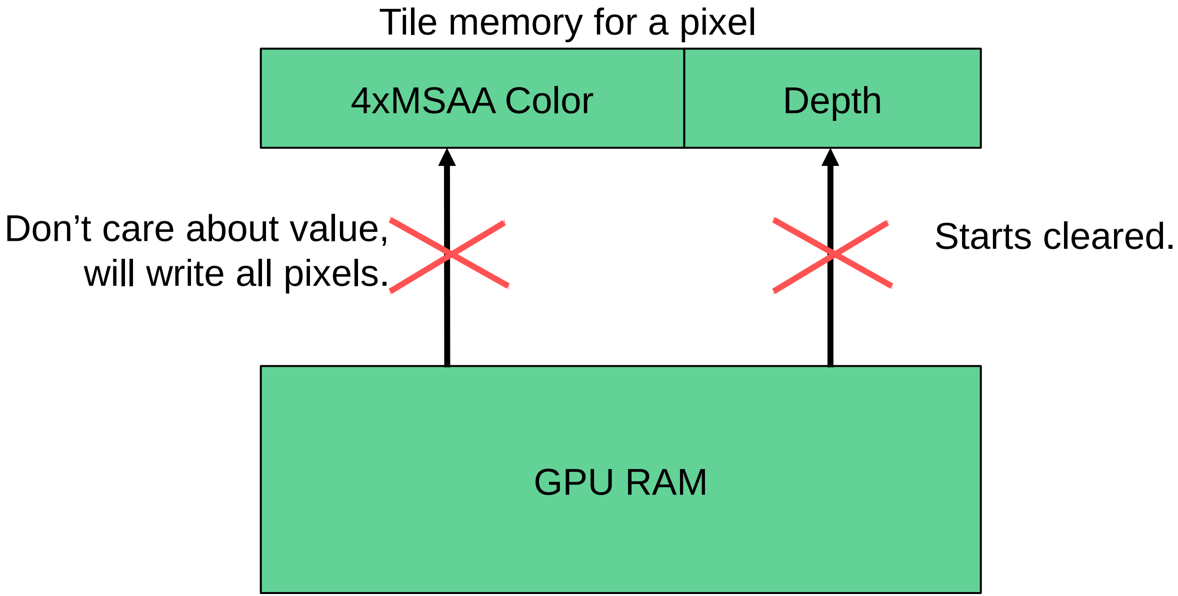
More optimizations

- What if
  - We know the initial depth is constant?
  - We only use the depth for testing and don’t care about the final value?
  - We don’t care about the initial color?

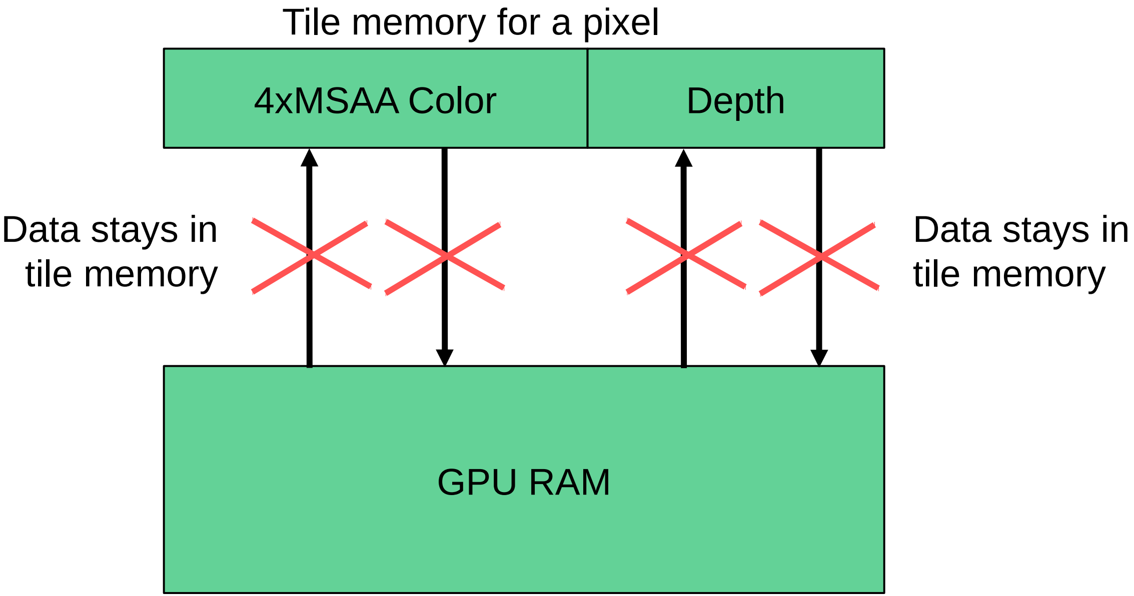
Graphics API render passes

- The define a range of operations rendering to textures
  - How the textures are loaded (if at all)
  - How they are stored (if at all)
  - How multisampled textures are resolved

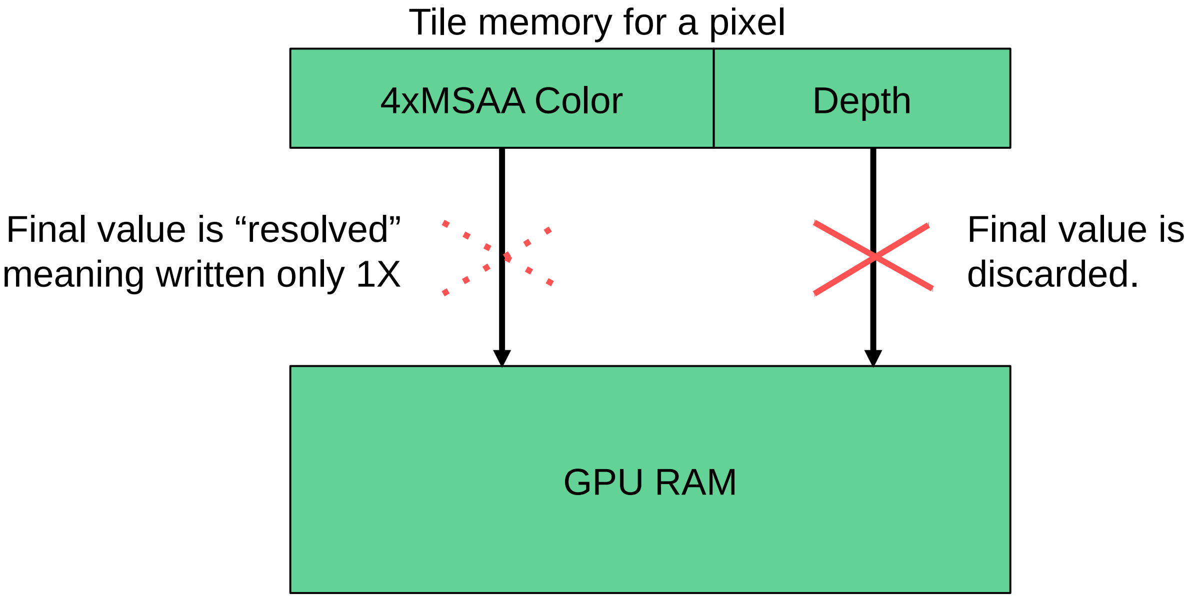
Example of render pass wins



Example of render pass wins



Example of render pass wins



Thank you!

Questions?