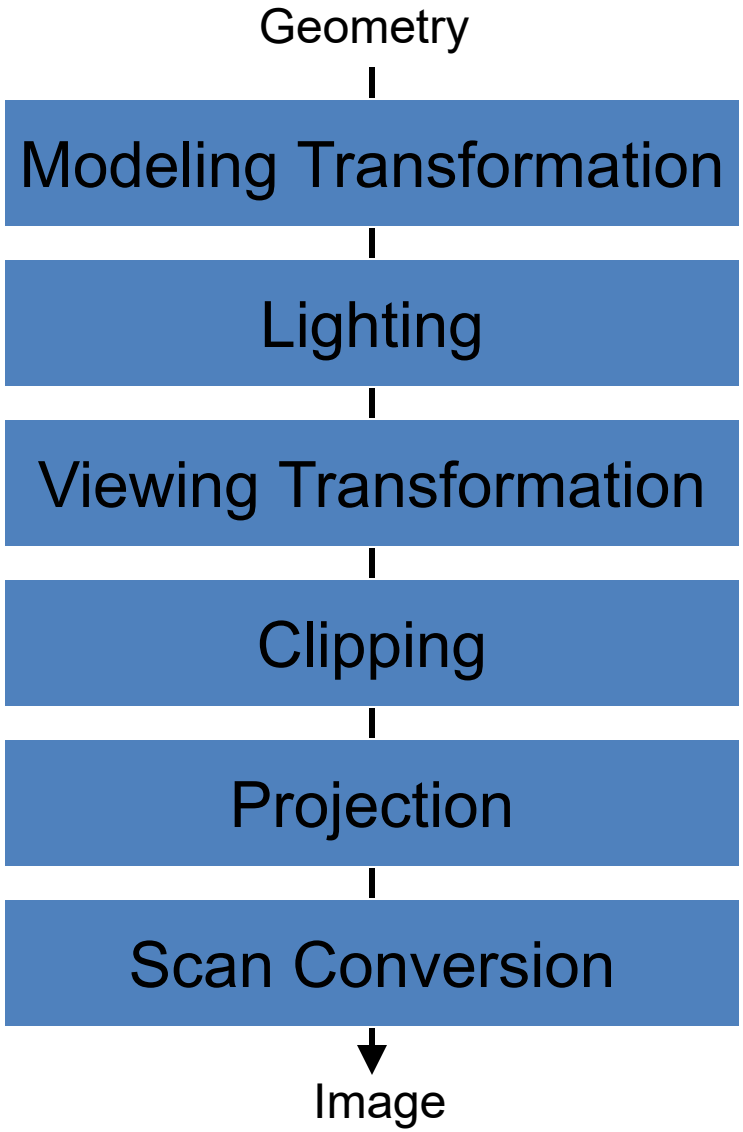
The Way of the GPU

(based on GPGPU SIGGRAPH Course)

CS535

Daniel G. Aliaga
Department of Computer Science
Purdue University

Computer Graphics Pipeline



Transform into 3D *world* coordinate system

Simulate illumination and reflectance

Transform into 3D *camera* coordinate system

Clip primitives outside camera's view

Transform into 2D camera coordinate system

Draw pixels (incl. texturing, hidden surface…)

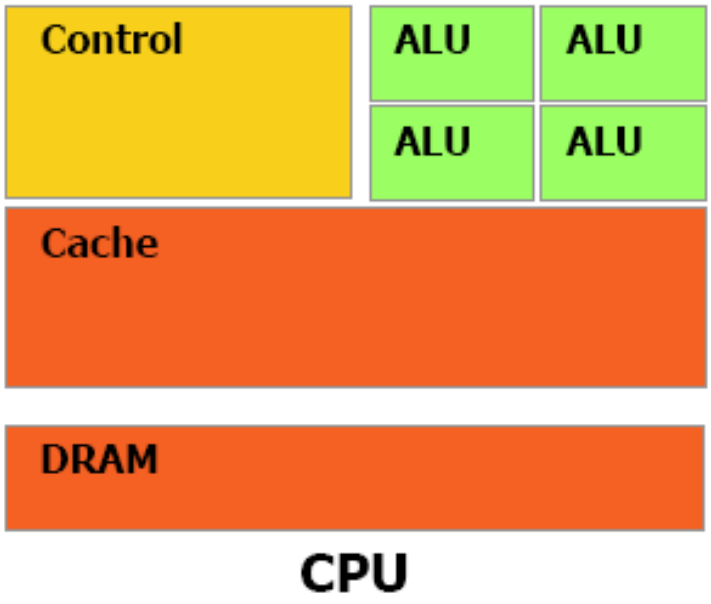
Today, we have GPUs…

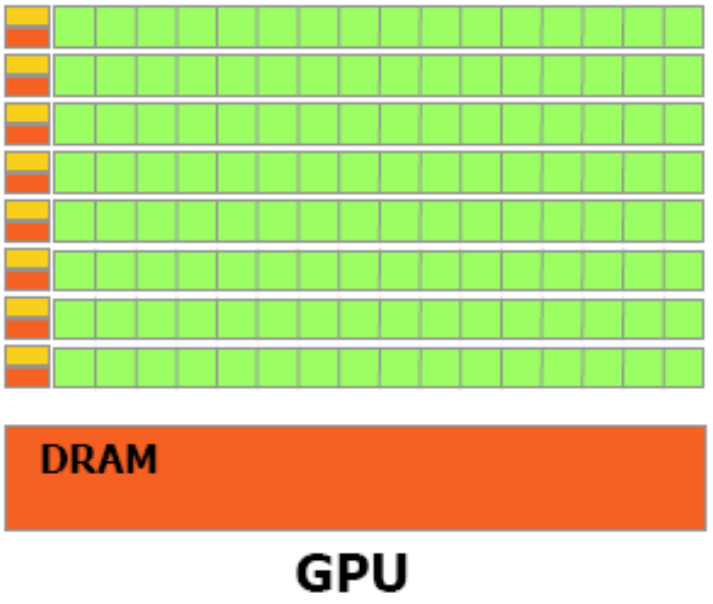
(GPU = graphical processing unit)

Motivation: Computational Power

- *Why are GPUs fast?*
  - Arithmetic intensity: the specialized nature of GPUs makes it easier to use additional transistors for computation not cache
  - Economics: multi-billion dollar video game market is a pressure cooker that drives innovation

Modern GPU has more ALU's





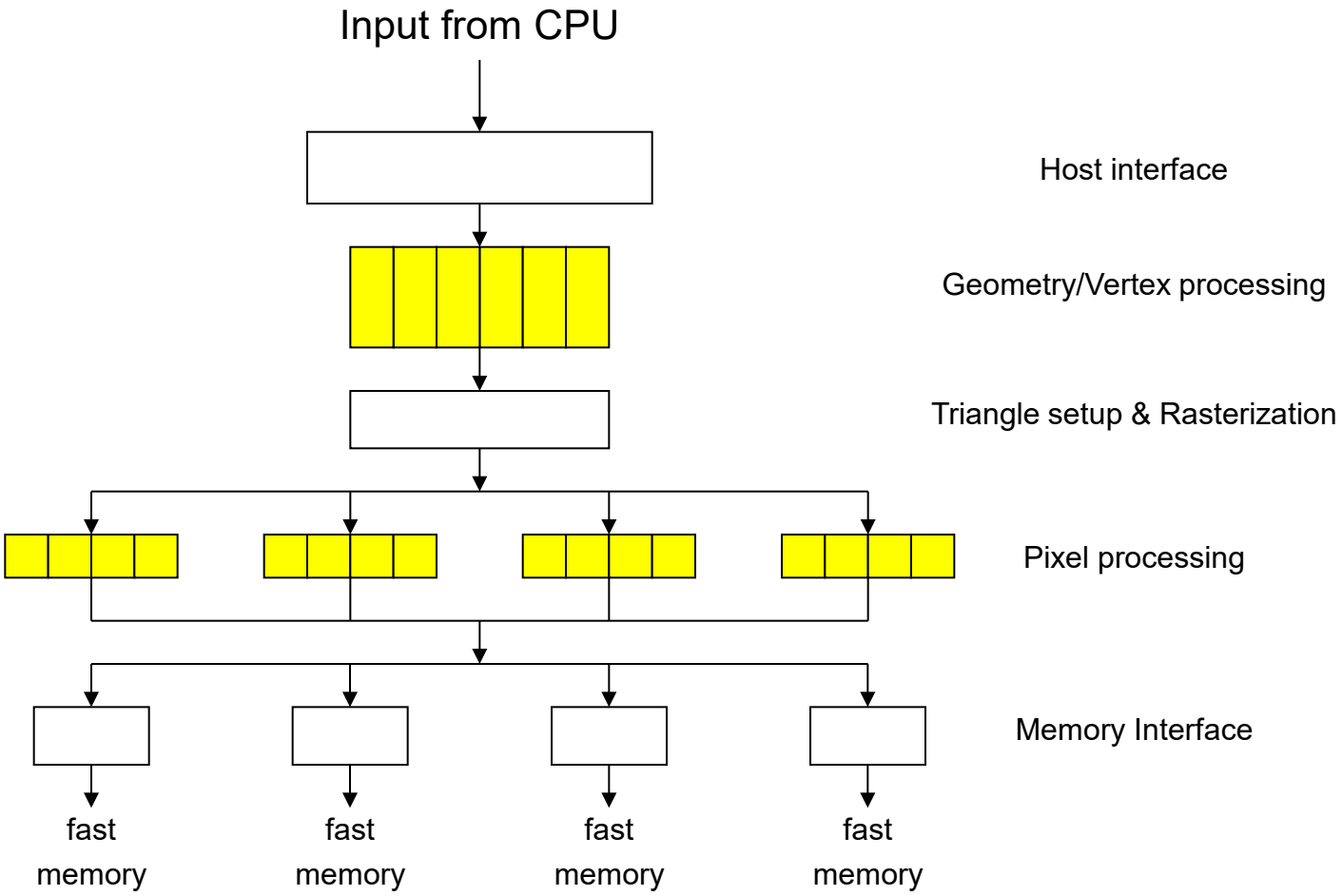
nVIDIA GPU

- GTX 4090
  - 16,384 cores (i.e., mini processors)
  - FLOPS: 83-191 TFLOPS
  - Temp: 90 C (water boils at 100 C)
  - Power: 850 Watts (~a microwave oven)

- 1 Datacenter ~32,000 GPUs
  - ~27 Megawatts
  - (watts needed to for about 2,700 average US homes)

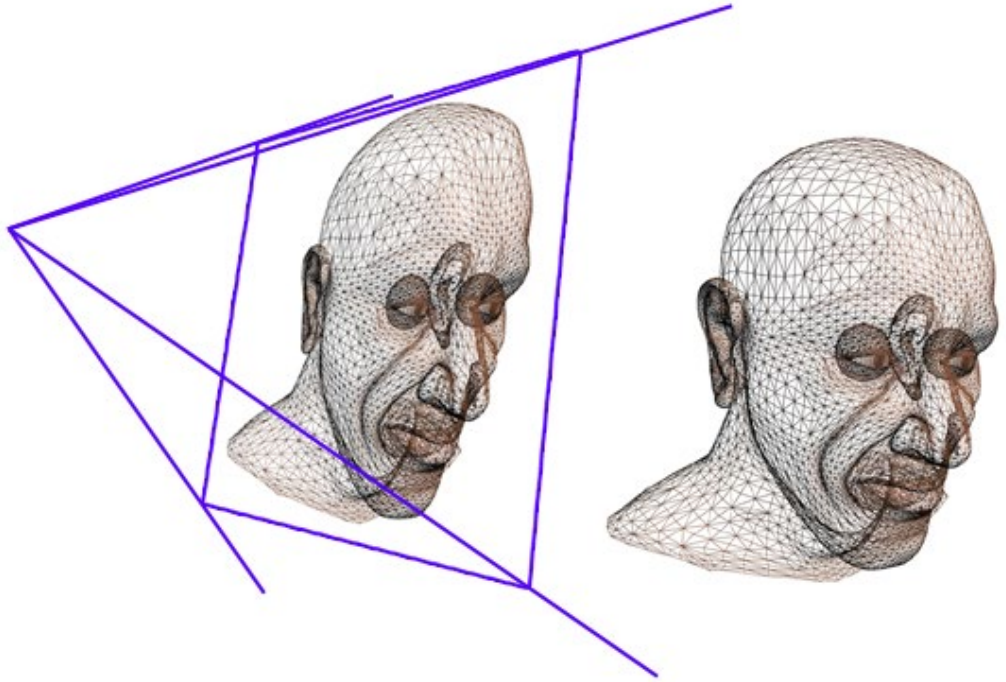
- For all datacenters (~11,000 datacenters globally)
  - Power of about ~30,000,000 average US homes

Diagram of a Modern GPU



GPU Pipeline: Transform

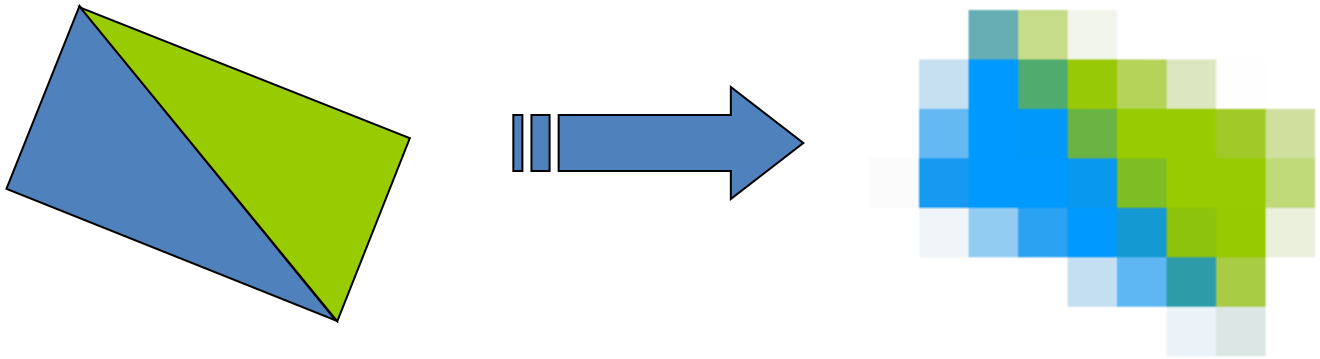
- Geometry/Vertex processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-primitive and per-vertex lighting



GPU Pipeline: Rasterize

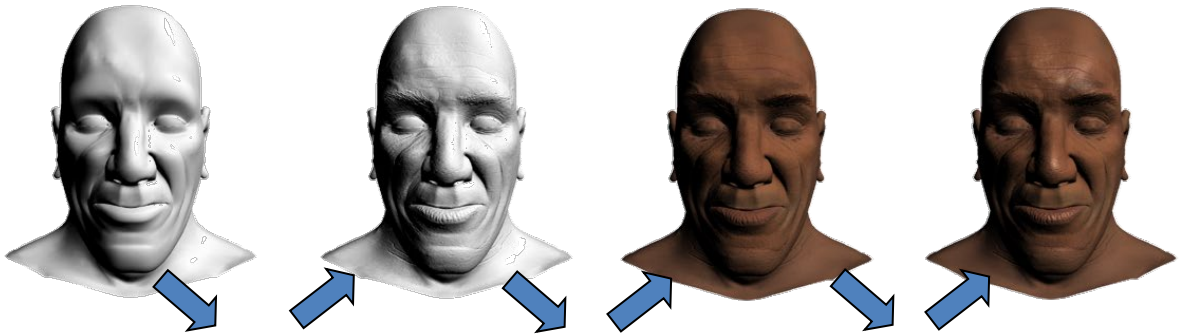
(typically not programmable)

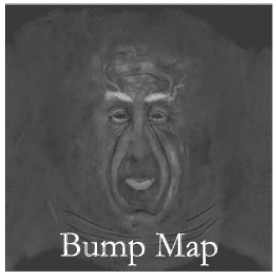
- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Fragment = image fragment
      - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels



GPU Pipeline: Shade

- Fragment/Pixel processor (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)





High Level Shading Languages

- Cg, HLSL, & OpenGL Shading Language
  - Cg:
    - http://www.nvidia.com/cg
  - HLSL:
    - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/highlevellanguageshaders.asp
  - OpenGL Shading Language:
    - http://www.3dlabs.com/support/developer/ogl2/whitepapers/index.html

A really naïve shader


```
frag2frame Smooth(vert2frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1,
                  uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
    frag2frame OUT;

    float2 center = IN.TexCoord0.xy;
    float4 U = f4texRECT(Source, center);

    // Calculate Red-Black (odd-even) masks
    float2 intpart;
    float2 place = floor(1.0f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
    float2 mask = float2((1.0f-place.x) * (1.0f-place.y), place.x * place.y);

    if (((mask.x + mask.y) && params.y) || (!(mask.x + mask.y) && !params.y))
    {
        float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f), params.x*center.y - 0.5f*(params.x-1.0f));
        ...
        float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
        float central = -2.0f*(O.x + O.y);

        float poisson = ((params.x*params.x)*U.z + (-O.x * f1texRECT(Source, float2(neighbor.x, center.y)) +
                                                    -O.x * f1texRECT(Source, float2(neighbor.y, center.y)) +
                                                    -O.y * f1texRECT(Source, float2(center.x, neighbor.z)) +
                                                    -O.z * f1texRECT(Source, float2(center.x, neighbor.w)))) / O.w;
        OUT.COL.x = poisson;
    }
    ...
    return OUT;
}
```

Computational Frequency

- Think of your CPU, vertex, and fragment programs as different levels of nested looping.

```
...
foreach tri in triangles {
    // run the vertex program on each vertex
    v1 = process_vertex(tri.vertex1);
    v2 = process_vertex(tri.vertex2);
    v3 = process_vertex(tri.vertex2);

    // assemble the vertices into a triangle
    assembledtriangle = setup_tri(v1, v2, v3);

    // rasterize the assembled triangle into [0..many] fragments
    fragments = rasterize(assembledtriangle);

    // run the fragment program on each fragment
    foreach frag in fragments {
        outbuffer[frag.position] = process_fragment(frag);
    }
}
...
```

Computational Frequency

- Branches
  - Avoid these, especially in the inner loop – i.e., the fragment program.

Computational Frequency

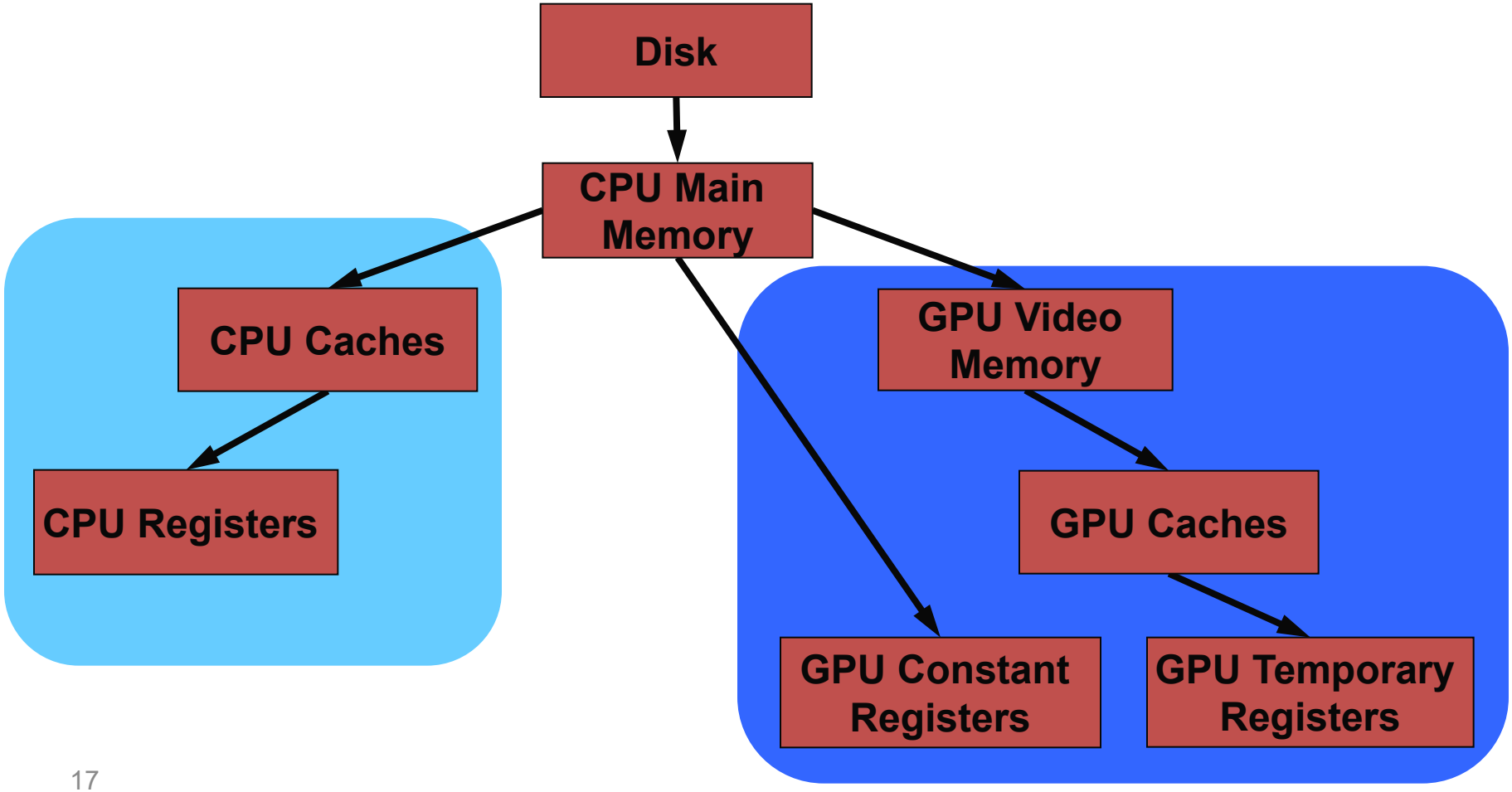
- Static branch resolution
  - write several variants of each fragment program to handle boundary cases
  - eliminates conditionals in the fragment program
  - equivalent to avoiding CPU inner-loop branching

Computational Frequency

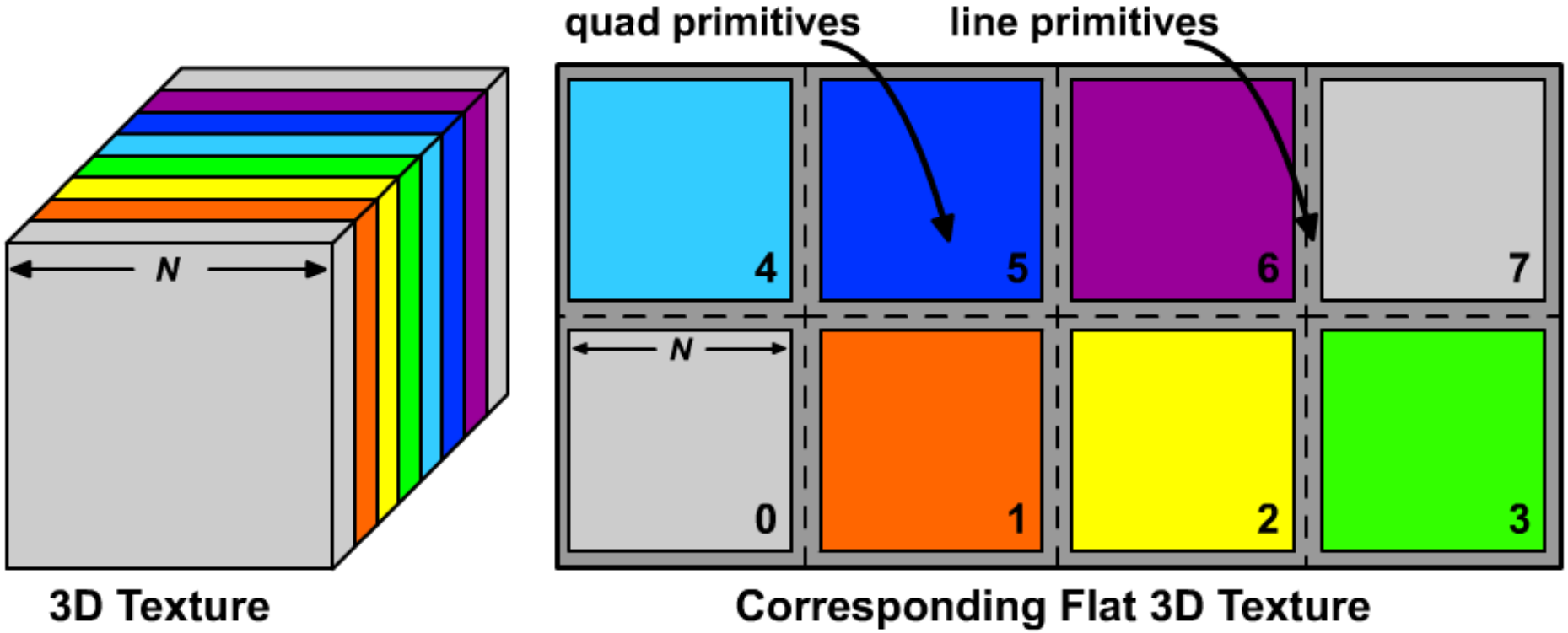
- Dynamic branching
  - Use only when needed
    - Good perf requires spatial coherence in branching

Memory Hierarchy

- CPU and GPU Memory Hierarchy



Flat 3D Textures



Flat 3D Textures

- Advantages
  - One texture update per operation
  - Better use of GPU parallelism
  - Non-power-of-two Textures
  - Quick simulation preview
- Disadvantage
  - Must compute texture offsets