

CS 380 - GPU and GPGPU Programming

Lecture 4: GPU Architecture 3

Markus Hadwiger, KAUST

Reading Assignment #2 (until Feb. 17)



Read (required):

- GLSL book, chapter 4 (*The OpenGL Programmable Pipeline*)
- GPU Gems 2 book, chapter 30 (*The GeForce 6 Series GPU Architecture*)
available online:

http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

Quiz #1: Feb. 17



Organization

- First 15 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Programming Assignments: Schedule



Assignment #1:

- Querying the GPU (OpenGL and CUDA) due Feb 10

Assignment #2:

- Phong shading and procedural texturing (GLSL) due Mar 3

----- Spring Break: Mar. 28 – Apr. 5 -----

Assignment #3:

- Image Processing with (a) GLSL, and (b) CUDA due Apr 7

Assignment #4:

- Conjugate Gradient Linear Systems Solver (CUDA) due Apr 28

Per-Pixel(Fragment) Lighting

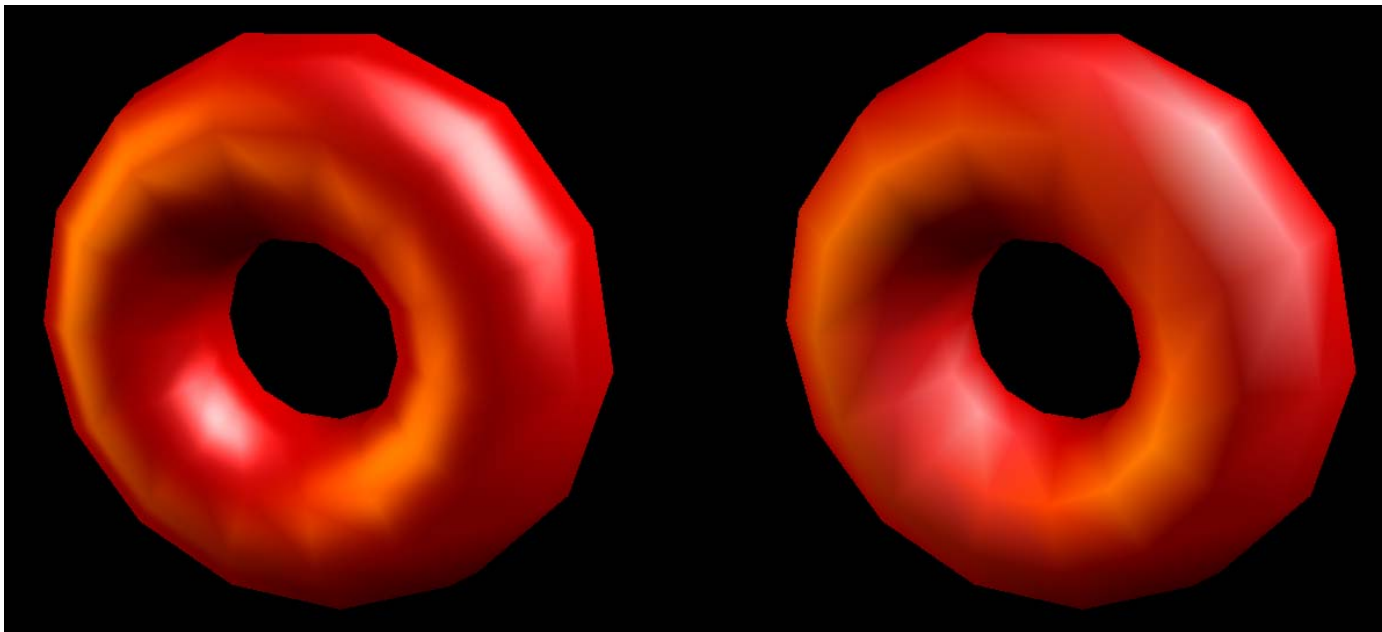


Simulating smooth surfaces by calculating illumination for each fragment

Example: specular highlights (Phong illumination/shading)

Phong shading:
per-fragment evaluation

Gouraud shading:
linear interpolation from vertices



Per-Pixel Phong Lighting (Cg)



```
void main(float4 position : TEXCOORD0,  
          float3 normal   : TEXCOORD1,  
  
          out float4 oColor : COLOR,  
  
          uniform float3 ambientCol,  
          uniform float3 lightCol,  
          uniform float3 lightPos,  
          uniform float3 eyePos,  
          uniform float3 Ka,  
          uniform float3 Kd,  
          uniform float3 Ks,  
          uniform float  shiny)  
{
```

Per-Pixel Phong Lighting (Cg)



```
float3 P = position.xyz;
float3 N = normal;
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);

float3 ambient = Ka * ambientCol;

float3 L          = normalize(lightPos - P);
float  diffLight = max(dot(L, N), 0);
float3 diffuse    = Kd * lightCol * diffLight;

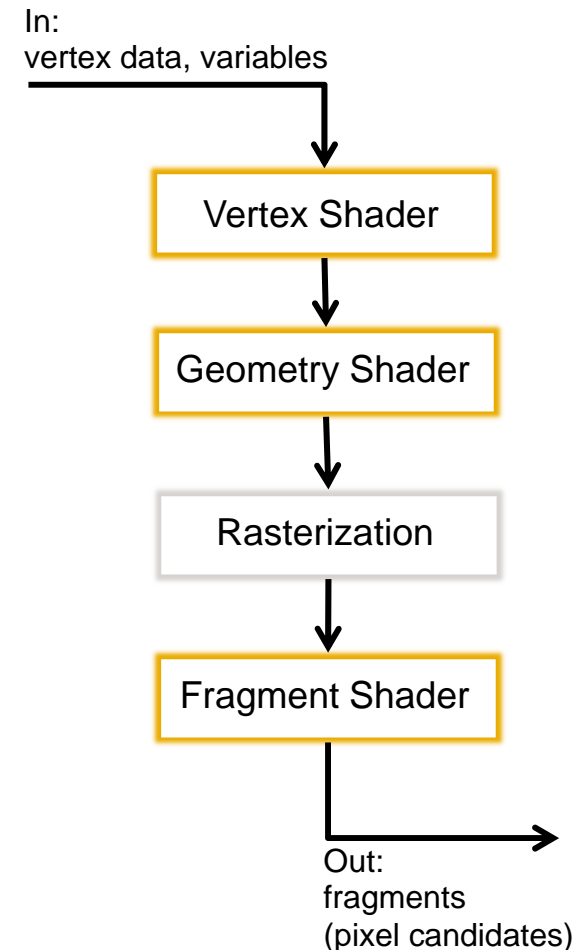
float  specLight = pow(max(dot(H, N), 0), shiny);
float3 specular  = Ks * lightCol * specLight;

oColor.xyz = ambient + diffuse + specular;
oColor.w = 1;
}
```

Assignment 2 – Shaders – Su., 03.03.



- What you'll learn
 - Basic graphics pipeline
 - How to use vertex shaders
 - How to use fragment shaders
- What you'll pick up on the way
 - GLSL
 - Phong lighting and shading



Assignment 2 – Shaders – Su., 03.03.



- What's already there
 - Different models
 - Shader setup
 - Loading
 - Compiling
 - Variables
 - Gouraud shading



Assignment 2 – Shaders – Su., 03.03.



- What you'll have to do
 - Per fragment lighting (instead of per vertex)
 - Procedural texturing “OpenGL Shading Language” book (11.1-11.3)
 - Optional: (11.4) procedural bump mapping
- Where to start
 - readme.txt
 - Understand the framework code (how to edit shaders, etc.)
 - Look for TODOs



SIGGRAPH 2009

NEW ORLEANS

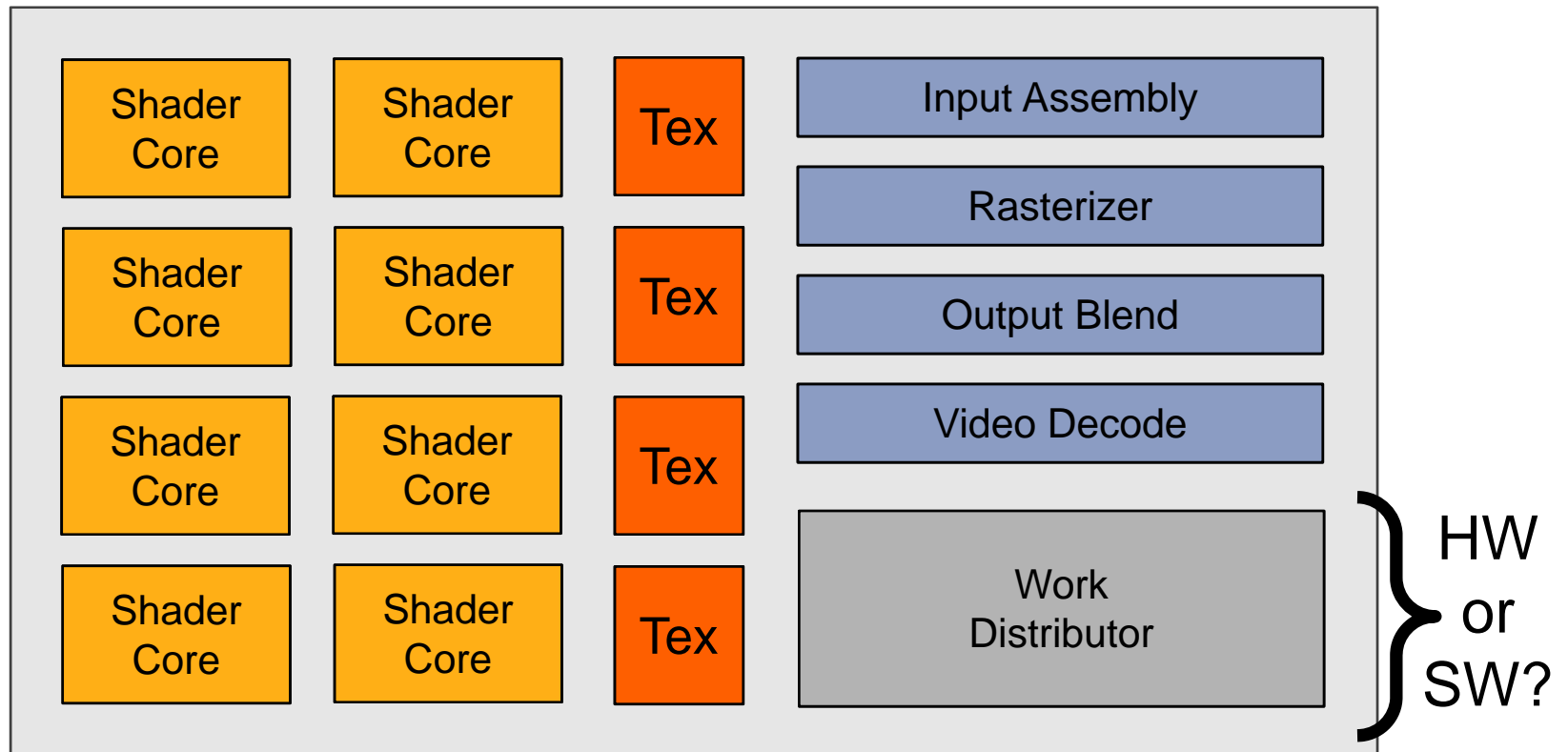
From Shader Code to a **Teraflop**: How Shader Cores Work

Kayvon Fatahalian
Stanford University

Part 1: throughput processing

- Three key concepts behind how modern GPU processing cores run code
- Knowing these concepts will help you:
 1. Understand space of GPU core (and throughput CPU processing core) designs
 2. Optimize shaders/compute kernels
 3. Establish intuition: what workloads might benefit from the design of these architectures?

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

A diffuse reflectance shader

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Independent, but no explicit parallelism

Compile shader

1 unshaded fragment input record



```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp ( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```



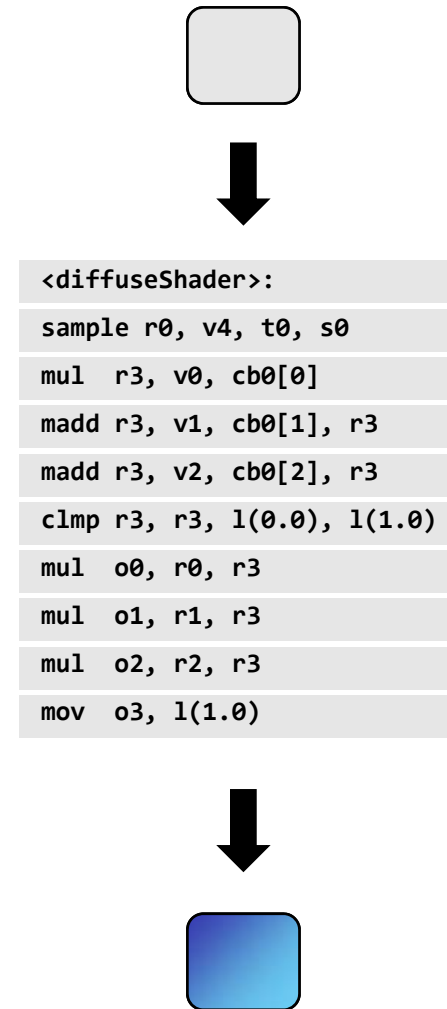
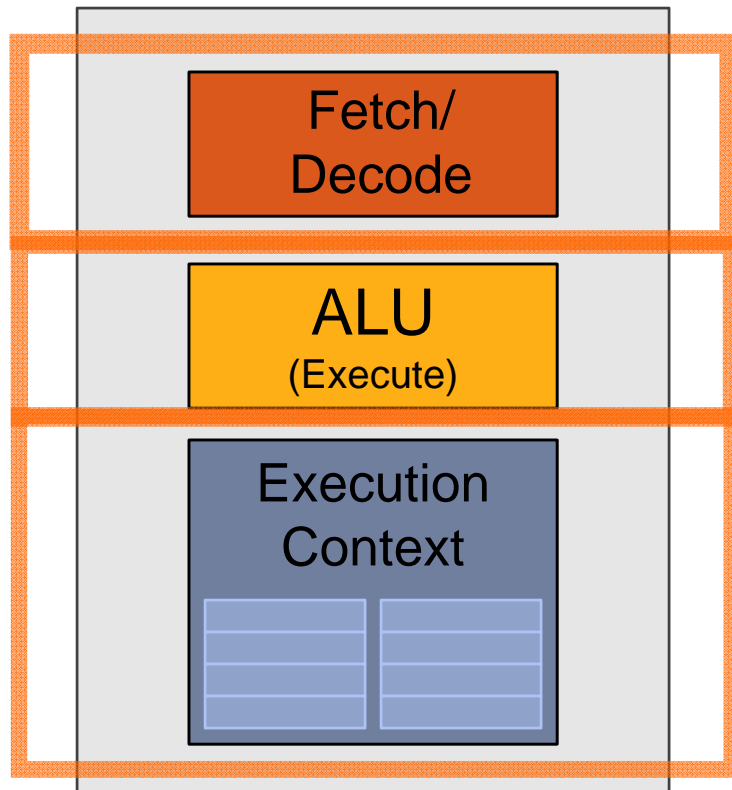
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



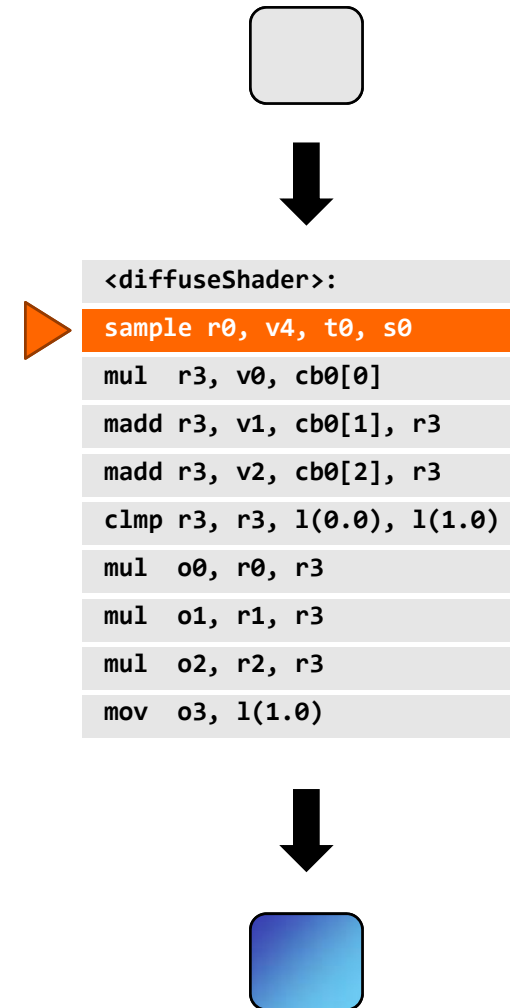
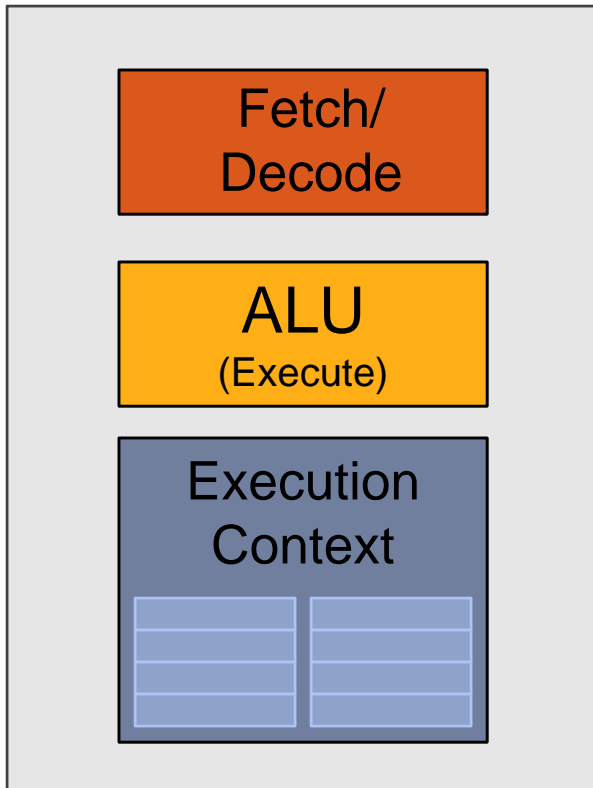
1 shaded fragment output record



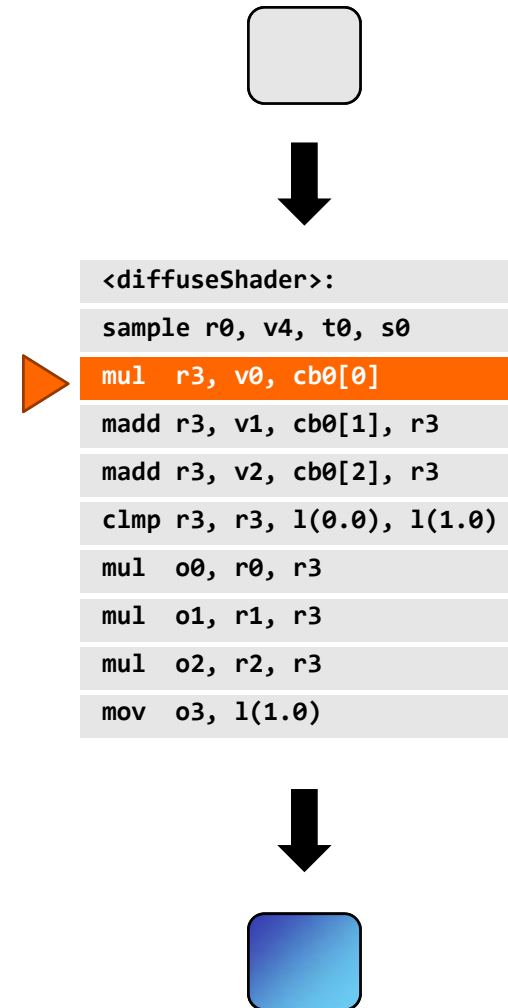
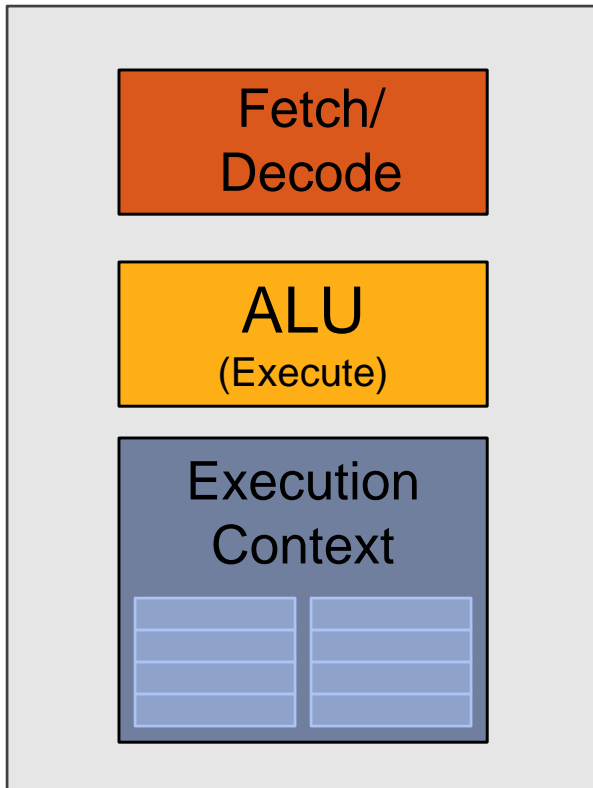
Execute shader



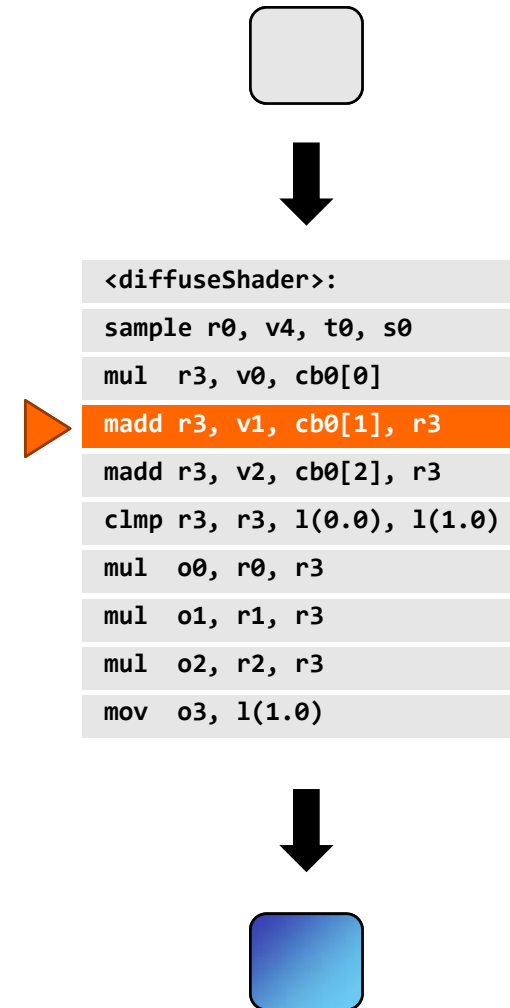
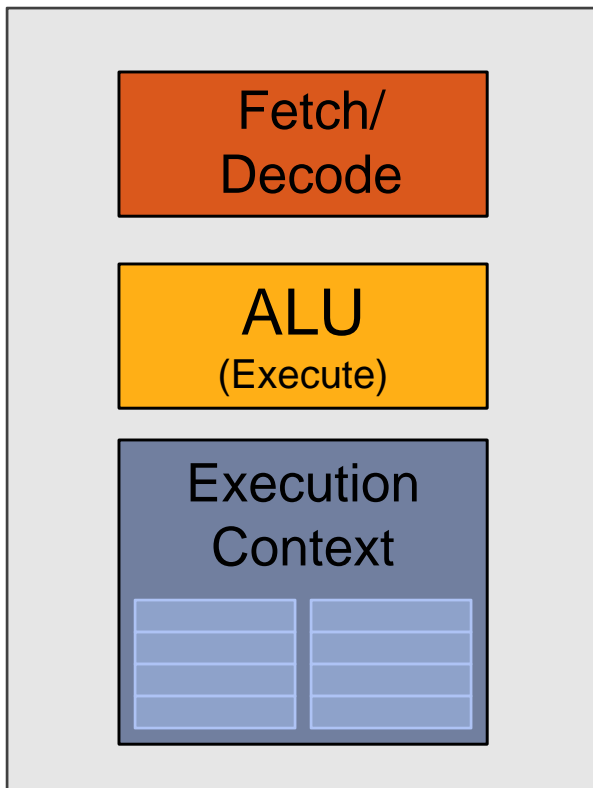
Execute shader



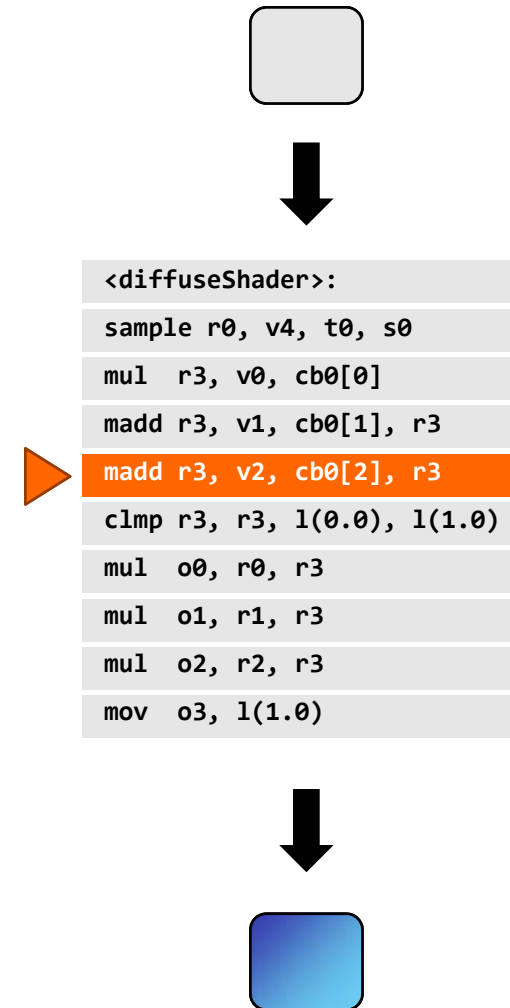
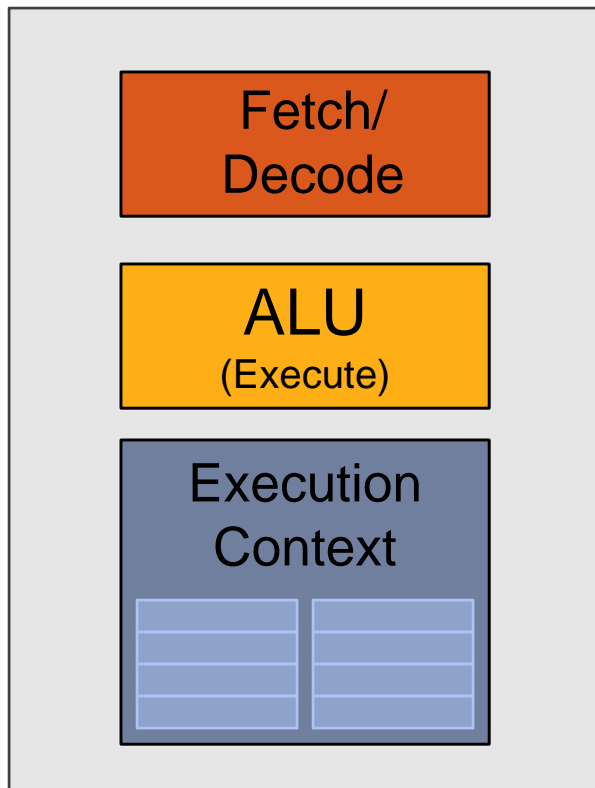
Execute shader



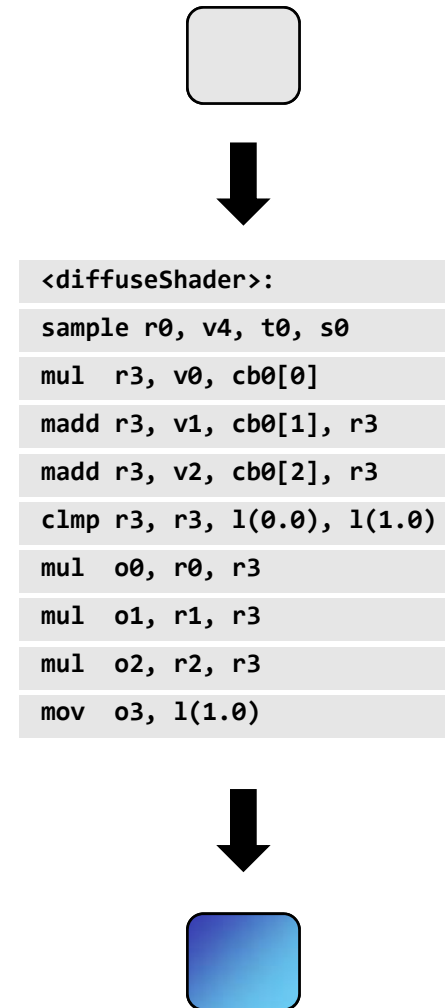
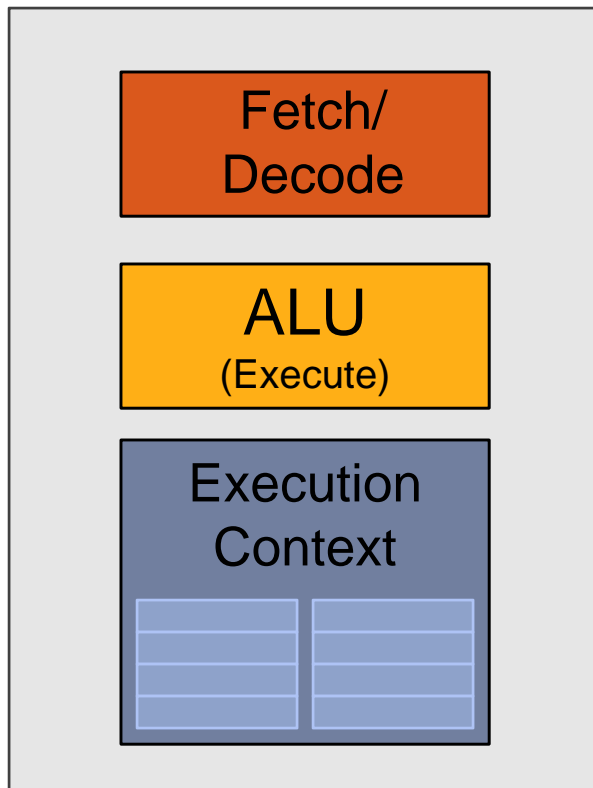
Execute shader



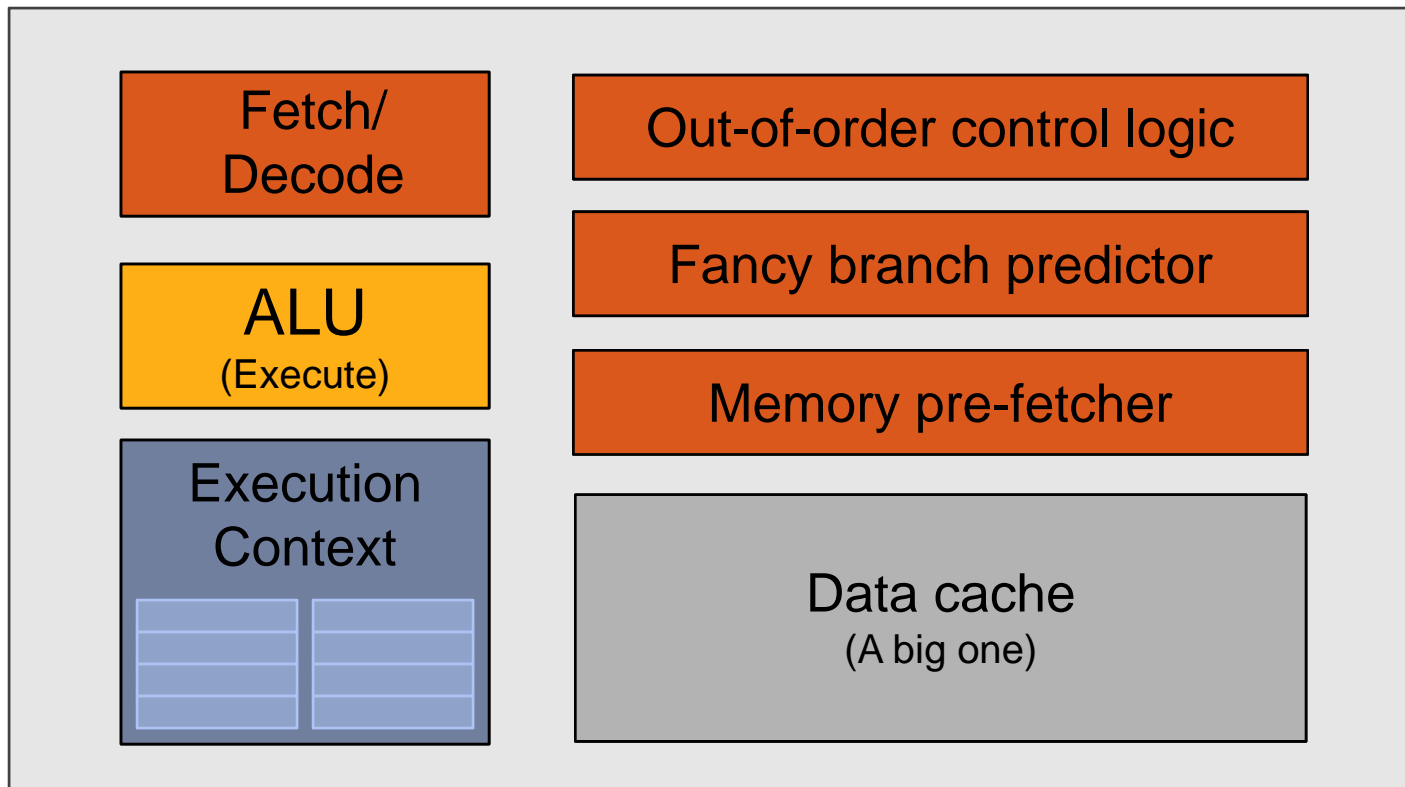
Execute shader



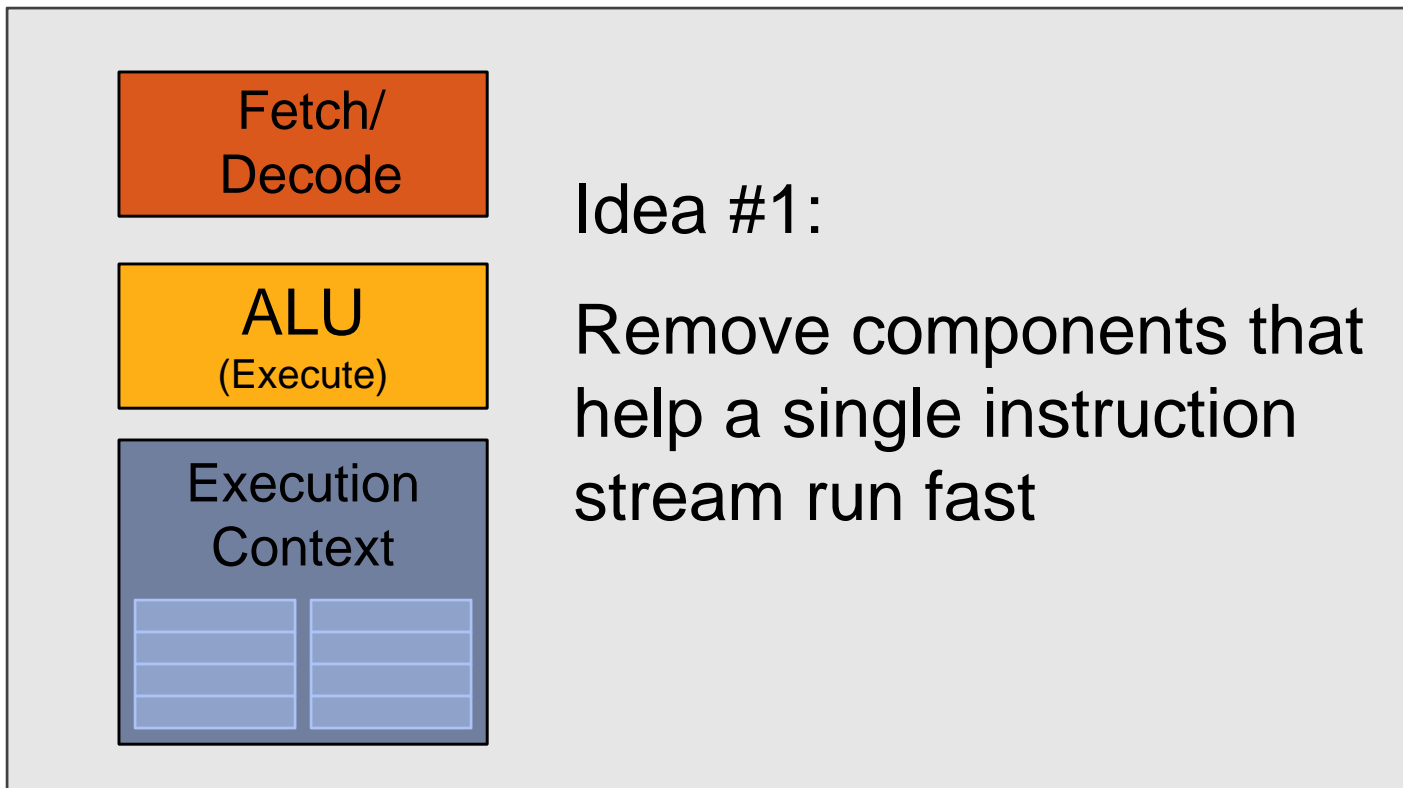
Execute shader



CPU-“style” cores

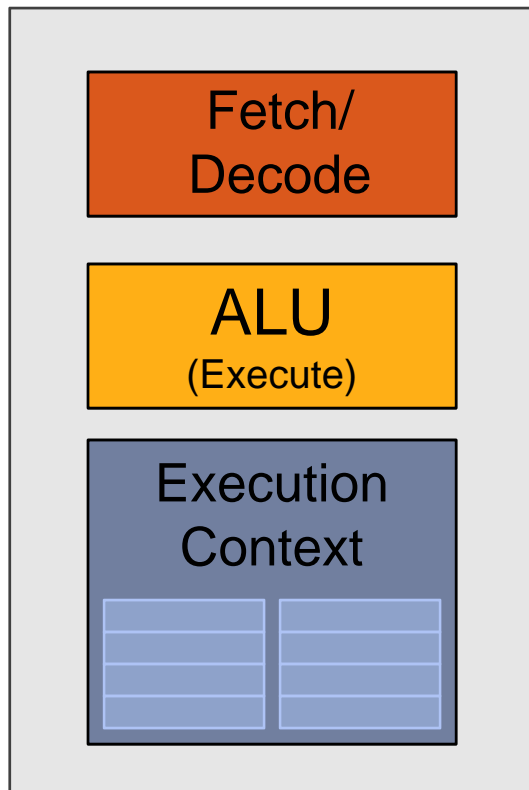
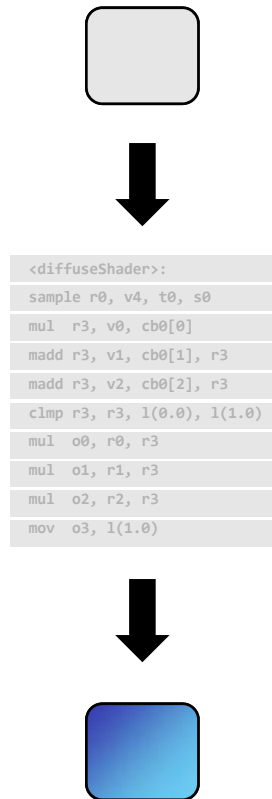


Slimming down

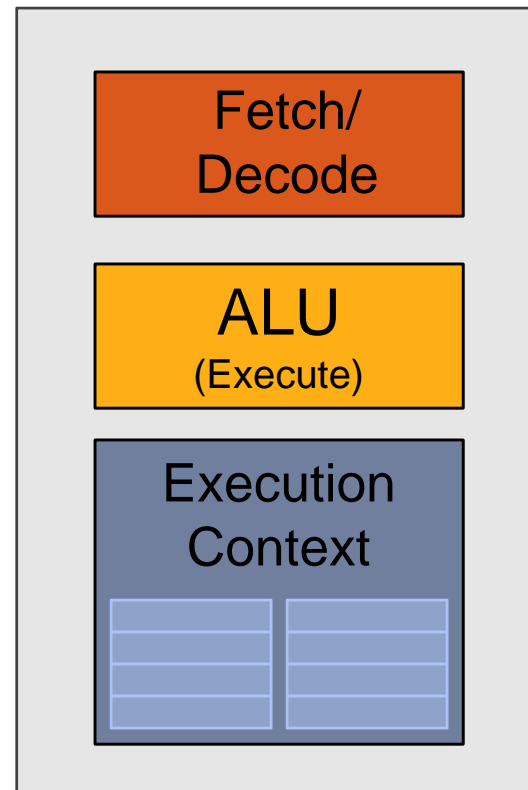
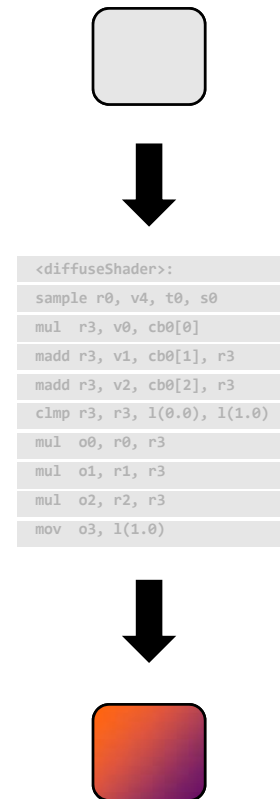


Two cores (two fragments in parallel)

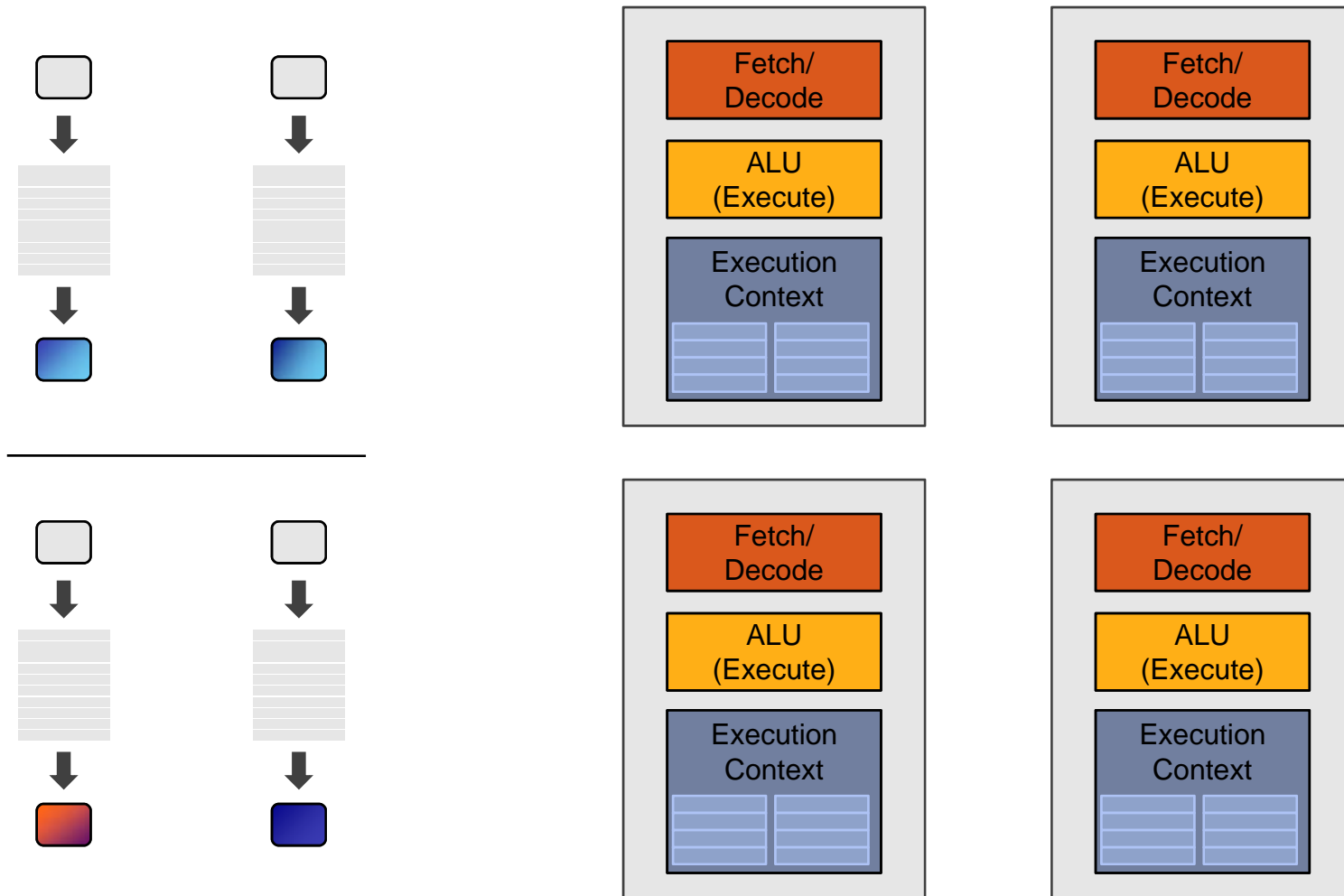
fragment 1



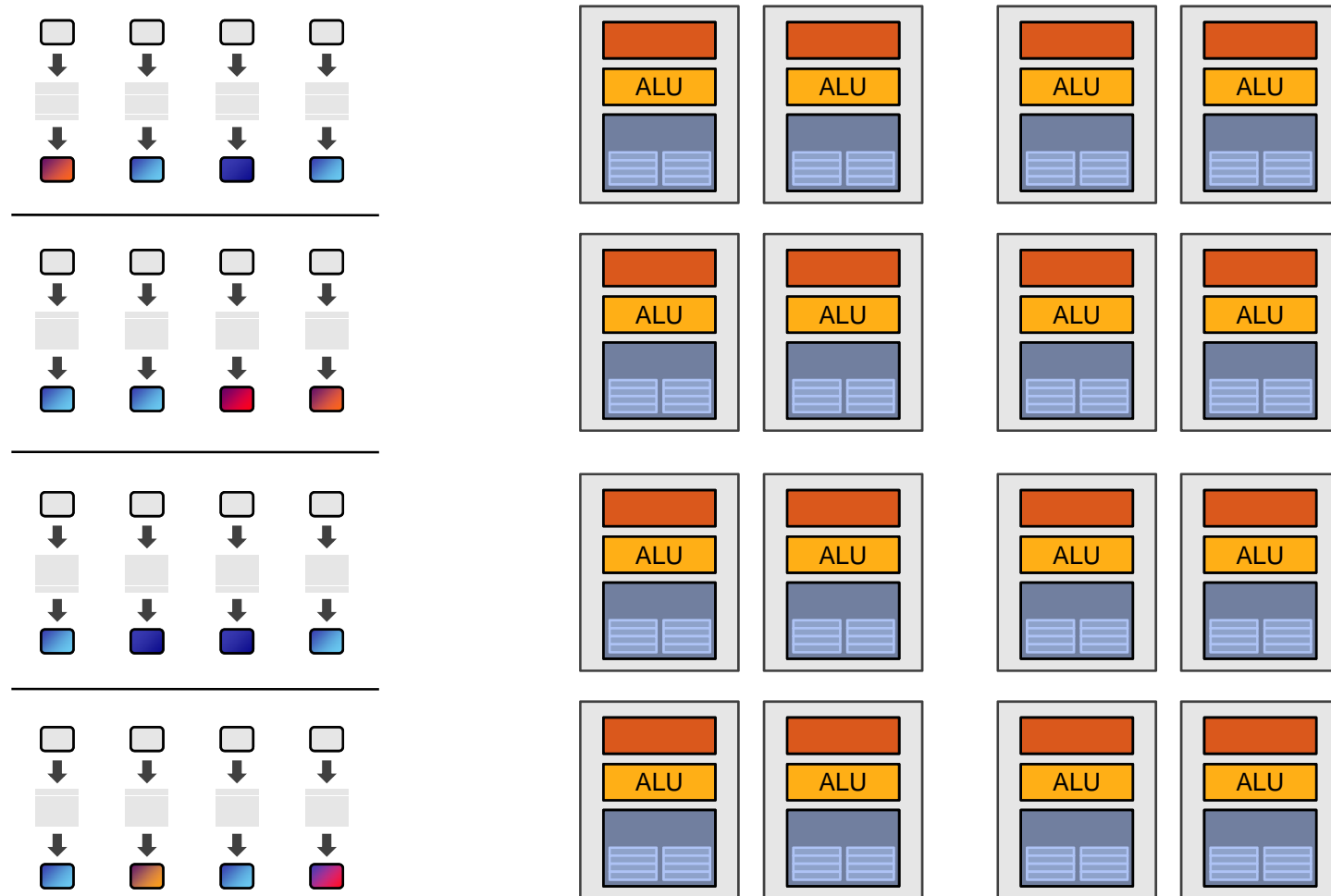
fragment 2



Four cores (four fragments in parallel)

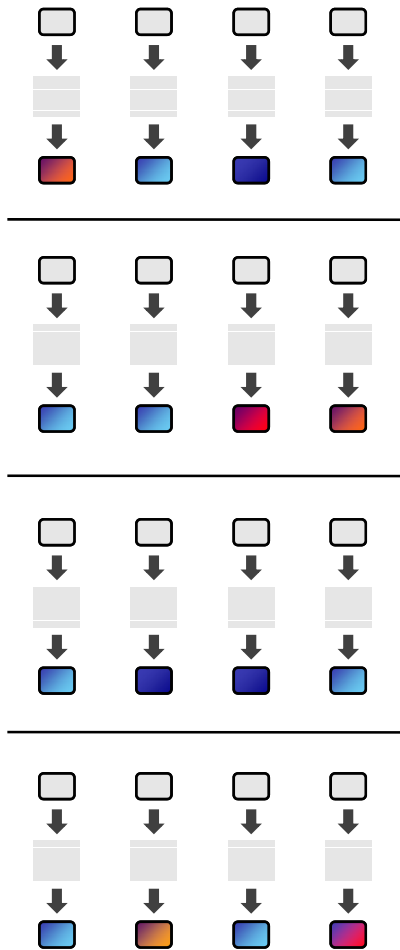


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

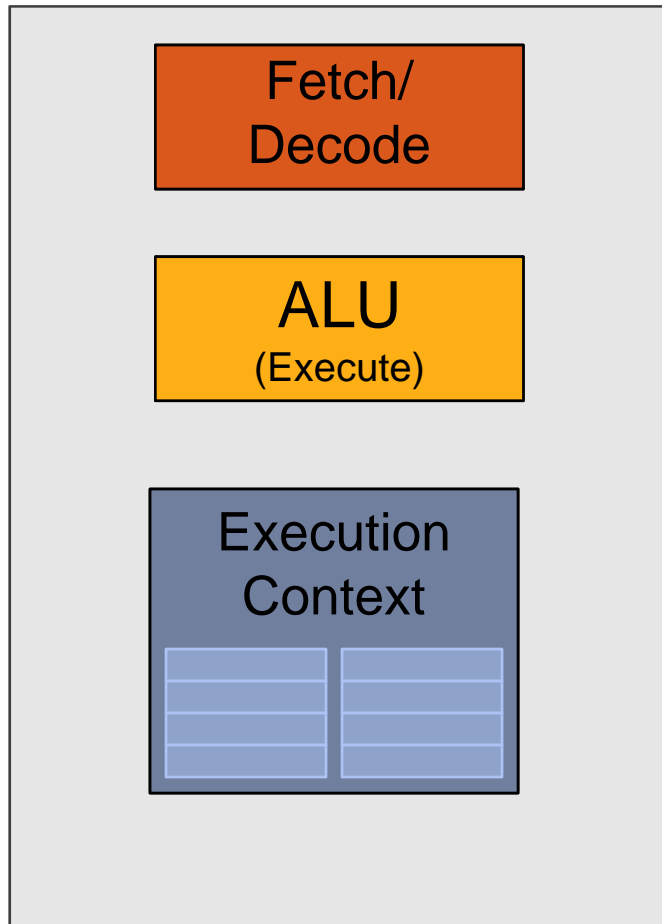
Instruction stream sharing



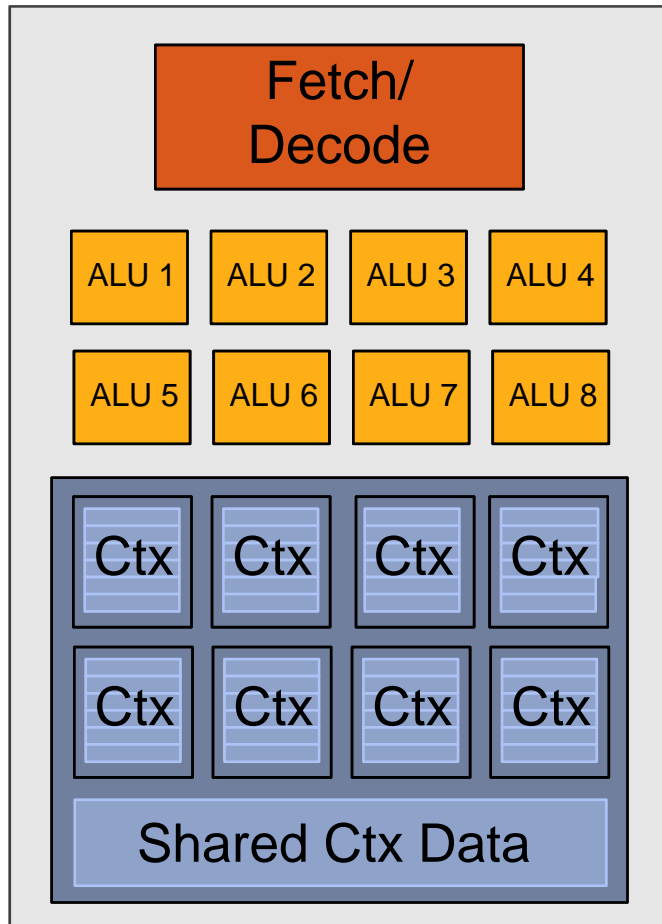
But... many fragments should be able to share an instruction stream!

```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clamp r3, r3, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```

Recall: simple processing core



Add ALUs



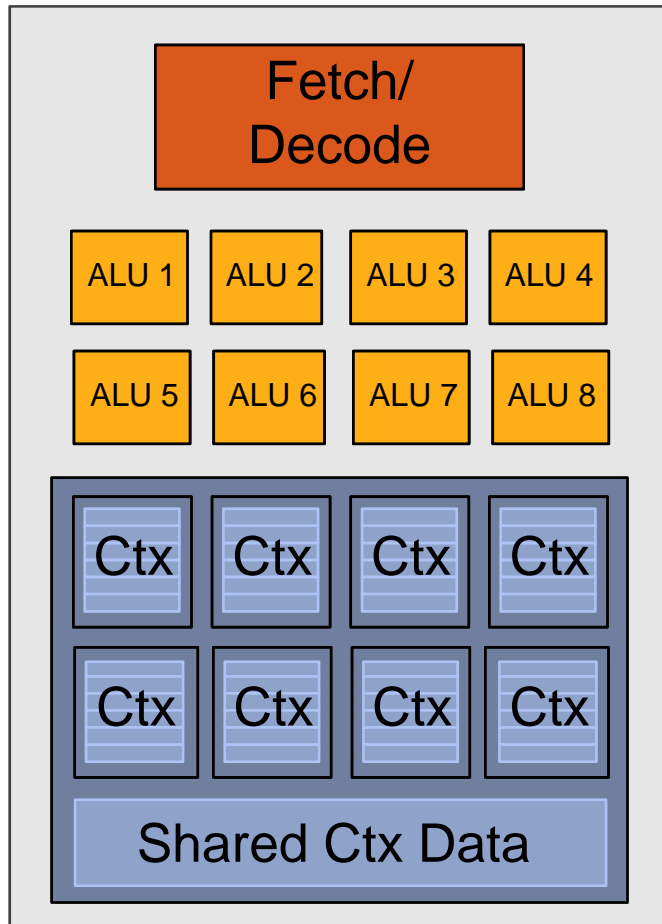
Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

(or SIMT, SPMD)

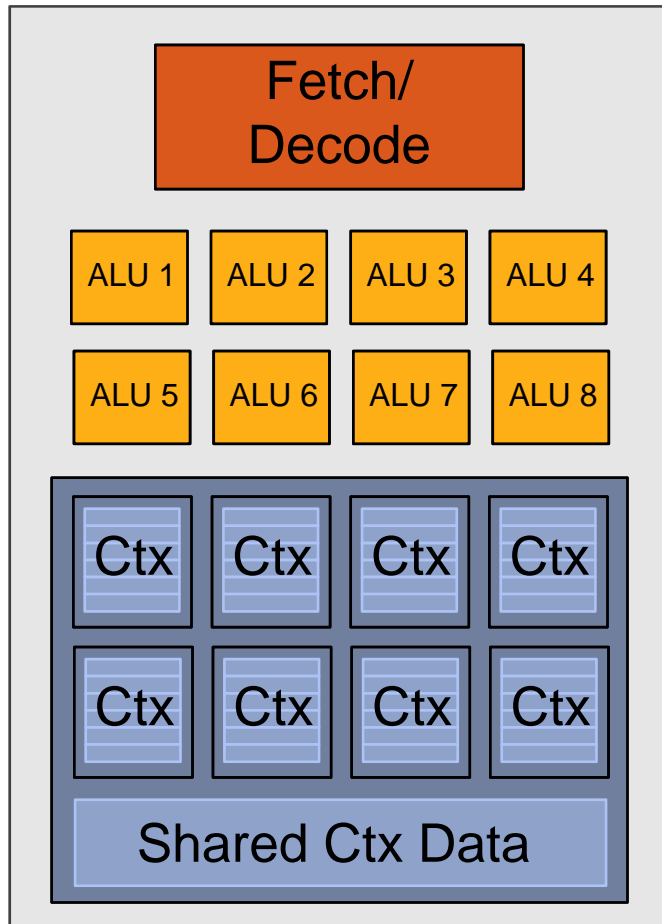
Modifying the shader



```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clamp r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```

Original compiled shader:
Processes one fragment
using scalar ops on scalar
registers

Modifying the shader

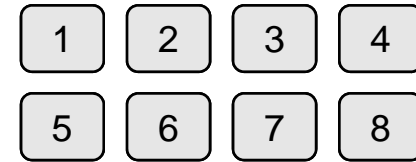
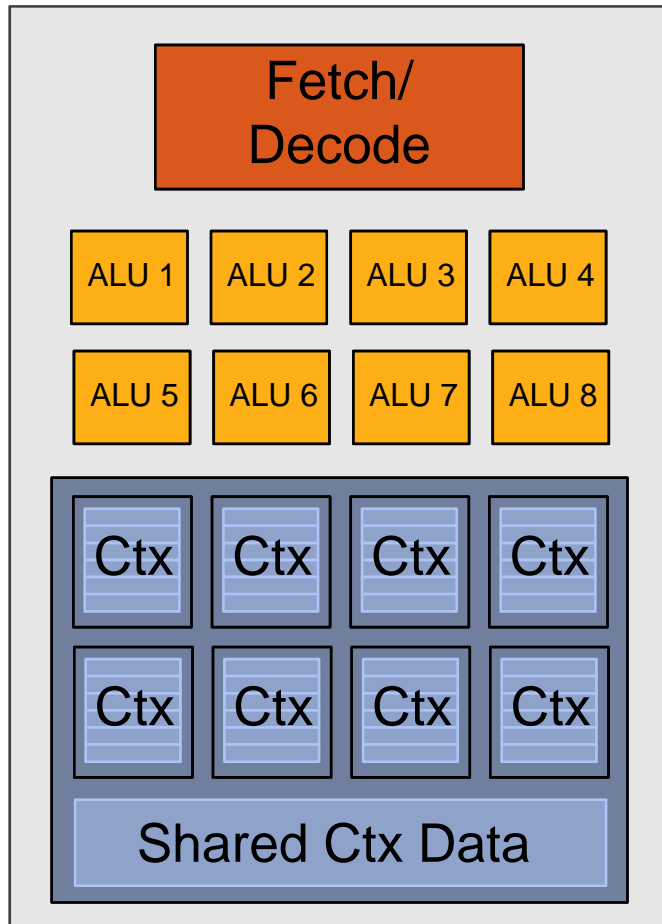


```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul vec_r3, vec_v0, cb0[0]  
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul vec_o0, vec_r0, vec_r3  
VEC8_mul vec_o1, vec_r1, vec_r3  
VEC8_mul vec_o2, vec_r2, vec_r3  
VEC8_mov vec_o3, 1(1.0)
```

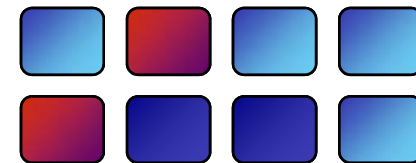
New compiled shader:

Processes 8 fragments
using vector ops on vector
registers

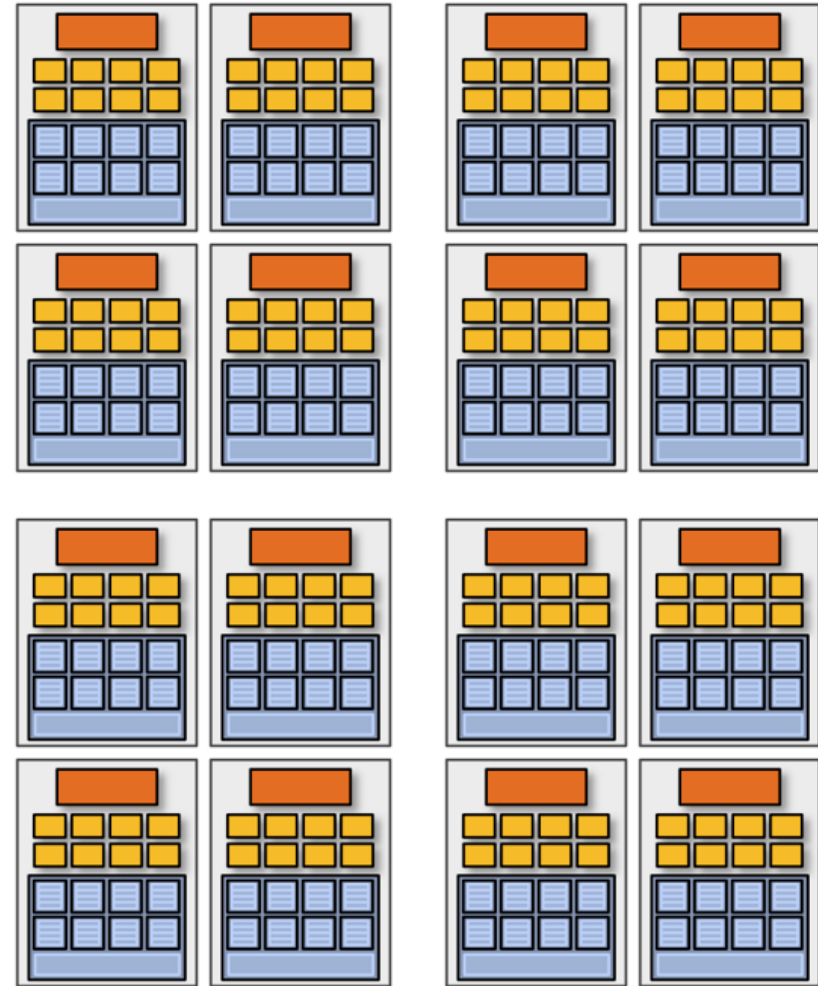
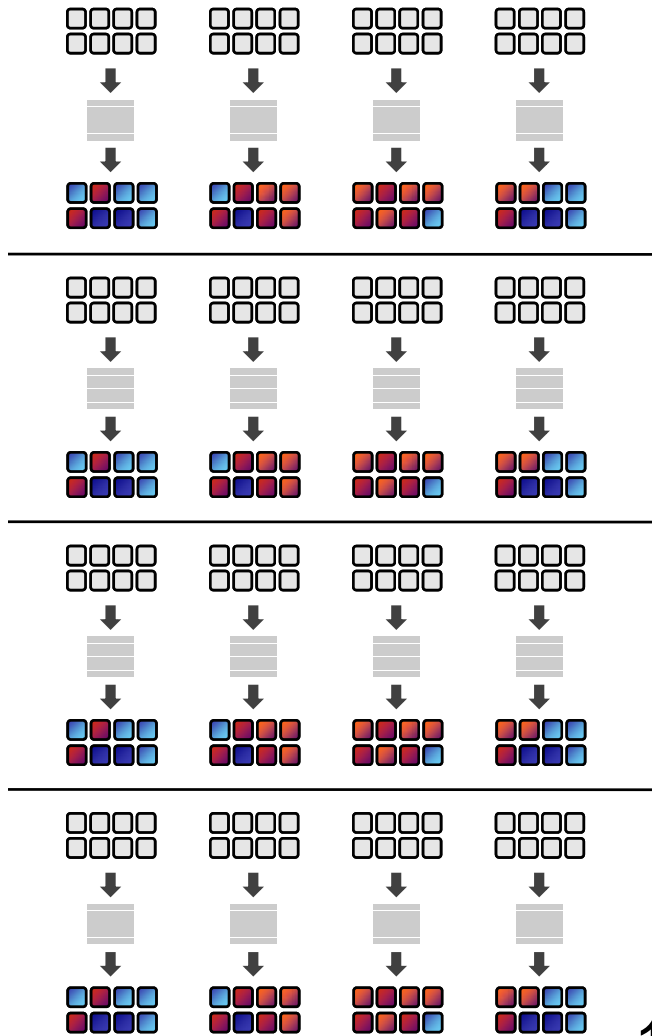
Modifying the shader



```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul vec_r3, vec_v0, cb0[0]  
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul vec_o0, vec_r0, vec_r3  
VEC8_mul vec_o1, vec_r1, vec_r3  
VEC8_mul vec_o2, vec_r2, vec_r3  
VEC8_mov vec_o3, 1(1.0)
```

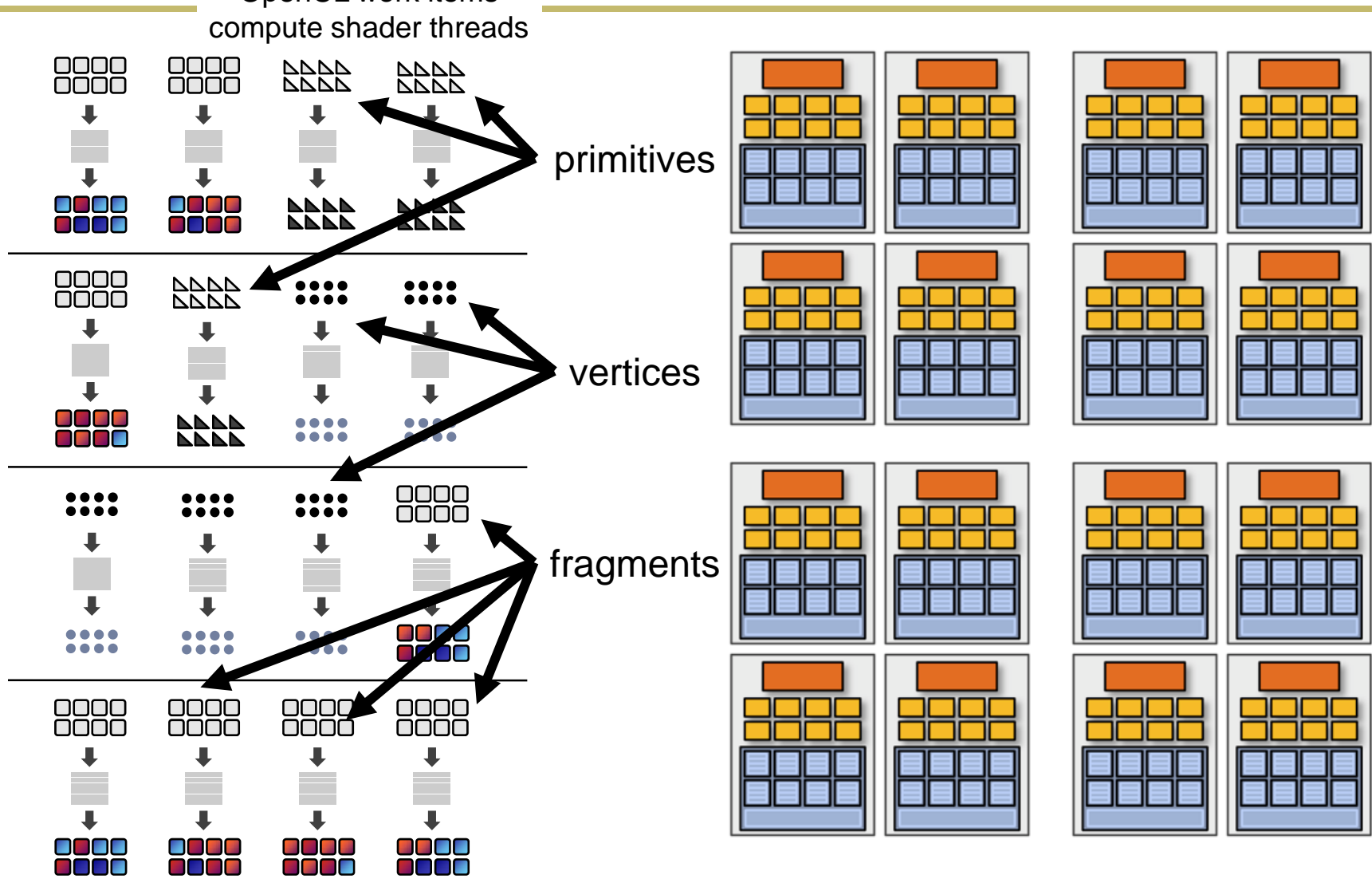


128 fragments in parallel



16 cores = 128 ALUs
= 16 simultaneous instruction streams

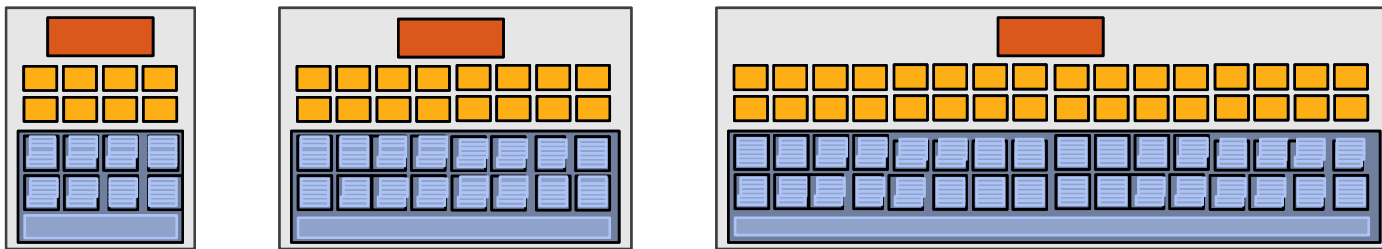
128 [vertices / fragments primitives CUDA threads OpenCL work items compute shader threads] in parallel



Clarification

SIMD processing does not imply SIMD instructions

- Option 1: Explicit vector instructions
 - Intel/AMD x86 SSE, Intel Larrabee
- Option 2: Scalar instructions, implicit HW vectorization
 - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
 - NVIDIA GeForce (“SIMT” warps), AMD Radeon architectures



In practice: 16 to 64 fragments share an instruction stream

Thank you.

Thanks for slides

- Kayvon Fatahalian