

CS 380 - GPU and GPGPU Programming

Lecture 5: GPU Architecture, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 18)



Read (required):

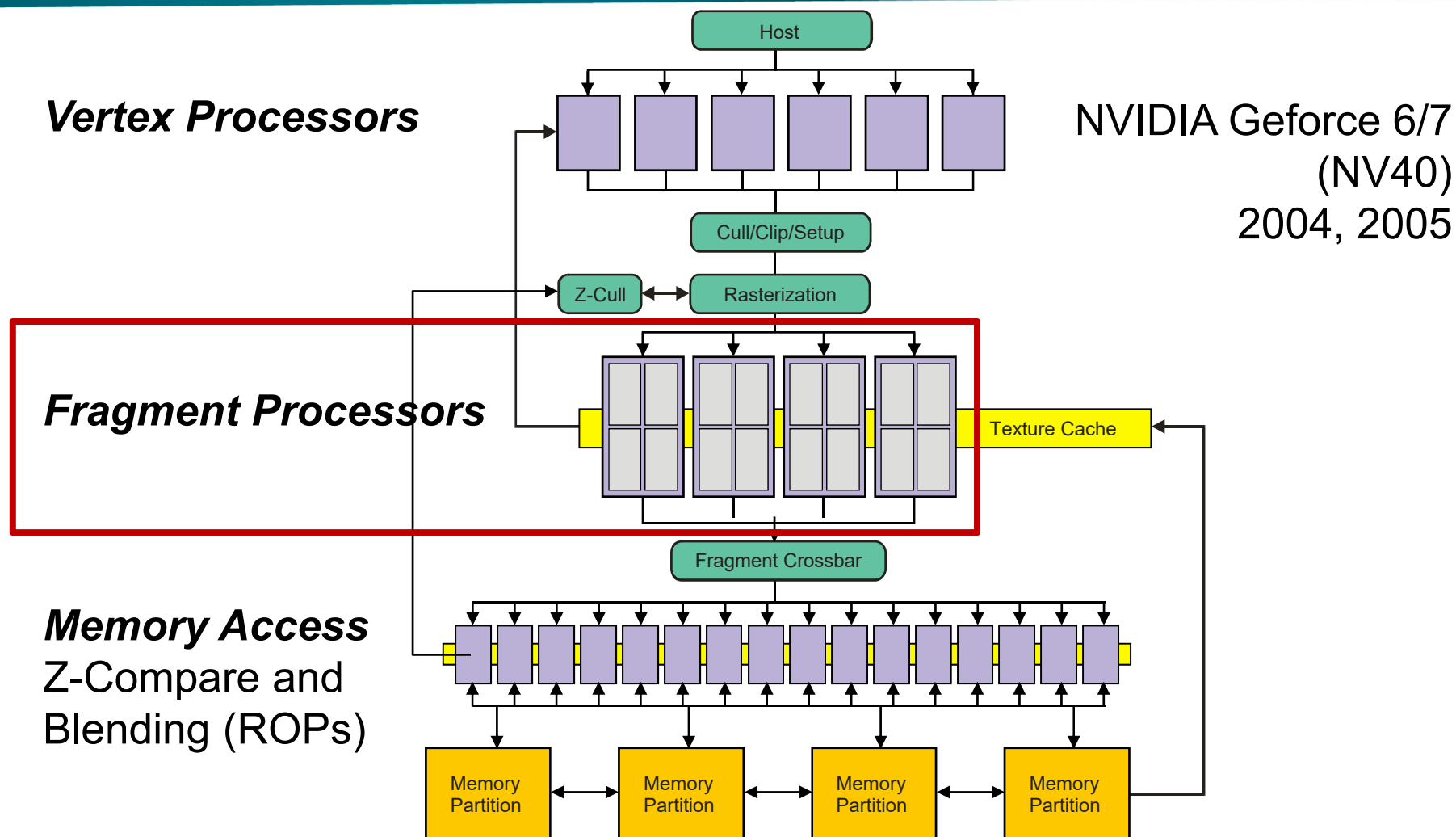
- Programming Mass. Parallel Proc. book, 4th ed., Chapter 1 (*Introduction*)
- Programming Mass. Parallel Proc. book, 2nd ed., Chapter 2 (*History of GPU Computing*)
- OpenGL 4 Shading Language Cookbook, Chapter 2
- OpenGL Shading Language 4.6 (current: Jul 10, 2019) specification: Chapter 2
<https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>
- Download OpenGL 4.6 (current: May 5, 2022) specification
<https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf>

Read (optional):

- Orange (GLSL) book, Chapter 7 (OpenGL Shading Language API)
- OpenGL 4 Shading Language Cookbook, Chapter 1

GPU Architecture

GPU Structure Before Unified Shaders

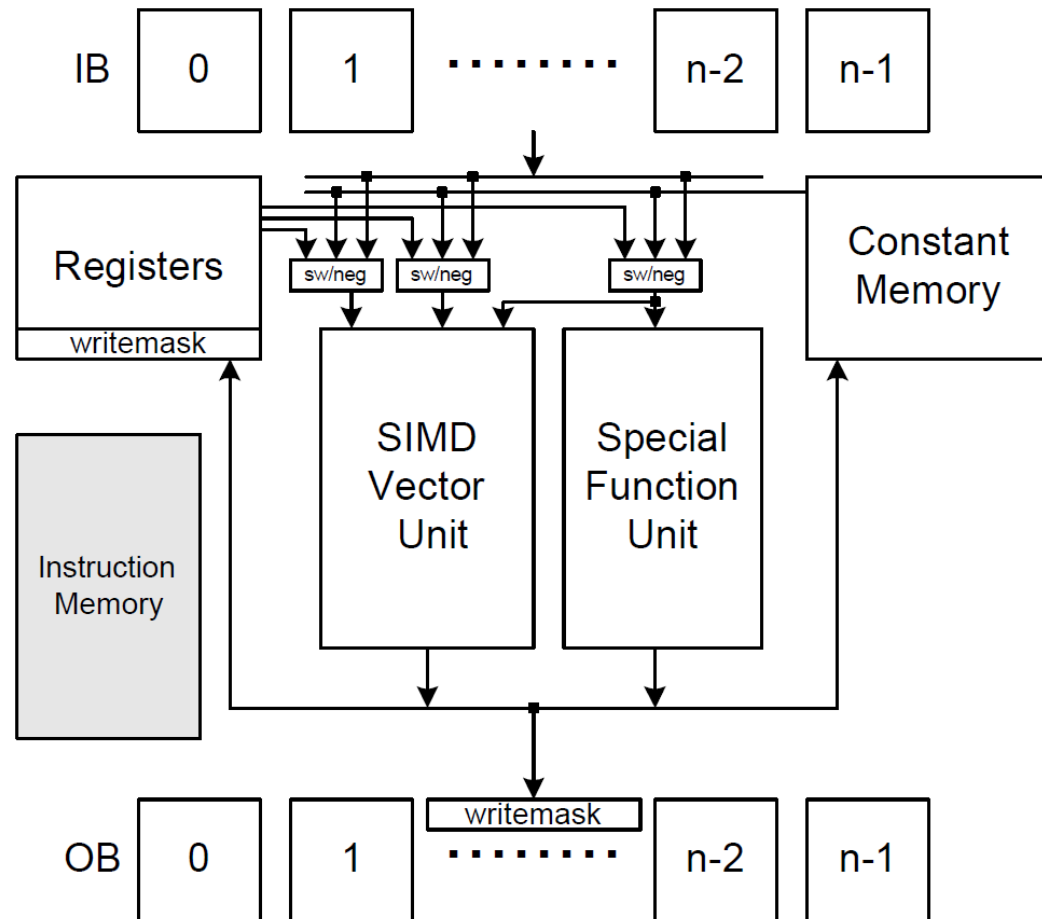


Legacy Vertex Shading Unit (1)



Geforce 3 (NV20), 2001

- floating point
4-vector
vertex engine
- still very
instructive for
understanding
GPUs in general



Lindholm et al., A User-Programmable Vertex Engine, SIGGRAPH 2001

Legacy Vertex Shading Unit (3)



Vector instruction set, very few instructions; **no branching yet!**

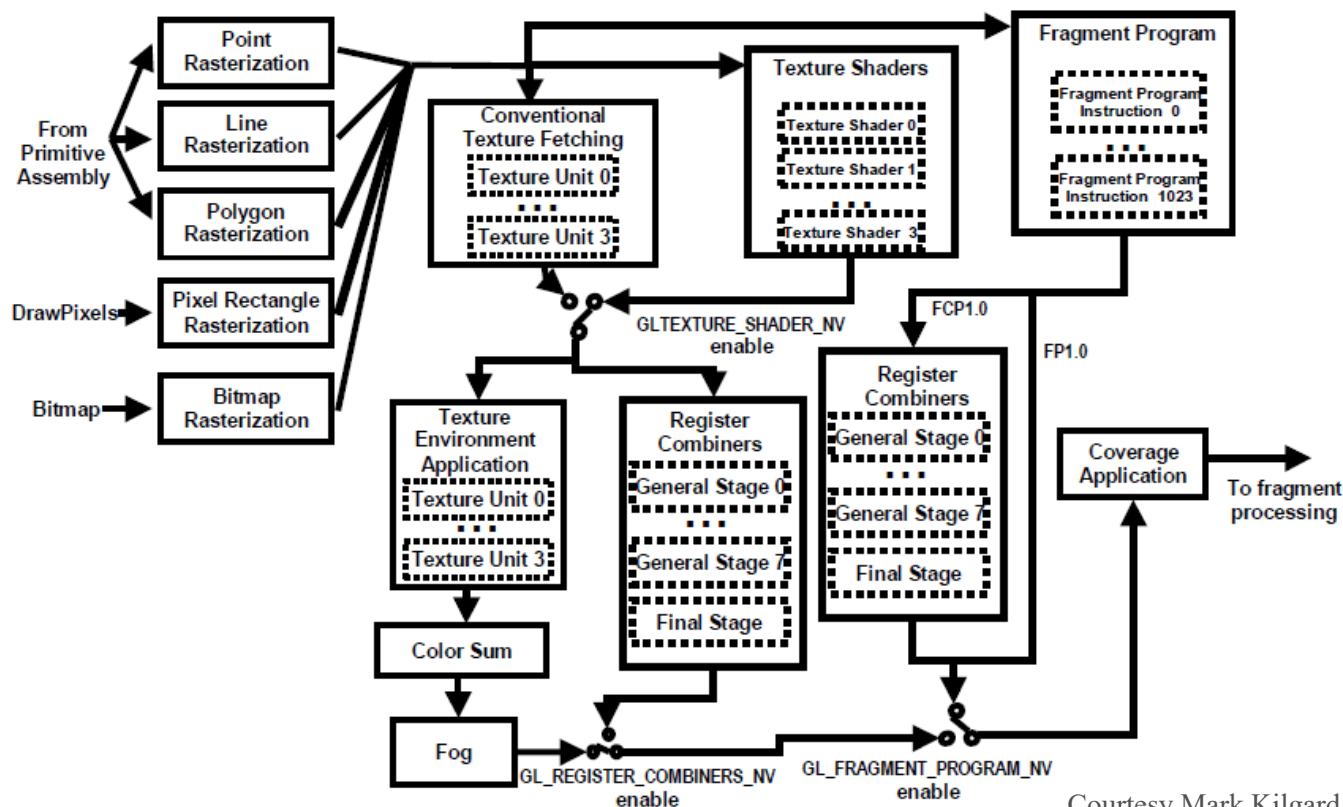
OpCode	Full Name	Description
MOV	Move	vector -> vector
MUL	Multiply	vector -> vector
ADD	Add	vector -> vector
MAD	Multiply and add	vector -> vector
DST	Distance	vector -> vector
MIN	Minimum	vector -> vector
MAX	Maximum	vector -> vector
SLT	Set on less than	vector -> vector
SGE	Set on greater or equal	vector -> vector
RCP	Reciprocal	scalar-> replicated scalar
RSQ	Reciprocal square root	scalar-> replicated scalar
DP3	3 term dot product	vector-> replicated scalar
DP4	4 term dot product	vector-> replicated scalar
LOG	Log base 2	miscellaneous
EXP	Exp base 2	miscellaneous
LIT	Phong lighting	miscellaneous
ARL	Address register load	miscellaneous

Fast Forward to Programm. Fragment Shading



NV30 OpenGL Fragment Texturing & Coloring

GeForce FX (5),
2003



NVIDIA Proprietary

Courtesy Mark Kilgard

Legacy Fragment Shading Unit (1)



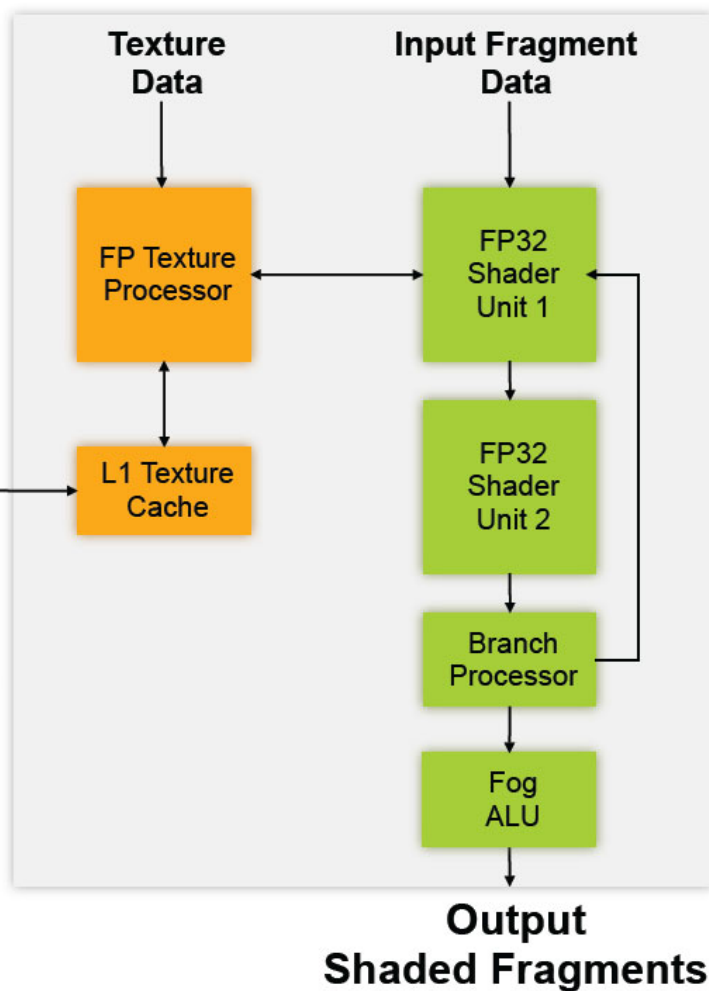
GeForce 6 (NV40), 2004

- dynamic branching

Texture Filter
Bi / Tri / Aniso
1 texture @ full speed
4-tap filter @ full speed
16:1 Aniso w/ Trilinear (128-tap)
FP16 Texture Filtering

L2 Texture
Cache

SIMD Architecture
Dual Issue / Co-Issue
FP32 Computation
Shader Model 3.0



Shader Unit 1
4 FP Ops / pixel
Dual/Co-Issue
Texture Address Calc
Free fp16 normalize
+ mini ALU

Shader Unit 2
4 FP Ops / pixel
Dual/Co-Issue
+ mini ALU

Legacy Fragment Shading Unit (2)



Example code

```
!!ARBfp1.0

ATTRIB unit_tc = fragment.texcoord[ 0 ];
PARAM  mvp_inv[] = { state.matrix.mvp.inverse };
PARAM  constants = {0, 0.999, 1, 2};

TEMP pos_win, temp;

TEX pos_win.z, unit_tc, texture[ 1 ], 2D;

ADD pos_win.w, constants.y, -pos_win.z;
KIL pos_win.w;

MOV result.color.w, pos_win.z;

MOV pos_win.xyw, unit_tc;
MAD pos_win.xyz, pos_win, constants.a, -constants.b;

DP4 temp.w, mvp_inv[ 3 ], pos_win;
RCP temp.w, temp.w;

MUL pos_win, pos_win, temp.w;

DP4 result.color.x, mvp_inv[ 0 ], pos_win;
DP4 result.color.y, mvp_inv[ 1 ], pos_win;
DP4 result.color.z, mvp_inv[ 2 ], pos_win;

END
```

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, l(0.0), l(1.0)  
mul   o0, r0, r3  
mul   o1, r1, r3  
mul   o2, r2, r3  
mov   o3, l(1.0)
```



1 shaded fragment output record



Per-Pixel(Fragment) Lighting

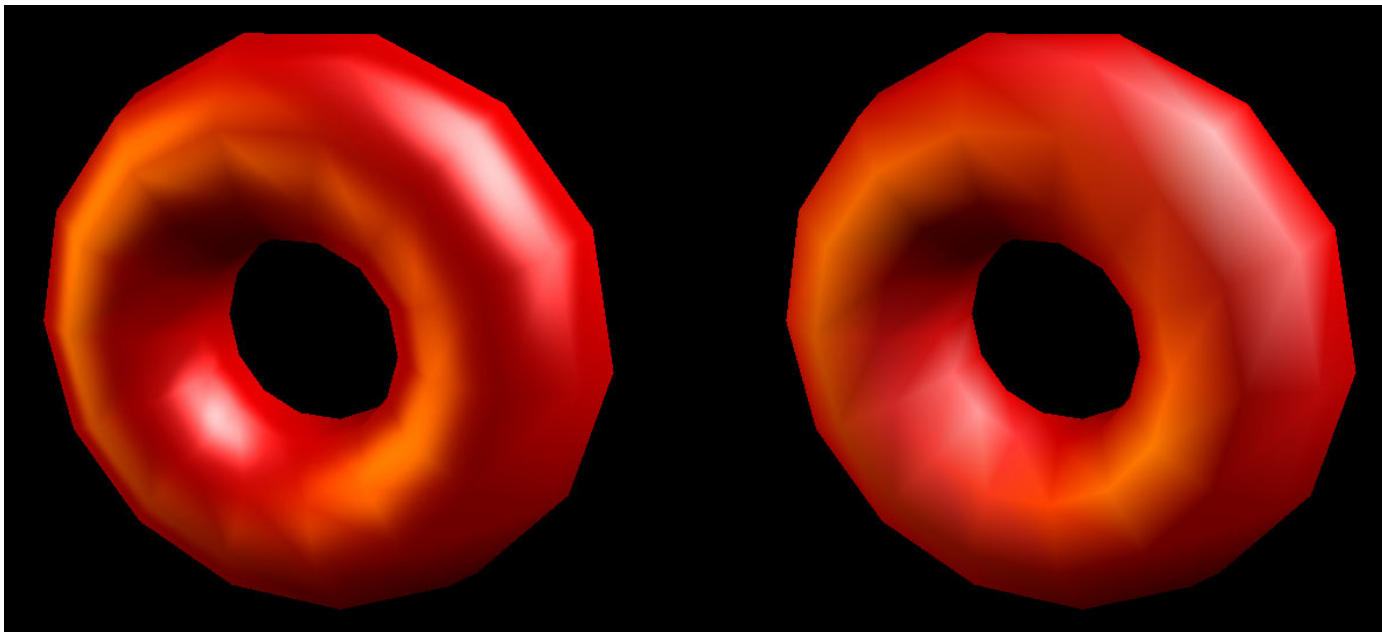


Simulating smooth surfaces by calculating illumination for each fragment

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;
}
```

GPU Architecture: General Architecture



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?

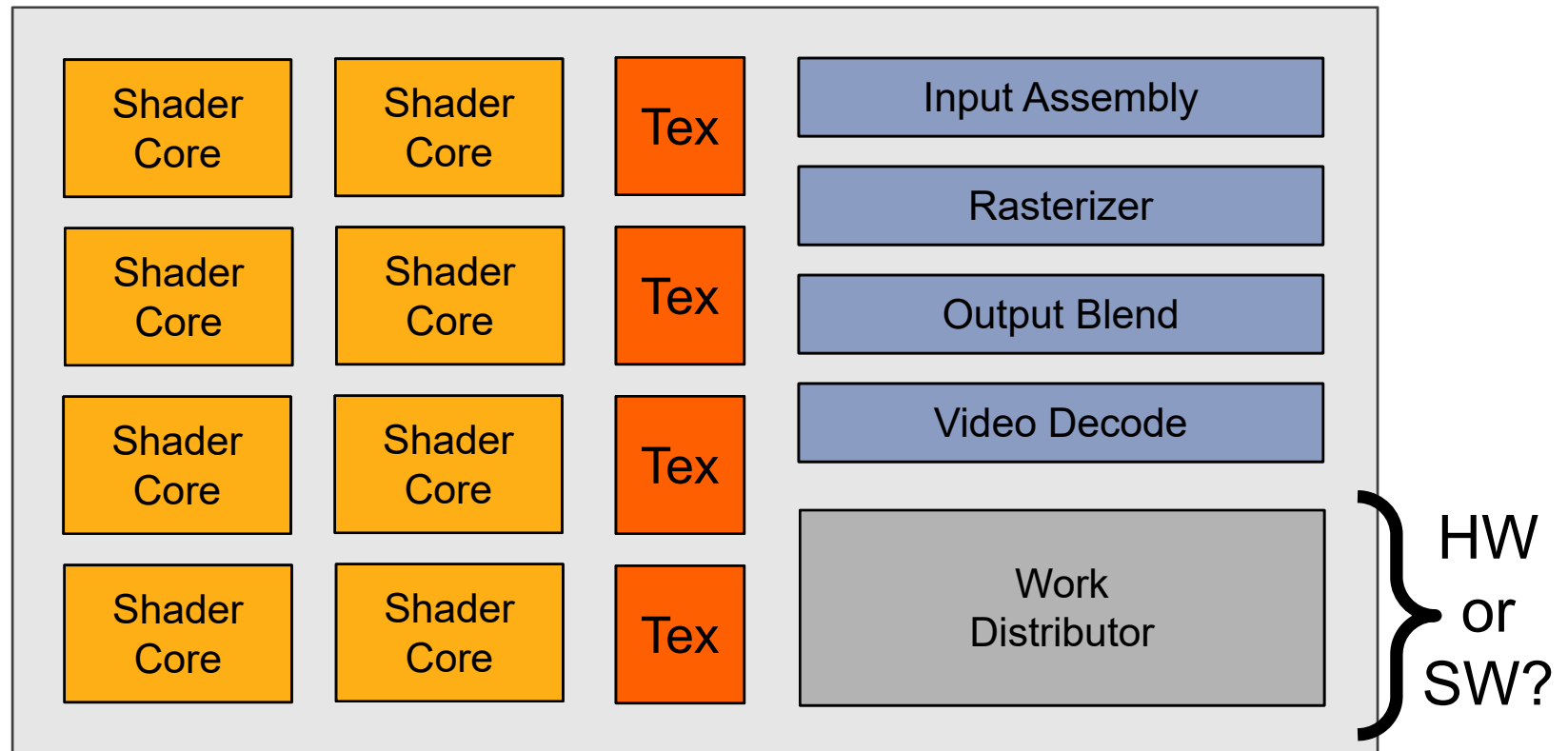
Where this is going...



Summary: three key ideas for high-throughput execution

- 1. Use many “slimmed down cores,” run them in parallel**
- 2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)**
 - Option 1: Explicit SIMD vector instructions**
 - Option 2: Implicit sharing managed by hardware**
- 3. Avoid latency stalls by interleaving execution of many groups of fragments**
 - When one group stalls, work on another group**

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);  
}
```



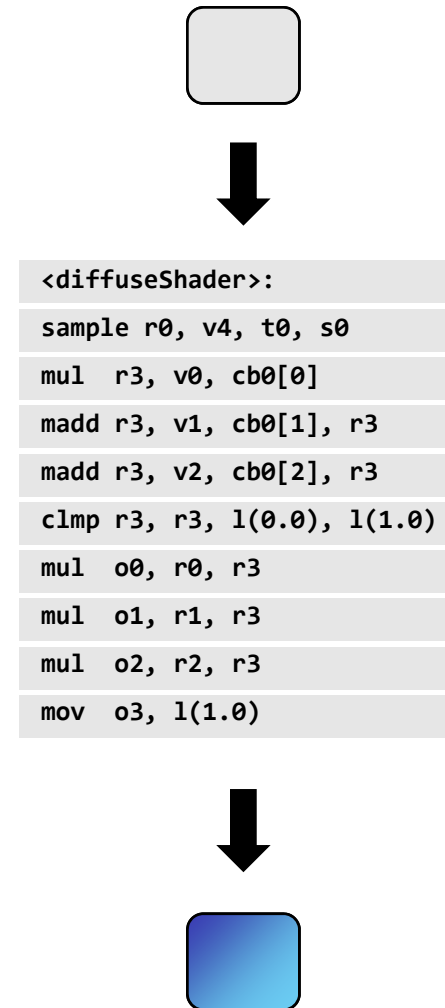
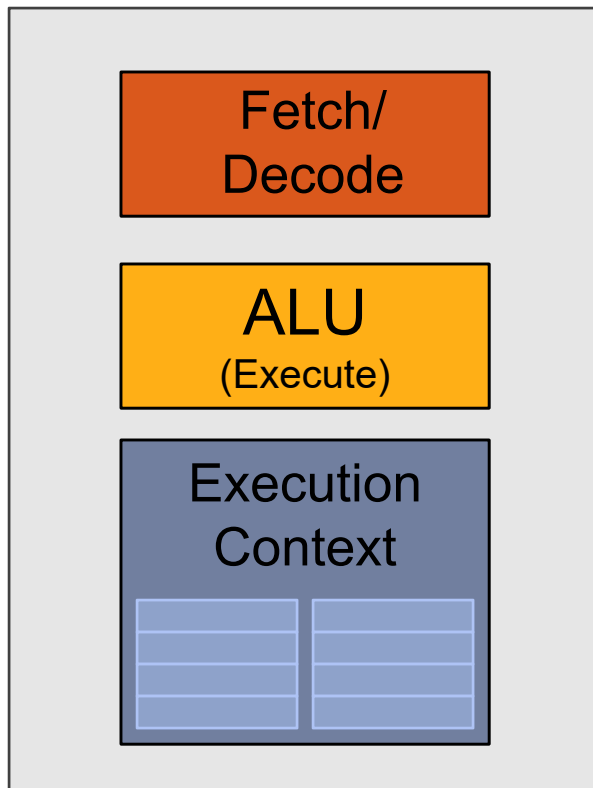
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul   r3, v0, cb0[0]  
madd  r3, v1, cb0[1], r3  
madd  r3, v2, cb0[2], r3  
clmp  r3, r3, 1(0.0), 1(1.0)  
mul   o0, r0, r3  
mul   o1, r1, r3  
mul   o2, r2, r3  
mov   o3, 1(1.0)
```



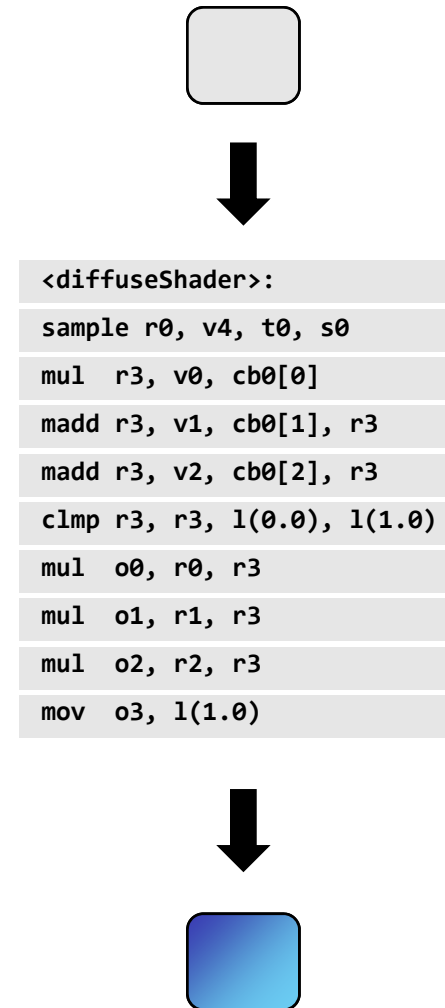
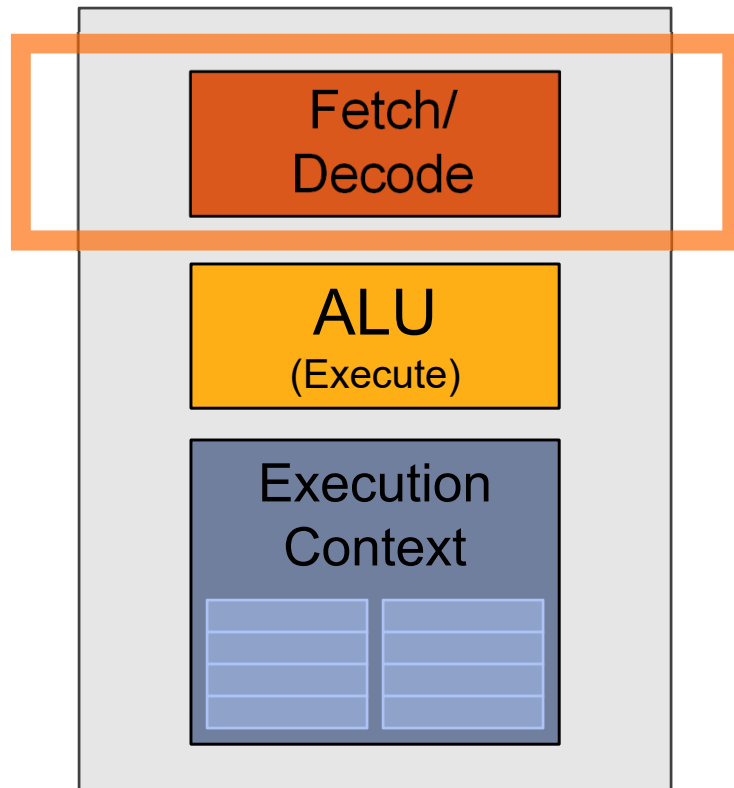
1 shaded fragment output record



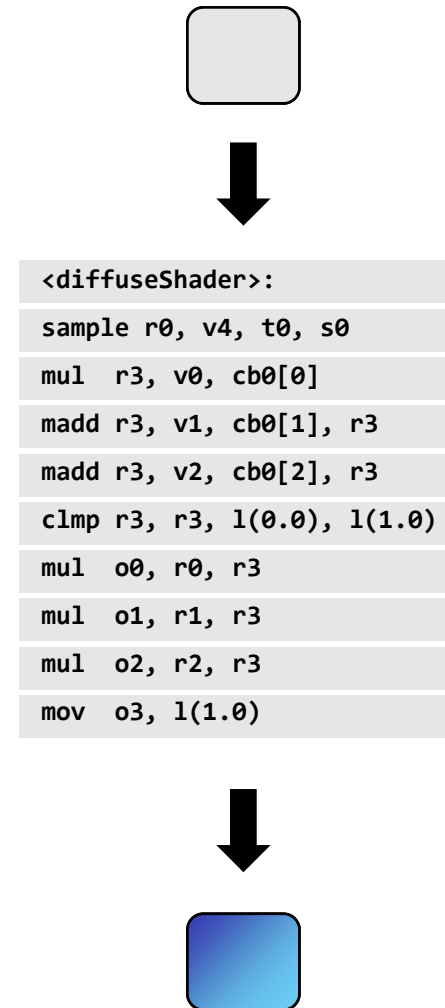
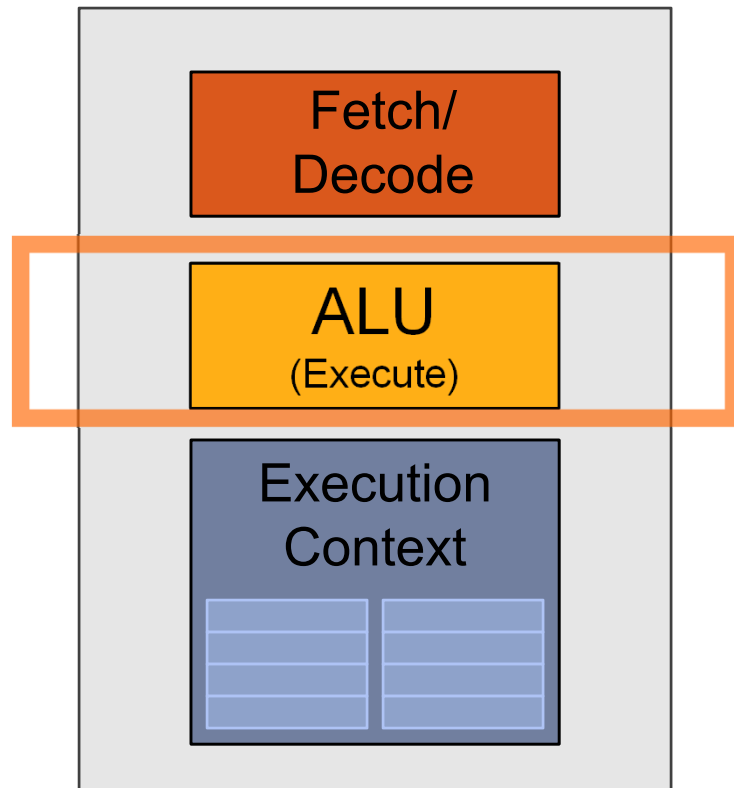
Execute shader



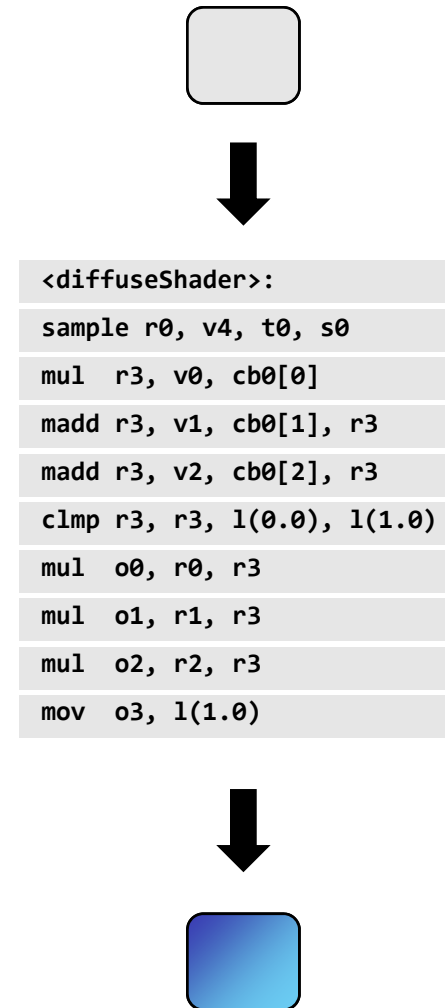
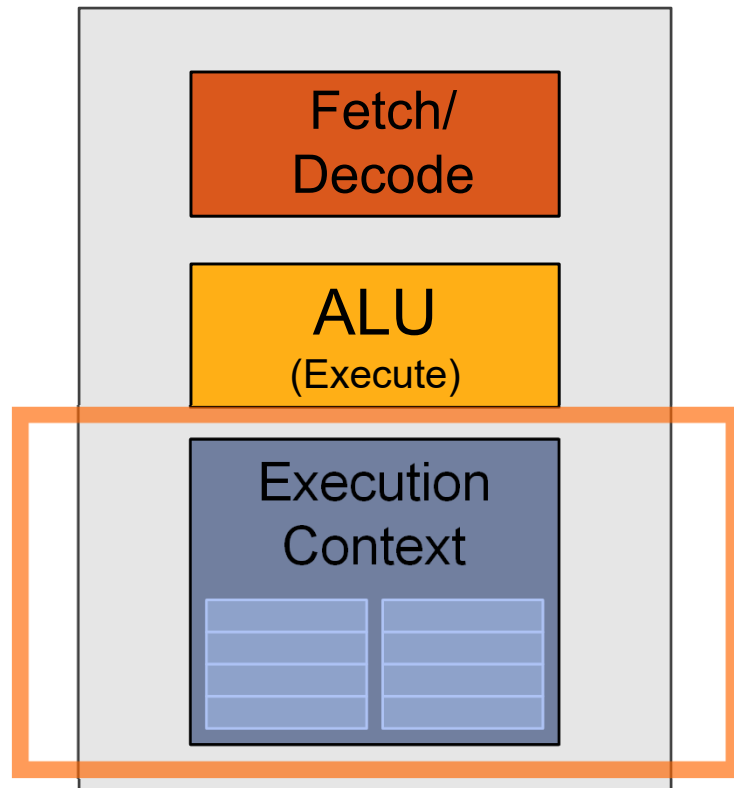
Execute shader



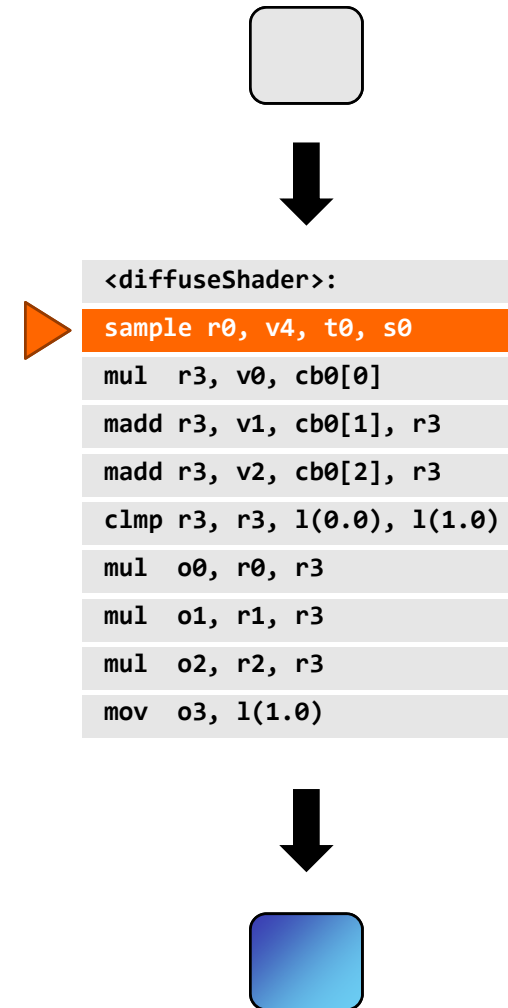
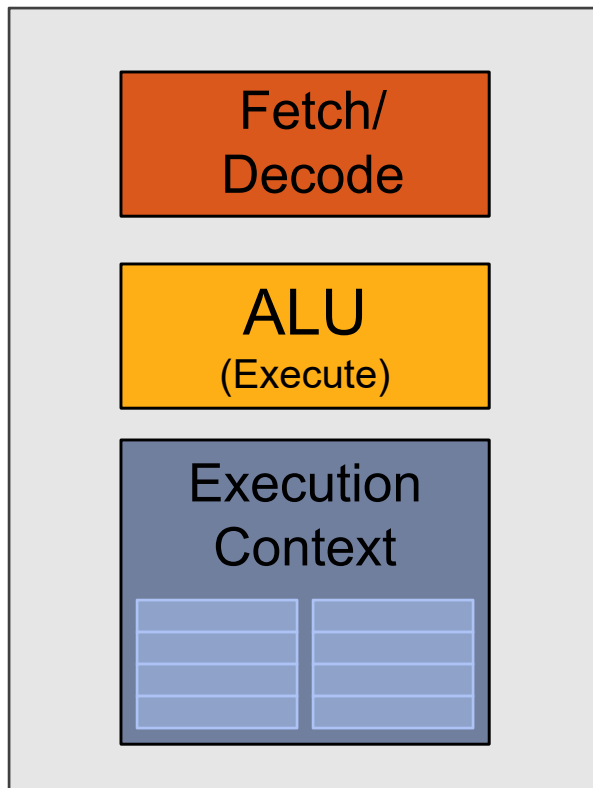
Execute shader



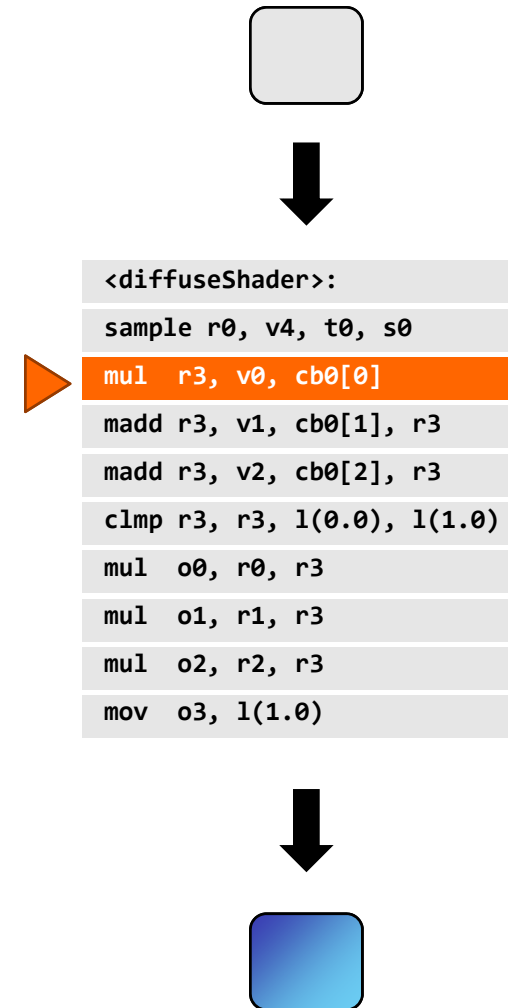
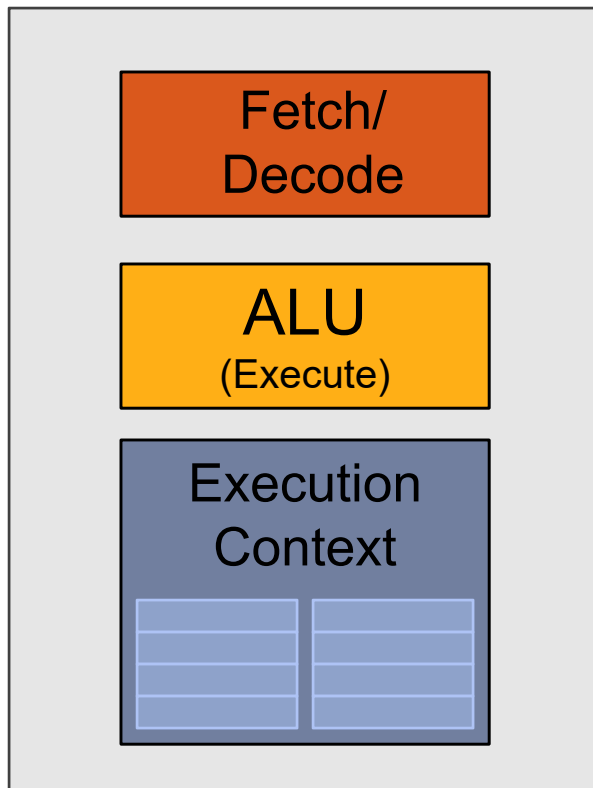
Execute shader



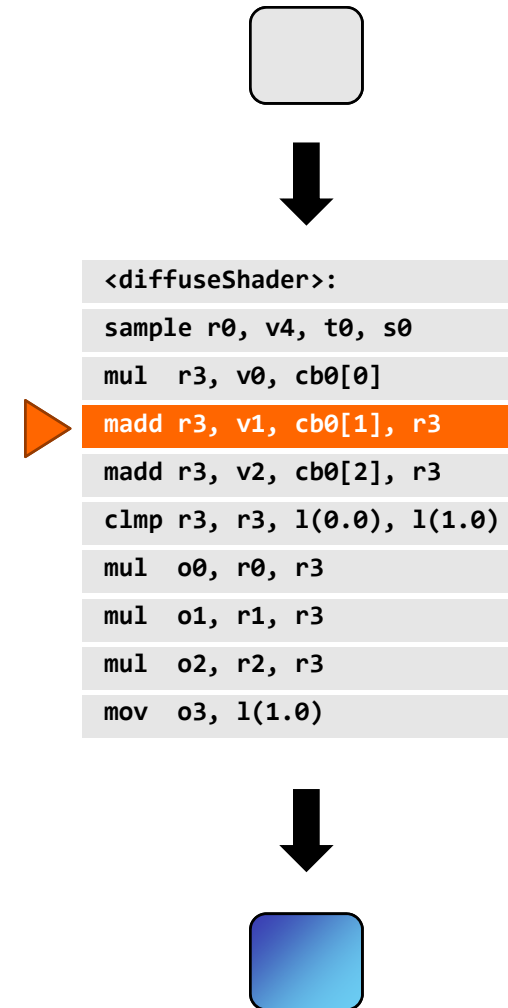
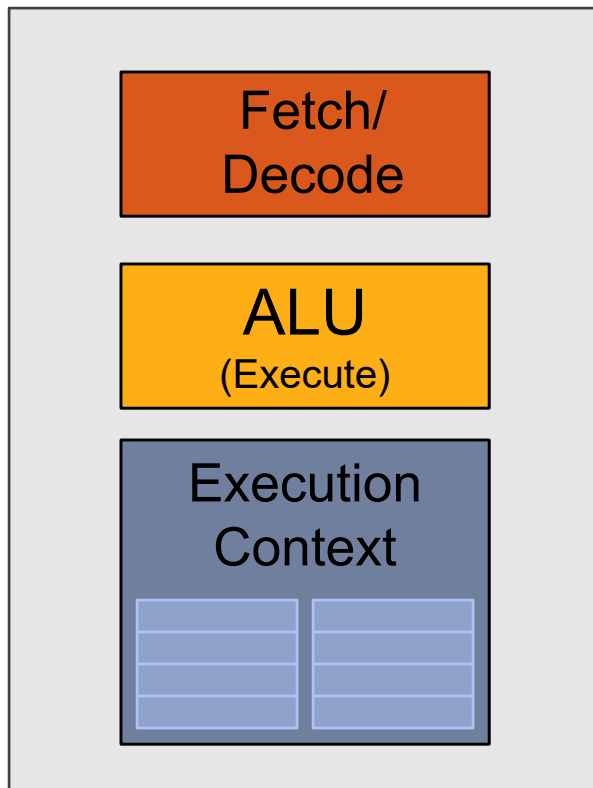
Execute shader



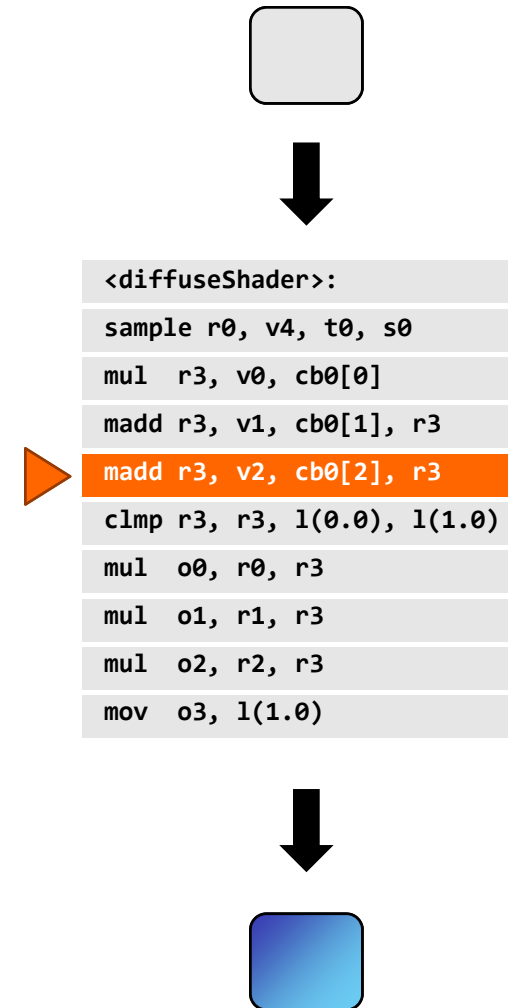
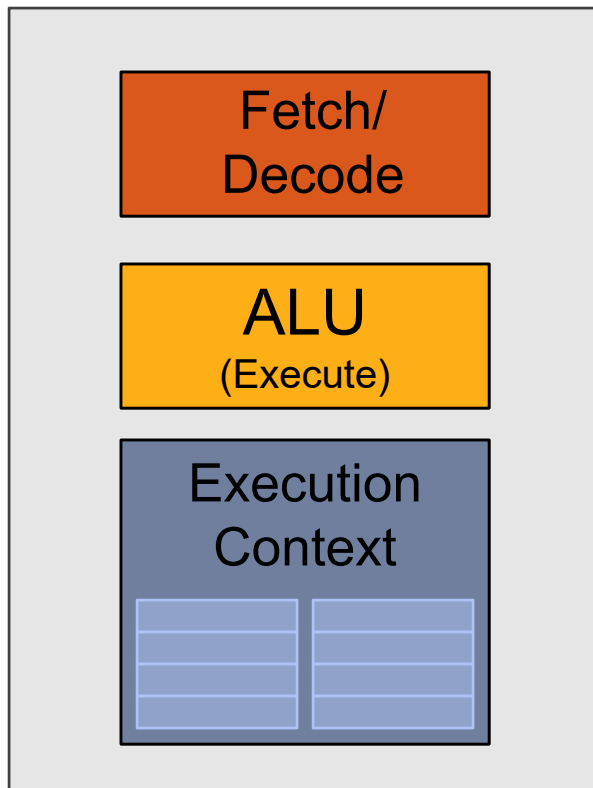
Execute shader



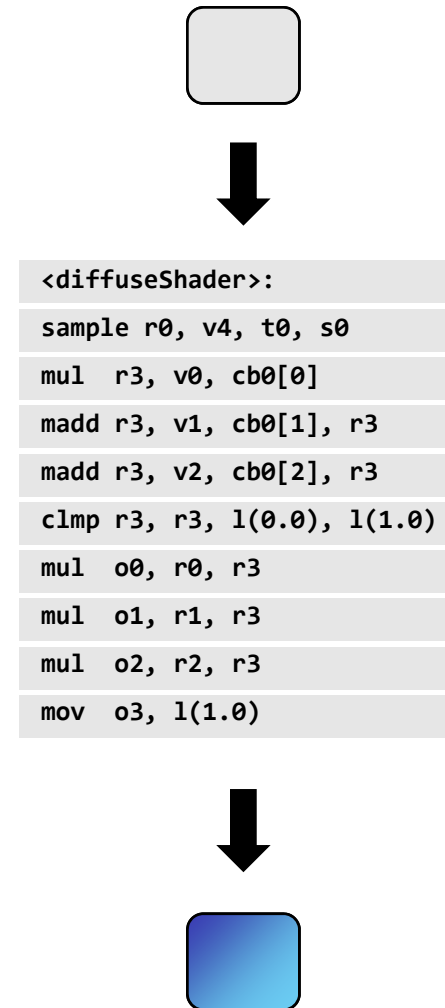
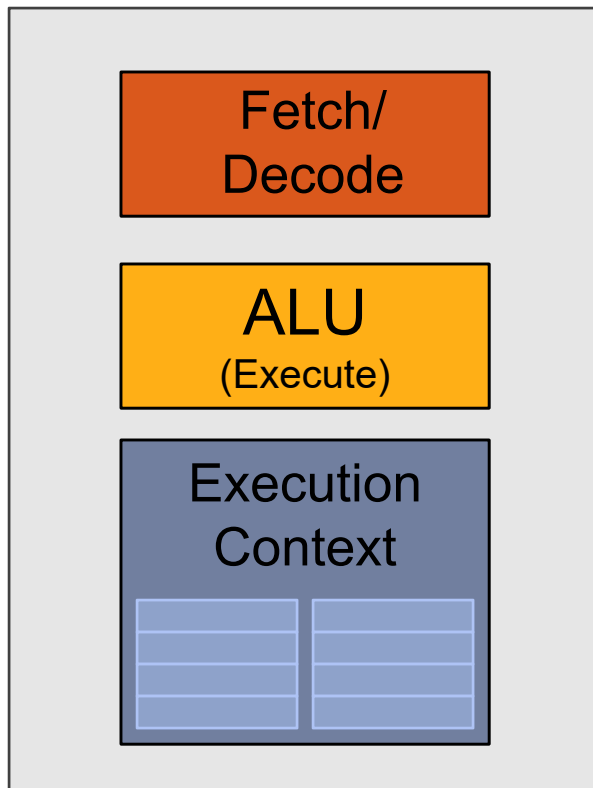
Execute shader



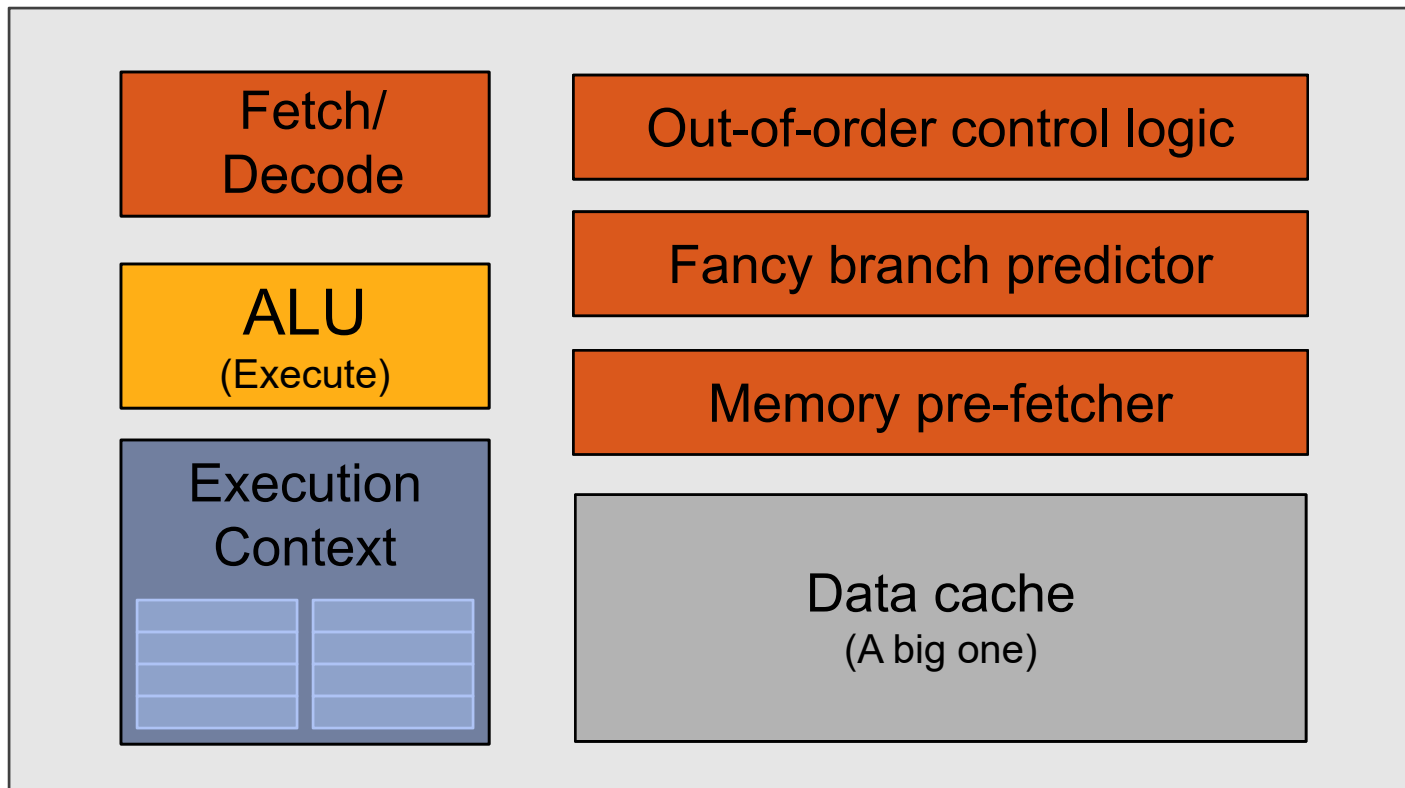
Execute shader



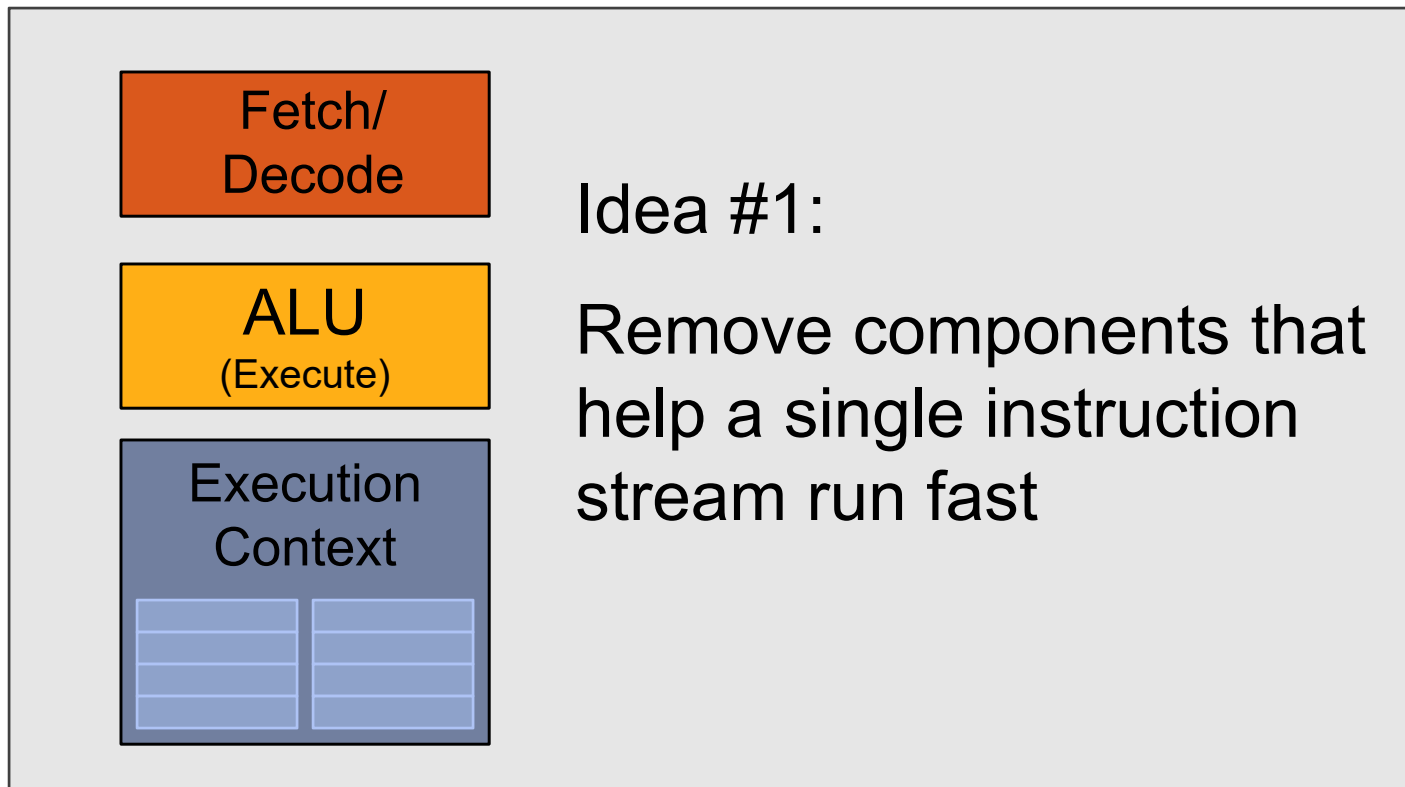
Execute shader



CPU-“style” cores

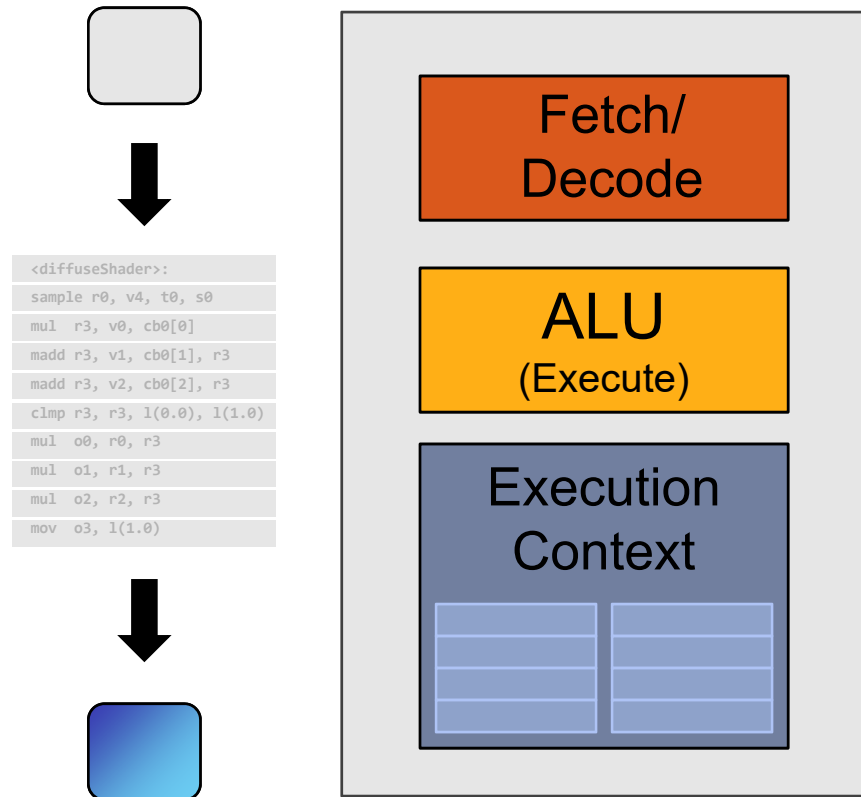


Idea #1: Slim down

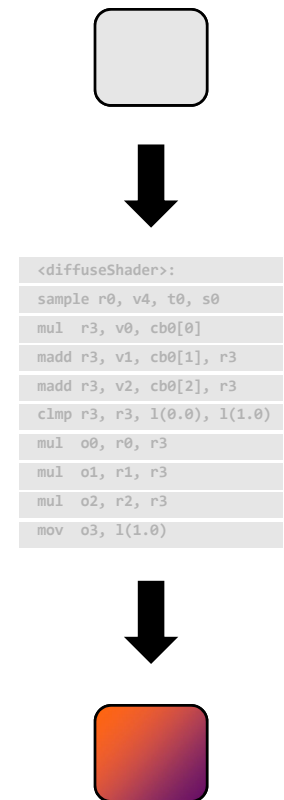


Two cores (two fragments in parallel)

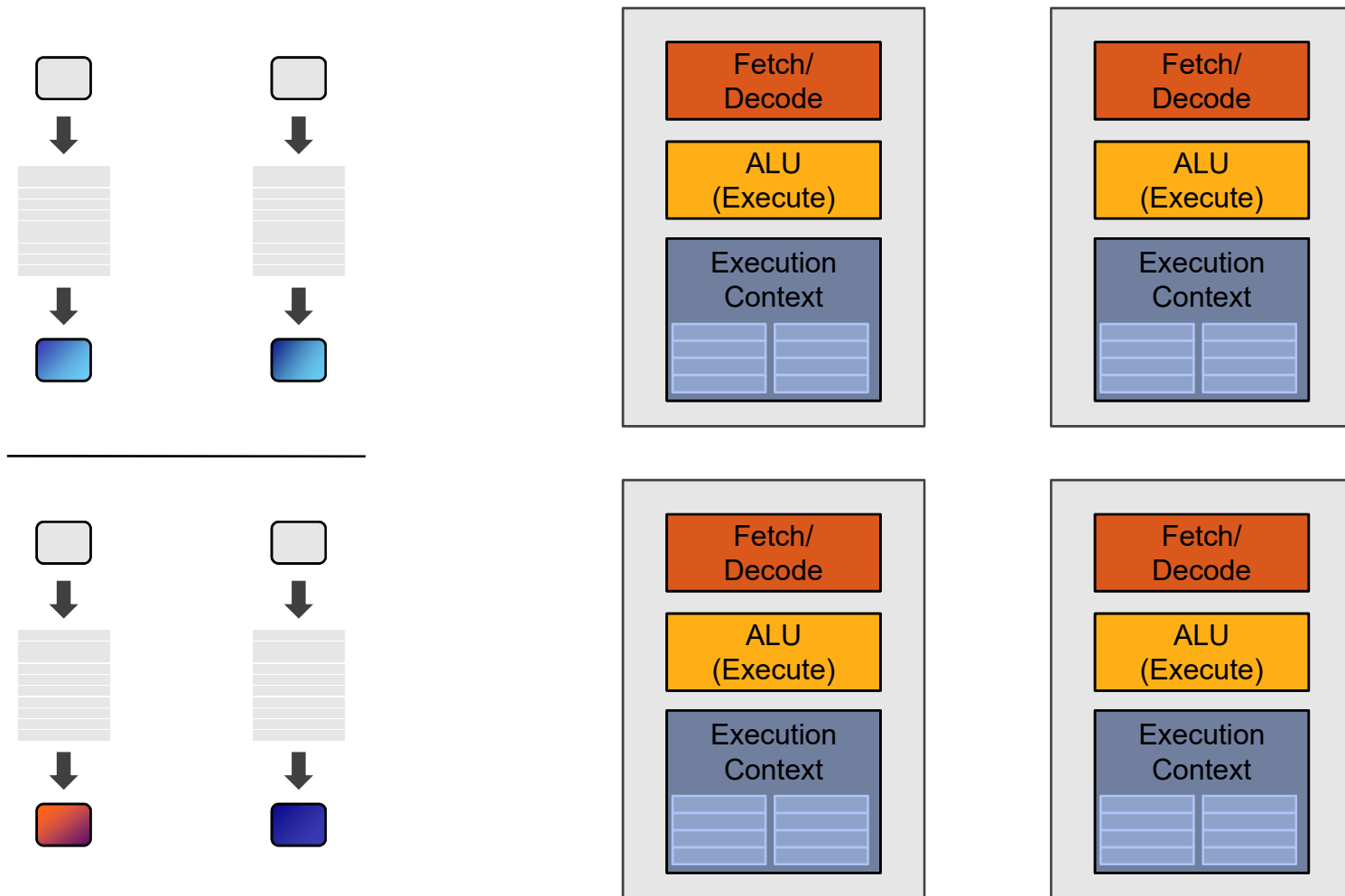
fragment 1



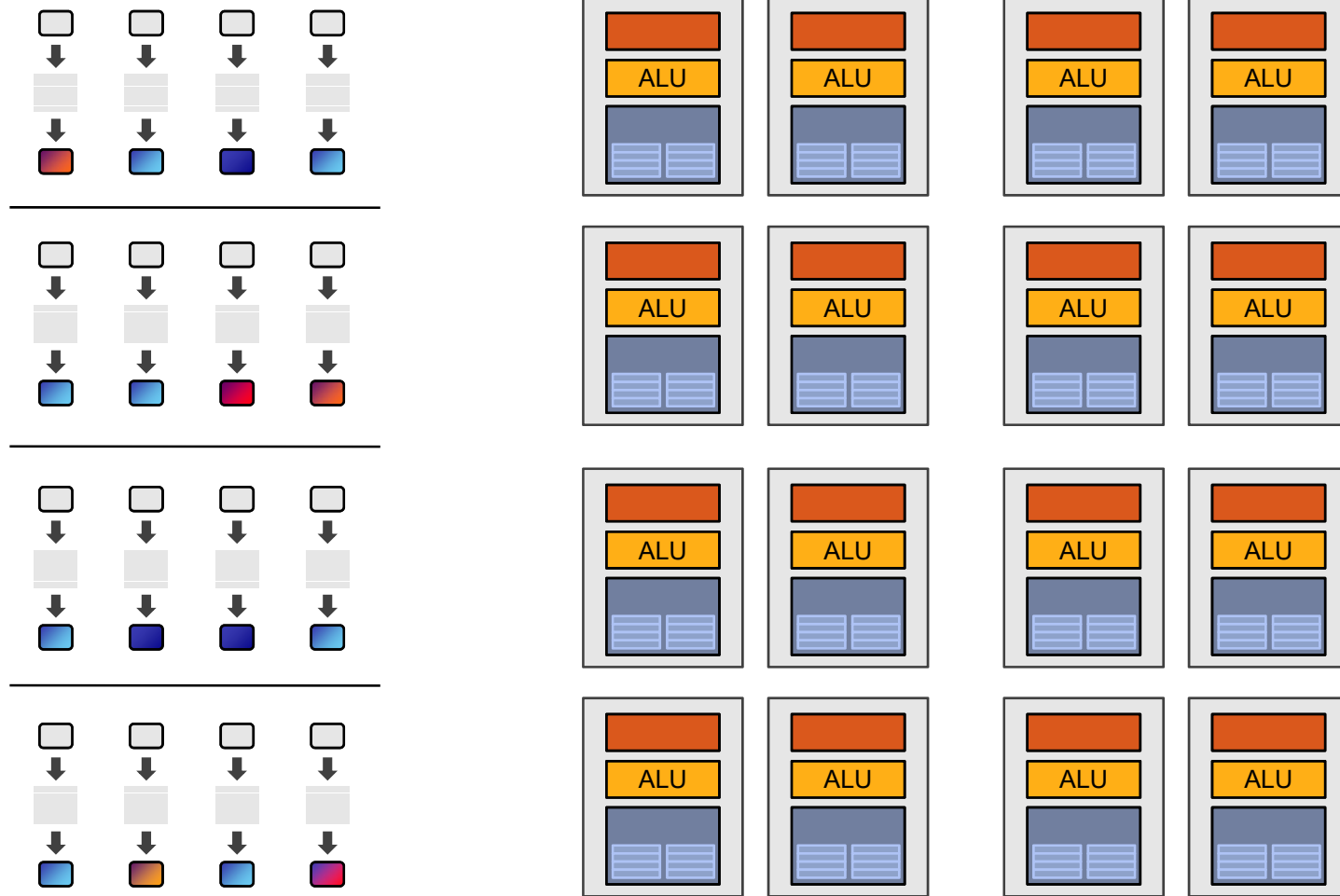
fragment 2



Four cores (four fragments in parallel)

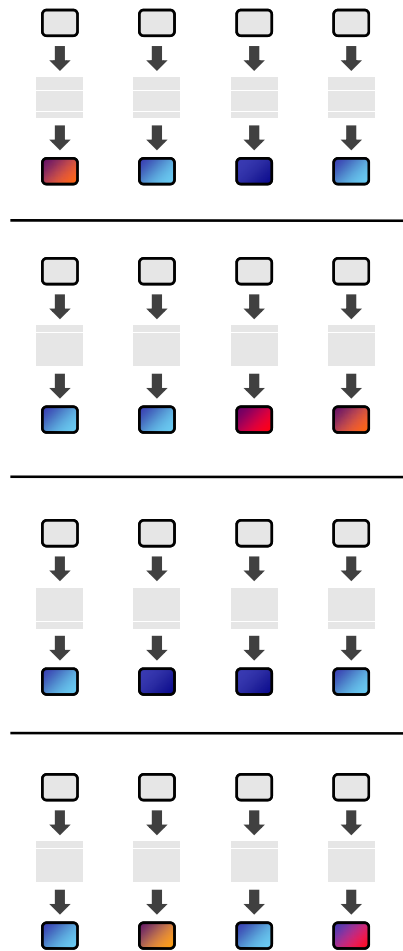


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

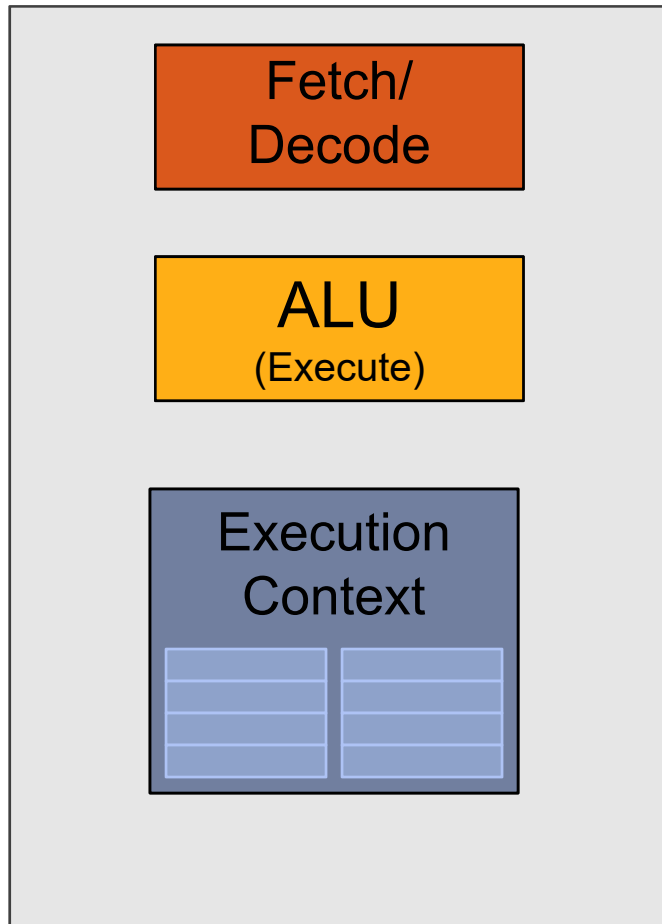
Instruction stream sharing



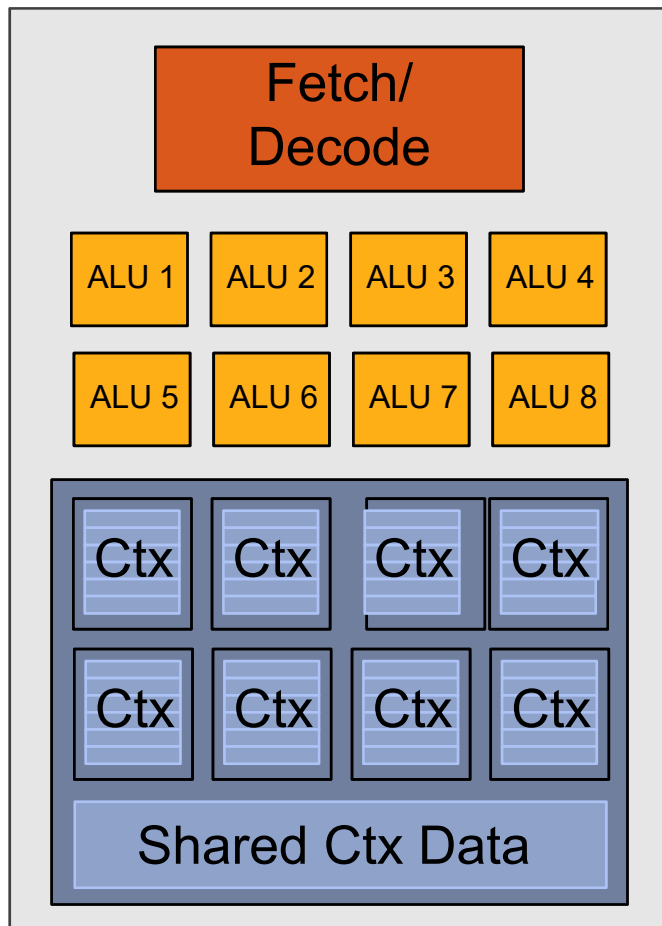
But... many fragments should be able to share an instruction stream!

```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp 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)
```

Recall: simple processing core



Idea #2: Add ALUs



Idea #2:

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

SIMD processing

(or SIMT, SPMD)

Thank you.