King Abdullah University of Science and Technology

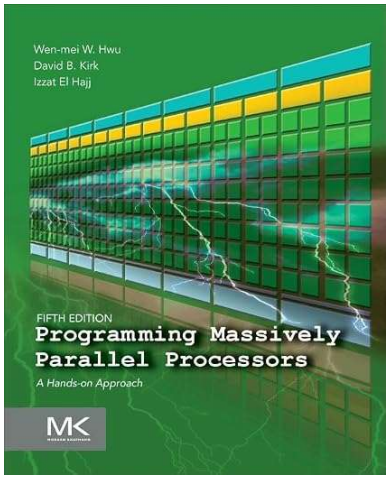
KAUST

CS 380 - GPU and GPGPU Programming
Lecture 6: GPU Architecture, Pt. 4

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 21)

Read (required):



- Programming Massively Parallel Processors book, 5th edition, **Chapter 2** (*Heterogeneous data-parallel computing*)
- Programming Massively Parallel Processors book, 5th edition, **Chapter 4** (*Compute architecture and scheduling*) **until 4.3**
- NVIDIA CUDA C++ Programming Guide
  (current: v13.4, Sep 9, 2026)

`https://docs.nvidia.com/cuda/cuda-programming-guide/pdf/cuda-programming-guide.pdf`

*Read **Chapter 1.3.1*** (*Compute Capability and Streaming Multiprocessor Versions*), ***Chapter 5.1*** (Compute Capabilities)

- Look at NVIDIA CUDA C++ Best Practices Guide
  (current: v13.4, Sep 9, 2026)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

Shared Memory Allocation

- **2 modes**
- **Static size within kernel**

```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

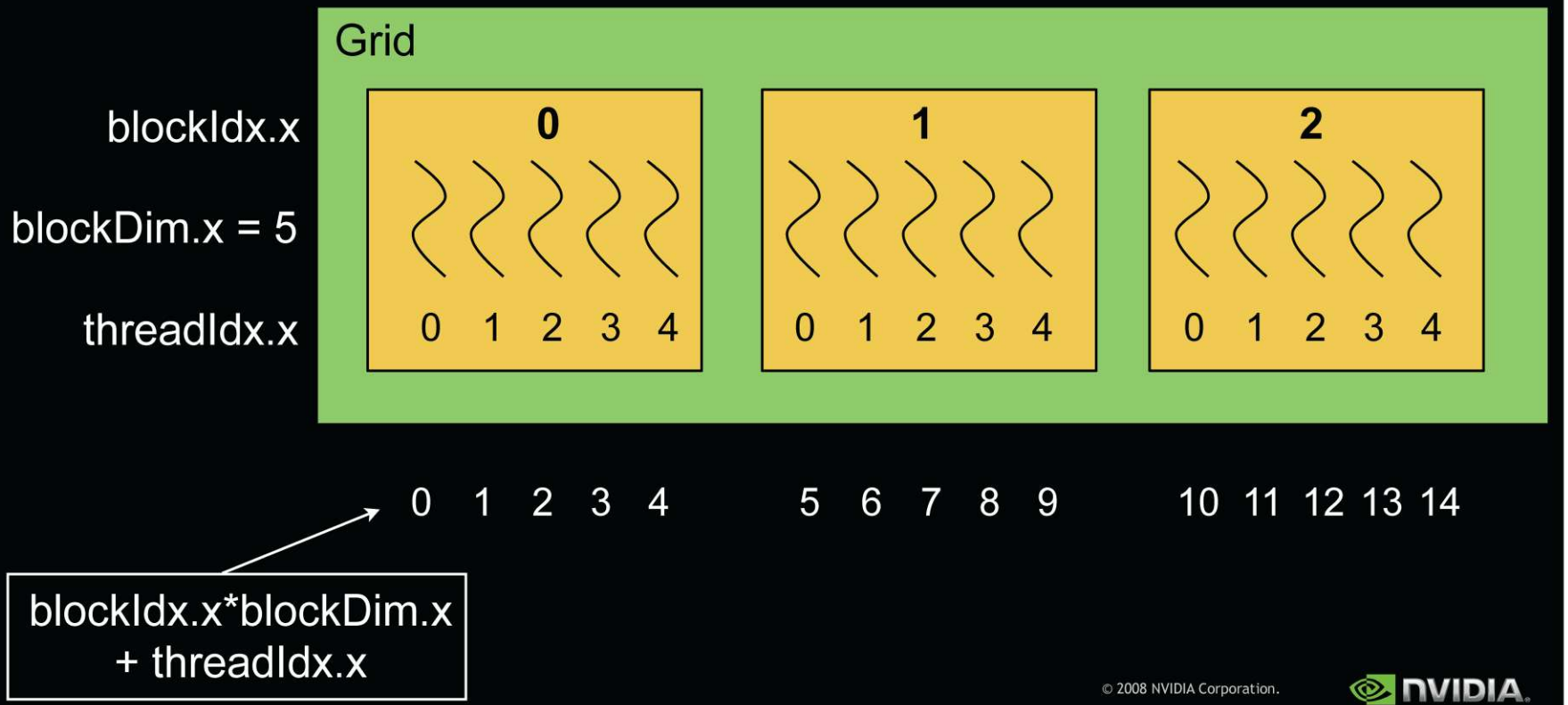
```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Increment Array Example

CPU program

```
void inc_cpu(int *a, int N)
{
  int idx;

  for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
}

int main()
{
  ...
  inc_cpu(a, N);
}
```

CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
 int idx = blockIdx.x * blockDim.x
           + threadIdx.x;
 if (idx < N)
   a[idx] = a[idx] + 1;
}

int main()
{
 …
 dim3 dimBlock (blocksize);
 dim3 dimGrid( ceil( N / (float)blocksize)  );
 inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

GPU Architecture: General Architecture

Concepts: Latency vs. Throughput

Latency

- What is the time *between start and finish* of an operation/computation?
- How long does it take between starting to execute an instruction until the execution is actually finished / its results are available?
- Examples: 1 FP32 MUL instruction; 1 vertex computation, …

Throughput

- How many computations (operations/instructions) *finish per time unit*?
- How many instructions of a certain type (e.g., FP32 MUL) finish per time unit (per clock cycle, per second)?

GPUs: ***High-throughput execution*** (at the expense of latency)
(but: *hide* latencies to avoid throughput going down)

Concepts: Types of Parallelism

Instruction level parallelism (ILP)

- In single instruction stream: Can consecutive instructions/operations be executed in parallel? (Because they don't have a dependency)
- Exploit ILP: Execute independent instructions (1) via pipelined execution (instr. pipe), or even (2) in multiple parallel instruction pipelines (superscalar processors)
- On GPUs: also important, but much less than TLP (compare, e.g., Kepler with current GPUs)

Thread level parallelism (TLP)

- Exploit that by definition operations in different threads are independent (if no explicit communication/synchronization is used, which should be minimized)
- Exploit TLP: Execute operations/instructions from multiple threads in parallel (which also needs multiple parallel instruction pipelines)
- **On GPUs: main type of parallelism**

more types:
- Bit-level parallelism (processor word size: 64 bits instead of 32, etc.)
- Data parallelism (SIMD/vector instructions), task parallelism, …

Concepts: Latency Hiding

Not about latency of single operation or group of operations:
It's about avoiding that the *throughput* goes below peak

Hide latency that *does* occur for one instruction (group) by *executing a different instruction (group)* as soon as current one stalls:

→ *Total throughput does not go down*

In GPUs, hide latencies via:

- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from down the inst. stream in same thread group
- Depending on GPU: TLP often sufficient, but sometimes also need ILP
- However: If in one cycle TLP doesn't work, ILP can jump in or vice versa

SIGGRAPH2009 NEW ORLEANS

From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian
Stanford University

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

Where this is going...

Summary: three key ideas for high-throughput execution

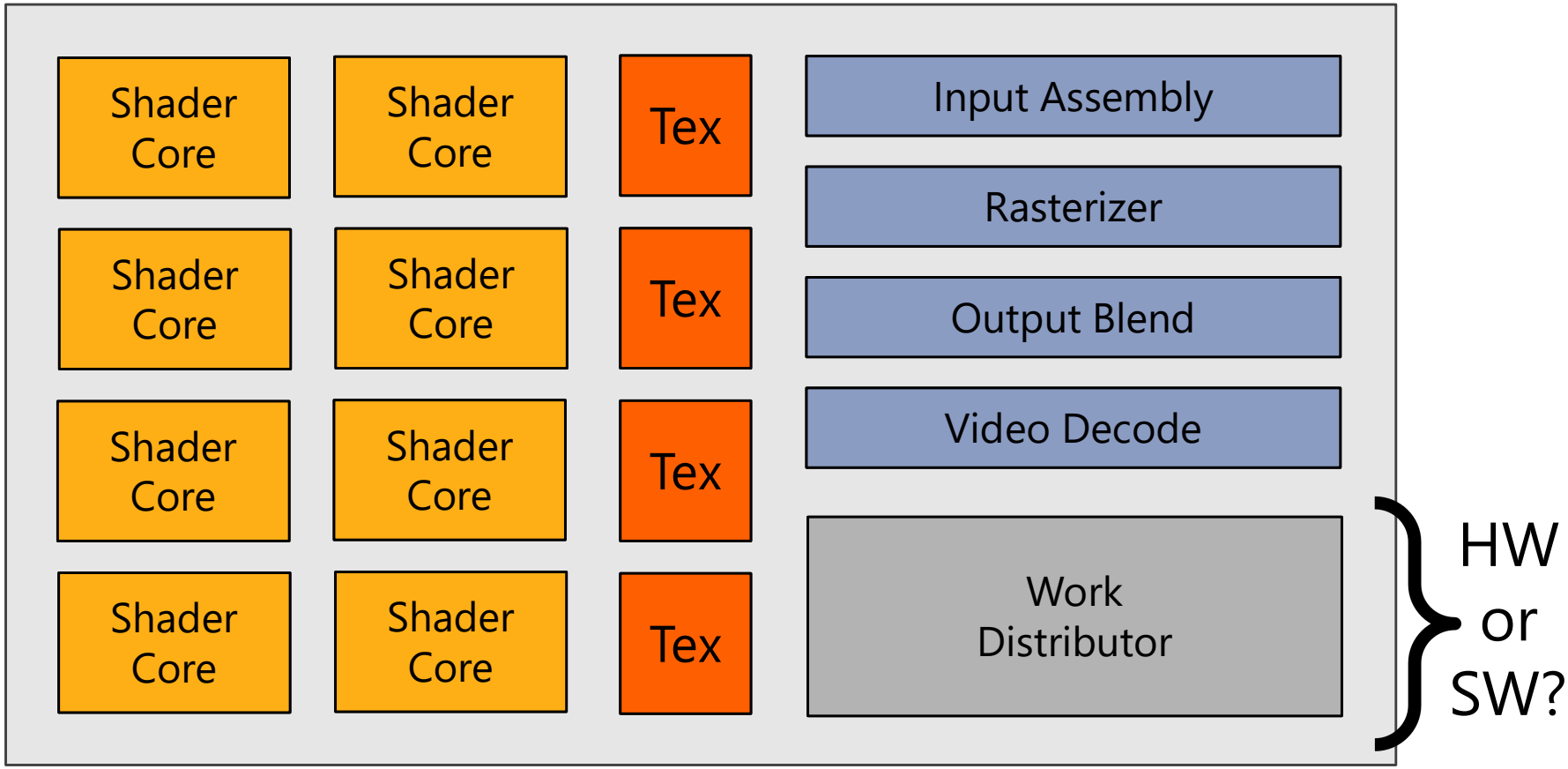
1. Use many "slimmed down cores," run them in parallel

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

   GPUs are here! (usually)

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

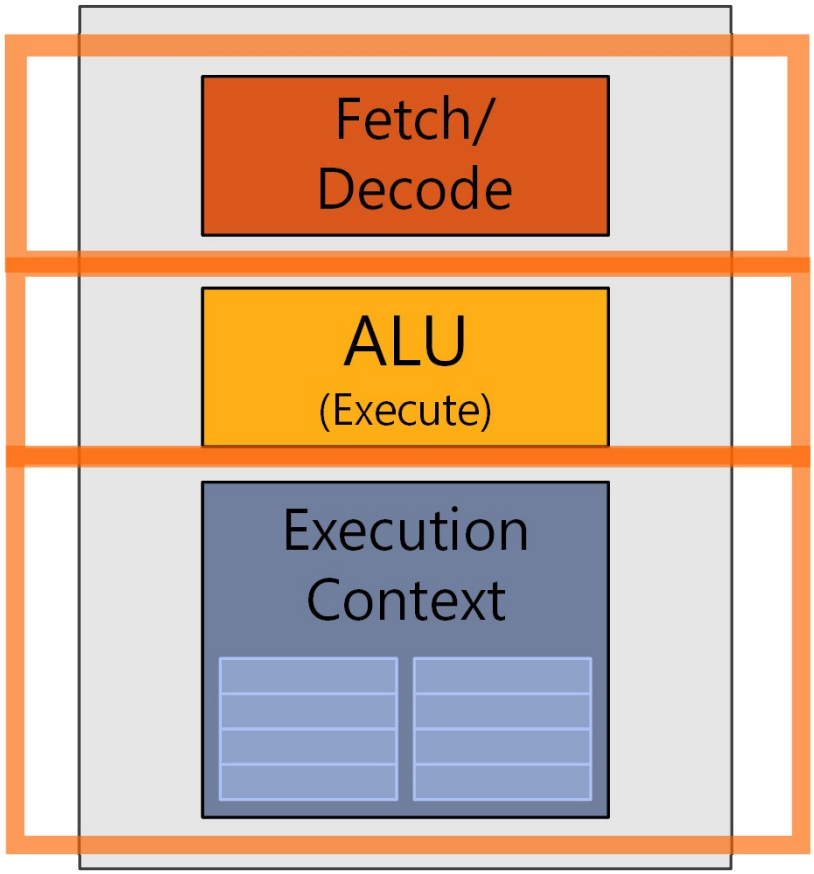
1 unshaded fragment input record

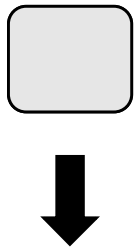
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

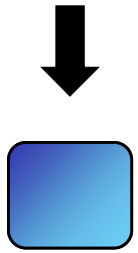
1 shaded fragment output record

Execute shader

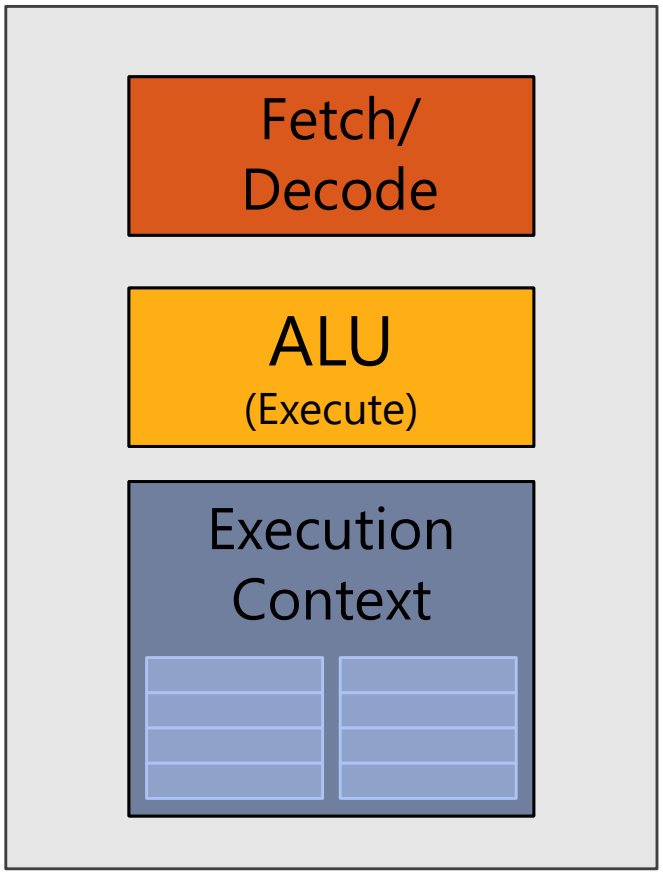


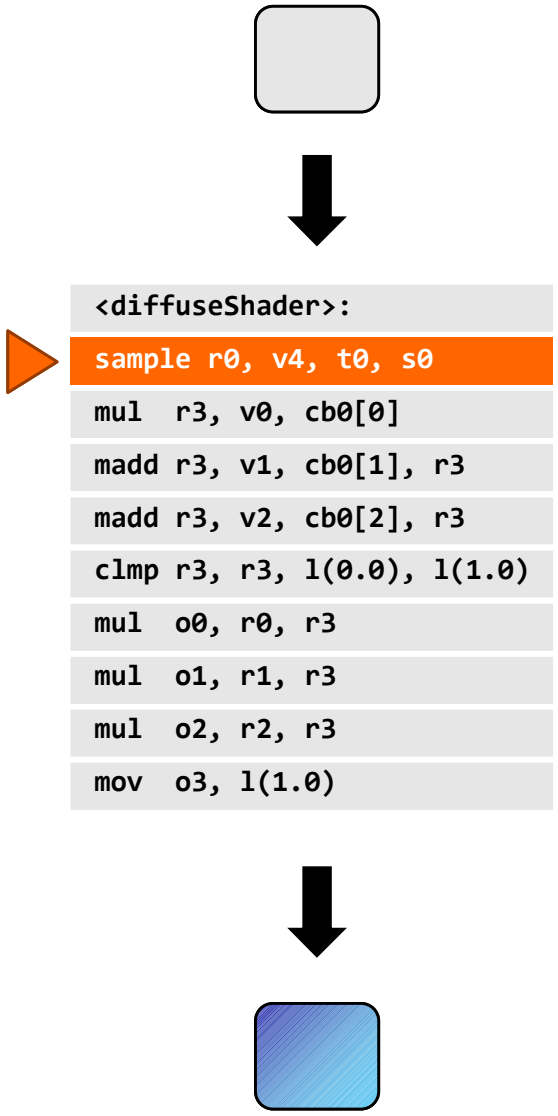


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

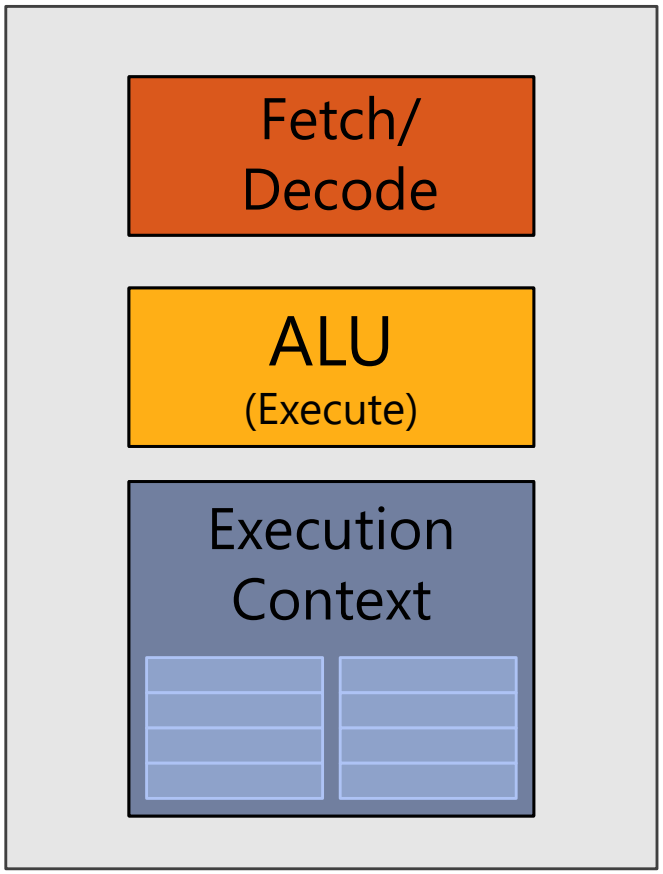


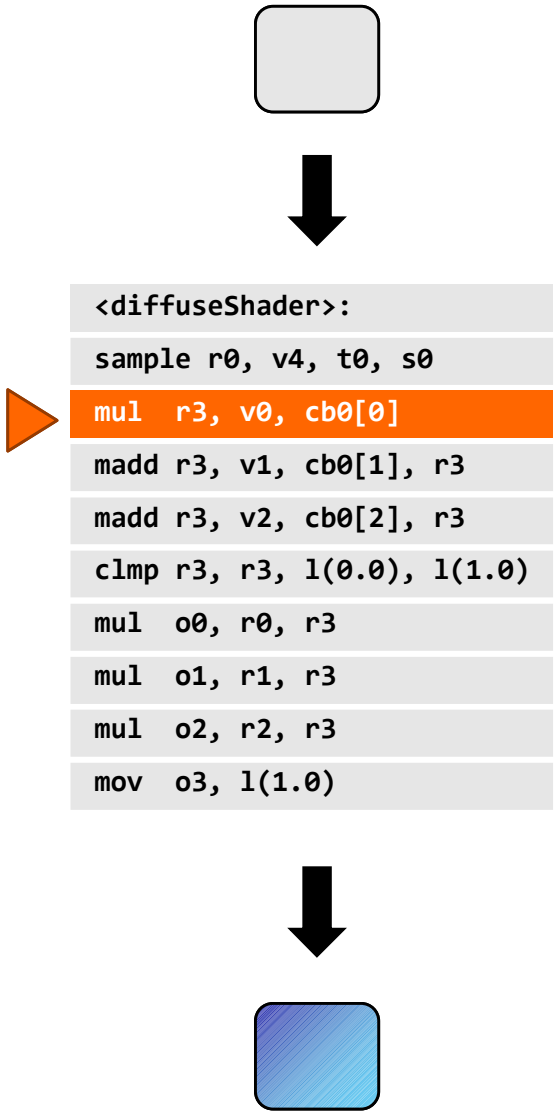
Execute shader



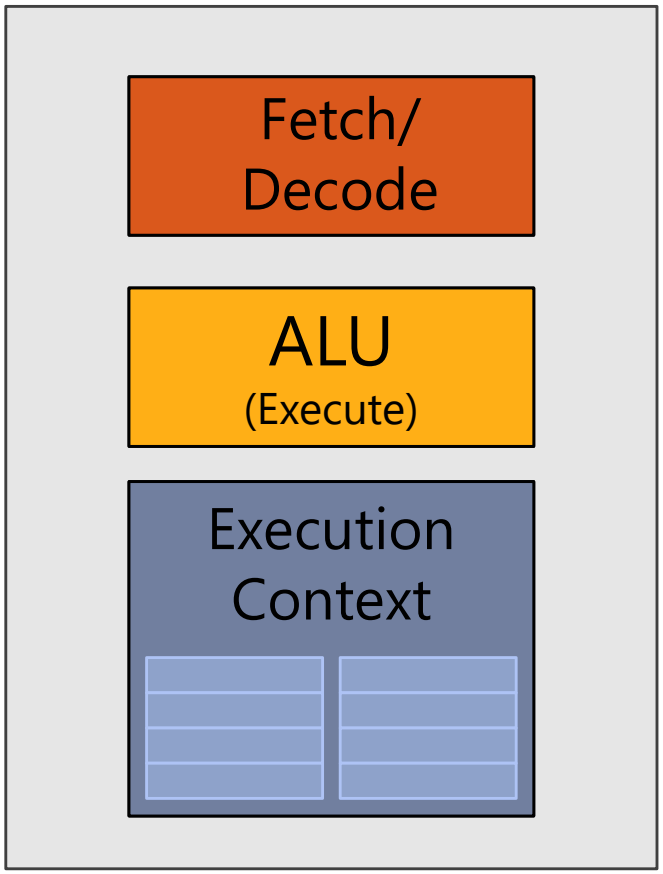


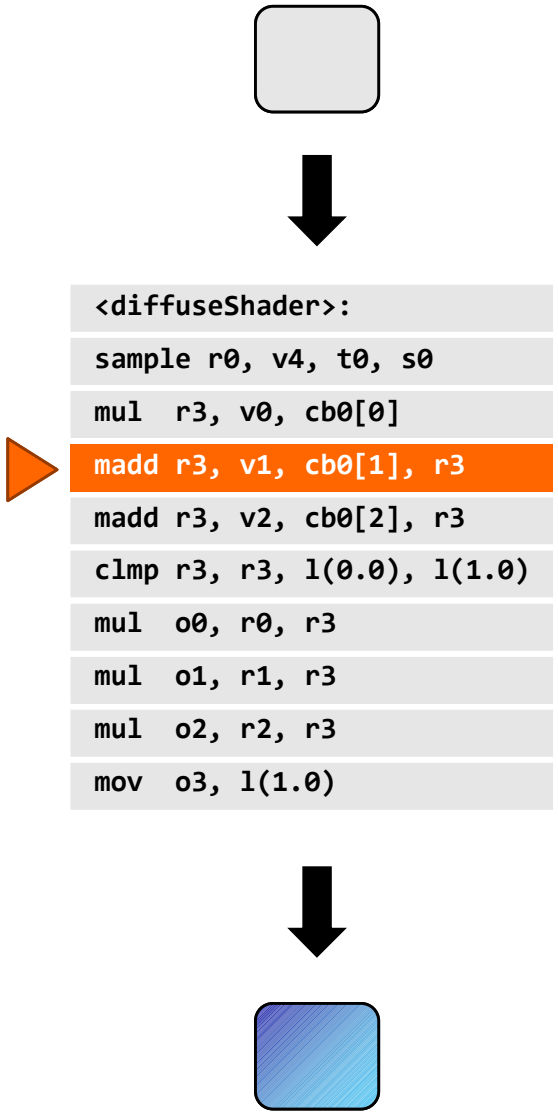
Execute shader



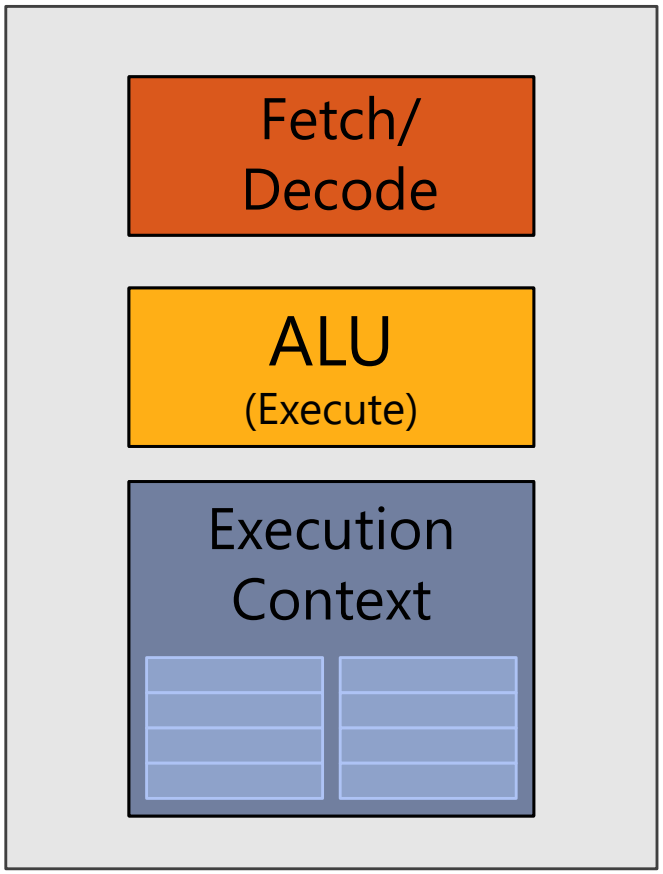


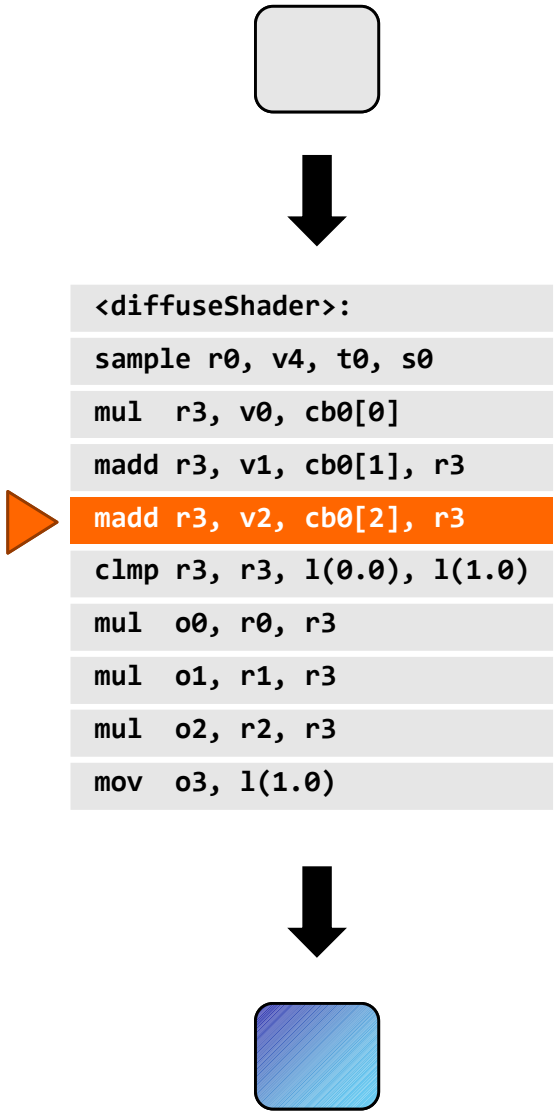
Execute shader



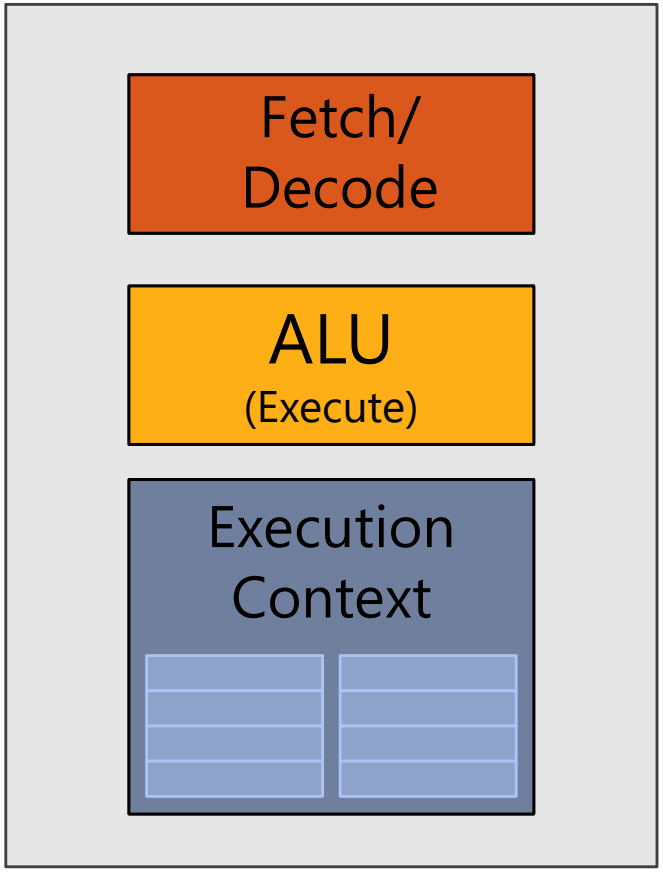


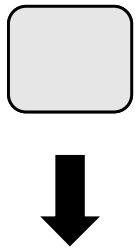
Execute shader



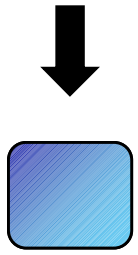


Execute shader



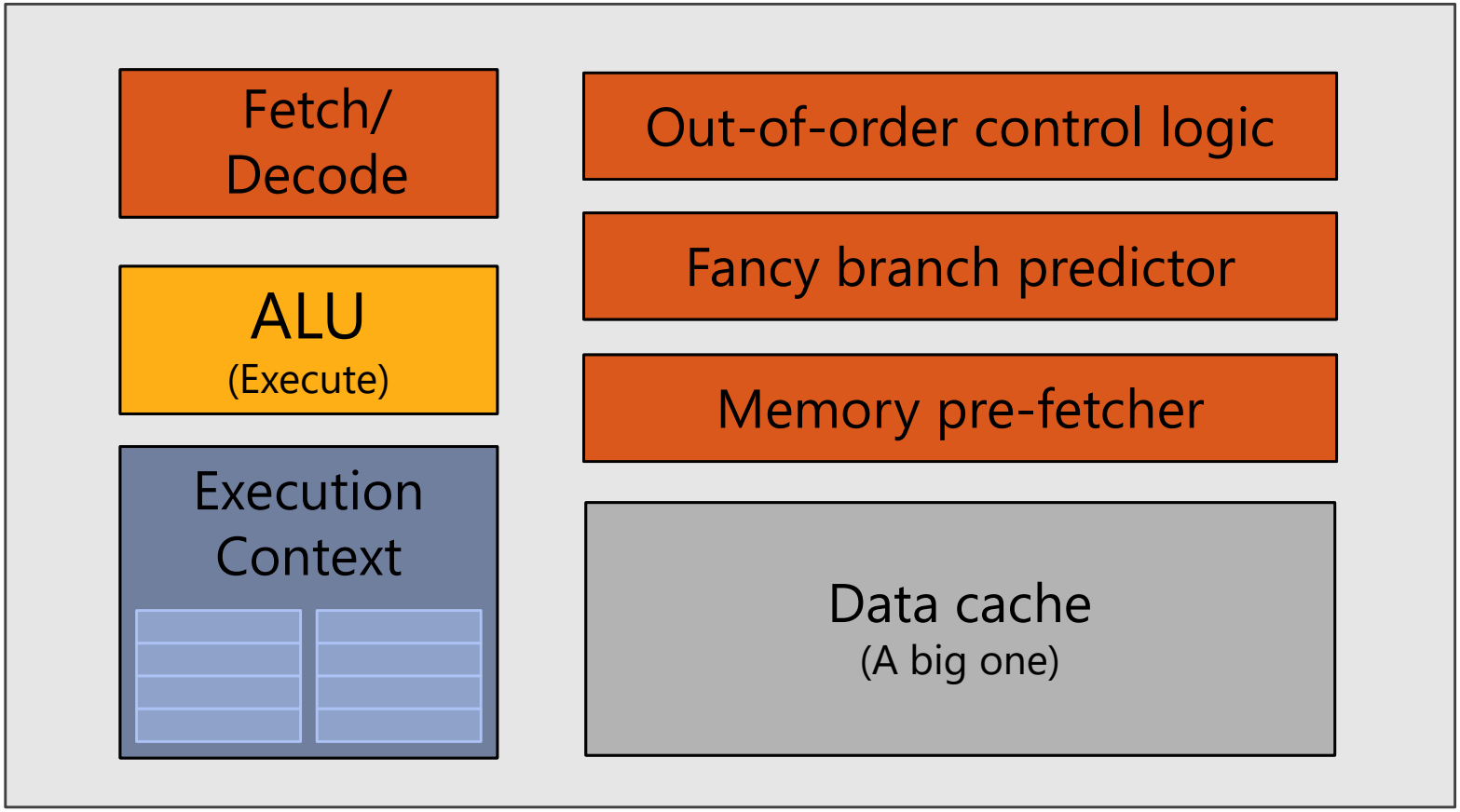


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

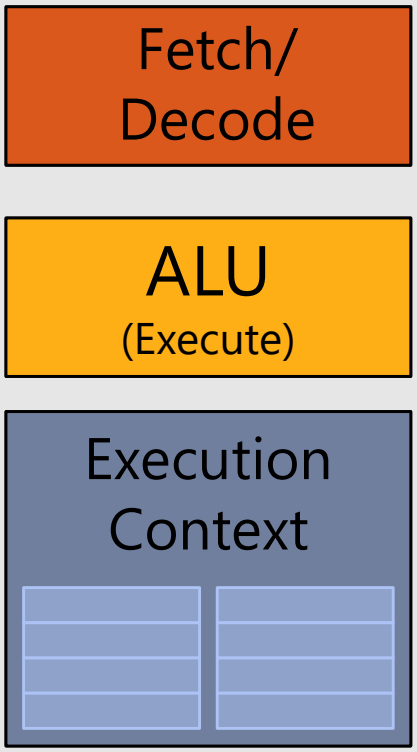


GPU Architecture
Big Idea #1

CPU-"style" cores



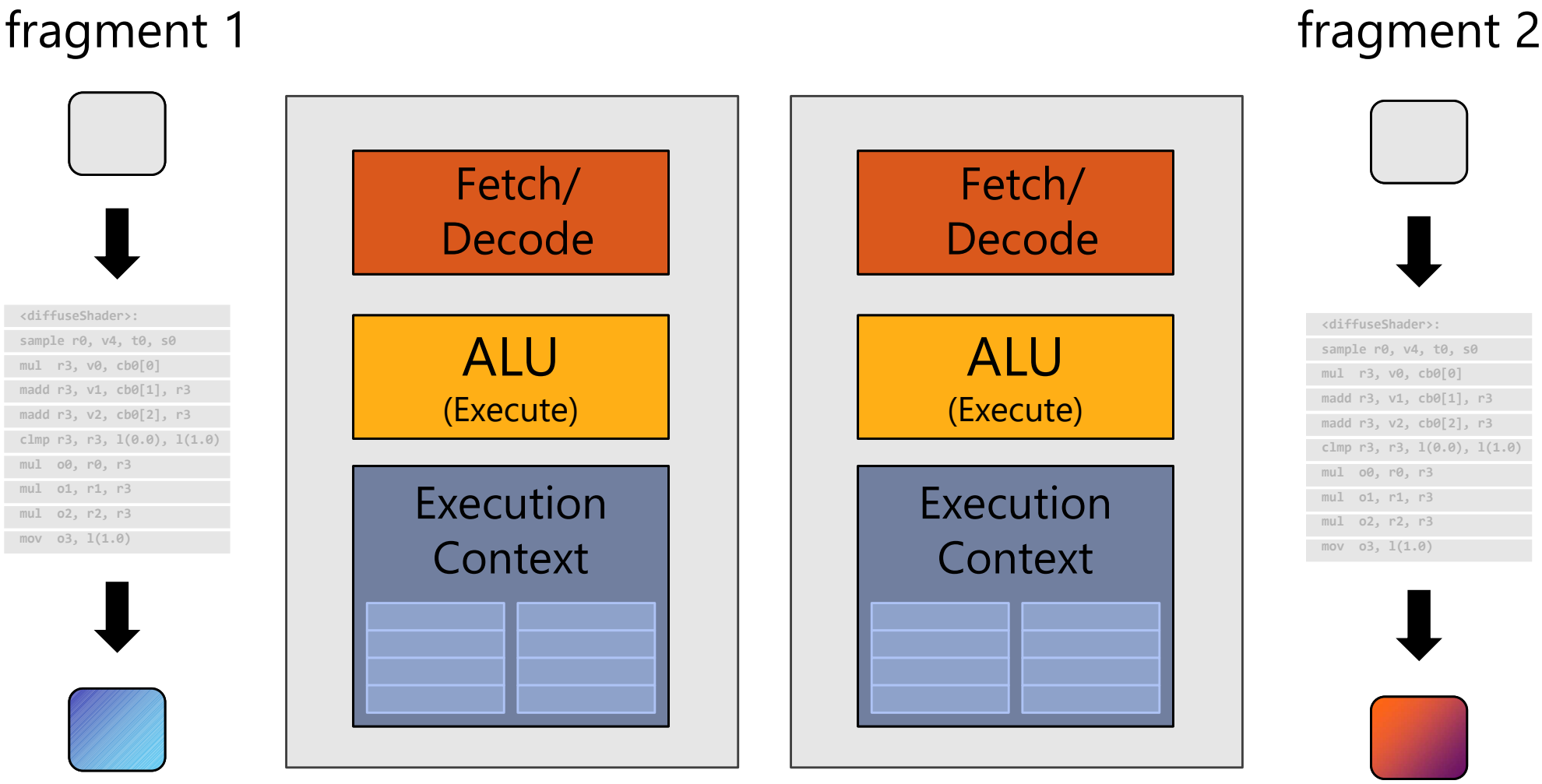
Idea #1: Slim down



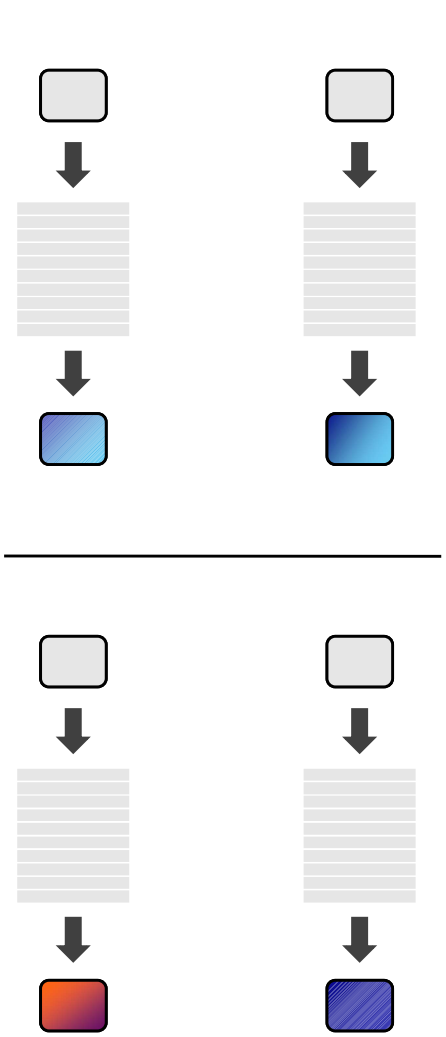
Idea #1:

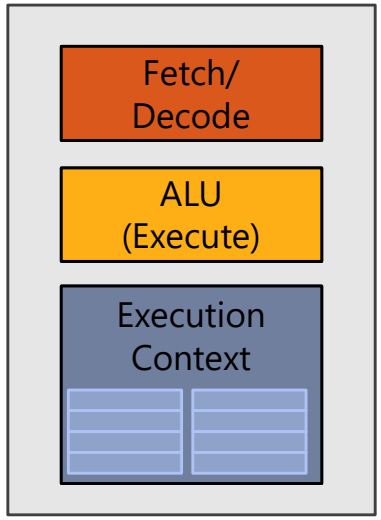
Remove components that help a single instruction stream run fast (i.e., that reduce latency of a single instruction stream)

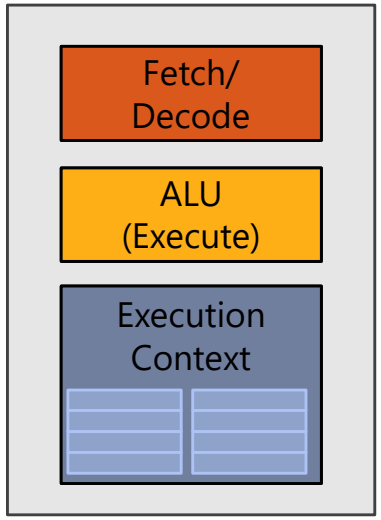
Two cores (two fragments in parallel)

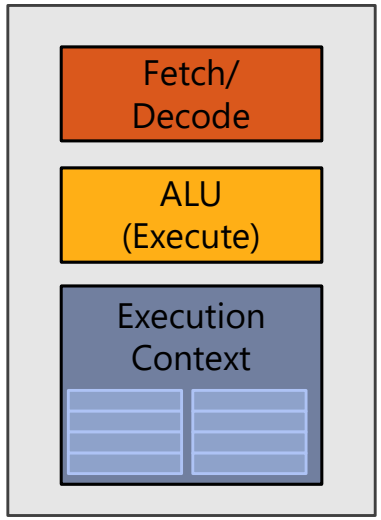


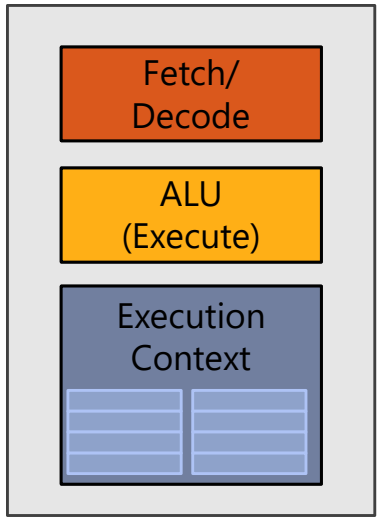
Four cores (four fragments in parallel)







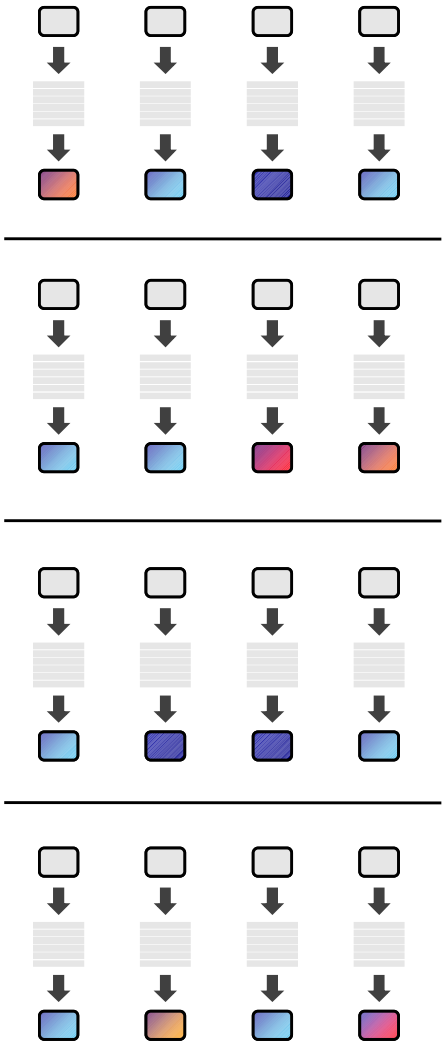




Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

Instruction stream sharing



But… many fragments should be able to share an instruction stream! → **big idea #2 !**

```
<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)
```

Thank you.