CMPE 110: Computer Architecture

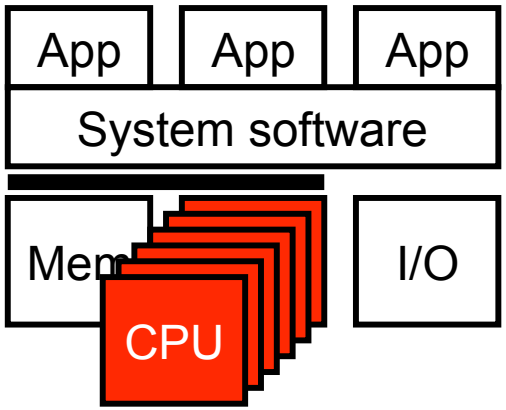
Week 11

GPUs

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Kayvon Fatahalian, Joe Devietti, Onur Mutlu, and others]

Review: Multicore



- Hardware multithreading vs. Multicore
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Memory consistency models
  - Sequential consistency

Review: Quick example of SC

Initially, A=0, B=0

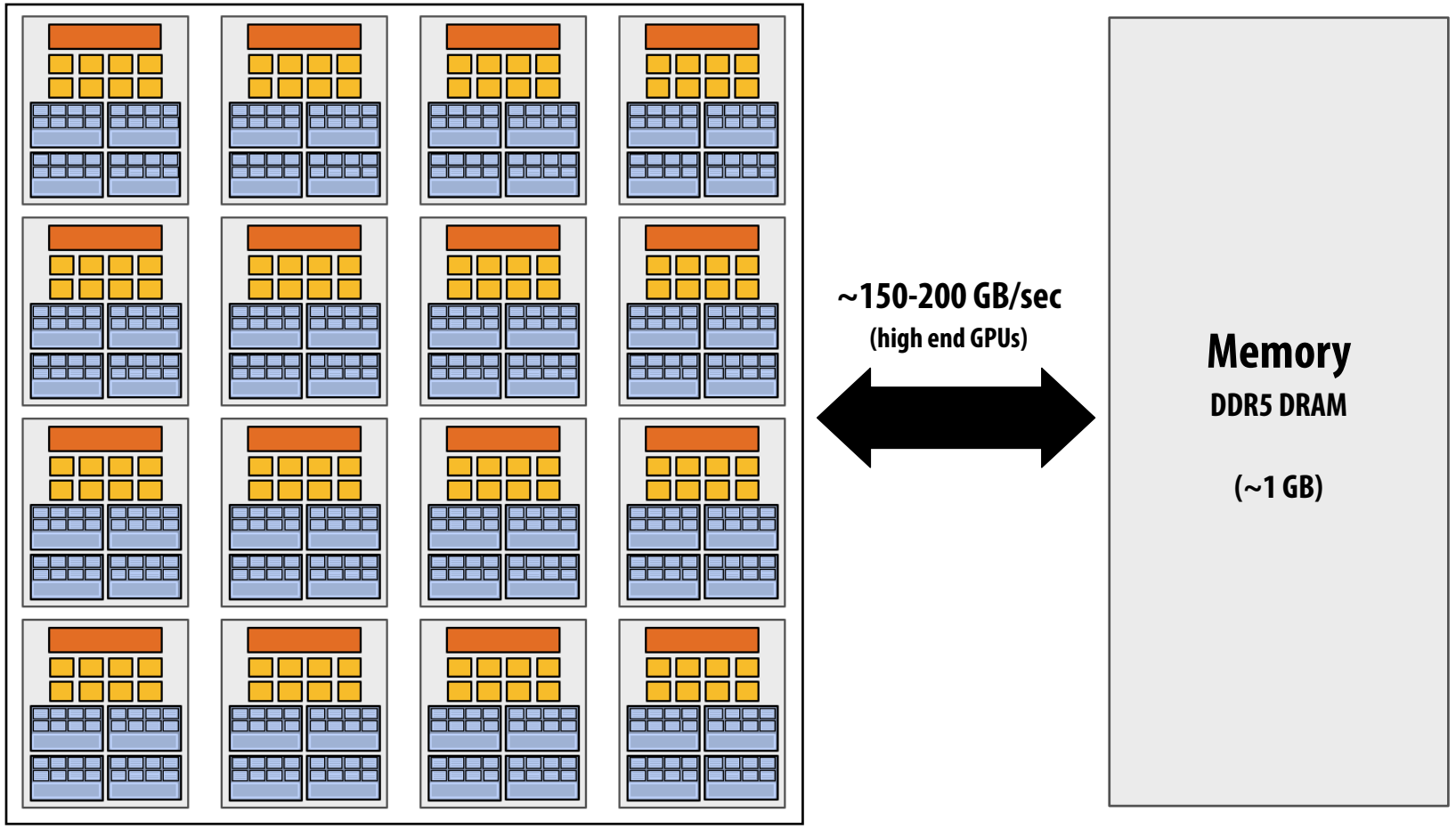
Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print "Hello";
```

Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print "World";
```

- What will get printed out?
- Sequential consistency (SC) =>
  - P1: the write to A has to complete before the read of B can begin
  - P2: the write to B has to complete before the read of A can begin
- Answer: Code will either print "hello" or "world" or nothing, but not both.



DATA-LEVEL PARALLELISM & GPU

How to Compute This Fast?

- Performing the **same** operations on **many** data items
  - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
    Z[I] = A*X[I] + Y[I];
           }
```

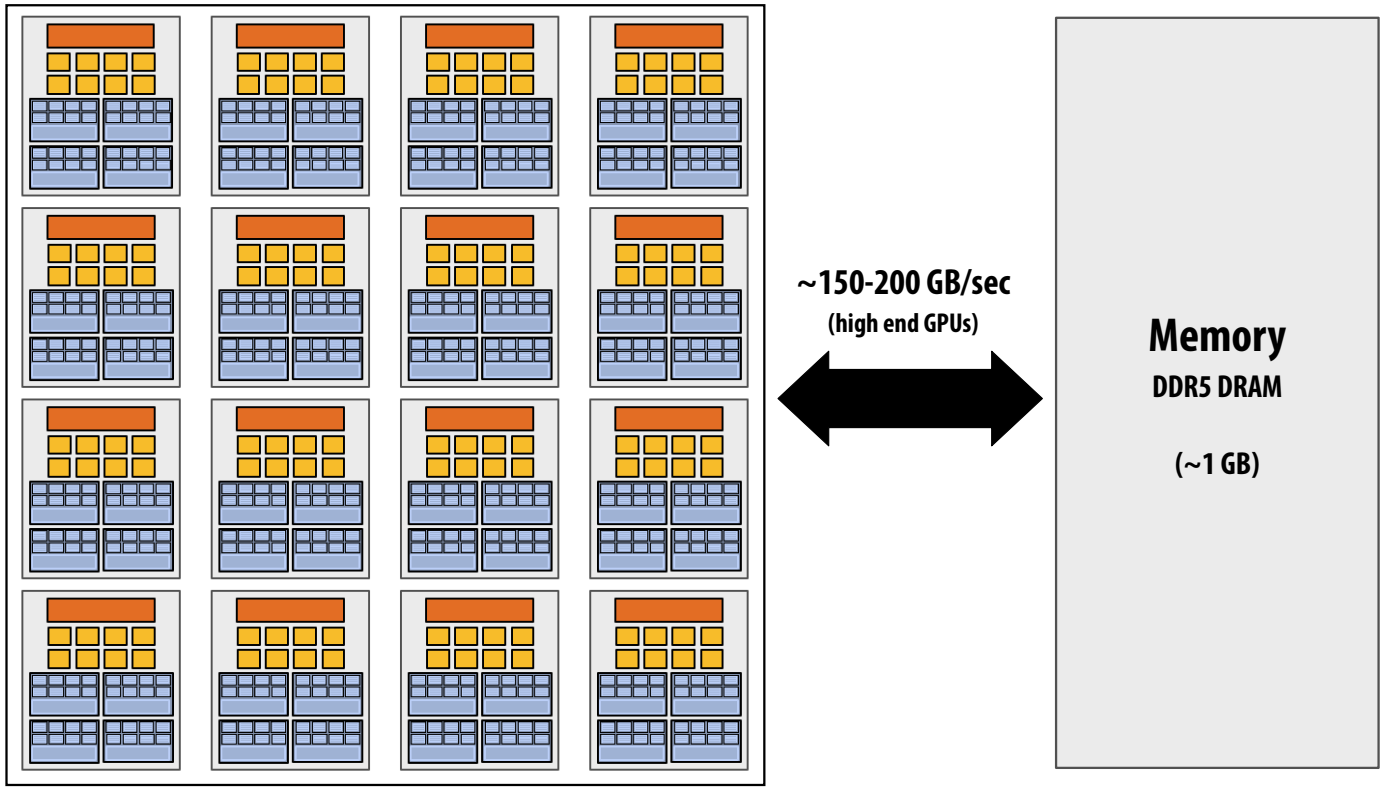
```
L1: ldf [X+r1]->f1   // I is in r1
    mulf f0,f1->f2   // A is in f0
        ldf [Y+r1]->f3
           addf f2,f3->f4
        stf f4->[Z+r1}
           addi r1,4->r1
          blti r1,4096,L1
```

- Thread-level parallelism (TLP) - coarse grained
  - Multicore
- Can we do something finer grained?

Data-Level Parallelism

- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
    - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point "vector registers"
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector or Vector+Scalar
      - addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, so one operation per cycle (not parallel)

Basic GPU architecture



- Multi-core chip
- SIMD execution within a single core (many ALUs performing the same instruction)
- Multi-threaded execution on a single core (multiple threads executed concurrently by a core)

What GPUs were originally designed to do: 3D rendering

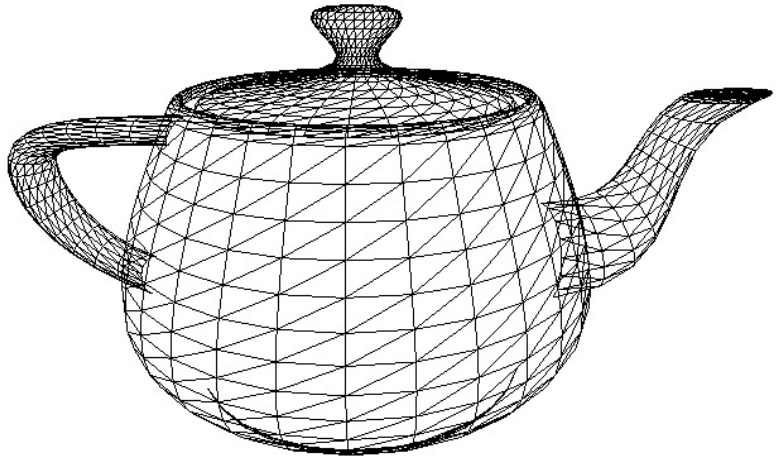


Image credit: Henrik Wann Jensen

Input: description of a scene:

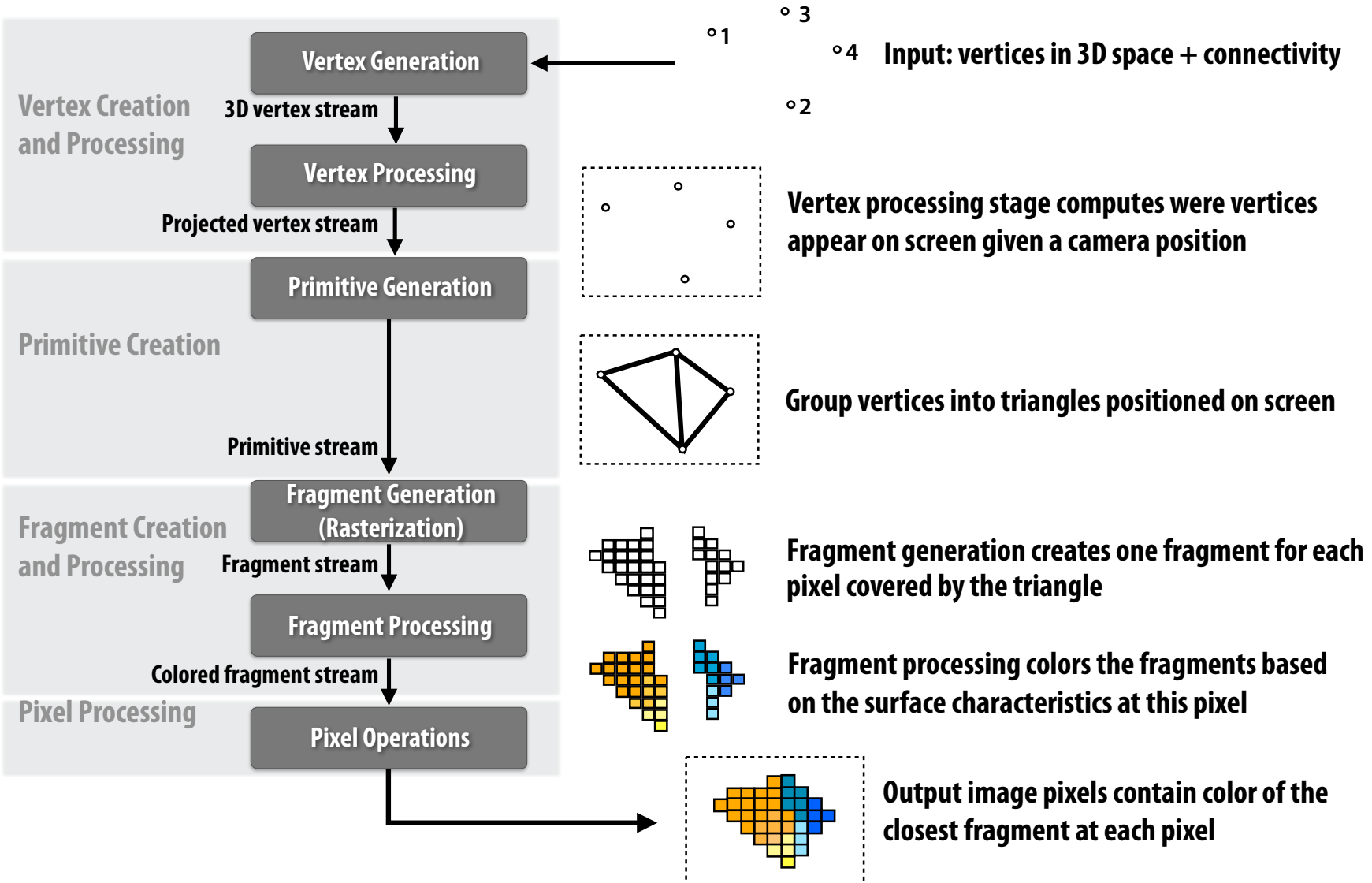
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

- Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

Graphics pipeline architecture

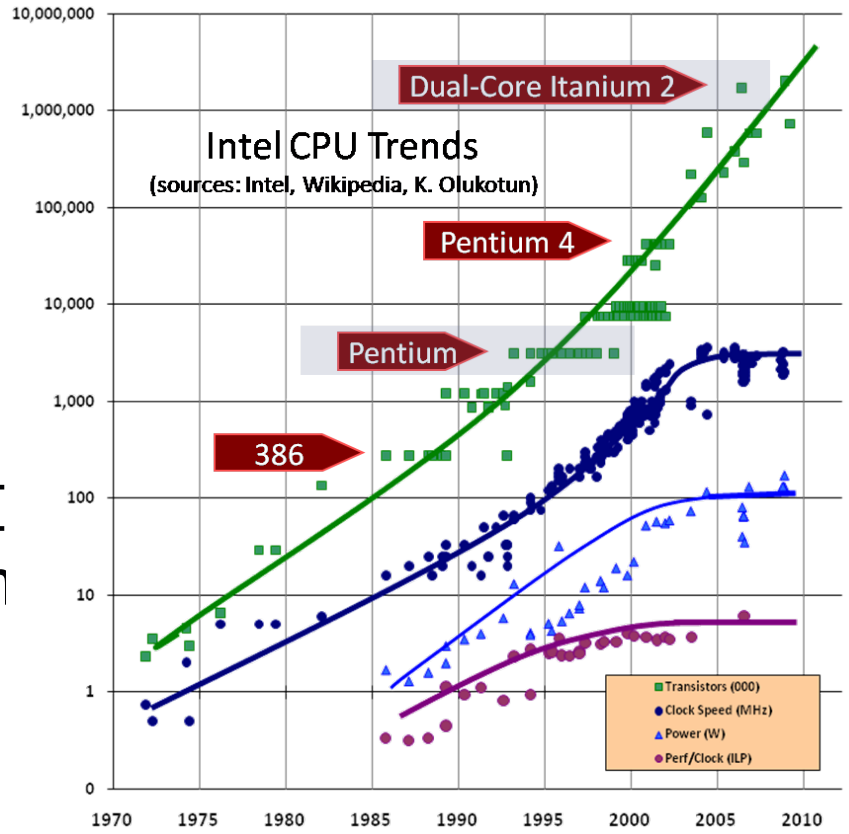
Performs operations on vertices, triangles, fragments, and pixels



Observation circa 2001-2003

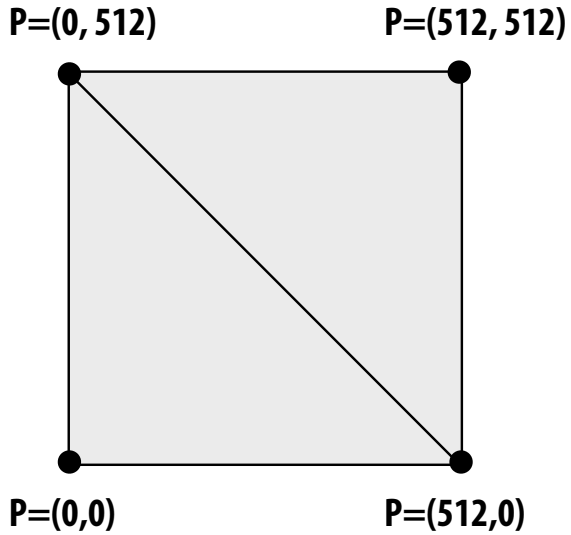
- These GPUs are very fast processors for performing the same computation (shaders) on collections of data (streams of vertices, fragments, pixels)

- Wait a minute! That sounds a lot like data-parallelism to me! I remember data-parallelism from exotic supercomputers in the 90s.



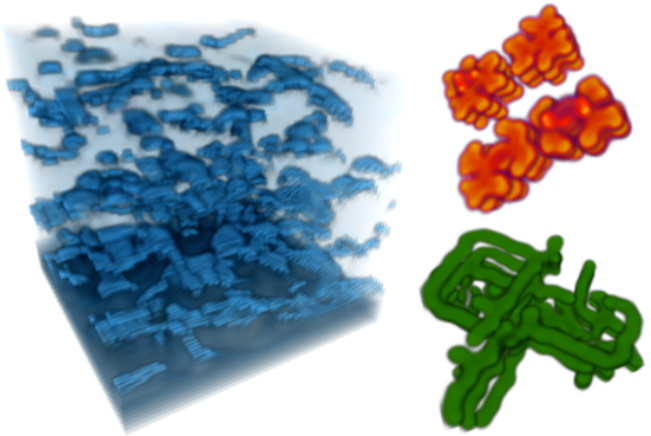
Hack! early GPU-based scientific computation

- Set OpenGL output image size to be output array size (e.g., 512 x 512)
- Render 2 triangles that exactly cover screen (one shader computation per pixel = one shader computation output image element)
- We now can use the GPU like a data-parallel programming system.
- Fragment shader function is mapped over 512 x 512 element collection.

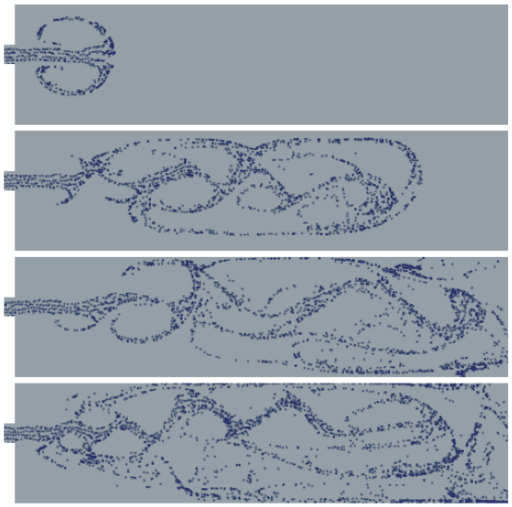


"GPGPU" 2002-2003

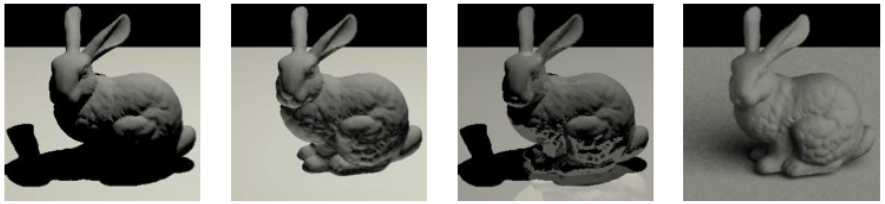
- GPGPU = "general purpose" computation on GPUs



Coupled Map Lattice Simulation [Harris 02]



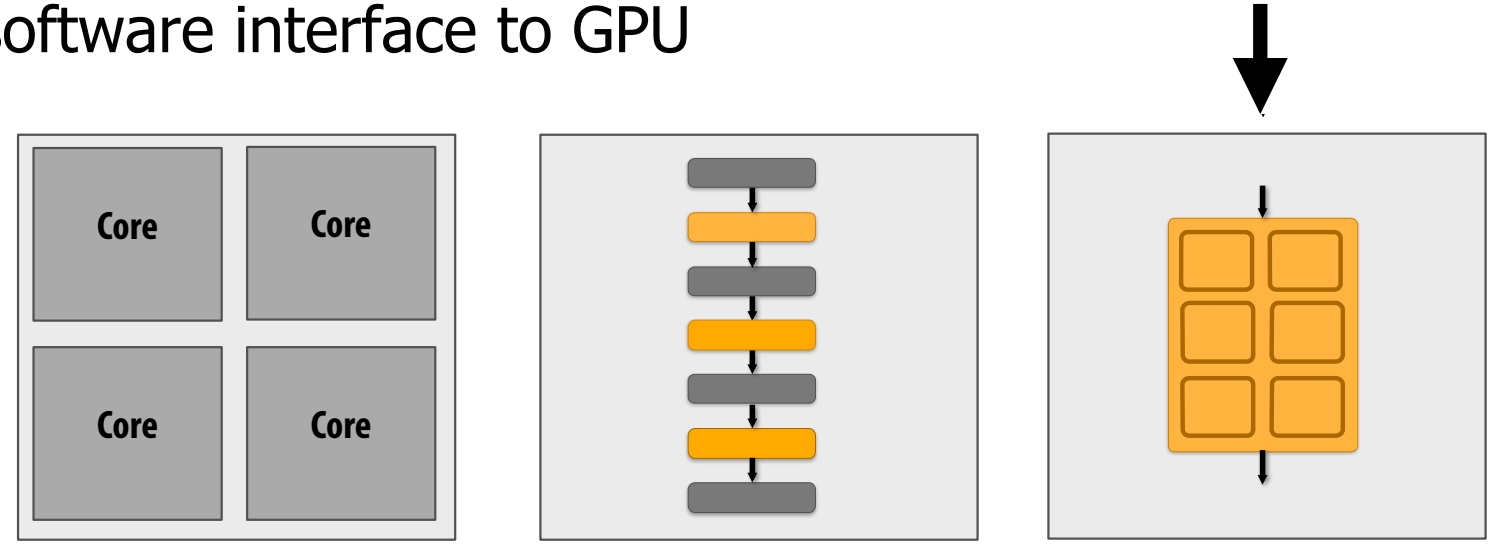
Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

NVIDIA Tesla architecture (2007)

- GeForce 8xxx series
- First alternative, non-graphics-specific ("compute mode") software interface to GPU



Multi-core CPU architecture

CPU presents itself to OS as a multi-processor system

ISA provides instructions for managing execution context (program counter, VM mappings, etc.) on a per core basis

Pre-2007 GPU architecture

GPU presents following interface** to system software (driver):

Set screen size
Set shader program for pipeline
DrawTriangles

Post-2007 "compute mode" GPU architecture

GPU presents a new data-parallel interface to system software (driver):

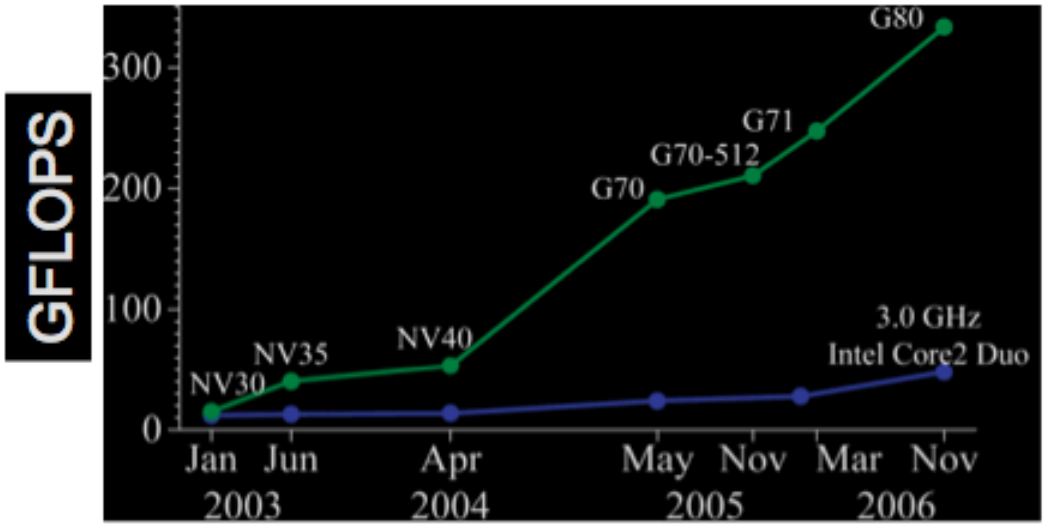
Set kernel program
Launch(kernel, N)

Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)
- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until recently, programmed through graphics API

GeForce 8800

Tesla S870



G80 = GeForce 8800 GTX
G71 = GeForce 7900 GTX
G70 = GeForce 7800 GTX
NV40 = GeForce 6800 Ultra
NV35 = GeForce FX 5950 Ultra
NV30 = GeForce FX 5800

  - GPU in every desktop, laptop, mobile device – massive volume and potential impact

The GPU programmable processing core

- Three major ideas that make GPU processing cores run fast
- Closer look at a real GPU design

* *Shader: In the field of computer graphics, a shader is a computer program that is used to do "shading"*

- *The production of appropriate levels of color within an image, or, in the modern era, also to produce special effects or do video post-processing.*

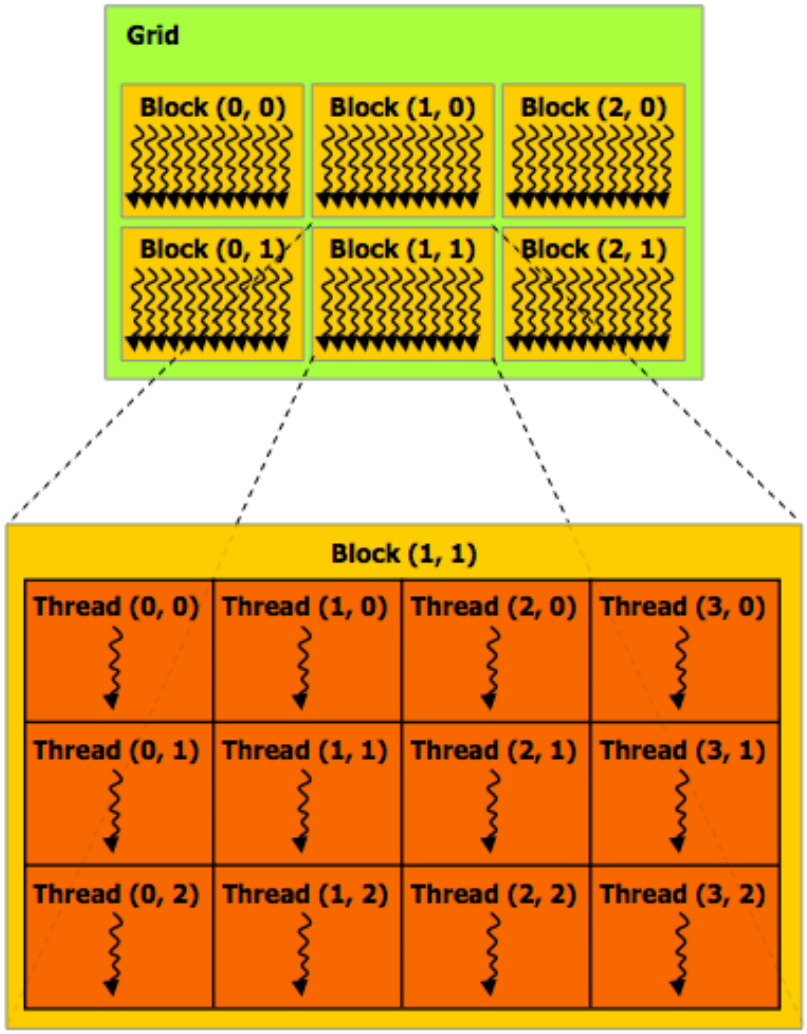
- Following slides c/o Kayvon Fatahalian's "Graphics and Imaging Architectures" and "Parallel Computer Architecture and Programming" courses
  - http://www.cs.cmu.edu/~kayvonf/

- I am going to describe NVIDIA GPU architectures using CUDA terminology, but the basic ideas apply to other families

CUDA programs consist of a hierarchy of concurrent threads

Thread IDs can be up to 3-dimensional (2D example below)

Multi-dimensional thread ids are convenient for problems that are naturally n-D




```
const int Nx = 12;
const int Ny = 6;

// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + B[j][i];
}

////////////////////////////////////////////

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 CUDA threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA programs consist of a hierarchy of concurrent threads

SPMD execution of device code:

Each thread computes its overall grid thread id from its position in its block (`threadIdx`) and its block's position in the grid (`blockIdx`)

"Device" code: SPMD execution kernel function (denoted by `__global__`) runs on co-processing device (GPU)

"Host" code : serial execution
Running as part of normal C/C++ application on CPU

Bulk launch of many threads
Precisely: launch a grid of thread blocks
Call returns when all threads have terminated

```
const int Nx = 12;
const int Ny = 6;

// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + B[j][i];
}

//////////////////////////////////////////

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Clear separation of host and device code

Separation of execution into host and device code is performed statically by the programmer

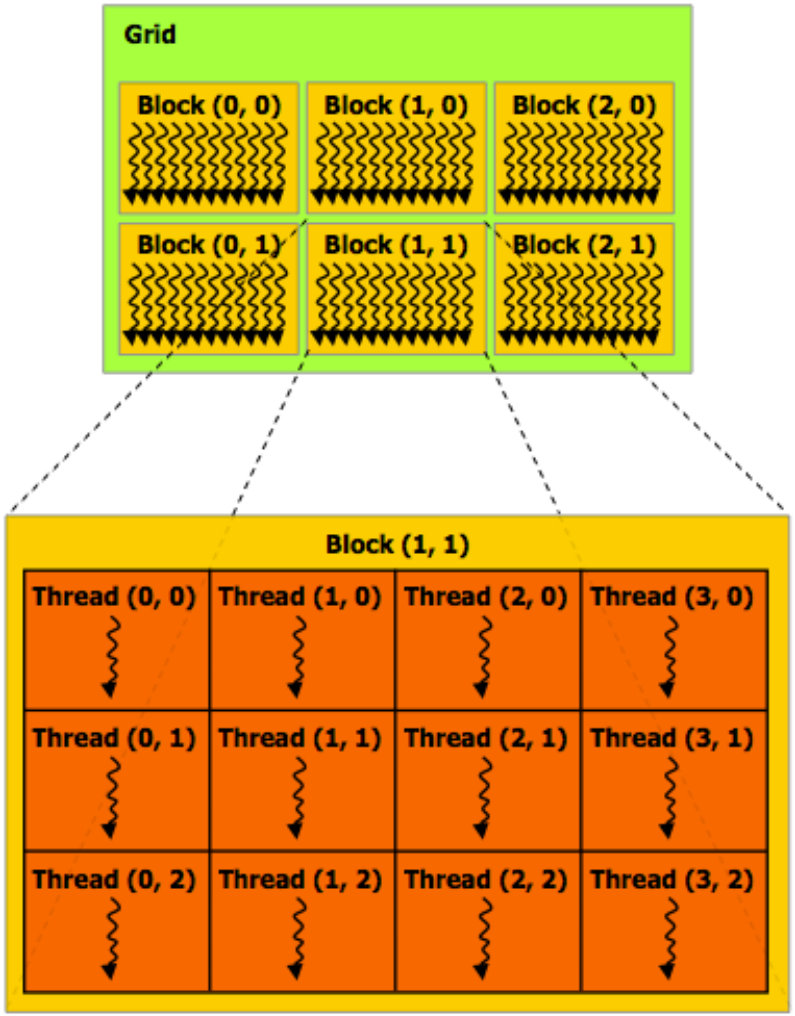
"Device" code
SPMD execution on GPU

"Host" code : serial execution on CPU

```
const int Nx = 12;
const int Ny = 6;

__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}

////////////////////////////////////////////

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Number of SPMD threads is explicit in program

Number of kernel invocations is not determined by size of data collection
(Kernel launch is not map(kernel, collection) as was the case with graphics shader programming)




```
const int Nx = 11;  // not a multiple of threadsPerBlock.x
const int Ny = 5;  // not a multiple of threadsPerBlock.y

__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   // guard against out of bounds array access
   if (i < Nx && j < Ny)
      C[j][i] = A[j][i] + B[j][i];
}

////////////////////////////////////////////

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x,
               (Ny+threadsPerBlock.y-1)/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA execution model

Host
(serial execution)

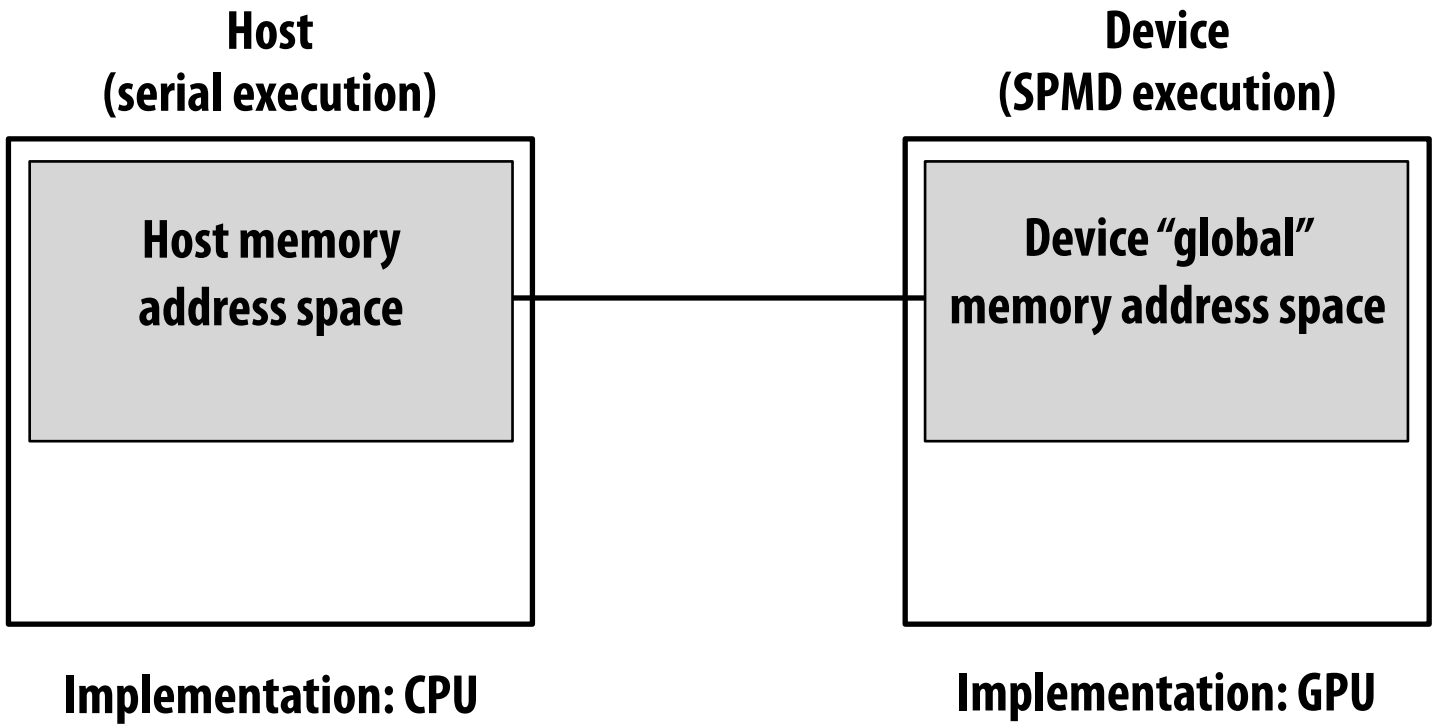
Implementation: CPU

Device
(SPMD execution)

Implementation: GPU

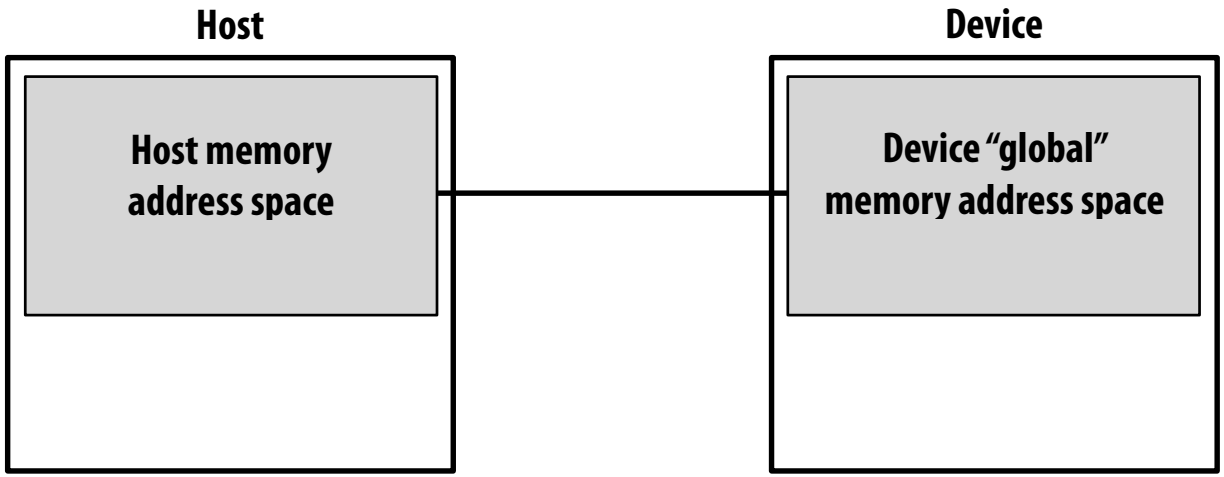
CUDA memory model

Distinct host and device address spaces



memcpy primitive

Move data between address spaces




```
float* A = new float[N];       // allocate buffer in host mem

// populate host address space pointer A
for (int i=0 i<N; i++)
   A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;                 // allocate buffer in
cudaMalloc(&deviceA, bytes);    // device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

What does cudaMemcpy remind you of?

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);
}
```

Shader programming model:

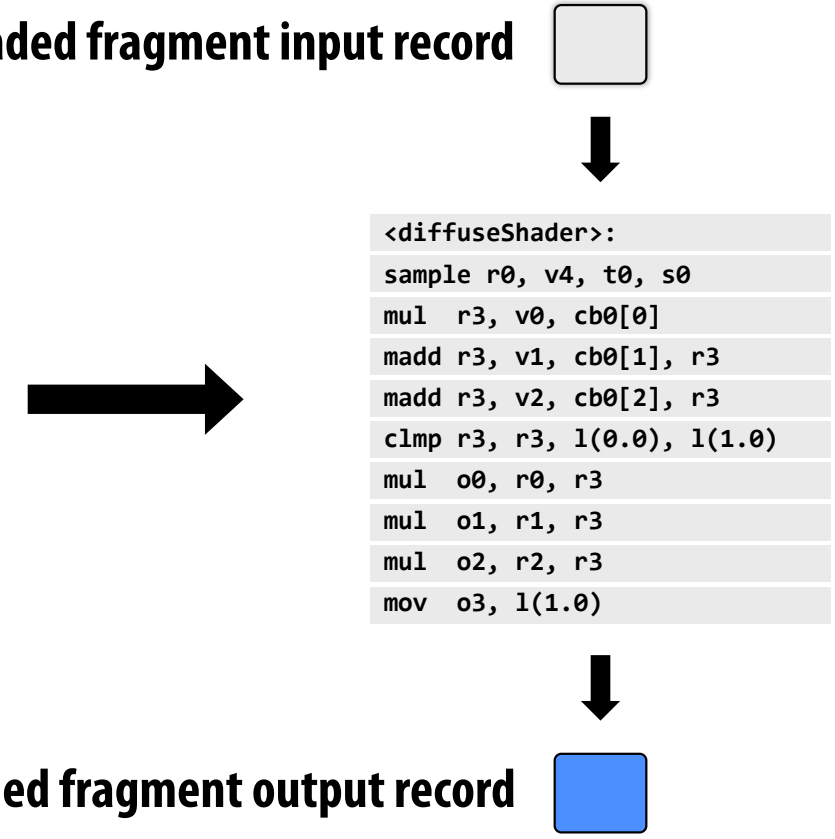
Fragments are processed *independently*, but there is no explicit parallel programming.

Independent logical sequence of control per fragment. ***

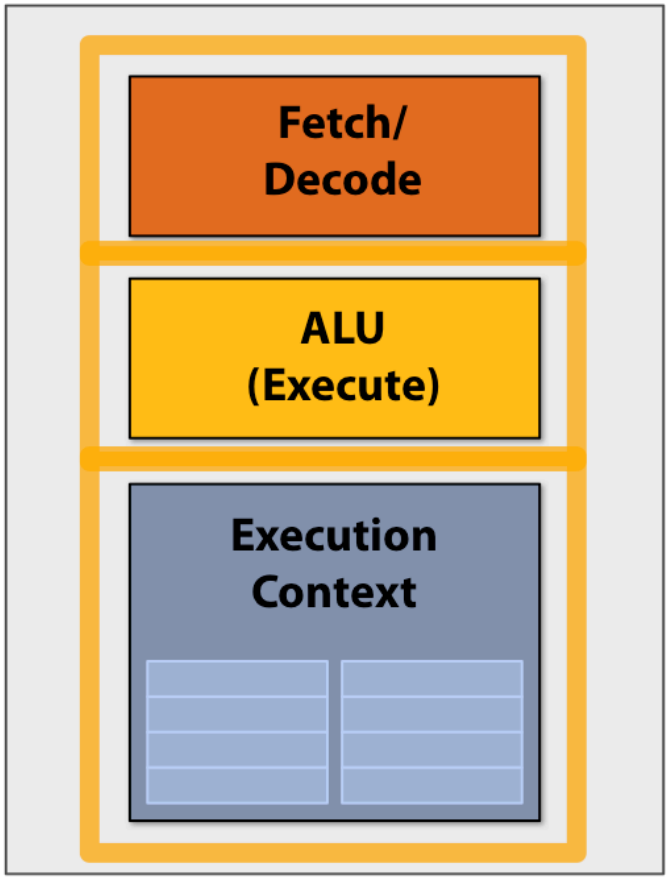
***** In this talk, references to "fragment" can be replaced with "vertex" (in the vertex shader), "primitive" (in the geometry or hull shaders), or "thread" (in OpenCL or CUDA)**

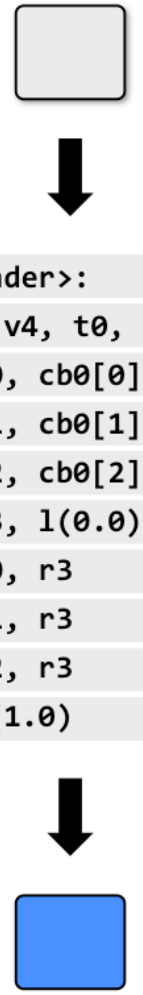
Compile shader

```
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);
}
```



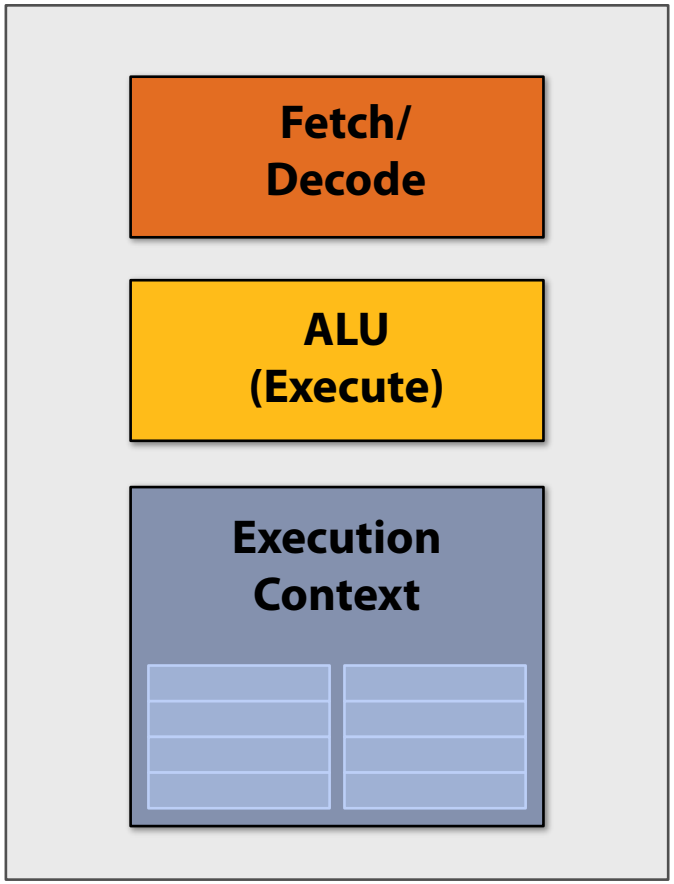
Execute shader

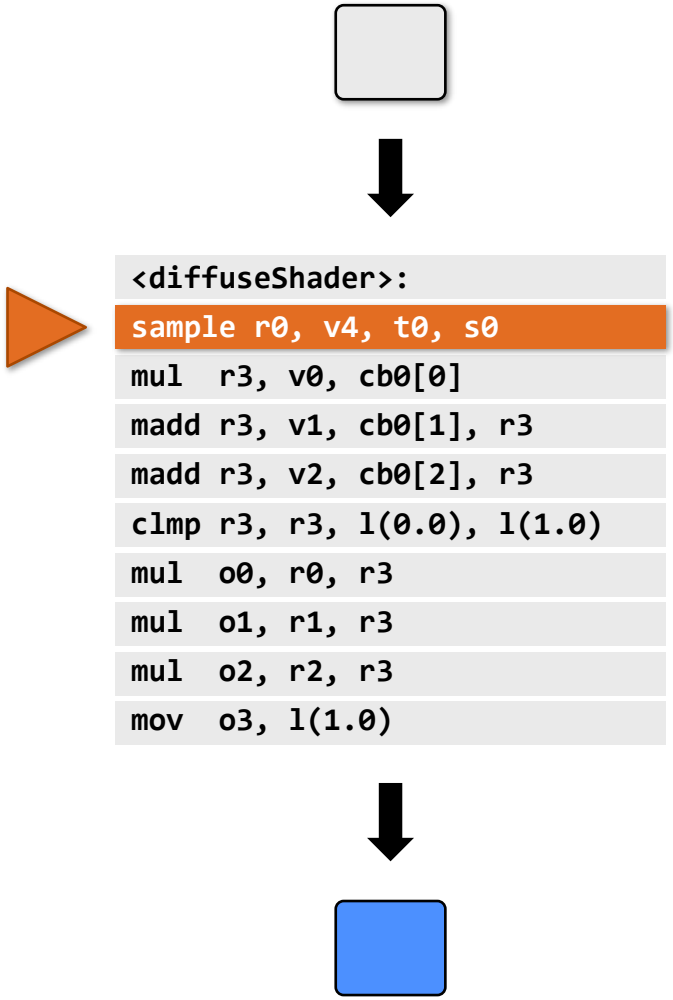




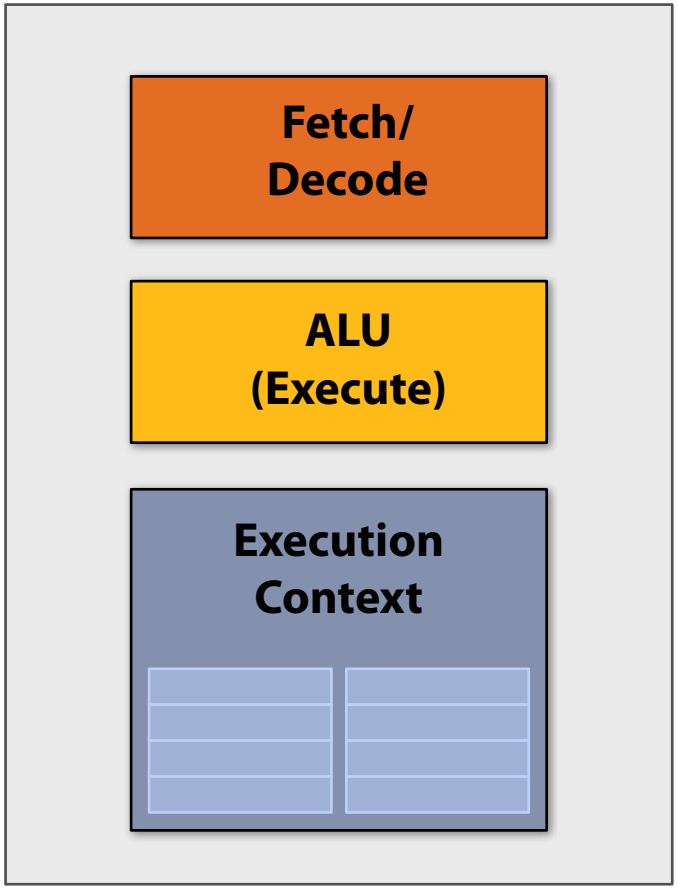
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

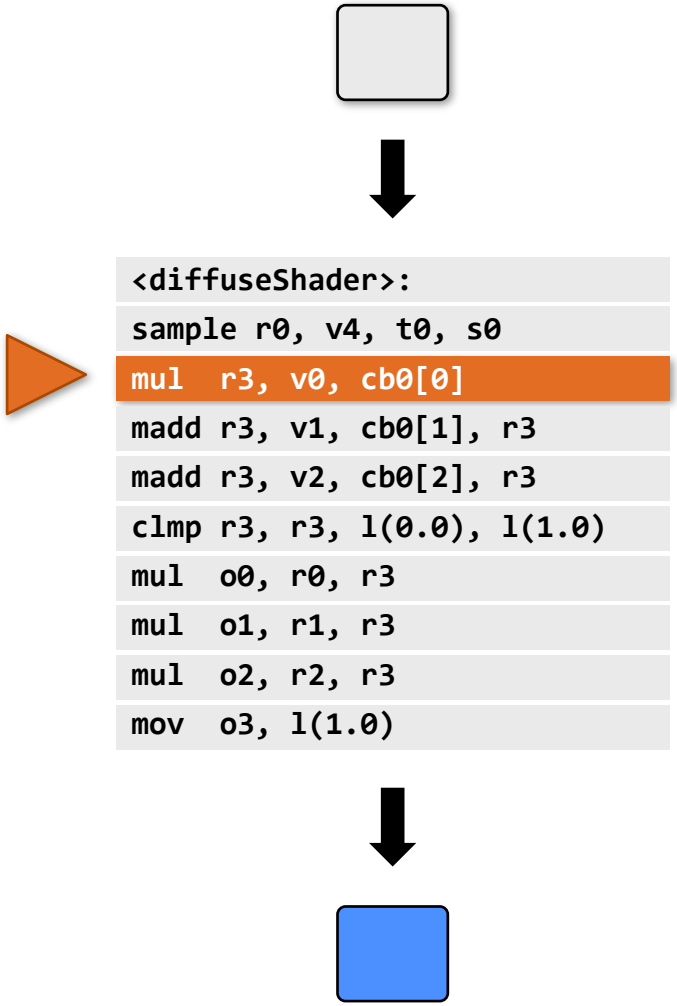
Execute shader



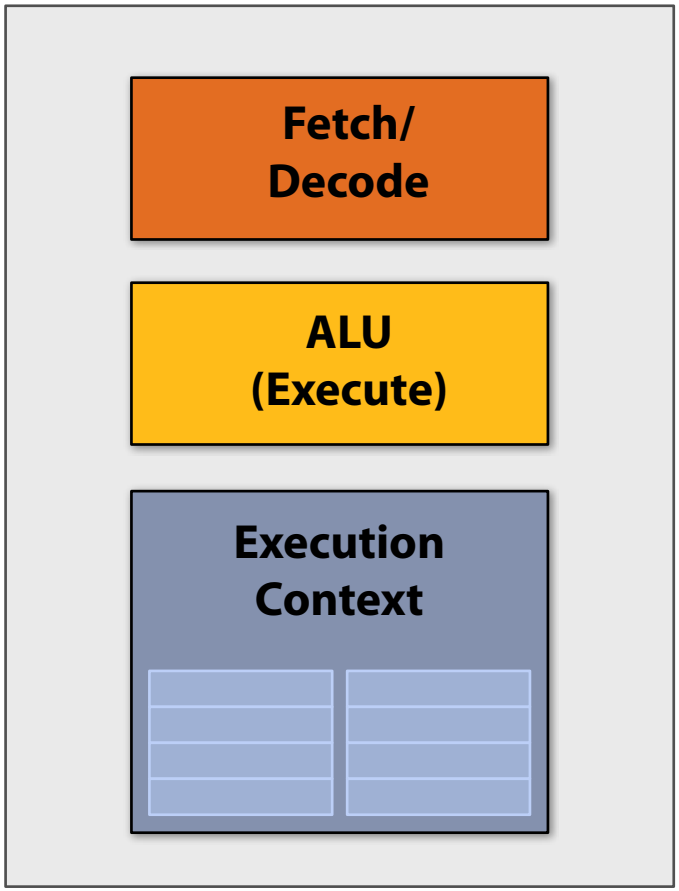


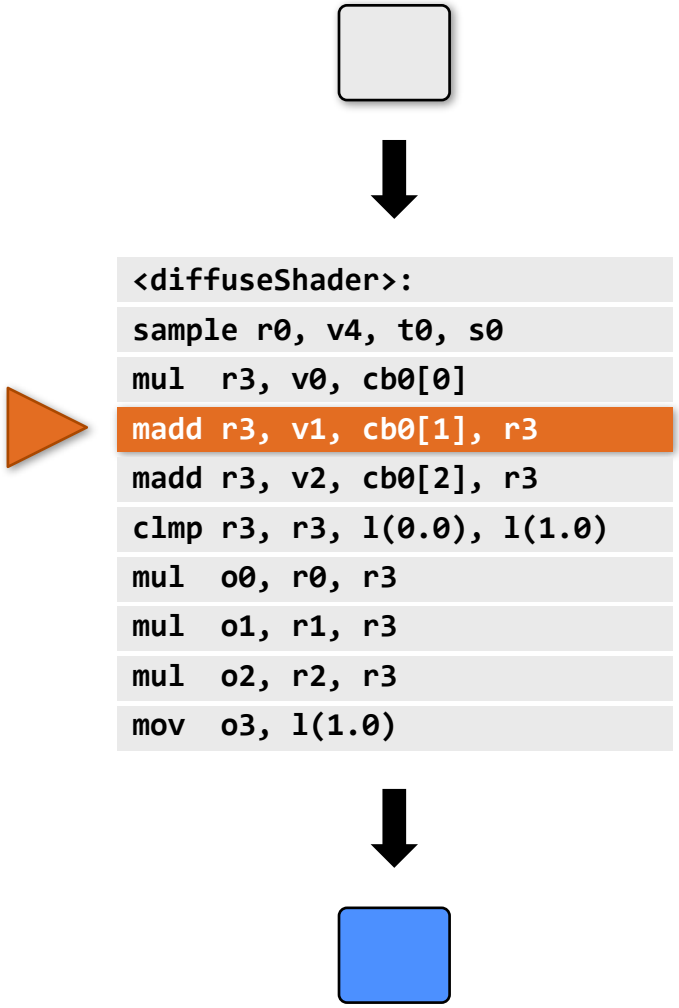
Execute shader





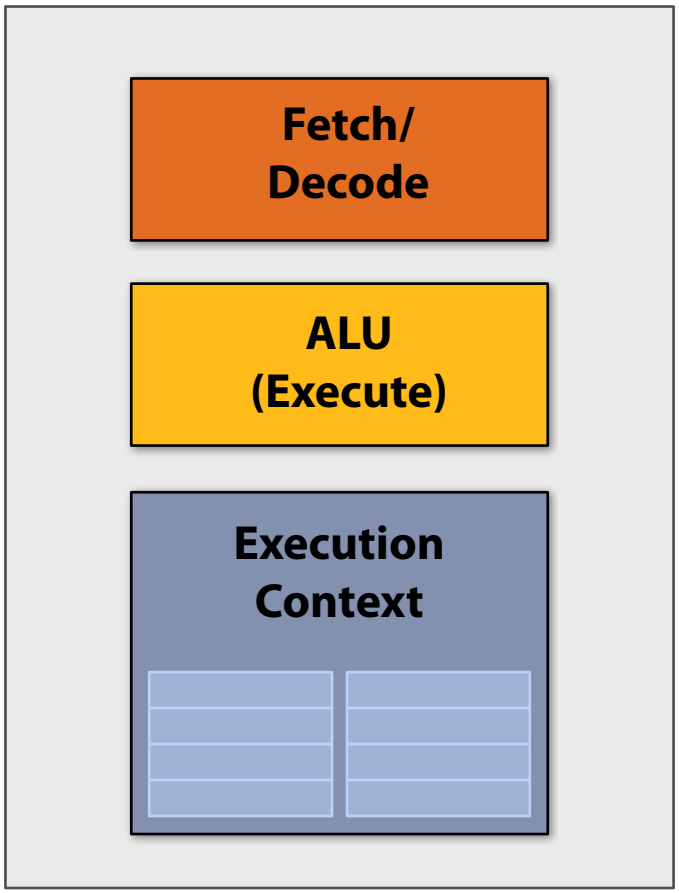
Execute shader

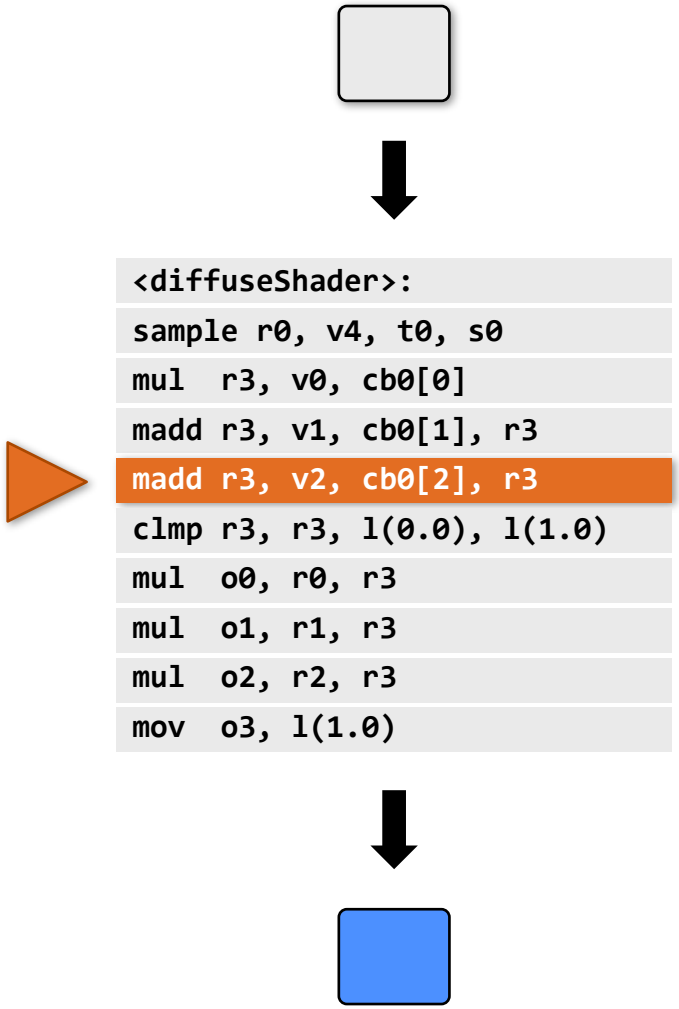




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

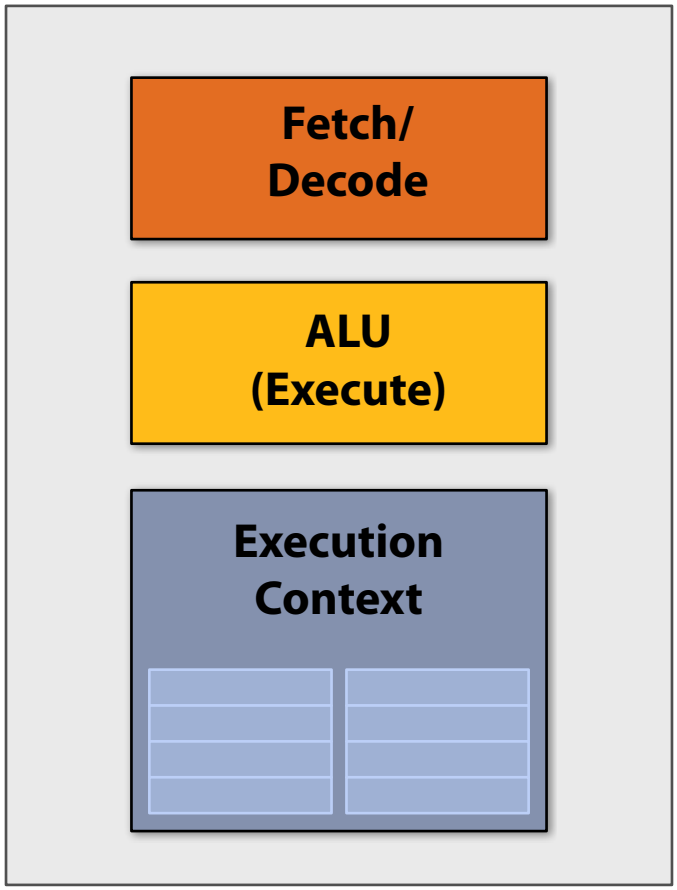
Execute shader

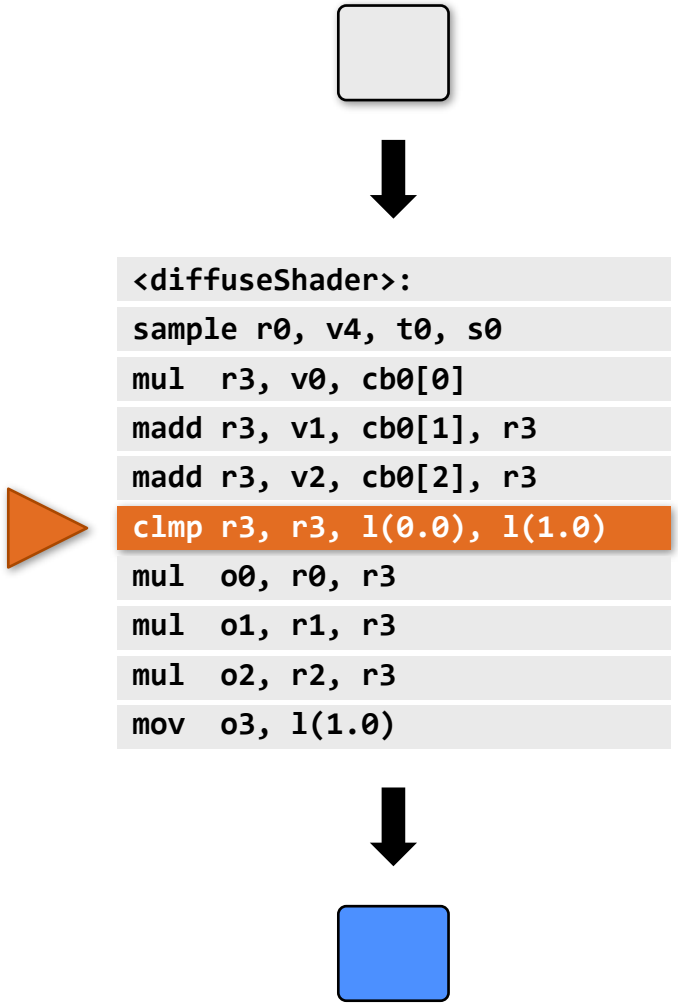




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

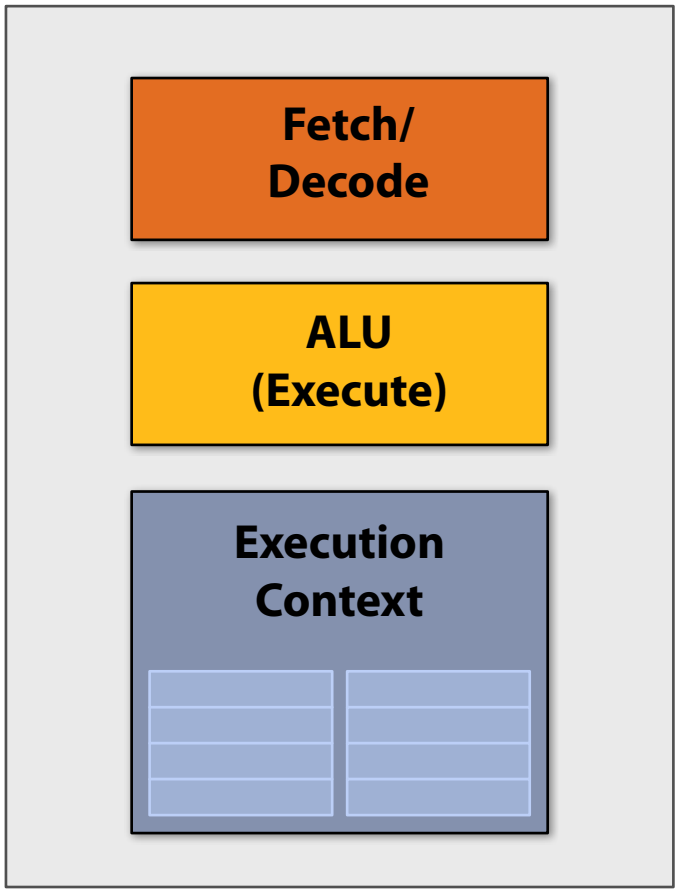
Execute shader

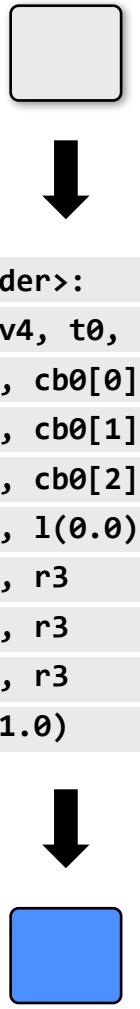




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader





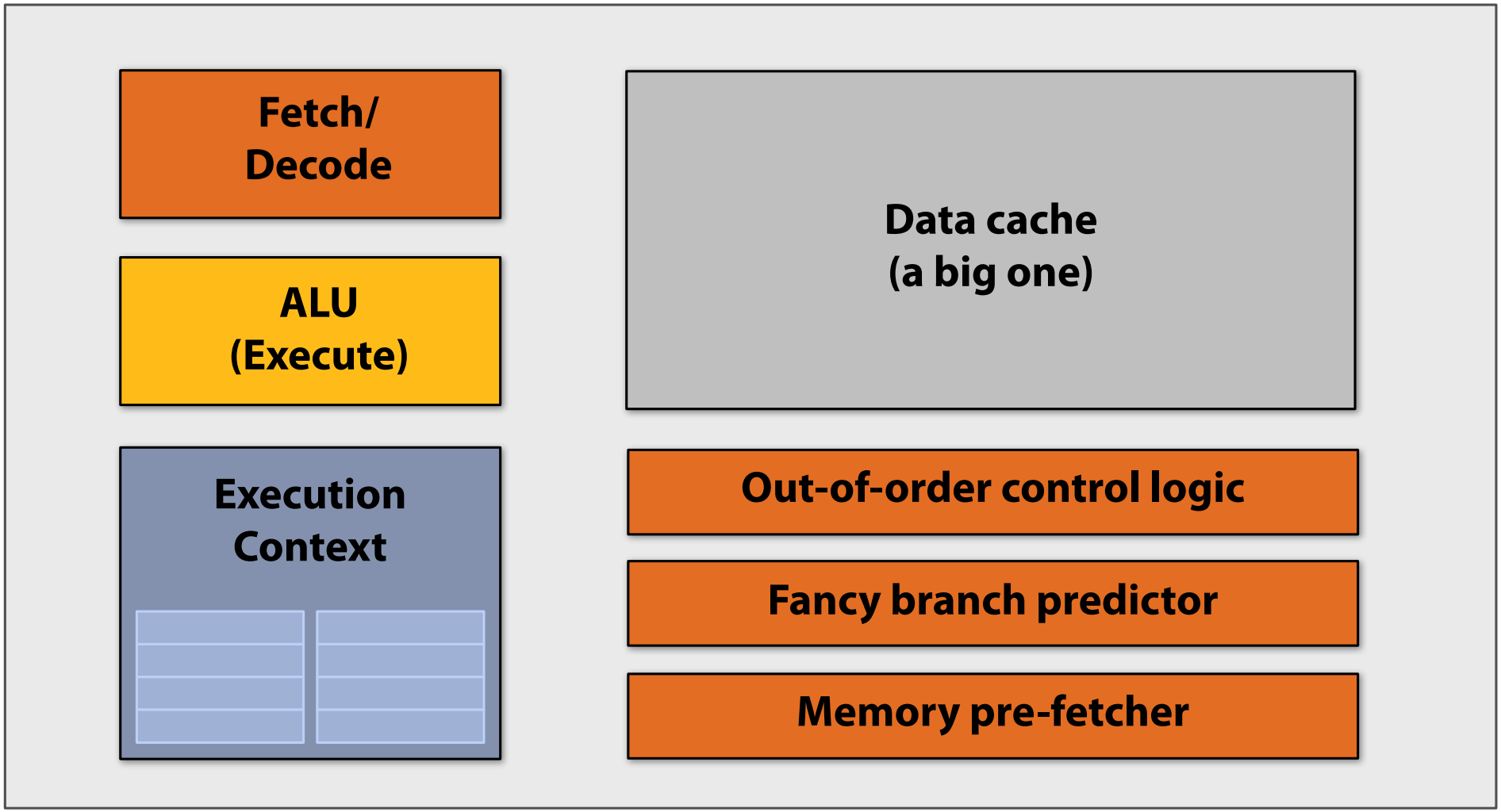
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Three major ideas that make GPU processing cores run fast

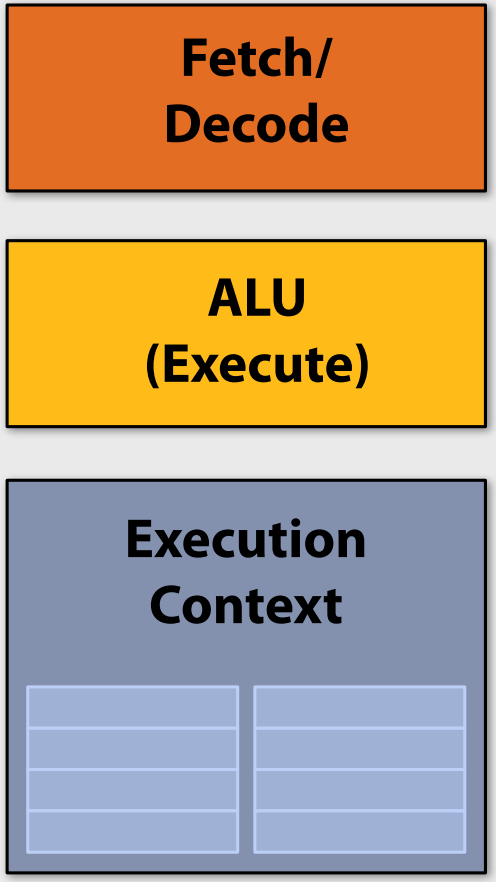
Goal: Increase parallelism

GPU PROCESSOR DESIGN

CPU-style cores



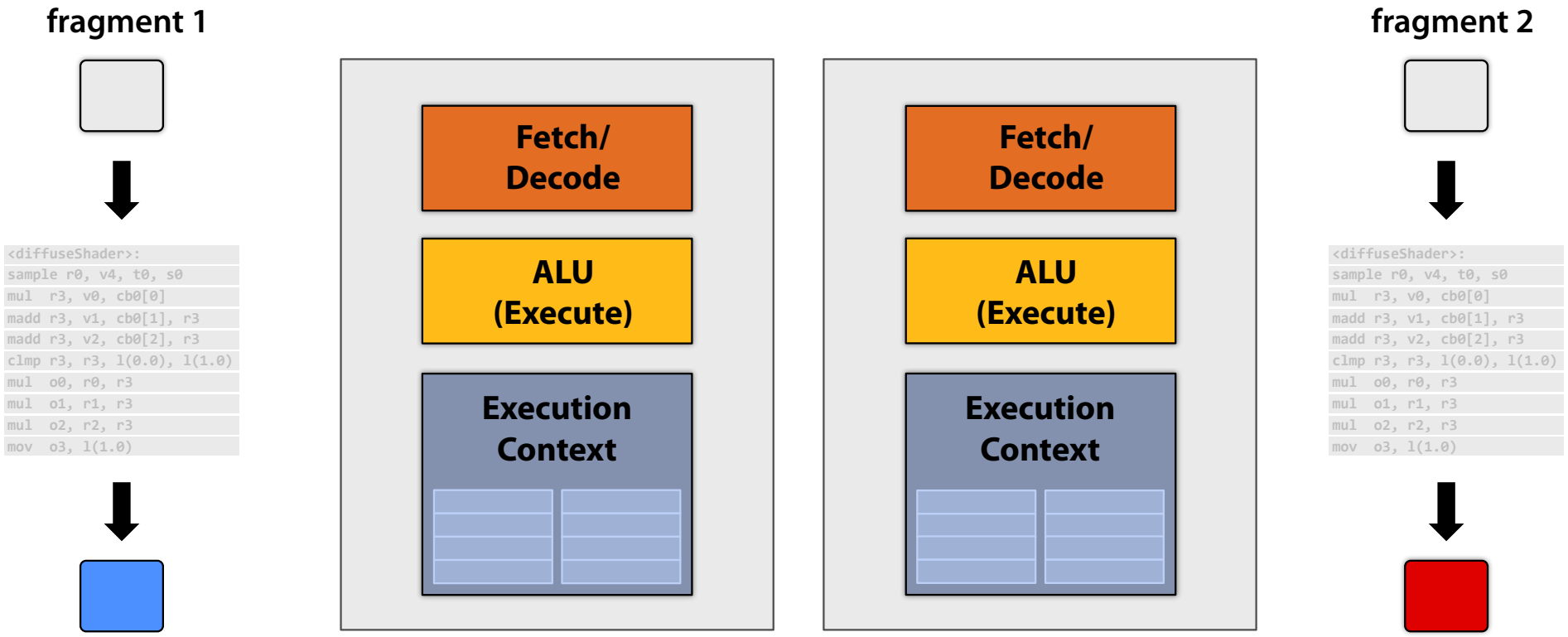
Slimming down



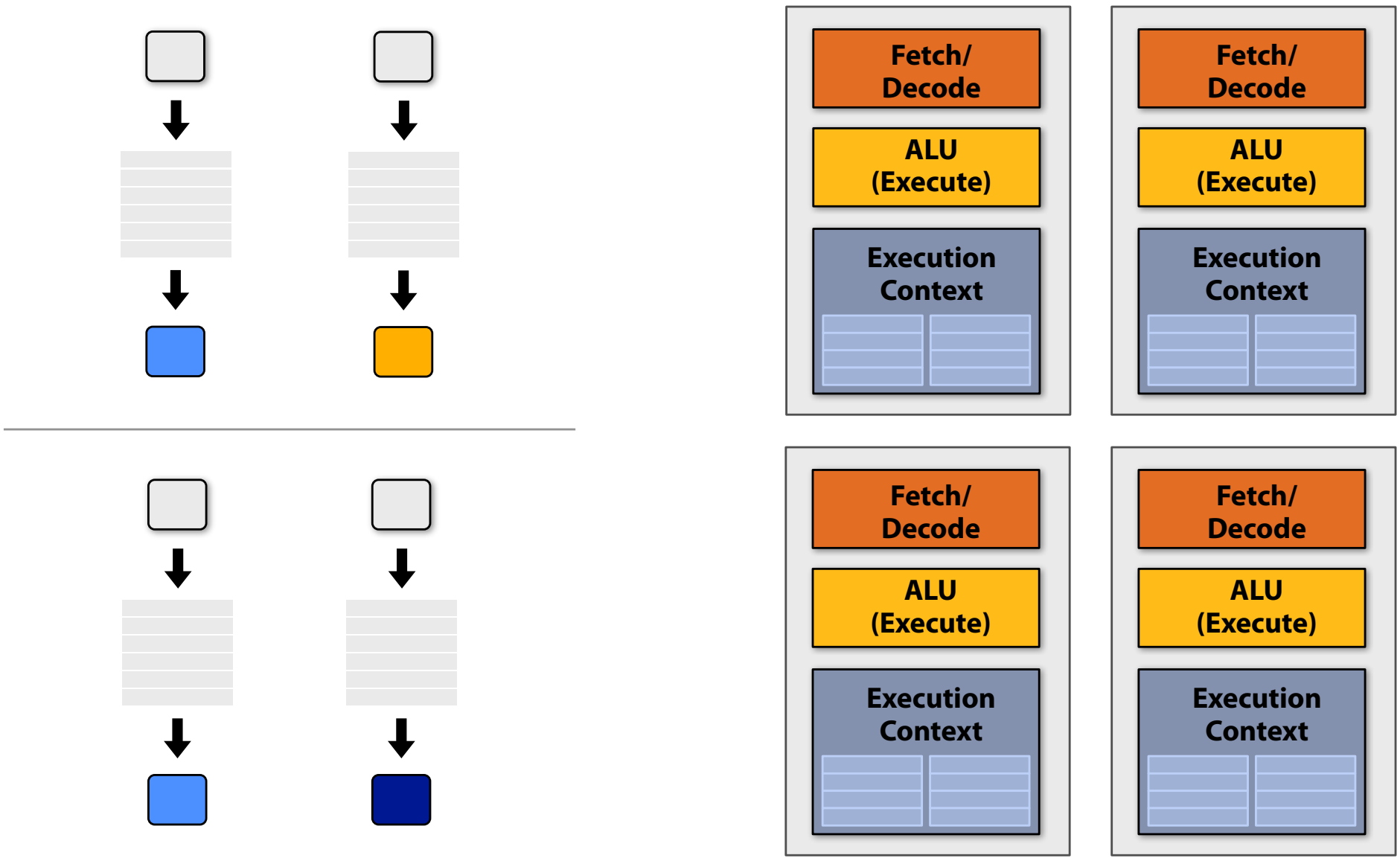
Idea #1:

Remove components that help a single instruction stream run fast

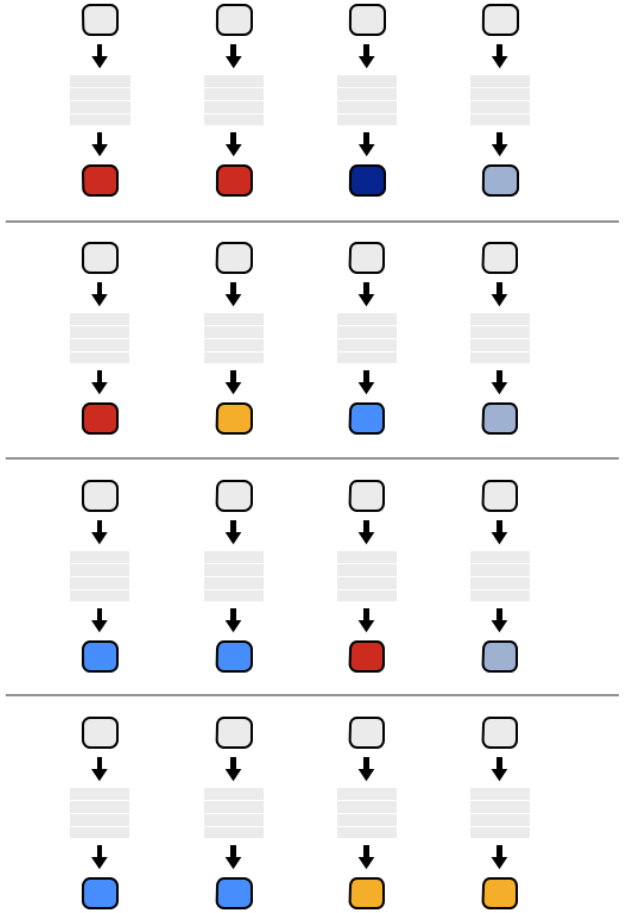
Tow cores (two fragments in parallel)

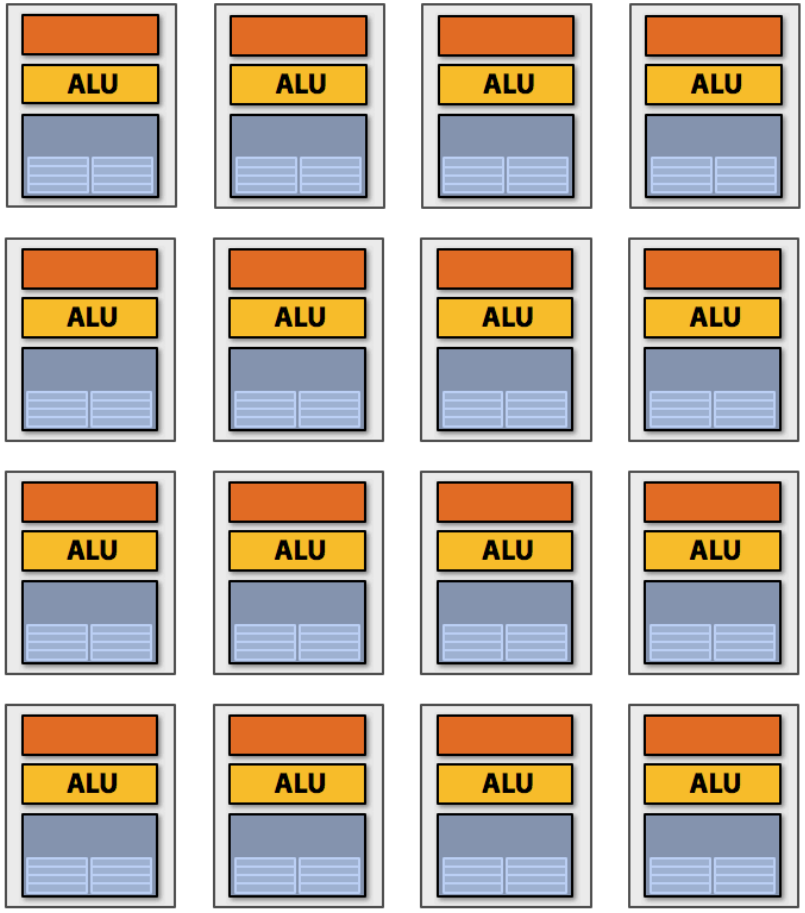


Four cores (four fragments in parallel)



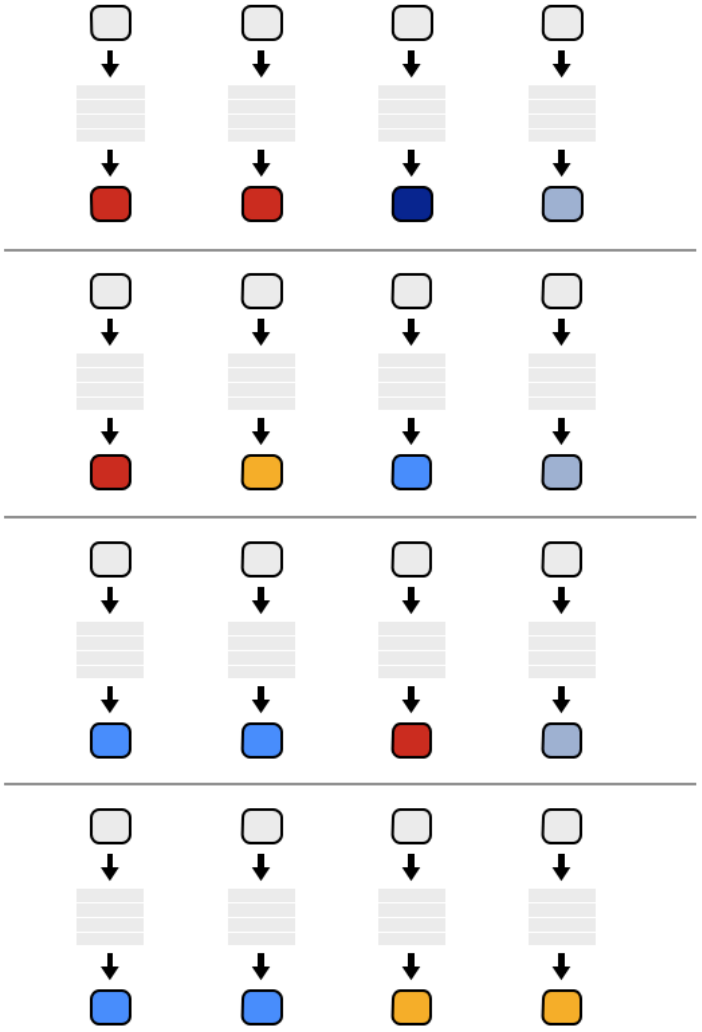
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

Instruction stream sharing



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

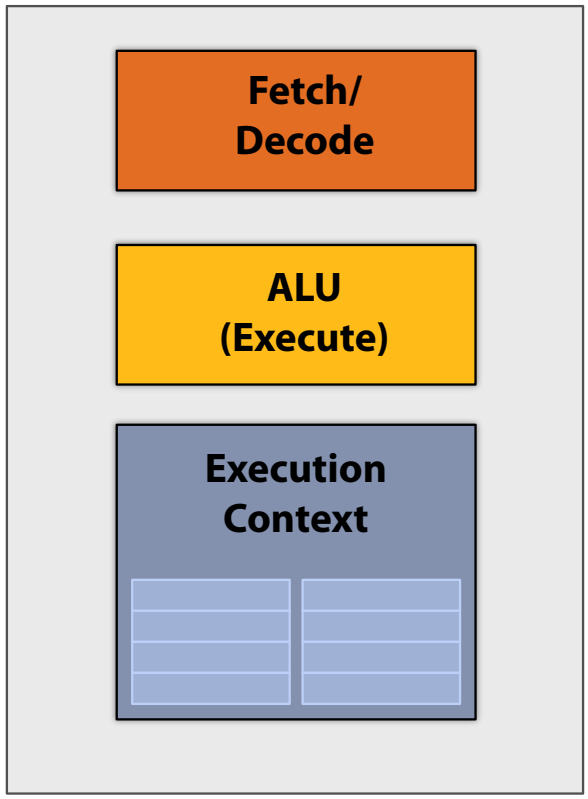
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

What we learned: Idea 1

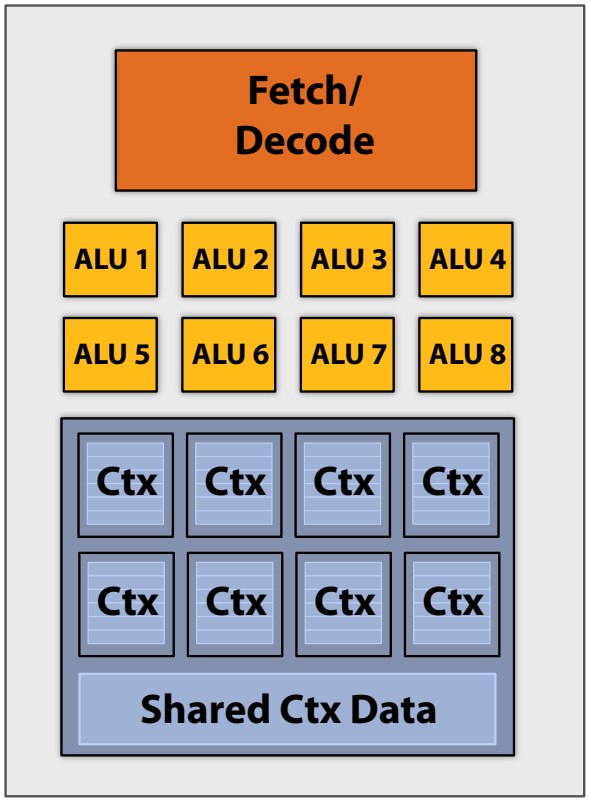
CPU

GPU

Recall: simple processing core



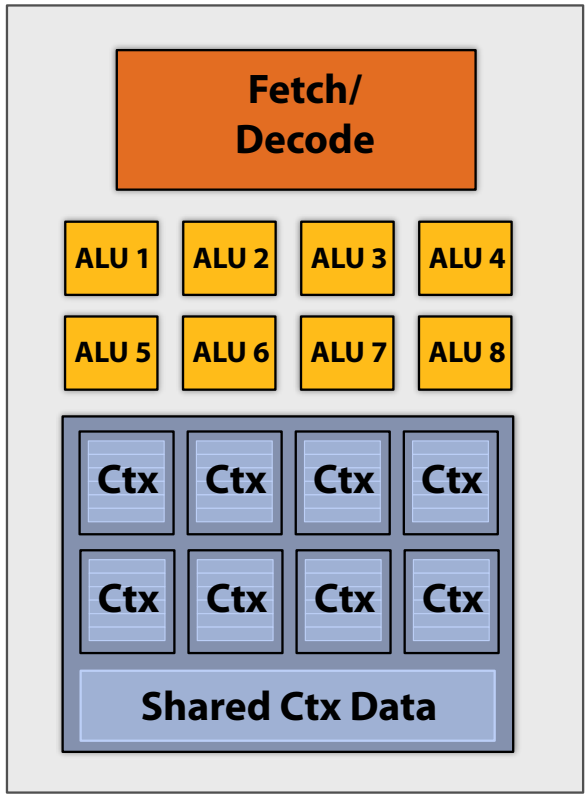
Add ALUs



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

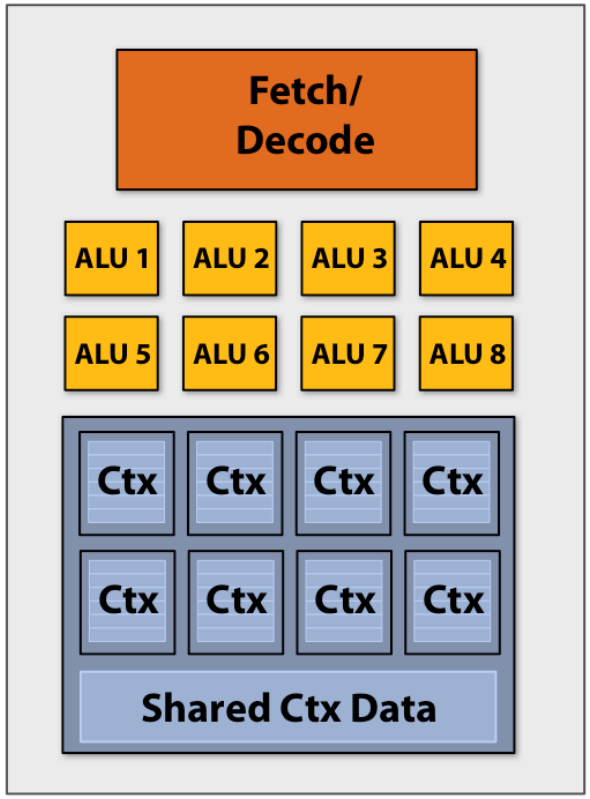



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

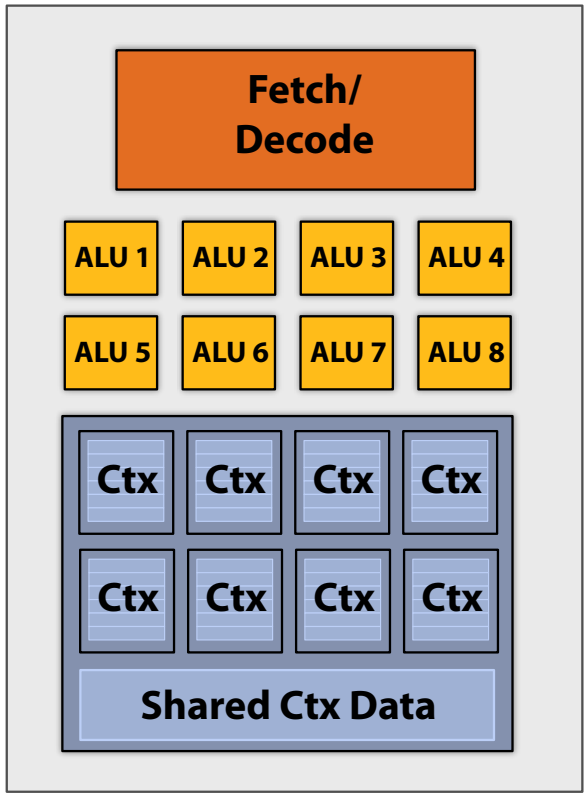


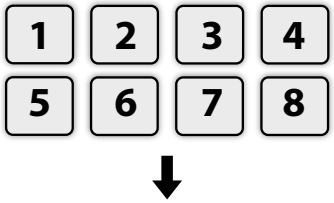

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul    vec_r3, vec_v0, cb0[0]
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp   vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul    vec_o0, vec_r0, vec_r3
VEC8_mul    vec_o1, vec_r1, vec_r3
VEC8_mul    vec_o2, vec_r2, vec_r3
VEC8_mov    o3, l(1.0)
```

New compiled shader:

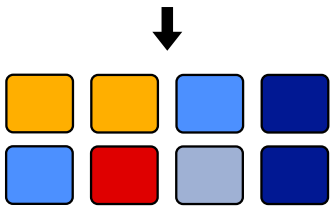
Processes eight fragments using vector ops on vector registers

Modifying the shader

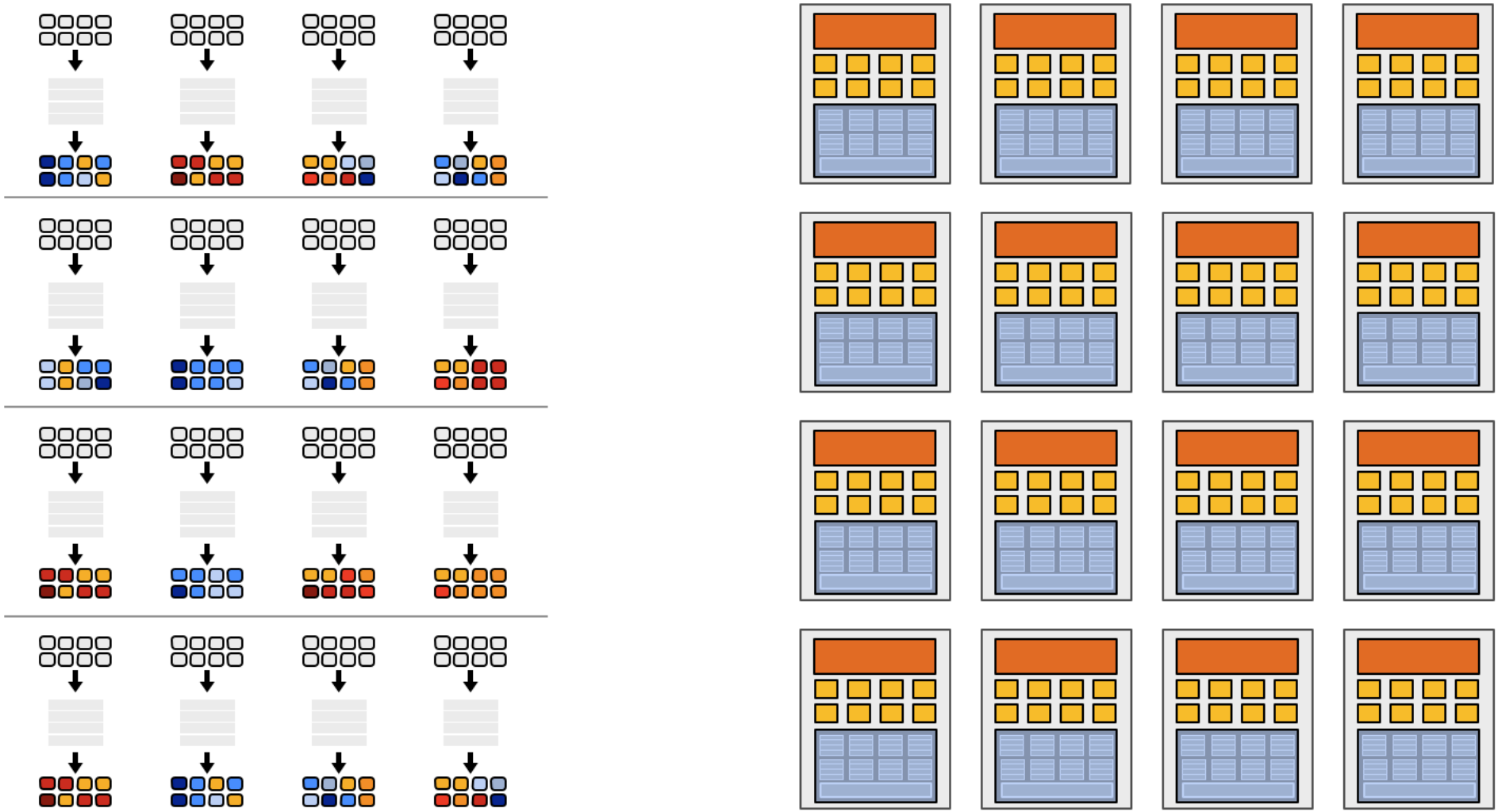





```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```

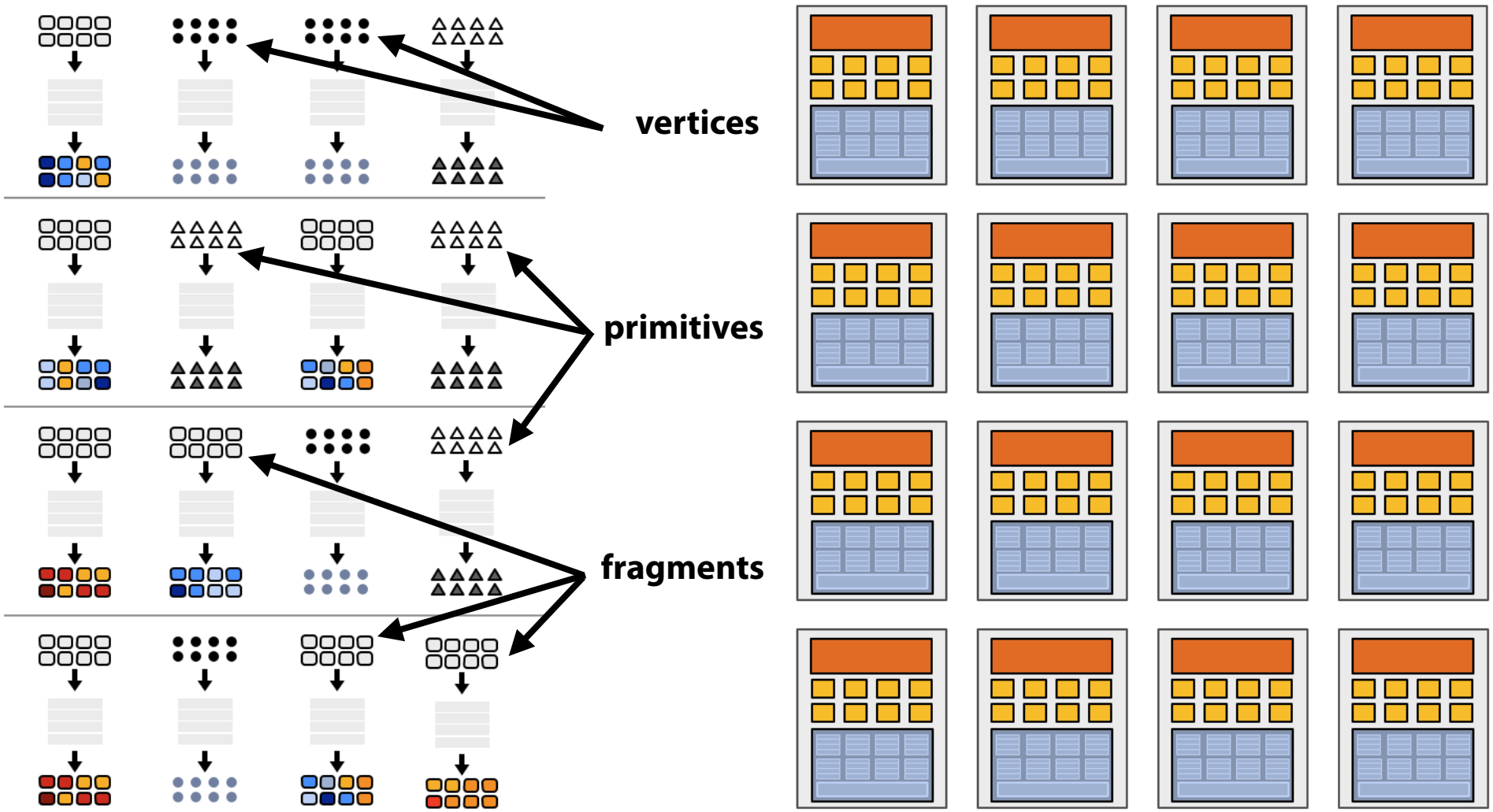


128 fragments in parallel



16 cores = 128 ALUs, 16 simultaneous instruction streams

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

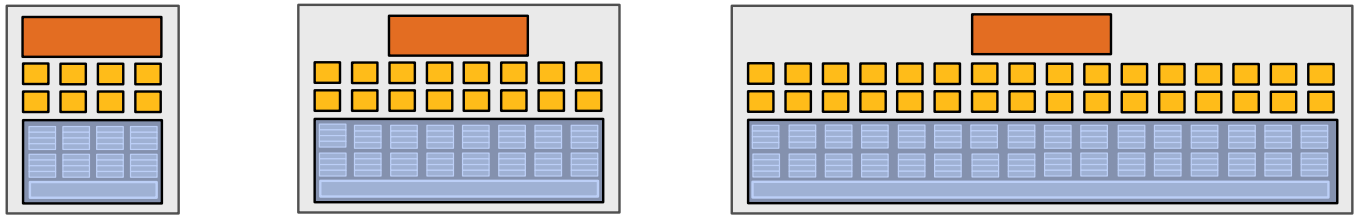


Clarification

SIMD processing does not imply SIMD instructions in an ISA

- Option 1: explicit vector instructions
  - x86 SSE (4-wide), Intel AVX (8-wide), Intel Larrabee (16-wide)

- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")

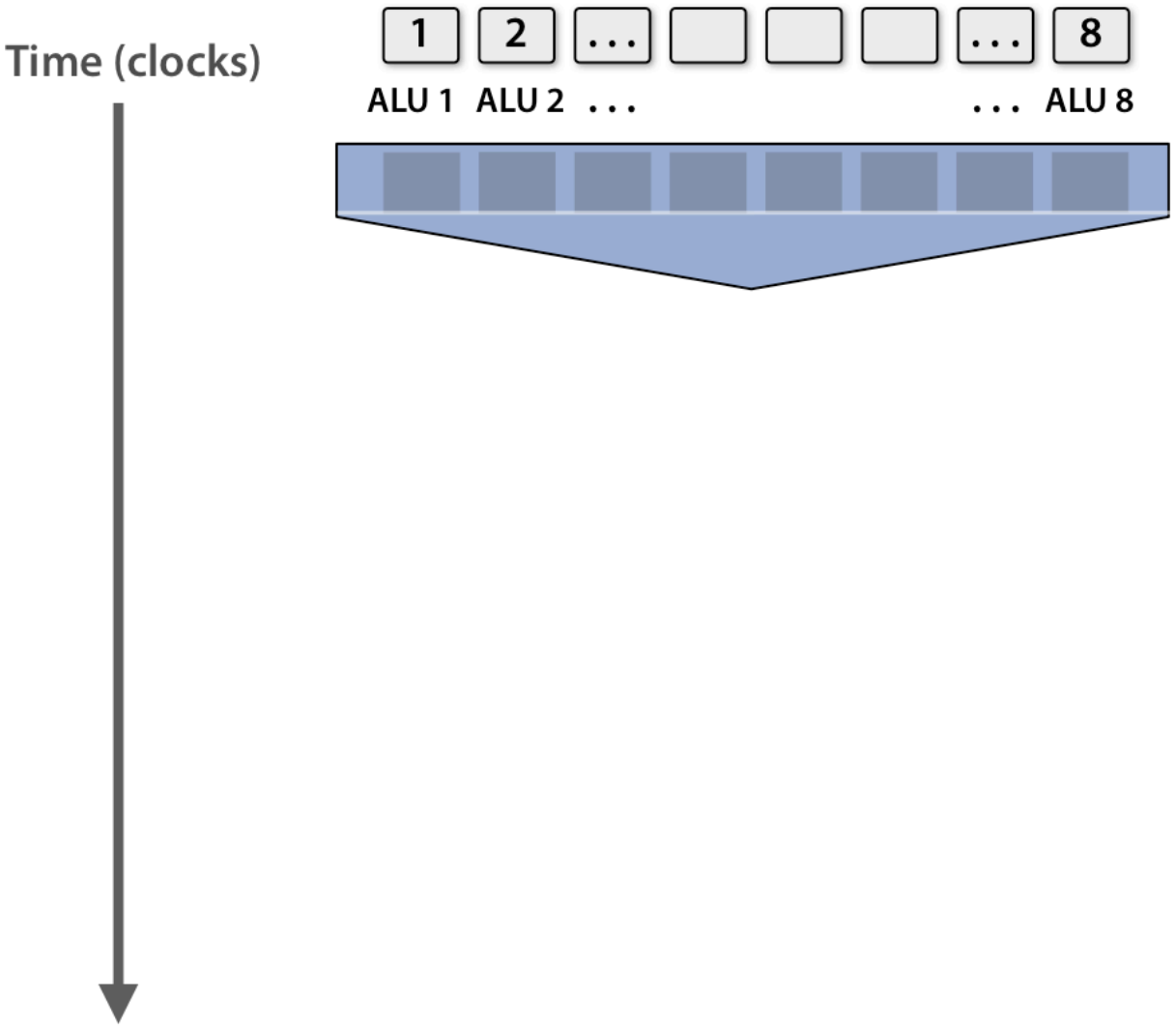


In modern GPUs: 16 to 64 fragments share an instruction stream.

What we learned: Idea 2

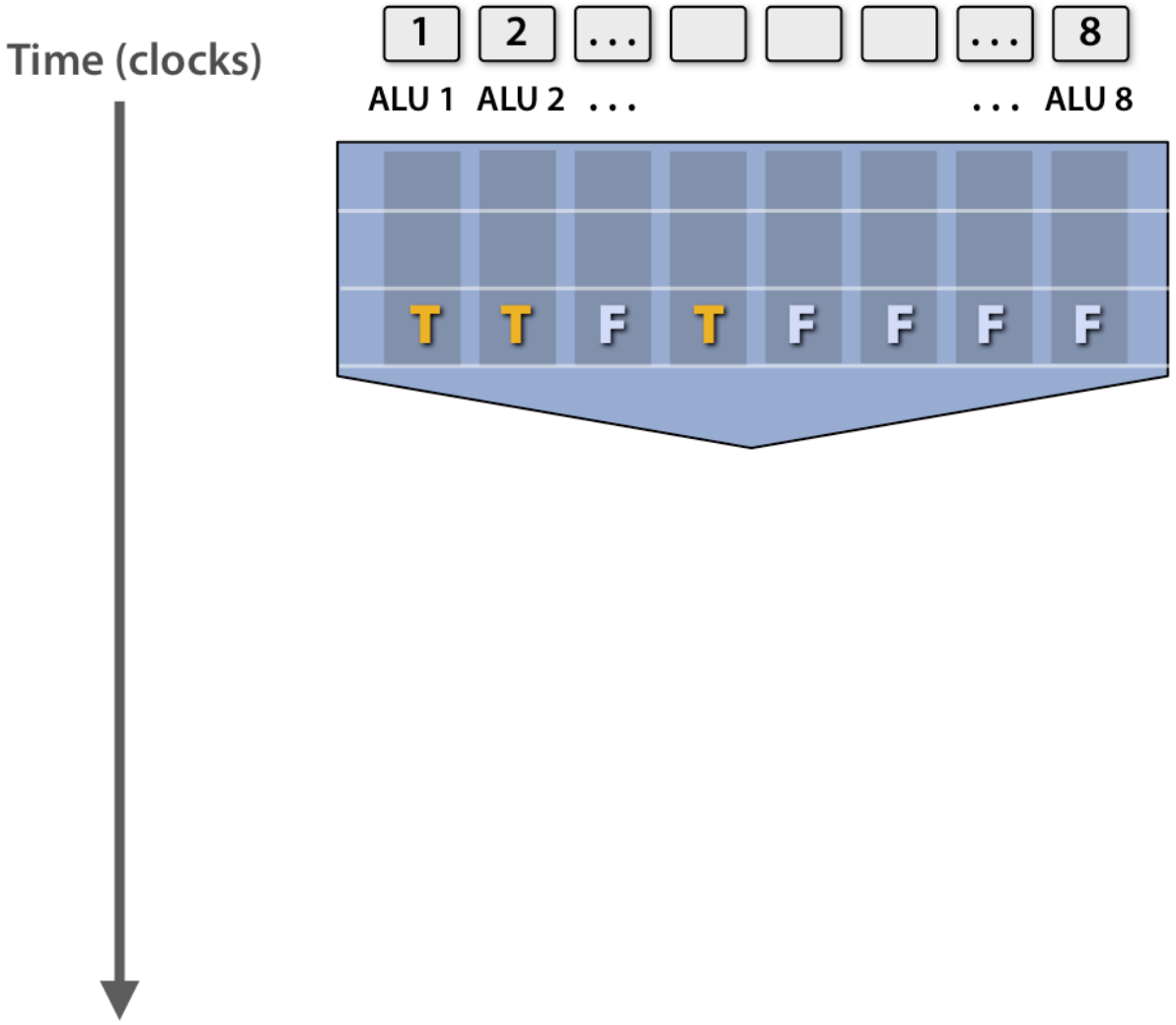
Each GPU core has many ALUs

But what about branches?



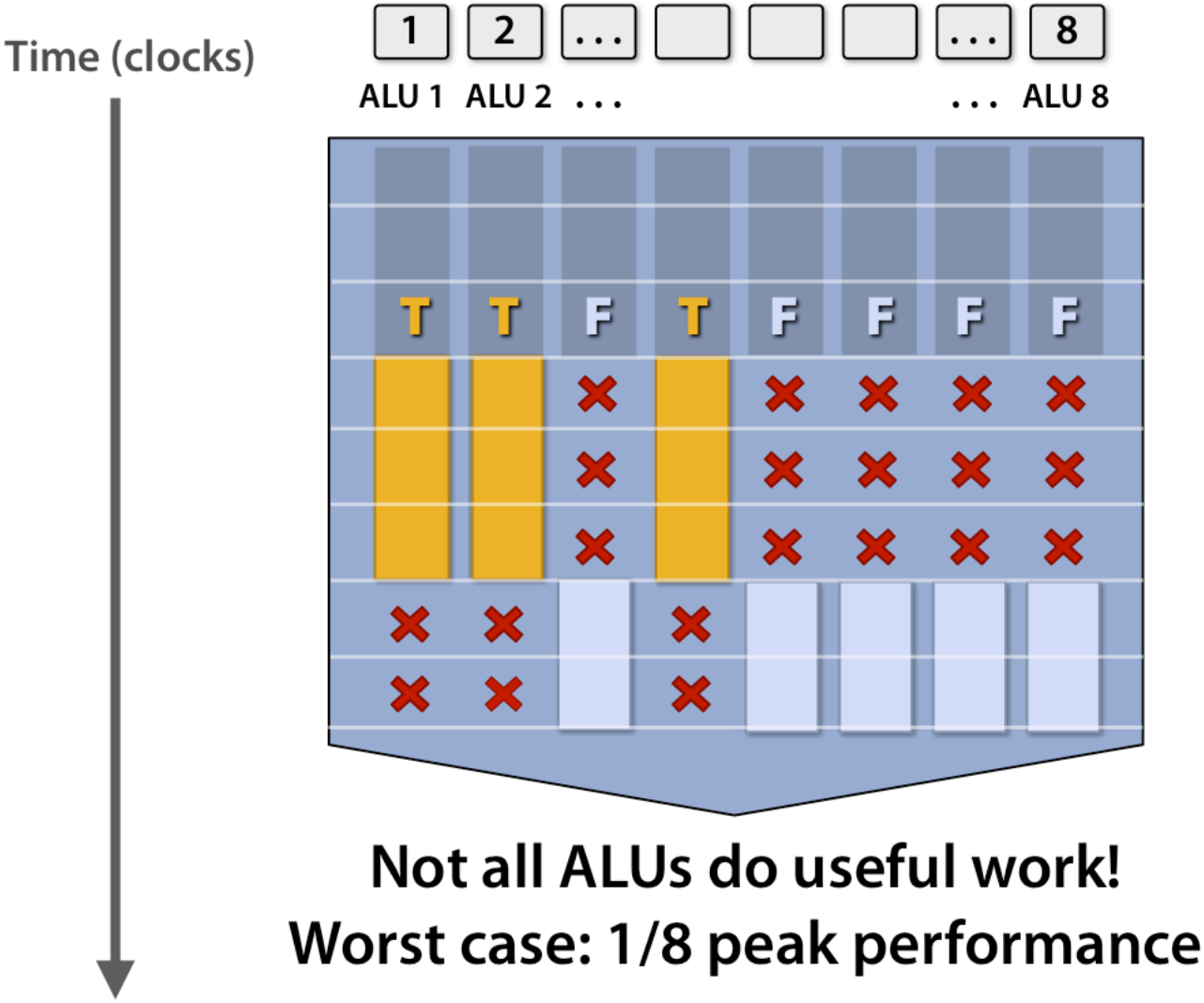

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

But what about branches?

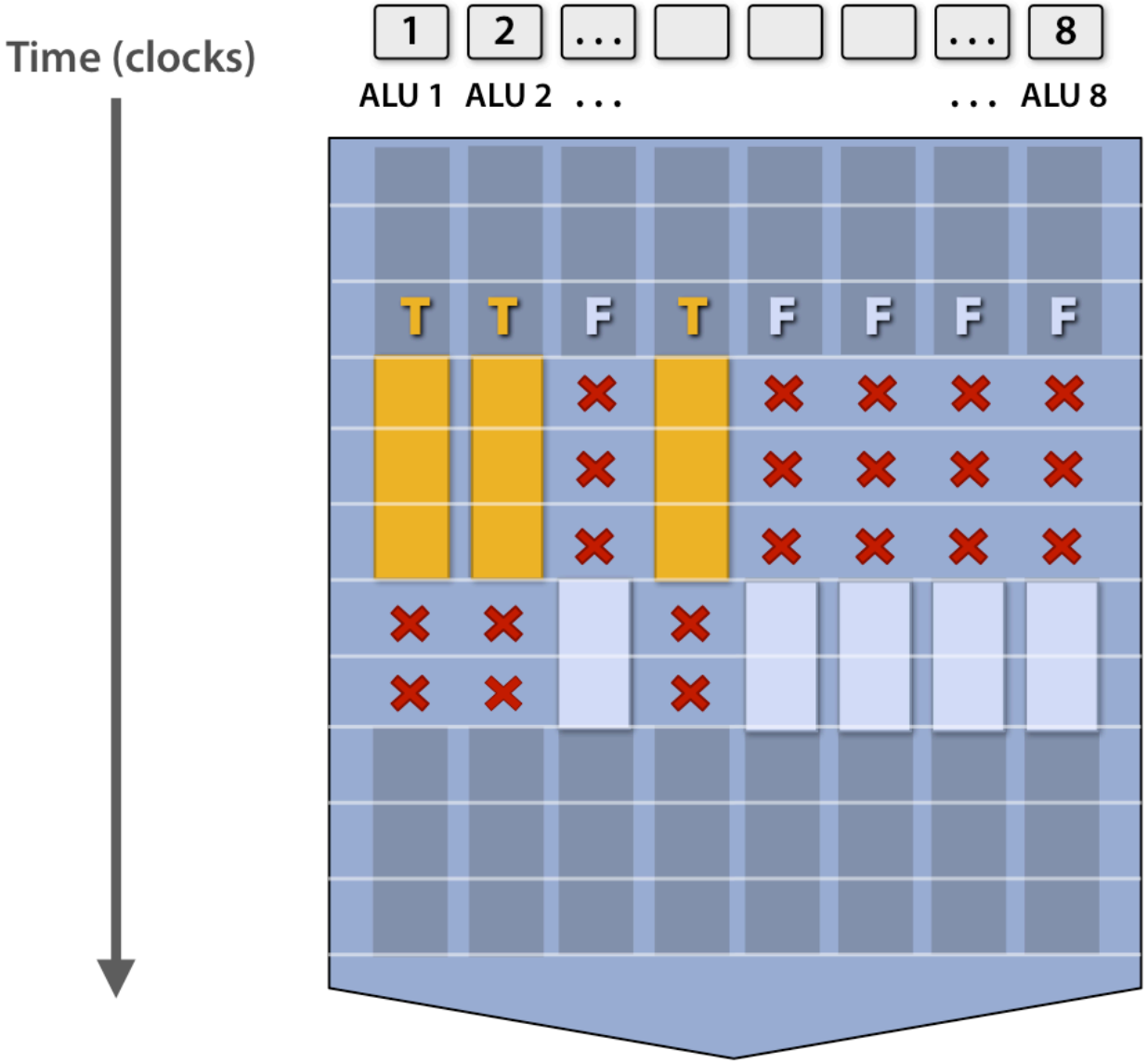



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Not all ALUs do useful work!

Worst case: 1/8 peak performance

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Terminology

- **"Coherent" execution*** (admittedly fuzzy definition): when processing of different entities is similar, and thus can share resources for efficient execution**
  - **Instruction stream coherence: different fragments follow same sequence of logic**
  - **Memory access coherence:**
    - **Different fragments access similar data (avoid memory transactions by reusing data in cache)**
    - **Different fragments simultaneously access contiguous data (enables efficient, bulk granularity memory transactions)**

- **"Divergence": lack of coherence**
  - **Usually used in reference to instruction streams (divergent execution does not make full use of SIMD processing)**

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100's to 1000's of cycles

We've removed the fancy caches and logic that helps avoid stalls.

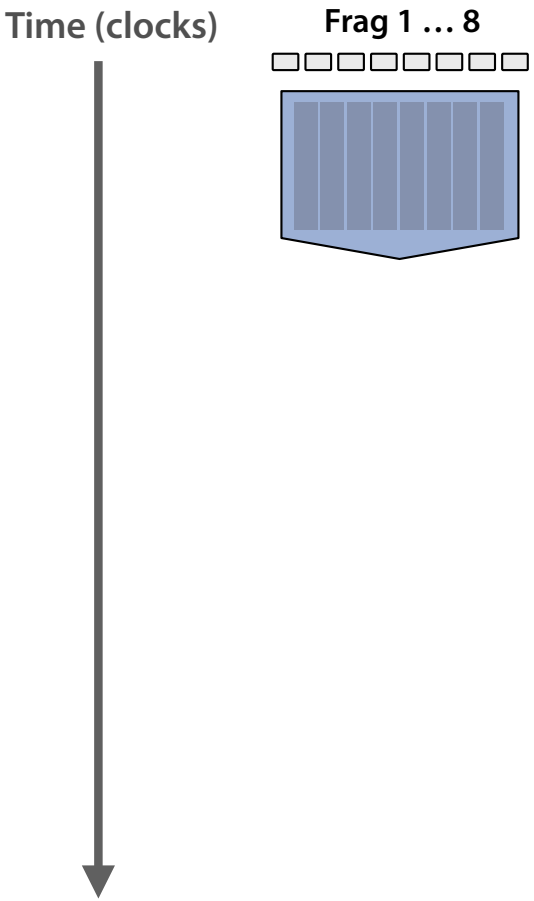
But we have LOTS of independent fragments.

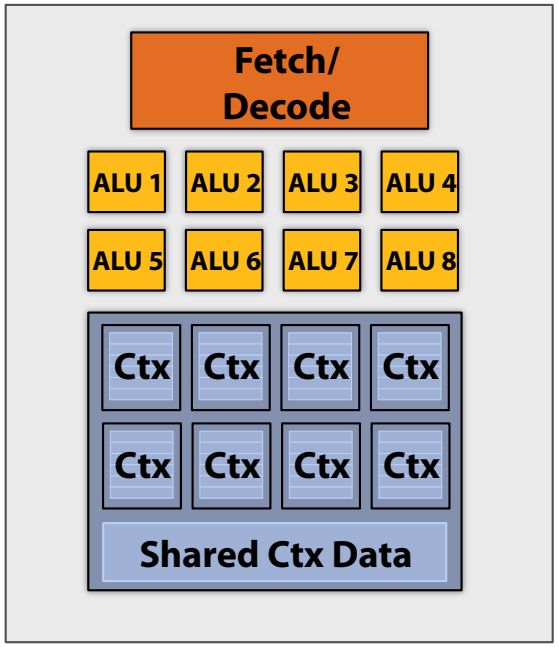
(Way more fragments to process than ALUs)

Idea #3:

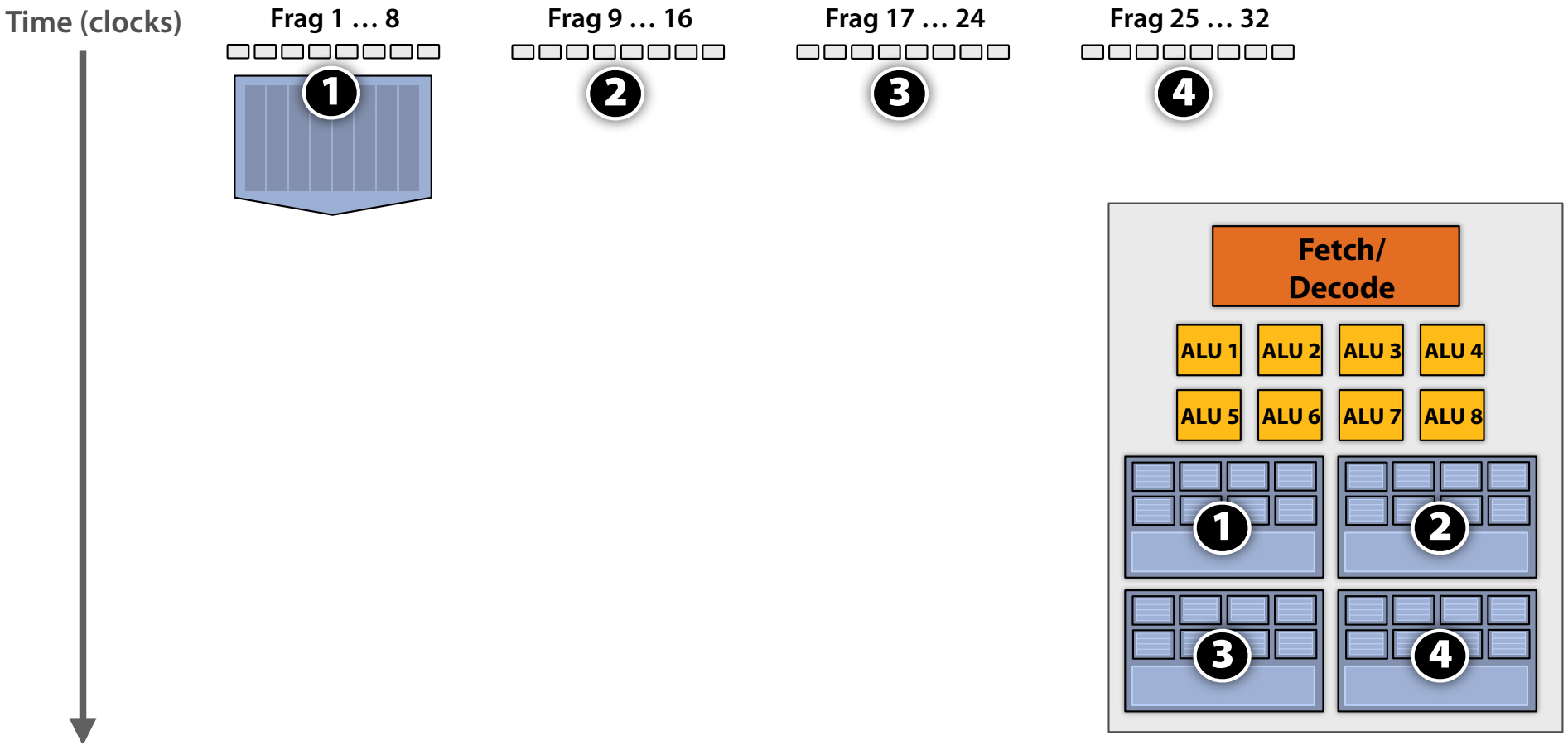
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

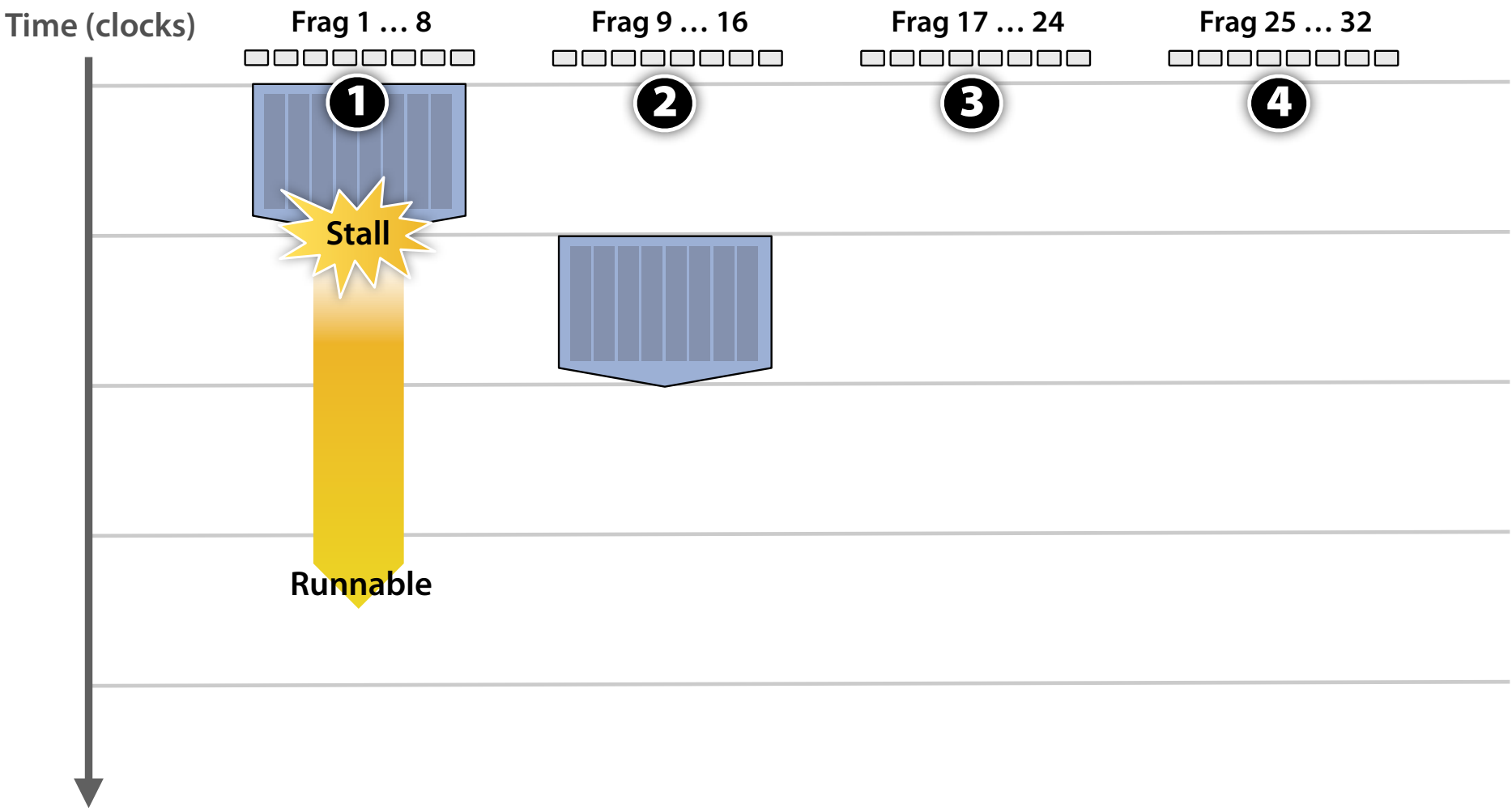




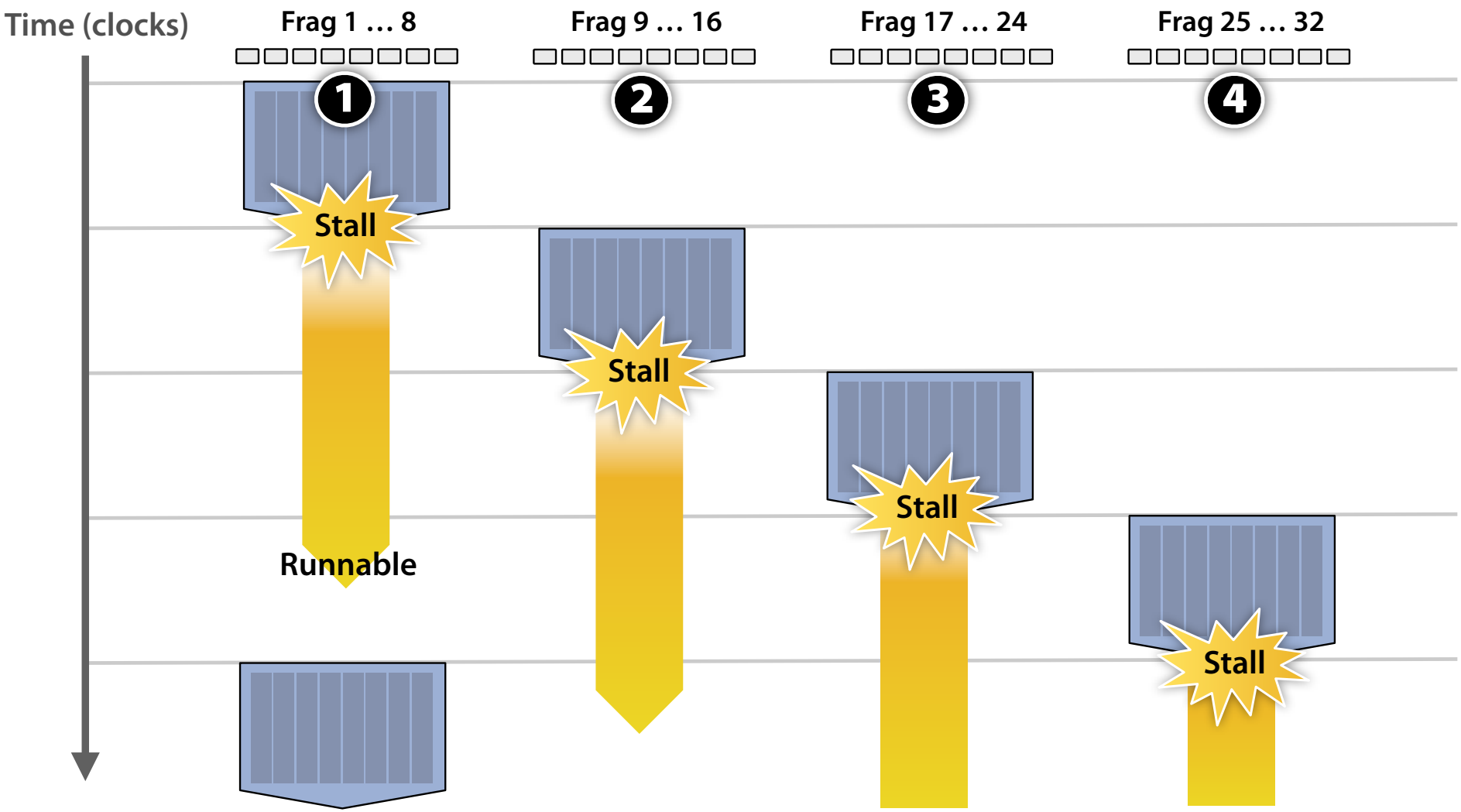
Hiding shader stalls



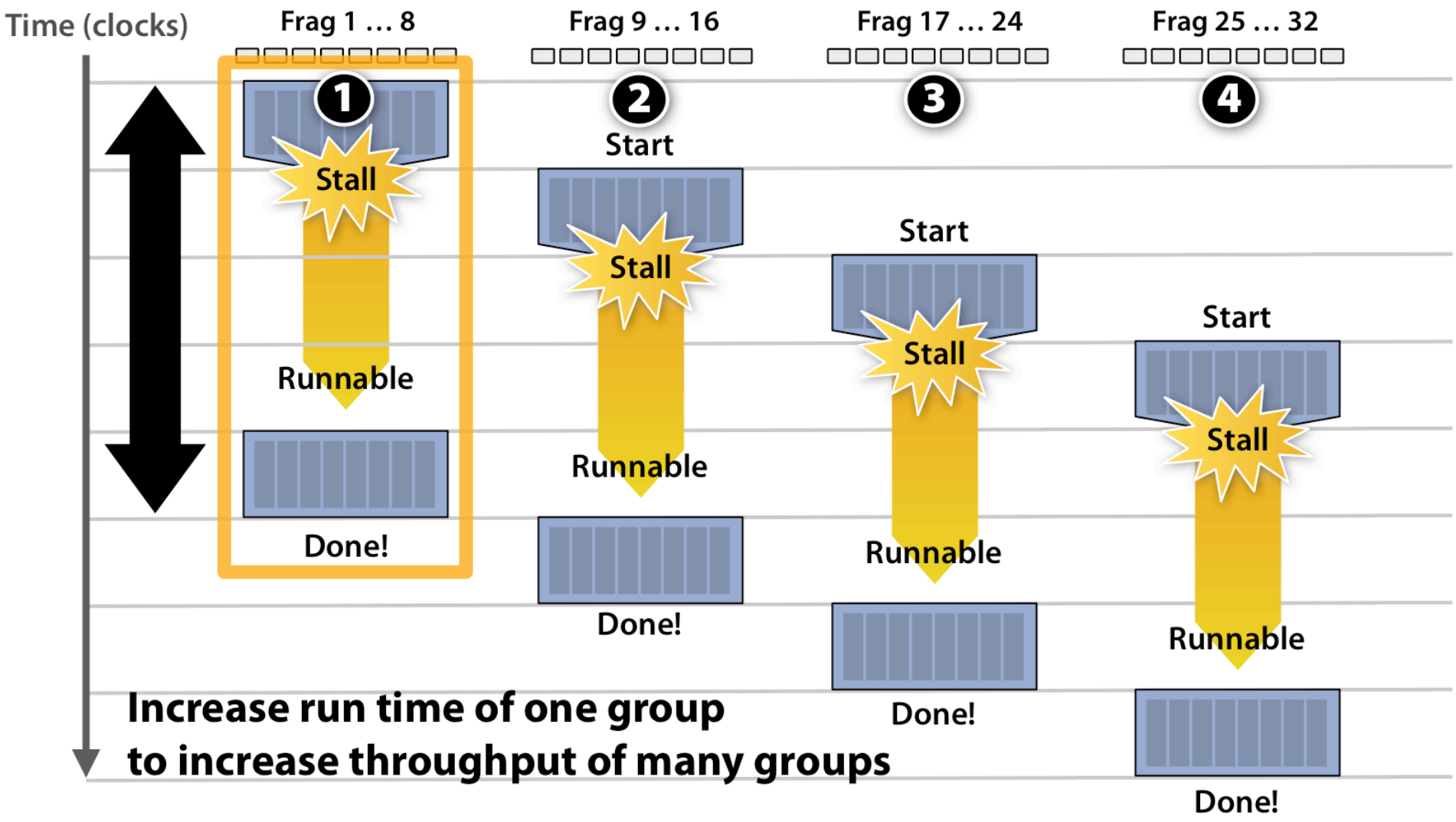
Hiding shader stalls



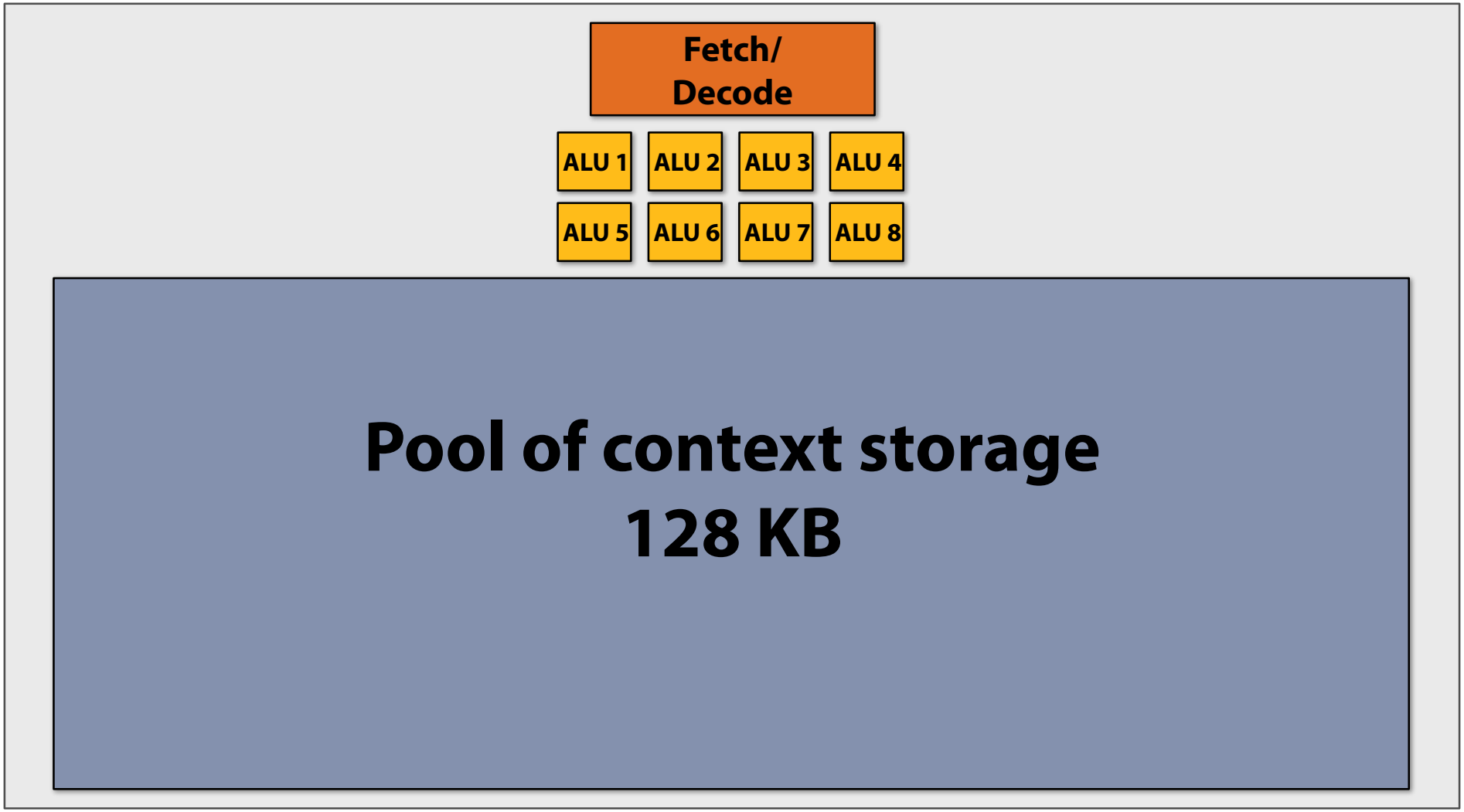
Hiding shader stalls



Throughput!

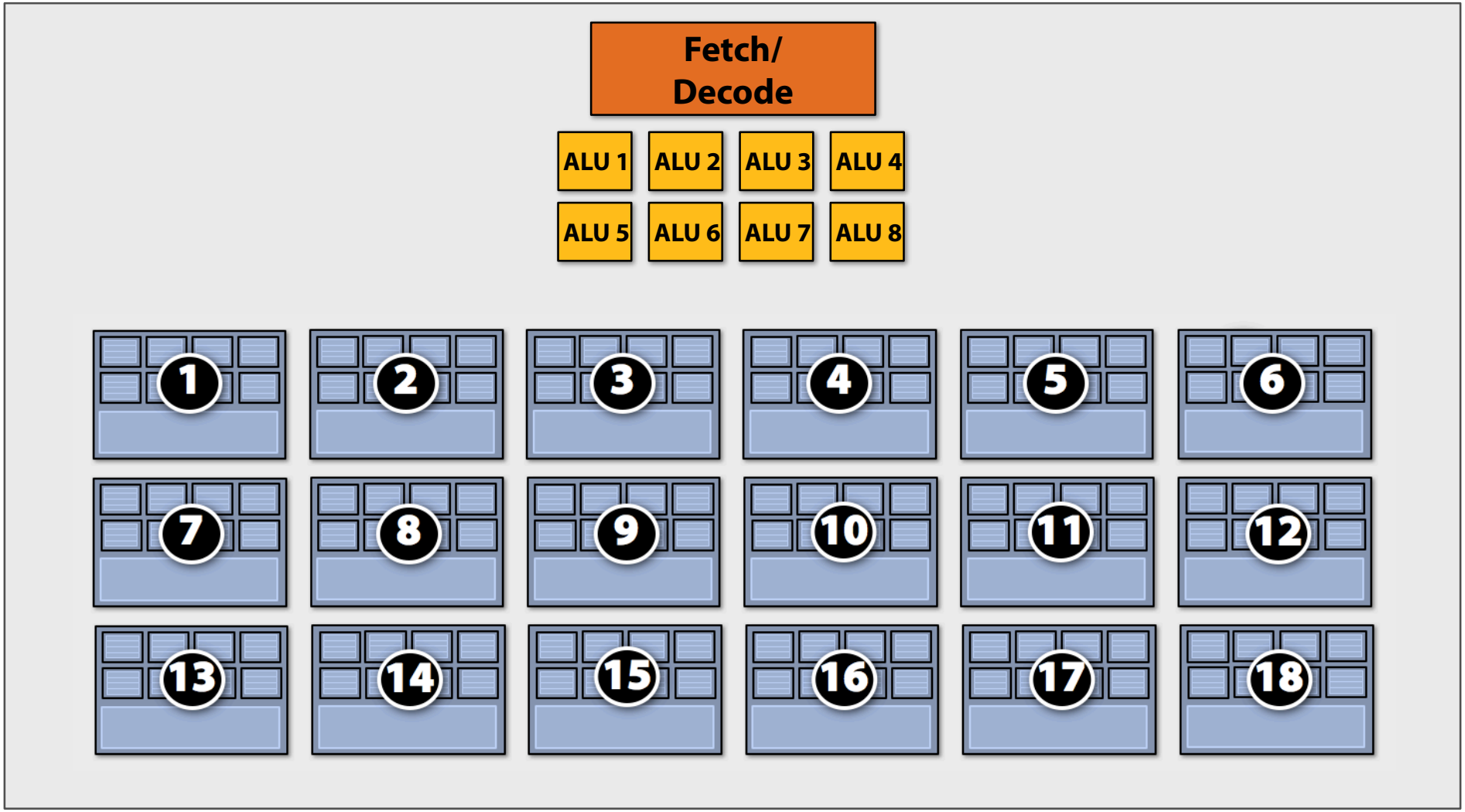

Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time of one group
to increase throughput of many groups

Storing contexts

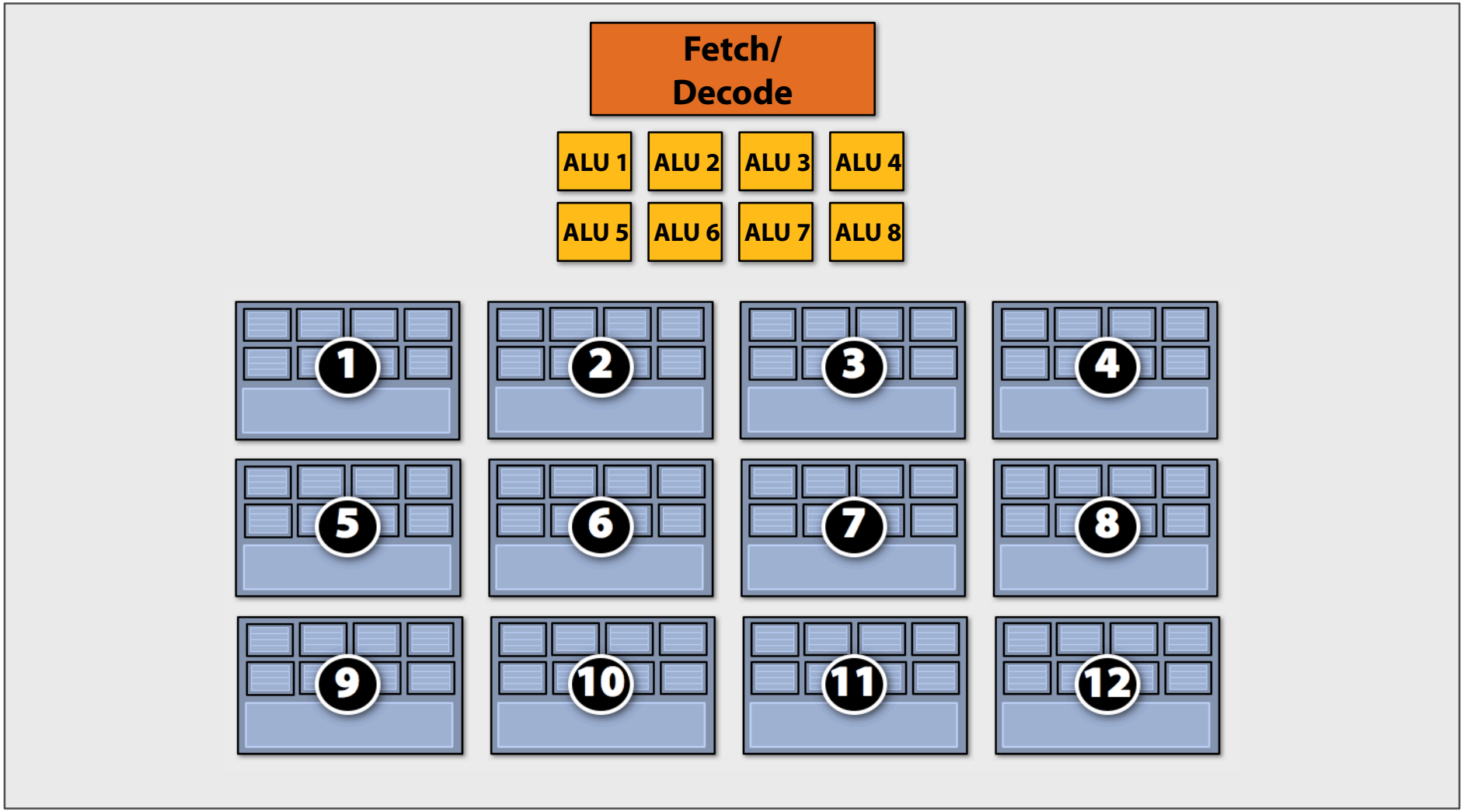


Eighteen small contexts

(maximal latency hiding)

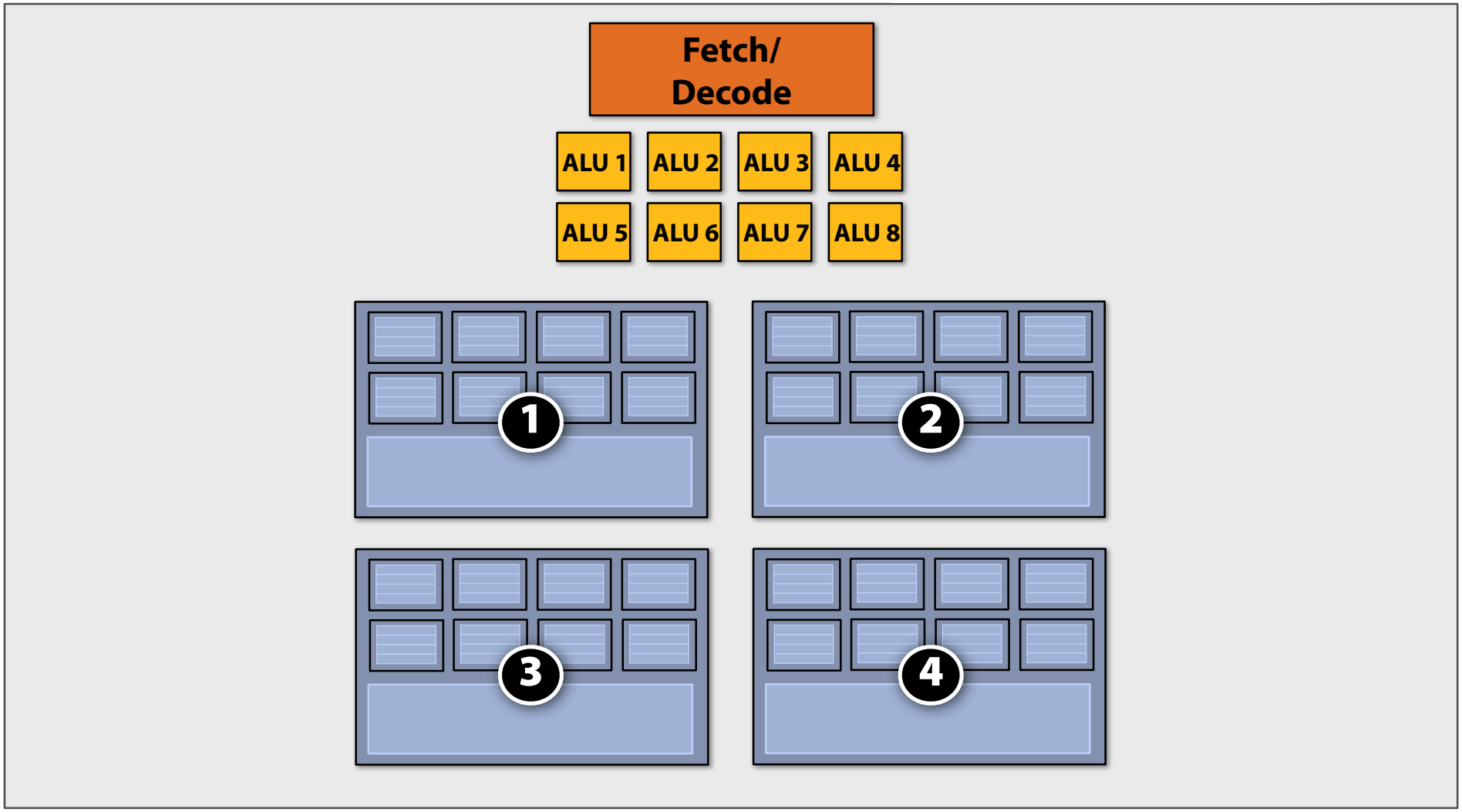


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

- Interleaving between contexts can be managed by hardware or software (or both!)
- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

What we learned: Idea 3

Interleave groups of threads