Computer Architecture

Lec 13: Data-Level Parallelism & Accelerators

This Lecture

- Data Level Parallelism
  - "medium-grained" parallelism between ILP and TLP
  - Still one flow of execution (unlike TLP)
  - Compiler/programmer must explicitly express it (unlike ILP)
- GPUs
  - Embrace data parallelism via "SIMT" execution model
  - Becoming more programmable all the time
- TPUs
  - Neural network accelerator
  - Fast matrix multiply machine

Data-Level Parallelism

How to Compute SAXPY Quickly?

- 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
```

- Instruction-level parallelism (ILP) - fine grained
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
  - Multicore
- Can we do some "medium grained" parallelism?

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)

Example Vector ISA Extensions (SIMD)

- Extend ISA with vector storage …
  - **Vector register**: fixed-size array of FP/int elements
  - **Vector length**: For example: 4, 8, 16, 64, …
- … and example operations for vector length of 4
  - Load vector: `ldf.v [X+r1]->v1`

    `ldf [X+r1+0]->v1`$_0$

    `ldf [X+r1+1]->v1`$_1$

    `ldf [X+r1+2]->v1`$_2$

    `ldf [X+r1+3]->v1`$_3$
  - Add two vectors: `addf.vv v1,v2->v3`

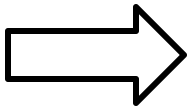
    `addf v1`$_i$`,v2`$_i$`->v3`$_i$ `(where i is 0,1,2,3)`
  - Add vector to scalar: `addf.vs v1,f2,v3`

    `addf v1`$_i$`,f2->v3`$_i$ `(where i is 0,1,2,3)`
- Today's vectors: short (128-512 bits), but fully parallel

Example Use of Vectors – 4-wide

```
ldf [X+r1]->f1
mulf f0,f1->f2
ldf [Y+r1]->f3
addf f2,f3->f4
stf f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions



```
ldf.v [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v [Y+r1]->v3
addf.vv v2,v3->v4
stf.v v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

- Operations
  - Load vector: **`ldf.v [X+r1]->v1`**
  - Multiply vector to scalar: **`mulf.vs v1,f2->v3`**
  - Add two vectors: **`addf.vv v1,v2->v3`**
  - Store vector: **`stf.v v1->[X+r1]`**
- Performance?
  - Best case: 4x speedup
  - But, vector instructions don't always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

Vector Datapath & Implementatoin

- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch (no extra N^2 checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid N^2 bypass problem)
  - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width
- Because they are just instructions…
  - …superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle

Vector Insn Sets for Different ISAs

- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - currently: AVX 512 offers 512b vectors
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - Scalable Vector Extensions (SVE): up to 2048b
    - implementation is narrower than this!
    - makes vector code portable

GPU

Part I: Graphics + GPU history

What GPUs were originally designed to do: 3D rendering

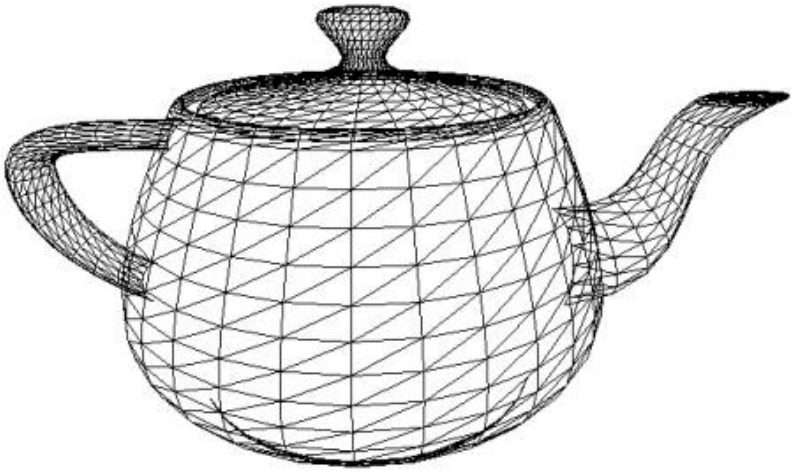


Image credit: Henrik Wann Jensen

Input: description of a scene:

3D surface geometry (e.g., triangle mesh)

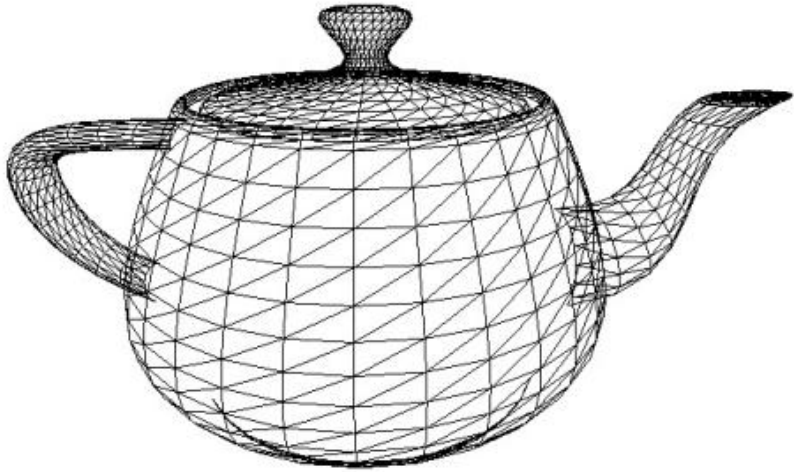
surface materials, lights, camera, etc.

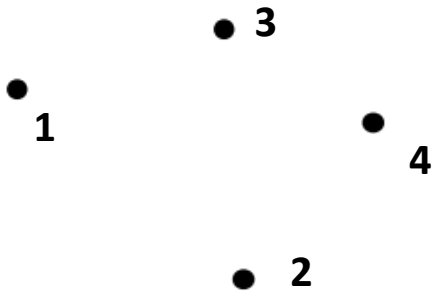
Output: image of the scene

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

Real-time graphics primitives (entities)

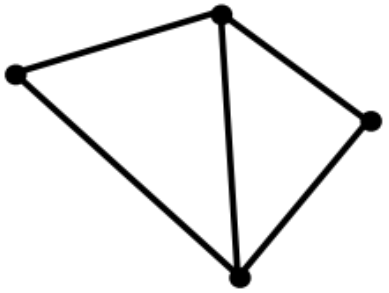
Represent surface as a 3D triangle mesh





Vertices

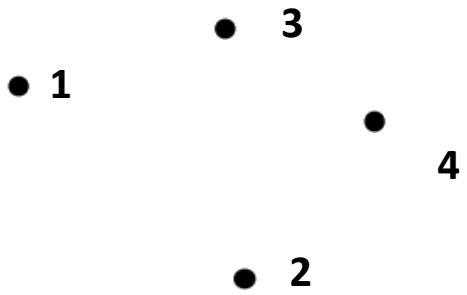
(points in space)



Primitives

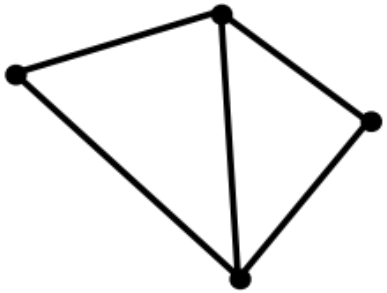
(e.g., triangles, points, lines)

Real-time graphics primitives (entities)



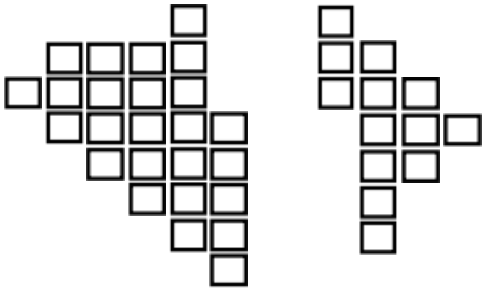
Vertices

(points in space)

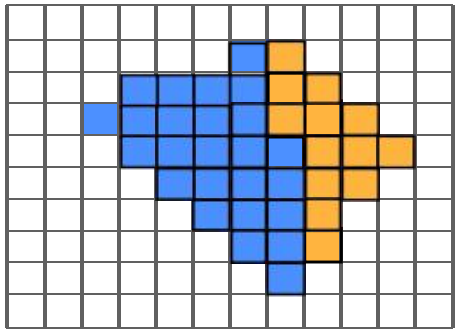


Primitives

(e.g., triangles, points, lines)

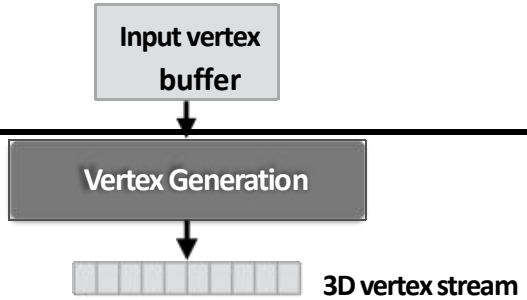


Fragments



Pixels (in an image)

Rendering a picture



Input: a list of vertices in 3D space (and their connectivity into primitives)

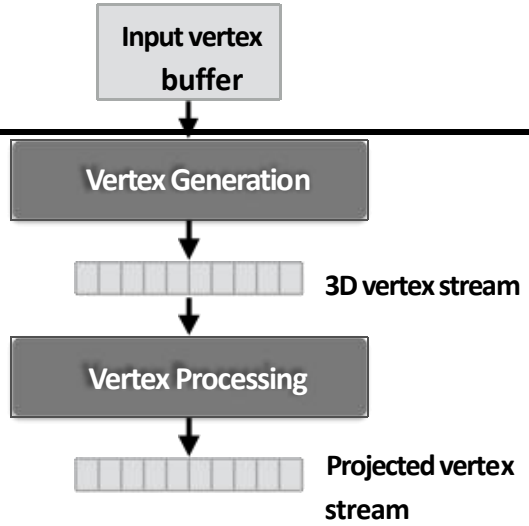
Example: every three vertices defines a triangle

```
list_of_positions =
{   v1x, v1y, v1x,
    v0x, v0y, v0z,
    v2x, v2y, v2z,
    v3x, v3y, v3x
};
```

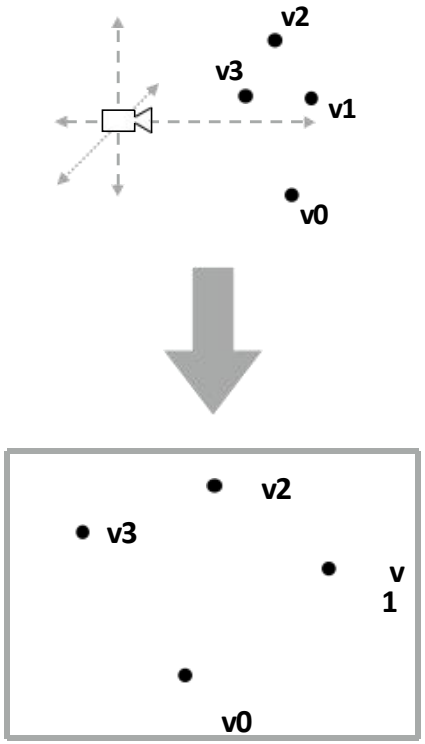
triangle 0 = {v0, v1, v2}

triangle 1 = {v1, v2, v3}

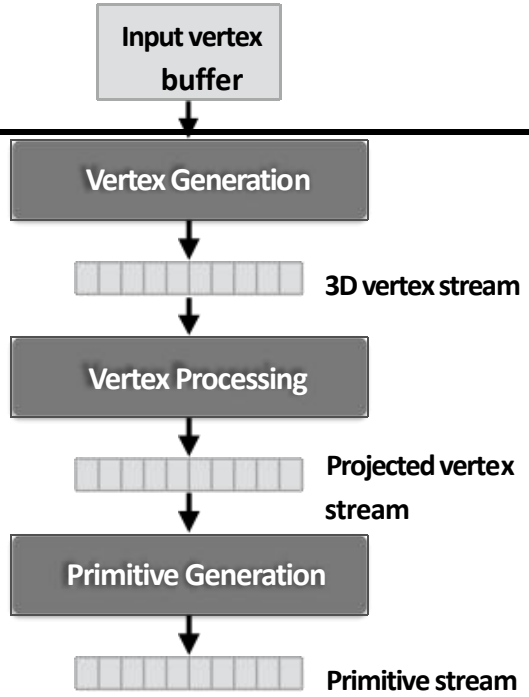
Rendering a picture



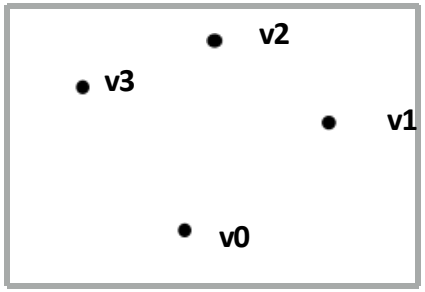
Step 1: given a scene camera position, compute where the vertices lie on screen

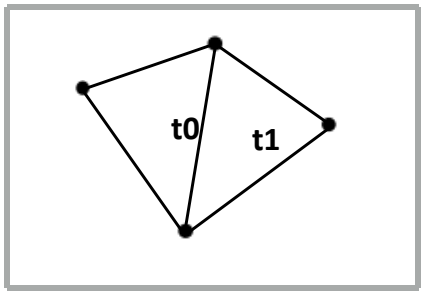


Rendering a picture



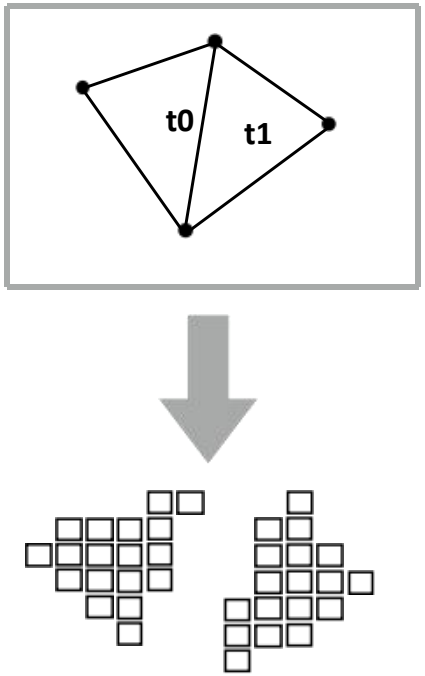
Step 2: group vertices into primitives

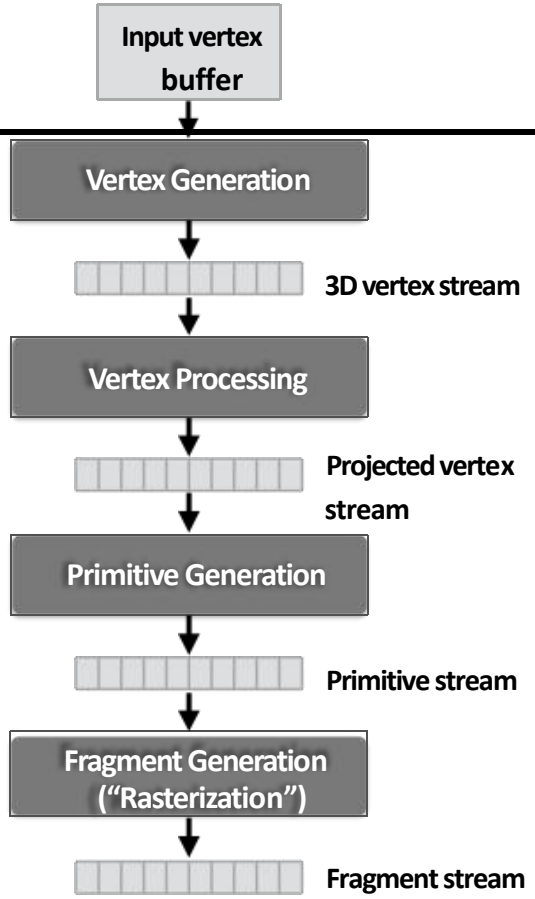




Rendering a picture

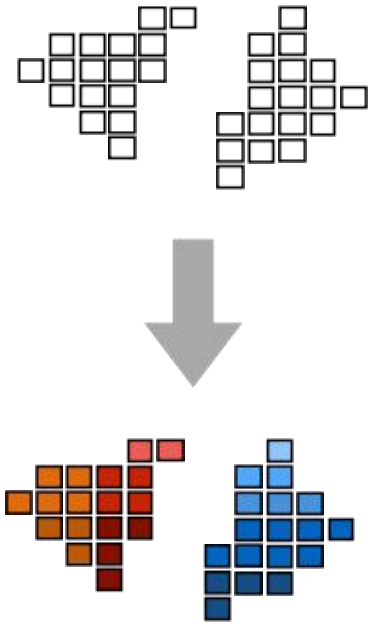
Step 3: generate one fragment for each pixel a primitive overlaps

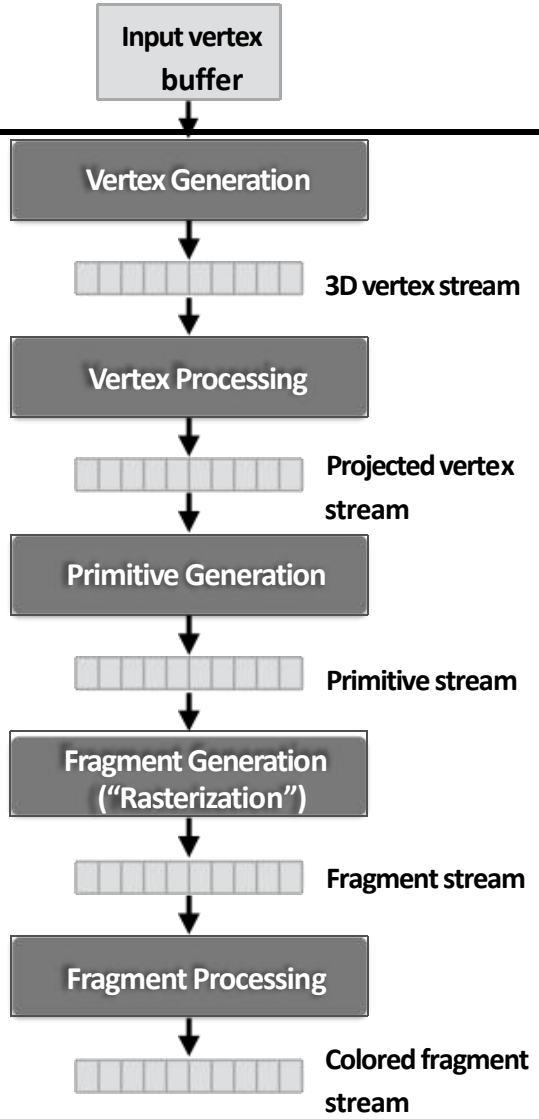




Rendering a picture

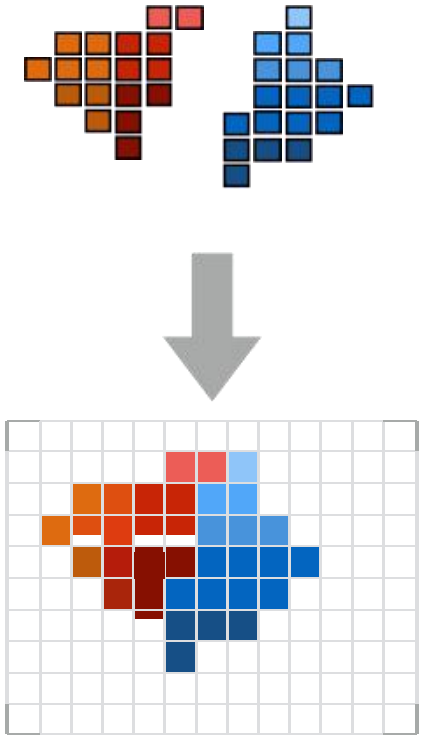
Step 4: compute color of primitive for each fragment (based on scene lighting and primitive material properties)

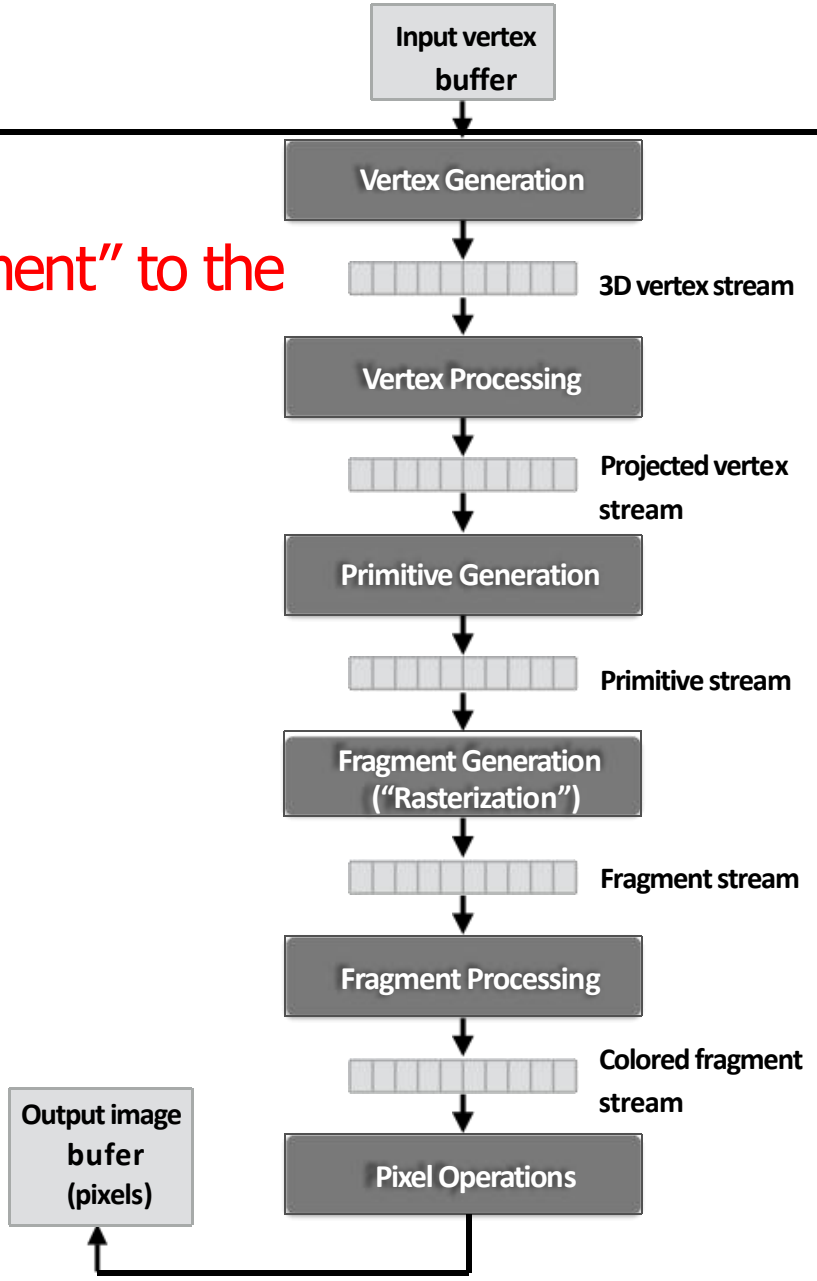




Rendering a picture

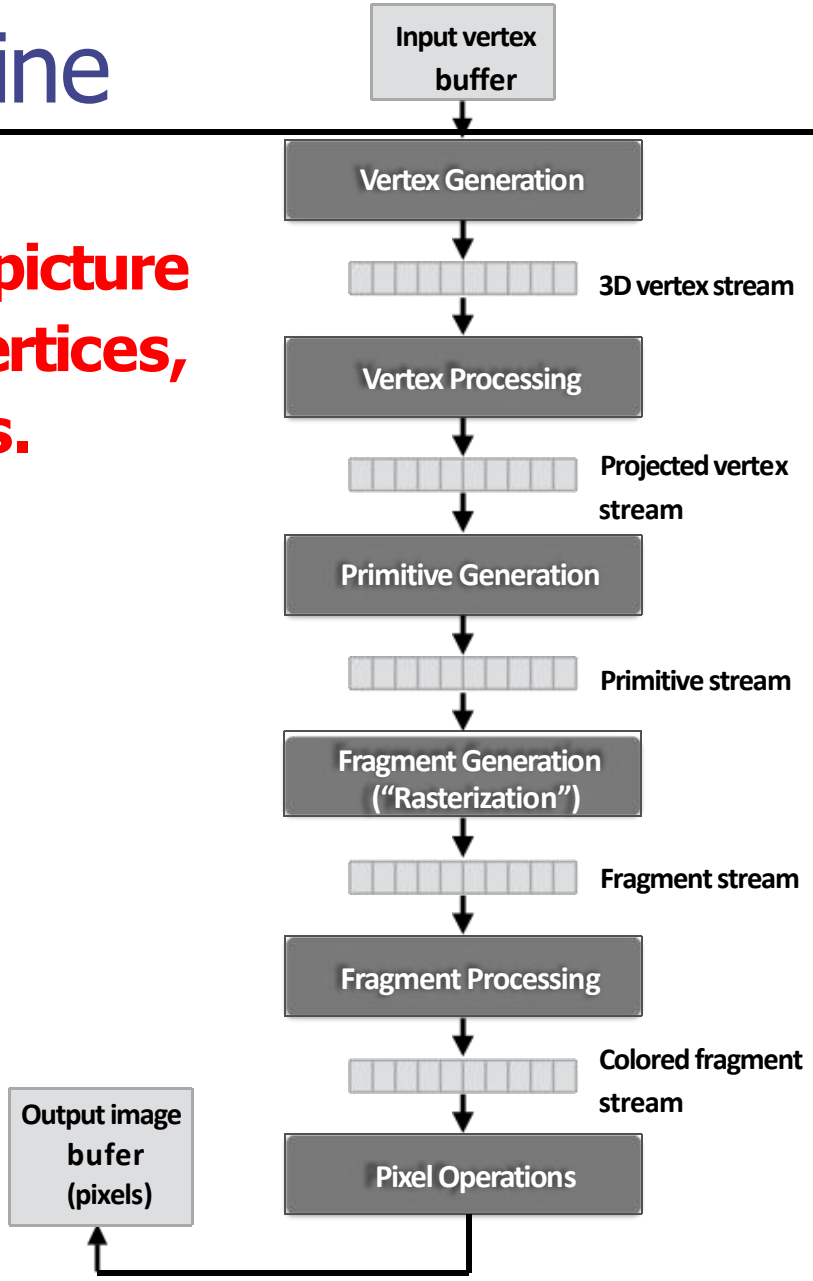
Step 5: put color of the "closest fragment" to the camera in the output image





Real-time graphics pipeline

Abstracts process of rendering a picture as a sequence of operations on vertices, primitives, fragments, and pixels.



Early graphics programming (OpenGL API)

- Graphics programming APIs provided programmer mechanisms to set parameters of scene lights and materials

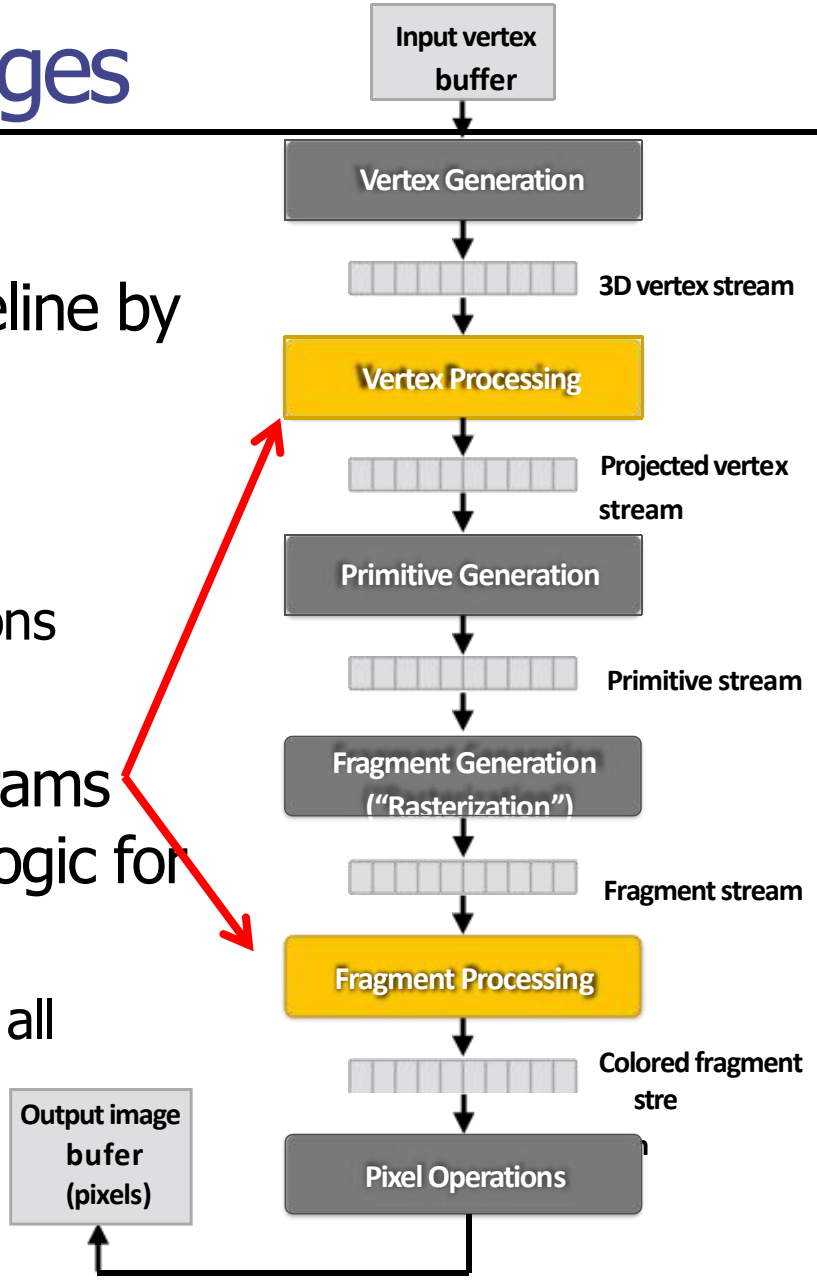
  - **`glLight(light_id, parameter_id, parameter_value)`**
    - Examples of light parameters: color, position, direction

  - **`glMaterial(face, parameter_id, parameter_value)`**
    - Examples of material parameters: color, shininess

Graphics shading languages

- Allow application to extend the functionality of the graphics pipeline by specifying materials and lights programmatically!
  - Support diversity in materials
  - Support diversity in lighting conditions

- Programmer provides mini-programs ("shaders") that define pipeline logic for certain stages
  - Pipeline maps shader function onto all elements of input stream



Example fragment shader program

Run once per fragment (per pixel covered by a triangle)

OpenGL shading language (GLSL) shader program:
defines behavior of fragment processing stage

```
uniform sampler2D myTexture;
uniform float3 lightDir;
varying vec3 norm;
varying vec2 uv;

void myFragmentShader( )
{
  vec3 kd = texture2D(myTexture, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return vec4(kd, 1.0);
}
```

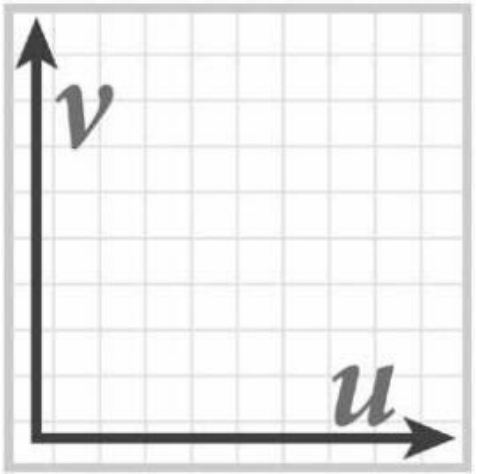
read-only global variables

per-fragment inputs

"fragment shader"
(a.k.a kernel function mapped onto input fragment stream)

per-fragment output: RGBA surface color at pixel

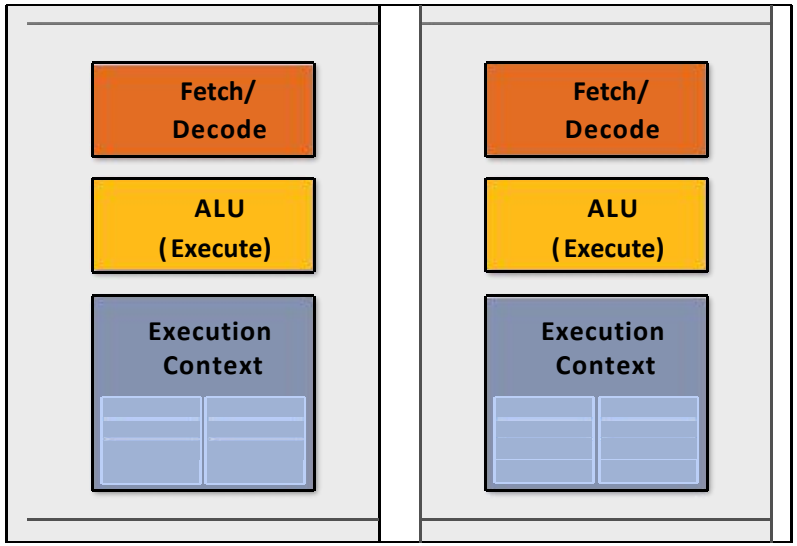
myTextureis a texture map



Part II: GPU compute mode

Review: how to run code on a CPU

- Lets say a user wants to run a program on a multi-core CPU…
  - OS loads program text into memory
  - OS selects CPU execution context
  - OS interrupts processor, prepares execution context (sets contents of registers, program counter, etc. to prepare execution context)
  - Go!
  - Processor begins executing instructions from within the environment maintained in the execution context

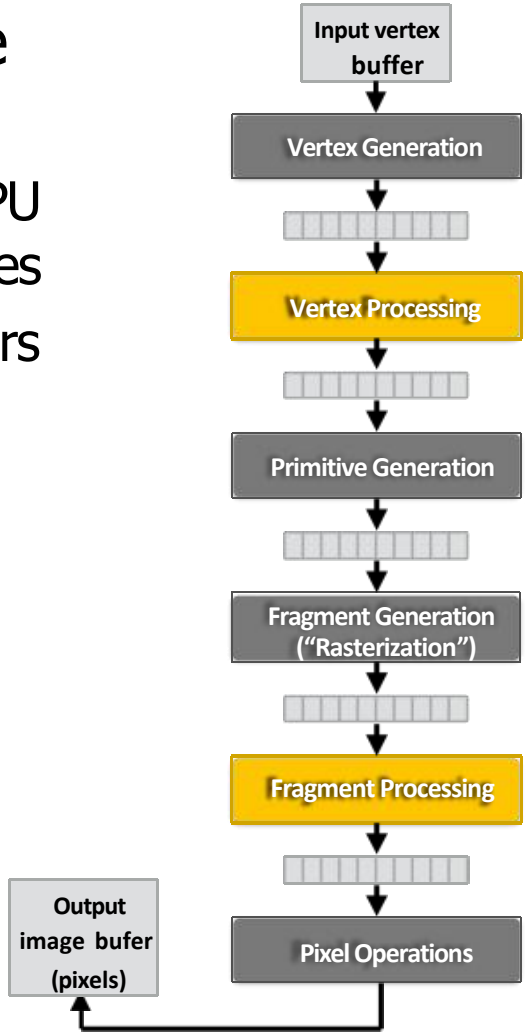


Multi-core CPU

How to run code on a GPU (prior to 2007)

- Lets say a user wants to draw a picture using a GPU…
  - Application (via graphics driver) provides GPU vertex and fragment shader program binaries
  - Application sets graphics pipeline parameters (e.g., output image size)
  - Application provides hardware a buffer of vertices
  - Go! launch(myKernel, N)

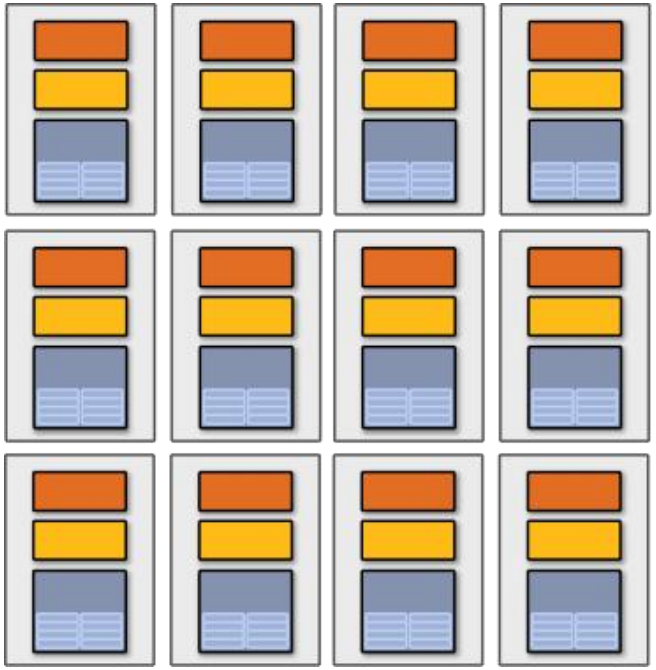
This was the only interface to GPU hardware. GPU hardware could only execute graphics pipeline computations.



NVIDIA Tesla architecture (2007)

First alternative, non-graphics-speciic ("compute mode") interface to GPU hardware

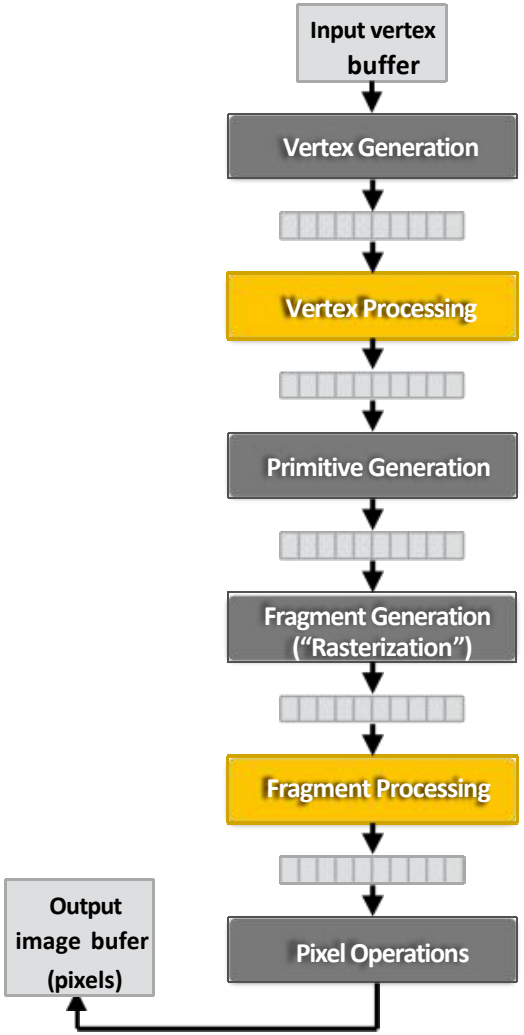
- Lets say a user wants to run a non-graphics program on the GPU's programmable cores…
  - Application can allocate buffers in GPU memory and copy data to/from buffers
  - Application (via graphics driver) provides GPU a single kernel program binary
  - Application tells GPU to run the kernel in an SIMD fashion ("run N instances")
  - Go! (drawPrimitives(vertex_bufer))



Aside: interestingly, this is a far simpler operation than drawPrimitives()

CUDA programming language

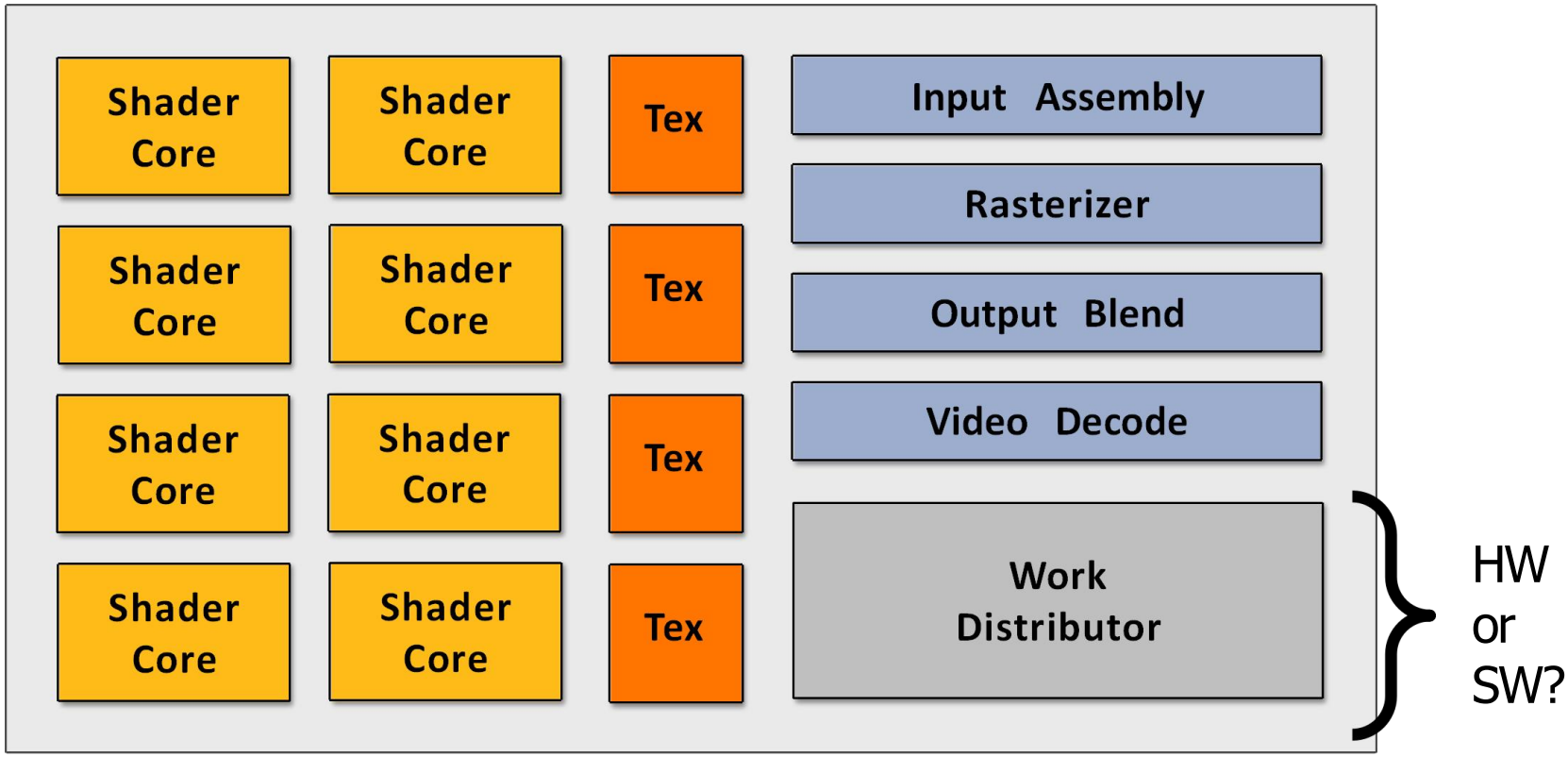
- Introduced in 2007 with NVIDIA Tesla architecture
- "C-like" language to express programs that run on GPUs using the compute-mode hardware interface
- Relatively low-level: CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs (design goal: maintain low abstraction distance)
- Note: OpenCL is an open standards version of CUDA
  - CUDA only runs on NVIDIA GPUs
  - OpenCL runs on CPUs and GPUs from many vendors
  - Almost everything I say about CUDA also holds for OpenCL
  - CUDA is better documented



Part III: GPU architecture

What's in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



A diffuse relectance shader

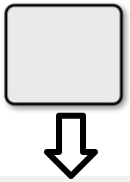
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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:

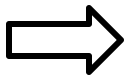
Fragments are processed independently,
but there is no explicit parallel programming

Compile shader

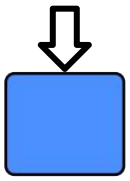
1 unshaded fragment input record



```
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);
}
```

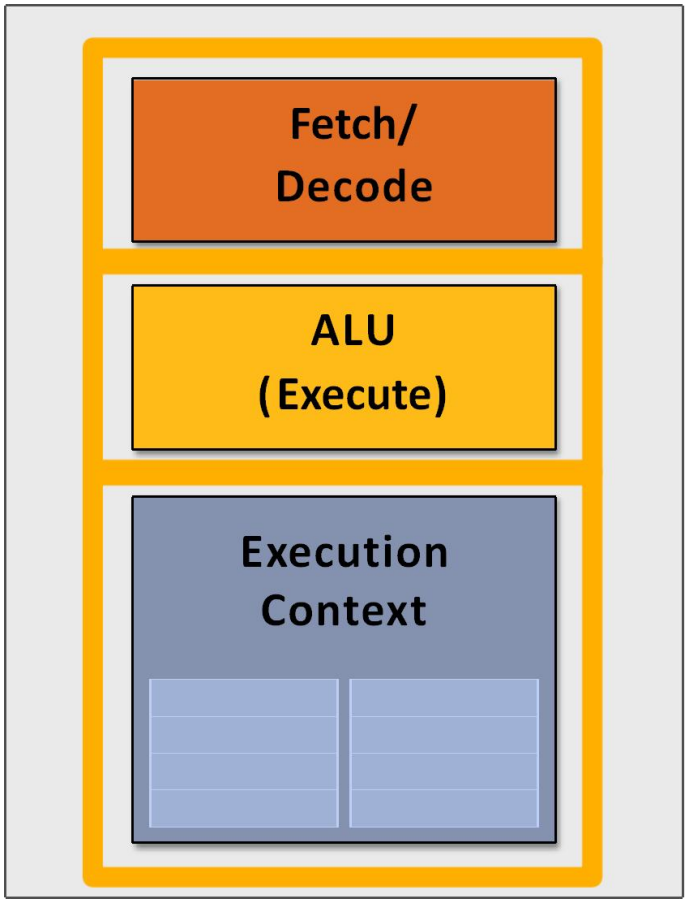


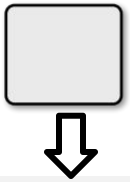
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```



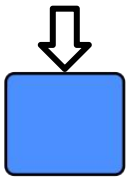
1 shaded fragment output record

Execute shader

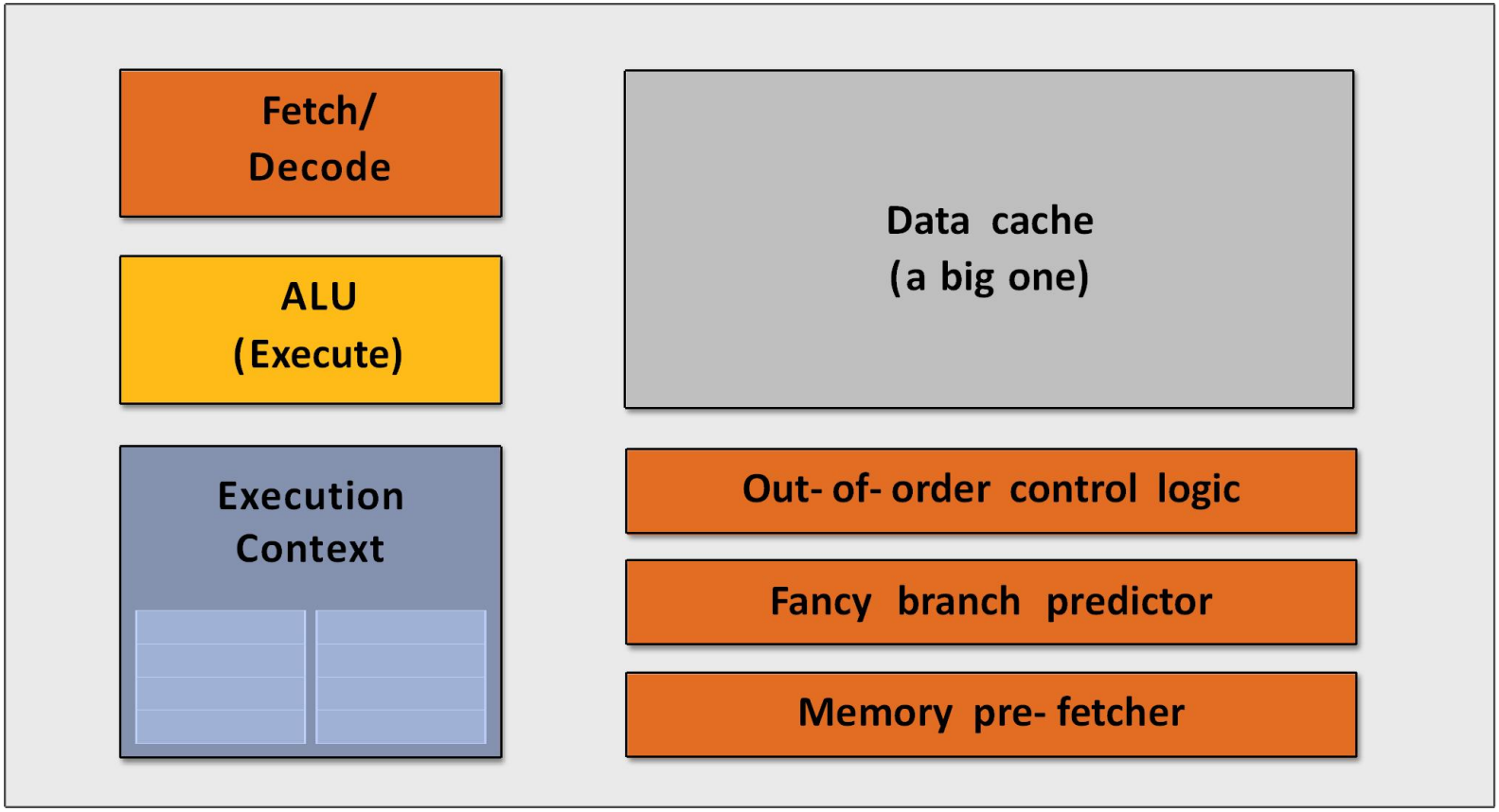




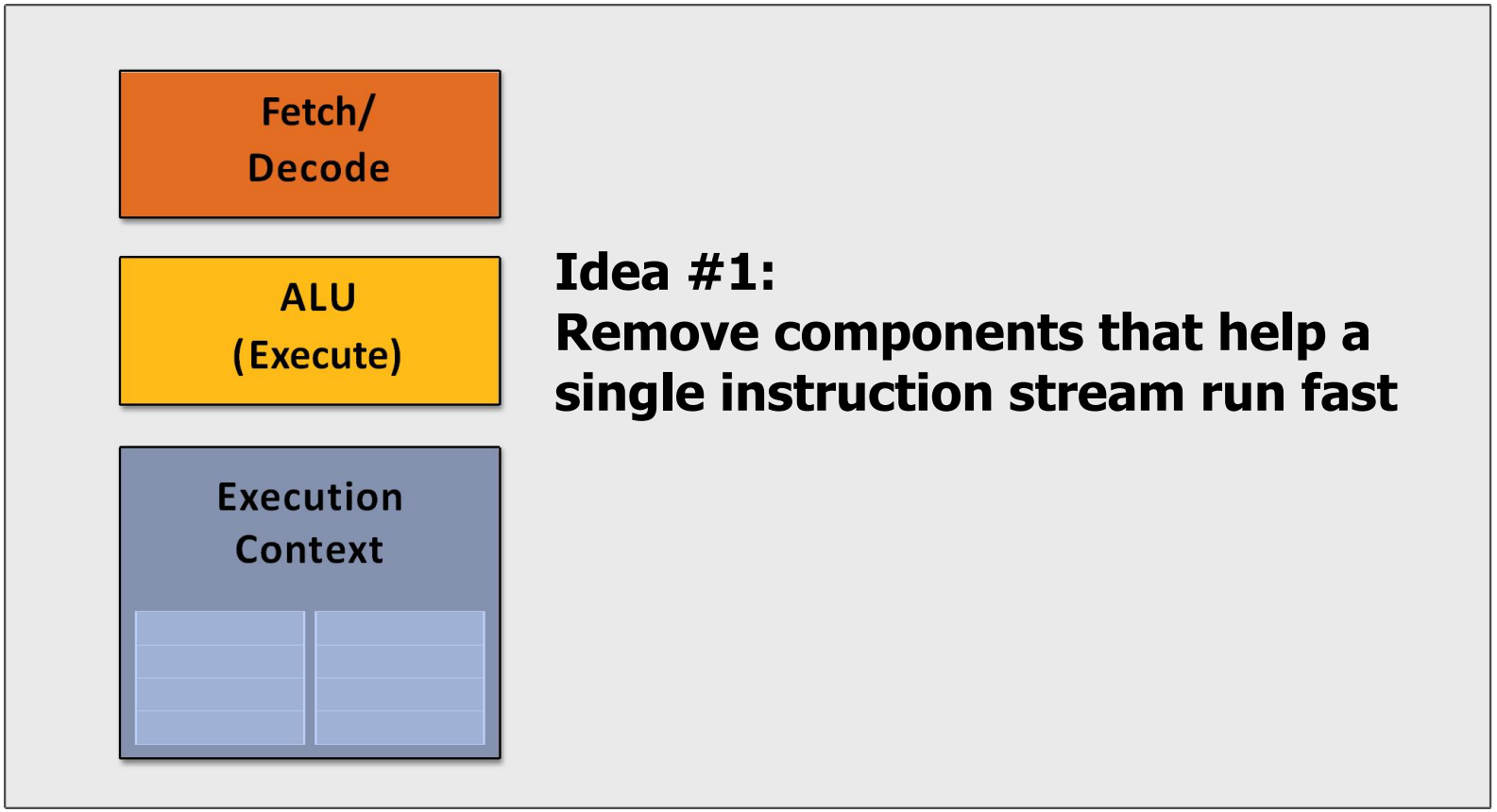
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```



"CPU-style" cores

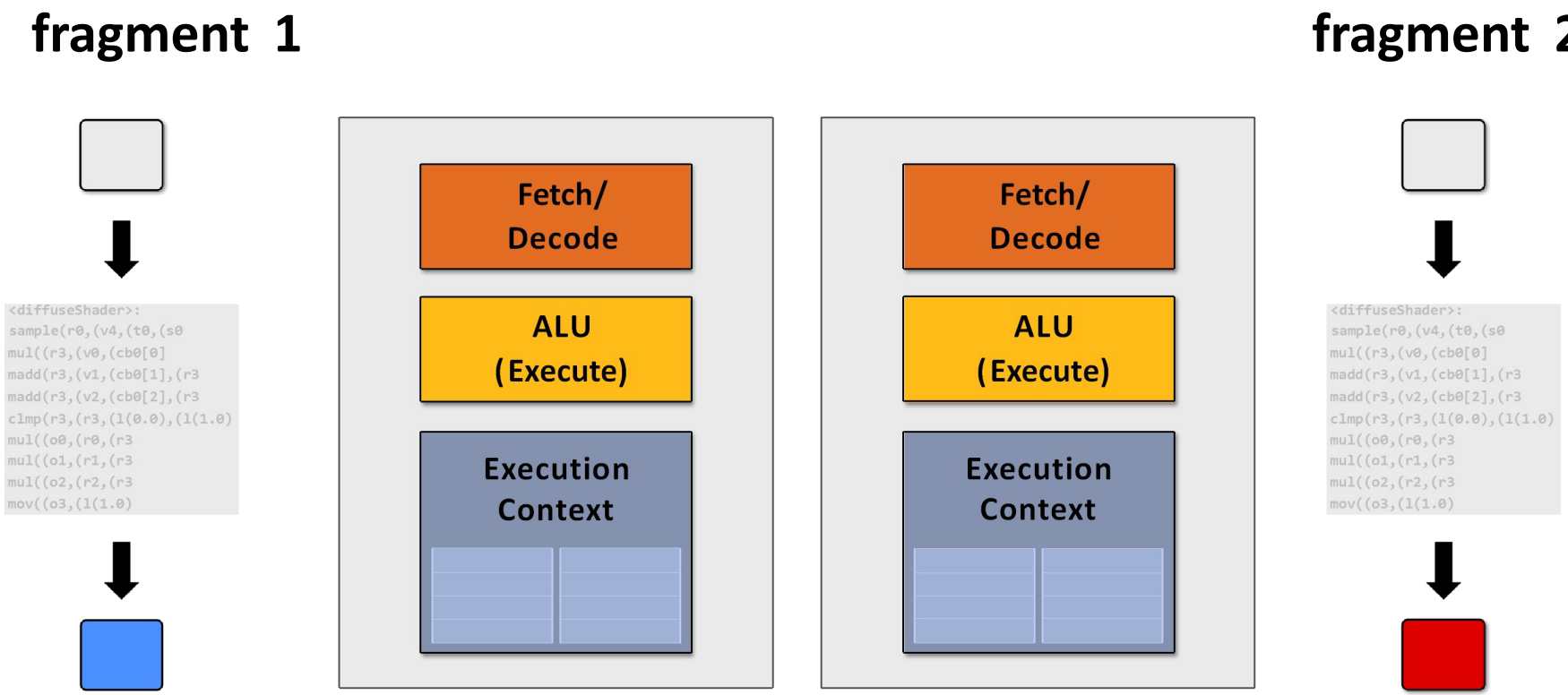


Slimming down

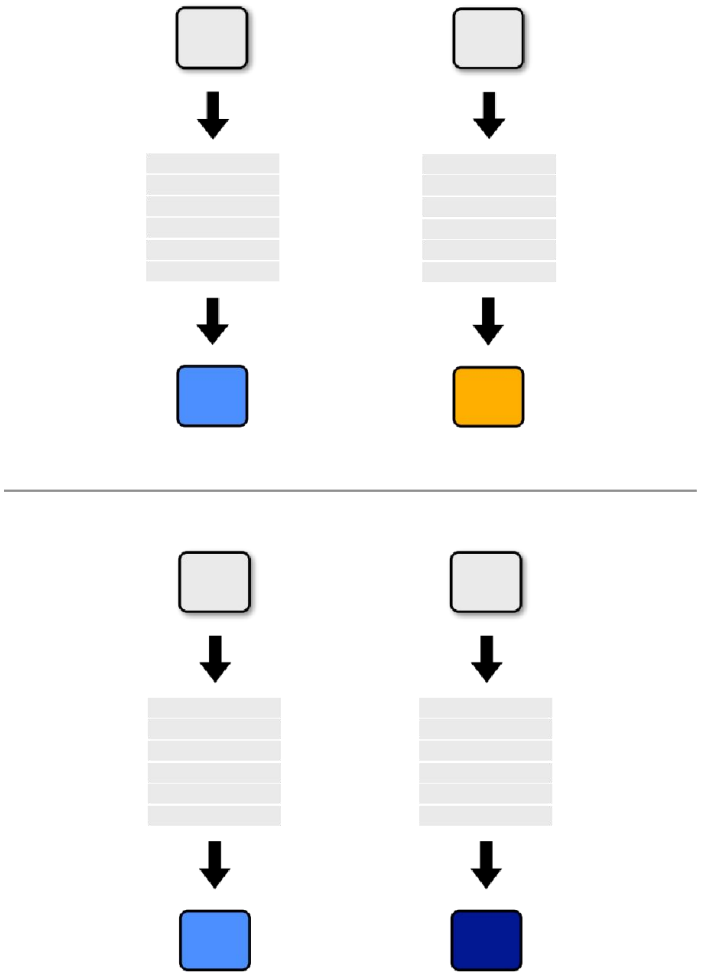


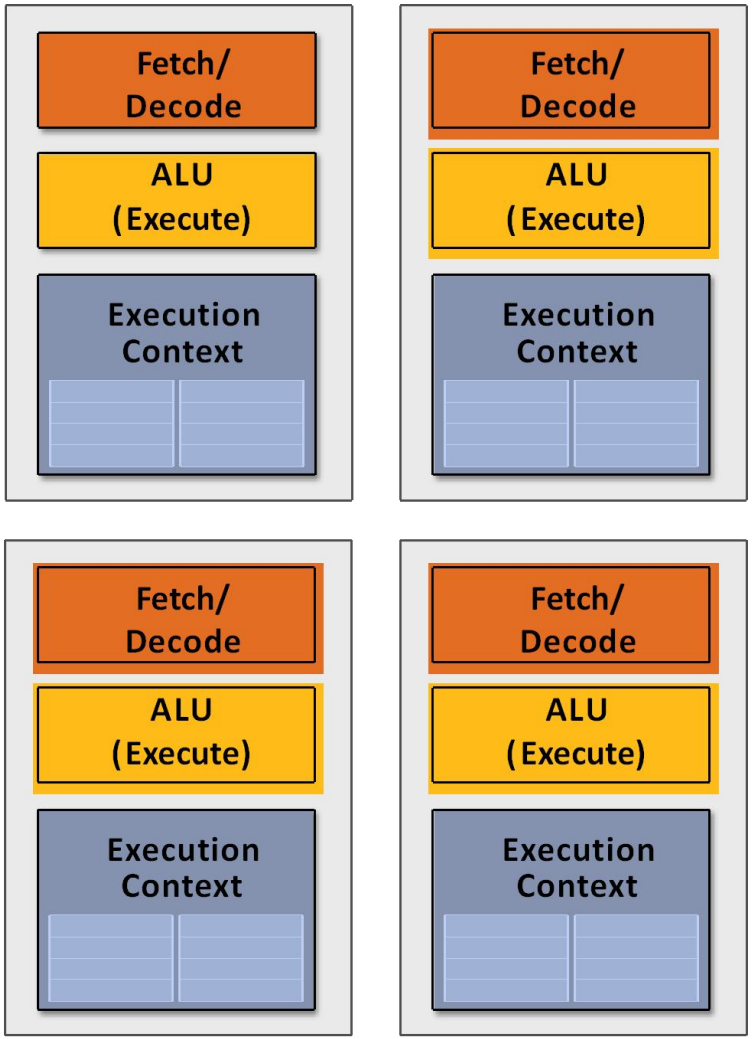
Idea #1:
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

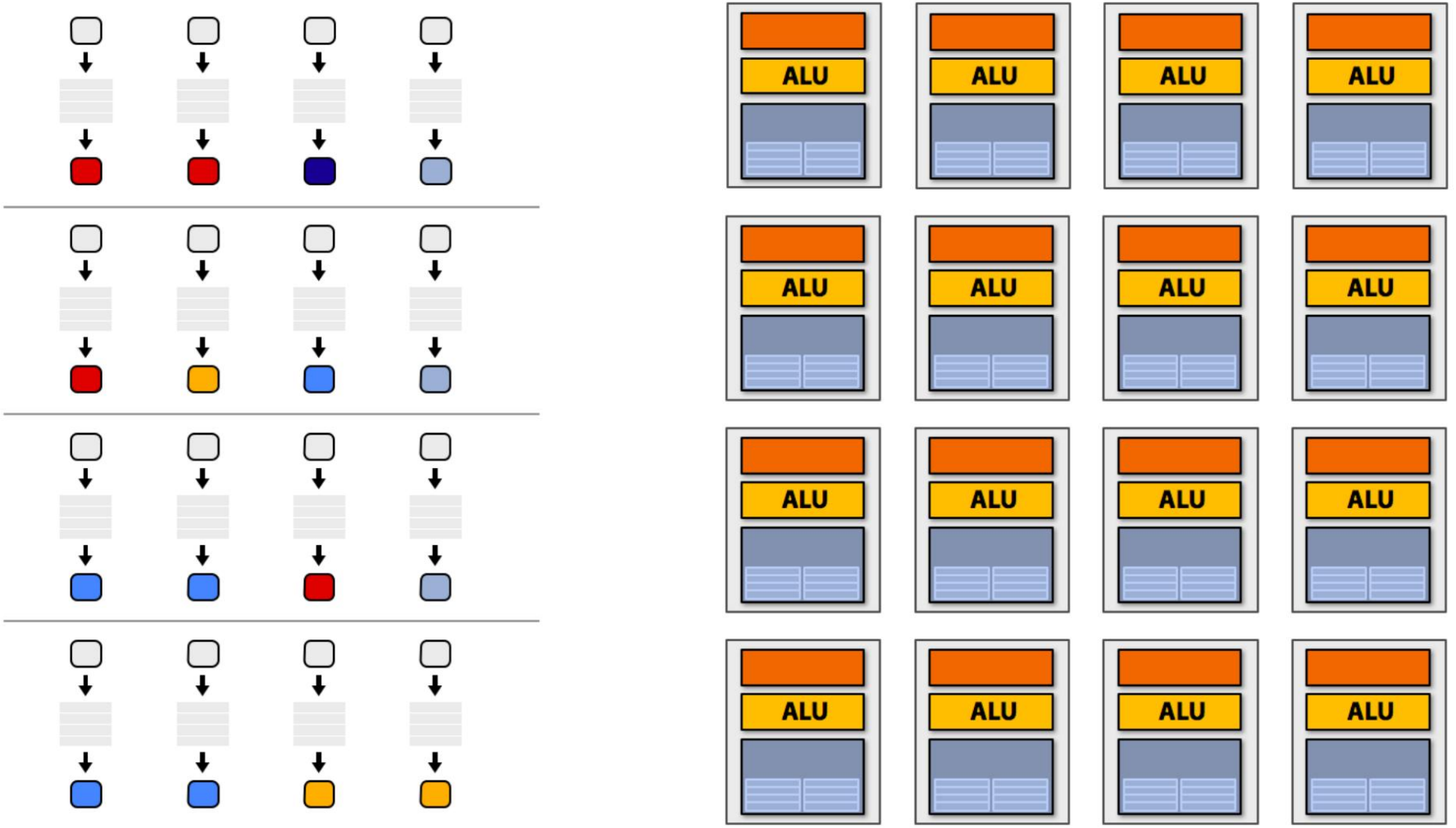


Four cores (four fragments in parallel)



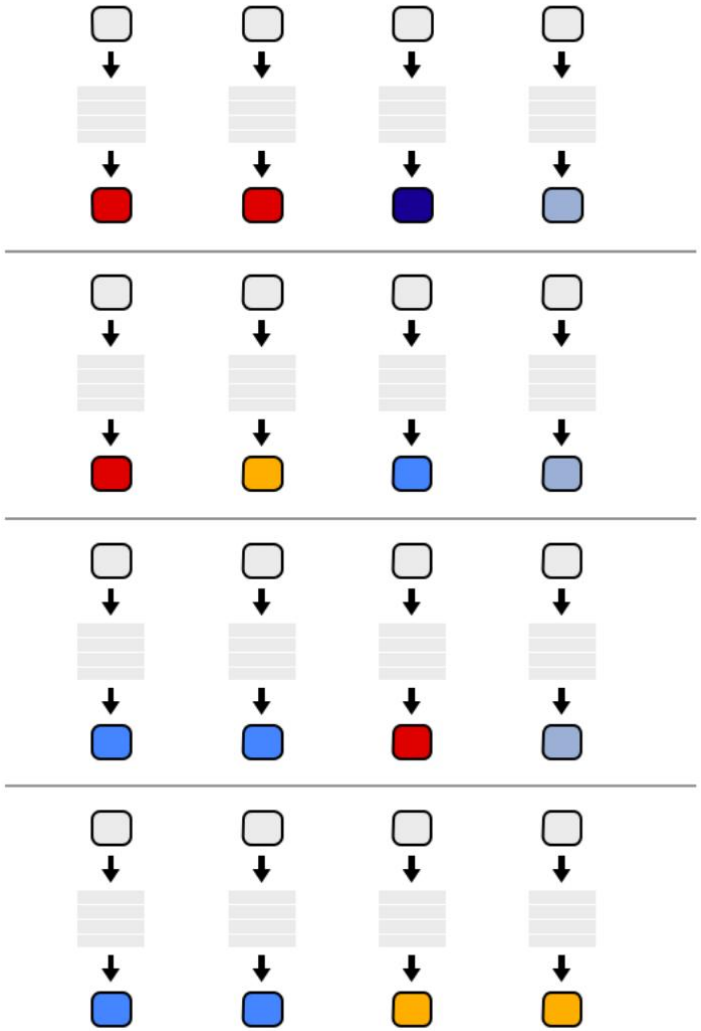


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

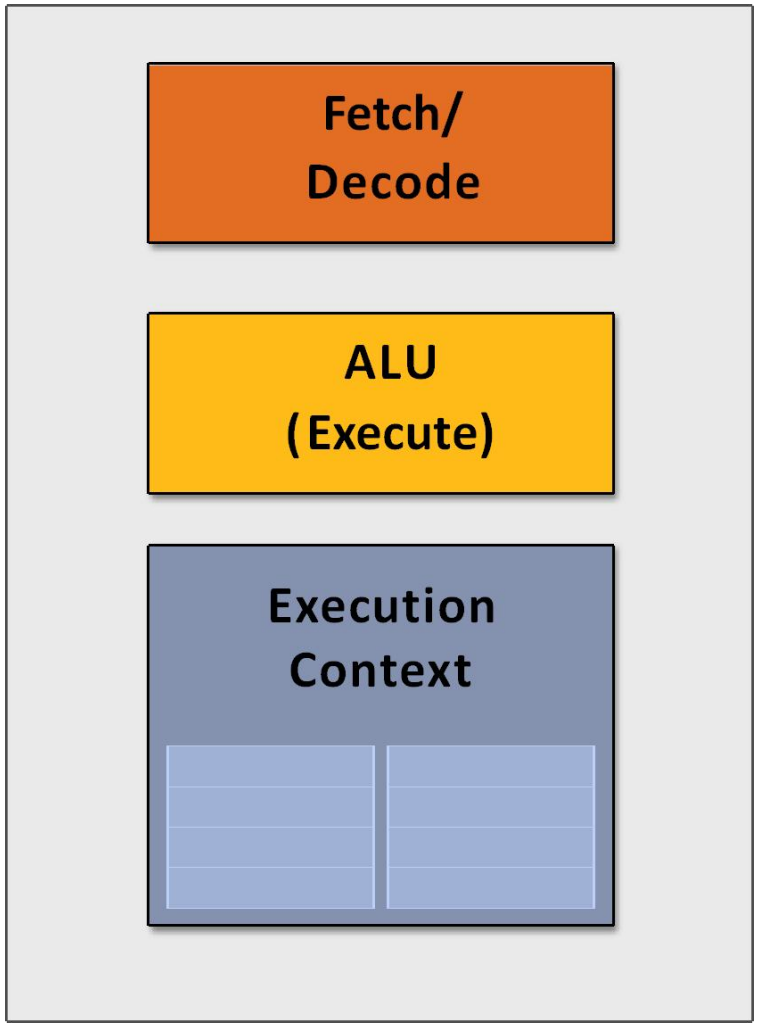
Instruction stream sharing



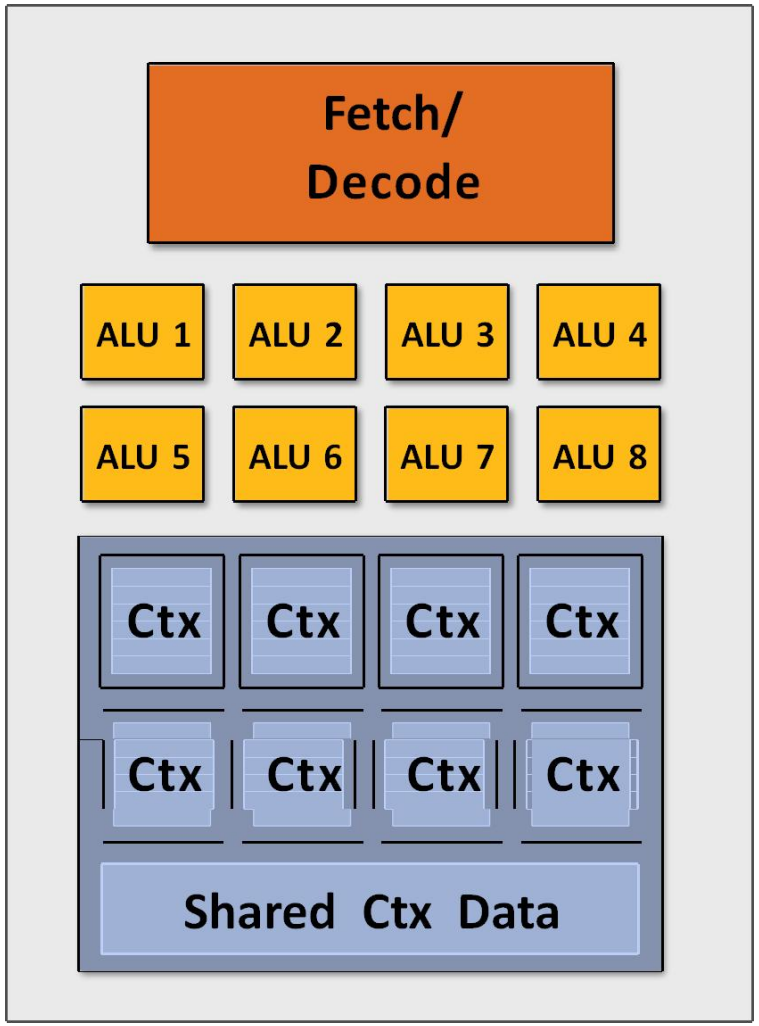
But ... many fragments should be able to share an instruction stream!

```
<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



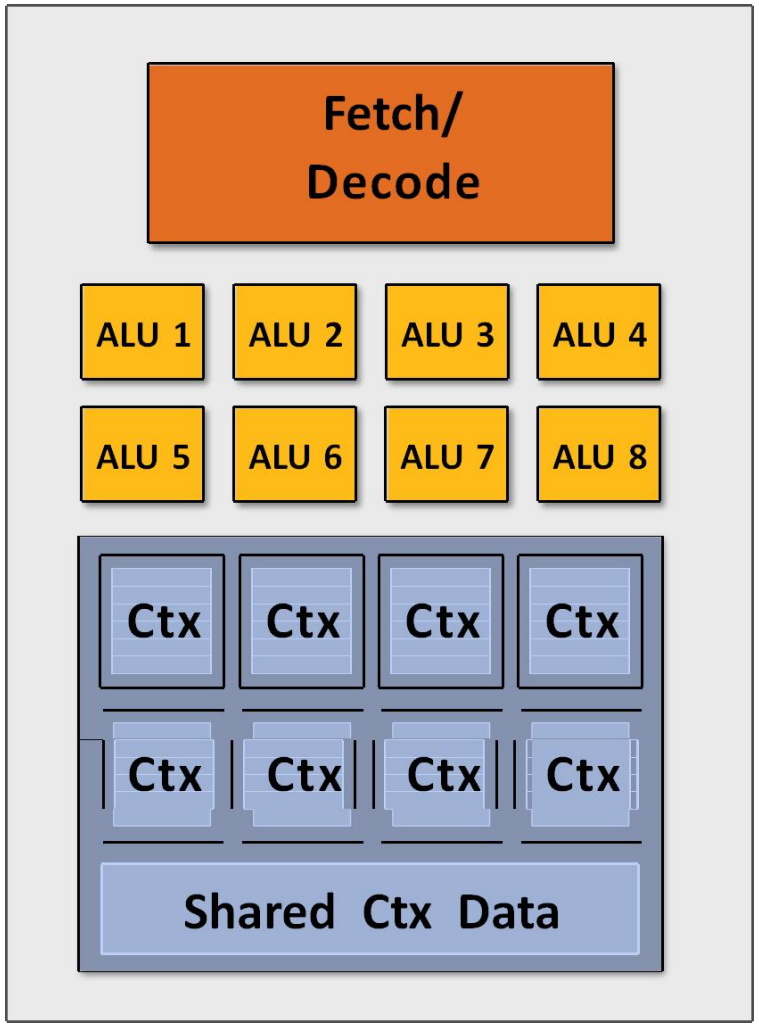
Add ALUs



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

SIMD processing

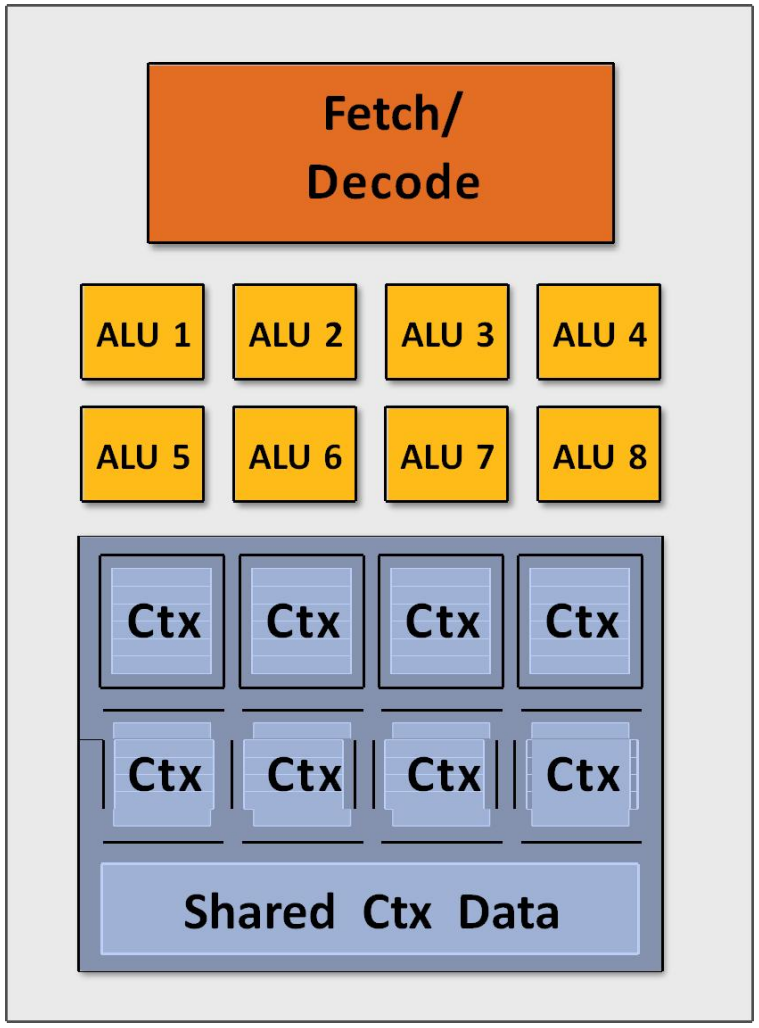
Modifying the shader




```
<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

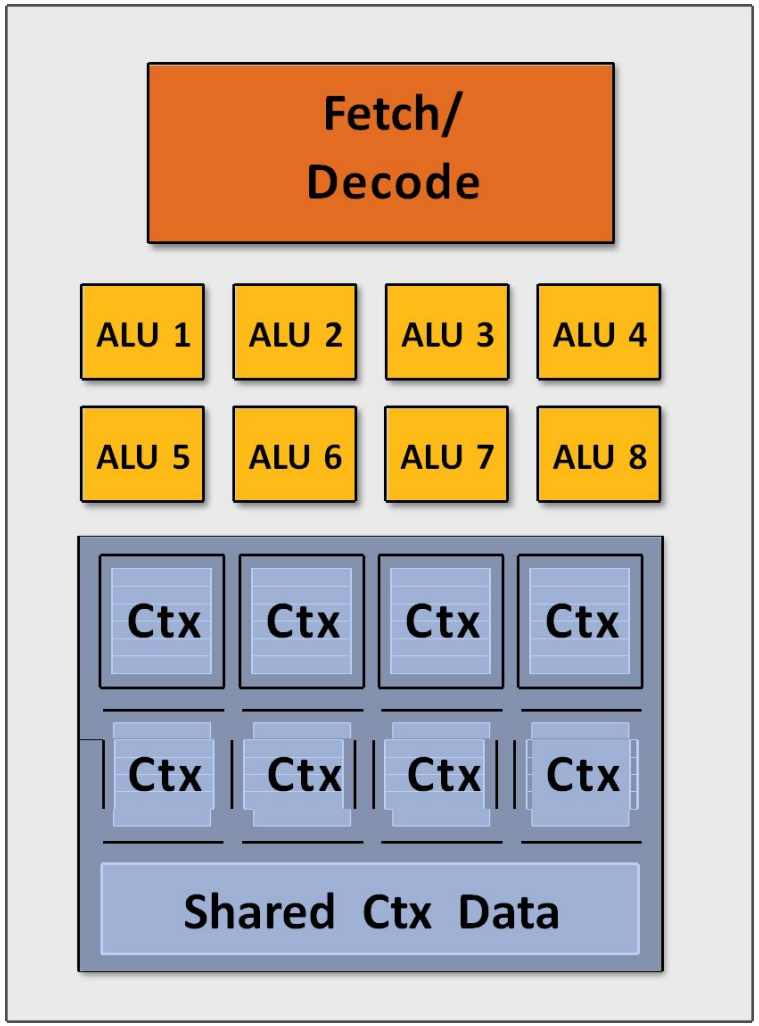
Modifying the shader

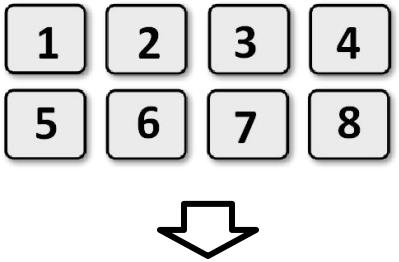



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2,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

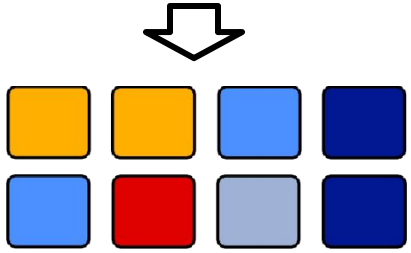
New compiled shader:
Processes eight fragments using vector ops on vector registers

Modifying the shader

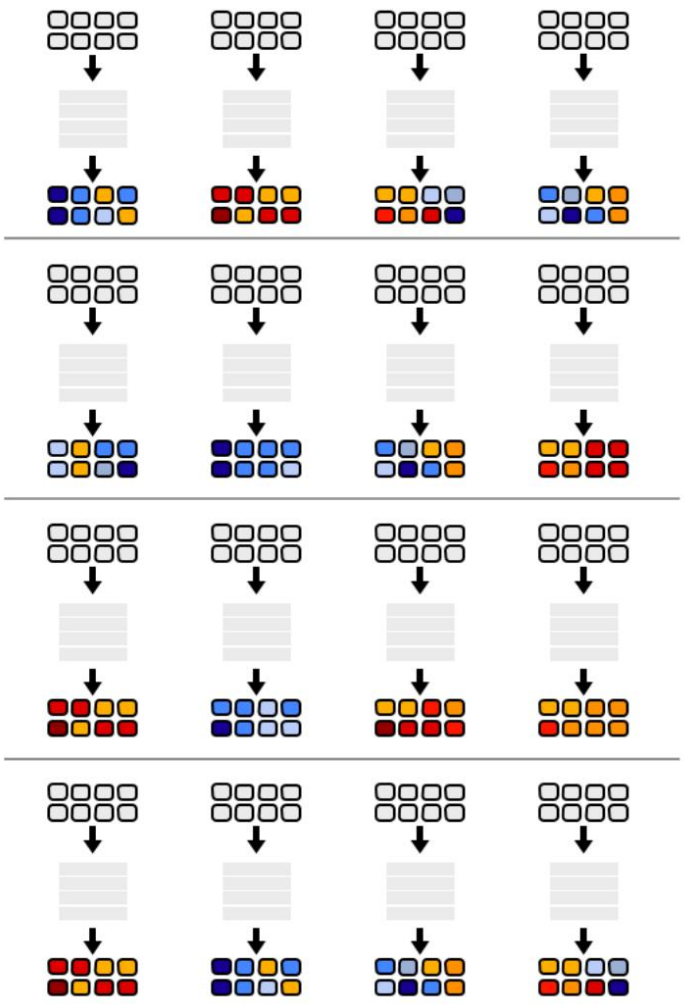


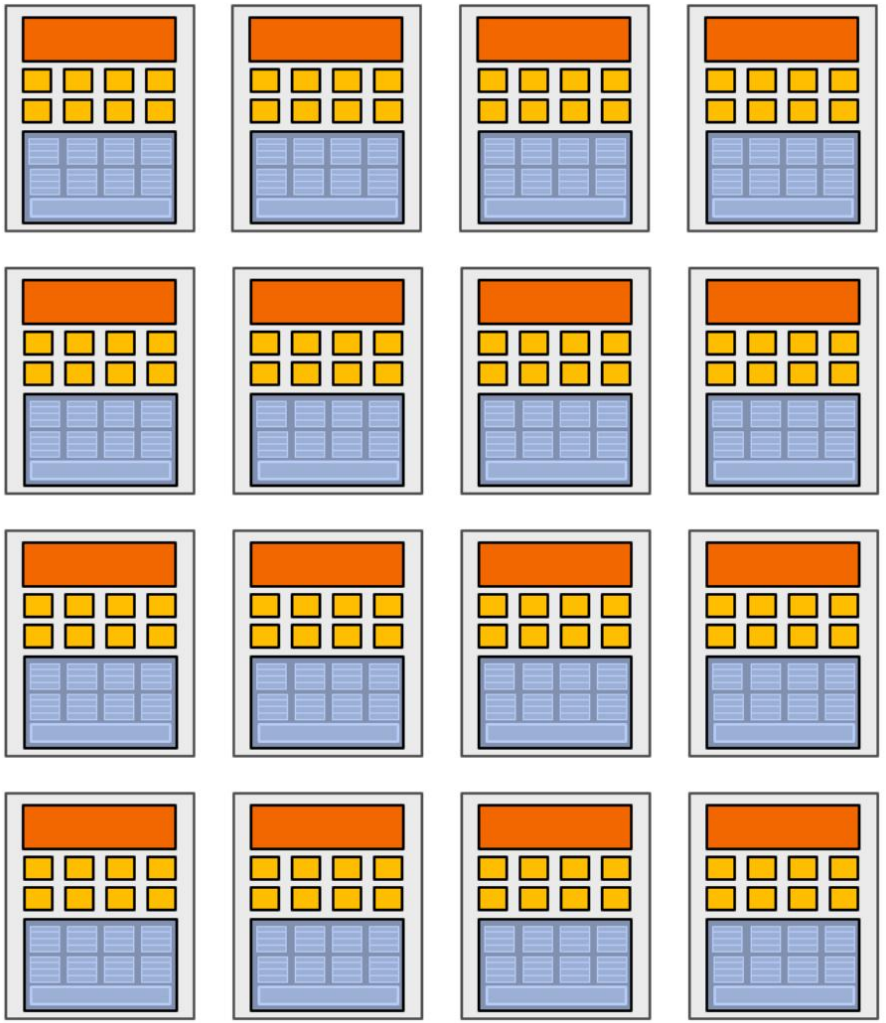



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2,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



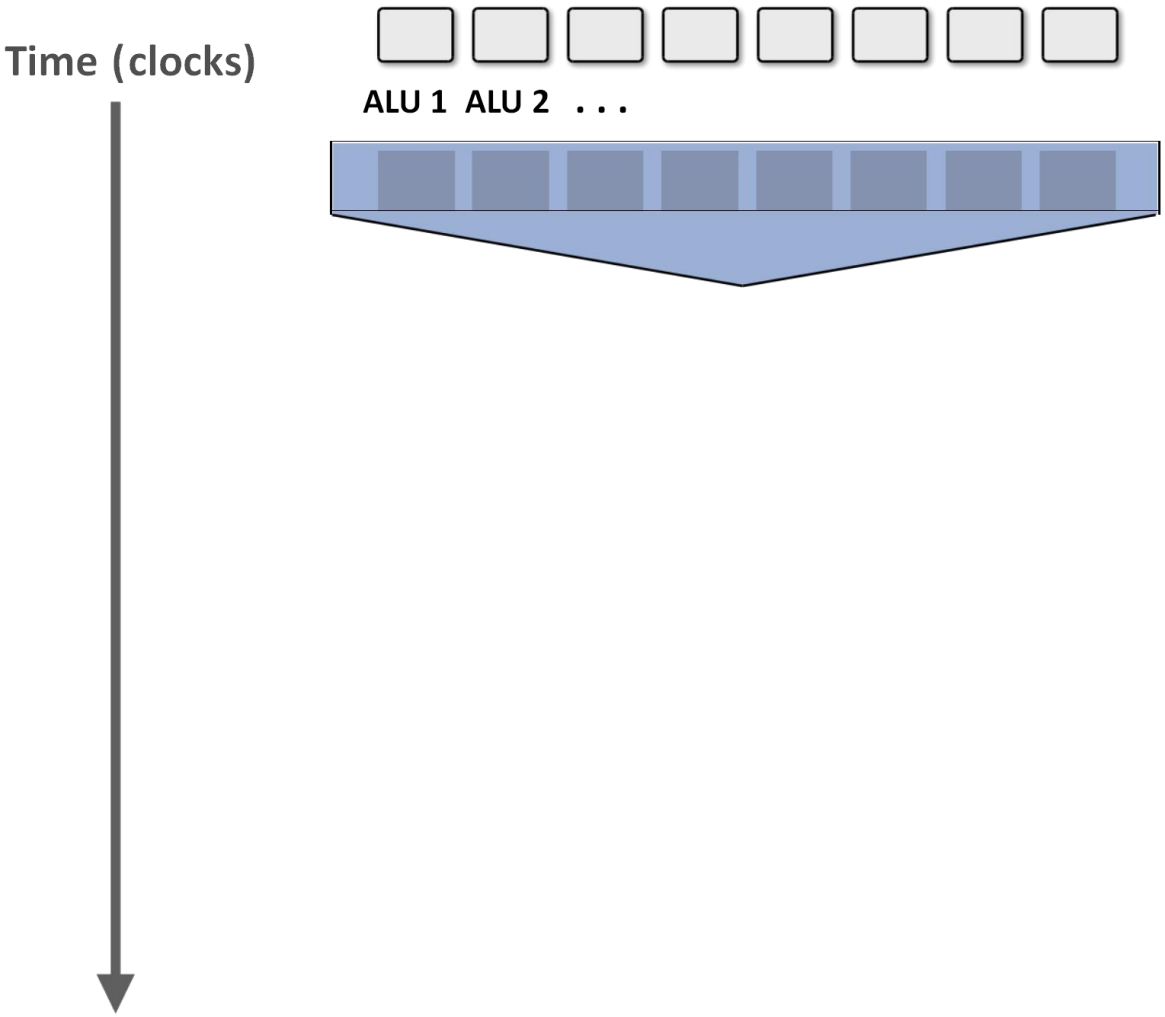
128 fragments in parallel





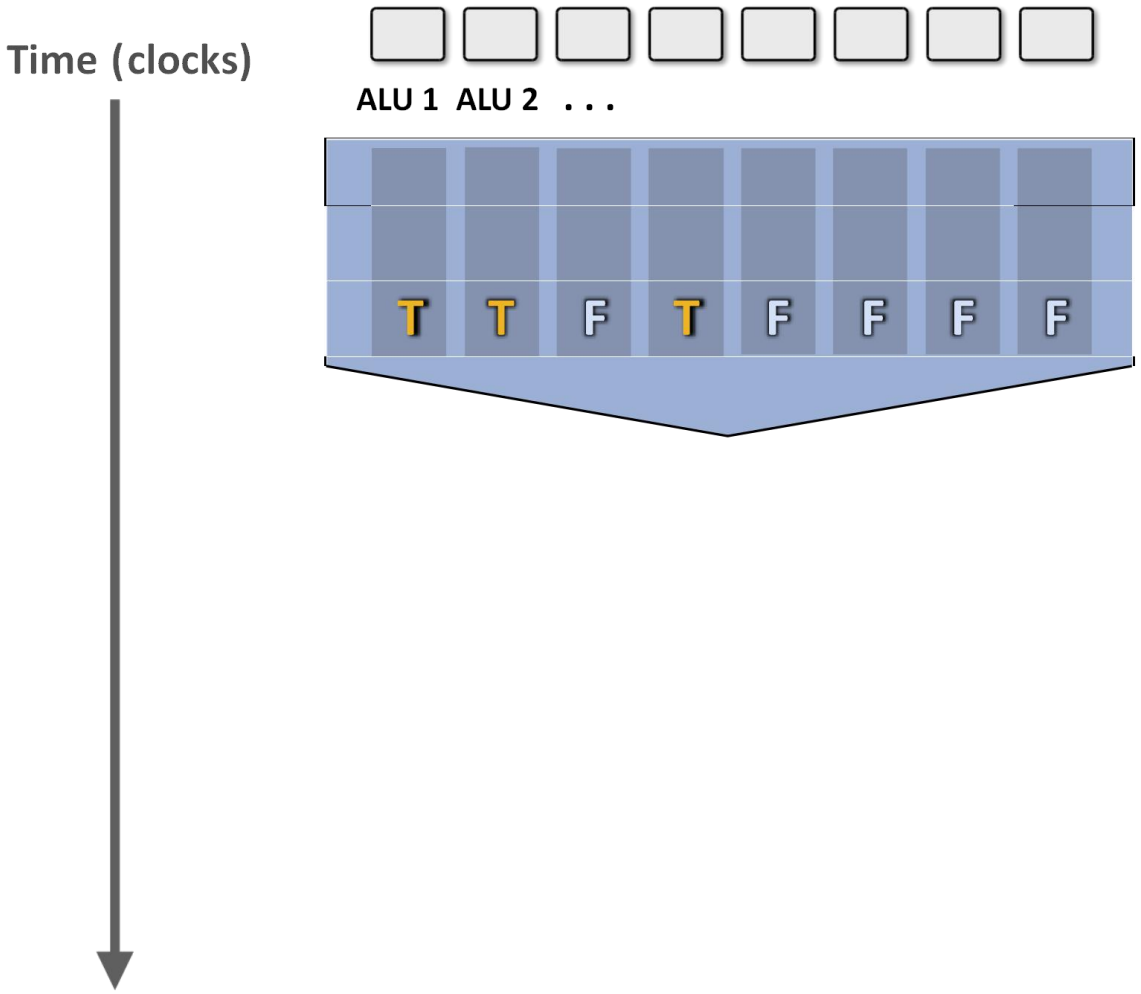
16 cores = 128 ALUs , 16 simultaneous instruction streams

But what about branches?



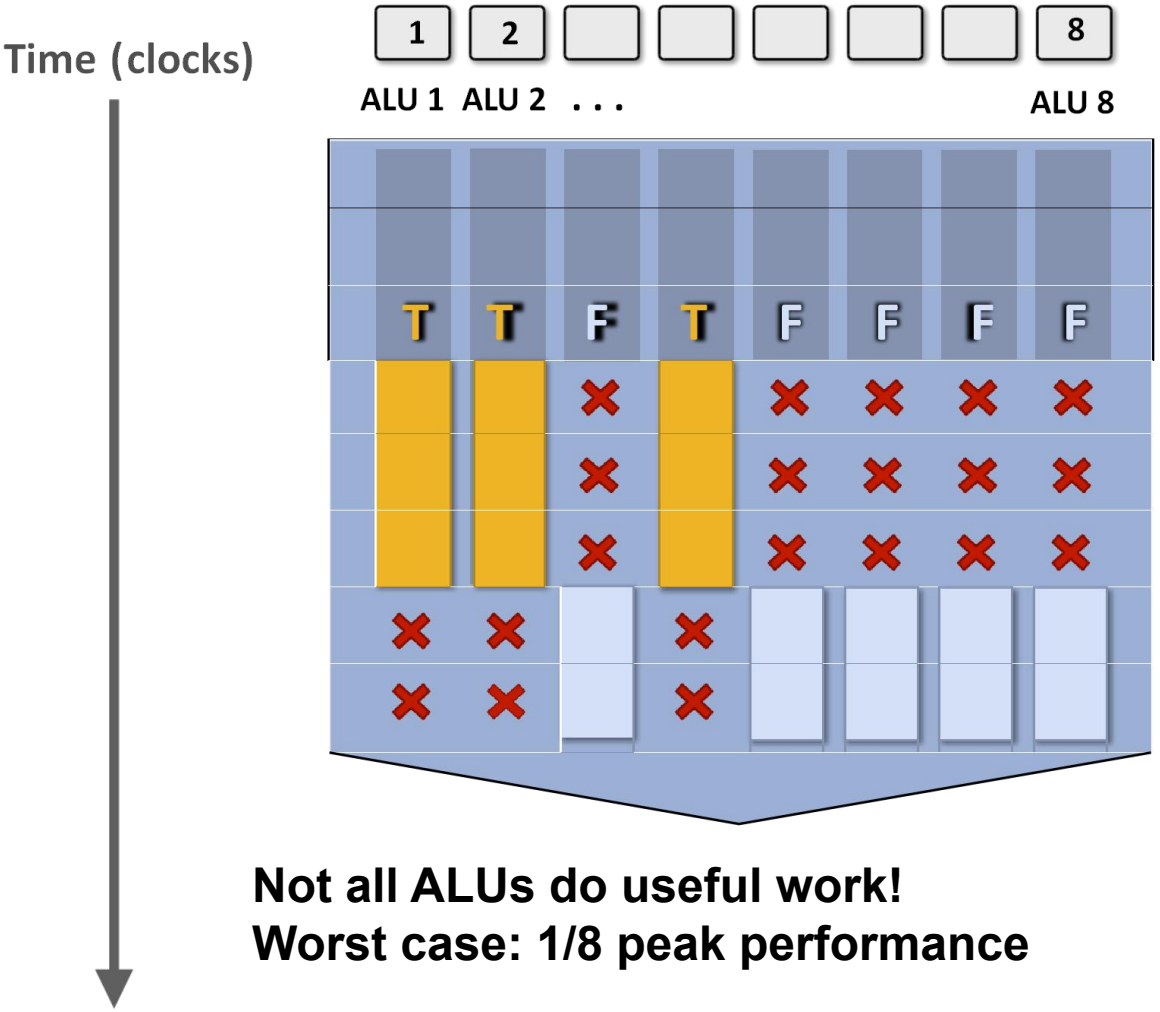

```
<unconditional
 shader(code>

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>
```

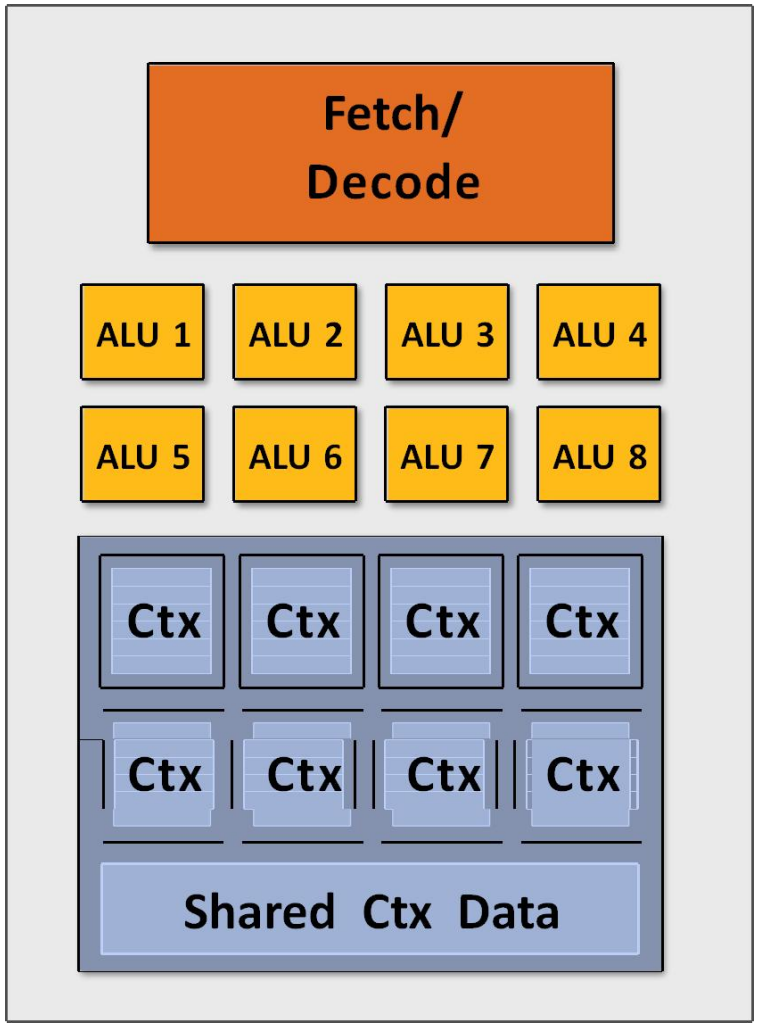
But what about branches?



Not all ALUs do useful work!
Worst case: 1/8 peak performance

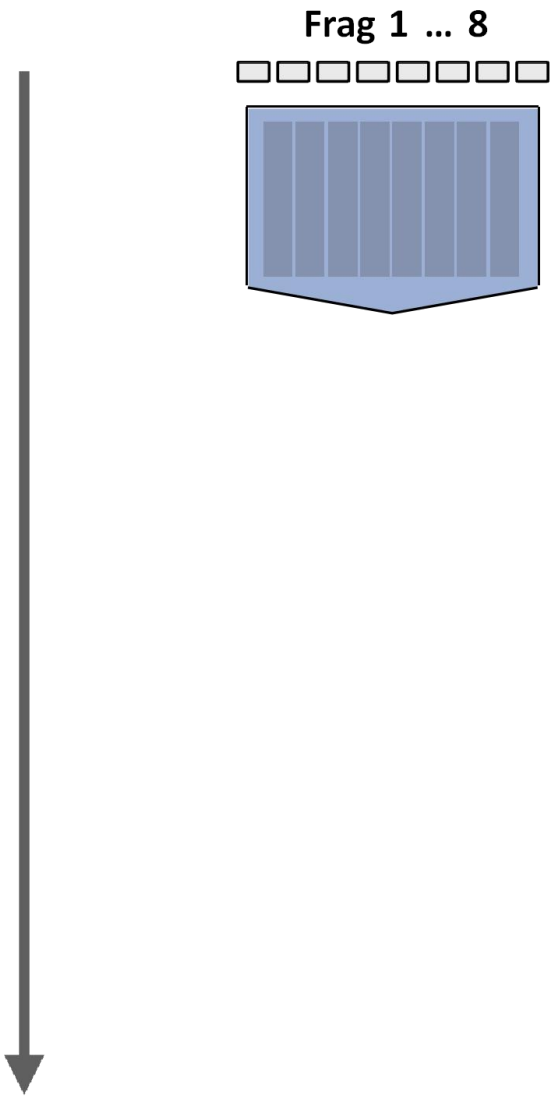
```
<unconditional
 shader(code>

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

<resume(unconditional
 shader(code>
```

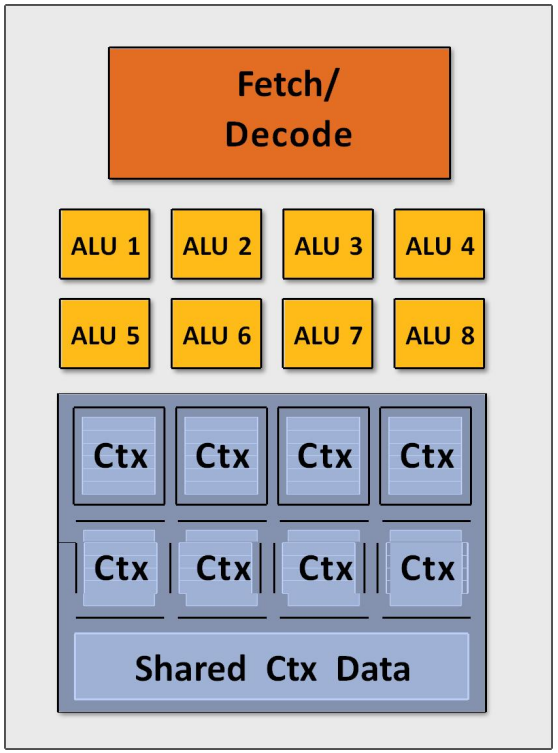
Hiding shader stalls



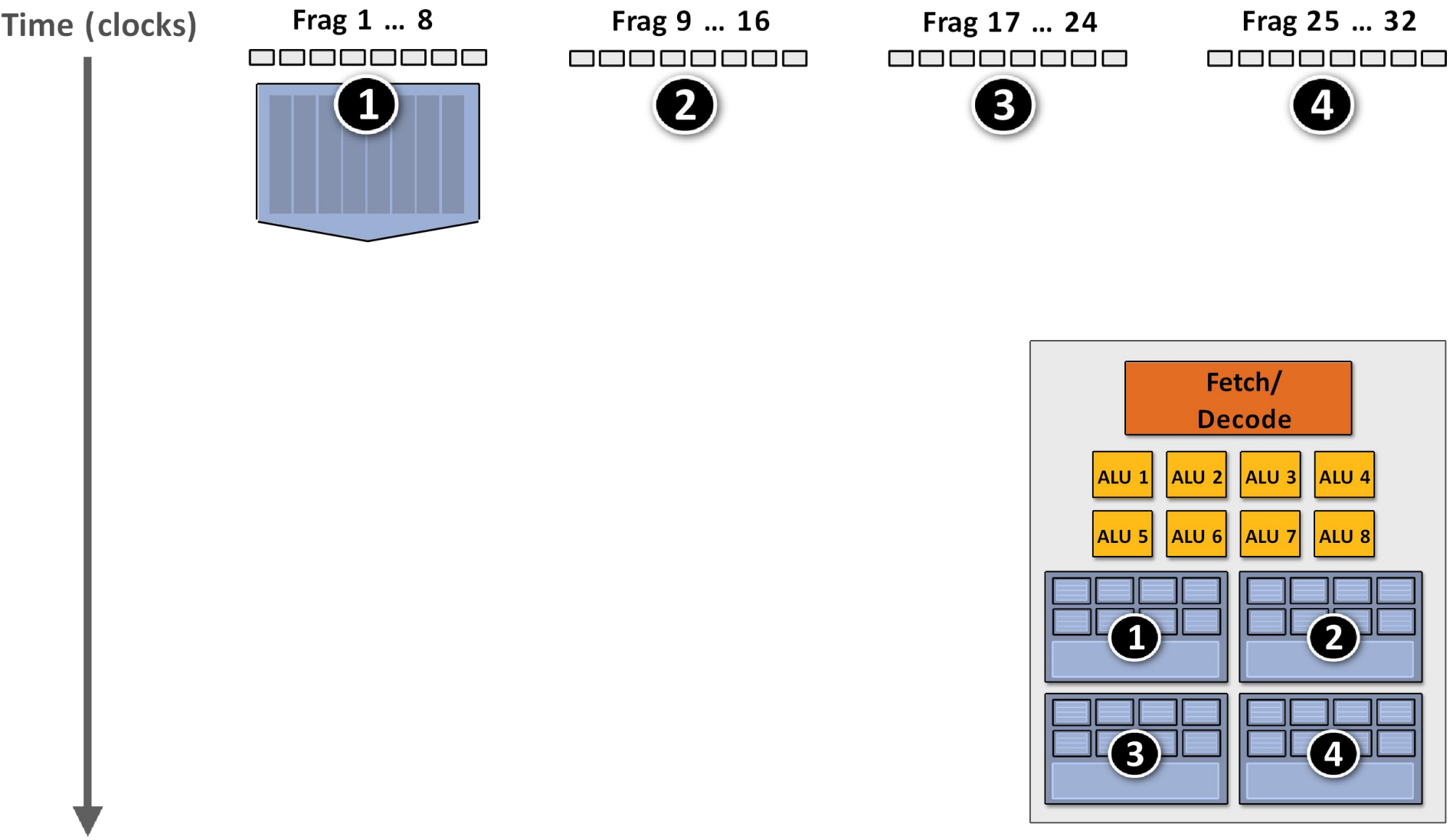
Idea #3:
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

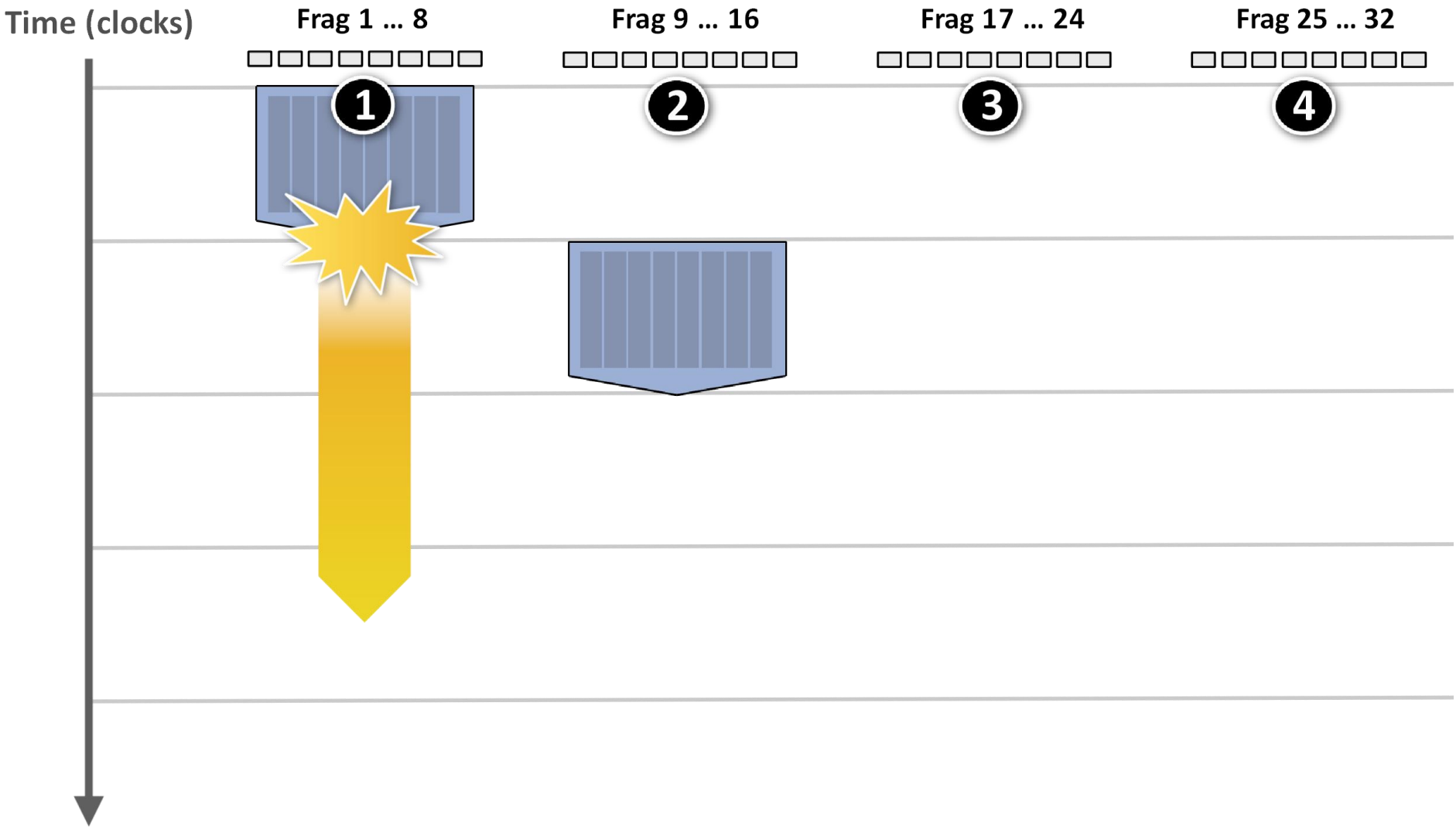




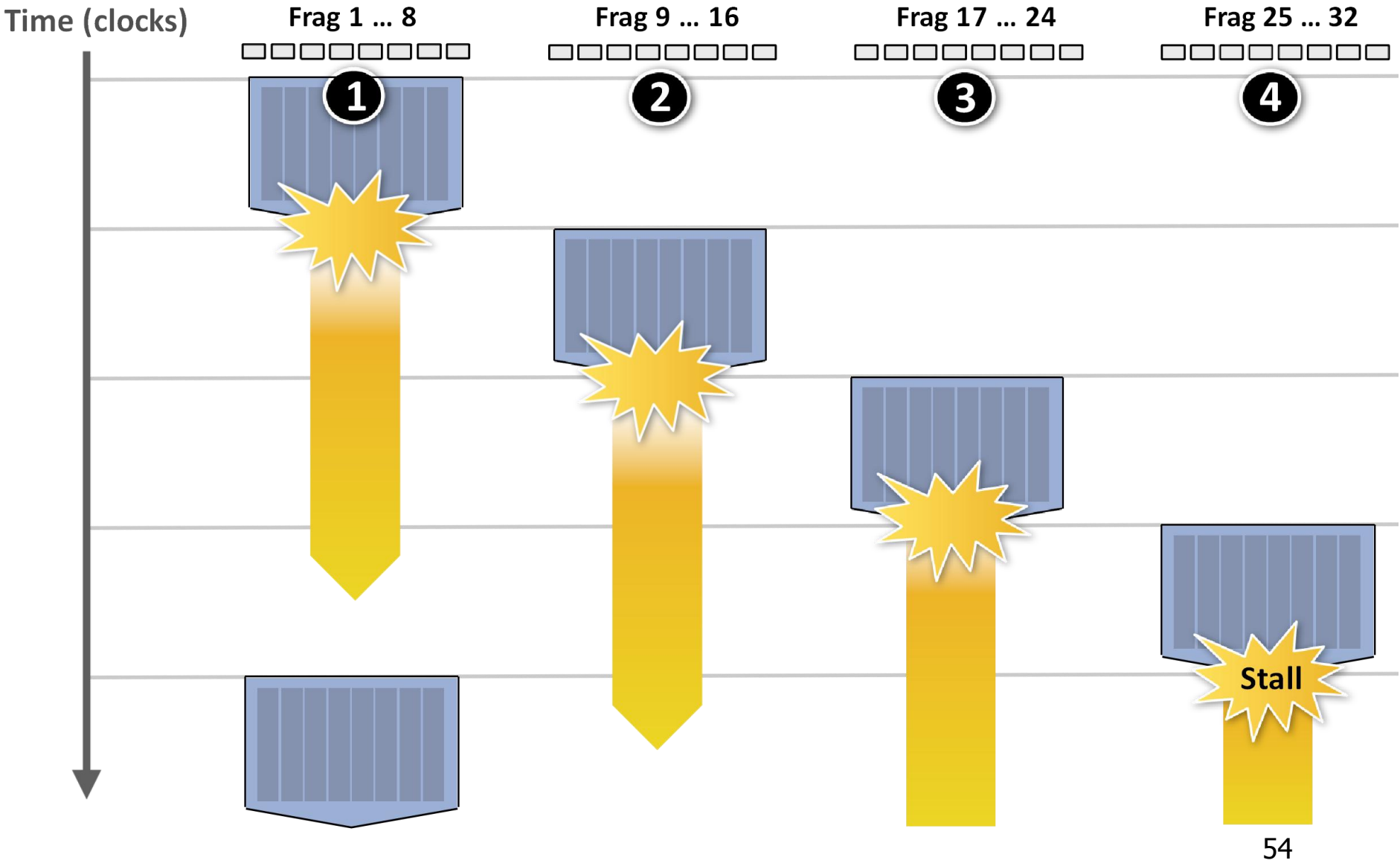
Hiding shader stalls



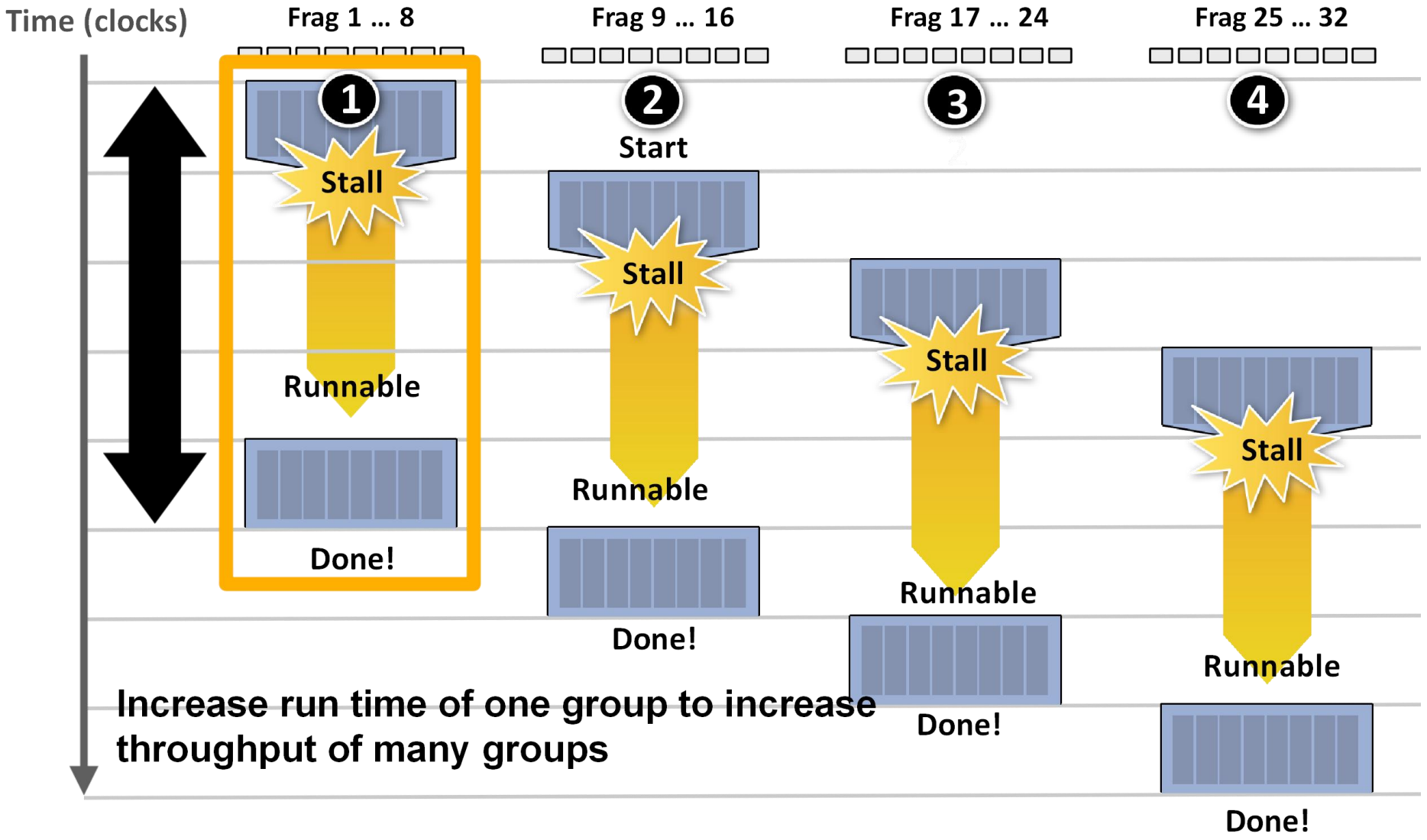
Hiding shader stalls



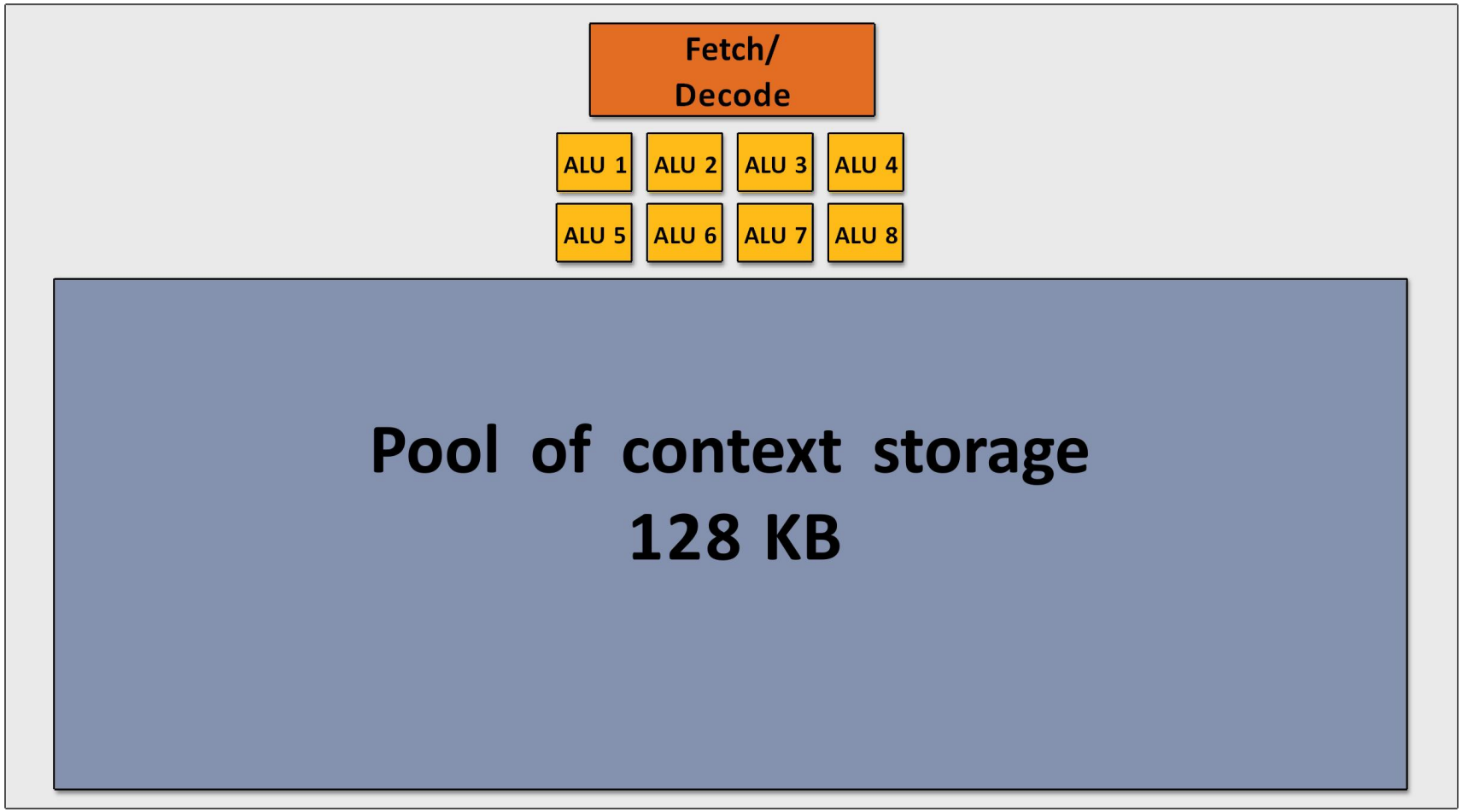
Hiding shader stalls



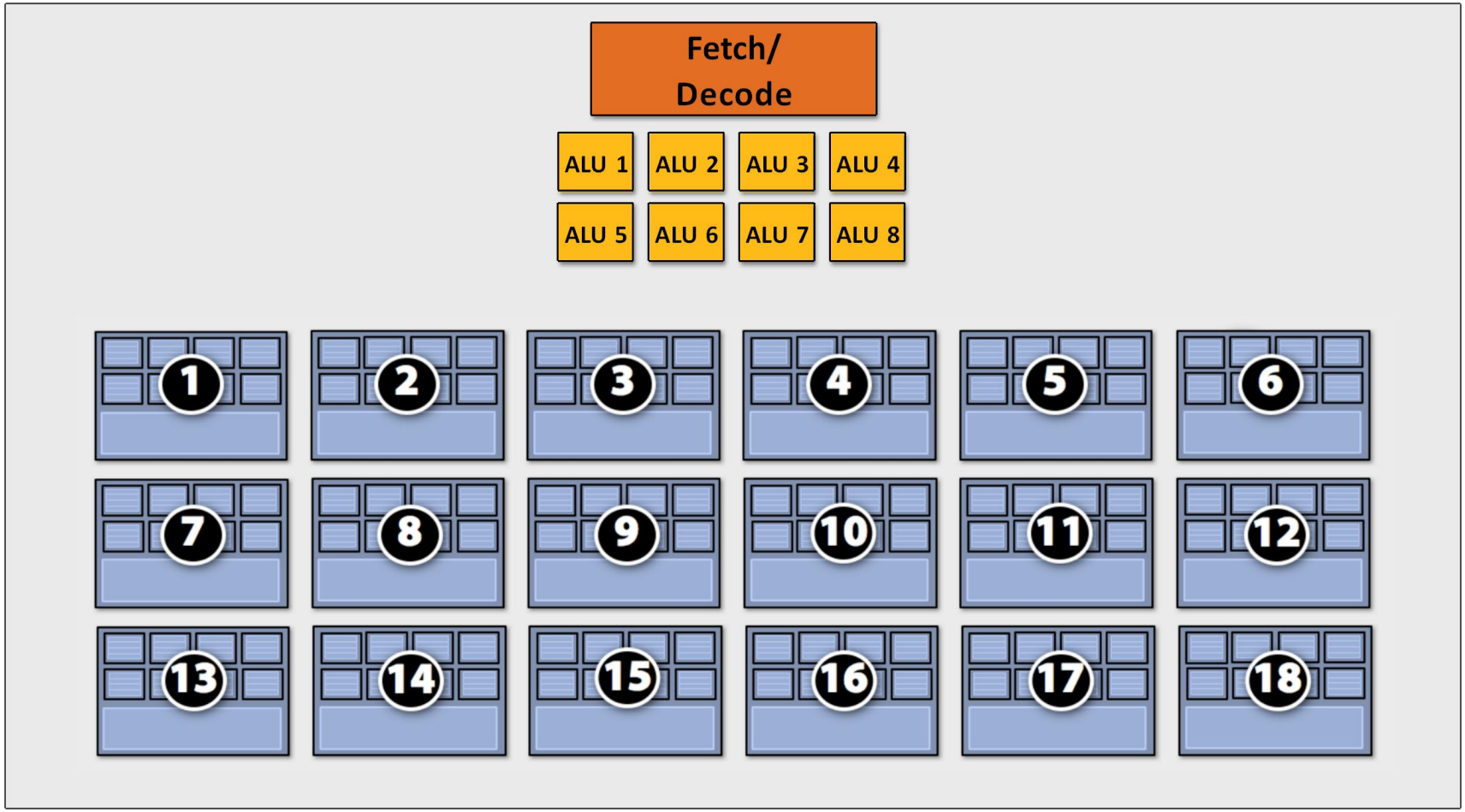
Throughput!


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

Storing contexts



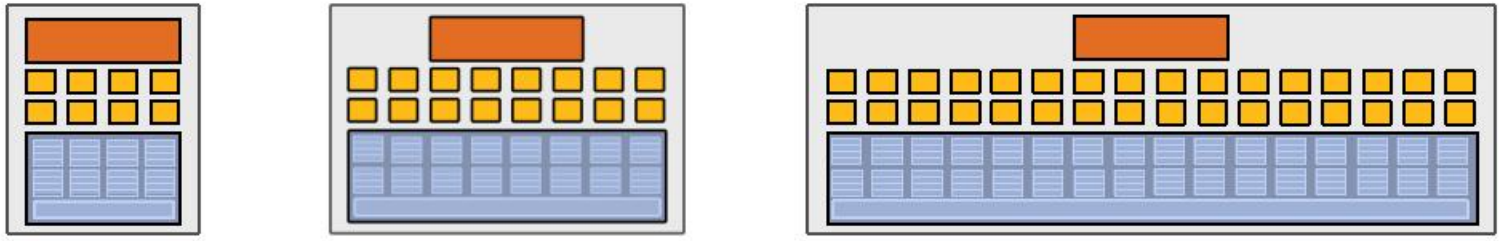
Eighteen small contexts



Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, Intel Larrabee
- Option 2: scalar instructions, implicit HW
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")



In practice: 16 to 64 fragments share an instruction stream.

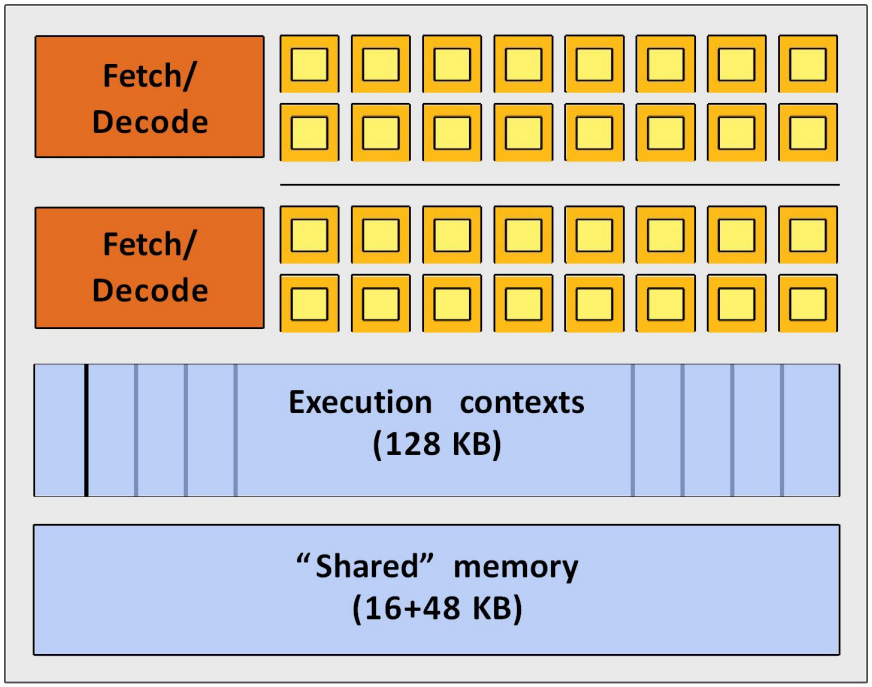
NVIDIA GeForce GTX 480 (Fermi)

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA-speak:
  - 480 stream processors (“CUDA cores”)
  - “SIMT execution”

- Generic speak:
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

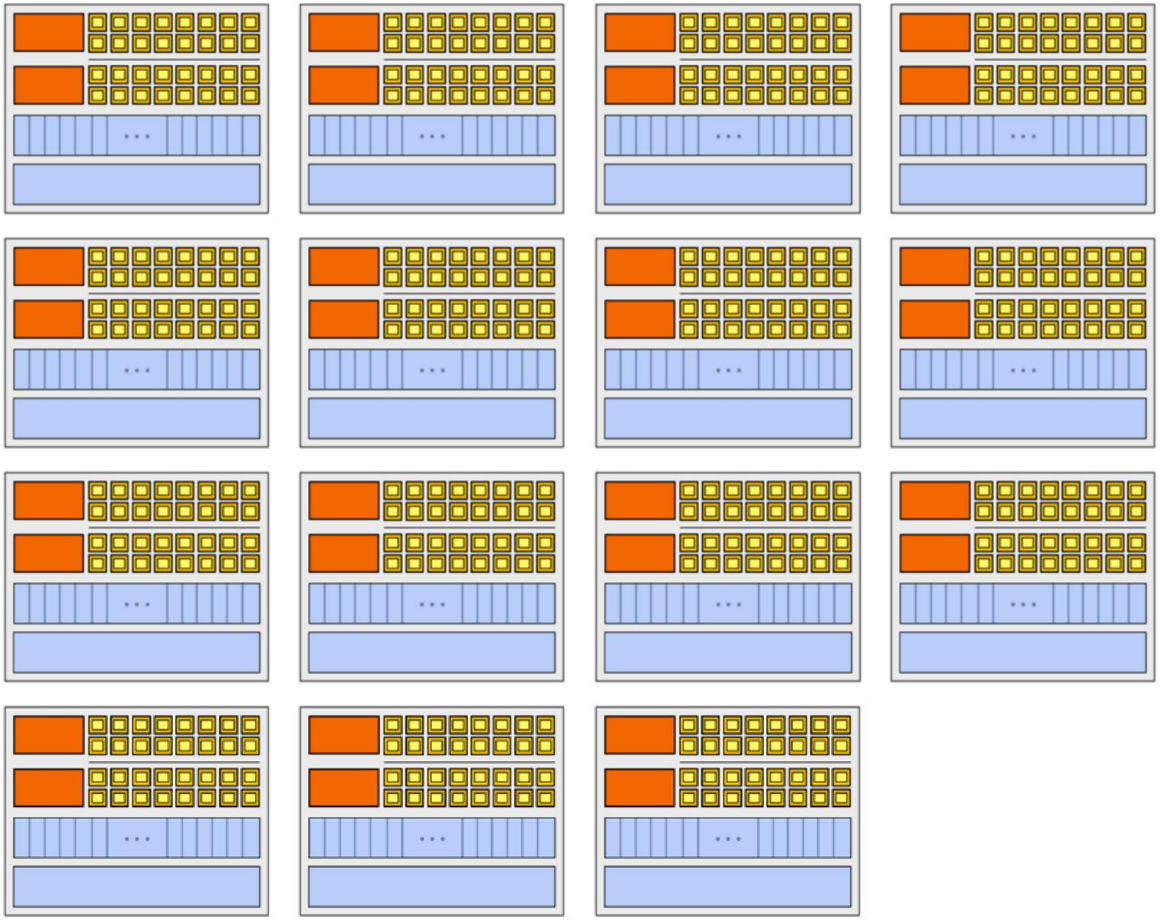
NVIDIA GeForce GTX 480 "SM"



= CUDA core
(1 MUL-ADD per clock)

- The SM contains 32 CUDA cores
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

NVIDIA GeForce GTX 480



There are 15 SMs on the GTX 480:

That's 23,000 fragments!

Or 23,000 CUDA threads!

GPU Summary

- Three major ideas that all modern processors employ to varying degrees
  - Employ multiple processing cores
    - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - Increase compute capability with little extra cost
  - Use multi-threading to make more eicient use of processing resources (hide latencies, ill all available resources)
- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound
- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales