Introduction to Graphics Processing Units

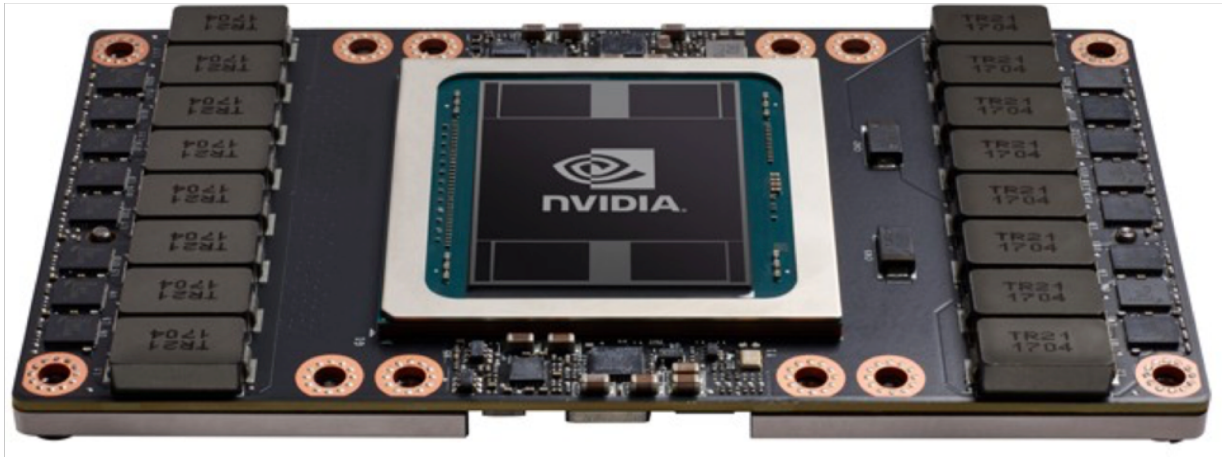
Lecture 19
December 11th, 2018

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Nvidia Volta Whitepaper; Justin Hensley from AMD (Throughput Computing)*

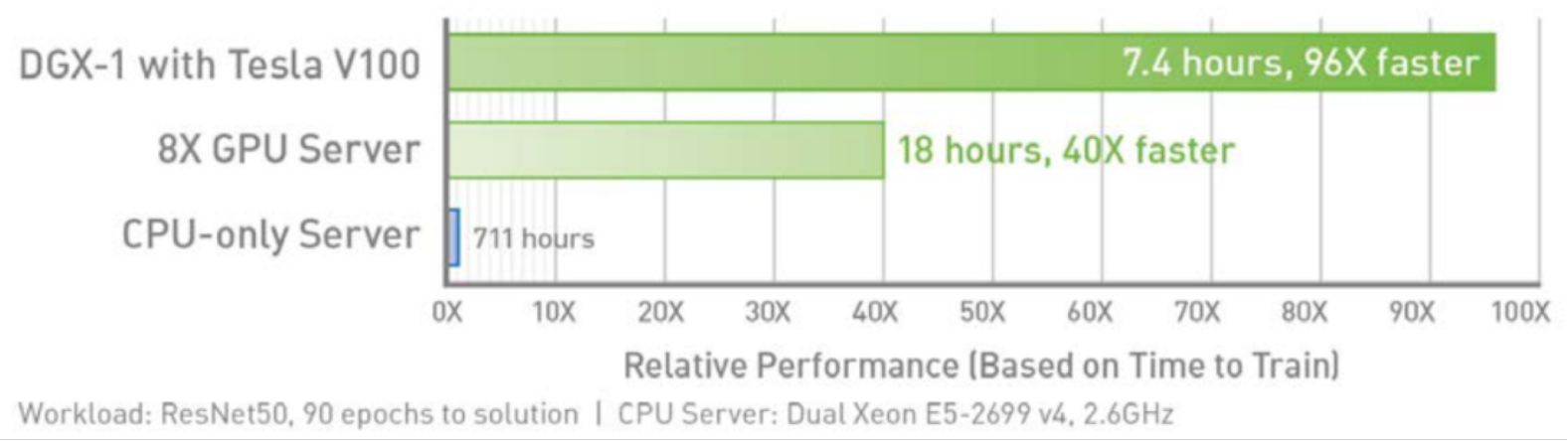
State-of-the-art GPU

- **Nvidia Tesla V100**
  - Announced in May 2017
  - 815mm^2@ TSMC 12nm, 21B transistors
  - 5,120 CUDA cores
  - 7.5 FP64 TFLOPS, 15 FP32 TFLOPS
  - NEW 120 Tensor TOPS
  - 20MB register file, 16MB cache
  - 16GB HBM2 DRAM @ 900 GB/s
  - 300 GB/s NVLink



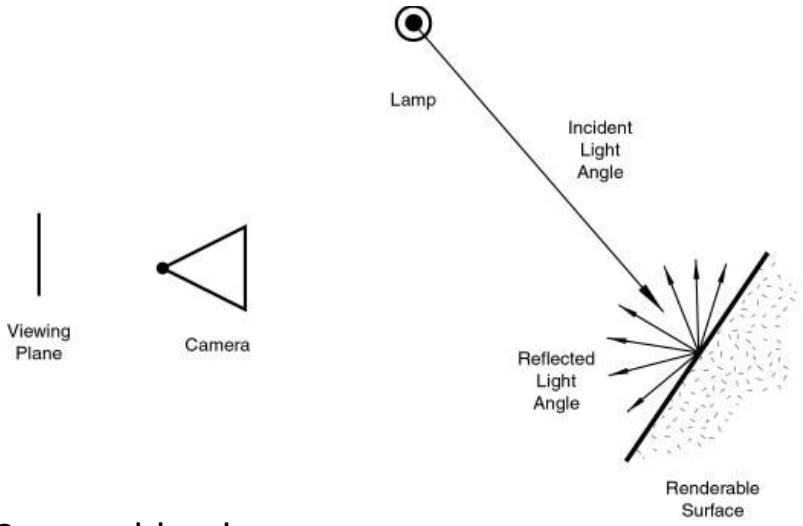
State-of-the-art GPU

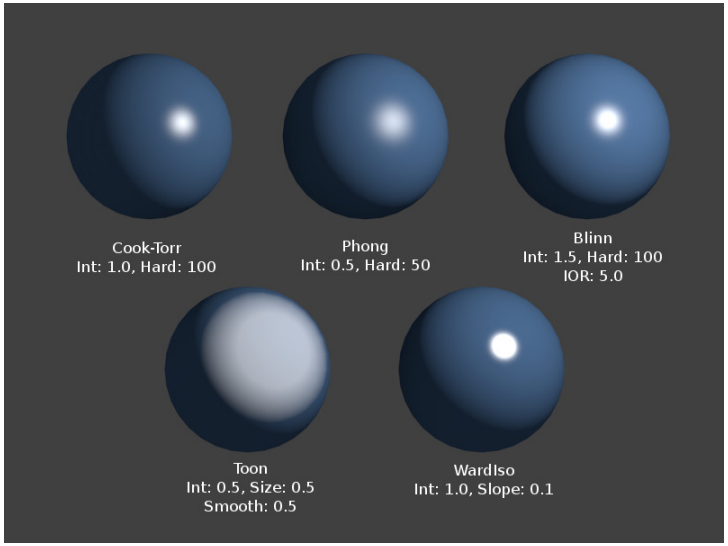
- **Nvidia Tesla V100**
  - Particularly well suited for deep learning (DL) workloads
  - DGX-1 server (with 8 V100 GPUs) delivers 96x faster training than CPU (Source: Nvidia Volta Architecture Whitepaper)



Hardware for Throughput Computing

- **Example: Diffuse shader**
  - Determines the general color of a material when light shines on it
  - Most shaders give a smooth falloff from bright to dark from the point of the strongest illumination to the shadowed areas.
  - Light striking a surface and then re-irradiated via a Diffusion phenomenon will be scattered.
  - If most of the light striking a surface is reflected diffusely, the surface will have a matte appearance.





Source: blender.org

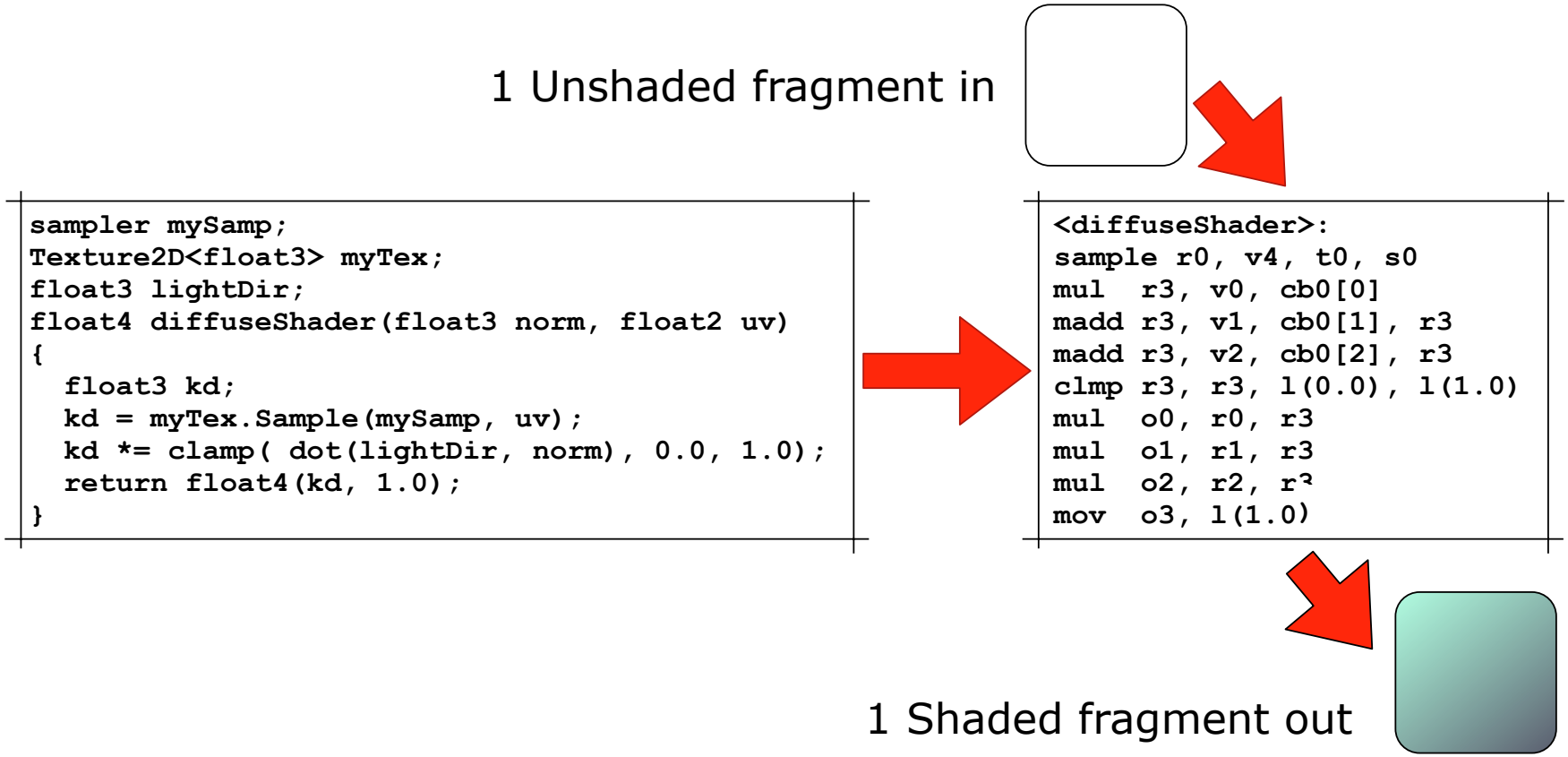
Hardware for Throughput Computing

- **Example: Diffuse shader**
  - Code looks like this.
  - Each invocation is independent, but explicitly exposed parallelism.

```
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);
}
```

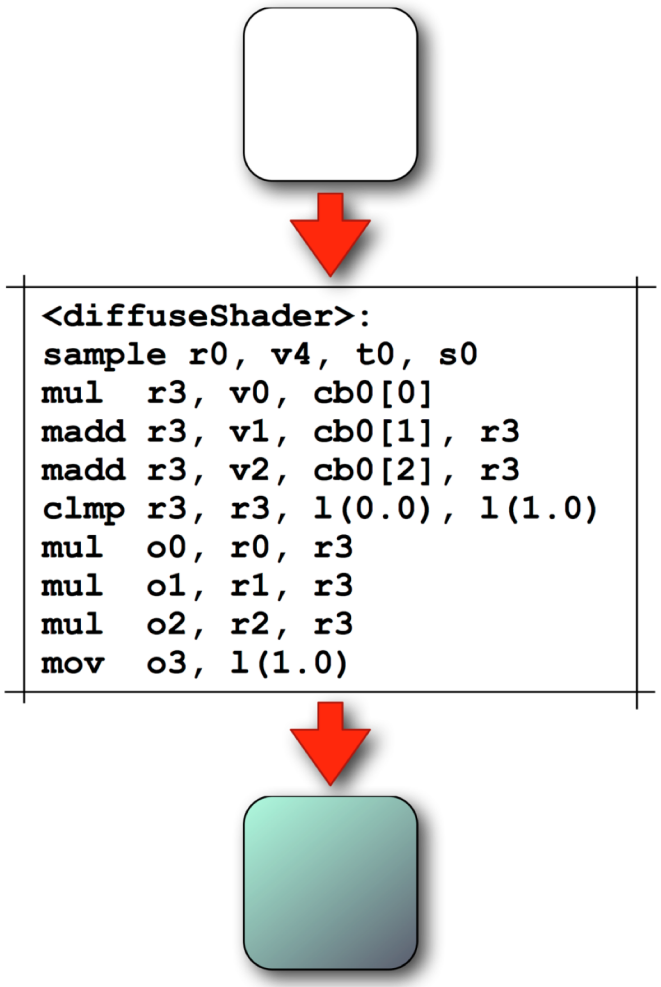
Hardware for Throughput Computing

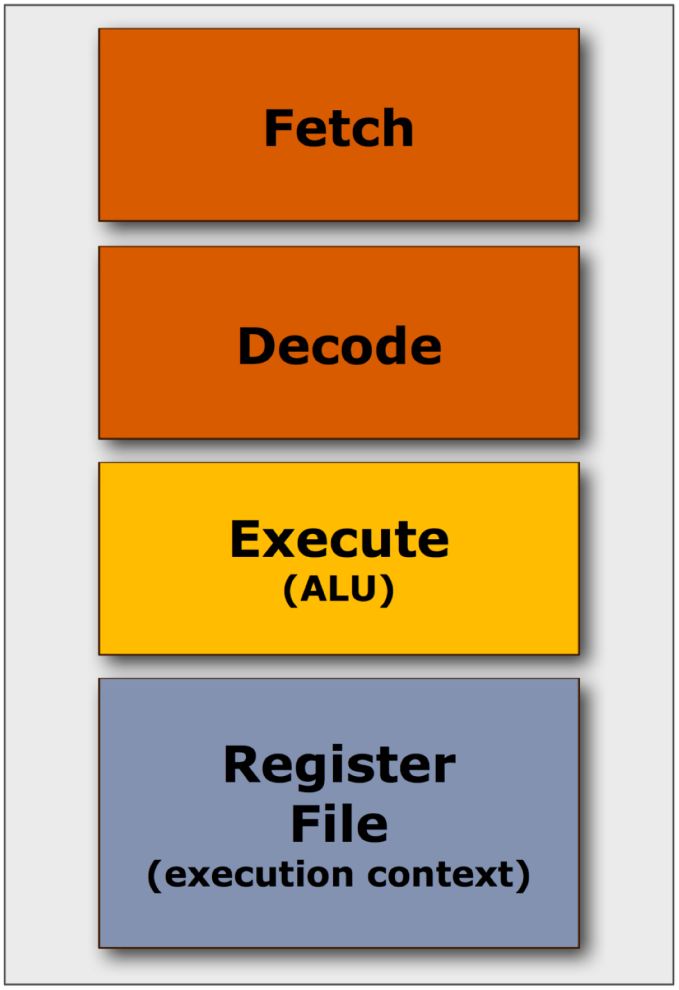
- **Shader is compiled.**



Hardware for Throughput Computing

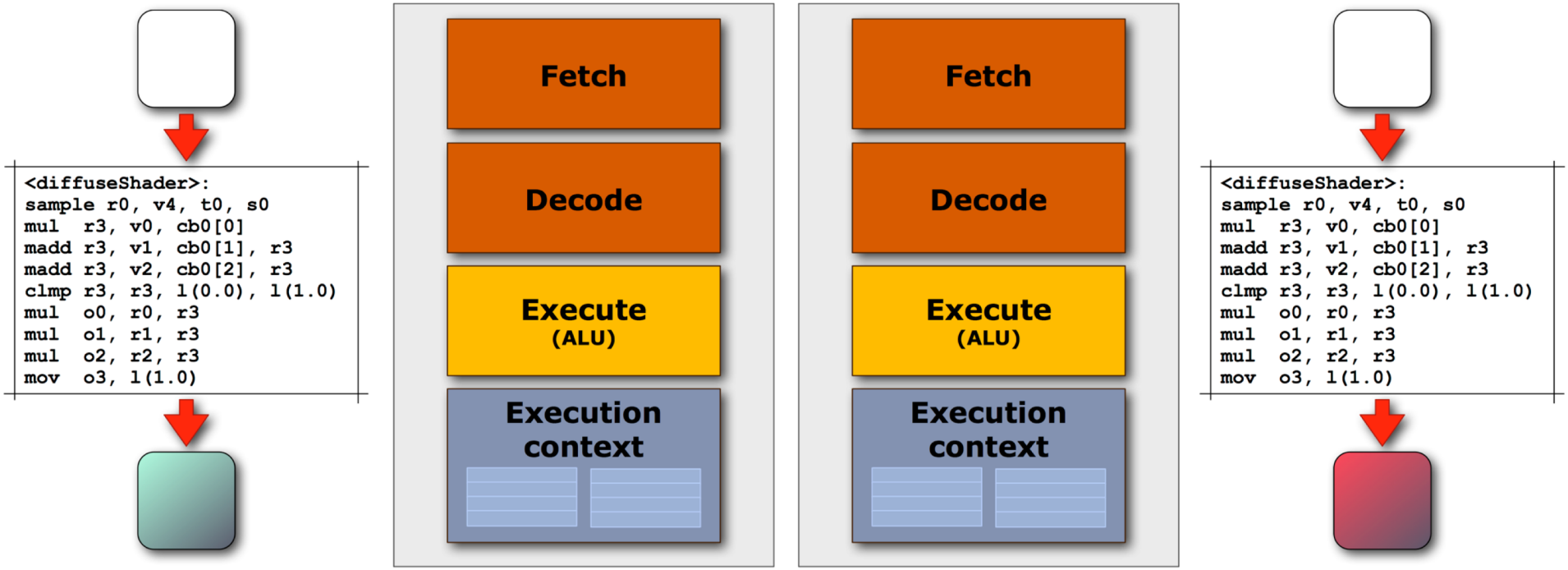
- **Execute the shader: processing one fragment**





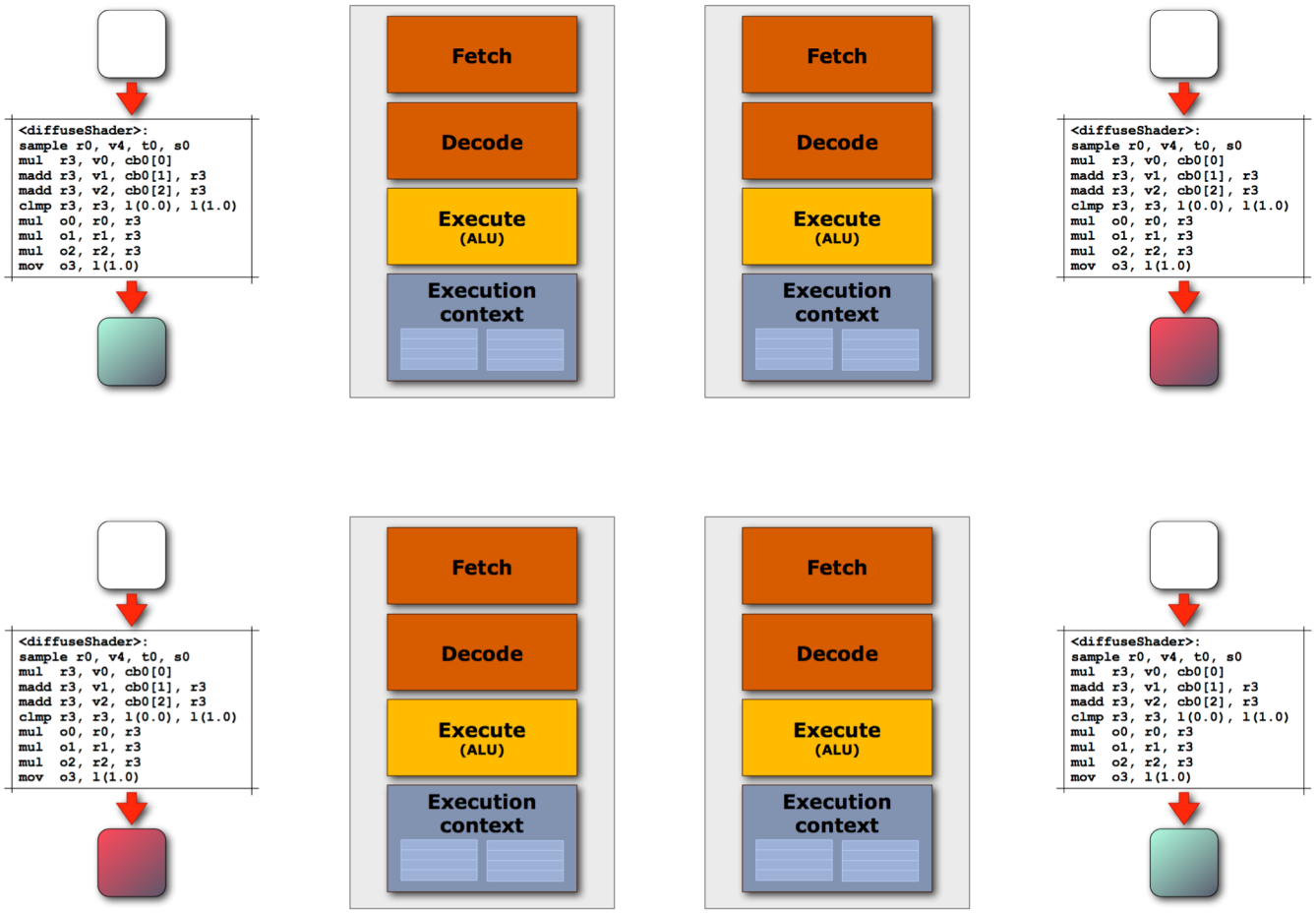
Hardware for Throughput Computing

- **Exploit data parallelism! – add two cores**
  - Each invocation is independent!



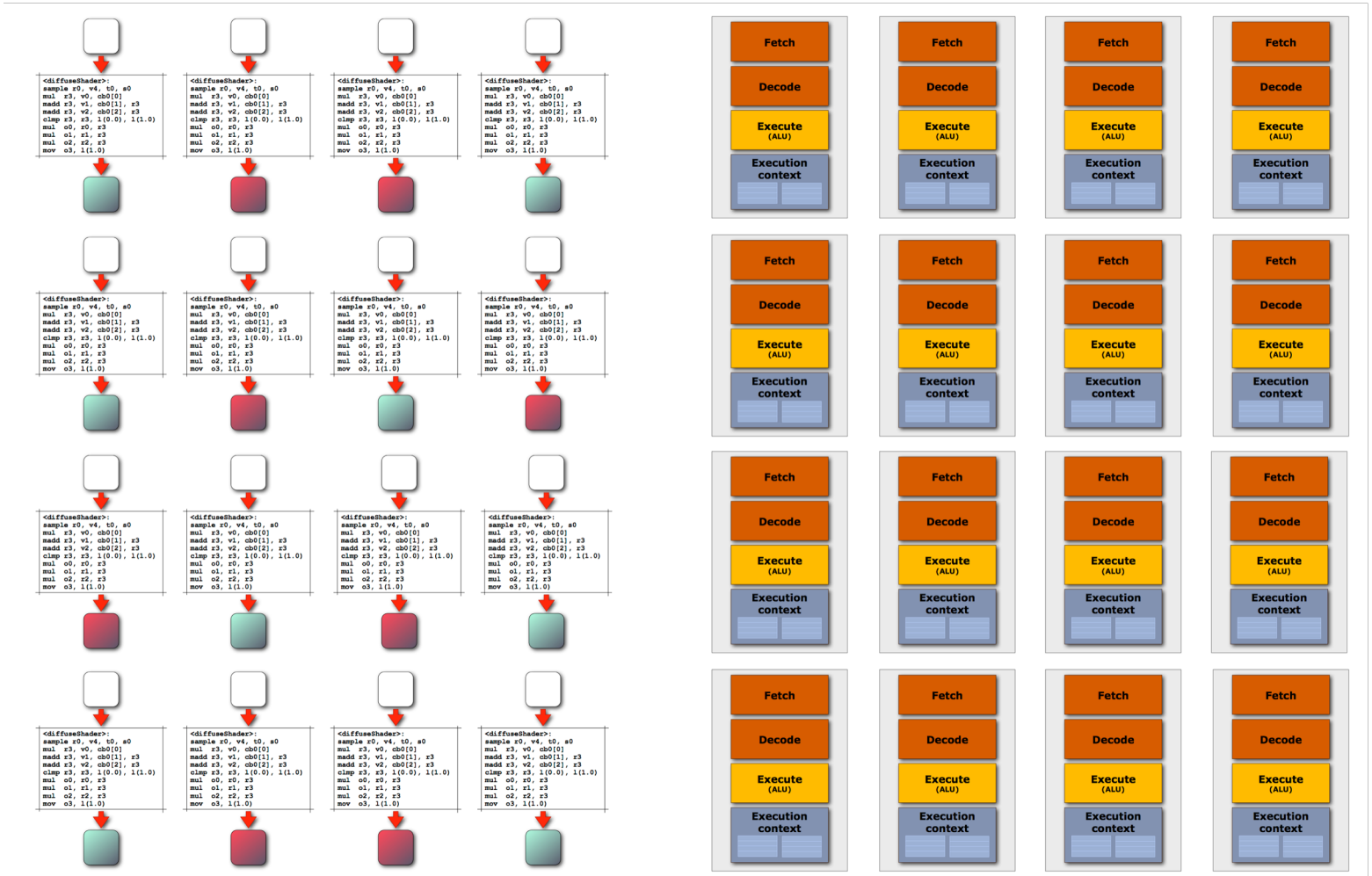
Hardware for Throughput Computing

- Add even more cores – four cores



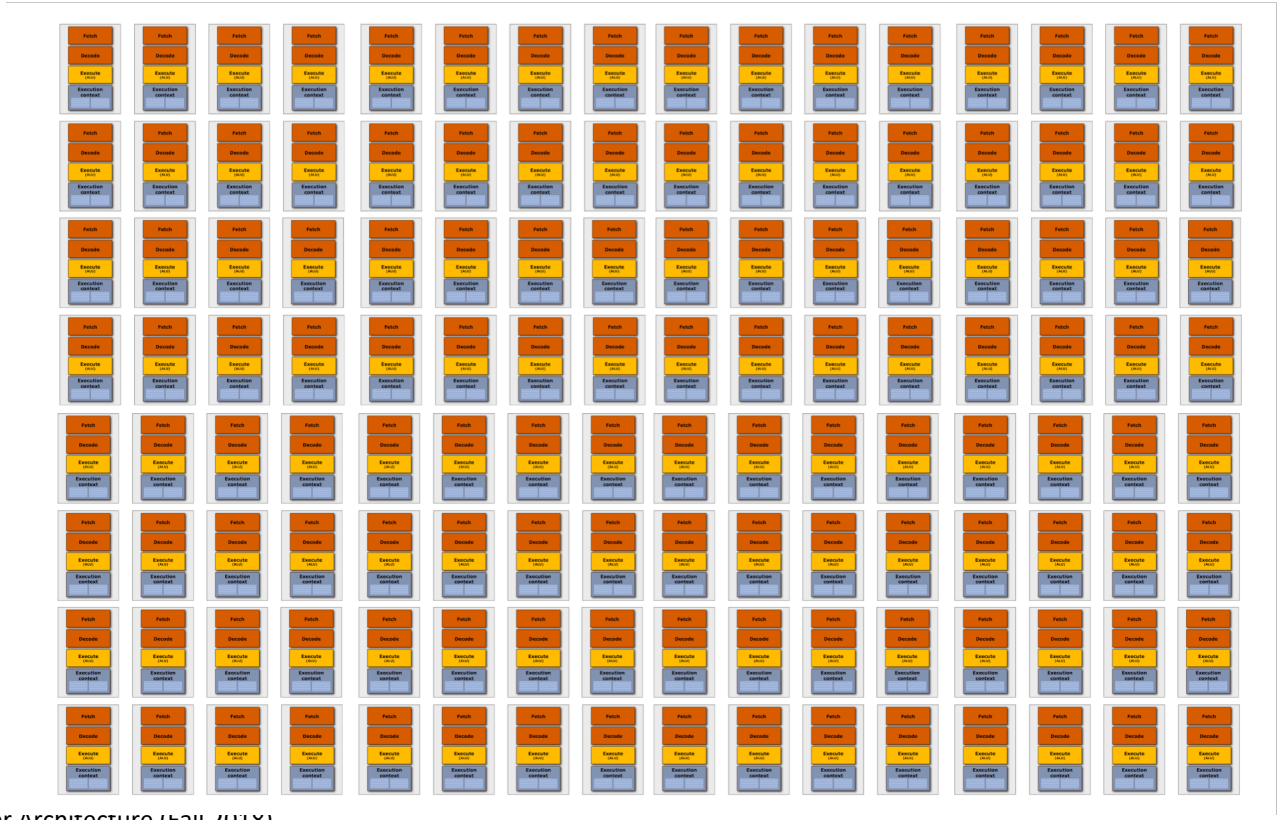
Hardware for Throughput Computing

- How about even more cores – 16 cores



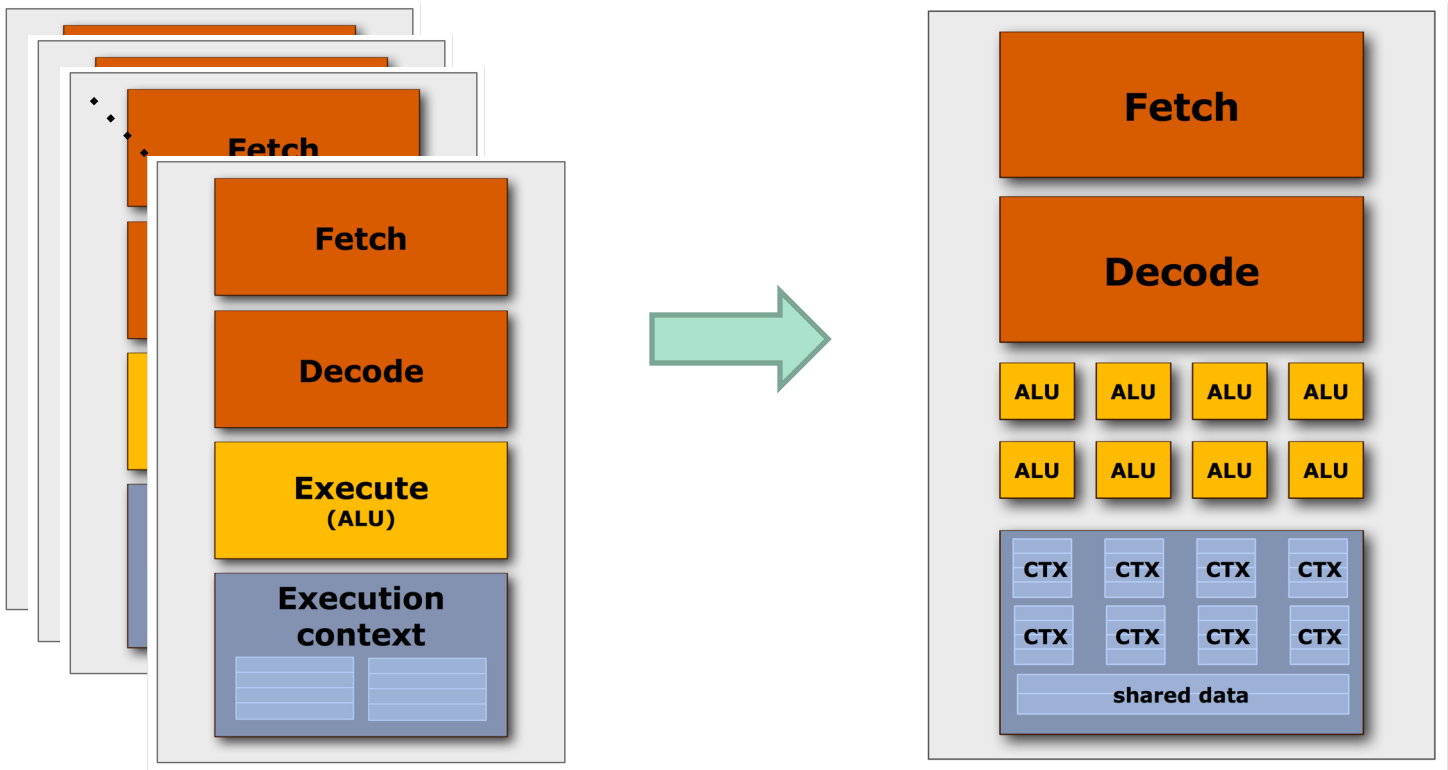
Hardware for Throughput Computing

- **128 cores?**
  - How you feed all these cores?
  - Exploit data parallelism! – Graphics requires hardware to process lots of items that share the same program.



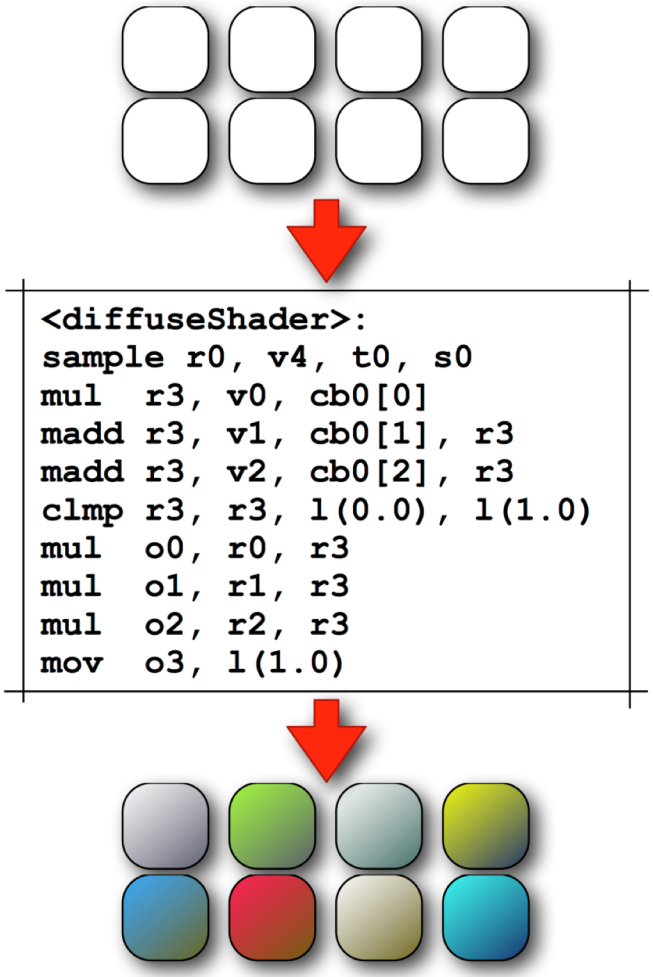
Hardware for Throughput Computing

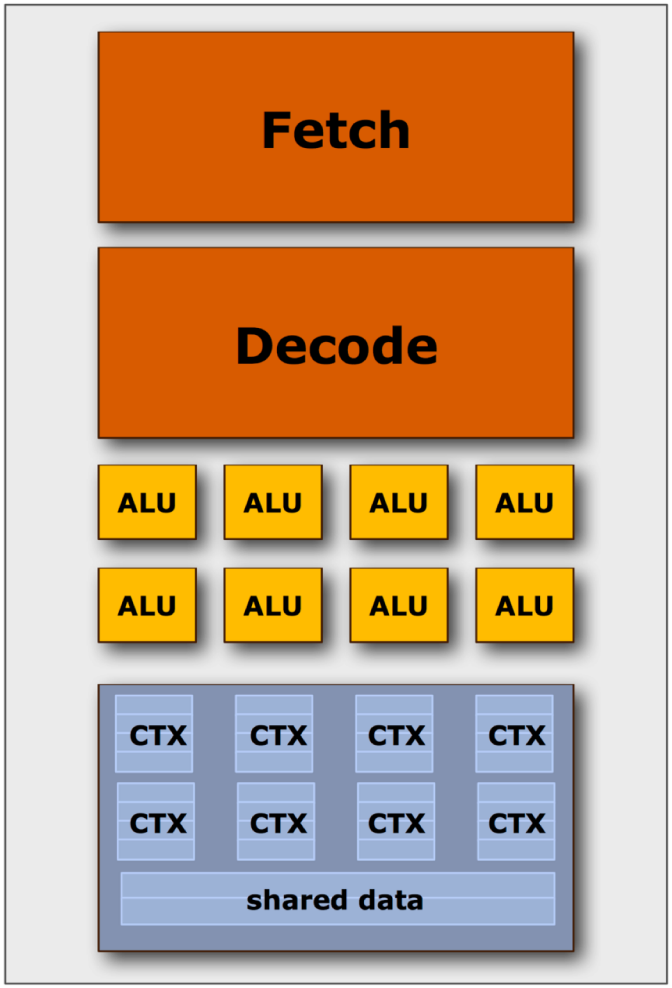
- **Back to simple core**
  - How do you feed all these cores?
  - Share cost of fetch/decode across many ALUs
  - SIMD processing!



Hardware for Throughput Computing

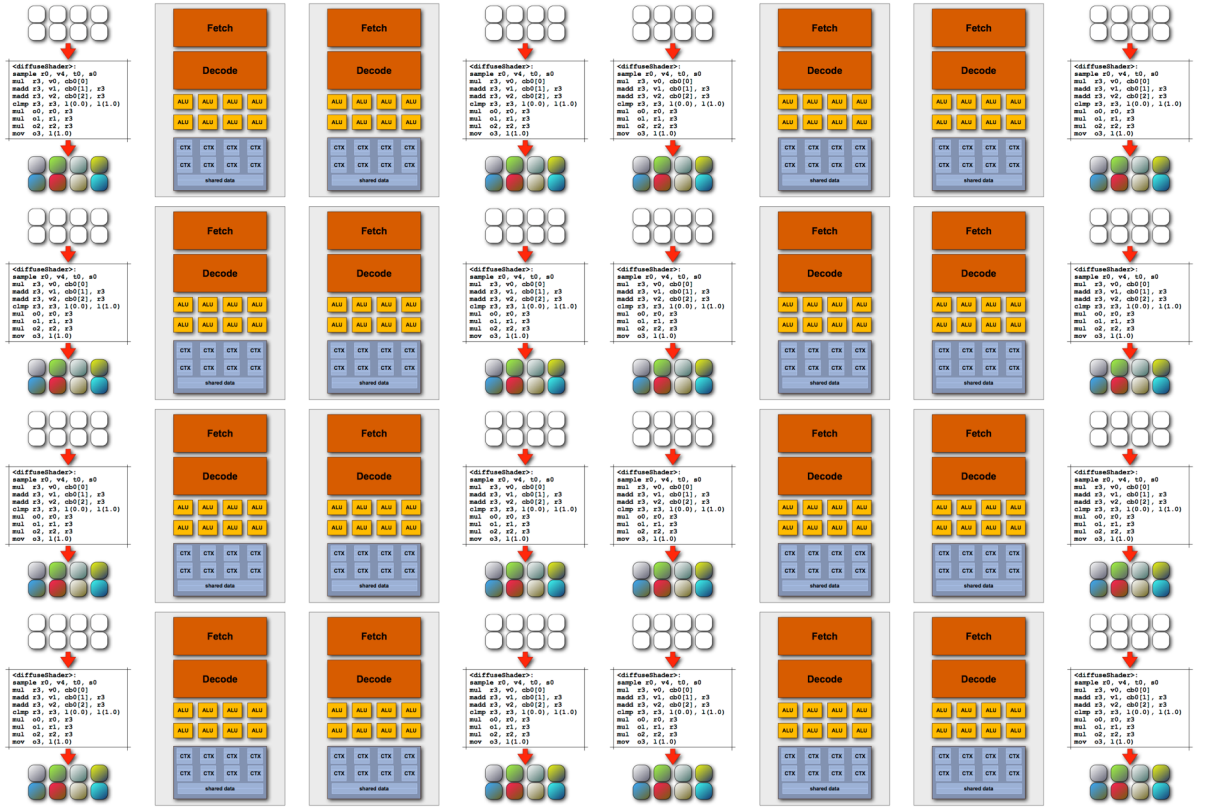
- Back to single core





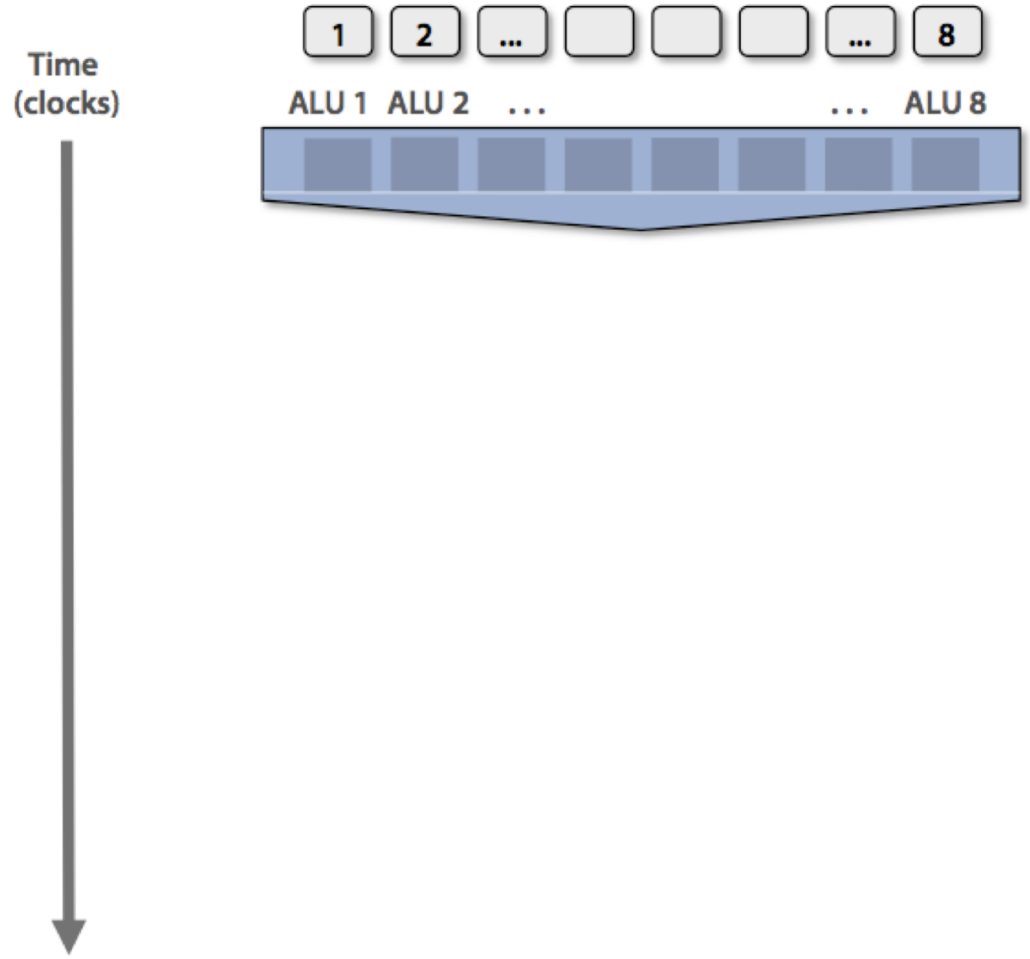
Hardware for Throughput Computing

- **128 fragments in parallel**
  - 16 cores → 128 ALUs (16 cores * 8 ALUs/core)
  - 16 independent instruction streams



Hardware for Throughput Computing

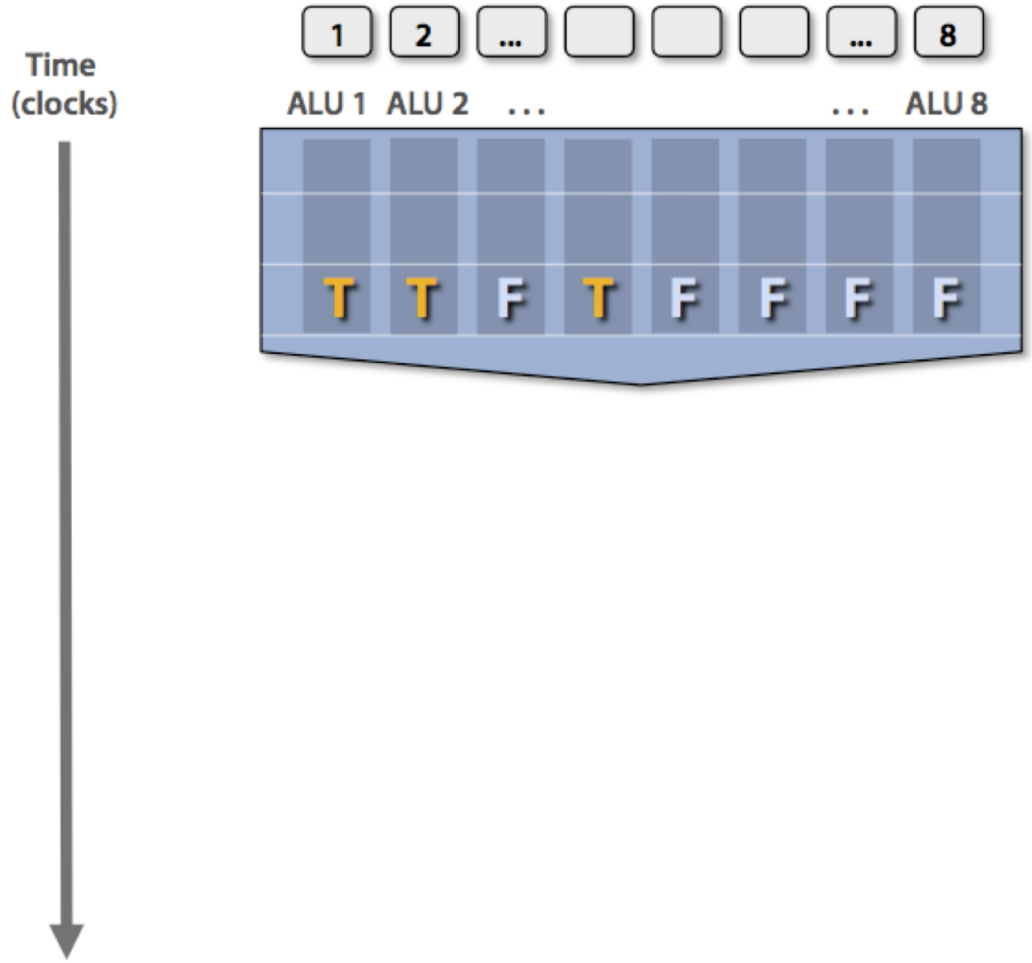
- **What about branching?**




```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

Hardware for Throughput Computing

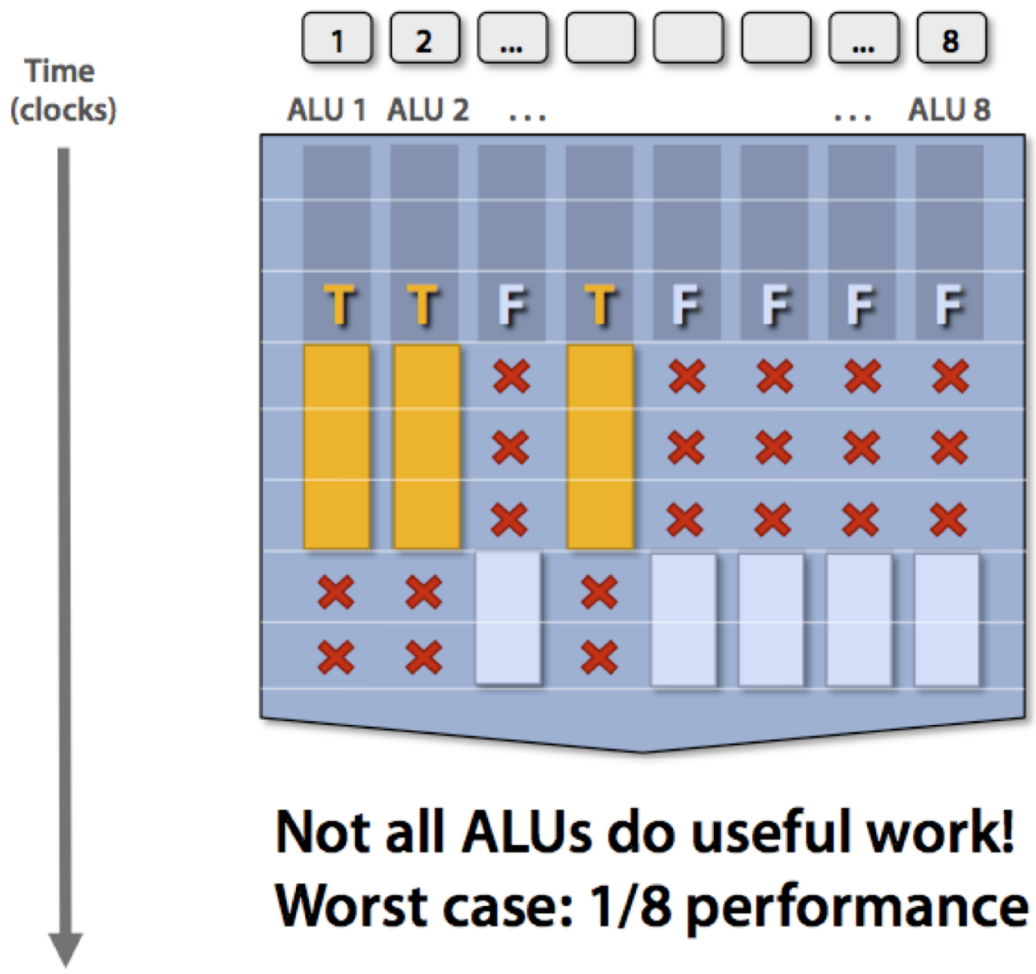
- **What about branching?**




```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

Hardware for Throughput Computing

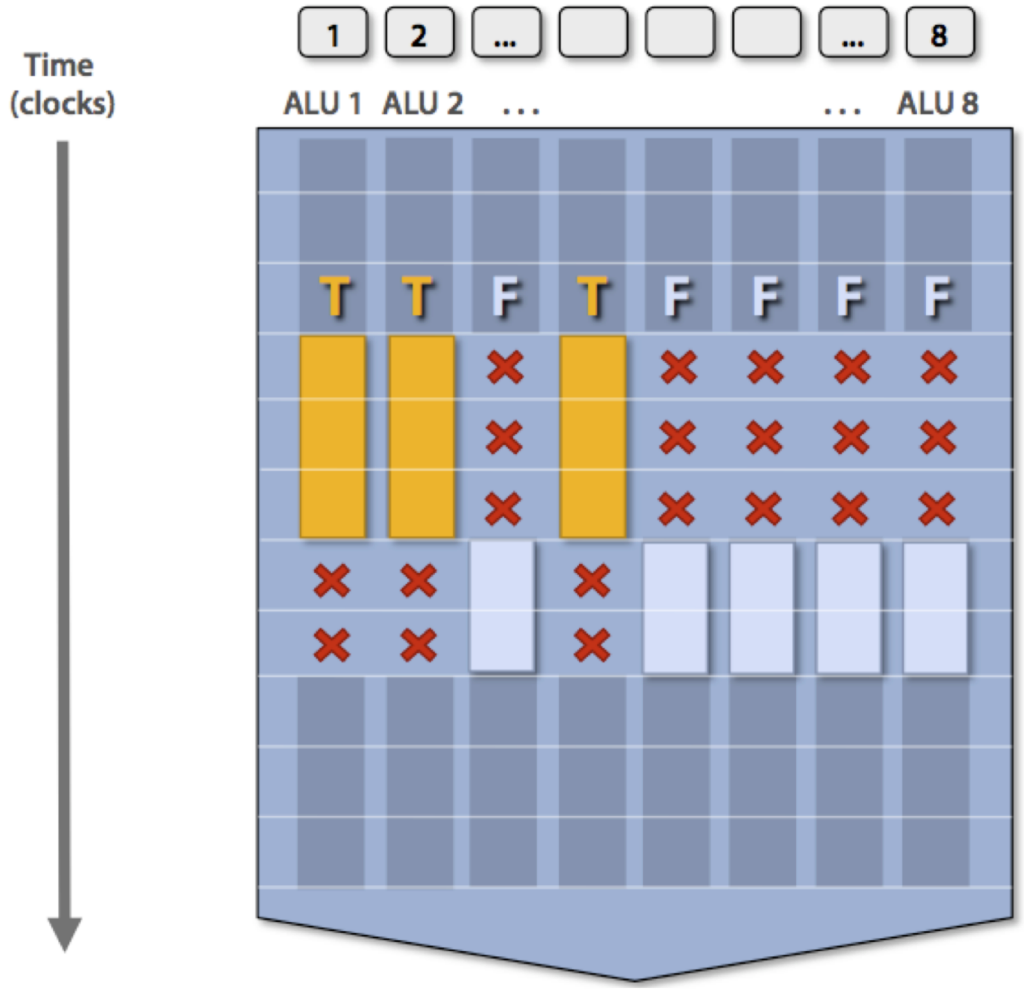
- What about branching?




```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

Hardware for Throughput Computing

- What about branching?



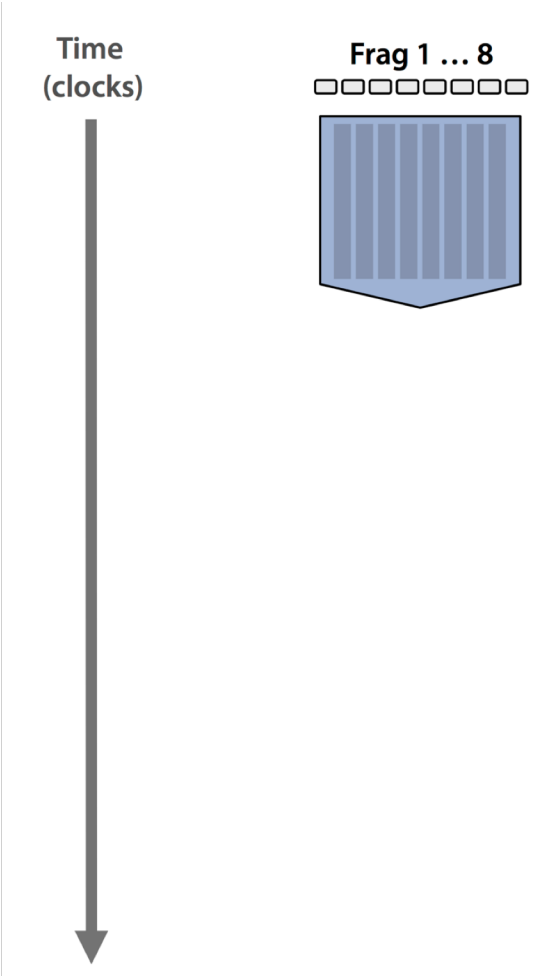

```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

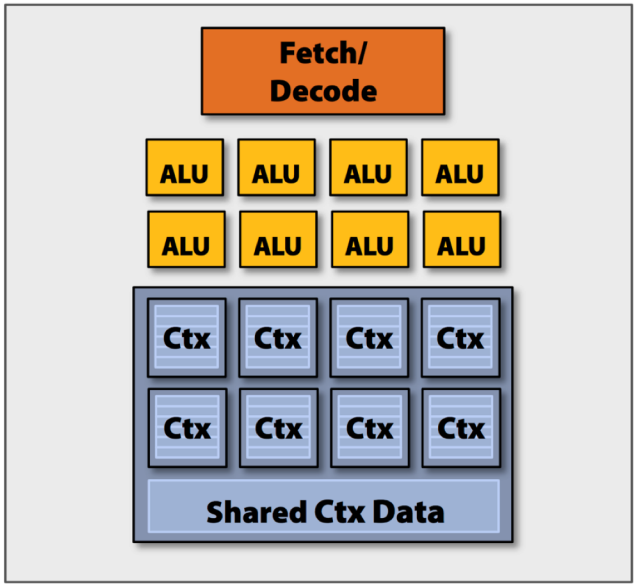
Hardware for Throughput Computing

- **How to handle stalls (e.g, cache misses)?**
  - Memory access latency = 100's to 1000's of cycles
    - Stalls occur when a core cannot run the next instruction
  - GPUs don't have the large / fancy caches and logic that helps avoid stall because of a dependency on a previous operation.
  - GPUs do have LOTS of independent "things" to work on
    - Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hardware for Throughput Computing

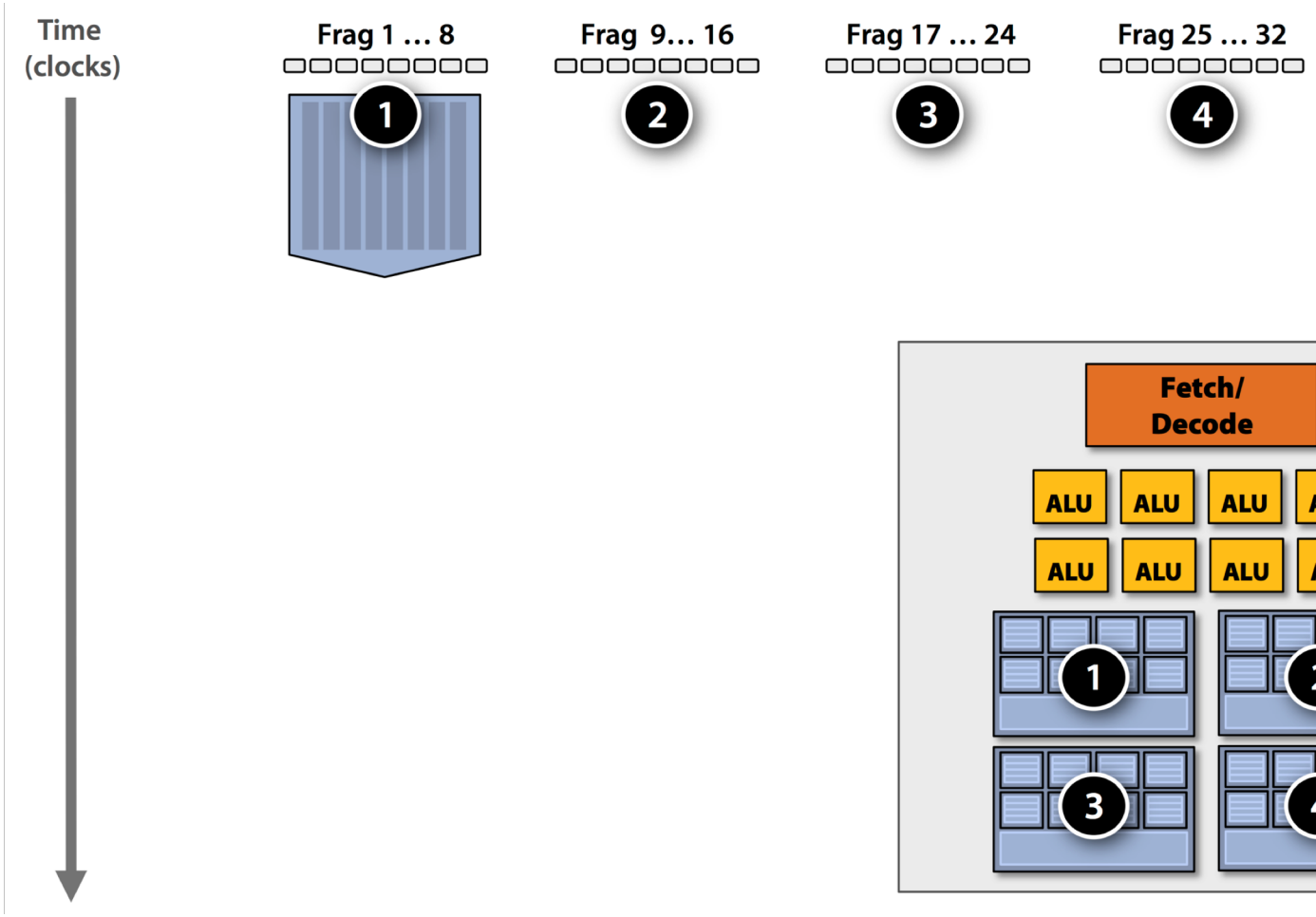
- **Hiding memory stalls**





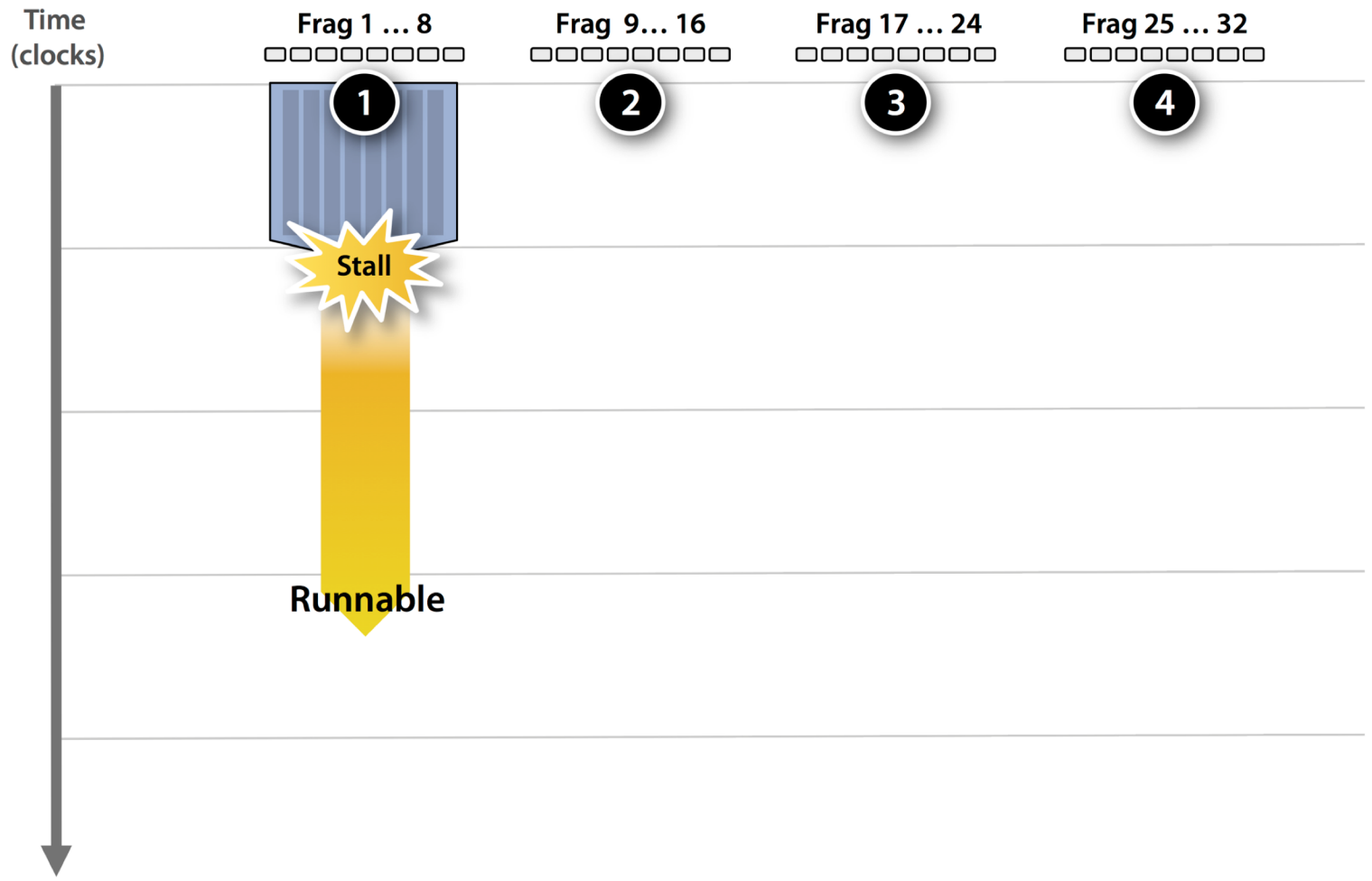
Hardware for Throughput Computing

- **Hiding memory stalls**



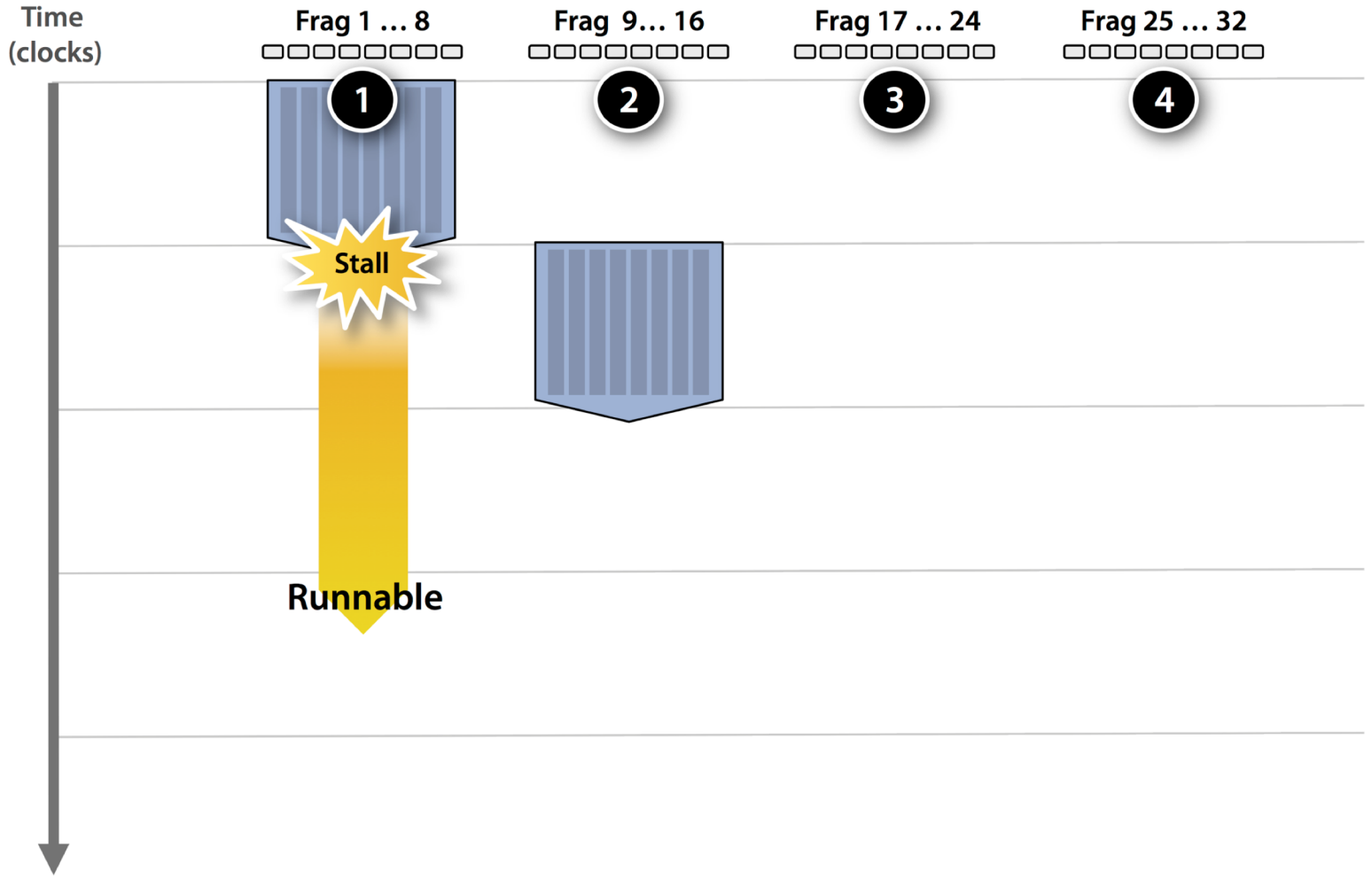
Hardware for Throughput Computing

- Hiding memory stalls



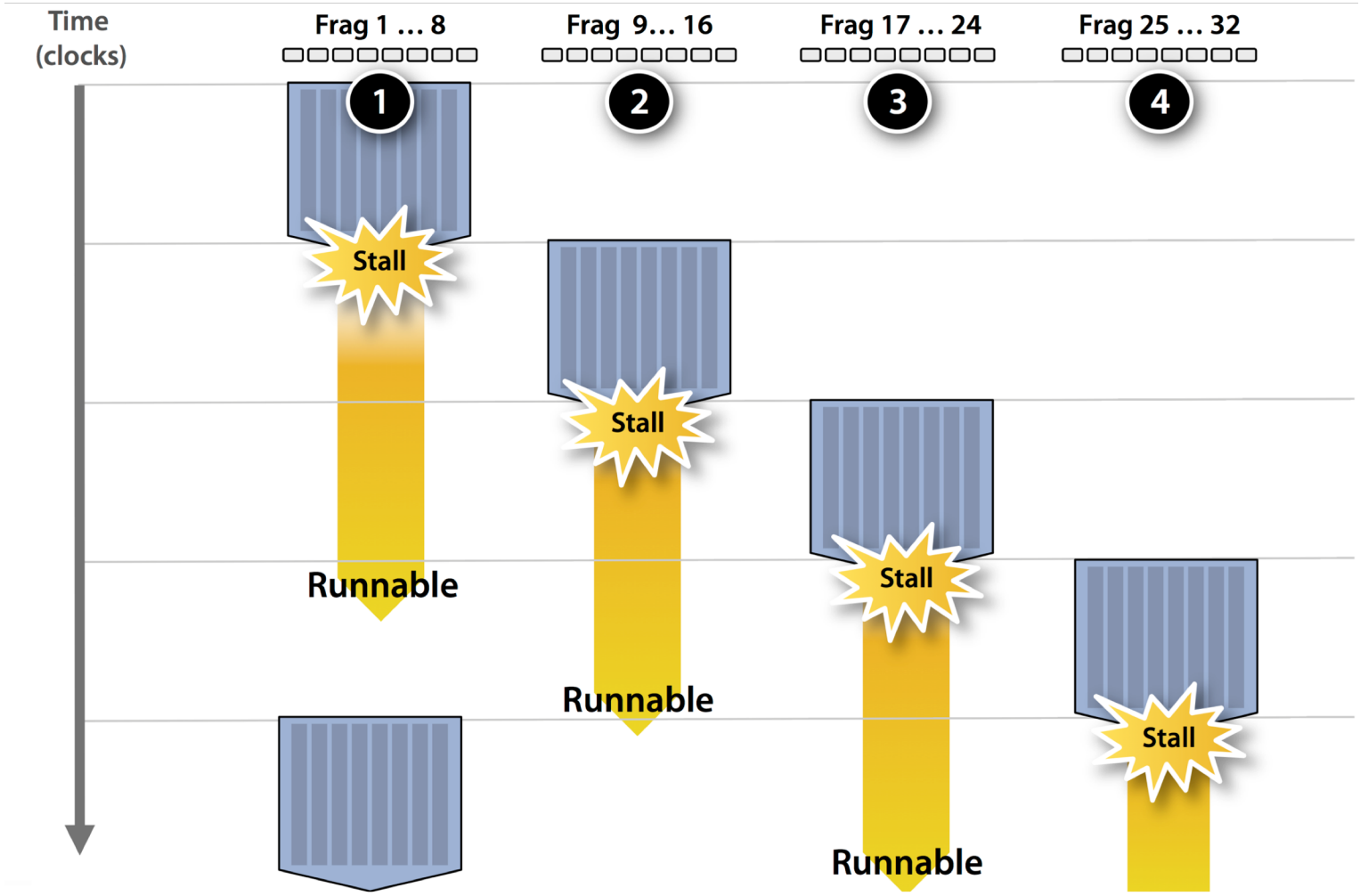
Hardware for Throughput Computing

- Hiding memory stalls



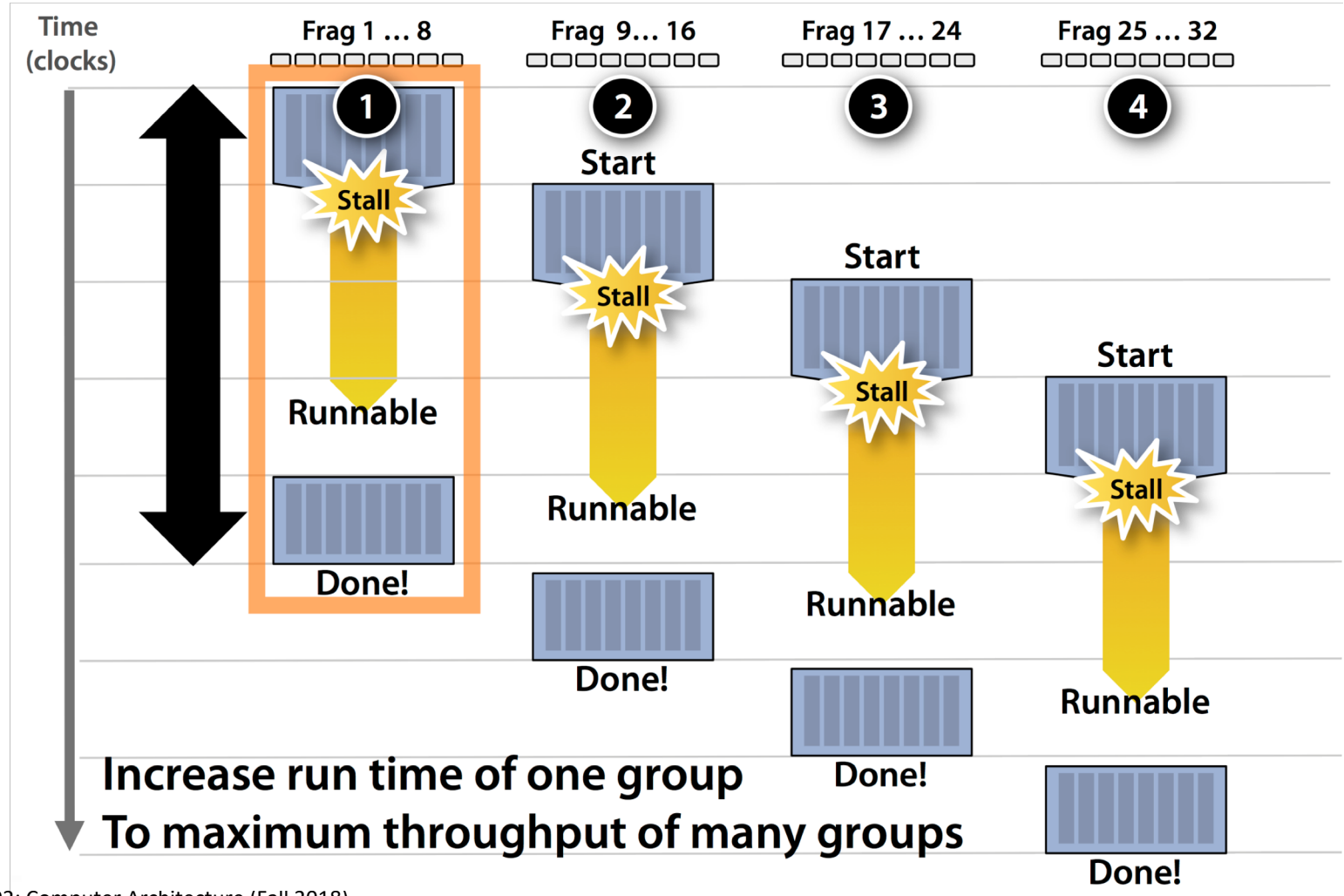
Hardware for Throughput Computing

- Hiding memory stalls



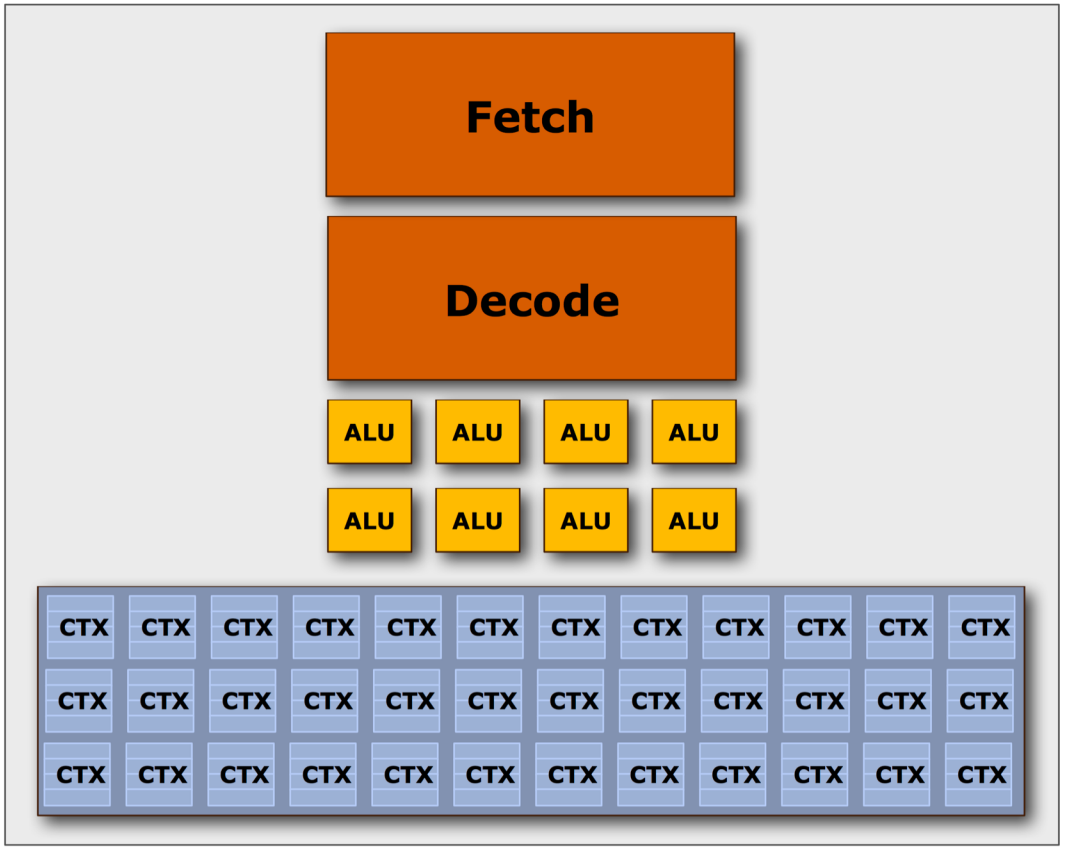
Hardware for Throughput Computing

- Hiding memory stalls



Hardware for Throughput Computing

- **How does it work?**
  - Small context – best latency hiding
  - Light register → usage more contexts → more elements in flight



Hardware for Throughput Computing

- **Summary of three key features of GPU**
  1. Use many “slimmed down cores” run in parallel

  2. Pack cores full of ALUs
     - By sharing instruction stream across groups of fragments
     - 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

Graphics Processing Units (GPUs)

- **Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance FP units**
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- **Over time, more programmability added (2001-2005)**
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- **Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations**
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

Graphics Processing Units (GPUs)

- **General-Purpose Processing on GPUs (GP-GPUs)**
  - In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
    - "Compute Unified Device Architecture"
    - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
  - Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
  - Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
  - This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics
    - Other vendors (AMD, Intel, ARM, etc.) have many similarities.

Graphics Processing Units (GPUs)

- **Simplified CUDA Programming Model**
  - Computation performed by a very large number of independent small scalar threads (CUDA threads or microthreads) grouped into thread blocks.

```
void daxpy(int n, double a, double*x, double*y) // C DAXPY
{  for (int i=0; i<n; i++)
       y[i] = a*x[i] + y[i]; }

// CUDA version
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__  // Piece run on GPU.
void daxpy(int n, double a, double*x, double*y)
{  int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n) y[i]=a*x[i]+y[i]; }
```

Summary

- **Three key features of GPUs**
  - A lot of simple cores
  - Full of ALUs by sharing instruction stream among fragments
  - Hiding memory stalls by zero-latency context switching

- **Compute Unified Device Architecture (CUDA)**
  - De-facto standard for GPU programming
  - Strengths: easy to program, success with complex applications, fast memory close to processors (registers + shared memory)
  - Weaknesses: better tools needed (profiling, memory layout, etc.), difficult to find optimal values for CUDA execution parameters (grid/block configurations, on-chip memory resources, etc.)