Introduction to OpenCL

Justin Hensley & Derek Gerstmann

SIGGRAPHASIA2010
SEOUL

Agenda for this tutorial

Introduction

Provide an understand of how GPUs work

(why is OpenCL designed the way it is)

Overview of OpenCL 1.0 specification

Slides will be posted at http://sa10.idav.ucdavis.edu/

Agenda for afternoon course

OpenCL Development Tips

OpenGL/OpenCL Interoperability

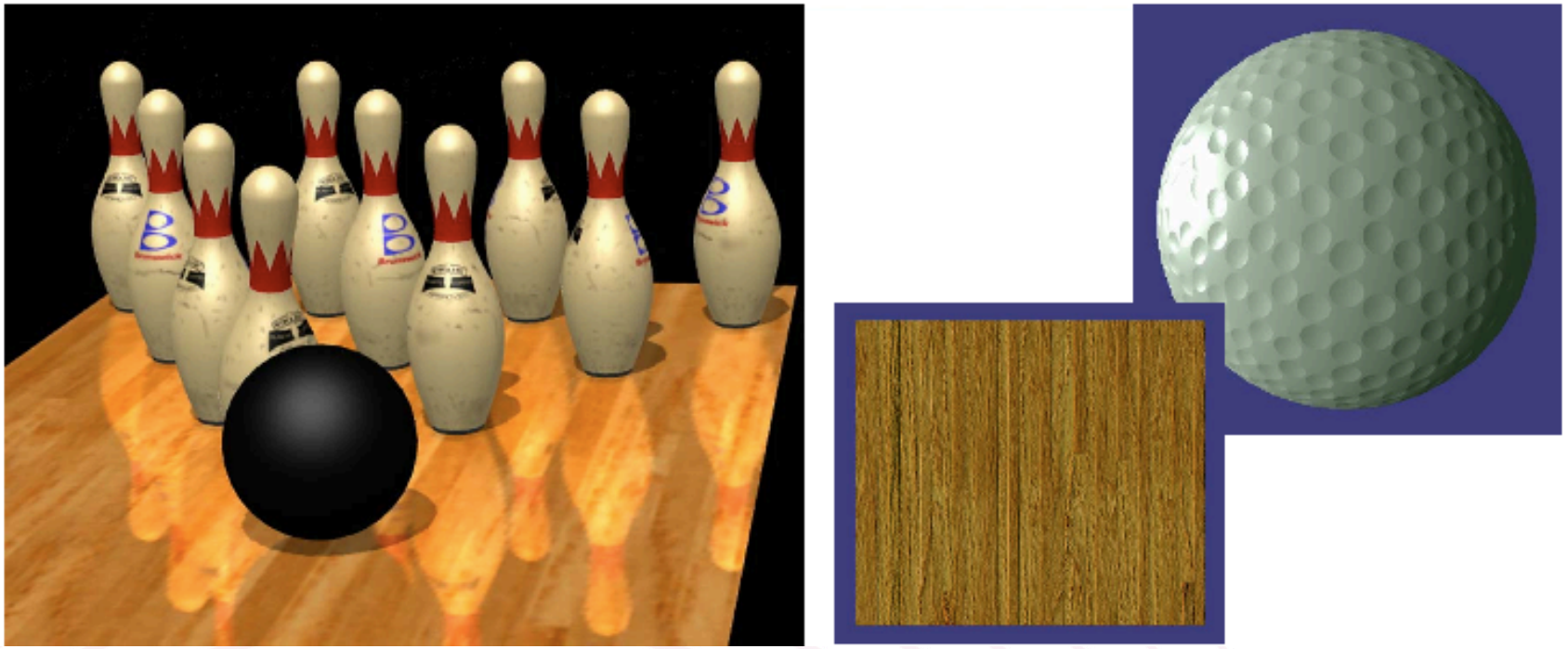
Rigid body particle simulation

Bullet cloth

Galaxy n-body simulation

Visualization using OpenCL

State of the Art, 1998



Marc Olano, Anselmo Lastra: A Shading Language on Graphics Hardware: The Pixelflow Shading System. SIGGRAPH 1998

Enablers for Throughput Computing

✔High Level Programming Language

Unfortunately, not Cost Effective!

This was a prototype
Meant to test concept
Hundreds of chips

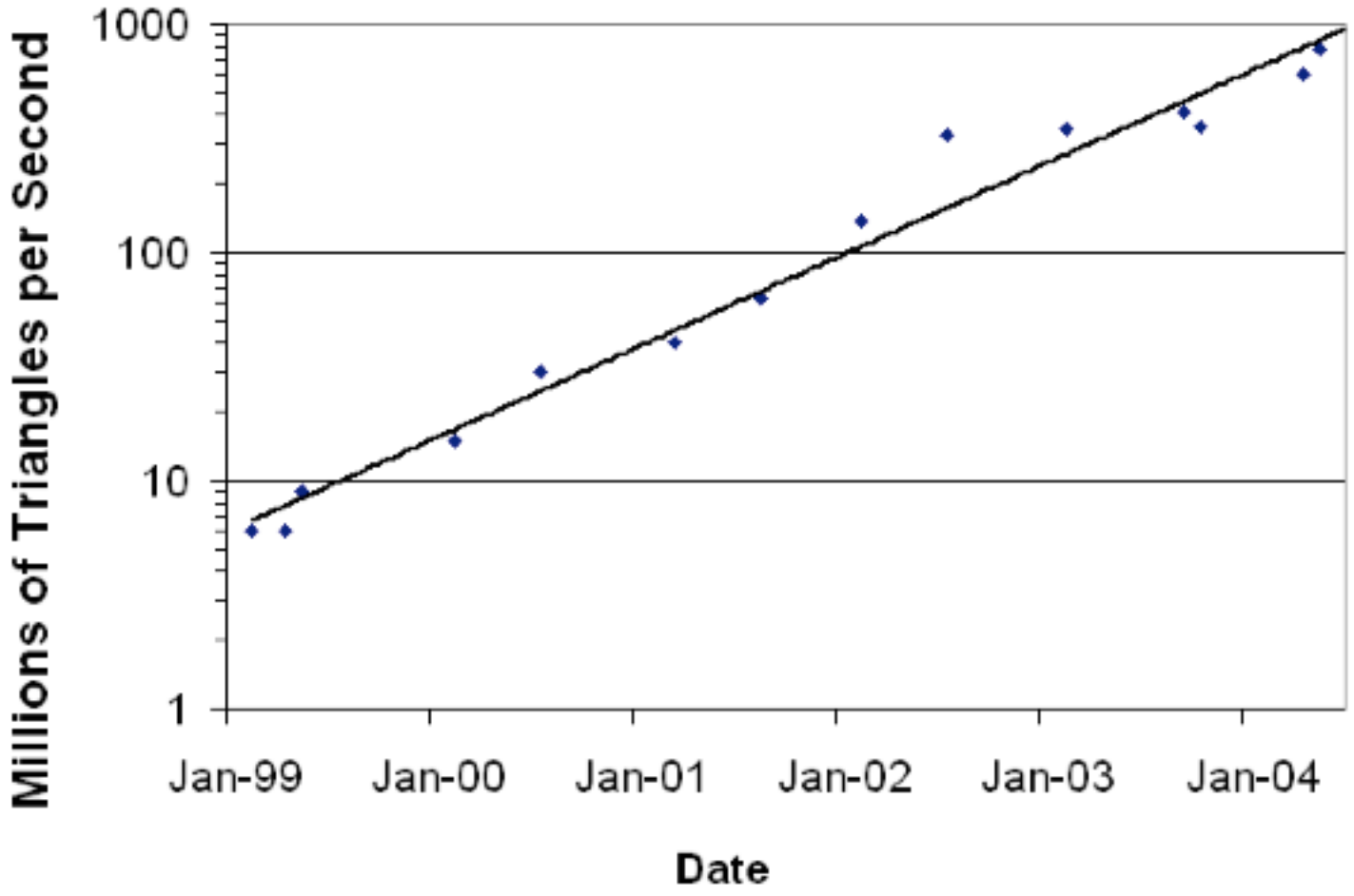
The Single Chip Revolution

As in CPUs, going to a single chip made for cost efficiency (as did the fact that PC graphics is a volume market)

Enablers for Throughput Computing

✔High Level Programming Language
✔Excellent Cost Performance

Skyrocketing Performance



Why Such a Fast Rate of Increase?

Architectures meant for **throughput**, not for **single thread**

- Not complexity of out-of-order issue
- Nor of multiple issue per core

Rendering is **latency tolerant**

Graphics generates many **parallelizable** tasks

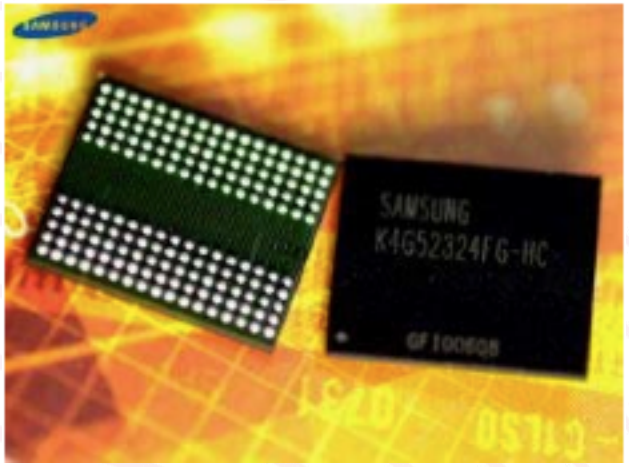
- Therefore much easier to scale

Enablers for Throughput Computing

- ✔High level language programming
- ✔Excellent cost performance
- ✔Rapid increases in performance over time

Memory Bandwidth

GPUs have driven high-performance memory market
GDDR series of DRAMs



Enablers for Throughput Computing

- ✔High Level Programming Language
- ✔Excellent Cost Performance
- ✔High Memory Bandwidth

Enablers for Throughput Computing

- ✔High Level Programming Language
- ✔Excellent Cost Performance
- ✔Rapid increases in performance over time
- ✔High Memory Bandwidth

What's Left?

Major stumbling block: special purpose nature of graphics hardware

Let's look at the ***Wheel of Reincarnation***

Wheel of Reincarnation

CACM paper by Myer and Sutherland in **1968**

Tongue in cheek look at design of a display processor

In those days it was for vector graphics

Authors examine their design process

Let's look and see whether it's familiar

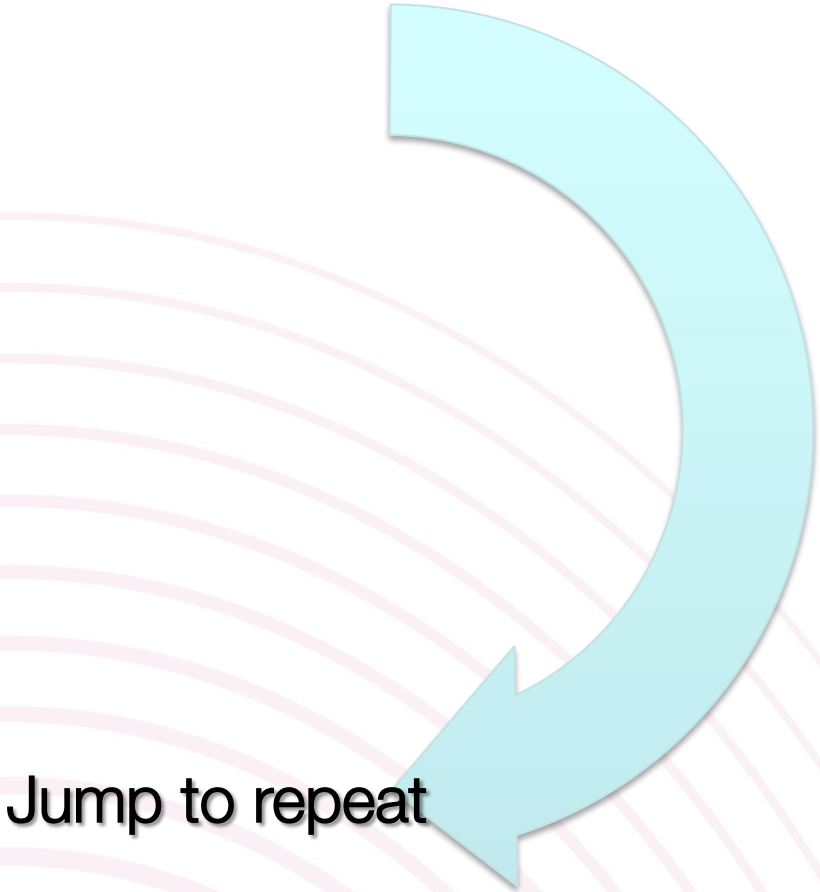
A Spin Around the Wheel

Display Channel for Points

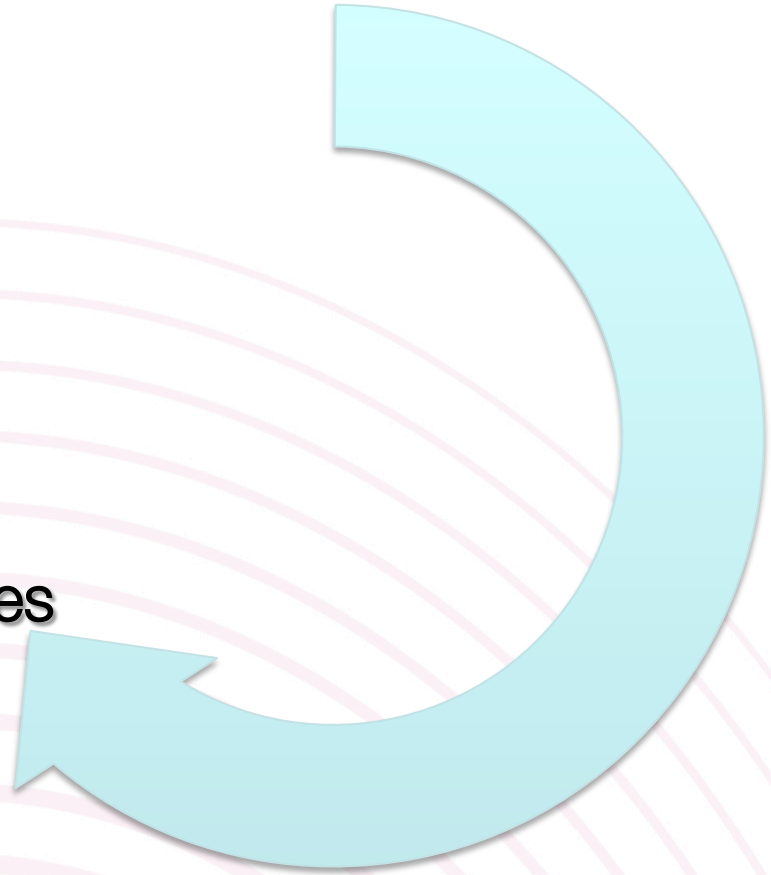
A Spin Around the Wheel

Line drawing hardware

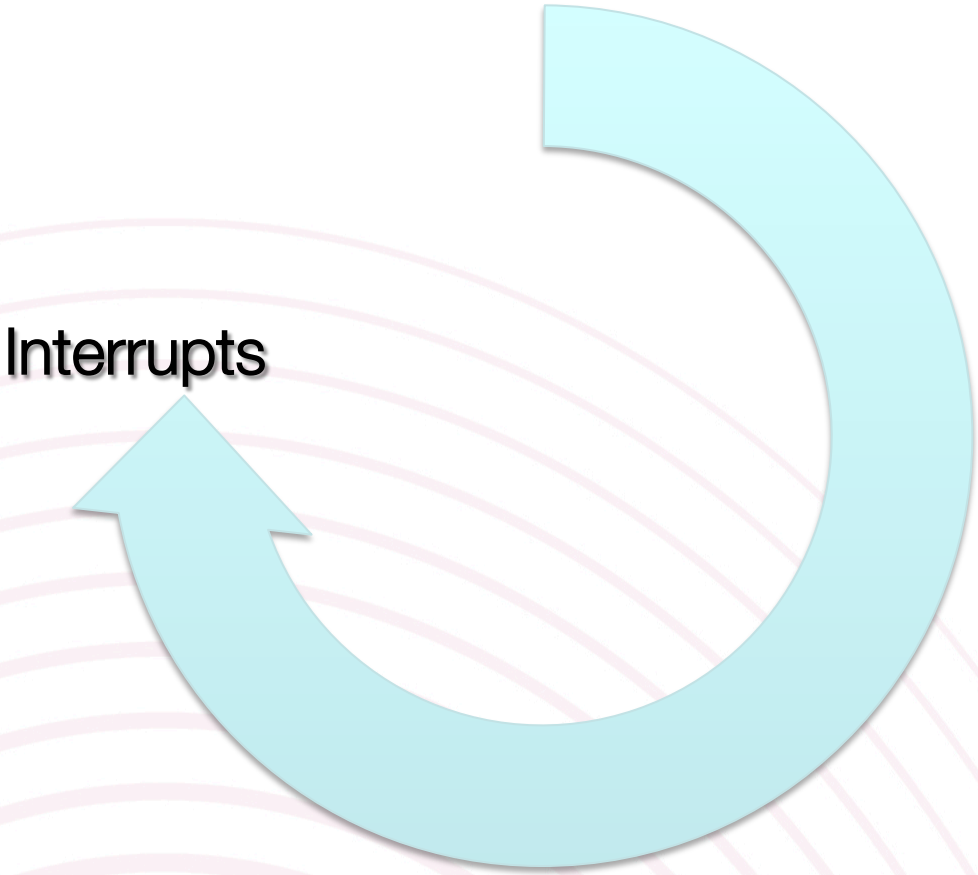
A Spin Around the Wheel



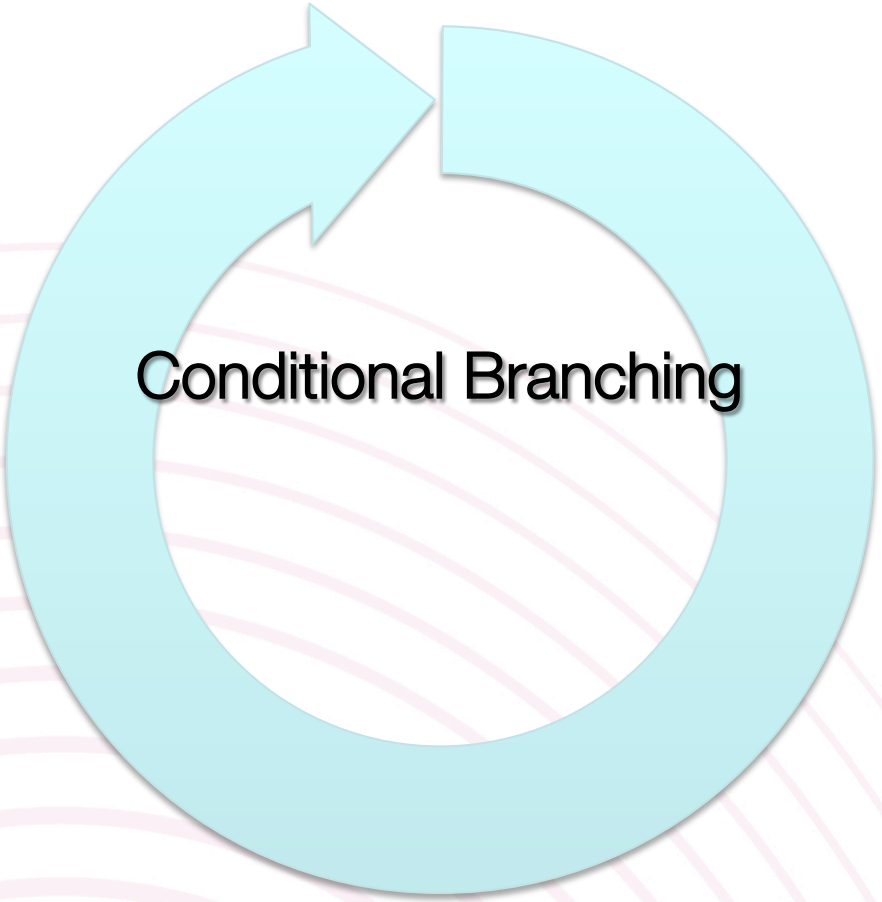
A Spin Around the Wheel



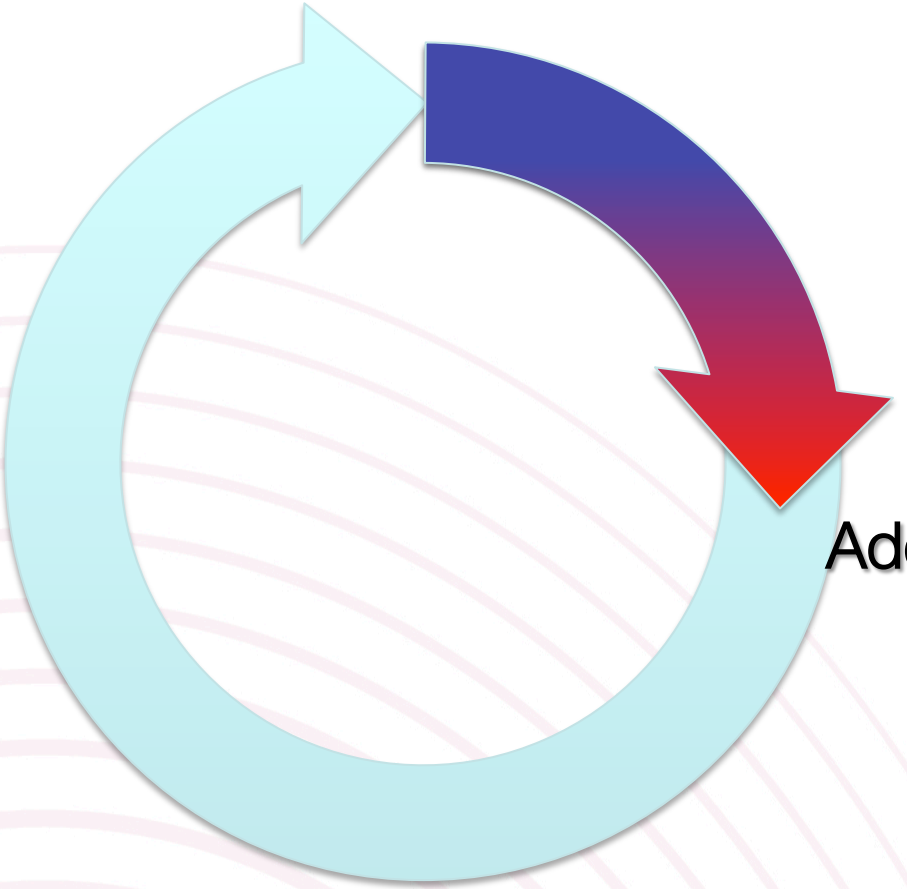
A Spin Around the Wheel



A Full Spin



Maybe a Little More?



Add A Channel?

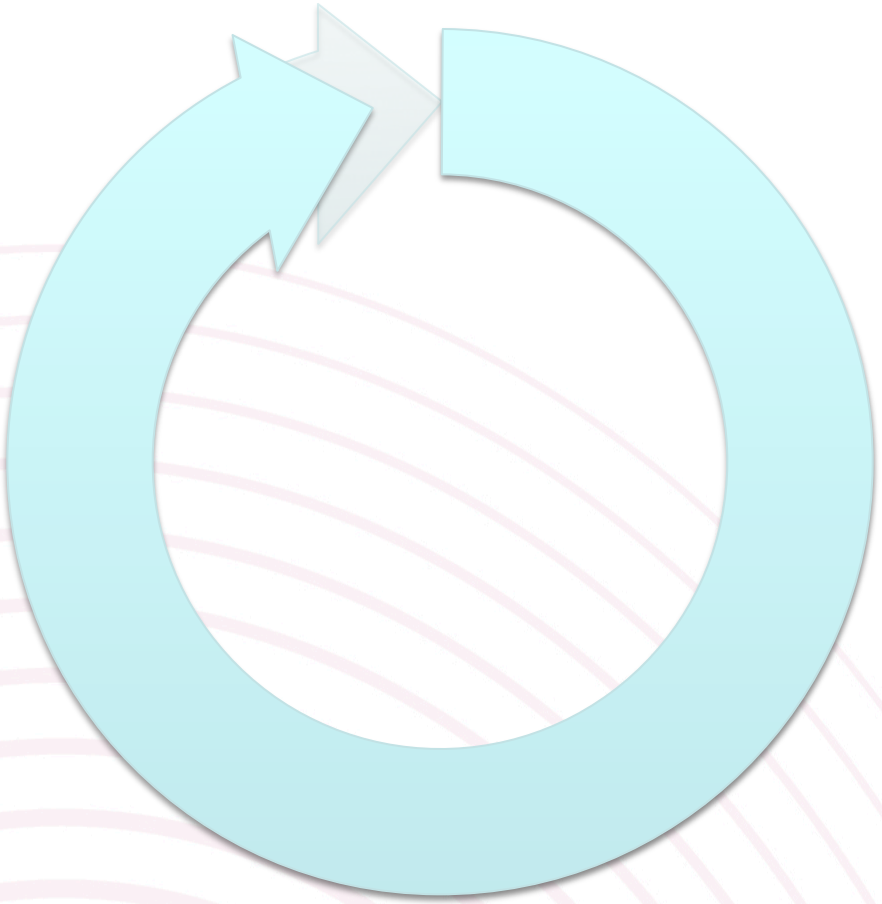
State of the Art Today

Commercial, high-end GPUs (such as from AMD and NVIDIA) have become very general

- Unified shader
- Some special purpose circuits: texturing, rasterization, blending, etc.

Where are we on the wheel?

Existing Graphics Processors



Enablers for Throughput Computing

✔High Level Programming Language
✔Excellent Cost Performance
✔Rapid increases in performance over time
✔High Memory Bandwidth
✔General Purpose

✔**Ready for Use!**

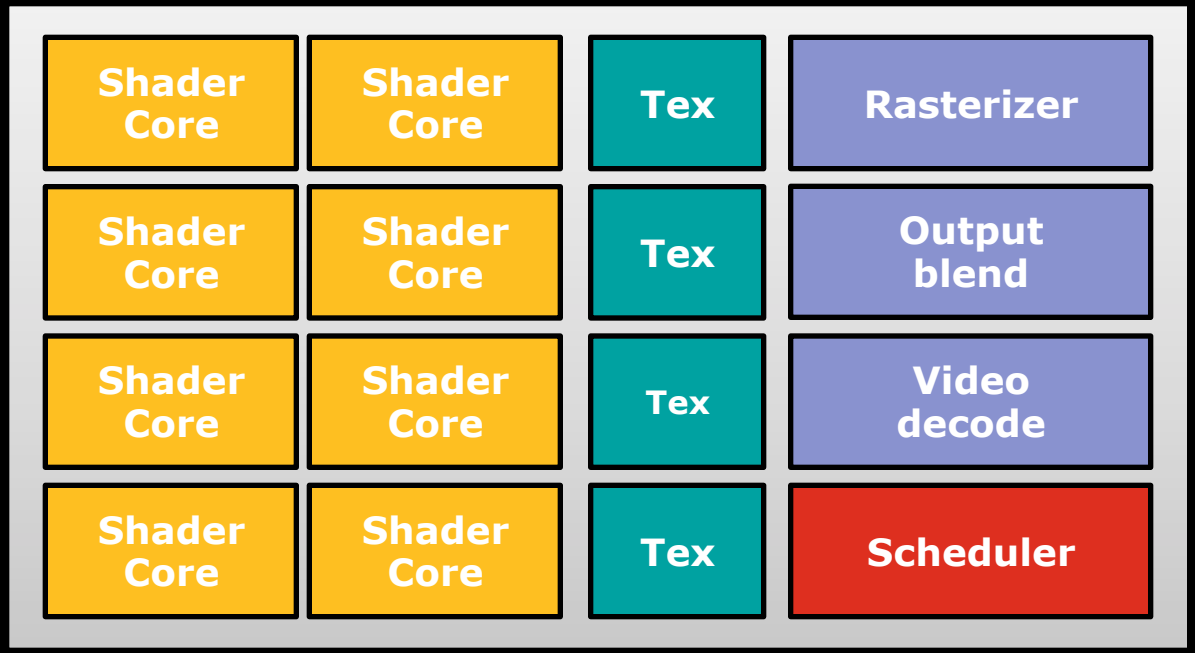
Agenda

- To provide a general background to modern GPU architectures
- To place the GPU designs in context:
  - With other types of architecture
  - With other GPU designs
- To give an idea of why certain optimizations become necessary on such architectures and why the architectures are designed in that way

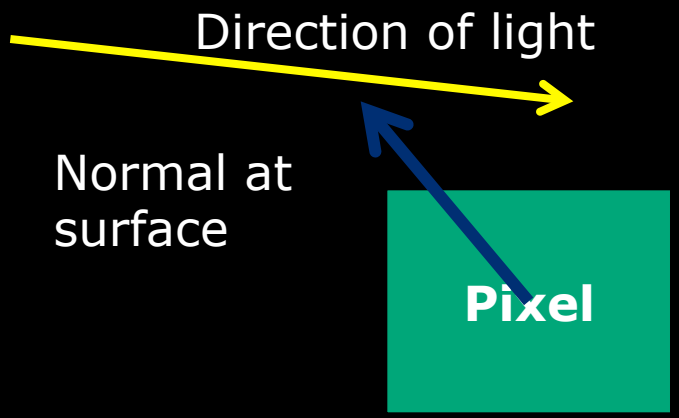
What is a GPU?

In a nutshell

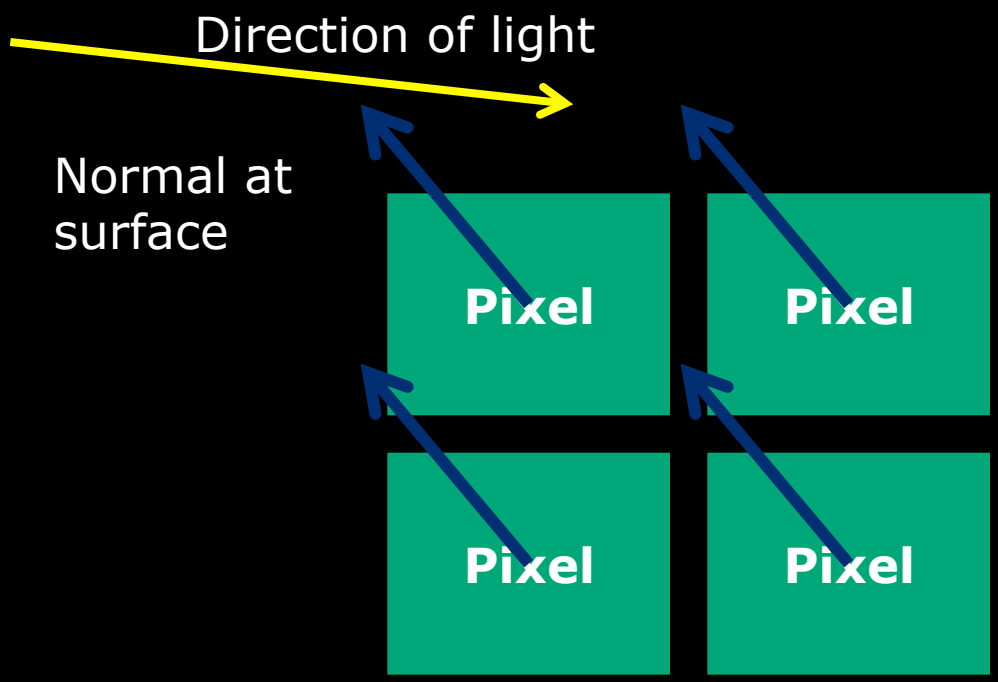
- The GPU is a multicore processor optimized for graphics workloads



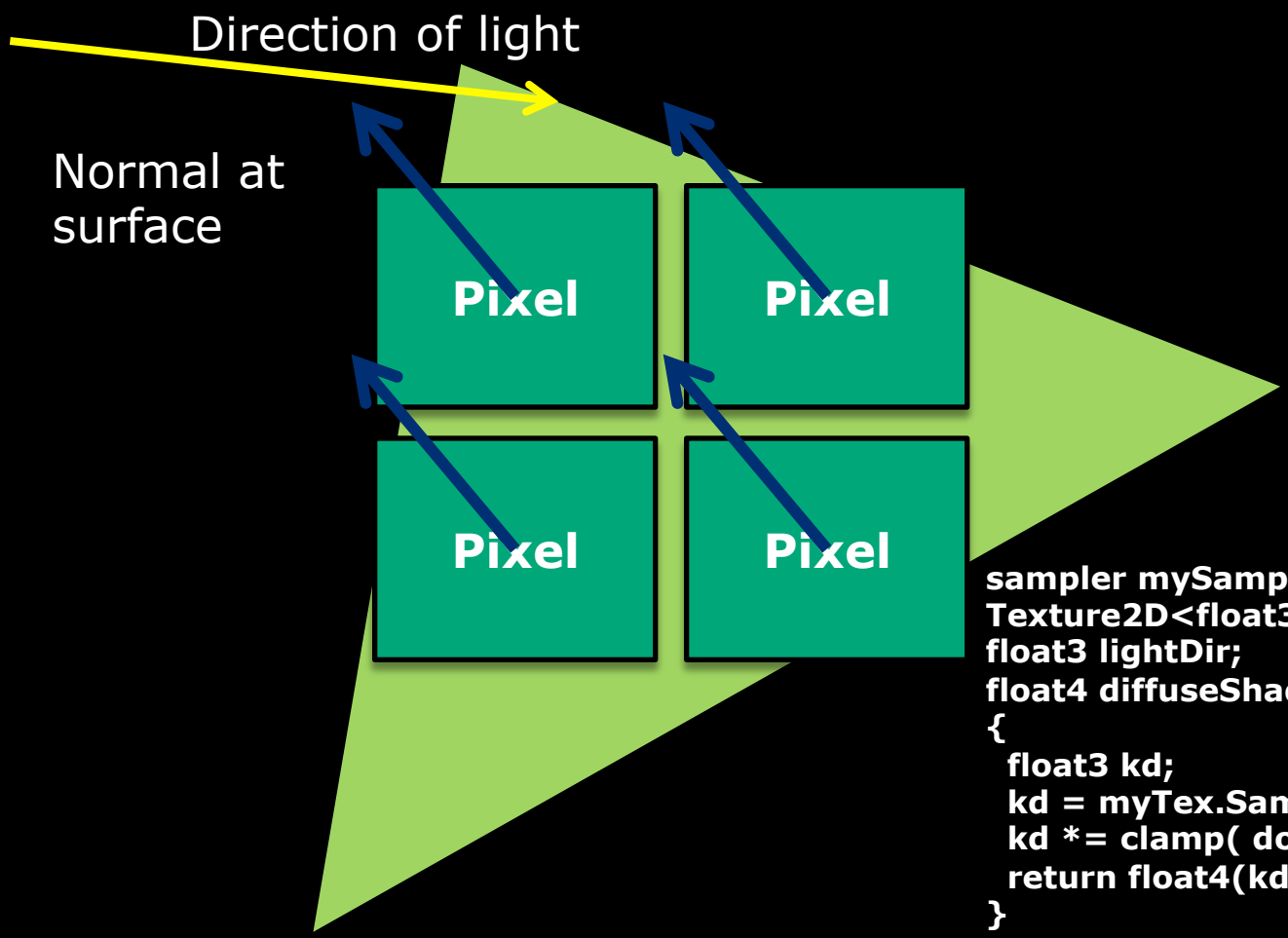
Processing pixels



Processing pixels

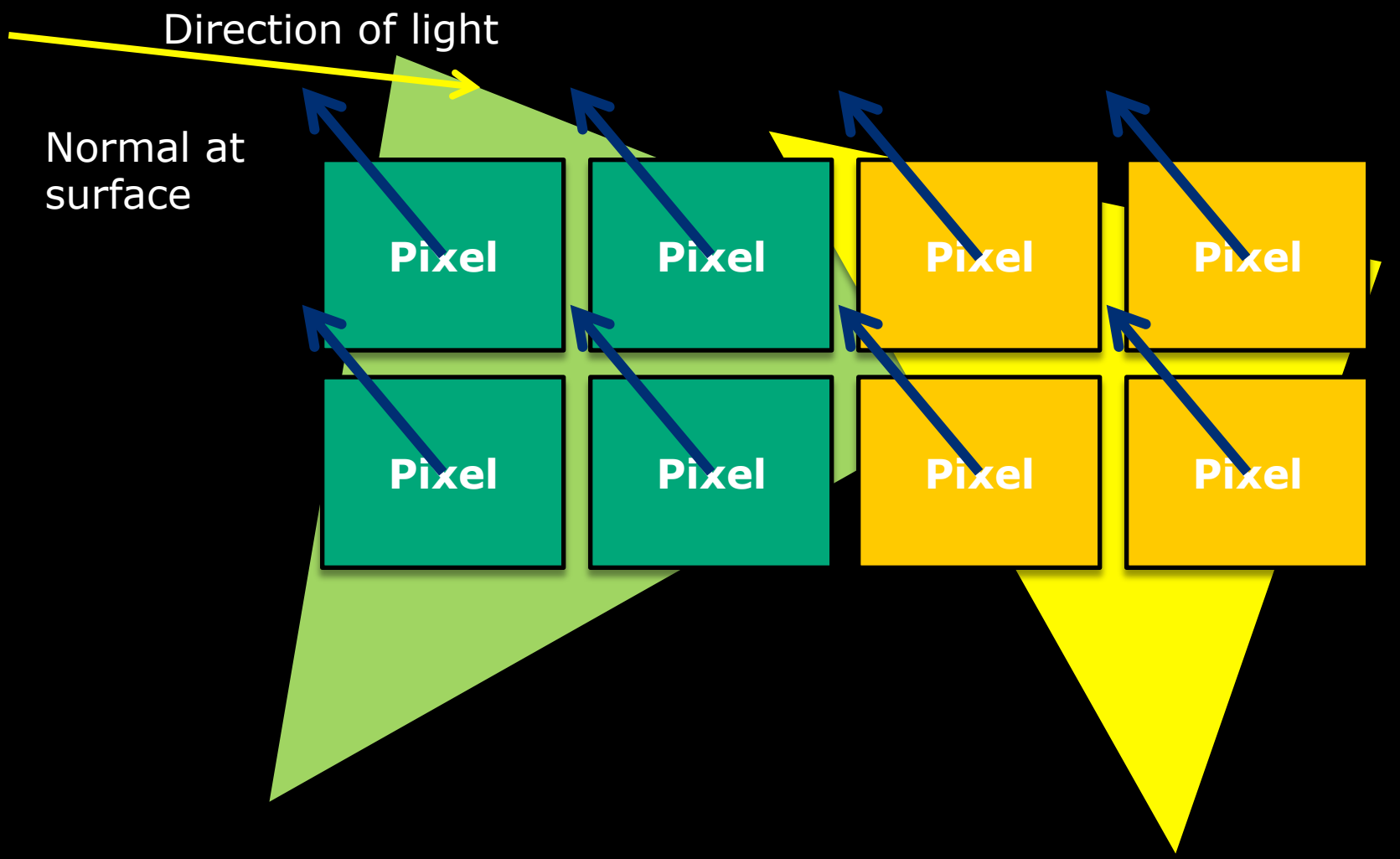


Processing pixels




```
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);
}
```

Processing pixels

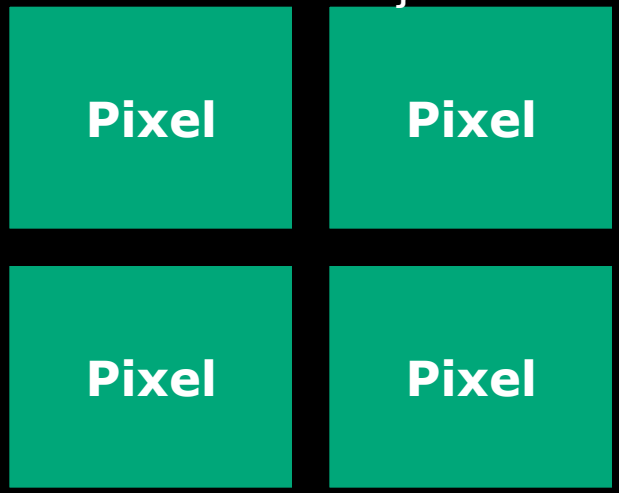


SIMD execution and its implications

SIMD pixel execution

```
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);
}
```

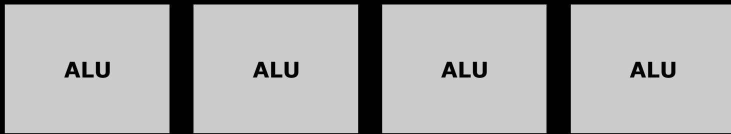
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```




```
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);
}
```

```
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);
}
```

Branches that diverge



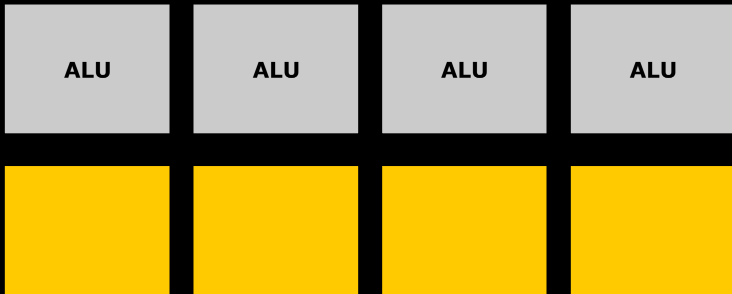

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Branches that diverge




```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Branches that diverge

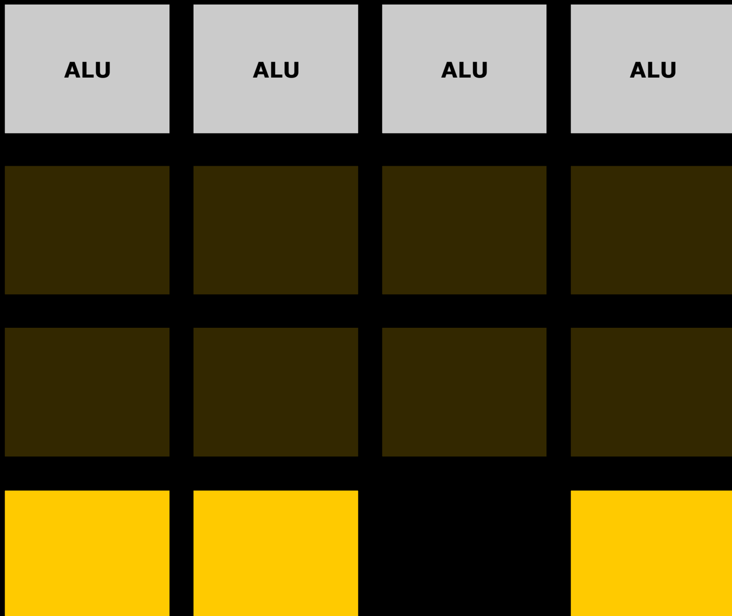

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Branches that diverge



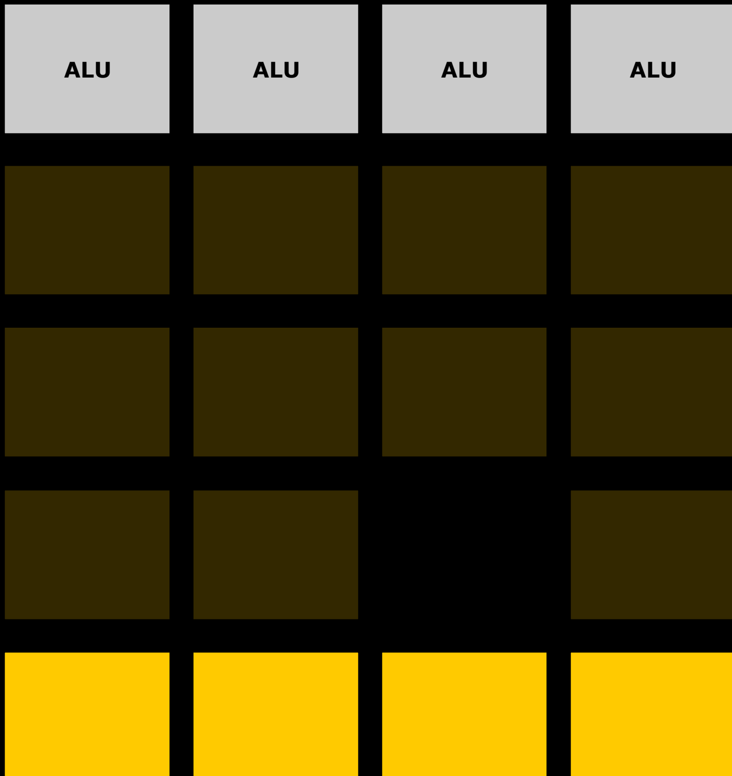

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
       output = threshold;
    else
       output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
       output = threshold;
    else
       output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
       output = threshold;
    else
       output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
       output = threshold;
    else
       output = brightness;
    return output;
}
```

Branches that diverge



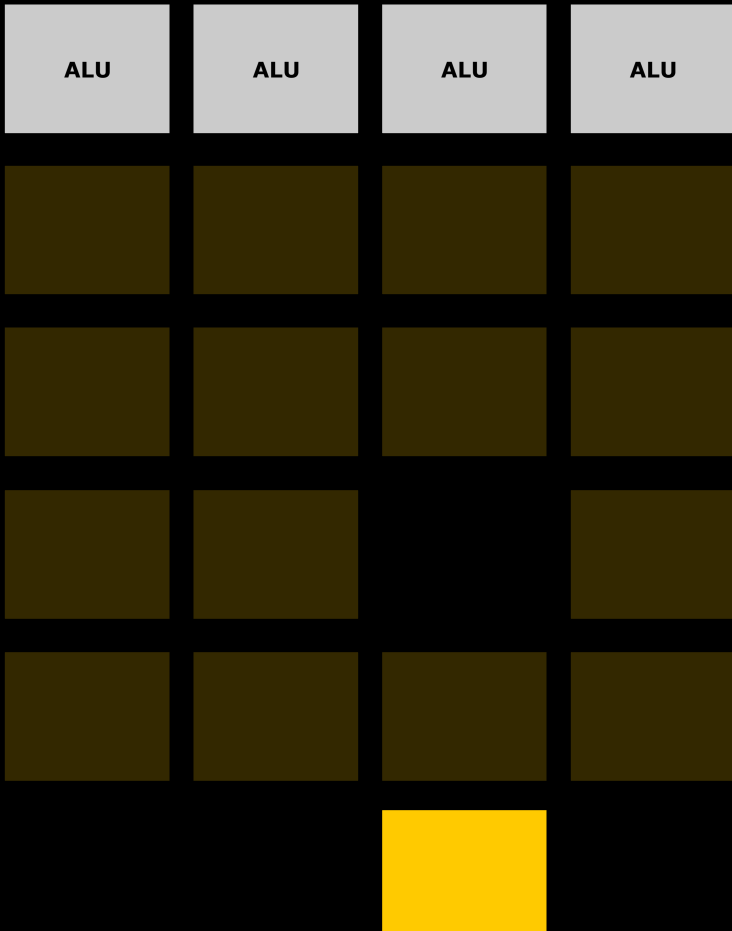

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Branches that diverge



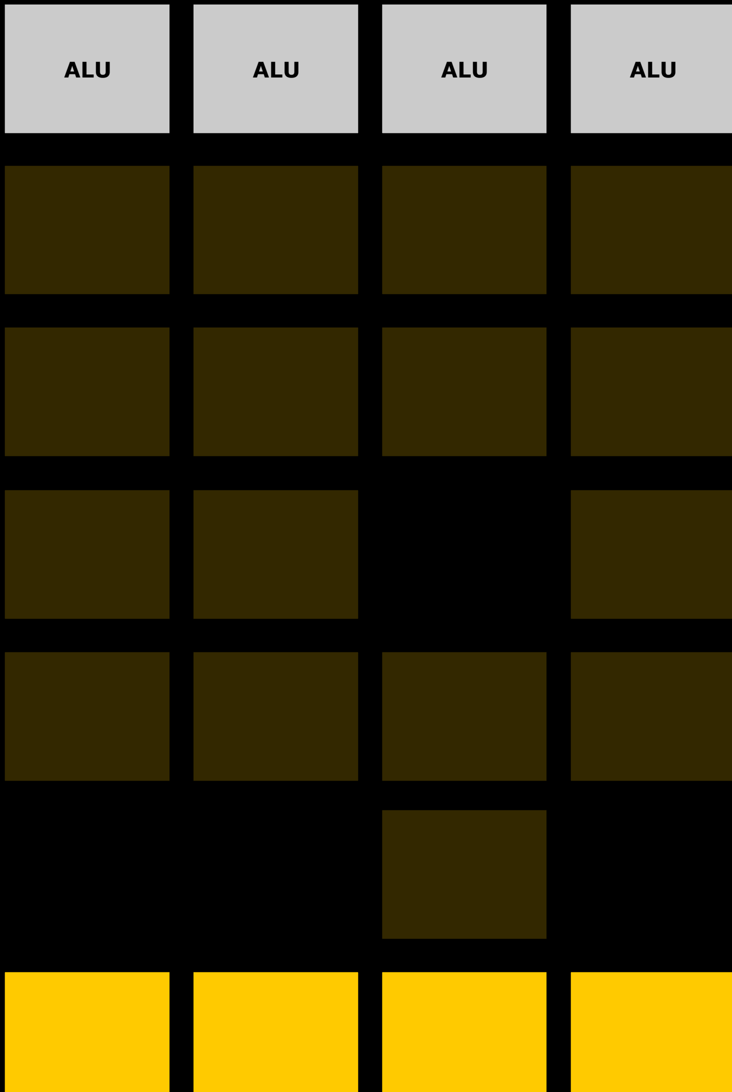

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Branches that diverge




```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```


```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

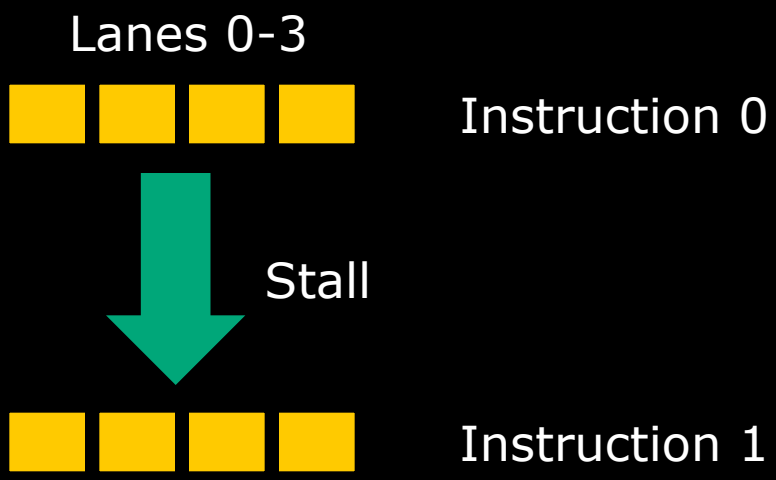
```
sampler mySamp;
Buffer<float> myTex;

float diffuseShader(
  float threshold, float index)
{
    float brightness = myTex[index];
    float output;
    if( brightness > threshold )
        output = threshold;
    else
        output = brightness;
    return output;
}
```

Why does this matter for Compute?

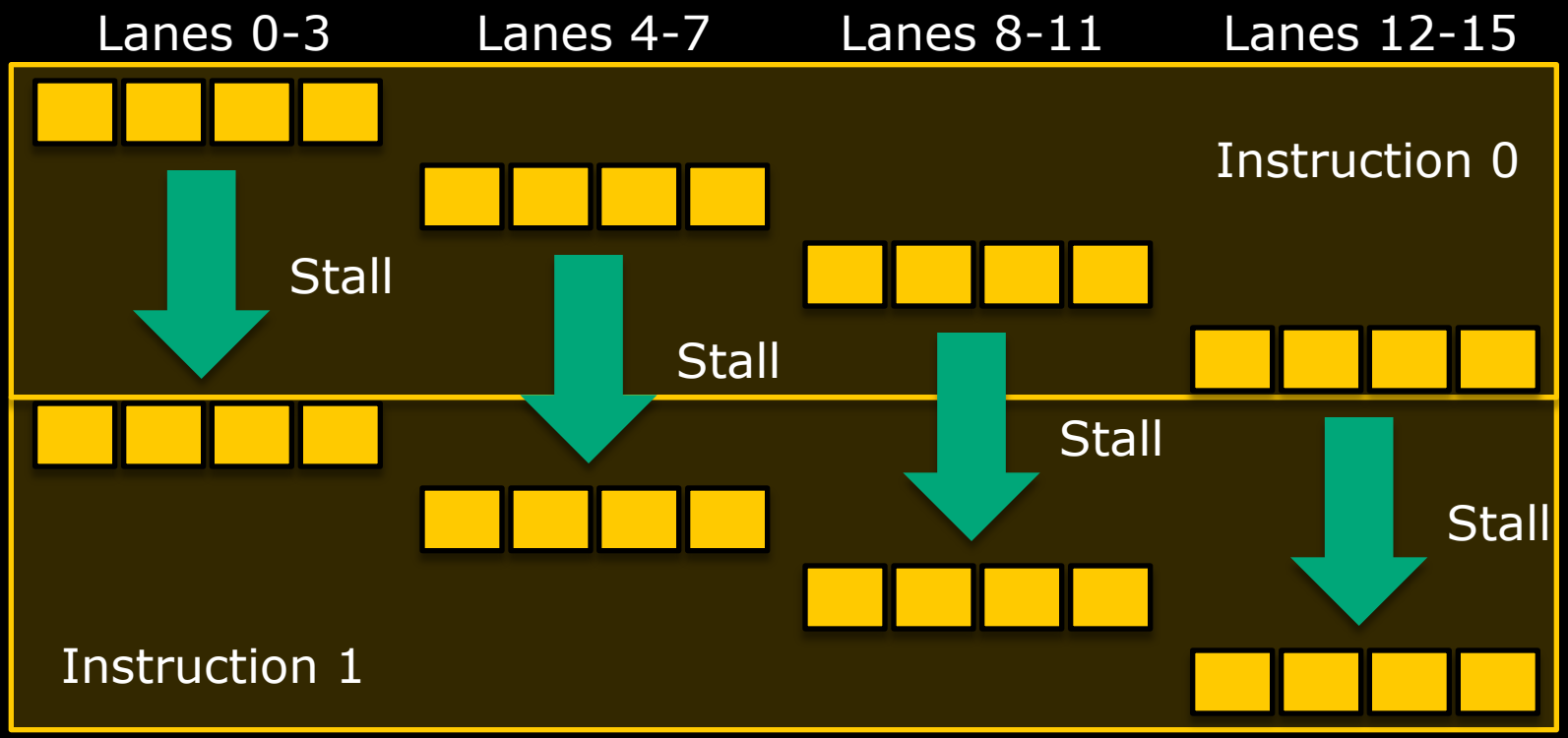
- Graphics code traditionally has relatively short shaders on large triangles
  - The level of branch divergence overall will not be high
- With graphics code you can not necessarily control it
  - SIMD batches are constructed by the hardware depending on the scene properties.
- For OpenCL code you are defining your execution space
  - You choose what work is performed by which work item
  - You choose how to structure your algorithm to avoid this divergence

Throughput execution and latency hiding

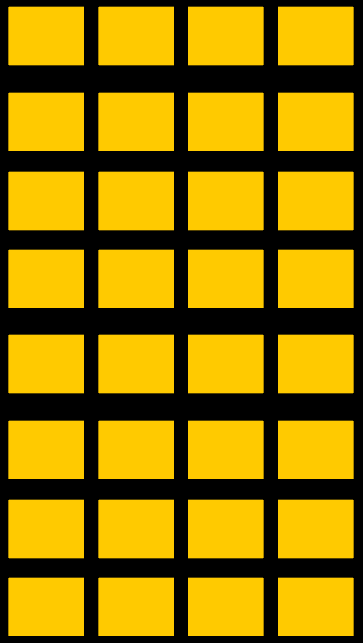
Covering pipeline latency



Covering pipeline latency: logical vector

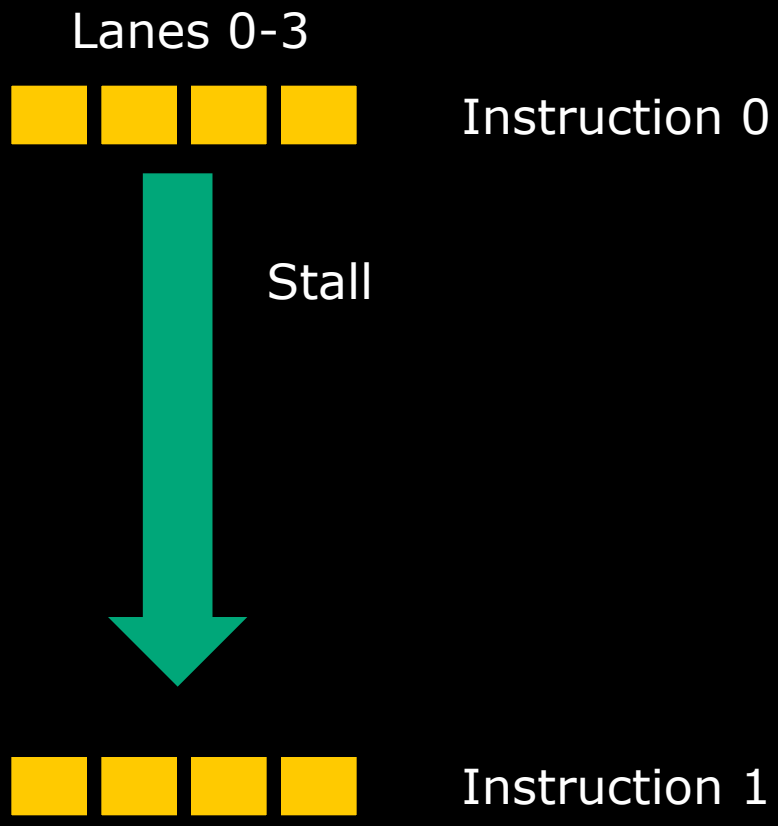


Covering pipeline latency: ALU operations

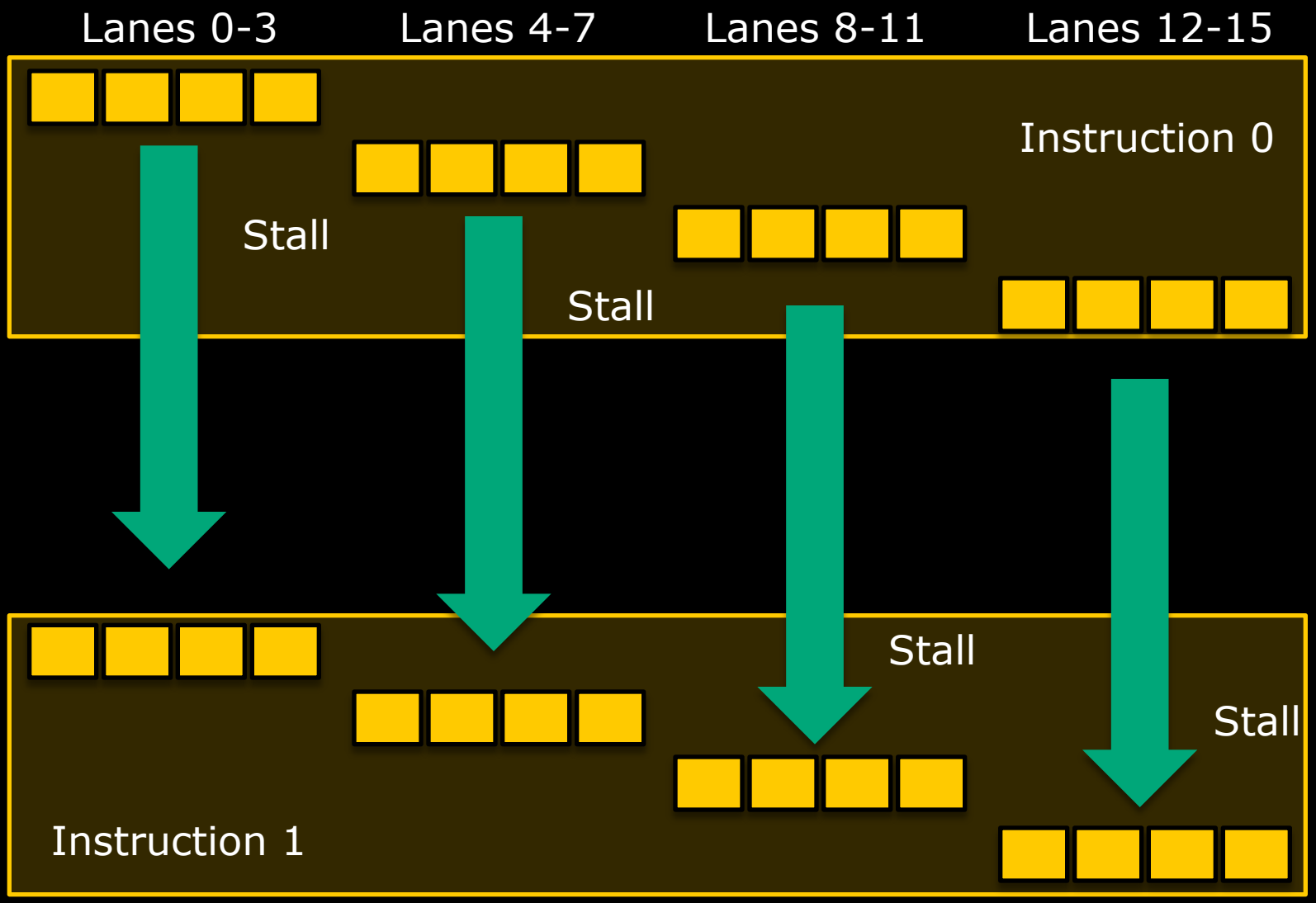


Lanes 0-3	Instruction 0
Lanes 4-7	Instruction 0
Lanes 8-11	Instruction 0
Lanes 12-15	Instruction 0
Lanes 0-3	Instruction 1
Lanes 4-7	Instruction 1
Lanes 8-11	Instruction 1
Lanes 12-15	Instruction 1

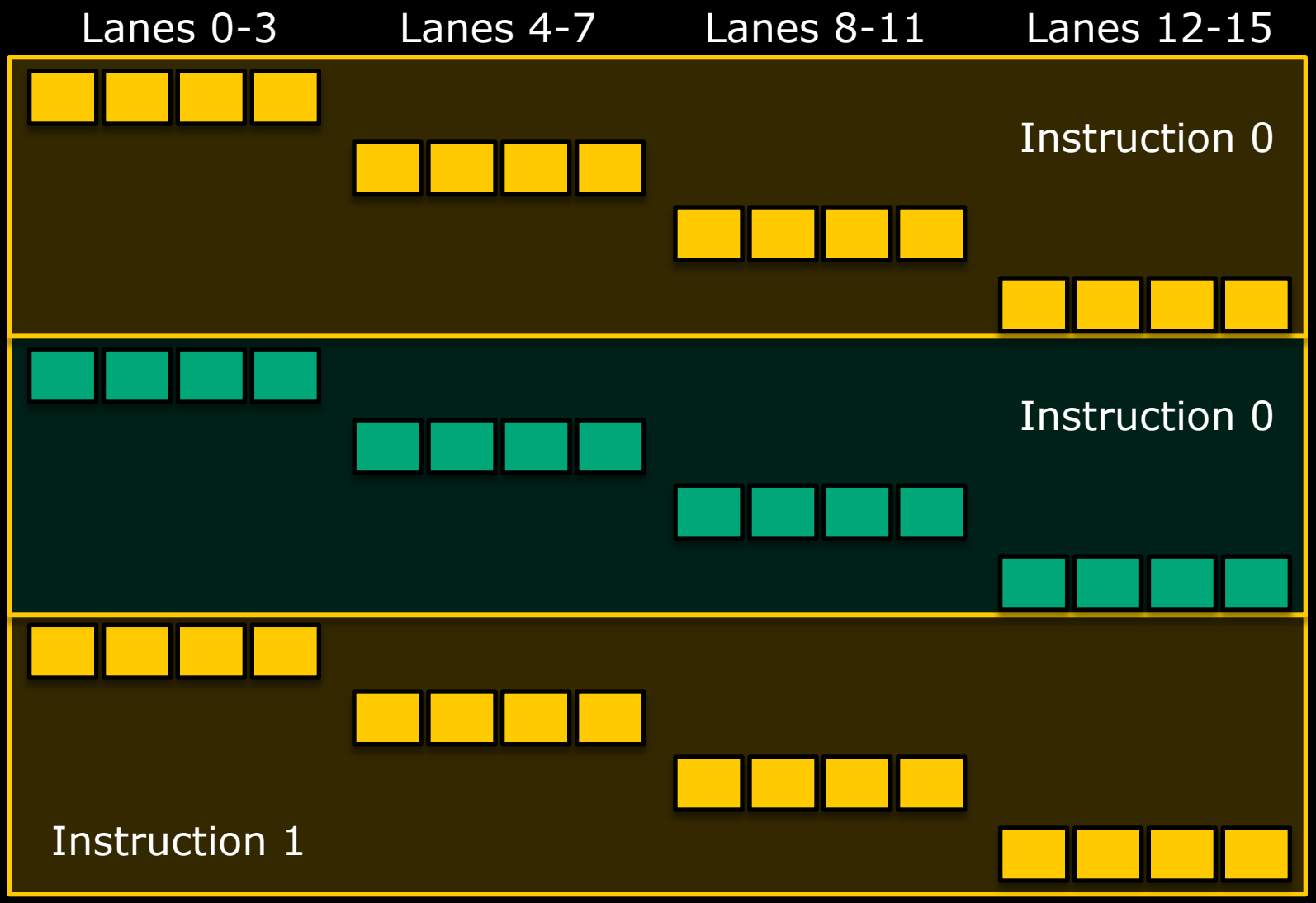
Covering memory latency



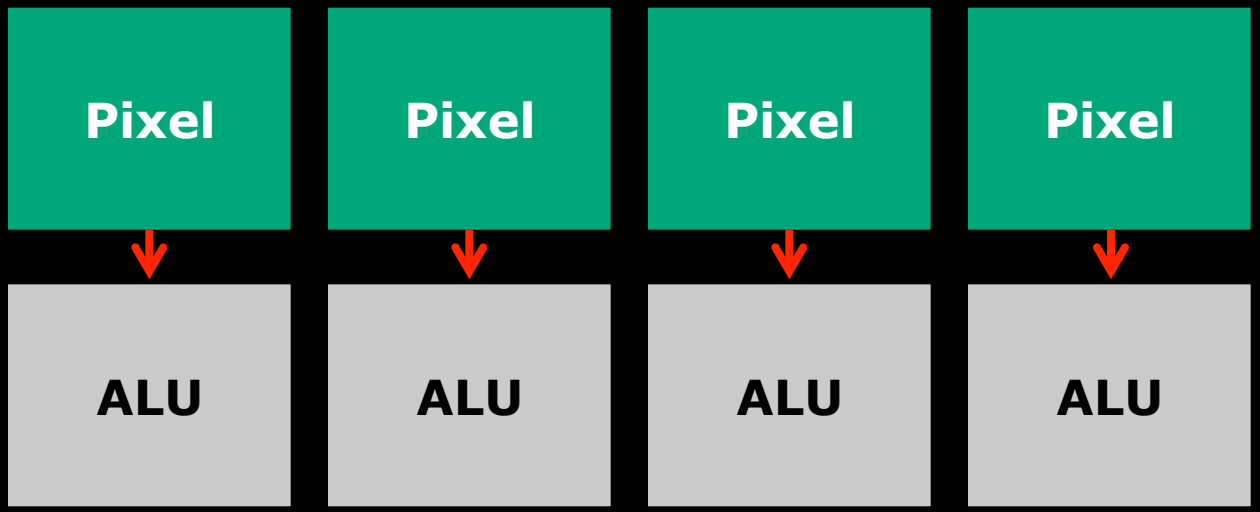
Covering memory latency: we still stall



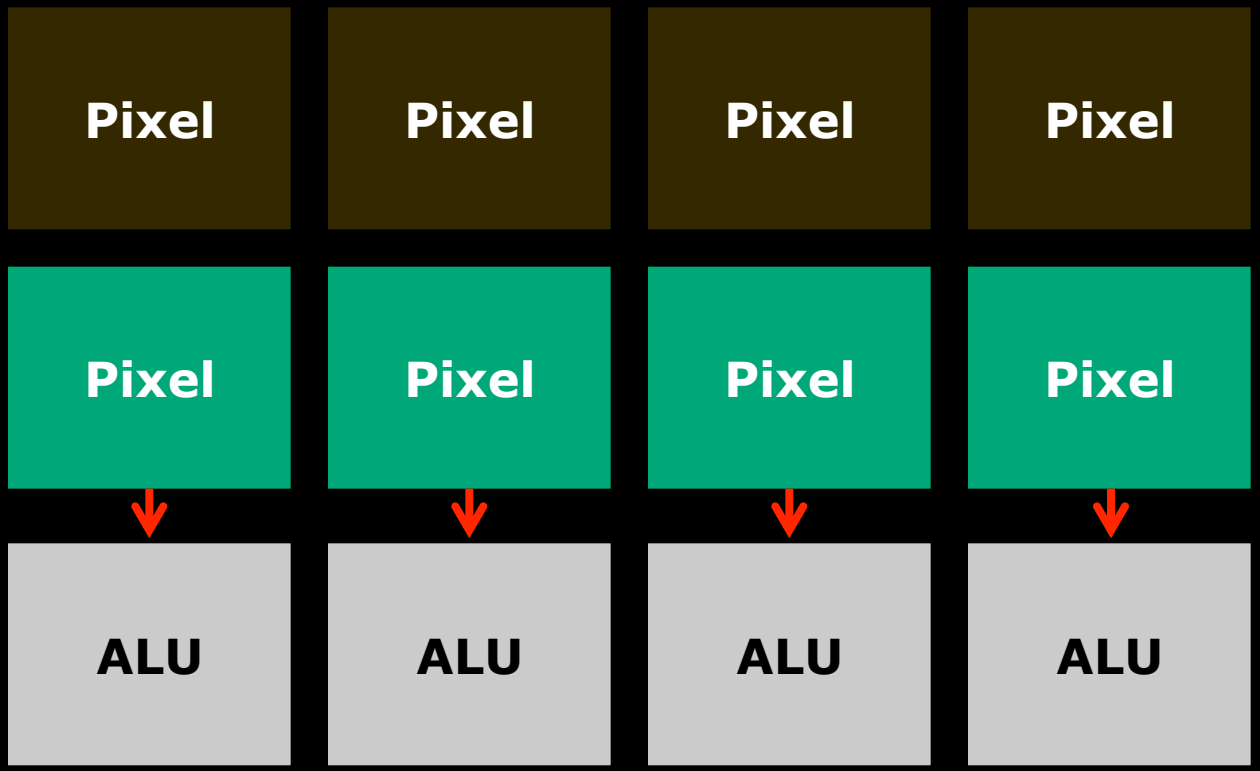
Covering memory latency: another wavefront



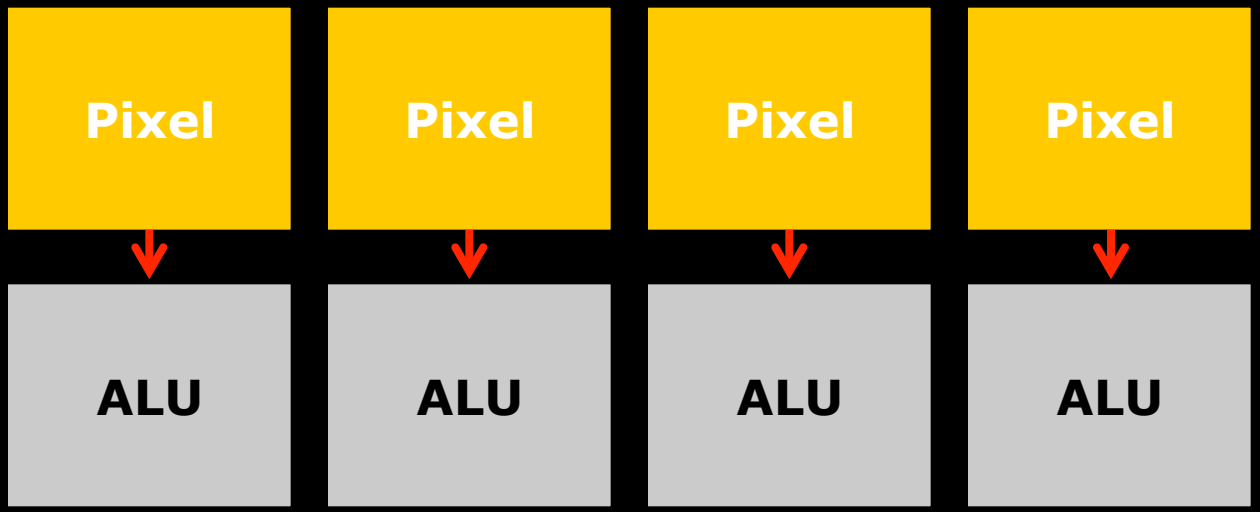
Latency hiding in the SIMD engine



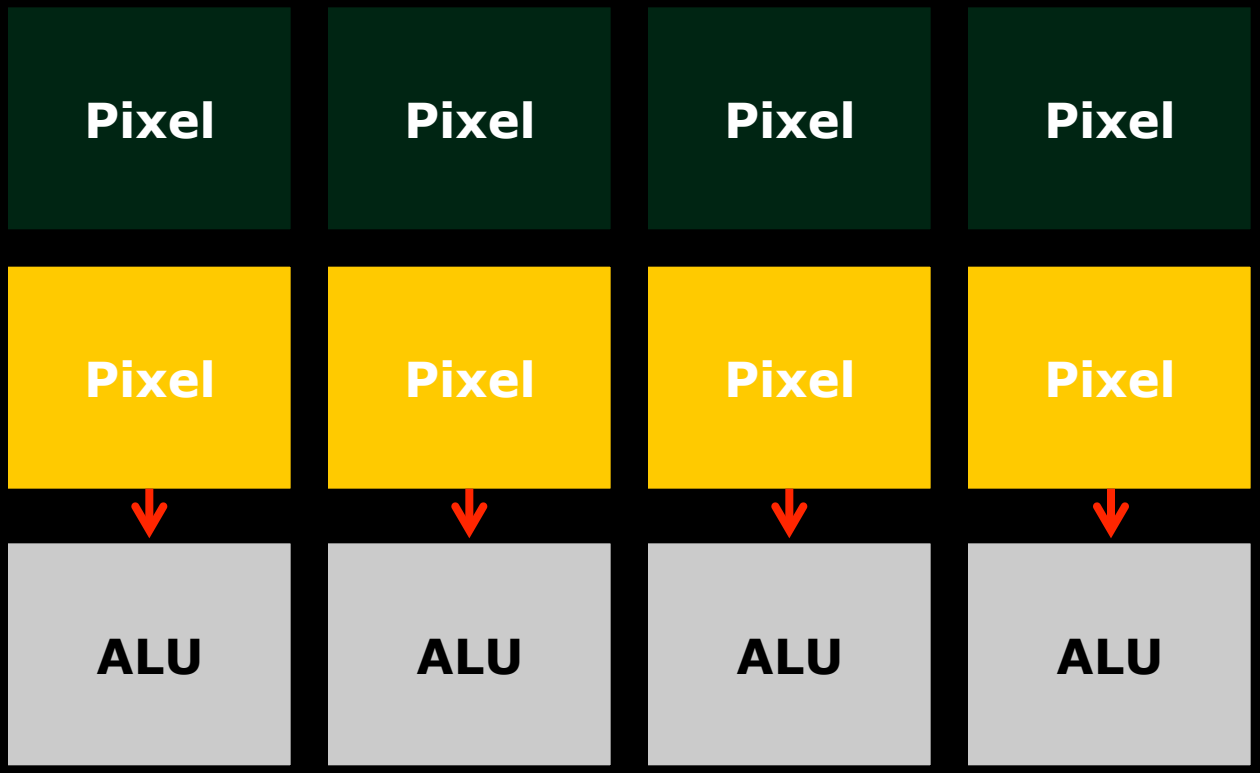
Latency hiding in the SIMD engine



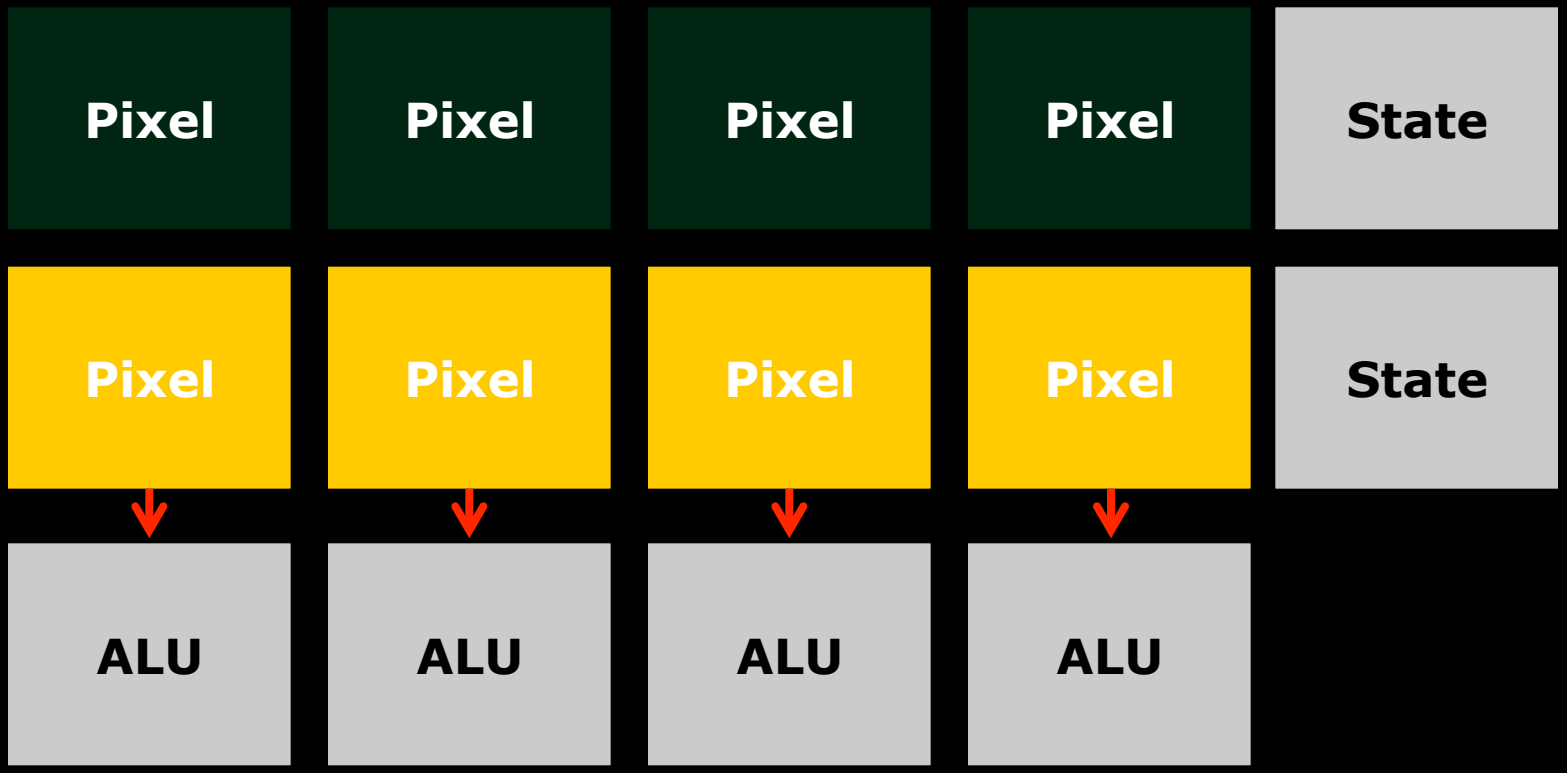
Latency hiding in the SIMD engine



A throughput-oriented SIMD engine



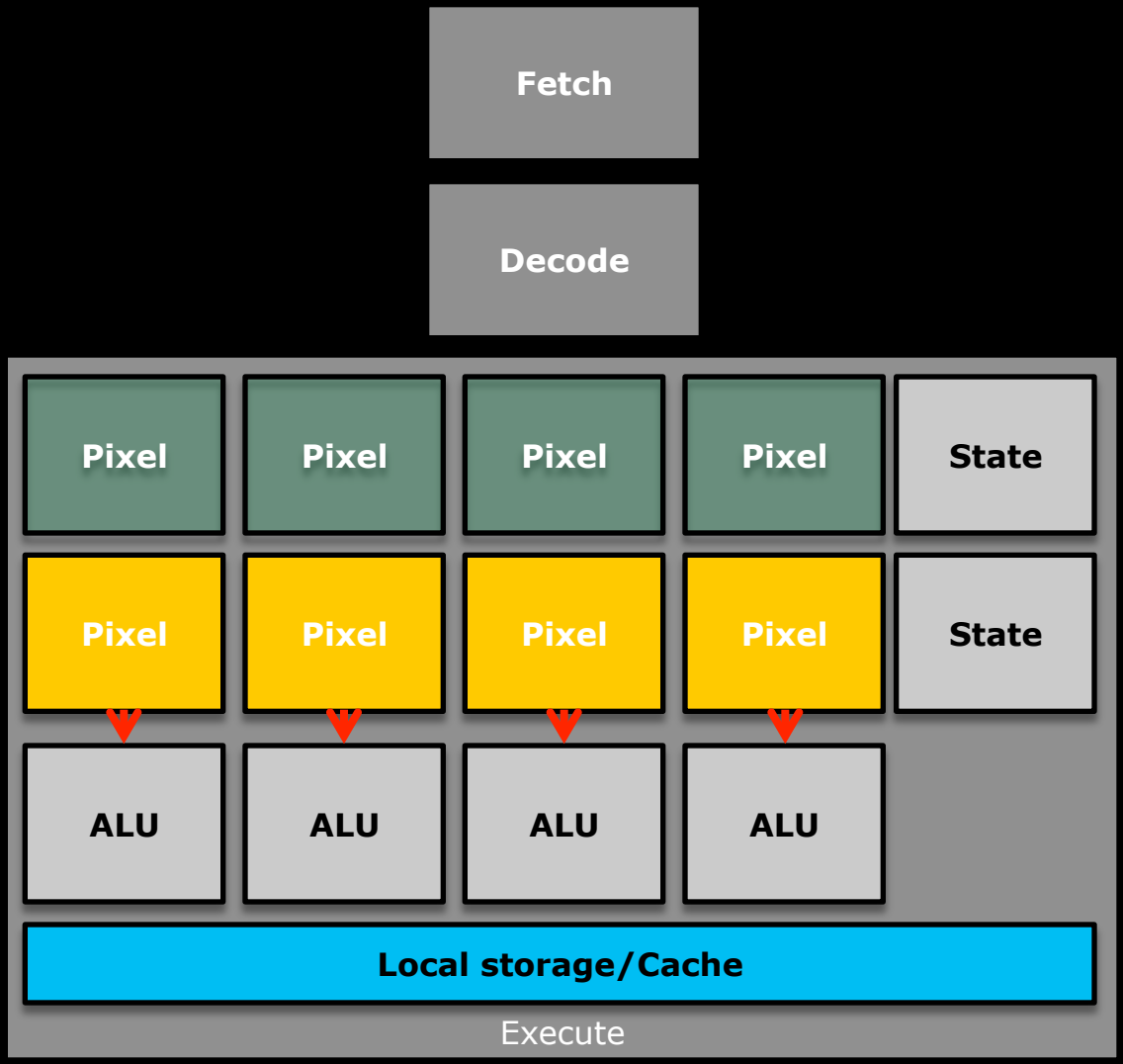
A throughput-oriented SIMD engine



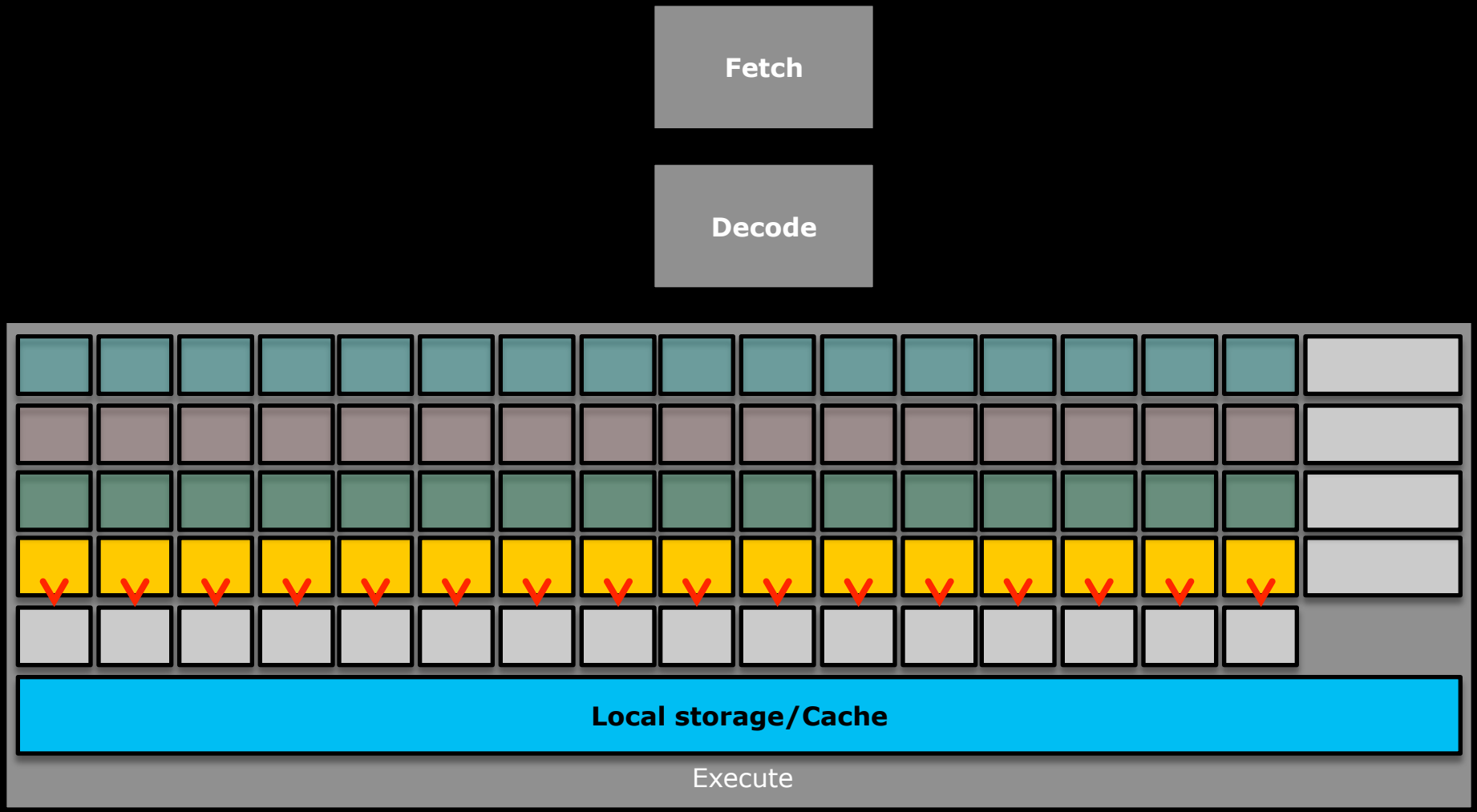
Adding the memory hierarchy

- Unlike most CPUs, GPUs do not have vast cache hierarchies.
  - Caches on CPUs allow primarily for lower access latency
- Heavy multithreading reduces the latency requirement
  - Latency is not an issue, we cover that with other threads
  - Total bandwidth still an issue, even with high-latency high-speed memory interfaces

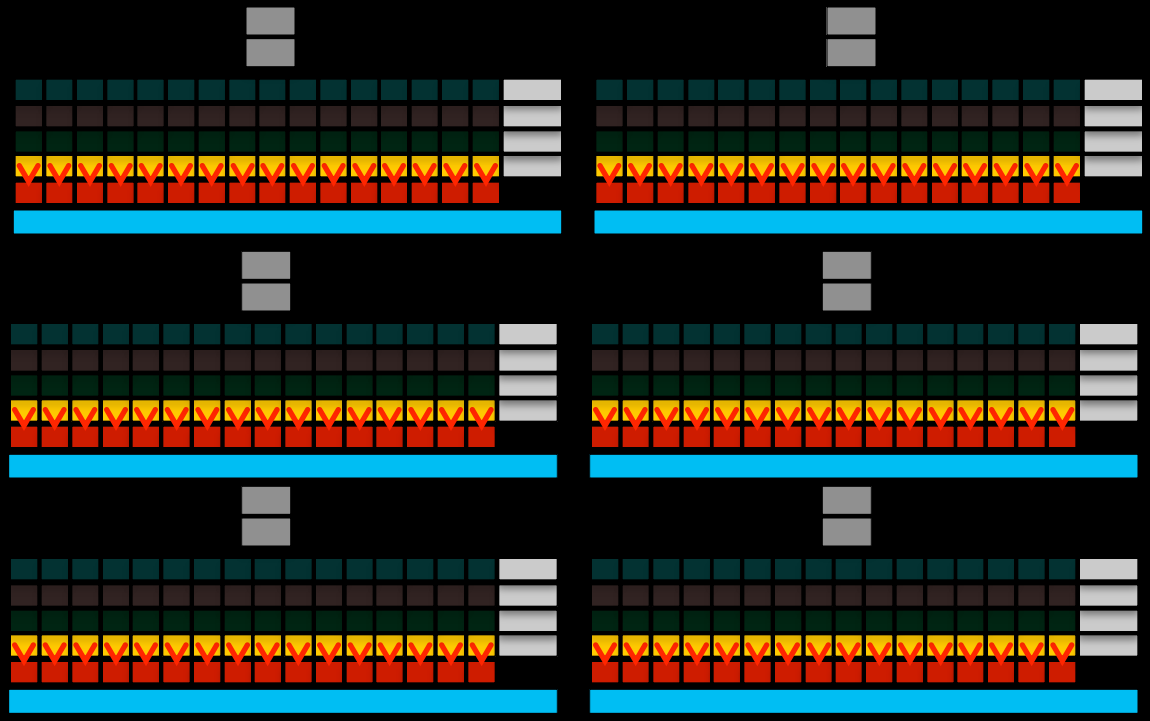
A throughput-oriented SIMD engine



A throughput-oriented SIMD engine



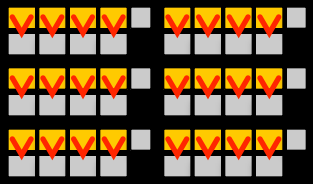
The GPU shader cores



The design space

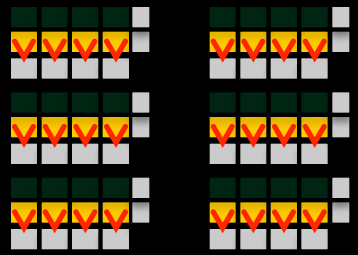
The AMD Phenom™ II X6

- 6 cores
- One state set per core
- 4-wide SIMD (actually there are two pipes)

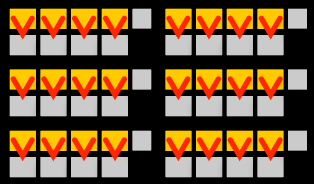


The Intel i7 6-core variants

- 6 cores
- Two state sets per core (SMT/Hyperthreading)
- 4-wide SIMD

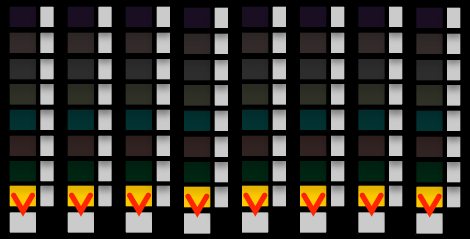


Phenom II X6

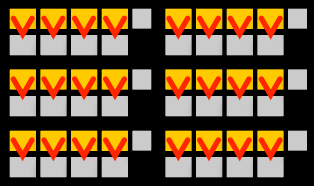


Sun UltraSPARC T2

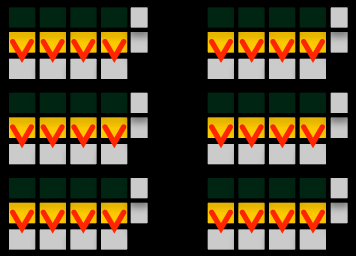
- 8 cores
- Eight state sets per core
- No SIMD



Phenom II X6

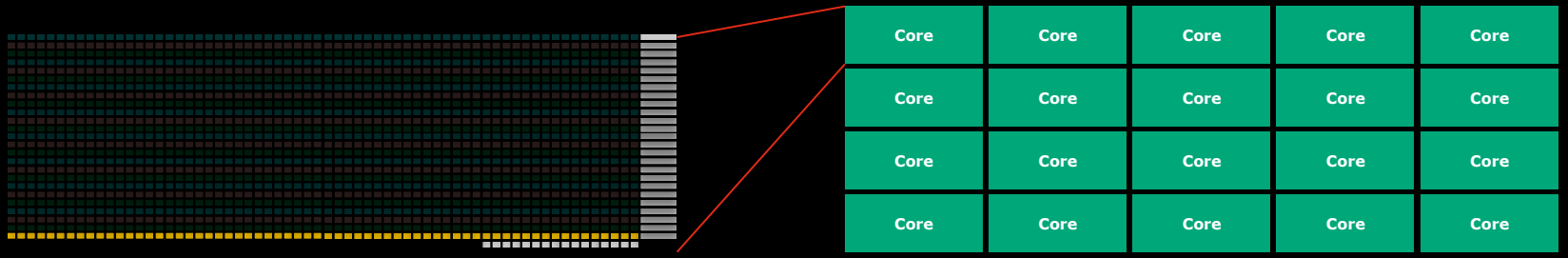


Intel i7 6-core

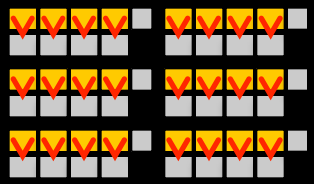


The AMD HD5870 GPU

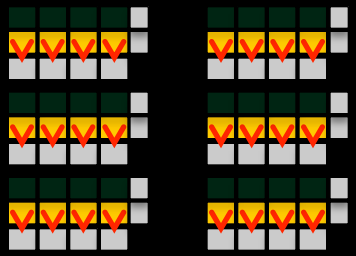
- 20 cores
- Up to 24 logical 64-SIMD wide state sets per core (number depends on register requirements)
- 16-wide physical



Phenom II X6



Intel i7 6-core



UltraSPARC T2

Summary

- We've:
  - looked at the basic principles of the GPU architecture in the processor design space
  - seen some of the tradeoffs that lead to GPU features