GRAPHICS HARDWARE

Niels Joubert, 4th August 2010, CS147

Today

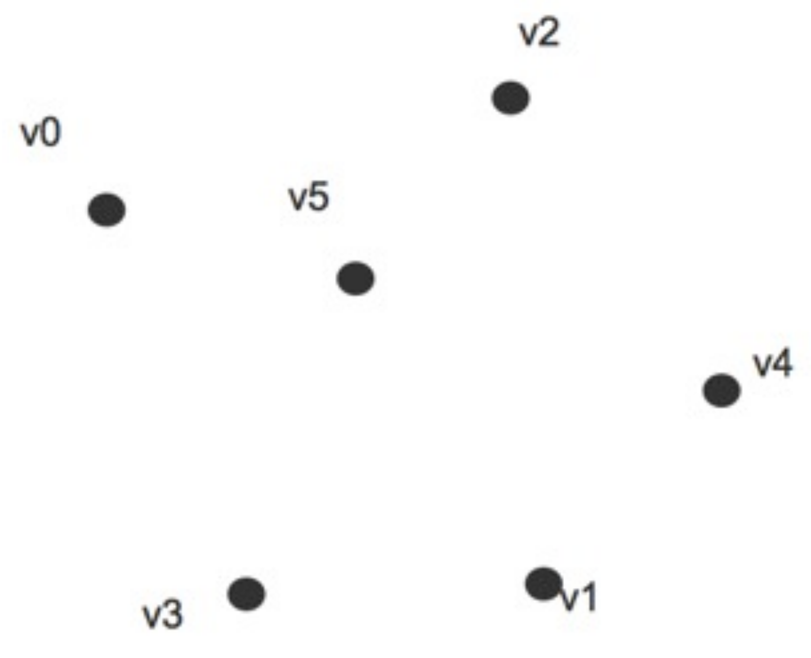
- Rendering Pipeline
- History
- Latest Architecture
- GPGPU Programming

RENDERING PIPELINE

Real-Time Graphics

Vertex Processing

Vertices are transformed into “screen space”

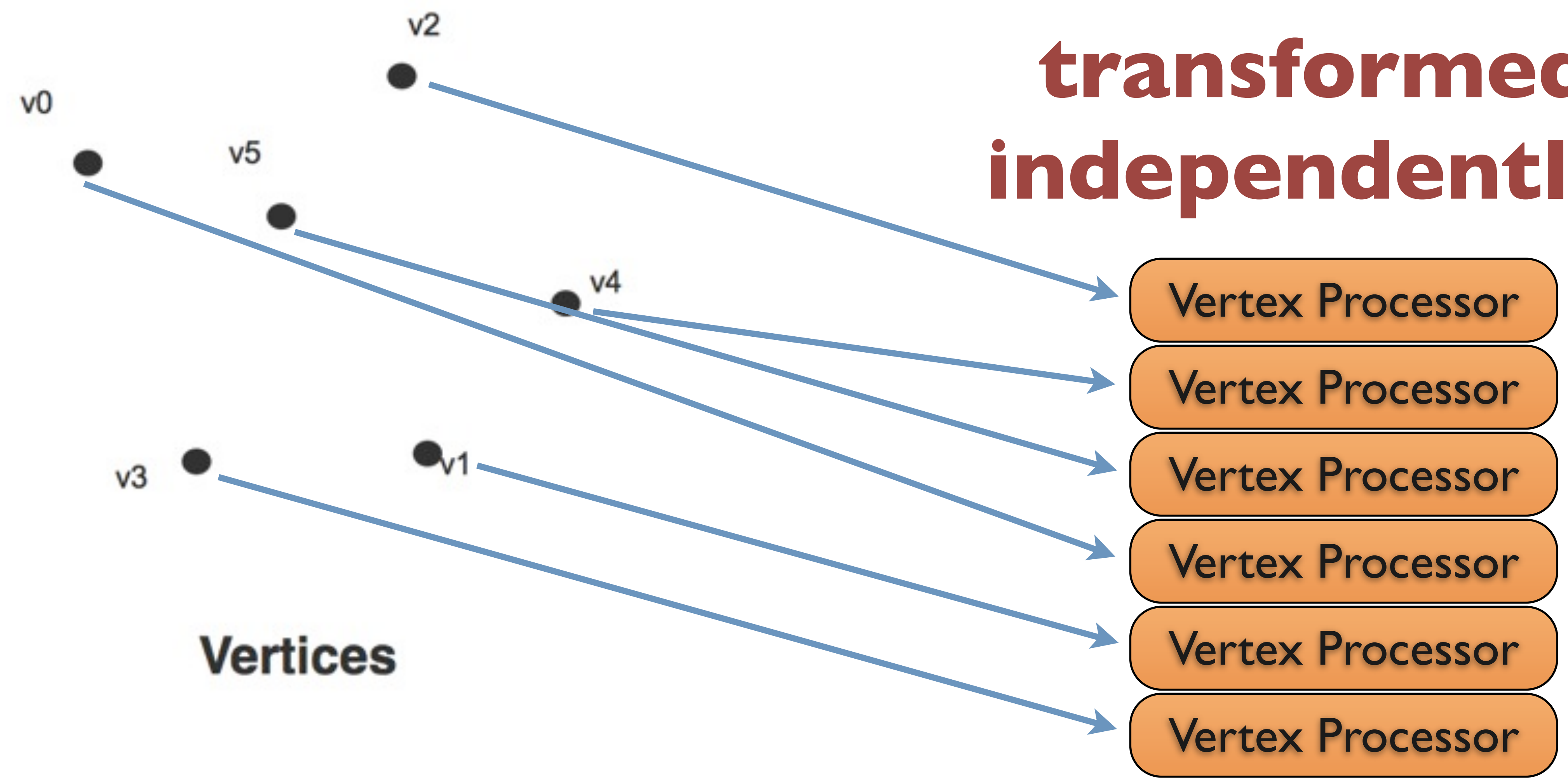


Vertices

Vertex Processing

Vertices are transformed into "screen space"

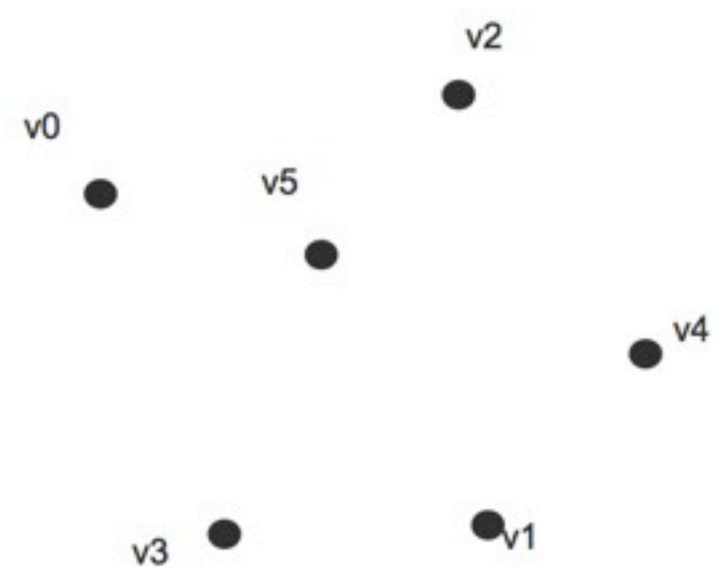
Each vertex is transformed independently!



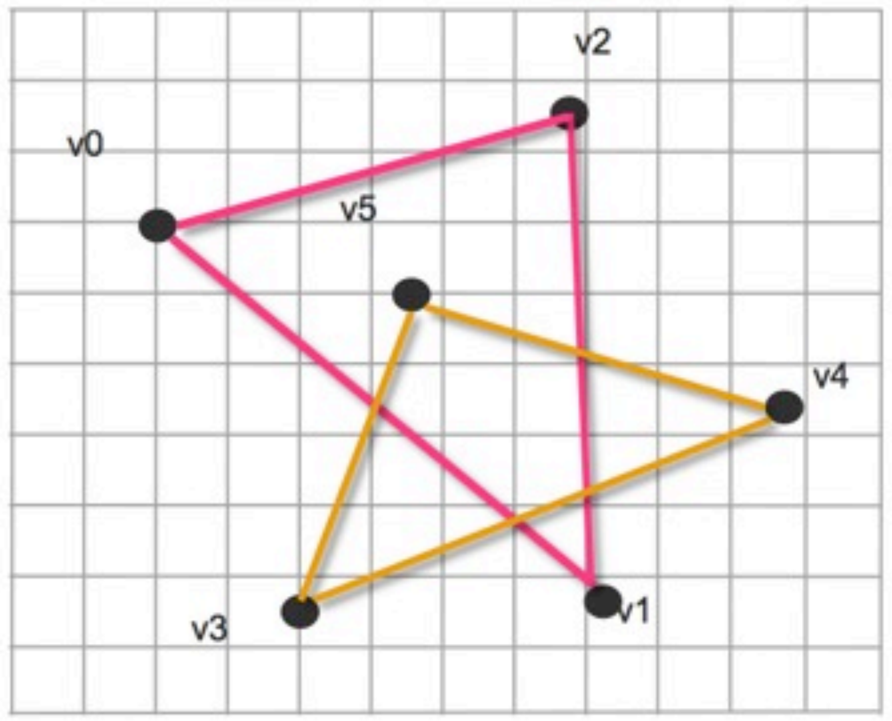
Primitive Processing

Vertices are organized into primitives

Primitives are clipped and culled



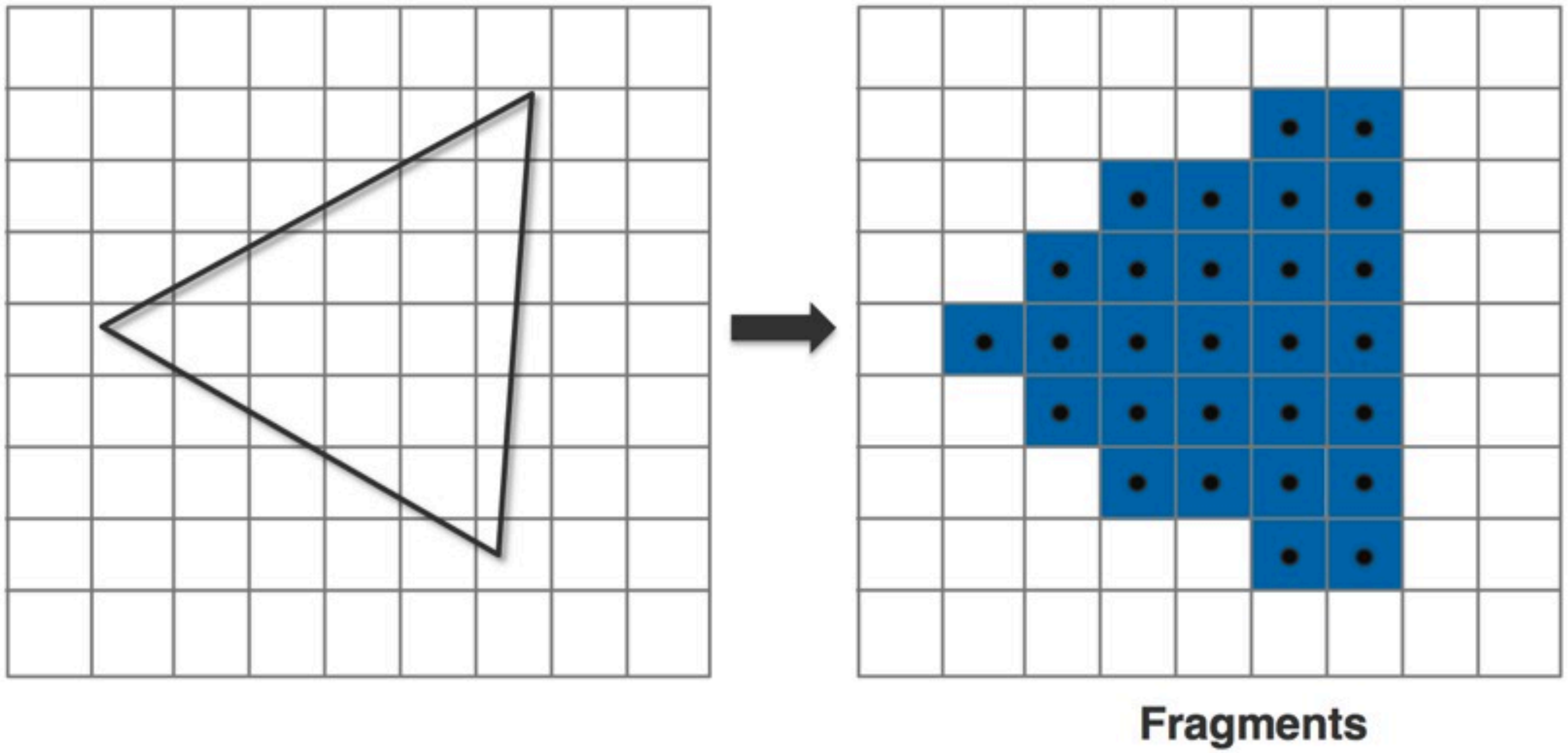
Vertices



Primitives (triangles)

Rasterization

Primitives are rasterized into pixel fragments.

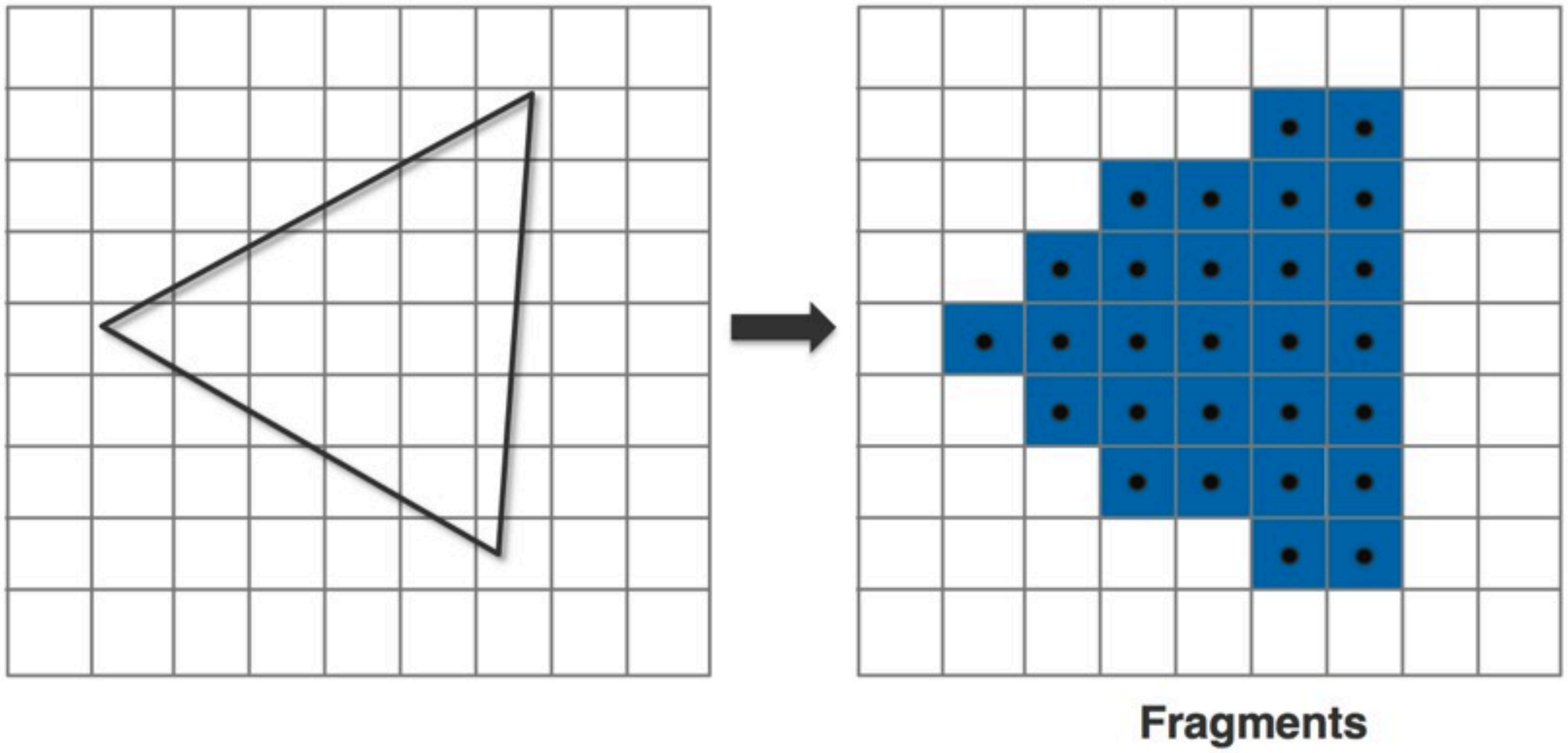


Rendering Pipeline

Rasterization

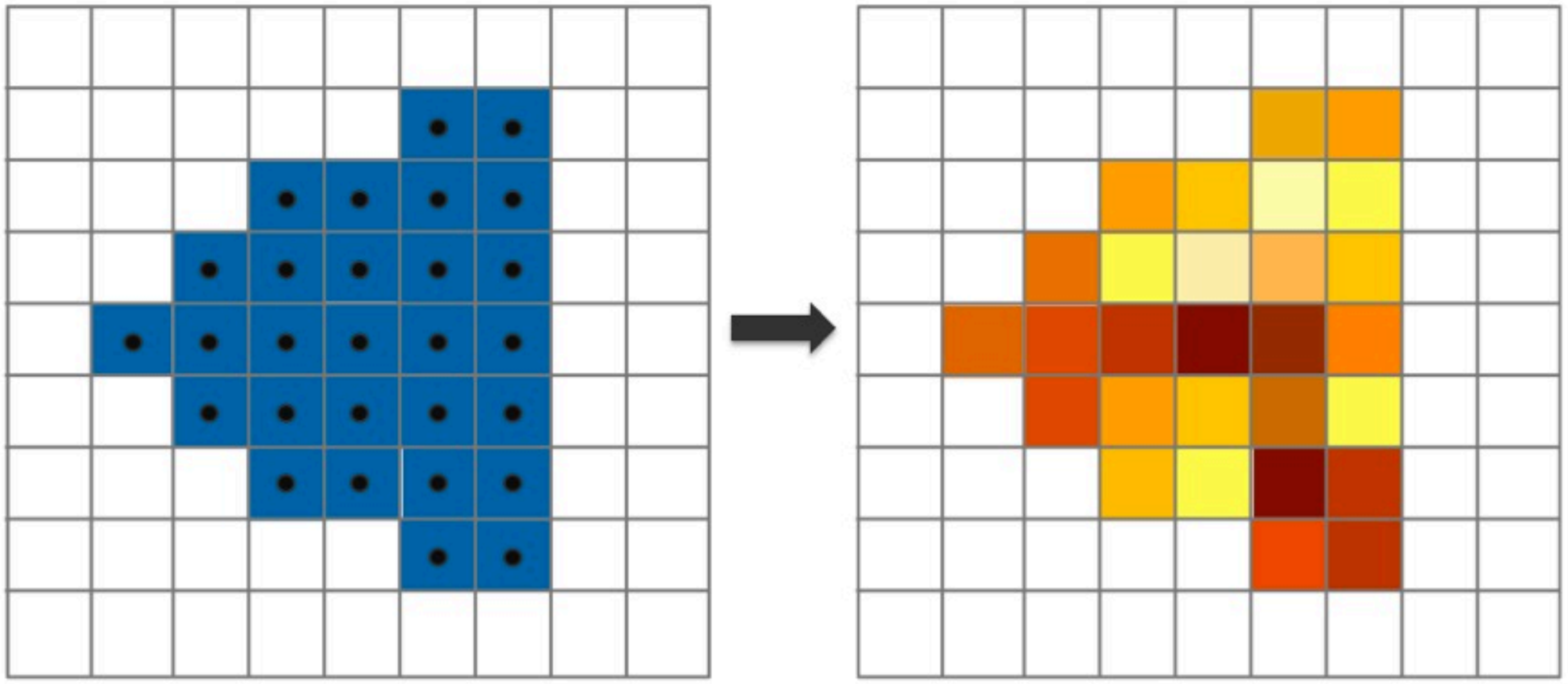
Each primitive is rasterized independently!

Primitives are rasterized into pixel fragments.



Fragment Processing

Fragments are shaded to compute a color at a pixel

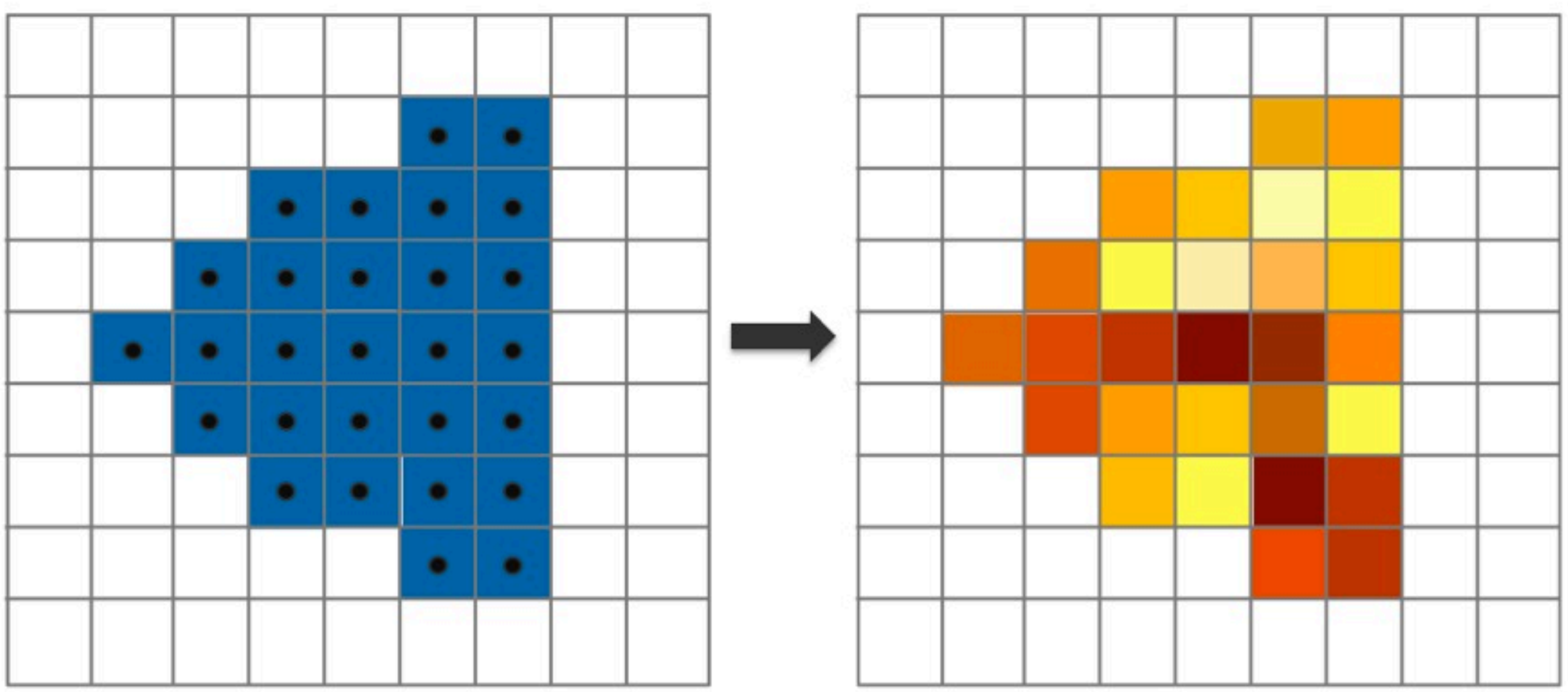


Shaded fragments

Fragment Processing

Each fragment is shaded independently!

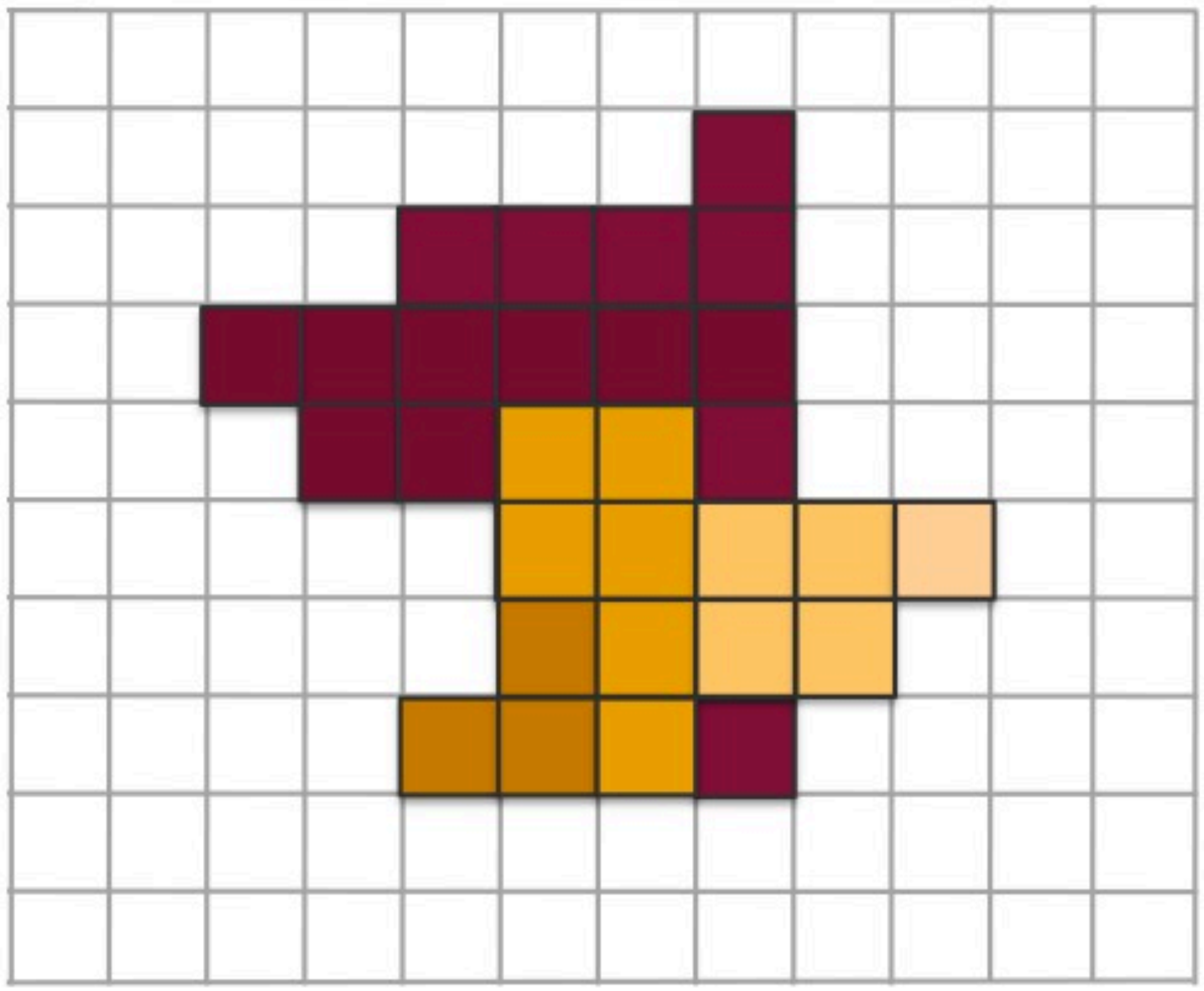
Fragments are shaded to compute a color at a pixel



Shaded fragments

Pixel Operations

Fragments are blended into the framebuffer



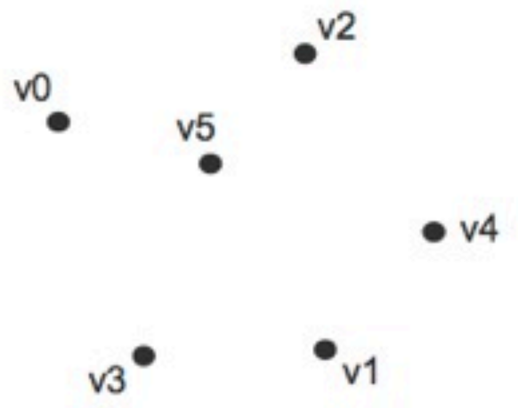
Pixels

Z-Buffer determines visibility

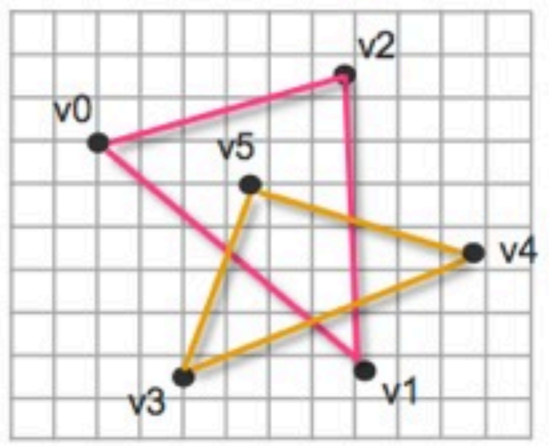
Frame buffer

- Memory location with aggregation ability
  - Many fragments end up on same pixel
  - All fragments are handled independently
  - Conflicts when writing to framebuffer?
  - Revisit this later! [Synchronization / Atomics]

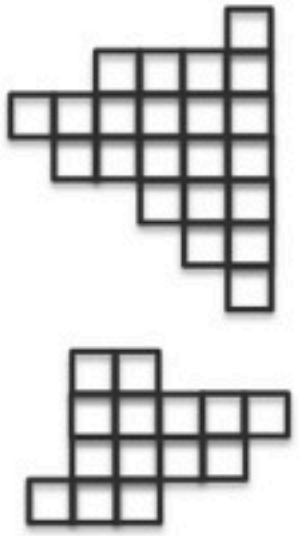
Pipeline Entities



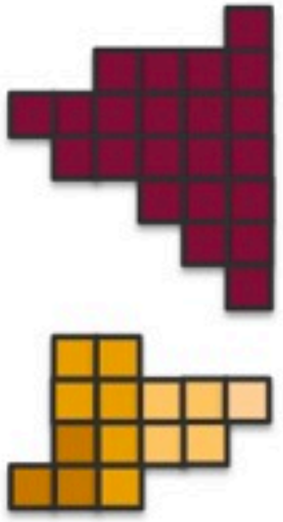
Vertices



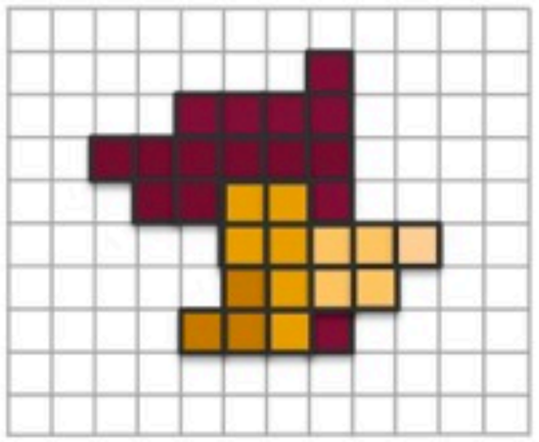
Primitives



Fragments

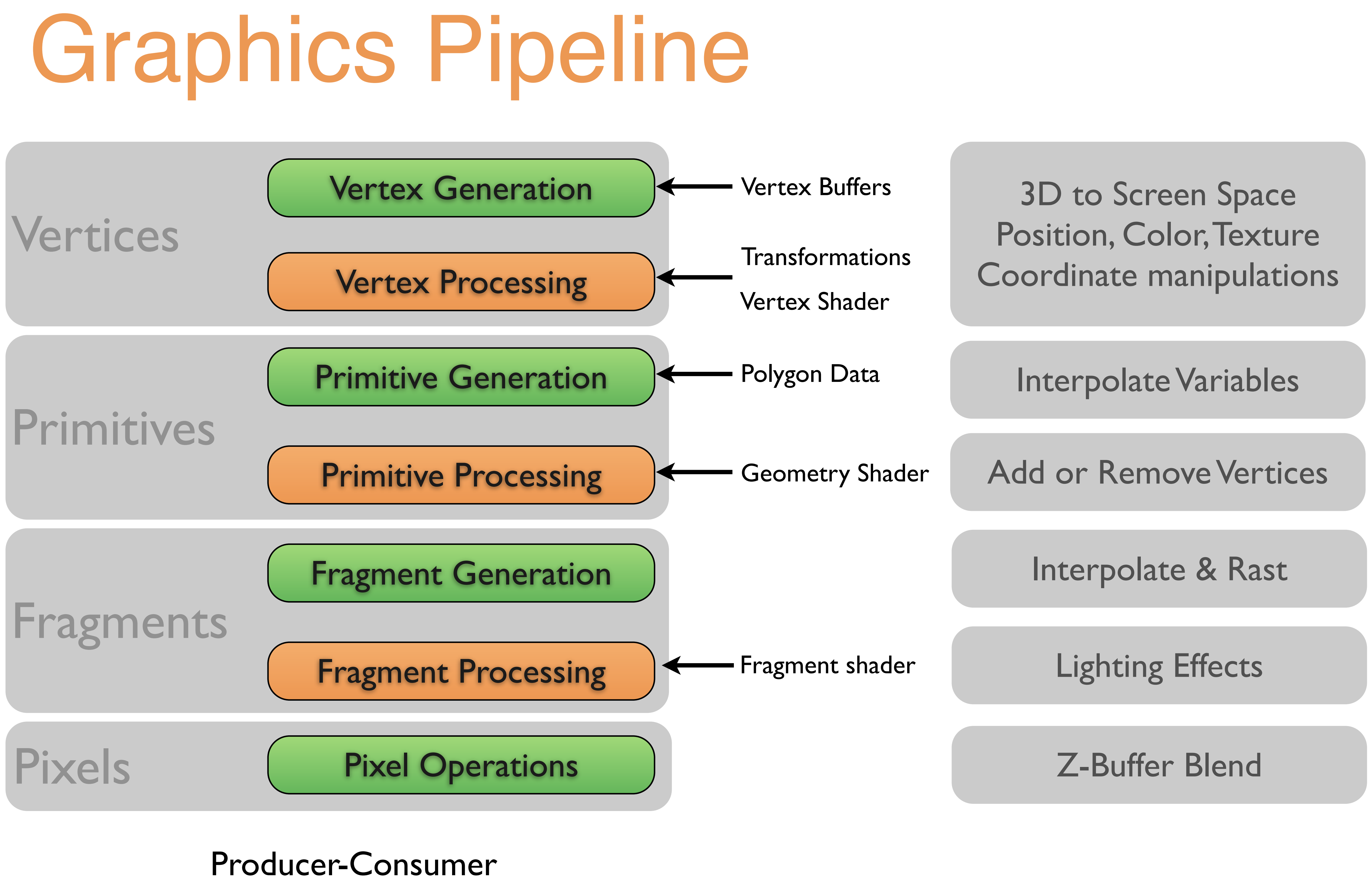


Fragments (shaded)



Pixels

Rendering Pipeline



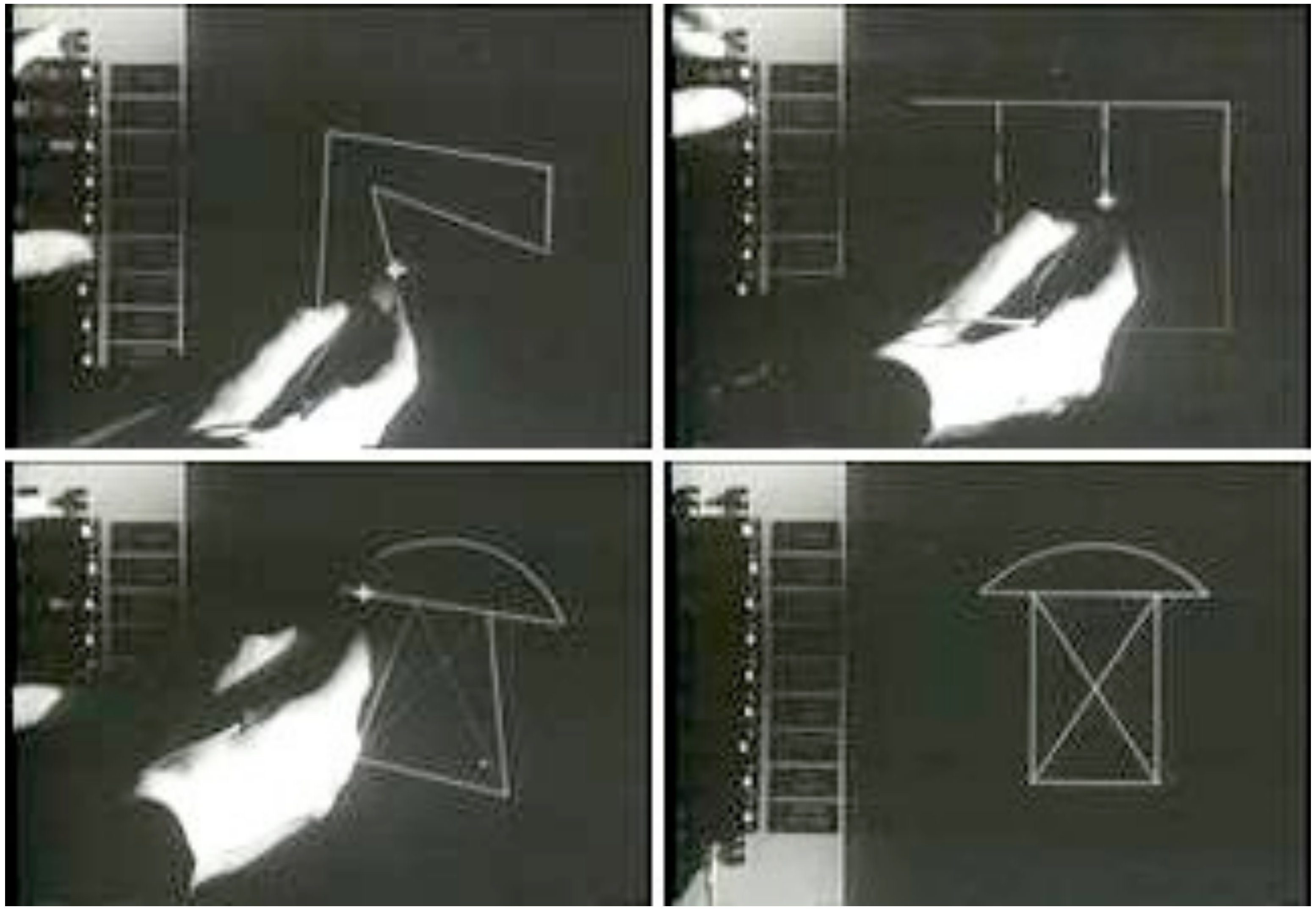
HISTORY

How we got to where we are

History: 1963

Sketchpad

The first GUI.



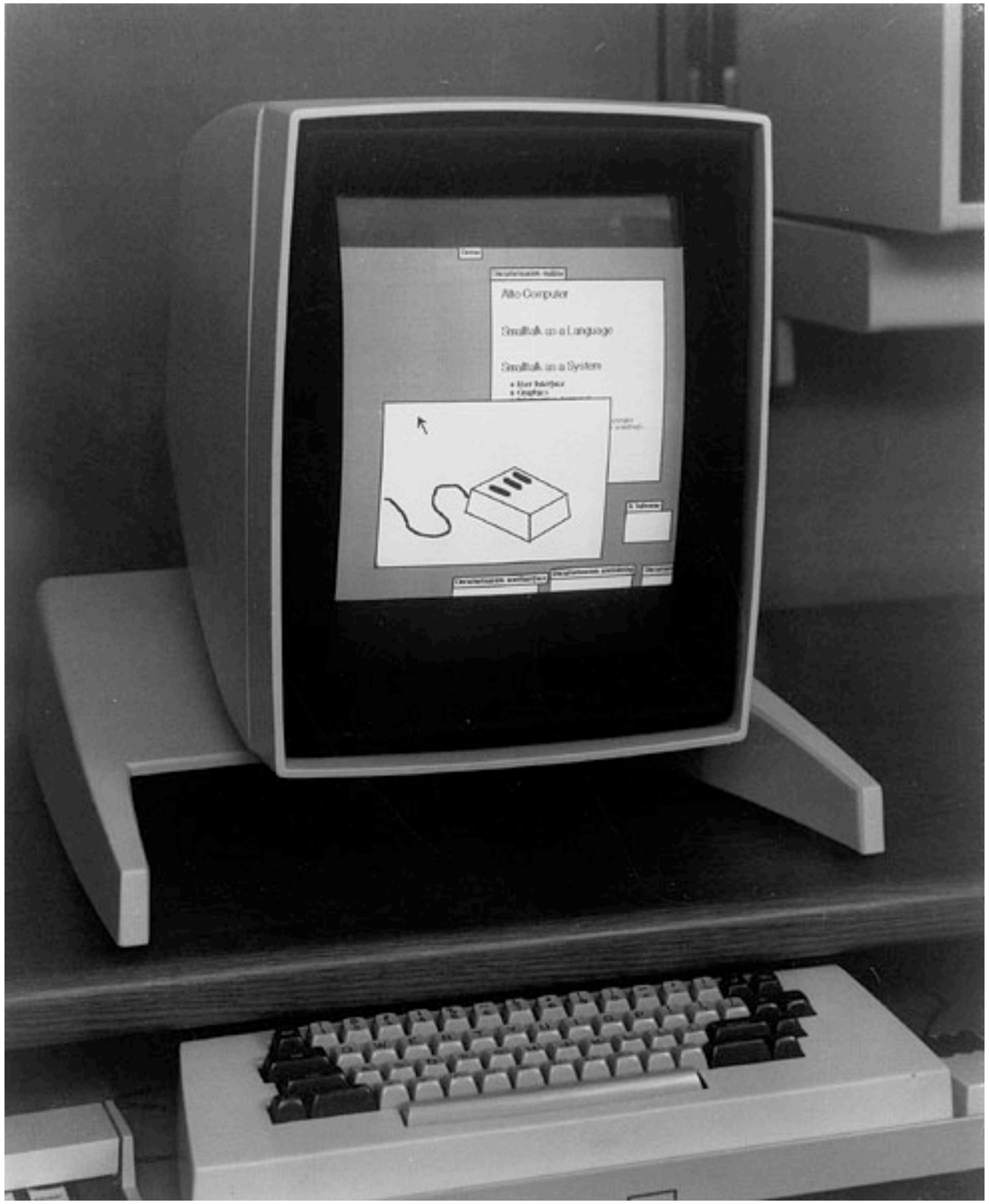
History: 1972

Xerox Alto

1972:

Mouse, Keyboard, Files, Folders, Draggable windows

"Personal Computer"



History: 1977

Apple II

Launched 1977

- 1Mhz 6502 processor
- 4KB RAM (expandable to 48KB for $1340)
- “Graphics for the Masses”

Did it have a “graphics card”?

History: 1977

Apple II

CPU:

- Writes “pixel” data to RAM

Video Hardware:

- Reads RAM in scanlines, generates NTSC

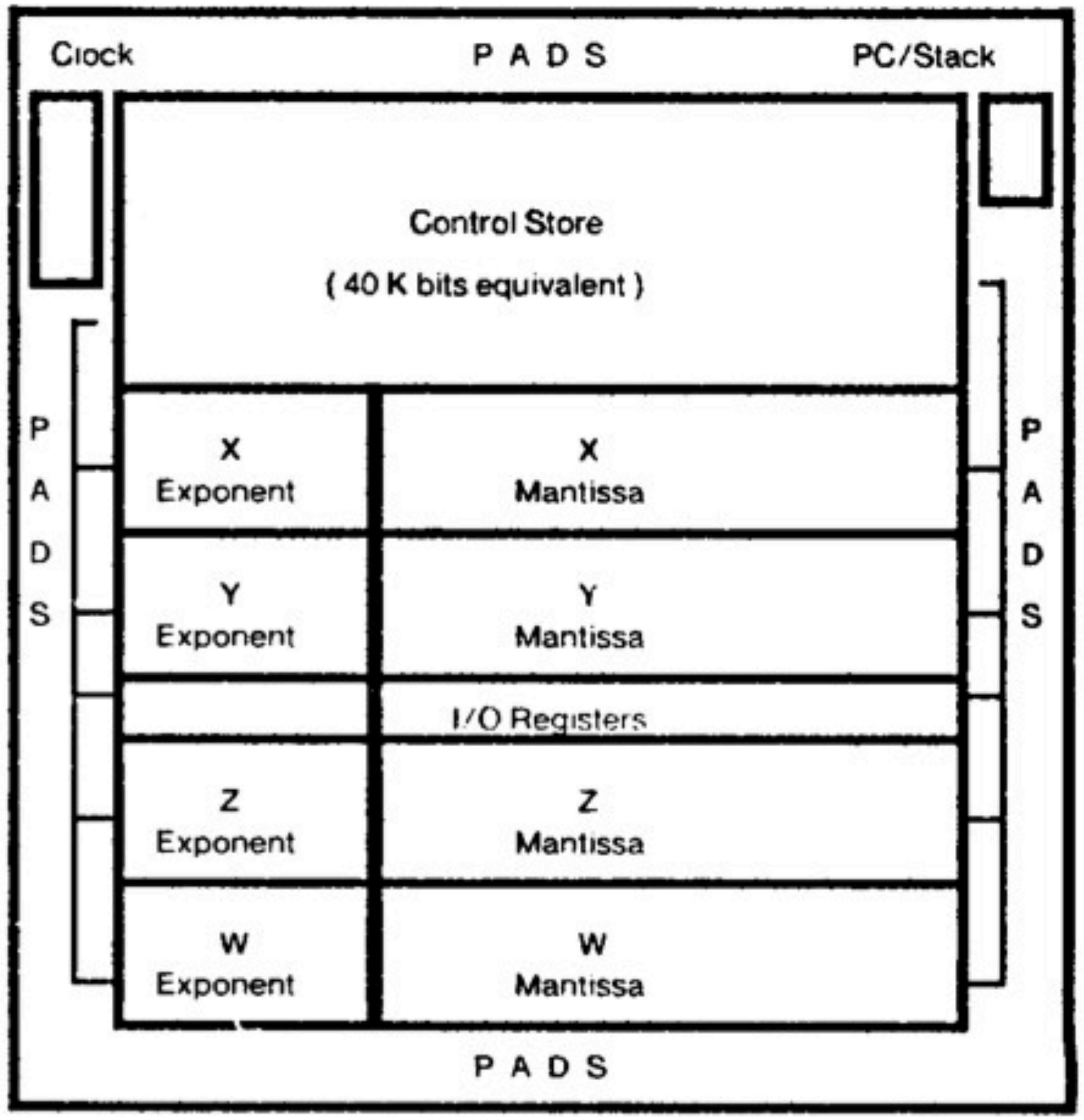
No, there is no graphics card.

History: 1982

The Geometry Engine

A VLSI Geometry System for Graphics
James H. Clark, Computer Systems Lab, Stanford University & Silicon Graphics, Inc.

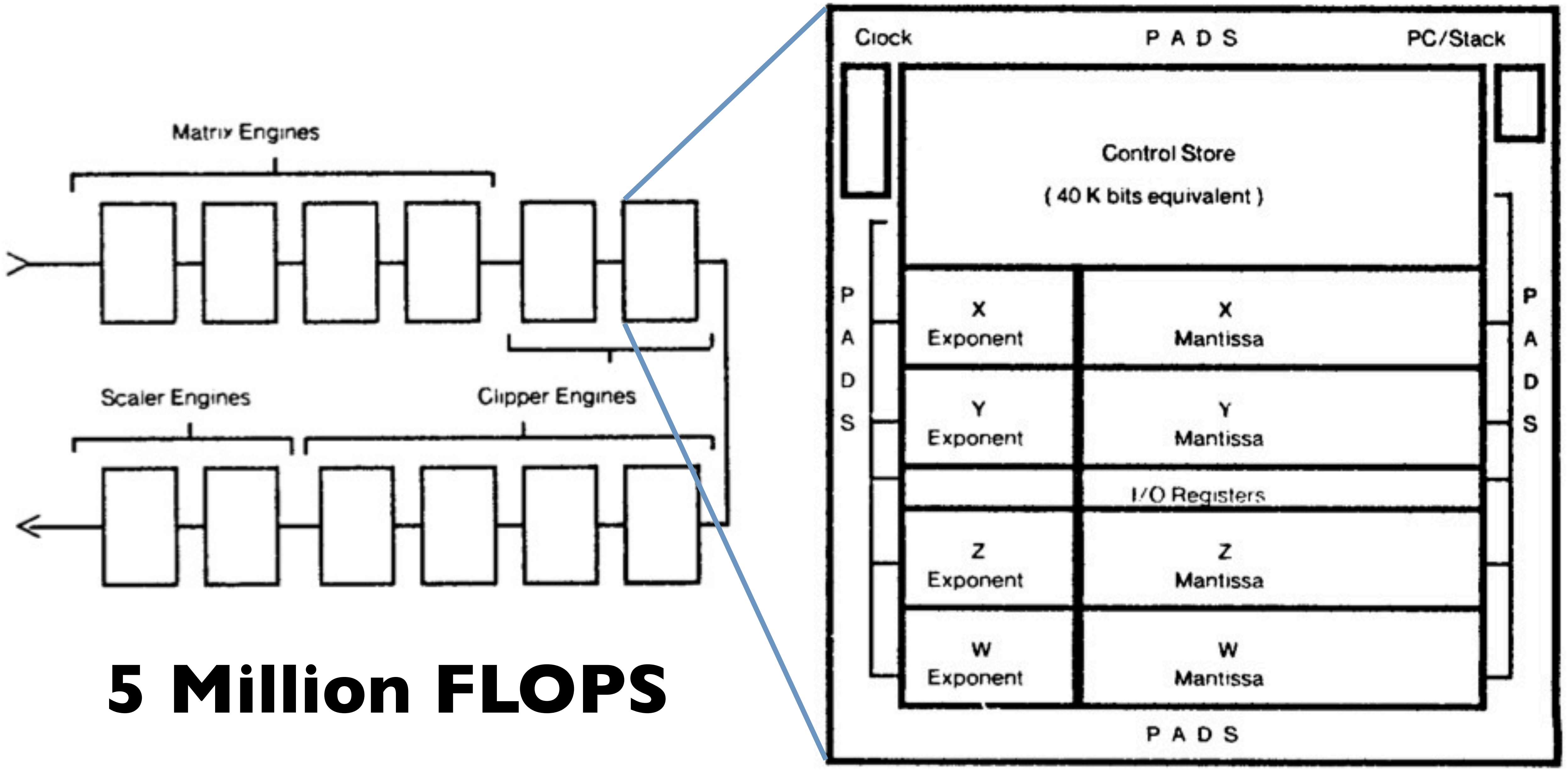
"The Geometry System is a ... computing system for computer graphics constructed from a basic building block, the *Geometry Engine.*



History: 1982

The Geometry Engine

A VLSI Geometry System for Graphics
James H. Clark, Computer Systems Lab, Stanford University & Silicon Graphics, Inc.

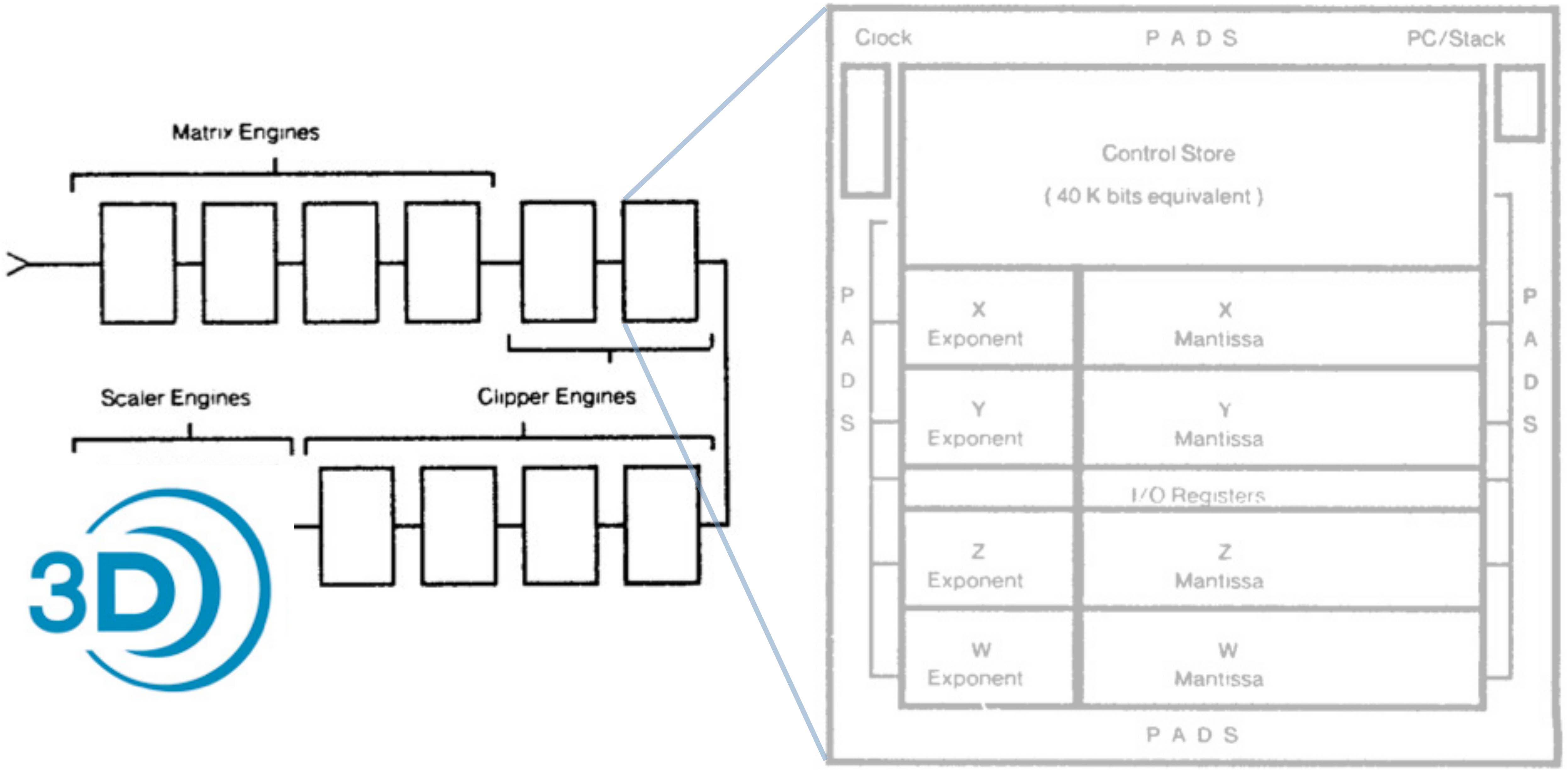


5 Million FLOPS

History: 1982

The Geometry Engine

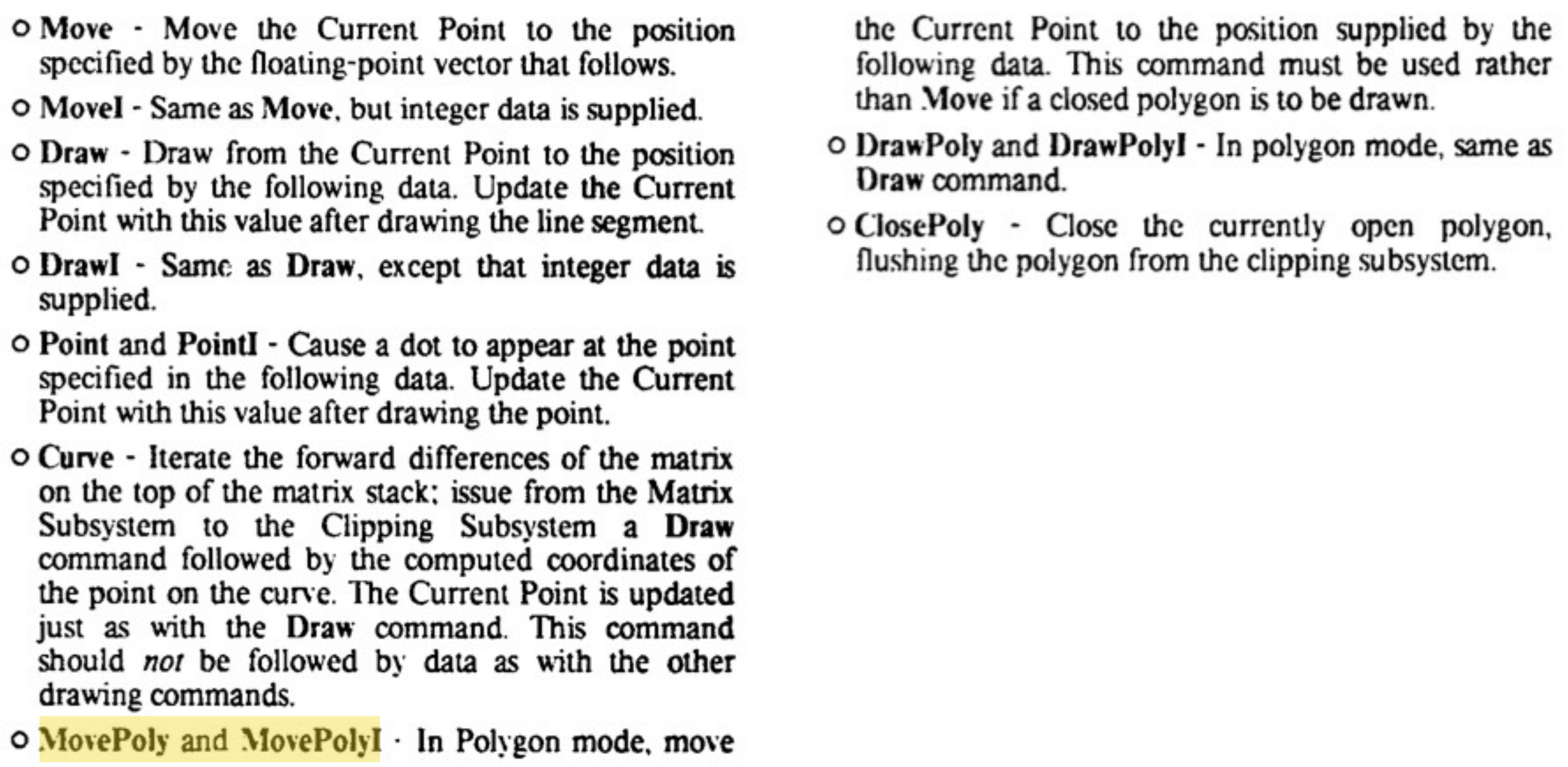
A VLSI Geometry System for Graphics
James H. Clark, Computer Systems Lab, Stanford University & Silicon Graphics, Inc.



History: 1982

The Geometry Engine

Instruction Set:

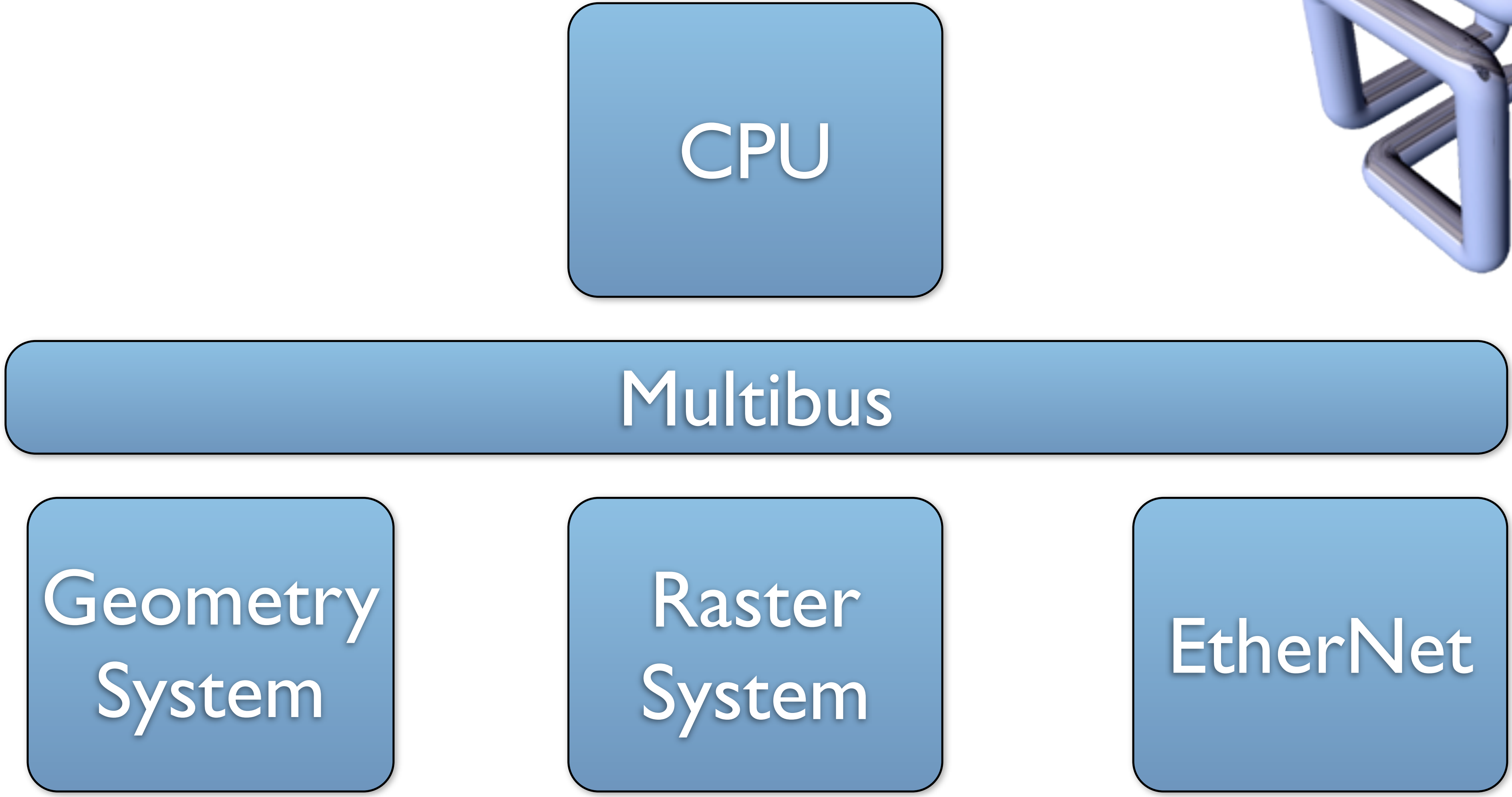


- **Move** - Move the Current Point to the position specified by the floating-point vector that follows.
- **MoveI** - Same as **Move**, but integer data is supplied.
- **Draw** - Draw from the Current Point to the position specified by the following data. Update the Current Point with this value after drawing the line segment.
- **DrawI** - Same as **Draw**, except that integer data is supplied.
- **Point** and **PointI** - Cause a dot to appear at the point specified in the following data. Update the Current Point with this value after drawing the point.
- **Curve** - Iterate the forward differences of the matrix on the top of the matrix stack; issue from the Matrix Subsystem to the Clipping Subsystem a **Draw** command followed by the computed coordinates of the point on the curve. The Current Point is updated just as with the **Draw** command. This command should *not* be followed by data as with the other drawing commands.
- **MovePoly** and **MovePolyI** - In Polygon mode, move the Current Point to the position supplied by the following data. This command must be used rather than **Move** if a closed polygon is to be drawn.
- **DrawPoly** and **DrawPolyI** - In polygon mode, same as **Draw** command.
- **ClosePoly** - Close the currently open polygon, flushing the polygon from the clipping subsystem.

History: 1982

IRIS - Integrated Raster Imaging System

Silicon Graphics Inc's first real-time 3D rendering engine.



SiliconGraphics
Computer Systems

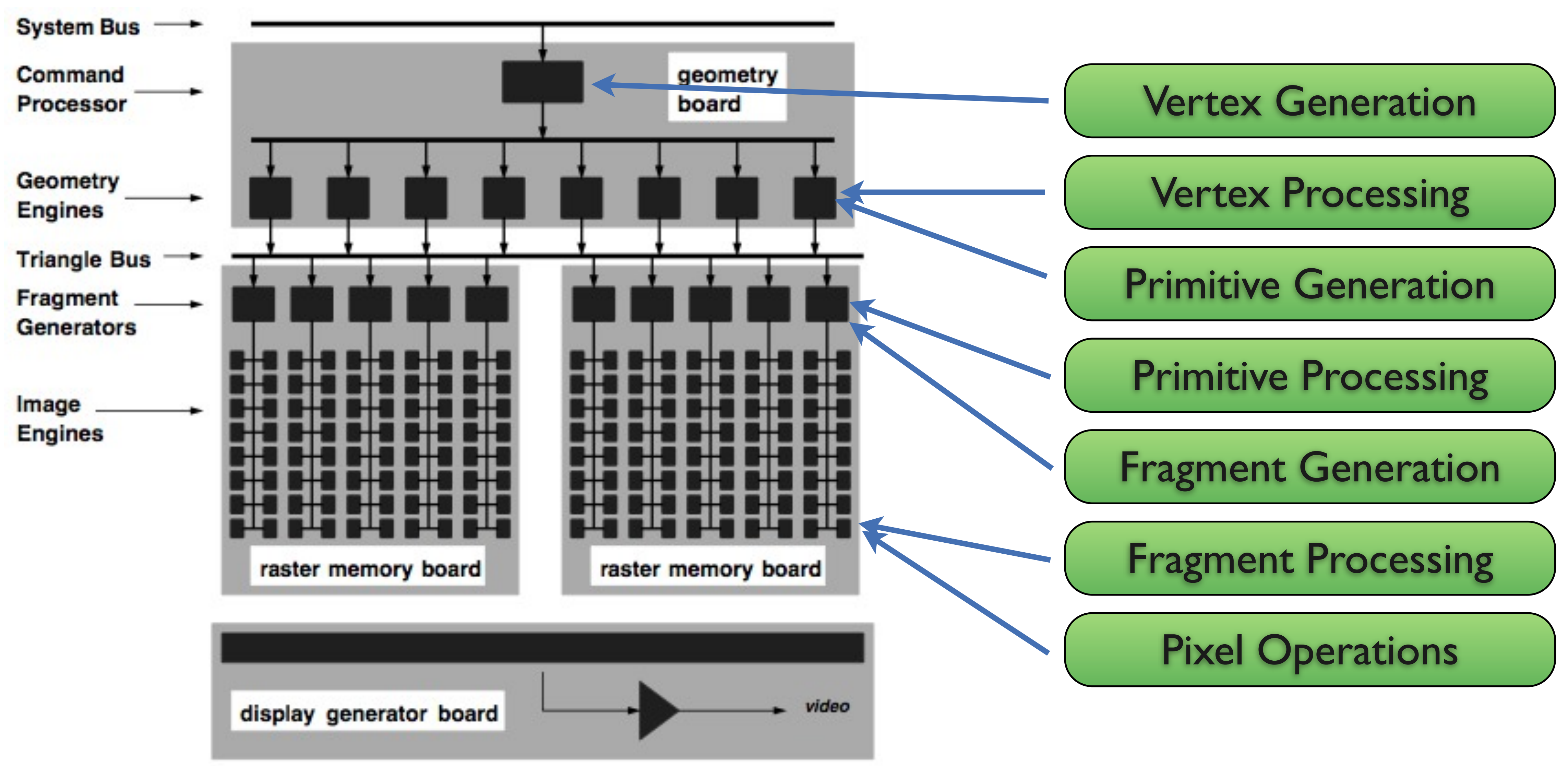
“BLIT” & Commodore Amiga

BLock **I**mage **T**ransfer

- Co-processor / logic block
- Rapid movement and modification of memory blocks
- Commodore Amiga had a complete blitter in hardware, in a separate “graphics processor”

Silicon Graphics RealityEngine

"Its target capability is the rendering of lighted, smooth shaded, depth buffered, **texture mapped**, antialiased triangles." - RealityEngine Graphics, K. Akeley, 1993



History: 1992 OpenGL 1.0 & 1995 Direct3D

OpenGL 1.0

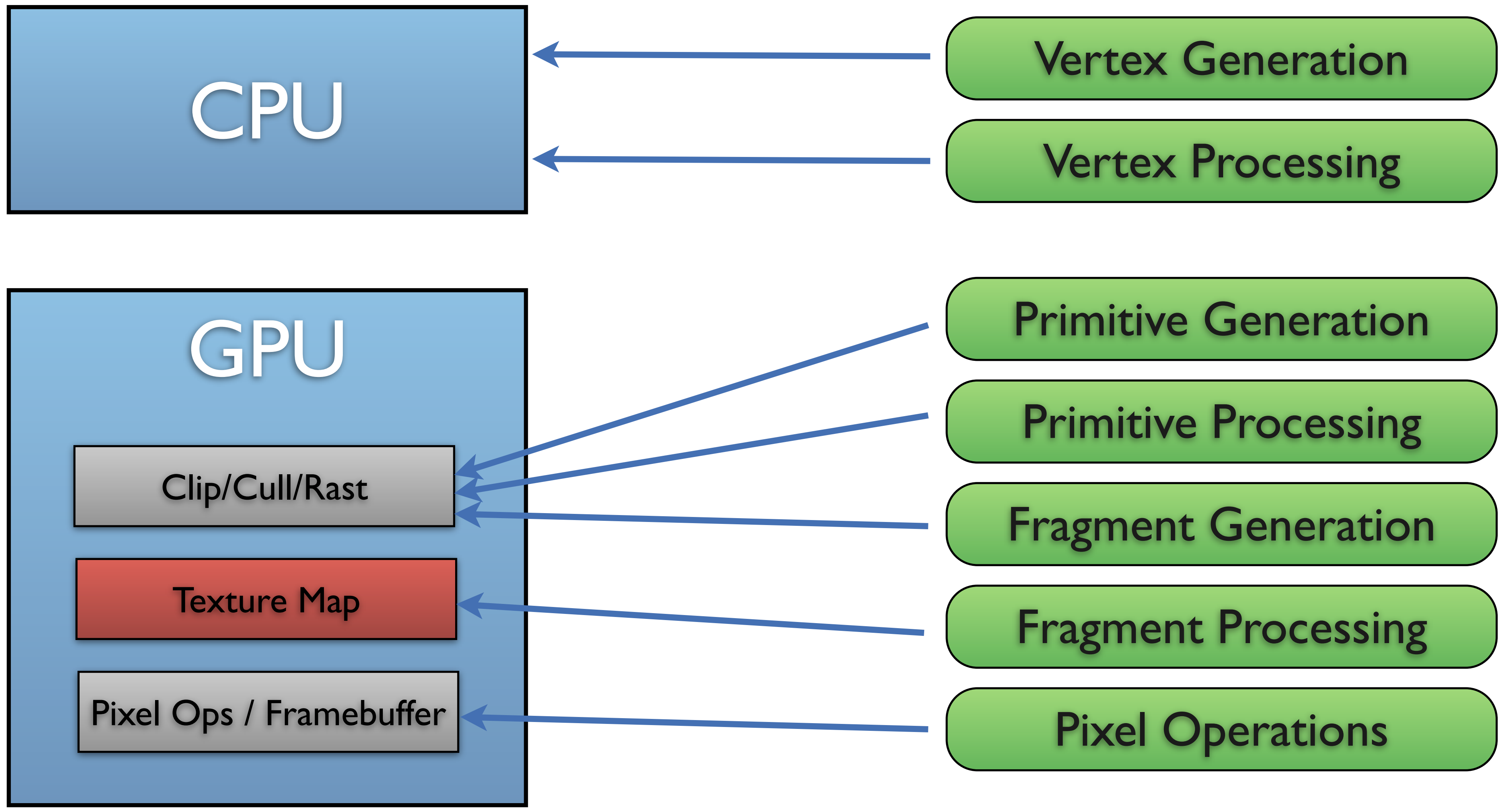
Silicon Graphics

- Proprietary IRIS GL API (state of the art)
- OpenGL as open standard derived from IRIS
- Standardised HW access, device drivers becomes important
- HUGE success:

OpenGL allows HW to evolve, SW to decouple

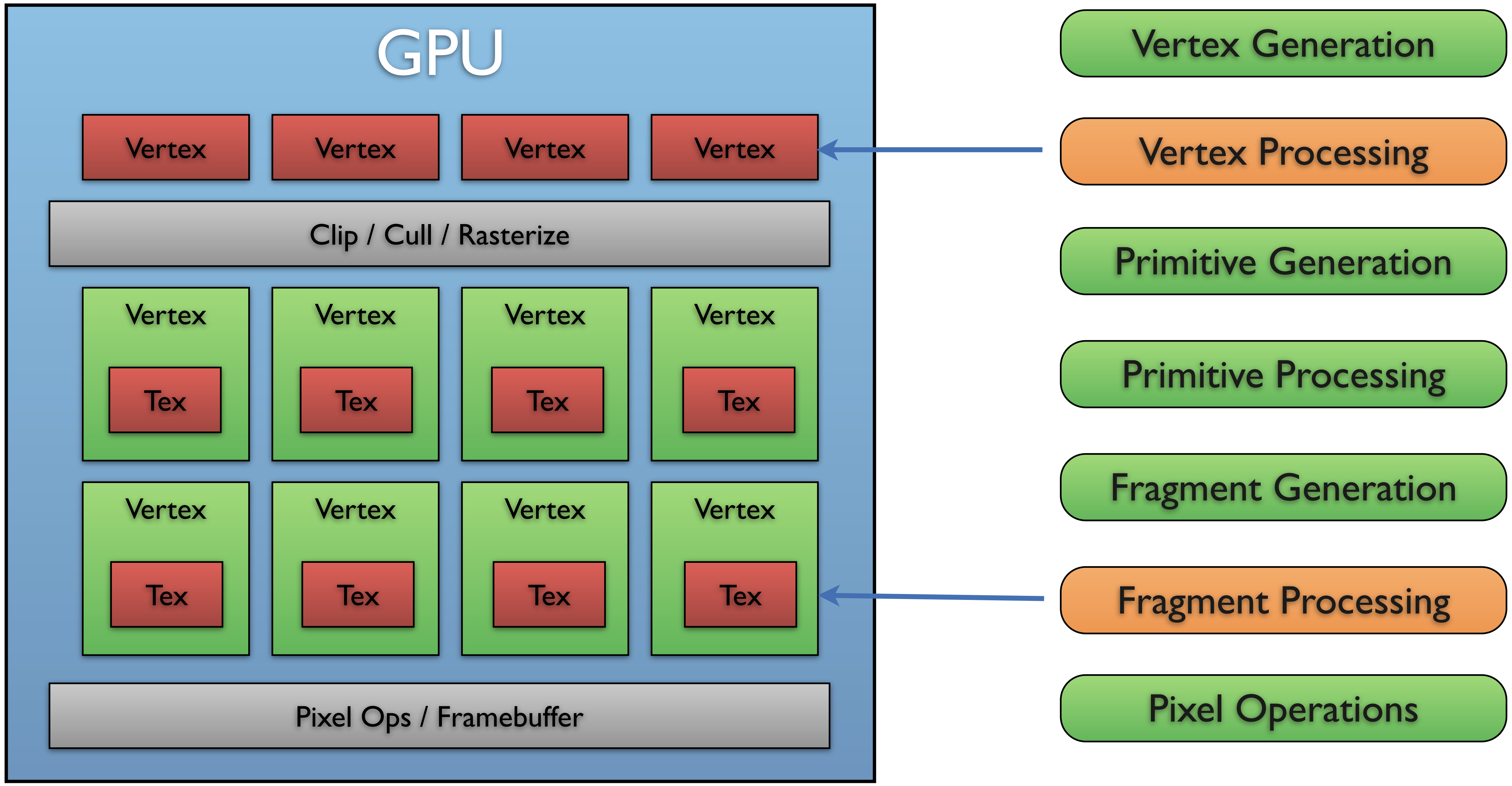
History: 1998

NVidia RIVA TNT



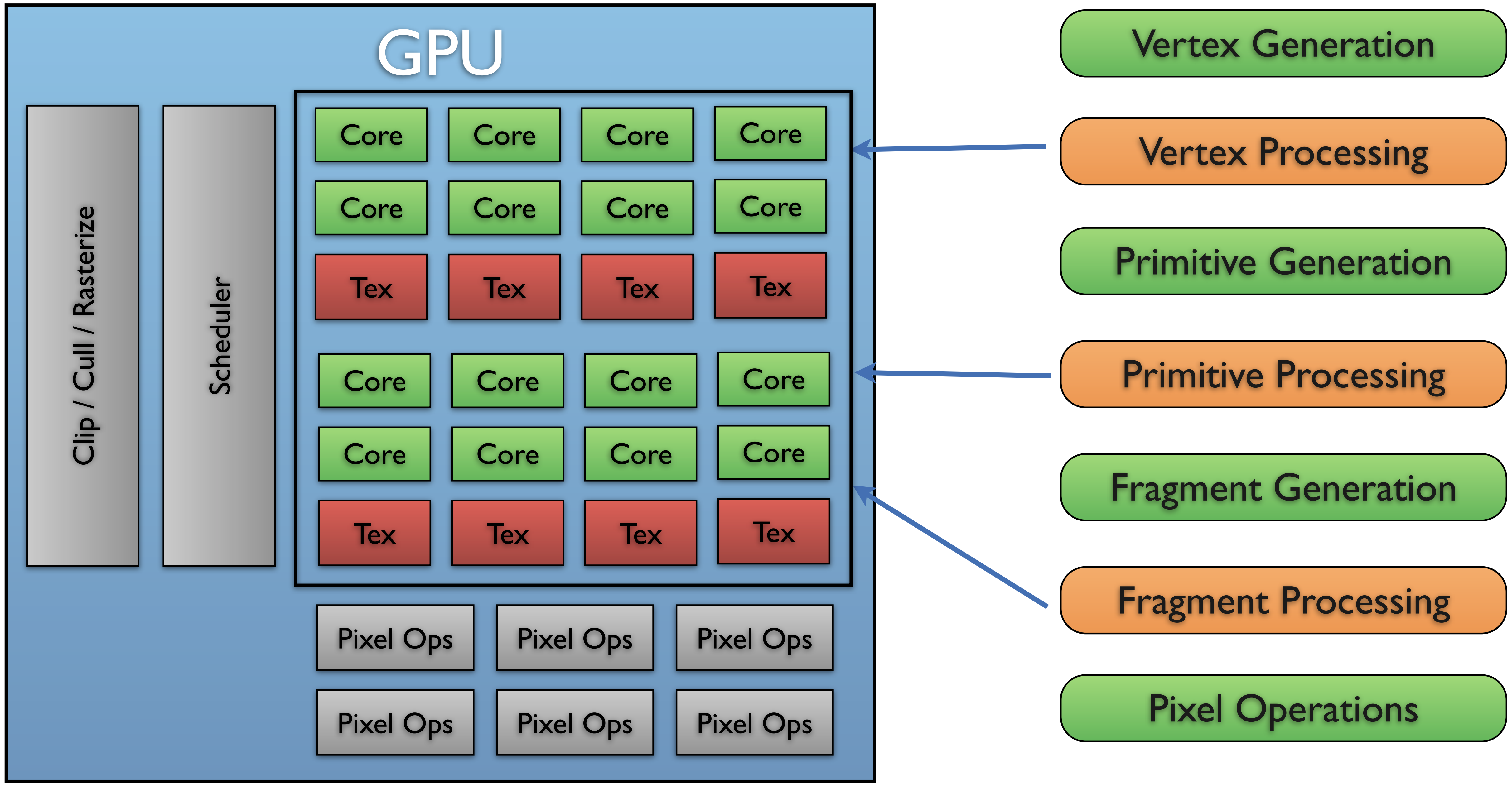
History: 2002

Direct3D 9, OpenGL 2.0



ATI Radeon 9700

"Unified Shading" GPUs



GeForce G80

GRAPHICS ARCHITECTURE

GPUs as Throughput-Oriented Hardware

CPU Evolution

Single stream of instructions REALLY FAST

- Long, deep pipelines
- Branch Prediction & Speculative Execution
- Hierarchy of Caches
- Instruction Level Parallelism (ILP)

Architecture

G5 (2003)

G5

Architecture

G5 (2003)

- 2 Ghz
- 1 Ghz FSB
- 4GB RAM
- 2 FPUs
- 50 million transistors
- 215 inst. pipeline
- Branch Prediction

Architecture

G5 (2003)

- 2 Ghz
- 1 Ghz FSB
- 4GB RAM
- 2 FPUs
- 50 million transistors
- 215 inst. pipeline
- Branch Prediction

Fermi (2010)

- 1.4 Ghz
- 1.8 Ghz FSB
- 4GB RAM (1.5 in GTX)

Architecture

G5 (2003)	Fermi (2010)
2 Ghz	1.4 Ghz
1 Ghz FSB	1.8 Ghz FSB
4GB RAM	4GB RAM (1.5 in GTX)
2 FPUs	960 FPUs
50 million transistors	3 billion transistors
215 inst. pipeline	
Branch Prediction	No Branch Prediction

Moore’s Law

“The number of transistors on an integrated circuit doubles every two years”

- Gorden E. Moore

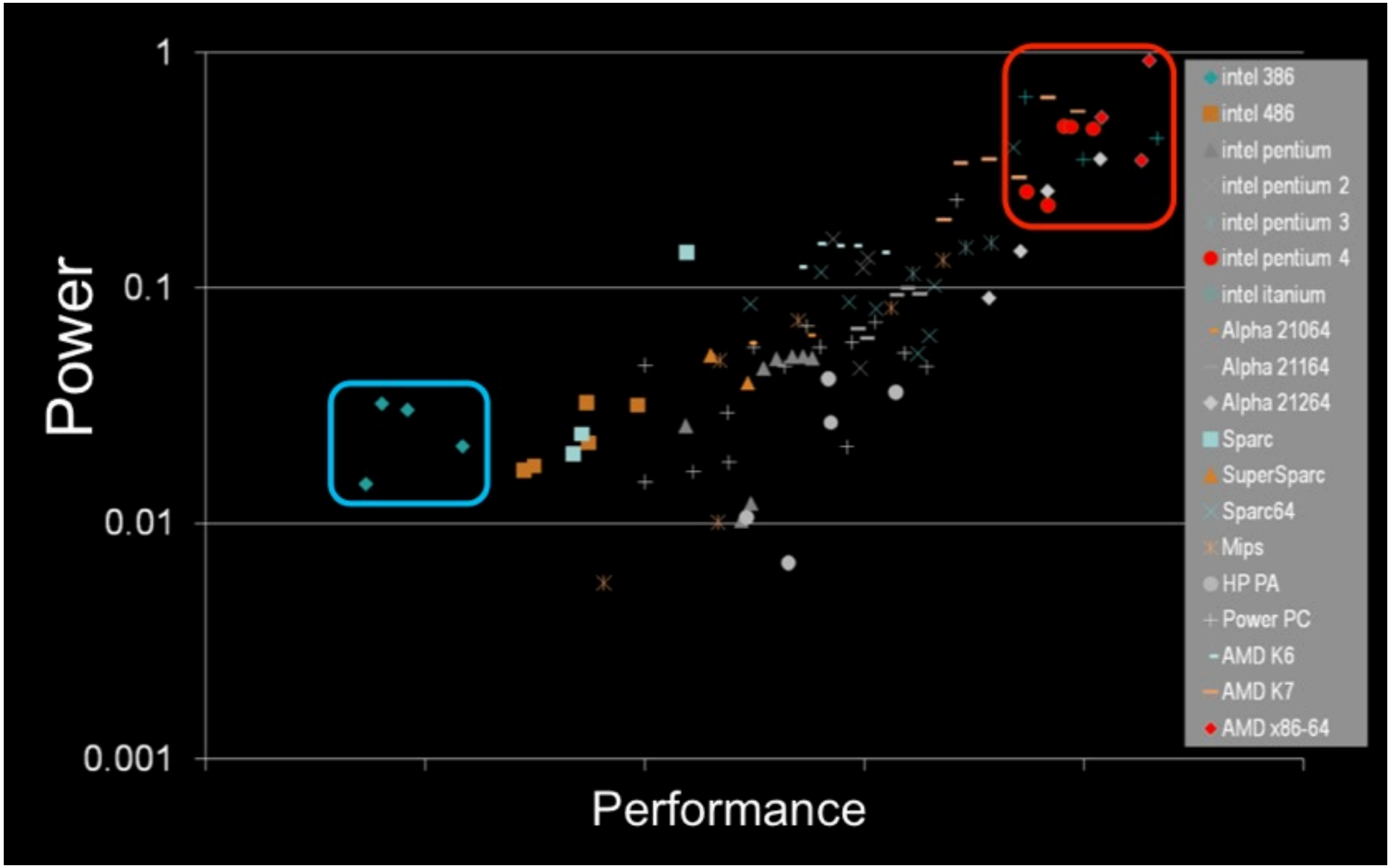
Moore’s Law

“The number of transistors on an integrated circuit doubles every two years”

- Gorden E. Moore

What Matters: How we use these transistors

Buy Performance with Power



Serial Performance Scaling

- Cannot continue to scale Mhz

  There is no 10 Ghz chip

- Cannot increase power consumption per area

  We’re melting chips

- Can continue to increase transistor count

Using Transistors

- Instruction-level parallelism

  out-of-order execution, speculation, branch prediction

- Data-level parallelism

  vector units, SIMD execution

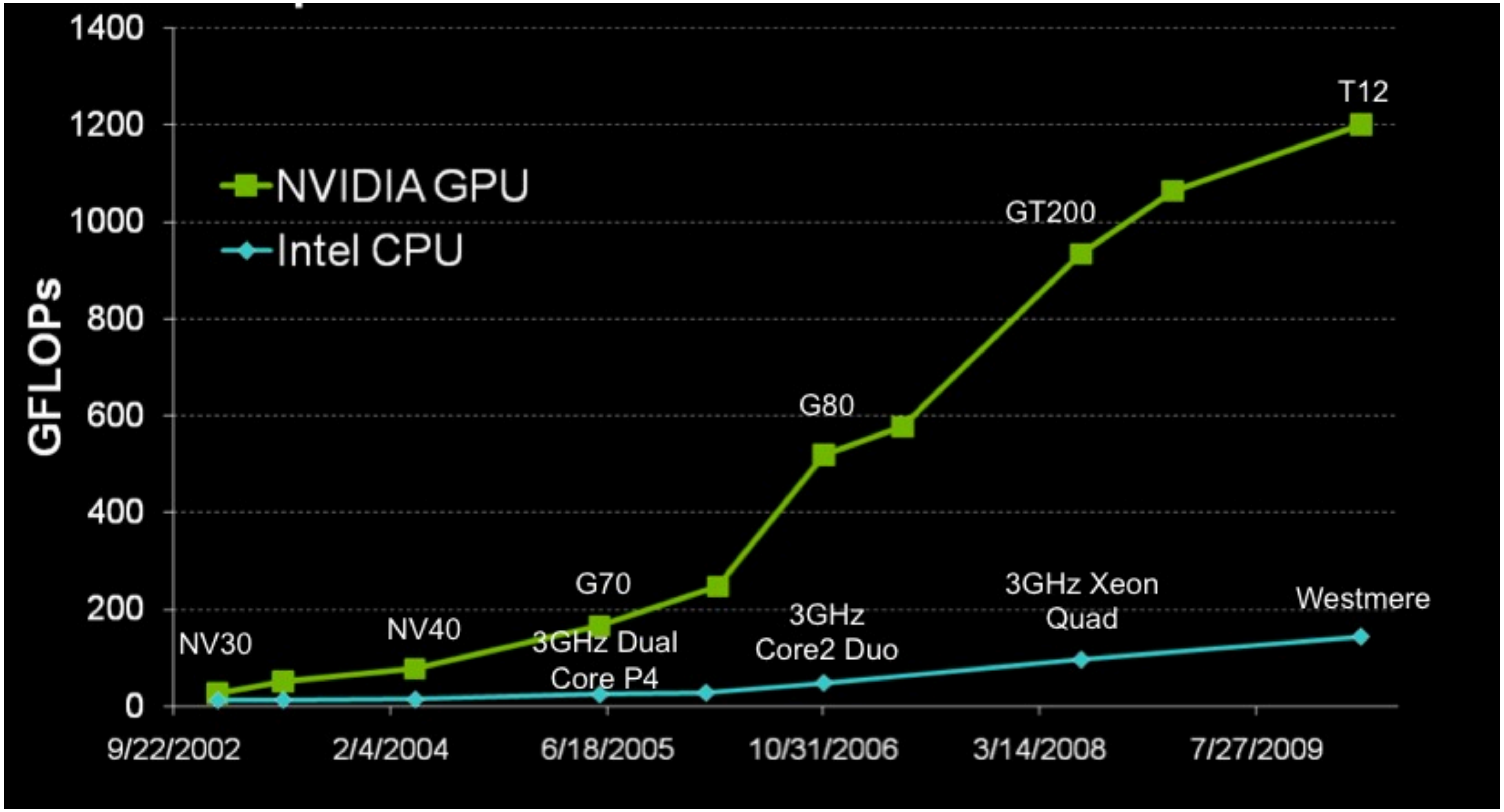
  SSE, AVX, Cell SPE, Clearspeed

- Thread-level parallelism

  multithreading, multicore, manycore

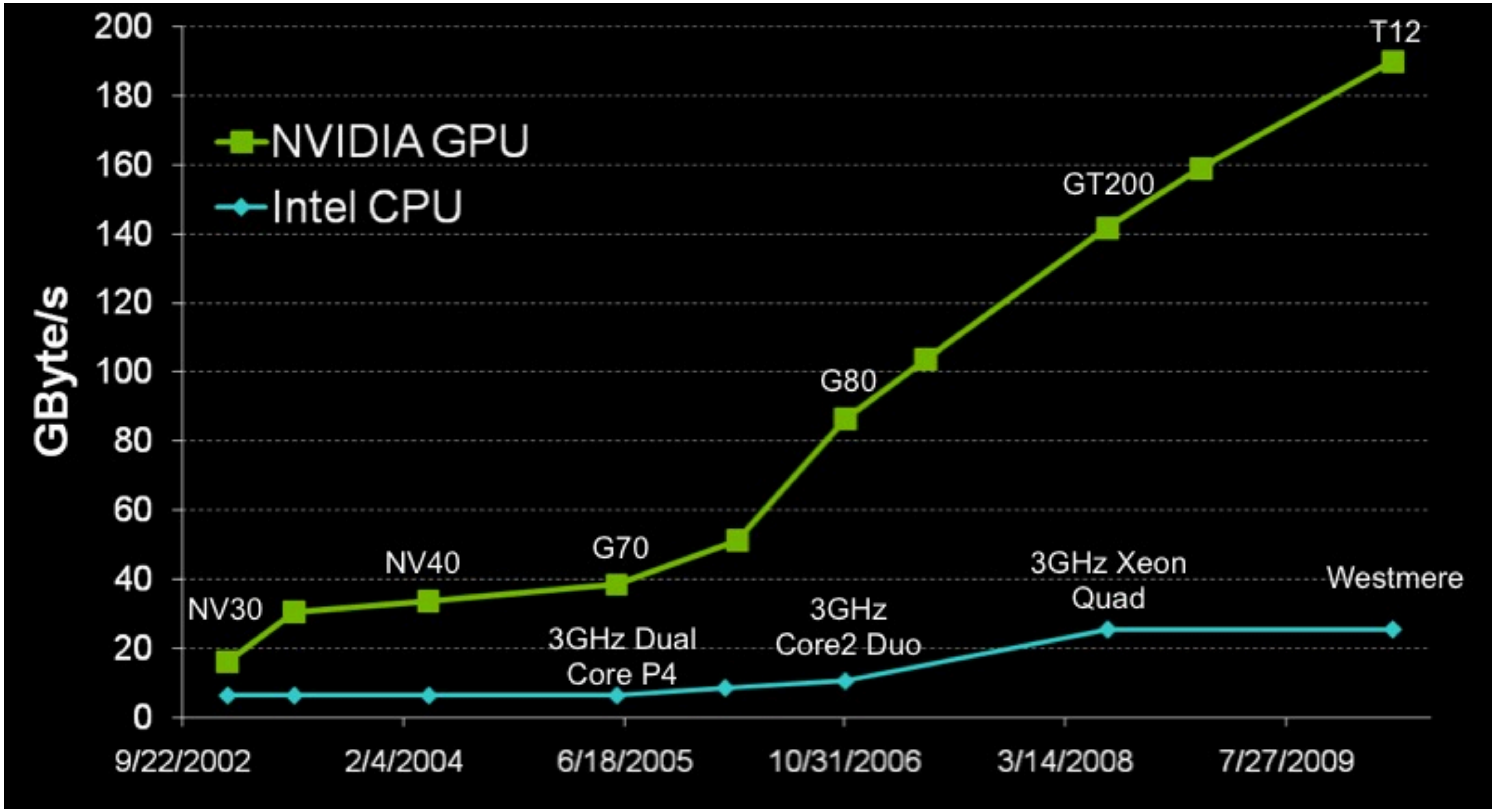
Why Massively Parallel Processing?

Computation Power of Graphic Processing Units



Why Massively Parallel Processing?

Memory Throughput of Graphic Processing Units



Why Massively Parallel Processing?

How can this be?

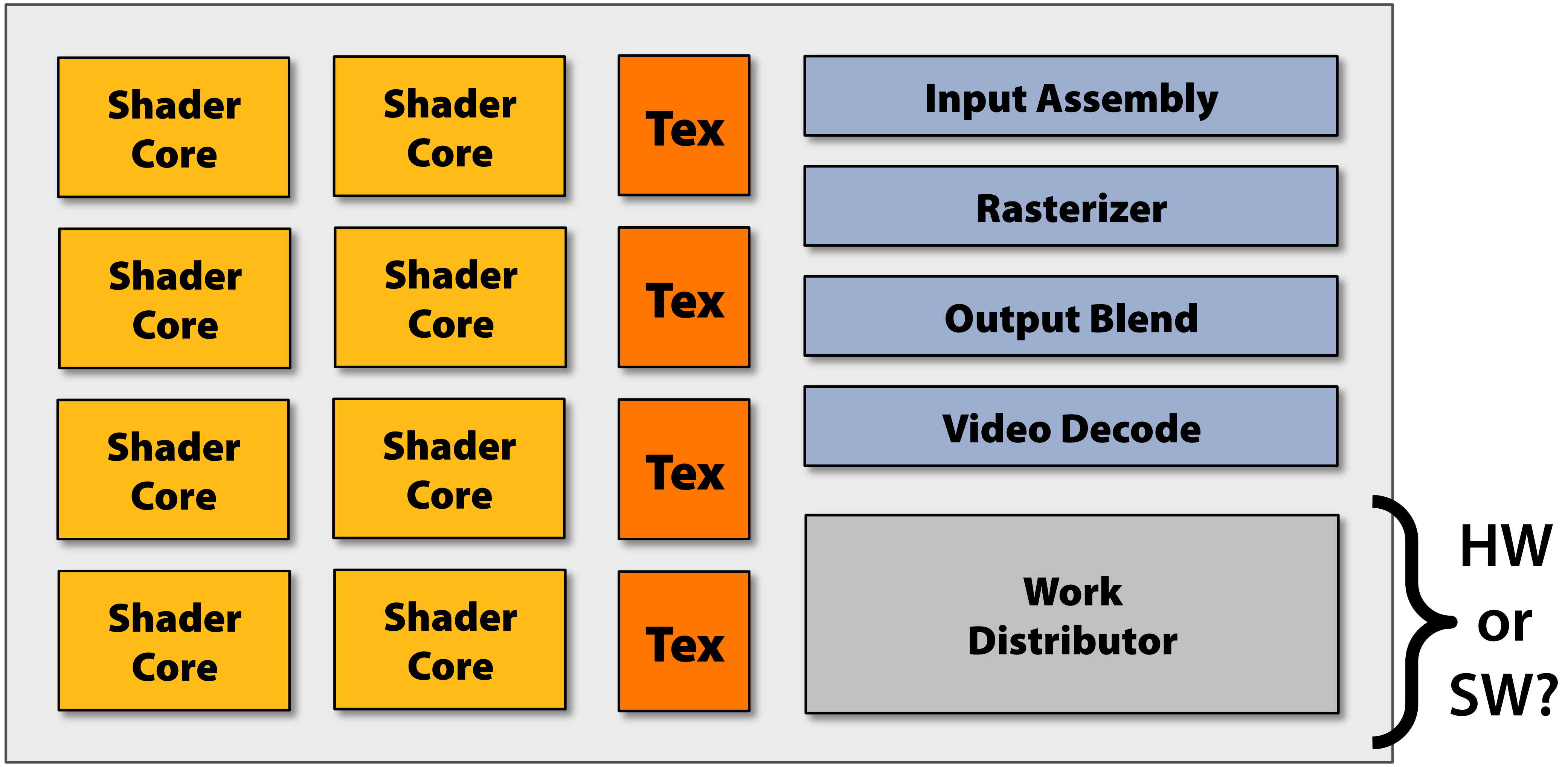
- Remove transistors dedicated to speed of a single stream of instructions
  - out-of-order execution, speculation, caches, branch prediction
  - CPU: minimize latency of an individual thread
- More memory bandwidth, more compute
  - Nothing else on the card! “Simple” design
  - GPU: maximize throughput of all threads.

SIGGRAPH2009 NEW ORLEANS

From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian
Stanford University

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

A diffuse reflectance shader

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


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

Independent, but no explicit parallelism

Compile shader

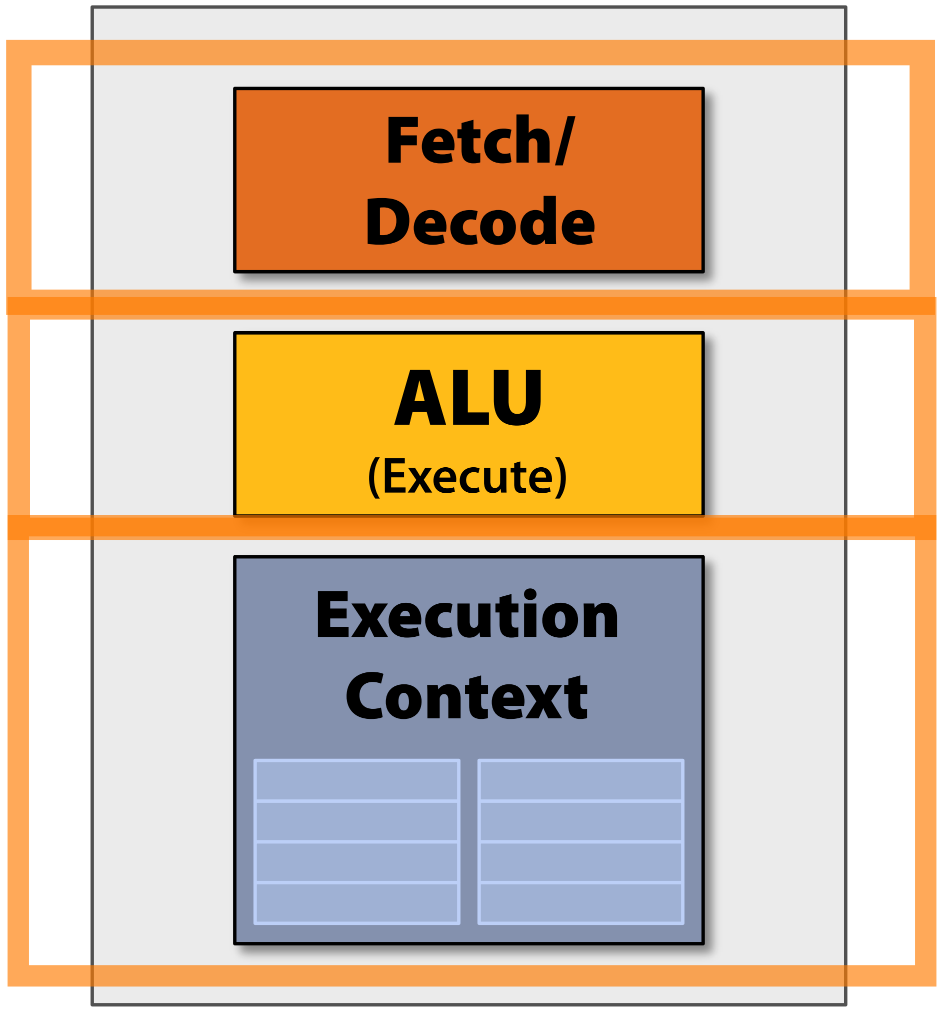
1 unshaded fragment input record

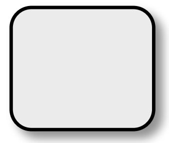
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

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

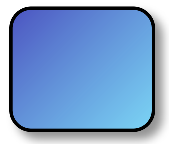
1 shaded fragment output record

Execute shader

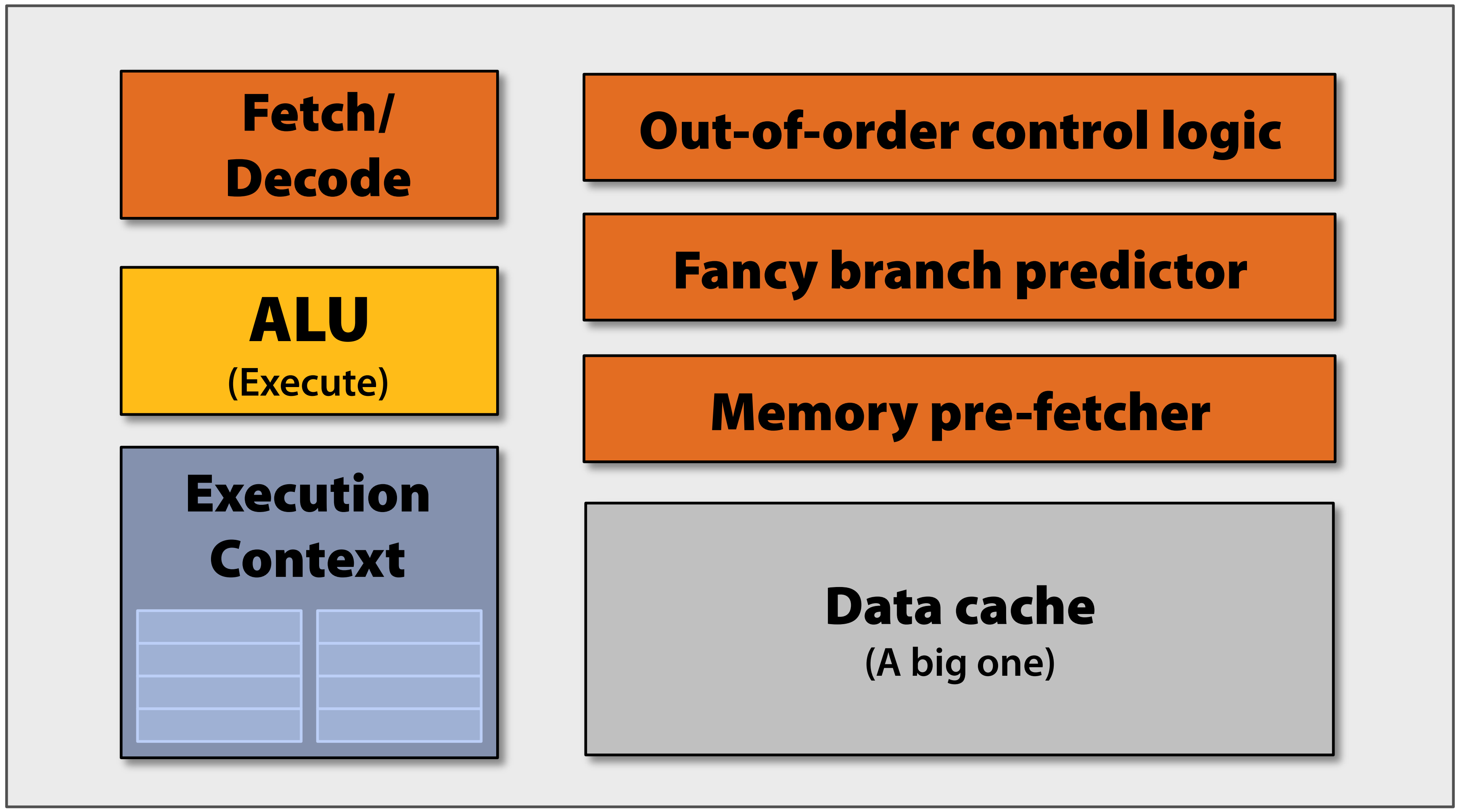




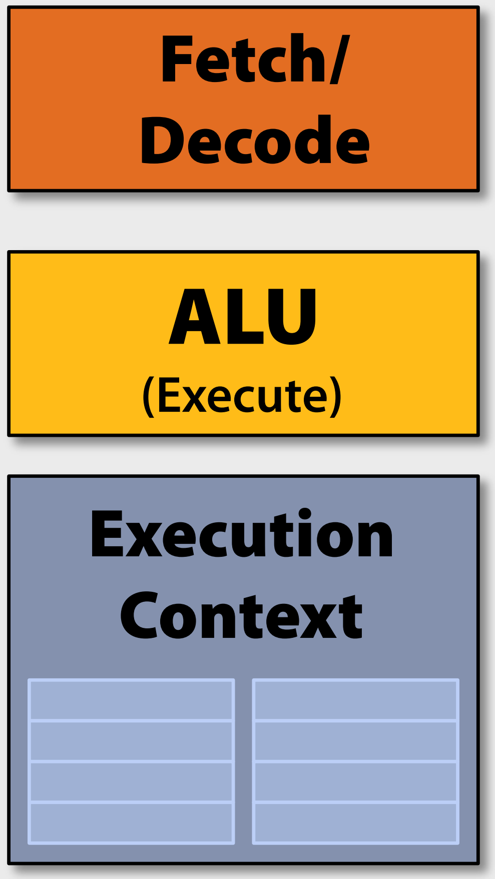
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



CPU-“style” cores



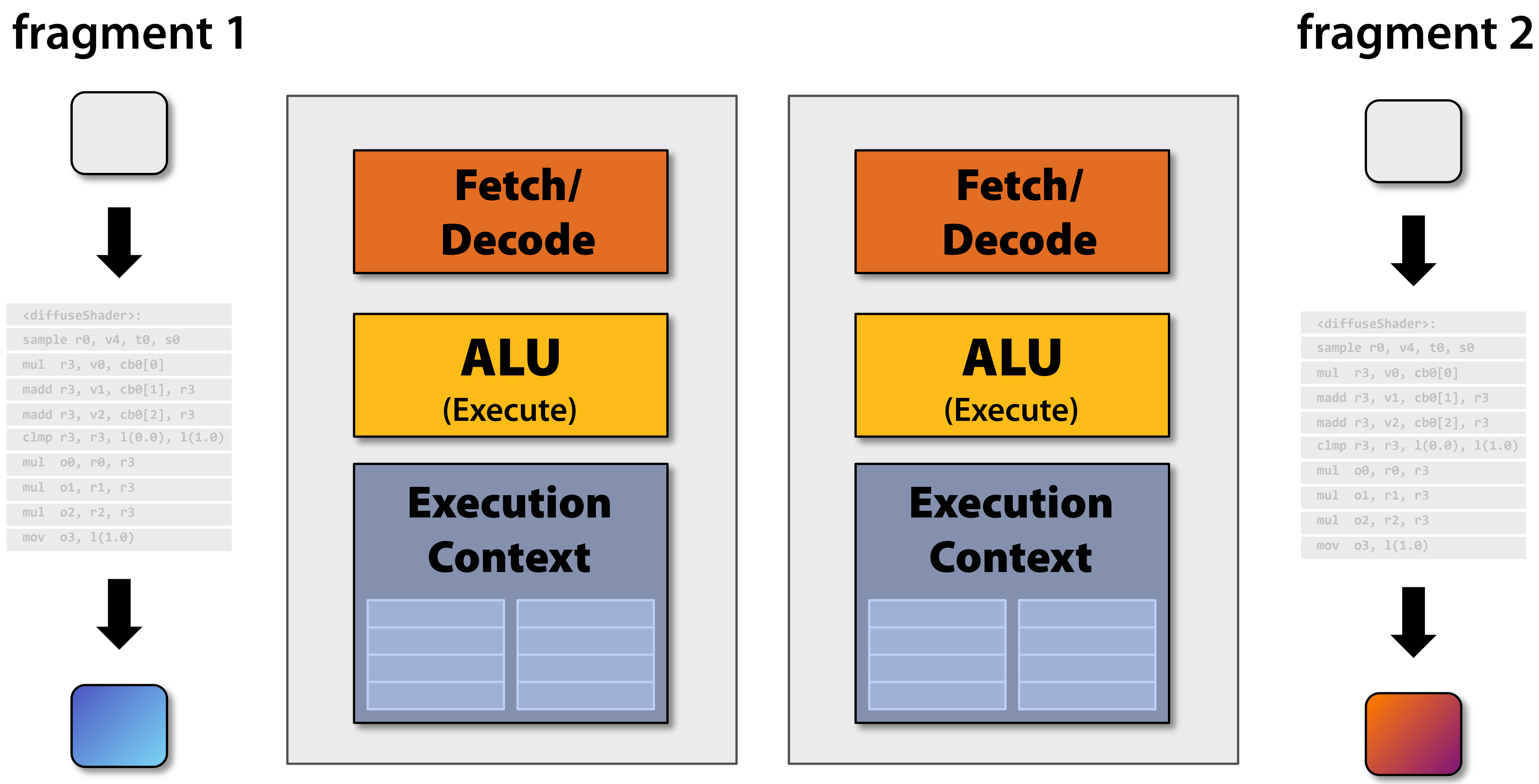
Slimming down



Idea #1:

Remove components that help a single instruction stream run fast

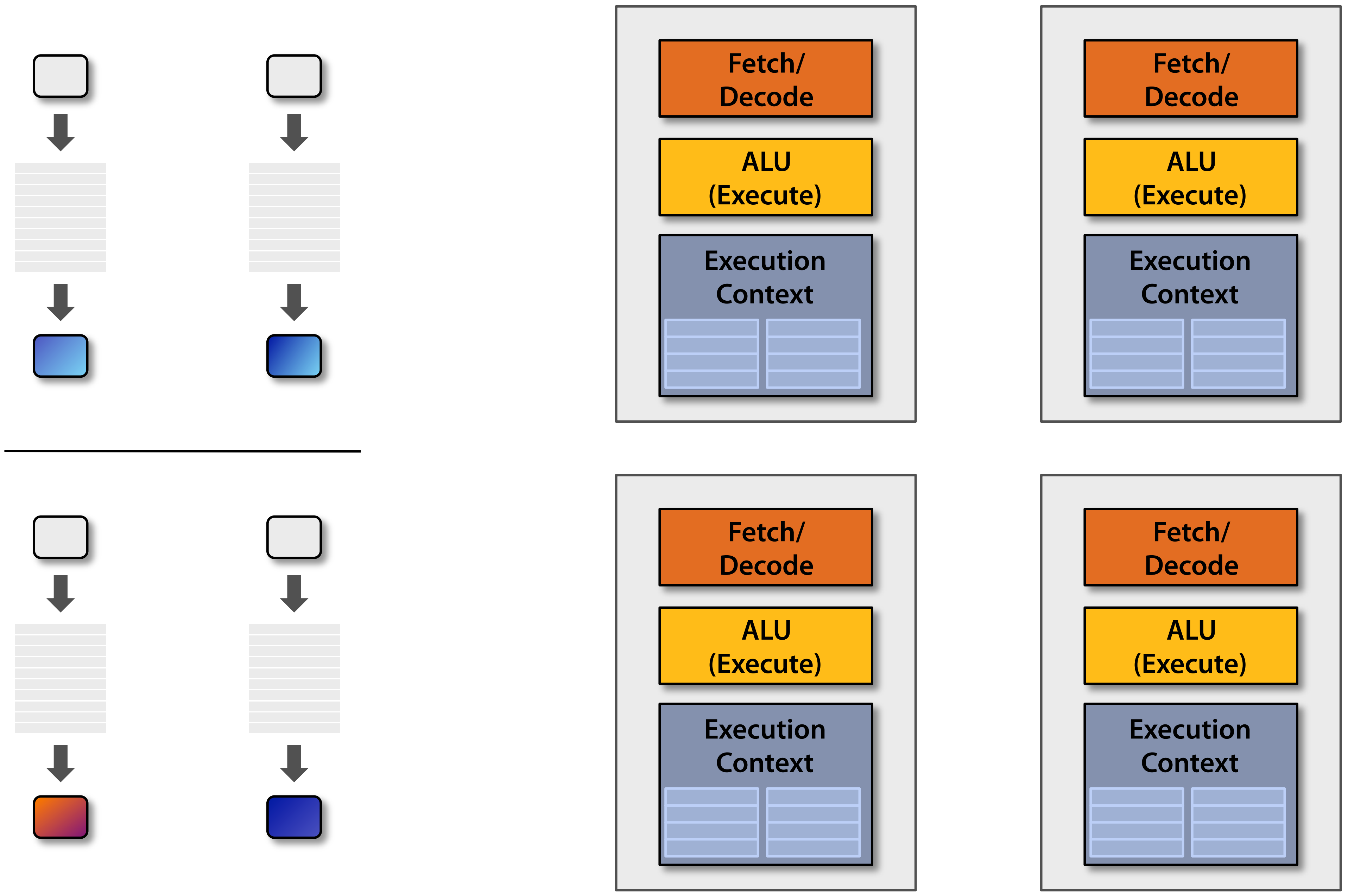
Two cores (two fragments in parallel)



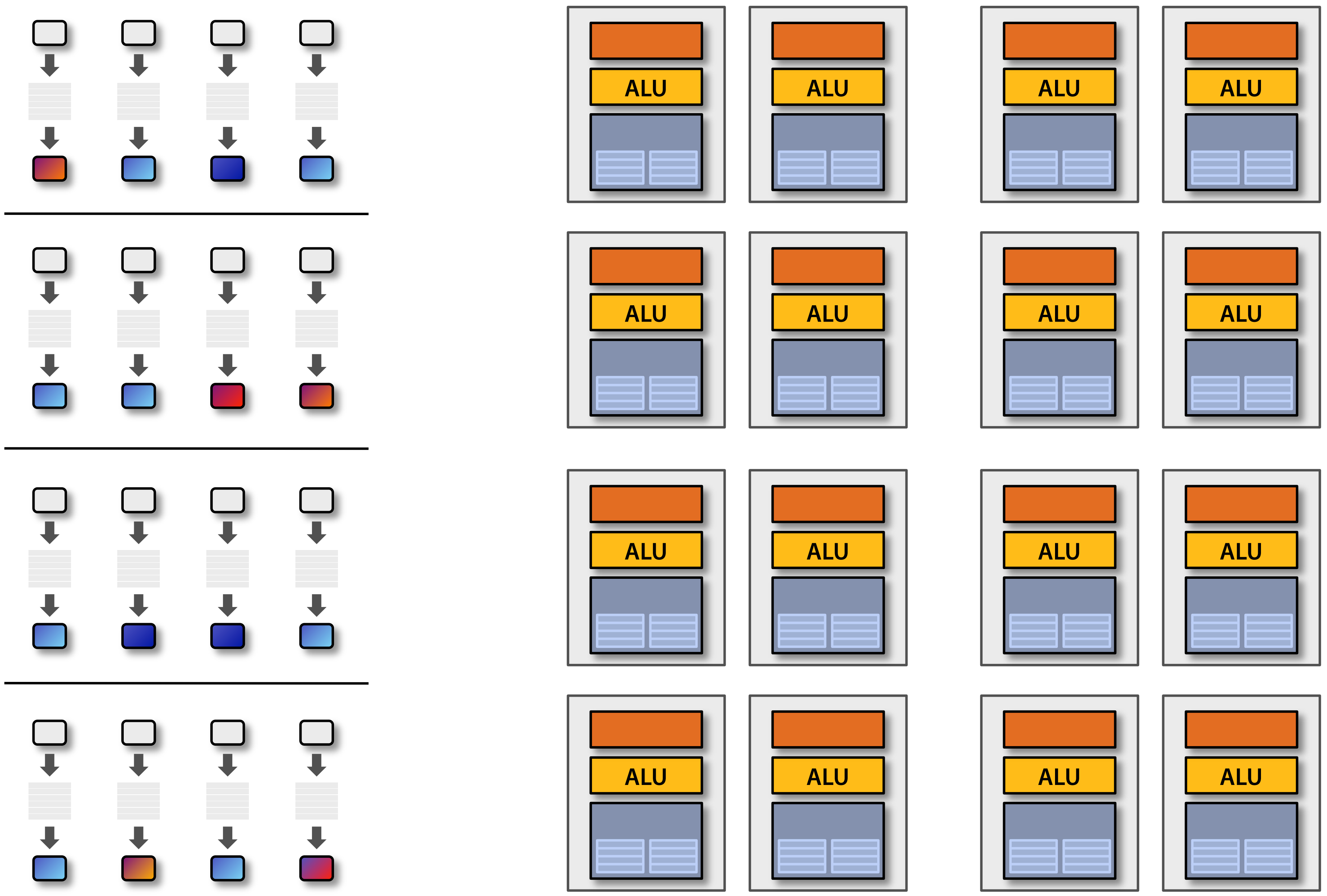
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

```
<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)
```

Four cores (four fragments in parallel)

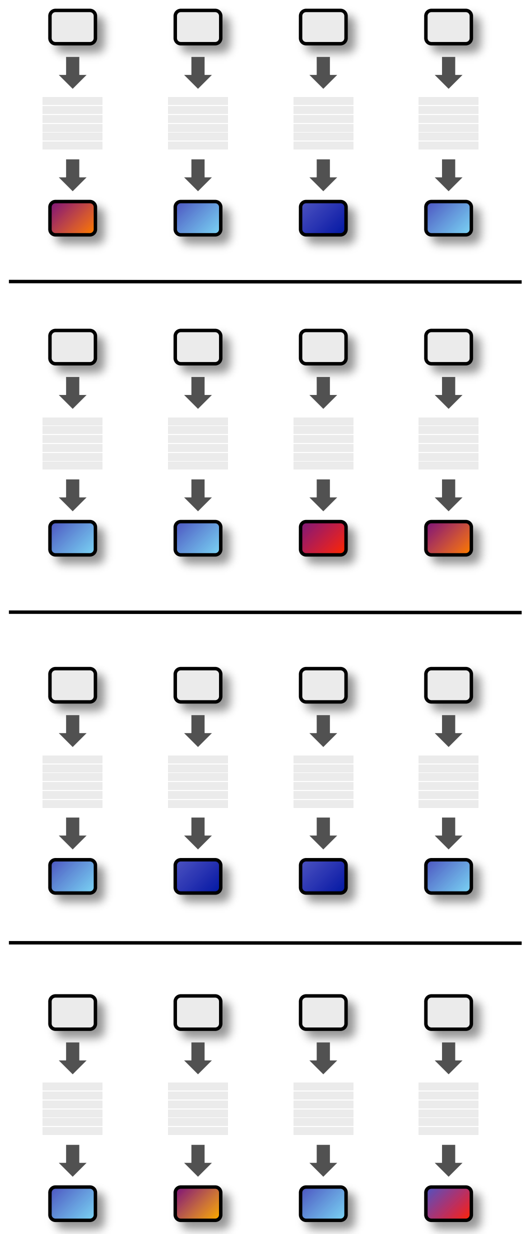


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

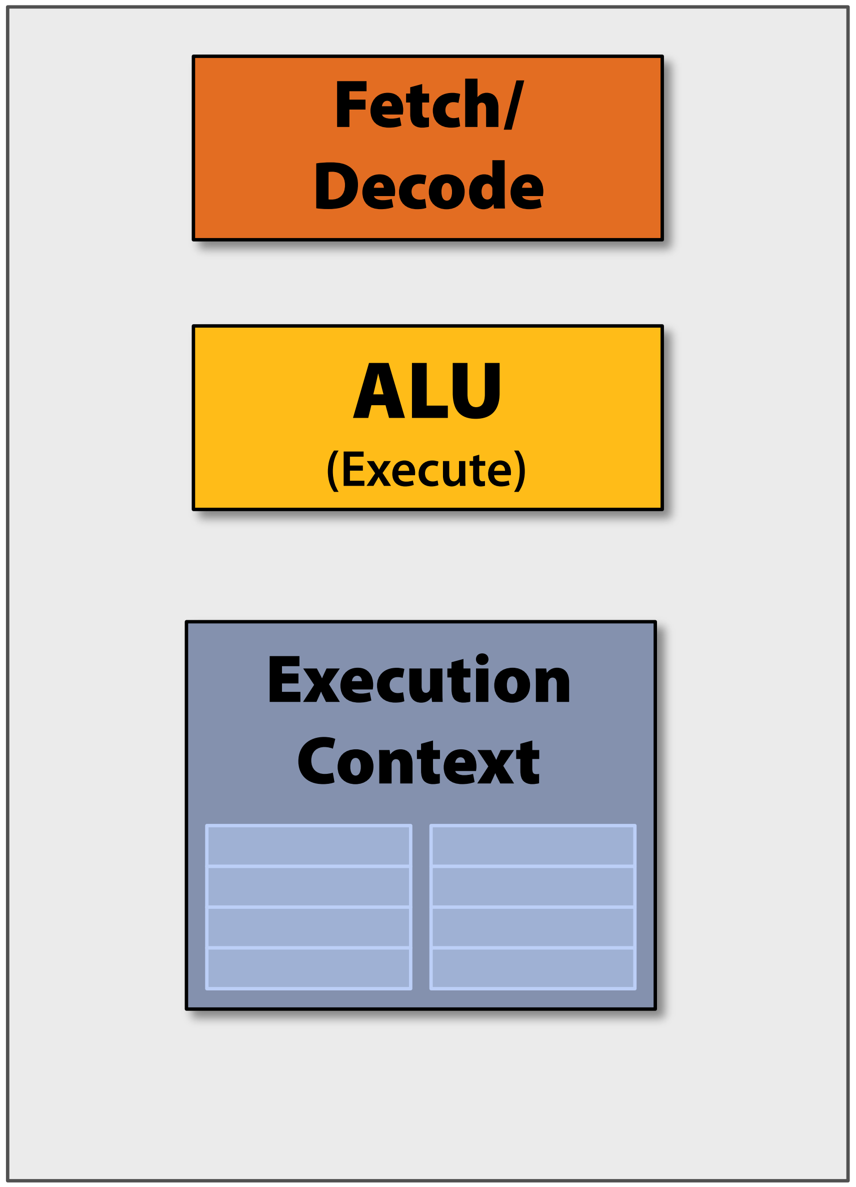
Instruction stream sharing



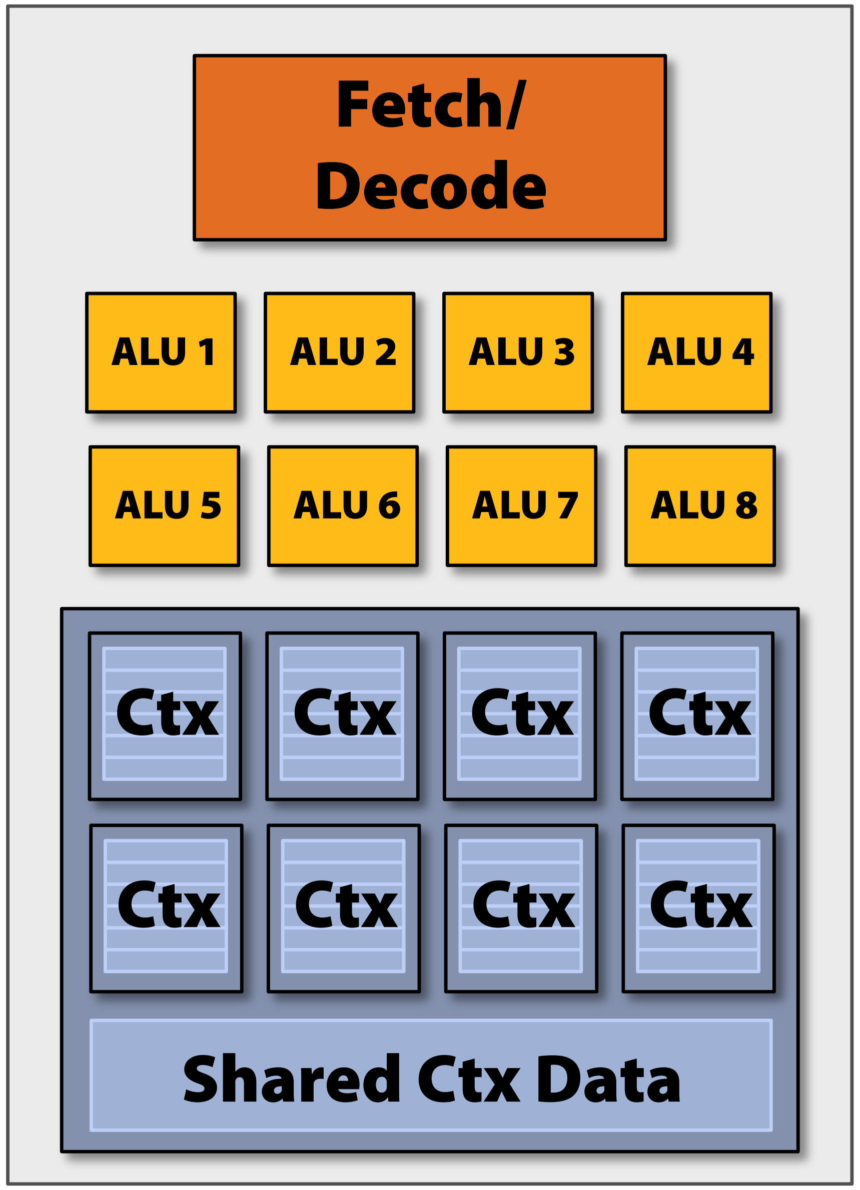
But… many fragments *should* be able to share an instruction stream!

```
<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

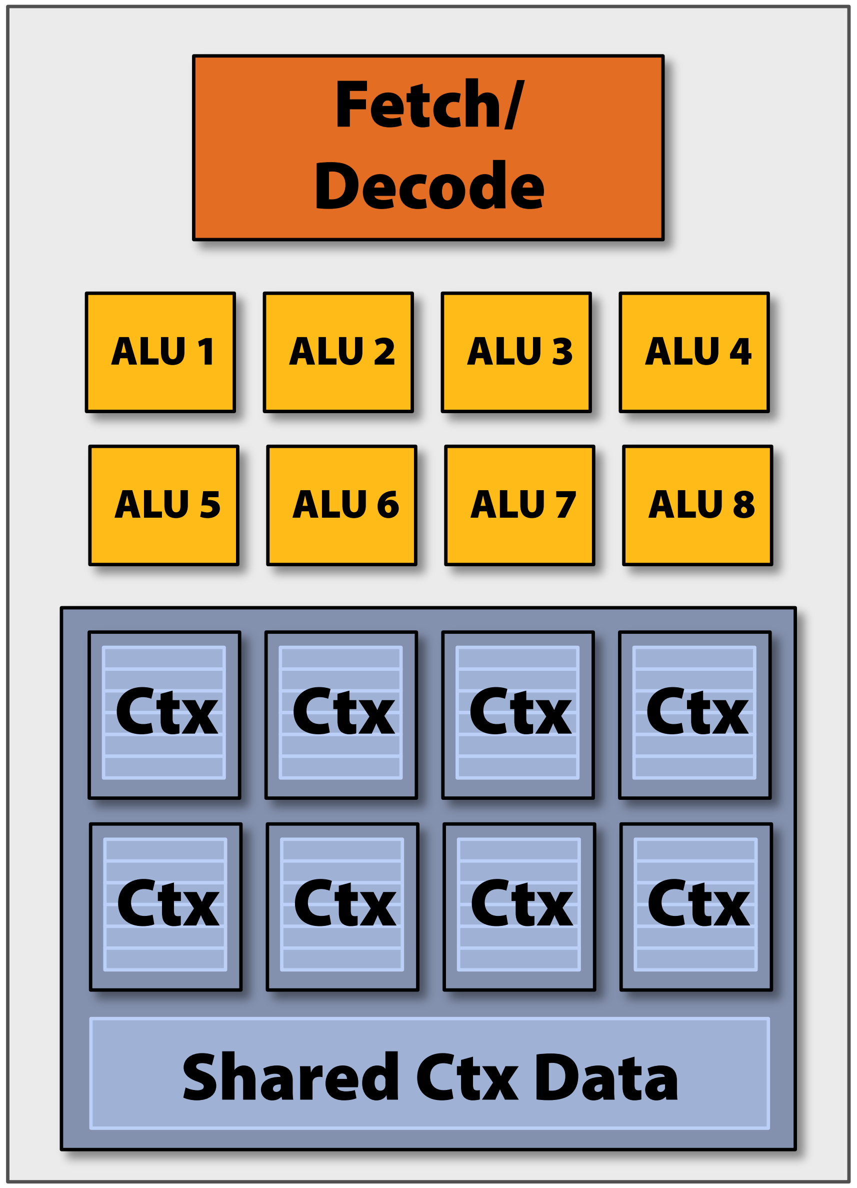


Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

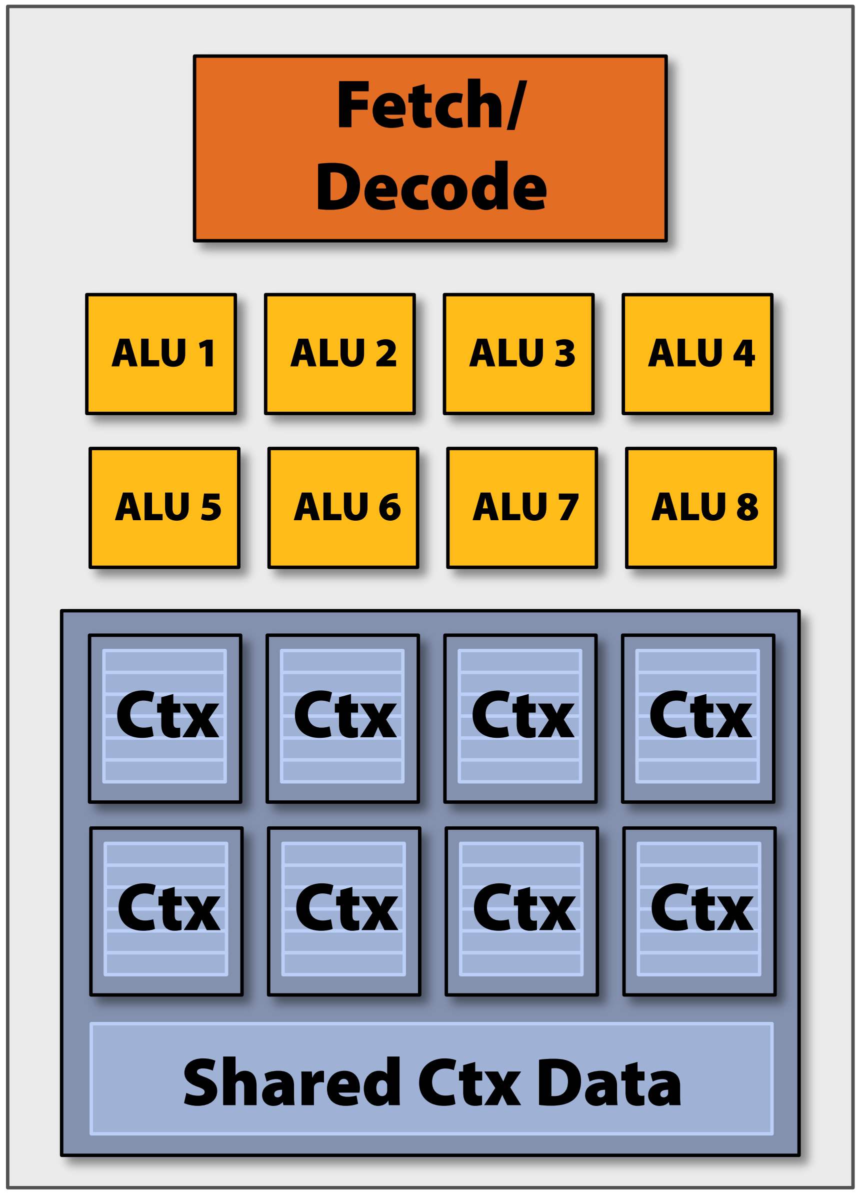


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

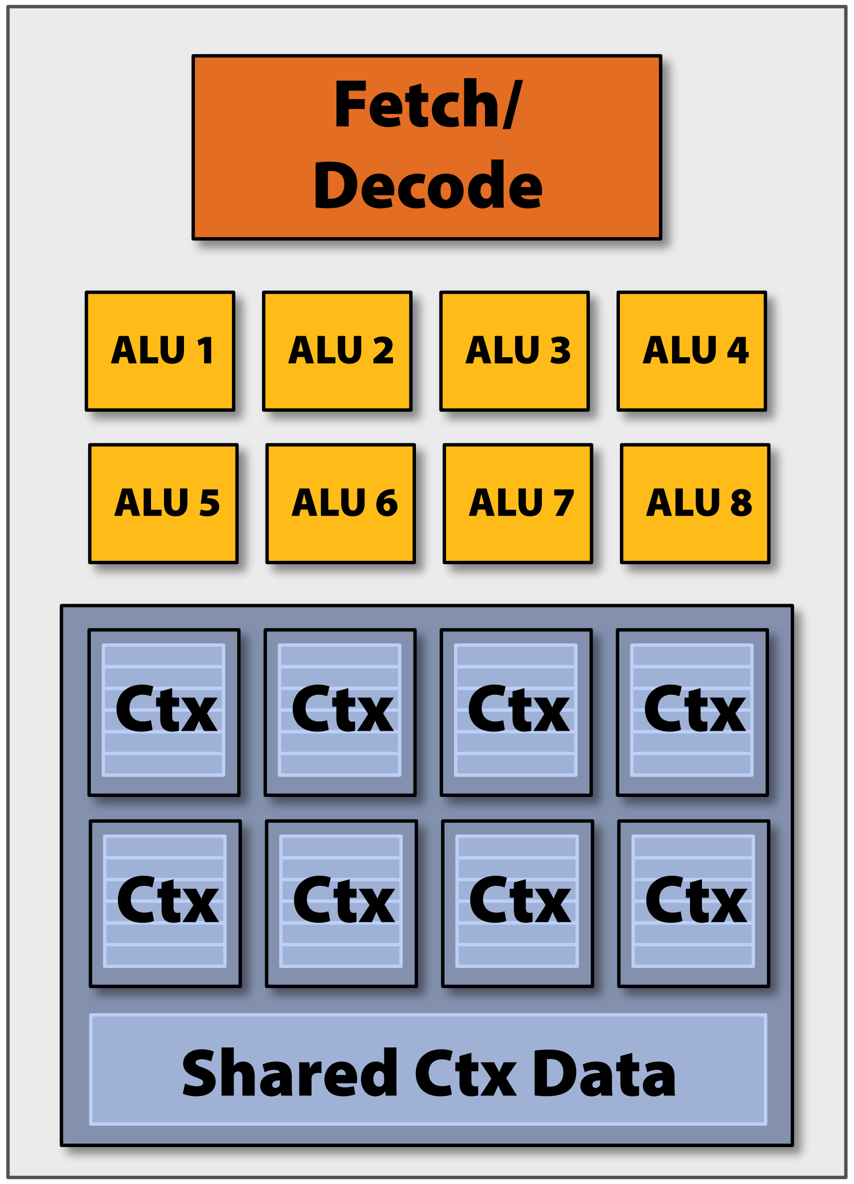


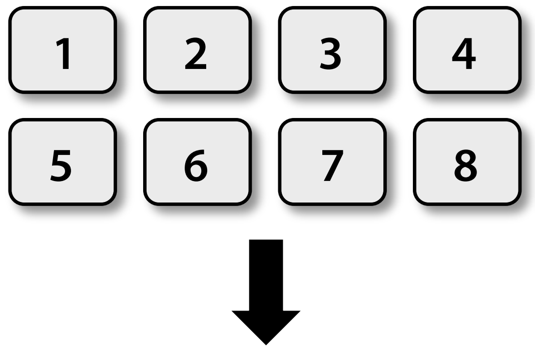
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

New compiled shader:

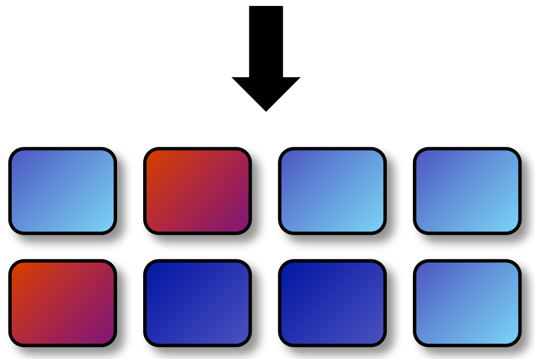
Processes 8 fragments using vector ops on vector registers

Modifying the shader

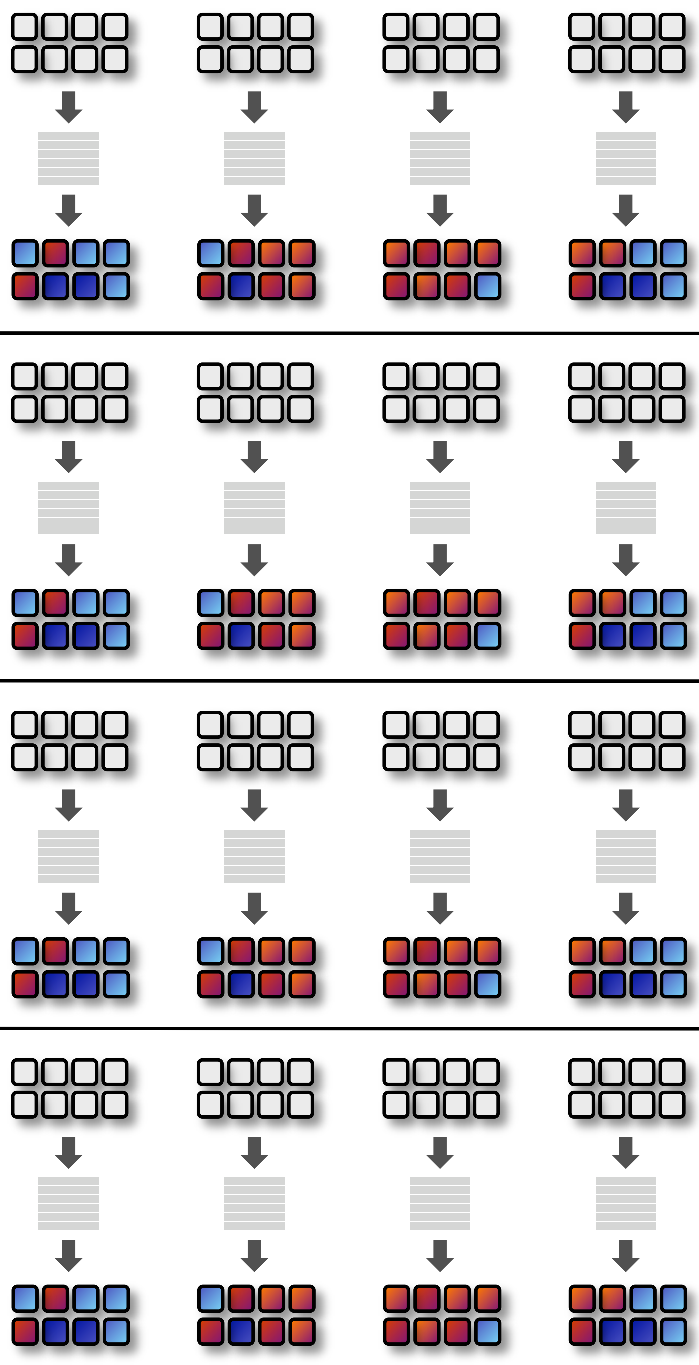


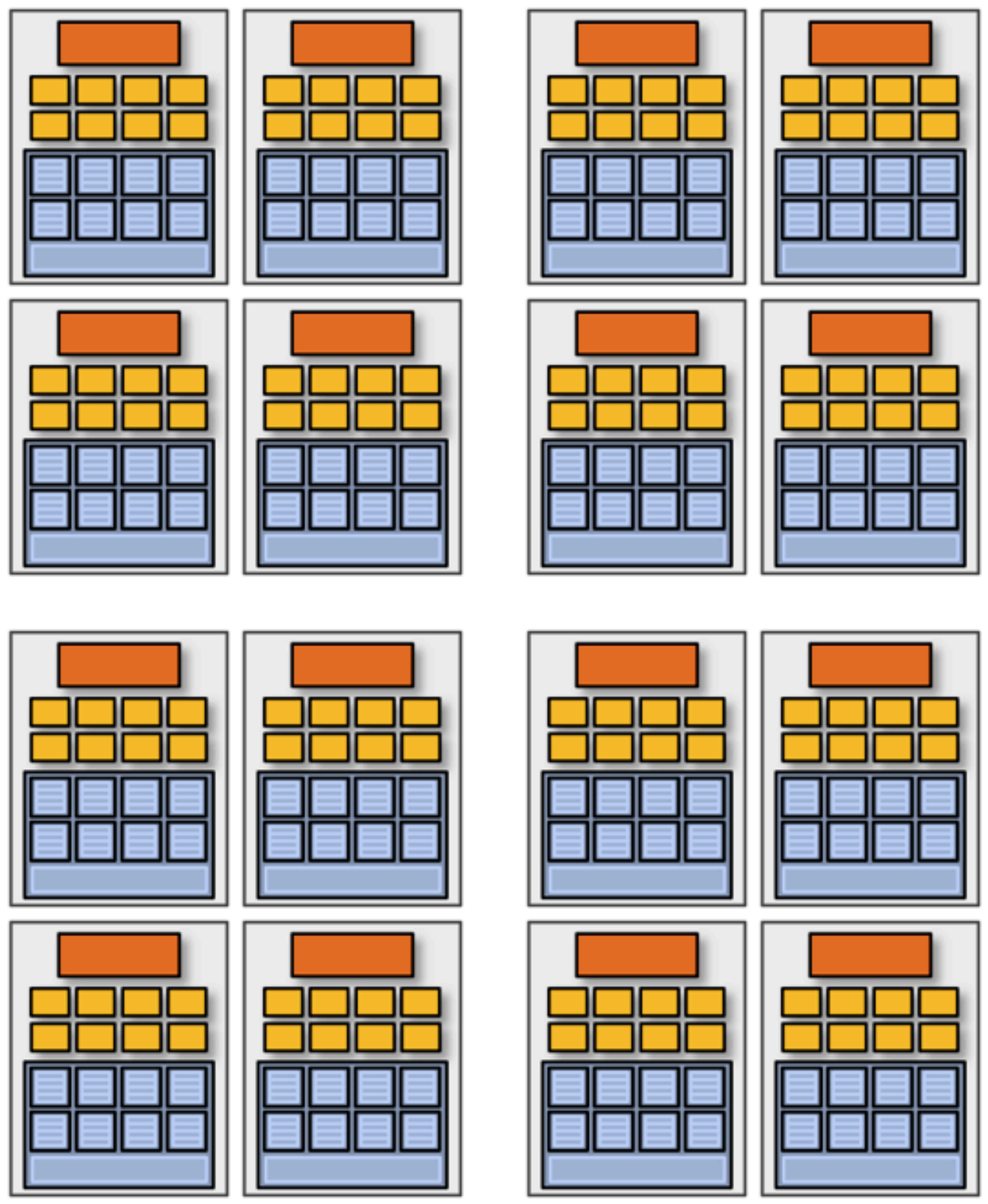


```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```



128 fragments in parallel

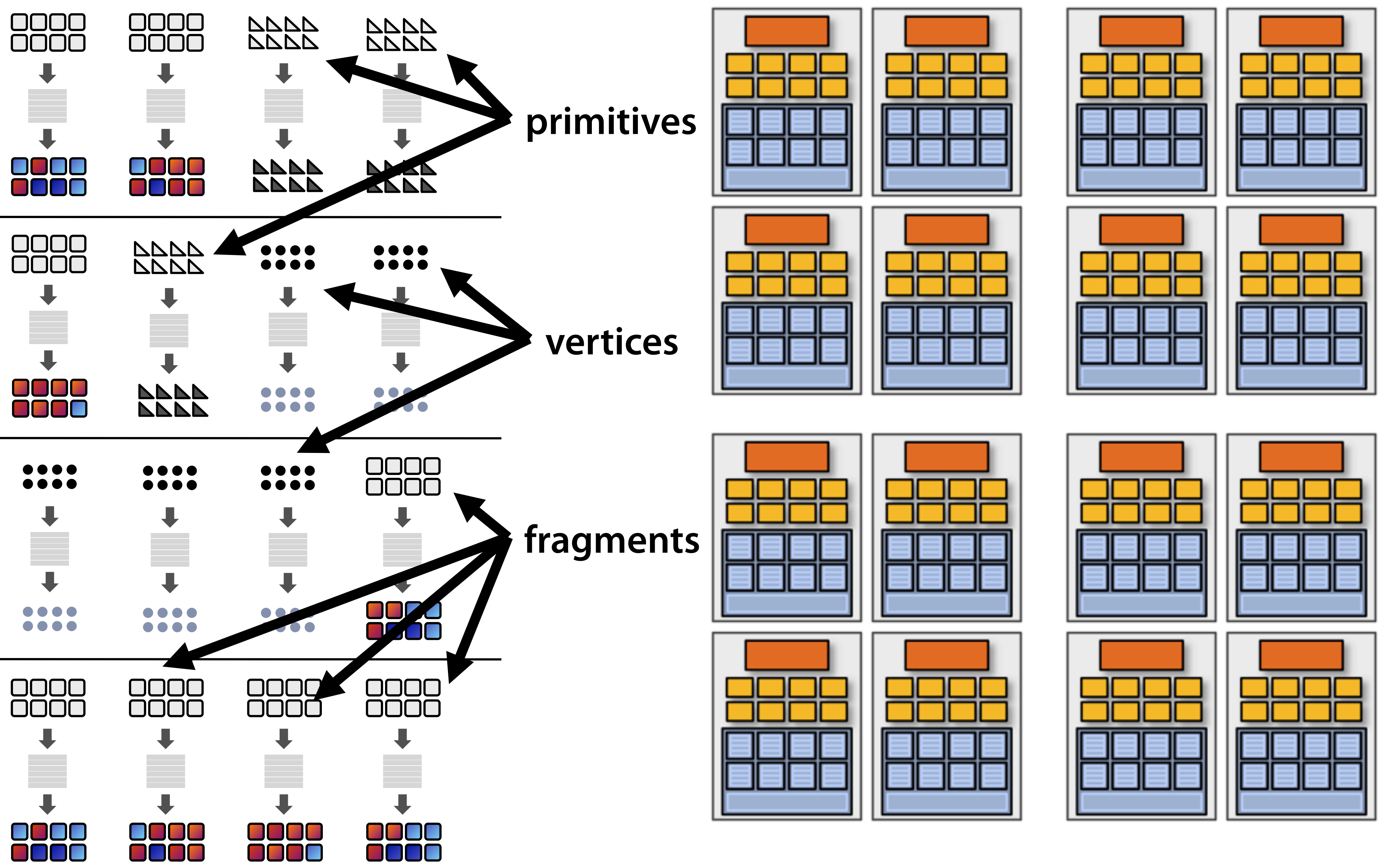




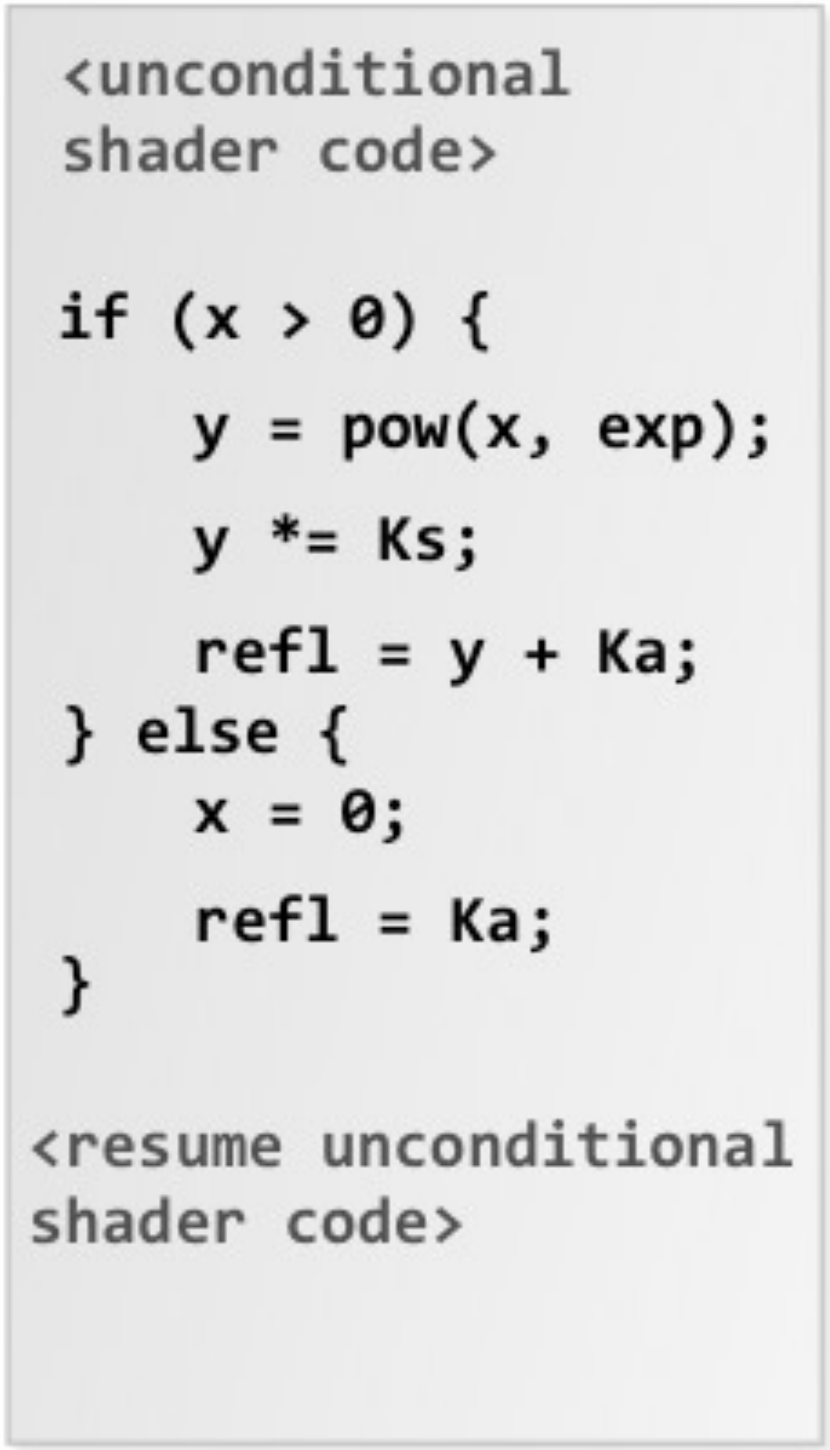
16 cores = 128 ALUs

= 16 simultaneous instruction streams

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

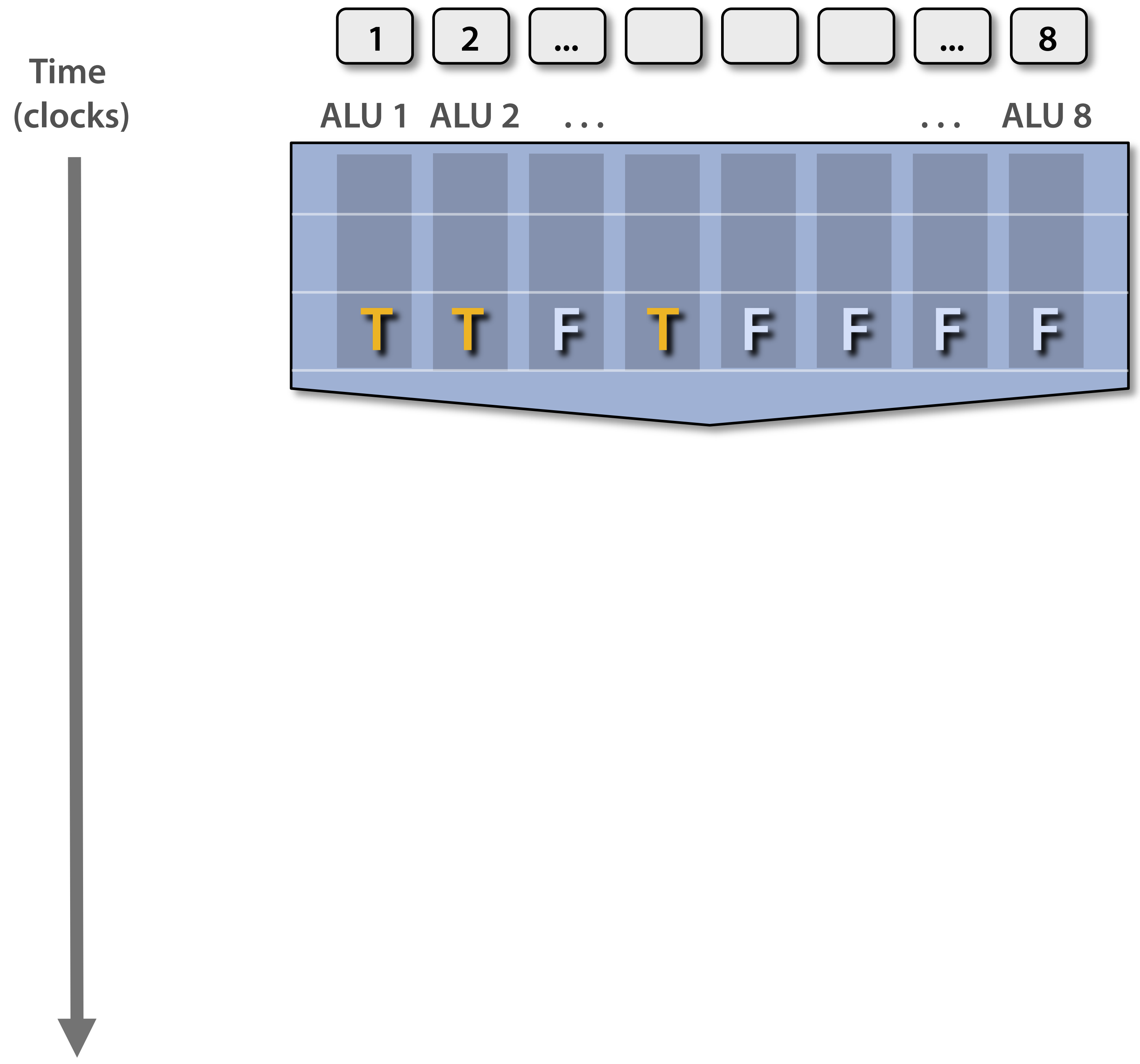


What is the problem?



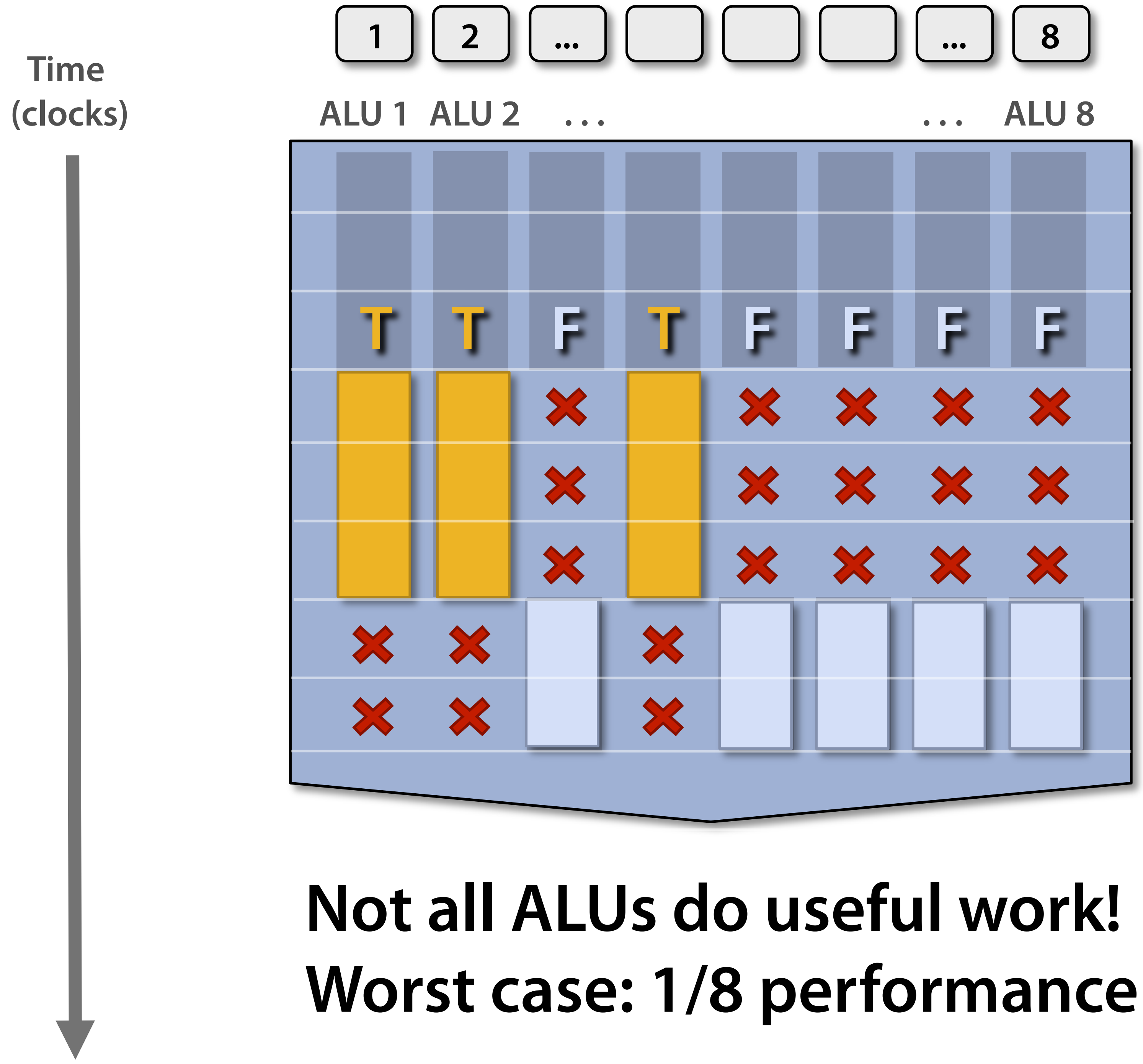
```
<unconditional
shader code>

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 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>
```

Plenty more intricacies!

No time now - See the "Beyond Programmable Shading" Siggraph talk

Think of a GPU as a multi-core processor optimized for maximum throughput: Many SIMD cores working together.

An efficient GPU workload...

Thousands on <u>independent</u> pieces of work

- Uses many ALUs on many cores

Amenable to instruction stream sharing

- Uses SIMD instructions

Compute-heavy:

- lots of math for each memory access

GF100 Architecture

30 SM’s on GTX480

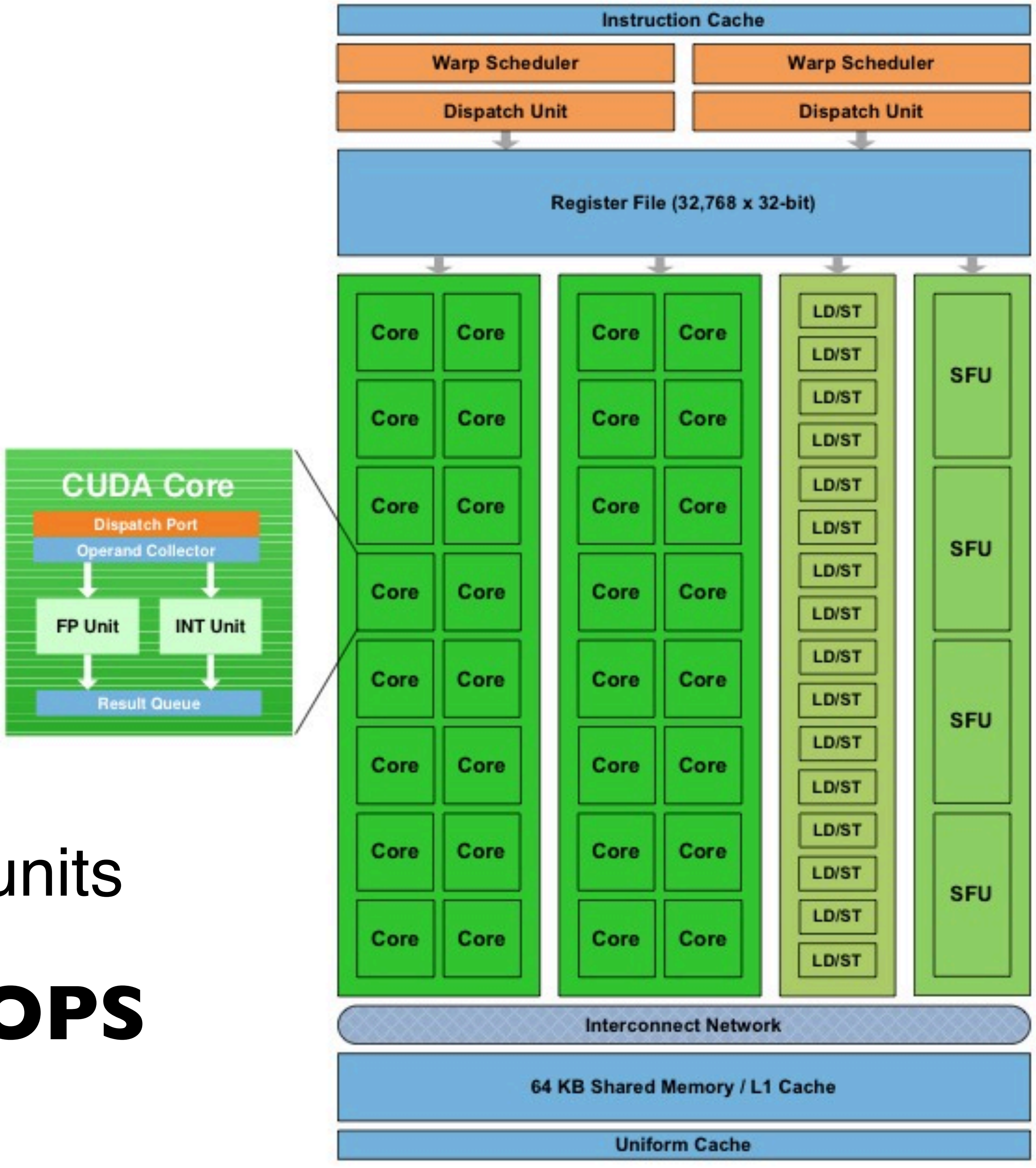
Resources:

- 2 x 16 “Cores”
- 16 Load/Store units
- 4 Special Functions

  Sin/Cos/Sqrt

- 16 Double Precision units

1.44 Terra FLOPS



Fermi Streaming Multiprocessor (SM)

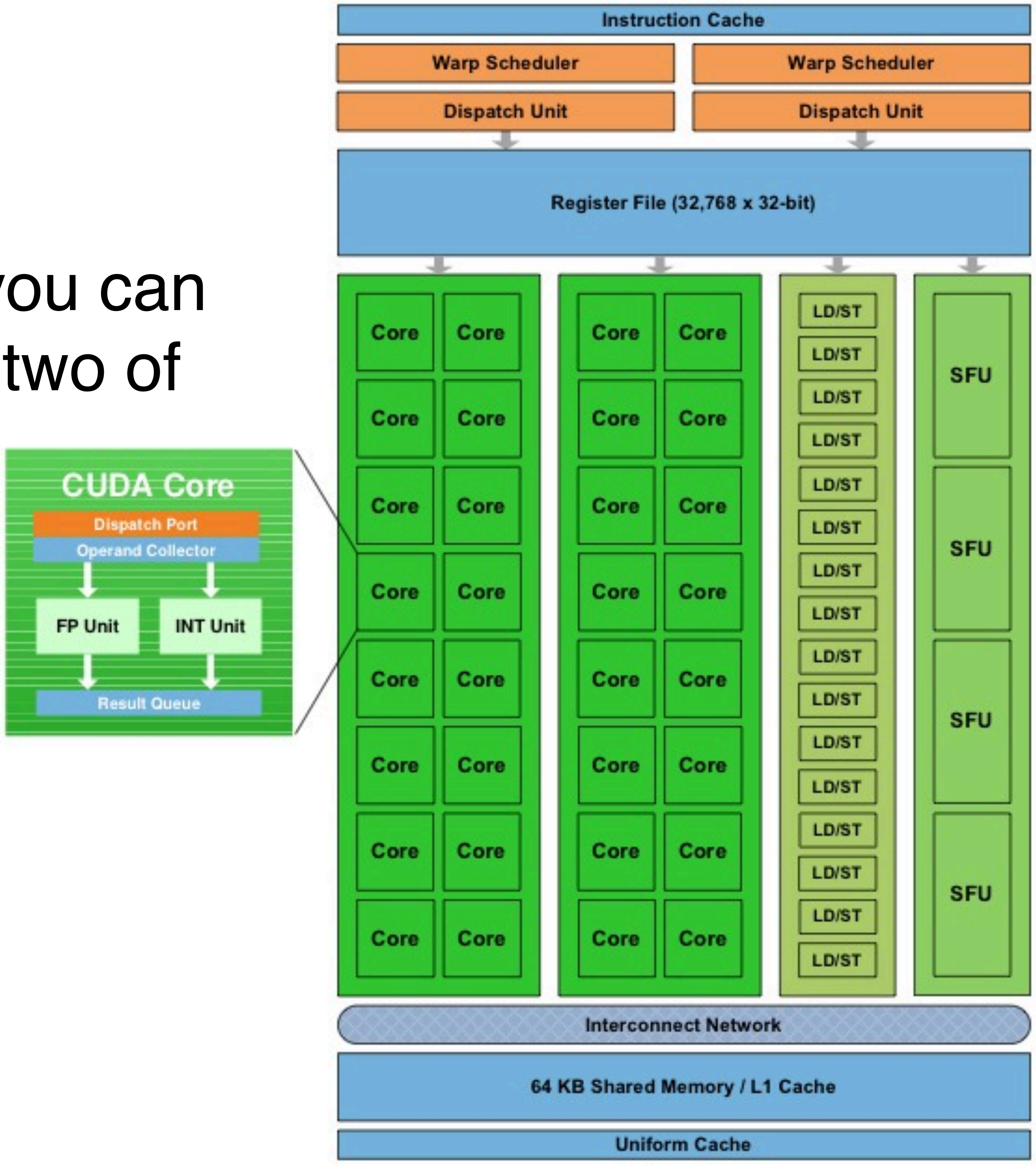
GF100 Architecture

Mental Model:

“On every clock cycle, you can assign an instruction to two of these resources”

Resources:

- 2 x 16 “Cores”
- 16 Load/Store units
- 4 Special Functions
- 16 Double Precision units



Fermi Streaming Multiprocessor (SM)

GPGPU

General Purpose Computing on GPUs

CUDA Stream Programming

C/C++ extended with:

- Kernels - function executed N times in parallel
- CPU/GPU Synchronization
- GPU Memory Management

CUDA Stream Programming

Example: Vector Addition Kernel

Device Code

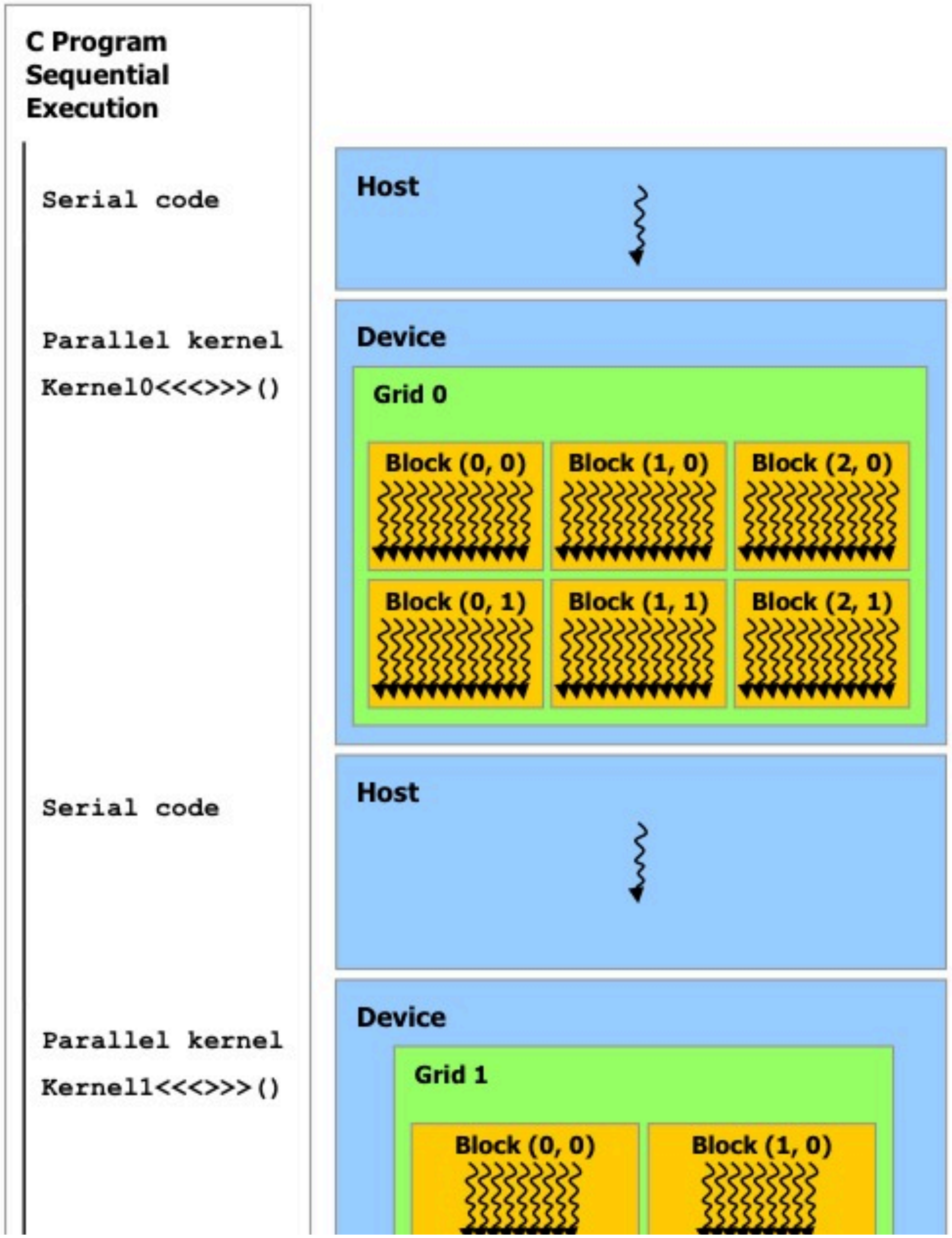
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```


C Program Sequential Execution
Serial code
Parallel kernel
Kernel0<<<>>>()
Serial code
Parallel kernel
Kernel1<<<>>>()
Host
Device
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Host
Device
Grid 1
Block (0, 0)
Block (1, 0)

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

```
Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

```
Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

```
Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Example: Shuffling Data

```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
   new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

Host Code

Alternatives

OpenCL

- Attempts to be OpenGL for GPGPU
- Almost identical to CUDA

Need for a higher level languages

- Jacket for MATLAB
- PyCUDA

The Future

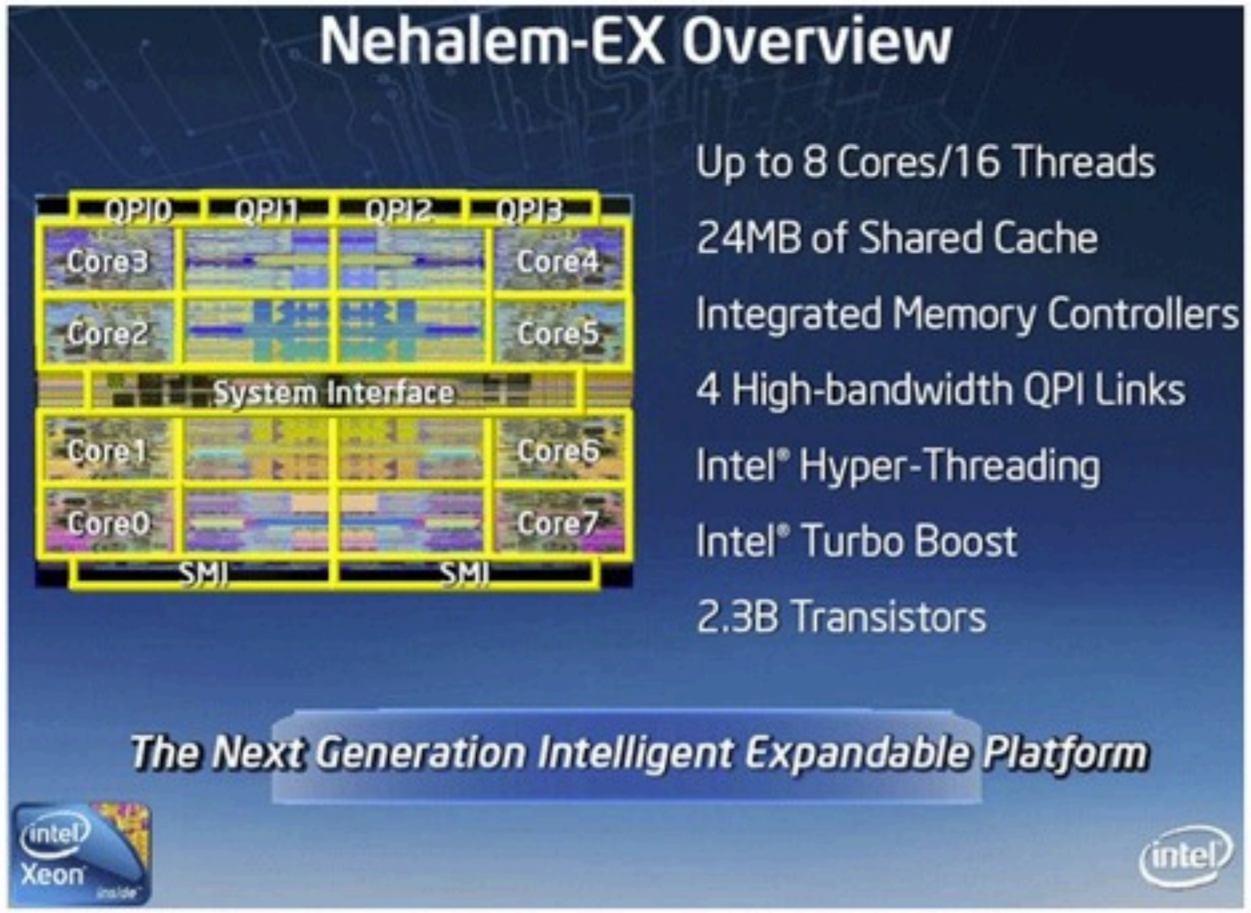
Massively Multi-Core Processors

The Future

MultiCore is Dead

Intel's new Nehalem-EX CPUs rock servers with eight cores, 16 threads, infinite sex appeal

By Tim Stevens posted May 27th 2009 8:06AM



Nehalem-EX Overview

- Up to 8 Cores/16 Threads
- 24MB of Shared Cache
- Integrated Memory Controllers
- 4 High-bandwidth QPI Links
- Intel® Hyper-Threading
- Intel® Turbo Boost
- 2.3B Transistors

The Next Generation Intelligent Expandable Platform

What's that, you have an array of six-core CPUs in your rack? That is so last year. You're going to feel pretty foolish when all the cool admins start popping eight-core chips up in their closets this fall. That's the number on offer in Intel's latest, the Nehalem-EX. It's an evolution of the architecture that some of you may be spinning in your Core i7 machines, but boosted to support up to 16 threads and 24MB of cache. 2.3 billion transistors make the magic happen here, and Intel is pledging a nine-times improvement in memory bandwidth over the Xeon 7400. Chips are set to start hitting sockets sometime later this year, and while nobody's talking prices, staying hip in the enterprise server CPU crowd doesn't come cheap.

"You have an array of six-core CPUs in your rack? You're going to feel pretty stupid when all the cool admins start popping eight-core chips.

Many-Core?

Number of cores so large that:

- Traditional caching models don’t work

    Cannot keep coherent cache

- Network on a chip?

Memory Models & More

Haven't even *touched* it

- Coherent and Uncoherent caches
- Uniform vs Non-Uniform Memory Access? TM?
- Special Purpose Hardware?

Schedulers?

Programming Languages

The Future

Intel Nehalem

Up to 12 cores

30% lower power usage

Similar programming abstractions:

- 12 cores, each with 128-bit wide SIMD units (SSE)

Intel SandyBridge

256-bit wide SIMD units (AVX)

Parallelism Everywhere

Putting all those transistors to use

- Many ALUs
- Many Cores
- Intricate Cache Hierarchies
- **Very difficult to program**

Graphics is *way ahead of the game*

Learn More:

“The Landscape of Parallel Computing Research”

http://view.eecs.berkeley.edu/wiki/Main_Page

Thank you!

Acknowledgements

Kayvon Fatahalian

- Many of these slides are inspired by or copied from him

Mike Houston

- CS448s “Beyond Programmable Shading”

Jared Hobernock & David Tarjan

- CS193g “Programming massively parallel processors”
- (I TA’d this last quarter)