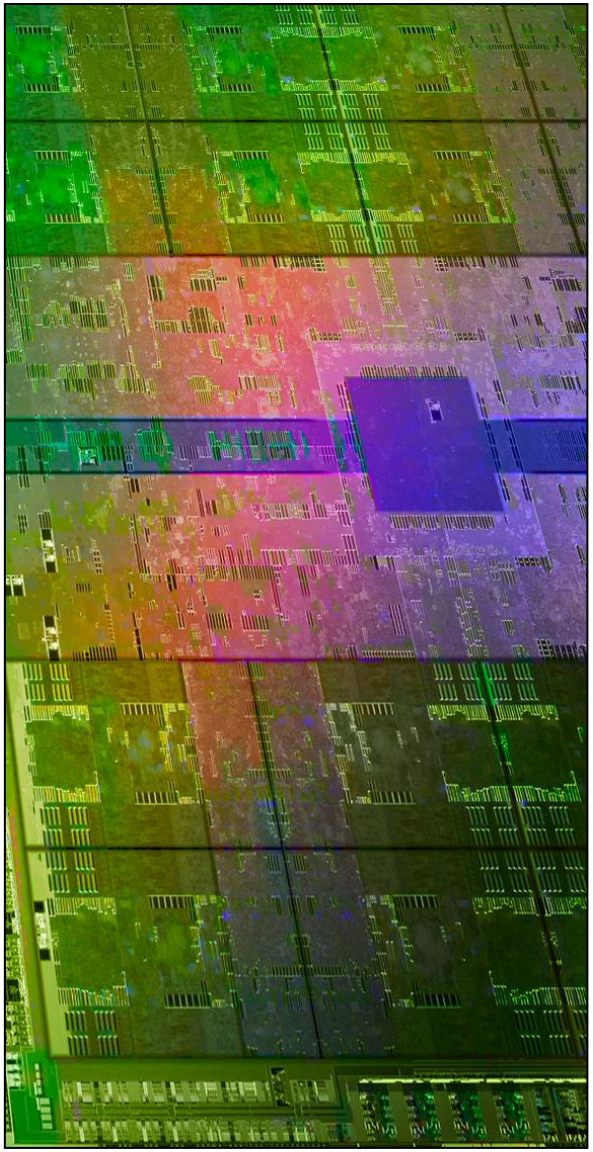
AUEB COMPUTER GRAPHICS GROUP

COMPUTER GRAPHICS COURSE

The GPU



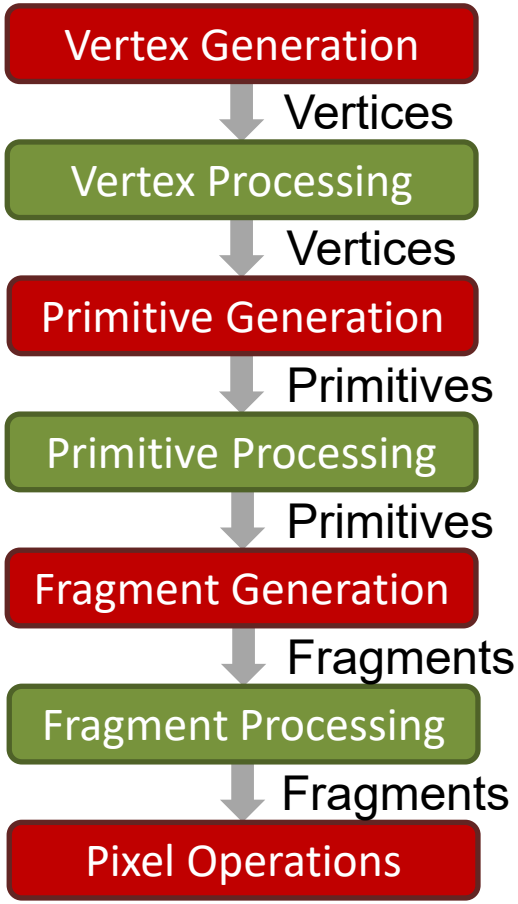
Andreas Vasilakis

The Hardware Graphics Pipeline (1)

- Essentially maps the above procedures to hardware stages
- Certain stages are optimally implemented in fixed-function hardware (e.g. rasterization)
- Other tasks correspond to programmable stages

The Hardware Graphics Pipeline (2)

- Vertex attribute streams are loaded onto the graphics memory along with
  - Other data buffers (e.g. textures)
  - Other user-defined data (e.g. material properties, lights, transformations, etc.)



Fixed stage

Programmable stage

Shaders

- A shader is a user-provided program that implements a specific stage of a rendering pipeline

- Depending on the rendering architecture, shaders my be designed and compiled to run in software renderers (on CPUs) or on H/W pipelines (GPU)

GPU Shaders

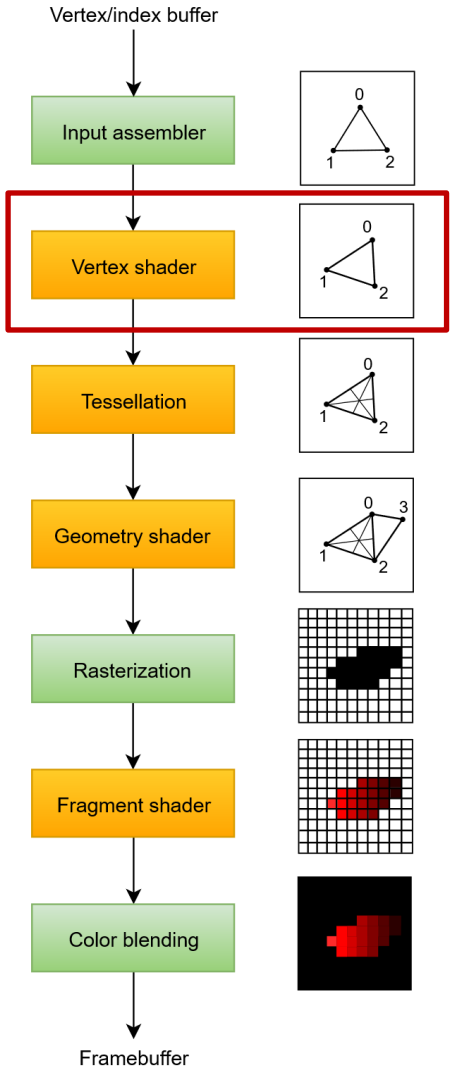
- The GPU graphics pipeline has several programmable stages
- A shader can be compiled, loaded and made active for each one of the programmable stages
- A collection of shaders, each one corresponding to one stage comprise a shader program
- Multiple programs can be interchanged and executed in the multiprocessor cores of a GPU

The Lifecycle of Shaders

- Shaders are loaded as source code (GLSL, HLSL etc)
- They are compiled and linked to shader programs by the driver
- They are loaded as machine code in the GPU
- Shader programs are made current (activated) by the host API (OpenGL, Direct3D etc)
- When no longer needed, they are released

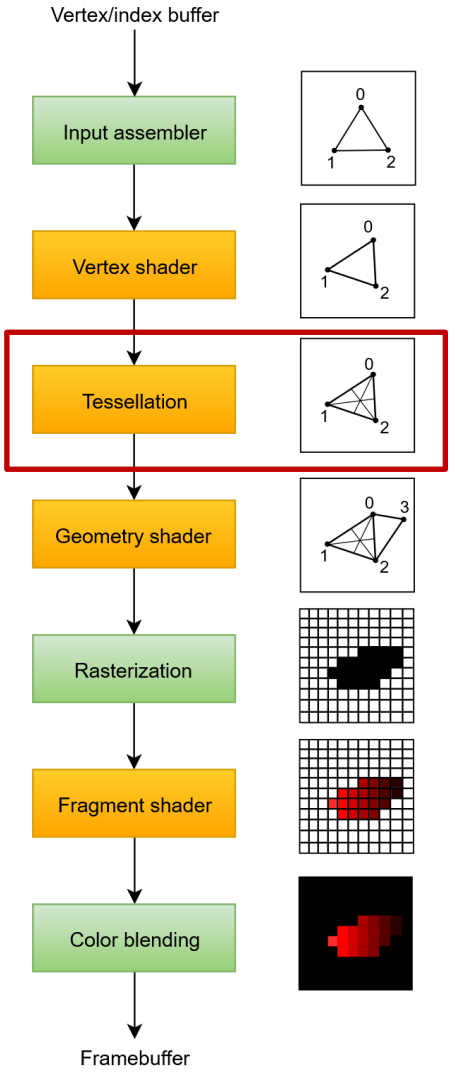
Programmable Stages – Vertex Shader

- Executed:
  - Once per input vertex
- Main role:
  - Transforms input vertices
  - Computes additional per vertex attributes
  - Forwards vertex attributes to the primitive assembly and rasterization (interpolation)
- Input:
  - Primitive vertex
  - Vertex attributes (optional)
- Output:
  - Transformed vertex (mandatory)
  - “out” vertex attributes (optional)



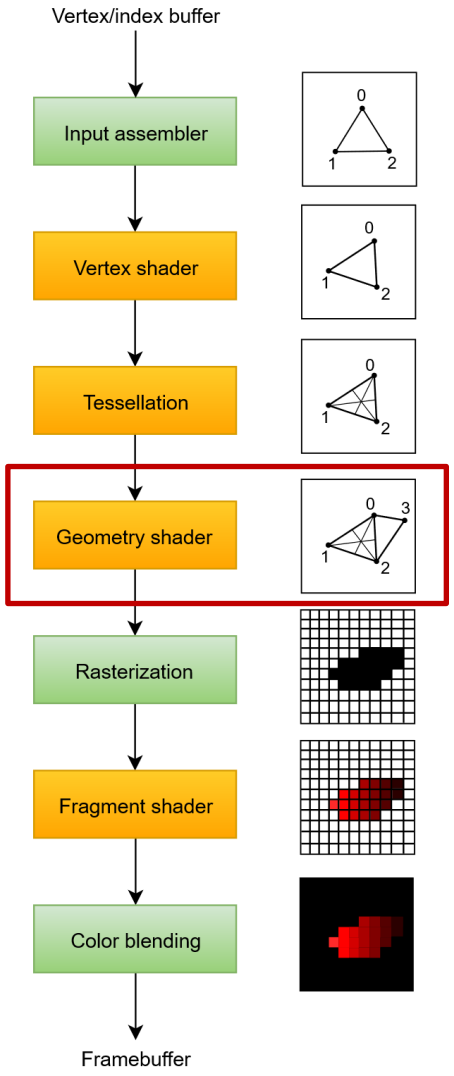
Programmable Stages – Tesselation

- An optional three-stage pipeline to subdivide primitives into smaller ones (triangle output)
- Stages:
  - **Tesselation Control Shader** (programmable): determines how many times the primitive is split along its normalized domain axes
    - Executed: once per primitive
  - **Primitive Generation**: Splits the input primitive
  - **Tesselation Evaluation Shader** (programmable): determines the positions of the new, split triangle vertices
    - Executed: once per split triangle vertex



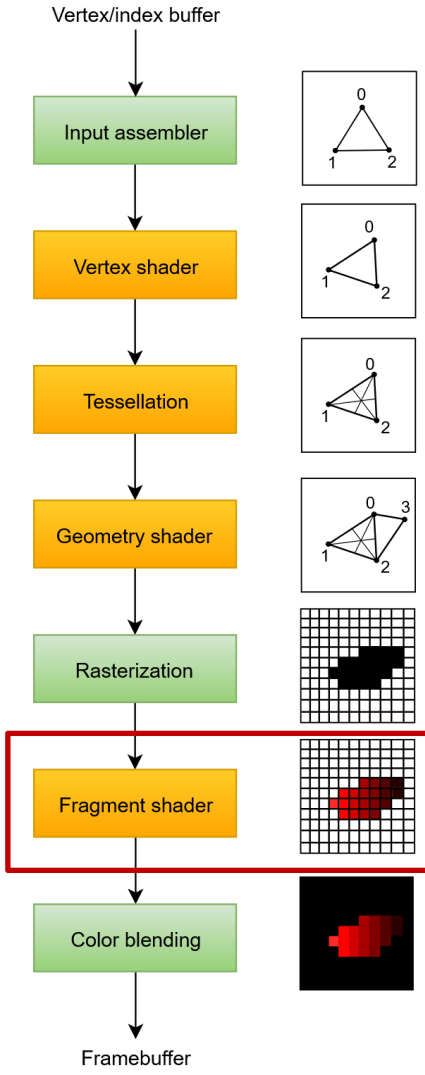
Programmable Stages – Geometry Shader

- Executed:
  - Once per primitive (before rasterization)
- Main role:
  - Change primitive type
  - Transform vertices according to knowledge of entire primitive
  - Amplify the primitive (generate extra primitives)
  - Wire the primitive to a specific rendering “layer”
- Input:
  - Primitive vertices
  - Attributes of all vertices (optional)
- Output:
  - Primitive vertices (mandatory)
  - “out” attributes of all vertices (optional)



Programmable Stages – Fragment Shader

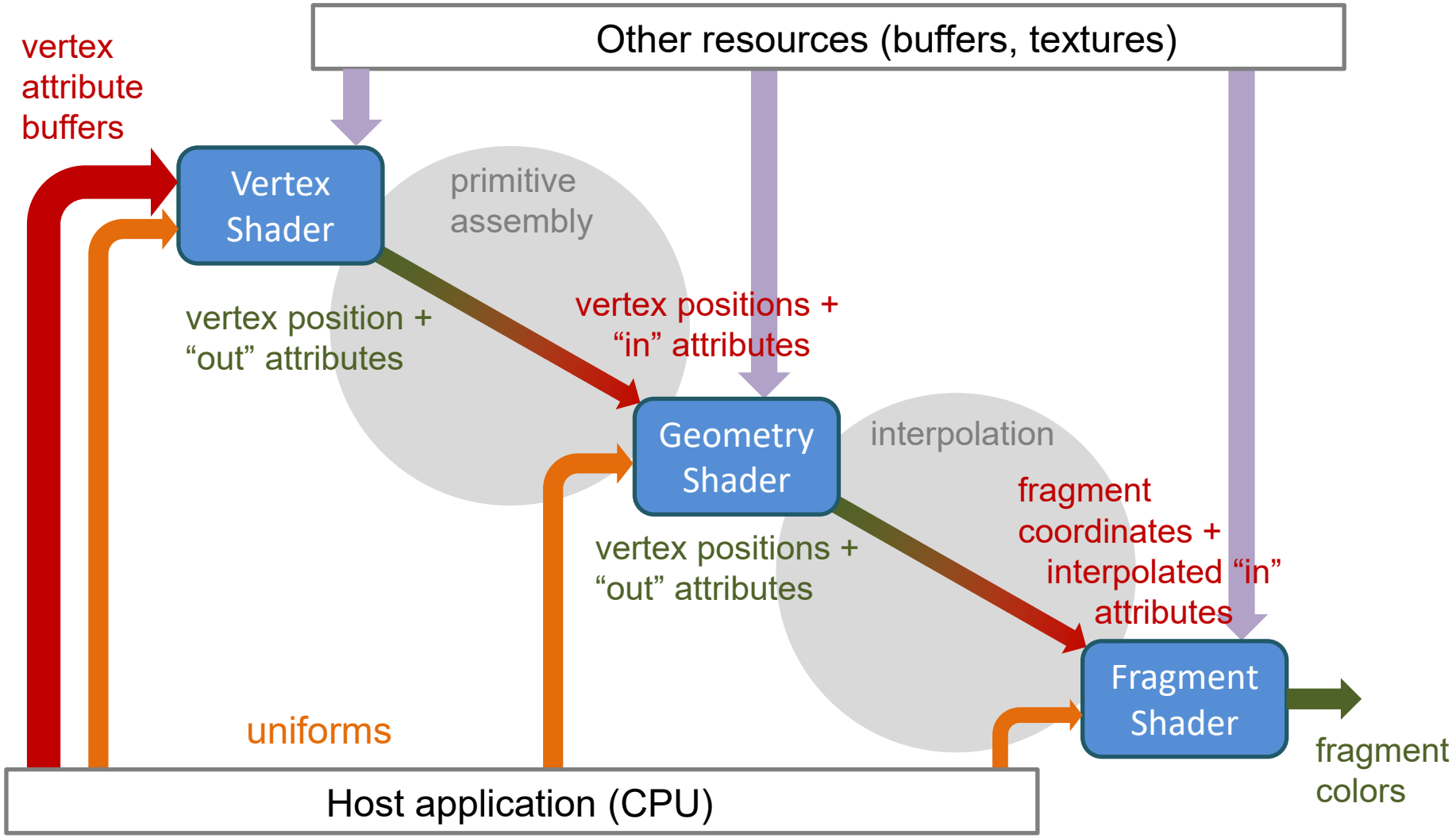
- Executed:
  - Once per fragment (after rasterization)
- Main role:
  - Determine the fragment's color and transparency
  - Decide to keep or "discard" the fragment
- Input:
  - Interpolated vertex data
- Output:
  - Pixel values to 1 or more buffers (simultaneously)



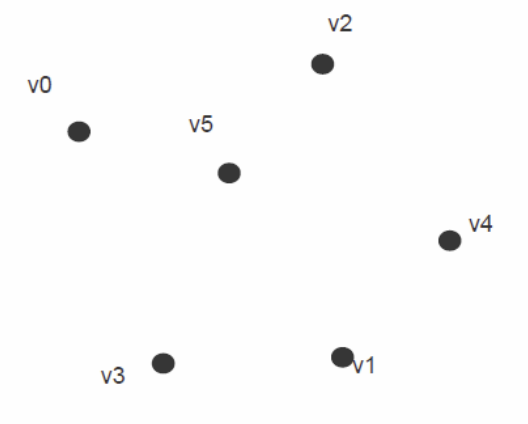
Shaders - Data Communication (1)

- Each stage passes along data to the next via input/output variables
  - Output of one stage must be consistent with the input of the next
- The host application can also provide shaders with other variables that are globally accessible by all shaders in an active shader program
  - These variables are called uniform variables/buffers

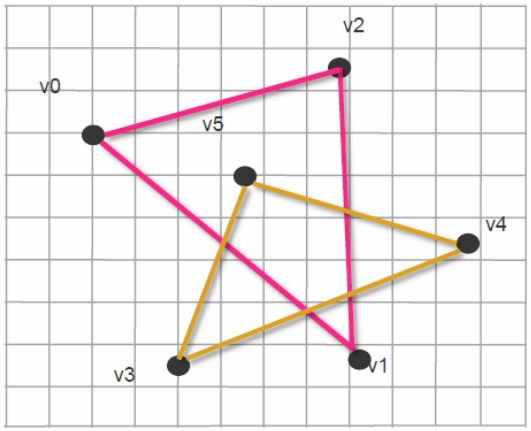
Shaders – Data Communication (2)



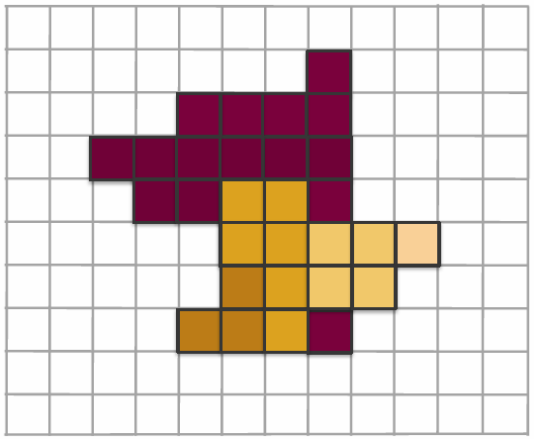
Shader Invocation Example



Vertex Shader
invoked 6 times



Geometry Shader
invoked 2 times



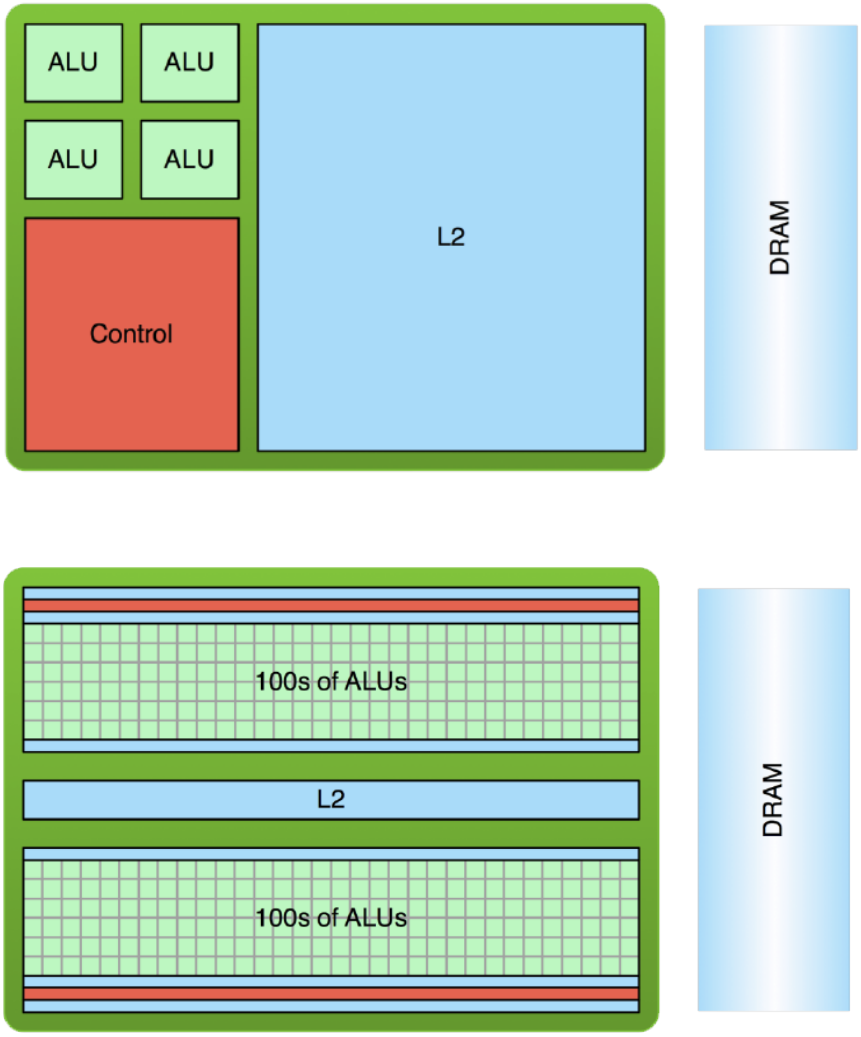
Fragment Shader
invoked 35 times
(for the hidden
fragments, too)

The Graphics Processing Unit



- GPU is practically a combination of a MIMD/SIMD supercomputer on a chip!
- Main purpose:
  - Programmable graphics co-processor for image synthesis
  - H/W acceleration to all visual aspects of computing, including video decompression
- Due to its architecture and processing power, it is nowadays also used for demanding general-purpose computations → GPUs are evolving towards this!

GPU: Architectural Goals



CPU

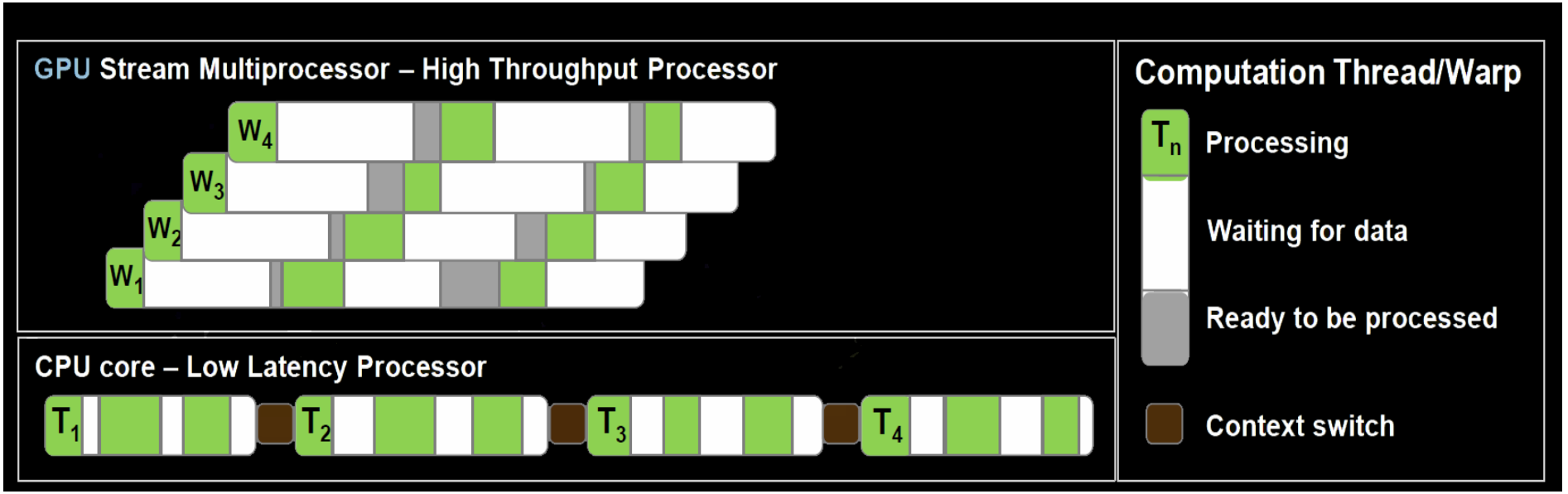
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More ALU transistors

Philosophy of Operation

- CPU architecture must **minimize latency** within each thread
- GPU architecture **hides latency** with computation from other threads

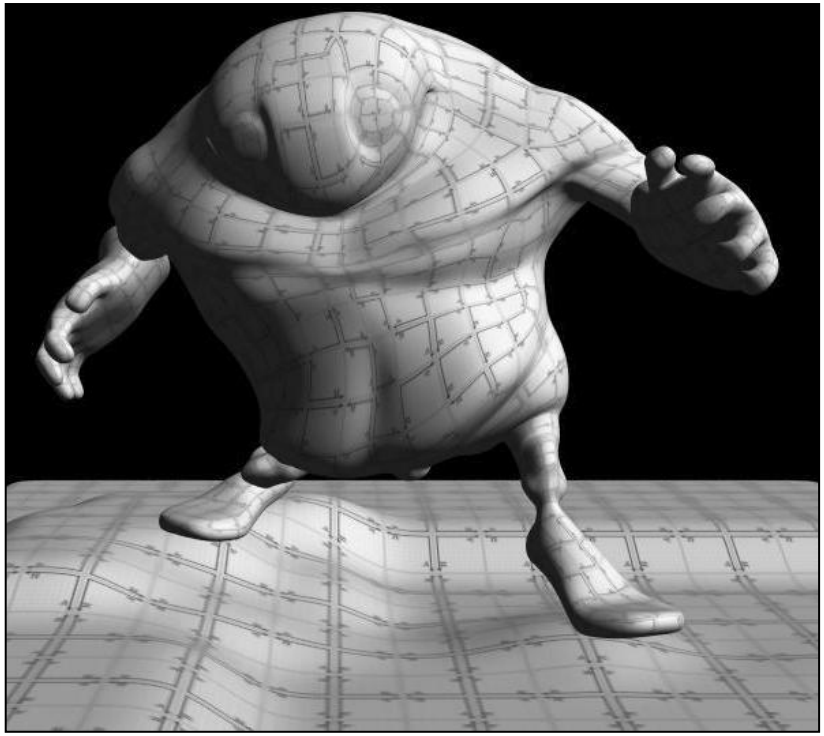


Mapping Shaders to H/W: Example (1)

- A simple Direct3D fragment shader example

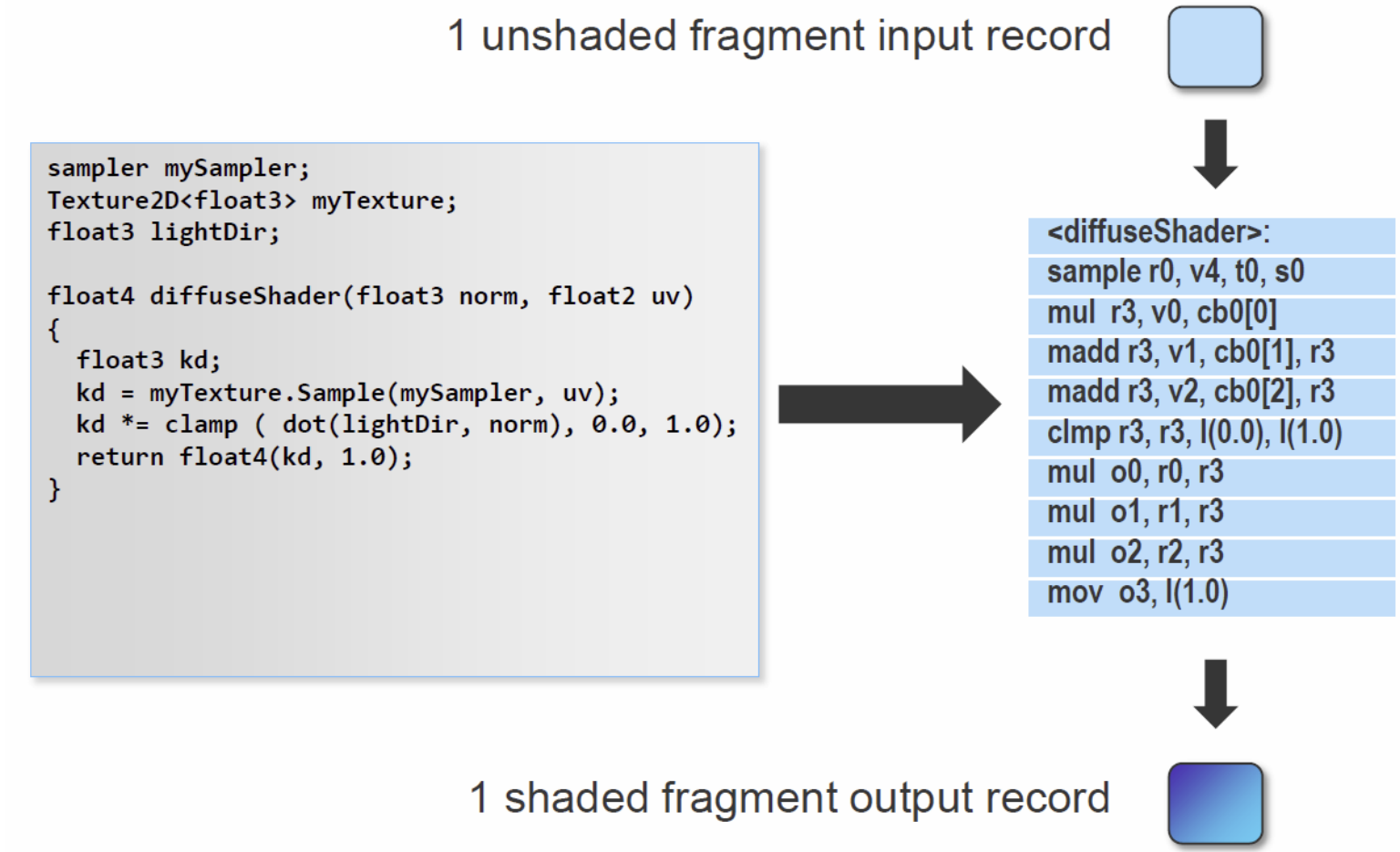
```
sampler mySampler;
Texture2D<float3> myTexture;
float3 lightDir;

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

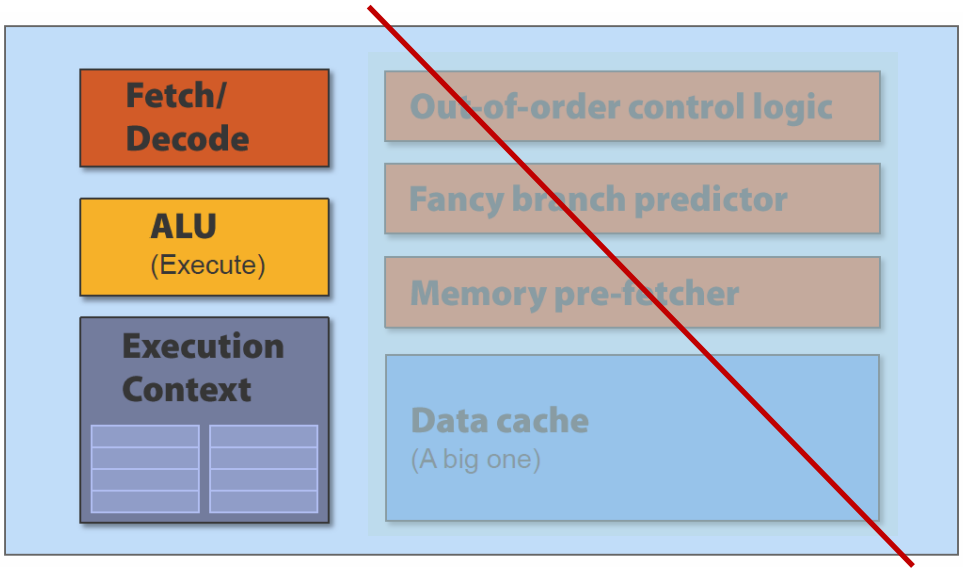


Mapping Shaders to H/W: Example (2)

Compile the Shader:



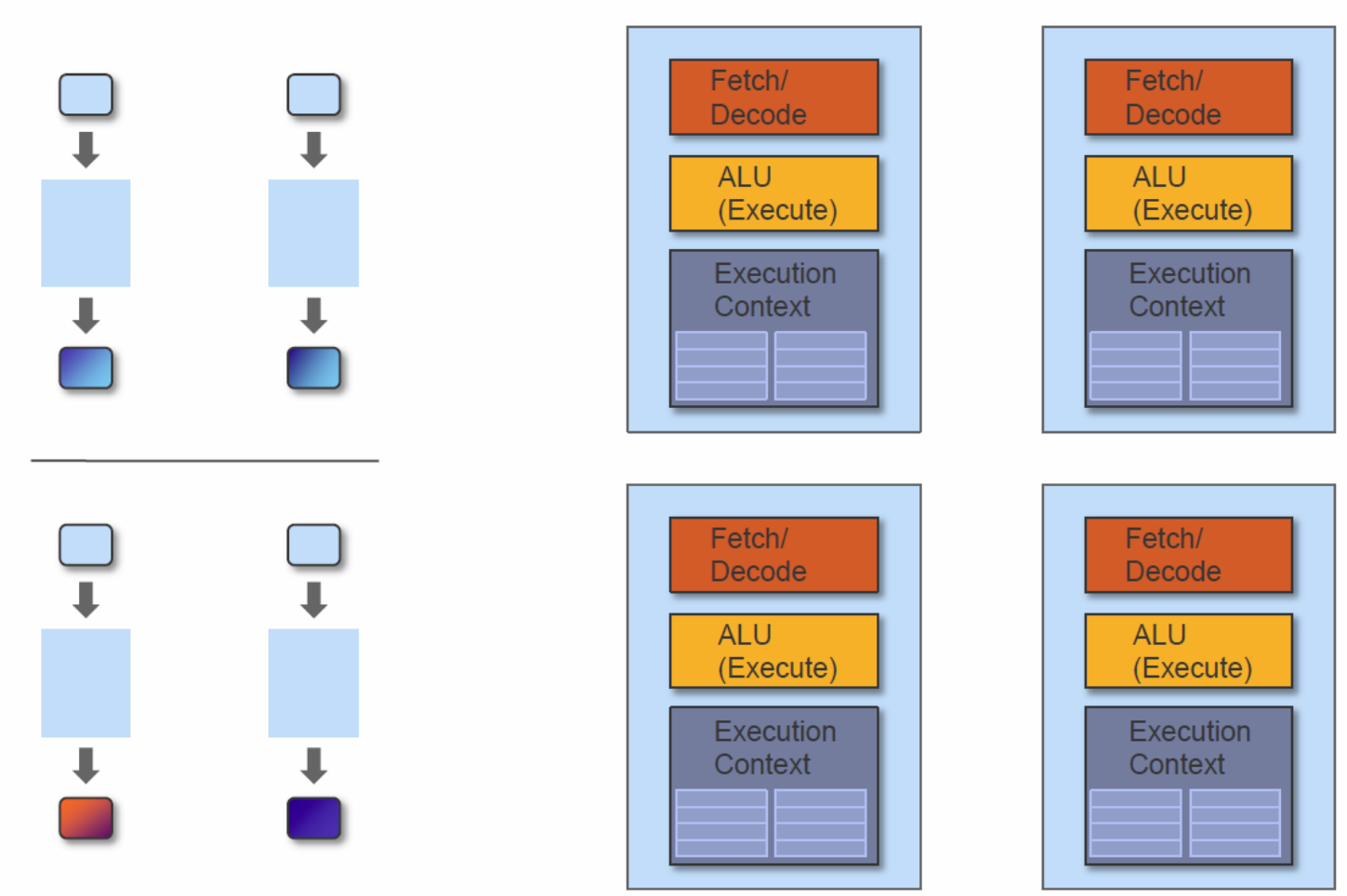
GPU: Slimming down the Cores



- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More computations → More ALU transistors →
- Need to lose some core circuitry →
- Remove single-thread optimizations

GPU: Multiple Cores

- Multiple threads



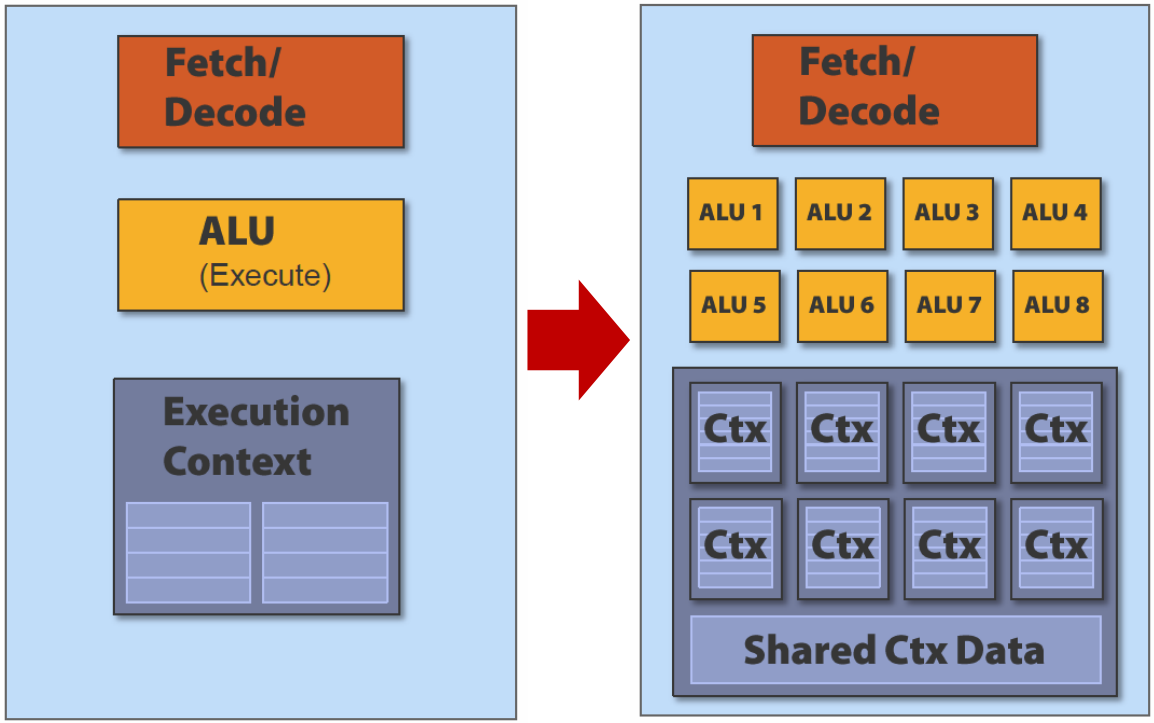
GPU: …More Cores



16 cores = 16 simultaneous instruction streams

What about Multiple Data?

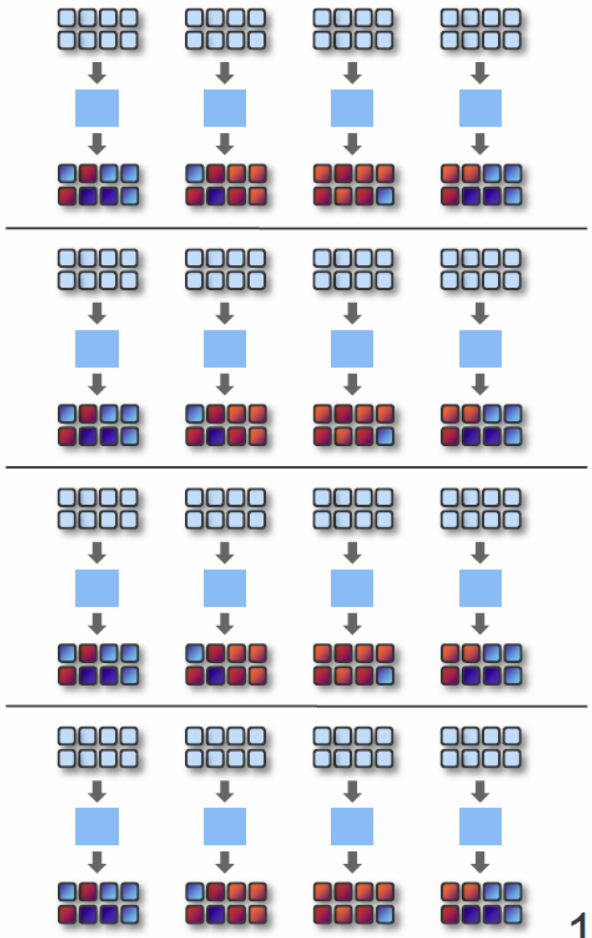
- Shaders are inherently executed many times over and over on multiple records from their input data streams (SIMD!)

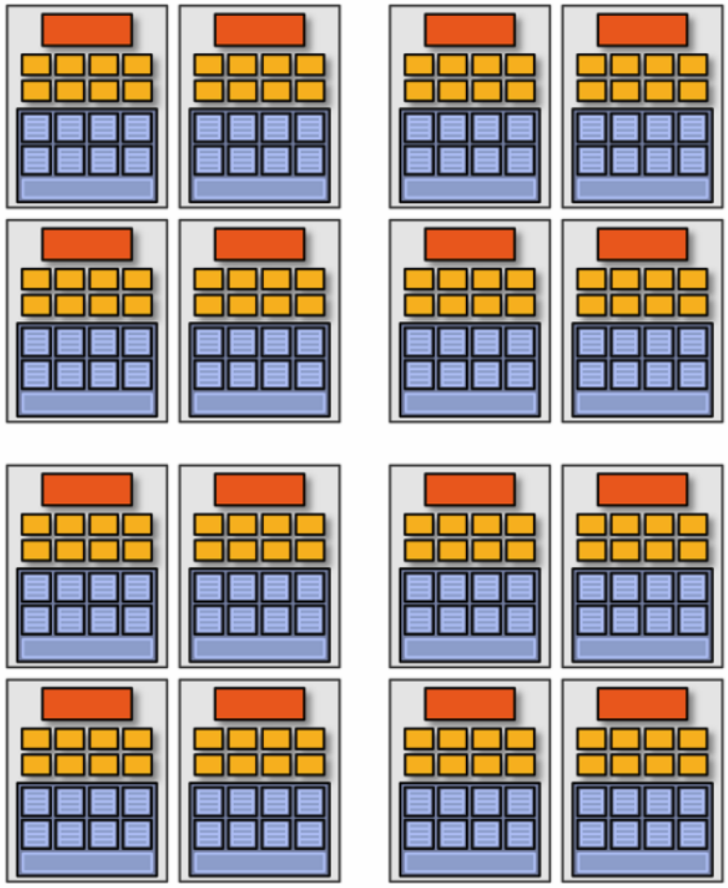


Amortize cost / complexity of instruction management to multiple ALUs → Share instruction unit

Adding It All Up: Multiple SIMD Cores

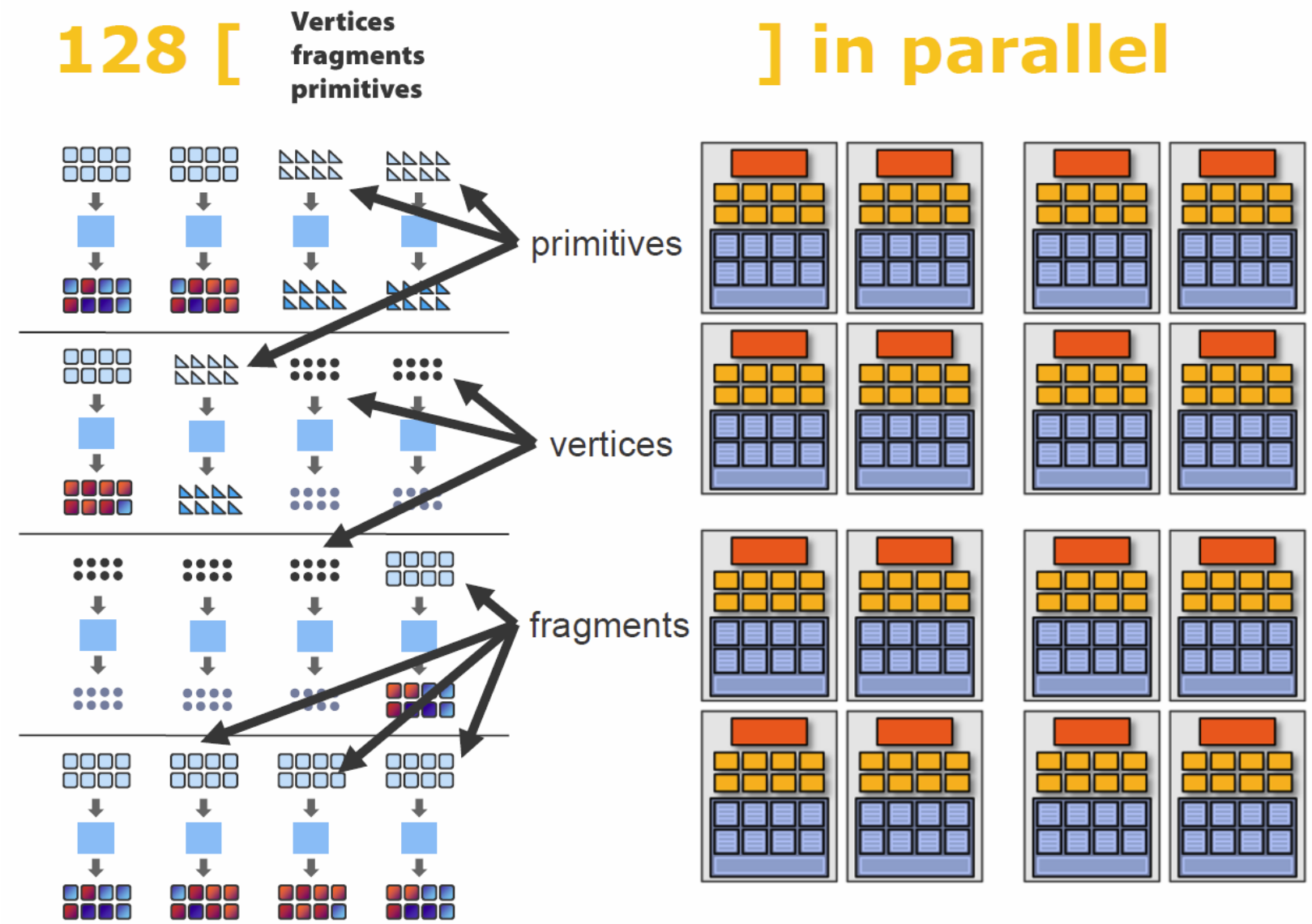
In this example: 128 data records processed simultaneously





16 cores = 128 ALUs
= 16 simultaneous instruction streams

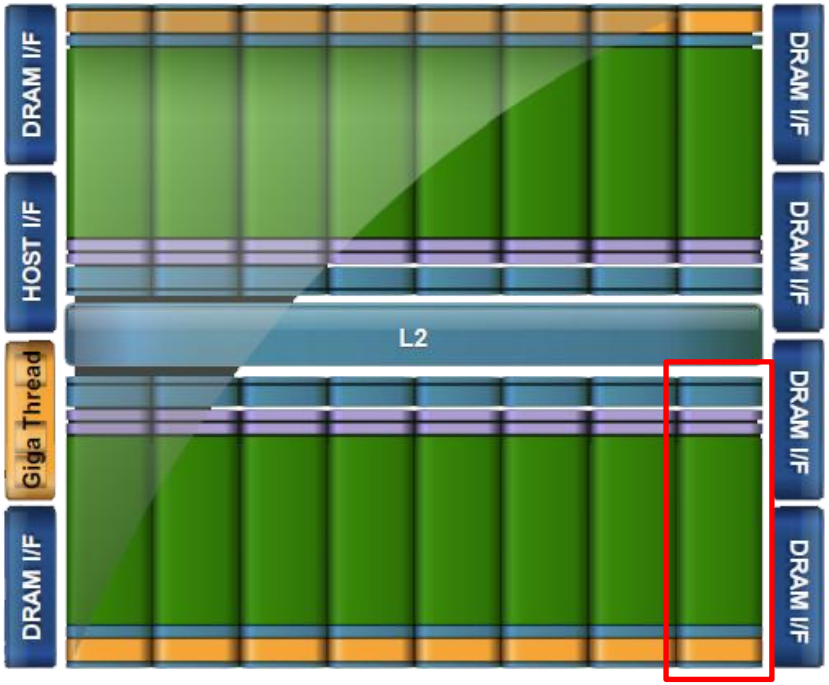
Multiple SIMD Cores: Shader Mapping



Unified Shader Architecture

- Older GPUs had split roles for the shader cores →
  - Imbalance of utilization
- Unified architecture:
  - Pool of “Stream Multiprocessors”
  - H/W scheduler to designate shader instructions to SMs

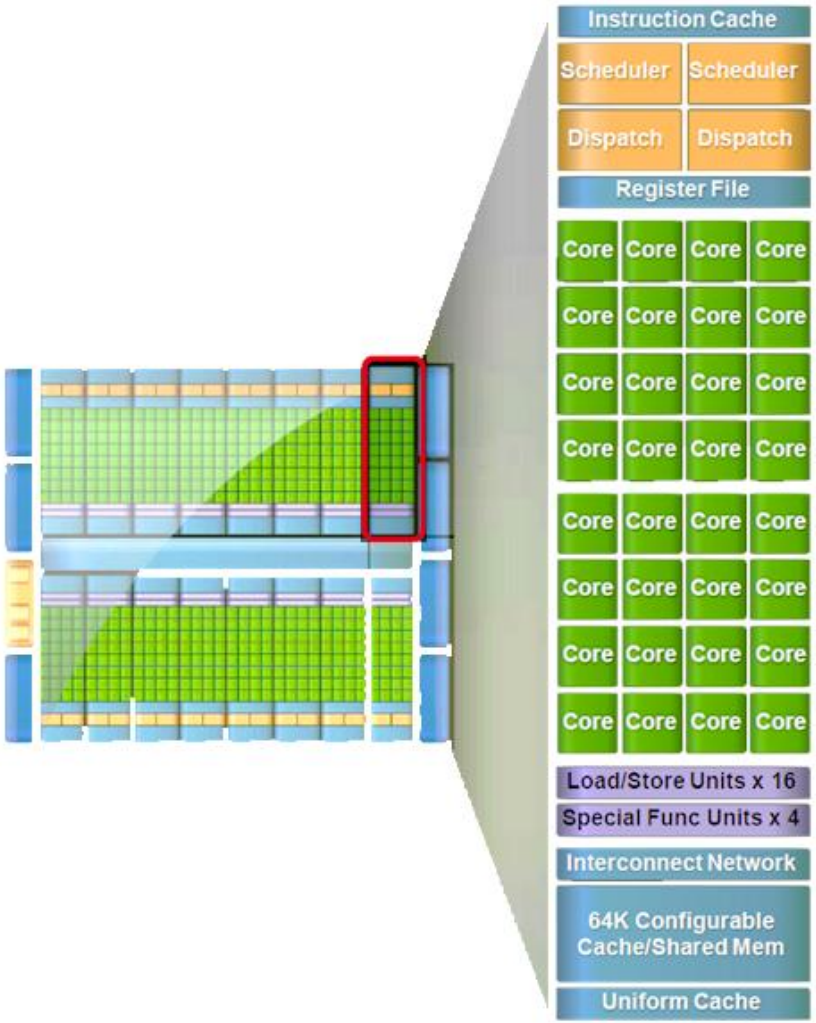
Under the Hood



Components:

- Global memory
  - Analogous to RAM in a CPU server
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
  - Control units, registers, execution pipelines, caches
- H/W scheduling

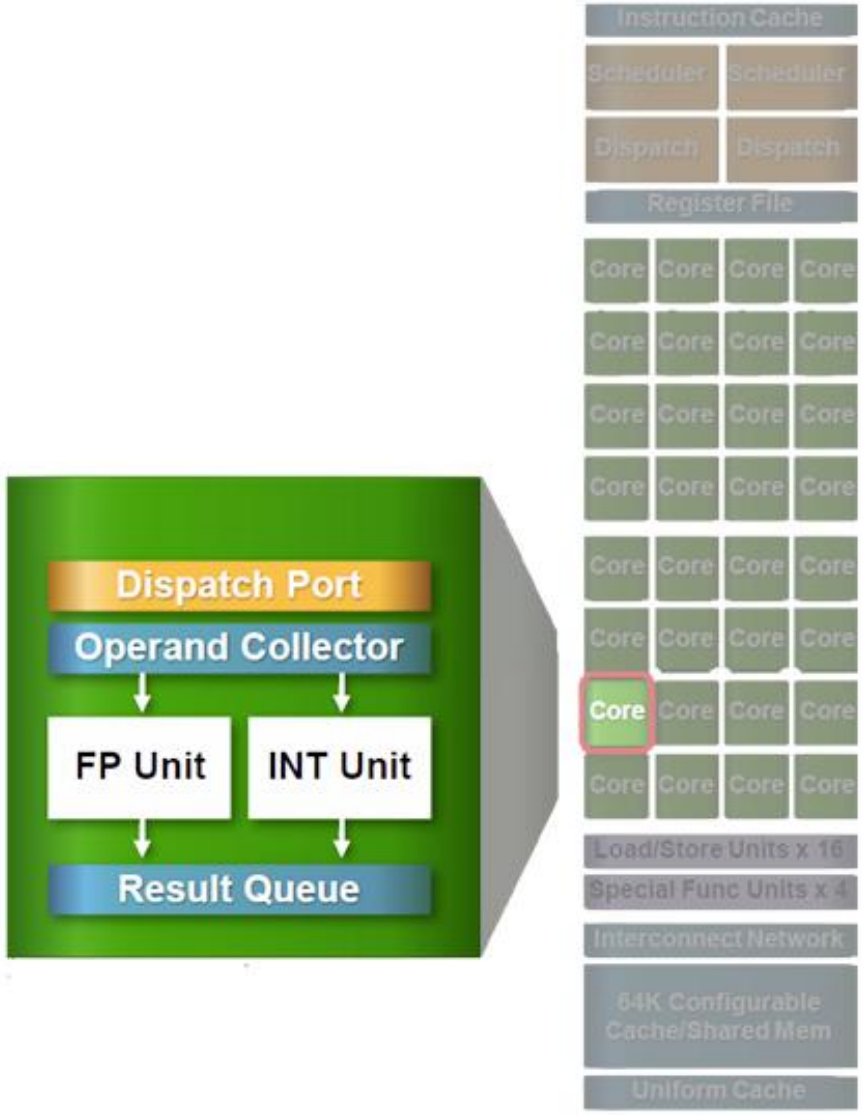
The Stream Multiprocessor



E.g. FERMI SM:

- 32 cores per SM
- Up to 1536 live threads concurrently (32 active: a “warp”)
- 4 special-function units
- 64KB shared mem+ L1 cache
- 32K 32-bit registers

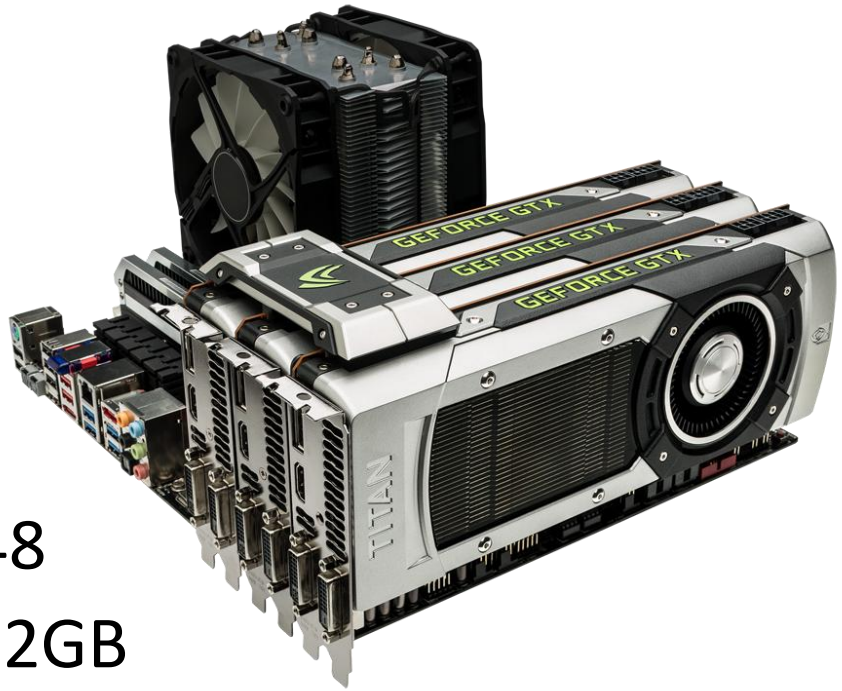
The “Shader” (Compute) Core



Each core:

- Floating point & Integer unit
- IEEE 754-2008 floating-point standard
- Fused multiply-add (FMA) instruction
- Logic unit
- Move, compare unit
- Branch unit

Some Facts



- Typical cores per unit:512-2048
- Typical memory on board: 2-12GB
- Global memory bandwidth: 200-300 GB/s
- Local SM memory aggregate bandwidth: >1TB/s
- Max processing power per unit:2-4.5 TFlops
- A single motherboard can host up to 3-4 units

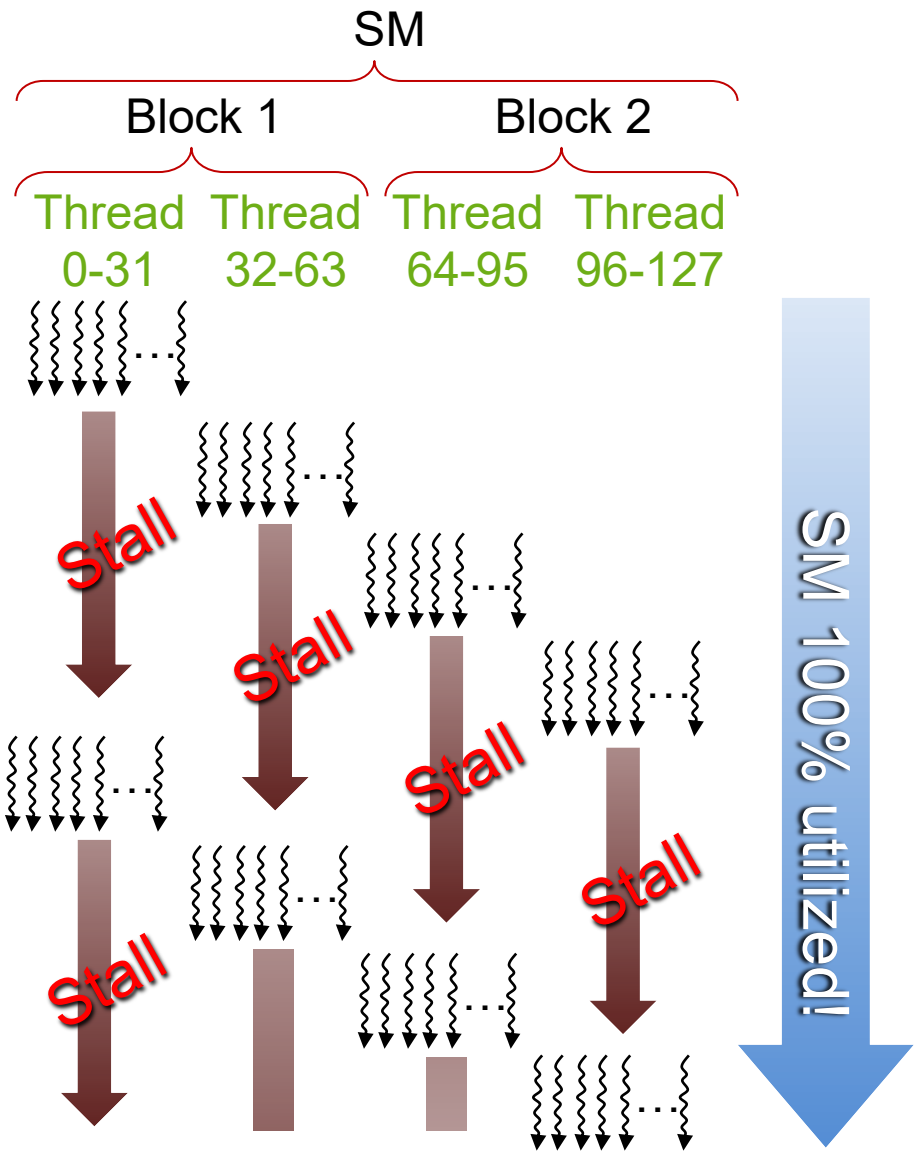
GPU Interconnection

Current typical configurations:

- CPU – GPU comminication via PCIe X16
  - Scalable
  - High computing power
  - High energy profile
  - Constrains on PCIe throughput
- Fused CPU – GPU
  - Potentially integrated SoC design (e.g. i5,i7, mobile GPUs)
  - High-bandwidth buses (CPU-memory-GPU, e.g. PS4)
  - Truly unified architecture design (e.g. mem. addresses)
  - Less flexible scaling (or none at all)

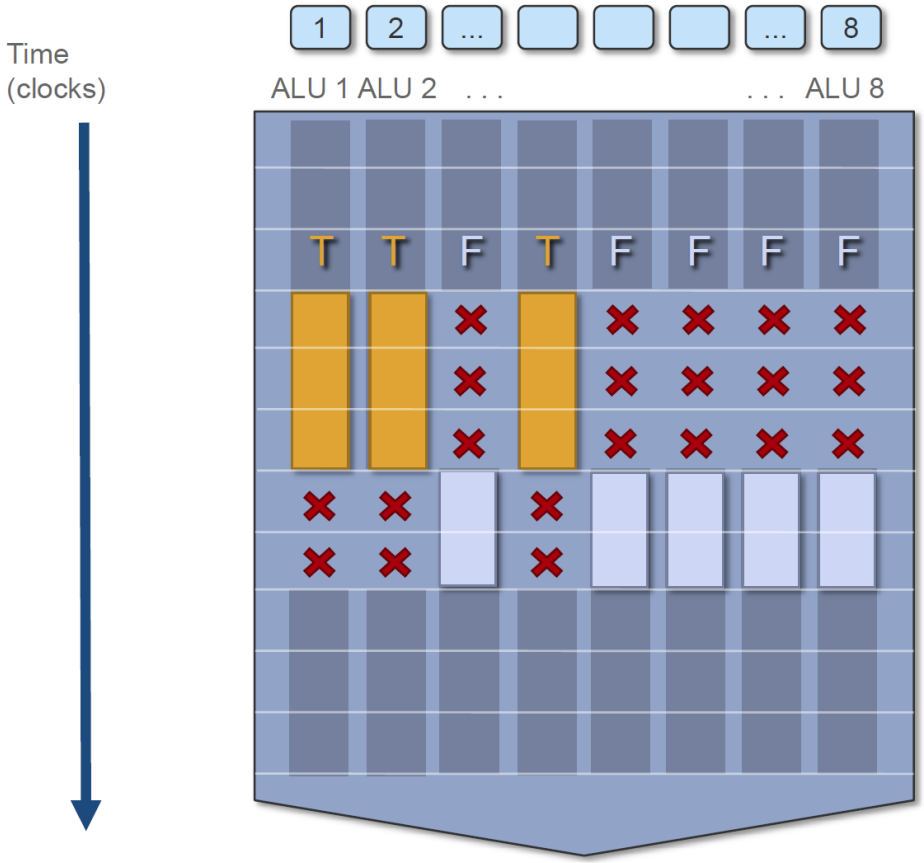
Utilization and Latency (1)

- Global memory access can seriously stall the SMs
  - up to 800 cycles is typical
- Solution: Many interleaved thread groups (“warps”) live on the same SM



Utilization and Latency (2)

- Divergent code paths (branching) pile up!
- Unrollable loops cost = max iterations




```
<unconditional
shader code>

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

<resume unconditional
shader code>
```

Contributors

- Georgios Papaioannou

- Sources:
  - [GPU] K. Fatahalian, M. Houston, GPU Architecture (Beyond Programmable Shading - SIGGRAPH 2010)
  - [CDA] C. Woolley, CUDA Overview, NVIDIA