NVIDIA CUDA
Hardware

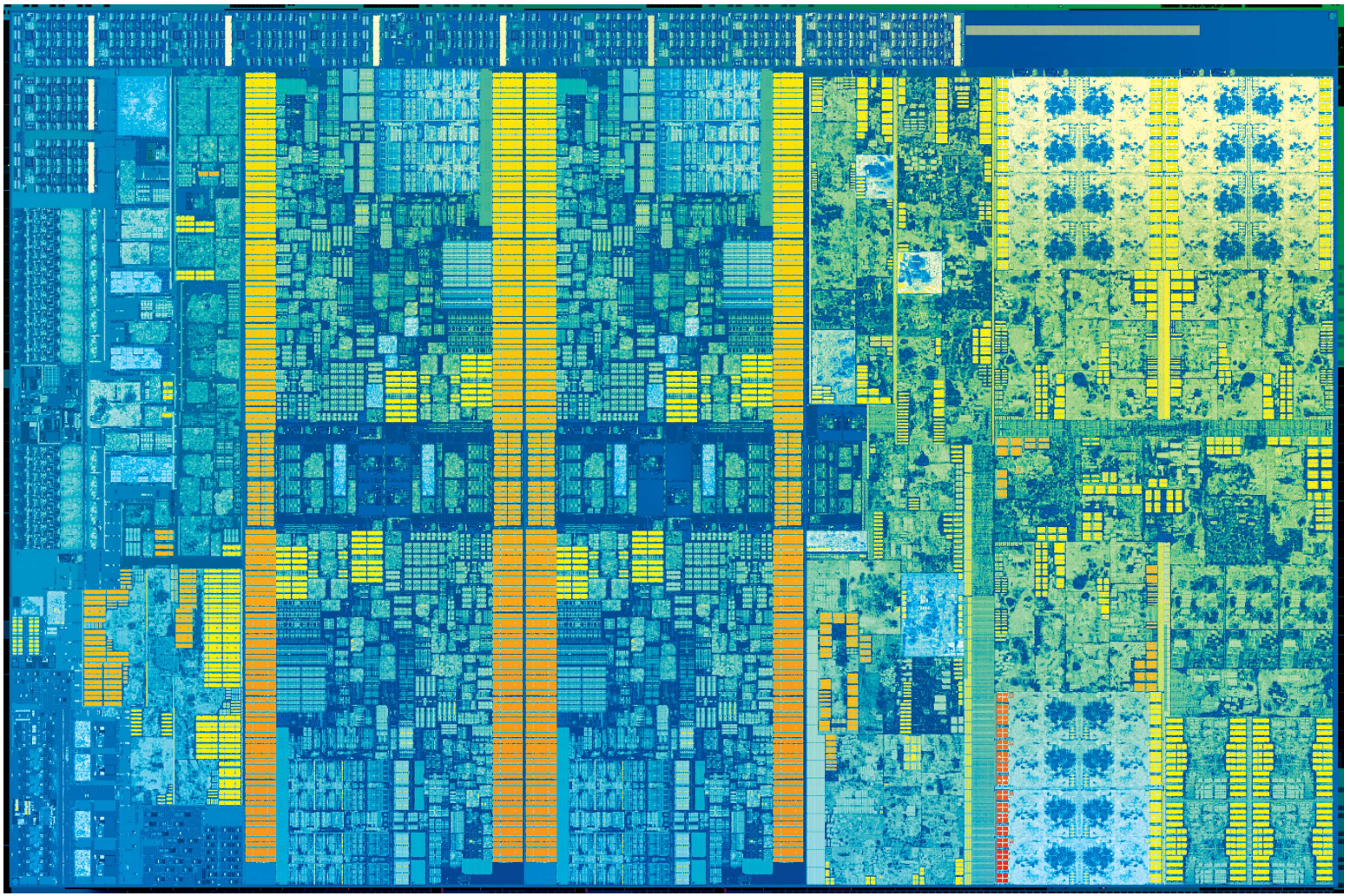
Overview

- Execution Model
- Memory Model
- CUDA API Basics
- Code Example

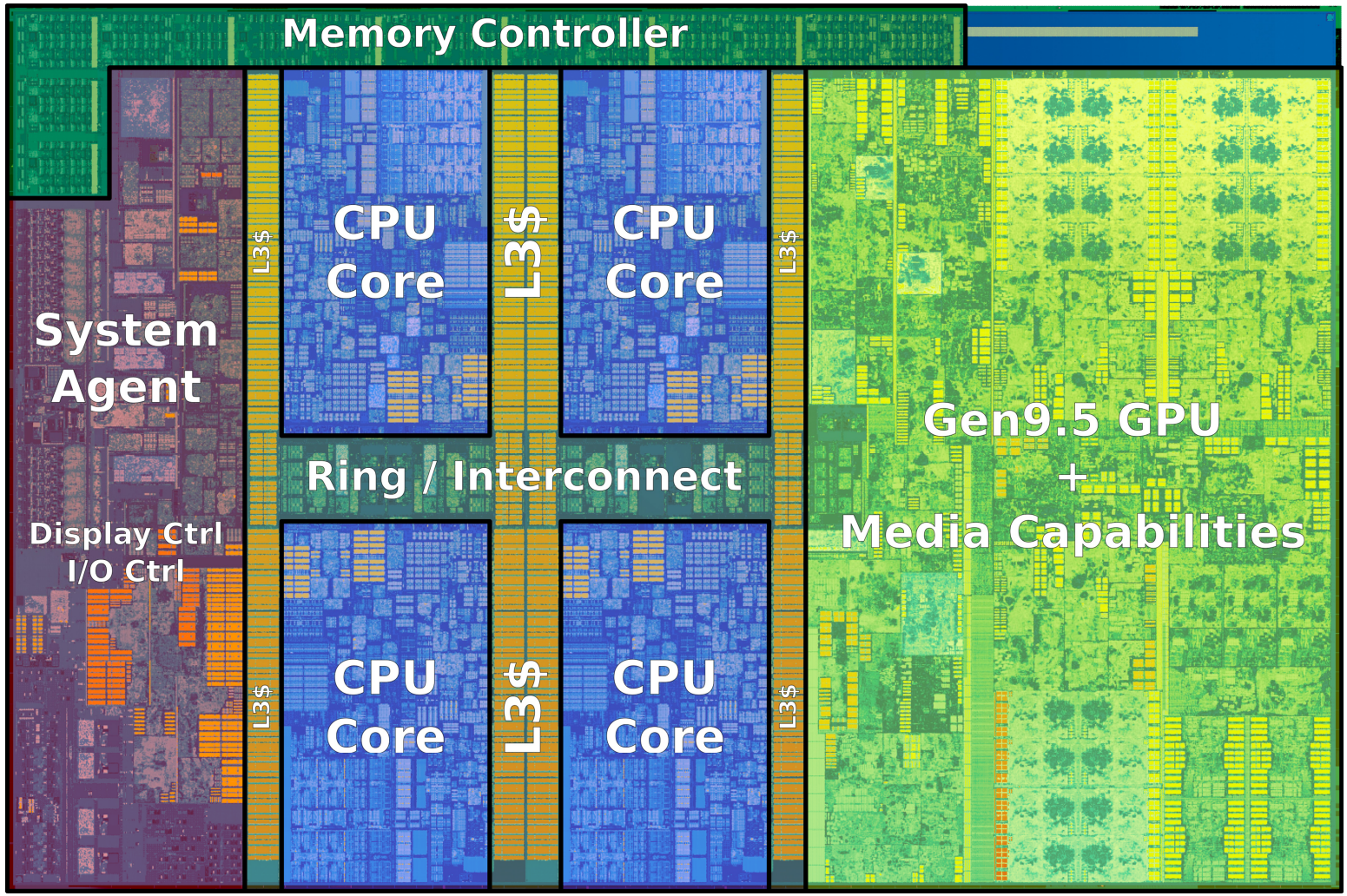
CUDA Real "Hardware"

	Intel Core i7 7700	NVIDIA Titan XP	AMD Radeon HD 8970
Transistors	2.27 billion	12 billion	4.31 billion
Processor frequency	3.6 GHz	1582 MHz	1000 MHz
Cores	4	3840	2408
Cache/Shared Memory	8 MB	256 KB x 12	N/A
Threads executed per cycle	8	3840	2408
Active hardware threads	8	30720	16384
Peak FLOPS	101 GFLOPS	12.15 TFLOPS	4.3 TFLOPS
Memory bandwidth	35.76 GiB/s	547.7 GiB/s	288 GiB/s
Power Consumption	95 Watts	250 Watts	250 Watts
Watt / GFLOPS	**0.94**	**0.02**	**0.058**

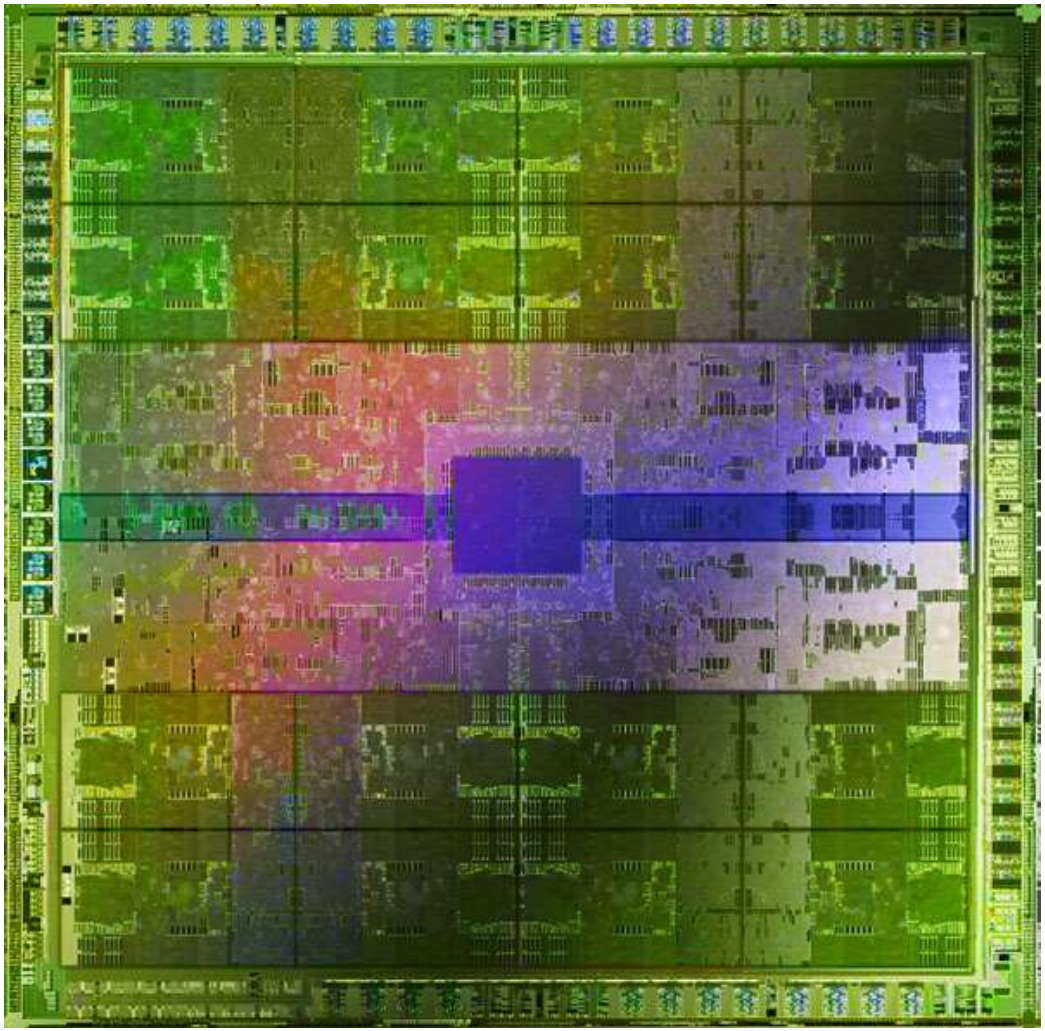
CPU



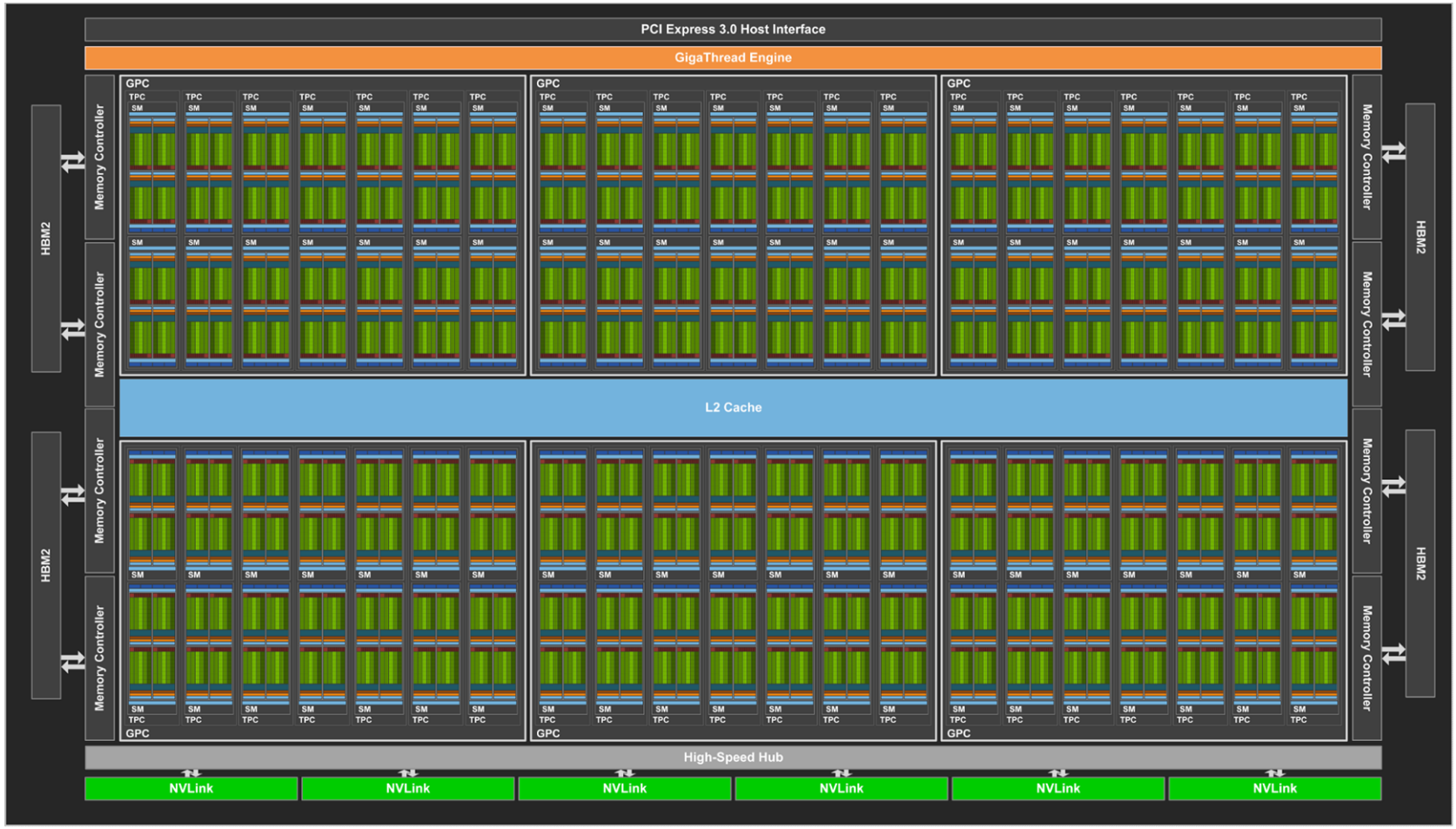
CPU


Memory Controller
System Agent
Display Ctrl
I/O Ctrl
L3$
CPU Core
CPU Core
L3$
L3$
Ring / Interconnect
CPU Core
CPU Core
L3$
L3$
Gen9.5 GPU
+
Media Capabilities

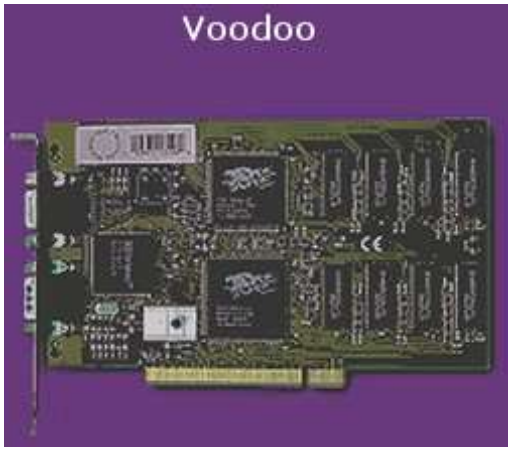
GPU



NVIDIA V100

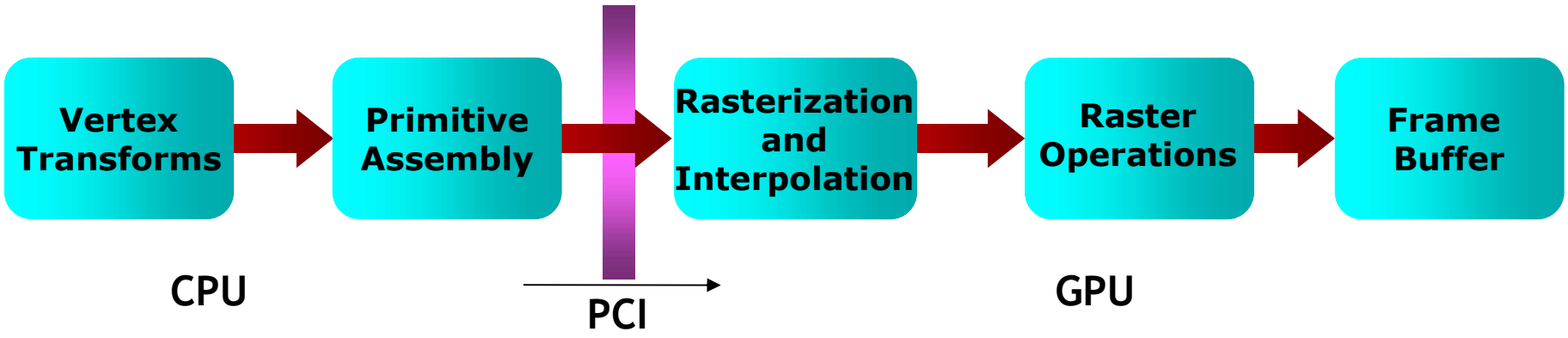

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

Generation I: 3dfx Voodoo (1996)



http://accelenation.com/?ac.id.123.2

- One of the first true 3D game cards
- Worked by supplementing standard 2D video card.
- Did not do vertex transformations: these were done in the CPU
- Did do texture mapping, z-buffering.

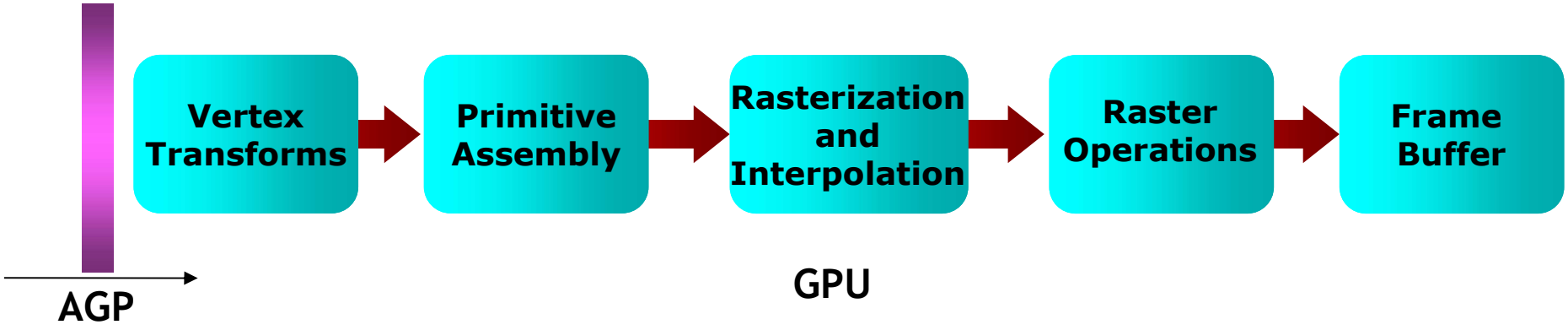


Generation II: GeForce/Radeon 7500 (1998)

GeForce 256

http://accelenation.com/?ac.id.123.5

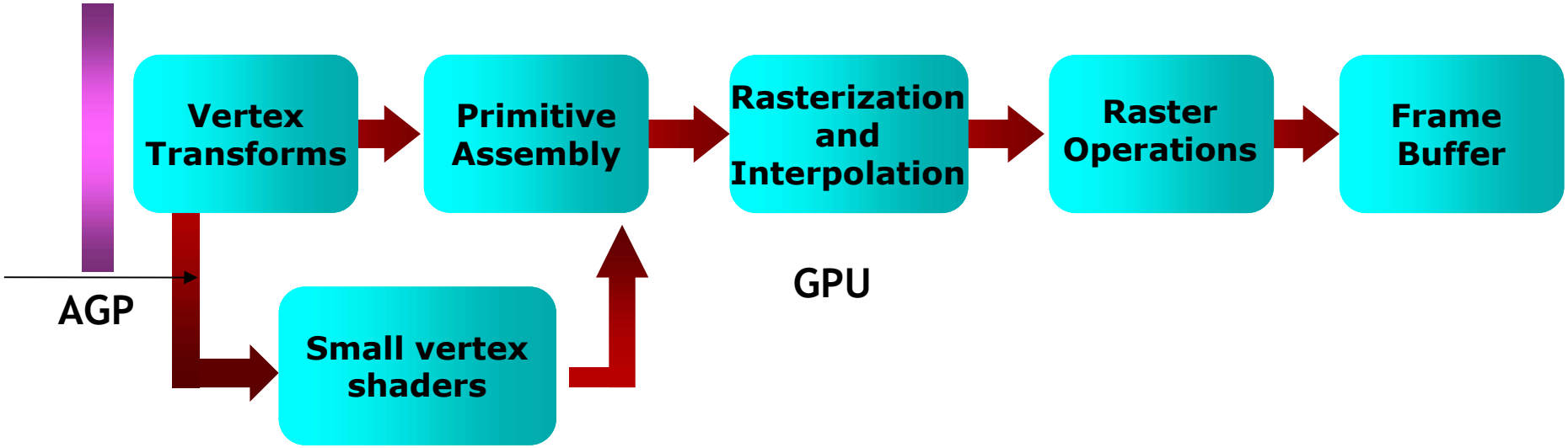
- Main innovation: shifting the transformation and lighting calculations to the GPU
- Allowed multi-texturing: giving bump maps, light maps, and others..
- Faster AGP bus instead of PCI



Generation III: GeForce3/Radeon 8500(2001)

http://accelenation.com/?ac.id.123.7

- For the first time, allowed limited amount of programmability in the vertex pipeline
- Also allowed volume texturing and multi-sampling (for antialiasing)

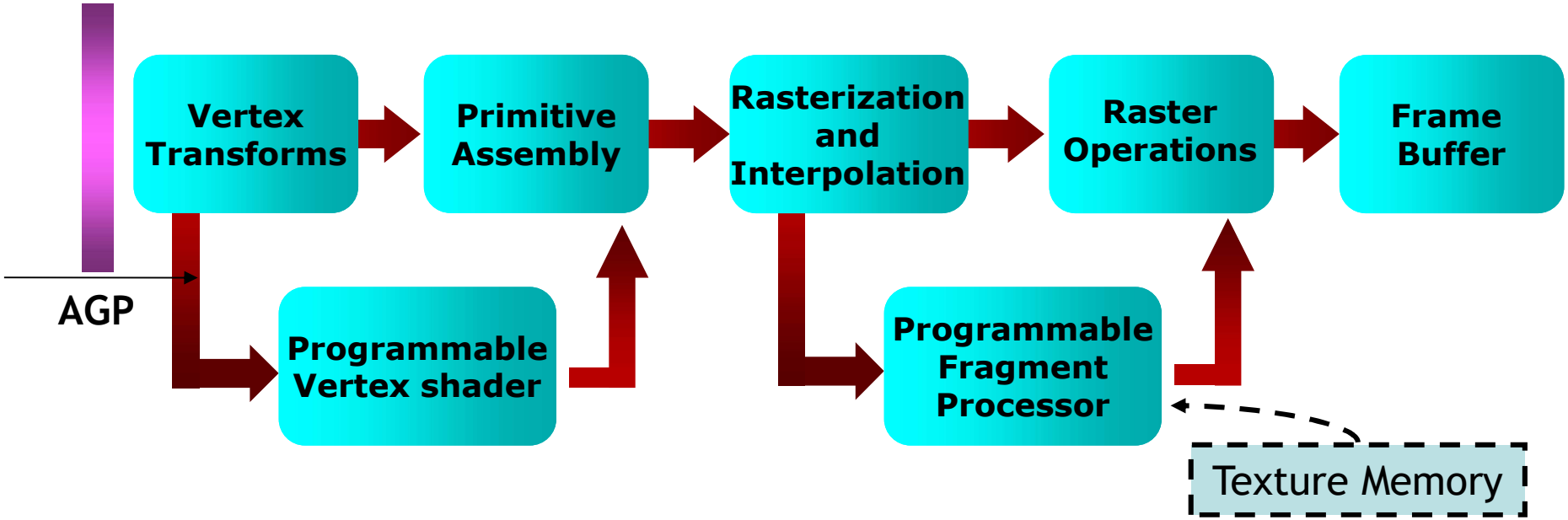


Generation IV: Radeon 9700/GeForce FX (2002)

GeForce FX

http://accelenation.com/?ac.id.123.8

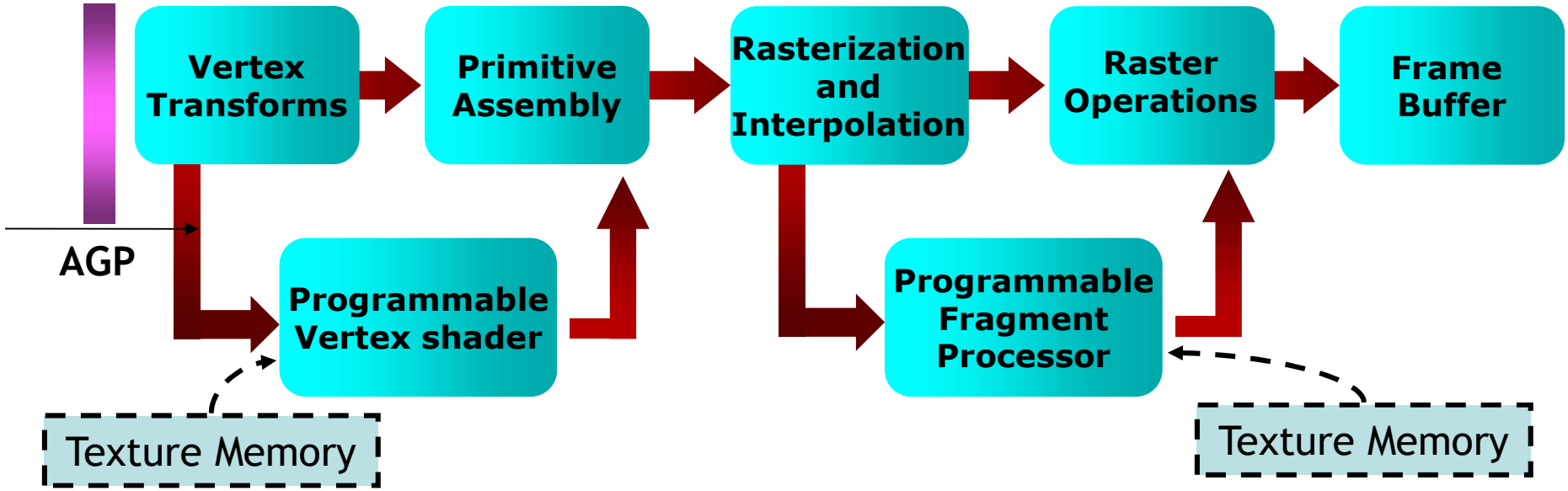
- This generation is the first generation of fully-programmable graphics cards
- Different versions have different resource limits on fragment/vertex programs



Generation IV.V: GeForce6/X800 (2004)

Not exactly a quantum leap, but…

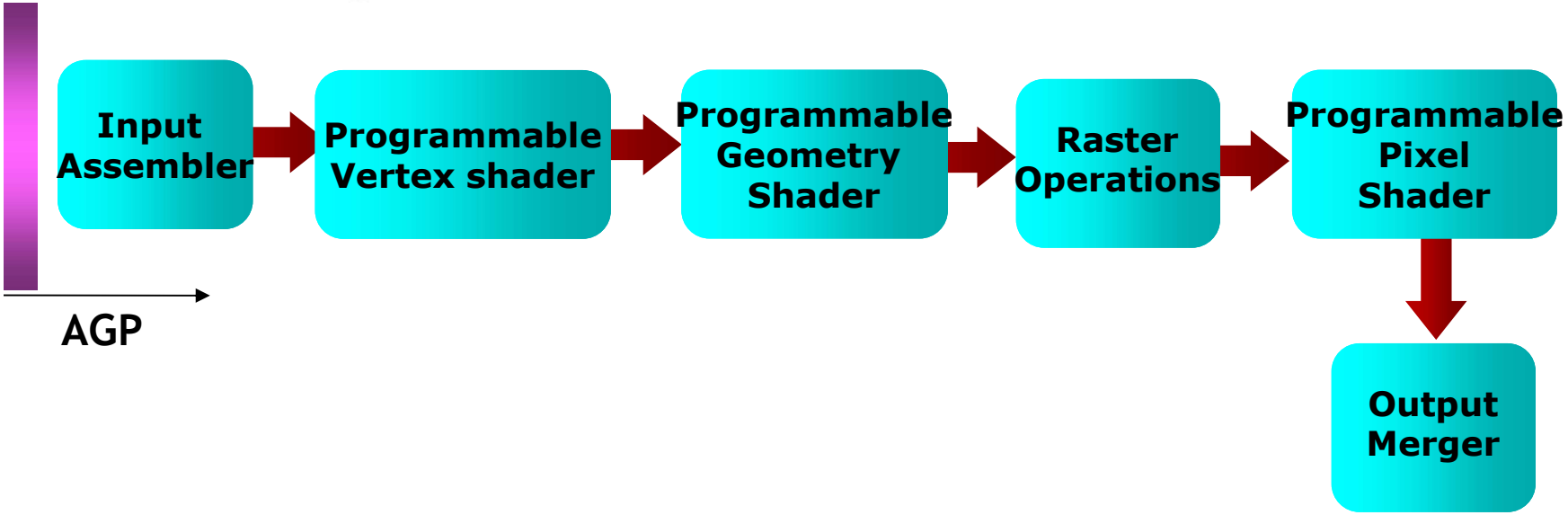
- Simultaneous rendering to multiple buffers
- True conditionals and loops
- Higher precision throughput in the pipeline (64 bits end-to-end, compared to 32 bits earlier.)
- PCIe bus
- More memory/program length/texture accesses
- Texture access by vertex shader

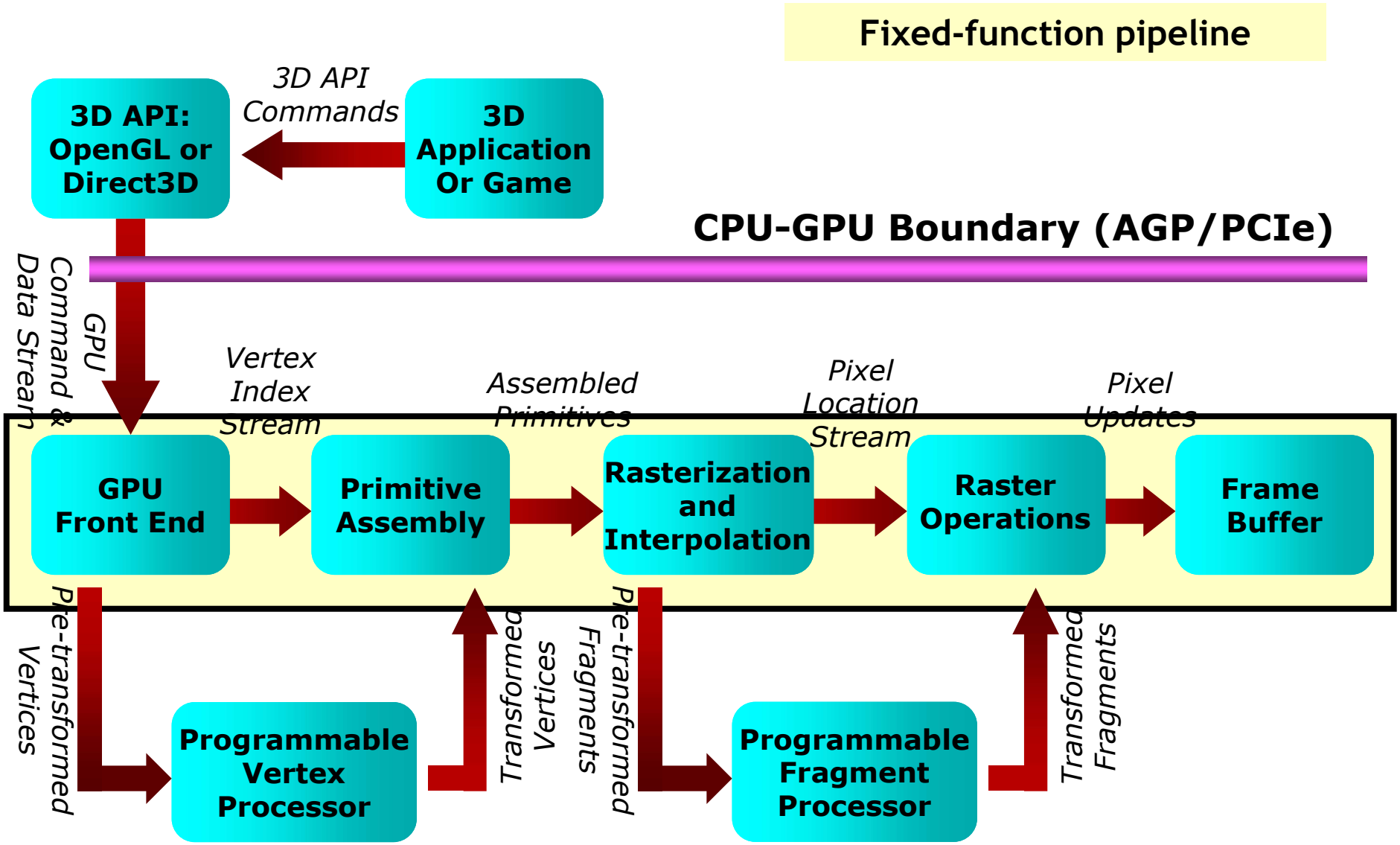


Generation V: GeForce8800/HD2900 (2006)

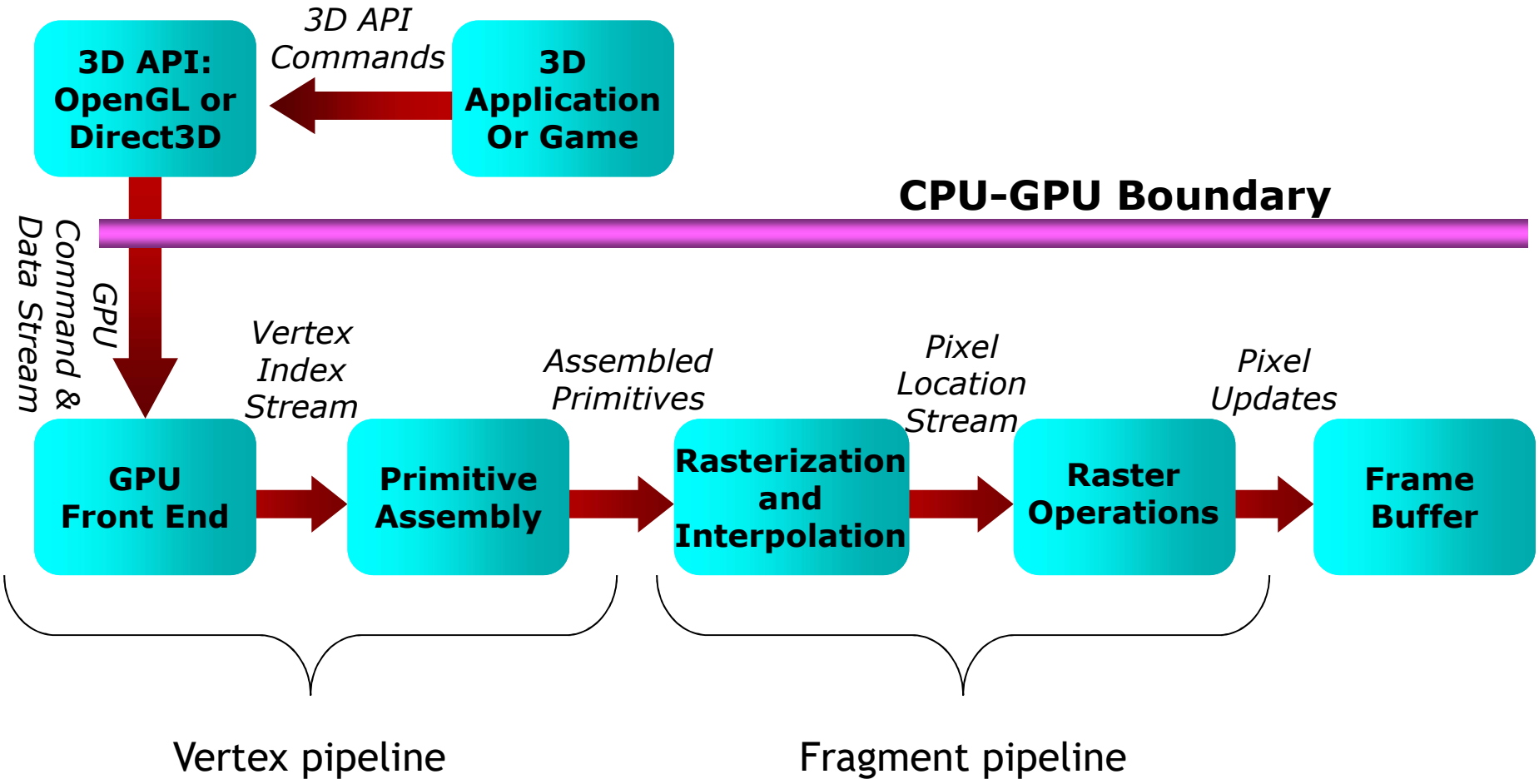
Complete quantum leap

- Ground-up rewrite of GPU
- Support for DirectX 10, and all it implies (more on this later)
- Geometry Shader
- Support for General GPU programming
- Shared Memory (NVIDIA only)




Fixed-function pipeline
3D API: OpenGL or Direct3D
3D API Commands
3D Application Or Game
CPU-GPU Boundary (AGP/PCIe)
GPU Command & Data Stream
Vertex Index Stream
Assembled Primitives
Pixel Location Stream
Pixel Updates
GPU Front End
Primitive Assembly
Rasterization and Interpolation
Raster Operations
Frame Buffer
Pre-transformed Vertices
Programmable Vertex Processor
Transformed Vertices
Pre-transformed Fragments
Programmable Fragment Processor
Transformed Fragments

Graphics pipeline



How to program GPU?

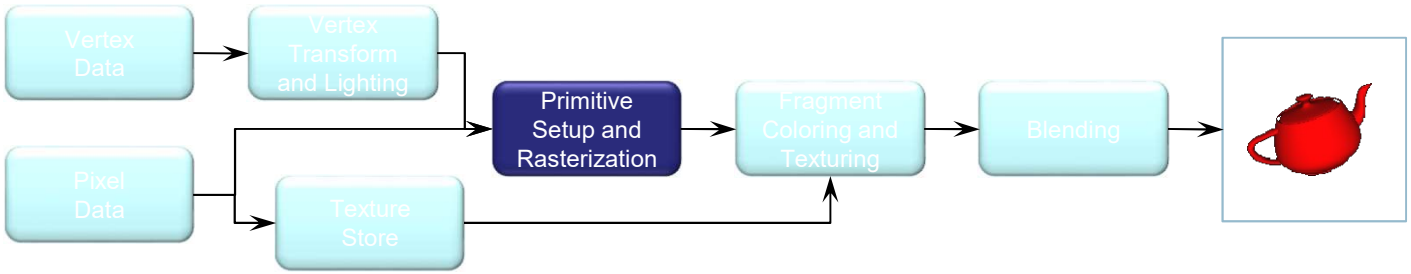
CUDA was not the first GPU programming framework.

OPENGL

Evolution of the OpenGL Pipeline

In the Beginning …

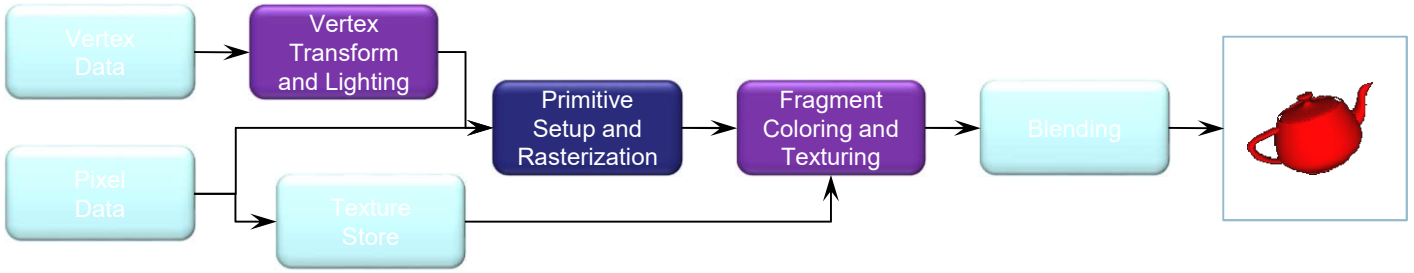
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



- The pipeline evolved
  - but remained based on fixed-function operation through
    OpenGL versions 1.1 through 2.0 (Sept. 2004)

Beginnings of The Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



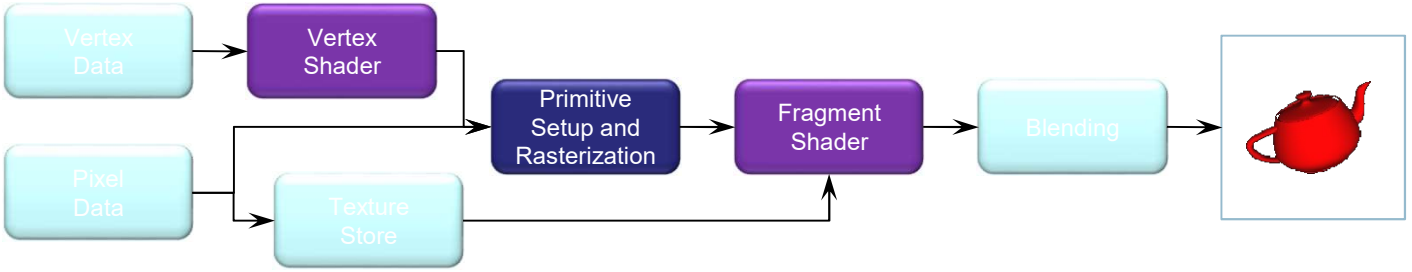
An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

The Exclusively Programmable Pipeline

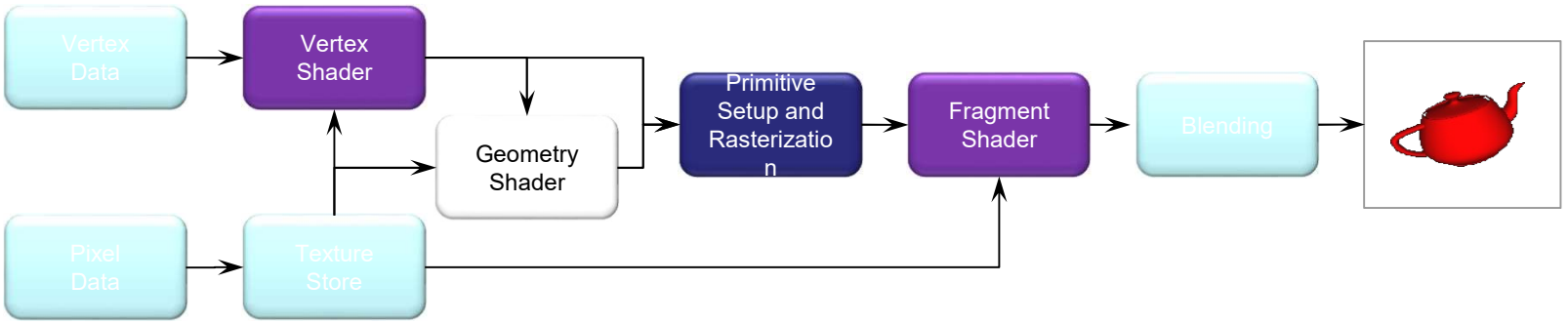
- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders



- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

More Programmability

- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders
  - modify geometric primitives within the graphics pipeline



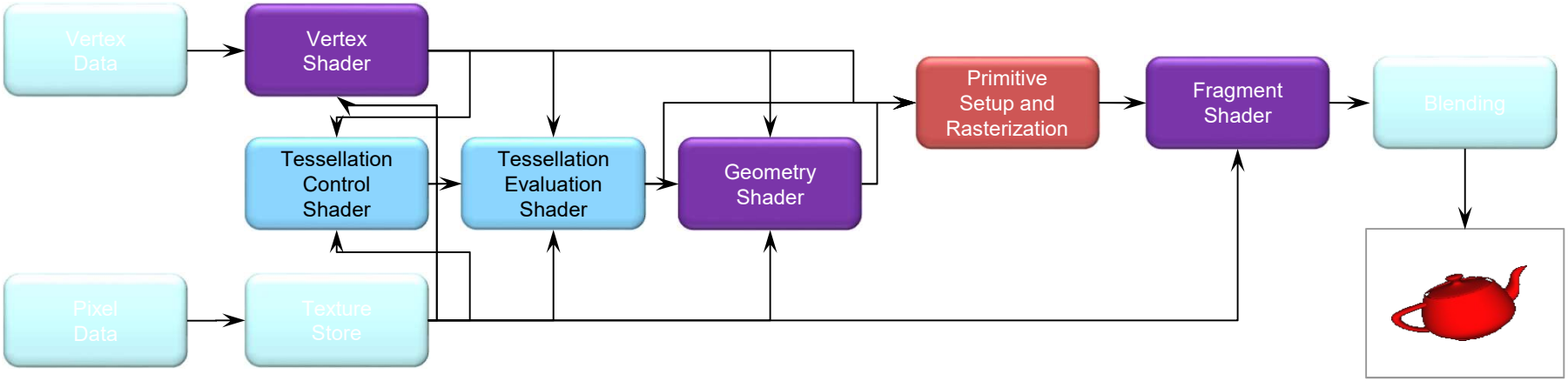
More Evolution – Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
    - it's like `GL_ARB_compatibility`, only not insane ☺
  - currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

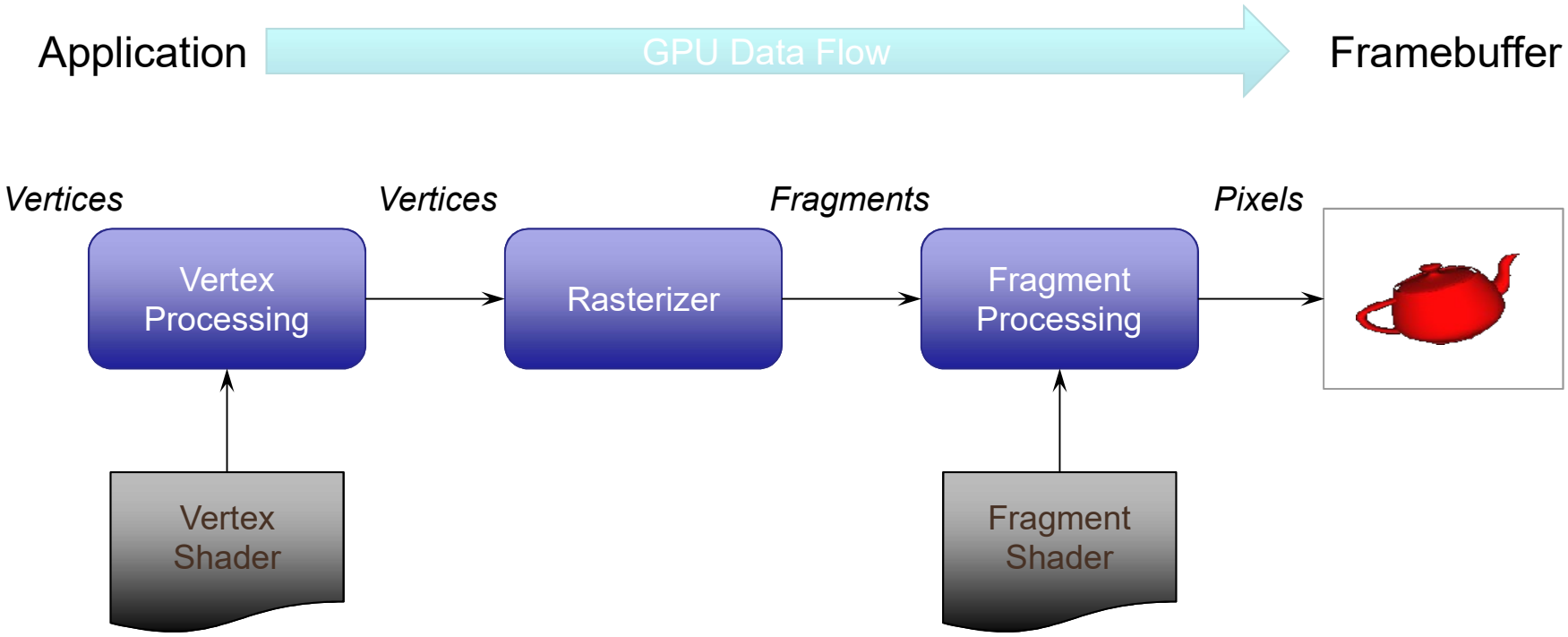
The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders
- Latest version is 4.3



OpenGL Application Development

A Simplified Pipeline Model



OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render

Shaders and GLSL

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
```

`v[1]`, `v.y`, `v.g`, `v.t` - all refer to the same element

- Component swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Functions

- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Simple Vertex Shader for Cube Example

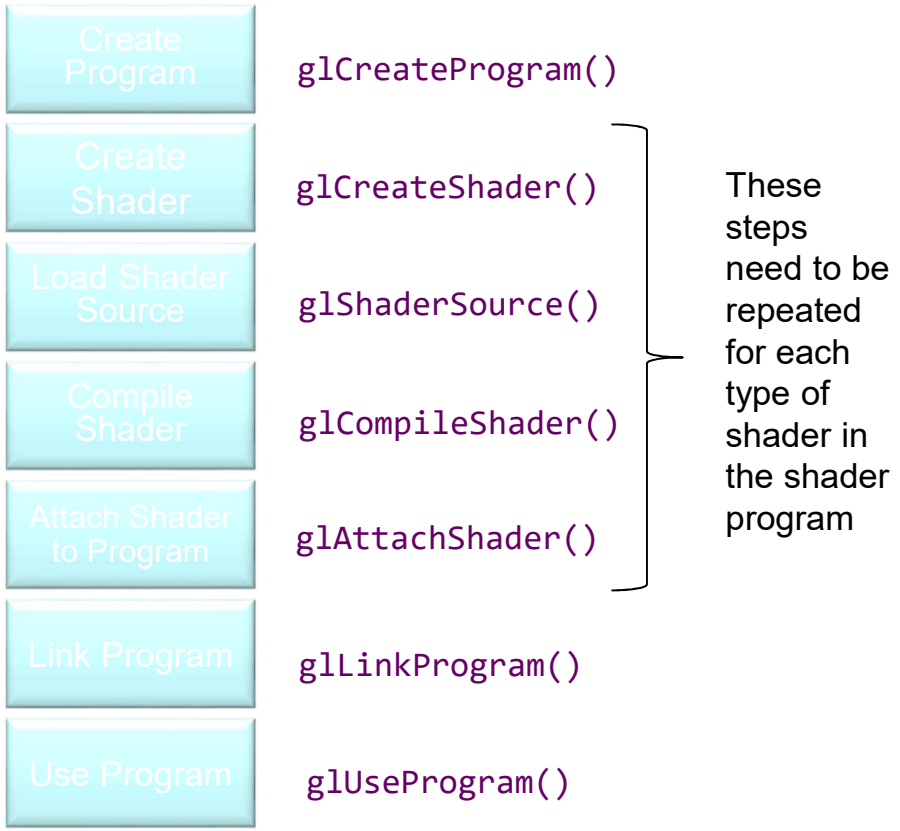
```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment's final
color

void main()
{
    fColor = color;
}
```

Getting Your Shaders into OpenGL

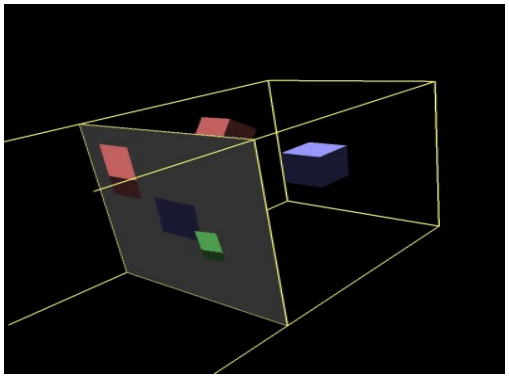
- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain



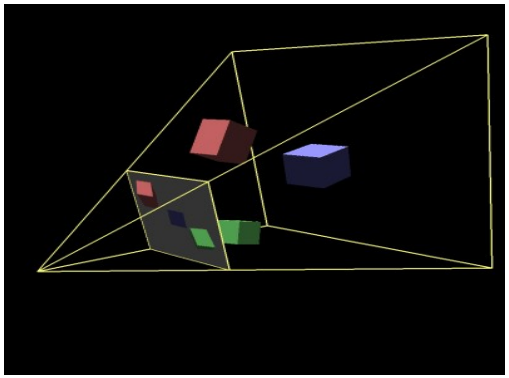
WHAT KIND OF COMPUTATION IN OPENGL?

Specifying What You Can See (Example)

Orthographic View



Perspective View



$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Creating the LookAt Matrix

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|}$$

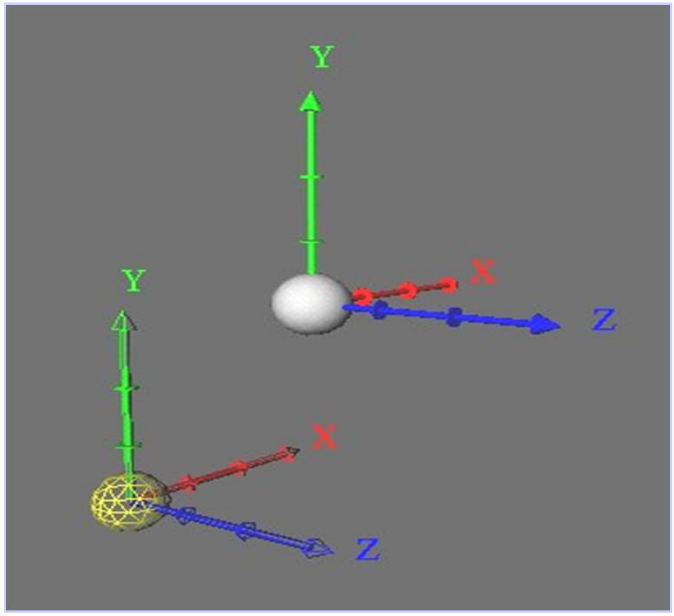
$$\hat{v} = \hat{u} \times \hat{n}$$

$$\Rightarrow \begin{pmatrix} u_x & u_y & u_z & -(eye \cdot \vec{u}) \\ v_x & v_y & v_z & -(eye \cdot \vec{v}) \\ -n_x & -n_y & -n_z & -(eye \cdot \vec{n}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Move the origin to a new location

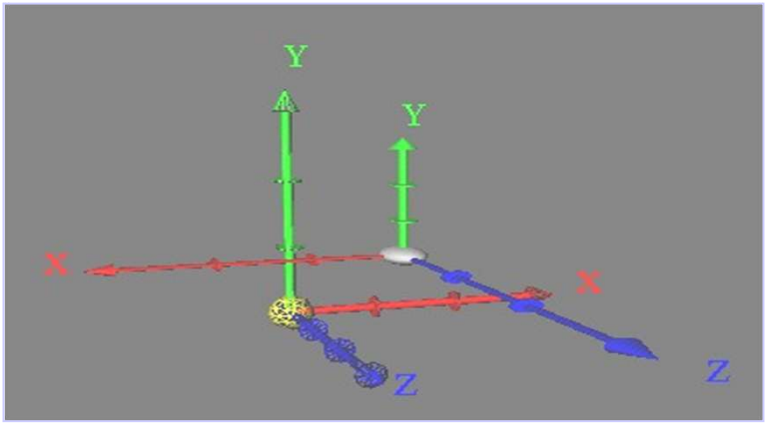
$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Scale

- Stretch, mirror or decimate a coordinate direction

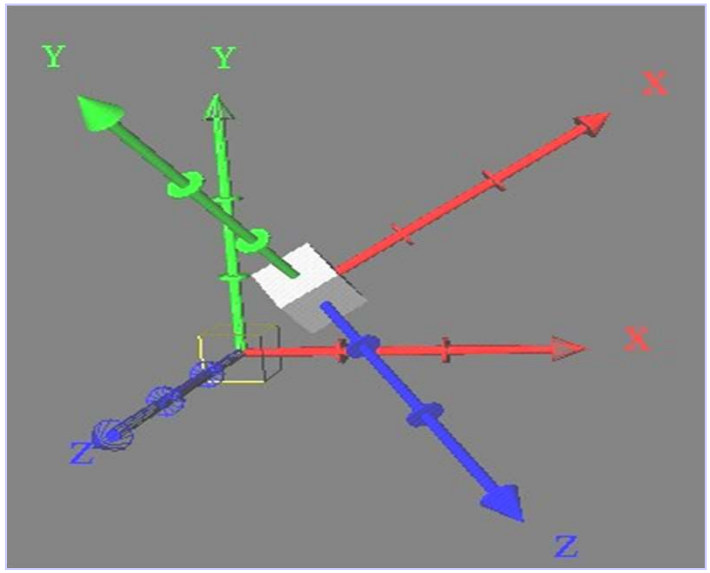
$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Note, there's a translation applied here to make things easier to see

Rotation

- Rotate coordinate system about an axis in space



Note, there's a translation applied here to make things easier to see

Rotation (cont'd)

$$\vec{v} = (x \quad y \quad z)$$
$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = (x' \quad y' \quad z')$$

$$M = \vec{u}^t\vec{u} + \cos(\theta)(I - \vec{u}^t\vec{u}) + \sin(\theta)S$$

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_{\vec{v}}(\theta) = \begin{pmatrix} & M & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vertex Shader for Rotation of Cube

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta
for
    // each of the three axes in one
computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Vertex Shader for Rotation of Cube (cont'd)

```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Vertex Shader for Rotation of Cube (cont'd)

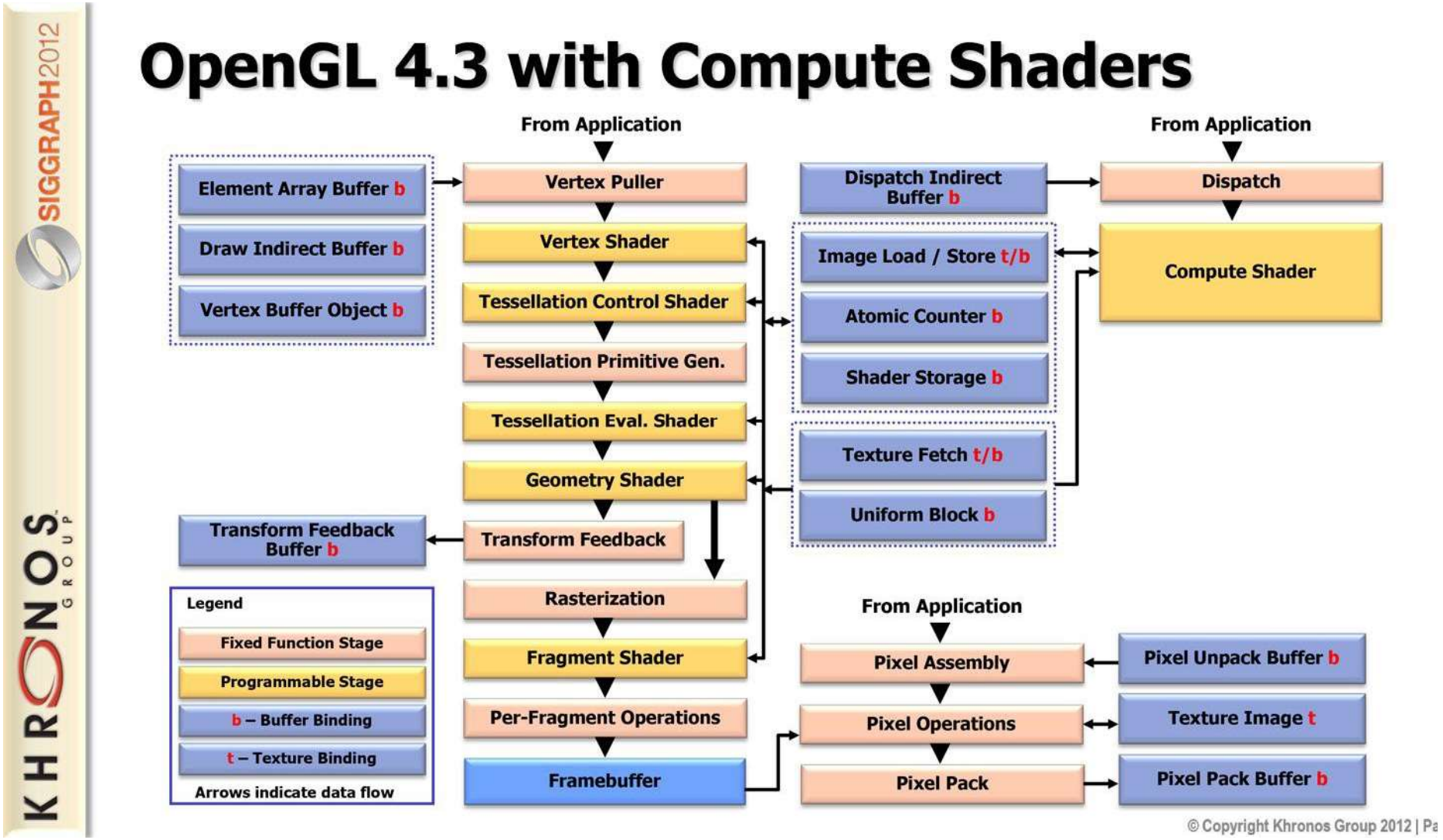
```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

OPENGL COMPUTE SHADER

Motivations

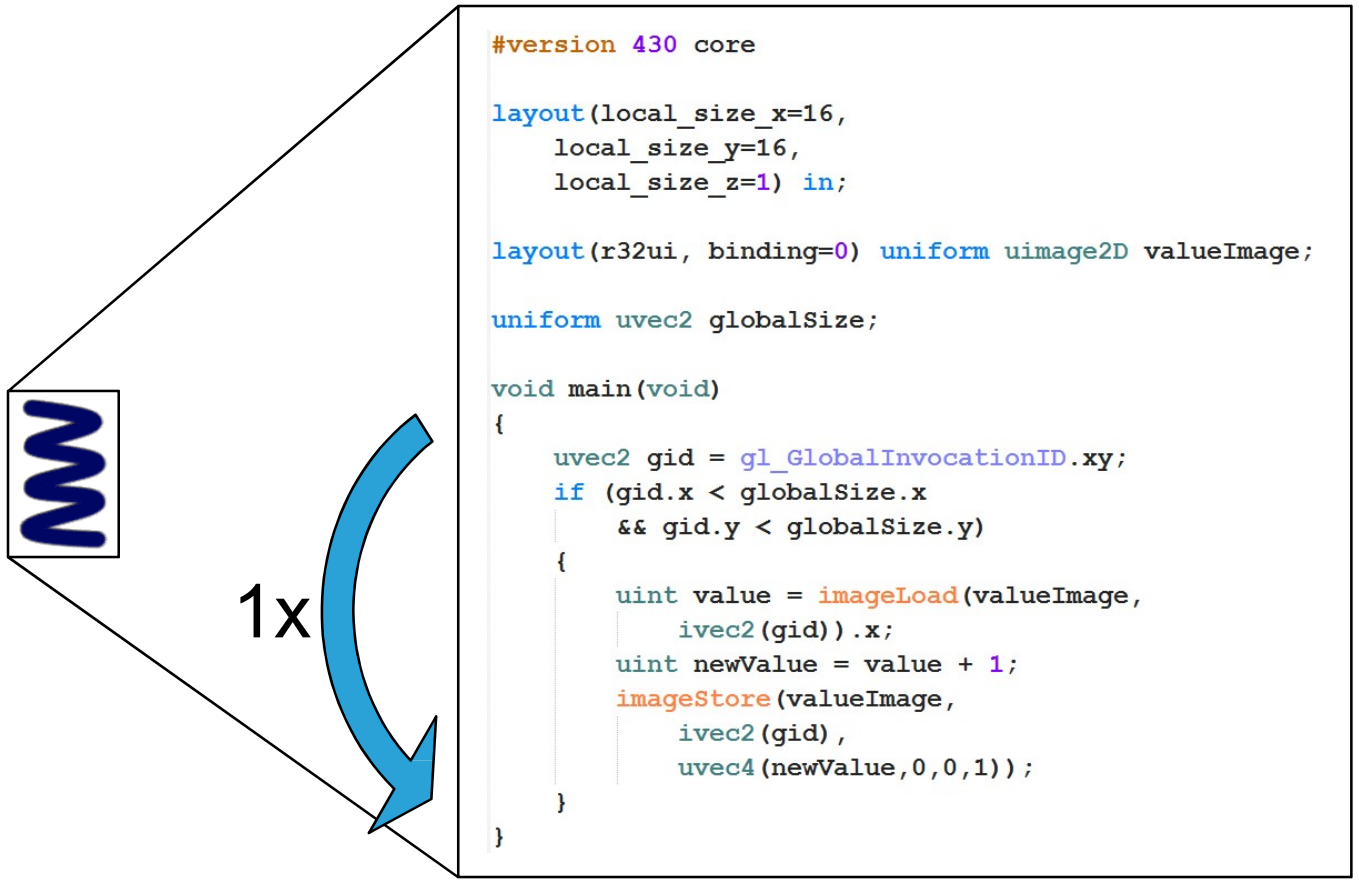
- Use parallel processing power of GPU for General Purpose (GP) computations
- Great for image processing, particles, simulations, etc.
- Implement any parallel SPMD algorithm!
  - **S**ingle **P**rogram, **M**ultiple **D**ata
- Why not OpenCL or CUDA?
  - One API for graphics and GP processing
    - Avoid interop
    - Avoid context switches
  - Same language: GLSL

Pipeline


OpenGL 4.3 with Compute Shaders
SIGGRAPH2012
KHRONOS GROUP
From Application
Element Array Buffer b
Draw Indirect Buffer b
Vertex Buffer Object b
Vertex Puller
Vertex Shader
Tessellation Control Shader
Tessellation Primitive Gen.
Tessellation Eval. Shader
Geometry Shader
Transform Feedback
Transform Feedback Buffer b
Rasterization
Fragment Shader
Per-Fragment Operations
Framebuffer
From Application
Dispatch Indirect Buffer b
Dispatch
Compute Shader
Image Load / Store t/b
Atomic Counter b
Shader Storage b
Texture Fetch t/b
Uniform Block b
From Application
Pixel Assembly
Pixel Operations
Pixel Pack
Pixel Unpack Buffer b
Texture Image t
Pixel Pack Buffer b
Legend
Fixed Function Stage
Programmable Stage
b – Buffer Binding
t – Texture Binding
Arrows indicate data flow
© Copyright Khronos Group 2012 | Pa

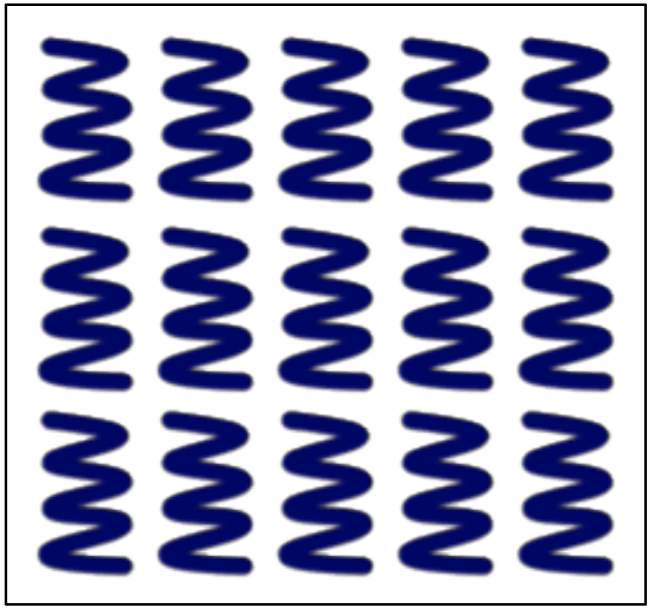
Thread Hierarchy

- Smallest execution unit:
- **Thread = Invocation**



Thread Hierarchy

- Grid of Threads:
  - **Work Group**
- 1D, 2D or 3D
- Size specified in shader code
- 2D example:

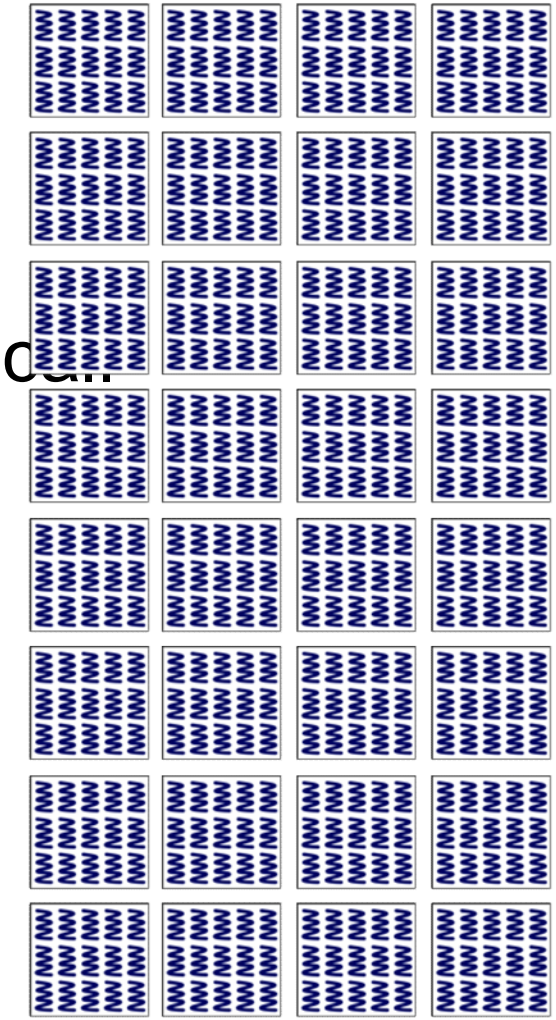


```
layout(
local_size_x      =5
local_size_y      =3,
   local_size_z   =1
) in;
```

Thread Hierarchy

- Grid of work groups:
  - **Dispatch**
- 1D, 2D or 3D
- Size specified in OpenGL call
- 2D example:

```
glDispatchCompute(
4       /*x*/,
8       /*y*/,
1   /*z*/
);
```

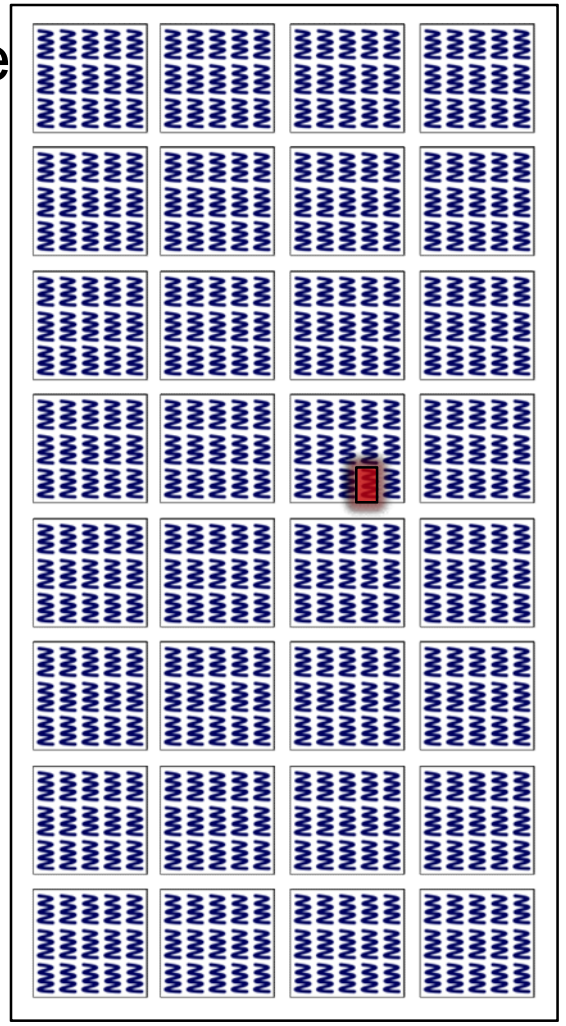


Thread Location

- GLSL built-in variables Have type

`uvec3`

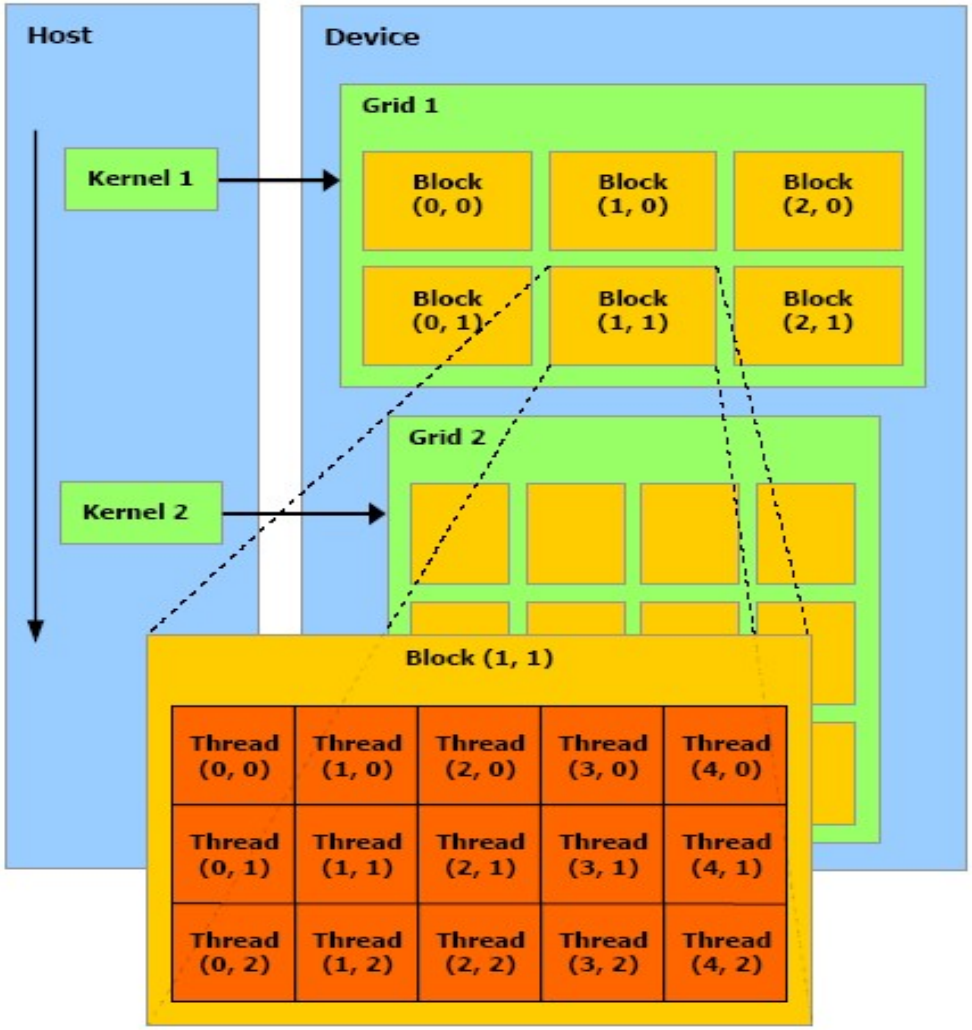
- 2D example:
- `gl_WorkGroupID`
- is `(2,4,0)`
  - `gl_LocalInvocationID`
  - is `(3,0,0)`
  - `gl_GlobalInvocationID`
  - is `(13,12,0)`



NOW, BACK TO CUDA

Case study: NVIDIA GTX 1080

CUDA Programming Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

CUDA Programming Model

- The GPU is seen as a *compute device* to execute a portion of an application that
  - Has to be executed many times
  - Can be isolated as a function
  - Works independently on different data
- Such a function can be compiled to run on the *device.* The resulting program is called a Kernel

CUDA Programming Model

- The batch of threads that executes a kernel is organized as a grid of thread blocks

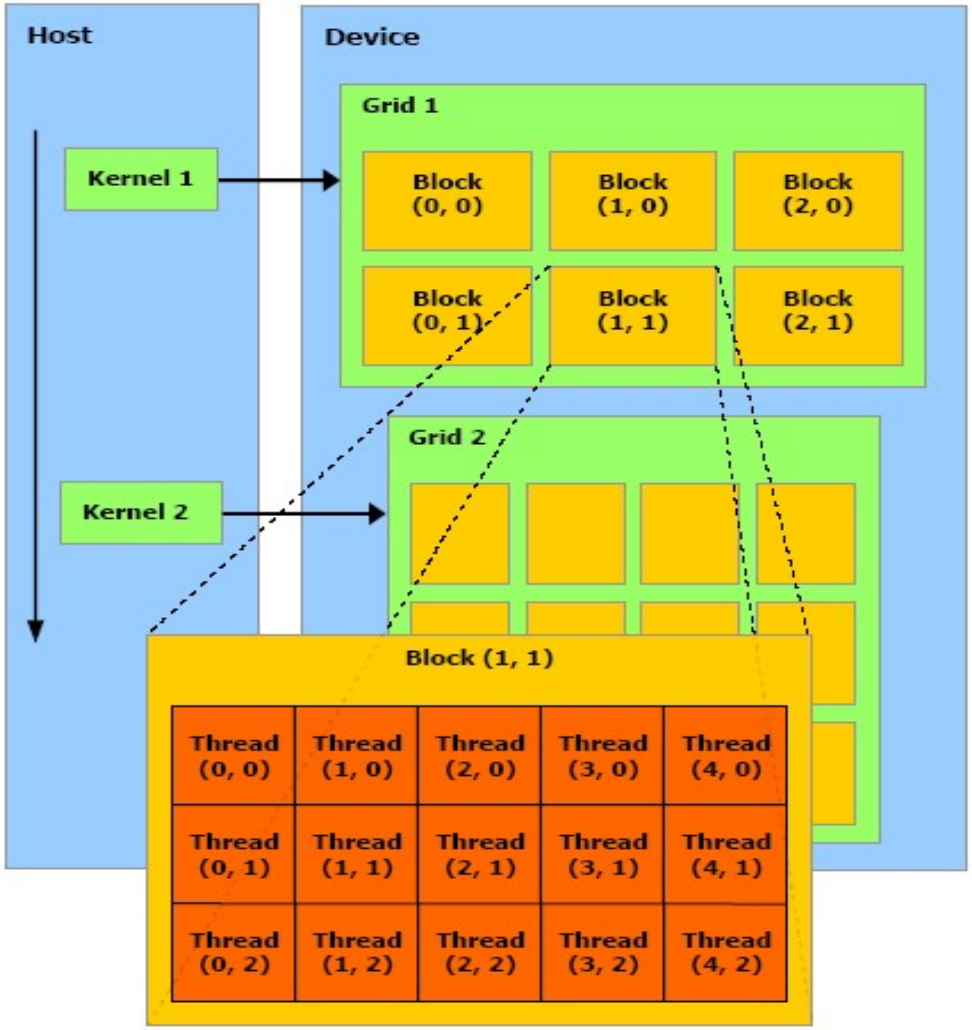
CUDA Programming Model

- Thread Block
  - Batch of threads that can cooperate together
    - Fast shared memory
    - Synchronizable
    - Thread ID
  - Limited number of threads in a block
- Threads in a block are executed in batches of SIMD threads
  - The SIMD thread batch is called Warp

CUDA Programming Model

- Grid of Thread Block
  - Allows larger numbers of thread to execute the same kernel with one invocation
  - Blocks identifiable via block ID
  - Leads to a reduction in thread cooperation

CUDA Programming Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Physical Limits for G80

Multiprocessors per GPU	16, 32, 320,…
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768/1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384

Physical Limits for G80

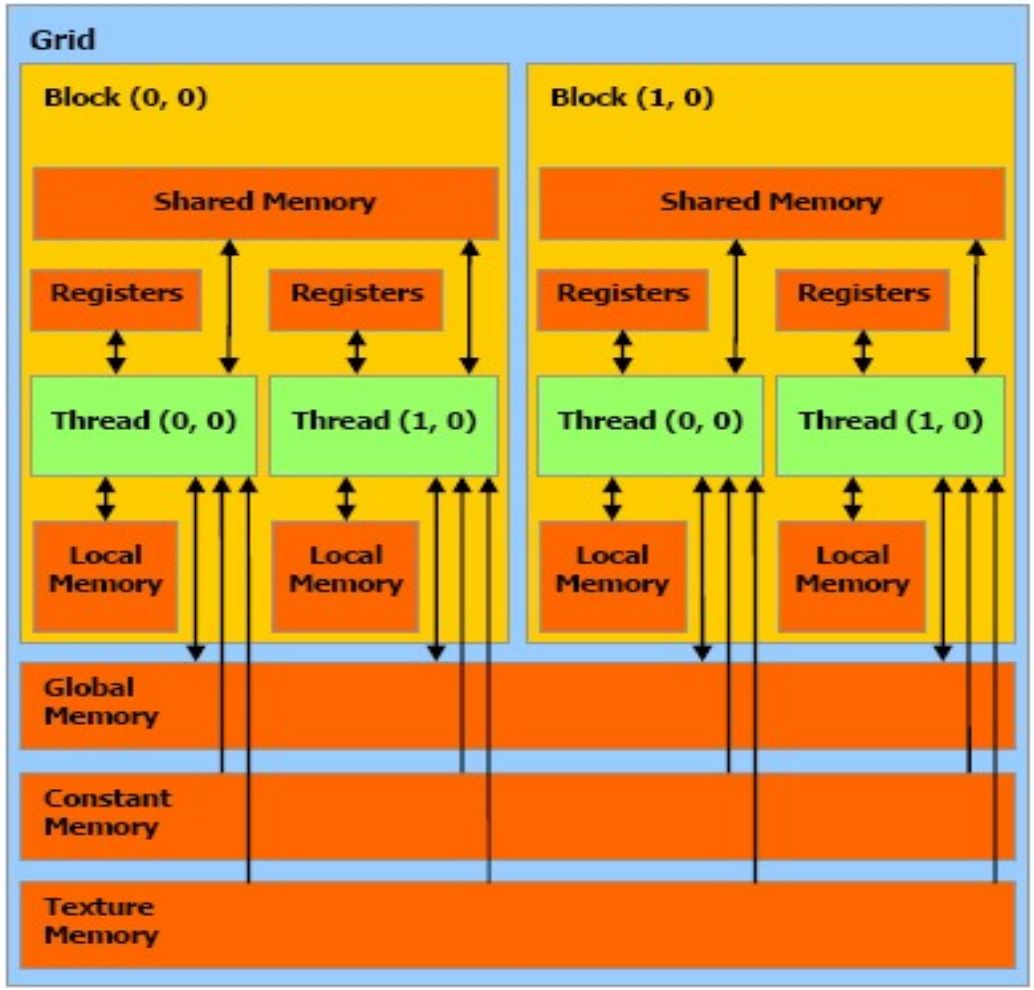
Maximum Thread Blocks Per Multiprocessor

Limited by Max Warps / Multiprocessor
Limited by Registers / Multiprocessor
Limited by Shared Memory / Multiprocessor

Overview

- Programming Model
- Memory Model
- CUDA API Basics
- Code Example

CUDA Memory Model



CUDA Memory Model

- A thread has only access to the device's DRAM and on-chip memory
  - Read-write per-thread *registers*,
  - Read-write per-thread *local memory*,
  - Read-write per-block *shared memory*,
  - Read-write per-grid *global memory*,
  - Read-only per-grid *constant memory*,
  - Read-only per-grid *texture memory*.

CUDA Memory Model

- global, constant, and texture memory
  - Can be read from or written to by the host
  - persistent across kernel calls by one application
  - optimized for different memory usages

CUDA Memory Model

- Global Memory
  - Is not cached
  - Important to follow the right access pattern
  - Read 64-bit or 128-bit words with single instruction
  - Alignment requirements automatically fulfilled for built-in types
  - Alignment requirements can be enforced by alignment specifiers.

CUDA Memory Model

- Constant Memory
  - Is cached

    → Read from memory only on cache miss

  - Constant cache is as fast as read from register when all threads read same address

  - Cost scales linearly with number of different addresses read by all threads

CUDA Memory Model

- Texture Memory
  - Is cached
    - → Read from memory only on cache miss.
- Texture cache
  - Optimized for 2D spatial locality,
  - Best performance for threads of same warp that read addresses that are close together.

CUDA Memory Model

- Memory Instructions
  - 2 clock cycles to issue memory instruction
  - Global memory additional 200 to 300 clock cycles of memory latency
  - Global memory latency can be hidden by thread scheduler → sufficient independent arithmetic instructions

CUDA Memory Model

- Memory Bandwidth
  - Depends on the memory access pattern
  - Device memory is of much higher latency and lower bandwidth than on-chip memory

→ Device memory accesses should be minimized

CUDA Memory Model

- Reduce access to the device memory by staging data into shared memory.

  Each thread of a block

  - loads data from device to shared memory,
  - synchronizes with other threads that each thread can safely read shared memory,
  - processes the data in shared memory,
  - synchronizes if necessary to make sure shared memory is up to date,
  - writes results back to device memory.

CUDA Memory Model

- Shared Memory
  - Is on-chip:
    - much faster than the local and global memory,
    - as fast as a register when no bank conflicts,
    - divided into equally-sized memory banks.
  - Successive 32-bit words are assigned to successive banks,
  - Each bank has a bandwidth of 32 bits per clock cycle.

CUDA Memory Model

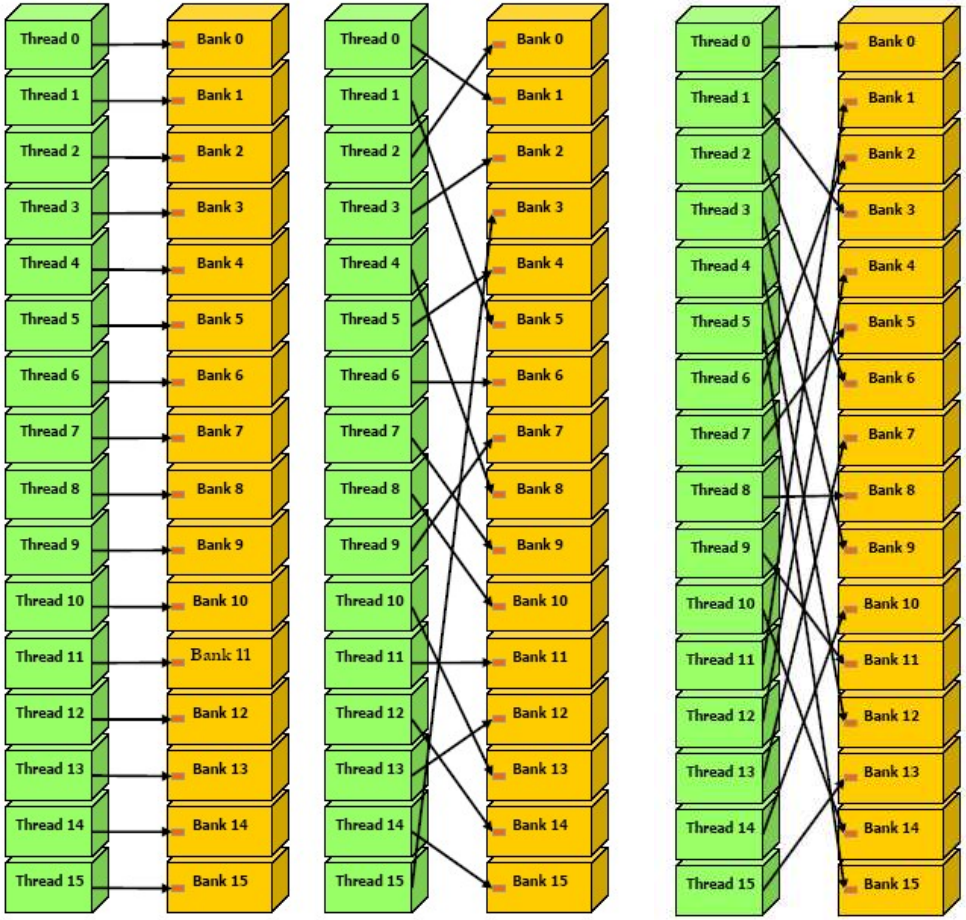
- Shared Memory

  Reminder: warp size is 32, number of banks is 16

  - memory request requires two cycles for a warp
    - One for the first half, one for the second half of the warp
    - →No conflicts between threads from first and second half

CUDA Memory Model

- Shared Memory



CUDA Memory Model

- Shared Memory
  - However we have to be careful accessing arrays with elements larger or smaller than 32-bit. E.g. char-array:

```
__shared__ char shared[32];
char data = shared[tid];
```

  - Conflict, since `shared[0]`, `shared[1]`, `shared[2]`, and `shared[3]` are in the same bank.

  - Solution:

```
char data = shared[BaseIndex + 4 * tid];
```

CUDA Memory Model

- Shared Memory
  - Structures:
    as many memory requests as it has members

Conflict:

```
struct type {
        float x, y;
}a[32];
…=a[tid].x+1;
```

No conflict:

```
struct type {
        float x, y, z;
}a[32];
…=a[tid].x+1;
```

Basic optimizations

- To achieve a good performance with a CUDA kernel function two important optimizations are to:
  - Achieve the highest GPU occupancy by optimizing for block level parallelism:
    - Registers
    - Shared memory
    - Block size
  - Organize memory layout and access patterns to reduce
    - Access to global memory
    - Bank conflicts for shared memory access