An Introduction to Modern GPU Architecture

Ashu Rege
Director of Developer Technology

Agenda

- Evolution of GPUs
- Computing Revolution
- Stream Processing
- Architecture details of modern GPUs

Evolution of GPUs

Evolution of GPUs (1995-1999)

- 1995 – NV1
- 1997 – Riva 128 (NV3), DX3
- 1998 – Riva TNT (NV4), DX5
  - 32 bit color, 24 bit Z, 8 bit stencil
  - Dual texture, bilinear filtering
  - 2 pixels per clock (ppc)
- 1999 – Riva TNT2 (NV5), DX6
  - Faster TNT
  - 128b memory interface
  - 32 MB memory
  - The chip that would not die ☺

Virtua Fighter
(SEGA Corporation)

NV1
50K triangles/sec
1M pixel ops/sec
1M transistors
16-bit color
Nearest filtering

1995

Evolution of GPUs
(Fixed Function)

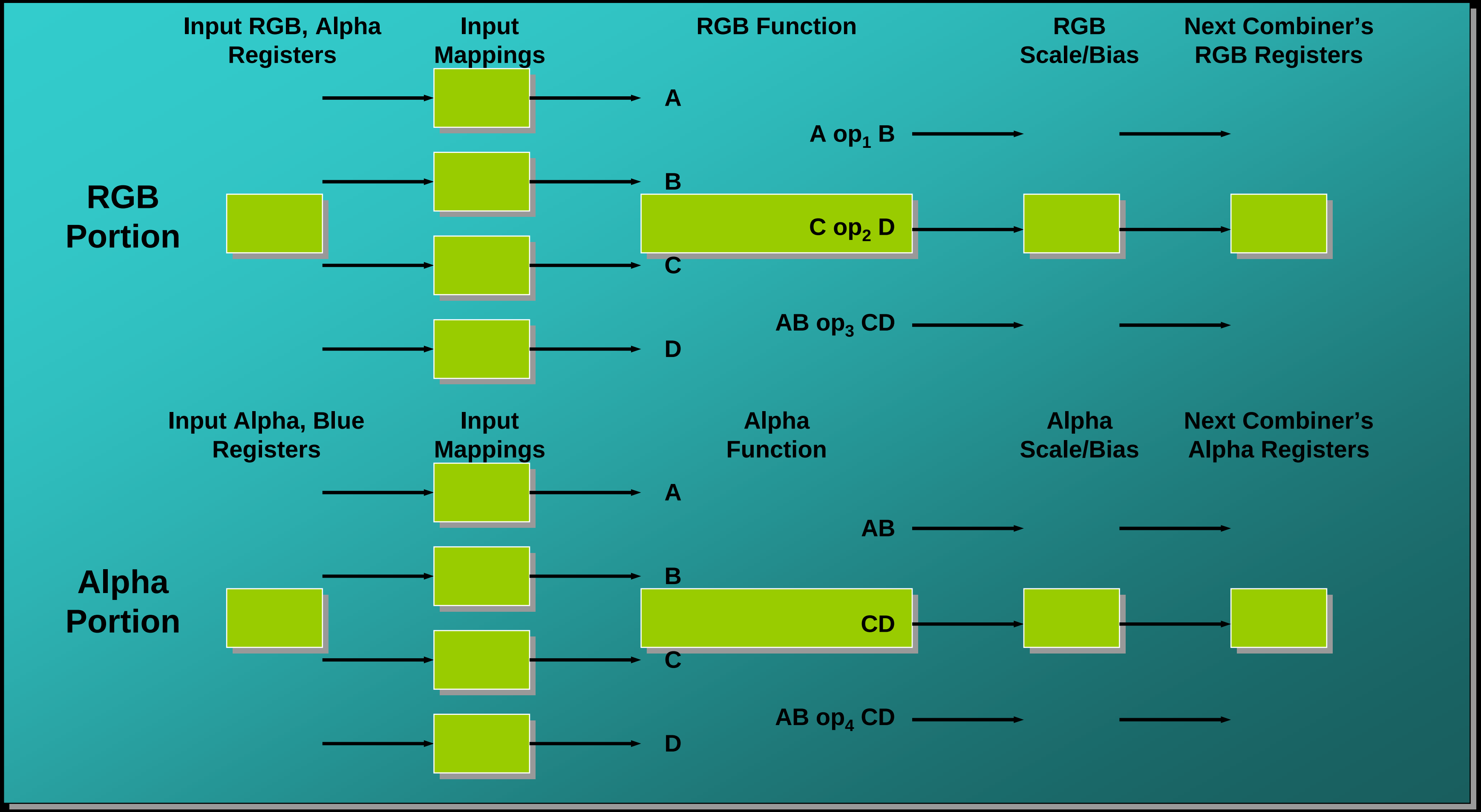
- GeForce 256 (NV10)
- DirectX 7.0
- Hardware T&L
- Cubemaps
- DOT3 – bump mapping
  - Register combiners
- 2x Anisotropic filtering
- Trilinear filtering
- DXT texture compression
- 4 ppc
- Term “GPU” introduced

Deus Ex
(Eidos/Ion Storm)

NV10
15M triangles/sec
480M pixel ops/sec
23M transistors
32-bit color
Trilinear filtering

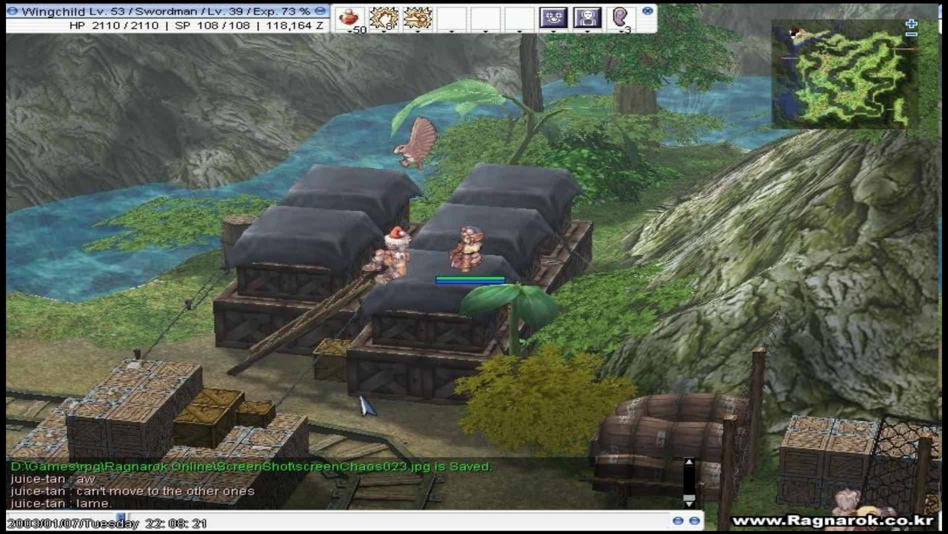
1999

NV10 – Register Combiners



Evolution of GPUs (Shader Model 1.0)

- GeForce 3 (NV20)
  - NV2A – Xbox GPU
- DirectX 8.0
- Vertex and Pixel Shaders
- 3D Textures
- Hardware Shadow Maps
- 8x Anisotropic filtering
- Multisample AA (MSAA)
- 4 ppc



Wingchild Lv. 53 / Swordman / Lv. 39 / Exp. 73 %

HP 2110 / 2110 | SP 108 / 108 | 118,164 Z

D:\Games\rpg\Ragnarok Online\ScreenShot\screenChaos023.jpg is Saved.
juice-tan : aw
juice-tan : can't move to the other ones
juice-tan : lame.

2003/01/07/Tuesday 22: 08: 21

Ragnarok Online
(Atari/Gravity)

NV20
100M triangles/sec
1G pixel ops/sec
57M transistors
Vertex/Pixel shaders
MSAA

2001

Evolution of GPUs (Shader Model 2.0)

- GeForce FX Series (NV3x)
- DirectX 9.0
- Floating Point and “Long” Vertex and Pixel Shaders
- Shader Model 2.0
  - 256 vertex ops
  - 32 tex + 64 arith pixel ops
- Shader Model 2.0a
  - 256 vertex ops
  - Up to 512 ops
- Shading Languages
  - HLSL, Cg, GLSL

Dawn Demo
(NVIDIA)

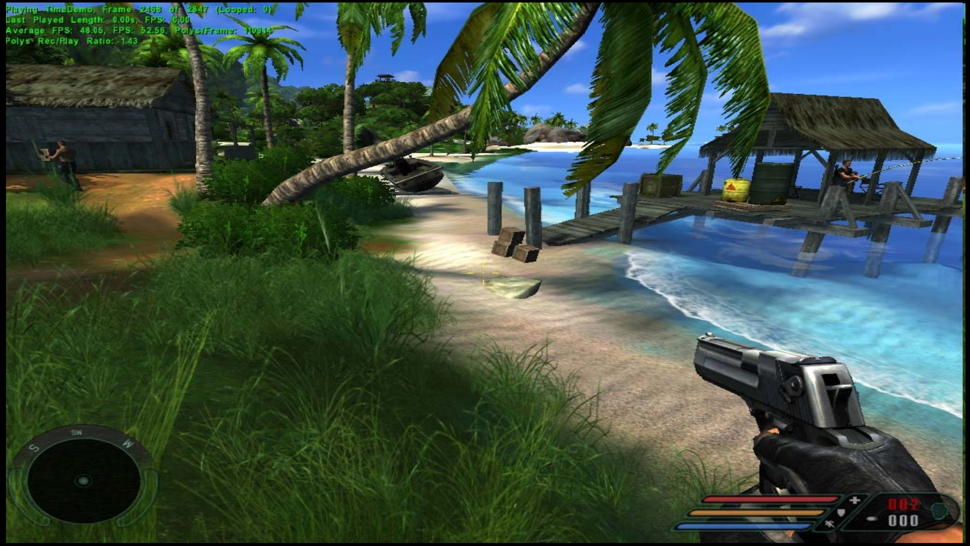
NV30
200M triangles/sec
2G pixel ops/sec
125M transistors
Shader Model 2.0a

2003

Evolution of GPUs (Shader Model 3.0)

- GeForce 6 Series (NV4x)
- DirectX 9.0c
- Shader Model 3.0
- Dynamic Flow Control in Vertex and Pixel Shaders[1]
  - Branching, Looping, Predication, …
- Vertex Texture Fetch
- High Dynamic Range (HDR)
  - 64 bit render target
  - FP16x4 Texture Filtering and Blending

[1] *Some flow control first introduced in SM2.0a*

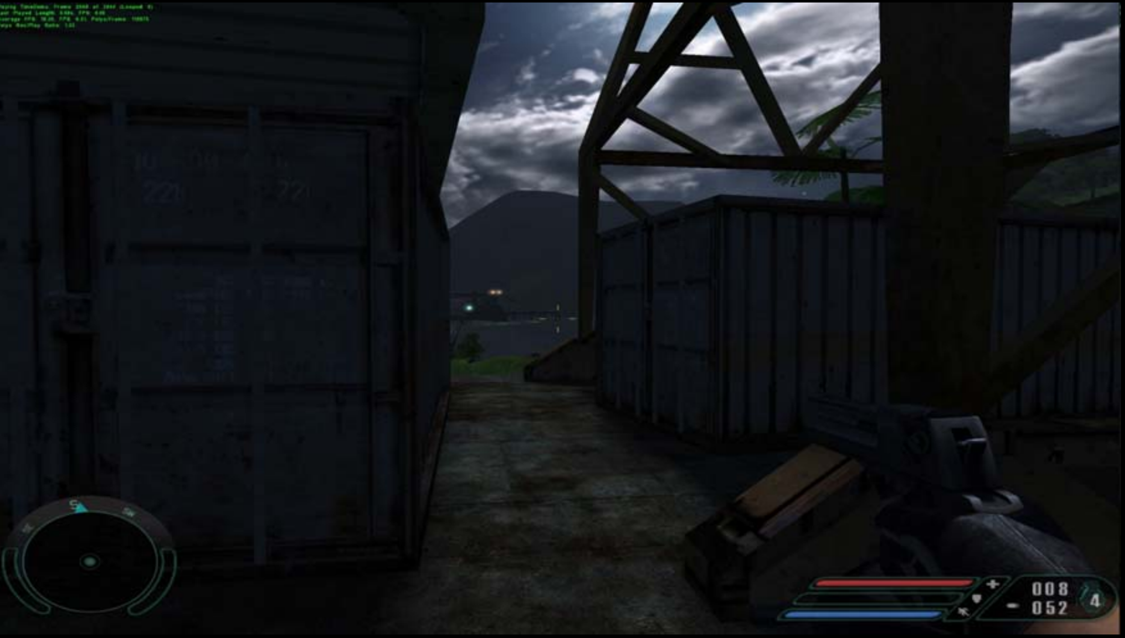


```
Playing TimeDemo, Frame 2468 of 2847 (Looped: 0)
Last Played Length: 0.00s, FPS: 0.00
Average FPS: 48.06, FPS: 52.56, Polys/Frame: 110344
Polys Rec/Play Ratio: 1.43
```

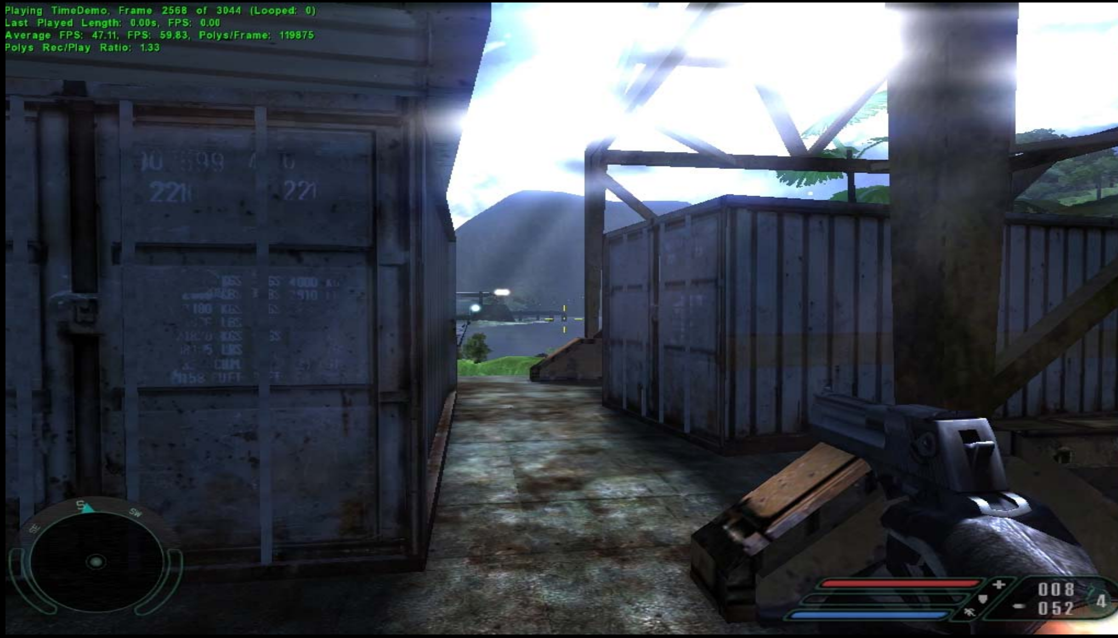
Far Cry HDR
(Ubisoft/Crytek)

NV40
600M triangles/sec
12.8G pixel ops/sec
220M transistors
Shader Model 3.0
Rotated Grid MSAA
16x Aniso, SLI
2004

Far Cry – No HDR/HDR Comparison



```
laying TimeDemo, Frame [illegible] of 3044 (Looped: 0)
ast Played Length: 0.00s, FPS: 0.00
verage FPS: [illegible], FPS: [illegible], Polys/Frame: [illegible]
olys Rec/Play Ratio: 1.33
```



```
Playing TimeDemo, Frame 2568 of 3044 (Looped: 0)
Last Played Length: 0.00s, FPS: 0.00
Average FPS: 47.11, FPS: 59.83, Polys/Frame: 119875
Polys Rec/Play Ratio: 1.33
```

Evolution of GPUs (Shader Model 4.0)

- GeForce 8 Series (G8x)
- DirectX 10.0
  - Shader Model 4.0
  - Geometry Shaders
  - No “caps bits”
  - Unified Shaders
- New Driver Model in Vista
- CUDA based GPU computing
- GPUs become true *computing processors* measured in GFLOPS

Crysis
(EA/Crytek)

G80
Unified Shader Cores w/ Stream Processors
681M transistors
Shader Model 4.0
8x MSAA, CSAA

2006

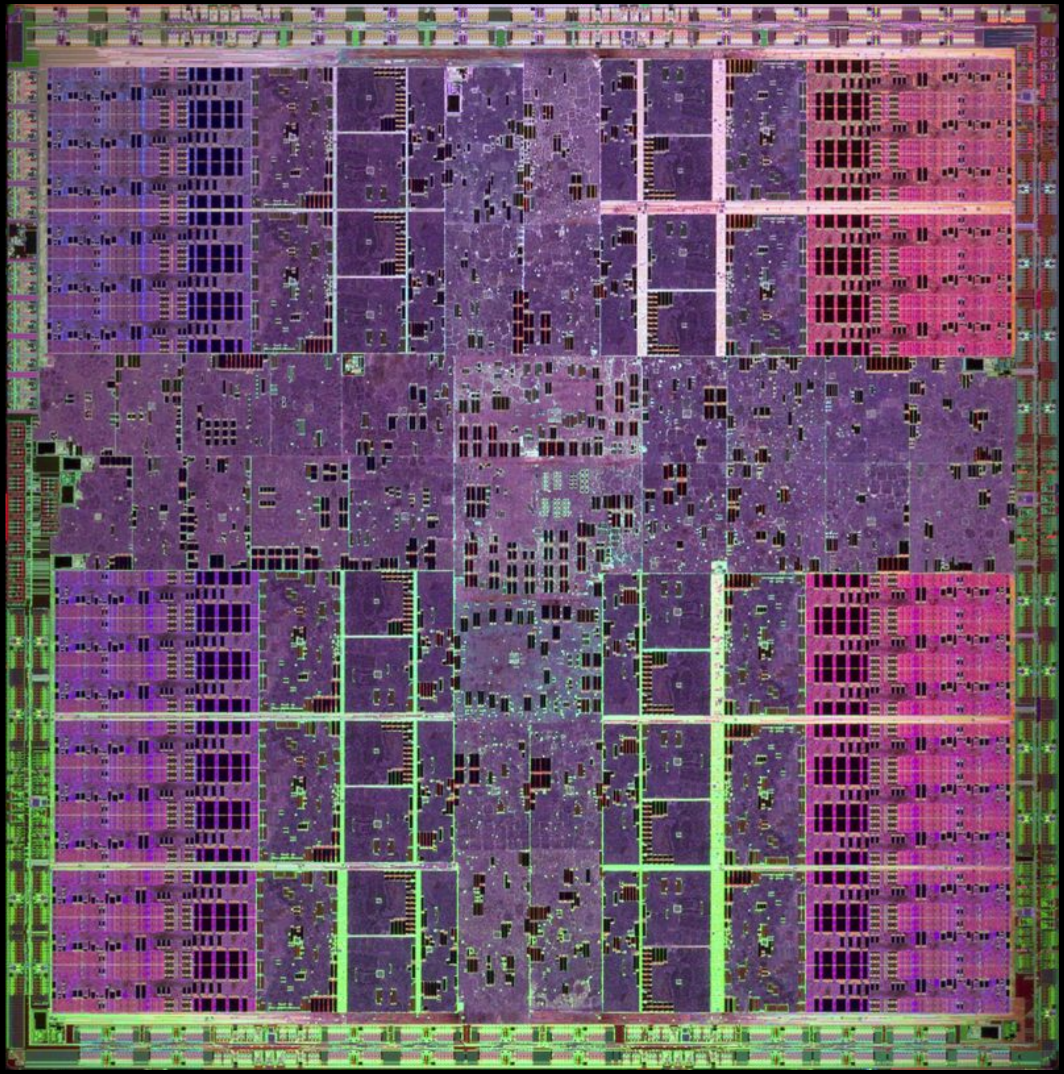
Crysis. Images courtesy of Crytek.

As Of Today…

- GeForce GTX 280 (GT200)
- DX10
- 1.4 billion transistors
- 576 mm^2 in 65nm CMOS

- 240 stream processors
- 933 GFLOPS peak
- 1.3GHz processor clock

- 1GB DRAM
- 512 pin DRAM interface
- 142 GB/s peak



Stunning Graphics Realism

Lush, Rich Worlds

Crysis © 2006 Crytek / Electronic Arts

POWERED BY NVIDIA.

Hellgate: London © 2005-2006 Flagship Studios, Inc. Licensed by NAMCO BANDAI Games America, Inc.

Incredible Physics Effects

Core of the Definitive Gaming Platform

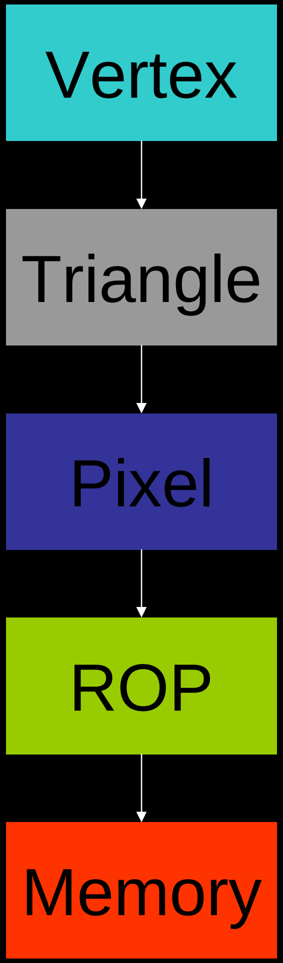
What Is Behind This Computing Revolution?

- Unified Scalar Shader Architecture
- Highly Data Parallel Stream Processing
- Next, let’s try to understand what these terms mean…

Unified Scalar Shader Architecture

Graphics Pipelines For Last 20 Years

Processor per function



Vertex

Triangle

Pixel

ROP

Memory

T&L evolved to vertex shading

Triangle, point, line – setup

Flat shading, texturing, eventually pixel shading

Blending, Z-buffering, antialiasing

Wider and faster over the years

Shaders in Direct3D

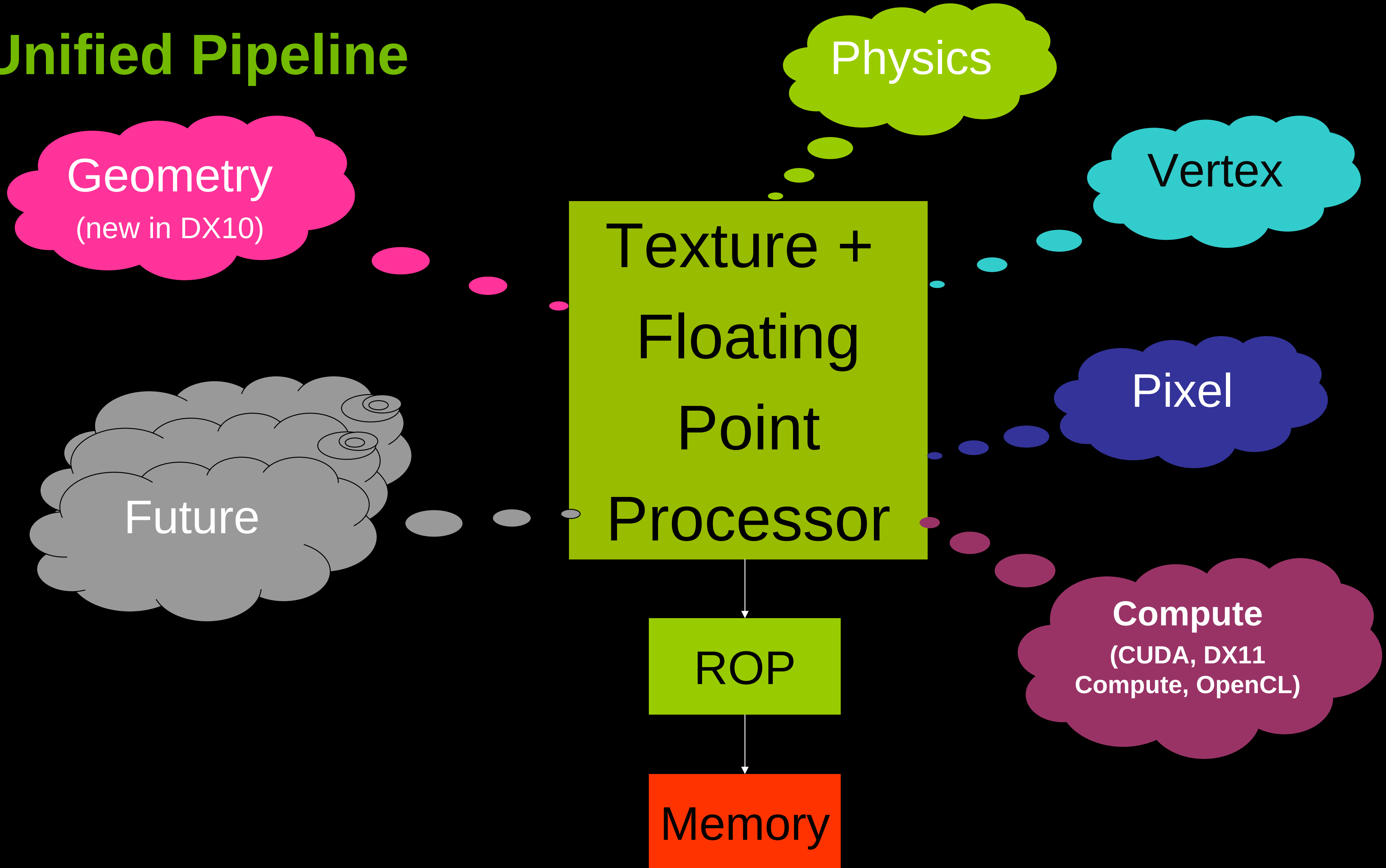
- **DirectX 9:**
  Vertex Shader, Pixel Shader

- **DirectX 10:**
  Vertex Shader, Geometry Shader, Pixel Shader

- **DirectX 11:**
  Vertex Shader, Hull Shader, Domain Shader, Geometry Shader, Pixel Shader, Compute Shader

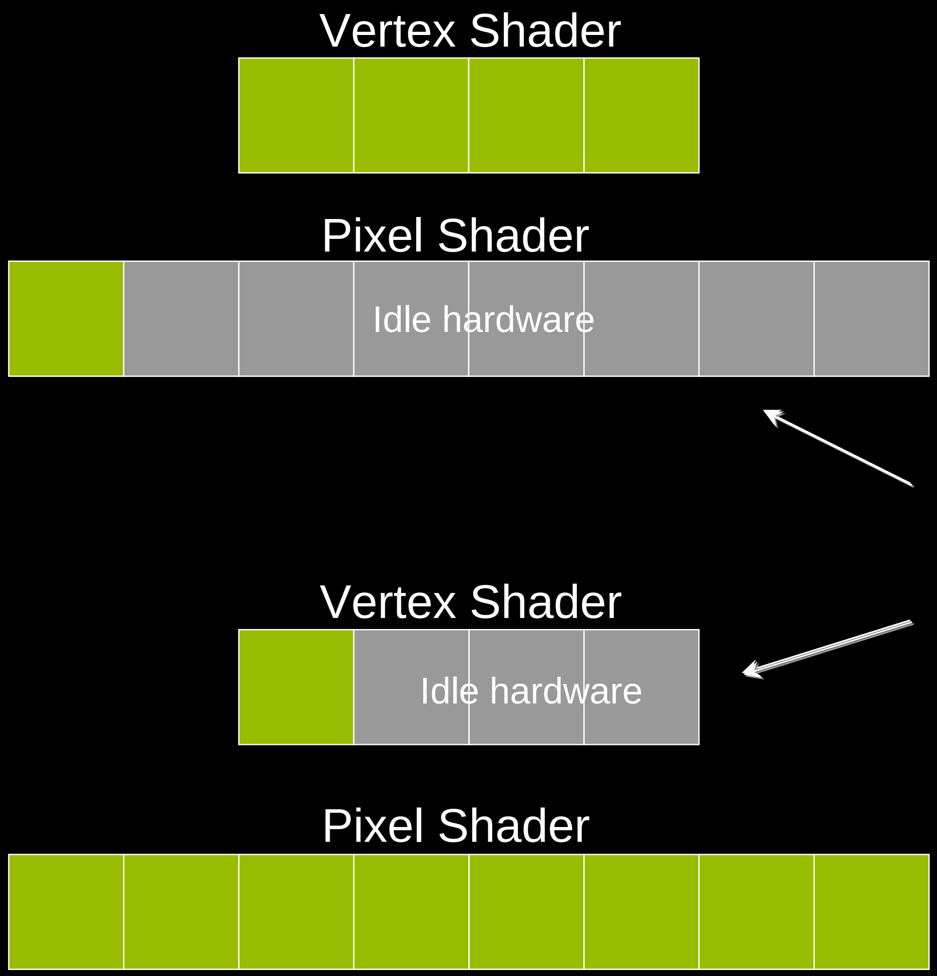
- **Observation: *All of these shaders require the same basic functionality*: Texturing (or Data Loads) and Math Ops.**

Unified Pipeline



Physics
Geometry
(new in DX10)
Vertex
Texture + Floating Point Processor
Pixel
Future
Compute
(CUDA, DX11 Compute, OpenCL)
ROP
Memory

Why Unify?



Vertex Shader

Pixel Shader

Idle hardware

Vertex Shader

Idle hardware

Pixel Shader

Unbalanced and inefficient utilization in non-unified architecture

Heavy Geometry

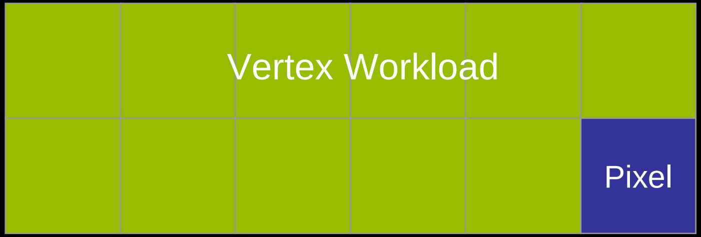
Workload Perf = 4

Heavy Pixel

Workload Perf = 8

Why Unify?

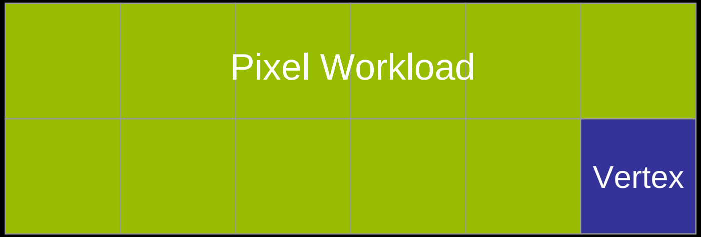
Unified Shader



Optimal utilization

In unified architecture

Unified Shader



Heavy Geometry

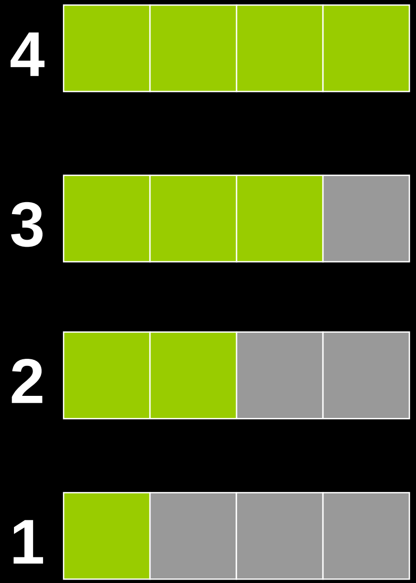
Workload Perf = 11

Heavy Pixel

Workload Perf = 11

Why Scalar Instruction Shader (1)

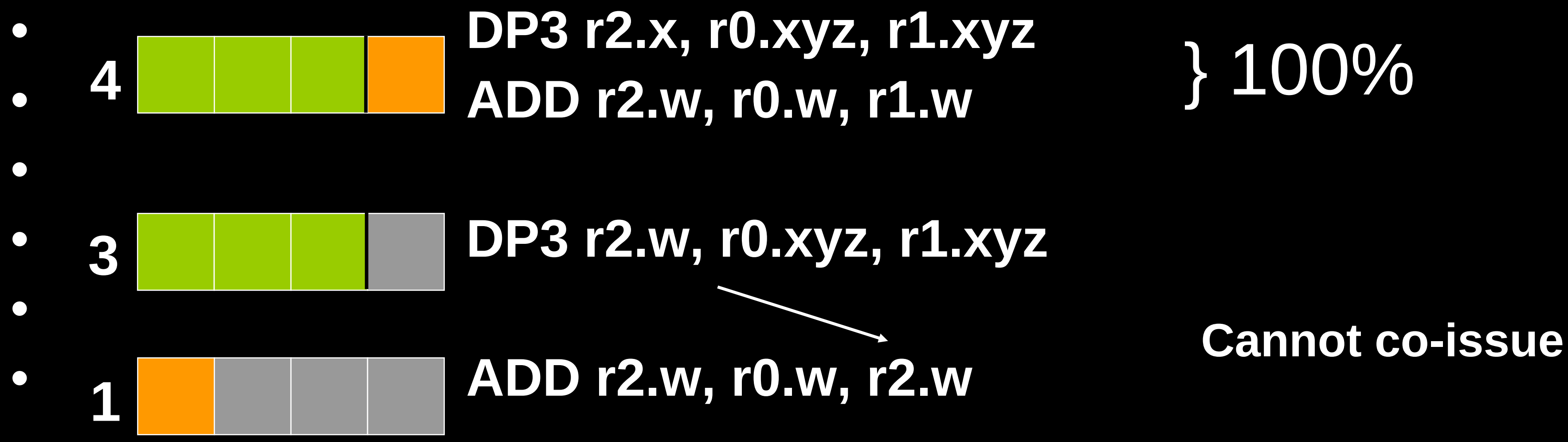
- Vector ALU – efficiency varies



- MAD r2.xyzw, r0.xyzw, r1.xyzw – 100% utilization
- DP3 r2.w, r0.xyz, r1.xyz – 75%
- MUL r2.xy, r0.xy, r1.xy – 50%
- ADD r2.w, r0.x, r1.x – 25%

Why Scalar Instruction Shader (2)

- Vector ALU with co-issue – better but not perfect



DP3 r2.x, r0.xyz, r1.xyz

ADD r2.w, r0.w, r1.w

} 100%

DP3 r2.w, r0.xyz, r1.xyz

ADD r2.w, r0.w, r2.w

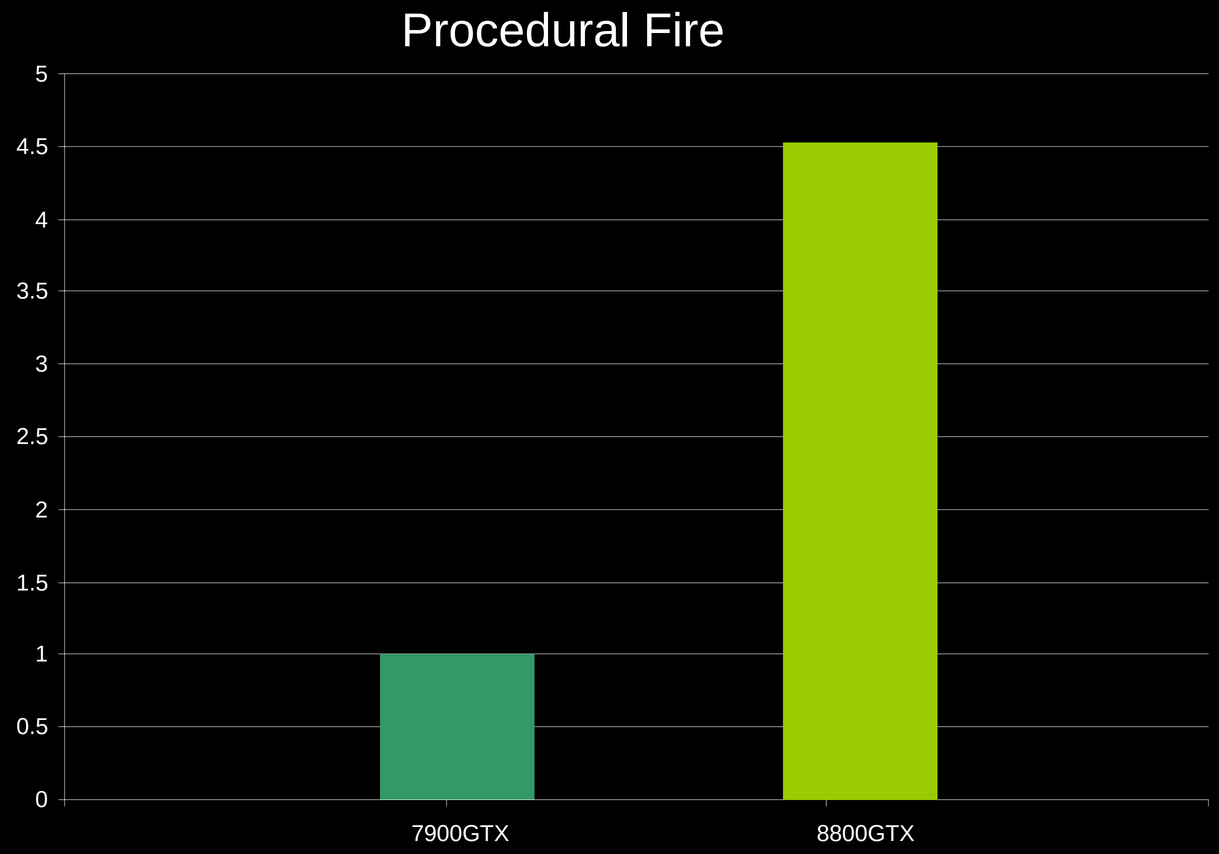
Cannot co-issue

- Vector/VLIW architecture – More compiler work required

- G8x, GT200: scalar – always 100% efficient, simple to compile
- Up to 2x effective throughput advantage relative to vector

Complex Shader Performance on Scalar Arch.

Procedural Perlin Noise Fire

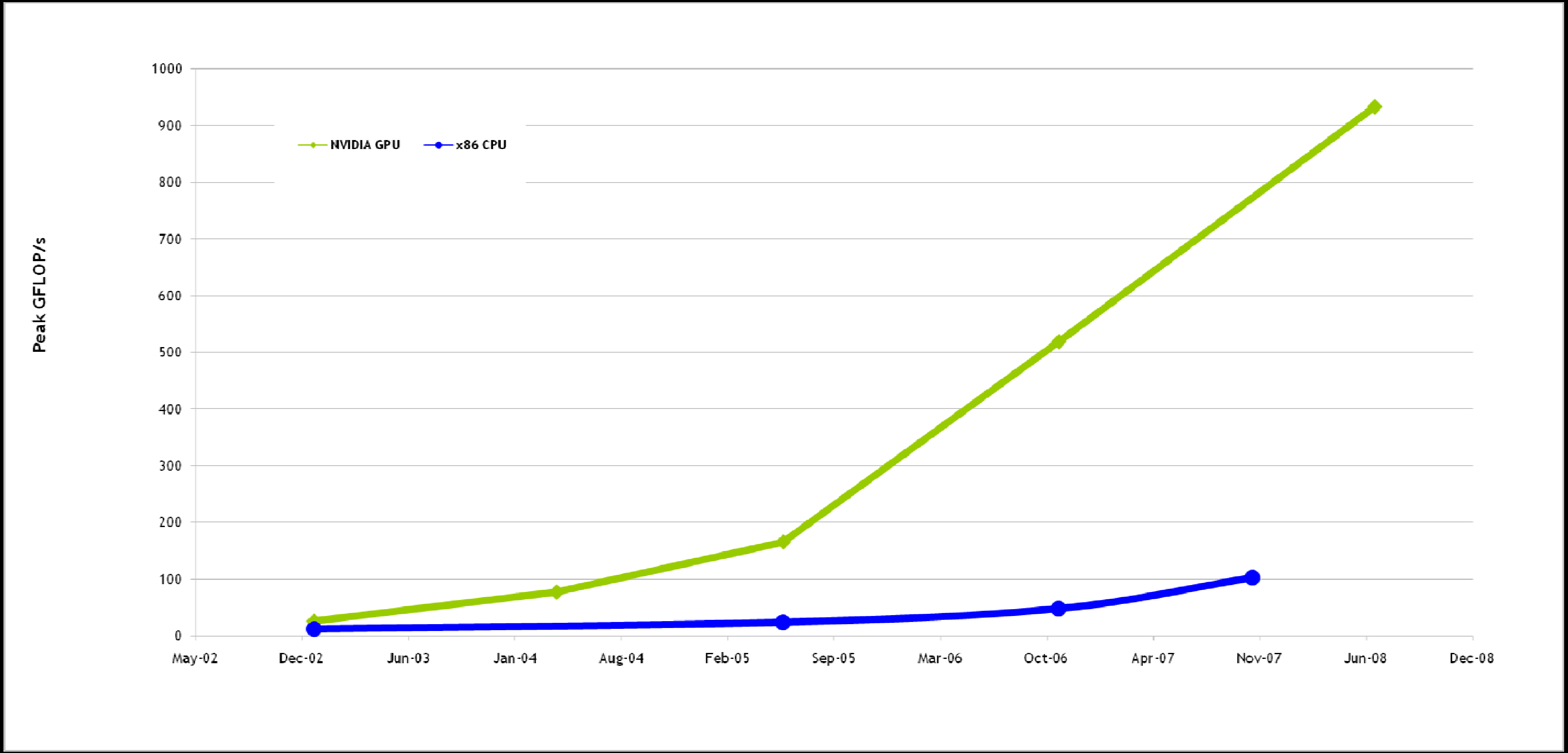


Conclusion

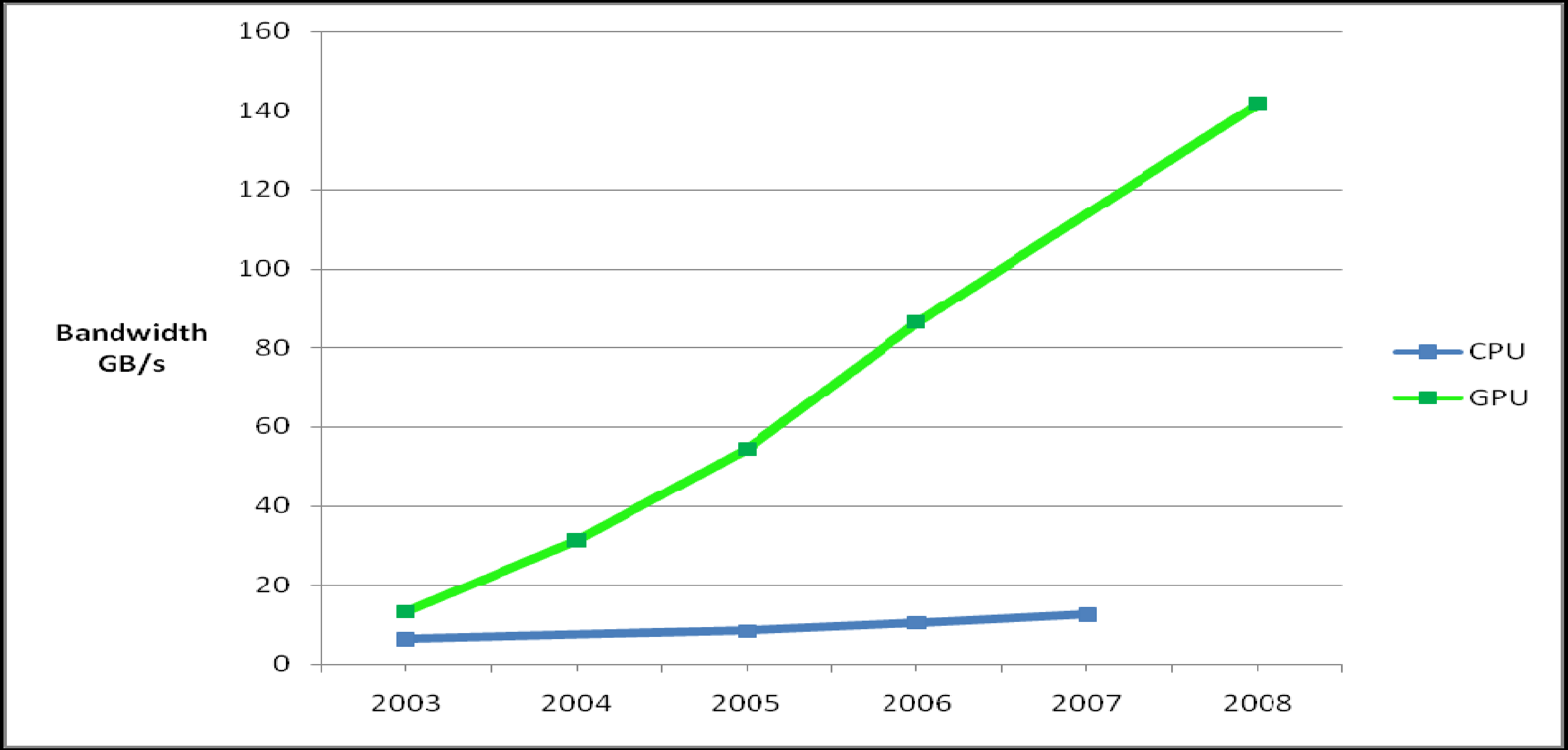
- **Build a *unified* architecture with *scalar* cores where all shader operations are done on the same processors**

Stream Processing

The Supercomputing Revolution (1)



The Supercomputing Revolution (2)



What Accounts For This Difference?

- Need to understand how CPUs and GPUs differ
  - Latency Intolerance versus Latency Tolerance
  - Task Parallelism versus Data Parallelism
  - Multi-threaded Cores versus SIMT (Single Instruction Multiple Thread) Cores
  - 10s of Threads versus 10,000s of Threads

Latency and Throughput

- "Latency is a *time delay* between the moment something is initiated, and the moment one of its effects begins or becomes detectable"
  - For example, the time delay between a request for texture reading and texture data returns
- Throughput is the amount of work done in a given amount of time
  - For example, how many triangles processed per second
- CPUs are low latency low throughput processors
- GPUs are high latency high throughput processors

Latency (1)

- **GPUs are designed for tasks that can tolerate latency**
  - **Example: Graphics in a game (simplified scenario):**

CPU	Generate Frame 0	Generate Frame 1	Generate Frame 2
GPU	Idle	Render Frame 0	Render Frame 1

Latency between frame generation and rendering (order of milliseconds)

- **To be efficient, GPUs must have *high throughput*, i.e. processing millions of pixels in a single frame**

Latency (2)

- CPUs are designed to minimize latency
  - Example: Mouse or keyboard input

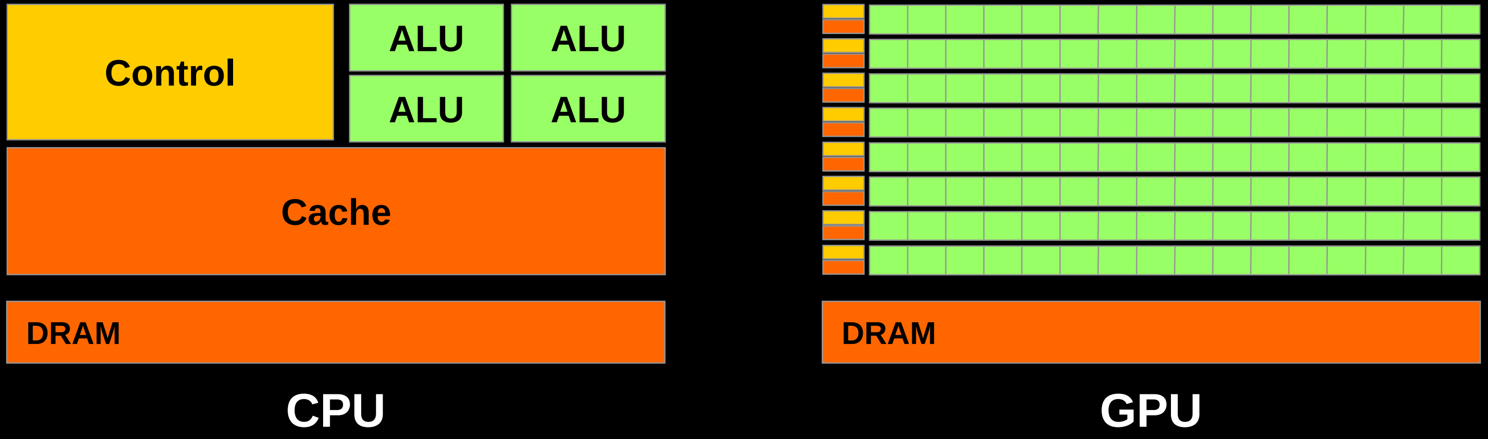
- Caches are needed to minimize latency

- CPUs are designed to maximize running operations out of cache
  - Instruction pre-fetch
  - Out-of-order execution, flow control

- → CPUs *need* a large cache, GPUs do not
  - GPUs can dedicate more of the transistor area to computation horsepower

CPU versus GPU Transistor Allocation

- GPUs can have more ALUs for the same sized chip and therefore run many more threads of computation



- Modern GPUs run 10,000s of threads concurrently

Managing Threads On A GPU

- How do we:
  - Avoid synchronization issues between so many threads?
  - Dispatch, schedule, cache, and context switch 10,000s of threads?
  - Program 10,000s of threads?

- Design GPUs to run specific types of threads:
  - Independent of each other – no synchronization issues
  - SIMD (Single Instruction Multiple Data) threads – minimize thread management
    - Reduce hardware overhead for scheduling, caching etc.
  - Program blocks of threads (e.g. one pixel shader per draw call, or group of pixels)

- Any problems which can be solved with this type of computation?

Data Parallel Problems

- Plenty of problems fall into this category (luckily ☺)
  - Graphics, image & video processing, physics, scientific computing, …
- This type of parallelism is called *data parallelism*
- And GPUs are the perfect solution for them!
  - In fact the more the data, the more efficient GPUs become at these algorithms
  - Bonus: You can relatively easily add more processing cores to a GPU and increase the throughput

Parallelism in CPUs v. GPUs

- CPUs use *task parallelism*
  - Multiple tasks map to multiple threads
  - Tasks run different instructions
  - 10s of relatively heavyweight threads run on 10s of cores
  - Each thread managed and scheduled explicitly
  - Each thread has to be individually programmed

- GPUs use *data parallelism*
  - SIMD model (Single Instruction Multiple Data)
  - Same instruction on different data
  - 10,000s of lightweight threads on 100s of cores
  - Threads are managed and scheduled by hardware
  - Programming done for batches of threads (e.g. one pixel shader per group of pixels, or draw call)

Stream Processing

- What we just described:
  - Given a (typically large) set of data ("stream")
  - Run the same series of operations ("kernel" or "shader") on all of the data (SIMD)

- GPUs use various optimizations to improve throughput:
  - Some on-chip memory and local caches to reduce bandwidth to external memory
  - Batch groups of threads to minimize incoherent memory access
    - Bad access patterns will lead to higher latency and/or thread stalls.
  - Eliminate unnecessary operations by exiting or killing threads
    - Example: Z-Culling and Early-Z to kill pixels which will not be displayed

To Summarize

- GPUs use *stream processing* to achieve high throughput
  - GPUs designed to solve problems that tolerate high latencies
  - High latency tolerance → Lower cache requirements
  - Less transistor area for cache → More area for computing units
  - More computing units → 10,000s of SIMD threads and high throughput
  - GPUs win ☺

- Additionally:
  - Threads managed by hardware → You are not required to write code for each thread and manage them yourself
  - Easier to increase parallelism by adding more processors

- So, fundamental unit of a modern GPU is a *stream processor…*

G80 and GT200 Streaming Processor Architecture

Building a Programmable GPU

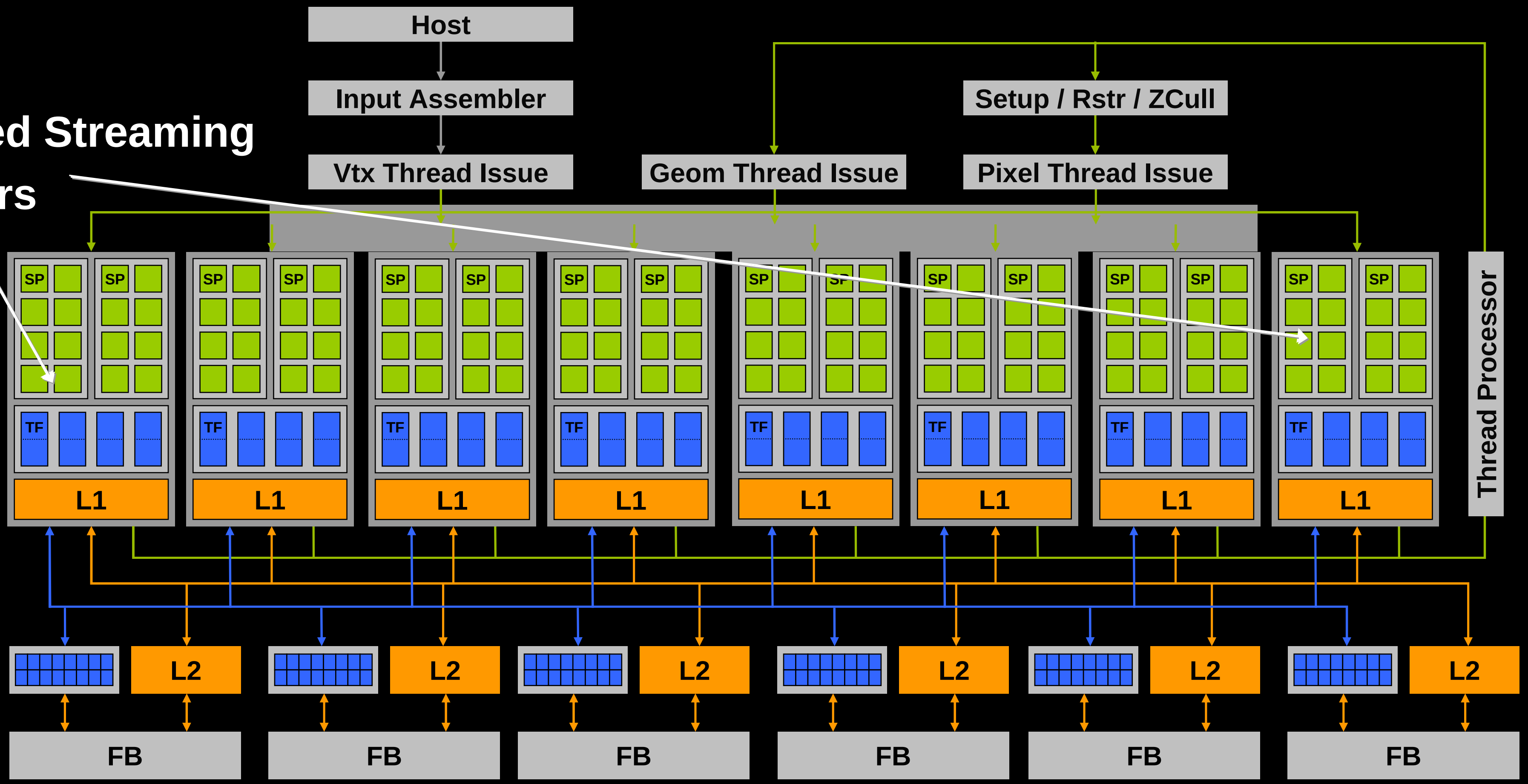
- The future of high throughput computing is programmable stream processing

- So build the architecture around the unified scalar stream processing cores

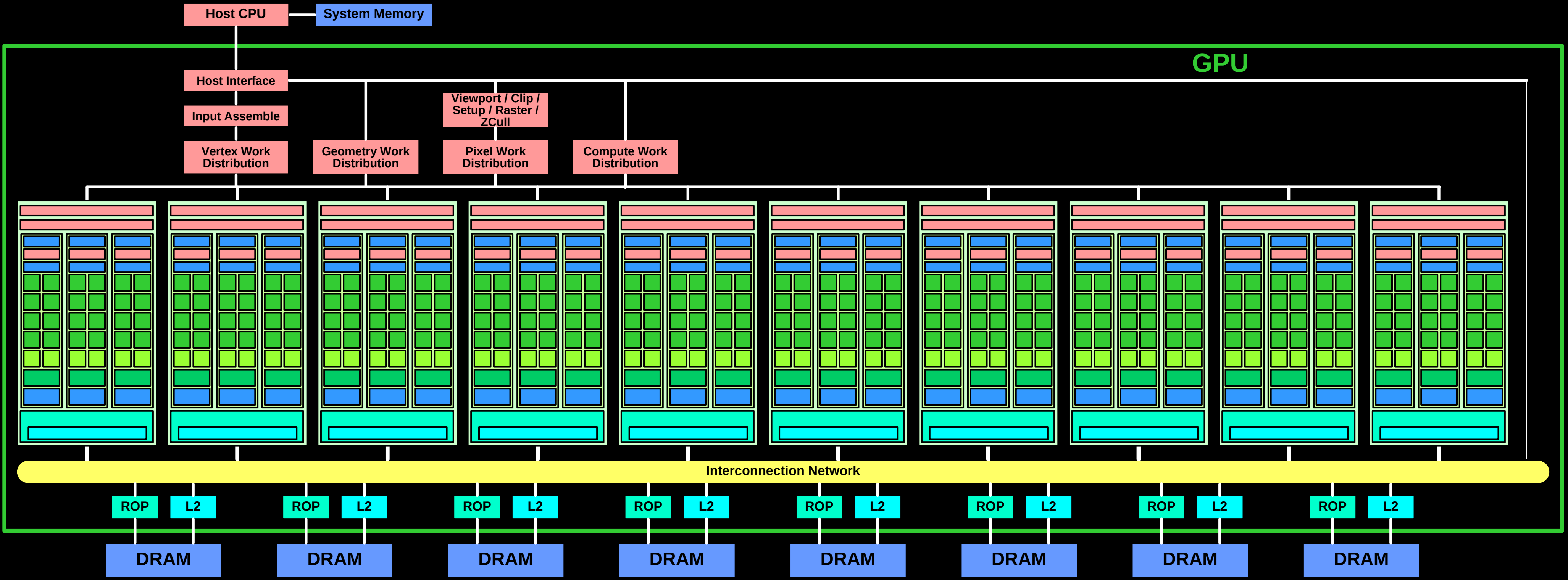
- GeForce 8800 GTX (G80) was the first GPU architecture built with this new paradigm

G80 Replaces The Pipeline Model

128 Unified Streaming Processors



GT200 Adds More Processing Power



8800GTX (high-end G80)

16 Stream Multiprocessors

- Each one contains 8 unified streaming processors – 128 in total

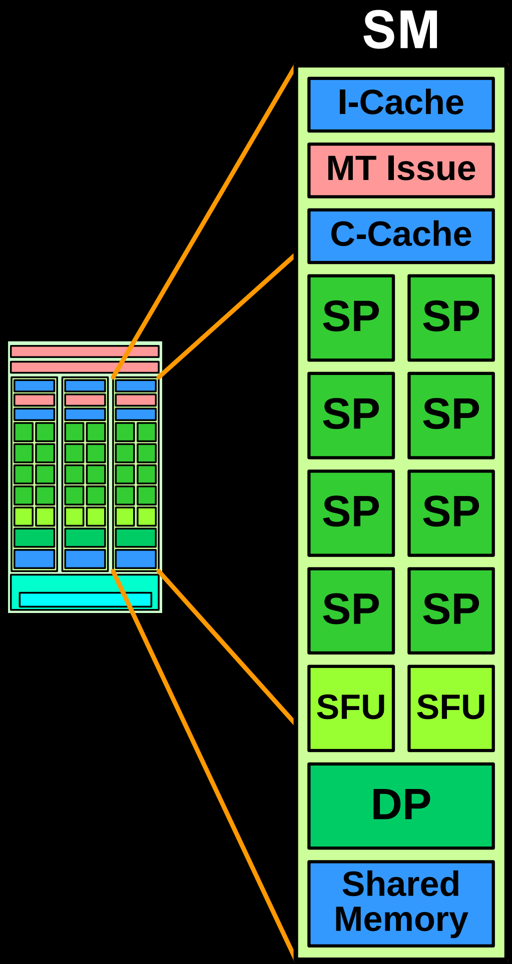
GTX280 (high-end GT200)

24 Stream Multiprocessors

- Each one contains 8 unified streaming processors – 240 in total

Inside a Stream Multiprocessor (SM)

- Scalar register-based ISA
- Multithreaded Instruction Unit
  - Up to 1024 concurrent threads
  - Hardware thread scheduling
  - In-order issue
- 8 SP: Thread Processors
  - IEEE 754 32-bit floating point
  - 32-bit and 64-bit integer
  - 16K 32-bit registers
- 2 SFU: Special Function Units
  - sin, cos, log, exp
- Double Precision Unit
  - IEEE 754 64-bit floating point
  - Fused multiply-add
- 16KB Shared Memory



Multiprocessor Programming Model

- Workloads are partitioned into blocks of threads among multiprocessors
  - a block runs to completion
  - a block doesn’t run until resources are available
- Allocation of hardware resources
  - shared memory is partitioned among blocks
  - registers are partitioned among threads
- Hardware thread scheduling
  - any thread not waiting for something can run
  - context switching is free – every cycle

Memory Hierarchy of G80 and GT200

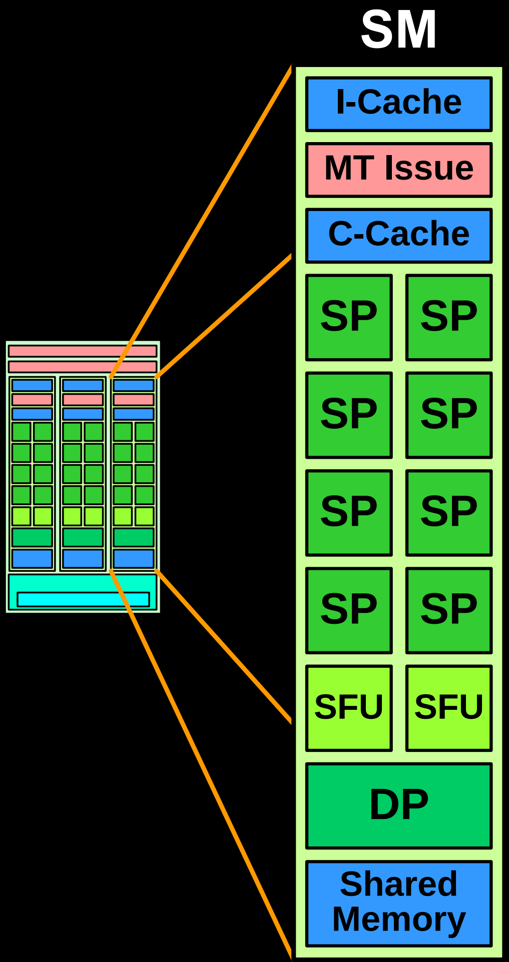
- SM can directly access device memory (video memory)
    - Not cached
    - Read & write
    - GT200: 140 GB/s peak
- SM can access device memory via texture unit
    - Cached
    - Read-only, for textures and constants
    - GT200: 48 GTexels/s peak
- On-chip shared memory shared among threads in an SM
    - important for communication amongst threads
    - provides low-latency temporary storage
    - G80 & GT200: 16KB per SM

Performance Per Millimeter

- For GPU, performance == throughput
  - Cache are limited in the memory hierarchy
- Strategy: hide latency with computation, not cache
  - Heavy multithreading
  - Switch to another group of threads when the current group is waiting for memory access
- Implication: need large number of threads to hide latency
  - Occupancy: typically 128 threads/SM minimum
  - Maximum 1024 threads/SM on GT200 (total 1024 * 24 = 24,576 threads)
- Strategy: Single Instruction Multiple Thread (SIMT)

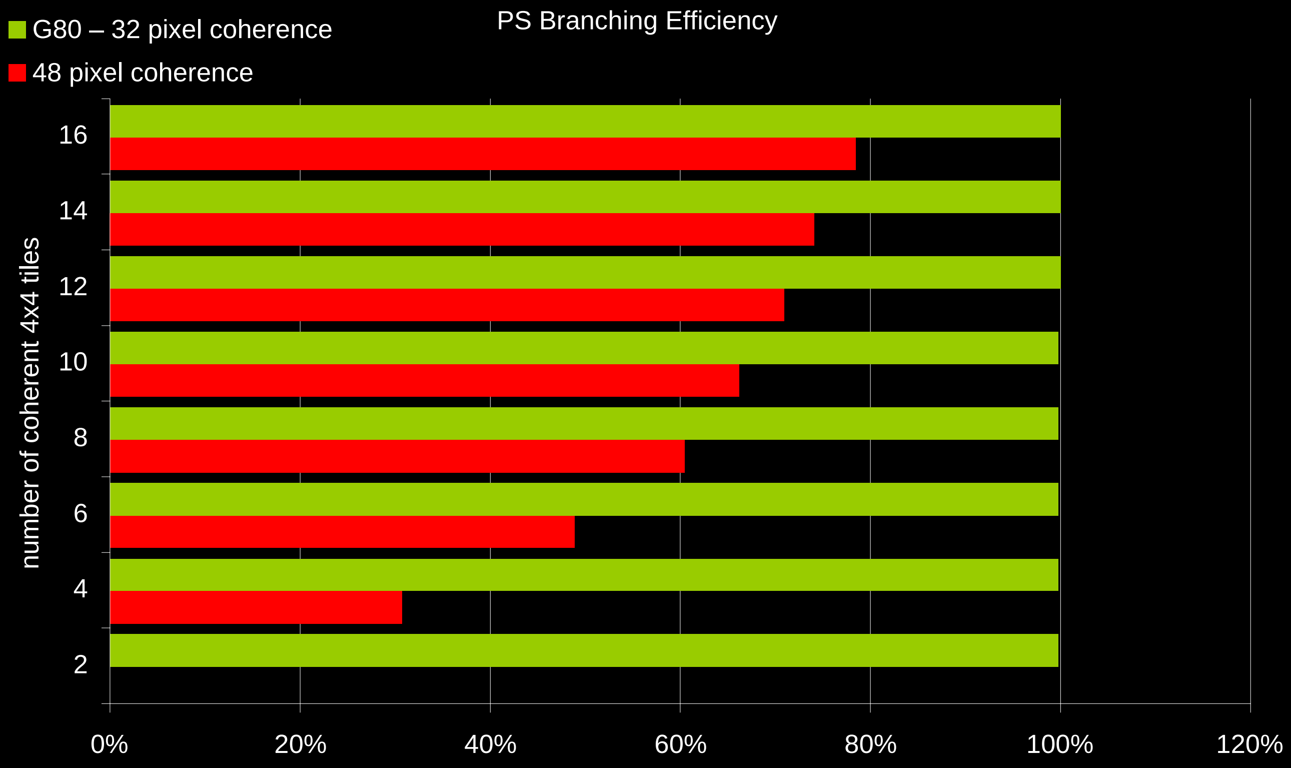
SIMT Thread Execution

- Group 32 threads (vertices, pixels or primitives) into warps
  - Threads in warp execute same instruction at a time
  - Shared instruction fetch/dispatch
  - Hardware automatically handles divergence (branches)
- Warps are the primitive unit of scheduling
  - Pick 1 of 24 warps for each instruction slot
- SIMT execution is an implementation choice
  - Shared control logic leaves more space for ALUs
  - Largely invisible to programmer



Shader Branching Performance

- **G8x/G9x/GT200 branch efficiency is 32 threads (1 warp)**
- **If threads diverge, both sides of branch will execute on all 32**
- **More efficient compared to architecture with branch efficiency of 48 threads**

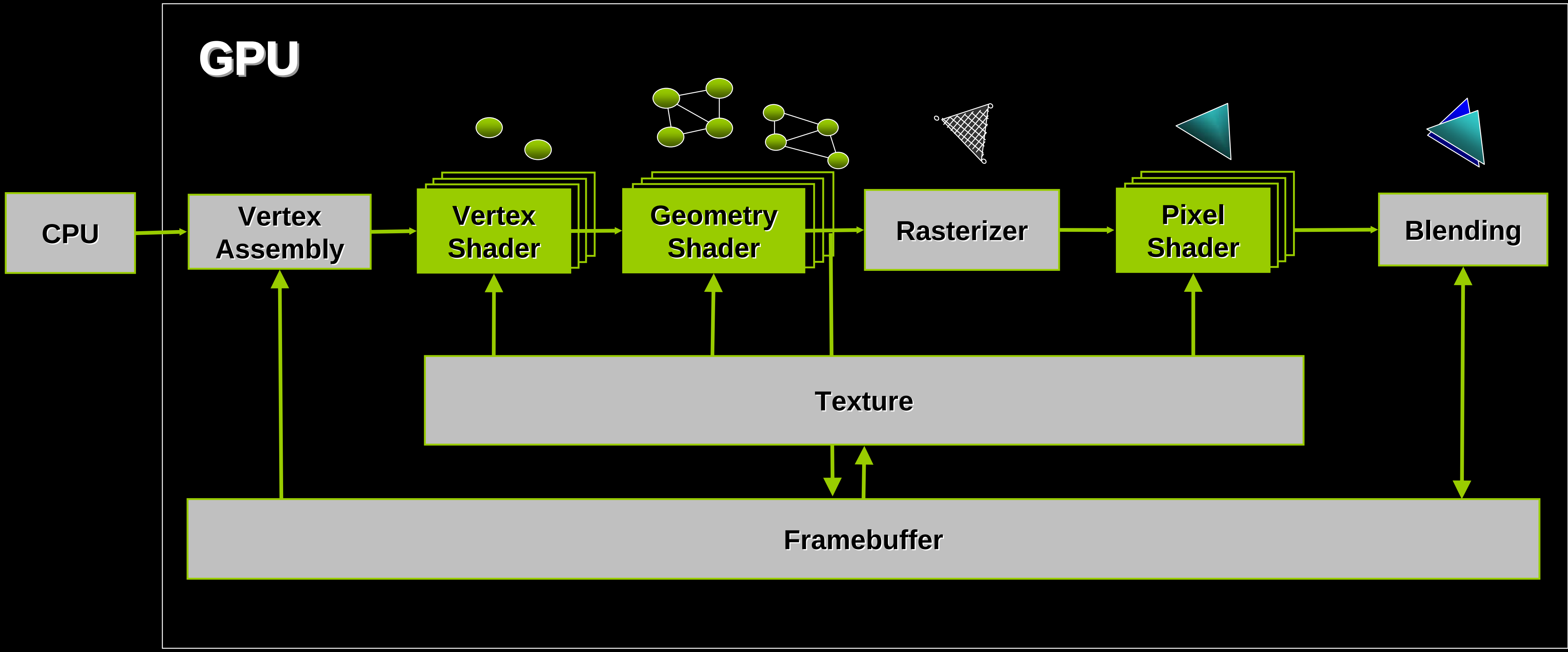


Conclusion:
G80 and GT200 Streaming Processor Architecture

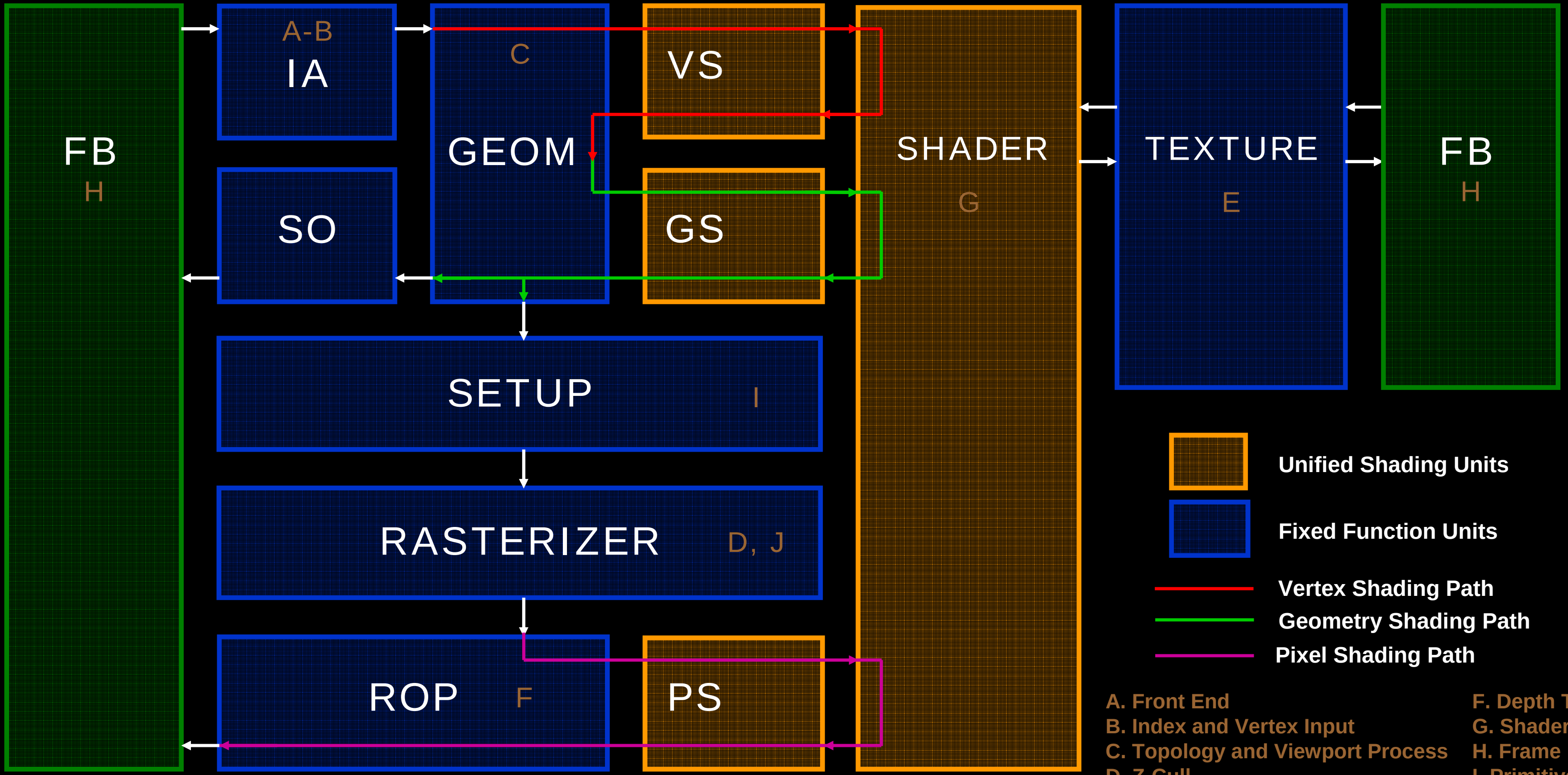
- Execute in blocks can maximally exploits data parallelism
  - Minimize incoherent memory access
  - Adding more ALU yields better performance
- Performs data processing in SIMT fashion
  - Group 32 threads into warps
  - Threads in warp execute same instruction at a time
- Thread scheduling is automatically handled by hardware
  - Context switching is free (every cycle)
  - Transparent scalability. Easy for programming
- Memory latency is covered by large number of in-flight threads
  - Cache is mainly used for read-only memory access (texture, constants).

Stream Processing for Graphics

DX10 Graphics Pipeline (Logical View)



DX10 API Mapped To GPU

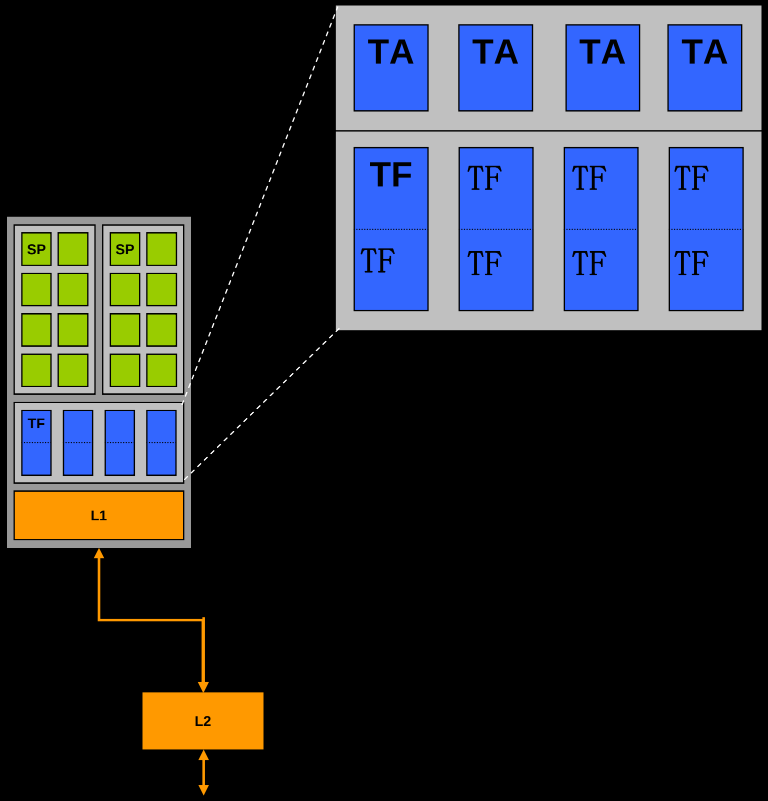


A. Front End
B. Index and Vertex Input
C. Topology and Viewport Process
D. Z-Cull
E. Texture Fetch

F. Depth Test, Alpha-Blending, AA
G. Shader Cores
H. Frame Buffer Controller
I. Primitive Setup
J. Rasterization

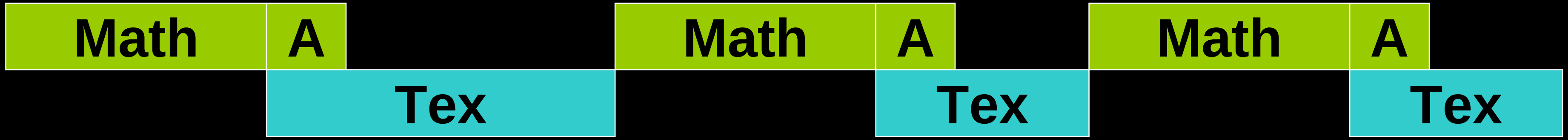
Texturing

- G80
  - Every 2 SMs connect to a TMU (Texture Mapping Unit)
  - 8 TMUs in total
- GT200
  - Every 3 SMs connect to a TMU
  - 10 TMUs in total
- Each TMU contains 4 TA (Texture Addressing) and 8 TF (Texture Filtering)
  - TA:TF = 1:2, allowing full-speed FP16 bilinear filtering and free 2xAF



Decoupled shader & texture pipelines

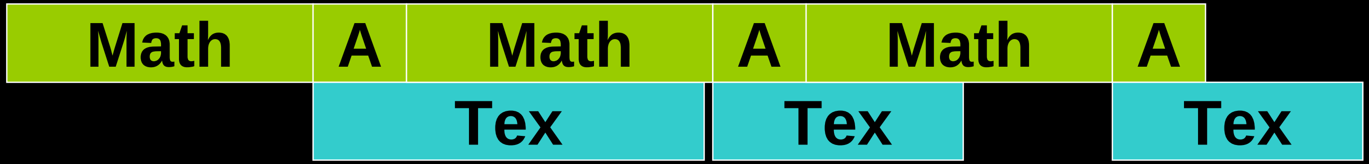
- G7x Architecture



Math A Math A Math A

Tex Tex Tex

- G8x and later Architecture



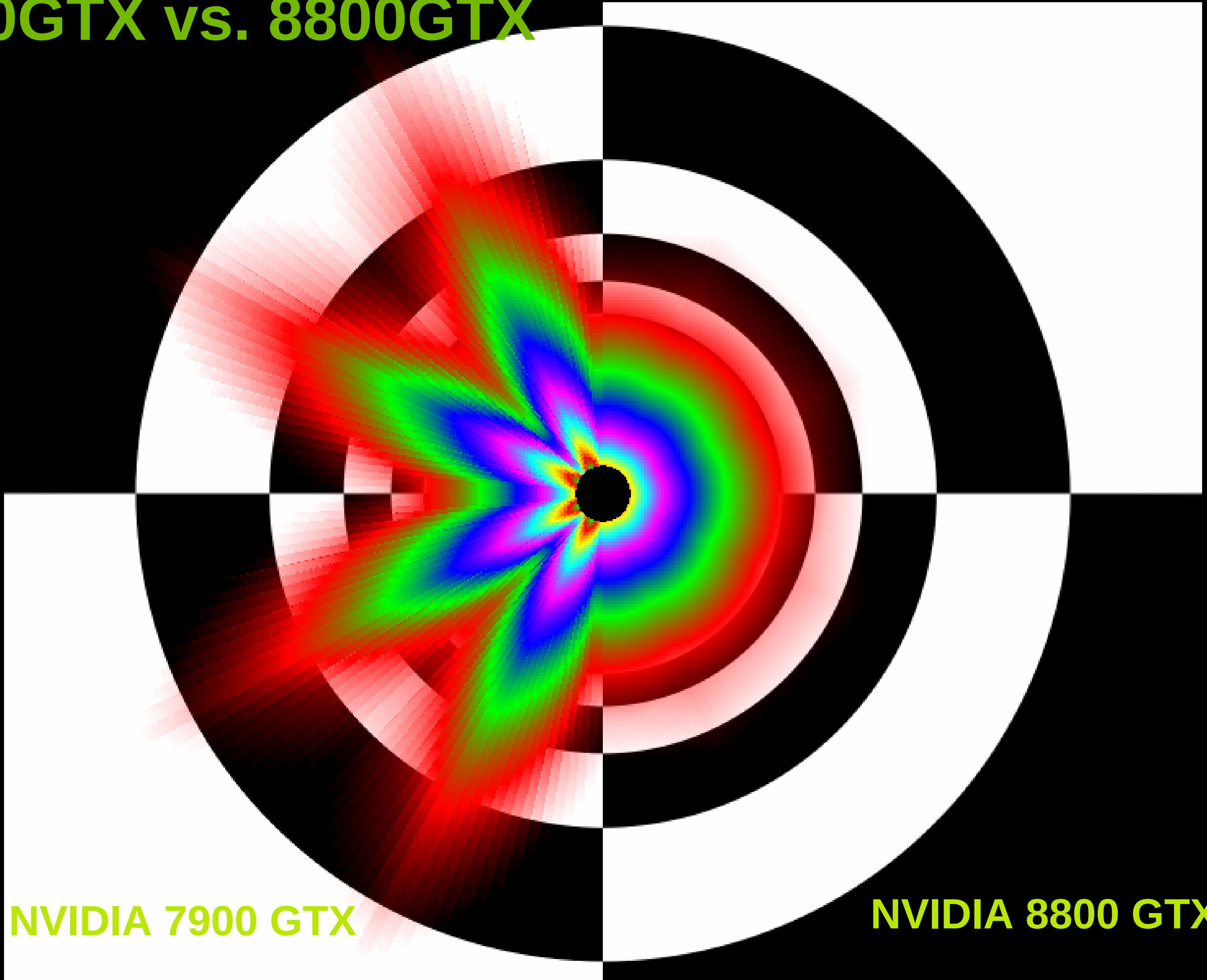
Math A Math A Math A

Tex Tex Tex

- Cover texture latency with thousands of threads in flight

Texture Filtering Quality

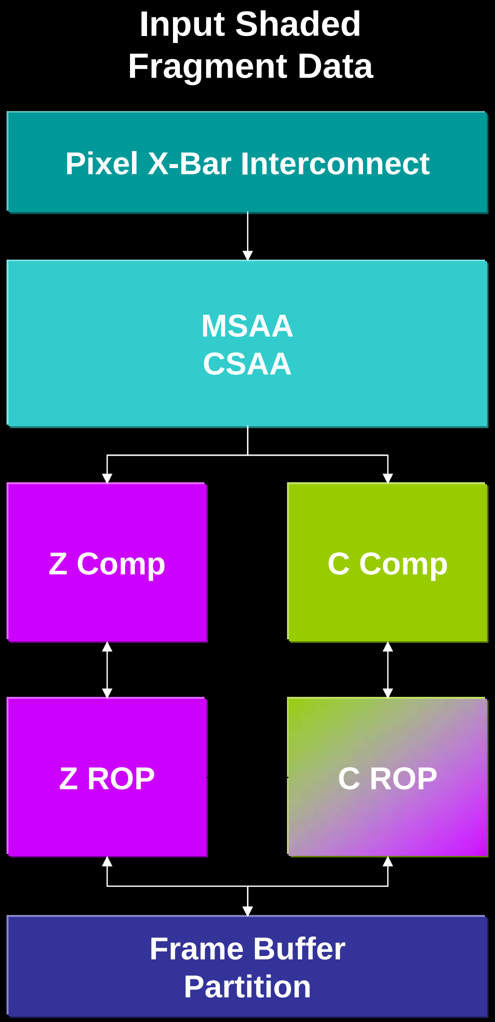
16xAF, 7900GTX vs. 8800GTX



NVIDIA 7900 GTX

NVIDIA 8800 GTX and after

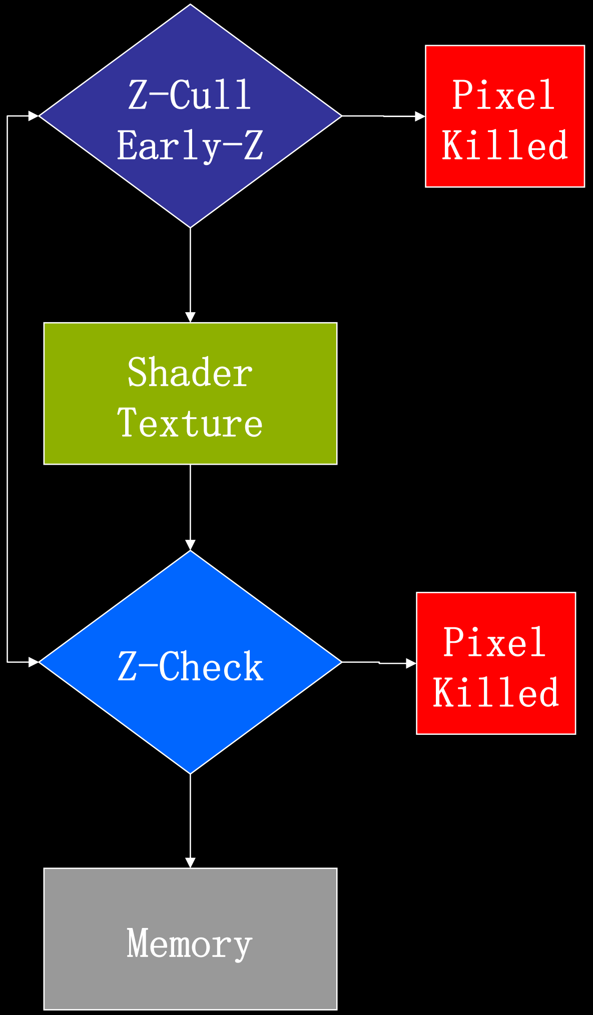
Detail of a single ROP pixel pipeline



- INT8, FP16 & FP32 formats Support
- Up to 16x Multisampling
- AA with HDR
- Color & Z Compression (2x)
- Multiple Render Targets (8)

ROP Unit in GPU: Early Z

- Z-Buffer removes pixels not visible
- Z-Culling removes regions of non-visible pixels @ high rate
- Early-Z removes individual pixel quads (2x2) before they are processed



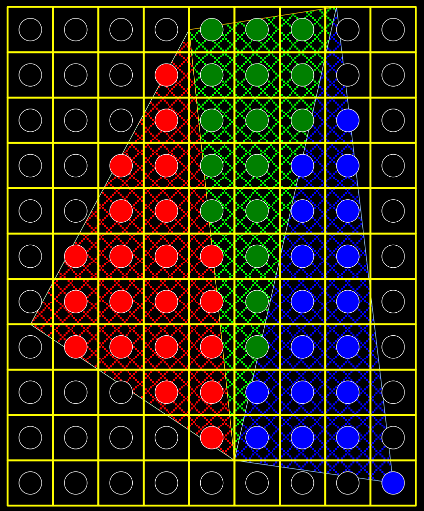
Next Generation Antialiasing

- High Quality AA with incredible performance
- Highest Quality AA
- AA with HDR
- Coverage Sampling AA (CSAA)

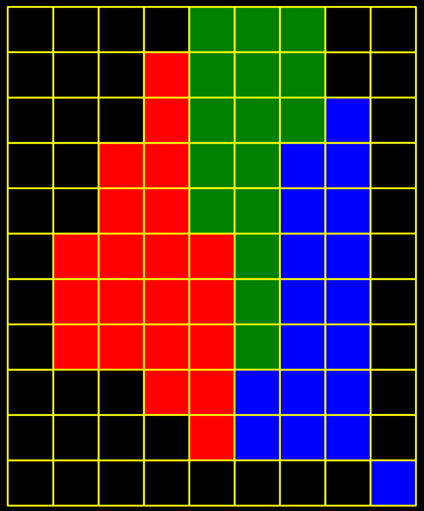
3DMark05 – Futuremark

Review: Aliased Rendering

- Only one sample per pixel
  - Coverage, Color, Depth and Stencil
- Precisely evaluated, but under-sampled



Triangle Geometry



Aliased

Anti-Aliasing Fundamentals

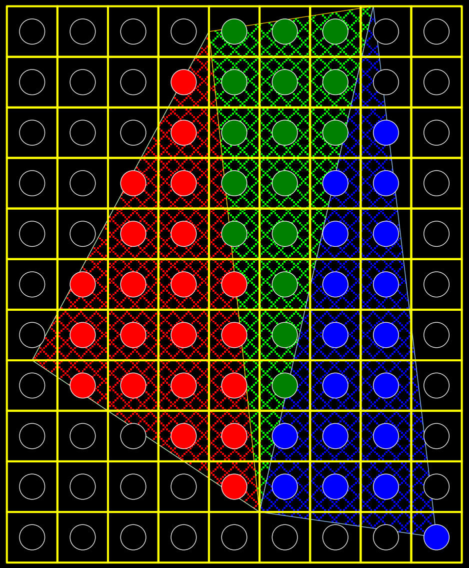
- ***Pixels in an image need to represent finite areas from the scene description, not just infinitely small sample points***

- **Ideal anti-aliasing would be a smooth convolution over the pixel area**
  - **Too expensive for general scenes**
- **So we approximate by taking many samples**

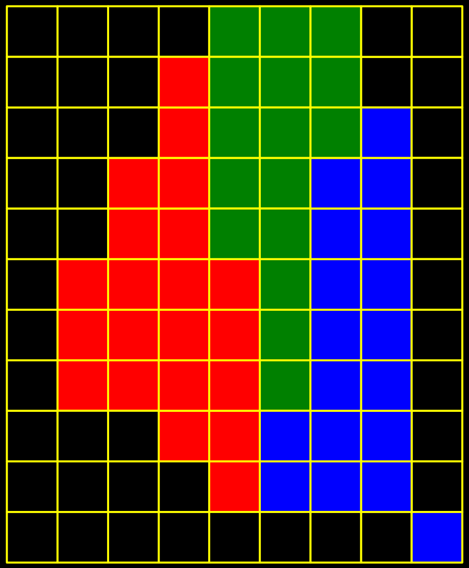
- **The more samples the better**

Anti-Aliasing

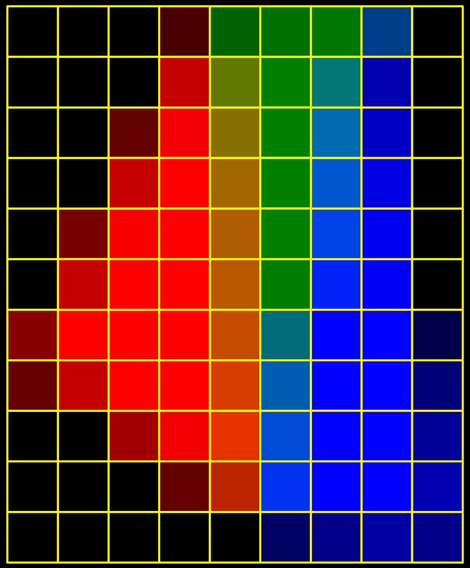
- Anti-aliasing properly accounts for the contribution of all the primitives that intersect a pixel



Triangle Geometry



Aliased



Anti-Aliased

Supersampling

- Store a color and depth value for each sub-sample at every pixel. Calculate unique color and depth values at every sub-sample for every pixel of every drawn triangle
- Strengths
    - Robust image quality by brute force
- Weaknesses
    - Inefficient: Expensive pixel shaders and texture fetches are executed for every sub-sample; wasteful because color results within a pixel are nearly identical

Multisampling (SGI, 1993)

- Store a unique color and depth value for each sub-sample for each pixel, but re-use one calculated color for all color sub-samples for a polygon
- Strengths
  - Only one color value calculated per pixel per triangle
  - Z and stencil evaluated precisely; interpenetrations and bulkheads handled correctly
- Weaknesses
  - Memory footprint N times larger than 1x
  - Expensive to extend to 8x quality and beyond

Motivation for CSAA

- Multisampling evolved from 1 → 2 → 4 samples
- Beyond 4 sub-samples, storage cost increases faster than the image quality improves
- Even more true with HDR
  - 64b and 128b per color sub-sample!

- For the vast majority of edge pixels, 2 colors are enough
  - What matters is more detailed coverage information

Coverage Sampled Antialiasing

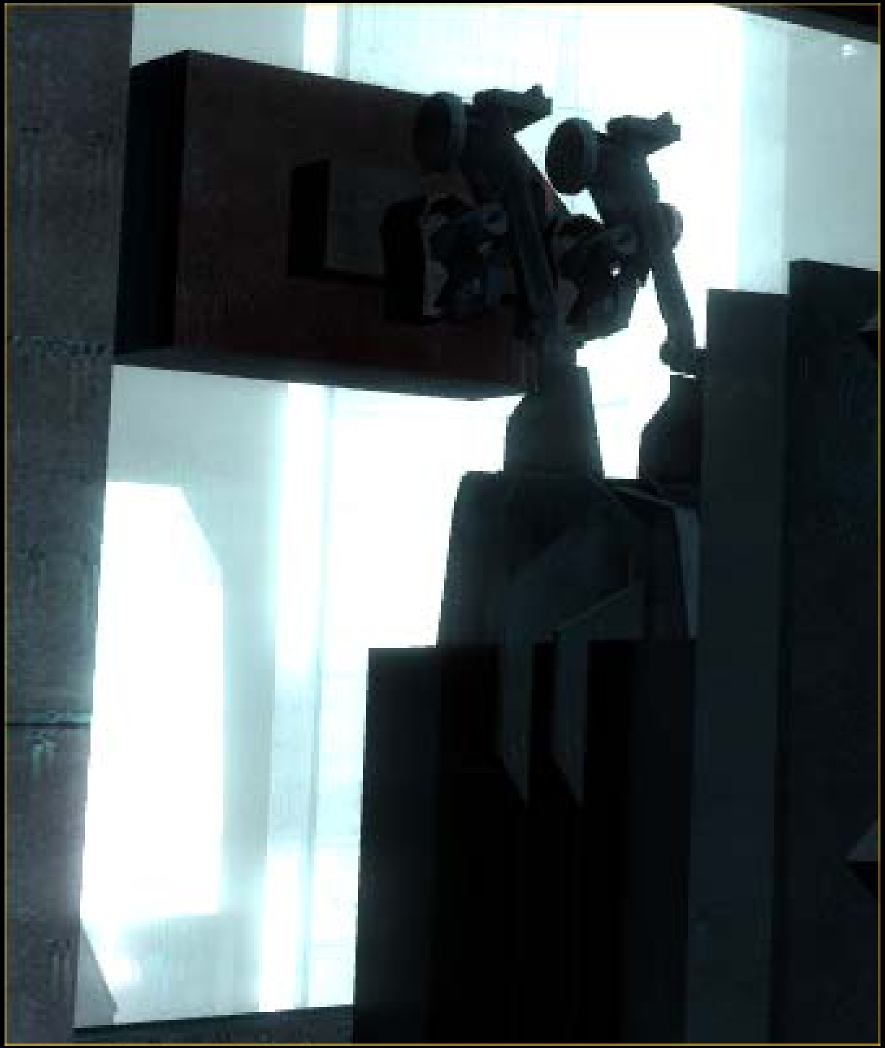
- Compute and store boolean coverage at 16 sub-samples
- Compress the redundant color and depth/stencil information into the memory footprint and bandwidth of 4 or 8 multisamples

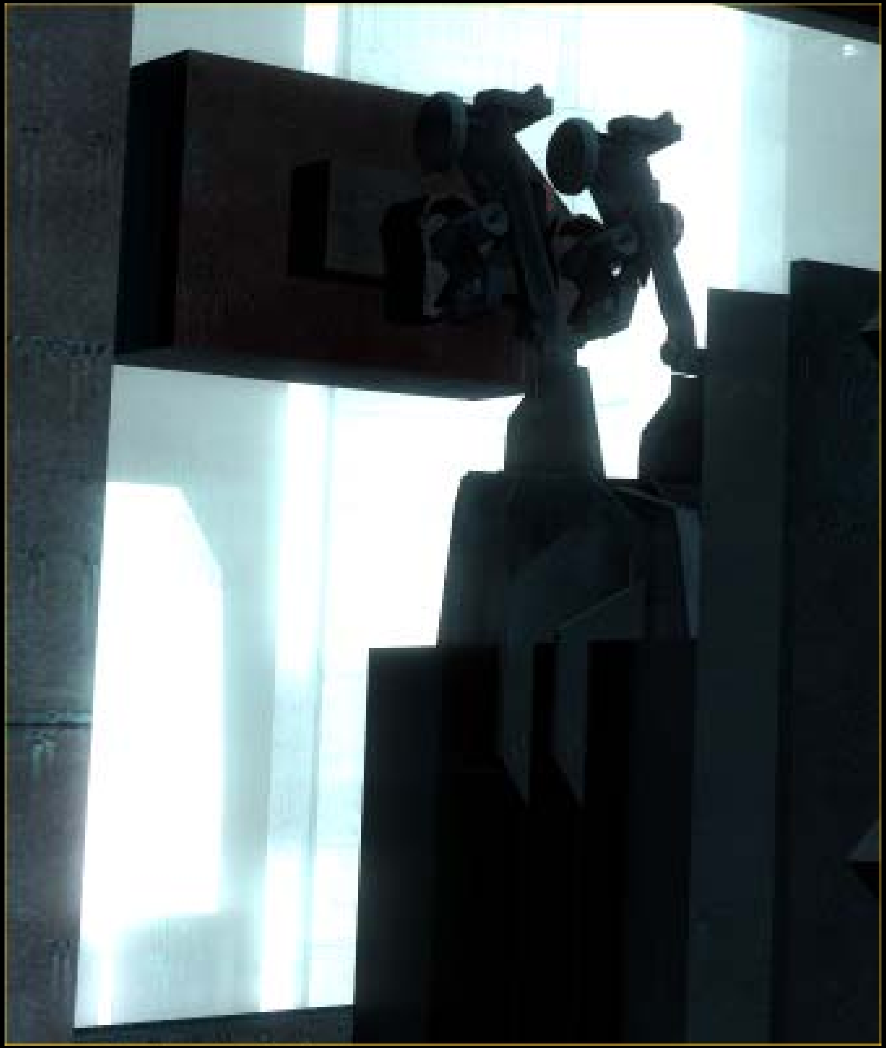
- Performance of 4xMSAA with 16x quality

CSAA Quality Levels

Quality Level:	4x	8x	8xQ	16x	16xQ
Texture/Shader Samples	1	1	1	1	1
Stored Color/Z Samples	4	4	8	4	8
Coverage Samples	4	8	8	16	16

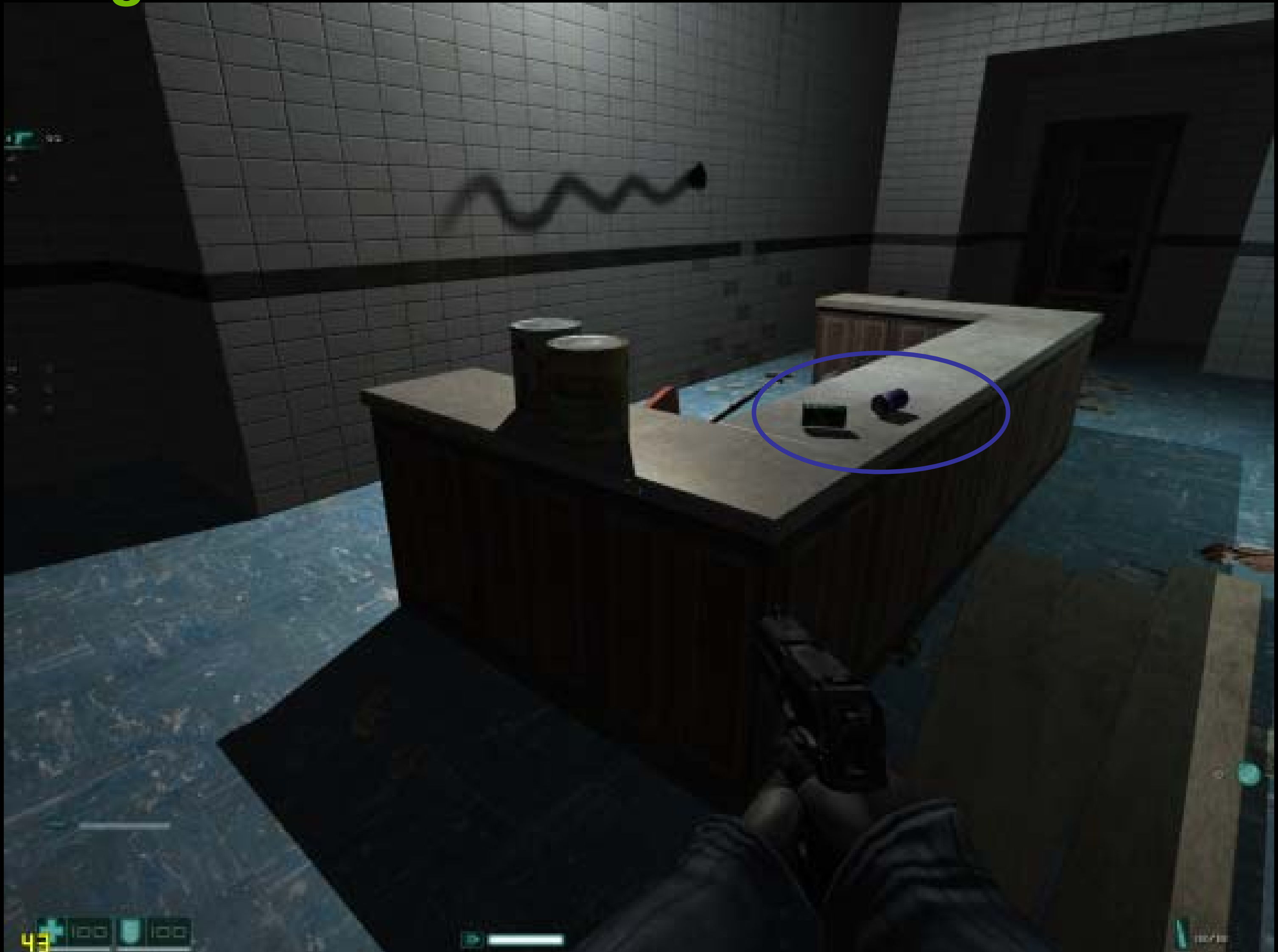
Half Life 2





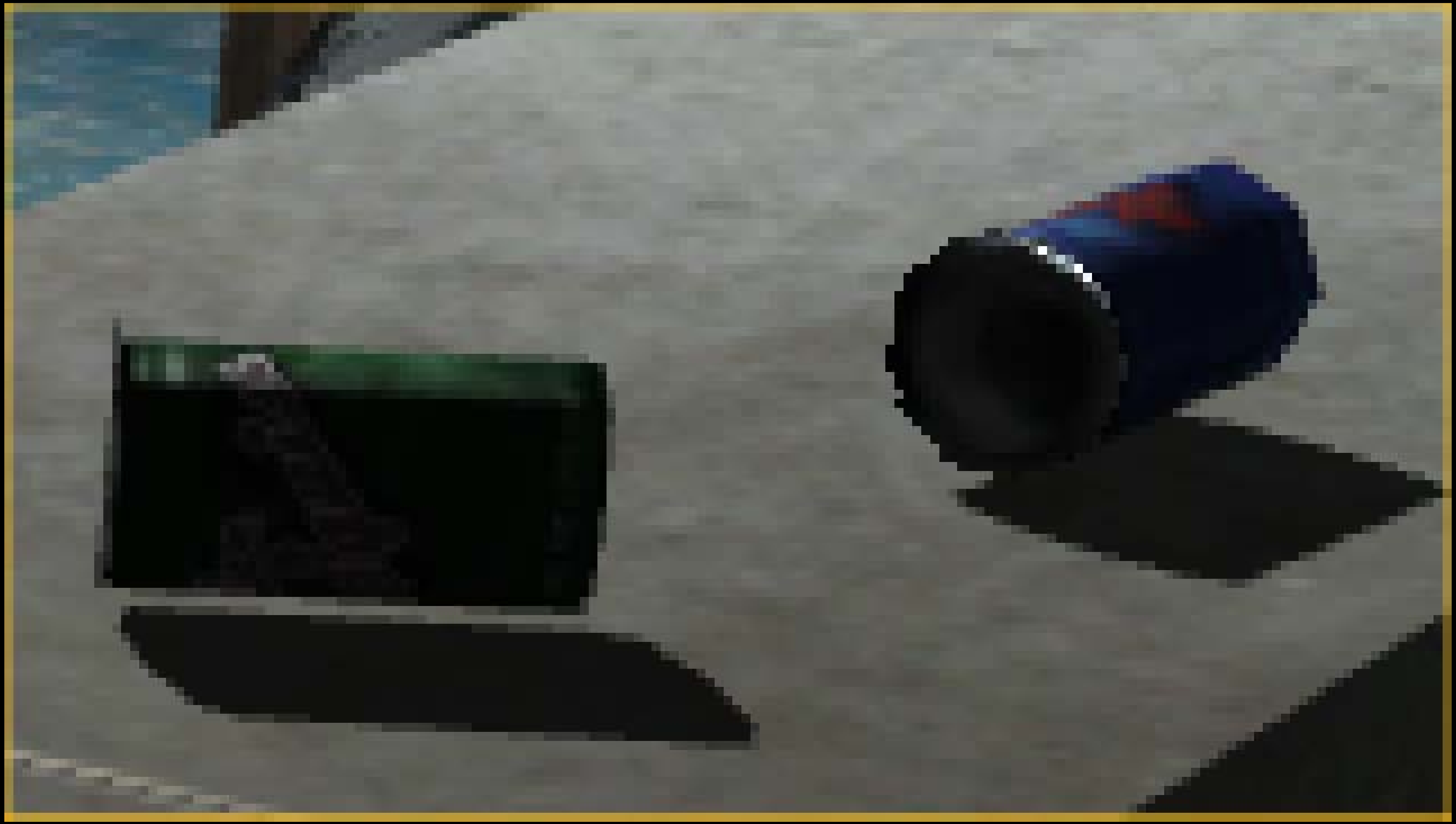
Half Life 2 – Valve Software

Shadow Edges in FEAR



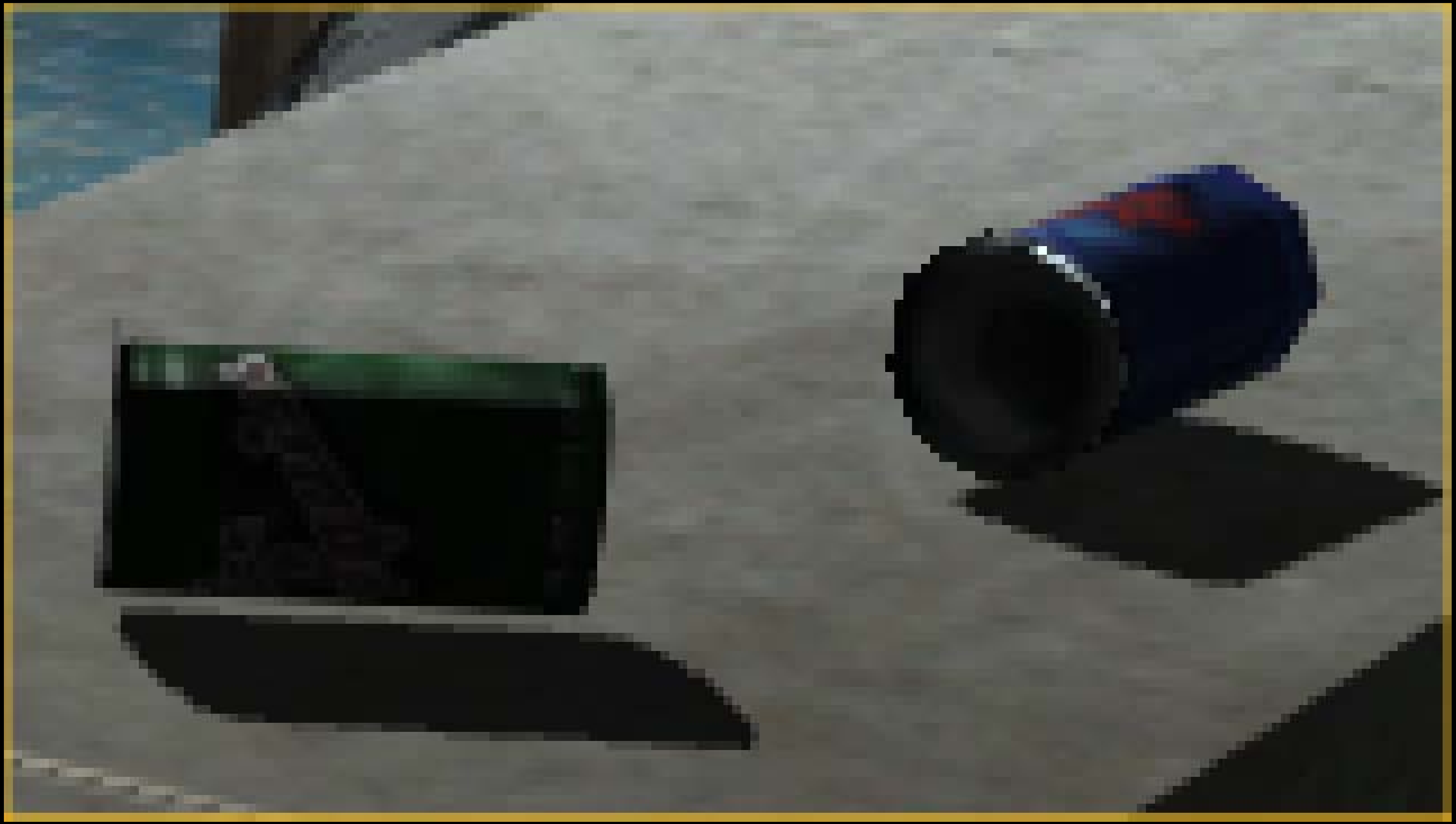
FEAR - Monolith

FEAR 4x



FEAR - Monolith

FEAR 16x



Higher Quality Object Edges

FEAR - Monolith

Summary of CSAA Advantages

- Only need to traverse the scene once
- Small, fixed video memory footprint
- Handles inter-penetrating objects
- Pixel shaders are only run where color detail is needed
- 16x stored positional coverage

- **16x quality for ~4x performance**

GPU beyond Graphics

CUDA

A scalable parallel programming model and software environment for parallel computing

Minimal extensions to familiar C/C++ environment

Heterogeneous serial-parallel programming model

The Democratization of Parallel Computing

- GPUs and CUDA bring parallel computing to the masses
  - Over 100M CUDA-capable GPUs sold to date
  - 60K CUDA developers
  - A “developer kit” (i.e. GPU) costs ~$200 (for 500 GFLOPS)
- Data-parallel supercomputers are everywhere!
  - CUDA makes this power accessible
  - We’re already seeing innovations in data-parallel computing

Massively parallel computing has become a commodity technology!

More about CUDA at Seminar

“Parallel Computing with CUDA”

Questions?