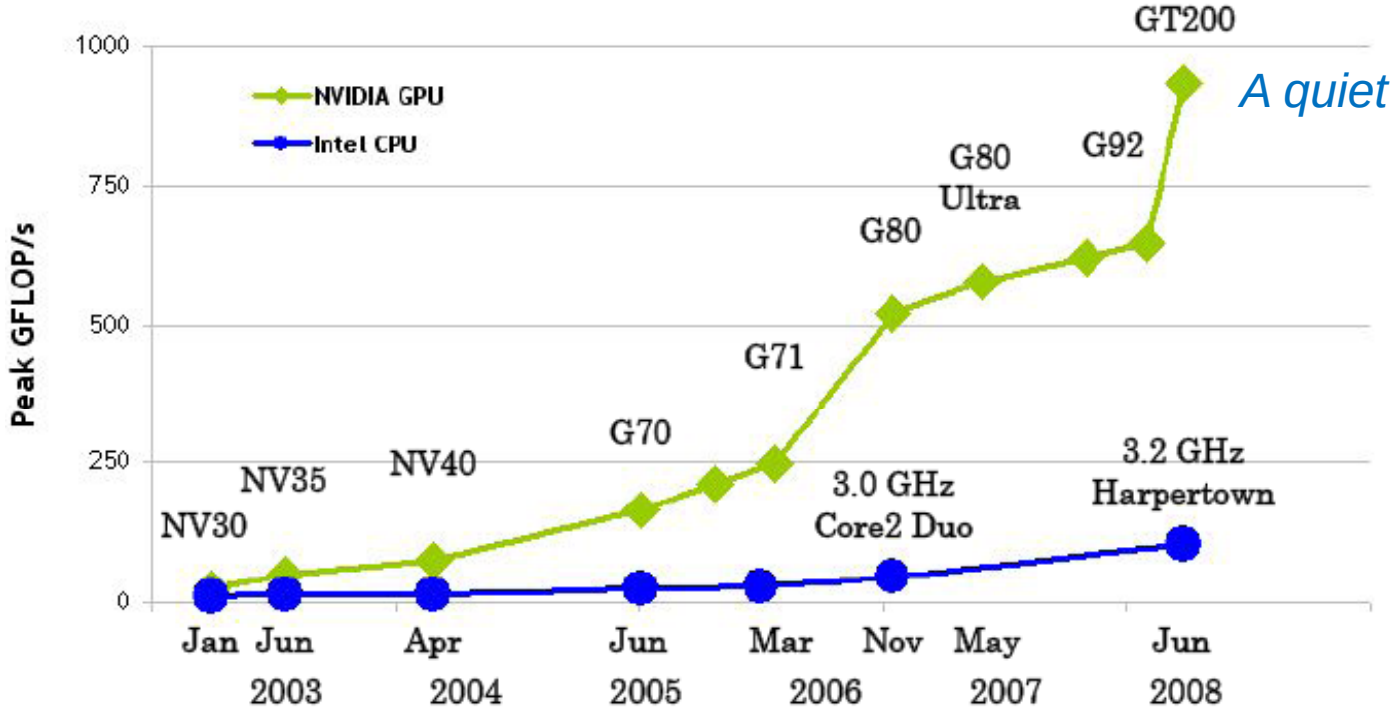
GPGPU Computing

Yong Cao

Why Graphics Card?

- It's powerful!



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

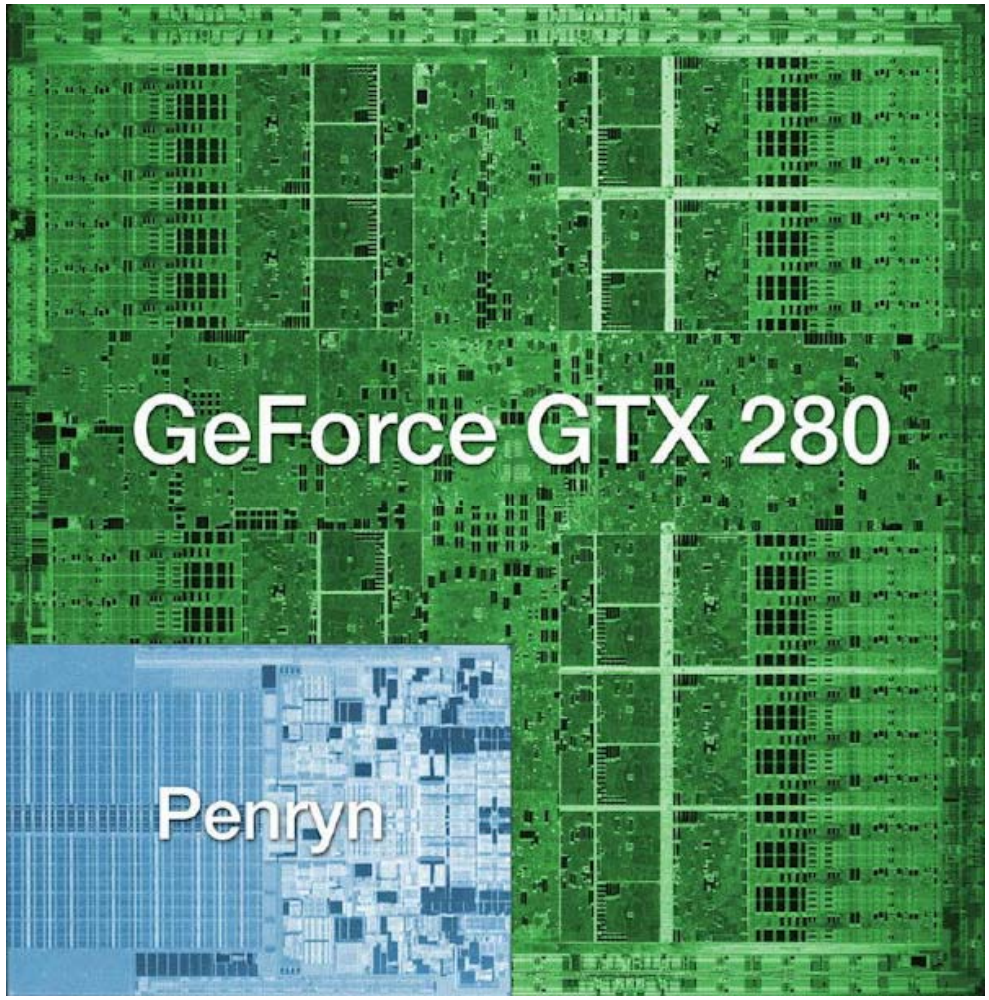
Why Graphics Card?

- It's powerful!

Processor	Processing Units	FLOPs per Unit	Clock Speed	Processing Power
High End Qual-Core CPU	16 (4 per core)	2	3000 MHz	96 GFLOPs
NVIDIA GTX285	240	3	1476 MHz	1063 GFLOPs
ATI Radeon HD 4870	800	2	750 MHz	1200 GFLOPs

Why Graphics Card?

- It's powerful!
  - Die size 576 mm^2
  - 1.4 Billion Transistors
  - Memory width: 512bit
  - Bandwidth 142GB/s



Max board Power:236 W
GTX285: 183 W (55nm)

Why Graphics Card?

- It's cheap and everywhere.
  - E.g. NIVIDA sold 100 Million high-end GPGPU devices.
  - A 128-core Geforce 9800GTX (648 GFLOPs) is $129 on Newegg.com.

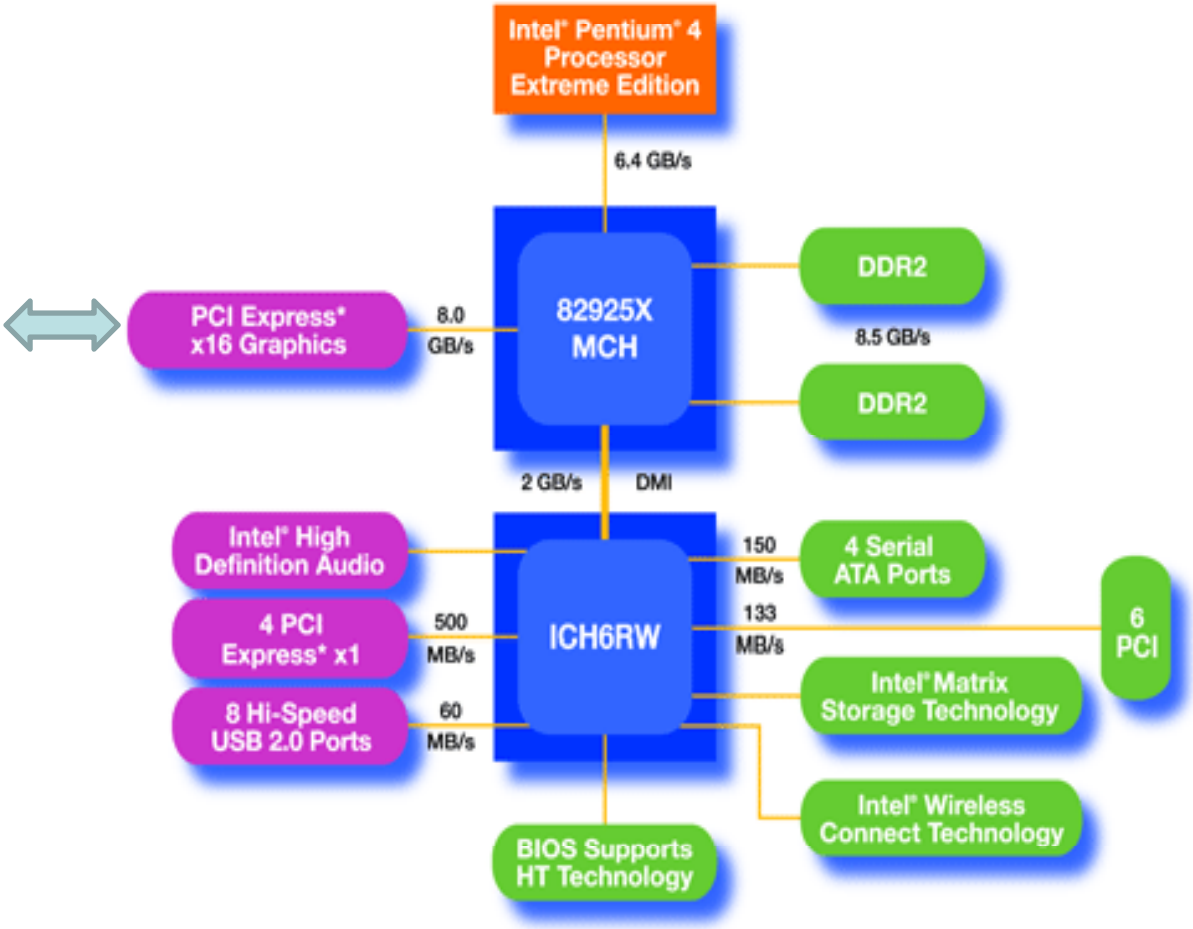
Why Graphics Card NOW?

- **Before:**
  - Two years ago, everyone is using Graphics API (Cg, GLSL, HLSL) for GPGPU programming.
  - Restrict random-read (using Texture), NOT be able to random-write. (No pointer!)

Why Graphics Card NOW?

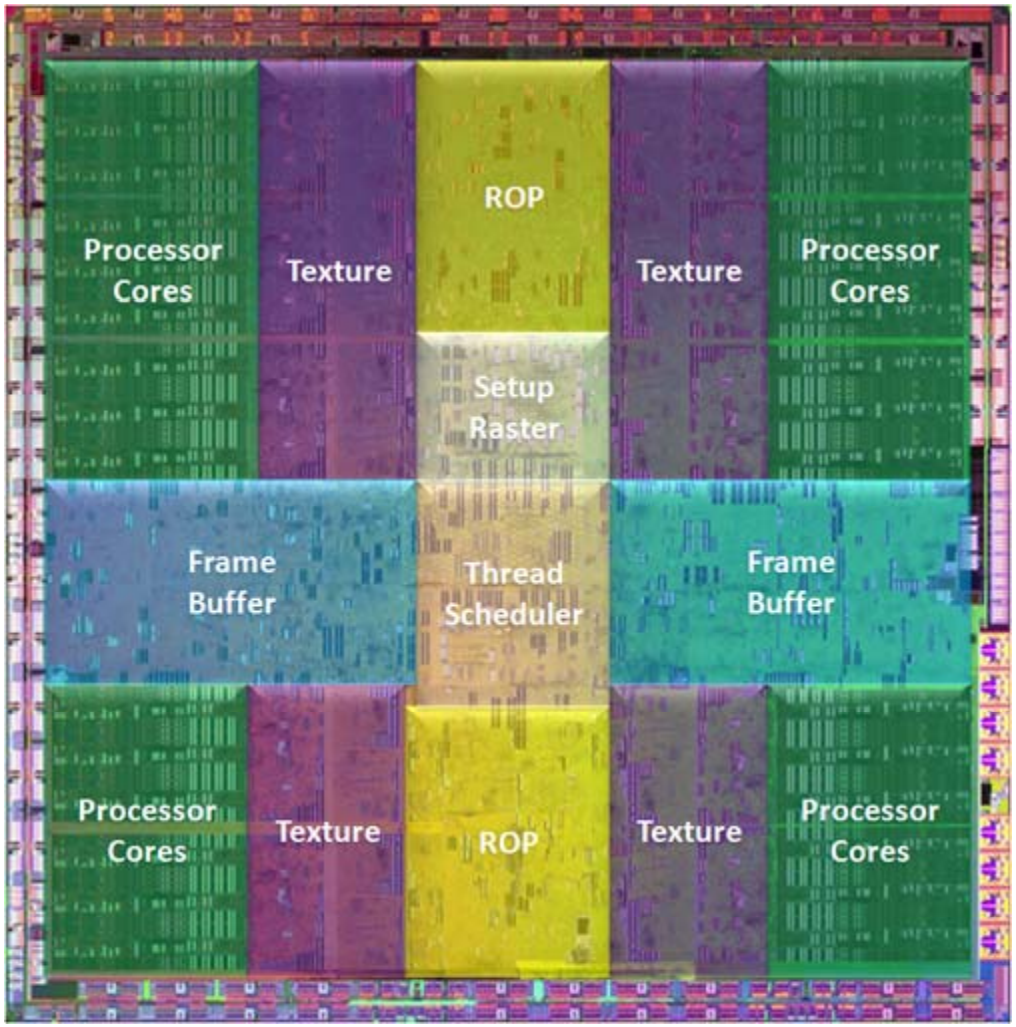
- Now:
  - NIVIDA released CUDA two years ago, since then
    - Thousands of CUDA software engineers
    - New job title “CUDA programmer”
    - Around 200 CUDA based technical publications!
  - Why?
    - Standard C language
    - Support Pointer! Random read and write on GPU memory.
    - Work with C++, Fortran

Where's GPU in the system



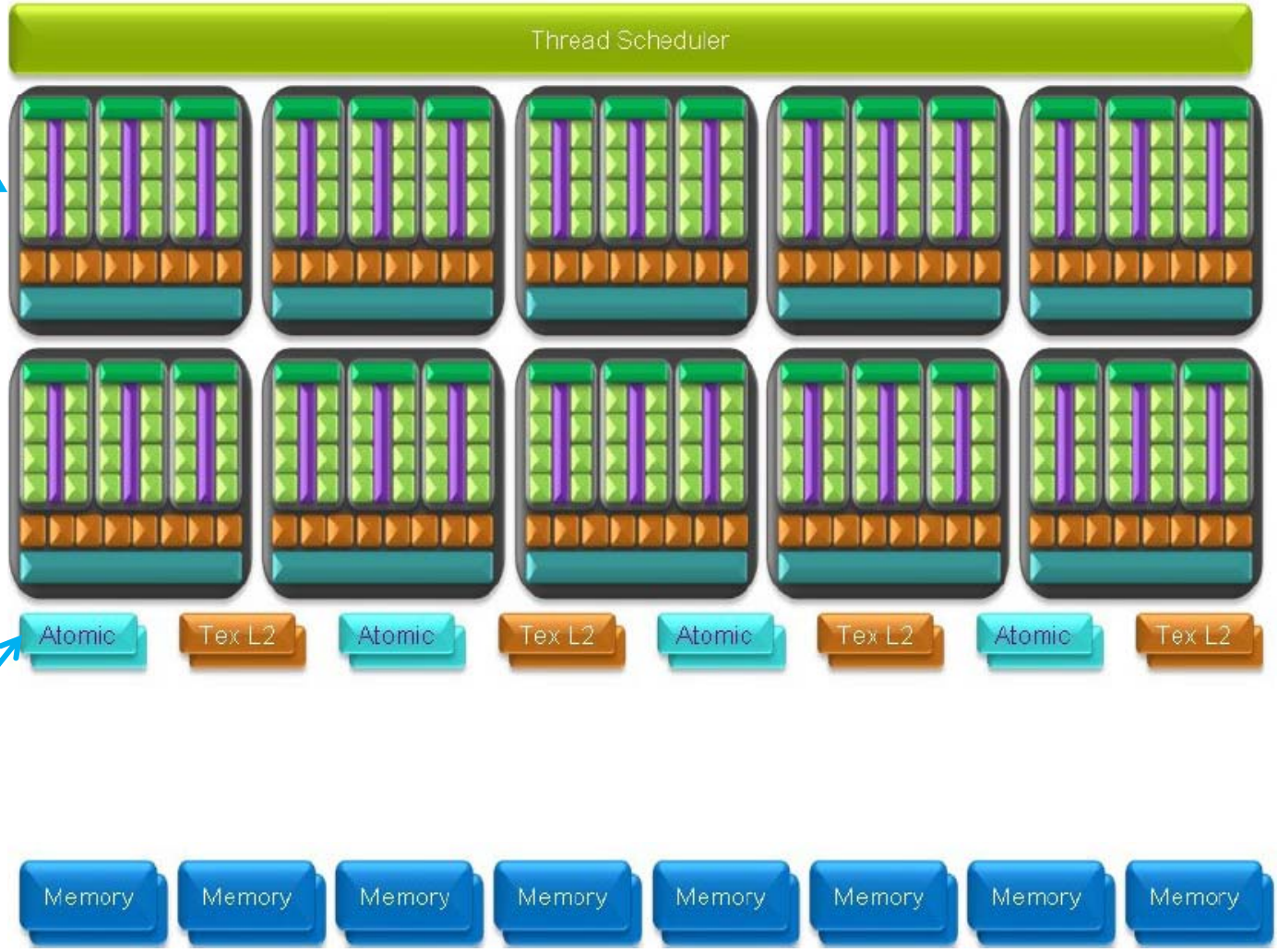
NVIDIA GPU Architecture

- Lots of ALUs
- Lots of Control
- Focus on Graphics Applicants (Data parallel)



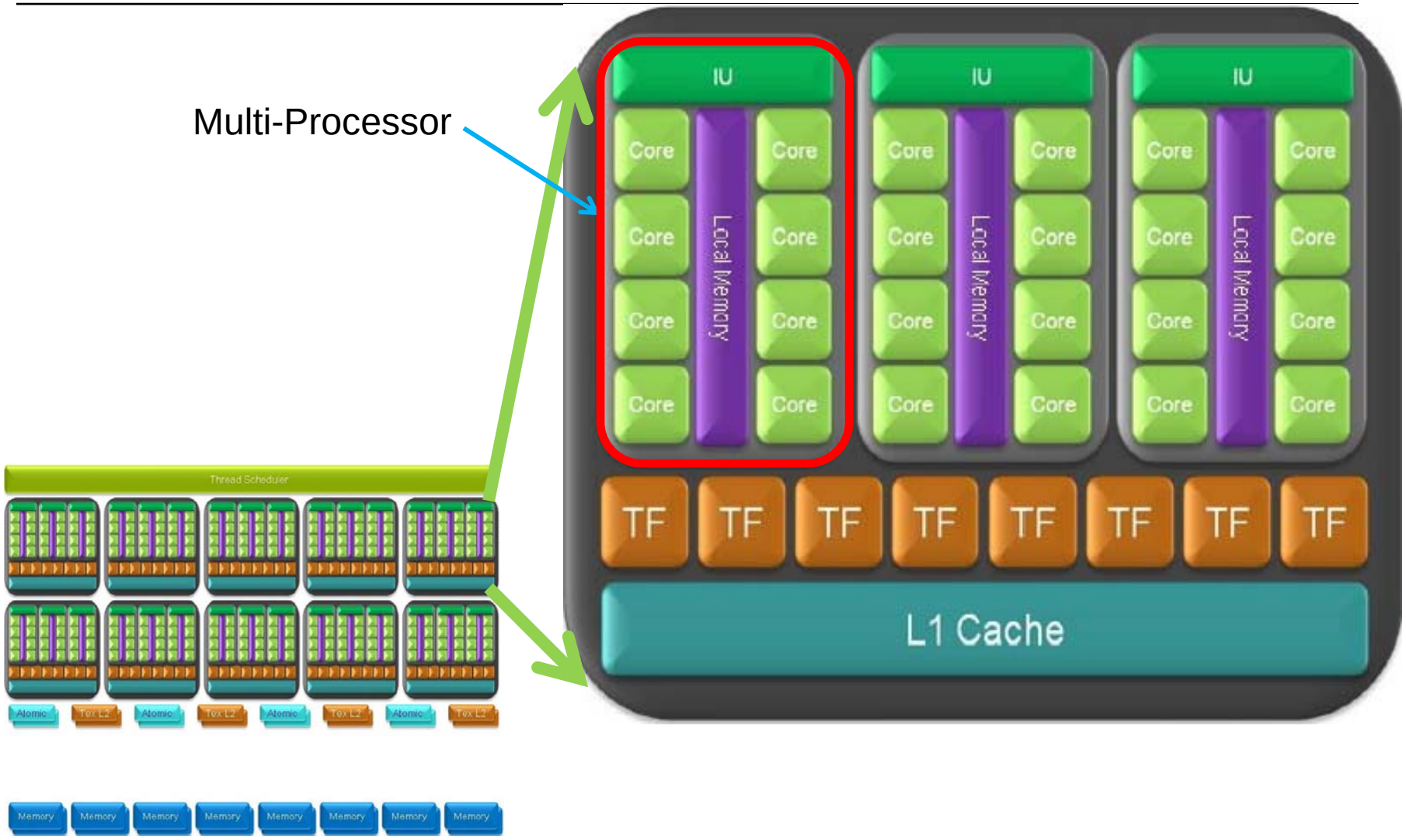
NVIDIA GT200 Architecture

Thread Processing Cluster

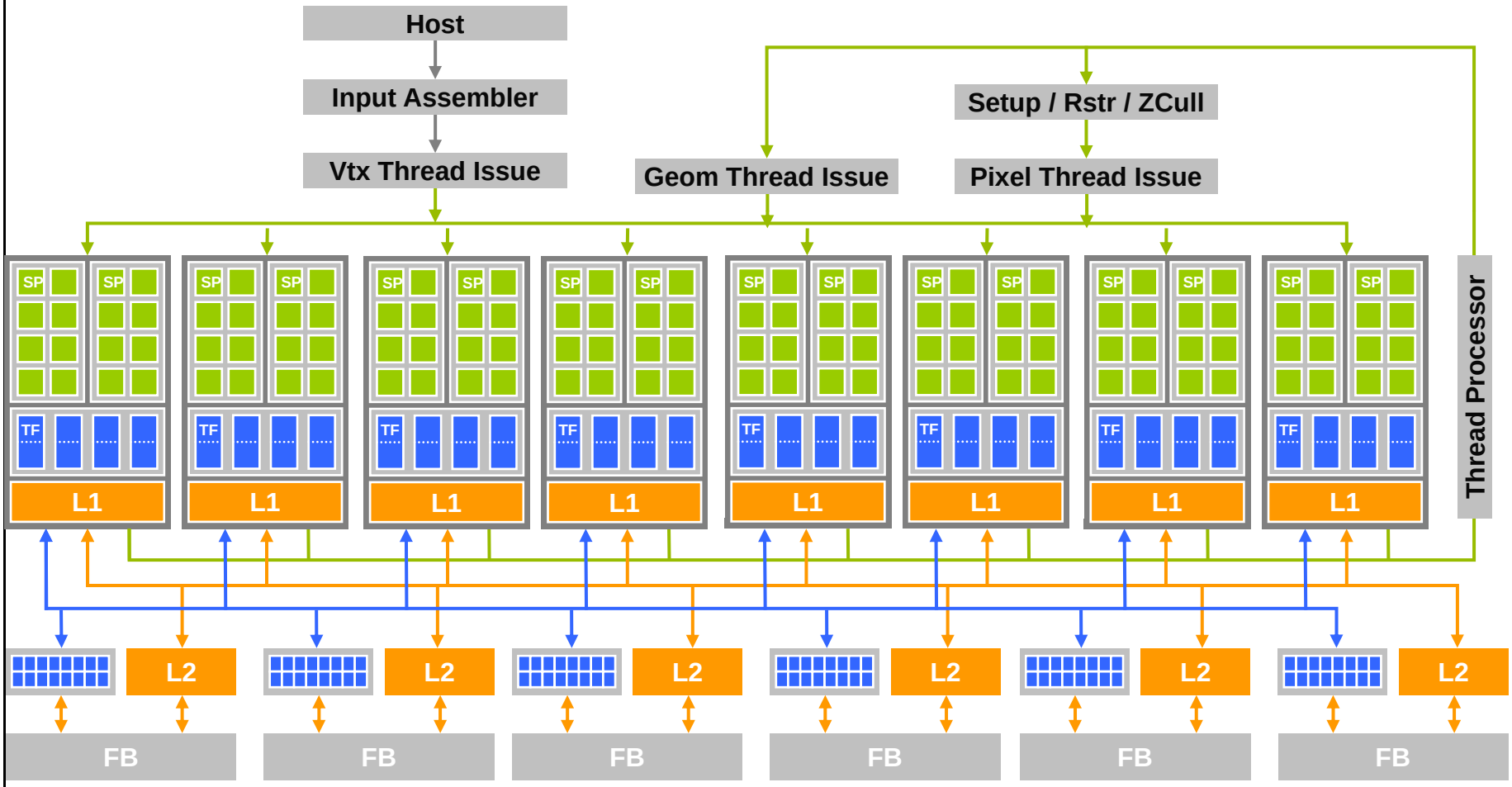


Atomic Memory Access Control

NVIDIA GT200 Architecture

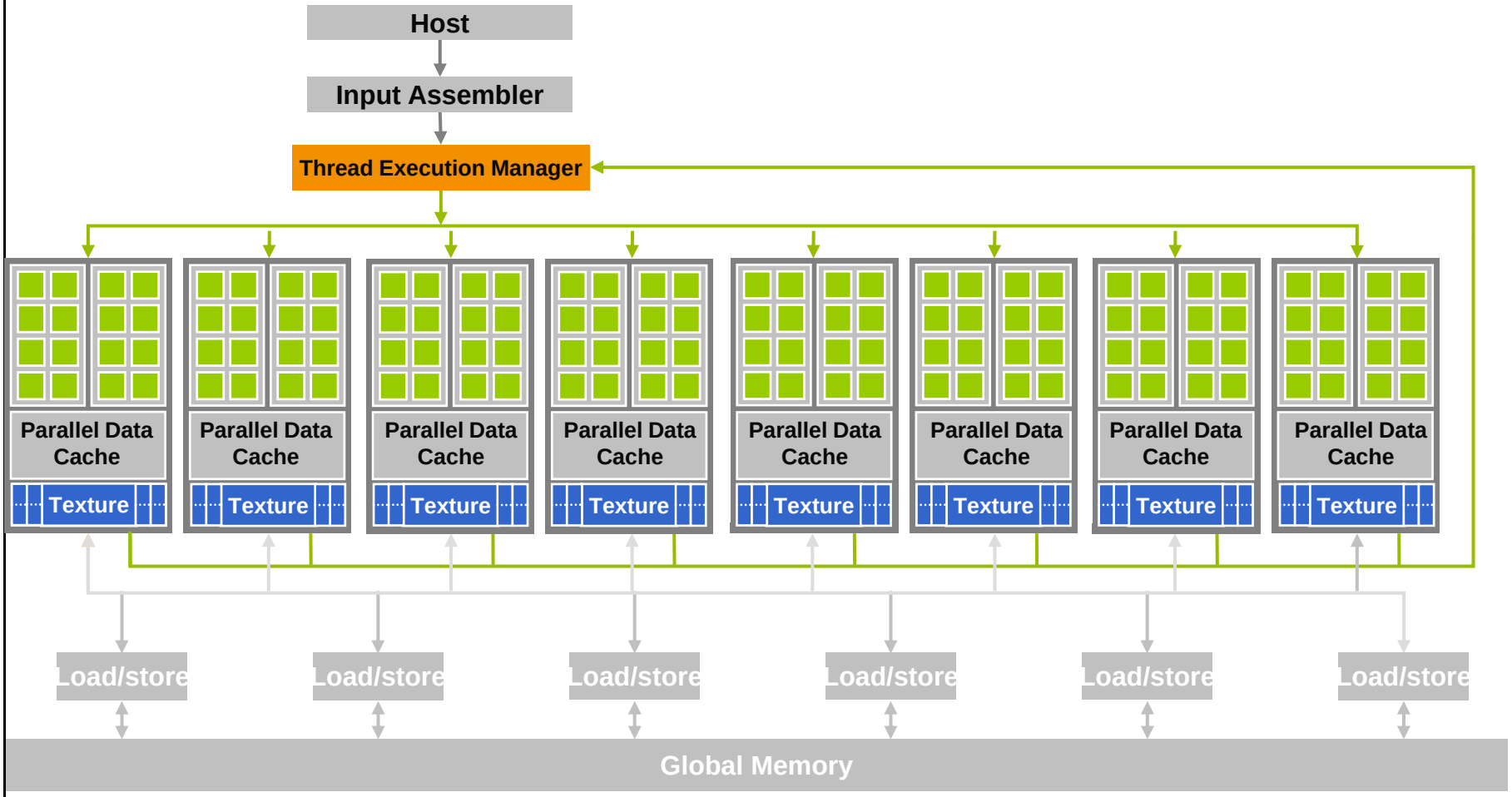


Execution Mode– Graphics



NVIDIA G80 GPU

Execution Mode– General Computing



NVIDIA G80 GPU

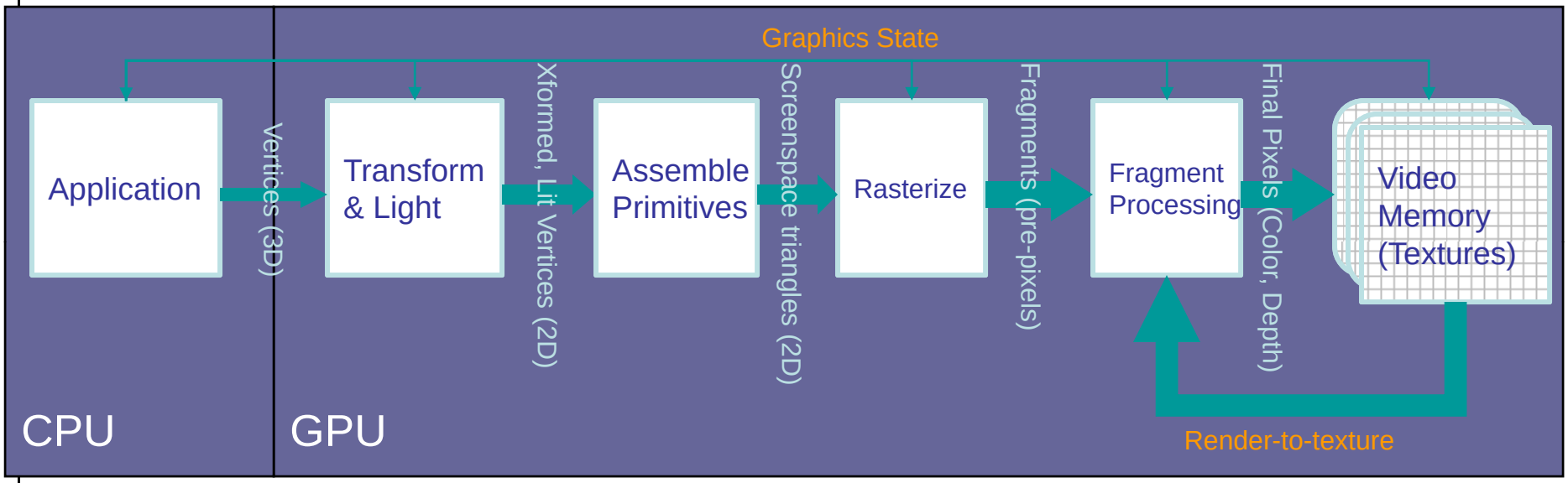
GPGPU from Graphics Point of View

- Graphics Processing Pipeline (on GPU)
  - Fixed function pipeline
  - Programmable pipeline with Shaders
- GPU Processing Model
  - Stream computing model

3D Graphics Applications

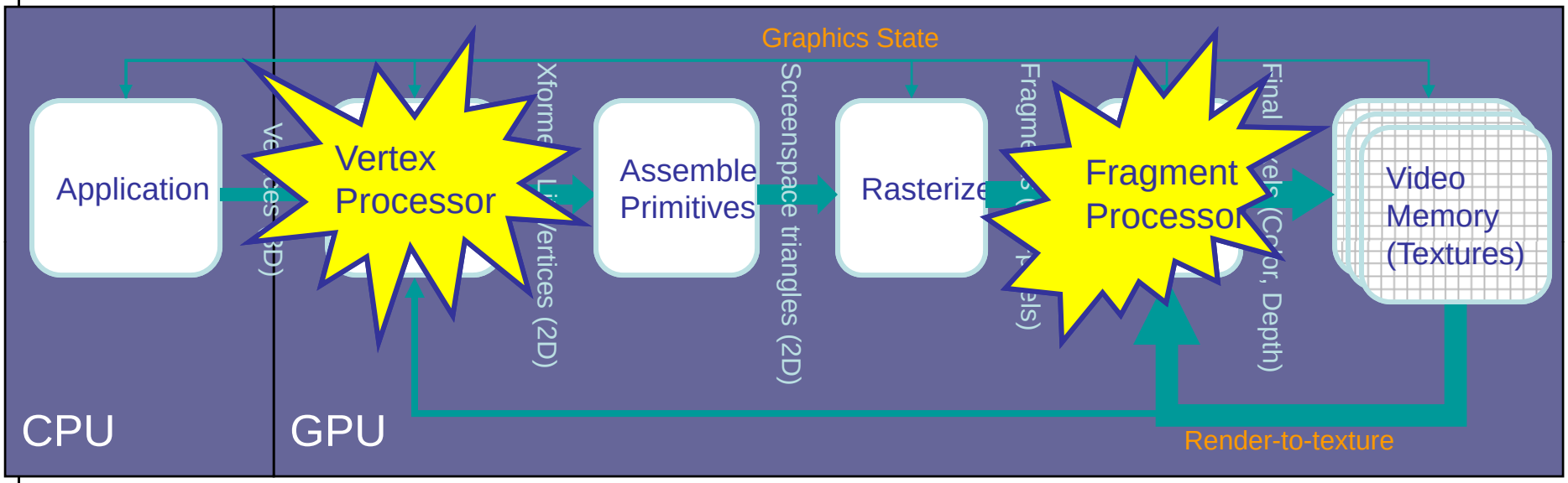
- Demos.

GPU Fundamentals: The Graphics Pipeline



- **A simplified graphics pipeline**

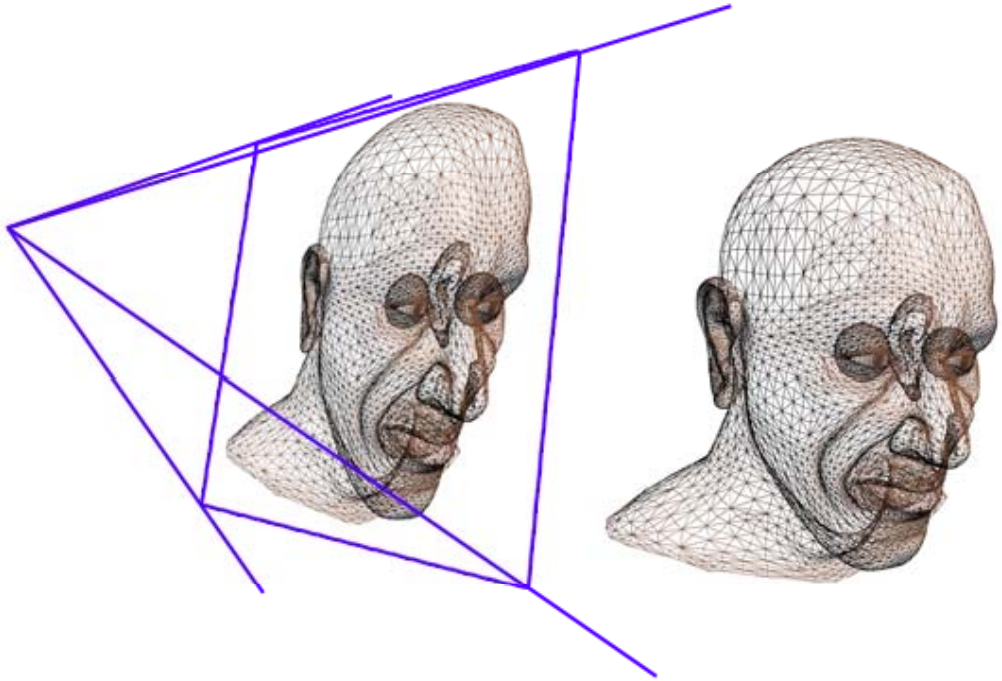
Programmable Graphics Pipeline - Shaders



- Programmable vertex processor!
- Programmable fragment processor!

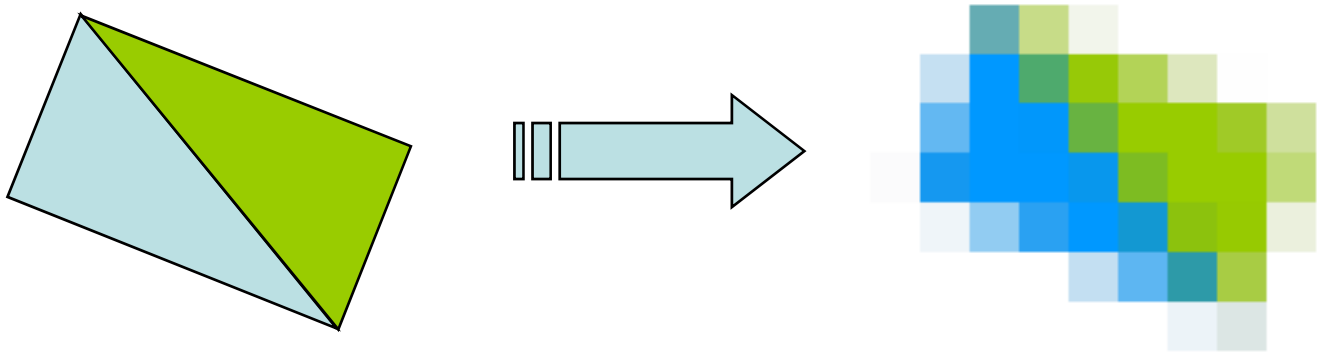
GPU Pipeline: Transform

- Vertex processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting



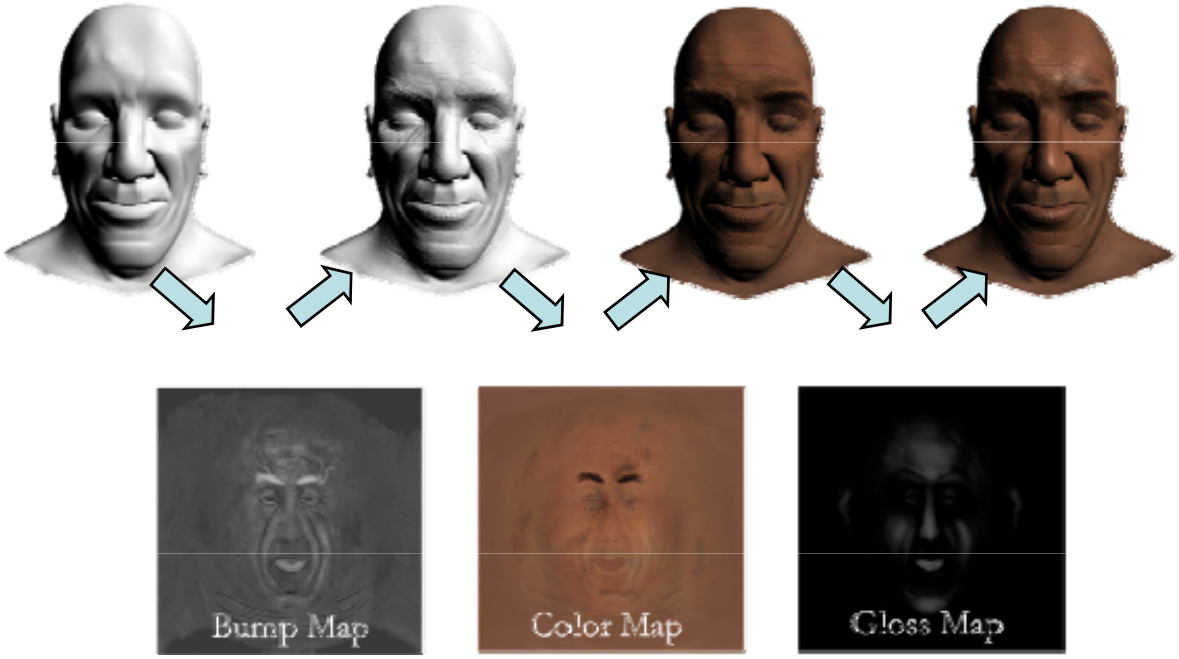
GPU Pipeline: Rasterize

- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Fragment = image fragment
      - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels



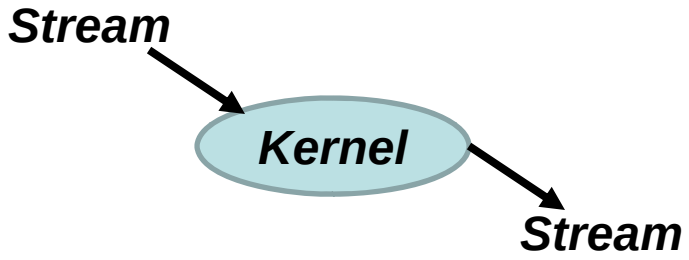
Pixel / Fragment Processor

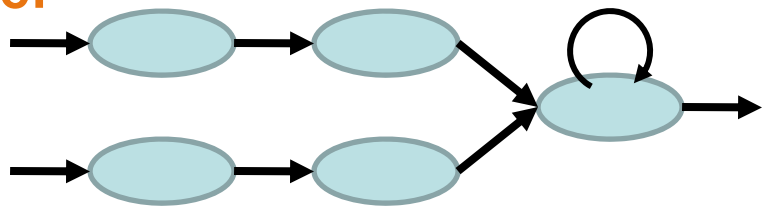
- Fragment processors (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)



GPU Programming Model: Stream

- Stream Programming Model
- Streams:
  - An array of data units
- Kernels:
  - Take streams as input, produce streams at output
  - Perform computation on streams
  - Kernels can be linked together





Why Streams?

- Ample computation by exposing parallelism
  - Stream expose data parallelism
    - Multiple stream elements can be processed in parallel
  - Pipeline (task) parallelism
    - Multiple tasks can be processed in parallel
- Efficient communication
  - Producer-consumer locality
  - Predictable memory access pattern
    - Optimize for throughput of all elements, not latency of one
    - Processing many elements at once allows latency hiding

Reading Material

- NVIDIA CUDA Programming Guide 2.0, Chapter One
  http://www.nvidia.com/object/cuda_develop.html and looking for documentation
- NVIDIA Geforce GTX 280 Technical Brief
  http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf