GPUs and Heterogeneous Computing Systems

Politecnico di Milano

v1

Christian Pilato <christian.pilato@polimi.it>

Outline

- Heterogenous Computing
- GPU vs CPU
- GPU structure

Procedural Synthesis in a nutshell

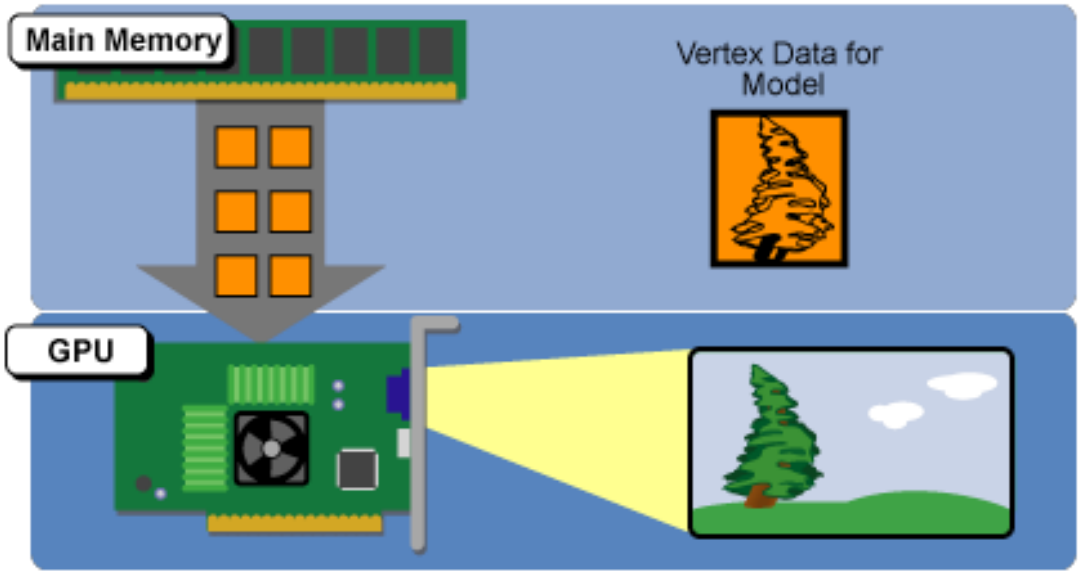
- Procedural synthesis is about making optimal use of system bandwidth and main memory by dynamically generating lower-level geometry data from statically stored higher-level scene data

- For 3D games
  - Artists use a 3D rendering program to produce content for the game
  - Each model is translated into a collection of polygons
  - Each polygon is represented in the computer's memory as a collection of vertices

- When the computer is rendering a scene in a game in real-time
  - Models that are being displayed on the screen start in main memory as stored vertex data
  - That vertex data is fed from main memory into the GPU
    - where it is then rendered into a 3D image and displayed on the monitor as a sequence of frames.

Limitations

- There are two problems
  - The costs of creating art assets for a 3D game are going through the roof along with the size and complexity of the games themselves
  - Console hardware's limited main memory sizes and limited bus bandwidth



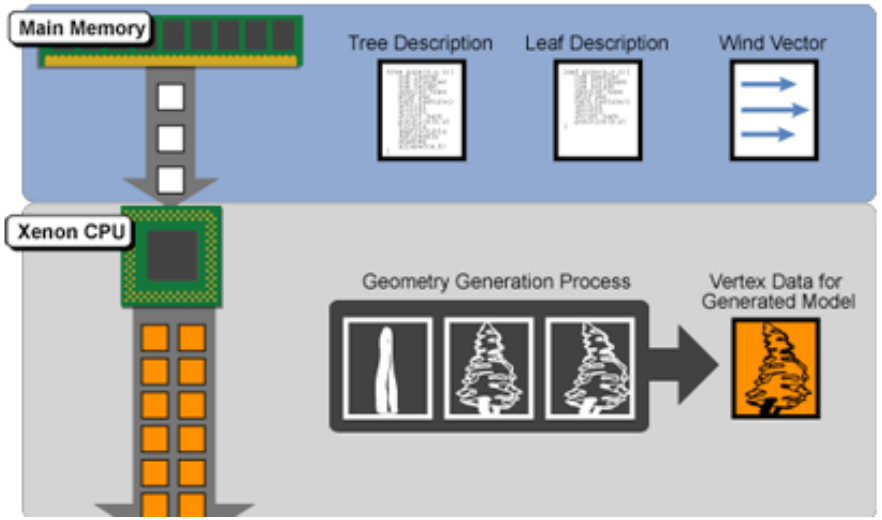
The Xbox 360's solution

The Xbox 360's solution

- Store high-level descriptions of objects in main memory

The Xbox 360's solution

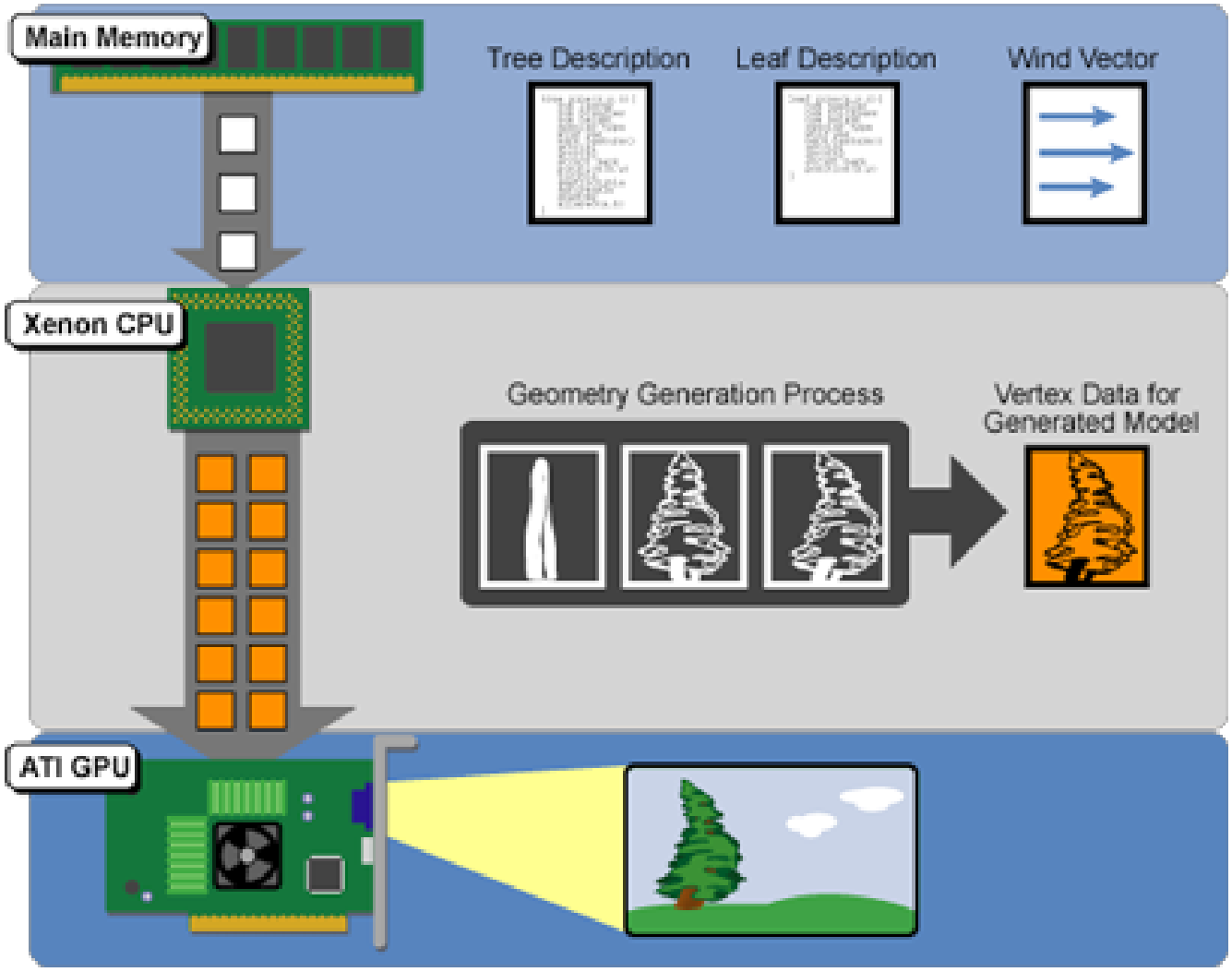
- Store high-level descriptions of objects in main memory
- Give the CPU the ability to procedurally generate the geometry of the objects on the fly
  - E.g., The vertex data is generated by one or more running threads



The Xbox 360's solution

- Store high-level descriptions of objects in main memory
- Give the CPU the ability to procedurally generate the geometry of the objects on the fly
  - E.g., The vertex data is generated by one or more running threads
- The GPU then takes that vertex information and renders the trees normally, just as if it had gotten that information from main memory

The Xbox 360's solution

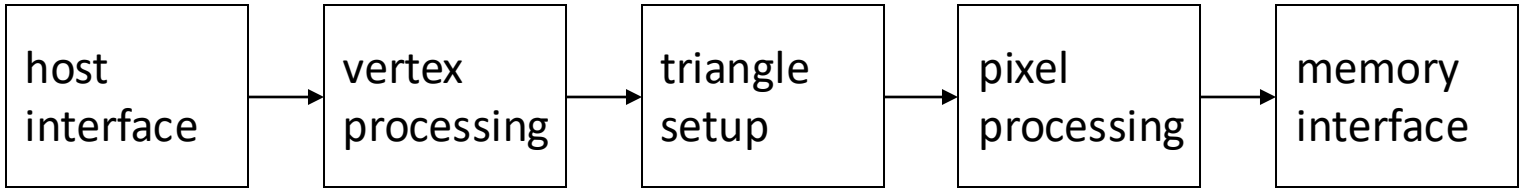


GPU vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds
- A GPU is, for the most part, deterministic in its operation (though this is quickly changing)
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces, as they need to shift around a lot more data than CPUs

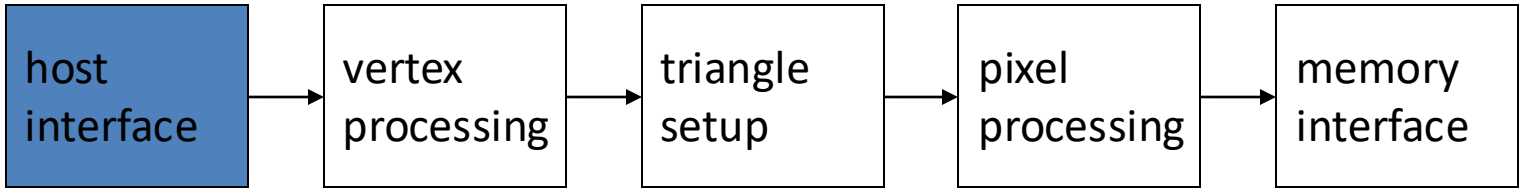
The GPU pipeline

- The GPU receives geometry information from the CPU as an input and provides a picture as an output
- Let's see how that happens



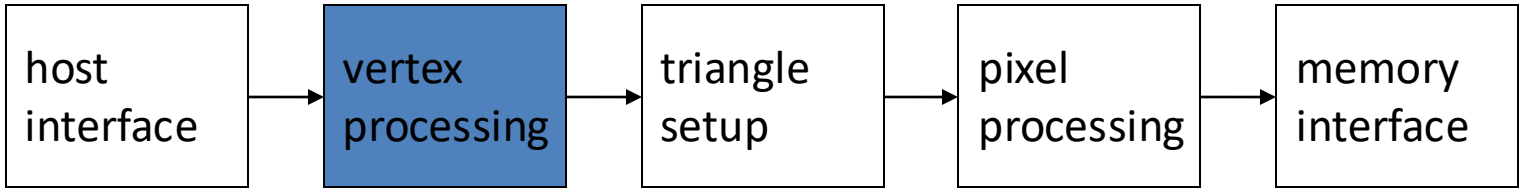
Host Interface

- The host interface is the communication bridge between the CPU and the GPU
- It receives commands from the CPU and also pulls geometry information from system memory
- It outputs a stream of vertices in object space with all their associated information (normals, texture coordinates, per vertex color etc)



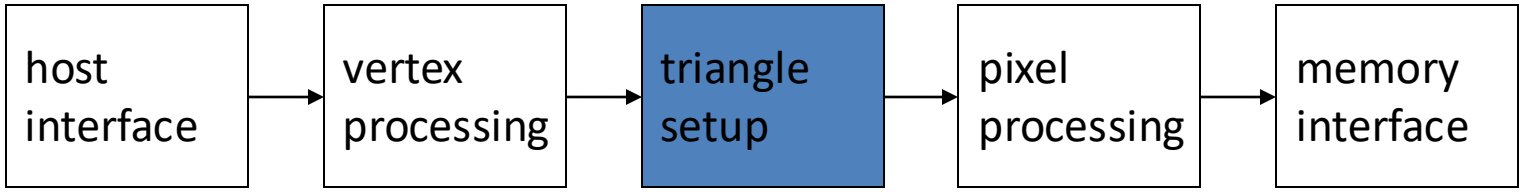
Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space
- This may be a simple linear transformation, or a complex operation involving morphing effects
- Normals, texcoords etc are also transformed
- No new vertices are created in this stage, and no vertices are discarded (input/output has 1:1 mapping)



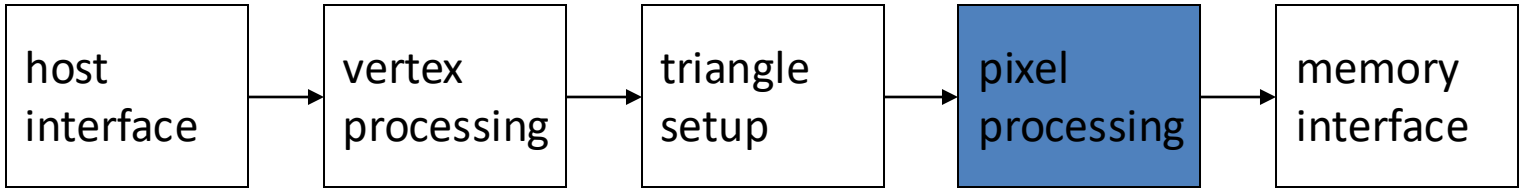
Triangle setup

- In this stage geometry information becomes raster information
  - screen space geometry is the input, pixels are the output
- Prior to rasterization, triangles that are backfacing or are located outside the viewing frustum are rejected
- Some GPUs also do some hidden surface removal at this stage



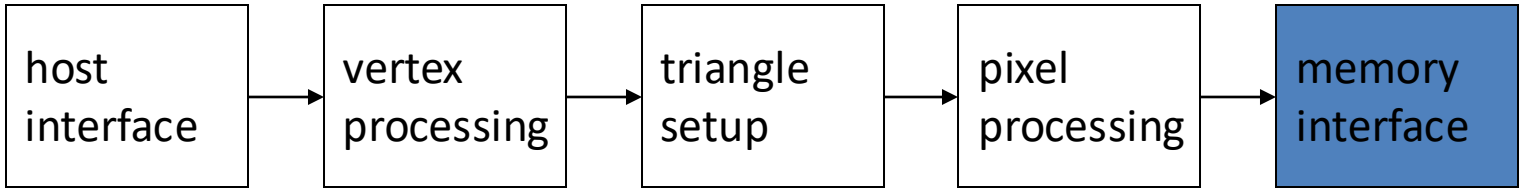
Fragment Processing

- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texcoord, etc), which are used to compute the final color for this pixel
- **The computations taking place here include texture mapping and math operations**
- Typically, the bottleneck in modern applications



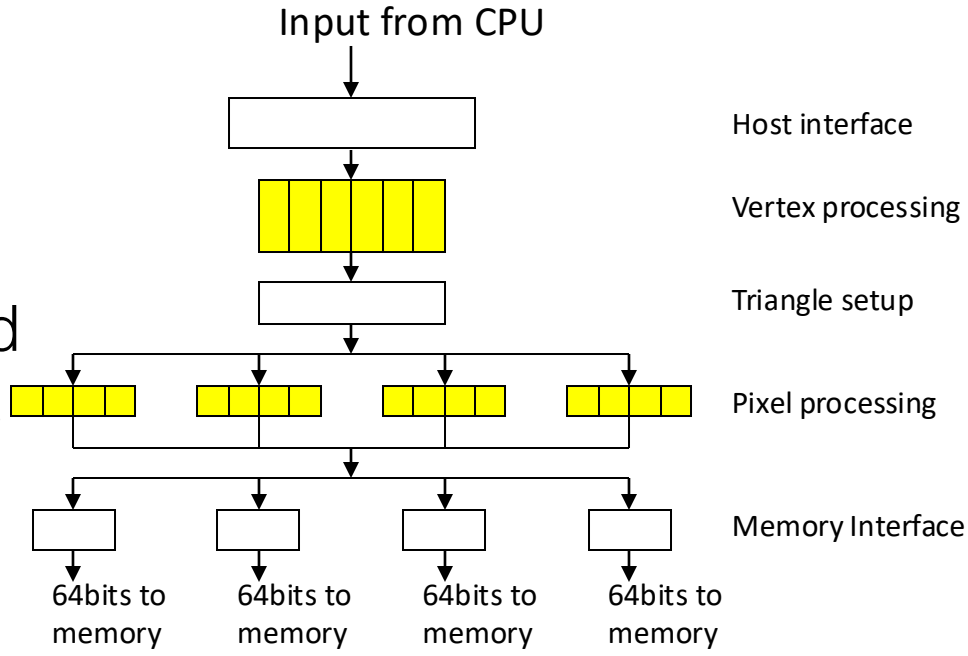
Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer
- Used to be the biggest bottleneck before fragment processing took over
- Before the final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size)



Programmability in the GPU

- Vertex and fragment processing, and now triangle set-up, are programmable
- **The programmer can write programs that are executed for every vertex as well as for every fragment**
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

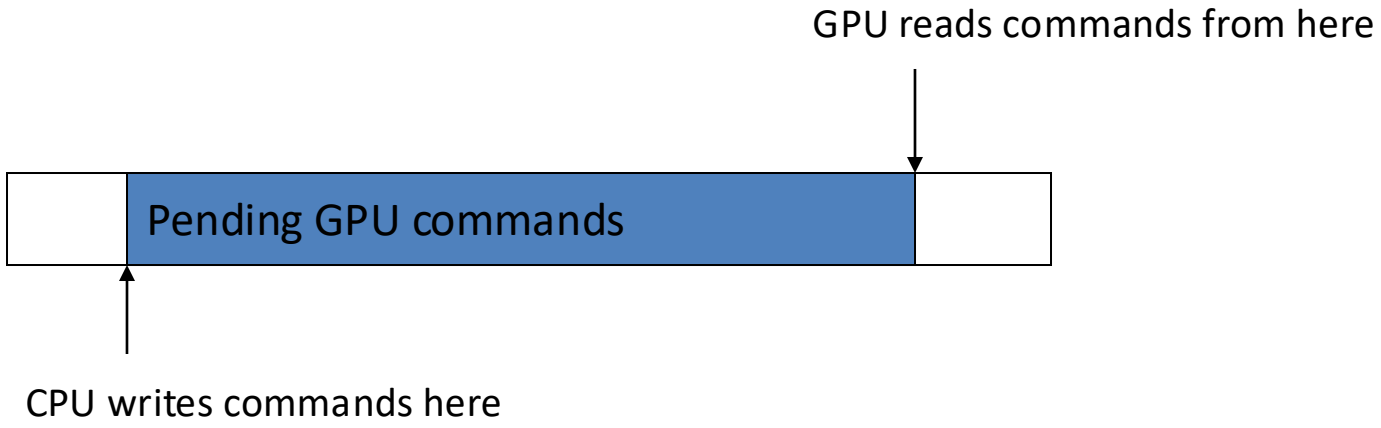


CPU/GPU interaction

- The CPU and GPU inside the system work in parallel with each other

CPU/GPU interaction

- The CPU and GPU inside the system work in parallel with each other
- There are two “threads” going on, one for the CPU and one for the GPU, which communicate through a command buffer:



CPU/GPU interaction (cont)

- If this command buffer is drained empty, we are CPU limited and the GPU will spin around waiting for new input.
  - NOTE: All the GPU power in the universe isn't going to make your application faster!

CPU/GPU interaction (cont)

- If this command buffer is drained empty, we are CPU limited and the GPU will spin around waiting for new input.
  - NOTE: All the GPU power in the universe isn't going to make your application faster!
- If the command buffer fills up, the CPU will spin around waiting for the GPU to consume it, and we are effectively GPU limited

CPU/GPU interaction (cont)

- Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model

CPU/GPU interaction (cont)

- Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model

Statement A
API call to draw object
Statement B

CPU/GPU interaction (cont)

- Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model
- In the CPU program below, the object is not drawn after statement A and before statement B:

Statement A
API call to draw object
Statement B

CPU/GPU interaction (cont)

- Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model
- In the CPU program below, the object is not drawn after statement A and before statement B:

```
Statement A
API call to draw object
Statement B
```

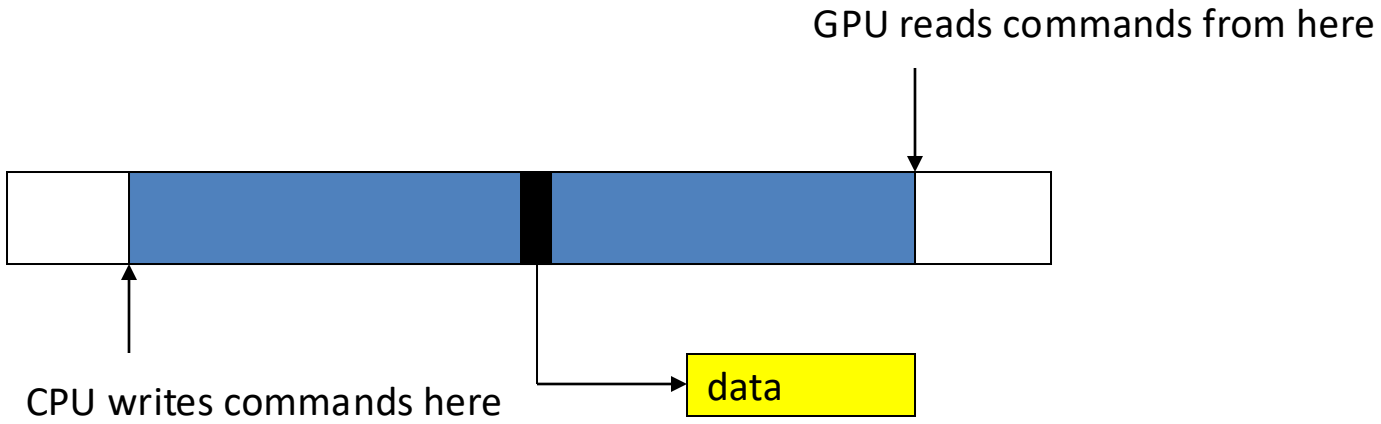
- Instead, the API call adds the command to draw the object to the GPU command buffer

Synchronization issues

- This leads to a number of synchronization considerations

Synchronization issues

- This leads to a number of synchronization considerations
- In the figure below, the CPU must not overwrite the data in the “yellow” block until the GPU is done with the “black” command, which references that data:

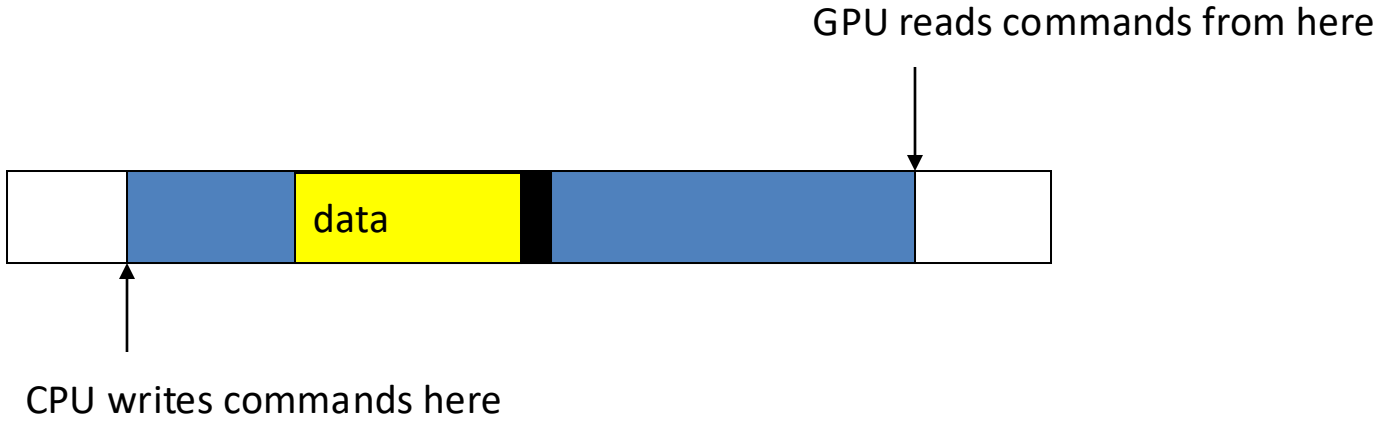


Syncronization issues (cont)

- Modern APIs implement semaphore-style operations to keep this from causing problems
- If the CPU attempts to modify a piece of data that is being referenced by a pending GPU command, it will have to spin around waiting until the GPU is finished with that command
- While this ensures correct operation, it is not good for performance since there are a million other things we'd rather do with the CPU instead of spinning
- The GPU will also drain a big part of the command buffer, thereby reducing its ability to run in parallel with the CPU

Inlining data

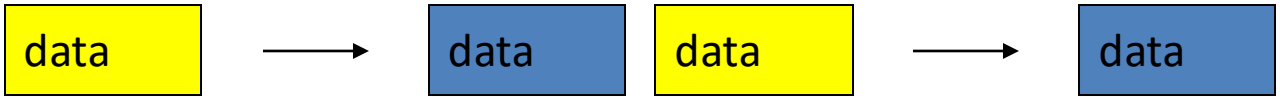
- One way to avoid these problems is to inline all data to the command buffer and avoid references to separate data:



- However, this is also bad for performance, since we may need to copy several megabytes of data instead of merely passing around a pointer

Renaming data

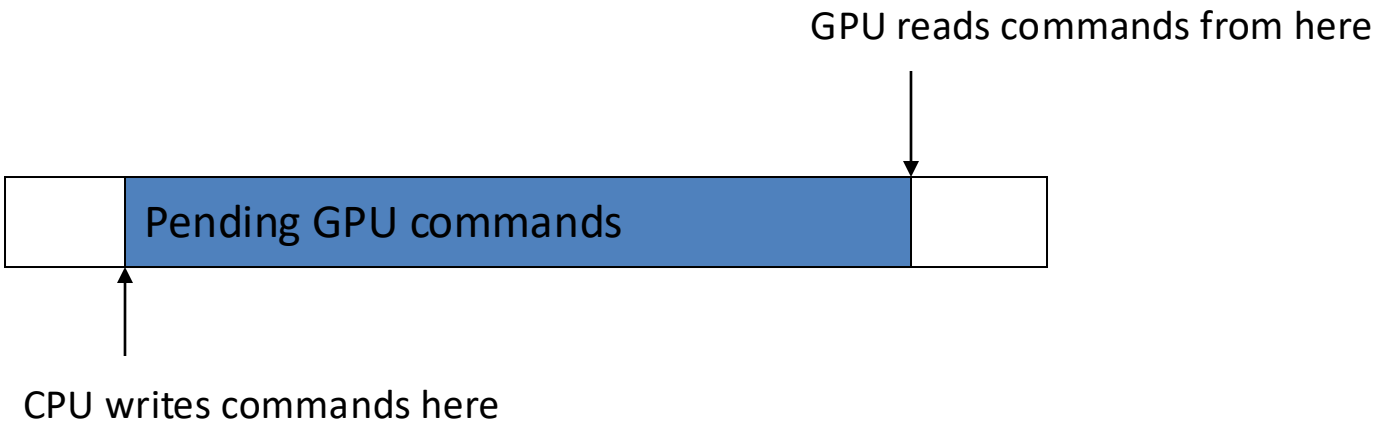
- A better solution is to allocate a new data block and initialize that one instead, the old block will be deleted once the GPU is done with it
- Modern APIs do this automatically, provided you initialize the entire block (if you only change a part of the block, renaming cannot occur)



- Better yet, allocate all your data at startup and don't change them for the duration of execution (not always possible, however)

GPU readbacks

- The output of a GPU is a rendered image on the screen, what will happen if the CPU tries to read it?



- The GPU must be synchronized with the CPU, i.e., it must drain its entire command buffer, and the CPU must wait while this happens

GPU readbacks (cont)

- We lose all parallelism, since first the CPU waits for the GPU, then the GPU waits for the CPU (because the command buffer has been drained)
- Both CPU and GPU performance take a nosedive

GPU readbacks (cont)

- We lose all parallelism, since first the CPU waits for the GPU, then the GPU waits for the CPU (because the command buffer has been drained)
- Both CPU and GPU performance take a nosedive

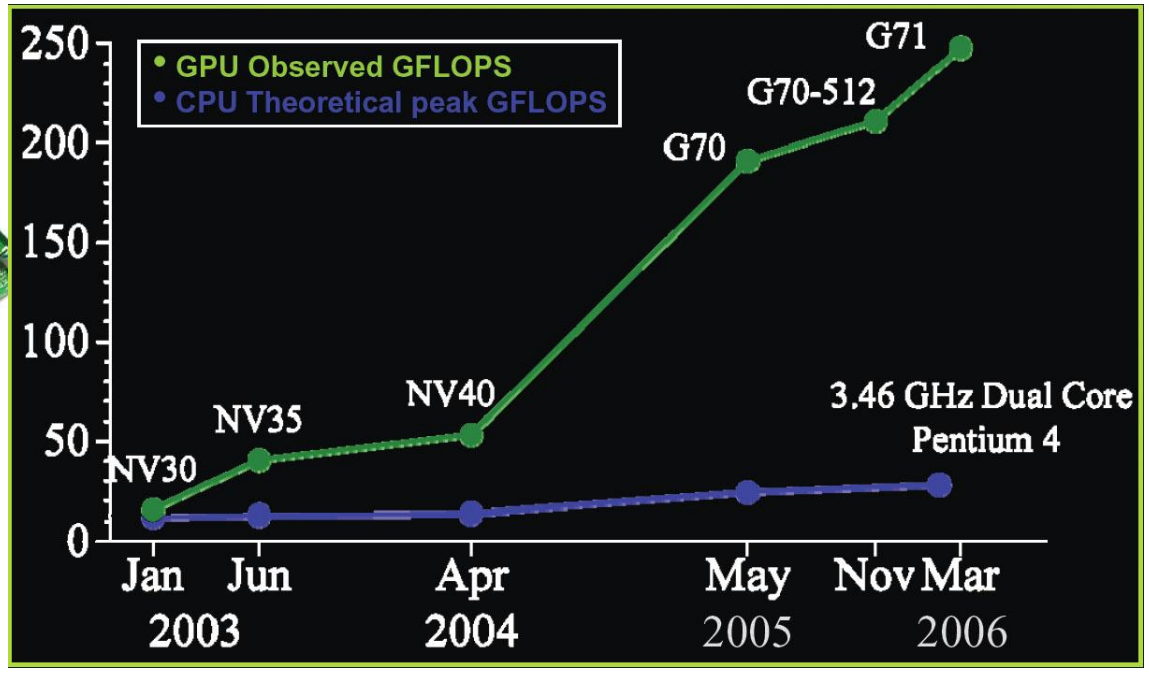
- Bottom line: the image the GPU produces is for your eyes, not for the CPU (treat the CPU -> GPU highway as a one way street)

Some more GPU tips

- Since the GPU is highly parallel and deeply pipelined, try to dispatch large batches with each drawing call
- Sending just one triangle at a time will not occupy all of the GPU's several vertex/pixel processors, nor will it fill its deep pipelines
- Since all GPUs today use the zbuffer algorithm to do hidden surface removal, rendering objects front-to-back is faster than back-to-front (painters algorithm), or random ordering
- Of course, there is no point in front-to-back sorting if you are already CPU limited

The nVidea G80 GPU

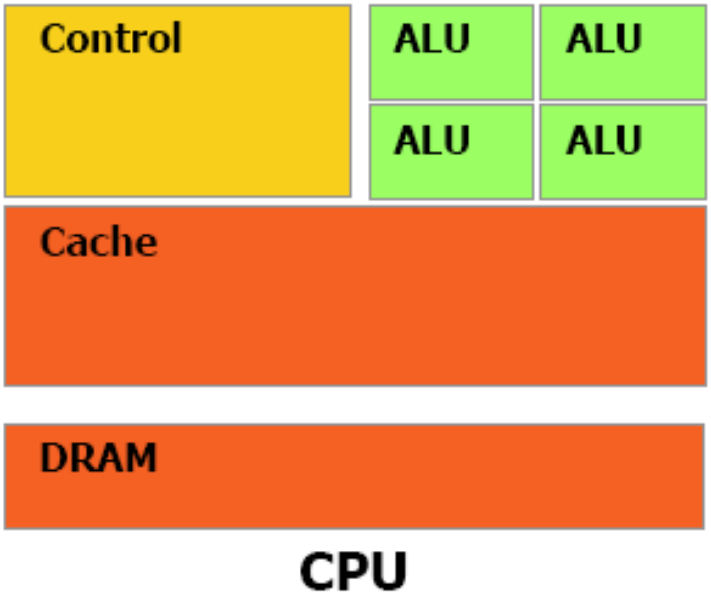
- 128 streaming floating point processors @1.5Ghz
- 1.5 Gb Shared RAM with 86Gb/s bandwidth
- 500 Gflop on one chip (single precision)



Why are GPUs so fast?

- Entertainment Industry has driven the economy of these chips?
  - Males (age 15-35) buy $10B in video games/year
- Moore's Law ++
- Simplified design (stream processing)
- Single-chip designs

Modern GPU has more ALU's



Intel Pentium 5 Prescott

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 1024 entries.

Return Stacks (4 x16 entries)

Trace Cache next IP's (4x)

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 128 entry, fully associative for 4k and 4M pages. In: Virtual address [47:12] Out: Physical address [39:12] + 2 page level bits

Instruction Fetch from L2 cache and Branch Prediction

Front Side Bus Interface, 533..800 MHz

Instruction Trace Cache

Execution Pipeline Start

Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

uOp Schedulers

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

FP Move Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Load / Store uOp Scheduler: (8x8 dependency matrix)

FP, MMX, SSE1..3

Floating Point, MMX, SSE1..3 Renamed Register File 256 entries of 128 bit.

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle

(2) Integer Renamed Register File 256 entries of 32 bit (+ 6 status flags) 12 read ports and six write ports

(3) Databus switch & Bypasses to and from the Integer Register File.

(4) Flags, Write Back

(5) Double Pumped ALU 0

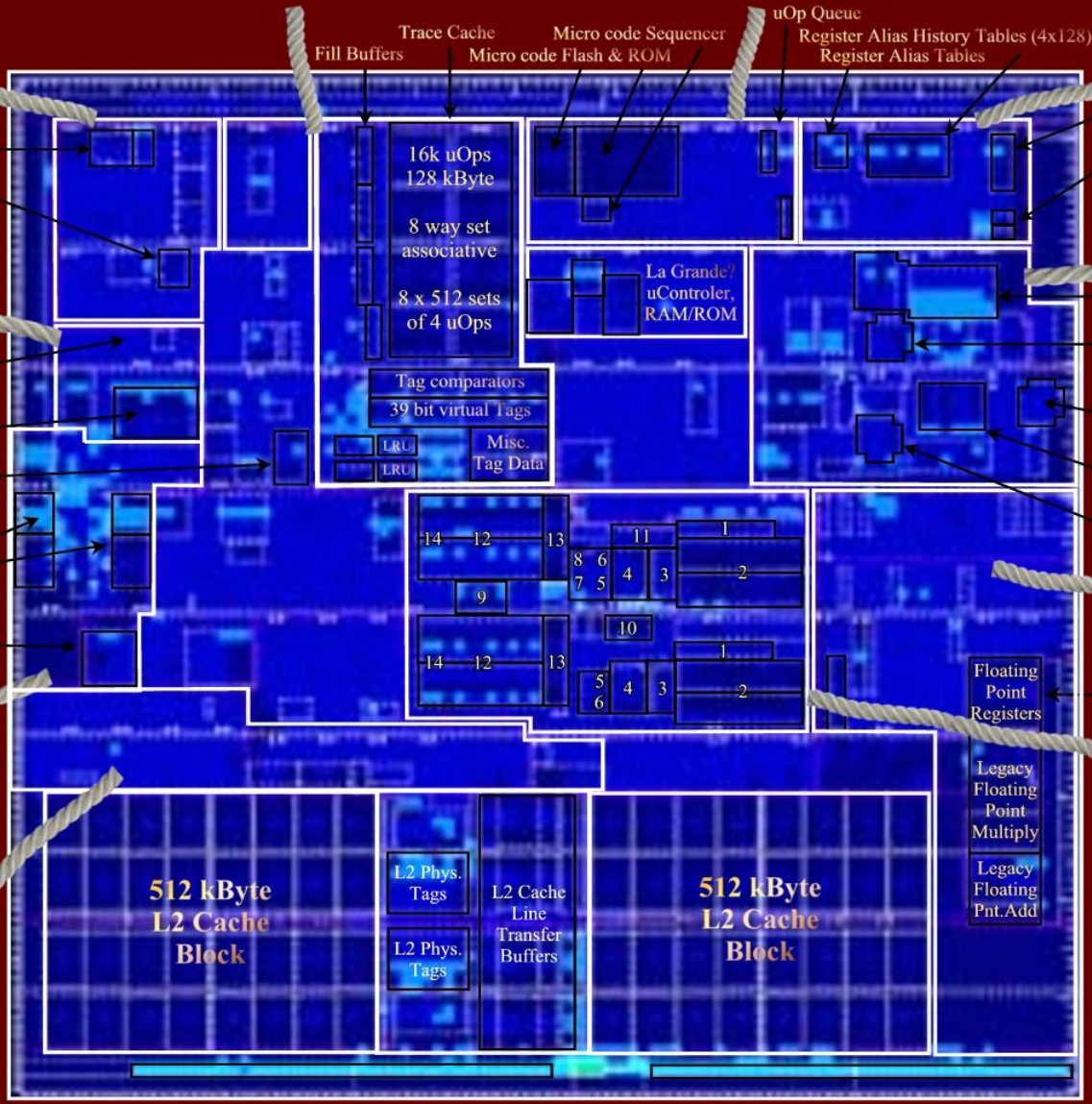
(6) Double Pumped ALU 1

(7) Load Address Generator Unit

(8) Store Address Generator Unit

(9) Load Buffer (96 entries)

(10) Store Buffer (48 entries)



(13) Databus multiplexing

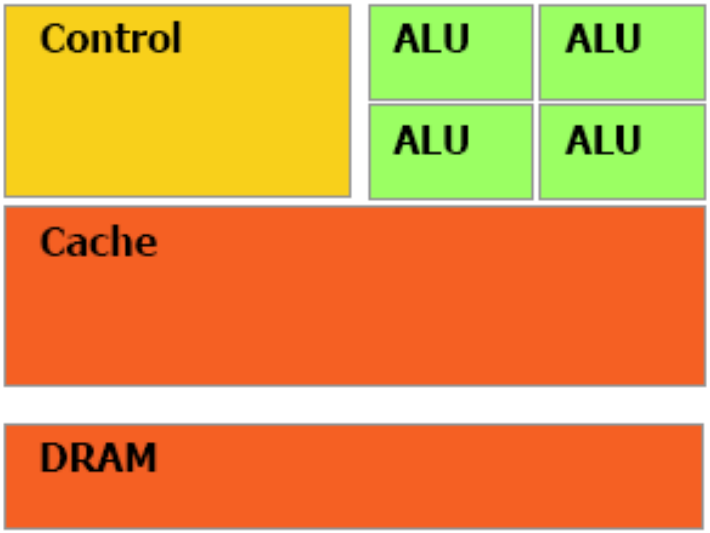
(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

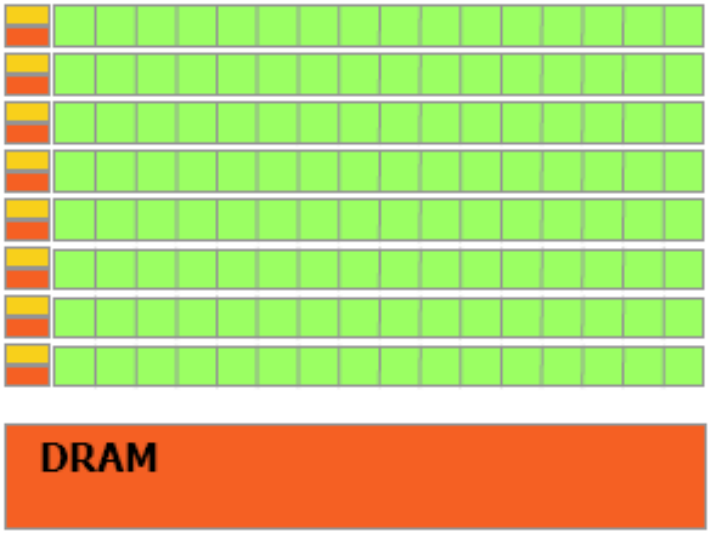
(11) ROB Reorder Buffer 4x64 entries

(12) 16 kByte Level 1 Data cache four way set associative. 1R/1W

April 19, 2003 www.chip-architect.com

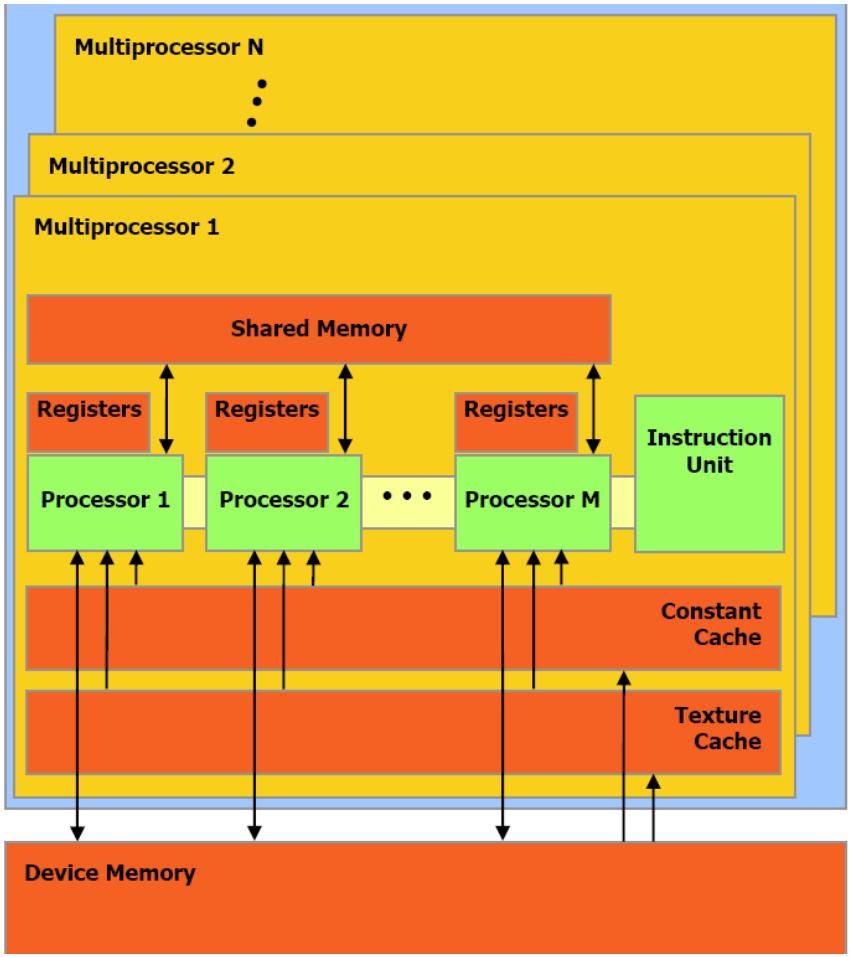
Modern GPU has more ALU's





GPUs devote more transistors to data processing

nVidia G80 GPU Architecture Overview



- 16 Multiprocessors Blocks
- Each MP Block Has:
  - 8 Streaming Processors (IEEE 754 spfp compliant)
  - 16K Shared Memory
  - 64K Constant Cache
  - 8K Texture Cache
- Each processor can access all of the memory at 86Gb/s, but with different latencies:
- Shared – 2 cycle latency
- Device – 300 cycle latency

A Specialized Processor

- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- Not As Efficient For
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

Heterogeneous System Architecture

More information available at: http://hsafoundation.com/

Heterogeneous System Architecture

More information available at: http://hsafoundation.com/

Questions?