ECE 571 – Advanced Microprocessor-Based Design Lecture 34

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

6 December 2024

Project/HW Reminder

- Homework #11 was posted, Nvidia Blackwell B100 Reading, due Monday
- Don't forget projects!

Things not mentioned last time

- Tradeoff rasterization vs ray-trace
- Rasterization really good at drawing lots of triangles
- No shadows, transparency, glass spheres, water effects
- Any games with those effects are faking them via software

Modern Graphics Cards

- Essentially high-end linear algebra / 3D rendering supercomputers
- Can draw a lot of power
- 2D (optional afterthought these days), possibly your 2d window is just a texture drawn on two triangles
- Probably have a “compositing” window manager
- Can contain other hardware accelerators (such as Video decoders)
- Video driver does a lot of heavy lifting, translates

the high-level APIs into what the underlying hardware expects

Interface – Integrated vs Standalone

- Integrated
  - Built into motherboard/chipset/processor
  - Can share memory (and bandwidth) with CPU
  - Traditionally less capable, but that is changing
- Standalone
  - Usually in PCIe slot, bandwidth constrained
  - Can draw lots of power
  - Can have multiple

Video RAM

- VRAM – dual ported. Could read out full 1024Bit line and latch for drawing, previously most would be discarded (cache line read)
- GDDR3/4/5 – traditional one-port RAM. More overhead, but things are fast enough these days it is worth it.
- Confusing naming, GDDR3 is equivalent of DDR2 but with some speed optimization and lower voltage (so higher frequency)

Busses

- DDC – i2c bus connection to monitor, giving screen size, timing info, etc.
- PCIe (PCI-Express) – most common bus in x86 systems
  Original PCI and PCI-X was 32/64-bit parallel bus
  PCIe is a serial bus, sends packets
  Can power 25W, additional power connectors to supply can have 75W, 150W and more
  Can transfer 8GT/s (giga-transfers) a second
  In general PCIe is limiting factor to getting data to GPU.

Connectors

CRTC (CRT Controller) Can point to same part of memory (mirror) or different.

- RCA – composite/analog TV
- VGA – 15 pin, analog
- DVI – digital and/or analog. DVI-D, DVD-I, DVD-A
- HDMI – compatible with DVI (though content restrictions). Also audio. HDMI 1.0 – 165MHz, 1080p

or 1920x1200 at 60Hz. TMDS differential signaling. Packets. Audio sent during blanking.

- Display Port – similar but not the same as HDMI
- Thunderbolt – combines PCIe and DisplayPort. Intel/Apple. Originally optical, but also Copper. Can send 10W of power.
- LVDS – Low Voltage Differential Signaling – used to connect laptop LCD

LCD Displays (sic)

- Crystals twist in presence of electric field
- Various types, one that is out of patent is 2-(4-alkoxyphenyl)-5-alkylpyrimidine with cyanobiphenyl
- Asymmetric on/off times
- Passive (crossing wires) vs Active (Transistor at each pixel)
- Passive have to be refreshed constantly
- Use only 10% of power of equivalent CRT
- Circuitry inside to scale image and other post-processing

- Need to be refreshed periodically to keep their image
- New "bistable" display under development, requires no power to hold state
- Aside, does it take more energy on vs off? Black vs white screen?

Other Display Tech

- LED
- OLED
- QLED (quantum dots)
- Plasma

Graphics Programming Interfaces

- OpenGL – SGI (Khronos)
- DirectX – Microsoft (Direct3d)
- Vulkan (sort of next gen OpenGL. Lower level, closer to hardware)
- Metal – from Apple
- WebGL – javascript/web
- OpenGL ES – embedded subset

GPGPU Programming Interfaces

- Interfaces needed, as GPU companies do not like to reveal what their chips due at the assembly level.
  - CUDA – Nvidia
  - ROCm – AMD
  - OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc
  - OpenACC?

GPUs

- Massively parallel matrix-processing CPUs that write to the frame buffer (or can be used for calculation)
- Originally just did lighting and triangle calculations. Now shader languages and fully generic processing
- Texture control, 3d state, vectors
- Front-buffer (written out), Back Buffer (being rendered) Z-buffer (depth)
- Display memory often broken up into tiles (improves cache locality)

GPU Low-Level Software

- APIs abstract away actual hardware, more than CPUs do
- Often you can't really do "assembly" language, or at least it's sometimes no documented
  - NVIDIA – undocumented
  - AMD, Intel – some things documented
  - Embedded (VideoCore, MALI) – reverse engineered?

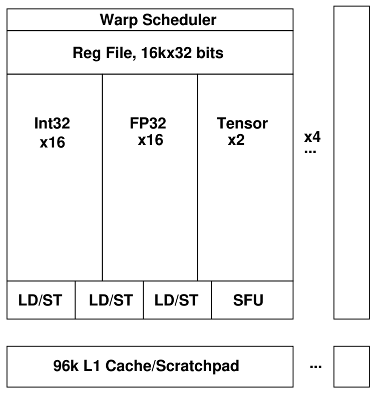
GPU Cores

- Often some debate about what constitutes a GPU core
- Different companies with different terms
  - Nvidia – SM (stream multiprocessors)
  - AMD – CU (compute unit) or WGP (Workgroup Processor)
  - Inte; – EU (execution unit) or Xe core
- Also custom cores
  - RT – ray tracing (BVH bounding volume hierarchy)
  - Tensor cores – AI, low-precision matrix

Very Wide Threads

- Warp – Nvidia
- Wavefront – AMD
- Wave – DX12
- Subgroup – Vulkan

Example Hardware



- Above based on older NVIDIA Tesla GPU
- Note register file acutually bigger than scratchpad cache
- Cache often software managed
- Latency hiding, when inevitable stall waiting for mem, run another thread group that's waiting

Low Level Code Theory

- CPU code you might do something like

```
for(x=0;x<1024;x++) {
            A[x]=B[x]*c+d;
}
```

- On a CPU at best if no dependencies can maybe run 4 - 6 things at a time?
- On a GPU, essentially unrolls the loop and you can run threads with maybe 1024 of these at once
- All 1024 (one for each x) load B, all multiply by c, all

add in d, all save out to A

- Trouble is flow control. If you have

```
if (x<10) something
else something_different;
```

you can't actually branch only some of the threads, instead all threads do both, and just the ones that aren't true ignore it. So potentially slower.

Key Ideas

- using many slimmed down cores
- have single instruction stream operate across many cores (SIMD)
- avoid latency (slow textures, etc) by working on another group when one stalls
- Avoid memory latency with calculation, not cache (which is how CPUs do it)

Latency vs Throughput

- CPUs = Low latency, low throughput
- GPUs = high latency, high throughput
- CPUs optimized to try to get lowest latency (caches); with no parallelism have to get memory back as soon as possible
- GPUs optimized for throughput. Best throughput for all better than low-latency for one

Why GPUs?

- Newer example:
  - Cascade Lake, 1 TFLOP (64-bit floating point)
  - NVIDIA 3090 36 TFLOPs
- Newer example
  - Raspberry Pi, 700MHz, 0.177 GFLOPS
  - On-board GPU: Video Core IV: 24 GFLOPS

Graphics vs Programmable Use

Vertex	Vertex Processing	Data	MIMD processing
Polygon	Polygon Setup	Lists	SIMD Rasterization
Fragment	Per-pixel math	Data	Programmable SIMD
Texture	Data fetch, Blending	Data	Data Fetch
Image	Z-buffer, anti-alias	Data	Predicated Write

GPU Benefits

- Specialized hardware, concentrating on arithmetic. Transistors for ALUs not cache.
- Fast 32-bit floating point (16-bit? 8? 4?)
- Driven by commodity gaming, so much faster than would be if only HPC people using them.
- Accuracy? 64-bit floating point? 32-bit floating point? 16-bit floating point? Doesn't matter as much if color slightly off for a frame in your video game.
- highly parallel

GPU Challenges

- Originally optimized for 3d-graphics, not always ideal for other things
- Need to port code, usually can't just recompile cpu code.
- Companies secretive.
- serial code
- a lot of control flow
- lot of off-chip memory transfers

Older / Traditional GPU Pipeline

- In old days, fixed pipeline (lots of triangles).
- Modern chips much more flexible, but the old pipeline can still be implemented in software via the fancier interface.

GPGPUs

- Started when the vertex and fragment processors became generically programmable (originally to allow more advanced shading and lighting calculations)
- By having generic use can adapt to different workloads, some having more vertex operations and some more fragment

Shader Programming

- There are competitions. Also see `shadertoy.com`
- Vertex Shader
  - Vertex transform
  - Object space to clip space
  - Compute colors, normals, texture co-ords
  - Can displace/distort (move vertices: wave flag)
  - Can animate (move vertices: move fish)
- Fragment Shader
  - Compute and color

- Get data from vorteces and textures
- Can make better materials. Glossy, reflections, bumpy, shadows

GLSL Shader Programming

- Similar to C code
- Based on OpenGL
- vertex
  - Each time screen drawn main() called once per vertex
  - Massively parallel
  - Have vars. Can get positions
- Fragment
  - Each time screen drawn main() called once per pixel
  - Can get x/y

Example Shader 3.0 (DX9) Capabilities – Vertex Processor

- They are up to Pixel Shader 5.0 now
- 512 static / 65536 dynamic instructions
- Up to 32 temporary registers
- Simple flow control
- Texturing – texture data can be fetched during vertex operations
- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:

- EXP, EXPP, LIT, LOGP (exponential)
- RCP, RSQ (reciprocal, r-square-root)
- SIN, COS (trig)

Example Shader 3.0 (DX9) capabilities– Fragment Processor

- 65536 static / 65536 dynamic instructions (but can time out if takes too long)
- Supports conditional branches and loops
- fp32 and fp16 internal precision
- Can do 4-wide MAD and 4-wide DP4 (dot product)

Program

- Typically textures read-only. Some can render to texture, only way GPU can share RAM w/o going through CPU. In general data not written back until entire chunk is done. Fragment processor can read memory as often as it wants, but not write back until done.

- Only handle fixed-point or floating point values

- Analogies:
  – Textures $==$ arrays

- Kernels $==$ inner loops
- Render-to-texture $==$ feedback
- Geometry-rasterization $==$ computation. Usually done as a simple grid (quadrilateral)
- Texture-coordinates $=$ Domain
- Vertex-coordinates $=$ Range

Flow Control, Branches

- only recently added to GPUs, but at a performance penalty.
- Often a lot like ARM conditional execution

Terminology (CUDA)

- Thread: chunk of code running on GPU.
- Warp: group of thread running at same time in parallel simultaneously
- Block: group of threads that need to run
- Grid: a group of thread blocks that need to finish before next can be started

Terminology (cores)

- Confusing. Nvidia would say GTX285 had 240 stream processors; what they mean is 30 cores, 8 SIMD units per core.