ECE 574 – Cluster Computing
Lecture 17

Vince Weaver

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

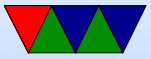
vincent.weaver@maine.edu

11am, Barrows 133

21 March 2024

Announcements

- HW#7 due Monday
- Don't forget project topics were due today! Please send if you haven't
- Reminder about talk on Friday
- Reminder office hours cancelled Monday due to Faculty Interview talk

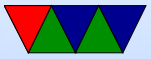


GPUs in the News

- Future of AMD vs CUDA

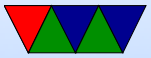
  https://www.hpcwire.com/2023/10/05/how-amd-may-get-across-the-cuda-moat/

- NVIDIA just announce Blackwell GPU
  - Named for David Blackwell, mathematician
  - Successor to Hopper, 5x performance (at 4 bit fp)
  - 20 PFLOPs (at 4 bit fp), 300W? B100?
  - Two dies, 10TB/s NVLINK, 8 HBM3e stacks (192GB)
  - Tower of them get get over an exaflop (a lot of power, and again 4 bit fp, for AI)



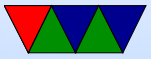
Interfaces for 3D Graphics

- 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



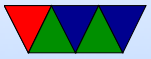
3D Graphics

- Two common ways to do 3D graphics
  - Ray tracing, very accurate, but slow
  - Rasterization, low quality, but fast enough to do in real time
- Can do either completely in software on CPU (and people did), but much faster if you accelerate with hardware



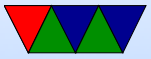
Ray-tracing / Ray-casting

- TODO: diagram
- Objects placed in 3d space
- Rays of light traced from eye through each pixel on screen until hit object
- Based on material they hit, reflect, refract, take on color, etc
- Can do reverse where light source sends out rays and you bounce them around until they hit pixel on screen



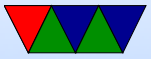
Question: how does Hardware Raytrace work

- Accelerate in hardware ray-tracing, though usually only partially
- NVIDIA: Optix Library
- You describe how rays behave
- Details are a bit hard to get



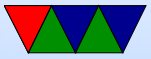
Ray-marching

- Just a place holder, it's a related technique often used in size-coded demoscene productions and I've been meaning to learn more about it



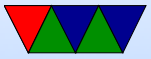
Rasterization

- TODO: show diagram
- Objects made up of many triangles (or quads)
- Send vertices to card
- Project to 2d screen
- Broken up to pixels and shaded/textured
  Color/shading based on angle with light source (normals)
- Clipping, depth



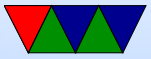
Rasterization on GPU

- CPU send list of vertices to GPU.
- Transform (vertex processor) (convert from world space to image space). 3d translation to 2d, calculate lighting. Operate on 4-wide vectors (x,y,z,w in projected space, r,g,b,a color space)
- Rasterizer – transform vertexes/vectors into a grid. Fragments. break up to pixels and anti-alias
- Shader (Fragment processor) compute color for each pixel. Use textures if necessary (texture memory, mostly



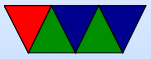
read)

- Write out to framebuffer (mostly write)
- Z-buffer for depth/visibility



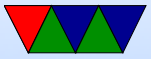
Rasterization Downsides

- Can't calculate shadows (have to do hacks)
- Can't easily do transparency (mirrors) or refraction (lenses)
- On the plus side it is fast



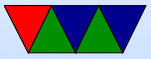
GPU Pipeline

- Old / Traditional
  - Implement rasterization in fixed hardware
  - Fixed pipeline (lots of triangles).
- Modern
  - Much more flexible, programmable almost general-purpose compute units
  - Old pipeline can still be implemented in software via the fancier interface



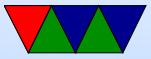
GPUs

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



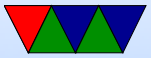
GPGPUs

- Can we use GPUs as an accelerator?
- 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



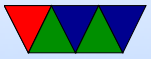
Is GPGPU worth it?

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



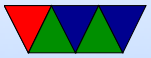
Graphics vs Programmable Use

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



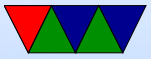
Key Idea

- 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)



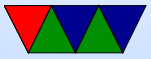
Latency vs Throughput

- CPUs = Low latency, low throughput
- GPUs = high latency, high throughput
- CPUs optimized to try to get lowest latency (caches); with no parallelism need fast access to memory to avoid stalls
- GPUs optimized for throughput. Best throughput for all better than low-latency for one



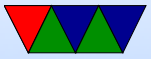
GPU Benefits

- 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



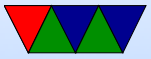
GPU Challenges

- 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



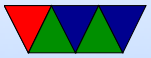
GPGPU Programming

- GPU companies do not like to reveal what their chips do at the low/assembly level
- Abstraction provided for programming them
  - CUDA (Nvidia)
  - OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc
  - ROCm (Radeon Open Compute Platform) (AMD)
  - OpenACC?

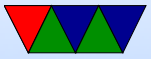


Shader Programming

- 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

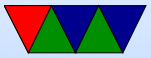


- Get data from vorteces and textures
- Can make better materials. Glossy, reflections, bumpy, shadows



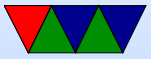
GLSL Shader Programming

- 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

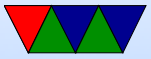


Example Shader 3.0 (DX9) Capabilities – Vertex Processor

- 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:

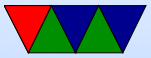


- EXP, EXPP, LIT, LOGP (exponential)
- RCP, RSQ (reciprocal, r-square-root)
- SIN, COS (trig)

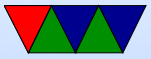


Example Shader 3.0 (DX9) capatbilities– Fragment Processor

- 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)

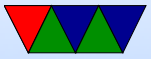


NVIDIA GPUs



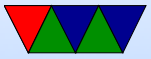
NVIDIA Generations

- Kepler
- Maxwell
- Pascal
- Turing (consumer)/Volta (pro)
- Ampere
- Lovelace/Hopper
- Blackwell

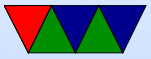


NVIDIA Workstation vs Gaming

- Quadro (Workstation) vs Geforce (Gaming) (note from 2023, they renamed these)
  - Quadro generally more RAM. higher Bus width
  - Fancier Drivers
  - Optimized for CAD type stuff and compute, rather than games
  - Higher reliability
  - Quadro better support for double-precision floats
  - More compute cores

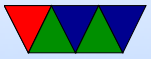


- Power limits

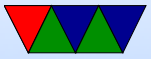


GPU hardware in my Lab

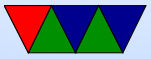
- Can use these for projects. I mostly get these for power measurement tests.
- NVIDIA RTX A2000 in Skylake
  - 6GB GDDR6, 192-bit, 288 GB/s
  - Ampere, PCIe4x16
  - 3328 CUDA cores, 104 tensor cores, 26 RT cores
  - 8 TFLOPS single-precision, RT 15.6 TFLOPS, Tensor 64 TFLOPS
  - 70W, DirectX 12.07, Vulkan 1.2



- NVIDIA RTX A2000 GA106 in RaptorLake
  - Roughly same as above but with 12GB RAM
- NVIDIA Quadro P2000 in Skylake (old)
  - 5GB GDDR5, 160-bit, 140 GB/s
  - 1024 cores, Pascal, PCIe3x16
  - 75W, DirectX 12.0, Vulkan 1.0
- NVIDIA Quadro P400 in Haswell-EP
  - 2GB GDDR5, 64-bit, up to 32 GB/s
  - 256 cores, Pascal architecture
  - 30W, OpenGL 4.5, DirectX 12.0
  - Low-power for server, as runs in 1U rack

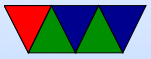


- NVIDIA Quadro K2200 in Quadro
  - So old the drivers don't want to support it anymore
  - 4GB GDDR5, 128-bit 80 GB/s
  - 640 cores, Maxwell architecture
  - 68W, OpenGL 4.5, DirectX 11.2



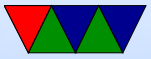
CUDA Programming background

https://docs.nvidia.com/cuda/cuda-c-programming-guide/



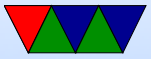
CUDA – installing

- On Linux need to install the proprietary NVIDIA drivers
- Have to specify nonfree on Debian.
- Debates over the years whether NVIDIA can have proprietary drivers; no one sued yet. (Depends on whether they are a "derived work" or not. Linus refuses to weigh in)
- Sometimes have issues where drivers won't install (currently having that issue on some of my machines)



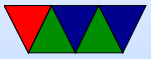
GPGPU Summary

- CPUs designed for fastest single-thread performance
  Lots of transistors for caching (to hide memory latency), control flow (branch predictors), caches
- GPUs designed to run as many threads as possible as once
  Use other methods to hide memory latency (mostly by moving to other threads if one batch is waiting)



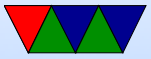
Programming a GPGPU (CUDA/OpenCL)

- Create a "kernel" which is a small GPU program that runs on a single thread. This will be run on many cores at a time.
- Allocate memory on the GPU and copy input data to it
- Launch the kernel to run many times in parallel. The threads operate in lockstep, all executing the same instruction in each thread.
- How is conditional execution handled? a lot like on ARM. If/then/else. If the particular thread does not



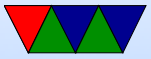
meet the condition, it just does nothing until the other condition finishes executing.

- If more threads are needed the available on the GPU, may need to break the problem up into smaller batches of threads.
- Once computing is done, copy results back to the CPU.



CPU vs GPU Programming Difference

- The biggest difference: NO LOOPS
- You essentially collapse your loop, and run all the loop iterations simultaneously.



Flow Control, Branches

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