ECE 574 – Cluster Computing
Lecture 24

Vince Weaver

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

vincent.weaver@maine.edu

4 November 2026

Announcements

- HW#8 due Friday, MPI followup
- Don't forget project topics due Friday
- Slurm/MPI on Haswell-EP is broken due to an OS upgrade gone wrong, I'll send an e-mail once I sort things out

Feedback on HW#6

- The biggest issue was scaling. Most people did not get better than 3x speedup even with the pragma for outside a loop
- My sample code I got up to 5x by reordering loops in addition
- Part of it is problem size and amdahl's law maybe? verify this

HW#6 – Results in General

- Ideally would have you measure multiple (10?) for each point and plot
- Know that's tedious
- Maybe print .csv and import into excel/sheets?
- perf actually can export .csv

HW#6 – Coarse Results

- In theory, just two sections, so should scale to two
- Moving to 4 probably won't increase performance

HW#6 – Fine Results

- In theory should scale decently up to 16 at least
- The biggest issue if not: you parallelized a loop with only three iterations (either the color loop or a matrix loop).
- OpenMP can't split up a loop of 3 iterations across 16 cores
- Ways to fix this:
  - re-arrange your loops so a big one is outer
  - try to use the collapsed loops support
  - you can re-write code to merge some of the loops together

HW#6 – Fine Results

- Even so you still might not scale well? Can vary on your code. Tracking down is the job of a performance engineer using tools like PAPI/perf. Can be a challenge.
- Also could be an issue of non-idle system

HW#6 – Why Print More Info

- Why did I have you print load/store time?
- Amdahl's law. Reduces overall potential speedup.
- Why does it vary so much? Not sure

HW#6 – Static vs Dynamic

- Setting up dynamic has a lot of overhead, and since our runs are quick and roughly same size shouldn't make a differenc
- Some people did find dynamic was better? Some much worse? Interesting. Again, analyzing why would be interesting.

HW#6 General Code Quality Feedback

- Be sure your code compiles, and that it doesn't crash
- OpenMP is supposed to be about taking existing correct linear code and dropping some pragmas in to make it parallel. You shouldn't have to majorly re-write your code.
  - If you are checking your thread-id and doing things based on it, it's probably not doing things properly
  - Will low-level messing about always work? (If you have 8 threads and you ask for 2, are you guaranteed they are 0 and 1?)

HW#6 More Comments

- If the homework says use sections, use sections. Don't use serial/tasks, or open-code your own sections implementation.
- Don't use nowait unless you know what you are doing
  - Great place for a code comment
  - "Because it crashes otherwise" is not a good reason

HW#6 Even More Comments

- Where you put your for (before d 0..2 or x 1..xsize), how it interacts with static vs dynamic
- Putting parallelization in inner loops instead of outer? Complex how this works? Implementation dependent? Probably not recommended but seems to work. Maxes out at total number of threads. Possibly some overhead for starting parallel over and over
- Reduction: you can use a reduction, but only if you are summing up results from a loop. So if you're using a loop to do the sums. Probably not worth it for the 9 additions in sobel

HW#6 Even More Comments 2

- Don't mix pthread and OpenMP code. It might work, but it's not necessary
- Let OpenMP set thread number for you, don't try to parse OMP_NUM_THREADS yourself

HW#6 OpenMP mysteries

- Cases the code sometimes works even if it shouldn't
- omp for nested inside of omp section. Technicall not allowed. Sometimes works, sometimes locks up.
- Use of private vars.
  - Loop indices are always treated as private
  - Not sure how some of the solutions worked without declaring sum to be private.
  - Not sure if it's compiler optimizations, or just luck

Computer 3D Graphics

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

- 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-casting / Ray-marching

- Hard to find good definition
- Ray-tracing the intersection of Ray with Object found with a single math calculation
- Ray-marching you estimate distances to objects and sort of “jump” to that location before doing more detailed check
- Ray-casting you sort of walk forward in steady increments until you hit objects

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

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

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 verteces 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 – Verte

- 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– Fragm

- 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)

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

GPGPU Motivation

- 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

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)

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)
  - ROCm (Radeon Open Compute Platform) (AMD)
  - OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc
  - OpenACC?
  - OneAPI (new attempt at CUDA replacement)
- Article on Future of AMD vs CUDA

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

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.

Programming a GPGPU (more details)

- 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