ECE 574 – Cluster Computing
Lecture 16

Vince Weaver

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

vincent.weaver@maine.edu

26 March 2019

Announcements

- HW#7 posted
- HW#6 and HW#5 returned
- Don't forget project topics due Thursday!
  Topics: memory/reliability on Pi? Launch in a balloon?
  Topics: GPU on Pi (OpenCL)

Notes on HW#5

- Were supposed to use sections directive for the coarse code

- Should parallelize your biggest loop, unless it is auto-collapsing, parallizing the colors loop of 0..3 won't scale very well
  This is why some were seeing bigger benefits of combine vs convolve

- Loop indices don't need to be marked as private, OpenMP assumes they are (so don't change their value outside the for statement)

Notes on MPI

- So many issues were C related. Fortran next time?
- Got MPI working on haswell-ep with mpich, needed to set `MpiDefault=pmi2` in `slurm.conf`

Notes on HW#6

- Biggest problem was calculating at different granularity to the gather
- Other problem was the final tail end of data to calculate in cases where not exact multiple of number or ranks.

Pi cluster

- 1 head node (16GB SD card), 24 sub-nodes. One currently seems to be down (reliability!)
- Read up on the cluster here:
  `https://www.mdpi.com/2079-9292/5/4/61/htm`
- Added your accounts, same password as haswell-ep (via hashes)
- Try not to use up too much disk space
- Also note the SD card is sorta slow, which with the network affects scaling a bit.

- Use slurm
- The batch scripts I gave you have a timeout of 5 minutes per job. Last time some people's code went crazy and ran forever and other people's jobs never ran
- Use `sinfo` or `squeue` to see cluster and job stats
- Use `scancel` to cancel a job
- If things going poorly, contact me
- Did update PAPI on all nodes which should be working

MPI and slurm

- HW `#SBATCH --tasks-per-node=4`
- -N = number of nodes
- -n = number of tasks, default is one task per node?
- N=4 tasks-per-node=4, 16
  N=4 tasks-per-node=4, sbatch -n 8, 16 (N=nodes, n=tasks)
  N=4 tasks-per-node=4, sbatch -N 8, 32

nothing, sbatch -N 8, 32

nothing, sbatch -n 8, (8, 2 nodes * 4 each)

nothing, sbatch -N 8 -n 8 (8, 8 nodes * 1 each)

Why use slurm?

- Can set account to charge
- Can handle checkpointing
- Can set constraints (run on machine with gpu, certain proc type)
- Contiguous allocations
- CPU freq, power capping
- Licenses avail (things like Matlab etc)
- Memory avail

Graphics Processing Units

- Retrospective on old graphics hardware
- Framebuffer is simple (though annoying pointer match like in sobel or worse). VGA Mode 13h, 0xa0000, 64kB
- Old video game systems didn't even have that. Why? 1MB for a framebuffer was expensive. Only 64k RAM total.
- Atari 2600 only had 128B of RAM, total. 40-bit framebuffer. Racing the beam.
- Also could do sprites or tile based.

GPUs

Interfaces

- Originally each vendor had own 3D interface, SGI standardized
- OpenGL – SGI
- Direct3D – Microsoft
- Vulkan – new interface with less baggage
- WebGL?
- Originally for HPC/CAD but gaming has brought down prices for everyone.

GPGPUS

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

Why GPUs?

- Old example:
  - 3GHz Pentium 4, 6 GFLOPS, 6GB/sec peak
  - GeForceFX 6800: 53GFLOPS, 34GB/sec peak
- Newer example
  - Raspberry Pi, 700MHz, 0.177 GFLOPS
  - On-board GPU: Video Core IV: 24 GFLOPS

GPGPU 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

GPU Benefits

- Specialized hardware, concentrating on arithmetic. Transistors for ALUs not cache.
- Fast 32-bit floating point (16-bit?)
- 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 Problems

- 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 with a lot of control flow runs poorly
- Off-chip memory transfers can be slow

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

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.

Older / Traditional GPU Pipeline

- 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

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

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

Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now

Shader 3.0 Programming – Vertex Processor

- 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)

Shader 3.0 Programming – 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)