ECE 571 – Advanced Microprocessor-Based Design Lecture 19

Vince Weaver

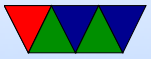
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

13 November 2014

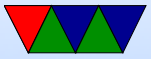
Announcements

- No class Tuesday
- Assign a homework with two papers to read to do instead, also feel free to work on your projects



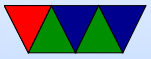
HW4 Review

-



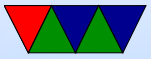
GPUs

- A lot of this info from the "GPU Gems" book available from Nvidia
- 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-latench for one



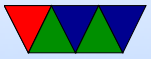
GPGPUs

- Started when the vertex and fragment processors became generically programmable (originally to allow more advanced shading and lighting calculations)



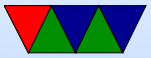
Graphics vs Programmable Use

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-alia	Data	Predicated Write



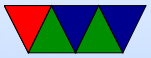
Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now

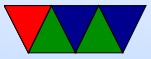


Shader 3.0 Programming – Vertex Processor

- 512 static / 65536 dynamic instructions
- Up to 32 temporary registers
- Simple flow control
- Texturing – texture data can be fetched during vertex operations

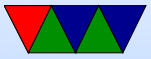


- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:
  - EXP, EXPP, LIT, LOGP (exponential)
  - RCP, RSQ (reciprocal, r-square-root)
  - SIN, COS (trig)



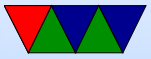
Shader 3.0 Programming – Fragment Processor

- 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-wite DP4 (dot product)



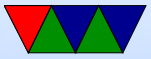
GPGPUS

- 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
  - OpenACC?

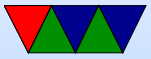


Program

- 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. Fraagment 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 (quadrilatteral)
- Texture-coordinates = Domain
- Vertex-coordinates = Range



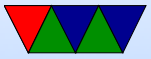
Kernel

Kernel is like the body of a for loop.

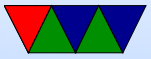
```
void do_something_1000x1000_grid(float in[1000][1

int x,y;

for (x = 0; x < 1000; x++) {
for (y = 0; y < 1000; y++) {
// next bit run 1000x1000 times, in parallel
out[x][y] = do_something(in[x][y]);
```

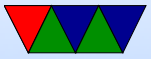


```
            }
        }
    }
```



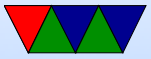
Flow Control, Branches

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

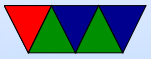


GPU Tasks

- Map – apply a kernel to each element of an array. For actual graphics, take each pixel and calculate what it should look like.
- Reduce – calculate a smaller stream from a larger one. After enough steps only one is left. For a 2D reduction (a grid) take 4 and make 1.
- Stream filtering – non-uniform reduction
- Scatter – send results to other processing elements

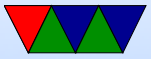


- Gather – gather results from other processing elements
- Sort
- Search – search for an item. Not necessarily faster, but can be done in parallel

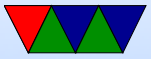


Writing code (CUDA)

- Init GPU
- Spcify kernel (in Cg C-like language)
- Load the program
- Set up input/output buffers
- Set the domain and range
- Set up constants

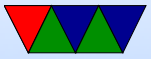


- Invoke the kernel
- Read data back from GPU (slow)



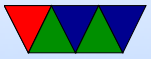
Terminology (CUDA)

- 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 groupd of thread blocks that need to finish before next can be started



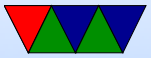
Terminology (cores)

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



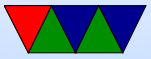
Power Angle

- Very high performance
- Simpler cores
- No wasted time on wrong-branches, control flow, cache.



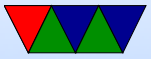
Other Implementations

- Xeon-Phi – massive array of full x86 cores. Larabee, etc.



Measuing Power/Performance

- Many chips have hardware performance counters. Not easy to get to. perf does not support (though patches for Intel GPU floating around).
- In general, can do aggregate but not profiling. Start when data submitted to GPU, only get results when task finished.
- In CUDA there are counters, PAPI supports them
- Also Nvidia has NVML which can provide some power



measurement.