# ECE 574 – Cluster Computing
# Lecture 16

Vince Weaver

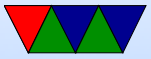
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

29 October 2015

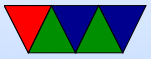
# Announcements

- HW#6 Will be posted still
  Delayed by cluster work



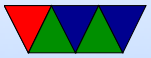
# HW#5 Review Continued

- Took a look at fine-grained with and without "sum" marked as shared using perf. cache-misses not much different. Big difference in "LLC-stores": fast case = 61M, slow case = 661M



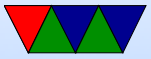
# GPGPU Key Ideas

- Using many slimmed down cores
- Have single instruction stream operate across many cores (SIMD)
- A void latency (slow textures, etc) by working on another group when one stalls

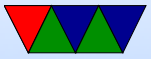


# GPU Benefits

- 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.

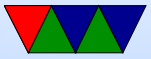


- highly parallel



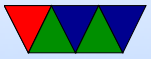
# GPU Problems

- 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



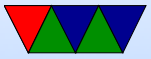
# Latency vs Throughput

- 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



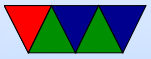
# Older / Traditional GPU Pipeline

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

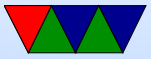


# Older / Traditional GPU Pipeline

- 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

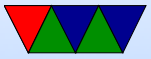


- Shader (Fragment processor) compute color for each pixel. Use textures if necessary (texture memory, mostly read)
- Write out to framebuffer (mostly write)



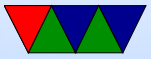
# GPGPUs

- 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



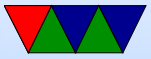
# Graphics vs Programmable Use

| | | | |
|---|---|---|---|
| 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 |



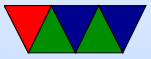
# Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now

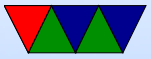


# Shader 3.0 Programming – Vertex Processor

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

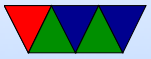


- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:
  - EXP, EXPP, LIT, LOGP (exponential)
  - RCP, RSQ (reciprocal, r-square-root)
  - SIN, COS (trig)



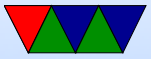
# Shader 3.0 Programming – Fragment Processor

- 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)



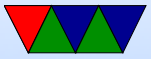
# GPGPUs

- 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?

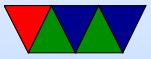


# Program

- 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. 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

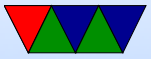


- Kernels == inner loops
- Render-to-texture == feedback
- Geometry-rasterization == computation. Usually done as a simple grid (quadrilateral)
- Texture-coordinates = Domain
- Vertex-coordinates = Range



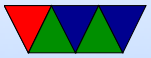
# Flow Control, Branches

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



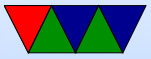
# Terminology (CUDA)

- 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



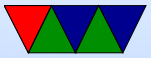
# Terminology (cores)

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

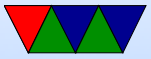


# CUDA Programming

- Since 2007
- Use `nvcc` to compile
- *host* vs *device*
  host code runs on CPU
  device code runs on GPU
- Host code compiled by host compiler (gcc), device code by custom NVidia compiler

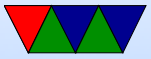


- `__global__` parameters to function – means pass to CUDA compiler
- cudaMalloc() to allocate memory and pointers that can be passed in
- call global function like this `add<<<1,1>>>(args)` where first inside brackets is number of blocks, second is threads per block
- cudaFree() at the end
- Can get block number with blockIdx.x and thread index



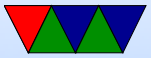
with threadIdx.x

- Can have 65536 blocks and 512 threads (At least in 2010)
- Why threads vs blocks?
  Shared memory, block specific
  `__shared__` to specify
- `__syncthreads()` is a barrier to make sure all threads finish before continuing

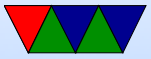


# Code Example

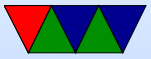
```
#include <stdio.h>

#define N 10


__global__ void add (int *a, int *b, int *c) {
    int tid=blockIdx.x;

    if (tid<N) {
        c[tid]=a[tid]+b[tid];
    }
}


int main(int arc, char **argv) {

    int a[N],b[N],c[N];
    int *dev_a,*dev_b,*dev_c;
    int i;

    /* Allocate memory on GPU */
```



```
cudaMalloc((void **)&dev_a,N*sizeof(int));
cudaMalloc((void **)&dev_b,N*sizeof(int));
cudaMalloc((void **)&dev_c,N*sizeof(int));

/* Fill the host arrays with values */
for(i=0;i<N;i++) {
    a[i]=-i;
    b[i]=i*i;
}

cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);

add<<<N,1>>>(dev_a,dev_b,dev_c);

cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

/* results */
for(i=0;i<N;i++) {
    printf("%d+%d=%d\n",a[i],b[i],c[i]);
}

cudaFree(dev_a);
cudaFree(dev_b);
```

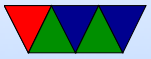


```
    cudaFree(dev_c);

    return 0;
}
```



# OpenCL

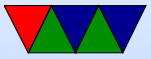
similar to Cuda at least conceptually



# Other Accelerator Options

- XeonPhi – came out of the larabee design (effort to do a GPU powered by x86 chips). Large array of x86 chips(p5 class on older models, atom on newer) on PCIe card. Sort of like a plug-in mini cluster. Runs Linux, can ssh into the boards over PCIe. Benefit: can use existing x86 programming tools and knowledge.

- FPGA – can have FPGA accelerator. Only worthwhile if you don't plan to reprogram it much as time delay in reprogramming. Also requires special compiler support



(OpenMP?)

- ASIC – can have hard-coded custom hardware for acceleration. Expensive. Found in BitCoin mining?
- DSPs – can be used as accelerators