

# ECE 574 – Cluster Computing

## Lecture 17

Vince Weaver

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

`vincent.weaver@maine.edu`

28 March 2019

# Announcements

- HW#8 (CUDA) posted.
- Project topics due.



# 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)



# GPU hardware for the class

- NVIDIA Quadro hardware – workstation rather than gaming
  - Higher reliability FP? ECC RAM (maybe?)
- 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
- NVIDIA Quadro K2200 in Quadro
  - 4GB GDDR5, 128-bitm 80 GB/s



- 640 cores, Maxwell architecture
- 68W, OpenGL 4.5, DirectX 11.2



# Programming a GPGPU

- 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 than 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.





# GPU vs GPGPU concepts

- Only handle fixed-point or floating point values
- Can use textures as arrays. Typically only 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.
- 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



# NVIDIA Terminology (CUDA)

- Thread: chunk of code running on GPU.
- Warp: group of thread running at same time in parallel simultaneously  
AMD calls this a “wavefront”
- 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 2006
- Use `nvcc` to compile
- `.cu` files. Note, technically C++ so watch for things like `new`
- `*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





# CUDA Programming

- See the NVIDIA “CUDA C Programming Guide”
- Compute Unified Device Architecture
- From CUDA C Programming guide from NVIDIA
- CUDA introduced in 2006
- Heterogeneous programming – there is a host executing a main body of code (a CPU) and it dispatches code to run on a device (a GPU)
- CUDA assumes host and device each have own separate DRAM memory



(newer cards can share address space via VM tricks)

- CUDA C extends C, define C functions "kernels" that are executed N times in parallel by N CUDA threads



# CUDA Debugging

- Can download special cuda-gdb from NVIDIA
- Plain printf debugging doesn't really work



# CUDA Coding

- version compliance – can check version number. New versions support more hardware but sometimes drop old
- nvcc – wrapper around gcc. global code compiled into PTX (parallel thread execution) ISA
- can code in PTX code directly which is sort of like assembly language. Won't give out *actual* assembly language. Why?
- CUDA C has mix of host and device code. Compiles the global stuff to PTX, compiles the <<< ... >>> into



code that can launch the GPU code

- PTX code is JIT compiled into native by the device driver
- You can control JIT with environment variables
- Only subset of C/C++ supported in the device code



# CUDA Hardware

- GPU is array of Streaming Multiprocessors (SMs)
- Program partitioned into blocks of threads. Blocks execute independently from each other.
- Manages/Schedules/Executes threads in groups of 32 parallel threads (warps) (weaving terminology) (no relation)
- Threads have own PC, registers, etc, and can execute independently
- When SM given thread block, partitions to warps and



each warp gets scheduled

- One common instruction at a time. If diverge in control flow, each way executed and thread not taking that path just waits.
- Full context stored with each warp; if warp is not ready (waiting for memory) then it may be stopped and another warp that's ready can be run



# CUDA Threads

- kernel defined using `__global__` declaration. When called use `<<<...>>>` to specify number of threads
- each thread that is called is assigned a unique ThreadID  
Use `threadIdx` to find what thread you are and act accordingly

```
__global__ void VecAdd(float *A, float *B, float *C) {  
    int i = threadIdx.x;  
  
    if (i<N)          // don't execute out of bounds  
        C[i]=A[i]+B[i];  
}
```





```

int main(int argc, char **argv) {
    ....
    /* Invoke N threads */
    VecAdd<<<1,N>>>(A,B,C);
}

```

- threadIdx is 3-component vector, can be seen as 1, 2 or 3 dimensional block of threads (thread block)
- Much like our sobel code, can look as 1D (just x), 2D, (thread iD is  $((y * \text{xsize}) + x)$  or  $(z * \text{xsize} * \text{ysize}) + y * \text{xsize} + x$ )
- Weird syntax for doing 2 or 3d.

```

__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i=threadIdx.x;
    int j=threadIdx.y;
    C[i][j]=A[i][j]+B[i][j];
}

```



```
}  
  
int numBlocks=1;  
dim3 threadsPerBlock(N,N);  
MatAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);
```

- Each block made up of the threads. Can have multiple levels of blocks too, can get block number with blockIdx
- Thread blocks operate independently, in any order. That way can be scheduled across arbitrary number of cores (depends how fancy your GPU is)



# CUDA Memory

- Per-thread private local memory
- Shared memory visible to whole block (lifetime of block)  
Is like a scratchpad, also faster
- Global memory
- also constant and texture spaces. Have special rules.  
Texture can do some filtering and stuff
- Global, constant, and texture persistent across kernel launches by same app.



# More Coding

- No explicit initialization, done automatically first time you do something (keep in mind if timing)
- Global Memory: linear or arrays.
  - Arrays are textures
  - Linear arrays are allocated with `cudaMalloc()`, `cudaFree()`
  - To transfer use `cudaMemcpy()`
  - Also can be allocated `cudaMallocPitch()` `cudaMalloc3D()` for alignment reasons



- Access by symbol (?)
- Shared memory, `__shared__`. Faster than Global also `__device__`

Manually break your problem into smaller sizes



# Misc

- Can lock host memory with `cudaHostAlloc()`. Pinned, can't be paged out. Can load store while kernel running if case. Only so much available. Can be marked `writecombining`. Not cached. So slow for host to read (should only write) but speeds up PCI transaction.



# Async Concurrent Execution

- Instead of serial/parallel/serial/parallel model
- Want to have CUDA running and host at same time, or with mem transfers at same time
  - Concurrent host/device: calls are async and return to host before device done
  - Concurrent kernel execution: newer devices can run multiple kernels at once. Problem if use lots of memory
  - Overlap of Data Transfer and Kernel execution
  - Streams: sequence of commands that execute in order,



but can be interleaved with other streams  
complicated way to set them up. Synchronization and  
callbacks





# Events

- Can create performance events to monitor timing
- PAPI can read out performance counters on some boards
- Often it's for a full synchronous stream, can't get values mid-operation
- NVML can measure power and temp on some boards?



# Multi-device system

- Can switch between active device
- More advanced systems can access each others device memory



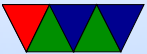
# Other features

- Unified virtual address space (64 bit machines)
- Interprocess communication
- Error checking



# Texture Memory

- Complex



# 3D Interop

- Can make results go to an OpenGL or Direct3D buffer
- Can then use CUDA results in your graphics program



# Code Example

```
#include <stdio.h>

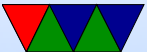
#define N 10

__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 argc, 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;
}

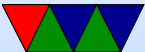
cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

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;  
}
```





# Code Examples

- Go through examples
- Also show off `nvidia-smi`



# CUDA Notes

- Nicely, we can use only block/thread for our results, even on biggest files
- In past there was a limit of 64k blocks with “compute version 2” but on “compute version 3” we can have up to 2 billion



# CUDA Examples

- Make builds them. .cu file, built with nvcc
- ./hello\_world bit of a silly example
- saxpy.c  
single  $a*x+y$   
CPU GPU run 320000000 1.12s 2.06
- What happen if thread count too high? Max threads per block 512 on Compute 2, 1024 on compute 3 Above



1024? Try saxpy\_block

- maximum block size 64k on Compute version 2, 2GB on Compute Version 3 200,000 50,000 cpu = 4.5s gpu = 0.8s



# CUDA Tools

- `nvidia-smi`. Various options. Usage, power usage, etc.
- `nvprof ./hello_world` profiling
- `nvvp` visual profiler, can't run over text console
- `nvidia-smi --query-gpu=utilization.gpu,power.draw --format=csv -lms 100`

