ECE 574 – Cluster Computing
Lecture 25

Vince Weaver

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

vincent.weaver@maine.edu

6 November 2026

Announcements

- HW#8 due Today
- Note I managed to mess up slurm/MPI (PMIX) on haswell-ep. Sent e-mail how to use mpi manually to finish it. Long story, but in theory kernel update and NVIDIA driver working?
  We won't really be using slurm rest of semester, though if you need for project let me know and I can spend more time on it.
- Don't forget project topics are due! Please send if you haven't
- HW#9 (CUDA) will be posted next week sometime

GPUs in HPC

- Widely used in Top500
- NVIDIA
  - A100 / H100 – large expensive GPUs
- AMD
  - Instinct

NVIDIA GPUs

- We'll look at NVIDIA GPUs first

NVIDIA Generations / Compute Capability

- Tesla/Fermi (deprecated)
- 3: Kepler
- 5: Maxwell
- 6: Pascal
- 7: Volta (pro) / 7.5: Turing (consumer)
- 8: Ampere
- 9: Lovelace/Hopper
- 10: Blackwell (optimized for AI)
- Feynman

Specs for Blackwell GPUs

- Named for David Blackwell, mathematician
- Successor to Hopper, 5x performance (at 4 bit fp)
- 20 PFLOPs (at 4 bit fp), 300W? B100?
- Two dies, 10TB/s NVLINK, 8 HBM3e stacks (192GB)
- Tower of them get get over an exaflop (a lot of power, and again 4 bit fp, for AI)

NVIDIA Workstation vs Gaming

- Quadro (Workstation) vs Geforce (Gaming) (note from 2023, they renamed these)
  - Quadro generally more RAM. higher Bus width
  - Fancier Drivers
  - Optimized for CAD type stuff and compute, rather than games
  - Higher reliability
  - Quadro better support for double-precision floats
  - More compute cores
  - Power limits

GPU hardware in my Lab (better)

- Can use these for projects. I mostly get these for power measurement tests.
- NVIDIA RTX A2000 in Skylake
  - 6GB GDDR6, 192-bit, 288 GB/s
  - Ampere, PCIe4x16
  - 3328 CUDA cores, 104 tensor cores, 26 RT cores
  - 8 TFLOPS single-precision, RT 15.6 TFLOPS, Tensor 64 TFLOPS
  - 70W, DirectX 12.07, Vulkan 1.2
- NVIDIA RTX A2000 GA106 in RaptorLake
  - Roughly same as above but with 12GB RAM

GPU hardware in my Lab (mid)

- NVIDIA Quadro P2000 in Skylake (old)
  - 5GB GDDR5, 160-bit, 140 GB/s
  - 1024 cores, Pascal, PCIe3x16
  - 75W, DirectX 12.0, Vulkan 1.0
- 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
  - Low-power for server, as runs in 1U rack

GPU hardware in my Lab (old)

- NVIDIA Quadro K2200 in Quadro
  - So old the drivers don't want to support it anymore
  - 4GB GDDR5, 128-bit 80 GB/s
  - 640 cores, Maxwell architecture
  - 68W, OpenGL 4.5, DirectX 11.2

CUDA Programming

- Since 2006
- Compute Unified Device Architecture
- See the NVIDIA "CUDA C Programming Guide"
- Use `nvcc` to compile
- .cu files. Note, technically C++ so watch for things like `new`

Useful CUDA Programming Reference

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/

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)
- Sometimes have issues where drivers won't install (currently having that issue on some of my machines)

NVIDIA Terminology (CUDA)

- Thread: chunk of code running on GPU.
- Warp: group of threads 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
- Streams?

Terminology (cores)

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

Notes from CUDA document

- Designed to be simple
- Three abstractions
  - hierarchy of thread groups
  - shared memories,
  - barrier synchronization
- Threads can be run in any order on any number of cores, the programmer doesn't have to worry about how many cores there are

CUDA Programming

- 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 Programming – Host vs Device

- *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

CUDA Compiling

- `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?
- PTX code is JIT compiled into native by the device driver
- You can control JIT with environment variables

CUDA Compiling – Version Compliance

- Various command line options, some do things like specify the "compute version" (GPU generation) to target
- version compliance – can check version number. New versions support more hardware but sometimes drop old

CUDA Programming – Overview

- Special kernel `__global__` function that runs on GPU limited what you can run there
- Only subset of C/C++ supported in the device code, compiled to PTX
- Special call with angle brackets to run in parallel `VecAdd<<<1,N>>>(A,B,C)`
- The kernel is run simultaneously on N different threads
- To get data on GPU need to cudaMalloc() it and then cudaMemcpy() there
- When done, need to cudaMemcpy() back

CUDA Programming – Memory Allocation

- `cudaMalloc()` allocates memory on the device
  `cudaMalloc((void **)&dev_a,N*sizeof(int));`
- `cudaFree()` to free when done
- `cudaMemcpy(dev_a,a,N*sizeof(int),`
  `cudaMemcpyHostToDevice);`
- `cudaMemcpy(c,dev_c,N*sizeof(int),`
  `cudaMemcpyDeviceToHost);`

CUDA Programming – Pointers

- Note: result of a `cudaMalloc()` might look like a pointer, but it's not
- You can't dereference memory allocated with `cudaMalloc()` on the CPU, the memory area is completely separate
- There is work on newer GPUs allowing unified CPU/GPU memory but we're going to assume that's not available

CUDA Programming – 2D/3D Arrays

- Can use `cudaMallocPitch()` and `cudaMalloc3D()`
- These will do proper padding/alignment when using 2d/3d arrays
- Have own `cudaMemcpy2D()` and `cudaMemcpy3D()`

CUDA Hardware – this might be dated

- 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

CUDA Hardware – continued

- 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

Simple CUDA Kernel Example

```
__global__ void VecAdd(float *A, float *B, float *C)
        int i = threadIdx.x;

        if (i<N)                    // don't execute ou
                C[i]=A[i]+B[i];
}

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