# CS 5220

## Accelerators

David Bindel

2024-10-08

# Architecture

# Pre-history

- GPU = Graphics Processing Units
- GPGPU = General Purpose GPU programming
  - Abuse shaders to do matrix multiply
  - Ignore the actual graphics output
  - Horrible code, but fast!
- Misc projects to systematize GPGPU (e.g. Brook)
- 2007: NVidia launches CUDA

# CPU vs GPU

Different fundamental goals

- CPU: Minimize latency
- GPU: Maximize throughput

Graphics (and ML) are high throughput!

# CPU reminder

Five-stage RISC basically has

- Fetch/decode
- Execute (ALU)
- Execution context (memory, register)

# CPU reminder

Modern machines have pipeline *and*

- Out-of-order engine
- Fancy branch prediction
- Memory pre-fetcher
- Large and complex data cache

Mostly to find parallelism and reduce latency.

# CPU to GPU

Simplify, simplify, simplify!

- Forget about making a *single* instruction stream run fast.
- Out-of-order, branch prediction, etc take space and energy
- What about more compute resources in simpler cores?
- Focus on data parallel operation for performance

# CPU to GPU

Simplify: Fetch/decode

- Share one fetch/decode across *warp* of threads
- SIMT: Single Instruction, Multiple Threads
- Each thread has own ALU and execution context
  - plus some shared context
- Result: more ALUs, fewer decoders

Simplify: Branches

```
if (x > 0)
    do_something();
else
    do_something_else();
```

Simplify: Branches

```
T1: do_something();       // Only if x > 0
T2: do_something_else(); // Only if x <= 0
```

- Process *both branches* sequentially
- Use *masking* to turn off thread on wrong branch
- “Control divergence” reduces performance

# CPU to GPU

Simplify: Latency tolerance

- Particular issue: *memory* latency (100s-1000s cycles)
- CPU solution: caches, prefetch, out-of-order
- GPU solution: more threads!
    - When one warp stalls, run another warp

# GPU architecture

- Array of *streaming multiprocessors* (SMs)
- Each SM has several CUDA cores
    - Cores share logic and control
- Ex: Ampere A100 GPU
    - 108 SMs/GPU
    - 64 cores/SM
    - 32 threads/warp
    - 6912 cores/GPU, 221184-way parallel

# Thread blocks

- Threads are organized into *blocks*
- Assign threads to SMs block-by-block
  - Can assign more than one block per SM
- Schedule blocks in units of warps
- Assignment to one SM enables
  - Barrier synchronization
  - Fast shared memory
- CUDA Occupancy Calculator
  - Checks if partitioning is HW appropriate

# Memory

- Global memory and constant memory
  - Constant memory only written by host
- Local memory (per thread)
  - Actually lives in global memory
- On-chip registers
  - Private per thread
  - Many registers for cheap context switch
- On-chip shared memory
  - Used to communicate in block

# Memory

No caches, but still:

- Global: large, but slow
- Shared: small, but fast

Therefore:

- Use tiling/blocking (as on CPU)
- Try to have threads access sequential global locations
  - HW *coalesces* such accesses – one DRAM burst
  - This is about how DRAM works, not cache lines!

# Optimizations

(From PMPP book)

- Maximize occupancy (tune use of SM resources)
- Enable coalesced global accesses (analogue of unit stride)
- Minimize control divergence (data/thread layout rearrangement)
- Tiling (as you know!)
- Privatization (work on local copies)
- Thread coarsening (reduce scheduling overhead)

Much is not so dissimilar to multicore CPU!

# Punchline

CPU

- Faster sequential execution
- More latency-reducing features
- Cache hierarchy
- More architectural complexity
- More energy/flop

GPU

- More parallelism
- More compute HW
- High BW on-chip SRAM + global DRAM
- Simplified core architecture
- Less energy/flop

# Punchline

GPUs are great for

- Graphics
- Large matrix computations
- Neural networks (and other ML)

... but you still want a CPU, too!

# Perlmutter setup

# Top level view

GPU nodes have:

- 1 AMD EPYC 7763 (Milan) CPU
- 64 CPU cores
- 256 GB of DDR4 DRAM
- 4 NVidia A100 (Ampere) Tensor Core GPUs per node with NVlink interconnect
    - Still just a PCI bus between CPU and GPU
- 40 GB high-bandwidth memory (HBM)/GPU

# A100 SM


Each SM has

- 64 FP32 cores
- 32 FP64 cores
- 64 KB registers
- 192 KB L1/shared memory
- Four Tensor Cores

# Tensor Cores

- Support matrix operations
- Mixed numeric data types
  - Ex: TF32 format with 10 bit mantissa (same as FP16)
  - Can serve as intermediate (FP32 format in/out)
  - Will return to floating point issues later in class

# Heterogeneous computing

# Basic model

CPU code calls GPU *kernels*

1. First, allocate memory on GPU
2. Copy data to GPU
3. Execute GPU program
4. Wait for completion
5. Copy results back to CPU

# Programming approaches

- CUDA
  - NVidia-only (HIP, etc are not)
  - “First mover” advantage
  - We will start here
- OpenMP
- Thrust, SYCL, Kokkos
- Halide, Taichi, etc

# Programming approaches

- CUDA
- OpenMP
  - Compiler-directive based
  - Largely subsuming OpenACC?
  - Plays nice with rest of OpenMP
- Thrust, SYCL, Kokkos
- Halide, Taichi, etc

# Programming approaches

- CUDA
- OpenMP
- Thrust, SYCL, Kokkos
  - Somewhat higher-level C++
  - Will not cover, but good for projects!
- Halide, Taichi, etc

# Programming approaches

- CUDA
- OpenMP
- Thrust, SYCL, Kokkos
- **Halide, Taichi, etc**
  - Specialized domain-specific languages
  - Raise the abstraction level
  - Also think JAX, etc
  - Will not cover, but good for projects!

# Vector addition

"Hello world": vector addition in CUDA

```
#include <iostream>
#include <vector>
#include <cassert>

// Compute y += x
void add(const std::vector<float>& x, std::vector<float>& y);

int main()
{
    int N = 1<<20;
    std::vector<float> x(N), y(N);
    for (auto& xi : x) xi = 1.0f;
    for (auto& yi : y) yi = 2.0f;
    add(x, y);
```

# Serial version

Serial version of add:

```
void add(const std::vector<float>& x, std::vector<float>& y)
{
    int n = x.size();
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}
```

# Reminder of basic model

CPU code calls GPU *kernels*

1. First, allocate memory on GPU
2. Copy data to GPU
3. Execute GPU program
4. Wait for completion
5. Copy results back to CPU

# CUDA kernel

```
__global__
void gpu_add(int n, float* x, float* y)
{
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}
```

- Use `__global__` for host-callable kernel on GPU
- Can also declare `__device__` or `__host__`
- This version is *not* parallel

# Device malloc

First, allocate memory on GPU:

```
int size = n * sizeof(float);
float *d_x, *d_y;
cudaMalloc((void**)& d_x, size);
cudaMalloc((void**)& d_y, size);
```

- Signature: `cudaMalloc(void** p, size_t size);`
- Resulting `p` refers to a *device* (global) memory location
- CUDA interface is C style (Thrust has device `vector`, for ex)
- Matching `cudaFree(void* p)` to free

# Device malloc

Should probably be more careful:

```
cudaError_t err = cudaMalloc((void**)& d_x, size);
if (err != cudaSuccess) {
    printf("%s in %s at line %d", cudaGetErrorString(err),
           __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

# Transfer to GPU

```
// cudaMemcpy(void* dest, void* src, size_t size, int directi
cudaMemcpy(d_x, x.data(), size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y.data(), size, cudaMemcpyHostToDevice);
```

- Use `data` for C pointer to `vector` storage
- Need matching copy back of `y` at end

# GPU call

```
gpu_add<<<1,1>(n, d_x, d_y);
```

- Call the kernel on GPU with 1 block, 1 thread
- Need more for parallelism!
- Will revisit this

# Transfer to host

```
cudaMemcpy(y.data(), d_y, size, cudaMemcpyDeviceToHost);
```

- This time we move data the other way
- No need to copy `x` back (only `y` updated)

# Full code

```
void add(const std::vector<float>& x, std::vector<float>& y)
{
    int n = x.size();

    // Allocate GPU buffers and transfer data in
    int size = n * sizeof(float);
    float *d_x, *d_y;
    cudaMalloc((void**)& d_x, size); cudaMalloc((void**)& d_y
    cudaMemcpy(d_x, x.data(), size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y.data(), size, cudaMemcpyHostToDevice);

    // Call kernel on the GPU (1 block, 1 thread)
    gpu_add<<<1,1>>>(n, d_x, d_y);

    // Copy data back and free GPU memory
```

# What's wrong here?

- Realistic: work doesn't justify memory transfer!
- Also: GPU is getting used as a serial device
- Also: The whole `malloc`/`free`/`memcpy` thing is not very C++!

# Kernel take 2

```
__global__
void gpu_add(int n, float* x, float* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
        y[i] += x[i]
}

// Call looks like
gpu_add<<<n_blocks,block_size>>>(n, x, y);
```

- Each thread handles one entry
- Use a 1D block layout
- Each block has `blockDim.x` threads
- Need at least `n` total threads

# Teaser: Thrust

```
int add(std::vector<float>& x, std::vector<float>& y)
{
    thrust::device_vector<float> x_d {x};
    thrust::device_vector<float> y_d {y};
    using namespace thrust::placeholders;
    thrust::transform(x_d.begin(), x_d.end(),
                      y_d.begin(), y_d.begin(),
                      _1 + _2);
    thrust::copy(y_d.begin(), y_d.end(), y.begin());
}
```

# Teaser: OpenMP

```
void gpu_add(int n, float* x, float* y)
{
    #pragma omp target map(to:x[0:n]) \
            map(tofrom:y[0:n])
    #pragma omp parallel for simd
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}
```

# Next time

- A little beyond “hello, world”
- Using the GPU with OpenMP