A High-Level Intro to CUDA

CS5220 Fall 2015
What is CUDA?

- **Compute Unified Device Architecture**
  - released in 2007
  - GPU Computing

- **Extension of C/C++**
  - requires NVCC (CUDA Compiler) and NVIDIA Graphics Card
Historical Background

● In the early days, no “GPUs”. Expensive computers had tiny math co-processors.
  ○ intersecting and transforming vectors, basic physics, textures, etc
  ○ The earliest games took advantage of these co-processors.
● Hardware changes!
  ○ Numerous vendors at first
  ○ now only NVIDIA and AMD (ATI)
● Not surprisingly, graphics cards were a great way to compute!
  ○ Simulations, Machine Learning, Signal Processing, etc etc
● Nowadays, GPUs are often the most expensive part of a computer
The Difference Between (Modern) CPUs and GPUs

- Starting Question: When would I use a CPU and when would I use a GPU?
- So far in this class, we’ve been using ~24 threads (~240 with offloading)
  - Need to find much more parallelism per GPU!
  - Think thousands of threads...
Current CPU Architecture
Current GPU Architecture

- PCI Express 3.0 Host Interface
- GigaThread Engine
- SMX
- Memory Controller
- L2 Cache
Let’s look a bit closer...
GPU Architecture

- Major Simplification: you can think of a GPU as a big set of vector (SIMD) units.
  - Programming with this model in mind won’t give you the best performance, but it’s a start
- A better view is thinking of a GPU as a set of multithreaded, multicore vector units.
  - see “Benchmarking GPUs for Dense Linear Algebra, Volkov and Demmel, 2008”
- These models abstract the architecture in various ways!
Side Discussion

- What are the differences between a GPU and a Xeon Phi (the latter of which we’ve been using?)
Heterogeneous Parallel Computing

Host: the CPU and its memory

Device: the GPU and its memory
Advantages of Heterogeneous Processing

- Use both the CPU and GPU
- You get the best of both worlds!
  - Do serial parts fast with CPU, do parallel parts fast with GPU
- How does this extend to larger computers?
  - Many of the fastest supercomputers are essentially sets of CPUs with attached GPU Accelerators, à la Totient (more unusual back in the day)
What is CUDA?

- An API (Application Program Interface) for general Heterogeneous Computing
  - before CUDA, one had to repurpose graphics-specific APIs for non-graphics work
  - Major headache
The Crux of CUDA

- Work on the host (CPU), copy data to the device’s memory (GPU RAM), where it will work on that data
- Device then copies data back to the host
- As with CPU programming, communication and synchronization are expensive!
  - Even more so with the GPU (information has to go through PCI-E bus)
  - You do not want to be constantly copying over small pieces of work.
A General Outline

do_something_on_host();
kernel<<<nBlk, nThd>>>(args);
cudaDeviceSynchronize();
do_something_else_on_host();
Example: Vector Addition

```c
__global__ void VecAdd(const float* A, const float* B, float* C, int N) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
}
```
CUDA Features: What you can do

- **Standard Math Functions** (think cmath.h)
  - trig, sqrt, pow, exp, etc

- **Atomic operations**
  - atomicAdd, atomicMin, etc
  - As with before, much faster than locks

- **Memory**
  - cudaMalloc, cudaFree
  - cudaMemcpy

- **Graphics**
  - Not in the scope of this class, lots of graphics stuff
What you can’t do:

- In Vanilla CUDA, not much else
  - no I/O, no recursion, limited object support, etc
- This is why we need heterogeneity.
CUDA Function Declarations

__global__
- Kernel function (must return void)
- Executed in parallel on device

__host__
- Called and executed on host

__device__
- Called and executed on device
__global__ void VecAdd(const float* A, const float* B, float* C, int N) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < N)
}
void main() {
    float *h_A, *h_B, *h_C; // host copies of a, b, c
    float *d_A, *d_B, *d_C; // device copies of a, b, c
    int size = N * sizeof(float);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Alloc space for host copies of a, b, c and setup input values
    h_A = (int*)malloc(size); random_ints(h_A, N);
    h_B = (int*)malloc(size); random_ints(h_B, N);
    h_C = (int*)malloc(size);
}
Vector Addition Cont.

// Copy inputs to device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Launch VecAdd() kernel on GPU
int Nblocks = (N + 255)/256;
int Nthreads = 256;
VecAdd<<<Nblocks, Nthreads>>>(d_A, d_B, d_C, N); //←---- Note the <<<blocksPerGrid, 256>>>

// Copy result back to host
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Cleanup
free(h_A); free(h_B); free(h_C);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
CUDA Thread Organization

- **CUDA Kernel call:**
  
  `VecAdd<<<Nblocks, Nthreads>>>(d_A, d_B, d_C, N);`

- When a CUDA Kernel is launched, we specify the # of thread blocks and # of threads per block
  - The `Nblocks` and `Nthreads` variables, respectively

- `Nblocks * Nthreads = number of threads`
  - Tuning parameters.
  - What’s a good size for `Nblocks`?
  - Max threads per block = 1024
CUDA Thread Organization: More about Blocking

- Each thread in a thread block shares a fast piece of shared memory
  - This makes communicating and synchronizing within a thread block fast!
  - Not the case for threads in different blocks
- Ideally, thread blocks do completely independent work
- Thread blocks encapsulate many computational patterns
  - think MatMul blocking, Domain Decomposition, etc
CUDA Thread Organization: More about Blocking

- Each block is further subdivided into warps, which usually contain 32 threads.
  - Threads in each warp execute in a SIMD manner (together, on contiguous memory)
  - Gives us some intuition for good block sizes.
- Just to reiterate
  - Threads are first divided into blocks
  - Each block is then divided into multiple warps
  - Threads in a warp execute in a SIMD manner
    - can get a little confusing!
CUDA Memory Model
CUDA Thread Organization Cont.

- What’s the maximum number of threads one can ask for?
  - Number of SMXs * Number of Warps per SMX * 32
  - maximum != optimal
CUDA Synchronization

- We’ve already mentioned atomic operations
- CUDA supports locking
- Using implicit synchronization from kernel calls
- CUDA functions
  - `syncthreads()` ...block level sync
  - `cudaDeviceSynchronize()`
Libraries

- Basic Libraries
  - cuBLAS
  - cuDPP (data parallel primitives i.e. reduction)
  - and more

- Many high-performance tools built on top of these basic libraries
  - MAGMA (LAPACK)
  - FFmpeg
  - cuFFT
  - and more
Profiling

- Nvidia Visual Profiler is NVIDIA’s CUDA profiler
  - lots of effort put into GUI and user friendliness
- Alternatives
  - nvprof is a command line profiler
Tuning for Performance

- Many things that we learned about writing good parallel code for CPUs apply here!
  - Program for maximal locality, minimal stride, and sparse synchronization.
  - Blocking, Buffering, etc

- More generally
  - GPU Architecture
  - Minimizing Communication and Synchronization
  - Finding optimal block sizes
  - Using fast libraries

- What if we wanted to optimize Shallow Waters solver in PA2?
Note: Thrust

- Designed to be the “cstdlib.h” of CUDA
- Incredibly useful library that abstracts away many tedious aspects of CUDA
- Greatly increases programmer productivity
Note: What if I don’t want to program in C/C++?

- Answer: PyCUDA, jCUDA, some others provide CUDA integration for as well
  - Not as mature as C/C++ versions, some libraries not supported
- The newest version of MATLAB also supports CUDA
- Fortran
- There is always a tradeoff…
Recent Developments in CUDA

- Checkout CUDA Developer Zone
- Lots of cool stuff
Alternatives

- OpenCL is managed by the Khronos Group and is the open-source answer to CUDA
- Performance wise, quite similar, but not as mature and not as many nice features
- Others
  - DirectCompute (MS)
  - Brook+ (Stanford/AMD)
Credit

CS267 (Berkeley)

CS5220 Lec Slides from last class iteration

Mythbusters