CUDA Introduction

Scott Grauer-Gray
GPU Programming

- So far in this class...
  - OpenCL
    - Works on AMD/NVIDIA GPUs, CPUs, other accelerators
    - Lots of control for optimization
    - Need to add lots of code
  - OpenACC
    - Works on AMD/NVIDIA GPUs
    - More limited control for optimization
    - Only need to add a few lines of code (compiler does most of the work)
Compute Unified Device Architecture (CUDA)

- Developed by NVIDIA
  - Only works on NVIDIA GPUs
  - Lots of control for optimization
  - Need to add a moderate amount of code
Compute Unified Device Architecture (CUDA)

- Introduced in February 2007
- NVIDIA consistently adding features
  - Features added with new hardware architectures
  - Software features including templates, inheritance, function pointers, and recursion not initially supported and now are
NVIDIA GPU Architecture

- Pre-CUDA
  - Separate vertex and fragment shaders
  - GPGPU possible via graphics APIs
    - Basically "trick" the GPU into thinking it's doing graphics when actually doing GPGPU
  - Pre-CUDA NVIDIA GPU in Playstation 3
NVIDIA GPU Architecture

- CUDA Architecture
  - CUDA introduced with GeForce 8-series
  - Introduced unified shaders (CUDA cores)
    - Replaced separate vertex and fragment shaders
  - Flagship GTX 8800 had 128 cores
    - Divided into 16 multi-processors (8 cores each)
NVIDIA GPU Architecture

- Improved with GT200 (Tesla) architecture
  - Successor to GeForce 8-series
  - 240 CUDA cores
    - Divided into 30 multiprocessors of 8 cores each
  - Added double-precision support
  - Doubled number of registers per multiprocessor
NVIDIA GPU Architecture

● Further evolved with introduction of Fermi
  ■ 512 CUDA Cores
    ● Divided into 16 multiprocessors of 32 cores each
  ■ Introduced L1 and L2 cache for global memory
    ● Makes GPU multiprocessor more "CPU-like"
  ■ Improved double-precision performance
  ■ Added ECC for error correction
    ● Makes GPGPU a viable option when reliable results are critical
NVIDIA GPU Architecture

- Current architecture iteration: Kepler (GK110)
  - 2688 cores (max currently enabled)
    - Available as K20 and K20X compute processors as well as GeForce Titan graphics card
    - Divided into 14 multiprocessors of 192 cores each
  - Further improved double-precision performance
  - Adds dynamic parallelism
    - Ability to launch GPU kernels within a GPU kernel
NVIDIA Compute GPU Line

- Introduced specifically for GPGPU
  - More DRAM than graphics cards
  - More thoroughly tested --> less likely to fail
  - Better double-precision performance (on Fermi)
  - Only option for GPGPU with ECC
  - Significantly more expensive
    - Intended for academia/businesses, not consumers

- Cards
  - G80 arch: C870
  - GT200 arch: C1060 (in cuda.acad)
  - Fermi arch: C2050/C2070
  - Kepler arch: K20/K20X
NVIDIA GPU Vs. CPU Performance
Compute Capability

- Used to distinguish between architectures
  - Different architectures have different features
  - Different specifications for max thread block size, number of registers per thread, etc
  - Multiple architectures in same "family" may have same compute capability
    - Means CUDA features are the same (likely just different number of CUDA cores)
    - GTX 680 and GTX 660m architectures differ in number of multiprocessors, but each have compute capability of 3.0
      - Same multiprocessor design, difference is that GTX 680 has many more multiprocessors
      - Doesn't affect CUDA features, only how fast program runs
Compute Capability

● Compute capability 1.0
  ○ Corresponds to first CUDA architecture (G80)

● Compute capability 1.1/1.2
  ○ Evolution on G80
  ○ Added various atomic operations on data in global memory

● Compute capability 1.3
  ○ Corresponds to GT200 (architecture for C1060 in cuda.acad)
  ○ Added double-precision support to CUDA
Compute Capability

- Compute capability 2.0/2.1
  - Corresponds to Fermi
  - Increased max shared memory from 16 KB to 48 KB
    - Program using additional shared memory won't compile for previous architectures
  - Decreased max registers per thread from 127 to 63

- Compute capability 3.5
  - Corresponds to Kepler GK110
    - Other Kepler architectures correspond to 3.0
  - Introduced dynamic parallelism
    - CUDA-only feature
    - NVIDIA could add feature to OpenCL w/ extension
  - Increases max registers per thread to 255
## Feature Support per compute capability:

<table>
<thead>
<tr>
<th>Feature Support</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x, 3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Warp Vote Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_ballot() (Warp Vote Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_threadfence_system() (Memory Fence Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_syncthreads_count(), _syncthreads_end(), _syncthreads_or() (Synchronization Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surface functions (Surface Functions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Funnel shift (see reference manual)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
# CUDA Specs by Compute Capability

From NVIDIA CUDA Documentation

- **Technical Specifications per compute capability:**

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum y- or z-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to linear memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and number of layers for a 1D layered texture reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to linear memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width, height, and number of layers for a 2D layered texture reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width, height, and depth for a 3D texture reference bound to a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a cubemap texture reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width (and height) and number of layers for a cubemap layered texture reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of textures that can be bound to a kernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding Compute Capacity / GPU Characteristics

- Want to determine features of particular NVIDIA card
  - Use deviceQuery CUDA program in NVIDIA SDK
    - Output gives GPU name
    - Compute capability
    - Number of cores
    - Amount of DRAM
    - Clock speeds
    - Other card specifications
Device Query output on cuda.acad (C1060 GPU)

CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "Tesla C1060"
CUDA Driver Version / Runtime Version 4.0 / 4.0
CUDA Capability Major/Minor version number: 1.3
Total amount of global memory: 4096 MBytes (4294770688 bytes)
(30) Multiprocessors x (8) CUDA Cores/MP: 240 CUDA Cores
GPU Clock Speed: 1.30 GHz
Memory Clock rate: 800.00 Mhz
Memory Bus Width: 512-bit
Max Texture Dimension Size (x,y,z) 1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(8192) x 512, 2D=(8192,8192)
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 16384 bytes
Total number of registers available per block: 16384
Warp size: 32
Maximum number of threads per block: 512
Maximum size of each dimension of a block: 512 x 512 x 64
Maximum size of each dimension of a grid: 65535 x 65535 x 1
Maximum memory pitch: 2147483647 bytes
Texture alignment: 256 bytes
Concurrent copy and execution: Yes with 1 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Concurrent kernel execution: No
Alignment requirement for Surfaces: Yes
Device has ECC support enabled: No
Device is using TCC driver mode: No
Device supports Unified Addressing (UVA): No
Device PCI Bus ID / PCI location ID: 2 / 0
Compute Mode: < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.0, CUDA Runtime Version = 4.0, NumDevs = 1, Device = Tesla C1060
deviceQuery test results...
PASSED

Press ENTER to exit...
CUDA Environment

- Similar to OpenCL
  - Kernel runs on many threads in parallel
  - Each thread has a unique ID
  - Threads are grouped in thread-blocks
    - Threads in same thread-block run on same multiprocessor
    - Threads within a thread-block can be synchronized and have access to same shared memory
CUDA Environment

- Different terminology compared to OpenCL
  - Thread in CUDA <-> Work-Item in OpenCL
  - Thread-Block in CUDA <-> Work-Group in OpenCL
  - Shared memory in CUDA <-> __local memory in OpenCL
  - Registers in CUDA <-> __private memory in OpenCL
CUDA Memory Model

- Memory model similar to OpenCL
  - Global memory
    - Stored in DRAM
    - Available to all threads
    - Access pattern important
      - Best for contiguous threads (in terms of thread ID) to access "contiguous" data in array in DRAM
      - Bad memory access pattern can slow down kernel
        - Penalty worse in older architectures
  - Shared memory
    - On-chip
    - Fast access
    - Shared within thread block
    - Can be used as user-managed cache
CUDA Memory Model

- **Constant memory**
  - Fast, read-only memory
  - Useful if all threads reading from same set of data

- ** Registers**
  - Fast access
  - Private to each thread
  - Limited number in each multiprocessor
  - Spill-over from registers goes to local memory (may be stored in slow global memory space)
    - Using too many registers may slow program down
    - Less of an issue w/ L1 and L2 cache
CUDA Environment

- CUDA Kernel almost same as in OpenCL
  - Different keywords
    - `__global__` used instead of `__kernel` for "entry" kernel from host
    - `threadIdx` and `blockIdx` (with dimensions x, y, and z) used instead
      get_global_id to retrieve thread ID
CUDA Environment

- Host simpler than OpenCL
  - No need to set platform, context, command queue, and compile device program at run-time
    - Any errors in kernel(s) will show up when compiling overall program
  - Still need to manually allocate data on GPU and transfer data from CPU to GPU, run kernel, and transfer data back
CUDA Memory Management

- "Array" on GPU
  - Treated similar to regular array
  - Stored in global memory on GPU
    - Pointer to location of array in GPU memory on host end
  - cudaMalloc - call on host to allocate memory to GPU array
  - cudaMemcpy - transfer data between GPU and host arrays
    - Last parameter to cudaMemcpy gives direction of memory transfer
      - Parameter is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost
  - cudaFree - call on host to free memory on GPU
CUDA Kernel Execution

- Similar to calling regular C function (but not the same)
  - Begin with kernel name (basically same as C function)
    - Need to pass in grid and thread block dimensions between <<< and >>> (different from normal C function)
    - Called execution configuration
    - Grid and thread block dimensions are of type dim3 - integer vector type
  - Function parameters follow in same manner as C function
  - Thread block dimensions equivalent to local work-group dimensions in OpenCL
    - Can be 1D, 2D, or 3D
    - Needs to be set in CUDA (while can be NULL in OpenCL)
CUDA Kernel Execution

- Grid dimensions correspond to number of thread blocks in x, y, and z directions
- **Example**: if processing 256,000 threads using 1D thread blocks of size 256, grid dimension would be 1000 in the x direction (and 1 in y and z directions)
Simple Program A

- Initialize two arrays of size N
- Add values element-by-element
- Place output summations in another array (of size N)
int main()
{
    float inArrayA[N];
    float inArrayB[N];
    float outArrayC[N];

    //function to set values in array in some manner
    initializeArray(inArrayA);
    initializeArray(inArrayB);

    for (int i=0; i < N; i++)
    {
        outArrayC[i] = inArrayA[i] + inArrayB[i];
    }

    //function to do "stuff" with the output data
    processOutputArray(outArrayC);

    return 0;
}
Simple Program A: CUDA code - KERNEL

- Minor differences compared to OpenCL kernel for same function
  - Use `__global__` keyword for entry kernel from host (instead of `__kernel`)
  - Use built-in variables `blockIdx`, `blockDim`, and `threadIdx` to retrieve thread ID
    - Parallelizing single for-loop
    - 1D thread block and grid in this program, so only using 'x' dimension

```c
__global__ void addArrays(float* inArrayA, float* inArrayB, float* outArrayC, int nVal)
{
    //retrieve thread ID
    int threadId = blockIdx.x*blockDim.x + threadIdx.x;

    //check if within computation bounds
    if ((threadId >= 0) && (threadId < nVal))
    {
        //run computation for loop iteration
        outArrayC[threadId] = inArrayA[threadId] + inArrayB[threadId];
    }
}
```
int main()
{
  float inArrayA[N];  float inArrayB[N];  float outArrayC[N];
  float* inArrayAGpu; float* inArrayBGpu; float* outArrayCGpu;

  //allocate data on GPU
  cudaMalloc(&inArrayAGpu, N*sizeof(float));
  cudaMalloc(&inArrayBGpu, N*sizeof(float));
  cudaMalloc(&outArrayCGpu, N*sizeof(float));

  //Set values in array in some manner
  initializeArray(inArrayA); initializeArray(inArrayB);

  //transfer input arrays to GPU
  cudaMemcpy(inArrayAGpu, inArrayA, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(inArrayBGpu, inArrayB, N*sizeof(float), cudaMemcpyHostToDevice);

  //Set thread block (to 256) and grid dimensions (using 1-D thread block and grid)
  dim3 blockDims(256, 1, 1); dim3 gridDims((int)(ceil((float)N / 256.0)), 1, 1);

  //run the addArrays CUDA kernel on the GPU
  addArrays <<< gridDims, blockDims >>> (inArrayAGpu, inArrayBGpu, outArrayCGpu, N);
  cudaThreadSynchronize();

  //transfer output array from GPU to host
  cudaMemcpy(outArrayC, outArrayCGpu, N*sizeof(float), cudaMemcpyDeviceToHost);

  //free allocated data on GPU
  cudaFree(inArrayAGpu); cudaFree(inArrayBGpu); cudaFree(outArrayCGpu);

  //Do "stuff" with the output data
  processOutputArray(outArrayC);

  return 0;
}
CUDA Libraries

- Allow programmer to use GPU acceleration (specifically CUDA) without needing to manually write kernels
  - Only work on NVIDIA GPUs
    - Kernels for particular functions already written and optimized
    - Likely better performance than writing own code for function
CUDA Libraries

○ Thrust
  ■ Parallel algorithms and data structures (including reductions)
  ■ Similar to C++ standard template library
○ cuBLAS
  ■ Matrix/linear algebra computation
○ cuSPARSE
  ■ Sparse matrix computation
○ cuRAND
  ■ Random number generation
○ NPP
  ■ Signal and Image processing
○ cuFFT
  ■ Fast fourier transform on GPU
Vector Add Using Thrust

```cpp
int main()
{
    // generate two vectors of random numbers on host
    thrust::host_vector<float> h_vect1(N);
    thrust::host_vector<float> h_vect2(N);
    thrust::generate(h_vect1.begin(), h_vect2.end(), rand);
    thrust::generate(h_vect1.begin(), h_vect2.end(), rand);

    // generate vector for output values
    thrust::host_vector<float> h_vectOut(N);

    // transfer the vectors to the device for computation
    thrust::device_vector<float> d_vect1 = h_vect1;
    thrust::device_vector<float> d_vect2 = h_vect2;
    thrust::device_vector<float> out_vect = h_vectOut;

    // run vector addition on the GPU using thrust library
    thrust::transform(d_vect1.begin(), d_vect1.end(), d_vect2.begin(), out_vect.begin(),
                      thrust::plus<float>());

    // transfer output vector back to host
    h_vectOut = out_vect;

    return 0;
}
```

- No need to set execution configuration and create CUDA kernel using thrust
  - Simplifies programming on the GPU
Other CUDA Tools

- CUDA profiler
  - Available using command line and as GUI
  - Can be used to time CUDA kernel(s)
  - Gives info about multiprocessor occupancy, memory access pattern, local memory usage, cache usage, etc
    - Need to specify what characteristics to measure
  - Output can be used to determine bottleneck(s) in kernel

- CUDA GDB
  - CUDA debugger for linux and mac os
  - Allows user to set breakpoints, step through CUDA applications, and inspect memory/variables of any thread
Other CUDA Tools

● **CUDA-MEMCHECK**
  ○ Functional correctness suite
  ○ Can precisely detect out-of-bounds and misaligned memory access errors
  ○ Report hardware exceptions
  ○ Report data races in shared memory

● **Nsight**
  ○ Available for Visual Studio and Eclipse
  ○ Provides integrated development environment for developers for building CUDA applications
  ○ Intended to help make CUDA programming as simple/straightforward as possible