CUDA and GPU Programming

University of Georgia
CUDA Teaching Center

Week 1: Introduction to GPUs and CUDA
April 3, 2013
Schedule of Topics

April 3: Introduction to GPUs and CUDA
April 10: CUDA Memory Model
April 17: Optimization and Profiling
April 24: “Real-World” CUDA Programming

Session Format:
3:30 – 4:30: Lecture presentation
4:30 – 5:00: Hands-on programming
GPU Resources at UGA

CUDA Teaching Center (cuda.uga.edu)
  Jennifer Rouan (jdrouan@uga.edu)
  Chulwoo Lim (churu@uga.edu)
  John Kerry (john@uga.edu)
  Ahmad Al-Omari (aomari@uga.edu)

GACRC (gacrc.uga.edu)
  Shan-ho Tsai (tsai@hal.physast.uga.edu)
Motivation: The Potential of GPGPU

- In short:
  - The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
  - Example applications range from in-game physics simulation to conventional computational science
  - NVIDIA architect John Danskin (GHo8) described the workload in a modern game: “AI (suitable for GPUs); physics (suitable for GPUs); graphics (suitable for GPUs); and a ‘perl script, which can be run on a serial CPU that takes five square millimeters and consumes one percent of a processor die”

- Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor
Recent GPU Performance Trends

Historical Single-/Double-Precision Peak Compute Rates

- **GFLOPS**
- **Date**
- **Precision**
  - SP
  - DP
- **Vendor**
  - AMD (GPU)
  - NVIDIA (GPU)
  - Intel (CPU)
  - Intel Xeon Phi
Successes on NVIDIA GPUs

Interactive visualization of volumetric white matter connectivity

Ionic placement for molecular dynamics simulation on GPU

Transcoding HD video stream to H.264

Fluid mechanics in Matlab using .mex file CUDA function

Astrophysics N-body simulation

Financial simulation of LIBOR model with swaptions

GLAME@lab: an M-script API for GPU linear algebra

Ultrasound medical imaging for cancer diagnostics

Highly optimized object oriented molecular dynamics

Cmatch exact string matching to find similar proteins and gene sequences

[courtesy David Luebke, NVIDIA]
Why is data-parallel computing fast?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
- So, more transistors can be devoted to data processing rather than data caching and flow control
SM Multithreaded Multiprocessor

- Each SM runs a block of threads
- SMs have 8, 16, or 32 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hw multithreaded (1024 in newer hw)
- 16KB Shared Memory (64KB in newer hw)
  - Concurrent threads share data
  - Low latency load/store
- 32 elements run at same time (SIMD) as a warp
Scaling the Architecture

- Same program
- Scalable performance
NVIDIA Kepler

Kepler: Fast & Efficient

SM
Fermi

CONTROL LOGIC
32 cores

3x
Perf / Watt

SMX
Kepler

CONTROL LOGIC
192 cores

Dynamic Parallelism
GPU Adapts to Data, Dynamically Launches New Threads

CPU
Fermi GPU

CPU
Kepler GPU

Hyper-Q
CPU Cores Simultaneously Run Tasks on Kepler

FERMI
1 MPI Task at a Time

KEPLER
32 Simultaneous MPI Tasks

Tesla K10: Same Power, 2x Performance of Fermi

<table>
<thead>
<tr>
<th>Product Name</th>
<th>M2090</th>
<th>K10</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Architecture</td>
<td>Fermi</td>
<td>Kepler GK104</td>
</tr>
<tr>
<td># of GPUs</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Single Precision Flops</td>
<td>1.3 TF</td>
<td>4.58 TF</td>
</tr>
<tr>
<td>Double Precision Flops</td>
<td>0.66 TF</td>
<td>0.190 TF</td>
</tr>
<tr>
<td># CUDA Cores</td>
<td>512</td>
<td>3072</td>
</tr>
<tr>
<td>Memory size</td>
<td>6 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Memory BW (ECC off)</td>
<td>177.6 GB/s</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>PCI-Express</td>
<td>Gen 2: 8 GB/s</td>
<td>Gen 3: 16 GB/s</td>
</tr>
</tbody>
</table>

http://www.theregister.co.uk/2012/05/15/nvidia_kepler_omega_gpu_revealed/page2.html
CUDA Software Development Kit

CUDA Optimized Libraries:
- math.h,
- FFT,
- BLAS,
- ...

Integrated CPU + GPU C Source Code

NVIDIA C Compiler

NVIDIA Assembly for Computing (PTX)

CUDA Driver

Debugger Profiler

GPU

CPU Host Code

Standard C Compiler

CPU
Compiling CUDA for GPUs

C/C++ CUDA Application

NVCC

CPU Code

PTX Code

Generic

PTX to Target Translator

GPU ... GPU

Target device code
Programming Model: A Highly Multi-threaded Coprocessor

• The GPU is viewed as a compute device that:
  • Is a coprocessor to the CPU or host
  • Has its own DRAM (device memory)
  • Runs many threads in parallel
• Data-parallel portions of an application execute on the device as kernels that run many cooperative threads in parallel
• Differences between GPU and CPU threads
  • GPU threads are extremely lightweight
    • Very little creation overhead
  • GPU needs 1000s of threads for full efficiency
    • Multi-core CPU needs only a few
Structuring a GPU Program

- CPU assembles input data
- CPU transfers data to GPU (GPU “main memory” or “device memory”)
- CPU calls GPU program (or set of kernels). GPU runs out of GPU main memory.
- When GPU finishes, CPU copies back results into CPU memory
- Recent interfaces allow overlap.
- What lessons can we draw from this sequence of operations?
Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate
- Blocks are independent
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One SIMT kernel is executed at a time
  - Many threads execute each kernel

- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA must use 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:

Device = GPU; Host = CPU

Kernel = function that runs on the device
Execution Model

Multiple levels of parallelism

- Thread block
  - Up to 512 threads per block
  - Communicate through shared memory
  - Threads guaranteed to be resident
    - threadIdx, blockIdx
  - __syncthreads()
- Grid of thread blocks
  - f<<<nbblocks, nthreads>>>(a, b, c)
Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate, runs to completion
- Several blocks can reside concurrently on one multiprocessor (SM)
  - Control limitations (of G8X/G9X GPUs):
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM (1024 in new hw)
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks
Key Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads
Hierarchical of concurrent threads

- Parallel *kernels* composed of many threads
  - all threads execute the same sequential program
  - (This is “SIMT”)
- Threads are grouped into *thread blocks*
  - threads in the same block can cooperate
- Threads/blocks have unique IDs
  - Each thread knows its “address” (thread/block ID)
What is a thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be *physical* threads
  - as on NVIDIA GPUs
- CUDA threads might be *virtual* threads
  - might pick 1 block = 1 physical thread on multicore CPU
- Very interesting recent research on this topic
What is a thread block?

- Thread block = *virtualized multiprocessor*
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) *parallel task*
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be *independent* tasks
  - program valid for *any interleaving* of block executions
Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD ... can easily deadlock
- Independence requirement gives scalability
CUDA Program Execution
CUDA Program Structure example

```c
int main(void) {
    float *a_h, *a_d;               // pointers to host and device arrays
    const int N = 10;                      // number of elements in array
    size_t size = N * sizeof(float);           // size of array in memory

    // allocate memory on host and device for the array
    // initialize array on host (a_h)
    // copy array a_h to allocated device memory location (a_d)

    // kernel invocation code – to have the device perform
    // the parallel operations

    // copy a_d from the device memory back to a_h
    // free allocated memory on device and host
}
```
Data Movement and Memory Management

• In CUDA, host and device have separate memory spaces
• To execute a kernel, the program must allocate memory on the device and transfer data from the host to the device
• After kernel execution, the program needs to transfer the resultant data back to the host memory and free the device memory
• C functions: `malloc()`, `free()`
  CUDA functions: `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()`
int main(void)
{
    float *a_h, *a_d; const int N = 10;
    size_t size = N * sizeof(float);           // size of array in memory

    a_h = (float *)malloc(size);                // allocate array on host
    cudaMalloc((void **) &a_d, size);         // allocate array on device
    for (i=0; i<N; i++) a_h[i] = (float)i;            // initialize array
    cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);

    // kernel invocation code

    cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(a_d); free(a_h);                    // free allocated memory
}
Kernel Invocation example

int main(void)
{
    float *a_h, *a_d; const int N = 10;
    size_t size = N * sizeof(float);           // size of array in memory

    a_h = (float *)malloc(size);                // allocate array on host
    cudaMalloc((void **) &a_d, size);         // allocate array on device
    for (i=0; i<N; i++) a_h[i] = (float)i;            // initialize array
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);

    int block_size = 4;                    // set up execution parameters
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
    square_array <<< n_blocks, block_size >>> (a_d, N);

    cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(a_d); free(a_h);                    // free allocated memory
}

Kernel Functions

- A kernel function specifies the code to be executed by all threads in parallel – an instance of single-program, multiple-data (SPMD) parallel programming.
- A kernel function declaration is a C function extended with one of three keywords: “__device__”, “__global__”, or “__host__”.

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>
CUDA Thread Organization

• Since all threads execute the same code, how do they determine which data to work on?
• CUDA provides built-in variables to generate a unique identifier across all threads in a grid
• Example (8 threads per block in the x-dimension):
  \[ i = blockIdx.x \times blockDim.x + threadIdx.x; \]

| blockIdx.x | 0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 1 | 2 2 2 2 2 2 2 2 |
| blockDim.x | 8 8 8 8 8 8 8 8 | 8 8 8 8 8 8 8 8 | 8 8 8 8 8 8 8 8 |
| offset | 0 0 0 0 0 0 0 0 | 8 8 8 8 8 8 8 8 | 16 16 16 16 16 16 16 16 |
| threadIdx.x | 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7 |
| unique threadID (offset + threadIdx.x) | 0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 |
CUDA kernel function:

```c
__global__ void square_array(float *a, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Compare with serial C version:

```c
void square_array(float *a, int N) {
    int i;
    for (i = 0; i < N; i++) a[i] = a[i] * a[i];
}
```
GPU Design Principles

- Data layouts that:
  - Minimize memory traffic
  - Maximize coalesced memory access

- Algorithms that:
  - Exhibit data parallelism
  - Keep the hardware busy
  - Minimize divergence
References

• Programming Massively Parallel Processors with CUDA by Stanford University

• CPU DB by Stanford VLSI Group
  http://cpudb.stanford.edu/visualize

• Introduction to Parallel Programming by Udacity
  https://www.udacity.com/course/cs344