Lecture 15 - CUDA

Mattan Erez

The University of Texas at Austin
Outline

- Bandwidths
- CUDA
  - Overview
  - Development process
  - Performance Optimization (just started in this lecture)
  - Syntax (skipped)

- Most slides courtesy Massimiliano Fatica (NVIDIA)
Bandwidths of GeForce 8800 GTX

- **Frequency**
  - 575 MHz with ALUs running at 1.35 GHz

- **ALU bandwidth (G FLOPs)**
  - \((1.35 \text{ GHz}) \times (16 \text{ SM}) \times ((8 \text{ SP}) \times (2 \text{ MADD}) + (2 \text{ SFU})) = \sim 388 \text{ G FLOPs}\)

- **Register BW**
  - \((1.35 \text{ GHz}) \times (16 \text{ SM}) \times (8 \text{ SP}) \times (4 \text{ words}) = 2.8 \text{ TB/s}\)

- **Shared Memory BW**
  - \((575 \text{ MHz}) \times (16 \text{ SM}) \times (16 \text{ Banks}) \times (1 \text{ word}) = 588 \text{ GB/s}\)

- **Device memory BW**
  - 1.8 GHz GDDR3 with 384 bit bus: 86.4 GB/s

- **Host memory BW**
  - PCI-express: 1.5 GB/s or 3 GB/s with page locking
Outline

• Bandwidths
• CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

• Most slides courtesy Massimiliano Fatica (NVIDIA)
CUDA is a programming system for utilizing the G80 processor for compute
- CUDA follows the architecture very closely

General purpose programming model
- User kicks off batches of threads on the GPU
- GPU = dedicated super-threaded, massively data parallel co-processor

Matches architecture features
Specific parameters not exposed
The CUDA Platform

- The G80 GPU is not integrated into the CPU
  - Connects through a PCI Express bus
  - Communicates through OS (drivers)
CUDA Programming System

- **Targeted software stack**
  - Compute oriented drivers, language, and tools

- **Driver for loading computation programs into GPU**
  - Standalone Driver - Optimized for computation
  - Interface designed for compute-graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management
Overall Performance Can be Limited by Interface

- H.264 LBM RC5-72 FEM RPES PNS SAXPY TPACF FDTD MRI-Q MRI-FHD

GPU Speedup Relative to CPU

- Kernel
- Application

- 210 79
- 457 431 316 263
Overall Performance Can be Limited by Interface

SGEMM performance

- GPU+I/O
- GPU+I/O Pinned
- GPU only

N

Gflops

0 512 1024 1536 2048 2560

0 20 40 60 80 100 120 140
CUDA API and Language: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance
CUDA Programming Model: 
A Highly Multithreaded 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 are executed on the device as kernels which run in parallel on many threads

• 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
CUDA is an Extension to C

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
foo.s

OCG

G80 SASS
foo.sass

CPU Host Code
foo.cpp

gcc / cl
CUDA is an Extension to C

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Function launch

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}
```

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Access Times

• Register – dedicated HW - single cycle
• Shared Memory – dedicated HW - two cycles
  – Hidden by warps
• Local Memory – DRAM, no cache - *slow*
• Global Memory – DRAM, no cache - *slow*
• Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
• Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
• Instruction Memory (invisible) – DRAM, cached
Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without blocking:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory
Programming Model: Common Programming Pattern

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory
  - Uncached

- So, a common way of scheduling some computation on the device is to block it up to take advantage of fast shared memory:
  - Partition the data set into data subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory
Programming Model: Square Matrix Multiplication Example

• \( P = M \times N \) of size \( \text{WIDTH} \times \text{WIDTH} \)

• With blocking:
  - One thread block handles one \( \text{BLOCK SIZE} \times \text{BLOCK SIZE} \) sub-matrix \( P_{\text{sub}} \) of \( P \)
  - \( M \) and \( N \) are only loaded \( \frac{\text{WIDTH}}{\text{BLOCK SIZE}} \) times from global memory

• Great saving of memory bandwidth!
A quick review

- **device** = GPU = set of multiprocessors
- **Multiprocessor** = set of processors & shared memory
- **Kernel** = GPU program
- **Grid** = array of thread blocks that execute a kernel
- **Thread block** = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>
CUDA: C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```c
__global__ void KernelFunc(...);
__shared__ int SharedVar;

KernelFunc<<< 500, 128 >>>(...);
```

- Explicit GPU memory allocation
  - `cudaMalloc()`, `cudaFree()`
- Memory copy from host to device, etc.
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
Example: Vector Addition Kernel

// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x + blockDim.x * blockId.x;
    oC[idx] = iA[idx] + iB[idx];
}
Example: Vector Addition Host Code

```c
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// ... initialize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float)));
cudaMalloc( (void**) &d_B, N * sizeof(float)));
cudaMalloc( (void**) &d_C, N * sizeof(float)));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```
Outline

• Bandwidths
• CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

• Most slides courtesy Massimiliano Fatica (NVIDIA)
Compilation

• Any source file containing CUDA language extensions must be compiled with **nvcc**

• NVCC is a **compiler driver**
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

• NVCC can output:
  - Either C code (CPU Code)
    • That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly

• Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (**cudart**)  
  - The CUDA core library (**cuda**)
Compiling CUDA

C/C++ CUDA Application

NVCC

PTX Code

PTX to Target Compiler

CPU Code

G80

GPU

Target code
Compiling CUDA

C/C++ CUDA Application

NVCC

PTX Code

PTX to Target Compiler

Target code

Virtual

Physical

G80

GPU

© NVIDIA Corp.
NVCC & PTX Virtual Machine

- **EDG**
  - Separate GPU vs. CPU code

- **Open64**
  - Generates GPU PTX assembly

- **Parallel Thread eXecution (PTX)**
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```c
float4 me = gx[gtid];
me.x += me.y * me.z;
```

```Assembly
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```
Role of Open64

Open64 compiler gives us

• A complete C/C++ compiler framework. Forward looking. We do not need to add infrastructure framework as our hardware architecture advances over time.

• A good collection of high level architecture independent optimizations. All GPU code is in the inner loop.

• Compiler infrastructure that interacts well with other related standardized tools.
Debugging Using the Device Emulation Mode

• An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

• When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of __syncthreads
Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results.

- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs
  - Different instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host.
Parameterize Your Application

• Parameterization helps adaptation to different GPUs

• GPUs vary in many ways
  - # of multiprocessors
  - Shared memory size
  - Register file size
  - Threads per block
  - Memory bandwidth

• You can even make apps self-tuning (like FFTW)
  - “Experiment” mode discovers and saves optimal config
Outline

• Bandwidths
• CUDA
  – Overview
  – Development process
  – Performance Optimization
  – Syntax

• Most slides courtesy Massimiliano Fatica (NVIDIA)
CUDA Optimization Priorities

• Memory coalescing is #1 priority
  – Highest !/$ optimization
  – Optimize for locality

• Take advantage of shared memory
  – Very high bandwidth
  – Threads can cooperate to save work

• Use parallelism efficiently
  – Keep the GPU busy at all times
  – High arithmetic / bandwidth ratio
  – Many threads & thread blocks

• Leave bank conflicts and divergence for last!
  – 4-way and smaller conflicts are not usually worth avoiding if avoiding them will cost more instructions