L16: Introduction to CUDA

November 1, 2012

Outline

- Overview of the CUDA Programming Model for NVIDIA systems
  - Presentation of basic syntax
- Simple working examples
  - See http://www.cs.utah.edu/~mhall/cs6963s09
- Architecture
- Execution Model

Reading

- David Kirk and Wen-mei Hwu manuscript or book
- CUDA Manual, particularly Chapters 2 and 4
  (download from nvidia.com/cudazone)
- Nice series from Dr. Dobbs Journal by Rob Farber
  - http://www.ddj.com/cpp/207200659

Administrative

Reminder: Project 4 due tomorrow at midnight
See instructions on mailing list
Final Project proposal due November 20
Today's Lecture

• Goal is to enable writing CUDA programs right away
  - Not efficient ones - need to explain architecture and mapping for that
  - Not correct ones (mostly shared memory, so similar to OpenMP)
  - Limited discussion of why these constructs are used or comparison with other programming
  - Limited discussion of how to use CUDA environment
  - No discussion of how to debug.

Why Massively Parallel Processor

• A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API

  - GPU in every PC and workstation - massive volume and potential impact

Minimal Extensions to C + API

• Declspecs
  - global, device, shared, local, constant
  - __device__ float filter[N];
  - __global__ void convolve (float *image)
    - __shared__ float region[N];
    - __constant__ constant

• Keywords
  - threadIdx, blockIdx
  - region[threadIdx] = image[i];

• Intrinsics
  - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
    - __syncthreads()
NVCC Compiler’s Role: Partition Code and Compile for Device

mycode.cu

__device__ dfunc() {
  int ddata;
}

__global__ gfunc() {
  int gdata;
}

Main() {}

__host__ hfunc() {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

Device Only

Interface

Host Only

int main_data;
__shared__ int sdata;

Main() {}

__host__ hfunc() {
  int hdata;
  <<<gfunc(g,b,m)>>>();
}

Compiled by native compiler: gcc, icc, cc

Compiled by nvcc compiler

CS4230

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

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 (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.x, y, z)

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...

Thread Batching: Grids and Blocks

Host

Device

Grid 1

Kernel 2

Block (1, 1)

Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)

Block (0, 0)
Block (1, 0)
Block (2, 0)

Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)

Block (1, 1)
Block (0, 1)
Block (2, 1)

Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Block (1, 2)
Block (0, 2)
Block (2, 2)

Device

Grid 1

Kernel 1

Block (1, 0)

Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)

Block (0, 0)
Block (1, 0)
Block (2, 0)

Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)

Block (1, 1)
**Simple working code example: Count 6**

- Goal for this example:
  - Really simple but illustrative of key concepts
  - Fits in one file with simple compile command
  - Can absorb during lecture

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does "6" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Known as a cyclic data distribution

---

**CUDA Pseudo-Code**

**MAIN PROGRAM:**
- Allocation memory on host for input and output
- Assign random numbers to input array
- Call host function
- Calculate final output from per-thread output
- Print result

**HOST FUNCTION:**
- Allocate memory on device for copy of input and output
- Copy input to device
- Set up grid/block
- Call global function
- Synchronize after completion
- Copy device output to host

**GLOBAL FUNCTION:**
- Thread scans subset of array elements
- Call device function to compare with "6"
- Compute local result

**DEVICE FUNCTION:**
- Compare current element and "6"
- Return 1 if same, else 0

---

**Main Program: Preliminaries**

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    /* initialization */
    outer_compute(in_array, out_array);
    ...
}
```

---

**Main Program: Invoke Global Function**

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute(int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    /* initialization */
    outer_compute(in_array, out_array);
    ...
}
```
Main Program: Calculate Output & Print Result

MAIN PROGRAM:
Initialization (OMIT)
• Allocate memory on host for input and output
• Assign random numbers to input array
Call host function
Calculate final output from per-thread output
Print result

#include <stdio.h>
define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute(int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
in_array, *out_array;
sum = 0;
/* initialization */ …
outer_compute(in_array, out_array);
for (int i=0; i<BLOCKSIZE; i++) {
    sum+=out_array[i];
}
printf("Result = %d\n",sum);
}

Host Function: Preliminaries & Allocation

HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    … do computation …
    cudaMemcpy(d_out_array, h_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}

Host Function: Copy Data To/From Host

HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    … do computation …
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}

Host Function: Setup & Call Global Function

HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    cudaThreadSynchronize();
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}

// Additional code for host function
Global Function

GLOBAL FUNCTION:
Thread scans subset of array elements
Call device function to compare with "6"
Compute local result

```
__global__ void compute(int *d_in, int *d_out)
{
  d_out[threadIdx.x] = 0;
  for (int i = 0; i < SIZE/BLOCKSIZE; i++)
  {
    int val = d_in[i*BLOCKSIZE + threadIdx.x];
    d_out[threadIdx.x] += compare(val, 6);
  }
}
```

Device Function

DEVICE FUNCTION:
Compare current element and "6"
Return 1 if same, else 0

```
__device__ int compare(int a, int b)
{
  if (a == b) return 1;
  return 0;
}
```

Reductions

- This type of computation is called a **parallel reduction**
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) [dimensionality reduced]
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs?

Standard Parallel Construct

- Sometimes called "embarrassingly parallel" or "pleasingly parallel"
- Each thread is completely independent of the others
- Final result copied to CPU
- Another example, adding two matrices:
  - A more careful examination of decomposing computation into grids and thread blocks
**Summary of Lecture**

- Introduction to CUDA
  - Essentially, a few extensions to C API supporting heterogeneous data-parallel CPU+GPU execution
    - Computation partitioning
    - Data partitioning (parts of this implied by decomposition into threads)
    - Data organization and management
    - Concurrency management
  - Compiler nvcc takes as input a .cu program and produces
    - C Code for host processor (CPU), compiled by native C compiler
    - Code for device processor (GPU), compiled by nvcc compiler
- Two examples
  - Parallel reduction
  - Embarassingly/Pleasingly parallel computation (your assignment)