L18: Introduction to CUDA

November 5, 2009

Administrative

• Homework assignment 3 will be posted today (after class)
  - Due, Thursday, November 5 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:
  "handin cs4961 hw3 <gzipped tar file>"

NEW: VTUNE PORTION IS EXTRA CREDIT!

• Mailing list set up: cs4961@list.eng.utah.edu

A Few Words About Final Project

• Purpose:
  - A chance to dig in deeper into a parallel programming model
  - Present results to work on communication of technical ideas

• Write a non-trivial parallel program that combines
  two parallel programming languages/models. In some
  cases, just do two separate implementations.
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of
    the code)
  - TBB + SSE-3
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA

• Present results in a poster session on the last day of
  class

Example Projects

• Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
    - Implement in MPI (+ SSE-3)
    - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
    - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains...
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
• Heterogeneous Memory Hierarchy

This lecture includes slides provided by:
Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)
see http://courses.ece.uiuc.edu/ece498/al1/
and Austin Robison (NVIDIA)

Reading

• David Kirk and Wen-mei Hwu manuscript (in progress)
  - http://www.tooldoc.com/CUDA-textbook-by-David-Kirk-
    from-NVIDIA-and-Prof-Wen-mei-Hwu-pdf.html
• CUDA 2.x 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

CUDA (Compute Unified Device Architecture)

• Data-parallel programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
• Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between host and GPU
  - Synchronization, memory management, testing, ...
• CUDA is one of first to support heterogeneous architectures (more later in the semester)
• CUDA environment
  - Compiler, run-time utilities, libraries, emulation, performance

What Programmer Expresses in CUDA

• Computation partitioning (where does computation occur?)
  - Declarations on functions __host__, __global__, __device__
  - Mapping of thread programs to device: compute <<<gs, bs>>>(<args>)
• Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data __shared__, __device__, __constant__, ...
• Data management and orchestration
  - Copying to/from host: e.g., cudaMemcpy(h_obj,d_obj, cudaMemcpyDeviceToHost)
• Concurrency management
  - E.g., syncthreads()
Minimal Extensions to C++ API

- **Declspecs**
  - `__device__ float filter[N];`
  - `__global__ void convolve (float *image) {}`
- **Keywords**
  - `threadIdx, blockIdx`
- **Intrinsics**
  - `__syncthreads()`
- **Runtime API**
  - Memory, symbol, execution management
- **Function launch**
  - `__device__ float filter[N];`
  - `__global__ void convolve (float *image)`
  - `__shared__ float region[M];`
  - `__syncthreads()`
  - `image[j] = result;`

```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

---

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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

---

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 different blocks cannot cooperate

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D \((\text{blockIdx.x, blockIdx.y})\)
  - Thread ID: 1D, 2D, or 3D \((\text{threadIdx.x, threadIdx.y})\)
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...

Simple working code example

- 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 "3" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

Working through an example

- Like MPI, we'll write some message-passing pseudo code for Count3 (from Lecture 4)

```c
int array[length];
int total;
for (i=0; i<length; i++)
{
  if (array[i] == 3) {
    prv_count++;  // Local variable
  }
  total += prv_count;  // Global variable
}
```

CUDA Pseudo-Code

MAIN PROGRAM:
- Initialization
- 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

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

GLOBAL FUNCTION:
- Thread scans subset of array elements
- Call device function to compare with "3"
- Compute local result

DEVICE FUNCTION:
- Compare current element and "3"
- Return 1 if same, else 0
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

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

__host__ void outer_compute(int *in_arr, int *out_arr) {
  ...
  cudaMemcpy(cudaMalloc(void **) &d_out_array, sizeof(int)),
  cudaMemcpy(void **) &d_out_array, BLOCKSIZE*sizeof(int));
  ...
}
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
- Copy device output to host

```c
__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(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
- Copy device output to host

```c
__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);
    compute<<<(1,BLOCKSIZE,0)>>>(d_in_array, d_out_array);
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
```

Global Function

**Global Function**: Thread scans subset of array elements
- Call device function to compare with "3"
- Compute local result

```c
__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, 3);
    }
}
```

Device Function

**Device Function**: Compare current element and "3"
- Return 1 if same, else 0

```c
__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}
```
Another Example: Adding Two Matrices

CPU C program
void add_matrix_cpu(float *a, float *b, float *c, int N)
{
    int i, j, index;
    for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
            index = i+j*N;
            c[index] = a[index] + b[index];
        }
    }
}

CUDA C program
__global__ void add_matrix_gpu(float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    int j = blockIdx.y*blockDim.y+threadIdx.y;
    int index = i+j*N;
    if (i < N && j < N)
        c[index] = a[index] + b[index];
}

Example source: Austin Robison, NVIDIA

Closer Inspection of Computation and Data Partitioning
- Define 2-d set of blocks, and 2-d set of threads per block
  
  ```c
  dim3 dimBlock(blocksize,blocksize);
  dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
  ```

  Each thread identifies what element of the matrix it operates on
  ```c
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  if (i < N && j < N)
      c[index] = a[index] + b[index];
  ```

Hardware Implementation: A Set of SIMD Multiprocessors
- A device has a set of multiprocessors
- Each multiprocessor is a set of 32-bit processors
  with a Single Instruction Multiple Data architecture
    - Shared instruction unit
    - At each clock cycle, a multiprocessor executes
      the same instruction on a group of threads called a warp
    - The number of threads in a warp is the warp size

Hardware Execution Model
I. SIMD Execution of warpsize=M threads (from single block)
   - Result is a set of instruction streams
     roughly equal to # blocks in thread divided by warpsize

II. Multithreaded Execution across different instruction streams within block
    - Also possibly across different blocks if there are more blocks than SMs

III. Each block mapped to single SM
    - No direct interaction across SMs
Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory.
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space

Programmer’s View: Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can read/write global, constant, and texture memory

Example SIMD Execution

"Count 3" kernel function

d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++)
    int val = d_in[i*BLOCsize + threadIdx.x];
    d_out[threadIdx.x] += compare(val, 3);
Example SIMD Execution

"Count 3" kernel function

```c
int val = d_in[i*BLOCKSIZE + threadIdx.x];
```
d_out[threadIdx.x] += compare(val, 3);
```c
```

Example SIMD Execution

"Count 3" kernel function

```c
int val = d_in[i*BLOCKSIZE + threadIdx.x];
```
d_out[threadIdx.x] += compare(val, 3);
```c
```

SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
  - 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
    - If one global memory access is needed for every 4 instructions
    - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

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