L1: Introduction
CS 6235: Parallel Programming for Many-Core Architectures

January 7, 2013

Outline of Today’s Lecture

- Introductory remarks
- A brief motivation for the course
- Course plans
- Introduction to CUDA
  - Motivation for programming model
  - Presentation of syntax
  - Simple working example (also on website)
- Reading:
  - CUDA 5 Manual, particularly Chapters 2 and 4
  - Programming Massively Parallel Processors, Chapters 1 and 2

This lecture includes slides provided by: Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)

Course Objectives

- Learn how to program “graphics” processors for general-purpose multi-core computing applications
  - Learn how to think in parallel and write correct parallel programs
  - Achieve performance and scalability through understanding of architecture and software mapping
- Significant hands-on programming experience
  - Develop real applications on real hardware
- Discuss the current parallel computing context
  - What are the drivers that make this course timely
  - Contemporary programming models and architectures, and where is the field going

Course/CUDA Introduction

CS6235

• Website: http://www.cs.utah.edu/~mhall/cs6235s13/
• Mailing lists:
  - cs6235s13@list.eng.utah.edu for open discussions on assignments
• Professor:
  Mary Hall
  MEB 3466, mhall@cs.utah.edu, 5-1039
  Office hours: M 12:20-1:00PM, Th 11:00-11:40 AM, or by appointment
• Teaching Assistant:
  TBD
Outcomes from Previous Classes

• Paper at POPL (premier programming language conference) and Masters project
  "Poster paper at PPoPP (premier parallel computing conference)
  "Evaluating Graph Coloring on GPUs," Pascal Grosset, Feihong Zhu, Shusen Liu, Mary Hall, Suresh Venkataramanan, Poster paper, PPoPP 11, Feb. 2011.
  "Posters at Symposium on Application Accelerators for High-Performance Computing http://saahpc.ncsa.illinois.edu/09/ (late April/early May submission deadline)
  "Gagandeep S. Sachdev, Vishay Vanjani and Mary W. Hall, Poster paper, July 2010.
  "Nvidia Project + new hardware
  "Osprey: Efficient Embedded Parallel Computing Technologies" with Nvidia
  "Kepler cluster coming in February or March

Grading Criteria

• Small projects (4): 35%
• Midterm test: 15%
• Project proposal: 5%
• Project design review: 10%
• Project presentation/demo: 15%
• Project final report: 20%

Primary Grade: Team Projects

• Some logistical issues:
  - 2-3 person teams
  - Projects will start in late February

• Three parts:
  - (1) Proposal; (2) Design review; (3) Final report and demo

• Application code:
  - I will suggest a few sample projects, areas of future research interest.
  - Alternative applications must be approved by me (start early).
**Collaboration Policy**

- I encourage discussion and exchange of information between students.
- But the final work must be your own.
  - Do not copy code, tests, assignments or written reports.
  - Do not allow others to copy your code, tests, assignments or written reports.

**Lab Information**

Primary lab
- Linux lab: MEB 3161

Secondary
- Tesla S1070 system in SCI (Linux)
- Coming soon: Kepler cluster in SCI

Tertiary
- CADE machines in WEB, both Linux and Windows
- Focus of course will be on Linux, however

Interim
- First assignment can be completed on any machine running CUDA (Linux, Windows, MAC OS)
- Other assignments must use lab machines for timing

---

**Text and Notes**


4. Additional readings associated with lectures.

---

**GPGPU (General-Purpose Computing on GPUs)**

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications (data parallel)
  - Demonstrations of 30-100X speedup over CPU

- Early challenges:
  - Architectures very customized to graphics problems (e.g., vertex and fragment processors)
  - Programmed using graphics-specific programming models or libraries

- Recent trends:
  - Some convergence between commodity and GPUs and their associated parallel programming models
The Fastest Supercomputers in the World

<table>
<thead>
<tr>
<th>Name</th>
<th>Reign</th>
<th>Location</th>
<th>What processors?</th>
<th>How many?</th>
<th>How fast?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan</td>
<td>NEW</td>
<td>Oak Ridge National Laboratory, USA</td>
<td>AMD Opterons and Nvidia Keplers</td>
<td>560,640 cores, half GPUs</td>
<td>17.6 PFlops</td>
</tr>
<tr>
<td>Sequoia (IBM BG/Q)</td>
<td>2012</td>
<td>Lawrence Livermore National Laboratory, USA</td>
<td>IBM Power BGC (custom)</td>
<td>1,572,864 cores</td>
<td>16.3 PFlops</td>
</tr>
<tr>
<td>K</td>
<td>2011</td>
<td>Riken, JAPAN</td>
<td>SPARC64 processors</td>
<td>705,024 cores</td>
<td>10.5 PFlops</td>
</tr>
<tr>
<td>Tianhe-1A</td>
<td>2010</td>
<td>National Supercomputing Center, CHINA</td>
<td>Intel Xeon and Nvidia Fermis</td>
<td>186,368 cores</td>
<td>2.57 PFlops</td>
</tr>
<tr>
<td>Jaguar (Cray XT5)</td>
<td>2010</td>
<td>Oak Ridge National Laboratory, USA</td>
<td>AMD 6-core, dual-processor Opterons</td>
<td>~37,000 processor chips (224,162 cores)</td>
<td>1.76 PFlops</td>
</tr>
<tr>
<td>RoadRunner</td>
<td>2009</td>
<td>Los Alamos National Laboratory, USA</td>
<td>AMD Opterons, IBM Cell/BE (Playstations)</td>
<td>~19,000 processor chips (129,600 cores)</td>
<td>1.1 PFlops</td>
</tr>
</tbody>
</table>

See www.top500.org

What Makes a Parallel Programming Model Successful for High-End Computing

- Exposes architecture's execution model, the principles of execution and what operations are supported well
- Must be possible to achieve high performance, even if it is painful
- Portable across platforms
- Easy migration path for existing applications, so nearby current approaches

What Makes a Parallel Programming Model Successful for the Masses

- Productivity
  - Programmer can express parallelism at a high level
  - Correctness is not difficult to achieve
- Portable across platforms
- Performance gains over sequential easily achievable

Future Parallel Programming

- It seems clear that for the next decade architectures will continue to get more complex, and achieving high performance will get harder.
- Most people in the research community agree that different kinds of parallel programmers will be important to the future of computing:
  - Programmers that understand how to write software, but are naive about parallelization and mapping to architecture (Joe programmers)
  - Programmers that are knowledgeable about parallelization, and mapping to architecture, so can achieve high performance (Stephanie programmers)
  - Intel/Microsoft say there are three kinds (Mort, Elvis and Einstein)
- Programming abstractions will get a whole lot better by supporting specific users.
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 placement 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

Today’s Lecture

- **Goal is to enable writing CUDA programs right away**
  - Not efficient ones - need to explain architecture and mapping for that (soon)
  - Not correct ones - need to discuss how to reason about correctness (also soon)
  - Limited discussion of why these constructs are used or comparison with other programming models (more as semester progresses)
  - Limited discussion of how to use CUDA environment (more next week)
  - No discussion of how to debug. We’ll cover that as best we can during the semester.

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)`
  - E.g. `__syncthreads()`

Minimal Extensions to C + API

- **Declspecs**
  - `__device__` float filter[N];
  - `__global__` void convolve (float *image)
    - `__shared__` float region[N];
      - `__syncthreads()`
- **Keywords**
  - threadIdx, blockIdx
  - `region[threadIdx] = image[i];`
- **Intrinsics**
  - `__syncthreads()`
    - `image[] = result;`
- **Runtime API**
  - `malloc`, `free`, `cudaMemcpy`, `cudaMemGetInfo`
    - `// Allocate GPU memory`
      - `void *myimage = cudaMalloc(bytes);`
      - `// 100 blocks, 10 threads per block`
        - `convolve<<<100, 10>>>(myimage);`
NVCC Compiler’s Role: Partition Code and Compile for Device

mycode.cu

Compiled by native compiler: gcc, icc, cc

Compiled by nvcc compiler

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, threadIdx.y)

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

Thread Batching: Grids and Blocks

Block and Thread IDs

Device

Host

Kernel 1

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

Kernel 2

Block (1,1) Block (2,1) Block (3,1)

Grid 1

Device

Host

Kernel 1

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

Kernel 2

Block (1,1) Block (2,1) Block (3,1)

Grid 1

Thread ID: (threadIdx.x, threadIdx.y)
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 "6" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

Known as a cyclic data distribution

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
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;
    ...
}

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

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 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 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
**Global Function**

**GLOBAL FUNCTION:**
Thread scans subset of array elements
Call device function to compare with "6"
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++)
        d_out[threadIdx.x] += compare(d_in[i*BLOCKSIZE + threadIdx.x], 6);
}
```

**Device Function**

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

```c
__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 \( \rightarrow \) 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)

Next Time

• Hardware Execution Model