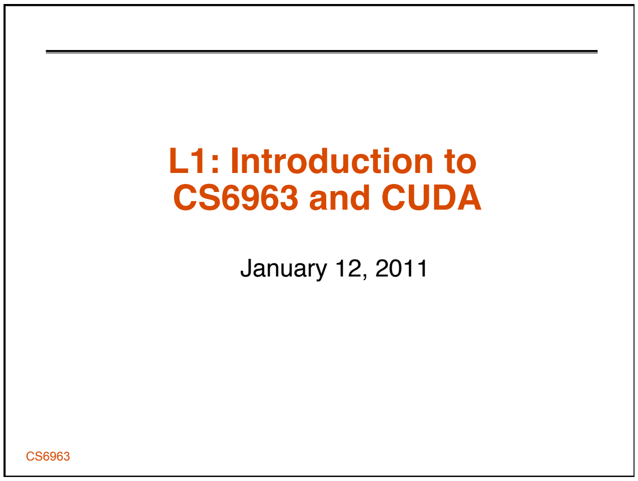
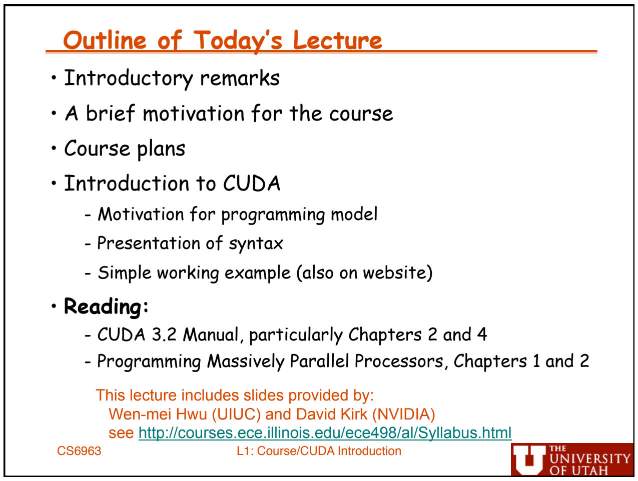


L1: Introduction to CS6963 and CUDA

January 12, 2011

CS6963

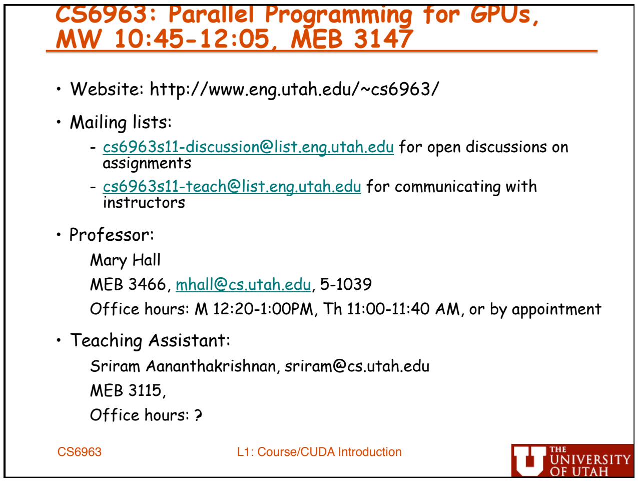


Outline of Today's Lecture

- 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 3.2 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)
see http://courses.ece.illinois.edu/ece498/al/Syllabus.html

CS6963 L1: Course/CUDA Introduction



CS6963: Parallel Programming for GPUs, MW 10:45-12:05, MEB 3147

- Website: http://www.eng.utah.edu/~cs6963/
- Mailing lists:
  - cs6963s11-discussion@list.eng.utah.edu for open discussions on assignments
  - cs6963s11-teach@list.eng.utah.edu for communicating with instructors
- Professor:

  Mary Hall

  MEB 3466, mhall@cs.utah.edu, 5-1039

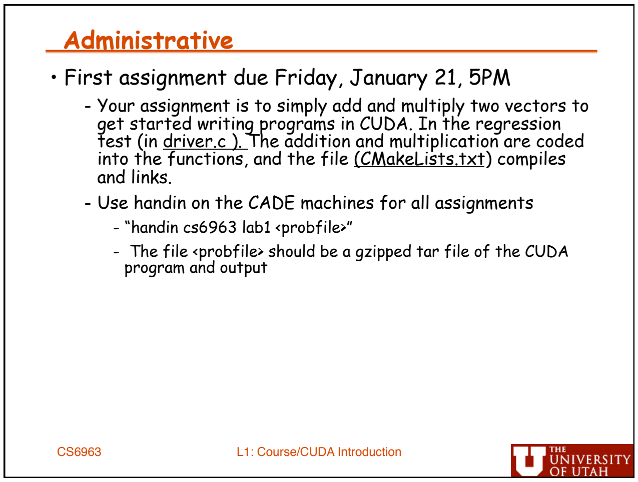
  Office hours: M 12:20-1:00PM, Th 11:00-11:40 AM, or by appointment
- Teaching Assistant:

  Sriram Aananthakrishnan, sriram@cs.utah.edu

  MEB 3115,

  Office hours: ?

CS6963 L1: Course/CUDA Introduction



Administrative

- First assignment due Friday, January 21, 5PM
  - Your assignment is to simply add and multiply two vectors to get started writing programs in CUDA. In the regression test (in driver.c). The addition and multiplication are coded into the functions, and the file (CMakeLists.txt) compiles and links.
  - Use handin on the CADE machines for all assignments
    - "handin cs6963 lab1 <probfile>"
    - The file <probfile> should be a gzipped tar file of the CUDA program and output

CS6963 L1: Course/CUDA Introduction

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

Outcomes from 2010 Course

- Paper at POPL (premier programming language conference) and Masters project

  “EigenCFA: Accelerating Flow Analysis with GPUs.” Tarun Prabhu, Shreyas Ramalingam , Matthew Might, Mary Hall, POPL ‘11, Jan. 2011.
- Poster paper at PPoPP (premier parallel computing conference)

  “Evaluating Graph Coloring on GPUs.” Pascal Grosset, Peihong Zhu, Shusen Liu, Mary Hall, Suresh Venkatasubramanian, Poster paper, PPoPP ‘11, Feb. 2011.
- Posters at Symposium on Application Accelerators for High-Performance Computinghttp://saahpc.ncsa.illinois.edu/10/ [Early May deadline]

  “Takagi Factorization on GPU using CUDA.” Gagandeep S. Sachdev, Vishay Vanjani and Mary W. Hall, Poster paper, July 2010.

  “GPU Accelerated Particle System for Triangulated Surface MeshesBrad Peterson, Manasi Datar, Mary Hall and Ross Whitaker, Poster paper, July 2010.
- Nvidia Project + new hardware
  - “Echelon: Extreme-scale Compute Hierarchies with Efficient Locality-Optimized Nodes
  - In my lab, GTX 480 and C2050 (Fermi)

Outcomes from 2009 Course

- Paper and poster at Symposium on Application Accelerators for High-Performance Computing http://saahpc.ncsa.illinois.edu/09/ (late April/early May submission deadline)
  - Poster:

    Assembling Large Mosaics of Electron Microscope Images using GPU - Kannan Venkataraju, Mark Kim, Dan Gerszewski, James R. Anderson, and Mary Hall
  - Paper:

    GPU Acceleration of the Generalized Interpolation Material Point Method Wei-Fan Chiang, Michael DeLisi, Todd Hummel, Tyler Prete, Kevin Tew, Mary Hall, Phil Wallstedt, and James Guilkey
- Poster at NVIDIA Research Summit http://www.nvidia.com/object/gpu_tech_conf_research_summit.html

  Poster #47 - Fu, Zhisong, *University of Utah (United States)*
  Solving Eikonal Equations on Triangulated Surface Mesh with CUDA
- Posters at Industrial Advisory Board meeting
- Integrated into Masters theses and PhD dissertations
- Jobs and internships

Grading Criteria

• Homeworks and mini-projects (4):	30%
• Midterm test:	15%
• Project proposal:	10%
• 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).

CS6963 L1: Course/CUDA Introduction

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.

CS6963 L1: Course/CUDA Introduction

Lab Information

Primary lab

- Linux lab: LOCATION

Secondary

- Tesla S1070 system in SCI (Linux)

Tertiary

- Windows machines in WEB, (lab5/lab6)
- Focus of course will be on Linux, however

Interim

- Until we get to timing experiments, assignments can be completed on any machine running CUDA 3.2 (Linux, Windows, MAC OS)

CS6963 L1: Course/CUDA Introduction

A Few Words About Tesla System

Nvidia Tesla system:
240 cores per chip, 960 cores per unit, 32 units.

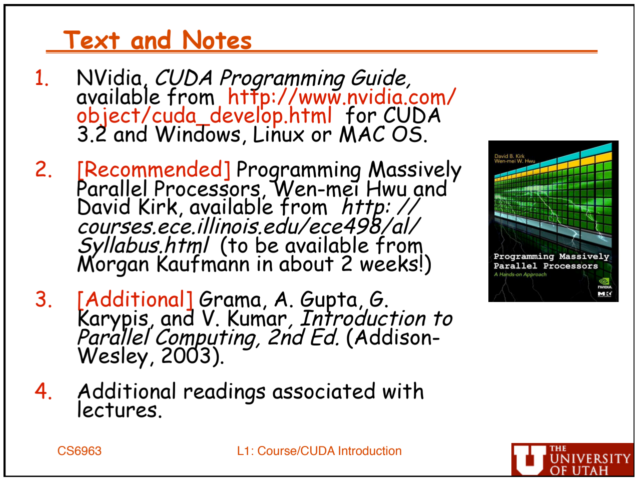
Over 30,000 cores!

Hosts are Intel Nehalems

PCI+MPI between units

NVIDIA Recognizes University Of Utah As A Cuda Center Of Excellence
University of Utah is the Latest in a Growing List of Exceptional Schools Demonstrating Pioneering Work in Parallel (JULY 31, 2008—NVIDIA Corporation)

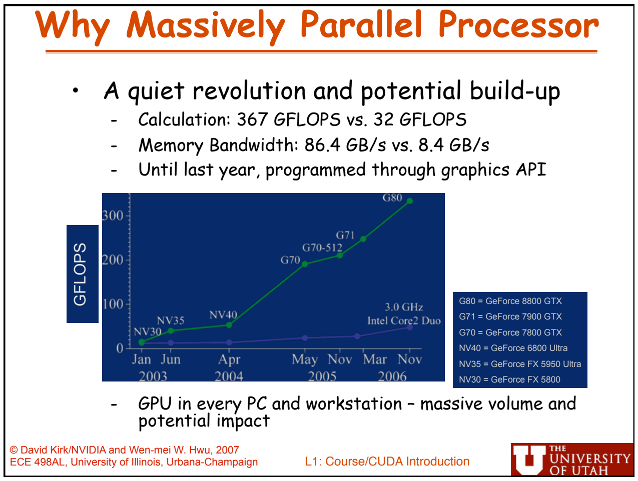
CS6963 L1: Course/CUDA Introduction



Text and Notes

1. NVidia, *CUDA Programming Guide*, available from http://www.nvidia.com/object/cuda_develop.html for CUDA 3.2 and Windows, Linux or MAC OS.
2. [Recommended] Programming Massively Parallel Processors, Wen-mei Hwu and David Kirk, available from *http://courses.ece.illinois.edu/ece498/al/Syllabus.html* (to be available from Morgan Kaufmann in about 2 weeks!)
3. [Additional] Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing, 2nd Ed.* (Addison-Wesley, 2003).
4. Additional readings associated with lectures.

CS6963 L1: Course/CUDA Introduction

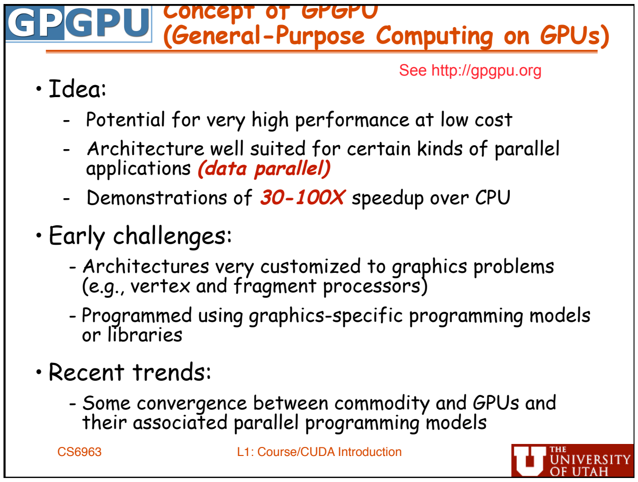


Why Massively Parallel Processor

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign L1: Course/CUDA Introduction

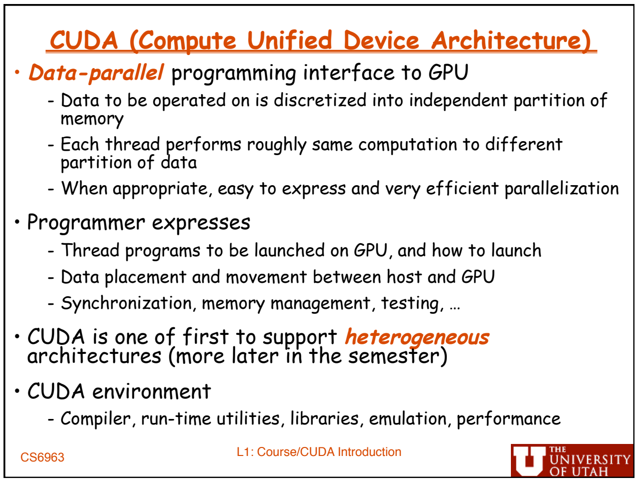


GPGPU Concept of GPGPU (General-Purpose Computing on GPUs)

See http://gpgpu.org

- 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

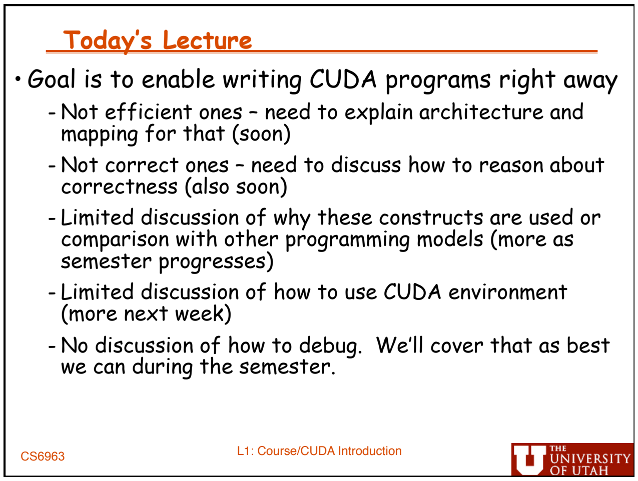
CS6963 L1: Course/CUDA Introduction



CUDA (Compute Unified Device Architecture)

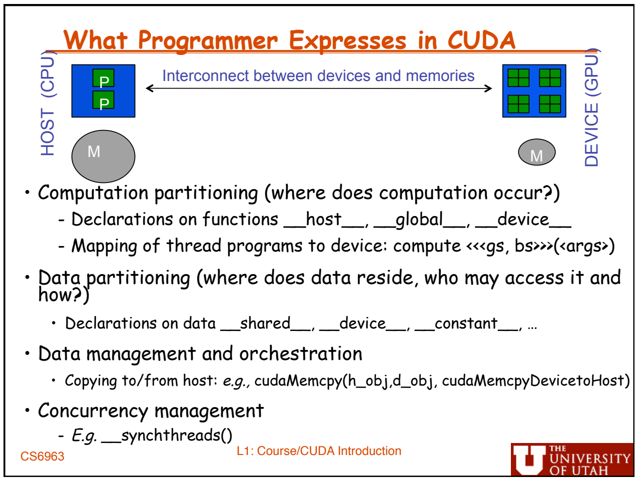
- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - 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

CS6963 L1: Course/CUDA Introduction



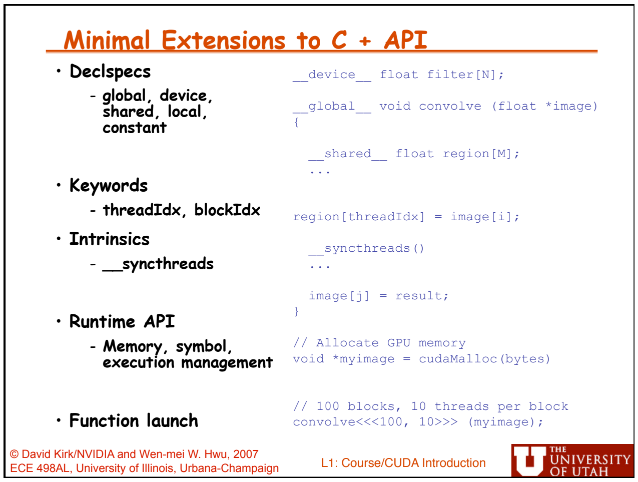
Today's Lecture

- 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

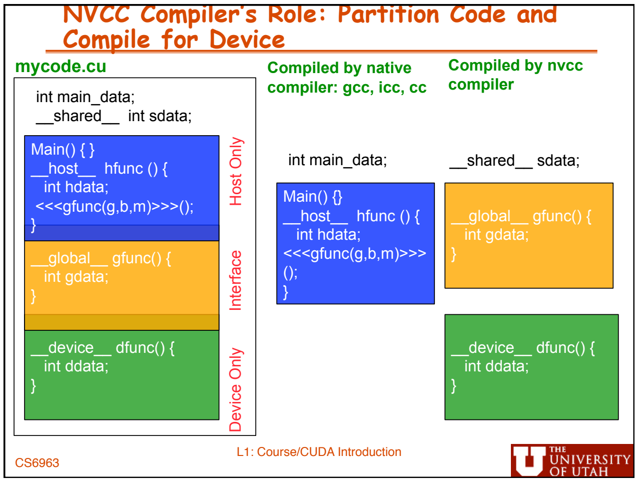
- Computation partitioning (where does computation occur?)
  - Declarations on functions __host__, __global__, __device__
  - Mapping of thread programs to device: compute <<<gs, bs>>>(<args>)
- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data __shared__,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



Minimal Extensions to C + API

- **Declspecs**
  - **global, device, shared, local, constant**
- **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];
  ...

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);
```



NVCC Compiler's Role: Partition Code and Compile for Device

mycode.cu

```
int main_data;
__shared__  int sdata;

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

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```

Compiled by native compiler: gcc, icc, cc

```
int main_data;

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

Compiled by nvcc compiler

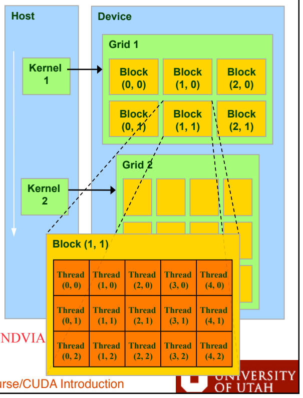
```
__shared__ sdata;

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```

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

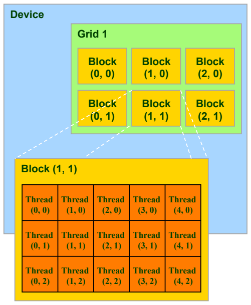
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared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



Courtesy: NDVIA

Block and Thread IDs

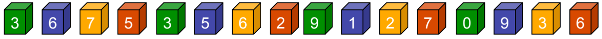
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

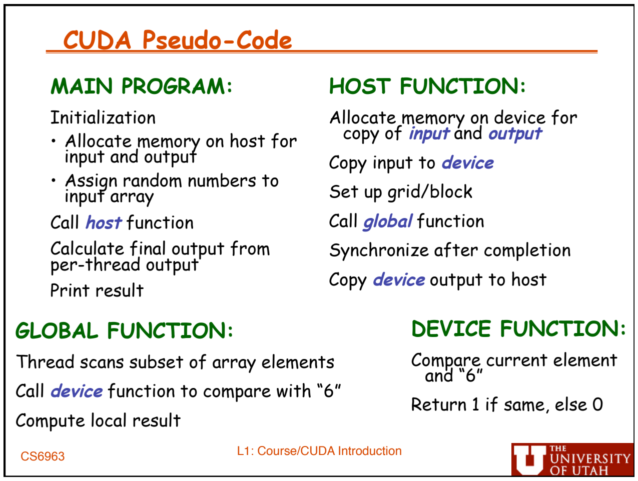
Simple working code example

- 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



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:

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

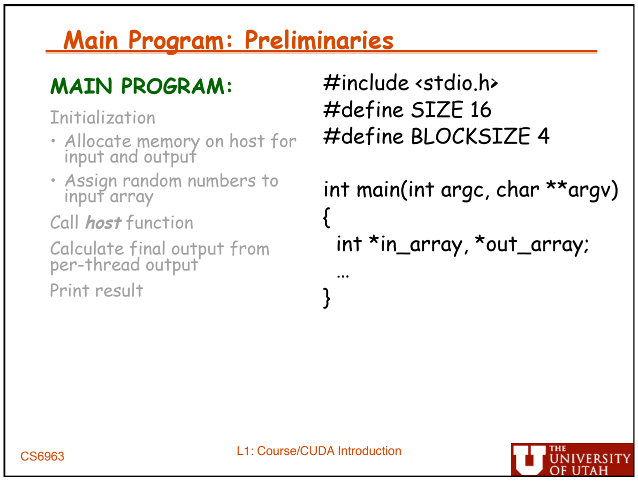
Call *device* function to compare with “6”

Compute local result

DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0



Main Program: Preliminaries

MAIN PROGRAM:

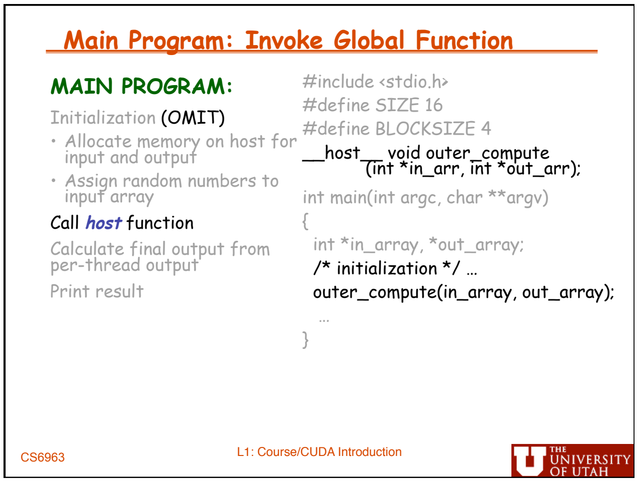
Initialization
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```
#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

MAIN PROGRAM:

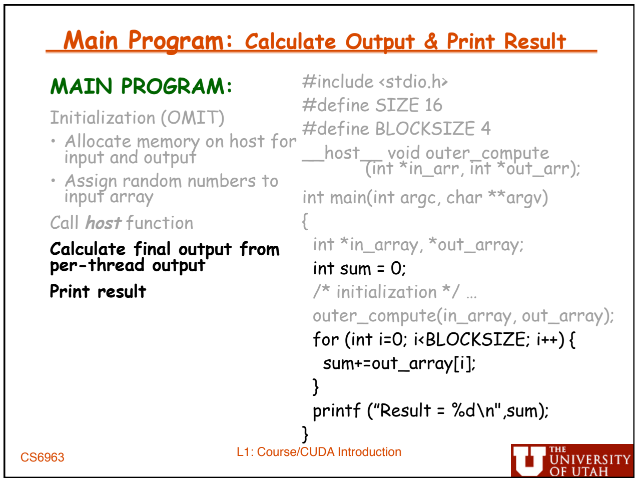
Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

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)
{
  int *in_array, *out_array;
  /* initialization */ …
  outer_compute(in_array, out_array);
  …
}
```



Main Program: Calculate Output & Print Result

MAIN PROGRAM:

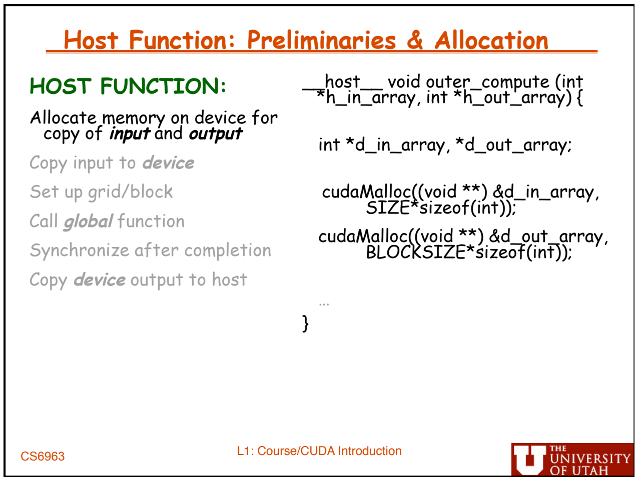
Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

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)
{
  int *in_array, *out_array;
  int 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***

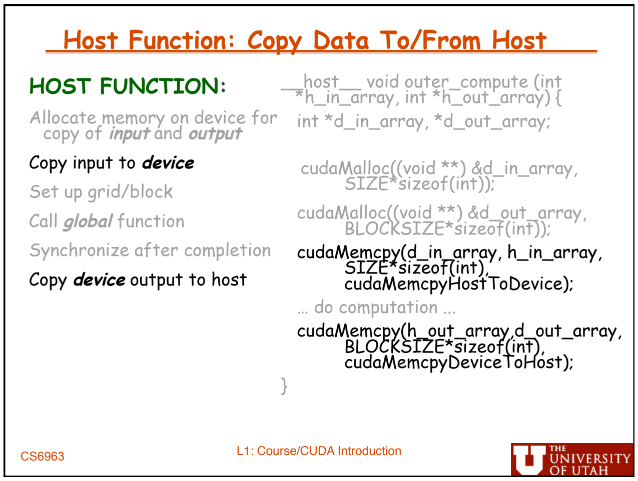
Set up grid/block

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));

 …
}
```



Host Function: Copy Data To/From Host

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

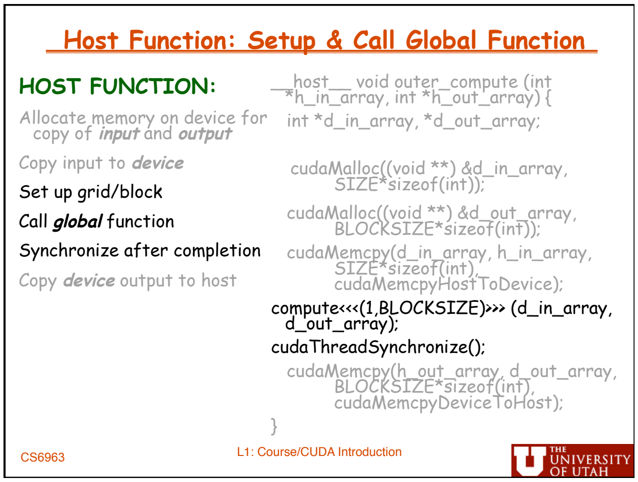
Set up grid/block

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;

  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***

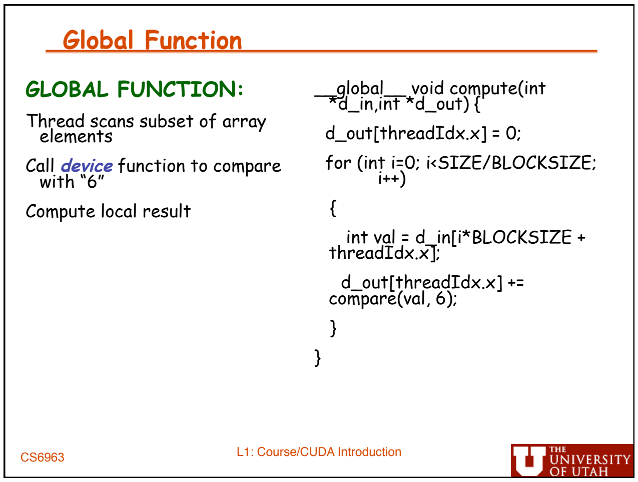
Set up grid/block

Call ***global*** function

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);
compute<<<(1,BLOCKSIZE)>>> (d_in_array,
 d_out_array);
cudaThreadSynchronize();
 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 "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

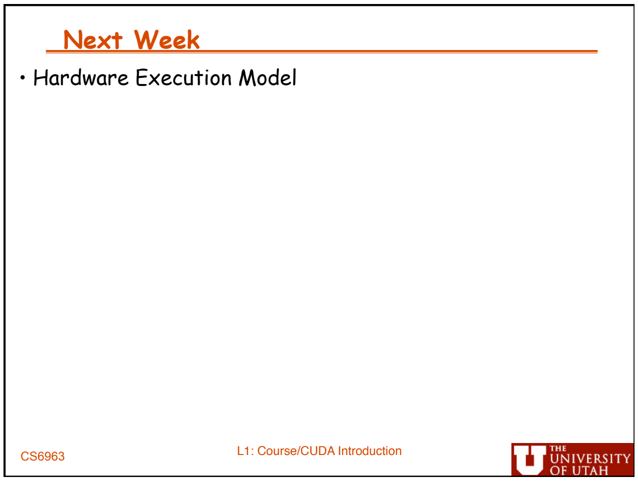
- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure ➔ computed result (perhaps single number) **[dimensionality reduced]**
- Why 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 "embarassingly 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 partititioning (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 Week

- Hardware Execution Model