Objective

- To understand the major elements of a CUDA program
- Introduce the basic constructs of the programming model
- Illustrate the preceding with a simple but complete CUDA program
Reading Assignment

• Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 3
• CUDA Programming Guide

CUDA/OpenCL- Execution Model

• Reflects a multicore processor + GPGPU execution model
• Addition of device functions and declarations to stock C programs
• Compilation separated into host and GPU paths
CUDA /OpenCL – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

```c
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);
```

```c
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```
CUDA /OpenCL - Programming Model

Note: Each thread executes the same kernel code!

Vector Addition – Conceptual View

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2011
ECE408 / CS483, University of Illinois, Urbana-Champaign
A Simple Thread Block Model

- Structure an array of threads into **thread blocks**
- A unique id can be computed for each thread
- This id is used for workload partitioning

\[ id = blockIdx \times blockDim + threadIdx; \]

Using Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions

\[ i = blockIdx.x \times blockDim.x + threadIdx.x; \]
\[ C_d[i] = A_d[i] + B_d[i]; \]
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate (only through global memory)

\[
i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x};
\]

\[
C_d[i] = A_d[i] + B_d[i];
\]

---

**blockIdx and threadIdx**

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
Data Partitioning

Partition multidimensional data across a multidimensional grid of threads

Vector Addition – Traditional C Code

```c
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

© David Kirk / NVIDIA and Wen-mei W. Hwu, 2007-2011
ECE408 / CS483, University of Illinois, Urbana-Champaign
Heterogeneous Computing vecAdd

Host Code

```c
#include <cuda.h>

void vecAdd(float* A, float* B, float* C, int n)
{
    // Local code
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...

    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – to have the device
    // to perform the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

Partial Overview of CUDA Memories

- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.
CUDA Device Memory Management API functions

- `cudaMalloc()`
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- `cudaFree()`
  - Frees object from device global memory
  - **Pointer** to freed object

Host-Device Data Transfer API functions

- `cudaMemcpy()`
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous
void vecAdd(float* A, float* B, float* C, int n) {
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

    // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &C_d, size);

    // Kernel invocation code – to be shown later

    // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n) {
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}

Example: Vector Addition Kernel

// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n) {
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}
Example: Vector Addition Kernel

```c
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

```c
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
```

More on Kernel Launch

```c
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);
    if (n%256) DimGrid.x++;
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
__global__
void vecAddKernel(float *A_d,
float *B_d, float *C_d, int n) {
    int i = blockIdx.x * blockDim.x
    + threadIdx.x;
    if (i < n) C_d[i] = A_d[i] + B_d[i];
}

__host__
Void vecAdd() {
    dim3 DimGrid = (ceil(n/256,1,1);
    dim3 DimBlock = (256,1,1);
    vecAddKernel<<<DimGrid,DimBlock>>>(
        A_d, B_d, C_d, n);
}

Kernel Execution in a Nutshell

More on CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function
- Each “__” consists of two underscore characters
- A kernel function must return void
- __device__ and __host__ can be used together
Compiling A CUDA Program
Integrated C programs with CUDA extensions

NVCC Compiler

Host Code
Host C Compiler/ Linker

Device Code (PTX)
Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs

The ISA

• An Instruction Set Architecture (ISA) is a contract between the hardware and the software.

• As the name suggests, it is a set of instructions that the architecture (hardware) can execute.
PTX ISA – Major Features

• Set of predefined, read only variables
  – E.g., %tid, %ntid, %ctaid, etc.
  – Some of these are multidimensional
    • E.g., %tid.x, %tid.y, %tid.z
  – Includes architecture variables
    • E.g., %laneid, %warpid
• Can be used for auto-tuning of code
• Includes undefined performance counters

PTX ISA – Major Features (2)

• Multiple address spaces
  – Register, parameter
  – Constant global, etc.
  – More later
• Predicated instruction execution
• More in PTX ISA v4.2
QUESTIONS?