ECE 8823A
GPU Architectures

Module 2:
Introduction to CUDA C

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
Compiling A CUDA Program

Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Device Code (PTX)

Host C Compiler/Linker

Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs

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

Serial Code (host)

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

Serial Code (host)

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

Note: Each thread executes the same kernel code!
A Simple Thread Block Model

id = blockIdx * blockDim + threadIdx;

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

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

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

```c
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

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);
}
```
Heterogeneous Computing vecAdd
Host Code

```c
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

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

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

Part 1

Host Memory

Device Memory

CPU

GPU

Part 2

Part 3

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 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;

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

2. // Kernel invocation code – to be shown later

3. // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    cudaMemcpy((void**) &C_d, C, size);
    cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}

Example: Vector Addition Kernel

// 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

// 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 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);
}

• Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit
  synch needed for blocking \( \rightarrow \) cudaDeviceSynchronize();
Kernel Execution in a Nutshell

```c
__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];
}
```

```c
__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);
}
```

More on CUDA Function Declarations

| __device__ float DeviceFunc() | Executed on: device | Only callable from: device |
| __global__ void KernelFunc()  |                      | device                      |
| __host__ float HostFunc()     |                      | host                        |

- **__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
Device Code (PTX)

Host C Compiler/
Linker
Device Just-in-Time
Compiler

Heterogeneous Computing Platform with
CPUs, GPUs

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

CPU and GPU Address Spaces

• Requires explicit management in each
  address space
• Programmer initiated transfers between
  address spaces
Unified Memory

- Single unified address space across CPU and GPU
- All transfers managed under the hood
- Smaller code segments
- Explicit management now becomes an optimization

Using Unified Memory

```cpp
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void) {
    int N = 1<<20;
    float *x, *y;
    // Allocate Unified Memory – accessible from CPU or GPU
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // Initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add<<<1, 1>>>(N, x, y);
    // Wait for GPU to finish
    cudaDeviceSynchronize();

    // Free memory
    cudaFree(x);
    cudaFree(y);
    return 0;
}
```

From https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/
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 (threadIdx), %ntid (blockDim), %ctaid (blockIdx), %nctaid(gridDim), 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

Versions

• Compute capability
  – Architecture specific, e.g., Volta is 7.0
• CUDA version
  – Determines programming model features
  – Currently 9.0
• PTX version
  – Virtual ISA version
  – Currently 6.1
QUESTIONS?