CSCI-GA.3033-004
Graphics Processing Units (GPUs): Architecture and Programming
Lecture 3: Introduction to CUDA

Some slides here are adopted from:
• NVIDIA teaching kit

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
GPU Computing Applications

<table>
<thead>
<tr>
<th>Libraries and Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuFFT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

CUDA-enabled NVIDIA GPUs

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Series</th>
<th>Series</th>
<th>Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>GeForce 1000 Series</td>
<td>Quadro P Series</td>
<td>Tesla P Series</td>
</tr>
<tr>
<td>Maxwell</td>
<td>GeForce 900 Series</td>
<td>Quadro M Series</td>
<td>Tesla M Series</td>
</tr>
<tr>
<td>Kepler</td>
<td>GeForce 700 Series, GeForce 600 Series</td>
<td>Quadro K Series</td>
<td>Tesla K Series</td>
</tr>
<tr>
<td>Fermi</td>
<td>GeForce 500 Series, GeForce 400 Series</td>
<td>Quadro Fermi Series</td>
<td>Tesla 20 Series</td>
</tr>
</tbody>
</table>

Entertainment | Professional Graphics | High Performance Computing |
3 Ways to Accelerate Applications

- Libraries: Easy to use, Most Performance
- Compiler Directives: Easy to use, Portable code
- Programming Languages: Most Performance, Most Flexibility
Parallel Computing on a GPU

- GPUs deliver up to 8,800+ GFLOPS
  - Available in laptops, desktops, and clusters

- GPU parallelism is doubling almost every year

- Programming model scales transparently
  - Data parallelism

- Programmable in C (and other languages) with CUDA tools

- Multithreaded SPMD model uses application data parallelism and thread parallelism.
  
  [SPMD = Single Program Multiple Data]
<table>
<thead>
<tr>
<th>GPU</th>
<th>Cores</th>
<th>Cores/SM</th>
<th>SM</th>
<th>Compute Capab.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 980</td>
<td>2048</td>
<td>128</td>
<td>16</td>
<td>5.2</td>
</tr>
<tr>
<td>GTX Titan</td>
<td>2688</td>
<td>192</td>
<td>14</td>
<td>3.5</td>
</tr>
<tr>
<td>GTX 780</td>
<td>2304</td>
<td>192</td>
<td>12</td>
<td>3.5</td>
</tr>
<tr>
<td>GTX 770</td>
<td>1536</td>
<td>192</td>
<td>8</td>
<td>3.0</td>
</tr>
<tr>
<td>GTX 760</td>
<td>1152</td>
<td>192</td>
<td>6</td>
<td>3.0</td>
</tr>
<tr>
<td>GTX 680</td>
<td>1536</td>
<td>192</td>
<td>8</td>
<td>3.0</td>
</tr>
<tr>
<td>GTX 670</td>
<td>1344</td>
<td>192</td>
<td>7</td>
<td>3.0</td>
</tr>
<tr>
<td>GTX 580</td>
<td>512</td>
<td>32</td>
<td>16</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Source: *Multicore and GPU Programming: An Integrated Approach* by G. Barlas

Latest compute capability (so far): 7.x
CUDA

- **Compute Unified Device Architecture**
- **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);
Parallel Threads

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

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

...
Thread Blocks

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization, ...
  - Threads in different blocks cannot cooperate

\[
\begin{align*}
    i &= \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x;} \\
    C_d[i] &= A_d[i] + B_d[i];
\end{align*}
\]

```cpp
for (int i = 0; i < blockDim.x; i++) {
    C_d[i] = A_d[i] + B_d[i];
}
```
Kernel

- Launched by the host
- Very similar to a C function
- To be executed on device
- All threads will execute that same code in the kernel.

Grid

- 1D or 2D (or 3D) organization of a block
- blockDim.x, blockDim.y, and blockDim.z
- gridDim.x, gridDim.y, and gridDim.z

Block

- 1D, 2D, or 3D organization of a block
- Block is assigned to an SM
- blockIdx.x, blockIdx.y, and blockIdx.z

Thread

- threadIdx.x, threadIdx.y, and threadIdx.z
GPU Allows Automatic Scalability

Assuming you have enough parallelism
Heterogeneous Computing

Diagram showing C Program Sequential Execution and Parallel kernels with grid and block structures.
**IDs**

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D, or 3D
  - Thread ID: 1D, 2D, or 3D

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

Courtesy: NDVIA
A Simple Example: Vector Addition

Vector A


Vector B


Vector C

A Simple Example: Vector Addition

// 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);
}
A Simple Example: Vector Addition

```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;
    ...
  1. // Allocate device memory for A, B, and C
     // copy A and B to device memory

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

  3. // copy C from the device memory
     // Free device vectors
}
```
CUDA Memory Model

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- **Device code can:**
  - R/W per-thread registers
  - R/W per-grid global memory
- We will cover more later
CPU & GPU Memory

• In CUDA, host and devices have separate memory spaces.
  – But ... Wait for lectures on advanced techniques

• If GPU and CPU are on the same chip, then they share memory space → fusion
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - Address of a pointer to the allocated object
    - **Size of** of allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
CUDA Device Memory Allocation

Example:

WIDTH = 64;
float * Md;
ing size = WIDTH * sizeof(float);
cudaMalloc((void**)&Md, size);
cudaFree(Md);
CUDA Device Memory Allocation

• `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    • Pointer to destination
    • Pointer to source
    • Number of bytes copied
    • Type of transfer
      – Host to Host
      – Host to Device
      – Device to Host
      – Device to Device

• Asynchronous transfer
CUDA Device Memory Allocation

Example:

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```
A Simple Example: Vector Addition

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);

    // Allocate device memory for C_d
    cudaMemcpy((void **) &C_d, C, size, cudaMemcpyDeviceToHost);

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

3. // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
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);
}

__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];
}

// Each thread performs one pair-wise addition

Unique ID
Unique ID
1D grid of 1D blocks

blockIdx.x * blockDim.x + threadIdx.x;
Unique ID
1D grid of 2D blocks

\[
\text{blockIdx.x} \times \text{blockDim.x} \times \text{blockDim.y} + \\
\text{threadIdx.y} \times \text{blockDim.x} + \\
\text{threadIdx.x};
\]
Unique ID
1D grid of 3D blocks

blockIdx.x * blockDim.x * blockDim.y * blockDim.z +
threadIdx.z * blockDim.y * blockDim.x +
threadIdx.y * blockDim.x +
threadIdx.x;
**Unique ID**

**2D grid of 1D blocks**

```c
int blockId = blockIdx.y * blockDim.x + blockIdx.x;

int threadId = blockId * blockDim.x + threadIdx.x;
```
Unique ID
2D grid of 2D blocks

```c
int blockIdx = blockIdx.x + blockIdx.y * gridDim.x;

int threadIdx = blockIdx * (blockDim.x * blockDim.y) +
(threadIdx.y * blockDim.x) + threadIdx.x;
```
Unique ID
2D grid of 3D blocks

```c
int blockId = blockIdx.x + blockIdx.y * gridDim.x;

int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z) +
    (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;
```
Unique ID
3D grid of 1D blocks

int blockId = blockIdx.x
    + blockIdx.y * gridDim.x
    + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockDim.x * blockDim.x + threadIdx.x;
Unique ID
3D grid of 2D blocks

```c
int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;
```
int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;
The Hello World of Parallel Programming: Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- **__global__** defines a kernel function. Must return **void**
- **__device__** and **__host__** can be used together

For functions executed on the device:
- No recursion ... for now ... (but wait till we talk about dynamic parallelism)
- No static variable declarations inside the function
- No indirect function calls through pointers
Note about __device__

- Kernel calls another kernel → Dynamic parallelism
- Needs compute capability 3.5 and higher.
- From second generation of Kepler
- Nested parallelism continues till depth 24.
- All child launches must complete in order for the parent kernel to be seen as completed.
- More details later ….
The Hello World of Parallel Programming: Matrix Multiplication

Data Parallelism:
We can safely perform many arithmetic operations on the data structures in a simultaneous manner.
The Hello World of Parallel Programming: Matrix Multiplication

C adopts raw-major placement approach when storing 2D matrix in linear memory address.
The Hello World of Parallel Programming: **Matrix Multiplication**

```c
int main(void) {
    1. // Allocate and initialize the matrices M, N, P
       // I/O to read the input matrices M and N
       ....

    2. // M * N on the device
       MatrixMultiplication(M, N, P, Width);

    3. // I/O to write the output matrix P
       // Free matrices M, N, P
       ...
       return 0;
}
```

A Simple main function: executed at the host
The Hello World of Parallel Programming: Matrix Multiplication

// Matrix multiplication on the (CPU) host
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
The Hello World of Parallel Programming: Matrix Multiplication

```c
void MatrixMultiplication(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

    // Transfer M and N to device memory
    cudaMalloc((void**) &Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc((void**) &Pd, size);

    MatrixMulKernel(Md, Nd, Pd, Width);

    // Transfer P from device to host
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```
The Hello World of Parallel Programming: Matrix Multiplication

// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}

The Kernel Function
More On Specifying Dimensions

// Setup the execution configuration
\texttt{dim3} dimGrid(x, y, z);
\texttt{dim3} dimBlock(x, y, z);

// Launch the device computation threads!
MatrixMulKernel\texttt{<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width)};

Important:
• \texttt{dimGrid} and \texttt{dimBlock} are user defined
• \texttt{gridDim} and \texttt{blockDim} are built-in predefined variable accessible in kernel functions
The above thread configuration is launched via:

```
    dim3 block(3,2);
    dim3 grid(4,3,2);
    foo<<<grid, block>>>();
```

The `<<<>>>` part of the launch statement, is called the **execution configuration**.

Source: *Multicore and GPU Programming: An Integrated Approach* by G. Barlas
Be Sure To Know:

• Maximum dimensions of a block
• Maximum number of threads per block
• Maximum dimensions of a grid
• Maximum number of blocks per thread
<table>
<thead>
<tr>
<th>Item</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. number of grid dimensions</td>
<td>1.x</td>
</tr>
<tr>
<td>Grid maximum x-dimension</td>
<td>$2^{16} - 1$</td>
</tr>
<tr>
<td>Grid maximum y/z-dimension</td>
<td>$2^{16} - 1$</td>
</tr>
<tr>
<td>Max. number of block dimensions</td>
<td>3</td>
</tr>
<tr>
<td>Block max. x/y-dimension</td>
<td>512</td>
</tr>
<tr>
<td>Block max. z-dimension</td>
<td></td>
</tr>
<tr>
<td>Max. threads per block</td>
<td>512</td>
</tr>
<tr>
<td>GPU example (GTX family chips)</td>
<td>8800</td>
</tr>
</tbody>
</table>

Source: *Multicore and GPU Programming: An Integrated Approach* by G. Barlas
Tools

Integrated C programs with CUDA extensions
(*.cu files)

NVCC Compiler

Host Code

Device Code (PTX)

Host C Compiler/ Linker

Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs
Conclusions

• Data parallelism is the main source of scalability for parallel programs
• Each CUDA source file can have a mixture of both host and device code.
• What we learned today about CUDA:
  – `KernelA<<< nBlk, nTid >>>`(args)
  – `cudaMalloc()`
  – `cudaFree()`
  – `cudaMemcpy()`
  – `gridDim` and `blockDim`
  – `threadIdx.x` and `threadIdx.y`
  – `dim3`