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

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

- GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling 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
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

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

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

```c
i = blockIdx.x * blockDim.x + threadIdx.x;
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 and blockDim.y
  • gridDim.x and gridDim.y

Block
• 1D, 2D, or 3D organization of a block
  • blockIdx.x, blockIdx.y, and blockIdx.z

Thread
• threadIdx.x, threadIdx.y, and threadIdx.z
**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


\[ A[0] + B[0] + C[0] = C[0] \]
\[ A[N-1] = C[N-1] \]
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

#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 lecture 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** allocated object

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

Example:

WIDTH = 64;
float * Md;
int 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:

\[
\text{cudaMemcpy}(\text{Md}, M, \text{size}, \text{cudaMemcpyHostToDevice});
\]

\[
\text{cudaMemcpy}(M, \text{Md}, \text{size}, \text{cudaMemcpyDeviceToHost});
\]
A Simple Example: Vector Addition

```c
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
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for C_d
    cudaMalloc((void **) &C_d, size);

    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);
}
```

How to launch a kernel?
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);
}

// 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; // Unique ID
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
Unique ID
1D grid of 1D blocks

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

blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;

A Block
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 blockIdx = blockIdx.y * blockDim.x + blockIdx.x;

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

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

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

```
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

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

```cpp
int threadId = blockId * 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;
```
Unique ID

3D grid of 3D blocks

```c
int blockId = blockIdx.x
             + blockIdx.y * blockDim.x
             + blockDim.x * blockDim.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><code>__device__</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><code>__global__</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><code>__host__</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float HostFunc()</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 (but wait till we talk about dynamic parallelism)
  - No static variable declarations inside the function
  - No indirect function calls through pointers
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
// 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

```c
// 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;

    // P value 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
```c
dim3 dimGrid(x, y, z);
dim3 dimBlock(x, y, z);
```

// Launch the device computation threads!
```c
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Important:
- `dimGrid` and `dimBlock` are user defined
- `gridDim` and `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>1.x</th>
<th>2.x</th>
<th>3.x</th>
<th>5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. number of grid dimensions</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Grid maximum x-dimension</td>
<td>$2^{16} - 1$</td>
<td>$2^{31} - 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid maximum y/z-dimension</td>
<td></td>
<td></td>
<td>$2^{16} - 1$</td>
<td></td>
</tr>
<tr>
<td>Max. number of block dimensions</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Block max. x/y-dimension</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block max. z-dimension</td>
<td></td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Max. threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU example (GTX family chips)</td>
<td>8800</td>
<td>480</td>
<td>780</td>
<td>980</td>
</tr>
</tbody>
</table>

Source: *Multicore and GPU Programming: An Integrated Approach* by G. Barlas
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
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()
  – blockDim and blockDim
  – threadIdx.x and threadIdx.y
  – dim3