Hardware/Software Co-Design

Introduction to CUDA

Miaoqing Huang
University of Arkansas
Fall 2011
1. Evolutions of NVIDIA GPU

2. CUDA Basic

3. Detailed Steps
   - Device Memories and Data Transfer
   - Kernel Functions and Threading
Outline

1. Evolutions of NVIDIA GPU
2. CUDA Basic
3. Detailed Steps
   - Device Memories and Data Transfer
   - Kernel Functions and Threading
128 streaming processors in 16 streaming multiprocessors
Evolvements of NVIDIA GPU CUDA Basic

Architecture of GT200 GPU (e.g., GeForce GTX 295)

- 240 streaming processors in 30 streaming multiprocessors
Architecture of Fermi GPU (e.g., GeForce GTX 480)

- 512 Fermi streaming processors in 16 streaming multiprocessors
## Comparison among Three Architectures

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Capability</td>
<td>None</td>
<td>30 FMAs/clk</td>
<td>256 FMAs/clk</td>
</tr>
<tr>
<td>Single Precision Capability</td>
<td>128 MADs/clk</td>
<td>240 MADs/clk</td>
<td>512 FMAs/clk</td>
</tr>
<tr>
<td>Special Function Units / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Shared Memory / SM</td>
<td>16KB</td>
<td>16KB</td>
<td>48KB/16KB</td>
</tr>
<tr>
<td>L1 Cache / SM</td>
<td>None</td>
<td>None</td>
<td>16KB/48KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768KB</td>
</tr>
<tr>
<td>ECC Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Multiply-Add (MAD)

\[
A \times B = \text{Product} \quad \text{(truncate extra digits)}
\]

\[
C = \text{Result}
\]

### Fused Multiply-Add (FMA)

\[
A \times B = \text{Product} \quad \text{(retain all digits)}
\]

\[
C = \text{Result}
\]
Compute Capability

- The *compute capability* of a device is defined by a major revision number and a minor revision number.
- Devices with the same major revision number are of the same core architecture:
  - Fermi architecture: 2.x
  - Prior devices: 1.x
- "NVIDIA CUDA C Programming Guide"
  - Appendix A lists of all CUDA-enabled devices along with their compute capability
  - Appendix F gives the technical specifications of each compute capability
Outline

1. Evolvements of NVIDIA GPU
2. CUDA Basic
3. Detailed Steps
   - Device Memories and Data Transfer
   - Kernel Functions and Threading
Data Parallelism
Square Matrix Multiplication Example

- $P = M \times N$ of size WIDTH $\times$ WIDTH
Data Parallelism
Square Matrix Multiplication Example

- \( P = M \times N \) of size \( \text{WIDTH} \times \text{WIDTH} \)
- The approach in C

```c
for (i=0; i<\text{WIDTH}; i++) {
    for (j=0; j<\text{WIDTH}; j++) {
        ......
        P[i][j] = ........;
        ......
    }
}
```
Data Parallelism
Square Matrix Multiplication Example

- \( P = M \times N \) of size \( \text{WIDTH} \times \text{WIDTH} \)
- The approach in C

```c
for (i=0; i<\text{WIDTH}; i++) {
    for (j=0; j<\text{WIDTH}; j++) {
        ......
        P[i][j] = ......
    }
    ......
}
```

- The approach in CUDA (the straightforward way)
  - One thread calculates one element of \( P \)
  - Issue \( \text{WIDTH} \times \text{WIDTH} \) threads simultaneously
The Heterogeneous Platform

- A typical GPGPU-capable platform consists of
  - One or more microprocessors (CPUs) – *host*
  - One or more GPUs – *device*
  - GPU devices are connected to CPUs through PCIe 2.0 bus

- A CUDA program consists of
  - The code on CPU – *software part*
  - The code on GPU – *hardware part*
CUDA Program Structure

- Serial Code (host)
- Parallel Kernel (device) 
  KernelA<<< nBlk, nTid >>>(args);
- Serial Code (host)
- Parallel Kernel (device) 
  KernelB<<< nBlk, nTid >>>(args);

- Integrated host+device application C program
  - Sequential or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code
A kernel is implemented as a grid of threads

- Threads in grid is further decomposed into blocks
- A grid can be up to three-dimensional

A thread block is a batch of threads that can cooperate with each other by:
A kernel is implemented as a grid of threads

- Threads in grid is further decomposed into blocks
- A grid can be up to three-dimensional

A thread block is a batch of threads that can cooperate with each other by:

- Synchronizing their execution
- Efficiently sharing data through shared memory

A block can be one or two or three-dimensional

Each block and each thread has its own ID
Matrix Multiplication: the `main()` function

```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 processor
    MatrixMultiplication(M,N,P,width);

    // 3.
    // I/O to write the output matrix P
    // Free matrices M, N, P

    ...... 

    return 0;
}
```
Matrix Multiplication: A Simple Host Version in C

```c
void MatrixMultiplication(float* M, float* N, float* P, int width) {
    for (int i=0; i<width; i++) {
        for (int j=0; i<width; j++) {
            float sum = 0;
            for (int k=0; k<width; k++) {
                float a = M[i*width+k];
                float b = N[k*width+j];
                sum += a*b;
            }
            P[i*width+j] = sum;
        }
    }
}
```
Matrix Multiplication: Move computation to the device

```c
void MatrixMultiplication(float* M, float* N, float* P, int width)
{
    int size = width*width*sizeof(float);
    float* Md, Nd, Pd;
    ......
    // 1.
    // Allocate device memory for M, N, // and P
    // Copy M and N to allocated device // memory locations
    // 2.
    // Kernel invocation code - to have // the device to perform the actual // matrix multiplication
    // 3.
    // Copy P from the device memory // Free device matrices
}
```
Outline

1. Evolvements of NVIDIA GPU
2. CUDA Basic
3. Detailed Steps
   - Device Memories and Data Transfer
   - Kernel Functions and Threading
Memory Hierarchy on Device

- Memory hierarchy on device
  - Global Memory
    - Main means of communicating between host and device
    - Long latency access
  - Shared Memory
    - Short latency
  - Register
    - Per-thread local variables

Data access

- Device code can read/write
  - Per-thread registers, per-block shared memory, per-grid global memory
- Host code can transfer data to/from
  - Per-grid global memory
CUDA Device Memory Allocation

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

```c
float* Md; //d indicates a device data
int size = width*width*sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

- `cudaFree()`
  - Frees object from device Global Memory
    - Pointer to freed object
**CUDA Host-Device Data Transfer**

- `cudaMemcpy()`
  - Memory data transfer
  - Requires four parameters
    1. Pointer to destination
    2. Pointer to source
    3. Number of bytes copied
    4. Type of transfer
  - Types of transfer
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToDevice`  
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```
Matrix Multiplication: After Integrating the Data Transfer

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

    // 1. Allocate and Load M, N to device memory
    cudaMalloc((void**)&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Pd, size);
    cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

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

    // 3. Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```
A kernel is implemented as a grid of threads

- Threads in grid is further decomposed into blocks
- A grid can be up to three-dimensional

A thread block is a batch of threads that can cooperate with each other by:

- Synchronizing their execution
- Efficiently sharing data through shared memory

A block can be one or two or three-dimensional

Each block and each thread has its own ID
Index (i.e., coordinates) of Block and Thread

- Each block and each thread are assigned an index, i.e., `blockIdx` and `threadIdx`
  - `blockIdx.x, blockIdx.y, blockIdx.z`
  - `threadIdx.x, threadIdx.y, threadIdx.z`
Index (i.e., coordinates) of Block and Thread

Each block and each thread are assigned an index, i.e., `blockIdx` and `threadIdx`

- `blockIdx.x, blockIdx.y, blockIdx.z`
- `threadIdx.x, threadIdx.y, threadIdx.z`

  - `threadIdx.y` → row, `threadIdx.x` → column
Define the Dimension of Grid and Block

- Use pre-defined `dim3` to define the dimension of grid and block
  - Use `gridDim` and `blockDim` to get the dimensions of the grid and the block

```c
dim3 dimGrid(width_g, height_g, depth_g);
dim3 dimBlock(width_b, height_b, depth_b);
```

- Total number of threads issued
  - `width_g \times height_g \times depth_g \times width_b \times height_b \times depth_b`

- Launch the device computation threads

```c
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```
// Matrix multiplication kernel
// -- per thread code

__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd,
    int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the
    // element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < width; ++k) {
        float Melement = Md[ty*width+k];
        float Nelement = Nd[k*width+tx];
        Pvalue += Melement * Nelement;
    }
    Pd[ty*width+tx] = Pvalue;
}
## CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on</th>
<th>Only callable from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>globe</strong> void DeviceFunc()</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
  - **Must be** void function
- **__device__** and **__host__** can be used together
  - Generate two versions of the same function during the compilation
- For functions executed on the device
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
  - No indirect function calls through pointers
- All functions are host functions by default