Hardware/Software Co-Design

CUDA Threads

Miaoqing Huang
University of Arkansas
Fall 2011
Outline

1. Simple Solution to the Matrix Multiplication Problem

2. CUDA Threads
   - Using blockIdx and threadIdx
   - More on Threads
Outline

1. Simple Solution to the Matrix Multiplication Problem

2. CUDA Threads
   - Using blockIdx and threadIdx
   - More on Threads
CUDA Thread Organization

- 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
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

```
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

```
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);
```
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`
  - `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
  - `threadIdx.y` → row, `threadIdx.x` → column
Simple Solution to the Matrix Multiplication Problem
CUDA Threads

Kernel Function Specification and the Launch

```c
// 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;
}
```

```c
dim3 dimGrid(1,1);
dim3 dimBlock(width,width);
MatrixMulKernel
    <<<<dimGrid,dimBlock>>>
    (Md,Nd,Pd,width);
```
Limitations to the Simple Solution

- The threads in the same block are scheduled into the same streaming multiprocessor
  - The processing power of the other SMs in the same device are wasted!
Limitations to the Simple Solution

- The threads in the same block are scheduled into the same streaming multiprocessor
  - The processing power of the other SMs in the same device are wasted!

```
threadID = threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...
```

Thread Block 0

```
threadID = threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...
```

Thread Block 1

```
threadID = threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...
```

Thread Block N - 1

```
threadID = threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...
```
Limitations to the Simple Solution

- The threads in the same block are scheduled into the same streaming multiprocessor
  - The processing power of the other SMs in the same device are wasted!

- The total number of threads residing in the same block is limited
  - G80 and GT200: up to 512
  - Fermi: up to 1024
Outline

1. Simple Solution to the Matrix Multiplication Problem

2. CUDA Threads
   - Using blockIdx and threadIdx
   - More on Threads
Outline

1. Simple Solution to the Matrix Multiplication Problem

2. CUDA Threads
   - Using blockIdx and threadIdx
   - More on Threads
Identifying the Location of a Thread in the Grid

- Define the dimension of grid and block using `dim3 struct`
  - Maximum x-, y- or z- dimension of a grid: 65,535
  - Maximum number of threads in a block: 512 (G80/GT200) or 1024 (Fermi)

- Locate the position of a thread by using
  - `blockIdx` and `threadIdx`
  - `blockDim` and `gridDim`

- `threadIdx.x: 0 ... blockDim.x-1`
- `blockIdx.x: 0 ... gridDim.x-1`

```
threadID = blockIdx.x * blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
... Thread Block 0
```
```
Thread Block 1
```
```
Thread Block N - 1
```

```
Thread Block 0
```
```
Thread Block 1
```
```
Thread Block N - 1
```
Matrix Multiplication Using Multiple Blocks

- Matrix multiplication can be implemented efficiently using CUDA threads.
- Using `blockIdx` and `threadIdx`

The diagram illustrates the concept of dividing large matrices into smaller tiles for parallel processing.

- **M** and **P** matrices are divided into tiles of size `TILE_WIDTH`.
- **N** matrix is also divided similarly.

Additionally, the document covers more on threads and other aspects of matrix multiplication in CUDA.
Change the Order of Coordinates of Matrix Element

- \( t_y \rightarrow \) row
- \( t_x \rightarrow \) column
Simple Solution to the Matrix Multiplication Problem
CUDA Threads

Using blockIdx and threadIdx
More on Threads

Change the Order of Coordinates of Matrix Element

=\begin{array}{cccc}
(0,0,1) & (1,0,1) & (2,0,1) & (3,0,1) \\
\text{Thread} & \text{Thread} & \text{Thread} & \text{Thread} \\
(0,0,0) & (1,0,0) & (2,0,0) & (3,0,0) \\
\text{Thread} & \text{Thread} & \text{Thread} & \text{Thread} \\
(0,1,0) & (1,1,0) & (2,1,0) & (3,1,0) \\
\end{array}

\begin{tabular}{|c|c|c|c|}
\hline
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
\hline
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
\hline
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
\hline
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3} \\
\hline
\end{tabular}

t_y \rightarrow \text{row}

t_x \rightarrow \text{column}
Simple Solution to the Matrix Multiplication Problem

CUDA Threads

Matrix Multiplication Using Multiple Blocks

Using blockIdx and threadIdx

More on Threads

Block(0,0)  Block(1,0)

Block(0,1)  Block(1,1)

TILE_WIDTH = 2

Pd_{0,0}  Pd_{1,0}  Pd_{2,0}  Pd_{3,0}

Pd_{0,1}  Pd_{1,1}  Pd_{2,1}  Pd_{3,1}

Pd_{0,2}  Pd_{1,2}  Pd_{2,2}  Pd_{3,2}

Pd_{0,3}  Pd_{1,3}  Pd_{2,3}  Pd_{3,3}
Matrix Multiplication Using Multiple Blocks

CUDA Threads

Using blockIdx and threadIdx

More on Threads

Simple Solution to the Matrix Multiplication Problem

TILE_WIDTH = 2

Block(0,0) Block(1,0)

Block(0,1) Block(1,1)

Pd0,0 Pd1,0 Pd2,0 Pd3,0

Pd0,1 Pd1,1 Pd2,1 Pd3,1

Pd0,2 Pd1,2 Pd2,2 Pd3,2

Pd0,3 Pd1,3 Pd2,3 Pd3,3

Md0,0 Md1,0 Md2,0 Md3,0

Md0,1 Md1,1 Md2,1 Md3,1

Nd0,0 Nd1,0 Nd1,1 Nd1,2 Nd1,3

Nd0,1 Nd1,1 Nd1,2 Nd1,3

Nd0,2 Nd1,2 Nd1,3

Nd0,3 Nd1,3

Pd0,0 Pd1,0 Pd2,0 Pd3,0

Pd0,1 Pd1,1 Pd2,1 Pd3,1

Pd0,2 Pd1,2 Pd2,2 Pd3,2

Pd0,3 Pd1,3 Pd2,3 Pd3,3

Pd0,0 Pd1,0 Pd2,0 Pd3,0

Pd0,1 Pd1,1 Pd2,1 Pd3,1

Pd0,2 Pd1,2 Pd2,2 Pd3,2

Pd0,3 Pd1,3 Pd2,3 Pd3,3
Revised Matrix Multiplication Kernel Using Multiple Blocks

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < width; ++k) {
        Pvalue += Md[Row*width+k] * Nd[k*width+Col];
    }

    Pd[Row*width+Col] = Pvalue;
}
```
Revised Matrix Multiplication Kernel Using Multiple Blocks

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width) {
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < width; ++k) {
        Pvalue += Md[Row*width+k] * Nd[k*width+Col];
    }
    Pd[Row*width+Col] = Pvalue;
}
```

```c
// Setup the execution configuration
dim3 dimGrid(width/TILE_WIDTH, width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);
```
Calculation Capability of the Tile-based Matrix Multiplication

The biggest size of the matrix the kernel can handle on Fermi GPUs.
Calculation Capability of the Tile-based Matrix Multiplication

The biggest size of the matrix the kernel can handle on Fermi GPUs

- \((65, 535 \times 32) \times (65, 535 \times 32)\)
Calculation Capability of the Tile-based Matrix Multiplication

1. The biggest size of the matrix the kernel can handle on Fermi GPUs
   - \((65, 535 \times 32) \times (65, 535 \times 32)\)

2. How to handle the matrix whose size is bigger than the above size?
Calculation Capability of the Tile-based Matrix Multiplication

1. The biggest size of the matrix the kernel can handle on Fermi GPUs
   - \((65, 535 \times 32) \times (65, 535 \times 32)\)

2. How to handle the matrix whose size is bigger than the above size?
   - Divide the \(P_d\) matrix into submatrices of a size permitted by the kernel
   - Run the same kernel multiple times manually
The biggest size of the matrix the kernel can handle on Fermi GPUs

- $(65,535 \times 32) \times (65,535 \times 32)$

How to handle the matrix whose size is bigger than the above size?

- Divide the $P_d$ matrix into submatrices of a size permitted by the kernel
- Run the same kernel multiple times manually

```
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width, submatrix_ID);
```
Outline

1. Simple Solution to the Matrix Multiplication Problem

2. CUDA Threads
   - Using blockIdx and threadIdx
   - More on Threads
Thread Synchronization

- All threads in a block execute the same kernel program (SPMD)
- Threads in the same block can synchronize while doing their share of the work
  - `__syncthreads()` statement in branches
  - Either all threads or none execute the same statement

```c
__global__ void Kernel(...) {
    ......
    if (condition) {
        // synchronization 1
        __syncthreads();
    }
    ......
    if (condition) {
        // synchronization 2
        __syncthreads();
    } else {
        // synchronization 3
        __syncthreads();
    }
    ......
}
```
Thread Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
- Each block can execute in any order relative to other blocks
- A kernel scales across any number of parallel processors
Thread Assignment

- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to 8 blocks in each SM as resource allows
    - Each block can contain up to 1024 threads
  - Up to 1536 threads in each SM on Fermi GPUs
- Threads run concurrently
  - SM maintains thread/block id numbers
  - SM manages/schedules thread execution
Thread Assignment

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected
# Technical Specification of GPU Architectures

<table>
<thead>
<tr>
<th>Technical Specification</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max x-, y- or z-dimension of a grid</td>
<td>65,535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Max x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Max z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of resident blocks per SM</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of resident warps per SM</td>
<td>24</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Max number of resident threads per SM</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>Number of 32-bit register per SM</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
</tr>
<tr>
<td>Max amount of shared memory per SM</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
</tr>
</tbody>
</table>