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

Lecture 4: CUDA Threads

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Software -> Hardware

• From a programmer’s perspective:
  – Blocks
  – Kernel
  – Threads
  – Grid

• Hardware Implementation:
  – SMs
  – SPs (per SM)
  – Warps
Some Restrictions First

• All threads in a grid execute the same kernel function
• A grid is organized as a 2D (or 3D if compute capability beyond 2.0) array of blocks (gridDim.x, gridDim.y, and gridDim.z)
• Each block is organized as 3D array of threads (blockDim.x, blockDim.y, and blockDim.z)
• Once a kernel is launched, its dimensions cannot change.
• All blocks in a grid have the same dimension
• The total size of a block has an upper bound
• Once assigned to an SM, the block must execute in its entirety by the SM
Compute Capability

- A standard way to expose **hardware resources** to applications.
- CUDA compute capability starts with 1.0 and latest one is 5.5 (as of today)
- API: `cudaGetDeviceProperties()`
Figure 3.2. An Example of CUDA Thread Organization.

Host

Kernel 1

Device

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Block (1, 1)

Grid 2

Kernel 2

Block (1, 1)

(0,0,0) (1,0,0) (2,0,0) (3,0,0)

(0,0,1) (1,0,1) (2,0,1) (3,0,1)

(0,0,1) (1,0,1) (2,0,1) (3,0,1)

Thread (0,0,0) Thread (1,0,0) Thread (2,0,0) Thread (3,0,0)

Thread (0,1,0) Thread (1,1,0) Thread (2,1,0) Thread (3,1,0)

Courtesy: NDVIA
• Thread ID is unique within a block
• Using block ID and thread ID we can make unique ID for each thread per kernel
Revisiting 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;
}

This is what we did before...
What is the main shortcoming??
Revisiting Matrix Multiplication

Can only handle limited elements in each dimension!

Reason: We used 1 block, and a block is limited to X threads (X depends on GPU type)
Revisiting
Matrix Multiplication

• Break-up Pd into tiles
• Each block calculates one tile
  – Each thread calculates one element
  – Block size equals tile size
Revisiting Matrix Multiplication

TILE_WIDTH = 2

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

<table>
<thead>
<tr>
<th>P_{0,0}</th>
<th>P_{1,0}</th>
<th>P_{2,0}</th>
<th>P_{3,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{0,1}</td>
<td>P_{1,1}</td>
<td>P_{2,1}</td>
<td>P_{3,1}</td>
</tr>
<tr>
<td>P_{0,2}</td>
<td>P_{1,2}</td>
<td>P_{2,2}</td>
<td>P_{3,2}</td>
</tr>
<tr>
<td>P_{0,3}</td>
<td>P_{1,3}</td>
<td>P_{2,3}</td>
<td>P_{3,3}</td>
</tr>
</tbody>
</table>

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

by 1

TILE_WIDTH = 2

Nd

Pd

Pd_{sub}
Revisiting Matrix Multiplication

```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);

__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 idenx 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;
}
```
Synchronization

_called by a kernel function
• The thread that makes the call will be held at the calling location until every thread in the block reaches the location
• Beware of if-then-else
• Threads in different blocks cannot synchronize -> CUDA runtime system can execute blocks in any order
The ability to execute the same application code on hardware with different number of execution resources is called transparent scalability.
Thread Assignment

• Threads assigned to execution resources on a block-by-block basis.
• CUDA runtime automatically reduces number of blocks assigned to each SM until resource usage is under limit.
• Runtime system:
  – maintains a list of blocks that need to execute
  – assigns new blocks to SM as they compute previously assigned blocks
• Example of SM resources
  – computational units
  – number of threads that can be simultaneously tracked and scheduled.
GT200 can accommodate 8 blocks/SM and up to 1024 threads can be assigned to an SM. What are our choices for number of blocks and number of threads/block?

Thread scheduling is an implementation concept.
FERMI
Warps

• Once a block is assigned to an SM, it is divided into units called warps.
  – Thread IDs within a warp are consecutive and increasing
  – Warp 0 starts with Thread ID 0

• Warp size is implementation specific.

• Warp is unit of thread scheduling in SMs
Warps

• Partitioning is always the same
• DO NOT rely on any ordering between warps
• Each warp is executed in a SIMD fashion (i.e. all threads within a warp must execute the same instruction at any given time).
  – Problem: branch divergence
Branch Divergence in Warps

- occurs when threads inside warps branches to different execution paths.

50% performance loss
Example of underutilization

Computational Resource Utilization

32 warps, 32 threads per warp, round-robin scheduling
Dealing With Branch Divergence

• A common case: avoid divergence when branch condition is a function of thread ID
  – Example with divergence:
    • If (threadIdx.x > 2) { }
    • This creates two different control paths for threads in a block
  – Example without divergence:
    • If (threadIdx.x / WARP_SIZE > 2) { }
    • Also creates two different control paths for threads in a block
    • Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

• There is a big body of research for dealing with branch divergence
Dealing With Branch Divergence

Predication

\(<p1>\) LDR r1, r2, 0

- If p1 is TRUE, instruction executes normally
- If p1 is FALSE, instruction treated as NOP

Example of Predication
Latency Tolerance

• When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution -> latency hiding
• Priority mechanism used to schedule ready warps
• Scheduling does not introduce idle time -> zero-overhead thread scheduling
• Scheduling is used for tolerating long-latency operations, such as:
  – pipelined floating-point arithmetic
  – branch instructions
This ability of tolerating long-latency operation is the main reason why GPUs do not dedicate as much chip area to cache memory and branch prediction mechanisms as traditional CPUs.
Exercise: Suppose 4 clock cycles are needed to dispatch the same instruction for all threads in a Warp in G80. If there is one global memory access every 4 instructions, how many warps are needed to fully tolerate 200-cycle memory latency?
Exercise

The GT200 has the following specs (maximum numbers):
• 512 threads/block
• 1024 threads/SM
• 8 blocks/SM
• 32 threads/warp

What is the best configuration for thread blocks to implement matrix multiplications 8x8, 16x16, or 32x32?
Myths About CUDA

• **GPUs have very wide (1000s) SIMD machines**
  – No, a CUDA Warp is only 32 threads

• **Branching is not possible on GPUs**
  – Incorrect.

• **GPUs are power-inefficient**
  – Nope, performance per watt is quite good

• **CUDA is only for C or C++ programmers**
  – Not true, there are third party wrappers for Java, Python, and more
## G80, GT200, and Fermi

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>GF100</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 Floating Point</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Single Precision Floating Point</td>
<td>128 MAD ops / clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops / clock</td>
</tr>
<tr>
<td>Special Function Units / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
Conclusion

• We must be aware of the restrictions imposed by hardware:
  – threads/SM
  – blocks/SM
  – threads/blocks
  – threads/warps

• The only safe way to synchronize threads in different blocks is to terminate the kernel and start a new kernel for the activities after the synchronization point