Lecture 19: CUDA II

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 ($grid\text{Dim}.x$, $grid\text{Dim}.y$, and $grid\text{Dim}.z$).
- Each block is organized as a 3D array of threads ($block\text{Dim}.x$, $block\text{Dim}.y$, and $block\text{Dim}.z$).
- 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 7.x (as of today)
- API: `cudaGetDeviceProperties()`
cudaError_t cudaGetDeviceProperties(
    struct cudaDeviceProp * prop, int device
)

struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem; /* in bytes */
    size_t sharedMemPerBlock; /* in bytes */
    int regsPerBlock;
    int warpSize;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate; /* in KHz */
    size_t totalConstMem;
    int major;  int minor;
    int multiProcessorCount;
    int concurrentKernels;
    int unifiedAddressing;
    int memoryClockRate;
    int memoryBusWidth;
    int l2CacheSize;
    int maxThreadsPerMultiProcessor;
    ... and a lot of other stuff
}

cudaError_t cudaGetDeviceCount(
    int * count
)
## Compute Capability Example

### Table 1. A Comparison of Maxwell GM107 to Kepler GK107

<table>
<thead>
<tr>
<th>GPU</th>
<th>GK107 (Kepler)</th>
<th>GM107 (Maxwell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>384</td>
<td>640</td>
</tr>
<tr>
<td>Base Clock</td>
<td>1058 MHz</td>
<td>1020 MHz</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>N/A</td>
<td>1085 MHz</td>
</tr>
<tr>
<td>GFLOP/s</td>
<td>812.5</td>
<td>1305.6</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Shared Memory / SM</td>
<td>16KB / 48 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Active Blocks / SM</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>5000 MHz</td>
<td>5400 MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>80 GB/s</td>
<td>86.4 GB/s</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>256 KB</td>
<td>2048 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>64W</td>
<td>60W</td>
</tr>
<tr>
<td>Transistors</td>
<td>1.3 Billion</td>
<td>1.87 Billion</td>
</tr>
<tr>
<td>Die Size</td>
<td>118 mm²</td>
<td>148 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28 nm</td>
<td>28 nm</td>
</tr>
</tbody>
</table>
Figure 3.2. An Example of CUDA Thread Organization.
• 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

```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;

    // 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

```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;

    // 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;
}
```

Can only handle 16 elements in each dimension!

Reason:
We used 1 block, and a block is limited to 512 threads (1024 in newer GPUs)
Revisiting Matrix Multiplication

- Break-up $P_d$ 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)  Block(0,1)  Block(1,1)

P_{0,0} P_{1,0} P_{2,0} P_{3,0}
P_{0,1} P_{1,1} P_{2,1} P_{3,1}
P_{0,2} P_{1,2} P_{2,2} P_{3,2}
P_{0,3} P_{1,3} P_{2,3} P_{3,3}

TILE_WIDTH = 2

Nd  Pd

Pd_{sub}
Revisiting Matrix Multiplication

```
// 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 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;
}```
Synchronization

```c
__syncthreads()
```

- 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
Each block can execute in any order relative to other blocks.

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.
  – Registers
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.
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.
  - But so far all NVIDIA GPUs have warp = 32 threads.
- 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
Computational Resource Utilization

Example of underutilization

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
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
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