VSCSE Summer School 2008
Accelerators for Science and Engineering
Applications – GPUs and Multi-cores

Lecture 3: CUDA Threads
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
Matrix Multiplication Using Multiple Blocks

• Break-up $P_d$ into tiles
• Each block calculates one tile
  – Each thread calculates one element
  – Block size equal tile size
A Small Example

TILE_WIDTH = 2

Block(0,0) \rightarrow Block(1,0)

\begin{array}{cccc}
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} \\
\end{array}

Block(0,1) \leftrightarrow Block(1,1)
A Small Example: Multiplication

<table>
<thead>
<tr>
<th>Md_{0,0}</th>
<th>Md_{1,0}</th>
<th>Md_{2,0}</th>
<th>Md_{3,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Md_{0,1}</td>
<td>Md_{1,1}</td>
<td>Md_{2,1}</td>
<td>Md_{3,1}</td>
</tr>
<tr>
<td>Pd_{0,0}</td>
<td>Pd_{1,0}</td>
<td>Pd_{2,0}</td>
<td>Pd_{3,0}</td>
</tr>
<tr>
<td>Pd_{0,1}</td>
<td>Pd_{1,1}</td>
<td>Pd_{2,1}</td>
<td>Pd_{3,1}</td>
</tr>
<tr>
<td>Pd_{0,2}</td>
<td>Pd_{1,2}</td>
<td>Pd_{2,2}</td>
<td>Pd_{3,2}</td>
</tr>
<tr>
<td>Pd_{0,3}</td>
<td>Pd_{1,3}</td>
<td>Pd_{2,3}</td>
<td>Pd_{3,3}</td>
</tr>
</tbody>
</table>
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;
}
```
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

Courtesy: John Nickolls, NVIDIA
Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
G80 Example: Executing Thread Blocks

• Threads are assigned to Streaming Multiprocessors in block granularity
  – Up to 8 blocks to each SM as resource allows
  – SM in G80 can take up to 768 threads
    • Could be 256 (threads/block) * 3 blocks
    • Or 128 (threads/block) * 6 blocks, etc.

• Threads run concurrently
  – SM maintains thread/block id #s
  – SM manages/schedules thread execution
G80 Example: Thread Scheduling

• 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?
  – Each Block is divided into 256/32 = 8 Warps
  – There are 8 * 3 = 24 Warps
G80 Example: Thread Scheduling
(Cont.)

• SM implements zero-overhead warp scheduling
  – At any time, only one of the warps is executed by SM
  – 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

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 18-22, 2008
G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?

  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
Some Additional API Features
Application Programming Interface

• The API is an extension to the C programming language

• It consists of:
  – Language extensions
    • To target portions of the code for execution on the device
  – A runtime library split into:
    • A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    • A host component to control and access one or more devices from the host
    • A device component providing device-specific functions
Language Extensions: Built-in Variables

- `dim3 blockDim;`
  - Dimensions of the block in threads

- `dim3 blockIdx;`
  - Block index within the grid

- `dim3 threadIdx;`
  - Thread index within the block

- `dim3 blockDim;`
  - Dimensions of the grid in blocks (gridDim.z unused)
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`  
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.

- When executed on the host, a given function uses the C runtime implementation if available
- These functions are only supported for scalar types, not vector types
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`
Host Runtime Component

• Provides functions to deal with:
  – **Device** management (including multi-device systems)
  – **Memory** management
  – **Error** handling

• Initializes the first time a runtime function is called

• A host thread can invoke device code on only one device
  – Multiple host threads required to run on multiple devices
Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block