CUDA Threads

Mark Greenstreet & Ian M. Mitchell

CPSC 418 – March 8 & March 10, 2017

- Kernel organization: grids, blocks & threads.
- Hardware organization: SMs, SPs & warps.

Compute Capability

- Lots of nVidia jargon here.
- Lots of very specific constraints on hardware capabilities.
- Values of those constraints depend on the *compute capability*: essentially a version number for the GPU hardware.
  - CS department lab ({`lin01`, `lin02`, ..., `lin25`}`.ugrad.cs.ubc.ca`) has GeForce GTX 550 Ti which feature compute capability 2.1.
  - Examples of recent GPUs:
    - ★ Compute capability 3.5: GT 730 & GTX 780.
    - ★ Compute capability 5.0: GTX 750, 8xxM & 960M.
    - ★ Compute capability 5.2: GTX 9xx, 965M.
    - ★ Compute capability 6.1: GTX 10xx.
  - More details at the CUDA wikipedia page.

Thread organization: Grids, Blocks and Threads

- When a kernel is launched, it creates a collection of threads.
- This collection is called a **grid**.
  - A grid is organized as an array of **blocks**
  - Each block is an array of **threads**
  - Array sizes are fixed once a kernel is launched.
- Why so many details?
  - Switching between blocks is done (I infer) by software in the GPU.
  - Switching between threads in a block is done by hardware.
  - By distinguishing blocks from threads, the CUDA model exposes the performance issues to the programmer.

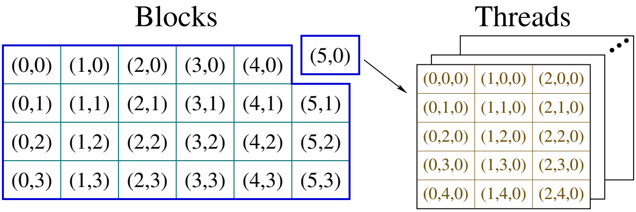
A grid is an array of blocks

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)

A grid

- Blocks are scheduled by the GPU **software**.
- Blocks can be arranged as 1D, 2D or 3D array.
  - Dimensions are called "`x`", "`y`" and "`z`".
- There can be **lots** of blocks:
  - Each dimension can be up to $2^{16} - 1 = 65535$.
  - CC 3.0+ allows `x` dimension up to $2^{31} - 1$ blocks.

Each block is an array of threads



Where do they put all those threads?

- Threads are scheduled by the GPU **hardware**.
- Threads can be arranged as a 1D, 2D, or 3D array
  - Grid and block dimensions and sizes may be different.
- There can be a moderate number of threads in each dimension:
  - `x` or `y` up to 1024 threads.
  - `z` up to 64 threads.
- However, total number of threads per block (product of all dimensions) is also capped at 1024.

Threads and blocks: launching a kernel

- Let's say we have:

  `__global__ void kernel_fun(`*args*`)`

- To launch this kernel, we execute a statement like:

  `kernel_fun<<<`*dimGrid*`, `*dimBlock*`>>>(`*actuals*`);`

  where
  - *dimGrid* specifies the dimension(s) of the grid (an array of blocks):
    - ★ *dimGrid* can be an `int`, in which case the array is 1D.
    - ★ *dimGrid* can be a `dim3`, for example:

      `dim3(6,4,1)`
  - *dimBlock* specifies the dimension(s) of each block (an array of threads):
    - ★ *dimBlock* can be an `int` or a `dim3`.

Threads and Blocks within a Kernel's Grid

- Within a running kernel, CUDA-C provides four built-in variables to determine the position of a thread within the grid: `blockDim`, `blockIdx`, `threadDim`, and `threadIdx`.
- There is a naming pattern:
  - Each of these structures has three fields: `x`, `y` and `z` corresponding to the three possible dimensions.
  - `blockDim.?` gives the size of the grid in each dimension `x`, `y` or `z`.
  - `threadDim.?` gives the size of each block in each dimension.
  - `blockIdx.?` gives the indices of the thread's block within the grid.
  - `threadIdx.?` gives the indices of the thread within its block.
- For dimensions which are absent:
  - `blockDim` or `threadDim` will be 1.
  - `blockIdx` or `threadIdx` will be 0.

Threads and Blocks: Where are We?

- Note the constraints:

$$
\begin{aligned}
0 &\leq \texttt{blockIdx.x} < \texttt{blockDim.x}\\
0 &\leq \texttt{blockIdx.y} < \texttt{blockDim.y}\\
0 &\leq \texttt{blockIdx.z} < \texttt{blockDim.z}\\
0 &\leq \texttt{threadIdx.x} < \texttt{threadDim.x}\\
0 &\leq \texttt{threadIdx.y} < \texttt{threadDim.y}\\
0 &\leq \texttt{threadIdx.z} < \texttt{threadDim.z}
\end{aligned}
$$

- Because the size of blocks are limited, it is common to use code such as:

```
uint my_idx = blockDim.x*blockIdx.x + threadIdx.x;
```

to combine the block and thread indices into a single index.

Bounds checking: launching kernels

- Consider executing `kernel_fun` on an array of `n` elements.
- Because `n` might be large, we'll use `n/256` blocks of 256 threads.
  - THINK: what if `n` is not a multiple of 256?
  - We'll round up to make sure we have enough threads.
- The kernel launch looks like:

```
kernel_fun<<<ceil(n/256.0), 256>>>(n, myArray);
```

  - Why divide by `256.0` instead of `256`?
  - Why use `ceil`?

Bounds checking: in the kernel

- The kernel launch looks like:

```
kernel_fun<<<ceil(n/256.0), 256>>>(n, myArray);
```

- THINK: what if `n` is not a multiple of 256?
  - We'll launch more than `n` threads?
  - For example, if `n==1000`, then we'll launch 4 blocks of 256 threads. A total of 1024 threads.
  - What will the last 24 threads do?
- Add a test:

```
uint my_idx = blockDim.x*blockIdx.x + threadIdx.x;
if(my_idx < n) {
    ...
}
```

SMs, SPs and Warps (oh my!)

- Each *streaming multiprocessor* (SM) has multiple *streaming processors* (SPs) and can be responsible for multiple groups of 32 threads called *warps*.
    - From the *New Oxford American Dictionary*: (the) “warp” is “the threads on a loom over and under which other threads (the weft) are passed to make cloth”
- Details, details…
    - These concepts are not part of the CUDA platform and API: Code is written in terms of a grid of blocks of threads.
    - You can write correct code without thinking about these details.
    - If you want to write fast code, you must take them into account.
    - The block vs grid structure exposes these details if you want to take advantage of them.

SMs, SPs and Warps: What are They?

- Each streaming multiprocessor (SM) in the GPU executes threads in SIMD fashion.
  - All threads in a block are assigned to the same SM.
  - Each SM has a single (or small number of?) instruction fetch unit(s) and a larger number of execution units.
- Each SM has multiple streaming processors (SPs) that actually execute an instruction.
  - The SPs are specialized: ALUs, load / store, special function units.
  - A single SP can perform a single operation on a small set of threads.
- A warp is a collection of 32 threads that execute together on the same SP.

SMs, SPs and Warps: Why do We Care?

- Fill your warps: Ensure the number of threads in a block is a multiple of the warp size to avoid idle hardware.
- Have lots of warps: If one warp is waiting on a long latency operation, the SM can find another warp to execute.
  - Provides *latency tolerance* or *latency hiding*.
- Watch out for hardware limits (per SM).
  - Maximum number of resident blocks (8 in 2.x, 32 in 6.x).
  - Maximum number of resident warps (48 in 2.x, 64 thereafter).
  - Maximum number of resident threads (1536 in 2.x, 2048 thereafter).
  - Exceeding these limits will not crash the system, but will result in slower execution.
- Watch out for thread divergence.
  - If different threads in the same warp are following different code paths, all possible paths will be executed sequentially and those threads not on the current path will be idle.
  - Execution is still correct, but much slower.

A Warped Example: Reduce (part 1)

- Consider a reduce of an array `data` containing `n` elements using `n/2` threads (assume `n` is power of 2).
- Simple code:

```
for(int stride = 1; stride < n; stride += stride) {
   if((my_idx & (stride-1)) == 0)
      data[2*my_idx] += data[2*my_idx + stride];
   __syncthreads();
}
```

- The `__syncthreads()` call ensures that every thread has completed an iteration of the loop before any thread starts the next iteration.
  - More discussion on slide 18.

A Warped Example: Reduce (part 2)

- Consider `n == 16`
  - First iteration, for `i` in 0, . . . , 7:
    `data[2*i] += data[2*i]+1`
    Now, all the even indexed elements have their sum with their odd counterpart.
  - Second iteration, for `i` in 0, 2, 4, 6:
    `data[2*i] += data[2*i]+2`.
    All elements with indices that are multiples of four, have their sum with the next three elements.
  - Third iteration leads with `data[0]` and `data[8]` holding sums for their halves of the array.
  - The fourth iteration puts the complete sum into `data[0]`.
- There are at most 8 threads working, so everything fits within a single warp.

A Warped Example: Reduce (part 3)

- What if `n==1024`?
  - We have 512 threads: 16 warps of 32 threads.
  - In the first iteration all threads are active.
  - In the next iteration each warp has 16 active threads, so the GPU has to execute the code for all 16 warps even though half the threads do nothing.
  - In subsequent iterations, the warps are more and more poorly utilized.
- This solution is correct, but much of the parallel hardware will sit idle much of the time.
- We would like to pack the busy threads into the minimum number of warps.

Warp Speed!

```
for(int stride = n/2; stride > 0; stride >>= 1) {
   if(my_idx < stride)
      data[my_idx] += data[my_idx] + stride;
   __syncthreads();
}
```

- Consider `n == 1024` again.
  - In the first iteration, there are 16 active warps – all threads in each warp are busy.
  - In the second iteration, there are 8 active warps – all threads in each active warp are busy.
  - Similarly, for the 3rd through 5th iterations
- The number of active warps decreases after each iteration, but all threads in each active warp are busy.
  - The inactive warps have no pending instructions, so they will not be scheduled and will not occupy processing resources.

Synchronization

- The reduce example used `__syncthreads()`: all the threads in the block must execute this statement before any can continue beyond it.
  - Be **very** careful about thread divergence: All threads in the block must meet at the **same** barrier.
  - That means the **same** line of code.
  - In loops, that means the **same** iteration.
  - Executing different `__syncthreads()` commands will cause the kernel to hang.
- Also, `__syncthreads()` only synchronizes between threads within a single block.
  - Note that threads within a warp already stay synchronized because they are executed together.
  - The only way to synchronize between threads in different blocks is to finish the kernel and launch another.
- We'll cover synchronization in more detail later.

Preview

March 10: CUDA Threads, Part 2	
March 13: CUDA Memory	
Reading	Kirk & Hwu Ch. 4
March 15: CUDA Memory: examples	
March 17: CUDA Performance	
Reading	Kirk & Hwu Ch. 5
March 20: Matrix multiplication with CUDA, Part 1	
March 22: Matrix multiplication with CUDA, Part 2	
March 24 – April 3: Other Topics • more parallel algorithms, e.g. dynamic programming? • reasoning about concurrency, e.g. termination detection • other paradigms, e.g. Scala and futures?	
April 5: Party: 50th Anniversary of Amdahl's Law	

Review

- In CUDA, what is a grid, a block, and thread?
- Why does CUDA allow millions of thread blocks but only 1024 threads per block?
- How does a programmer specify the number of blocks and number of threads when launching a CUDA kernel?
- How does a thread determine its position within the grid?
- Why do threads need to check their indices against array bounds?
- What is a warp? Why does it matter?