CUDA Threads

Mark Greenstreet

CpSc 418 – Mar. 7, 2016

- Threads organization: grids, blocks, threads, and warps.
- Synchronization
- Examples

Thread organization: grids, blocks and threads

- Lots of nVidia jargon here.
  - When a kernel is launched, it creates an array of threads.
  - This array is called a **grid**.
- A grid is organized as an array of **blocks**
- Each block is an array of **threads**
- 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.

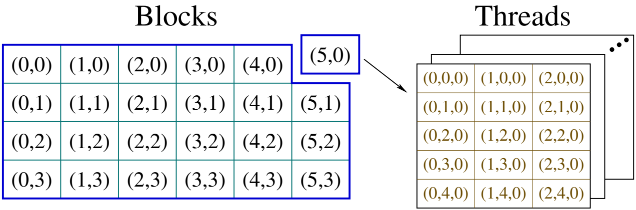
A grid is an array of blocks

(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 a 1D or 2D array.
- There can be **lots** of blocks:
  - Each dimension can be up to $2^{16} = 65536$.

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.
- There are a limited number of threads per block:
  - The total number of threads (product of all dimensions) is at most 256 to 1024, depending on the GPU.

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* is specifies the dimension(s) of the grid (an array of blocks):
    - ★ *dimGrid* can be an `int`, in which case the array is one dimensional of that size.
    - ★ or, *dimGrid* can be a `dim3`, for example:

      `dim3(6,4,1)`
    - ★ The last component of the `dim3` is the z-dimension, which is ignored when describing a grid. To avoid confusion, the standard practice it to use a value of `1`.
  - *dimBlock* is specifies the dimension(s) of each block (an array of threads):
    - ★ *dimGrid* can be an `int` or a `dim3`.
    - ★ If *dimGrid* is a `dim3`, all three dimensions are used.

Threads and blocks: within a kernel

- With a kernel, CUDA-C provides four variables to determine the position of a thread within the grid: `blockDim`, `blockIdx`, `threadDim`, and `threadIdx`.
- `blockDim.x` and `blockDim.y` give the size of the grid in the `x`- and `y`-dimensions.
- `threadDim.x`, `threadDim.y`, and `threadDim.z` give the size of each block.
- `blockIdx.x` and `blockIdx.y` give the indices of the thread's block within the grid. Note that:
  - $0 \leq$ `blockIdx.x` $<$ `BlockDim.x`, and
  - $0 \leq$ `blockIdx.y` $<$ `BlockDim.y`.
- Likewise, `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` give the indices of the thread within its block.
- 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) {
    ...
}
```

Warps

- Warps refer to how the hardware executes threads.
  - The programmer writes code with grid consisting of blocks of threads.
  - You can write **correct** code without paying attention to warps.
  - But you need to think about warps to write `fast` code.
- Each streaming multiprocessor (SM) in the GPU executes threads in SIMD fashion.
  - A warp is a collection of threads that execute together on the same SM.
- Why we care:
  - It helps performance to make the number of threads in a block a multiple of the warp size.
  - Thread divergence is an issue when different threads in the same warp follow different control paths.
- Etymology: "warp" is a term from weaving:

> *"the threads on a loom over and under which other threads (the weft) are passed to make cloth"*

From the *New Oxford American Dictionary* (on my laptop).

A Warped Example: Reduce (part 1 of 2)

- Consider a reduce of an array, `data`, of `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(); % see slide 13
}
```

- 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]`.
- What if `n==1024`? See the next slide.

A Warped Example: Reduce (part 2 of 2)

- 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 – 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.
- We would like to pack the busy threads into the minumum number of warps.

Faster Warps

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

- Consider `n == 1024`.
- 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.
  - All threads in each active warp are busy.

Synchronization

- The reduce example used `__syncthreads()`: all the threads in the block must execute this statement before any continue beyond it.
  - Be **very** careful about thread divergence.
  - All threads in the block must meet at the barrier.
  - They must all meet at the **same** barrier.
- We'll cover synchronization in more detail on March 9.

Some examples

See examples.cu.

Preview

March 9: Synchronization and Scheduling
Reading: Kirk & Hwu, Chapter 4.
March 11: The GPU Memory Model 1
Reading: Kirk & Hwu, Chapter 5.
March 14: The GPU Memory Model 2
Reading: Kirk & Hwu, Chapter 5.
March 16: GPU Performance 1
Reading: Kirk & Hwu, Chapter 6.
March 18: GPU Performance 2
Reading: Kirk & Hwu, Chapter 6.
March 21: Parallel Sorting
Reading: TBD.

But of course, we'll adjust this as we go.

Review

- In CUDA, what is a grid, a block, and thread?
- Why does CUDA allow millions of thread blocks but only 256 to 1024 threads per block?
- How does a programmer specify the number of thread blocks and number of threads when launching a CUDA kernel?
- How does a thread determing its position within the thread grid? "global memory" in CUDA programming.
- Why do threads need to check their indices against array bounds?
- What is a warp? Why does it matter?