CUDA Threads

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – February 26, 2018

Table of Contents

Outline

GPU Architecture Summary

Focus on the nVidia architecture, but others are very similar.

- Lots of cores:
  - Dozens of SIMD processors.
  - Each SIMD processor has 32 pipelines.
- Deep, simple, execution pipelines
  - Optimized for floating point.
  - No bypassing: use multi-threading for performance.
  - Branches handled by predicated execution

"When you come to a fork in the road, take it."
(Often attributed to Yogi Berra.)

- Limited on-chip memory of various types.
  - 1–2 MB total (compare to big CPUs with 32–64MB of L3 cache).
  - The programmer manages data placement.

Why this Architecture? First Answer: Performance

Today's processors are constrained by how much performance you can get using $\sim$ 200 watts.

- Lots of energy needed to move bits and/or perform operations quickly.
  - $E \sim d/t^{\alpha}$, where E is energy, d is distance, t is time per operation, and $1 < \alpha < 2$ depending on design details.
  - Corollary: $P \sim d/t^{\alpha+1}$. Power grows someplace between quadratically and cubically with clock frequency.
- GPUs optimize performance/power through use of:
  - SIMD: instruction fetch and decode moves lots of bits. Amortize over many cores.
  - Simple pipelines: bypassing means moving bits quickly. GPUs omit bypasses.
  - High latency: avoid pipeline stages that must do a lot in a hurry.
  - Expose the memory hierarchy: let the programmer control moving data bits around.

Why this Architecture? Second Answer: Economics

GPUs are designed for the consumer graphics market, and happen to be useful for parallel numerical computing.

- High-volume market amortizes high design cost over large number of units sold.
- Cheap sells.

  “I think there is a world market for about five computers”
  (Often (mis)attributed to IBM’s T. J. Watson.)

  - It was the unprecedented drop in price/performance created by the integrated circuit (Moore’s law) which made this prediction so completely wrong.
  - Add only features which will sell more chips.
- Market niches are important.
  - More memory would help GPUs for numerical computing, but little payoff for graphics and it starts to look like a CPU.

Data Parallelism

- Spotting it: When you see a `for`-loop:
  - Is the loop-index used as an array index?
  - Are the iterations independent?
- If the order of iterations does not matter, do them in parallel!

- Data-parallel problems:
  - Run well on GPUs because each element (or segment) of the array can be handled by a different thread.
  - Are good candidates for most parallel techniques because the available parallelism grows with the problem size.
- Compare with “task parallelism” where the problem is divided into the same number of tasks regardless of its size.

Outline

Thread organization: Grids, Blocks and Threads

- When a kernel is launched, it creates a collection of threads 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 to the programmer the difference in behavior and consequent difference in performance.

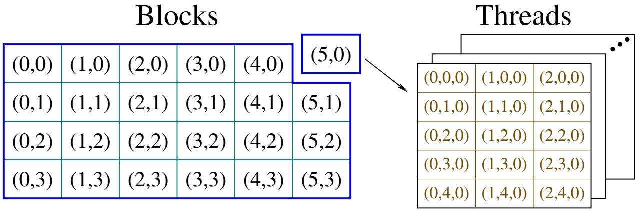
A grid is an array of blocks

(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

Assume that we have a kernel function:

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

Then to launch this kernel, we execute a statement:

`kernel_fun<<<`*dimGrid*`, `*dimBlock*`>>>(`*actual args*`);`

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 also be an `int` or a `dim3`.

Why are grids and blocks 1D, 2D or 3D?

Threads and Blocks: Which Thread Are We?

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

Threads and Blocks: Which Thread Are We?

- Note the constraints:

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

- Because the size of blocks is severely 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.

Example: Consider indexes of threads associated to pixels in K&H Figure 3.5.

Threads and Blocks: Synchronization

How do we ensure that all threads have completed certain tasks?

- Within a block use `__syncthreads()`: All the threads in the block must execute this statement before any can continue beyond it.
  - That means the **same** line of code.
  - In loops, that means hitting the barrier in the **same** iteration.
  - In conditionals be **very** careful about thread divergence: All threads in the block must meet at the **same** barrier.
  - Executing different `__syncthreads()` commands will cause the kernel to hang.
- Within a grid: Finish the kernel and launch another.
- We'll cover synchronization in more detail later.

Threads and Blocks: Bounds checking

Consider executing `kernel_fun` on an array of `n` elements.

- Because `n` might be large, we'll use `n/256` blocks of 256 threads.
  - If `n` is not a multiple of 256 we must round up the number of blocks 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`?
- When executing the kernel, need to handle extra threads.
  - For example, consider `n = 1000`, so there are 4 blocks of 256 threads for a total of 1024 threads.
  - Kernel must include a test to ensure the extra threads are idle:

    ```
    uint my_idx = blockDim.x*blockIdx.x + threadIdx.x;
    if(my_idx < n) {
      <reading and writing memory locations in arrays of size n>
    }
    ```

Outline

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 instruction fetch unit and a larger number of execution units.
- Each SM has multiple streaming processors (SPs) that actually execute instructions.
  - What marketing calls "NVidia CUDA Cores."
  - Most SPs handle basic ALU operations (integer and floating point).
  - Also separate specialized execution units, such as load/store or special functions.
  - A single SP can start a single operation on a single thread each cycle.
- On each cycle, each SM dispatches an instruction for the threads of an executable warp to its SPs.

Compute Capability: Version numbers for your GPU

- Architecture and hardware constraints affect performance.
  - Specific values of those constraints depend on chip / card.
  - Software user base will have a wide variety of chips / cards.
- Impractical to optimize high-level implementation or even “parallel thread execution” (PTX) intermediate assembly code for all possible chips / cards.
- Enter *compute capability*: essentially a version number for the GPU hardware.
  - Graphics lab machines `linXX.ugrad.cs.ubc.ca` (where XX takes values $01, 02, \ldots, 25$) have GeForce GTX 1060 3GB cards which feature compute capability 6.1.
  - Examples of NVidia 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.

Warning: We are using the NVidia jargon.

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.

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?
- Why are NVidia GPUs have both a model number (such as “GTX 1060”) and a compute capability (such as “CC 6.1”)?