CUDA threads v cores

Jump to bottom

zoran edited this page on Sep 9 · 1 revision

One of the most common misunderstandings with CUDA programming arises from assuming that CUDA cores and threads are synonymous. This post explains away this potentially confusing idea, with an overview of device architecture and relevant features of the CUDA APIs.

Streaming Multiprocessors

So first, let's outline what a typical CUDA device is like. Physically, an Nvidia GPU is arranged as a collection of Streaming Multiprocessors (SMs). This will be brought up during lectures in due course. Each individual SM holds a fixed number of Streaming Processors, just as the name SM would suggest. These are what we call the CUDA cores. The total number of CUDA cores available on a GPU is the number of cores per SM times the total number of SMs.

Consider the K80 GPU. It has 26 SMs. Each SM unit is comprised of 192 processor cores. Which gives the K80 card a total of $$192 \times 26 = 4992$$ streaming processor cores. These are called CUDA cores.

To emphasize the point here clearly - there are *hardware* restrictions that dictate in what manner and order threads are executed, and also how many threads can be executed at a time.

Threads and Blocks

Multiple thread-blocks can be scheduled on an SM for execution. There is a hardware-defined limitation on the number of threads each of those blocks can have - $$1024$$ threads/block -- no more. But, there's a catch! After reading this, you might conclude that this means there are 'physical blocks' somewhere in the SM, each having a maximum capacity for threads. **That is not the case.** Blocks are a way of organizing thread indexing for the benefit of the programmer. Therefore, **blocks are a concept, not a physical property of the GPU.**

Whether you want to 'arrange' those threads in a:

- 1D block [$$(X, 1, 1), X \leq 1024$$],
- 2D block [$$(X, Y, 1), X\times Y \leq 1024$$],
- or 3D block [$$(X, Y, Z), X \times Y \times Z \leq 1024$$]

-- is up to you. This can depend on the kind of program you are writing. So, you can find good reasons to use one kind of block dimensionality even though all 3 are available. Keep in mind, just usable is not enough - we want the best in terms of speed and/or efficiency. For small arrays this decision is trivial. You can choose any and they will work.

Here is a realistic example of when choosing the right block dimensionality is crucial:

Suppose you're working with very large matrices, and your kernel is some kind of iterative problem solver. The algorithm for that solver is such that the matrix of shape $$(M, N)$$ must be treated as comprised of sub-matrices of size $$(q, p)$$ $†$, and some affine matrix transformation has to be performed on those sub-matrices iteratively. In this scenario, while one could write a kernel using 1D thread indexing, but quite obviously, it'll be easier to write code utilizes 2D indexing.

$†$: *for example, a $$4 \times 4$$ matrix can be viewed as composed of 4 sub-matrices of shape $$(2, 2)$$*

Threads vs. Cores

There is one key rule you must remember when programming with CUDA. From a programmer's perspective, CUDA cores are 'invisible' to you, and so the number of CUDA cores available on your GPU is irrelevant. Here's why:

- Recall that the K80, for example, has 26 SMs, which together contribute toward a total of 4992 CUDA cores.
- Note how 4992 isn't a multiple of 1024. And yet, the upper limit of 1024 threads per block is a hardware limit.
- This should be a clue that its a bad idea to assume that there's a one-to-one correspondence between the number of CUDA cores being used and the number of threads running at execution time. *There isn't.*

If you are still not convinced, consider this - the CUDA APIs manage which threads are executed on which streaming processor/core of an SM, while the runtime API manages contexts. There could be more than one context running on the same device, which means there is simply no way of identifying which thread is running on which core, and indeed, even if one core is servicing just one thread. As this Nvidia engineer puts it -- there really is no sensible concept of threads running on a CUDA core.

In summary, it might or might not be accurate to assume # cores = # threads, especially with more recent Nvidia GPUs. However, there is no way of knowing for sure, and trying to find out is not possible.

Misc Notes.

1. Some additional information about the Tesla K80 card: it actually houses *two* chips. (Unimportant but these are the Kepler GK210 chips, Kepler being the architecture that Nvidia was using at the time). From the perspective of our course, when we talk about a GPU, these individual chips are not what we're referring to, but the two as a whole. Back in the early days of GPU programming most graphic cards only had one chip, so no distinction was necessary. So, when we query the device in CUDA (or OpenCL), it doesn't return the resources available on a per-chip basis. The numbers we see are the collective resources of both chips that constitute the graphic card. Each chip has 13 SMs.
2. This article will be updated with information about warps once the concept is introduced in class.

Clone this wiki locally

```
https://github.com/eecse4750/e4750_2021Fall_students_repo.wiki.git
```