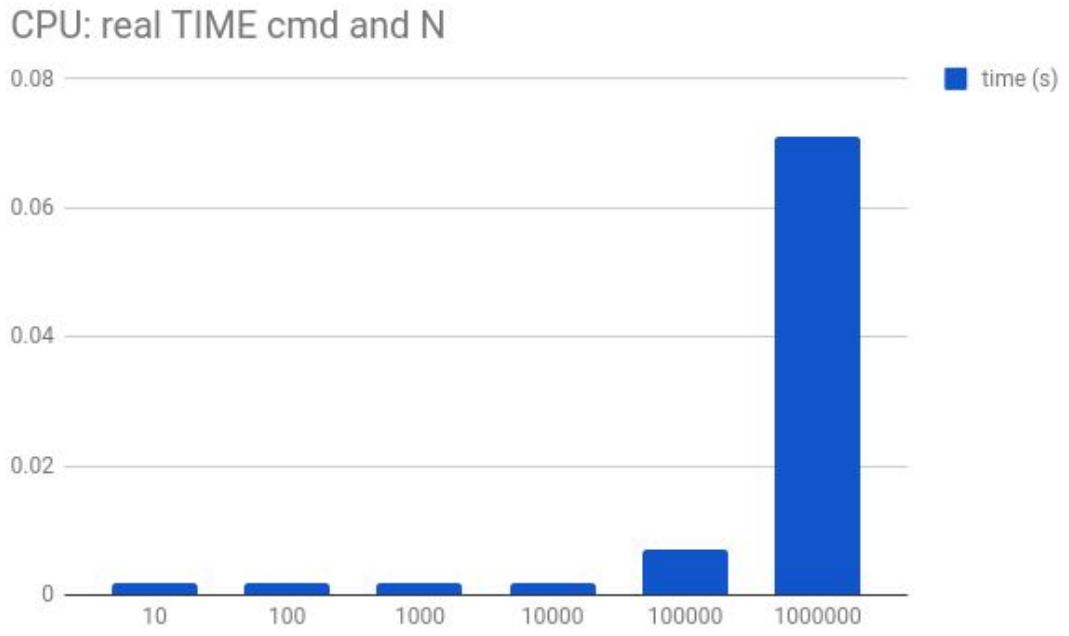
Michael Lukiman
Professor Z
Nov 2018

CSCI-GA.3033-004 Graphics Processing Units (GPUs): Architecture and Programming
Lab 2 - Prime Sieve Variations on GPU vs. Sequential

I utilized cuda5 with settings *nvcc -o Xcuda -arch=sm_35 CUDAprimes.cu* and then made a bash function to loop through different parameters. I used argv[1] parameters inclusive of the instructions (10^1 … 10^n).

For the sequential implementation, I broke down the sieve into 2 main steps: shooting the multiples of a number (eliminating nonprimes) and the loop of which numbers to shoot multiples of. Since all multiples of 2 will be eliminated first, there will not be any redundant elimination, e.g. shooting the multiples of 4 is redundant. The "shoot-loop" verifies this since it iterates from small to large. Then the "shoot" function actually eliminates multiples of the passed-in number. It does not divide, instead, it 'walks' along the nonprimes through addition, marking them.

For a baseline, the speed of sequential CPU-only case is plotted as followed, with a log10 x-axis (absolute numbers as shown averaged).



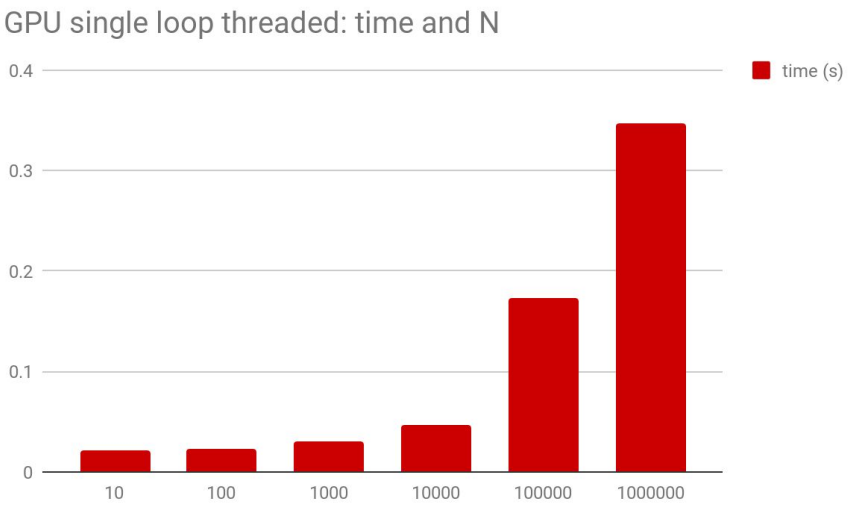
By a factor of 10, the runtime increases by a proportional factor, as expected with sequential code.

For the GPU variation, it can actually be parallelized in two ways, corresponding to the 2 main 'for' loops above.

1) I made a variation where each thread would help run a single shooting session, i.e. shooting 4, 6, 8, 10, 11 … 2*n in parallel, then multiples of 3 in parallel, and so on.
2) Another variation is each thread running separate shooting sessions in sequence, i.e. TID 0 eliminates 4, 6, 8, … in sequence while TID 1 eliminates 3, 6, 9, … in sequence at the same time. Note that this has some redundancy, (i.e. 6 is marked twice) but only one additional kernel needs to be launched, as opposed to the previous one where a kernel is launched per number in sequence.

Both implementations have a parallelized generator function which populates the array with [2 through N]. One of the above implementations coalesces with smaller intervals while the other does not. The question is, given the overhead of launching kernels, what is faster? Cue the quantitative data.

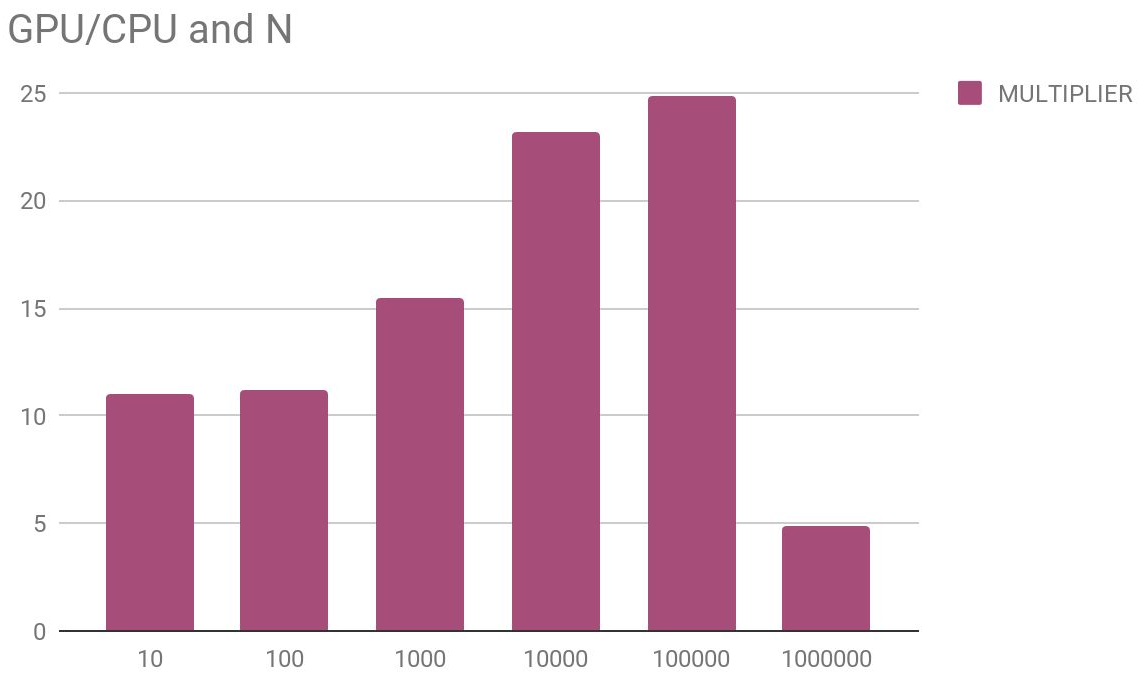
First, a disclaimer - the cuda5 platform was sometimes on heavy load during these tests, even after the averaging. Thus, cudaDeviceSynchronize and cudaMalloc time scales may be dependent on other students' jobs running at the same time. We will treat data on this case-by-case consideration, and look at rates of growth. These include times of waiting for all considered.



For the first approach, we can safely say that parallelization helps in the proportion of growth, as a factor of 10x N requires a runtime less than 10x longer, e.g. ~2x for the latter two. Let's dive into nvprof outputs for a more concrete breakdown:

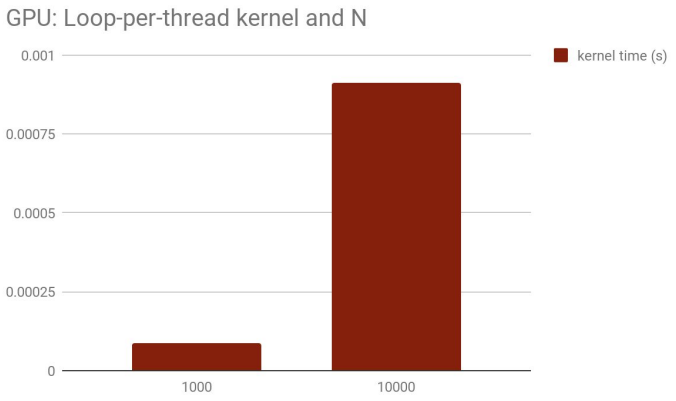
```
==133715== NVPROF is profiling process 133715, command: ./Xcuda 1000000
==133715== Profiling application: ./Xcuda 1000000
==133715== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.89%  131.45ms     11041  11.905us  11.456us  20.896us  shoot(unsigned int, unsigned int,
unsigned int*)
  1.09%  1.4543ms         1  1.4543ms  1.4543ms  1.4543ms  [CUDA memcpy DtoH]
  0.01%  14.368us         1  14.368us  14.368us  14.368us  generate(unsigned int, unsigned int*)
  0.00%     800ns         1     800ns     800ns     800ns  [CUDA memset]

==133715== Unified Memory profiling result:
Device "GeForce GTX TITAN X (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   31718  22.336KB  4.0000KB  60.000KB  691.8750MB  82.87197ms  Device To Host
```

The actual nonprime elimination set of kernels takes around 131ms total. Indeed, we can call it out here that under cuda5's realistic conditions and multiple kernels, it's actual heavier than the sequential approach at N = 1,000,000. The generator function is quite negligible and very fast at 14 microseconds. Again though, the rate of growth is forgiving. When comparing the ratio of GPU to CPU:



It ranges from a multiplier of 10 to 30, but gains some drastic advantages at N = 1 million on cuda5, since synchronization takes only twice as long whereas the CPU increase by a factor of 10, thus explaining the ~⅓ improvement. This takes advantage of all threads in strides. Again, these are from organic numbers, of which many causes can be attributed in realtime. Since the stride does not come into

play as much at small N, parallelization is not utilized to its fullest per launch. In the second variation, it improves per thread, but each thread is waiting for the bottleneck thread, which is the multiples of 2 (it eliminates the most nonprimes, and takes the longest). Thus it exhibits a pattern similar to sequential, although faster in this simple example:



The results are humbling, and shows that parallelizing primes seems to be an art of barrier establishing, i.e. when to launch the fewest amount of kernels while utilizing the most amount of threads. When one thread finishes, it should devote to help the longer running thread. Thus, we can come up with an ad-hoc strategy to reassign finished threads to help with finer multiples (i.e. x2, x3, etc), or divide those longer running threads into multiple threads, starting at different sections of the array. Block dividing can provide a different strategy.

Introducing an advanced feature, streams: we will create sets of multiples we are testing, from 2 to N+2/2. These will be chunked into however many streams we wish to use. Here I use 16 streams, since 8 returns slower as well as 32 when tuning. However, since streams ultimately have kernels that take up resources concurrently, there is a tradeoff, as seen in the nvprof of N = 100000.

```
==40757== Profiling application: ./XStreamsCuda 100000
==40757== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.92%  61.126ms        16  3.8204ms  3.6528ms  4.0000ms  shoot(unsigned int,
unsigned int, unsigned int*, int)
  0.07%  43.296us         1  43.296us  43.296us  43.296us  [CUDA memcpy DtoH]
  0.01%  4.3520us         1  4.3520us  4.3520us  4.3520us  generate(unsigned int,
unsigned int*)
  0.00%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memset]
```

The 16 streams (calls) add up to a higher value than the non-streamed version, displayed below:

```
==41028== Profiling result:
```

```
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.73%  18.624ms      5129  3.6310us  3.5200us  15.264us  shoot(unsigned int,
unsigned int, unsigned int*)
  0.23%  43.711us         1  43.711us  43.711us  43.711us  [CUDA memcpy DtoH]
  0.02%  4.6080us         1  4.6080us  4.6080us  4.6080us  generate(unsigned int,
unsigned int*)
  0.01%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memset]
```

It a full magnitude's worth faster. Thus, experimenting with different parallelization implementations, we have compared different complex techniques. It turns out that quantitatively, the interleaved option runs the quickest due to lowest average kernel time, even if it launches thousands of times. The code with advanced functions is also included, as well as the parallelization of the other loop (in a way gather vs. scatter). Precise functions like cudaMalloc and array chunking is successfully implemented in the streamed version, as can be introduced to the other files. As programmers become more free in their ability to utilize advanced features, it is a caveat that using these features do not necessarily promise a speedup - i.e. should be tested and tuned to the problem and problem size!