Graphics Processing Units - Fall 2018
Programming Assignment 2

Daniel Rivera Ruiz
Department of Computer Science
New York University
daniel.rivera@nyu.edu

1 Results

Table 1 shows the median running times (measured over 5 repetitions using the `time` command) for the two programs `seqgenprimes` (CPU) and `genprimes` (GPU) for different values of N. Figure 1 presents a bar chart for these results and a linear plot for the Speedup calculated as CPU time / GPU time.

Table 1

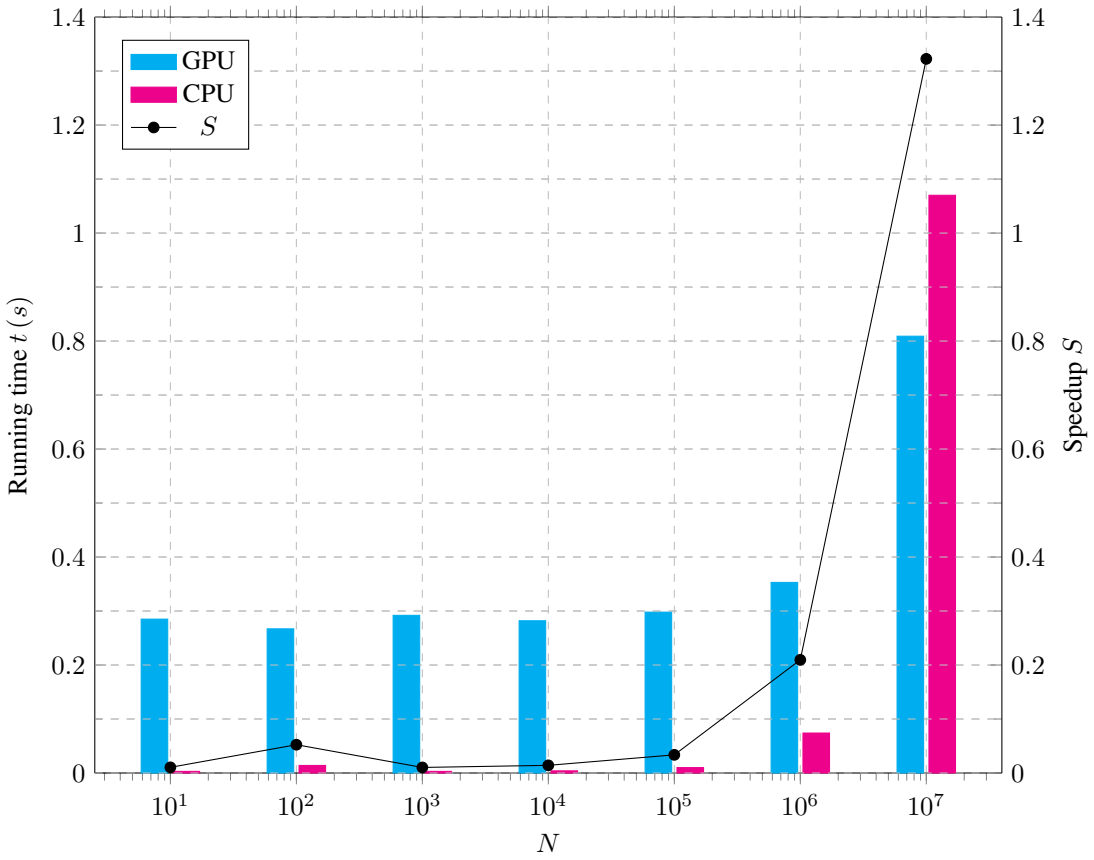
	Execution time $t(s)$		
N	GPU	CPU	Speedup
10	0.285	0.003	0.0105
100	0.267	0.014	0.0524
1 000	0.292	0.003	0.0103
10 000	0.282	0.004	0.0142
100 000	0.298	0.010	0.0336
1 000 000	0.353	0.074	0.2096
10 000 000	0.809	1.070	1.3226

2 Explanation

Looking at the results from the experiments, it is clear that this algorithm does not benefit from being implemented in a GPU instead of a CPU. The main reasons that support this argument are the following:

1. According to *nvprof*, a call to `cudaMalloc` to allocate 1 byte of data in the device takes no less than $200ms$. This number remains very similar regardless of the amount of bytes being allocated, which means that there is an inherent overhead to calling `cudaMalloc` that cannot be avoided. If the time required for a sequential algorithm to run in the CPU is less than this overhead (as it is the case of our algorithm for most values of N), then running the algorithm in the GPU will always be slower.
2. In addition to the overhead generated by `cudaMalloc`, there is also the overhead generated by `cudaMemcpy`. This value remains constant at around $35\mu s$ when copying a few bytes (similar to the constant overhead generated by `cudaMalloc`, but will grow linearly with the number of bytes being copied for larger values.
3. *Unified Memory* (which is the approach used in `genprimes.cu`) can help reduce the overall overhead for large values of N because it eliminates the need to perform the explicit transfer of data with `cudaMemcpy`. However, the inherent overhead associated to `cudaMallocManaged` is similar to that of `cudaMalloc` and therefore the GPU algorithm is still slower than the CPU one.
4. Given these conditions, it is clear that the bottleneck of the distributed algorithm will reside in the memory allocation for the device, accounting for over 75% of the overall execution time for most values of N. This means that even if we were to reduce the time taken to actually launch the kernel

Figure 1



(say by allocating shared memory or improving the algorithm itself) it wouldn't have a big impact in the overall execution (Amdhal's Law).

5. Finally, there is also the fact that the final step of the algorithm (writing the results to a file) cannot be parallelized and must always be executed in the CPU. The reasons for this are simple: writing to a file is a serialized process (there cannot be multiple threads writing to the same file at once), plus the GPU has no notion of files nor it has access to the file system, so all the results obtained in the GPU must go to system memory first.

It is worth noticing, however, that for the largest value of $N = 10\,000\,000$ considered in this assignment, the GPU starts to outperform the CPU by a little margin. This means that for even larger values of N we could expect the GPU to perform even better, since the overall execution time of the algorithm will keep increasing and the effect of memory allocation and data transfer overheads will be shrinked.

Table 2: Median time (in ms) required by CUDA API calls for different amounts of bytes according to *nvprof*

No. of Bytes	`cudaMalloc`	`cudaMemcpy`	`cudaMallocManaged`
1	213.030	0.034518	205.220
10	213.550	0.033841	214.070
100	217.200	0.030343	202.690
1 000	210.300	0.028796	217.090
10 000	209.990	0.037619	208.120
100 000	203.640	0.344380	232.390
1 000 000	183.750	1.778	205.010
10 000 000	200.020	12.992	207.360

Figure 2: Median time (in ms) required by CUDA API calls for different amounts of bytes according to *nvprof*