Lachlan Sinclair

sinclala@oregonstate.edu

5/17/2020

CS 475 Spring

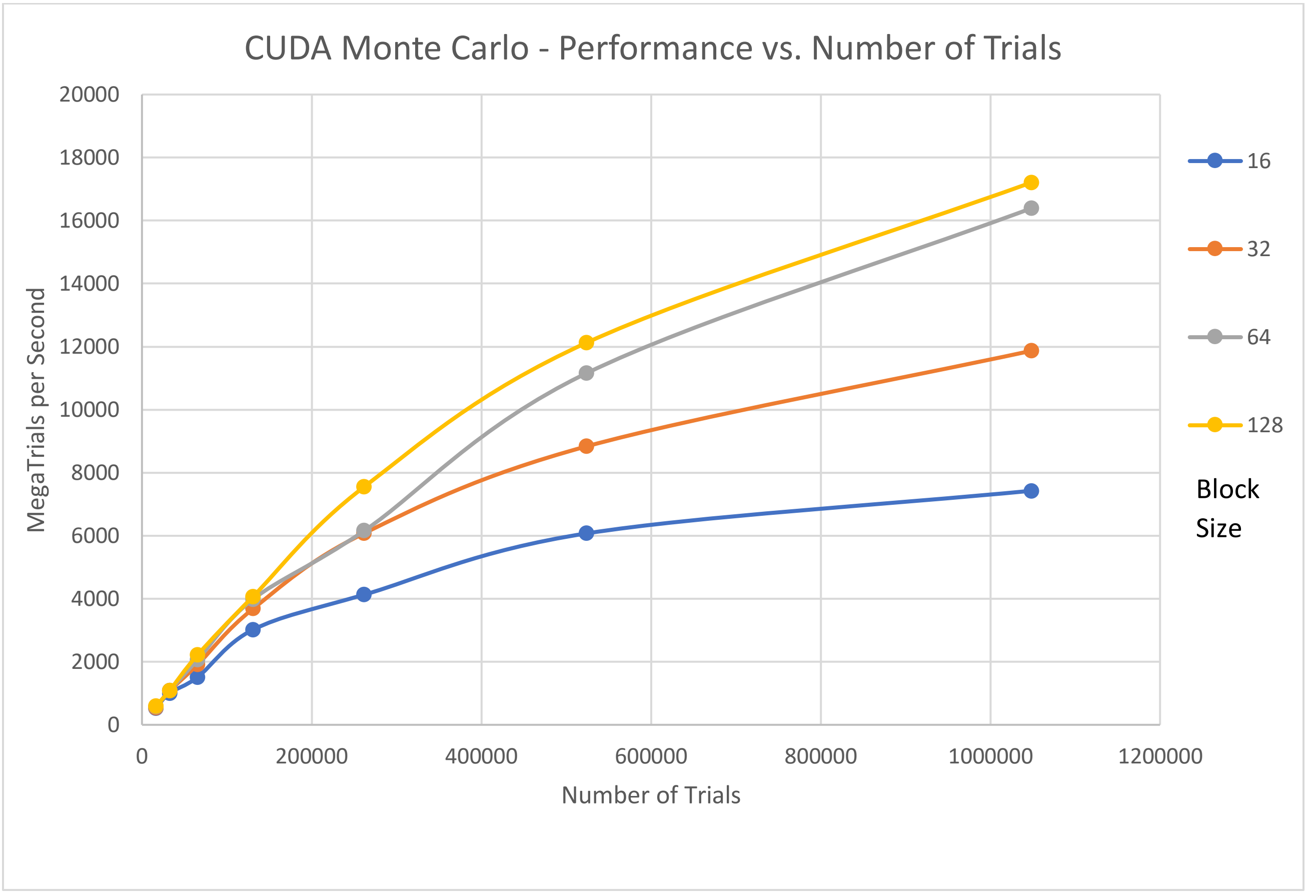
Project #5: CUDA Monte Carlo

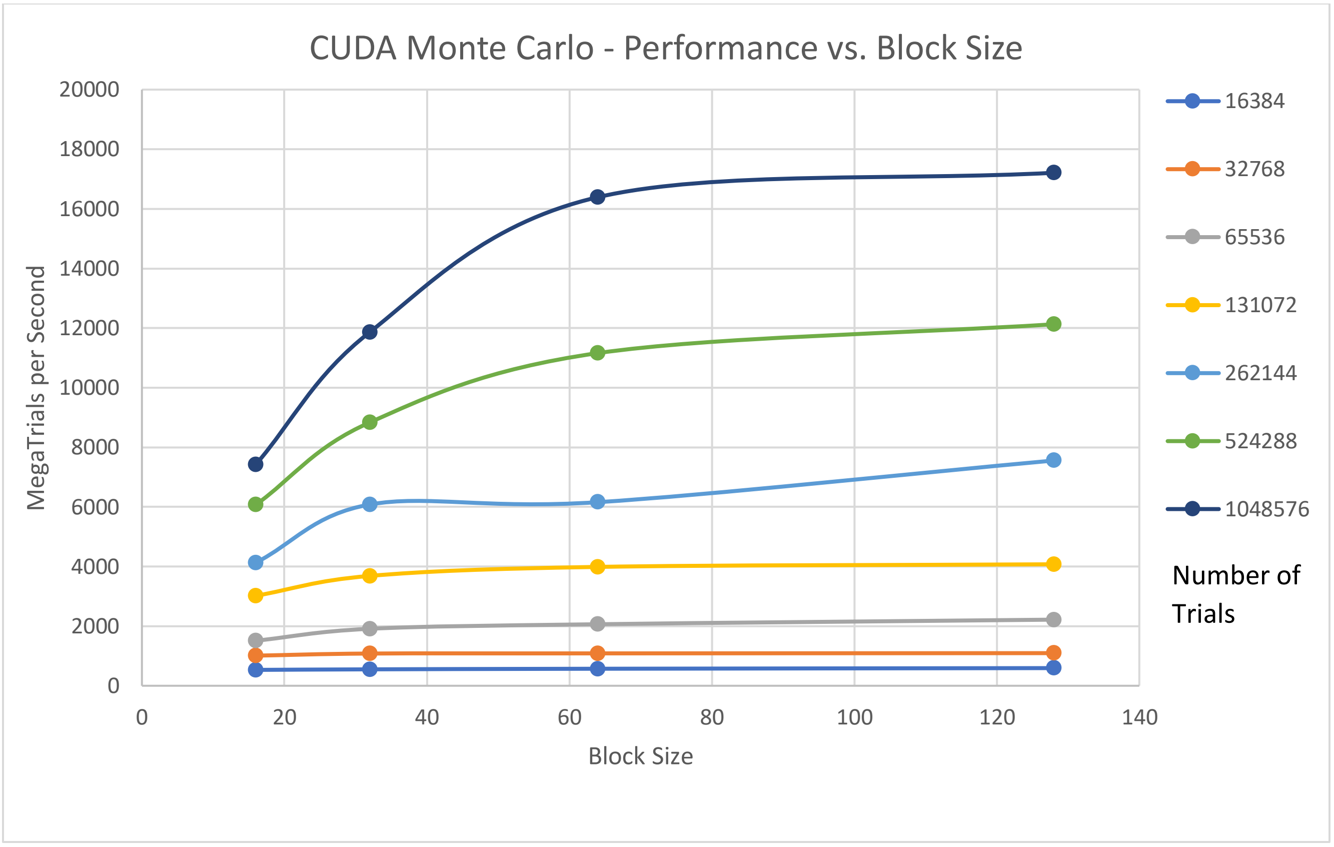
System: For this project I used the DGX system. I generated a script as described in the DGX lecture to submit a CUDA job.

Results:

Block Size	Number of Trials	MegaTrials/Second	Probability
16	16384	533.3333	42.108
16	32768	1010.8588	42.065
16	65536	1512.5554	41.905
16	131072	3020.649	42.267
16	262144	4131.1145	41.912
16	524288	6074.8979	42.037
16	1048576	7427.0171	42.08
32	16384	551.7241	42.316
32	32768	1081.3094	42.331
32	65536	1912.2315	42.459
32	131072	3686.7688	42.005
32	262144	6081.6629	42.062
32	524288	8837.1088	41.911
32	1048576	11868.1634	42.01
64	16384	571.4286	41.455
64	32768	1087.0488	42.239
64	65536	2064.5162	41.858
64	131072	3988.3154	41.988
64	262144	6159.3985	41.831
64	524288	11160.7629	42.148
64	1048576	16392.1954	41.998
128	16384	592.5926	41.553
128	32768	1095.1871	41.382
128	65536	2218.8516	42.105
128	131072	4075.622	42.11
128	262144	7557.1957	41.959
128	524288	12127.3127	41.985
128	1048576	17210.0845	42.002

	16384	32768	65536	131072	262144	524288	1048576
16	533.3333	1010.859	1512.555	3020.649	4131.115	6074.898	7427.017
32	551.7241	1081.309	1912.232	3686.769	6081.663	8837.109	11868.16
64	571.4286	1087.049	2064.516	3988.315	6159.399	11160.76	16392.2
128	592.5926	1095.187	2218.852	4075.622	7557.196	12127.31	17210.08





Explanation:

This program works by utilizing CUDA to perform the Monte Carlo calculations for randomly generated values through GPU parallelism. The code is quite similar to project one's code, however rather than using OpenMP the Monte Carlo function is ran on the GPU through CUDA. Instead of using a loop as seen in the OpenMP version of the project, the CUDA implementation relies on using the gid to access specific elements of the randomly generated arrays. Doing this creates an implied for loop and ensures that the GPU performs the calculation for all values in the randomly generated arrays. These calculations are done in parallel to one another, so we expect see a performance increase.

In the main table the probability for this configuration of the Monte Carlo simulation produces a probability of roughly 42%. This differs from project one since the conditions for the simulation are different.

Analysis:

The performance of all block sizes gradually rose as the number of trials rose, the larger the block size the more significant the increase in performance was. I hypothesize this continual improvement based on array size is because as the number of trials grow, the amount of time spent by the GPU cores performing operations in parallel dominates the time required for the CPU to make the call to the GPU. Also, this could be due to allowing for a single instruction to be used on more data, effectively utilizing more of the CUDA cores (arithmetic logic units) without having to alter the instruction cache which increases the computations per second.

An interesting trend in the performance vs. block size graph is that the lines for 16k-131k number of trials stay almost perfectly level. This indicates that regardless of the block size, if the number of computations is small a significant increase in performance will not be obtained. I think this is because with that small amount of data, using the four tested block sizes, the GPU can process all elements almost immediately using the provide cores. Once the array size increases past 260k the lines are longer level, indicating that this is no longer the case.

When looking at the larger number of trials in both graphs it is apparent that block sizes of 64 and 128 dominate in terms of performance. A block size of 32 still performs reasonably well, and a block size of 16 is by a wide margin the worst performing. This is because 16 is half the size of a warp and the other three block sizes are all intervals of a warp. A block size of 16 is causing 16 arithmetic units in that warp to be inactive which in turn causes a huge hit in performance. With a block size of 64 and 128 there are additional warps that can step in and execute when another warp is blocked for whatever reason, giving them a performance advantage over a block size of 32.

In project one I only ran tests up to a trial count of 500k. Comparing the fastest run on a trial count of 500k to the fastest performance using the GPU on similar trial size (524k) you get a speed up of (12127/197=61.5) 61.5. A speed up of that magnitude is massive, using the GPU to parallelize this function out classes using CPU multithreading. The nature of this problem is data parallel which lends it to run quickly on a GPU. A standard set of calculations is being performed on all of the elements in the arrays which can be described as simply streaming data through an identical set of processes which is how the lectures described the ideal usage of GPU.

Also, I found the if statements to be of interest, the lectures mentioned how GPU's handle them very inefficiently because they execute both branches regardless. However, in this program, one branch in all the if statements has no code to execute so the performance loss should be small if anything.

The sweet spot of block size as discussed in the lectures appears to be around 128 as the performance gain between 64 and 128 is not that large. For this problem and the benchmarks produced, a block size of 128 appears to be a good trade off in terms of the number of threads per block and the number of blocks produced by that block size for the array sizes tested.