John Olgin
olginj@oregonstate.edu
CS 475 – Spring 2019
Project #6
CUDA Monte Carlo

Description

I'll be analyzing the effects of GPU parallelism on the performance of a simple Monte Carlo simulation. This program was developed on macOS Mojave Version 10.14 using Sublime Text. However, CUDA is only supported on Nvidia GPUs, so all testing was done on the rabbit machine provided by Oregon State University.

The ultimate goal here is to calculate the probability of a laser beam hitting an infinite plate, which is adjacent to the beam's origin and runs along the x-axis. The beam will move in a positive x and y direction. A circle will be randomly placed in the beam's path, right on top of the beam's origin, or completely out of the beam's path. The CUDA program will generate a probability of the beam hitting the circle and deflecting into the infinite plate. To utilize CUDA correctly, I filled three arrays containing randomly generated values for the x-coordinate, y-coordinate and radius. I also created one more array of the same size to hold a flag value signaling whether or not the plate was hit for set of values. The four arrays were all passed into the CUDA-approved function call. This function was where the CUDA computation actually determined if the plate was hit.

Since I was measuring the effects of numtrials and blocksize on performance, I kept track of calculations per second performed by the program for different blocksizes while holding numtrials constant, and vice versa. The unit of measurement used is MegaTrials/s, which is calculated as follows:

(size*numtrials) / duration / 1000000

I used only the values for numtrials and blocksizes stated in the Project 6 requirements section. Blocksize describes the number of threads used on each block and numtrials describes the size of the overall problem in total simulations. The raw data, graphs and commentary for my results are provided below.

Data

Numtrials

Blocksize	16000	32000	64000	128000	256000	512000
16	0.41	0.42	0.42	0.42	0.42	0.42
32	0.41	0.42	0.42	0.42	0.42	0.42
64	0.41	0.42	0.42	0.42	0.42	0.42

Probability

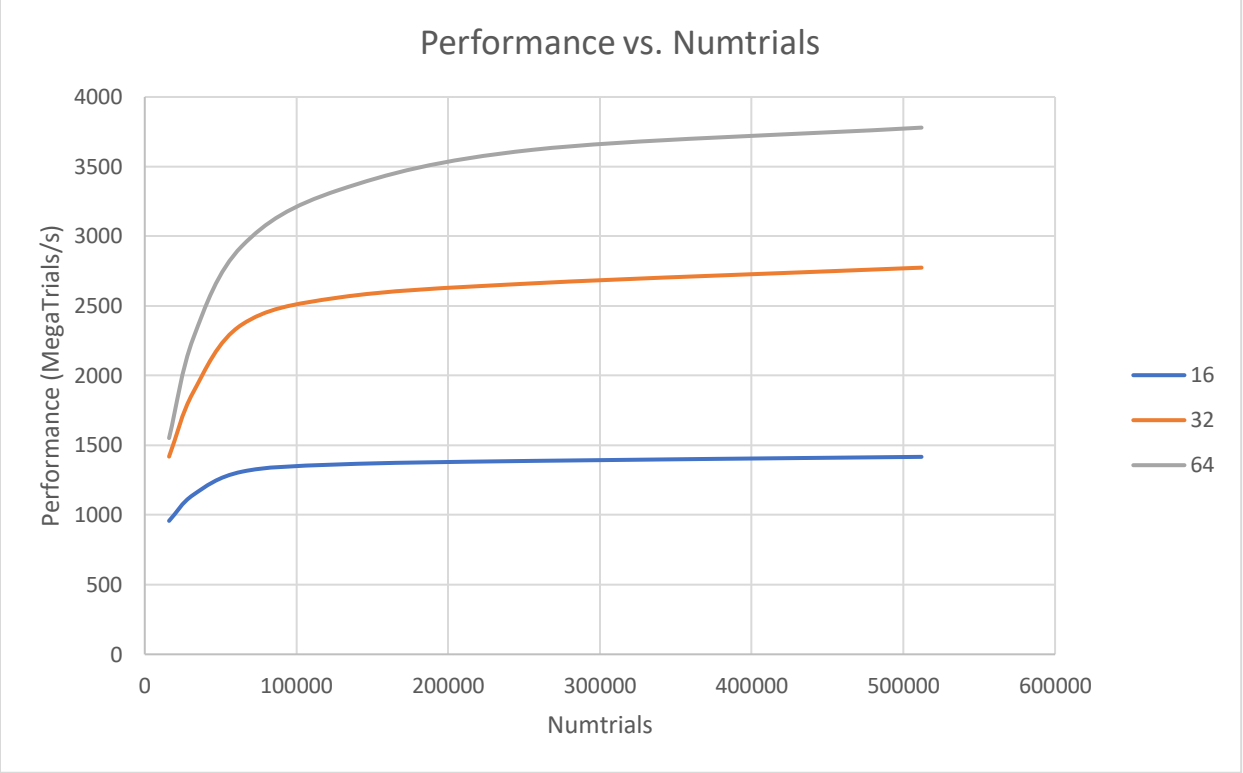
		Numtrials					
Blocksize		16000	32000	64000	128000	256000	512000
	16	957.71	1145	1310.62	1362.94	1388.29	1417.24
	32	1419.04	1882.39	2366.47	2560.89	2662.63	2775.06
	64	1551.93	2269.89	2929.33	3332.56	3621.12	3780.09

Measured in MegaTrials/s

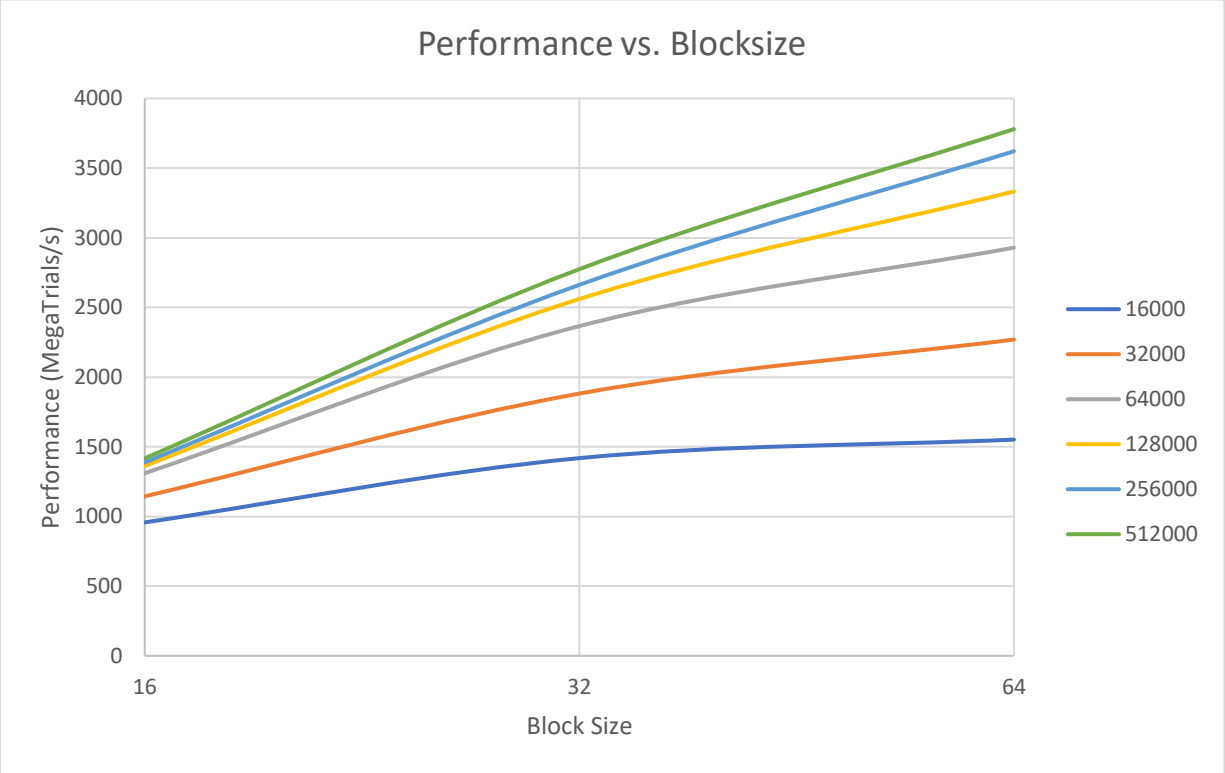
Probability is around .42 for all combinations above 32000.

Graphs

Graph 1



Graph 2



The shape of the curves in Graph 1 show us that problem size only has a significant effect on performance up to a certain value. The 32- and 64-thread block sizes both experienced sharp increases in performance within the 16000- and 64000-element dataset range. However, this is followed by a slowed rate of performance increase around the 100000-element mark, which is maintained, and seemingly becomes more constant, throughout the rest of the graph. The 16-thread block size didn't experience nearly the same magnitude of performance increase as the other two sizes. While their performances increased two-fold or more, the 16-thread block size experienced an increase of less than 50% over the graph's entirety. It also appears to become nearly constant at a smaller dataset size, around 64000.

Considering how CUDA works, the curves' patterns are reasonable. As the size of the problem increases, the scheduler also has to schedule increasingly more blocks for each multiprocessor. In other words, the scheduler will have to do more work as we increase the size of the dataset since block size is held constant. SMs will also be under a heavier load since they will have to perform more computations as the problem expands. Given the increased workload on both the scheduler and SMs, it's intuitive that computations will take more time and the rate of performance increase will decelerate. At this point, the only way to significantly increase performance is to increase block size again.

For Graph 2, performance growth is more consistent and doesn't taper off as quickly. A few curves are almost linear due to their steady increase in performance from one block size to

another. Additionally, the larger dataset sizes seem to perform better at virtually every increase in block size. They also maintain their slope (rate of performance increase) more consistently than that of the smaller datasets. For example, the 128000-, 256000- and 512000-element datasets have nearly the same slope throughout the graph, almost appearing linear as previously mentioned. However, the 16000- and 32000-element datasets only experience a marginal increase in performance after a block size of 32. The rate of increase becomes so trivial that I would expect it to become nearly constant if I were to double block size one or two more times.

The pattern in these curves can be attributed to both block size and problem size. Larger problems tend to gain more efficiencies from parallelism, so it makes sense that the larger datasets have steeper slopes than do the smaller ones. This is especially true considering the smaller block sizes we're using in this project. The range of valid block size values are between 1 and 512, so using our smaller sizes on large problems will result in significant performance boosts. The SMs are kept sufficiently busy by the number of blocks. Idling time is minimized and latency is well-hidden, thus improving performance. However, if we used larger blocks, say between 256-512, we'd likely see even the larger curves stabilize. Some might have experienced a decrease in performance if block size became too large in proportion to the problem. We can see some evidence of this in the 16000-element dataset size. The problem is too small to experience significant gains in performance with our range of block sizes. It is likely beginning to produce too few blocks to schedule across the SMs to fully utilize the GPUs power, which is why it seems to be nearing its peak performance at only 64-threads per block.

In Graph 1, as mentioned above, the 16-thread block size has a significant lack of performance compared to the other two block sizes. I believe this can be attributed to the block size being less than the standard warp size of 32. When this happens, the remaining threads go unused. That is, they aren't fetching operands, writing output or executing other instructions, they're just idle. In other words, the SM is starved for something to do and its ability to hide latency is hindered, thus hurting performance. Since the other two block sizes (32 and 64) are multiples of 32, they operated with fully utilized warps and kept the SM busy. Therefore, they gained more of the efficiencies in performance.

There are implications for GPU parallelism when considering this concept. When a developer's primary focus is to create the most efficient program possible, he/she should consider the use of block sizes in multiples of 32. While this does limit the number of useful values within the valid range (32 – device's maximum block size), it also provides a very finite set of values from which the developer can find the so-called "sweet spot". This value will vary for different code, but it will be relatively easy to find since the most efficient values are already known to be some multiple of 32. Doing this will ensure the program is fully utilizing the SM with minimal to no idling time. Additionally, it's noteworthy to mention that larger block sizes aren't always the most efficient. Depending on the size of the problem, blocks that are too large may result in too few blocks to schedule over all the SMs, which can lead to an underutilized GPU. Therefore, an awareness of the overall size of the problem will also benefit the use of GPU parallelism.