ECE 60827 - Programming Assignment 1 - Samuel Nathan Gardner (gardn188)

Part A: Single Precision A * X Plus Y

For the first part of Programming Assignment 1, the single precision A*X plus Y loop was analyzed. A CPU implementation was given, and I wrote a GPU implementation which initializes vector X and Y with vectorSize random elements, calls the kernel using a constant 256 threads per block, and splits vectorSize threads across ceil(vectorSize/256.0) blocks, and after kernel execution copies the solution Y vector back from the GPU. The 'A' scaling factor in this case is a random floating point value generated in CPU code and passed to the kernel. You can see the breakdown of API calls with significant execution time percentage as well as the vectorSize tested, all data below was collected for charts with a 256 thread per block size for consistency:

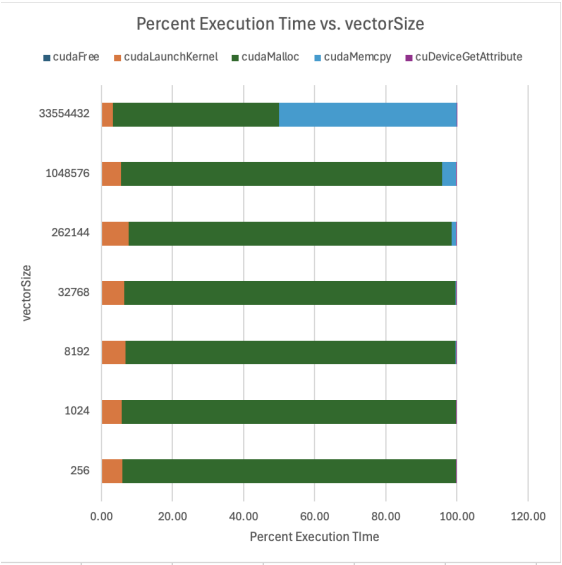

Figure 1. Percent Execution Time versus Sample Size

Data was obtained by varying the VECTOR_SIZE defined in lab1.cuh. From this data, you can see the cudaMalloc runtime dominates in most cases, except for when the vector size gets very large, then cudaMemcpy begins to dominate. When the vector size gets larger, more random numbers are being generated on the CPU and are being transferred to the device over a relatively slow PCIe bus. This is a draw-back of the required implementation, because data transfers from host to device are quite slow compared generally to compute on the device. `saxpy_gpu` was the only kernel running in testing so it took 100% of the kernel execution time.

For total execution time, a breakup is reported below for the GPU activities as they scale with vectorSize:

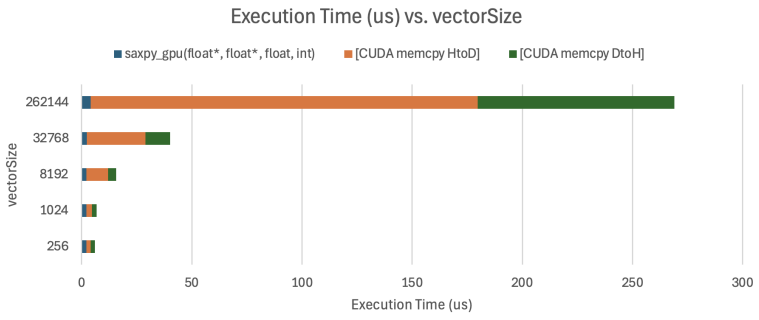

Figure 2. Execution Time (us) vs. vectorSize

You can see that execution time across the data transfers scale as the vector size scales, but that the kernel execution time scales more manageably. This is clearly showing data transfers over PCIe are rapidly deteriorating performance by using more execution time with larger vectorSize.

Part B: Monte Carlo estimation of the value of pi

For the second part of Programming Assignment 1, Monte Carlo simulation was performed for approximating the value of pi. The approximation was done by a series of threads, all responsible for generating sampleSize random points in a unit square, and calculating the sum of the points that fell within a quarter unit circle inside the unit square, which was considered a hit. After all threads calculated the hit sum for sampleSize random points in each thread, an array of size generateThreadCount (the number of threads generated) was transferred back from the device to the host and summed for a total hit count on the CPU. An average (approximately equal to pi/4) was then able to be calculated by dividing the total hit count by the number of generated threads times the sample size (i.e. the number of attempted random points). This average was multiplied by 4 to obtain pi. You can see the breakdown of API calls with significant execution time percentage as well as the (Threads, SampleSize) combination tested, all data below was collected for charts with a 256 thread per block size for consistency:

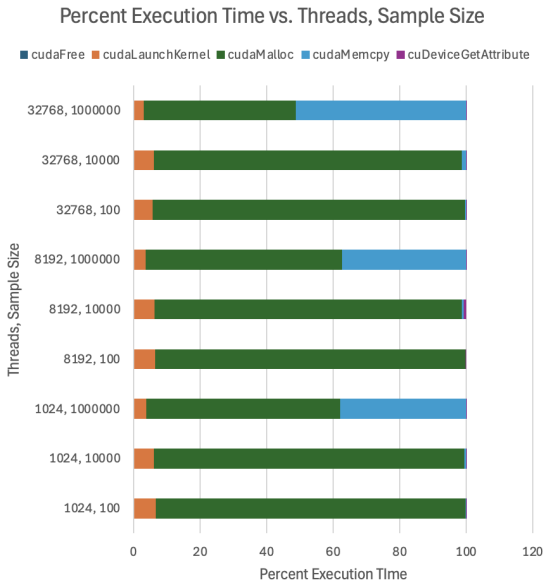

Figure 3. Percent Execution Time versus Threads, Sample Size

Data was obtained by varying the GENERATE_BLOCKS (generateThreadCount) and MC_SAMPLE_SIZE (sampleSize) defined in lab1.cuh. Similarly to SAXPY, cudaMalloc dominates in most of the combinations listed above. For the cases with a very large sampleSize, cudaMemcpy begins to take a very large portion of the runtime. This is caused by inefficiencies in the required algorithm with data transfer using cudaMemCpy back from the device to host, because with the large sample size comes larger sums and more data to transfer over the relatively slow PCIe bus. `generatePoints` was the only kernel running in testing so it took 100% of the kernel execution time.

For execution time, a breakup is reported below for GPU activities with varying Threads, Sample Size combinations:

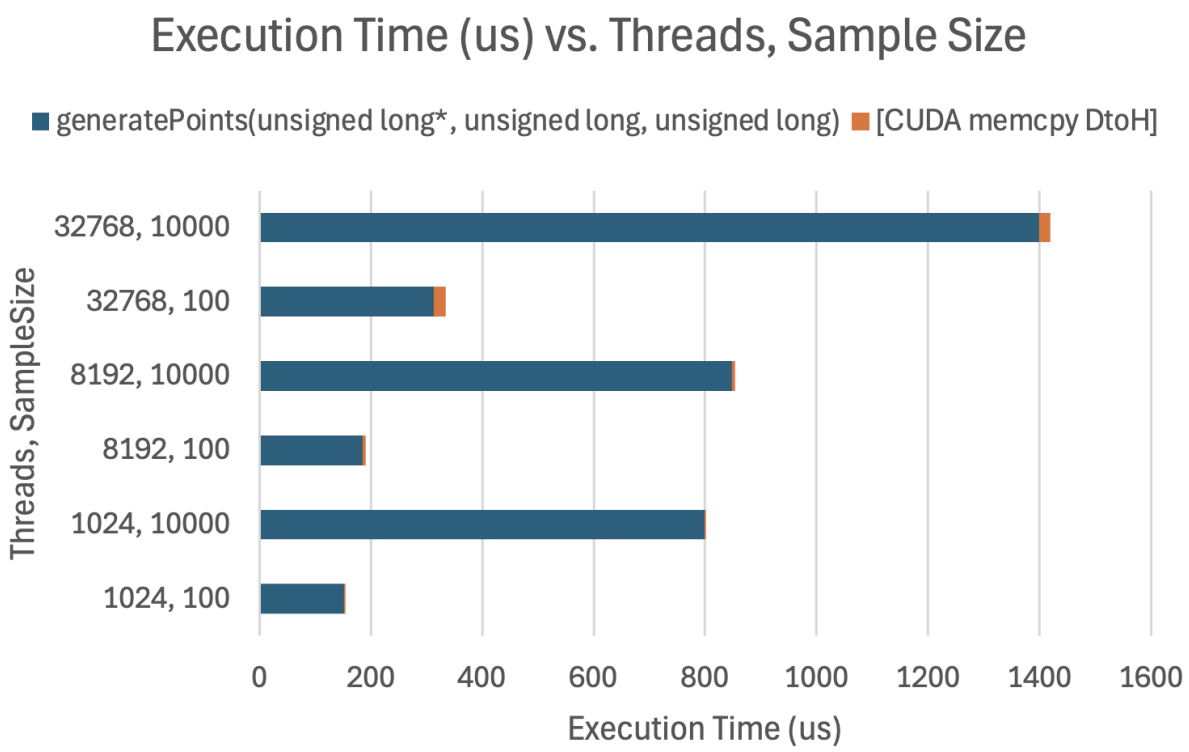

Figure 4. Execution Time (us) vs. Threads, Sample Size

The test case with a sample size of 10^6 was omitted from the chart above because it significantly supersedes the other cases in terms of execution time. You can see that for growing inputs the execution time of the kernel always grows, but especially notice that for growing number of threads, the CUDA memcpy device to host grows larger in comparison to the execution time of the kernel. This is known to be the case because the data being transferred back from the device is the size of threads, so this shows another transfer bottleneck in action.

Comparison between Part A and B

An optimization that can be made considering this data presented is designs that minimize the transfer of data between device and host and vice versa, because such data is sent over a slower interface. It is clear from the data above that complex analysis is required to determine factors related to how you parallelize algorithms, and a bad decision can have major repercussions on execution time and efficiency. It is shown above that varying some factors significantly negatively impacts transfer times and other harmful metrics for little benefit, and that there is a fine line to analyze when optimizing parallel algorithms.