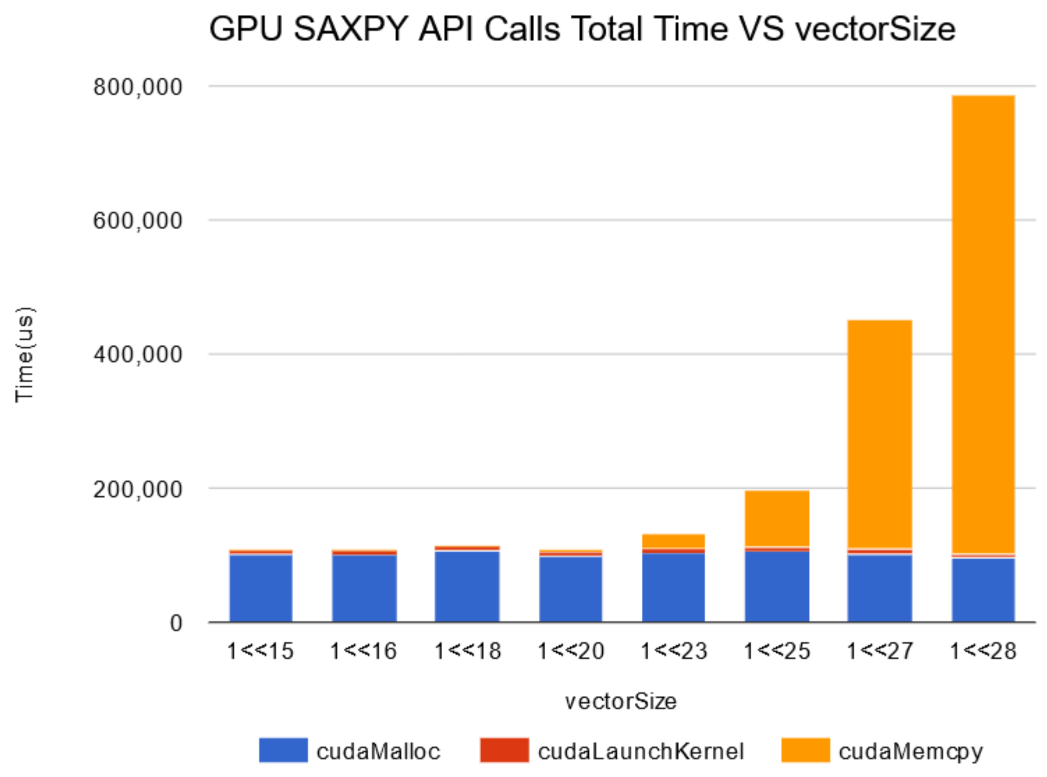
Programming Assignment 1 Report

Name: Shraddha Sahoo Username: shraddha101x

Note: For the below analysis, DEBUG_PRINT_DISABLE has been defined and there are 256 threads per block (blockDim = 256).

PART A: Single-precision A · X Plus Y (SAXPY)

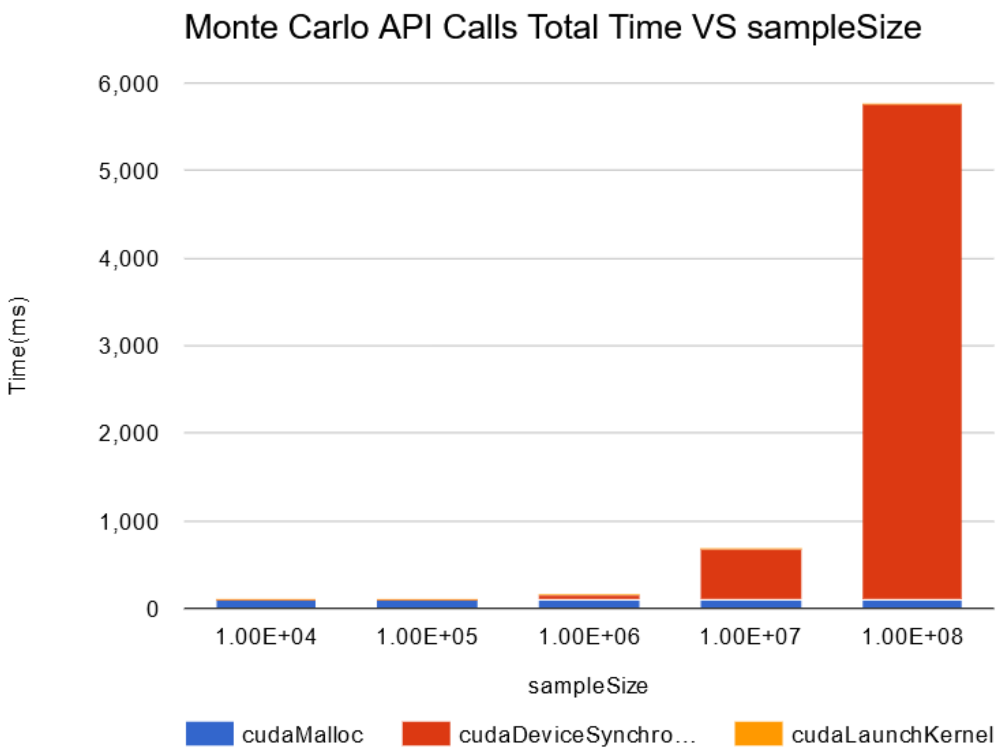


The top three major time-consumers as reported by nvprof are cudaMalloc, cudaLaunchKernel and cudaMemcpy. Other API calls are ignored because they take a low portion of the total execution time. From the above graph it can be observed that time taken by cudaMalloc is almost constant across all vector sizes. This could indicate that it takes a fixed amount of time for memory allocation regardless of vector size. The time taken by cudaLaunchKernel is the least and almost constant. This could be because the kernel is small and simple with only 1 multiplication and 1 addition operation. The time taken by cudaMemcpy seems to be increasing exponentially with vector size. This implies that with large vectors, it takes more time for transfer to data between CPU and GPU.

PART B: Monte Carlo estimation of the value of π

(a) The below analysis has been done for:

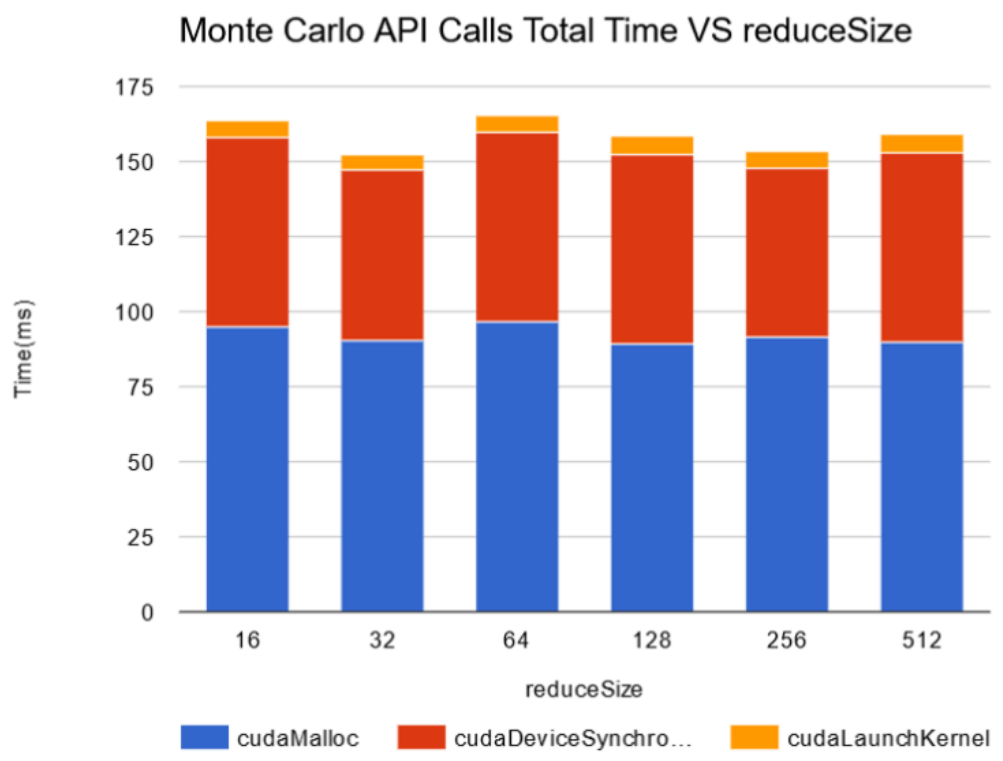
- generateThreadCount/GENERATE_BLOCKS = 1024
- reduceThreadCount/REDUCE_BLOCKS = 32
- reduceSize/REDUCE_SIZE = 32



Here sampleSize is same as SAMPLE_SIZE. The top three major time-consumers as reported by nvprof are cudaMalloc, cudaLaunchKernel and cudaDeviceSynchronize. Other API calls are ignored because they take a low portion of the total execution time. From the graph it can be observed that the time taken by cudaMalloc is almost constant across all sample sizes thus time taken for memory allocation is nearly constant. Time taken for cudaLaunchKernel is also the least and constant. As the number of sample sizes increases, it takes more time for cudaDeviceSynchronize API. For larger sample sizes, it takes longer for individual threads to complete the iterations to calculate hitpoints per thread. It is important for kernel executions to be completed before the CPU continues.

(b) The below analysis has been done for:

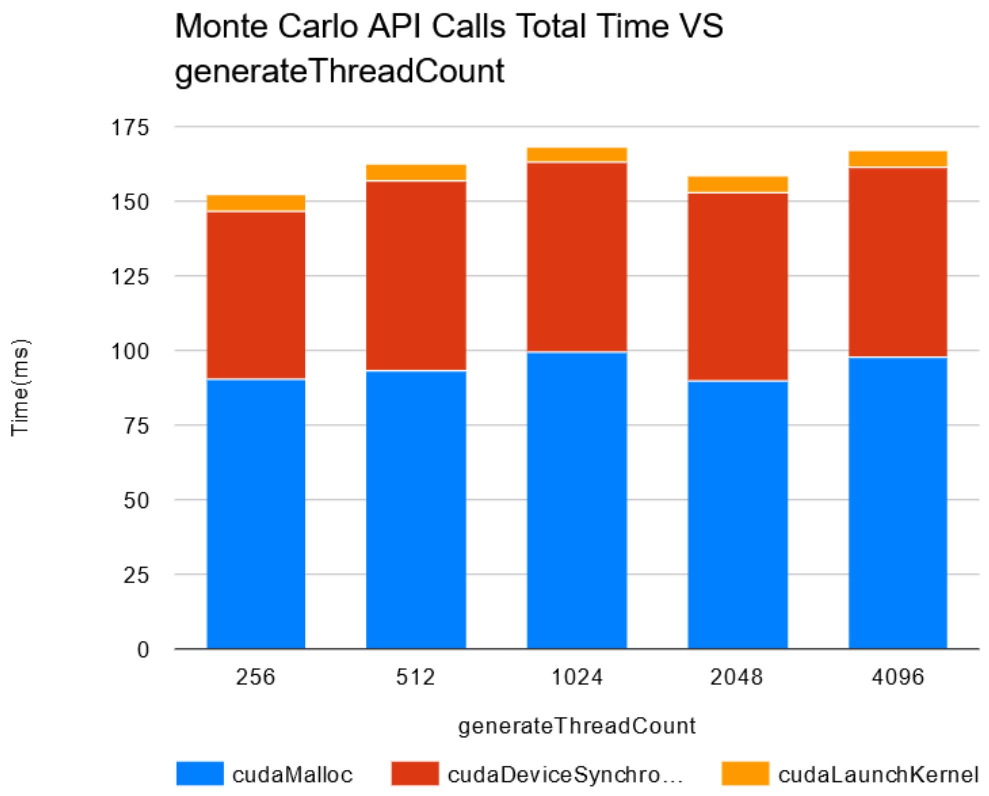
- generateThreadCount/GENERATE_BLOCKS = 1024
- reduceThreadCount/REDUCE_BLOCKS = 32
- sampleSize/SAMPLE_SIZE = 1e6



Here reduceSize is same as REDUCE_SIZE. Top major time-consumers as reported by nvprof are cudaMalloc, cudaLaunchKernel and cudaDeviceSynchronize. Other API calls are ignored because they take a low portion of the total execution time. As it can be seen from the graph, cudaMalloc, cudaLaunchKernel and cudaDeviceSynchronize take almost constant time across all reduce sizes. From previous graphs, memory allocation doesn't scale significantly with changes in reduceSize. The GPU is able to manage kernel launches efficiently, even as the reduction size changes. It likely handles the reduction in a parallel and efficient manner, thus needing almost constant time for synchronization.

(c) The below analysis has been done for:

- reduceThreadCount/REDUCE_BLOCKS = 32
- sampleSize/SAMPLE_SIZE = 1e6
- reduceSize/REDUCE_SIZE = 32



Here generateThreadCount is same as GENERATE_BLOCKS. Top major time-consumers as reported by nvprof are cudaMalloc, cudaLaunchKernel and cudaDeviceSynchronize. Other API calls are ignored because they take a low portion of the total execution time. As it can be seen from the graph, cudaMalloc, cudaLaunchKernel and cudaDeviceSynchronize take almost constant time across all generateThreadCounts.

Having almost constant time for cudaMalloc means that memory operations are not a bottleneck in this scenario. Near constant time for cudaLaunchKernel could indicate that the workload per thread is balanced, and the kernel doesn't perform significantly more or fewer computations as generateThreadCount changes. Due to fixed number of inter-thread dependencies from fixed sampleSize, cudaDevicesynchronize step completes consistently.