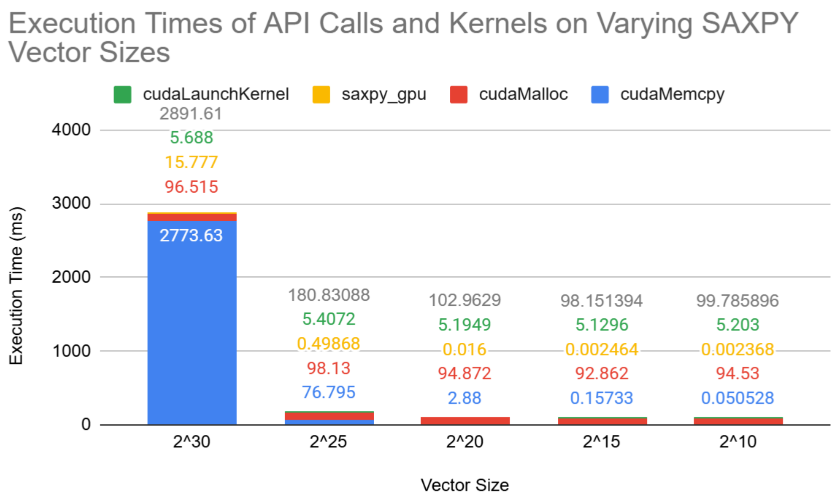
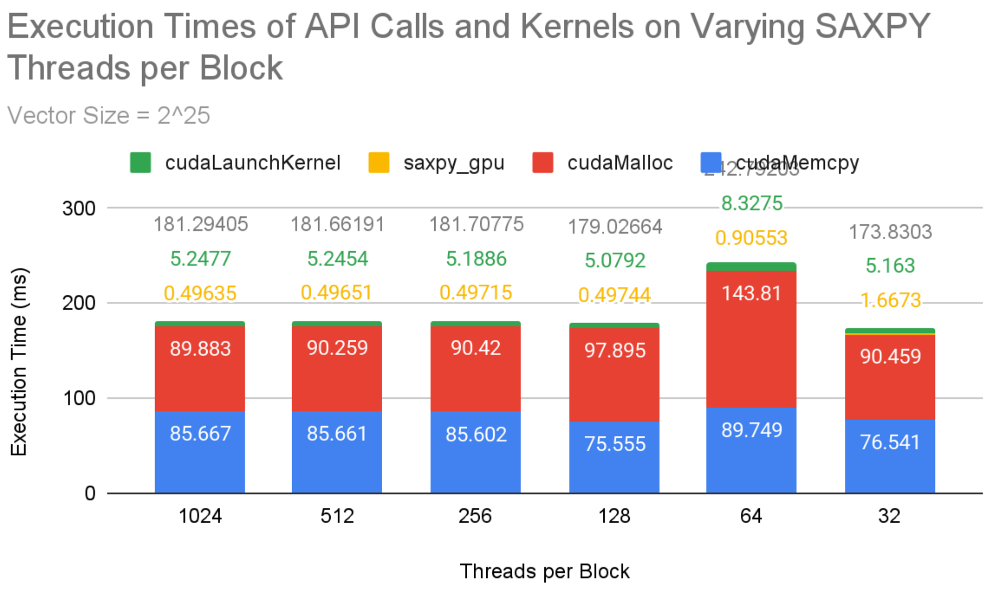
Part 1: SAXPY

The first chart varies the vector size of the SAXPY operation and measures the change in execution time due to various aspects of the CUDA program. The categories can be split into three distinct types: constant (cudaMalloc, cudaLaunchKernel), compute (saxpy_gpu), and memory (cudaMemcpy). Until the vector size reaches roughly 2^25, the majority of the execution time is spent on constant behaviors, mainly cudaMalloc. However, after the vector size passes 2^25, the execution time is dominated by cudaMemcpy. This shows that execution time is memory-bound, since copying data between device and host uses significantly more execution time than the kernel itself. Both the kernel and cudaMemcpy scale as vector size increases.

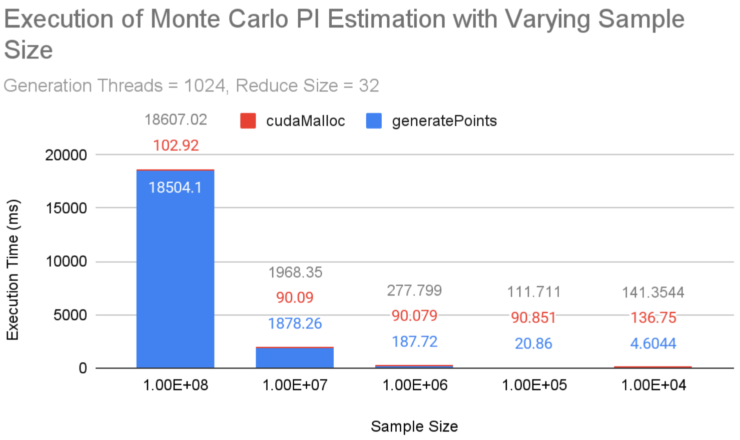


The second chart shows how execution time varies as the threads per block varies, i.e. scheduling more blocks rather than more threads per block. 2^25 vector size was chosen since this was when compute started becoming a more significant factor in execution time. No significant change was observed here, although when threads per block was very low (64 and 32), the kernel time doubled and quadrupled, respectively. This is likely because more thread blocks needed to be scheduled since there was wasted resource usage within each thread block.

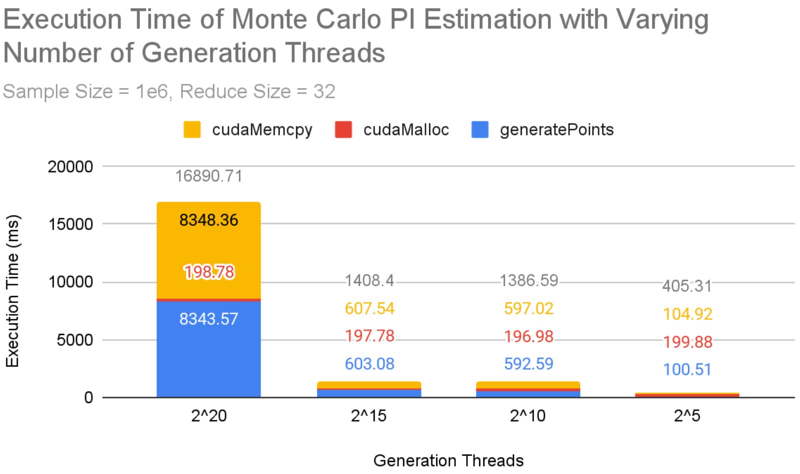


Part 2: Monte Carlo PI Estimation

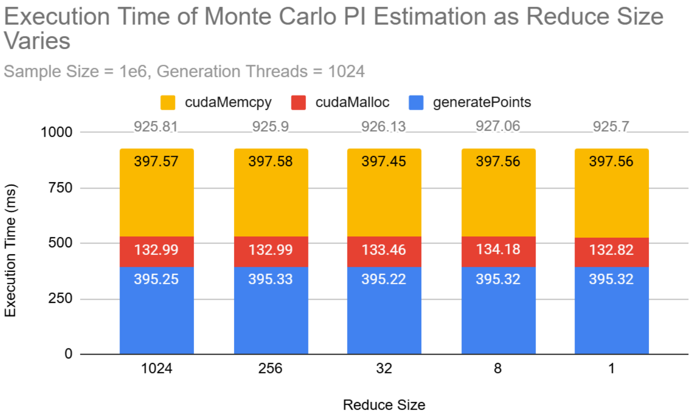
The first chart changes the sample size, the number of points calculated by each thread, and measures execution time. Unlike the SAXPY kernel, this program is compute-bound, as execution is heavily dominated by the kernel. The time spent on the generatePoints kernel seems to linearly scale with the sample size. This makes sense since each thread sequentially does the calculation for *sample size* times.



This chart shows execution time as the number of generation threads increases. An interesting finding is that execution time stays the same from 2^10 to 2^15 threads. This likely because as threads increase, the program is increasing compute in a parallel fashion, rather than a sequential fashion. Thus, it makes good use of the GPU's resources. However, once threads increase further, it saturates the compute of the GPU and execution time explodes. It is also worth noting that accuracy of the pi estimation increases as (generation threads * sample size) increases. Thus, it seems more feasible to increase the generation threads, since this will increase the accuracy more without as big of a penalty on execution time.



The last comparison done was changing the *reduce size* parameter. However, this led to no significant changes in execution time.


Execution Time of Monte Carlo PI Estimation as Reduce Size Varies
Sample Size = 1e6, Generation Threads = 1024
🟨 cudaMemcpy
🟥 cudaMalloc
🟦 generatePoints
Execution Time (ms)
Reduce Size