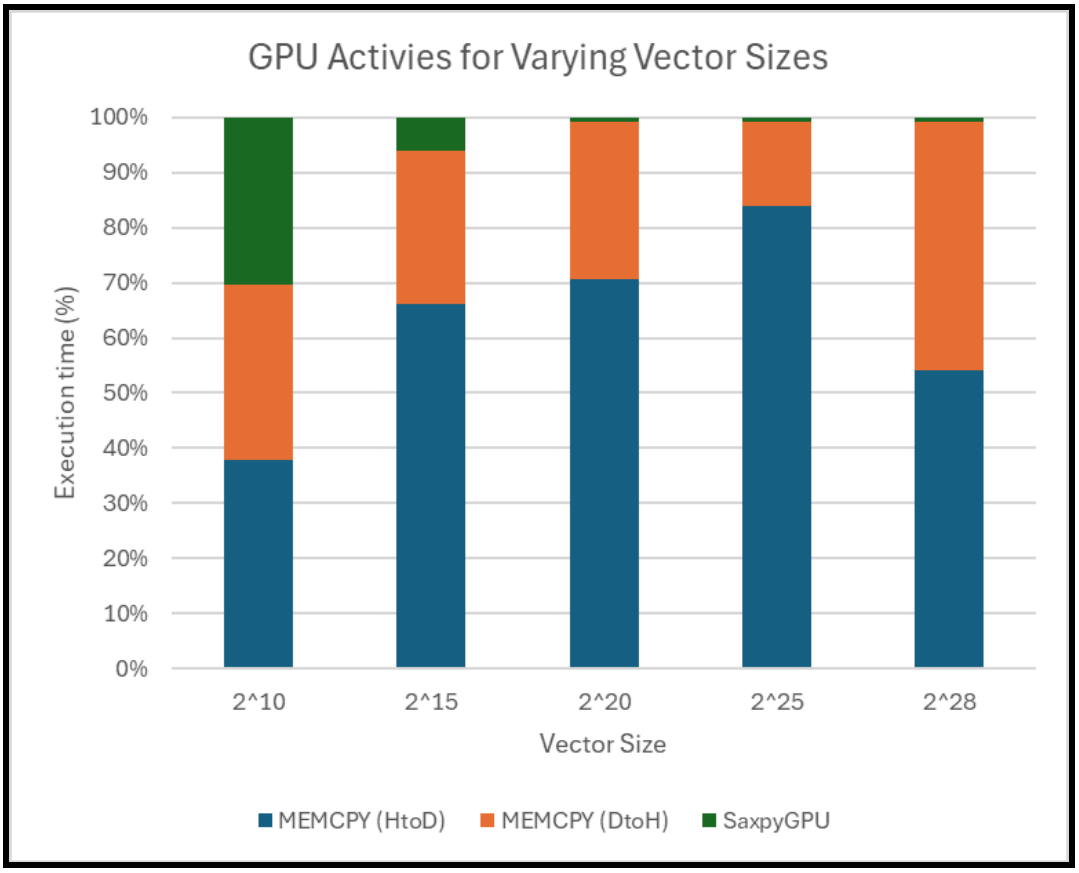
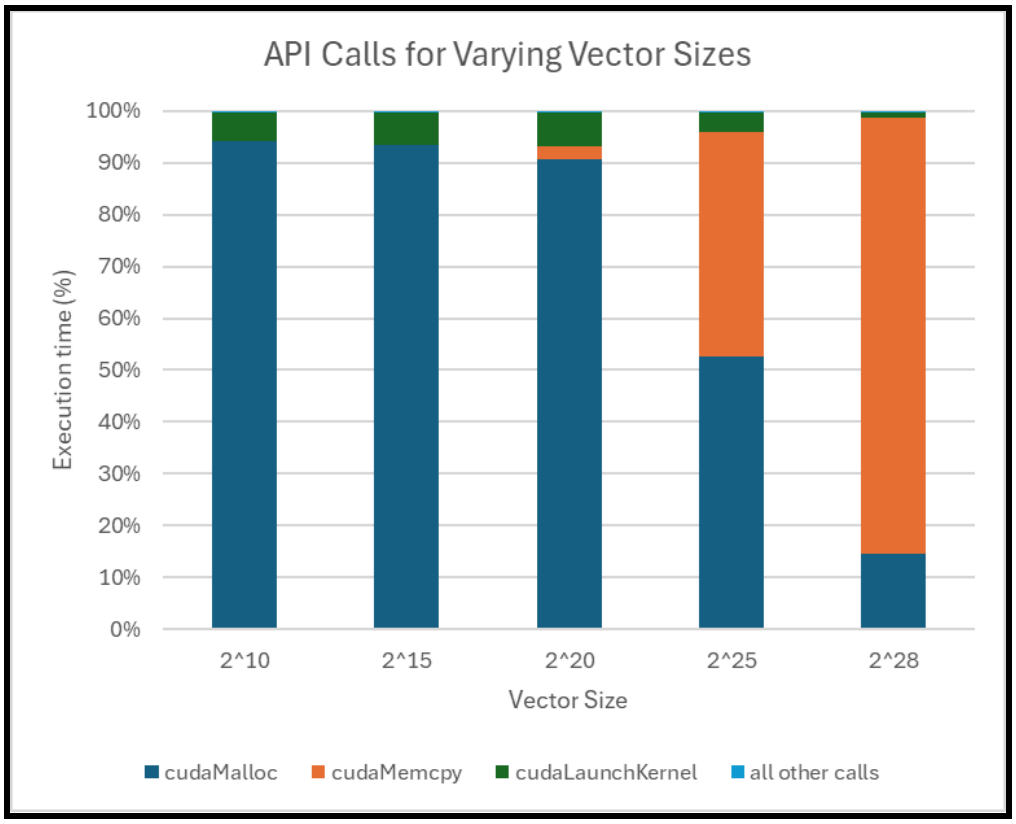
CUDA Programming Part 1

All data for the bar charts was collected using the nvprof profiler in the gpu.scholar.rcac.purdue.edu cluster. For part A, the VECTOR_SIZE variable was the parameter changed to investigate execution times. For part B, the MC_SAMPLE_SIZE variable was the parameter changed to investigate execution time

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

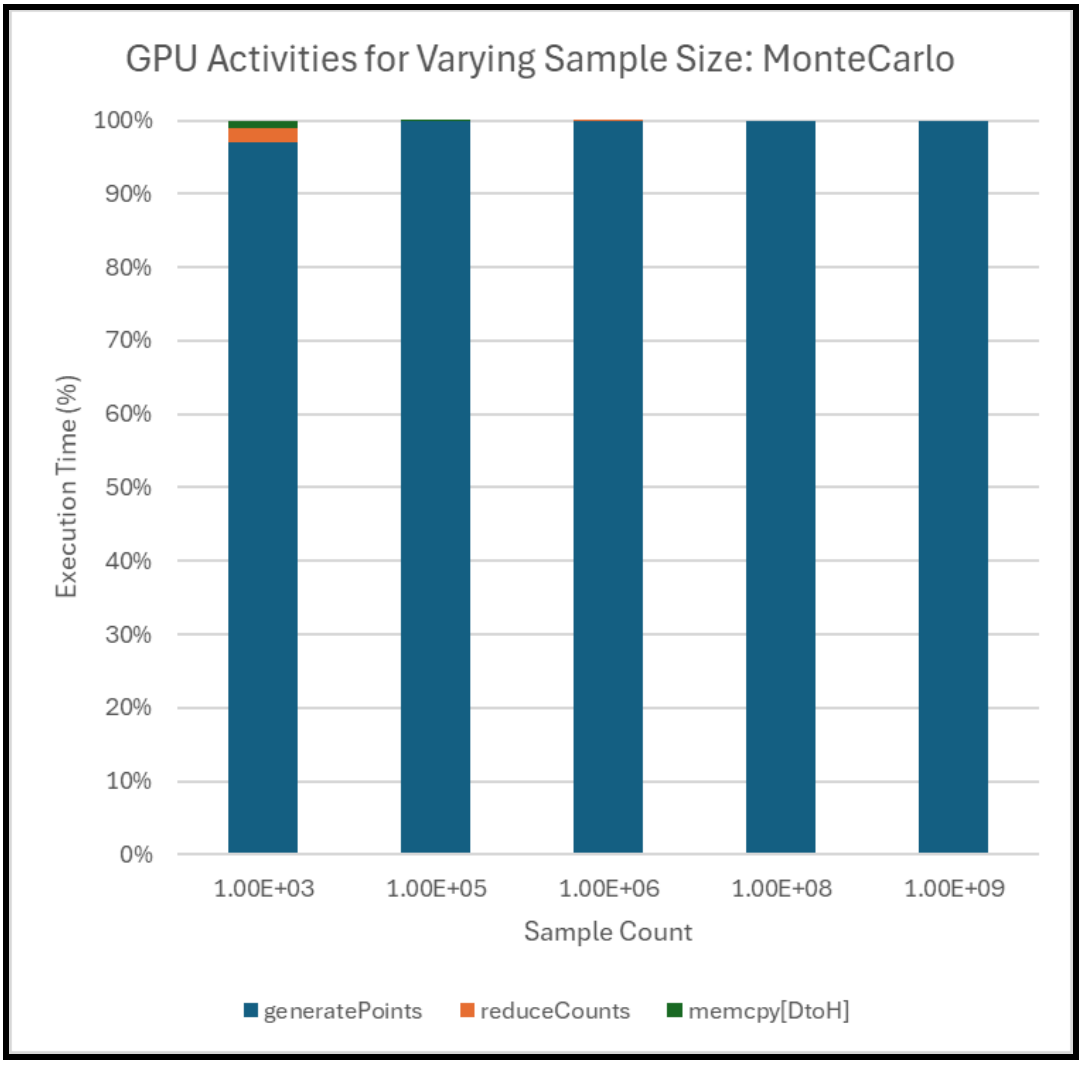


We have some interesting observations to glean from the stacked bar chart. For one, as the vector size increases, SaxpyGPU takes up a smaller and smaller portion of the overall run time. This is most likely due to the vector size being large enough that we are getting much better utilization, meaning way less cuda cores will be idle at the higher vector sizes.

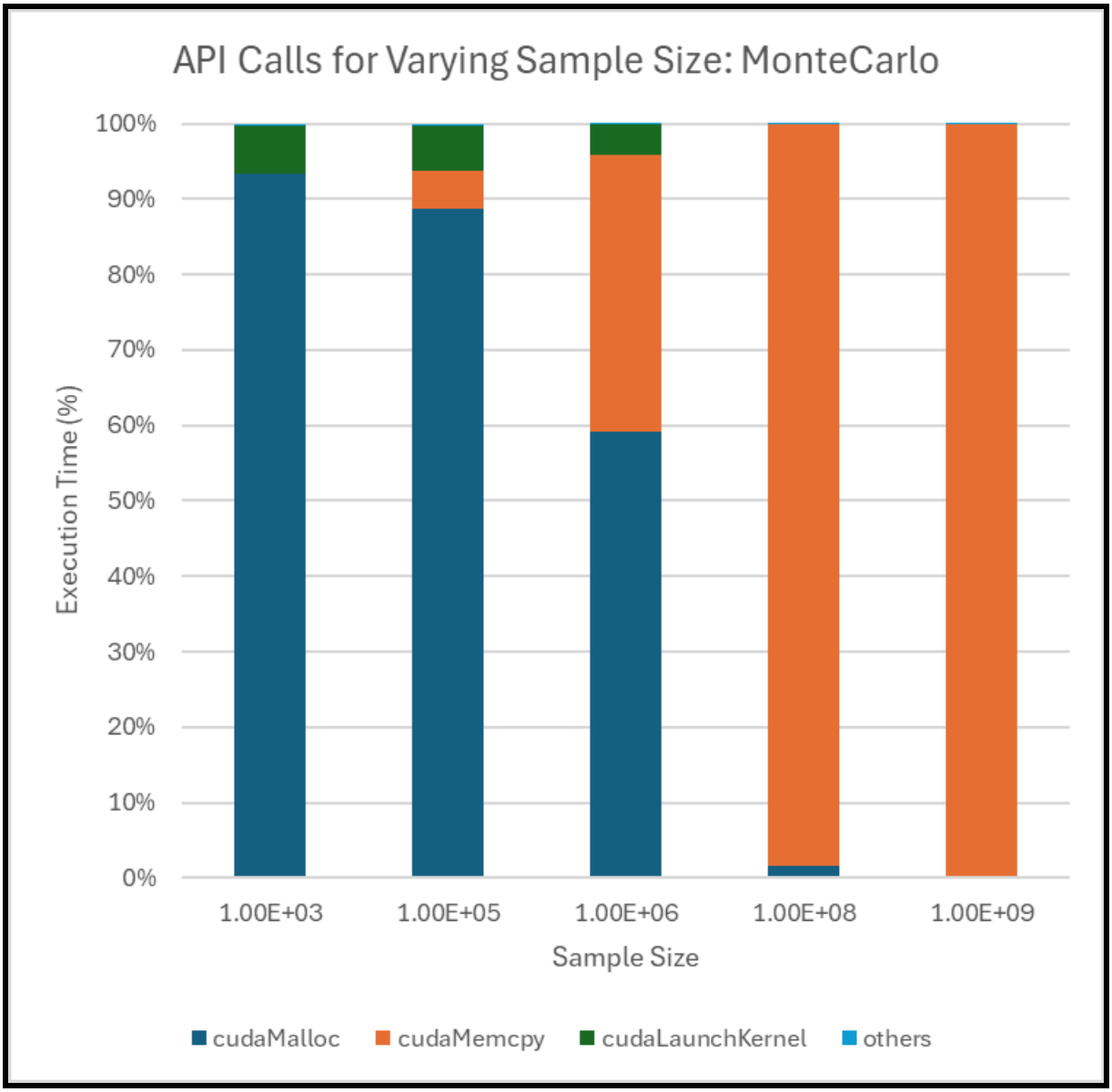


One important thing to note about the execution time for API Calls is that cudaLaunchKernel decreases in percentage as vector size increases. This is because this kernel launch is a fixed overhead, so as vector size increases and programs take longer to run, it takes up a much smaller portion of the overall execution time. We should also note that cudaMemcpy goes from an insignificant portion of execution time at 2^10 to a large portion of execution time at 2^28. This could be due to a few factors. For one, the amount of data we have scales linearly with vector size, so the larger the vector size the more data we must allocate and then transfer. Additionally, it's quite possible that CUDA kernel execution scales better than memory transfer, as cudaMemcpy is a memory bound execution. This means we now have a memory-bound operation rather than a compute-bound operation, so the memory transfer has now become the bottleneck in our system.

Part B: Monte Carlo estimation of the value of π



One obvious thing about this bar chart is that our generatePoints function takes up a huge majority of the execution time even at the lowest sample count (1e3). One main difference in how this function was written versus the previous was that cudaMemcpy was not used before the kernel was called (HtoD), as the kernel itself generated the data on the gpu using the curand_uniform function. This means that our generatePoints kernel is doing nearly all the work of the program and will scale in time as our sample count increases. On the other hand, reduce counts and memcpy[DtoH] are constant in time, so as sample count increases and execution time increases with it, these will become a smaller portion of the total execution time.



The results for the API calls for Monte Carlo have very similar observations as to what was seen in part A with the saxpy kernel. As our sample size increases, cudaMemcpy goes from a small portion of the overall execution percentage to being the largest portion. The amount of data used scales with the sample size, as as we have to copy more and more data, we will no longer be compute bound but rather memory bound, leading to a memory operation such as cudaMemcpy taking up more and more of the execution time as the program takes longer to run and has a significantly higher amount of data.