Garrett Lail

ECE 60827 Lab 1 Report

Analysis of CUDA Performance for SAXPY and Monte Carlo Estimation

The first program executed on the GPU, SAXPY, is an embarrassingly parallel workload. Vectors are streamed in, used once, then written back. Due to the lack of reuse, this program is expected to be heavily memory-bound – and the speed at which vectors can be streamed into GPU memory, as well as between the CPU and GPU, would be the primary limitation. Below are results of the execution of a simple SAXPY kernel on vectors of six lengths.

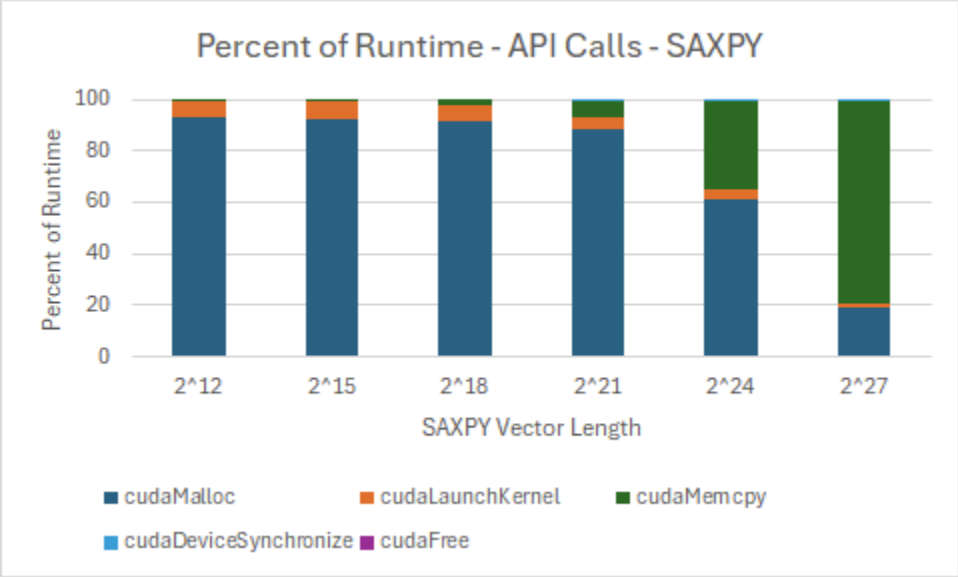

Figure 1: SAXPY kernel execution on vectors of six sizes. Plots the relative runtime of CUDA API calls. The amount of memory streamed between the CPU and GPU ranges from 32 KB to 1 GB of total data across the minimum and maximum vector lengths.

The vectors used in this experiment massively differ in their sizes, rapidly increasing the amount of memory which needs to be transferred between the CPU and GPU. As can be seen, the CudaMalloc() operation results in the most overhead in the program while vector length is small. However, as the size of the vector grows, the CPU-GPU communication becomes the limiting factor.

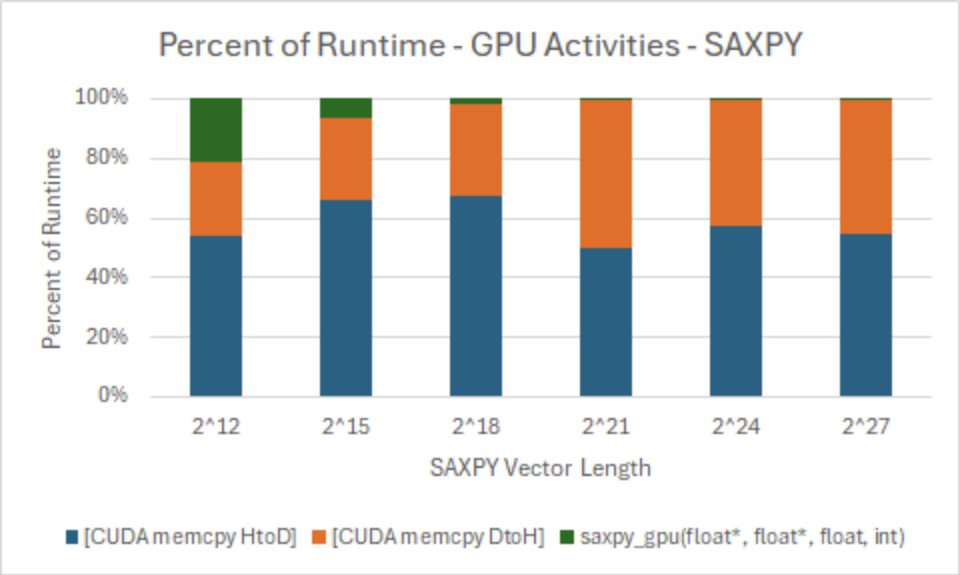

Figure 2: SAXPY kernel execution on vectors of six sizes. Plots the relative runtime of GPU Activities such as kernel execution and memory transfer.

The GPU activities plot offers more insight into how memory-bound SAXPY is. Even on small vectors, it is apparent that the transfer of memory from CPU to GPU and vice-versa is very expensive. The runtime of computation only averages around 20% for the smallest vector size of around 32KB. Additionally, as the size of the vector increases, the runtime of computation becomes negligible.

Transitioning to look at the Monte Carlo estimation of PI, the program is very different to SAXPY. Whereas the SAXPY kernel requires the CPU to transfer large vectors between itself and the GPU, the Monte Carlo kernel generates its values on the GPU. It also uses a reduction kernel to sum the partial results of the primary point-sampling kernel, which further reduces the requirement for memory copying between the device and host.

Figure 3 clearly shows how compute-dominated the Monte Carlo estimation is. Each thread calculates between 20000 to 2 billion pseudo-random numbers as well as computes the euclidean distance in 2-dimensions for the sampled floating-point values.

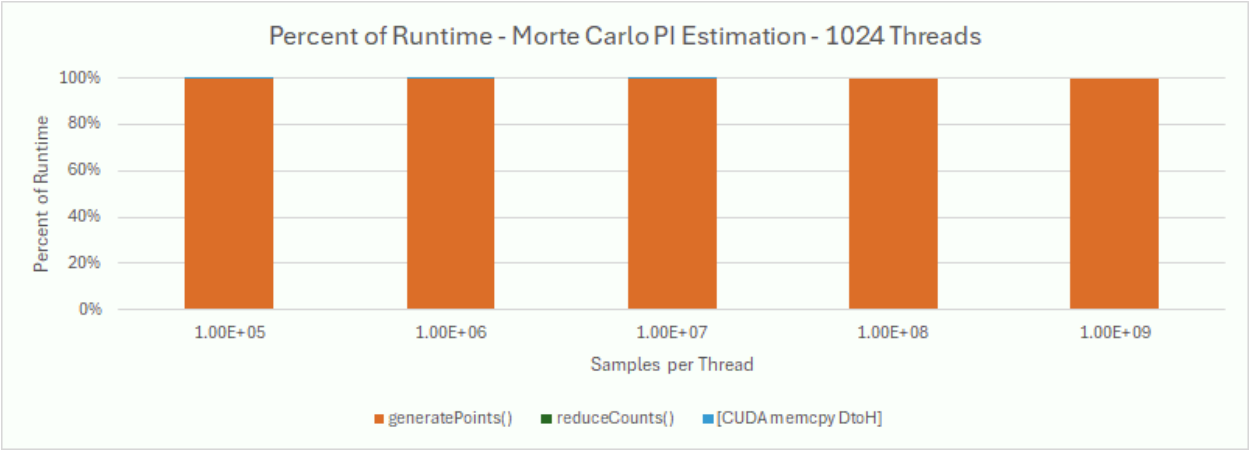

Figure 3: Runtime of GPU activities during Monte Carlo estimation of PI.

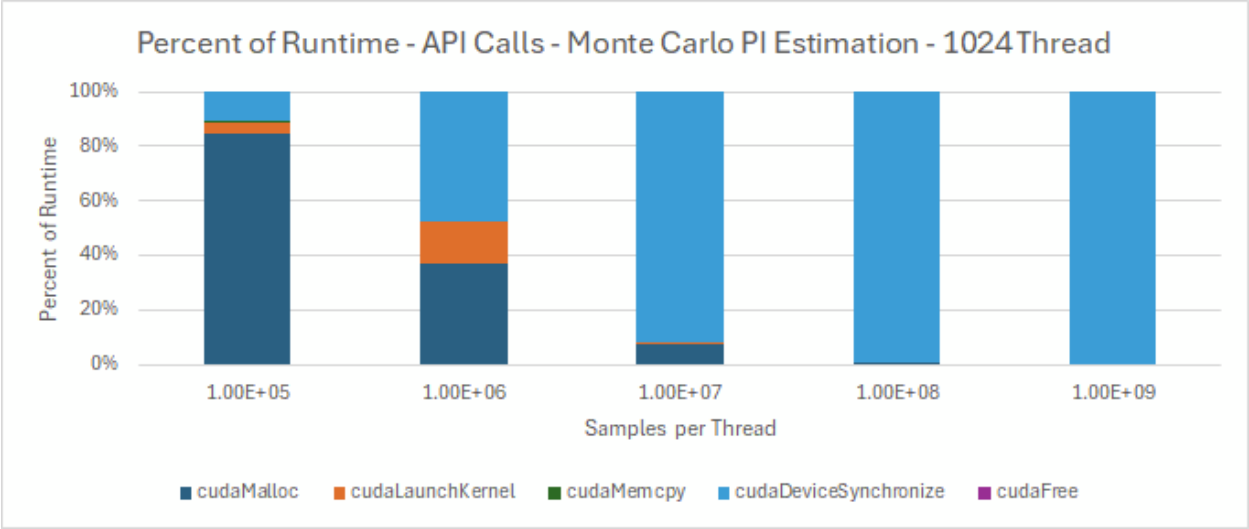

Figure 4: Runtime of CUDA API calls during Monte Carlo estimation of PI.

The program, when used with smaller sampling rates which reduce the compute workload of each thread, has significant overhead when allocating memory on the GPU. The kernel launch overhead is appreciable as well. However, as the workload size increases, these overheads are increasingly amortized and compute time (as seen by cudaDeviceSynchronize) dominates.

In figure 5, the number of samples calculated across all threads is held constant, but the number of threads used varies. 1024 * 10^7 total samples are calculated.

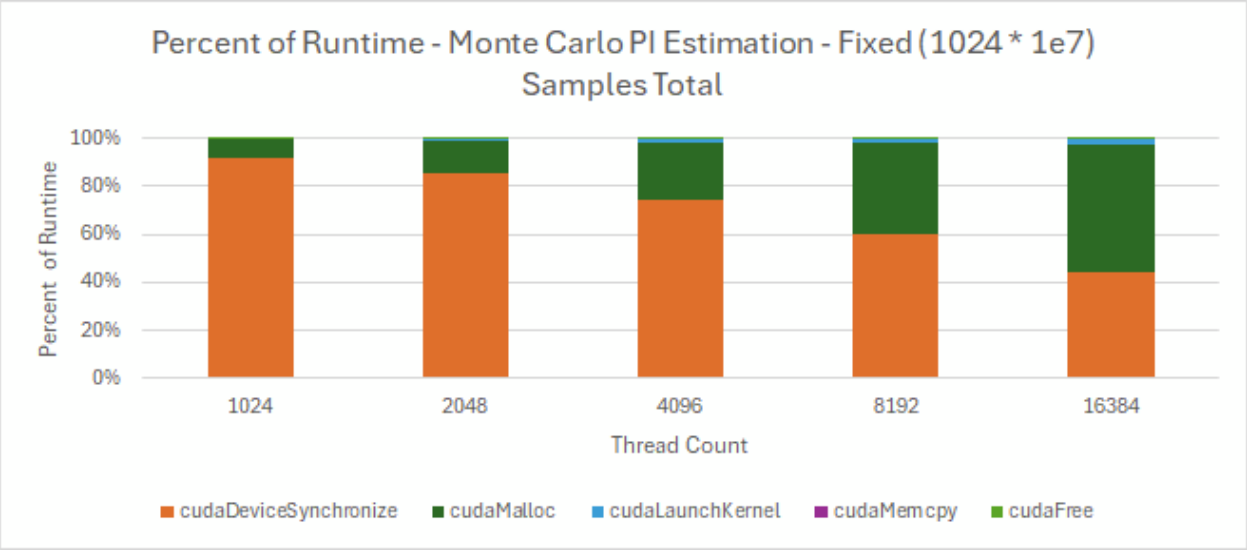

Figure 5: Runtime of CUDA API calls during Monte Carlo estimation of PI.

As expected from prior results, as the amount of compute power increases (with more threads), the memory allocation overhead becomes more apparent. The problem size is fixed throughout this experiment, so adding more threads results in overall faster computation but far less efficient computation. This harkens back to Amdahl's Law, which defines how the overall speedup of the program is bounded by what portion of the program is actually parallelizable. Memory allocation is not, and therefore represents a fundamental limitation on the speed of kernel execution.