Jake Nagel
ECE60827 CUDA Assignment 1

Part A : SAXPY GPU

The traditional SAXPY Loop was implemented as a parallel workload to be performed on a GPU. A CUDA Program was authored using a kernel to perform the computation provided a random scale and a pointer to the two vectors “x” and “y.” The “x” and “y” vectors are first initialized by the host and populated with random values. The “vectorInit()” function provided by the CPU Library was used to do this. After running “saxpy_gpu(),” the “verifyVector()” function provided by the CPU Library was used to confirm correct execution.

By using provided API calls from the CUDA Library, source code was authored to allocate host and device memory, copy data to and from the host to the device and de-allocate memory from the device. For this experiment, various vector lengths were tested and the NVIDIA Profiler tool was used to gather the results.

Results and Observations

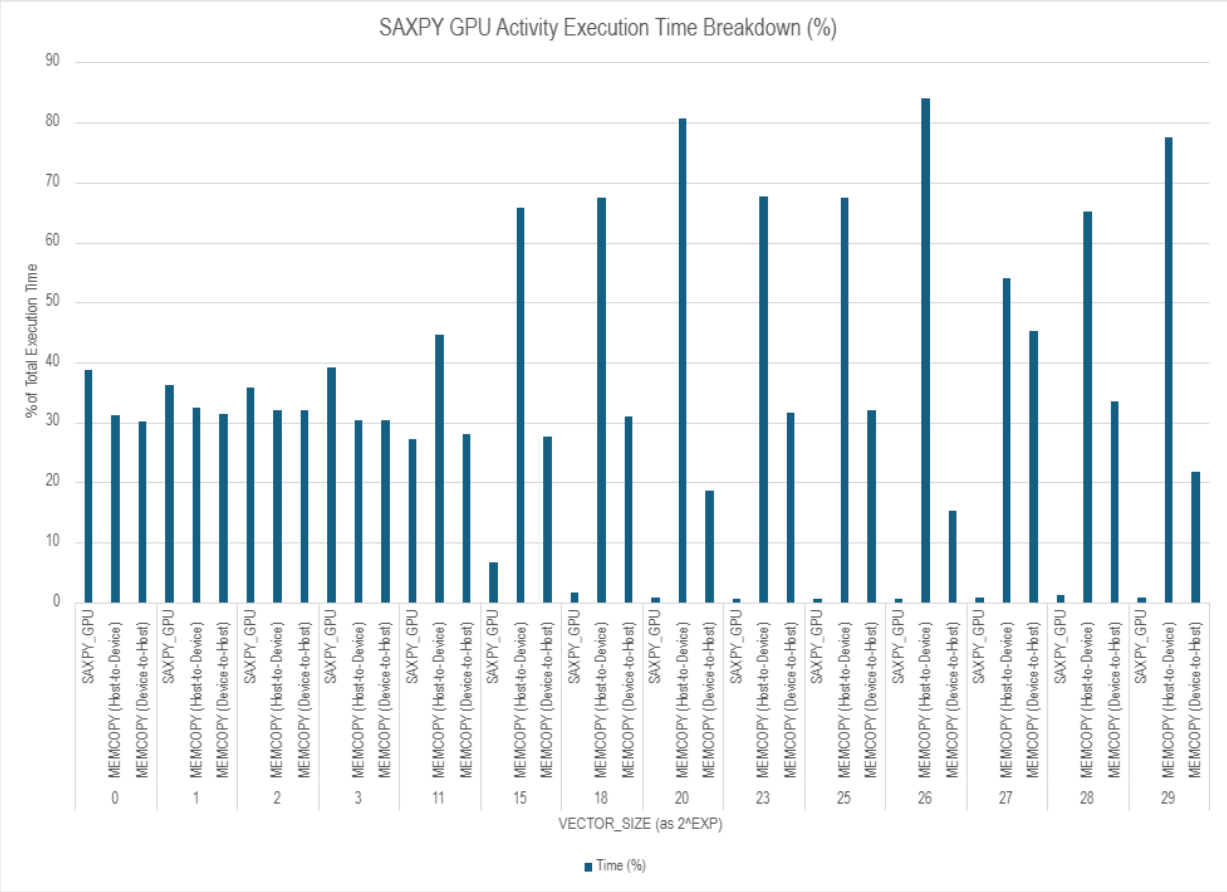

Figure 1: SAXPY GPU Activity Execution Time Breakdown

Jake Nagel
ECE60827 CUDA Assignment 1

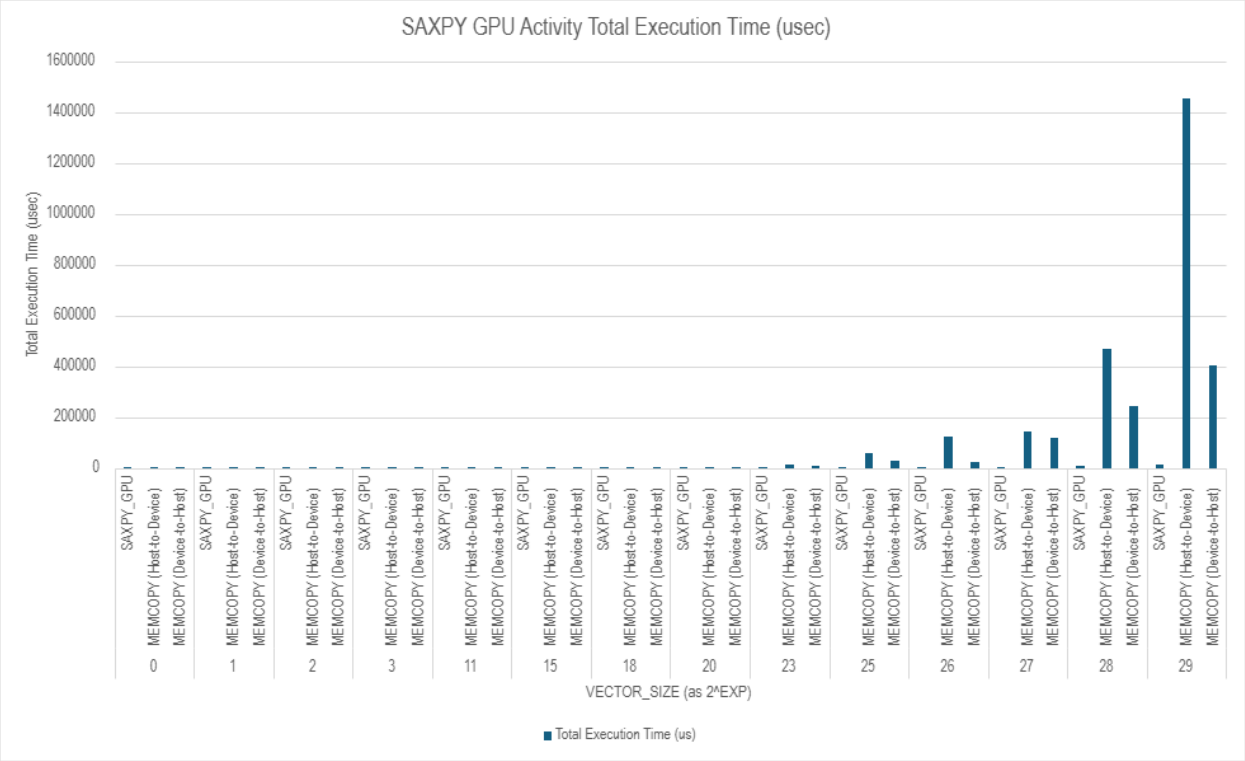

Figure 2 : SAXPY GPU Activity Total Execution Time (usec)

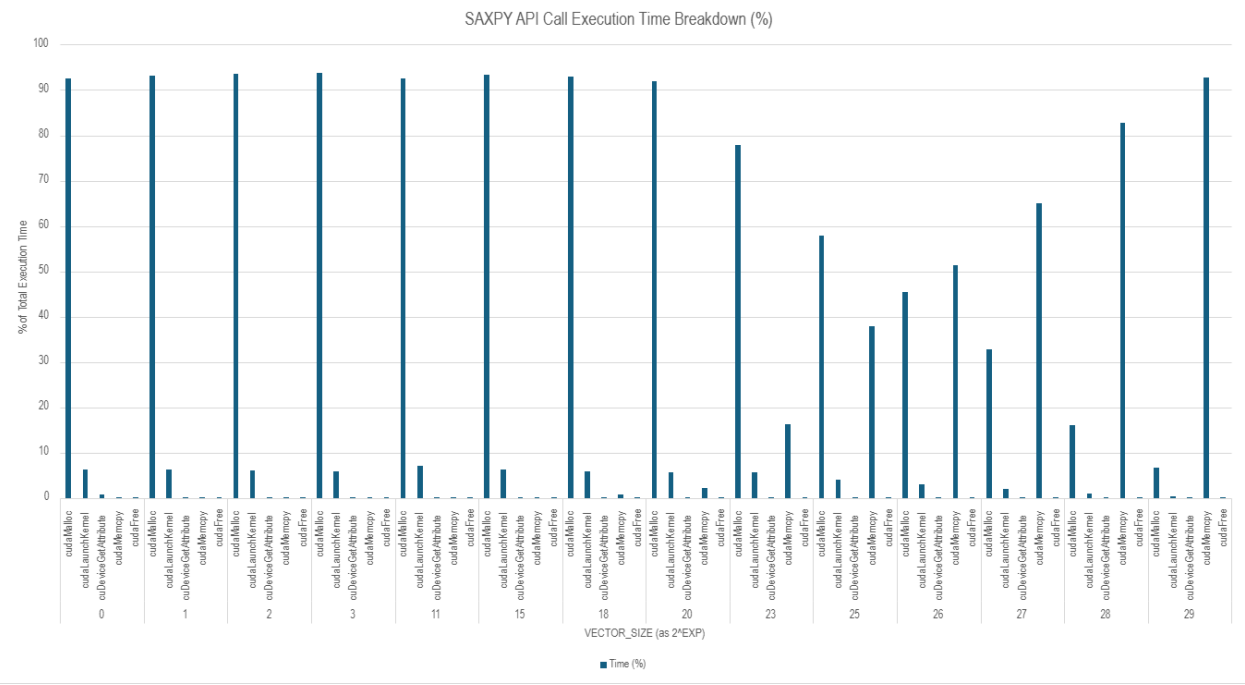

Figure 3 : SAXPY API Call Execution Time Breakdown (%)

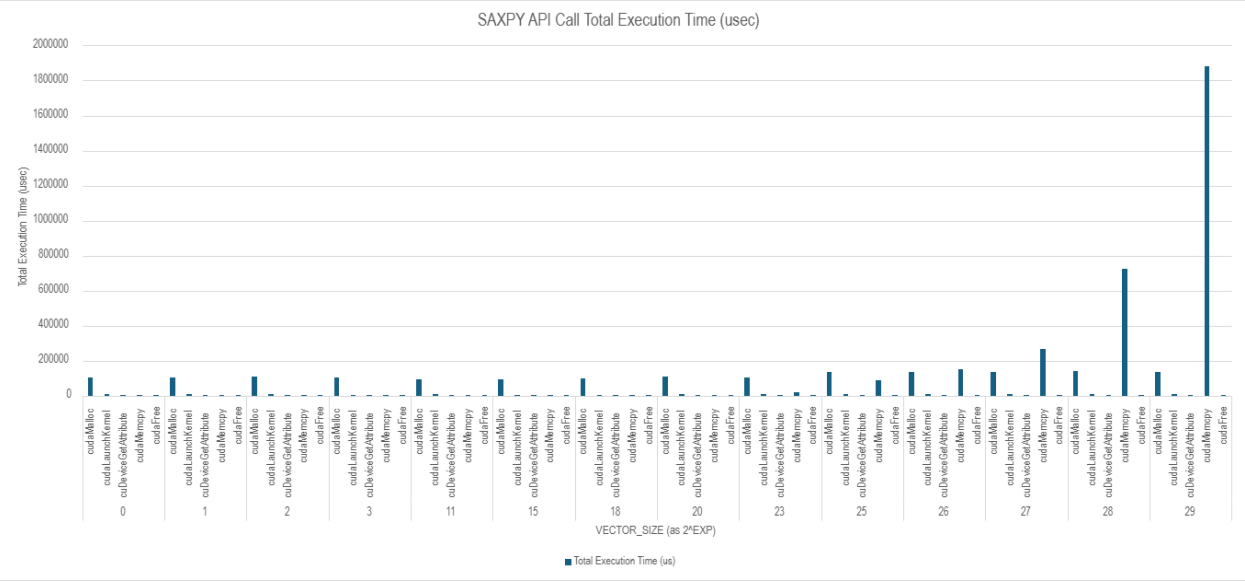

Figure 4 : SAXPY API Call Total Execution Time (usec)

Shown in Figure 1 - Figure 4, the vector size of “x” and “y” was varied using powers of 2 to modify “VECTOR_SIZE” defined in “lab1.cuh.” The exponent is shown on the bottom of the horizontal axis (Note – 29 is really 2^29 – 1 as this is the maximum size warranted).

A fixed thread block size of 256 was used for this experiment. It was not surprising to observe that as the data set is exponentially larger, the amount of execution time for the GPU spent copying data from the host became exponentially larger.

Part B : Monte Carlo Estimation of Pi

To parallelize the workload to compute Pi via the Monte Carlo method, the “generatePoints” kernel was authored to randomize the coordinate arrays “x” and “y” using the cudaRand() library, then iterate over the sample_size to compute the partial sum in each thread’s corresponding “pSums” array. An attempt was made to utilize the “reduceCounts” kernel to use parallel reduction of the partial sum result from generatePoints; however, further understanding on how to use shared memory in the kernel is underway. After performing this exercise without it, it is apparent that having this feature is a necessity for large data sets (> 1 billion samples).

Jake Nagel
ECE60827 CUDA Assignment 1

Results and Observations

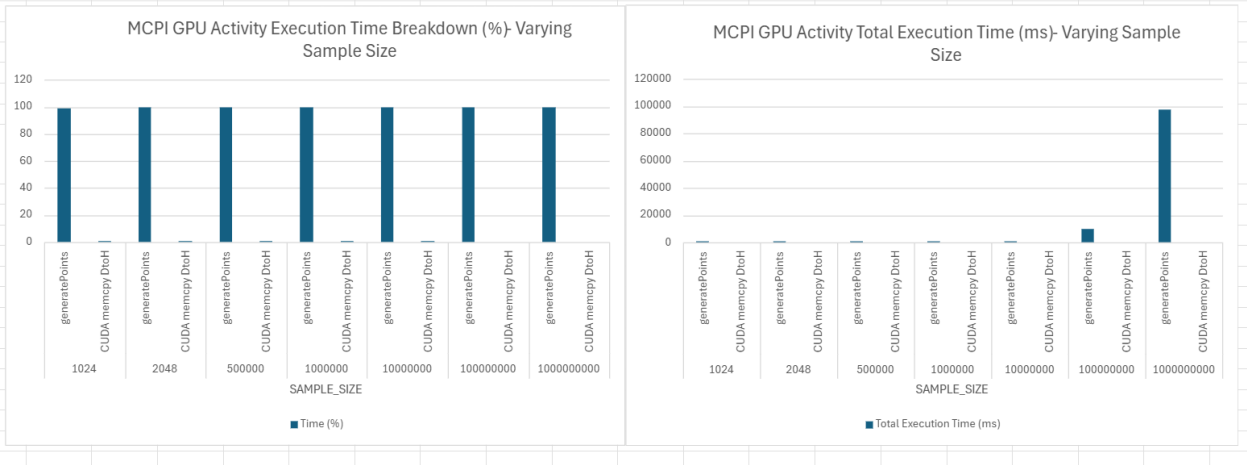

Figure 5 : Monte Carlo Pi Estimation - GPU Activity - Varying Sample Size

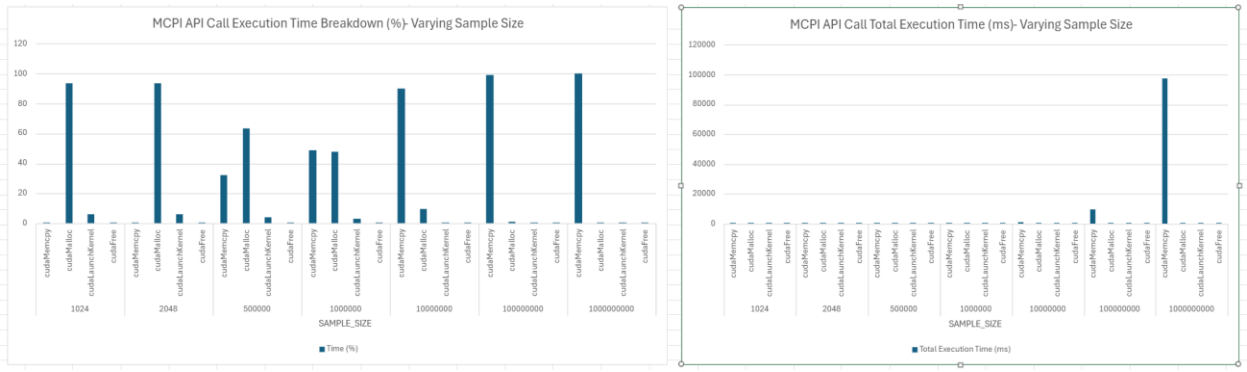

Figure 6 : Monte Carlo Pi Estimation - API Calls - Varying Sample Size

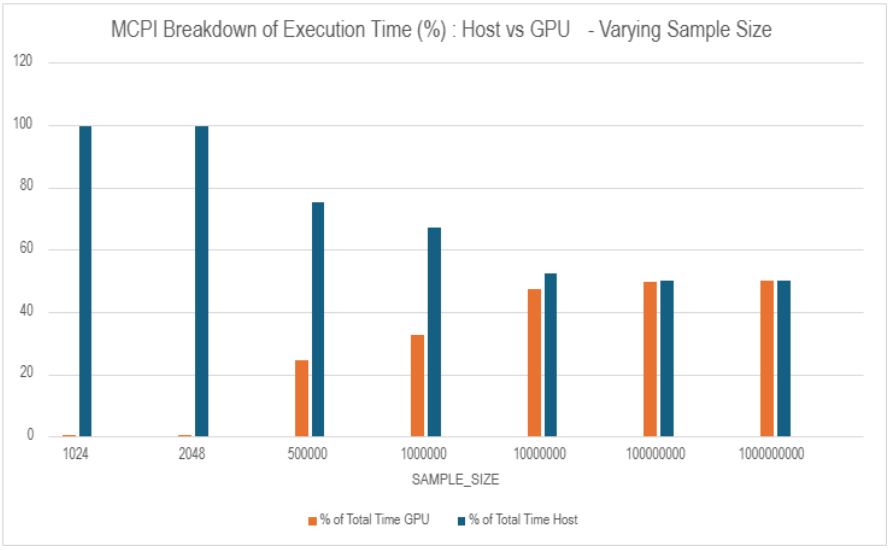

Figure 7 : Breakdown of Total Execution Program Time Between Host and GPU with Increasing sample_size

Figure 5 - Figure 7 show test results using nvprof for fixing the number of threads "GENERATE_BLOCKS" defined in "lab1.cuh" at 1024 and varying the sample size processed by each thread. The total execution time was also added in Figure 7 to show how, without reduceCounts the GPU and CPU workload even out. reduceCounts would certainly optimize this. Sample Size also affects the % error in the PI calculation. The error was found to be 0.07% at 1024 samples, which is **910 times higher than** the result of 0.000084% when using 1,000,000 samples.

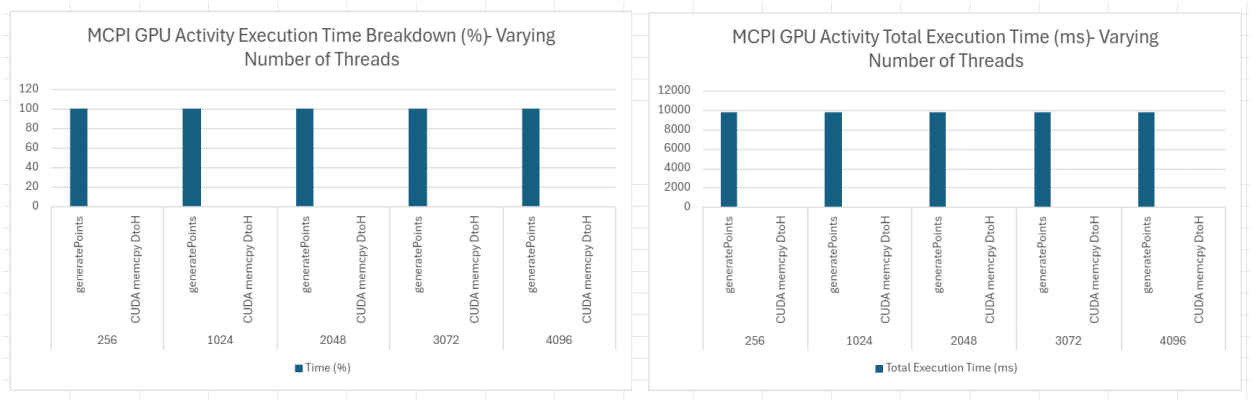

Figure 8 : Monte Carlo Pi Estimation - GPU Activity - Varying Number of Threads

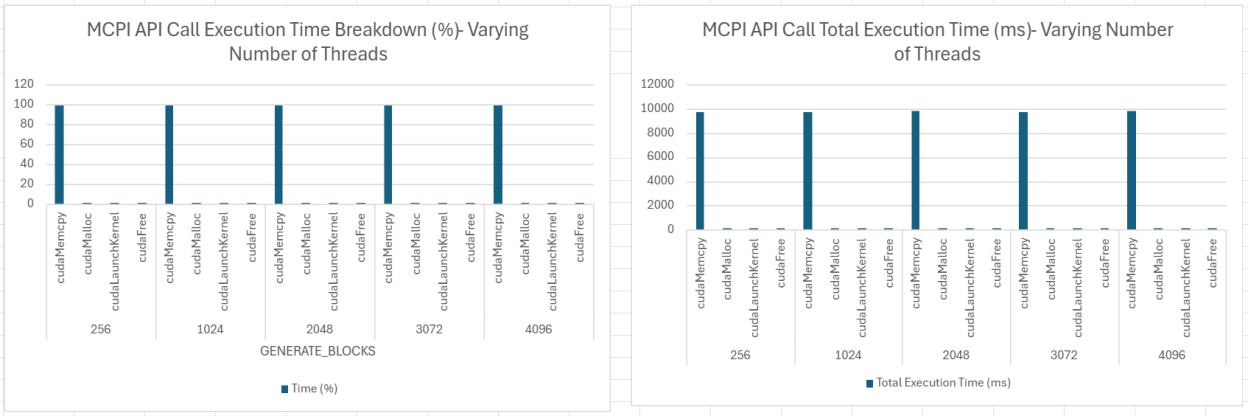

Figure 9 : Monte Carlo Pi Estimation - API Calls – Varying Number of Threads

A test was performed to vary the number of threads which process SAMPLE_SIZE number of points. From looking at the API Calls, it is apparent that the bottleneck is in copying data between the GPU and the Host – the cudaMemcpy is the largest . This is another reason using a parallel reduction within the GPU is necessary, especially in scenarios where memory latency is a critical path.