ECE 60827 CUDA Assignment 1 Report

Spring 2025
By: Bhavya Patel

Introduction

This goal for this homework is to explore the embarrassing parallelization nature of two tasks using CUDA programming. The SAXPY operation and the Monte Carlo estimation of π. I used GPU acceleration to optimize performance over traditional CPU implementations. SAXPY involves element-wise scaling and addition of vectors, making it susceptible for parallelization. Similarly, the Monte Carlo method estimates π by randomly generating points and determining their position relative to a unit circle. By implementing these algorithms demonstrates the efficiency gains achieved through parallel computing while providing hands-on experience with CUDA.

Results

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

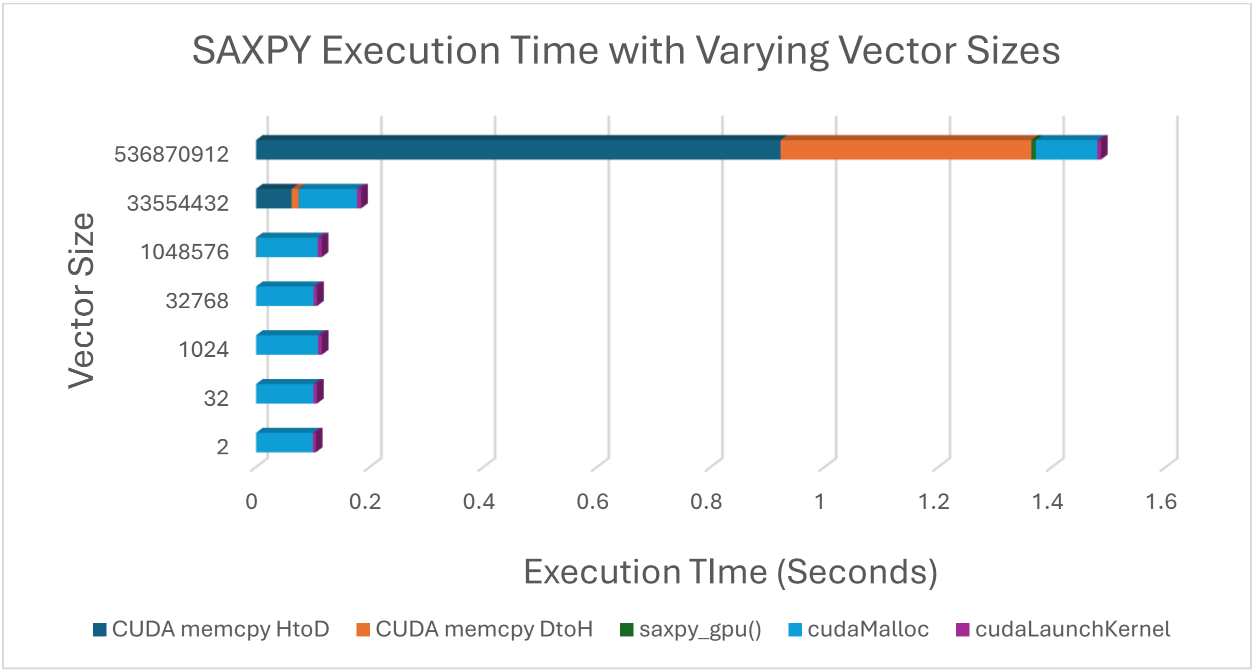

Figure 1: SAXPY with Varying Vector Sizes Results

The graph above shows the execution time of the SAXPY program when utilizing GPU acceleration. It provides a breakdown of the major components of execution in comparison to the overall runtime while also showing how increasing the vector size affects performance. For this experiment, I used 512 threads per thread block and calculated the number of thread blocks as the ceiling of vectorSize divided by threads per thread block. As expected, increasing the vector size leads to an increase in execution time. However, the more insightful observations come from analyzing the specific components contributing to execution time. The data shows that as the vector size increases, the time spent transferring memory between the host and device grows significantly. For smaller vector sizes, memory transfer time is negligible compared to cudaMalloc execution time. Additionally, it is important to note that the execution times for cudaMalloc and cudaLaunchKernel remain constant regardless of vector size. The primary factor that changes is the memory transfer time between the host and device. Finally, examining the execution time of the saxpy_gpu kernel itself reveals that, although it does increase with vector size, it remains a negligible portion of the overall execution time.

Now I will also look at the difference in execution time when looking at a different amount of threads in a threadblock. For this experiment I will use 128 and 512 threads per threadblock and compare it to the execution times of varying vector sizes.

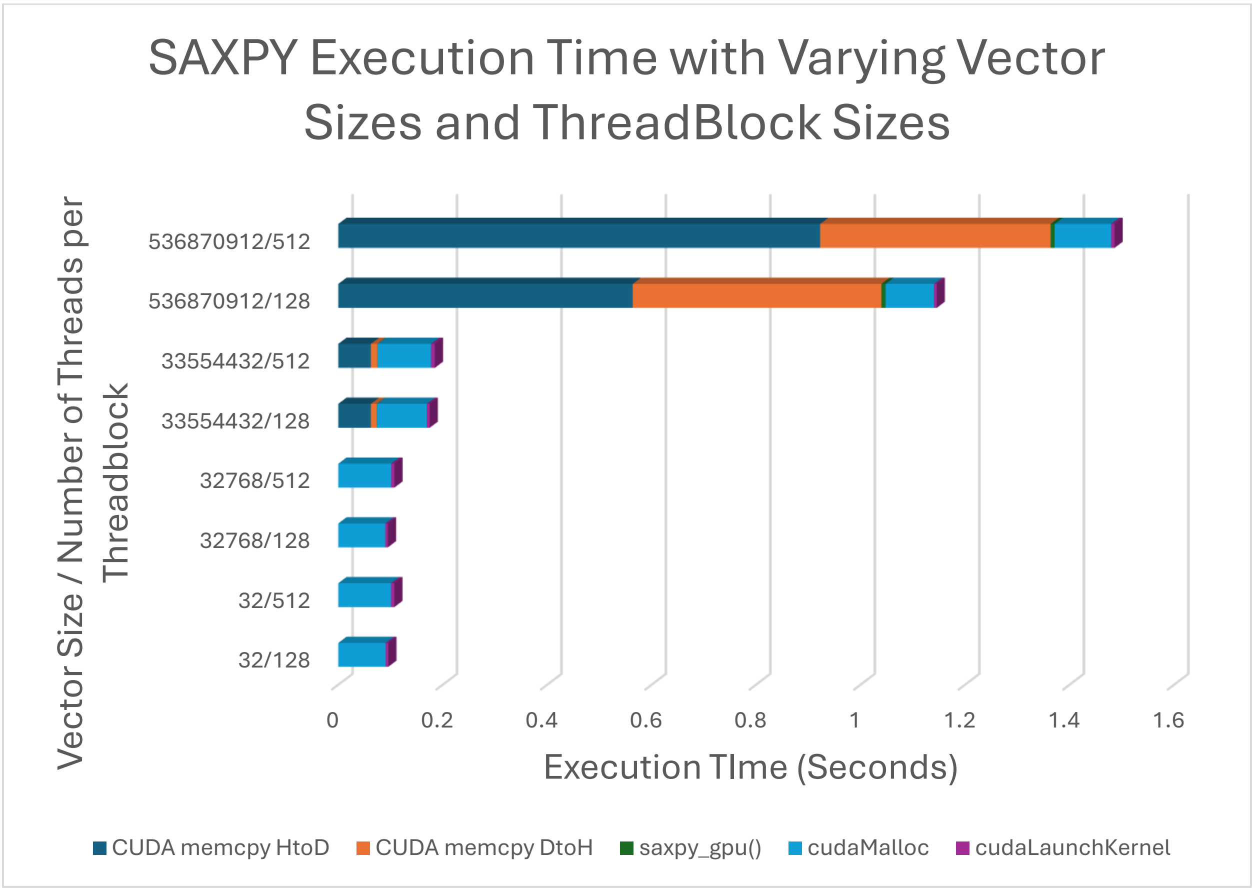

Figure 2: SAXPY with Varying Vector and Threadblock Sizes Results

In the above figure we can see that the general trend is that less threads per threadblock achieves better performance with SAXPY. Now the easiest vector size to see this difference is for 536870912 vector size. We can see that 512 threads spend much more time in the cudamemcpy than 128 threads. This is the reason why 128 threads per threadblock had a lower execution time than 512 threads per threadblock.

PART B: Monte Carlo estimation of the value of π

For this task there are no specific instructions on what results to present, so I have chosen to explore the impact of varying three key parameters, generateThreadCount, reduceSize, and sampleSize. I decided not to modify the number of threads per thread block because this was already explored in Part A. For my experiments I established a base configuration with 256 threads per thread block, a sampleSize of 1e6, a generateThreadCount of 1024, and a reduceSize of 32.

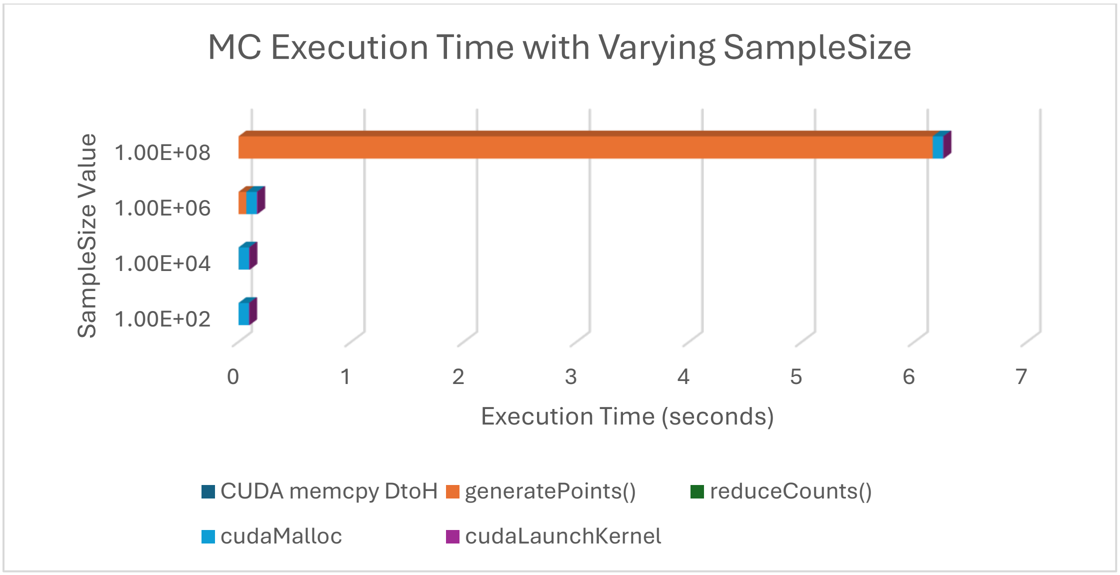

Figure 3: Monte Carlo with Varying SampleSize Values Execution Results

This figure illustrates how varying the SampleSize changes the execution time. We can see a noticeable increase in execution time with larger SampleSize values, which makes sense because each thread spends more time generating points in the generatePoints() kernel. Interestingly, the differences in execution time across the different SampleSize values are relatively small, showing that the kernel's execution time is mostly influenced by the SampleSize, with little variation beyond that.

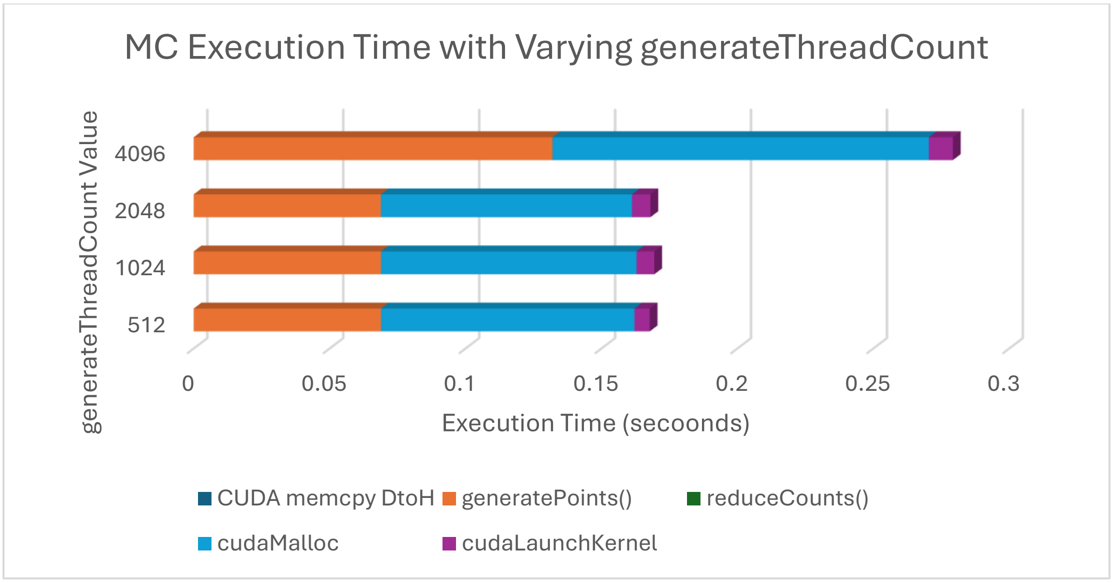

Figure 4: Monte Carlo with Varying generateThreadCount Values Execution Results

In the figure we are exploring the uses of various generateThreadCount values. You would think if you linearly increased the number of threads you are generating then the execution time should increase in the same manner. However, we can see that the execution time stagnate until you reach the value of 4096. This may be due to the fact that the number of threadblocks all are allocated to a SM without overlap which results in the same execution time. Then when the values was increased to 4096 then some SMs had two threadblocks to execute which resulted in the generatePoints() kernel to take twice as long to execute than previously.

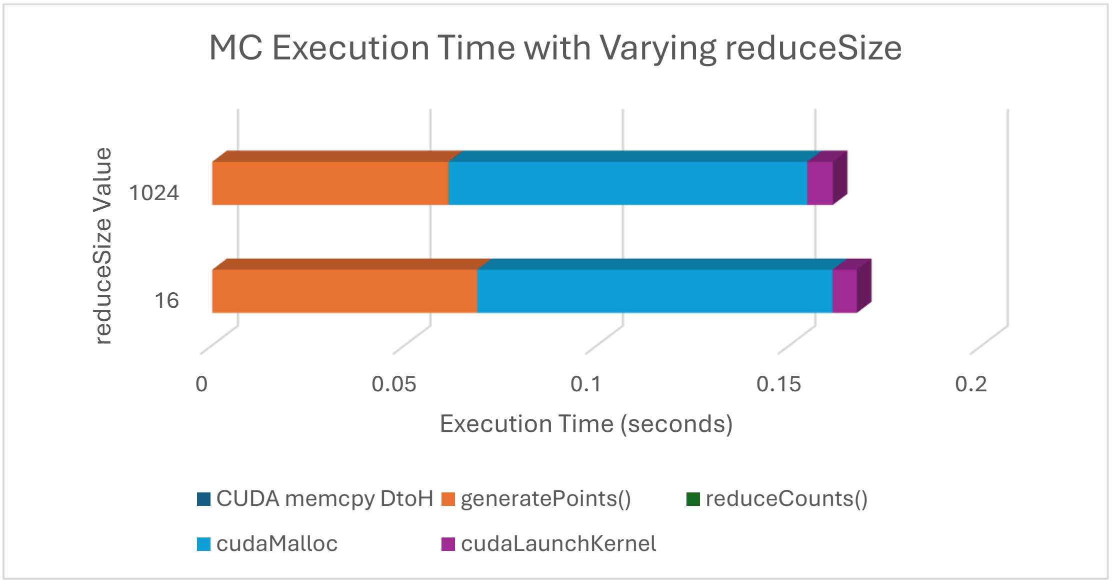

Figure 4: Monte Carlo with Varying reduceSize Values Execution Results

When evaluating the above figure it seems like increasing the reduceSize value slightly improved the performance, but I would disagree. The only function that improved when changing reduceSize was generatePoints(), which does not make sense because reduceSize has no affect on generatePoints. So am concluding that changing reduceSize does not change the execution time.