ECE 60827 Programming Assignment 1 - Marty Martin - 2/1/2025

Introduction

In this assignment, I explored two basic parallel workloads using CUDA. The two algorithms are a single precision Ax plus Y (SAXPY) kernel and an implementation of the Monte Carlo algorithm, used to estimate the value of pi. In addition to implementing the two using CUDA C++, I explore some performance characteristics of each algorithm.

Part 1: SAXPY

The SAXPY kernel for this assignment performs a simple vector accumulation, meaning the result is stored back into the same memory space as the bias value, y. To test SAXPY, a vector of random values, in this case, a randomly initialized vector of integers [0, 100) of user defined size were generated. The scale value used for computation was also to be randomly generated. To characterize performance statistics, all simulations used a constant scale factor of 1.7. This value was arbitrarily selected. Furthermore, all simulations used a thread block size of 256. Looking at the profiles simulation for different vector sizes, it seemed that the cudaMalloc function was not affected by the size of the vector allocated so I removed it from the chart to better characterize trends despite taking up a meaningful portion of overall runtime. As you can see in figure 2, as the size of the input vector increased, the time spent by cudaMemcpy dramatically increased on both the host and device runtime. The overall duration of each cudaMemcpy proportionally scaled for my test vector size of [2^20 - 2^25].

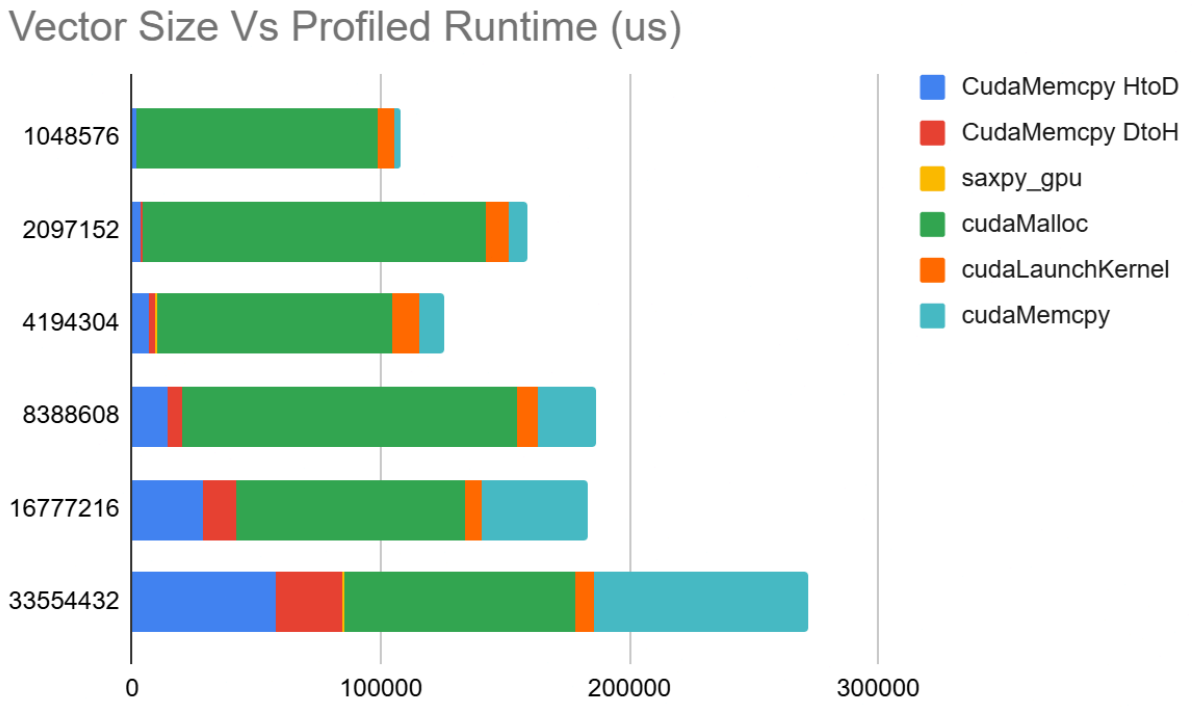

Figure 1: Profiled SAXPY GPU. GPU side cudaMalloc() seemed to not be affected by the vector size.

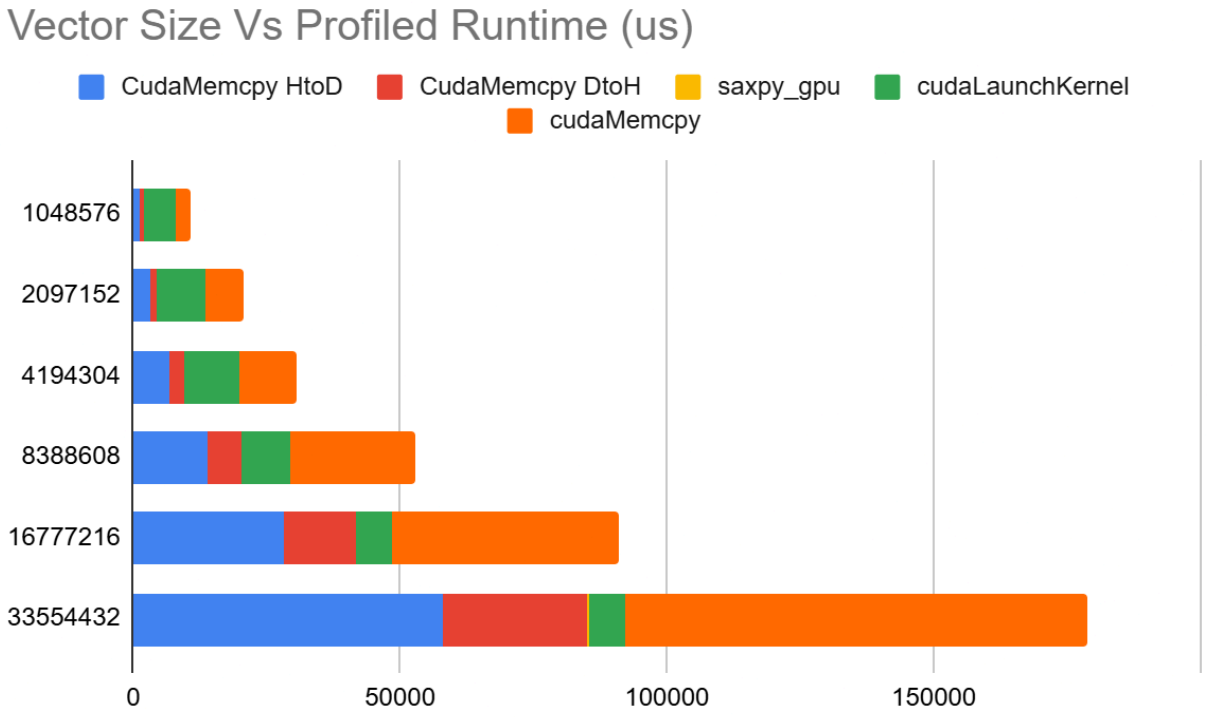

Figure 2: Profiled SAXPY GPU without cudaMalloc

Part 2: Monte Carlo

The Monte Carlo method is a technique used in many computational algorithms. It involves randomly sampling a curve to determine probability. In this case, I use it to estimate the value of pi. To achieve this I sample two points from a uniform distribution [0,1] to act as a Cartesian x and y coordinate. I sum the number of coordinates that lie within the unit circle (distance <= 1 from origin) and compare it to the total number of points generated. Since I only generate points in the first quadrant of the coordinate plane, the algorithm only predicts one quarter of the value of pi, so my final value is multiplied by 4 to account for this. To make the number generation more parallel, each thread performed a user-defined number of random point generations and each partial result was summed after all threads completed.

I chose to take on the optional portion of this assignment which involves reducing the size of the output memory transfer by pooling together portions of the partial sum array into more coarse chunks. I accomplished this by having each ‘reduce’ thread sum together a certain section of the entire partial sum array based on the threadID and function arguments which determine reduction characteristics. To characterize the performance statistics, I tested the number of sample points in the range [2^20 - 2^25]. I also tested having each thread responsible for 32 or 64 partial sums when reducing. Across all the vector sizes tested, I discovered that a large majority (90%+) of all runtime was spent on the generate points kernel code. This action in the GPU is complemented by the CPU waiting in the cudaDeviceSynchronize instruction waiting

to reduce after all data is generated. Similar to SAXPY, the total runtime seemed to scale proportionally with the input vector size.

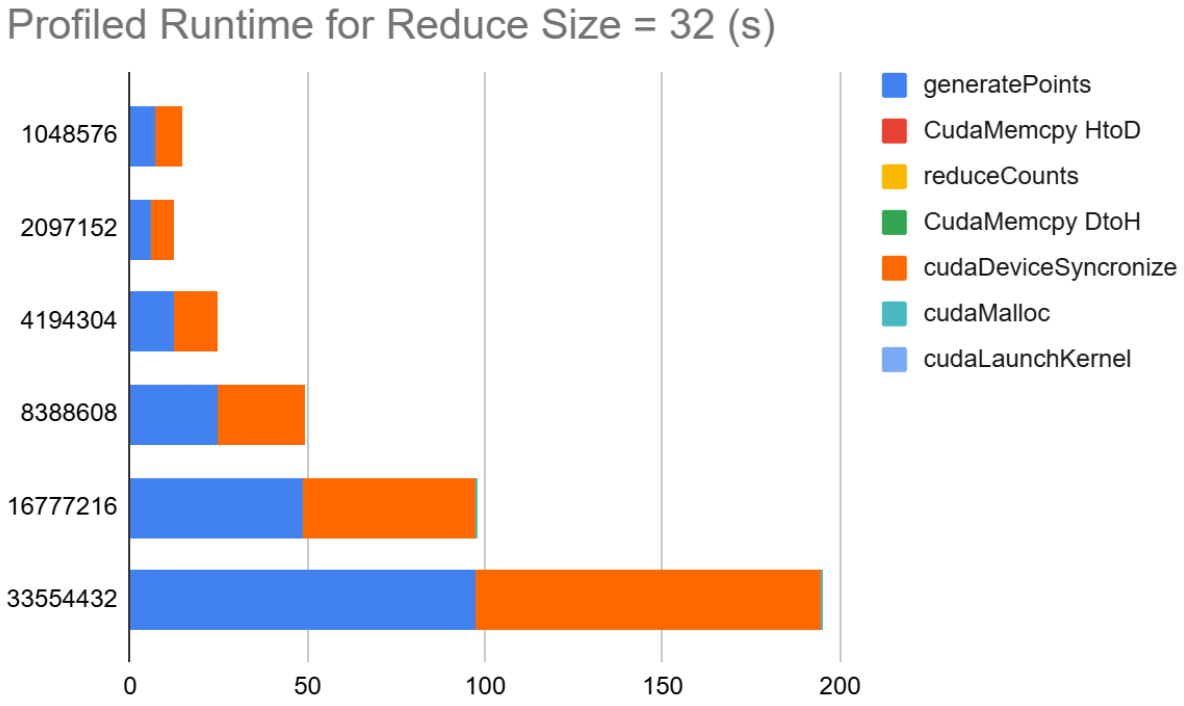

Figure 3: Profiled Monte Carlo GPU with reduceSize of 32. A majority of runtime is taken by the generatePoints kernel

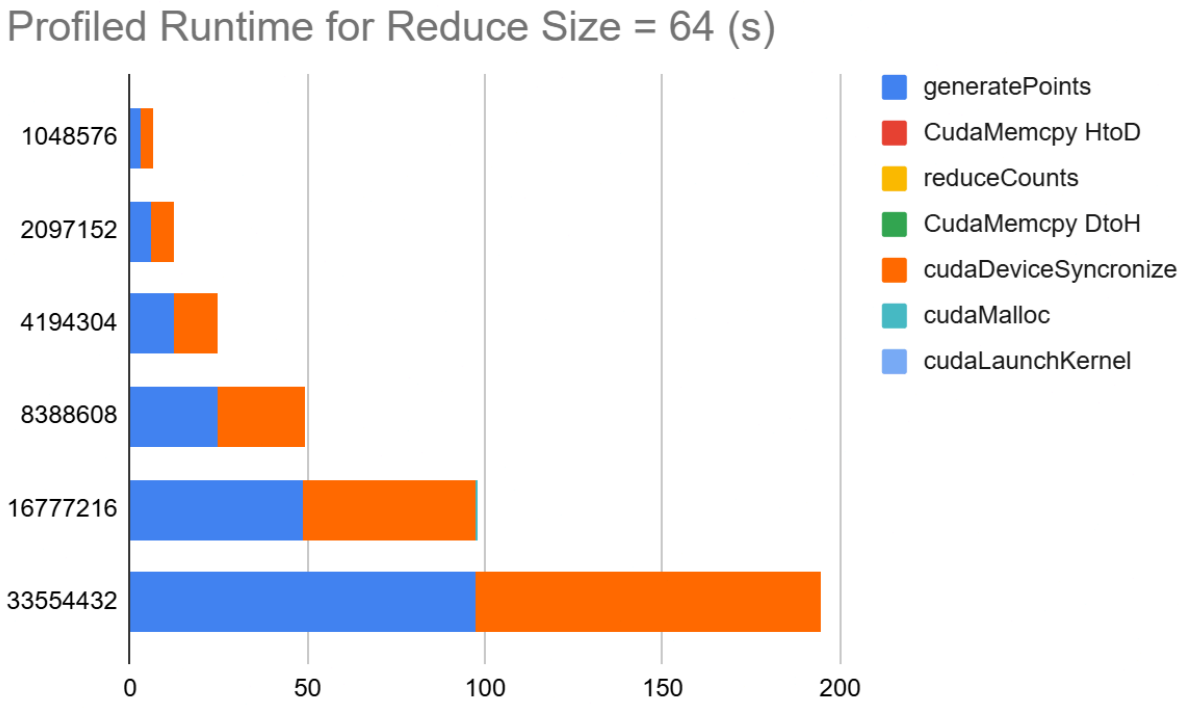

Figure 4: Profiled Monte Carlo GPU with reduceSize of 64. There are some minor time gains on the smaller test cases, but large cases performed similarly to size 32.