Programming Assignment 1 Report

Noel Sengel
Programmable Accelerator Architectures

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

The output of SAXPY on the GPU is:

Scale is 5.00129

First 5 of vector X: 4.6079 3.524 9.74528 4.17751 6.03867

First 5 of vector Y: 1.76742 1.93983 4.85859 4.19014 1.06798

First 5 values of output vector Y: 24.8129 19.5644 53.5975 25.0831 31.2691

Found 0 / 32768 errors

Using nvprof, the user can profile the speed of the program for performance and optimization. To see the differences in program runtimes, the vectorSize was varied from small to large. The three vector sizes chosen were 1024, 32768, and 1,048,576. The three GPU activities in the code is cudaMemCpy from the host to the device, the kernel saxpu_gpu, and cudaMemCpy from the device to host. When varying the vector sizes, the runtime of each of these activities changes.

Looking at the chart, as vector size increases, the time percentage spent running the cudaMemCpy from the host to device increases while the other two activities decrease. This makes sense because as the vector size increases, more space in memory needs to be copied from the CPU to the GPU. This agrees with memory being a limiting factor in execution time on the GPU. Even though all three increase in execution time as the vector size increases, the memory takes up so memory copy takes up so much time drastically changing the percentage of time running.

For the API calls, the main time is spent in cudaMalloc which slightly decreases as vector size increases. More notable is the cudaMemCpy API call that changes drastically when going from vector size of 32768 to 1,048,576. This once again agrees with memory being a limiting factor.

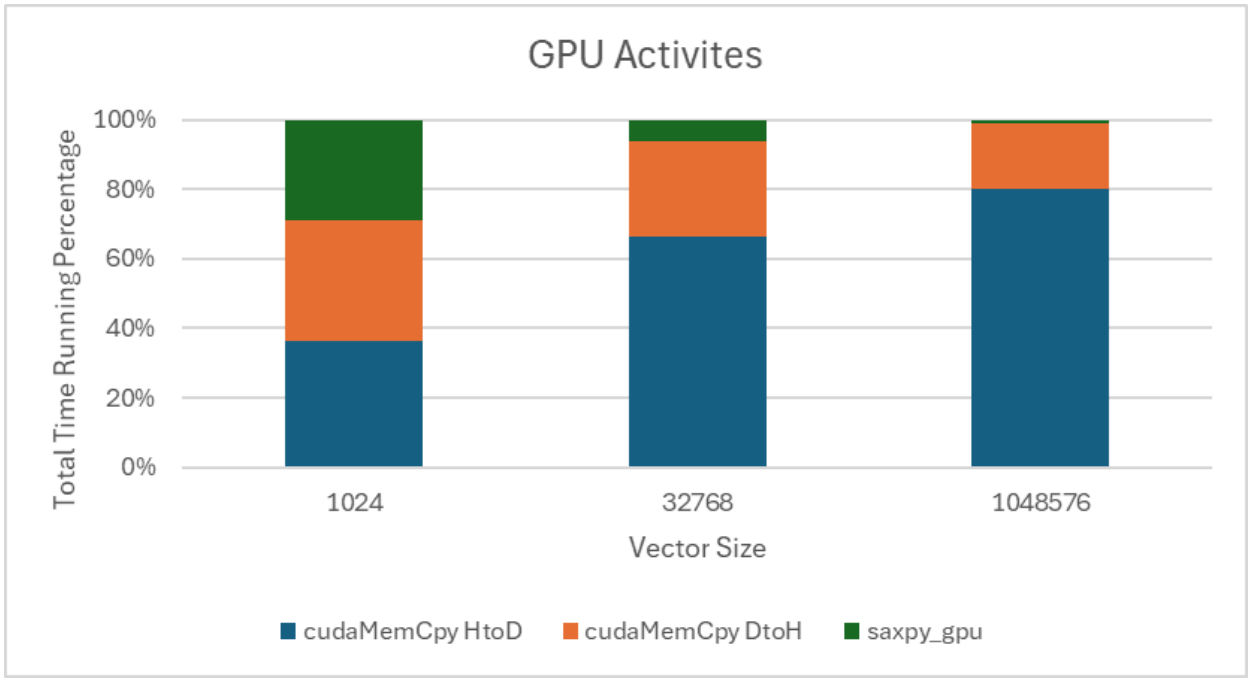

Chart 1: GPU Activities SAXPY

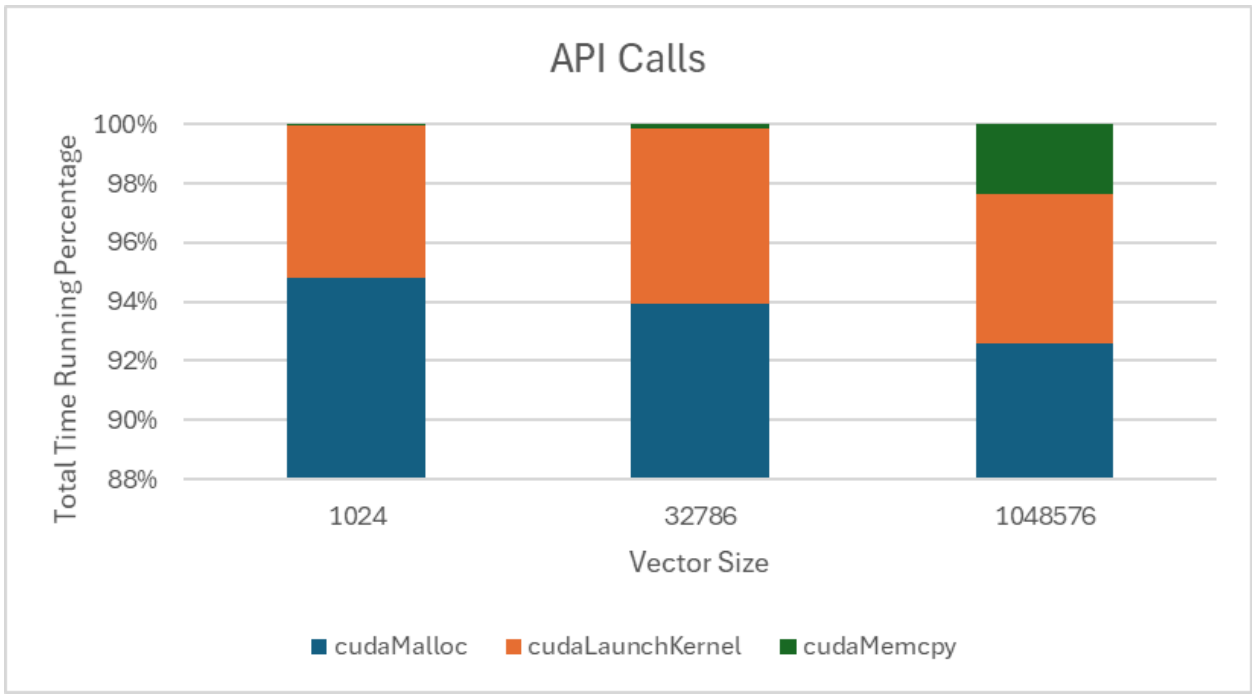

Chart 2: API Calls SAXPY

Part B: Monte Carlo estimation of the value of pi

The output of Monte-Carlo Pi Estimation on the GPU is:
Running Monte-Carlo Pi Estimation on GPU!

Estimated Pi = 3.14152
It took 0.169881 seconds.

... Done!

Using nvprof on the Monte-Carlo simulation, details of program executing/running time can be examined. The MC_SAMPLE_SIZE was varied with values of 1024, 32768, and 1048576.

The major GPU activities were generatePoints, reduceCounts, and cudaMemCpy device to host. As the sample size increased, the generatePoints function increased drastically taking up to almost 100 percent of the execution time. This is due to the computation increase and global memory usage from 1024 points to 1048576. The usage of global memory is slow and became a bottleneck in this program. Even though reduce counts, and cudaMemCpy also increase in execution time, they are far outweighed by generatePoints.

For the API calls, there are three: cudaMalloc, cudaLaunchKernel, and cudaMemCpy. cudaMalloc started off at a high percentage and lowered as the size increased due to cudaMemCpy increasing. This is due to the d_totals increasing in size and having to be copied back from the device to the host.

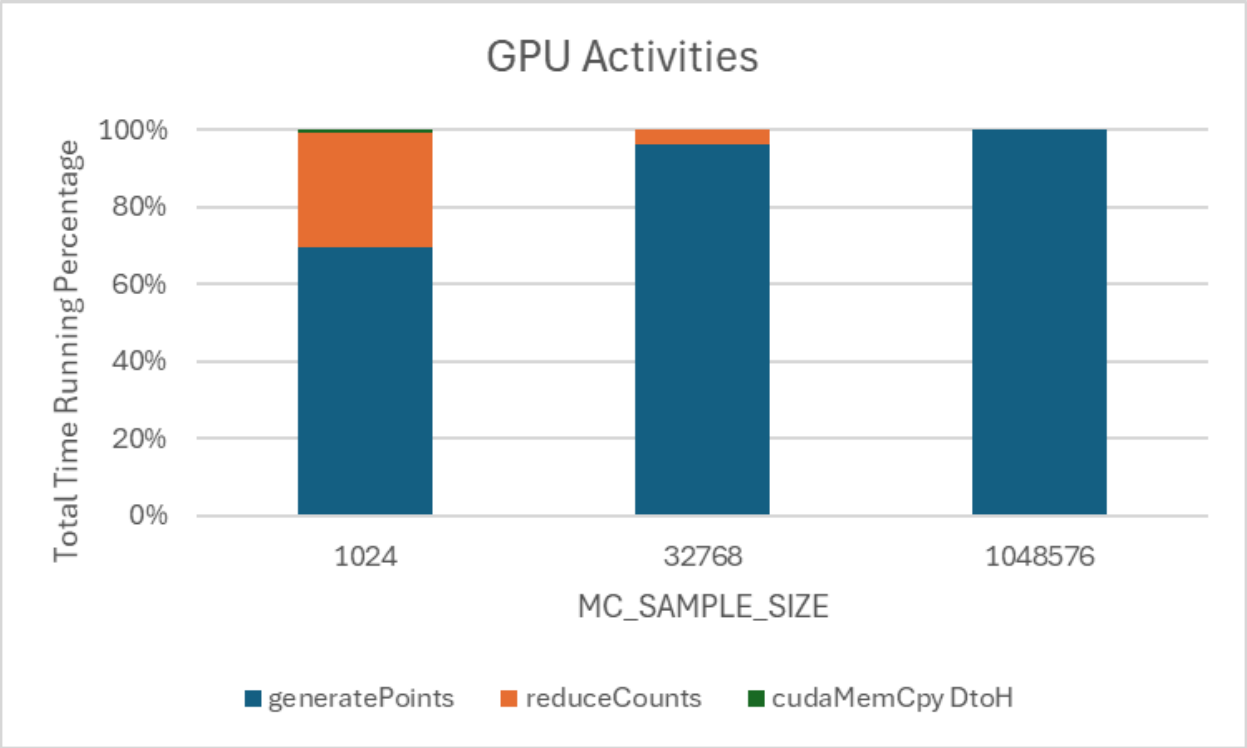

Chart 3: GPU Activities Monte Carlo

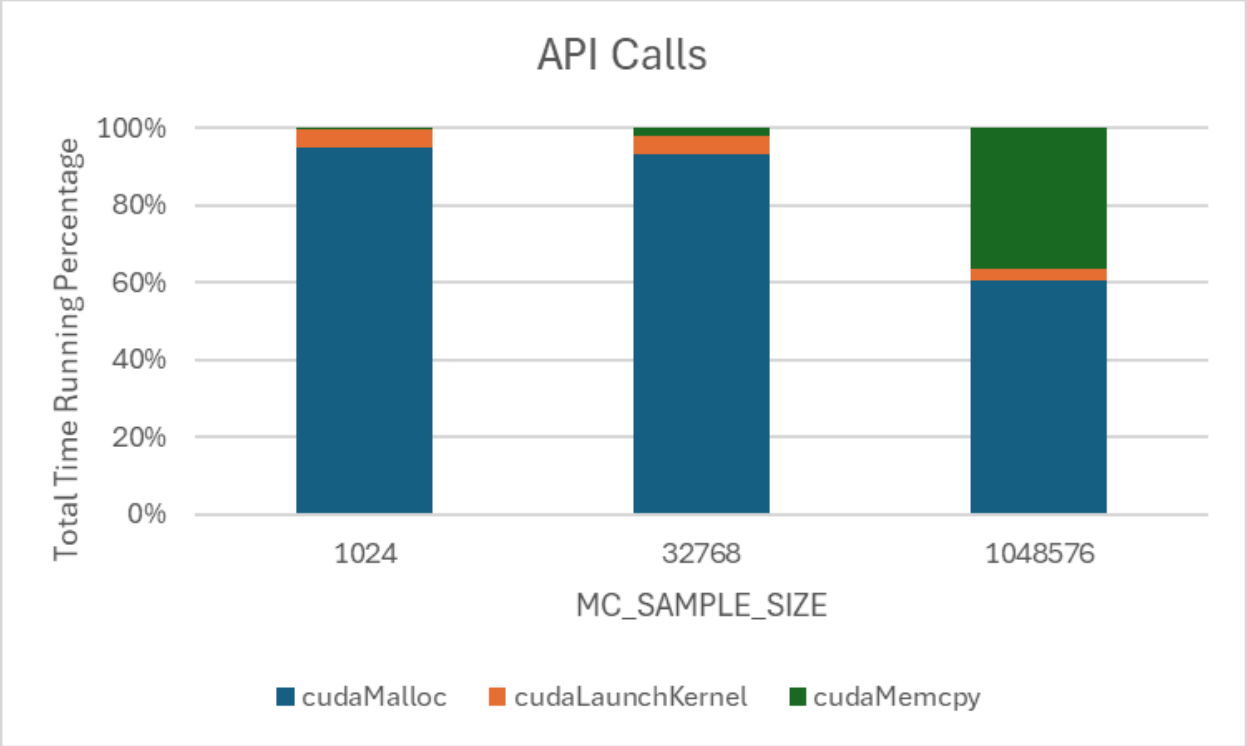

Chart 4: API Calls Monte Carlo