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 program's performance was analyzed by changing the vector sizes of 1024, 32,768, and 1,048, 576. The three major GPU activities in the code are cudaMemCpy (host to device), the saxpy_gpu kernel, and cudaMemCpy (device to host)

As vector size increases, the percentage of execution time spent on cudaMemCpy (host to device) increases, while kernel execution and device-to-host memory transfers occupy a smaller fraction. This is due to PCIe bandwidth limitations, where larger vector sizes require more data movement, making memory transfers a bottleneck. Although all three operations increase in absolute execution time, cudaMemCpy grows at a faster rate, significantly impacting the overall runtime distribution.

For API calls, the primary time-consuming operation is cudaMalloc, which remains relatively consistent but decreases in percentage as vector size grows. More notably are cudaMemCpyAPI calls increase drastically when transitioning from 32,768 to 1,048,576, reinforcing the impact of memory transfer overhead. This behavior is expected since cudaMalloc occurs once per allocation, whereas memory transfers scale with data size.

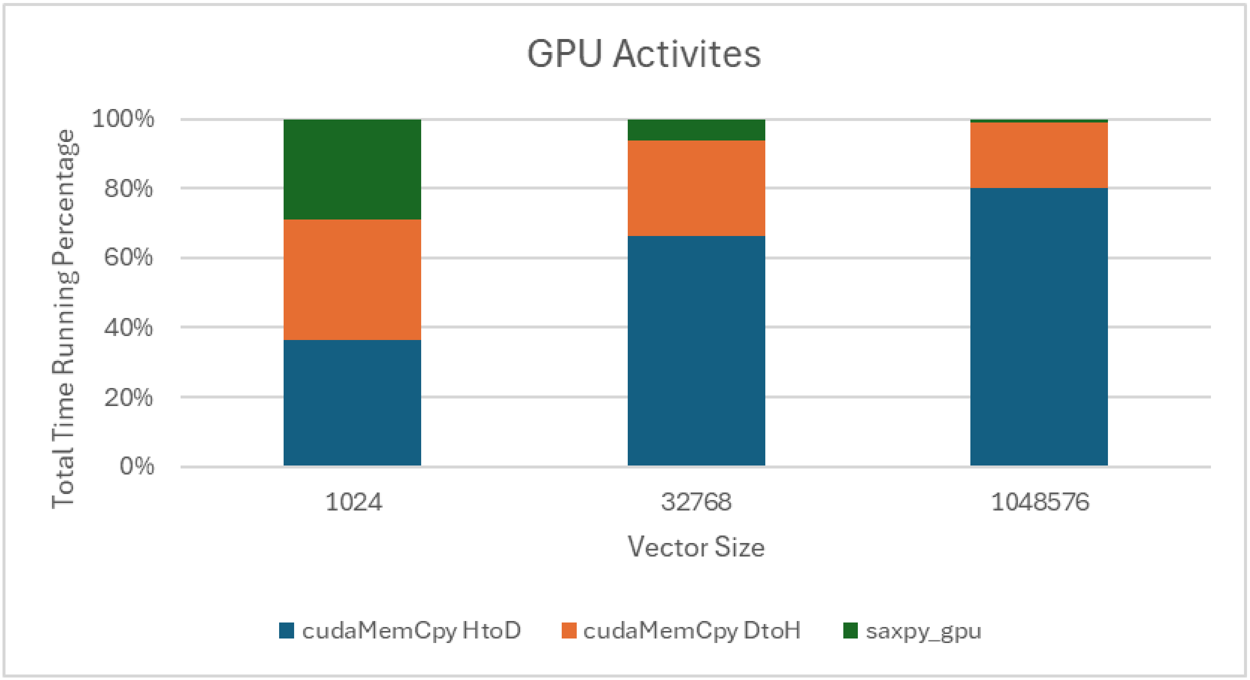

Chart 1: GPU Activities SAXPY

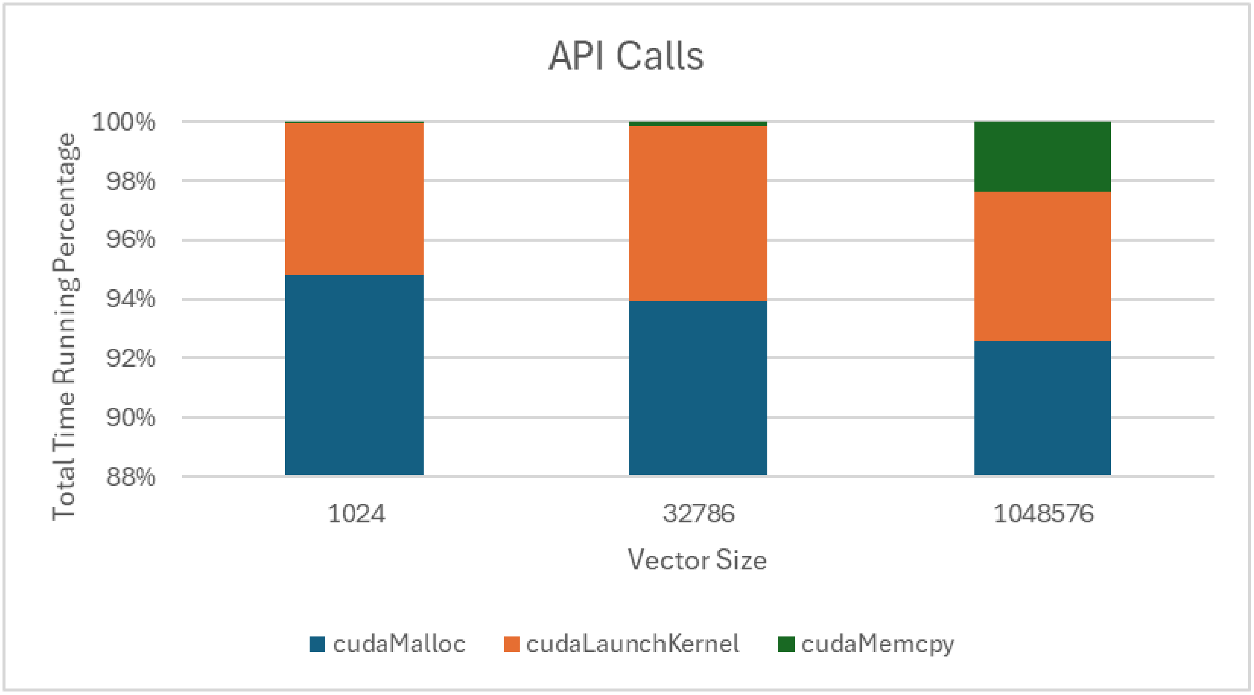

Chart 2: API Calls SAXPY

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, execution time was analyzed while varying MC_SAMPLE_SIZE 1024, 32,768, and 1,048,576.

The major GPU activities include generatePoints, reduceCounts, and cudaMemCpy (device to host). As sample size increases, the generatePoints function dominates execution time, accounting for nearly 100% of runtime at the largest sample size. This is primarily due to increased computational workload and global memory accesses, which are relatively slow. Since each generated sample requires coordinate calculations and condition checks, the function scales non-linearly with input size.

Although reduceCounts and cudaMemCpy also increase in execution time, their contribution remains much smaller compared to generatePoints. The heavy reliance on global memory further exacerbates execution time, as uncoalesced memory accesses introduce latency. Optimizing memory usage, such as using shared memory or reducing global memory transactions, could mitigate this issue.

For API calls, cudaMalloc, cudaLaunchKernel and cudaMemCpy contribute to execution time. cudaMalloc initially represents a significant fraction but decreases in percentage as the sample size increases, similar to SAXPY. The most notable shift occurs in cudaMemCpy , particularly when copying results back to the host. As MC_SAMPLE_SIZE increases, the size of d_totals grows, leading to larger memory transfers and increased overhead.

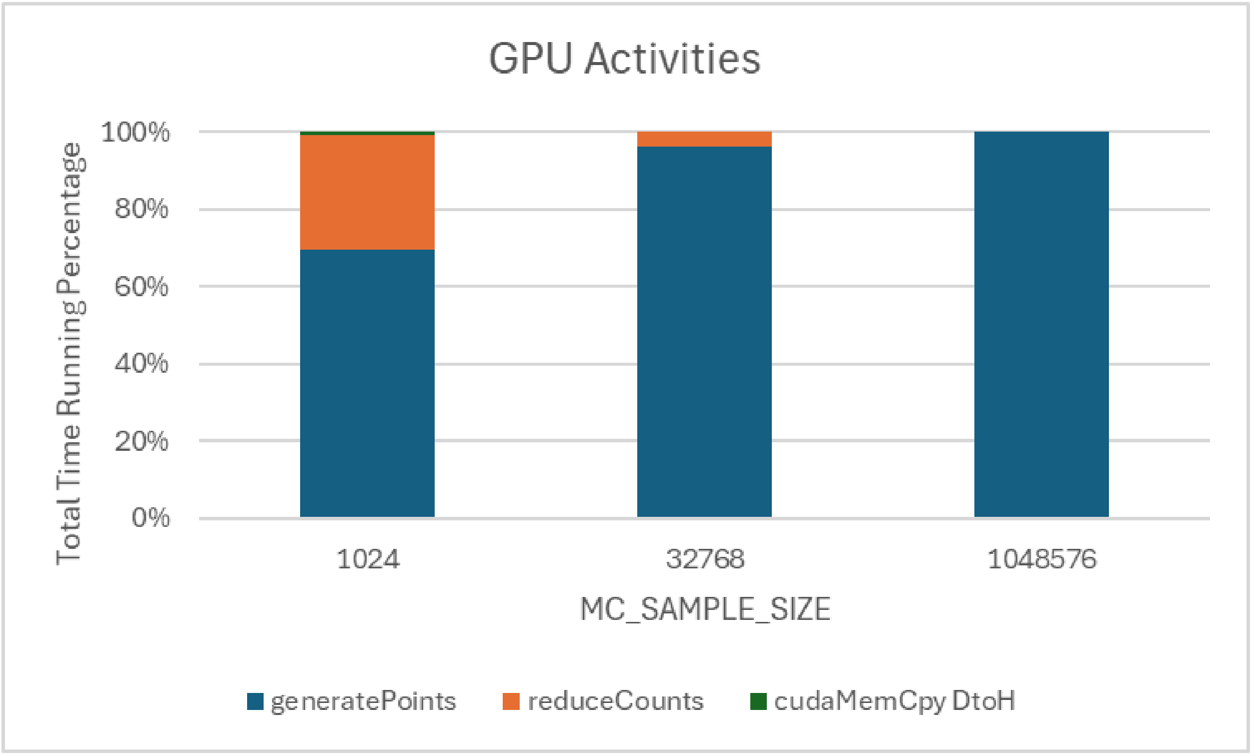

Chart 3: GPU Activities Monte Carlo

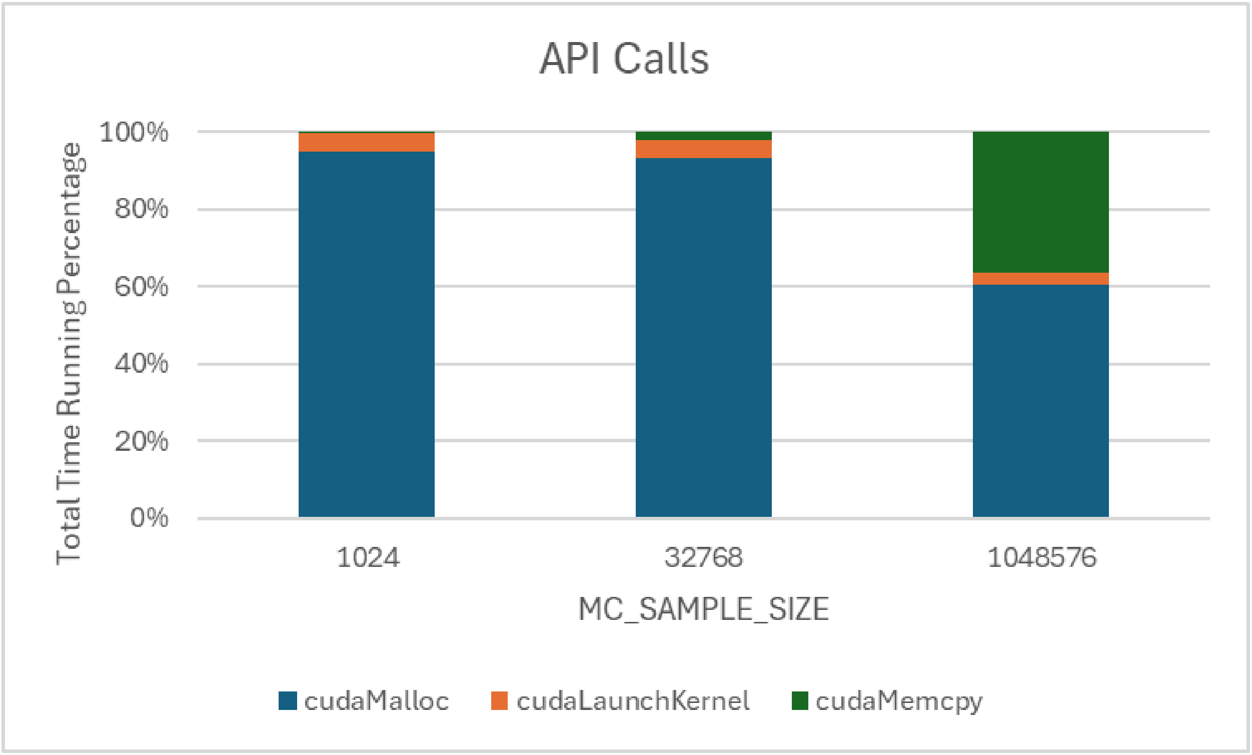

Chart 4: API Calls Monte Carlo

Conclusion:
The performance analysis using nvprof confirms that memory operations in the usage of GPUs are a bottleneck in performance. This occurred in both SAXPY, and Monte Carlo simulations. For SAXPY, host-to-device transfers dominate the execution time as vector size grows, furthering the bandwidth constraints of the PCIe. For Monte Carlo, the generatePoints function accounts for almost all of the execution time at large sample sizes due to inefficient global memory accesses and heavy computation workload. Reducing memory transfer overhead and optimizing memory access patterns could aid in performance.