ECE60827: CUDA Programming Part 1

Vedant Paranjape (PUID: 37709323)
Github ID: VedantParanjape

Part A

Single-precision A · X Plus Y (SAXPY)

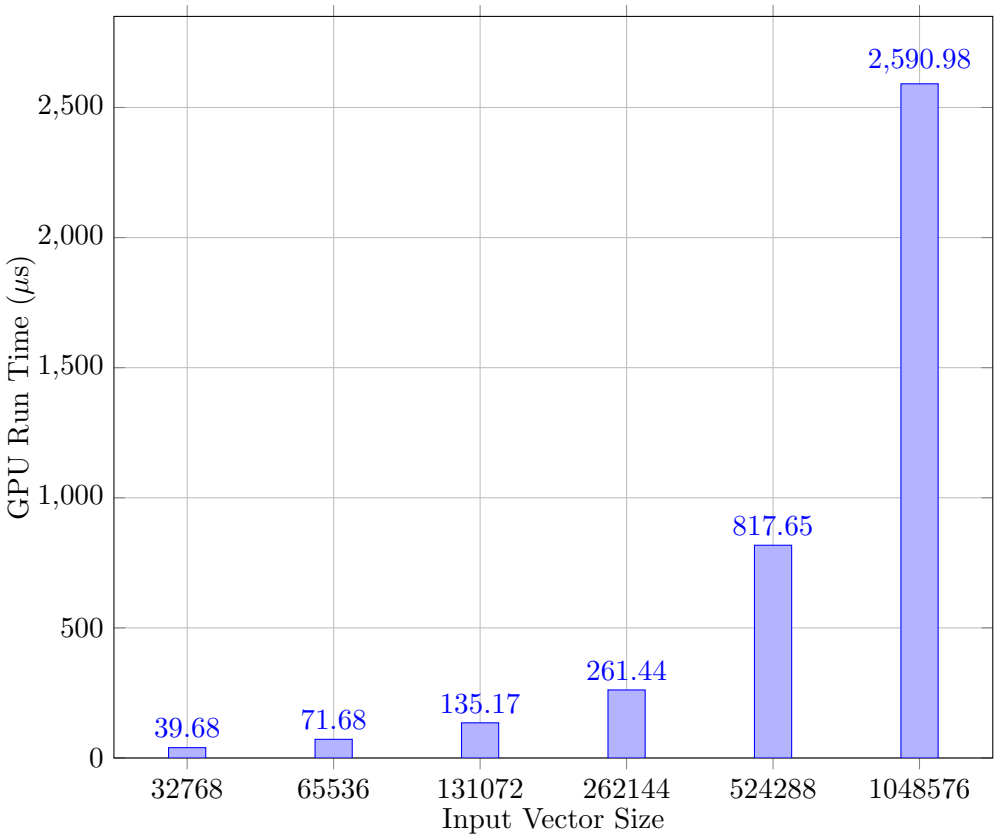

Figure 1: A plot comparing the vectorSize for SAXPY program vs execution time on Nvidia GPU

In this part, I implemented the SAXPY kernel to speed up SAXPY computation on Nvidia GPUs. For evaluation, I varied the input vector size for this program. In figure 1, I have plotted the GPU runtime for this kernel for various input sizes. In my implementation, I used 256 threads per block to increase the SM utilization. In the graph, we can see that as the input sizes are increased, the execution time increases as well.

I have also provided a breakdown of the execution time in table 1 and plotted a stacked bar chart in figure 2. The runtime of the saxpy_gpu kernel is too small compared to CudaMemcpy, so the stack is visibly very small in the graph. Please refer to table 1 for the accurate values. As evident from the data, the GPU spends most of its time moving the x and y array between host CPU and GPU. The actual execution time of saxpy_gpu kernel is a small percentage of the total execution time. Even though it is a small percentage, the runtime of saxpy_gpu increases with an increase in the input size. So does the time spent in copying memory between the host CPU and the GPU.

Input Vector Size	CUDA memcpy HtoD (μs)	CUDA memcpy DtoH (μs)	saxpy_gpu kernel (μs)	Total time (μs)
32768	26.175	11.072	2.432	39.679
65536	47.712	21.184	2.784	71.68
131072	90.464	40.992	3.712	135.168
262144	176.16	80.8	4.48	261.44
524288	648.63	162.75	6.272	817.652
1048576	1511.2	1064.0	15.775	2590.975

Table 1: Table of breakup of runtime for varying vectorSize for SAXPY program on Nvidia GPUs

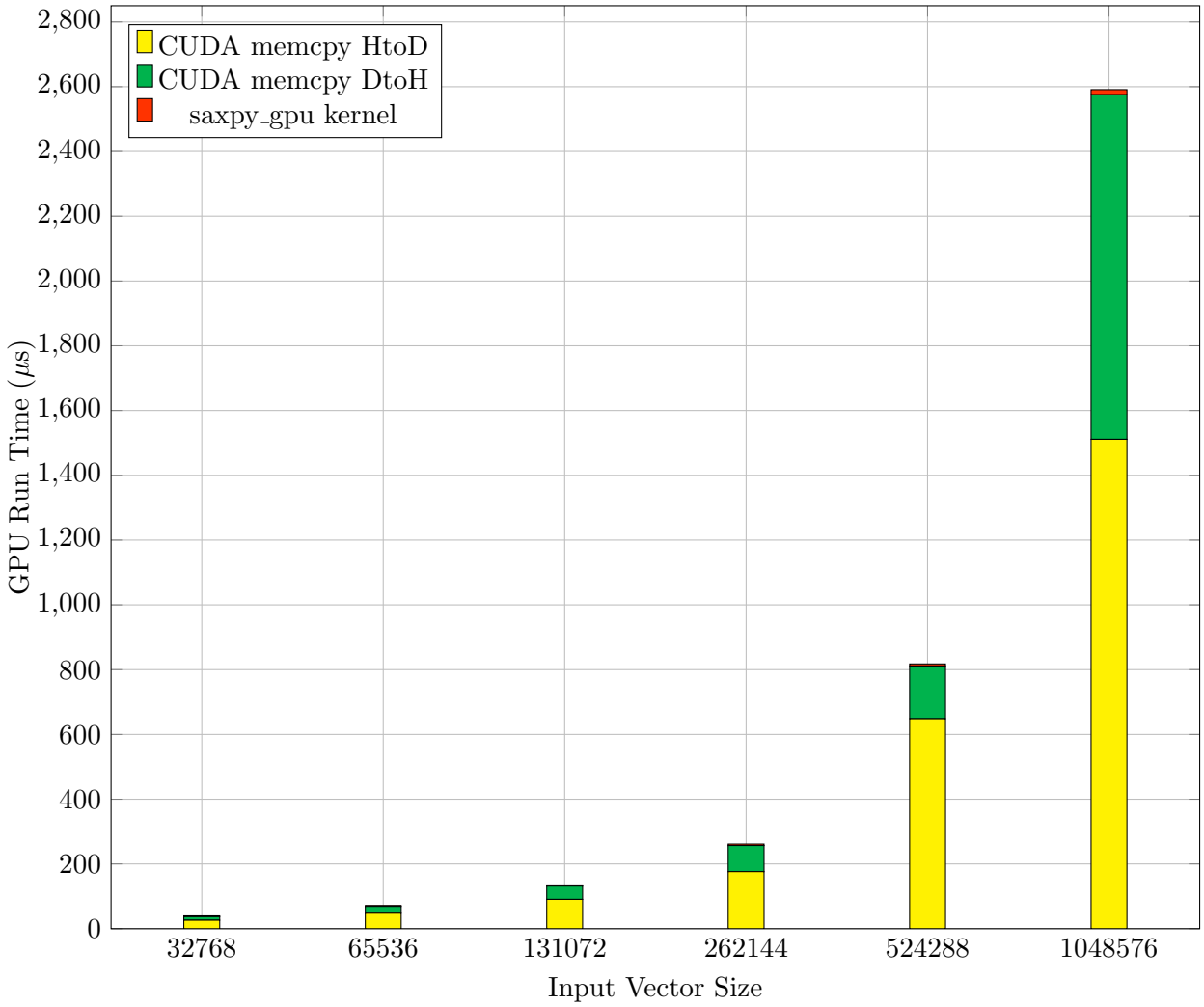

Figure 2: Breakup of runtime for varying vectorSize for SAXPY program on Nvidia GPUs

Part B

Monte Carlo estimation of the value of π

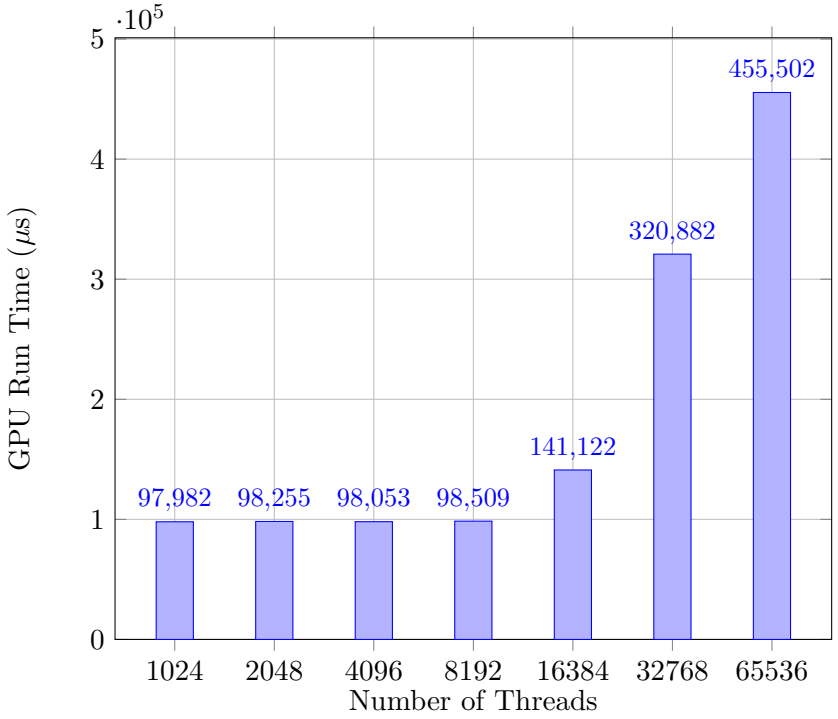

Figure 3: A plot comparing the number of threads used for Monte-Carlo estimation program vs execution time on Nvidia GPU

In this part, I implement the Monte Carlo estimation of the π program for Nvidia GPUs. In this estimation, we need to generate random numbers between 0 and 1 and then find their distance from (0, 0). The random points can be generated on the CPU and then copied to the GPU for processing, or they can be generated on the GPU itself. I went with the latter approach and ended up using the CuRAND library. This saves execution time by eliminating the need to copy from Host to Device.

For evaluation, I have varied the number of threads to be used for this computation. In figure 3, I have plotted the GPU runtime for this program for various numbers of threads. I have also provided a breakdown of the execution time in table 3 and plotted a stacked bar chart in figure 5. The runtime of the generatePoints kernel is high compared to other operations, so the other values are visibly very small in the graph. Please refer to table 3 for the accurate values. I also vary the number of samples to be computed per thread. I have plotted a bar chart depicting the same in figure 4 with y-axis on a log scale. As we can see, the runtime follows an upward trend with an increase in sample size keeping the number of threads constant at 1024. I have listed the breakdown of the runtime in table 2. The runtime for reduceCounts kernel and CUDA memcpy stays the same for varying sample sizes, but the runtime for generatePoints kernel follows an upward trend.

If we look at the execution time, it is fairly similar for 1024 to 8192 threads, but it blows up as soon as we go higher. This is because while calling the generatePoints kernel, we set the threads per block to 256. With 16384 threads and 64 blocks the GPU is over-saturated, and hence we see an increase in execution time as we increase the number of threads after 8192. The reduceCounts kernel take $\sim$5 μs for 1024 and 2048 threads, and increases after that, stabilizing at $\sim$10 μs. The time spent copying the totals array from Device to Host also follows a slightly upward trend.

Input Sample Size	generatePoints kernel (μs)	reduceCounts kernel (μs)	CUDA memcpy DtoH (μs)	Total time (μs)
10^6	97597.0	5.216	1.792	97604.008
10^7	976650.0	5.216	1.792	976657.008
10^8	9744480.0	5.216	1.792	9744487.008
10^9	97426400.0	5.216	1.760	97424606.976

Table 2: Table of breakup of runtime for varying sample size per thread used for Monte-Carlo estimation program vs execution time on Nvidia GPU

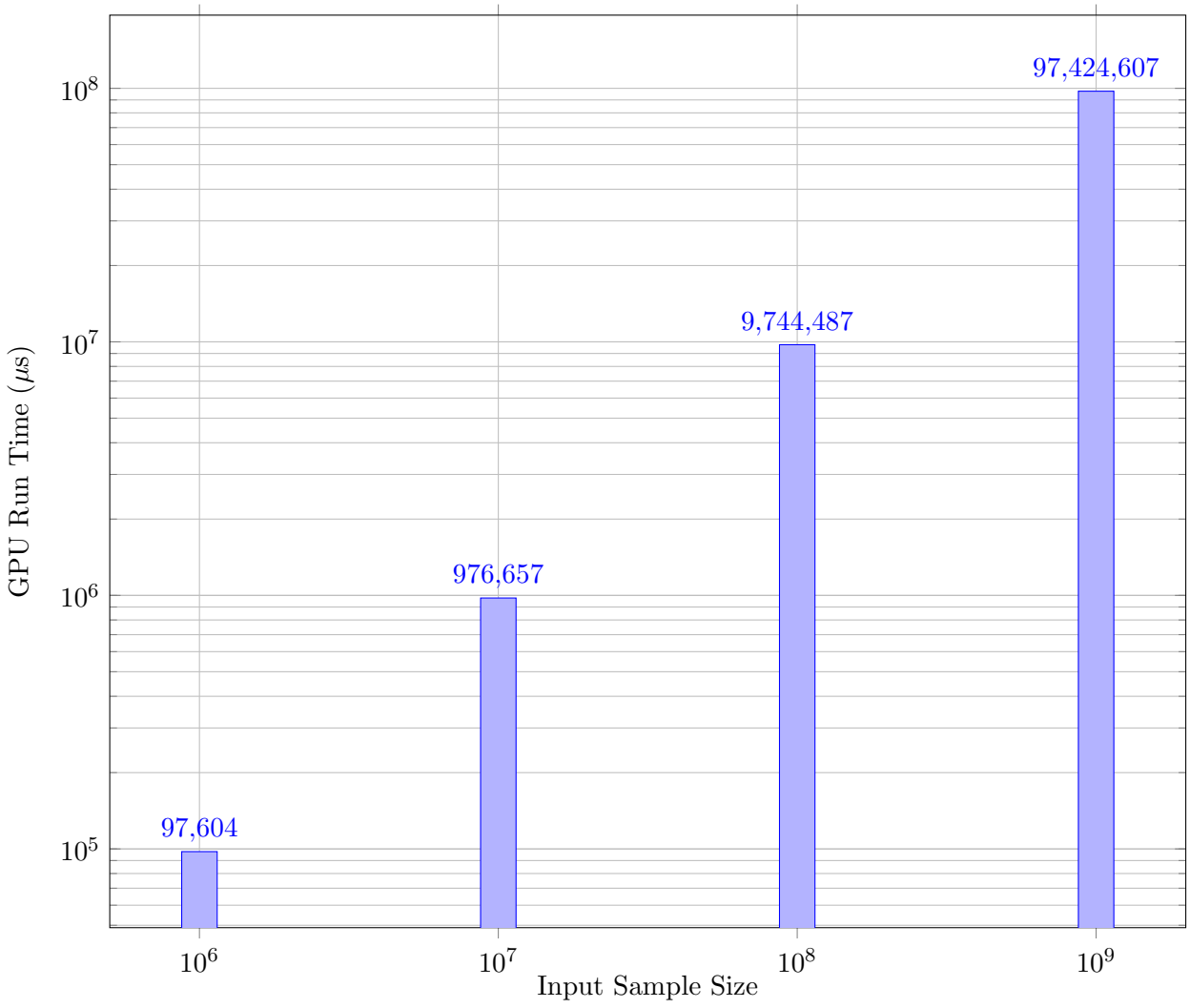

Figure 4: A plot comparing the sample size per thread used for Monte-Carlo estimation program vs execution time on Nvidia GPU

Input Size	generatePoints kernel (μs)	reduceCounts kernel (μs)	CUDA memcpy DtoH (μs)	Total time (μs)
1024	97975.0	5.248	1.76	97982.008
2048	98248.0	5.184	1.792	98254.976
4096	98044.0	6.912	1.760	98052.672
8192	98497.0	10.015	1.792	98508.807
16384	141110.0	10.080	1.856	141121.936
32768	320870.0	9.9530	2.048	320882.001
65536	455490	10.048	2.3040	455502.352

Table 3: Table of breakup of runtime for varying number of threads for Monte-Carlo estimation program on Nvidia GPUs.

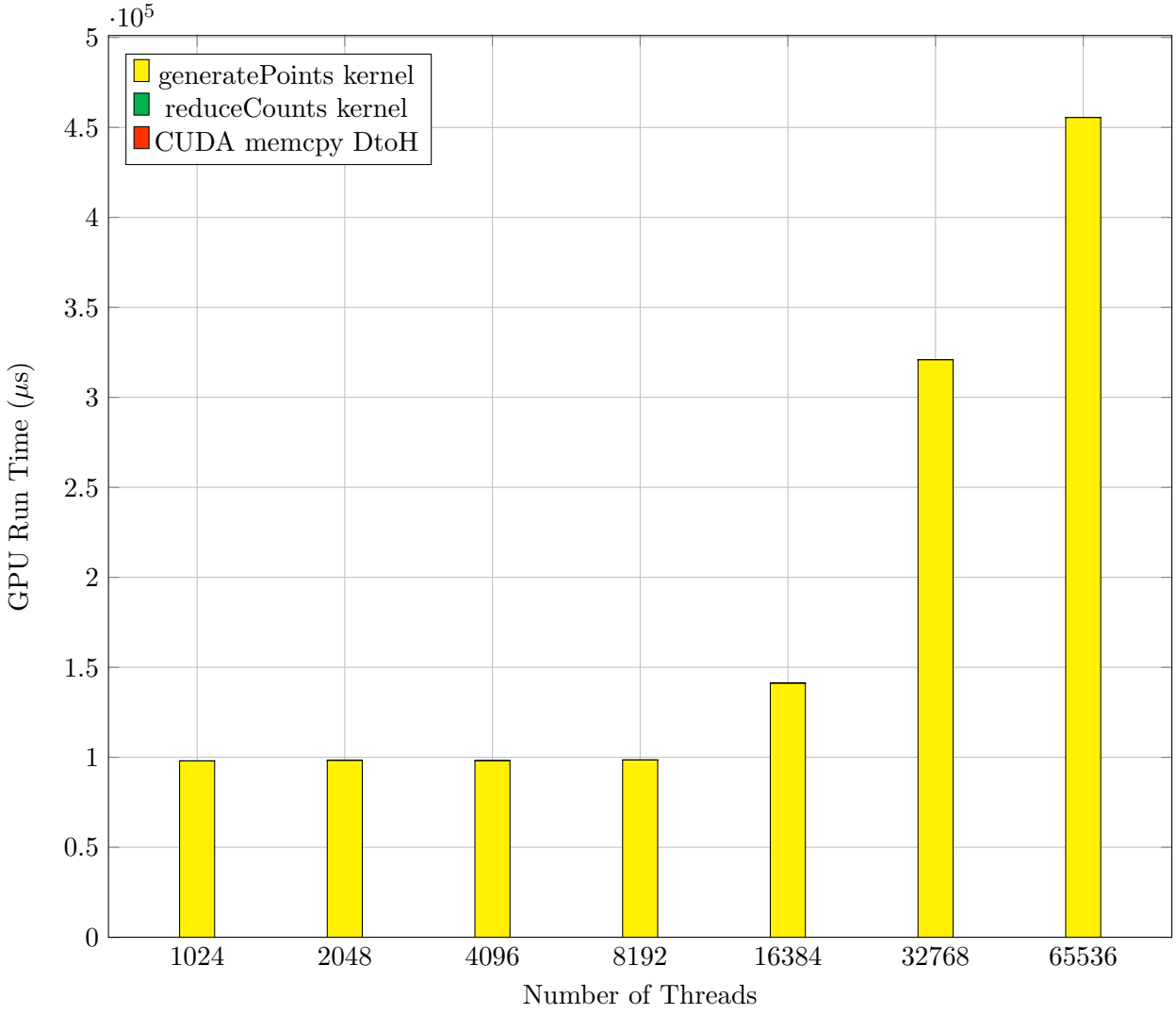

Figure 5: Breakup of runtime for varying number of threads for Monte-Carlo estimation program on Nvidia GPUs.