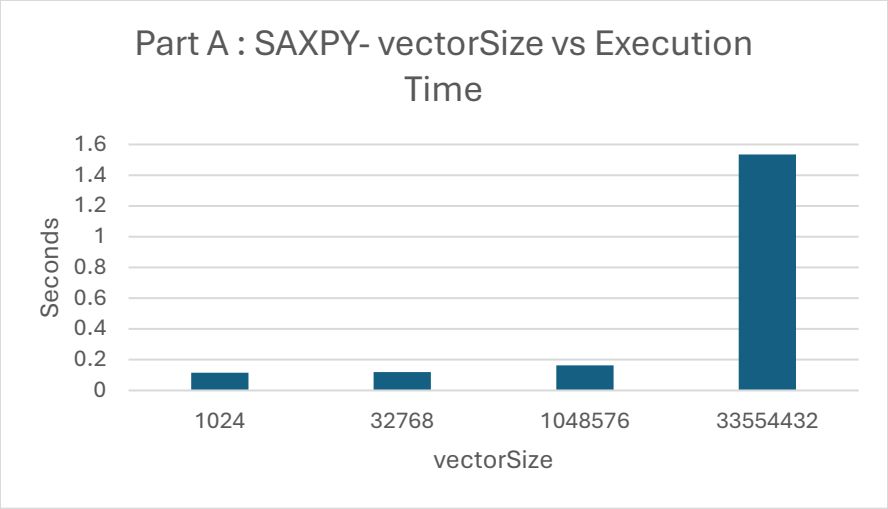
ECE 60827 - Kevin Darmasaputra
Github username: pischelus

CUDA Programming Assignment #1 Report

Part A – SAXPY

#1 – vectorSize vs execution time

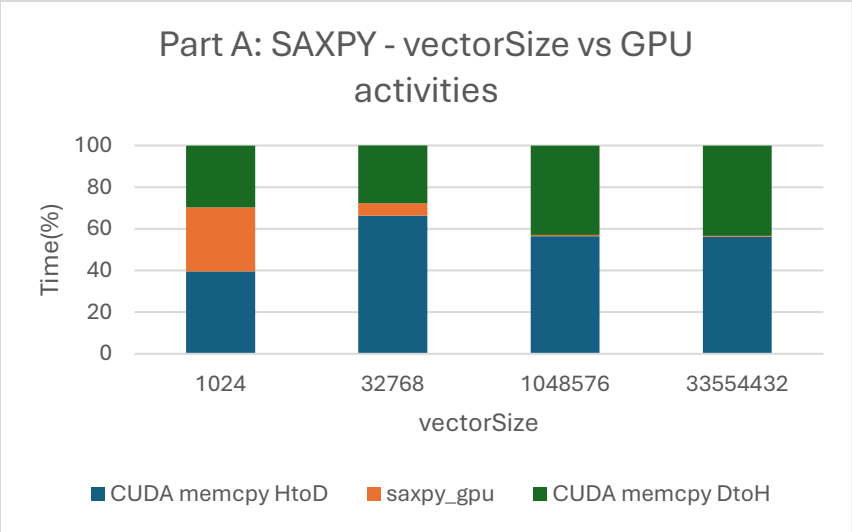
vectorSize	seconds
1024	0.116381
32768	0.11884
1048576	0.162987
33554432	1.53554



As expected, the bigger the vectorSize is, the longer the execution time is. One noticeable observation is the execution time increase only slightly for the 1st 3 sizes (1024, 32768, 1048576), but it increases by a huge factor from 1048576 to 33554432. This is because the operation is memory bound due to memcpy for larger vectorSize. The parameter that supports this argument will be shown in the subsequent section.

#2 – vectorSize vs GPU activities

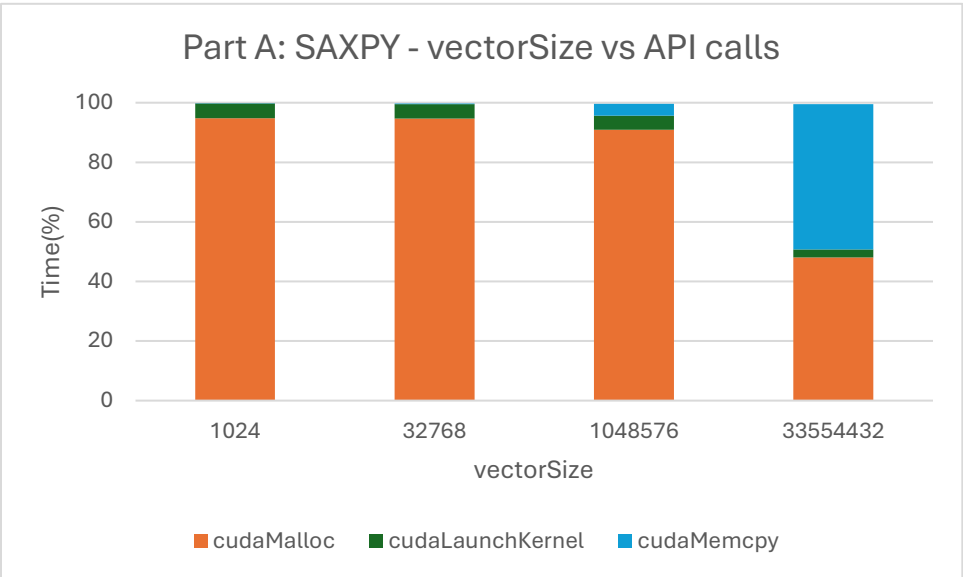
vectorSize	GPU activities (%)		
	CUDA memcpy HtoD	saxpy_gpu	CUDA memcpy DtoH
1024	39.62	30.66	29.72
32768	66.27	5.93	27.81
1048576	56.48	0.63	42.89
33554432	56.18	0.49	43.33



Based on the stacked bar chart above, we can see the % of time spent in saxpy_gpu keeps decreasing with increasing vectorSize, this is because of the reason mentioned in #1, saxpy is memory bound. The kernel code saxpy_gpu can be executed fast in GPU and both memcpy dominates the overall time spent. In all 4 vectorSize, CUDA memcpy HtoD always takes slightly higher % in time spent compared to CUDA memcpy DtoH, this is due to HtoD copies from 2 variables and DtoH only needs to copy over one variable, which is the result.

#3 – vectorSize vs API calls

	API calls (%)		
vectorSize	**cudaMalloc**	**cudaLaunchKernel**	**cudaMemcpy**
1024	94.74	4.95	0.05
32768	94.64	4.87	0.22
1048576	90.93	4.76	3.9
33554432	48.06	2.7	48.73



For vectorSize 1024, 32768, and 1048576, cudaMalloc dominates the % time spend in API calls. For vectorSize 1048576, we can start seeing a portion of cudaMemcpy catches up with cudaLaunchKernel. And for the highest vectorSize, cudaMemcpy takes higher % in time compared to cudaMalloc, they both takes about 48-49% of % time. It is worth noting that cudaMalloc and cudaLaunchKernel remain constant (~100ms for cudaMalloc and ~5ms for cudaLaunchKernel) through all vectorSize, while cudaMemcpy keeps getting larger. Hence, the growing time % for cudaMemcpy in stacked bar chart above.

Part B – Monte Carlo

#1 – sampleSize, generateThreadCount, reduceSize vs GPU activities

This part combined 3 varying parameters (sampleSize, generateThreadCount, and reduceSize) into one section when comparing against their GPU activities because they behave similarly.

For varying sampleSize: generateThreadCount is held at 1024 and reduceSize is held at 32
For varying generateThreadCount: sampleSize is held at 1e6 and reduceSize is held at 32
For varying reduceSize: sampleSize is held at 1e6 and generateThreadCount is held at 1024

	GPU activities (%)		
sampleSize	generatePoints	reduceCounts	CUDA memcpy DtoH
1e5	99.9	0.07	0.03
1e6	99.99	0.01	0
1e7	100	0	0
1e8	100	0	0

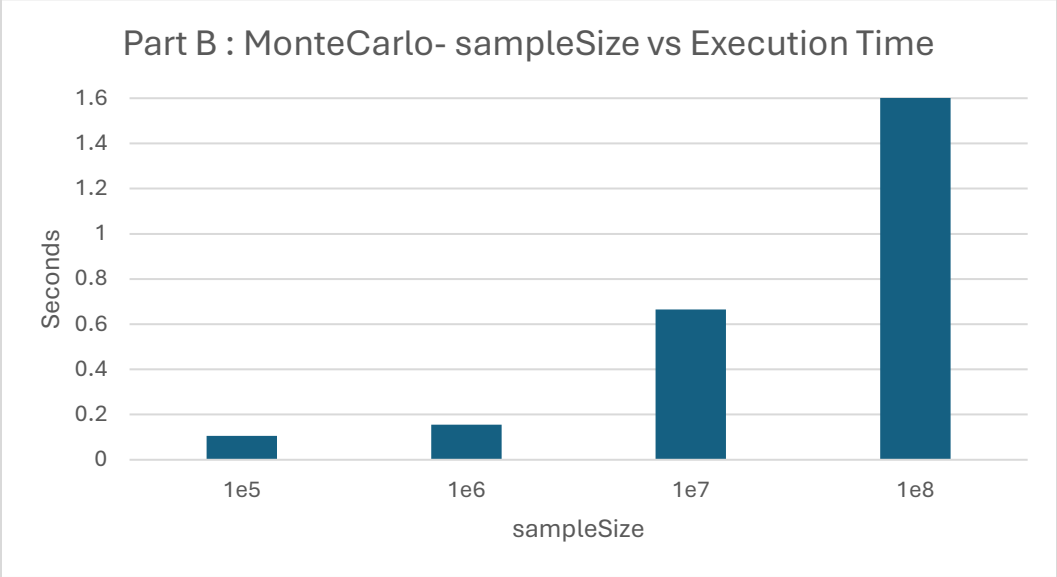
	GPU activities (%)		
generateThreadCount	generatePoints	reduceCounts	CUDA memcpy DtoH
1024	99.99	0.01	0
32768	99.99	0.01	0
1048576	100	0	0
33554432	100	0	0

	GPU activities (%)		
reduceSize	generatePoints	reduceCounts	CUDA memcpy DtoH
16	100	0	0
32	99.99	0.01	0
64	99.99	0.01	0
128	99.98	0.02	0

No matter what the sampleSize, generateThreadCount, and reduceSize are, GPU spent most of its time generatePoints. This is expected because it's the activity that generate the most threads, even if we look at the individual thread, generatePoints has to run as many as sampleSize. As a result, 99.9%+ is dominated by generatePoints. Stacked bar chart for these 3 are not shown since it'll just show a single-color bar for generatePoints.

#2 – sampleSize vs execution time (generateThreadCount = 1024, reduceSize = 32)

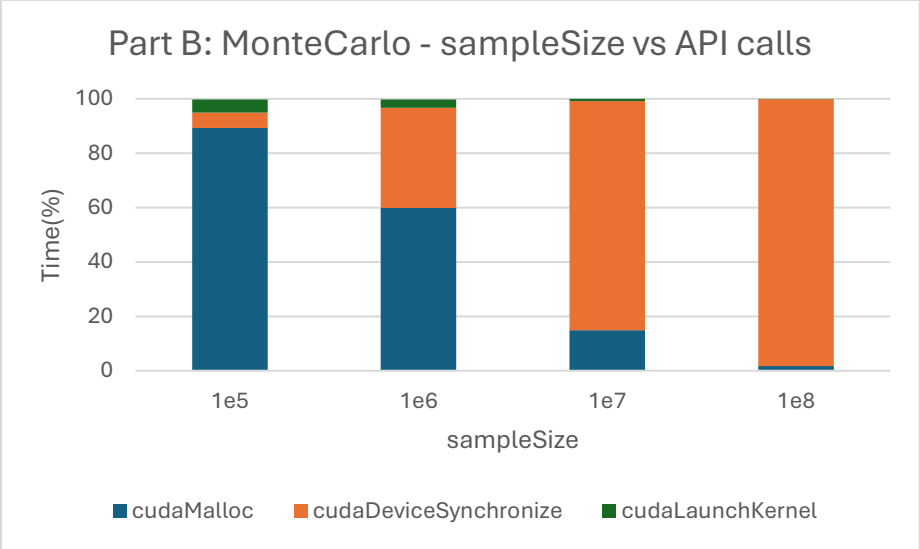
sampleSize	seconds
1e5	0.105865
1e6	0.155479
1e7	0.665153
1e8	5.75336



As expected, the bigger the sampleSize is, the longer the execution time will be. One noticeable observation is the execution time increase only slightly for the 1st 2 sizes (1e5 and 1e6), but the increase starts to become larger for sample size 1e7 and 1e8. This is because the operation is compute bound for very large number, and it spend most of the time waiting for the GPU to finish the computation. Since the sampleSize is huge, there are more threads and threadblocks, as a result not all threads can run in parallel due to GPU limitation. The parameter that supports this argument will be shown in the subsequent section.

#3 – sampleSize vs API calls (generateThreadCount = 1024, reduceSize = 32)

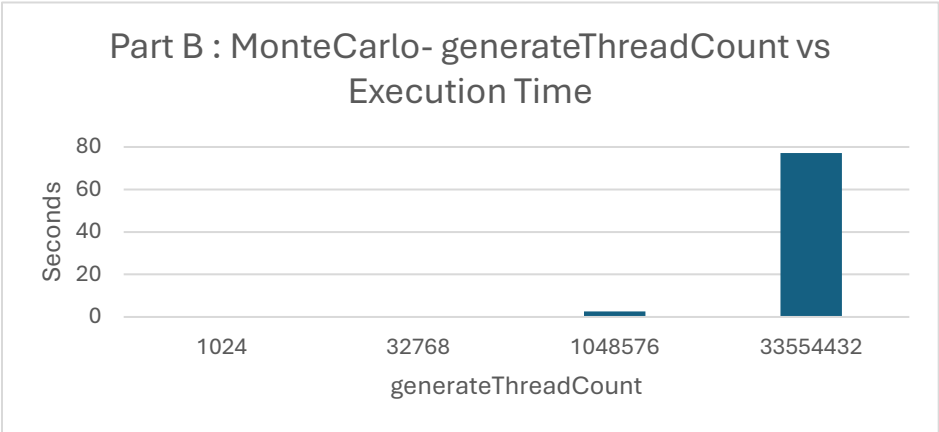
	API calls (%)		
sampleSize	cudaMalloc	cudaDeviceSynchronize	cudaLaunchKernel
1e5	89.26	5.73	4.73
1e6	59.86	36.9	3.06
1e7	14.91	84.27	0.78
1e8	1.76	98.13	0.1



cudaMalloc and cudaLaunchKernel remain constant through the sampleSizes. It's about ~100ms for cudaMalloc and ~5ms for cudaLaunchKernel. Since both cudaMalloc and cudaLaunchKernel remain constant, the increasing cudaDeviceSynchronize will eventually dominates the API calls time %. cudaDeviceSynchronize keeps increasing because it is compute bound. It spends most of the time just waiting for GPU to finish the computation. For reference: sampleSize 1e5, cudaDeviceSynchronize is ~6.5ms. sampleSize 1e8, cudaDeviceSynchronize is ~5.6s. The difference is about three orders of magnitude. Hence, the growing time % for cudaDeviceSynchronize in stacked bar chart above.

#4 – generateThreadCount vs execution time (sampleSize = 1e6, reduceSize = 32)

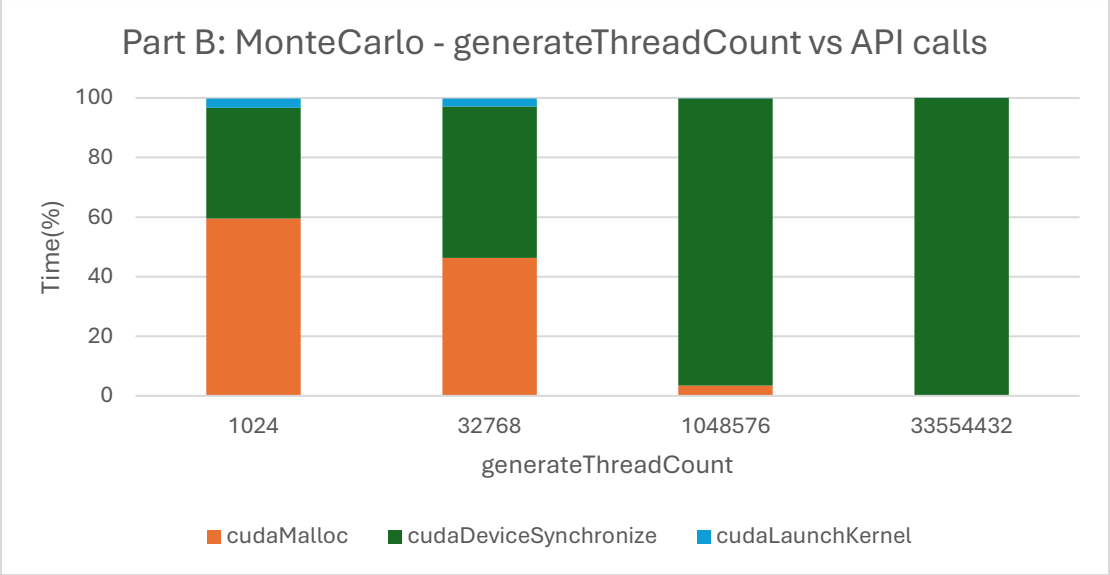
generateThreadCount	seconds
1024	0.147584
32768	0.188878
1048576	2.61685
33554432	77.1579



Looking at the effect of generateThreadCount, it mimics the effect of sampleSize, only much bigger. It's expected to have longer execution time with increasing generateThreadCount. The increase starts to become much larger for generateThreadCount >> 1048576 (2^{20}). This is because the operation is compute bound for very large number of threads. It spends most of the time waiting for the GPU to finish the computation. Since we have a very huge number of threads, there will be more threadblocks that need to be processed. Not all threadblocks can be run in parallel due to GPU limitation. The parameter that supports this argument will be discussed in the subsequent section.

#5 – generateThreadCount vs API calls (sampleSize = 1e6, reduceSize = 32)

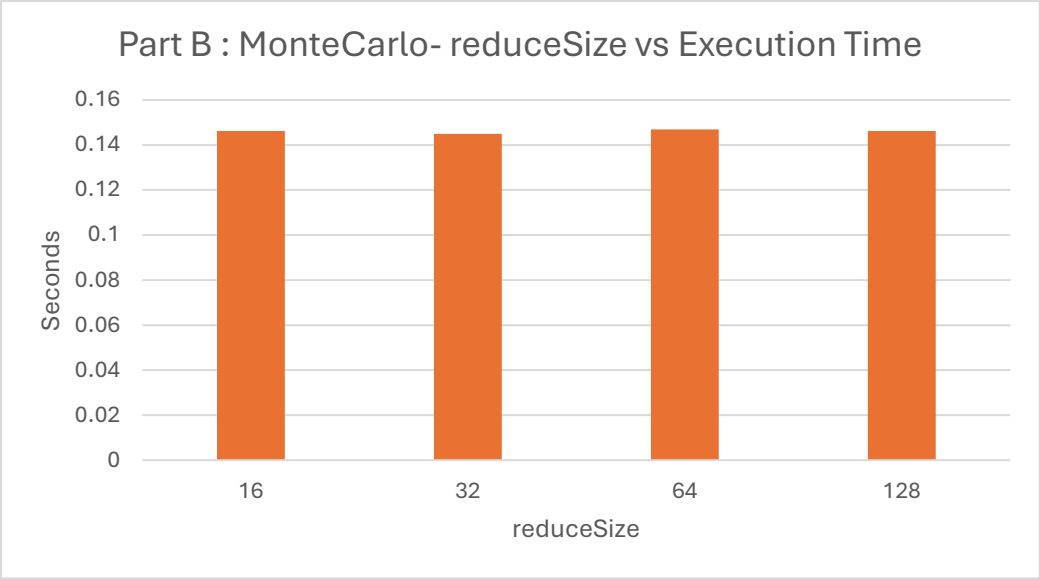
	API calls (%)		
generateThreadCount	**cudaMalloc**	**cudaDeviceSynchronize**	**cudaLaunchKernel**
1024	59.51	37.22	3.09
32768	46.27	50.87	2.7
1048576	3.47	96.3	0.2
33554432	0.14	99.84	0.01



cudaMalloc and cudaLaunchKernel remain constant through the sampleSizes. It's about ~100ms for cudaMalloc and ~5ms for cudaLaunchKernel. Since both cudaMalloc and cudaLaunchKernel remain constant, the increasing cudaDeviceSynchronize will eventually dominates the API calls time %. cudaDeviceSynchronize keeps increasing because it is compute bound. It spends most of the time just waiting for GPU to finish the computation. For reference: generateThreadCount 1024, cudaDeviceSynchronize is ~63ms. generateThreadCount 3355432, cudaDeviceSynchronize is ~77s. The difference is about three orders of magnitude. Hence, the growing time % for cudaDeviceSynchronize in stacked bar chart above.

#6 – reduceSize vs execution time (sampleSize = 1e6, generateThreadCount = 1024)

reduceSize	seconds
16	0.146164
32	0.144873
64	0.146948
128	0.146197



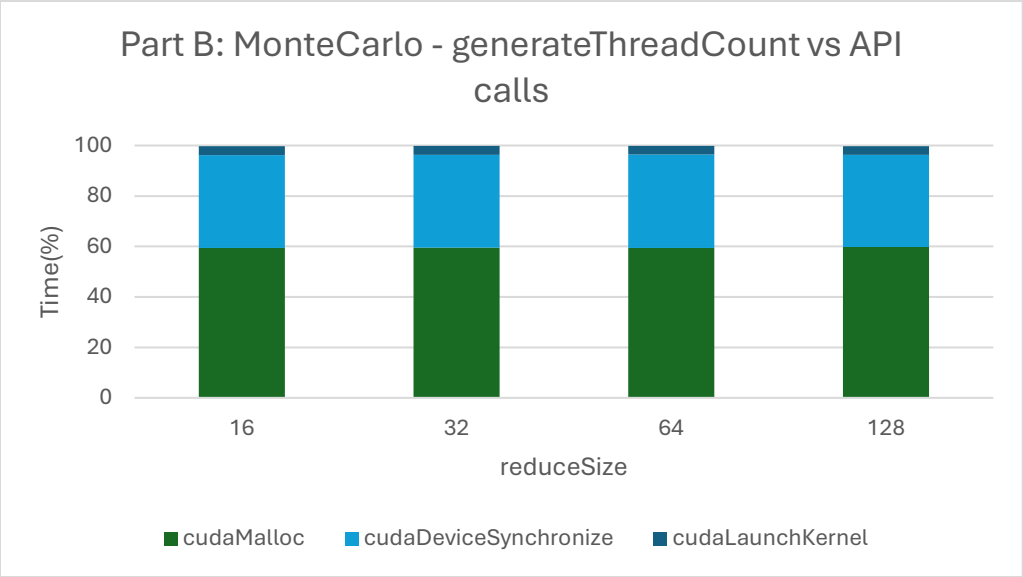
If we observe the chart above, varying reduceSize doesn't really change the execution time. This is because reduceSize is only used to reduce the partial sums using reduceCounts function. reduceCounts function doesn't really take much time compared to other GPU activities. As shown on part #1, it only takes about 0-0.02% time of GPU activities. Below shows what happens to reduceCounts with varying reduceSize:

reduceSize	reduceCounts time (us)
16	3.52
32	4.256
64	6.016
128	9.312

A single digit microsecond increase is not visible in stacked bar chart above, which explains our observation.

#7 – reduceSize vs API calls (sampleSize = 1e6, generateThreadCount = 1024)

	API calls (%)		
reduceSize	cudaMalloc	cudaDeviceSynchronize	cudaLaunchKernel
16	59.46	36.75	3.58
32	59.53	36.74	3.54
64	59.38	37.05	3.38
128	59.78	36.58	3.43



Just like observation of reduceSize vs execution time, we don't expect any noticeable difference in reduceSize vs API calls either. Any variation of time difference between cudaMalloc, cudaDeviceSynchronize, and cudaLaunchKernel can be attributed to variations in GPU runtime. As explained in #6, reduceCounts doesn't take much time in the overall scheme compared to other functions.