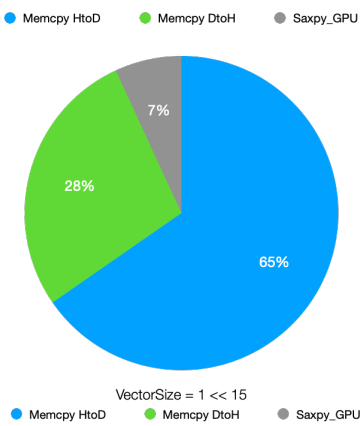
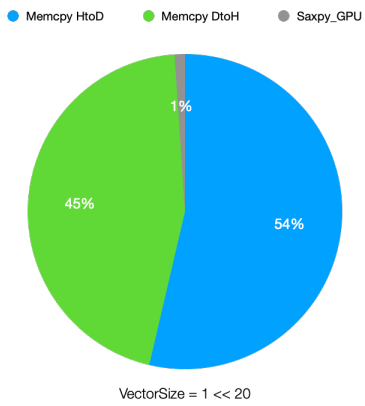
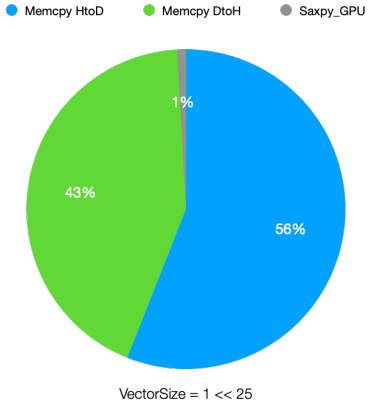
ECE 60827 LAB1 report Name: SILIN LI

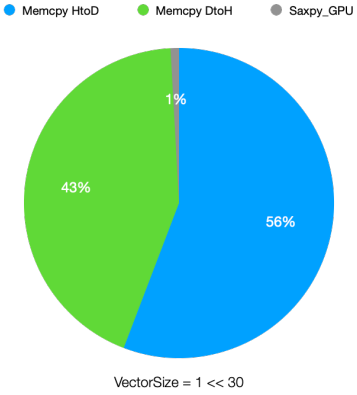
Part1: GPU SAXPY

The four graphs below illustrate the proportion of time spent on various GPU activities. (The instructions require a stacked bar chart, but due to the significant differences in vector sizes, the stacked bar chart becomes difficult to interpret. Therefore, an alternative visualization is used for better readability.)









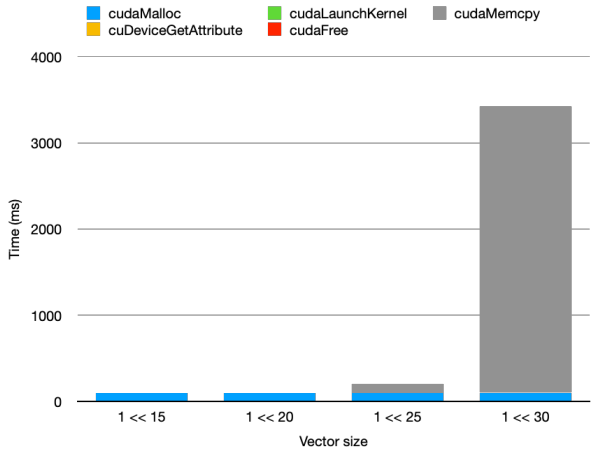
	Memcpy HtoD	Memcpy DtoH	Saxpy_GPU
1 << 15	0.0266ms	0.0113 ms	0.0028 ms
1 << 20	1.563 ms	1.319 ms	0.031 ms
1 << 25	57.14 ms	43.99 ms	0.905 ms
1 << 30	1850 ms	1436 ms	29 ms

Table1. The execution time in detail (GPU activities)

From the 4 pie charts shown above, we can see that the total execution time increased linearly with the increased size of the vector. E.g Every time the vector size increased by a factor of 32, each GPU activity proportion also increased by a factor of 32. Even though the SAXPY has a perfect nature that there is no dependency between each calculation, which means it can be processed in parallel perfectly. However, from the pie charts shown above, GPU spends more than 99% of its time on memory transactions when the vector is large (2^20 – 2^30 elements). Even when the vector size is small (2^15), the GPU still spends more than 90% of its time on memory transactions. This is because the SAXPY is purely memory bounded. When the elements are moved from host to device, each element is used only once, which means the GPU cannot leverage the power of parallel calculation. The SAXPY performance is bounded by the memory bandwidth. Even though the GPU performance can be improved by launching another kernel while SAXPY is doing the memory transaction, the absolute latency of SAXPY cannot be improved too much by GPU.

	cudaMalloc	cudaLaunchKernel	cudaMemcpy	cuDeviceGetAttribute	cudaFree
1 << 15	97.6	6.5	0.190	0.130	0.122
1 << 20	100	6.6	4.6	0.124	0.217
1 << 25	96.9	6.47	103.1	0.123	0.217
1 << 30	99.6	6.93	3316	0.152	4.6

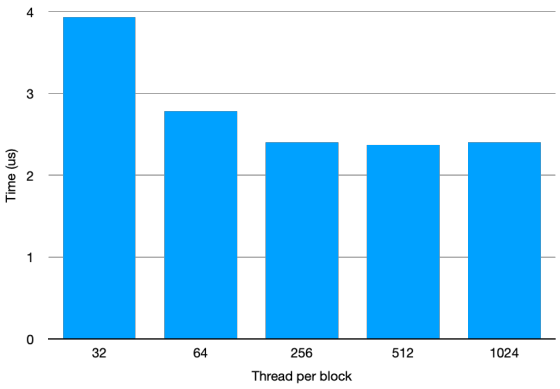
Table2. The execution time in detail (API calls)



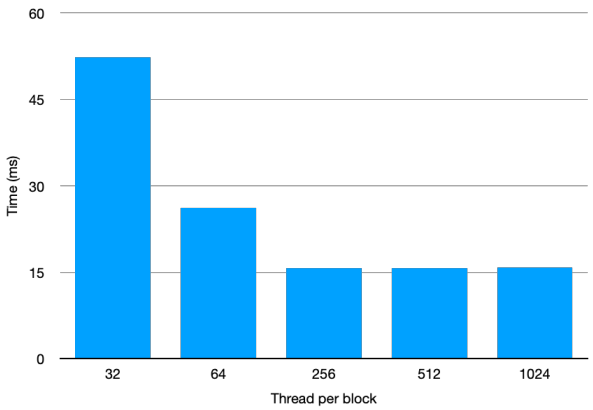
Time spend on API calls

From the stacked bar chart above, we can see that when the vector size gets larger, the “cudaMemcpu” dominate the execution time. This behavior agrees the argument we made above that SAXPY program is memory bounded. The interesting thing is that no matter how big the memory the program wants to allocate, the cudaMalloc API always takes the same amount of time. This behavior may have the relation between how CUDA finds a memory chunk in GPU and allocate.

The following graphs will show the saxpy_gpu kernel execution time under different thread per block and number of blocks settings. Since the GPU activities like DtoH memory transactions will not influenced by those parameters, the graphs will only show the saxpy_gpu kernel execution time.



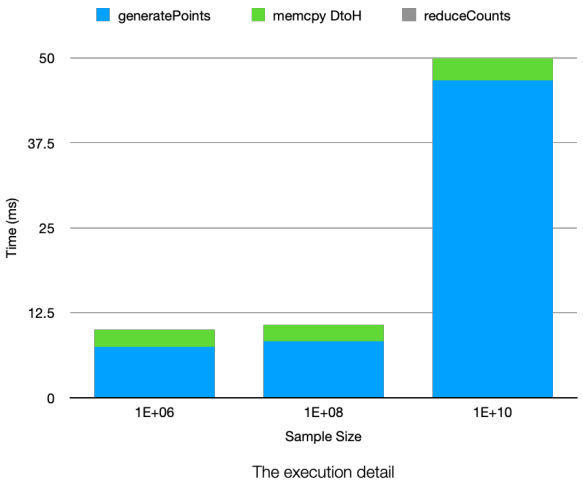
The execution time when vector size = 1<<15



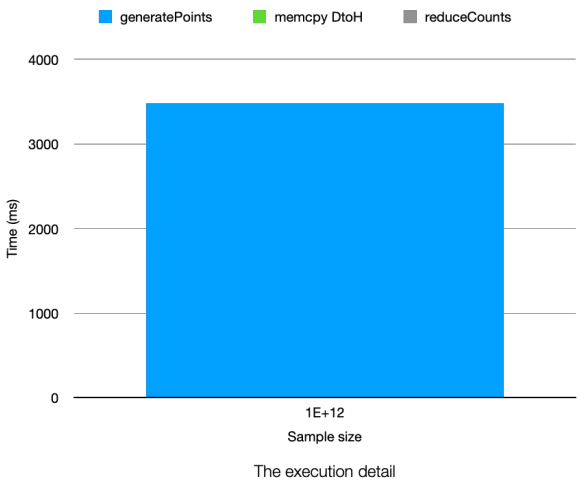
The execution time when vector size = 1<<30

The graphs above show the trend that the time consumed by the saxpy_gpu kernel will becomes lower when the thread_per_block get larger. Since in this problem, the shared memory is not needed, and even only two registers are enough for doing the calculation (The instruction requires that one thread only perform the calculation for one element). Therefore, as many threads as per SM can handle are needed to fully occupy the GPU resources. The trend becomes more obvious when the vector size gets larger. Since each thread only performs calculation for one elements, we need (vector size + thread_per_block – 1) / thread_per_block blocks to handle the calculations for all elements in the vector. However, if the thread_per_blcok is too small, there will be a high overhead on block scheduling since one GPU can only handle certain amount of blocks concurrently. The rest of blocks will sit in a queue and wait to be scheduled. This scheduling may cause a high overhead which leads a high execution time.

Part2: Monte Carlo estimation of the value of PI



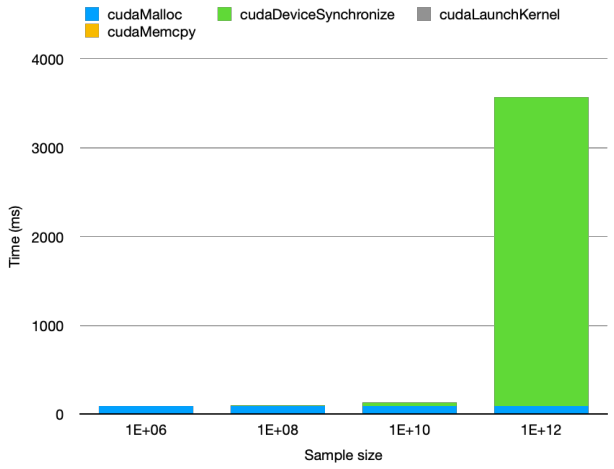
The execution detail



The execution detail

From the graphs above, we can see that unlike the memory bounded saxpy_gpu program, the Monte Carlo pi estimation program is calculation bounded. No matter how large the sample size is, the generatePoints kernel always dominates the execution time. Even with a small sample size, the generatePoints kernel still takes>70% of the execution time. This happens because the Monte Carlo program does not need too many memory transactions after executing the Reduce function. All the points can be generated inside GPU and all the calculations can be performed parallelly since there is zero dependency. After reduction, only a small amount of data needs to be copied to the CPU to do the further calculation. Even without reduction function, the memory transaction time is acceptable.

From the graphs, we can also see that when the sample size increased from 1e6 to 1e8, the execution times remained the same. I think this is because 1e6 or 1e8 samples cannot fully occupy the GPU resource, which means all the threads can be executed in parallel in one go. When the sample size gets larger, the GPU cannot run all threads at once, some threads have to wait for other threads.

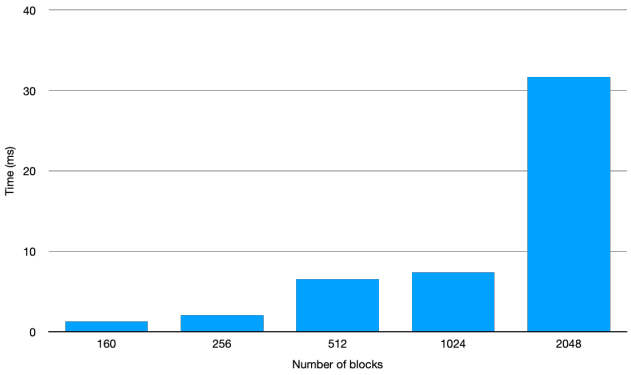


Time spend on API calls

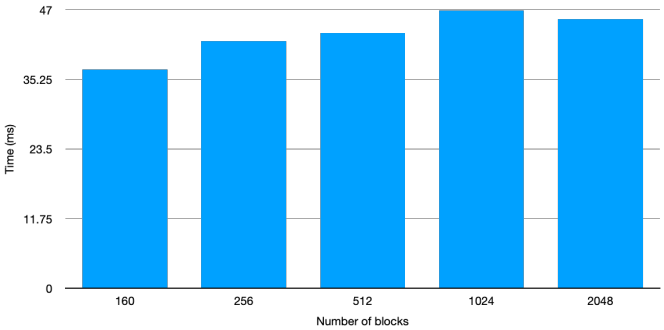
	cudaMalloc	cudaDeviceSynchronize	cudaLaunchKernel	cudaMemcpy
1E+06	90.97	7.463	5.260	4.238
1E+08	89.292	8.36	6.59	3.30
1E+10	90.153	46.71	6.084	4.216
1E+12	90.85	3478	6.33	4.25

Table3. The execution time in detail (API calls) for PI estimation

From the table and stacked bar graph above, the new observation would be that when the sample size gets larger, the time consumed on CUDA API “cudaDeviceSynchronize” grows exponentially. This behavior makes sense because there is an if statement in the Monte Carlo PI estimation kernel. After we get the multiplication result, we need to if it is greater or smaller than 1. This will cause the wrap to diverge. This may cause a different execution time of each thread, which needs to be synchronized in the end.



The execution time when sample size is 1e6



The execution time when sample size is 1e10

(The reason I only show the execution time of kernel is the same reason as before)

In my Monte Carlo π estimation program, I set the block size to 1024 threads to maximize GPU utilization. This decision follows the same reasoning as before—ensuring that the GPU's computational resources are fully utilized. Since the workload for each thread is relatively simple—generating random points, performing multiplications, and making comparisons—there is no need to consider shared memory or register usage.

From experimental results, the optimal performance is achieved when the total number of thread blocks is 160. This is because the GPU's Streaming Multiprocessors (SMs) can handle 2048 concurrent threads (as confirmed by the device query), meaning that two blocks per SM (each with 1024 threads) are required for full utilization. Given that the GPU has 80 SMs, distributing 160 thread blocks ensures efficient execution. Each thread within these blocks performs multiple calculations and increments a hit counter, maintaining high throughput.

This configuration minimizes block scheduling overhead, as evidenced by the performance degradation observed when the number of blocks exceeds 160. However, when the sample size is 1e6, and the number of thread blocks grows significantly, performance degrades exponentially. The reason for this is that when an excessive number of blocks is launched while the computation per thread remains minimal, many blocks remain idle. This happens because of the conditional check (if (threadIdx.x < sample size)), where a large portion of threads fail the condition and effectively perform no useful work. As a result, warp divergence occurs, leading to inefficient resource utilization and a sharp decline in performance.