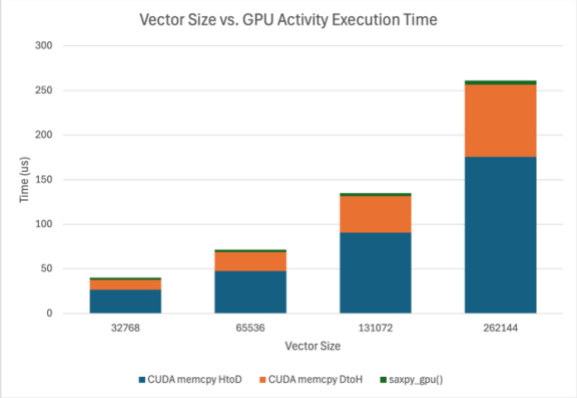
Pranav Srisankar
33469264
2/1/2024

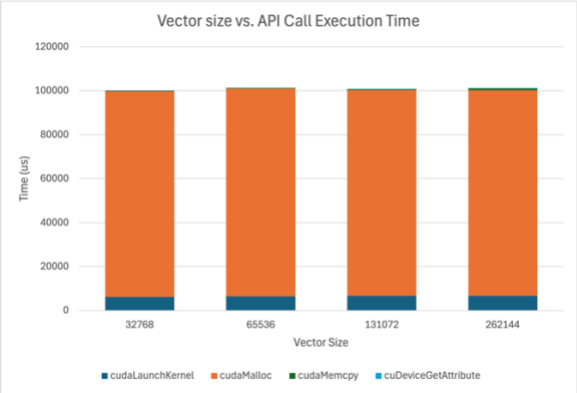
Lab 1 Report

Part A: SAXPY

Data



	GPU Activity Execution Time (us)		
Vector Size	CUDA memcpy HtoD	CUDA memcpy DtoH	saxpy_gpu()
32768	26.432	11.072	2.592
65536	47.552	21.184	2.783
131072	90.528	41.152	3.392
262144	175.615	80.832	4.545



	API Call Execution Time (us)			
Vector Size	cudaLaunchKernel	cudaMalloc	cudaMemcpy	cuDeviceGetAttribute
32768	6130.666	93656.986	192.778	140.965
65536	6375.913	94548.849	276.667	156.845
131072	6664.352	93628.665	465.203	140.656
262144	6649.152	93564.662	916.277	130.579

Pranav Srisankar
33469264
2/1/2024

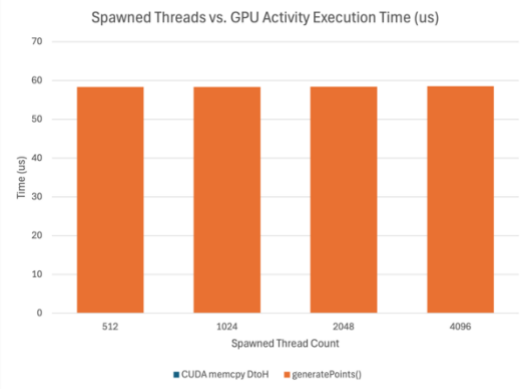
Analysis

The SAXPY kernel has a single variable to modify, vector size. The vector size dictates the size of the two vectors being operated on. In the program, the CPU first generates 2 vectors of random numbers between 0 – 100. Then, these vectors are copied to the GPU, where one is multiplied by a scale, then added to another, and saved. The resulting vector is then copied back to the CPU to be verified.

The results show that the run time is very dependent on the size of the vector, which is expected. The breakdown of the GPU activity runtime illustrates that very minimal time is spent on actual computation on vectors. Instead, majority of runtime is spent on copying data between the CPU and GPU. As two vectors are copied to and from the GPU, the runtime of GPU activities increases almost exponentially as vector size increases, as the increase in vector size results in significantly more data needed to be transferred between the host and the device.

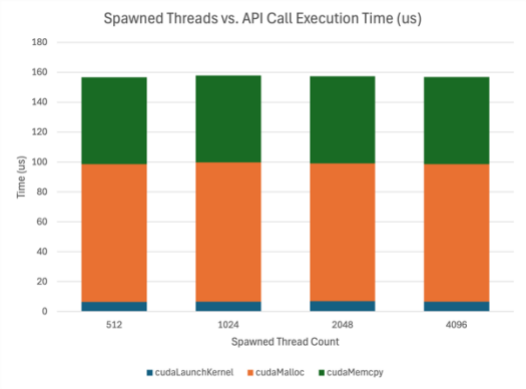
Part B: Monte Carlo Pi Approximation

Data



	GPU Activity Execution Time (us)	
Spawned Thread Count	CUDA memcpy DtoH	generatePoints()
512	0.002112	58.348116
1024	0.002272	58.326776
2048	0.002592	58.41655
4096	0.003808	58.53857

Pranav Srisankar
33469264
2/1/2024



	API Call Execution Time (us)		
Spawned Thread Count	cudaLaunchKernel	cudaMalloc	cudaMemcpy
512	6.411844	92.227617	58.143703
1024	6.58109	93.300468	58.087885
2048	6.969695	92.225016	58.187207
4096	6.609002	92.058441	58.238347

Analysis

The Monte Carlo Pi Approximation (MCPi) generates random points on a unit square, and checks if they are within a unit circle in the same square. Essentially, a random point is generated, then if its distance from (0, 0) is less than one, this is considered a “hit”. The probability of a hit is equivalent to π/4. It stands to reason that as more sample points are generated, the estimation of π becomes more and more precise and accurate.

To parallelize this on a GPU, two variables are needed, the total number of threads and the number of samples taken per thread. Each thread generates a random point and calculates the number of hits. Then, the number of hits per thread is returned to the CPU, which calculates the probability of a hit, and calculates π based on that. Increasing the number of samples taken only results in a longer run time for a single thread. So, for this reason, the variable to modify was chosen to be the total number of threads.

The sample size per thread was fixed to 1e6. Varying the total number of threads results in interesting results for MCPi. Like SAXPY, the time spent on data transfer between the CPU and GPU increases. It is important to note that there is no transfer from CPU to GPU, only the other way. However, the execution time of generatePoints(), the kernel, stays consistent, as every thread is doing the same thing. This behavior makes sense, as all threads should realistically spend the same time calculating hits. Differing from SAXPY, majority of the time is spent in the execution of the kernel, rather than the copying of

Pranav Srisankar
33469264
2/1/2024
memory. This also makes sense, as each kernel is iteratively generating a random point and checking for a hit. This results in runtime being dominated by the kernel itself.