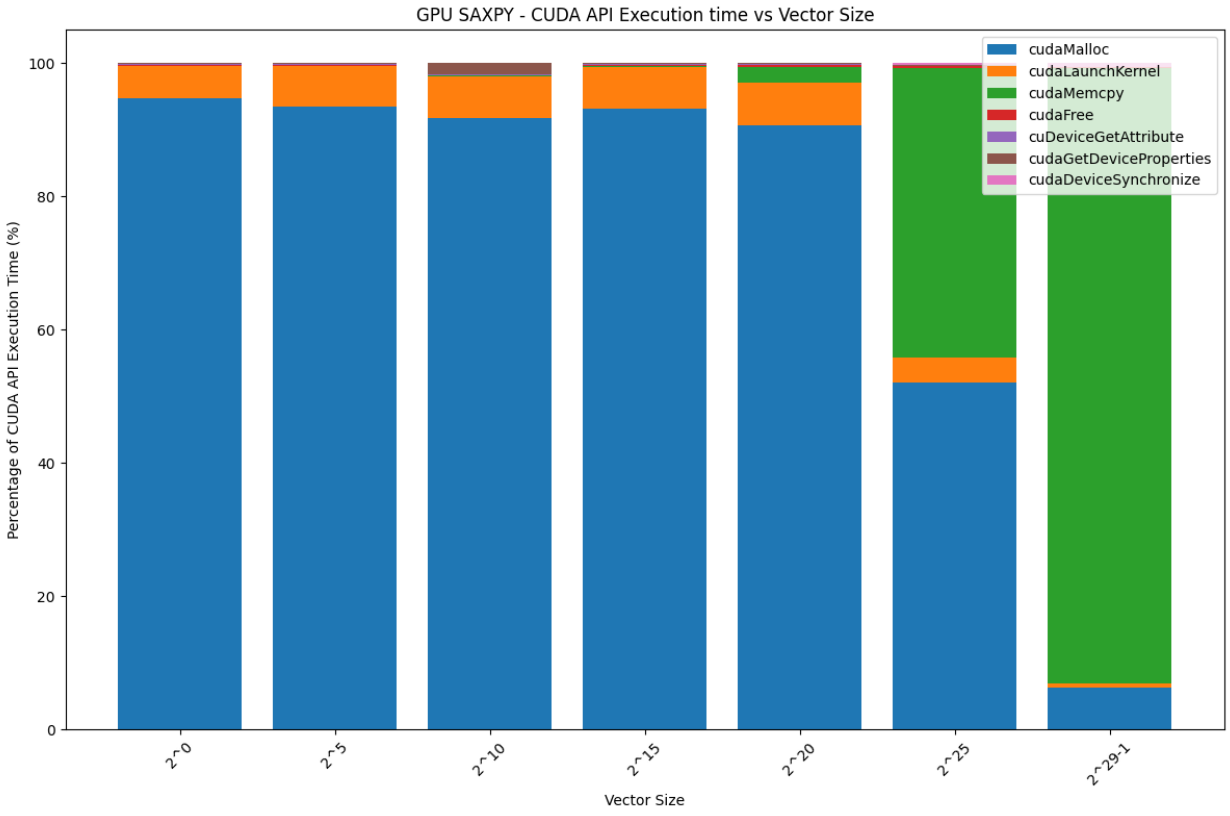
ECE60827: PA1

William Milne (GitHub ID: wmilnePurdue)

Email: milnew@purdue.edu

Part A – CUDA API Execution Time:

The test data collected by using the nvprof profiler for the "lab1" program GPU SAXPY operation with different vector sizes is illustrated below where the y-axis shows the breakdown of the execution time percentage for the main CUDA API functions.

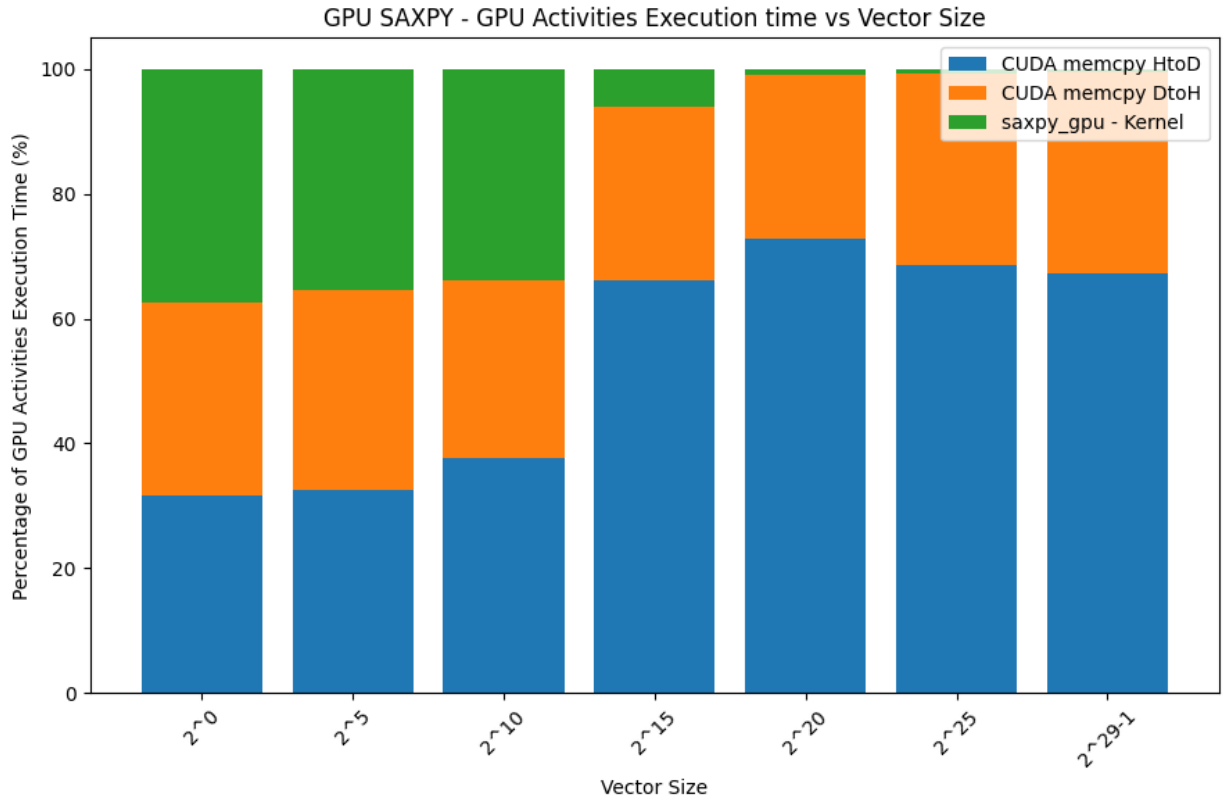


For the seven vector sizes that were tested and shown above, nearly 100% of the execution time can be linked to these API functions with the largest contributions from *cudaMalloc, cudaLaunchKernel and cudaMemcpy*. As the vector size gets large, the *cudaMalloc and cudaLaunchKernel* contributions to execution time diminish while cudaMemcpy contributions grow significantly. Although data is extremely limited, this could indicate as the vector gets larger, the overhead to the allocating memory and setting up the kernel on the GPU doesn't grow as much as the cost to transfer data between host

and device even though for small vector sizes the allocation and launching are more significant. Interestingly, the profile doesn't significantly change visually until vector size exceeds 2^20. Note: The remaining API functions listed but not mentioned yet, we're fairly consistent across these vector sizes, except for cudaDeviceSynchronize grew very quickly for the largest two sizes. With that being said, all contributions remained less than 1%.

Part A – GPU Activity Execution Time:

The same profiler also collected data on the GPU activity for the SAXPY operation which is presented in the image below

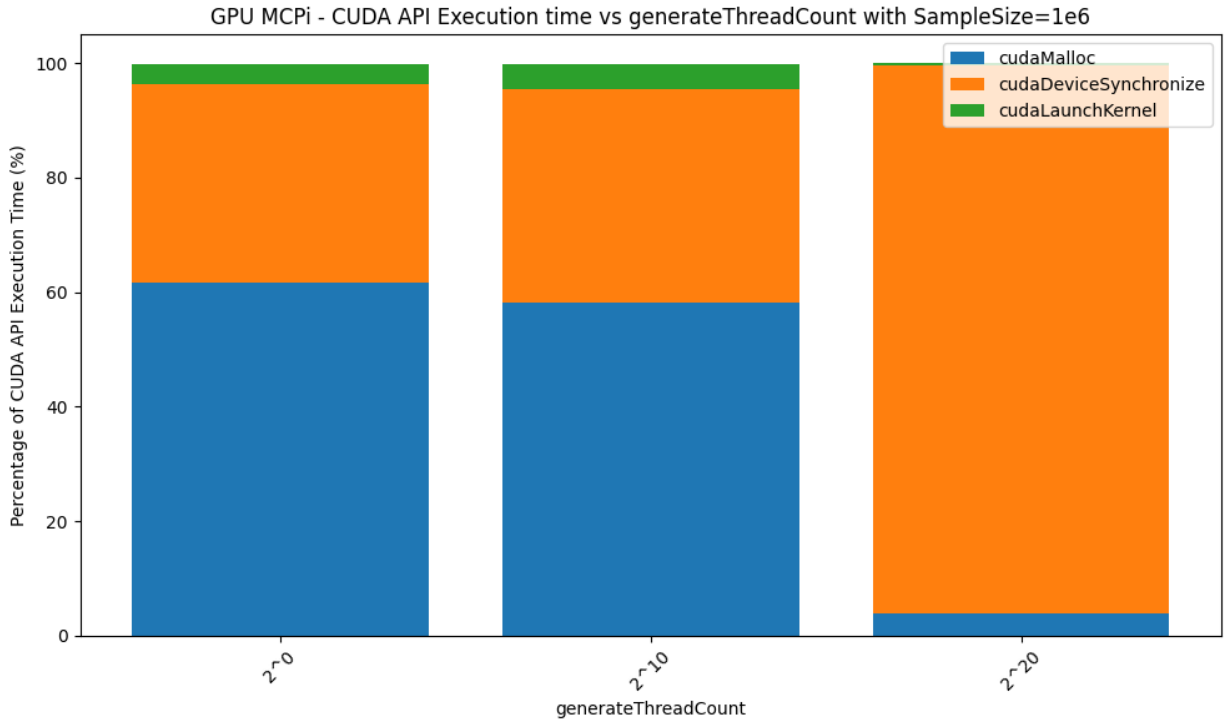


The plot above shows as vector size increases, the execution time of the *saxpy_gpu* kernel as a percentage decreases with a significant decline at 2^15 even though for smaller vector sizes it's a fairly substantial contributor. For the device to host memory copy, it is fairly stable (within 5% range for all vector sizes), but since we only copy one value from GPU to CPU and we copy two values from host to device (CPU to GPU), it's contribution to the percentage of execution time remains much lower. The graph shows transfer speed is very important for the speed of the SAXPY computation as kernel overhead diminishes.

Part B – Monte Carlo pi estimation:

For this part of the lab, consider the provided default macros as our default case to use statically when varying sample sizes and thread counts. These will be listed later. Like part a, the test data was collected during different builds/runs with the nvprof tool.

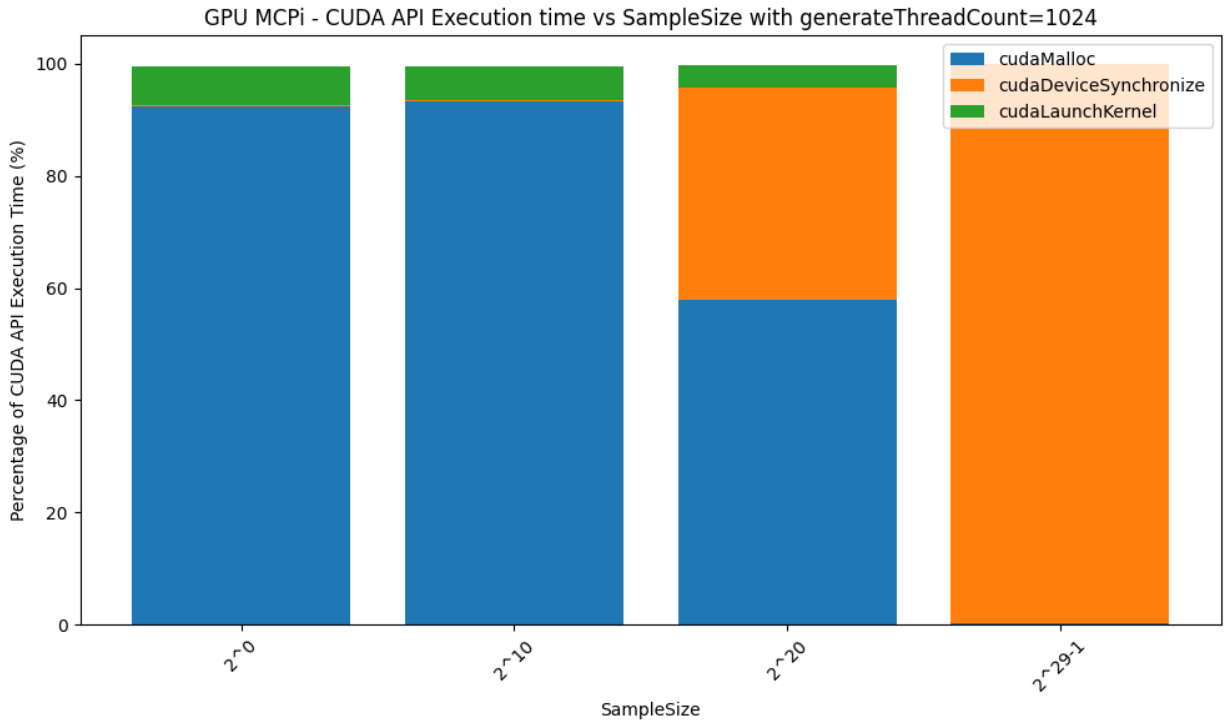
Also, I'm going to limit how many plots appear since the profile of the execution time percentages were dominated by certain contributors in both CUDA API and GPU activities case as will be seen below.



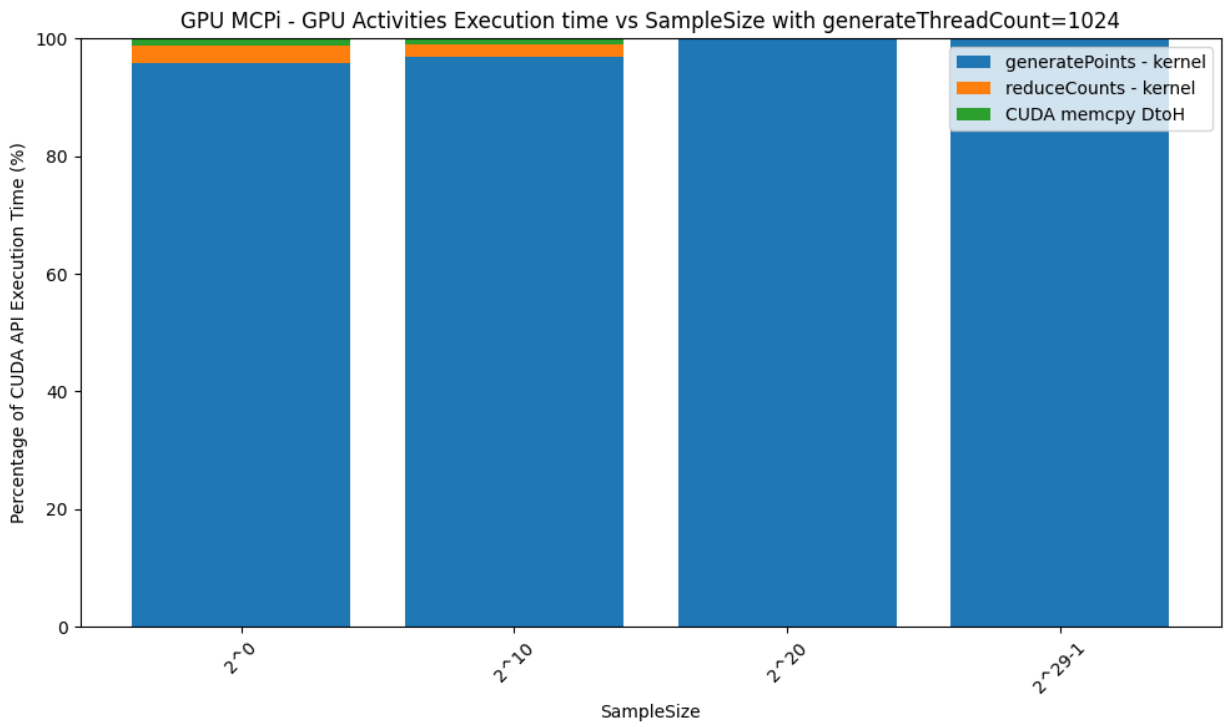
The plot above illustrates varying the *generateThreadCount* input to the *runGpuMCPi* function with the fixed SampleSize and *ReduceSize*=32. As the value for *generateThreadCount* increases, the overhead for the *cudaLaunchKernel* as a percentage of the total execution time diminishes but it is not normally large for all sizes. The overhead for cudaMalloc also dimishes, but is does contribute a larger percentage to the execution time for smaller *generateThreadCount* values. As generateThreadCount increases, the total runtime of the kernel increases so it may naturally make cudaDeviceSynchronize take longer. This also makes the proportion of time spent in cudaMalloc appear smaller.

NOTE: The GPU activity profile for a wide variety of changes (other than changing SampleSize from 1e6) for the activity described above, indicated that the *generatePoints* kernel always had >=99.99% of the execution time versus the *reduceCounts* kernel and the CUDA memcpy DtoH call.

The two plots below deal with varying SampleSize with the same fixed ReduceSize value above and a fixed *generateThreadCount* value of 1024.



A similar trend occurs when varying SampleSize as just described for varying *generateThreadCount*, so most of the same interpretation can apply. One difference is that at the extrema of the dependent variable in this case, their plots show the dominant CUDA API contributor to execution time having larger impacts.



The final plot above also shows a significant majority of execution time occurs in the *generatePoints* kernel, but in this test, there are non-neglible contributions when the SampleSize is smaller. Increasing SAMPLE_SIZE increases the computational load within the generatePoints kernel, causing its execution time to dominate the overall execution time. The overhead of other operations, like reduceCounts and the device to host memory transfer that occurs for the hit totals at worst case when no reduction is requested (which launches a relatively small number of threads (i.e. *generateThreadCount*) for the common case GPU activity (i.e. generatePoints kernel)), becomes a smaller proportion of the total time as the *generatePoints* kernel takes longer to execute with the increased SampleSize.