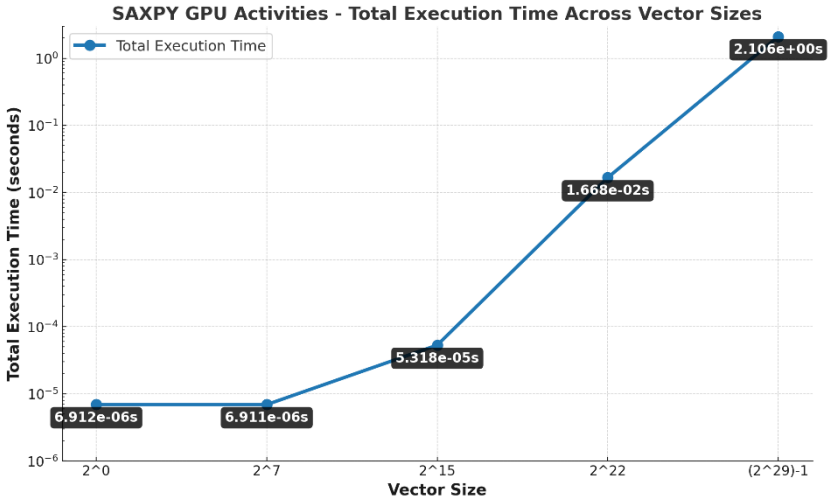
ECE60827 CUDA Programming Part 1

Ya-Han Lin (GitHub: Annie1103)

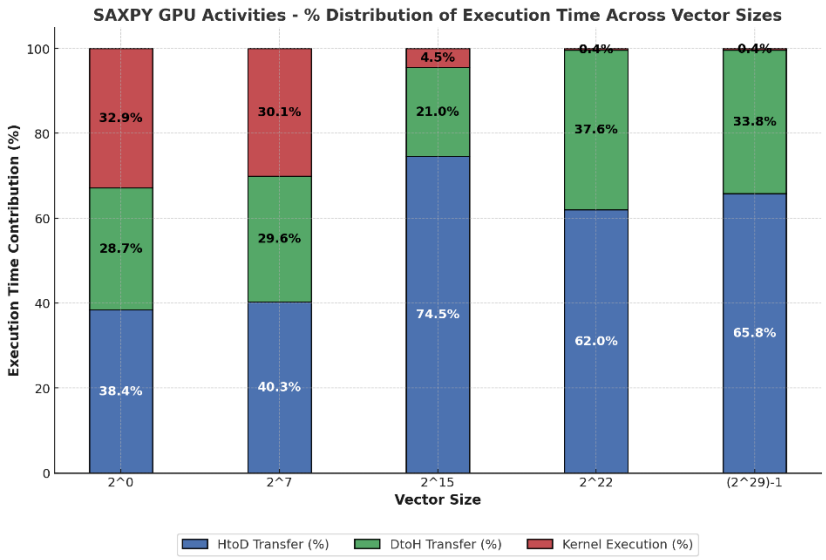
2025/2/1

Part A. GPU_SAXPY: Vector Size = [2^0, 2^7, 2^15, 2^22, (2^29-1)]

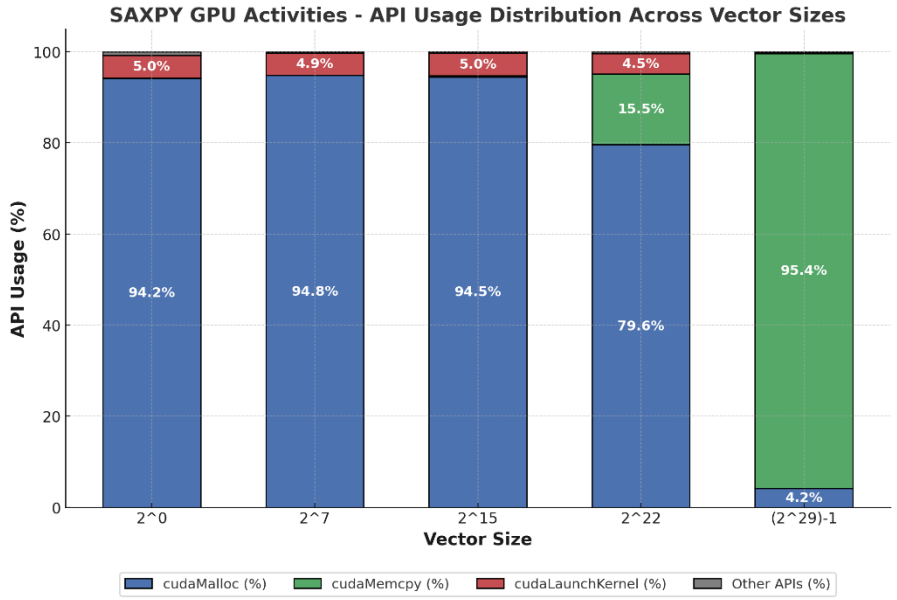
From the line graph of total execution time, we observe that as the vector size increases, the total execution time also increases significantly.



Diving deeper into the distribution of execution time, we see that Memory transfer (HtoD and DtoH) dominates execution time, especially for larger vector sizes. The kernel execution time (red section) is relatively small, indicating overall **performance is constrained by memory operations**.



Moreover, from the API usage distribution, for smaller vector sizes (2^0-2^15), cudaMalloc dominates execution time (~94%), highlighting that memory allocation overhead is significant when working with small data sizes. For extremely large vector sizes (2^29−1), cudaMemcpy consumes ~95% of the execution time, indicating that memory transfer is the primary bottleneck when working with large vectors.



Overall, **memory bandwidth is a key performance limiter** in SAXPY GPU execution.

Part B. Monte Carlo π estimation: Sample Size = [10^5, 10^6, 10^7, 10^8, 10^9]

As the sample size increases, the total execution time increases exponentially. From the execution time distribution, generatePoints() consistently consumes ~100% of execution time across all sample sizes (10^5 - 10^9), indicating that the primary computational effort is in generating random points and checking if they fall within the circle. Overall, Monte Carlo π estimation is highly **compute-bound** rather than memory-bound.