For **Challenge #13: Benchmarking Different SAXPY Problem Sizes**, I embarked on a hands-on exploration of CUDA programming to analyze how performance scales with input size. The main objective was to understand GPU execution behavior using the **SAXPY (Single-Precision A·X Plus Y)** operation, starting from the reference CUDA code provided on NVIDIA's official blog. I used Google Colab as my development platform, which offered access to a Tesla K80 GPU, making it ideal for experimentation without needing a dedicated GPU system.

Initially, I ported the SAXPY example to Google Colab, leveraging nvcc to compile .cu files directly within the environment. One of the first challenges I faced was the limited output visibility when executing the kernel across multiple matrix sizes. Although the kernel executed properly, I couldn’t see individual execution times per size. To solve this, I modified the C++ main() function to loop through increasing matrix sizes from 2152^{15} to 2252^{25}, measuring kernel execution time using std::chrono. To ensure accurate performance measurements, I surrounded the kernel call with cudaDeviceSynchronize() before and after, and printed each execution time to the terminal. Later, I also captured these values into a CSV file (saxpy\_times.csv) for easier visualization in Python.

A major obstacle came when attempting to use CuPy and Numba for kernel execution. CuPy failed due to conflicting CUDA runtime versions and missing shared libraries, while Numba produced errors in Colab due to lack of proper GPU configuration. These limitations prompted a fallback to raw CUDA C++ code compiled with nvcc, which turned out to be the most robust and reliable solution for Colab.

To visualize the benchmark results, I used Python and Matplotlib. However, the initial bar plots were misleading because the smallest input size (e.g., 2152^{15}) dominated the chart due to higher relative overhead, while larger inputs had very small execution times that were nearly invisible on a linear scale. Using GPT’s suggestion, I applied a **logarithmic Y-axis** to make performance differences more visually interpretable and, optionally, annotated the bars with exact timing values.

A particularly insightful enhancement was separating total execution time from GPU kernel execution time alone. By incorporating **cudaEvent\_t timing mechanisms**, I could distinguish time spent on memory allocations, host-device transfers, and kernel computation. This separation helped identify that for smaller sizes, memory operations dominated the timeline, while for larger problem sizes, kernel execution took precedence.

Throughout the challenge, GPT proved to be an invaluable assistant. It helped debug CuPy installation issues, provided fallback strategies using raw CUDA, guided me through writing CSV outputs in C++, and even generated Matplotlib scripts for visual comparison. Additionally, GPT helped interpret skewed plots and suggested advanced visualization fixes like log scaling and annotation.

In conclusion, this challenge deepened my understanding of CUDA programming, performance profiling, and GPU utilization patterns. It also highlighted the importance of choosing the right tools and visualization techniques to interpret benchmark data meaningfully. Most importantly, GPT significantly accelerated my learning and troubleshooting process, allowing me to successfully complete this challenge in an efficient and guided manner.