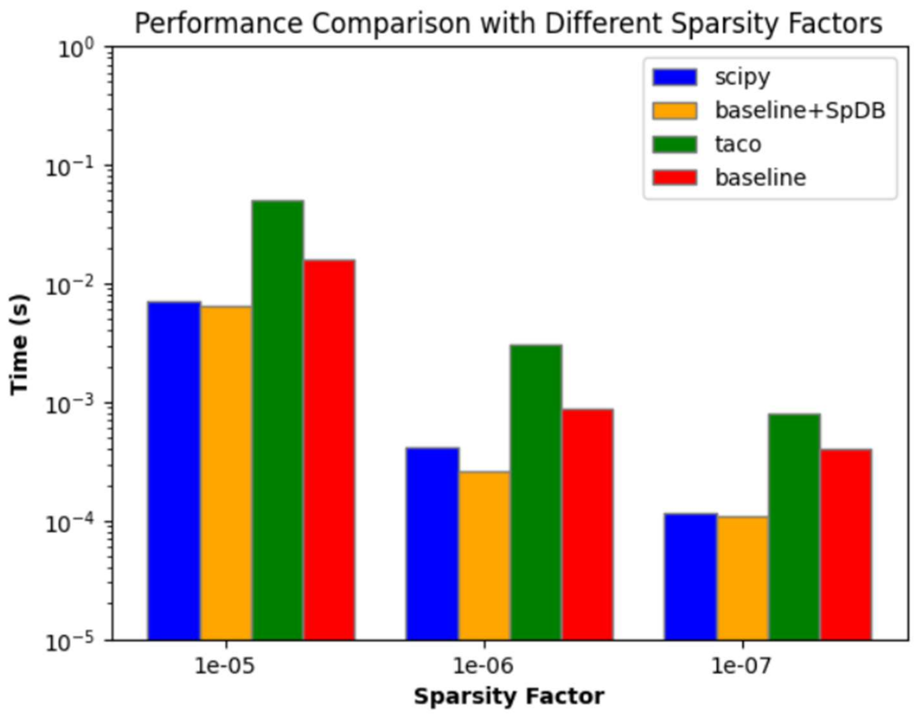
Internship Task Report:

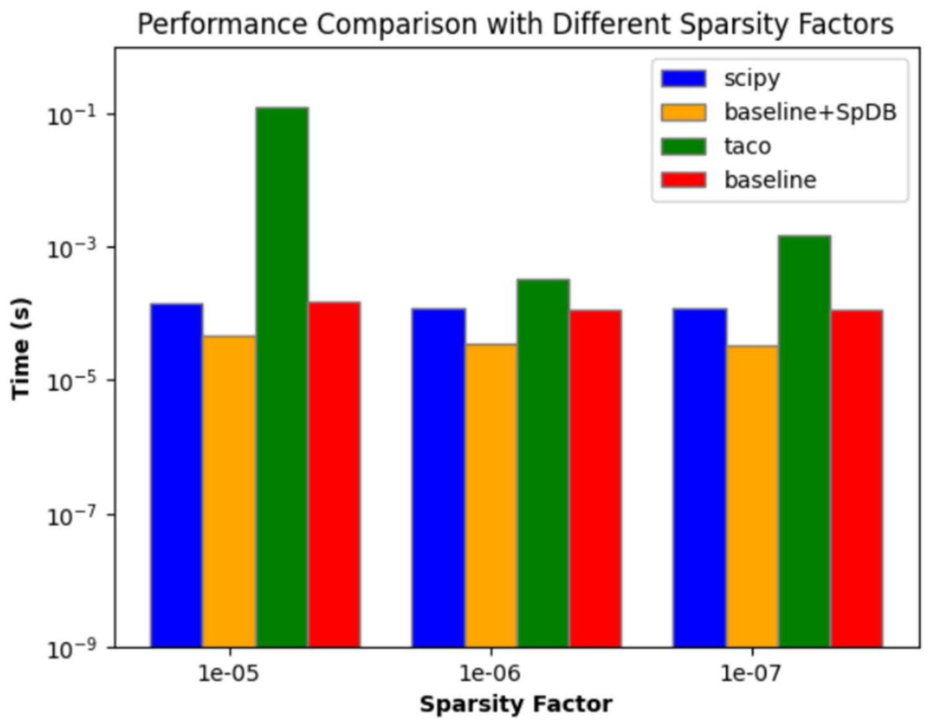
Step 0:

Before diving into coding, I conducted some research to explore the various methods for SpSpMM, ensuring that my time spent coding was invested in something worthwhile. During my investigation, I came across a survey that seemed relevant, so I decided to read it. Here is the link to the survey.

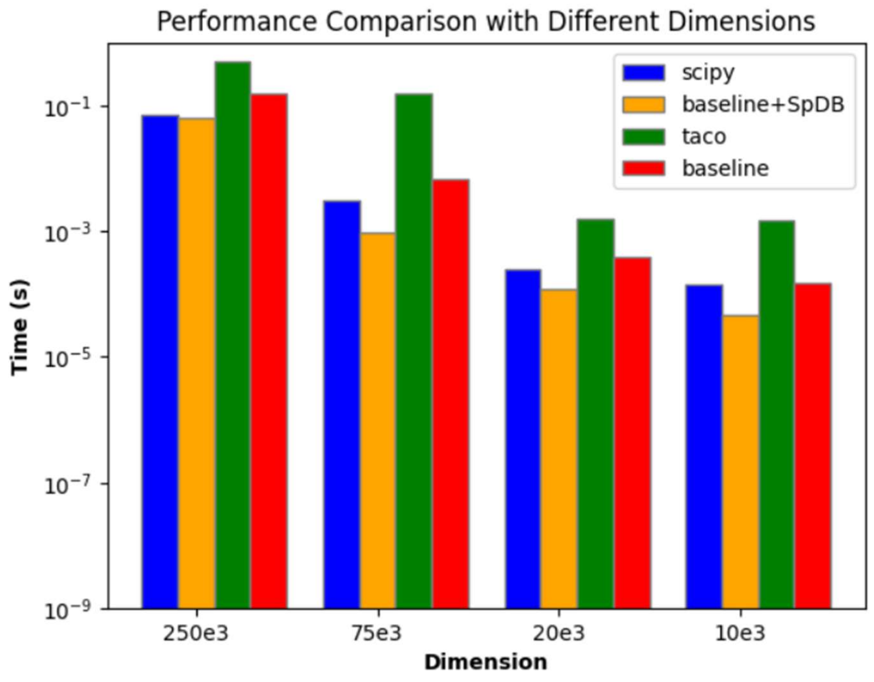
Step 1:



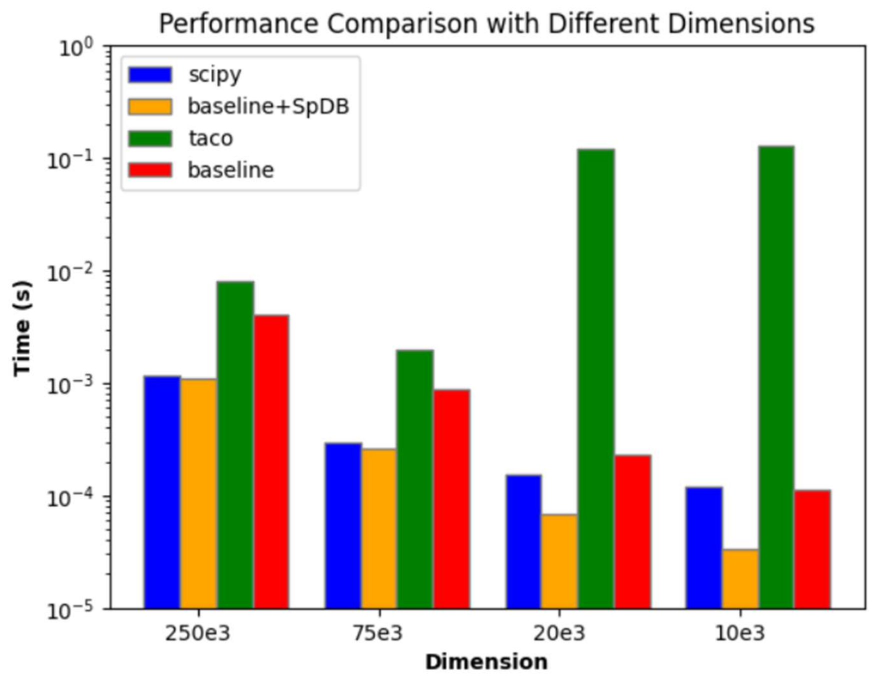
This figure describes the multiplication of two sparse matrices with dimensions 250,000 by 250,000, and varying sparsity factors.



This figure succinctly describes the visualization of two sparse matrices with dimensions 10,000 by 10,000, focusing on their differing sparsity patterns.



This figure succinctly describes a scenario involving low sparsity, characterized by a sparsity factor of 1e-05, across matrices of varying dimensions.



This figure succinctly describes a scenario involving high sparsity, characterized by a sparsity factor of 1e-07, across matrices of varying dimensions.

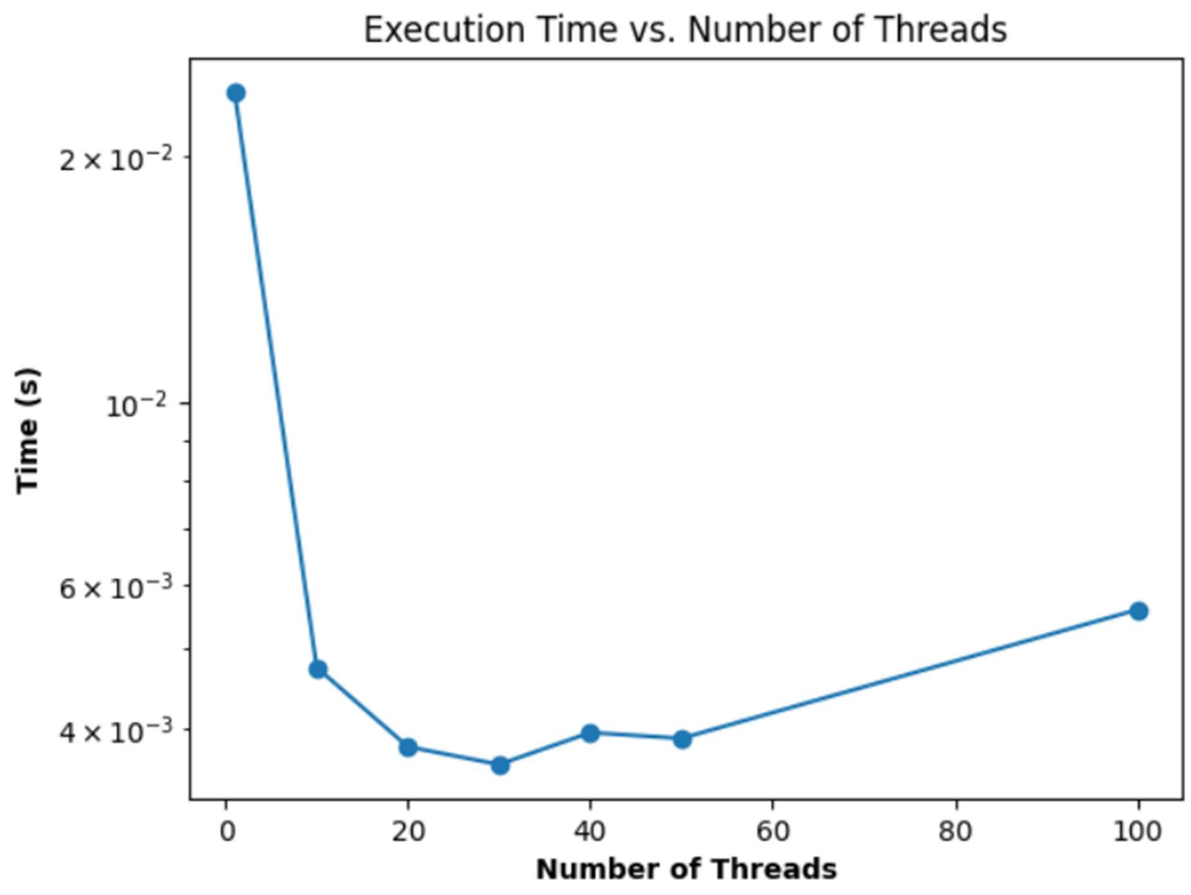
Step 2:

Comparing the latency of SpSpMM across different libraries with a baseline indicated room for improvement. To pinpoint the necessary enhancements, I deconstructed the baseline code and timed each step. Through this process, I conceived the idea of modifying the mapping structures to accelerate both the 'put' and 'get' operations. Instead of using the default hash function in boost::container::flat_map—which consumes considerable time because it is designed to accommodate various key types—our use case involves column numbers as integer keys. This allows for the use of simpler, faster hash functions, such as the modulus operator (%). To implement this, I developed a database called SpDB, specifically designed to utilize these more efficient hash functions.

There are several areas for improvement, some of which were mentioned in the survey. One aspect I wish to emphasize is the prediction of the sparse matrix size. The more accurately we can estimate the sparsity factor of the result matrix, the better the performance. This is true for my SpDB, which accounts for the sparsity factor. It can adapt to various sparsity factors by adjusting the size of its arrays. However, if the initial number of non-zero elements is close to the final count, it can achieve higher performance. To ensure a fair comparison between these libraries, I did not use any prediction strategy for the final matrix's sparsity in SpDB and set the initial sparsity equal to that of the input matrices.

One aspect I am curious about is comparing the memory consumption of these libraries. From this experiment, it appears that the trade-off between memory and computation time is significant, suggesting that exploring this further might be a beneficial next step.

Step 3:



To evaluate the efficiency of the parallelism implementation, I conducted computations using two matrices, each sized 10^5 by 10^5, with a sparsity factor of 1e-5. These matrices were multiplied to assess performance. As shown in the results, increasing the number of threads initially decreases the computation time. However, this reduction in time continues only up to a certain threshold, beyond which the overhead associated with thread management outweighs the time saved through multi-threading.

Reproducibility:

If you experience prolonged delays in matrix generation, initialization, or multiplication, or if you encounter an "Out of Memory" (OOM) error, please reduce the sample size by adjusting the settings in the generate_matrices.py file.