After developing our multithreaded matrix multiplication code, we tested its performance using different thread counts: 1, 2, 4, 8, 10, 20, 50, 100, 200, 400, and 500. Based on the matrix benchmark results, we realized that increasing the number of threads began to hinder the performance of the overall program after hitting a peak performance point. The results that we gathered on a local device with an Intel Core i5-10500H CPU 2.50GHz showed that the number of threads that had a "sweet spot" for our code was 4. When we had 4 threads, the delay was between 450-500 milliseconds, which was the best case for us overall.

One of the reasons why performance may suffer with too many threads is the fact that processors have a very limited number of available threads and cores. Too many threads with a limited number of threads/cores will lead to the processor being forced to rely on context switching, which will hinder the overall performance of our local device. Since context switching must focus on both scheduling and save/load the states of threads using overheads, the overall performance suffers.

Having to calculate and load both Matrix A and Matrix B entirely in memory will hinder the overall performance of our threads. Since Matrix B is accessed multiple times, and the desired value being sensitive to the coordinates of Matrix A, it was most practical to load Matrix B as a static value. On top of that, having to call upon an I/O operation for both Matrix A and Matrix B will also hinder the overall performance of our program. Since Matrix B will be accessed multiple times by Matrix A -the number of rows for Matrix A depends on the thread number- it is a better idea to keep Matrix B as static and load it separately in order to have a more efficient resource management.

After testing our matrix benchmark, we observed that the measured delays change. This can be related to other programs running on the system's background during execution, which may hinder the performance of specific time intervals. This may lead to small errors within the measured delay values. Since System.currentTimeMillis() will give us a rough estimation, we can't say it's a precisely reliable way of finding highly precise delays. However, it's still a valid way to find approximate delay values due to its nature.