

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size conversion chart | | | |
| Size on Graph | Square | Row (A\*B) | Column (A\*B) |
| 1 | 32x32 | (32x16)\*(16x32) | (16x32)\*(32x16) |
| 2 | 64x64 | (64x32)\*(32x64) | (32x64)\*(64x32) |
| 3 | 128x128 | (128x64)\*(64x128) | (64x128)\*(128x64) |
| 4 | 256x256 | (256x128)\*(128x256) | (128x256)\*(256x128) |
| 5 | 512x512 | (512x256)\*(256x512) | (256x512)\*(512x256) |
| 6 | 1024x1024 | (1024x512)\*(512x1024) | (512x1024)\*(1024x512) |
| 7 | 2048x2048 | (2048x1024)\*(1024x2048) | (1024x2048)\*(2048x1024) |

Problem 3

The only way to change the computational complexity from a Big O perspective is to use a different algorithm to solve the problem. The computational complexity does not change when a problem is parallelized. This can be seen in the relatively large data sizes (5, 6, and 7) of the “Matrix Multiplication Execution Time with Varying Matrix Size” graph. The execution times for the data sizes prior to these are relatively constant for the parallel implementations due to the overhead seen during parallel setup.

Problem 4

The speedup is higher for transposed matrices because of Row Major Order. Specifically, C++ retrieves data from RAM and stores it in Cache on a row-by-row basis. Thus, by iterating along rows through the second matrix, instead of along columns, the data for the matrix calculations will be in a lower cache level closer to being ready for use.