CSC 840

Matthew Wishoff

2/27/16

Project 2

Table of Contents

1.0 Abstract 3
2.0 Introduction 3
3.0 Method and Software used 4
4.0 Experimental Results and Analysis 5
4.1 Example of text file data 5
4.2 Matrix Multiplication Preprocessor Heavy Program Graphs 8
4.3 Matrix Multiplication without usage of the Preprocessor 9
4.4 Average of the difference between the programs 10
4.5 Analysis of Graphs 11
4.5.1 Three seperations 11
4.5.2 Odd memory/caching error 11
4.5.3 Increased iterations 11
4.5.4 AC power Vs battery power 13
4.5.5 Future testing 13
5.0 Conclusions 14
Appendix 15
a.) Preprocessor heavy program 15
b.) Brute force without preprocessor 19
c.) Preprocessed .i File 31

1.0 Abstract

In this project we will be looking at the efficiency of processing arrays using the preprocessor and not using the preprocessor. As people want to write more complex code it is important to maintain that code, and write it in an intelligible way. It is also important to keep in mind that you want efficient code, and when writing it in different ways you do not impact efficiency significantly or at all. In this paper we will be looking at compact code written using the preprocessor versus less compact code not using the preprocessor. The difference between these two is not subtle at all, and when looking at the code in the appendix you will immediately see the difference in maintenance, and the amount of effort it took to write each program. Doing tests on this program will dispel any notations about it being any less efficient than its counterpart.

2.0 Introduction

The question is now how are we going to compare these two programs? Well each program has timers in it that record the time it takes to evaluate matrix multiplication of different size matrices. There are 48 different ways to multiply matrices by switching and flipping indices, and both programs multiply matrices all 48 different ways. So what is the difference between these two programs? Well One uses the preprocessor, and one does not. The program that does not use the preprocessor is 600 lines of code, and would not be easy to maintain as it has 48 points of maintenance. While the program that uses the preprocessor is 180 lines of code, and only has one point of maintenance. The difference between these two programs is staggering, but we still need to measure if the preprocessor hits the performance of the program at all.

As said before we will be multiplying matrices 48 different ways by flipping and switching indices to get 48 different curves. The reason for this is we are changing the ways we access memory in these 48 different ways, and doing this will significantly impact performance and run time. As another goal of this paper is to identify the matrix multiplication pattern that produced the fastest runtimes. It's also important to note that patterns for different compilers, languages and processors will be different. This means that the same code that runs the fastest for C++ will not be the fastest for FORTRAN. Why is this? Because, FORTRAN stores and accesses memory column wise, while C++ does the same thing except row-wise. This means that one will be using random access to memory, and one will be using sequential access to memory for the same matrix multiplication pattern. And upon doing this you would see a great impact to performance, and run time of your program.

3.0 Method and Software used

I used two methods in this project. The first was a preprocessor heavy program that multiplied all 48 combinations of matrices sequentially; as well as, increasing the matrix size each iteration by 20 for all matrices from 100x100 to 500x500 see appendix a. The way it heavily used the preprocessor was with using macros, and symbol concatenation. And then calling one macro eight times each time multiplying the matrices 6 ways. Doing it this way I was able to write a program that was elegant and only 180 lines of code that multiplied a matrix 48 different ways. Another advantage of doing it this way is that there is one point of maintenance in the program, so in the future you can easily reduce or increase the number of tests quickly.

The second way I did this was the brute force way without using the preprocessor. Doing it this way I had to write the 48 ways to multiply a matrix manually See appendix b. One of the reasons this is a bad way to do it is because you create 48 points of maintenance so any future modifications are going to be difficult. However, for the purpose of this paper it was necessary to see if the preprocessor had any effect in the aspect of performance. It's also important to note that functions should not be used. Functions should not be used because functions will not decrease the performance of the matrix multiplication, but will decrease the performance of the program in general. Also even if you used functions you would not get one solid point of maintenance as you would when doing it with macros.

To perform these tests, and write the programs I used Dev-C++ a free open source IDE that is very easy to use. I also used Microsoft Visual studios to obtain the preprocessed file of the preprocessor heavy program. Both of these tests are written in C++, as in Java, Python, Ruby, and Javascript you do not have access to the preprocessor. In both of these programs after all the run times of the matrices have been added to a 2D array, I pipe the output to a comma delimited text file. This makes it very easy to read into Matlab, which is where I did all my data analysis and made all my graphs.

4.0 Experimental Results and Analysis

For my results I had them piped to a comma delimited text file so that I could easily send the data to be read by Matlab. An example of how my data would look in the text file is below. Doing it this way easily saved 15 minutes of copy pasting the data results into a Matlab matrix.

4.1 Example of text file data

0.00570687, 0.0095047, 0.0159355, 0.0239447, 0.033932, 0.0488438, 0.0638706, 0.083204, 0.107771, 0.132241, 0.168179, 0.200829, 0.245976, 0.291085, 0.341859, 0.410795, 0.471806, 0.552829, 0.640694, 0.74821, 0.902151

0.00491346, 0.00841536, 0.0135448, 0.0202046, 0.0282141, 0.0389601, 0.0519527, 0.0683769, 0.0868304, 0.105326, 0.12994, 0.159401, 0.190702, 0.224695, 0.26671, 0.310496, 0.356154, 0.406326, 0.472109, 0.535881, 0.613753

0.0051797, 0.00930708, 0.0157744, 0.0231037, 0.0334802, 0.0468136, 0.0629576, 0.0839649, 0.109314, 0.132334, 0.165522, 0.201801, 0.243791, 0.292171, 0.342692, 0.413064, 0.463188, 0.539125, 0.639697, 0.765578, 0.930419

0.0048979, 0.00828149, 0.0133898, 0.0217888, 0.0284771, 0.0389143, 0.0508453, 0.0670539, 0.0848548, 0.105873, 0.131066, 0.156672, 0.187857, 0.22697, 0.262822, 0.310031, 0.356936, 0.403503, 0.470649, 0.539553, 0.605616

0.00527098, 0.00933041, 0.0157128, 0.0254869, 0.0346265, 0.0475234, 0.0635917, 0.0852918, 0.109696, 0.137103, 0.168826, 0.202202, 0.248741, 0.291779, 0.350122, 0.415983, 0.47134, 0.543744, 0.653833, 0.784687, 0.96308

0.0051218, 0.00928707, 0.0154206, 0.0229828, 0.0327817, 0.0449468, 0.0597395, 0.0785929, 0.101373, 0.128223, 0.160551, 0.193367, 0.233429, 0.279645, 0.32877, 0.396974, 0.451521, 0.527583, 0.604348, 0.71615, 0.867847

0.00487249, 0.00853697, 0.0133245, 0.0199719, 0.0281761, 0.0391287, 0.0515783, 0.0675705, 0.0861227, 0.10755, 0.13068, 0.157907, 0.189214, 0.228494, 0.26435, 0.305456, 0.356756, 0.413879, 0.472896, 0.536143, 0.612586

0.00526043, 0.00952195, 0.0156539, 0.0232238, 0.0342269, 0.0466699, 0.0633675, 0.0830431, 0.106893, 0.132399, 0.165172, 0.203803, 0.242554, 0.288462, 0.343328, 0.408844, 0.466495, 0.552782, 0.629348, 0.73907, 0.903525

0.00520621, 0.00972542, 0.0156863, 0.0228042, 0.03352, 0.0461814, 0.0635371, 0.082936, 0.108742, 0.136155, 0.166434, 0.201998, 0.243645, 0.292606, 0.340196, 0.412553, 0.467181, 0.537252, 0.649887, 0.771636, 0.940343

0.00514515, 0.00922921, 0.0153729, 0.0224357, 0.0326632, 0.0438681, 0.0592148, 0.0774805, 0.099176, 0.125721, 0.156078, 0.189105, 0.22999, 0.274842, 0.327597, 0.386303, 0.446763, 0.51434, 0.594525, 0.710398, 0.860209

0.00536269, 0.00921509, 0.015575, 0.0232393, 0.0336617, 0.0472759, 0.0633531, 0.084453, 0.108786, 0.137012, 0.167748, 0.203199, 0.250557, 0.295329, 0.348382, 0.414891, 0.470018, 0.543505, 0.646005, 0.779842, 0.952181

0.00492167, 0.00833339, 0.0133222, 0.0203211, 0.0286334, 0.0391941, 0.0524081, 0.0662567, 0.085802, 0.105987, 0.131865, 0.15854, 0.186545, 0.222737, 0.268698, 0.310068, 0.359018, 0.411986, 0.472379, 0.537863, 0.608435

0.00544595, 0.00955707, 0.0160335, 0.0240821, 0.0354743, 0.0493574, 0.066096, 0.0867151, 0.112982, 0.138681, 0.174475, 0.21052, 0.256763, 0.301613, 0.360838, 0.433625, 0.490193, 0.561485, 0.677818, 0.803956, 0.96391

0.00495448, 0.00840299, 0.0135327, 0.0201735, 0.0284208, 0.0390457, 0.0514312, 0.0678528, 0.0883498, 0.106423, 0.130497, 0.159844, 0.19442, 0.222889, 0.264601, 0.313982, 0.359833, 0.407001, 0.469919, 0.535616, 0.630437

0.00513663, 0.00884782, 0.0145211, 0.0220612, 0.0311298, 0.0440409, 0.0587438, 0.0766336, 0.0986974, 0.122028, 0.153948, 0.187094, 0.22302, 0.269945, 0.321609, 0.380497, 0.425907, 0.501229, 0.571854, 0.680174, 0.841889

0.0048565, 0.00854845, 0.0133683, 0.0200361, 0.0279388, 0.0386634, 0.0517097, 0.0672595, 0.0846187, 0.104462, 0.129355, 0.156949, 0.190236, 0.226407, 0.26527, 0.305402, 0.354221, 0.407823, 0.4669, 0.53154, 0.601476

0.00514844, 0.0088058, 0.0143656, 0.0211702, 0.0298274, 0.0412921, 0.0551073, 0.0709602, 0.0938878, 0.116642, 0.145378, 0.173752, 0.212371, 0.261291, 0.303082, 0.358582, 0.410921, 0.481008, 0.552439, 0.652014, 0.795274

0.00536154, 0.00958824, 0.0161159, 0.0237681, 0.0347102, 0.0489299, 0.0652483, 0.0856585, 0.113191, 0.139141, 0.17279, 0.208778, 0.253679, 0.301979, 0.356257, 0.423753, 0.480808, 0.55141, 0.675466, 0.792886, 0.962777

0.00503399, 0.00887408, 0.0144491, 0.0213396, 0.0307141, 0.0417928, 0.056055, 0.0735026, 0.0952483, 0.119914, 0.146734, 0.180903, 0.21763, 0.261703, 0.303557, 0.367042, 0.421662, 0.488588, 0.568115, 0.672129, 0.812537

0.00530385, 0.00953795, 0.0154817, 0.0235019, 0.0343032, 0.0474445, 0.0637725, 0.0829779, 0.106629, 0.134944, 0.163013, 0.199106, 0.240907, 0.294191, 0.344209, 0.409963, 0.46494, 0.541598, 0.633276, 0.755458, 0.896397

0.00504824, 0.0089199, 0.0146801, 0.0220739, 0.0319692, 0.044687, 0.0580095, 0.076936, 0.100156, 0.124156, 0.154673, 0.187113, 0.226238, 0.271603, 0.316304, 0.375106, 0.441558, 0.499073, 0.581024, 0.683371, 0.842702

0.00510578, 0.0091561, 0.0151373, 0.0228039, 0.0322623, 0.0437838, 0.0580884, 0.0775448, 0.0987536, 0.125001, 0.156967, 0.189569, 0.23609, 0.27472, 0.322984, 0.383003, 0.452958, 0.517791, 0.593941, 0.71128, 0.853236

0.00501428, 0.00885149, 0.0139716, 0.021194, 0.0301445, 0.0415275, 0.0556797, 0.0724094, 0.0930122, 0.116784, 0.146809, 0.171662, 0.217849, 0.257439, 0.299557, 0.39035, 0.41025, 0.476613, 0.548715, 0.654196, 0.801093

0.00505707, 0.00891021, 0.0150111, 0.0226461, 0.0326214, 0.0455015, 0.0613752, 0.0791768, 0.101728, 0.12785, 0.159746, 0.191364, 0.230753, 0.276877, 0.330507, 0.385174, 0.442635, 0.515385, 0.5993, 0.702579, 0.861044

0.00533224, 0.00937623, 0.0160127, 0.0234685, 0.0341775, 0.0480265, 0.0637886, 0.0850678, 0.107629, 0.133004, 0.166379, 0.20455, 0.249472, 0.294905, 0.344074, 0.411888, 0.467732, 0.546786, 0.641634, 0.746835, 0.908194

0.00513546, 0.00872715, 0.0142104, 0.0208828, 0.029731, 0.0416964, 0.0560542, 0.0716224, 0.0922585, 0.117333, 0.145749, 0.173731, 0.21572, 0.256446, 0.296359, 0.357408, 0.409105, 0.475708, 0.544809, 0.656331, 0.798182

0.00499983, 0.00878839, 0.0145465, 0.0214197, 0.0302073, 0.0420488, 0.0561203, 0.073363, 0.0947497, 0.118866, 0.147182, 0.180147, 0.217579, 0.264939, 0.303683, 0.366812, 0.41959, 0.491042, 0.565159, 0.668933, 0.81155

0.00518238, 0.00898403, 0.0144021, 0.0221588, 0.0317364, 0.0436396, 0.0588868, 0.0781466, 0.0987413, 0.124497, 0.153157, 0.187377, 0.228125, 0.269866, 0.31929, 0.376813, 0.433674, 0.506998, 0.576084, 0.686349, 0.837324

0.00512485, 0.00906658, 0.0148897, 0.0226911, 0.0329435, 0.0446779, 0.0609385, 0.081021, 0.100353, 0.128054, 0.156726, 0.192961, 0.232915, 0.281647, 0.327079, 0.388905, 0.439721, 0.513779, 0.602354, 0.707742, 0.871121

0.00522788, 0.00930871, 0.015334, 0.0229576, 0.0330377, 0.0450072, 0.0600819, 0.078418, 0.100306, 0.125783, 0.156985, 0.193397, 0.232572, 0.276175, 0.332923, 0.39195, 0.450357, 0.5274, 0.602178, 0.714889, 0.870019

0.00497944, 0.00854802, 0.0134882, 0.020003, 0.0290444, 0.0398994, 0.0519004, 0.0663219, 0.0840989, 0.106633, 0.130975, 0.159995, 0.190519, 0.227078, 0.26838, 0.311665, 0.360266, 0.413945, 0.468725, 0.535248, 0.61286

0.00552672, 0.00993431, 0.0162387, 0.0244193, 0.0360016, 0.0506208, 0.0677695, 0.0900959, 0.116756, 0.142134, 0.180765, 0.217945, 0.265291, 0.315533, 0.369126, 0.46318, 0.507112, 0.57942, 0.690205, 0.830175, 1.02094

0.00503149, 0.00890357, 0.0145638, 0.0215086, 0.0302273, 0.0423317, 0.0562375, 0.0736349, 0.0941413, 0.118353, 0.147736, 0.176395, 0.217463, 0.264629, 0.309063, 0.363612, 0.419721, 0.49136, 0.566054, 0.674535, 0.815194

0.00549782, 0.00975138, 0.0162454, 0.0242148, 0.0351822, 0.0488605, 0.0658486, 0.0880631, 0.113243, 0.138831, 0.172786, 0.21235, 0.260799, 0.307327, 0.358405, 0.433878, 0.496694, 0.564662, 0.670181, 0.811859, 0.992968

0.00510405, 0.00915179, 0.0149885, 0.0224222, 0.0322845, 0.045013, 0.0602359, 0.081573, 0.101332, 0.129589, 0.155875, 0.191452, 0.233459, 0.287058, 0.324708, 0.388513, 0.447333, 0.509419, 0.601156, 0.702001, 0.864773

0.00486289, 0.00840904, 0.013594, 0.0200649, 0.0288187, 0.0392526, 0.0510456, 0.0682151, 0.0881099, 0.106831, 0.129269, 0.15875, 0.190444, 0.23158, 0.263712, 0.305264, 0.356313, 0.422219, 0.475132, 0.539267, 0.617836

0.00538965, 0.00977313, 0.01625, 0.0242045, 0.0350849, 0.0494989, 0.0659869, 0.0878229, 0.113135, 0.139726, 0.171468, 0.21177, 0.25364, 0.3045, 0.372658, 0.436497, 0.488188, 0.564154, 0.680346, 0.797945, 0.968852

0.00511879, 0.00885675, 0.0143337, 0.0211508, 0.0299699, 0.0413349, 0.0551971, 0.0715831, 0.0965344, 0.115705, 0.144314, 0.174987, 0.212936, 0.26197, 0.310792, 0.353283, 0.410339, 0.494556, 0.546112, 0.652765, 0.804441

0.00479294, 0.00841365, 0.0134323, 0.0197263, 0.0279799, 0.0386642, 0.0508943, 0.066146, 0.0852271, 0.104026, 0.144849, 0.157694, 0.190422, 0.224517, 0.263907, 0.307658, 0.358257, 0.406538, 0.464246, 0.533101, 0.599211

0.00501548, 0.00899198, 0.0145813, 0.0217669, 0.0313836, 0.0437658, 0.058278, 0.075559, 0.0988883, 0.121729, 0.156993, 0.185739, 0.226492, 0.272527, 0.318682, 0.377972, 0.428288, 0.500494, 0.582712, 0.681529, 0.838004

0.00488526, 0.00842668, 0.0134874, 0.019949, 0.0281821, 0.0387121, 0.051227, 0.0670486, 0.0864687, 0.105241, 0.131639, 0.15961, 0.191429, 0.228529, 0.26418, 0.309028, 0.361664, 0.411482, 0.470815, 0.539281, 0.609777

0.00542119, 0.00974124, 0.0160744, 0.0234553, 0.0348027, 0.0487092, 0.066662, 0.0852399, 0.112812, 0.137643, 0.17258, 0.208374, 0.25093, 0.308073, 0.355995, 0.426182, 0.480473, 0.556593, 0.654509, 0.811325, 0.949315

0.00503587, 0.00874184, 0.0145613, 0.0210209, 0.030737, 0.0422362, 0.0559742, 0.0733484, 0.0931298, 0.119614, 0.147833, 0.17938, 0.218512, 0.263328, 0.311333, 0.367915, 0.425681, 0.492518, 0.562722, 0.666235, 0.820877

0.0055646, 0.00989381, 0.0164677, 0.0240873, 0.0360145, 0.0501699, 0.0679845, 0.0914458, 0.115071, 0.144113, 0.177355, 0.226113, 0.268286, 0.316213, 0.372004, 0.447658, 0.502048, 0.584032, 0.698583, 0.831297, 1.03238

0.00493958, 0.00841119, 0.0134133, 0.0197226, 0.028542, 0.039076, 0.0552933, 0.0662926, 0.085894, 0.105053, 0.1311, 0.157946, 0.190893, 0.226474, 0.264431, 0.3105, 0.355776, 0.408628, 0.469275, 0.526549, 0.600619

0.00553542, 0.00964626, 0.0162317, 0.023663, 0.0357021, 0.0489777, 0.0675006, 0.0873415, 0.114769, 0.13744, 0.174634, 0.213153, 0.257074, 0.305136, 0.360779, 0.438871, 0.495215, 0.567306, 0.668016, 0.810341, 0.98173

0.00492117, 0.00842135, 0.0136231, 0.0196325, 0.0282671, 0.0394794, 0.0515249, 0.0682829, 0.0845241, 0.10697, 0.130327, 0.160042, 0.190659, 0.221483, 0.265901, 0.307084, 0.356338, 0.411471, 0.470377, 0.535585, 0.60587

0.00505318, 0.00910521, 0.0148748, 0.0219293, 0.0331668, 0.0449289, 0.0594251, 0.0786514, 0.101979, 0.126764, 0.159233, 0.19205, 0.230312, 0.27714, 0.32615, 0.383765, 0.443957, 0.512808, 0.594165, 0.704776, 0.851714

4.2 Matrix Multiplication Preprocessor Heavy Program Graphs

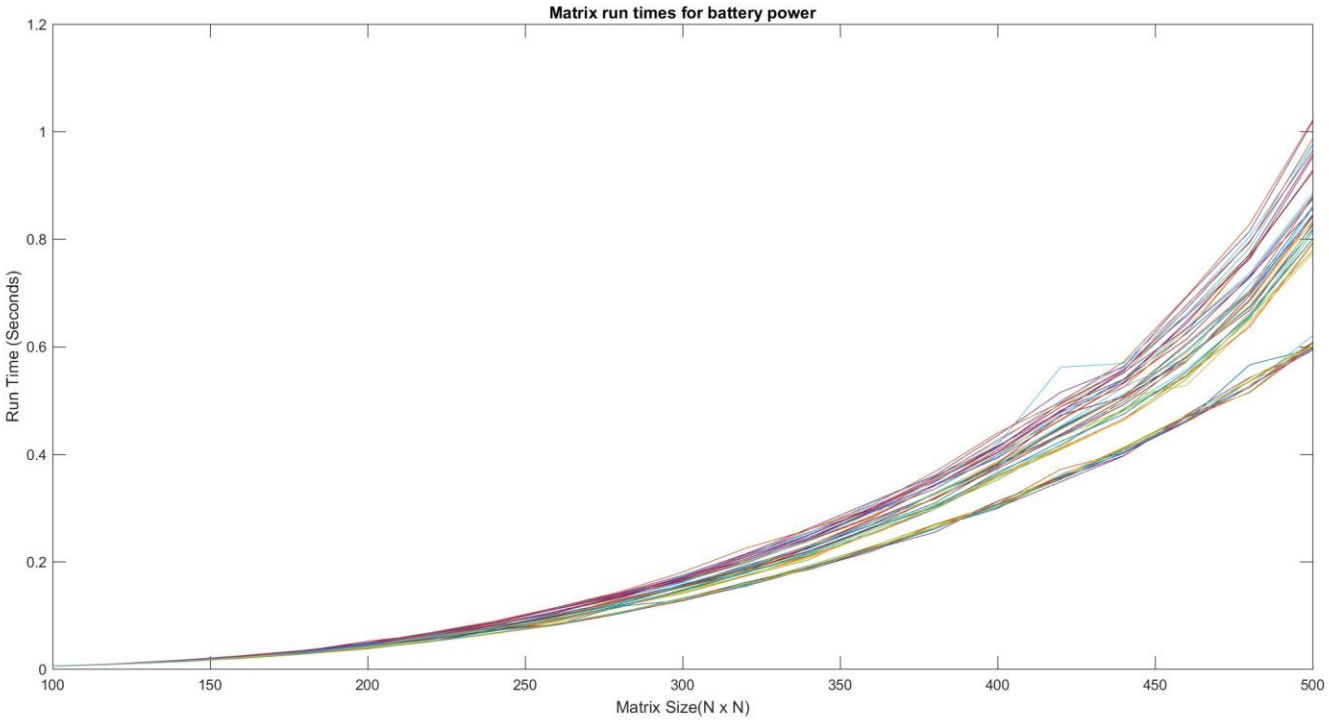

Figure 1

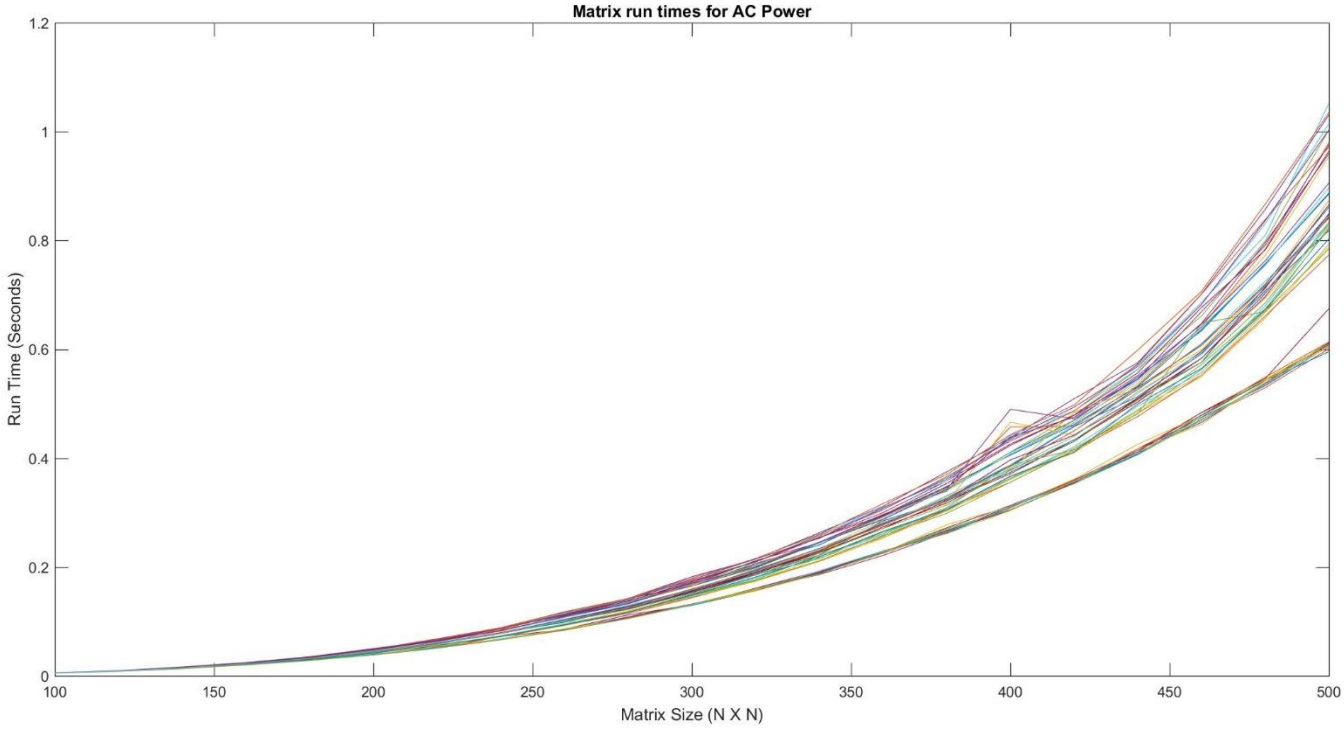

Figure 2

4.3 Matrix Multiplication without usage of the Preprocessor

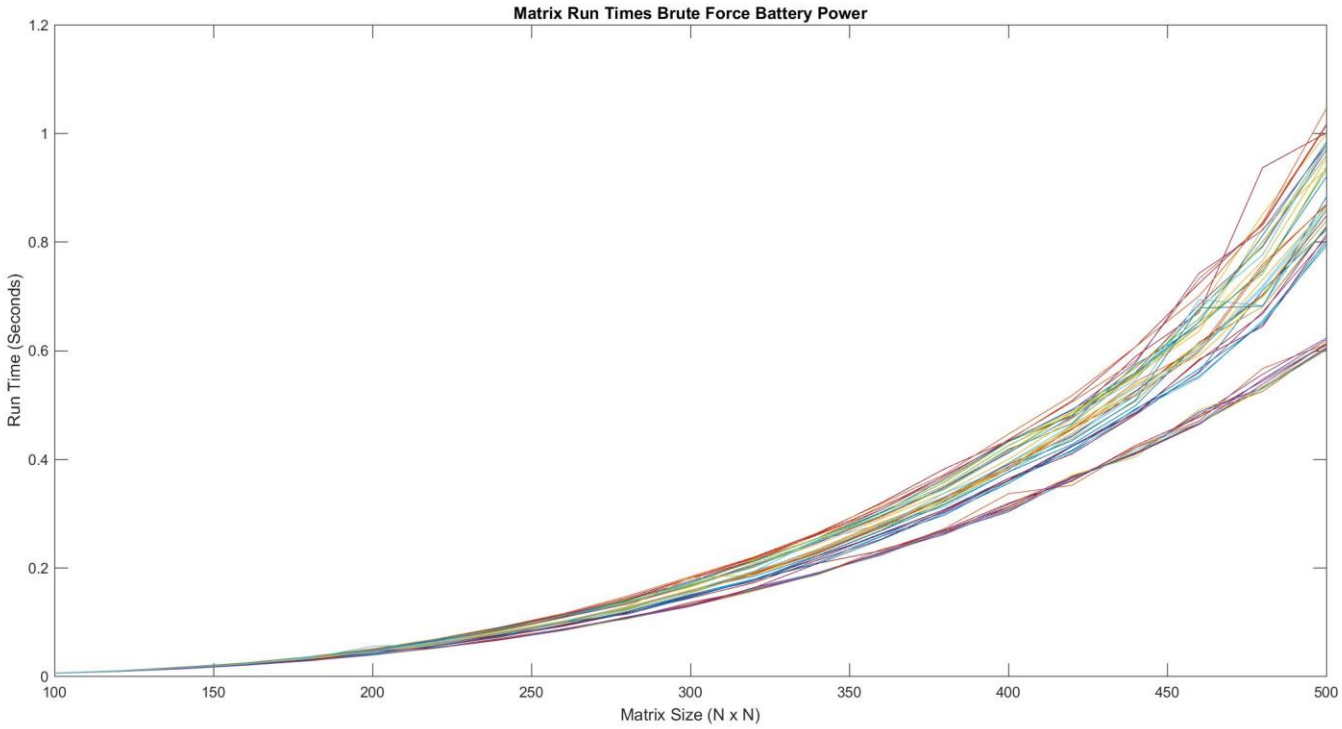

Figure 3

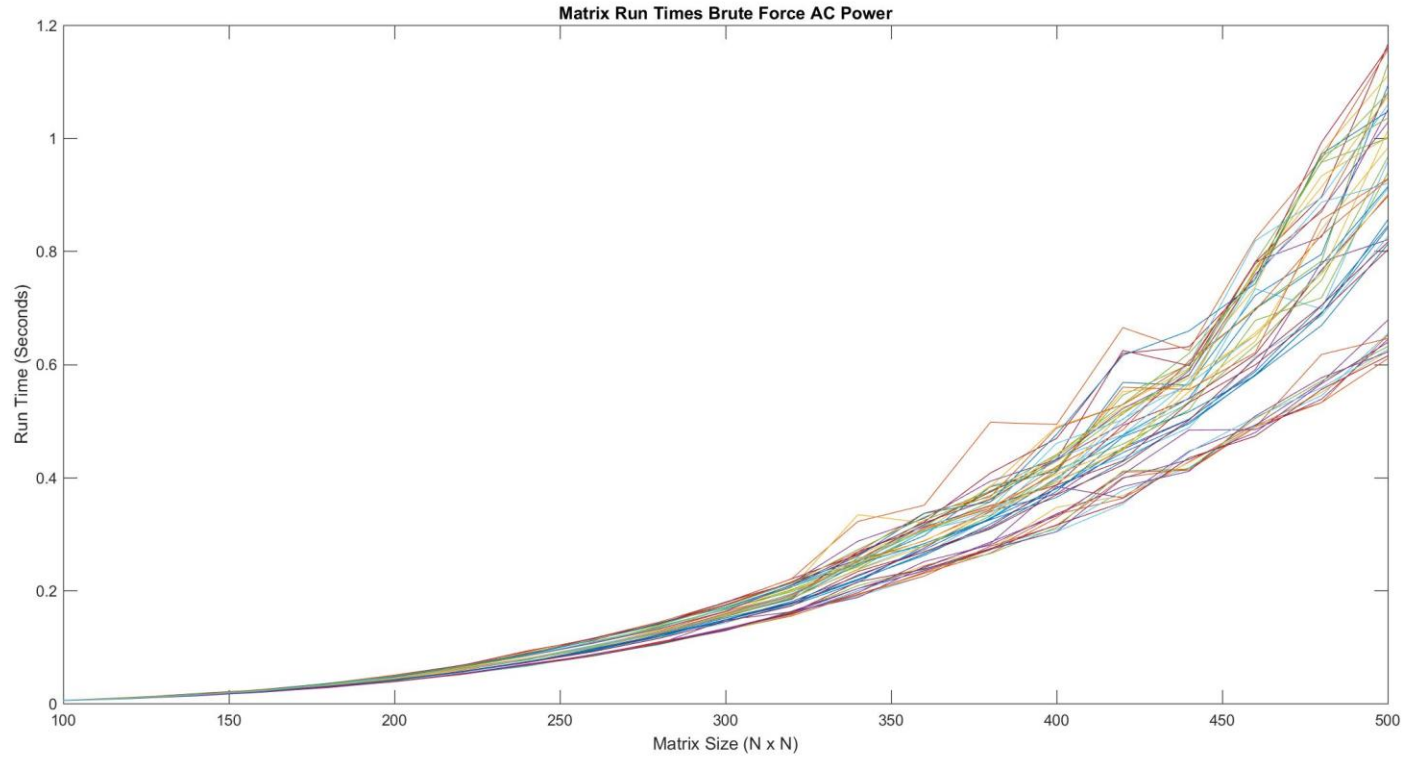

Figure 4

4.4 Average of the difference between the programs

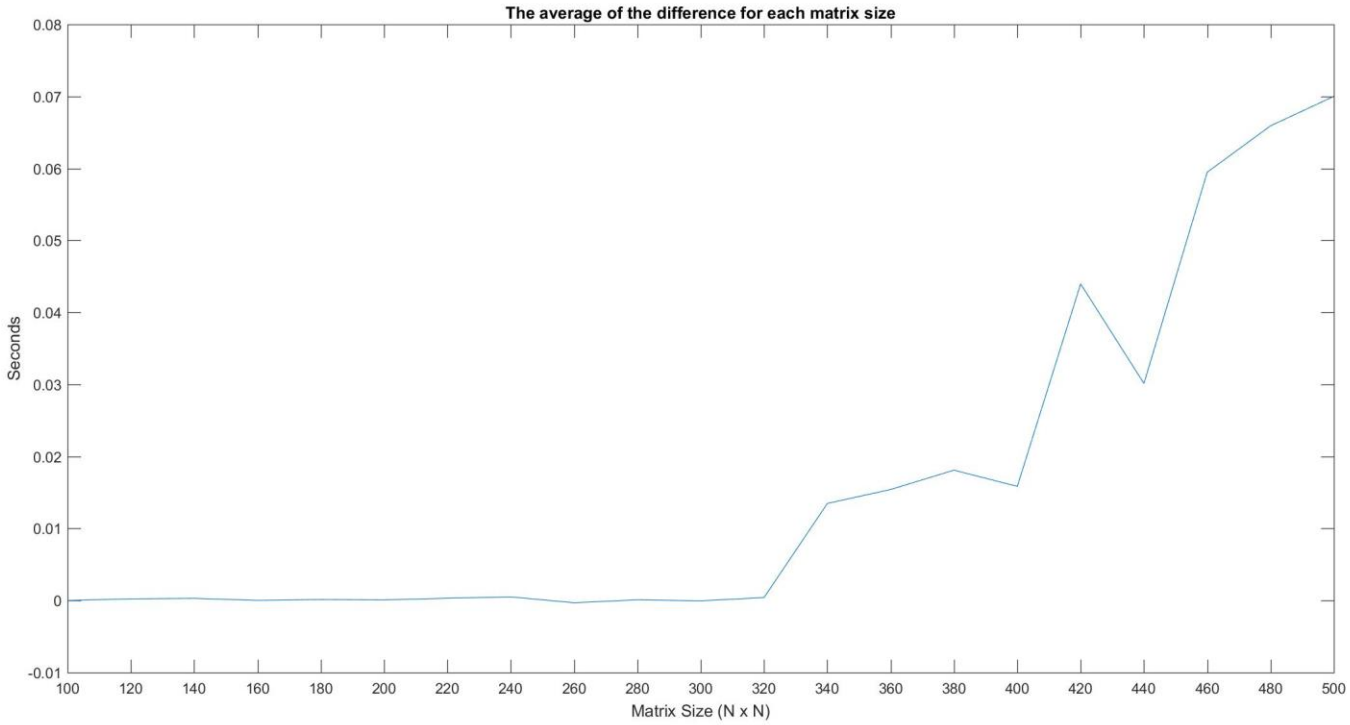

Figure 5

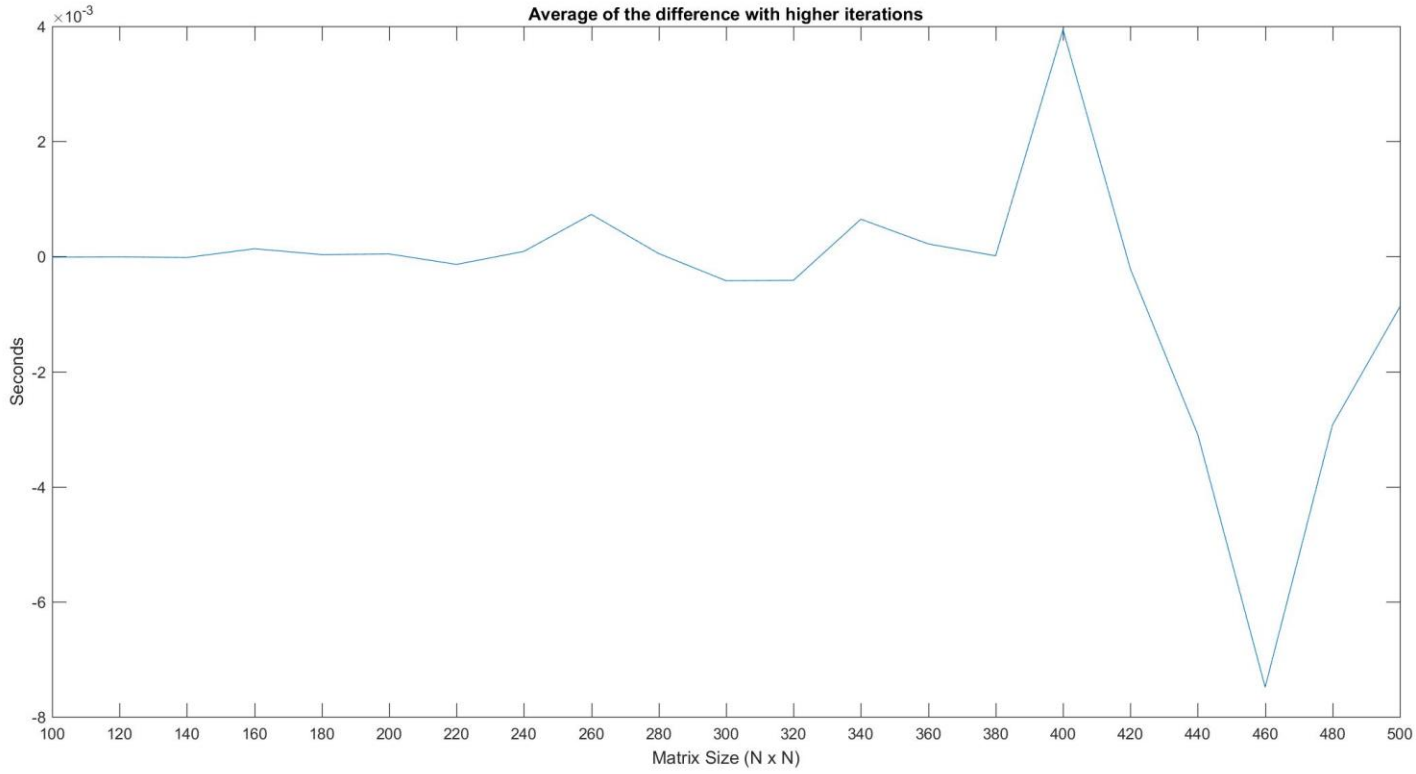

Figure 6

4.5 Analysis of Graphs

4.5.1 Three seperations

For figures 1 and 2 these are the preprocessor heavy programs, and how they performed when running the matrix multiplication tests. You can clearly see the fastest group of the three close to 0.6 seconds for 500x500 matrices. The fastest group has the inner most loop being the J indice. The last two groups are a bit merged together, but when 500x500 matrices are run they separate a little more so that you can see the serperation. The mid group of curves is the k indice, and finally the worst performing is the i indice being the inner most loop.

The reason for these loops having significantly different performance is due to the way they access memory. When the inner most loop is i memory access is column wise. This is a problem because when the matrice is stored in memory it is stored row wise. So the way the loop is accessing memory is with random access which is known to be slower than sequential access. The inner most loop being j must use sequential access, because it has the fastest run times of all 6 permutataions of the for loop indices for matrix multiplication.

4.5.2 Odd memory/caching error

An odd error that I see in figures 1 2 and 4 is that at around the matrix size of 400x400 there is a spike, and then returning to normal. Normally I wouldn't make anything of this spike it could be due to a clock issue like we talked about in class; although, the reason I think it is not is because it consistently happens at around 400x400 matrices. I talked to Sam Gluss and he was having the same kind of spike at around 440x400 matrices. It is because of these reasons why I feel it may be a caching issue, or possibly a memory issue. Where the cache is emptied and refilled with the information I am currently using. I believe with higher iterations of the matrix multiplication these kinds of errors will eventually disappear. However, the disadvantage to this is that the program will run for a significantly longer time than it would when running lower iterations for larger matrices. The reason why Sams spike is happening later in his tests might be attributed to having differeent hardware.

4.5.3 Increased iterations

For figures 1, 2, and 3 I made the base number of iterations 5. Normally the formula I used to reduce the number of iterations as the matrix size got larger was (NMax^3) / (NMin^3). The issue with this is that the curve does not look very smooth for very large matrices. Looking at figure 5 we see that the amount of error starts to increase drastically after the 300 x 300 matrices. Taking this into account I calculated (500^3) / (300^3) to be 4.63, and since it is stored as an integer it runs 4 times. The matrices after this run less than 4 times, so I made the base number of iterations 4. This allowed figures 1, 2, and 3 to have a smoother curve when compared to figure 4.

Taking into account what I did to ensure there was a good curve with little errors we can look at figures 5 and 6. With figure 5 that was when the number of interations had no base limit. For example with 100x100 matrices we get (500^3) / (100^3) = 125. So there would be 125 iterations for 100x100 matrices giving a very good average time for my system to execute the multiplication. On the other hand using the same formula for 500x500 matrices we get (500^3) / (500^3) = 1. The problem with this is that for 500x500 matrices it only runs one time, and that time is taken and added to the data set. So when taking the average between two data sets you get a large .07 second error. Therefore, to get around this I wanted to figure out how many iterations it would take to get a good average and have the smallest error possible, yet still have the program run quickly. For this you need to look at figure 5, and see that at matrice size 300x300 it is fine but at around 320x320 the error spikes and keeps increasing from there. Doing some basic math we find that (500^3) / (300^3) = 4.63. Truncating the decimal it amounts to 4 repetitions of array size 300x300. Going one step size greater to 320x320 matrice size we find that (500^3) / (320^3) = 3.81. Once again truncating the decimal we get 3 repetitions of this array size; however, at this point the error has deviated from being centered on zero. So anything below 4 iterations through variation of matrix multiplication gives an average that is not completely accurate. This is the reasoning why I decided to choose 5 for the number of iterations for each variant of matrix multiplication, 5 being 1 more than 4 makes for certain that the number of iterations will be sufficient enough to have close to zero error. Looking at figure 6 we can see that the error is so basically zero, as the Y axis is a 10^-3 value. In doing this I was successful in minimizing the error as close to zero as possible without drastically increasing the run time of my program. Making the base number of iterations 5 only extended the run time of my program by 10 minutes, making it a total of 20minute run time for data that is very accurate.

Looking at figure 6 more closely even though the error is so close to zero we can still see the odd caching or memory error at around 400x400 matrices. After this anomaly the graph returns to 0, so it's a consistent thing that effects the run time at around 400x400 matrices. I'm unsure if there is a way around it or if there is a way to identify what specifically is happening at this size matrice.

4.5.4 AC power Vs battery power

Looking at figures 1 and 2 for the preprocessor heavy program I did not see any significant differences. All of the differences that I saw were already explained by lack of iterations of a matrix size, or the memory/caching error that happens at around a 400 size matrix.

As with the program that did not use the preprocessor at all, still there were no significant differences. The high escalations you see in figure 4 are not due to AC power, but are instead due to lack of iterations through matrices of large size.

These two tests seemed odd to me as I figured that having the laptop plugged in might stabaliez performance having a constant stream of power to it. This however is not the case, and it doesn't seem to affect the performance of my program at all. Maybe if I increased the size of the matrices I could see more of a difference, between the performances of AC power and battery power.

4.5.5 Future testing

I think in the future it would be important to run tests on even larger matrices to see if the spike that happens at 400x400 matrices is a reoccurring consistent thing. That would be the first step I would take to identifying what is happening. Increasing the max size of the matrices will however increase the run time of the program.

Another test that would be interesting as well to run is only running one variation of matrice at a time, and then comparing that to the fully automated version. This would demonstrate that fully automating the process doesn't effect performance. Or if it did effect performance we would be able to determine the percentage of performance lost when doing this.

5.0 Conclusions

My experiments were a success showing that using the preprocessor heavily to reduce redundancy in code does not impact performance. Initially when looking at figure 5 I was concerned when I saw the error increasing to up to 0.07 seconds, but upon further testing I was able to clarify that it was not due to using the preprocessor. I feel using this structure in your code is good as it makes it so you have one point of maintenance for your entire program. Once again you cannot do this with just simple functions. Yes you will be able to break up your program into smaller chunks; however, you only make it worse by having 48 points of maintenance. If you wanted to change one thing in all the for loops you would have to do it many more times than if you were using my program where there is one point of maintenance. The reason why my program gets to have one point of maintenance is because of the use of the preprocessor. The preprocessor is the only thing that can use symbol concatnenation, and even better it happens before the compiler gets to do anything. If you compare the .i file which is the preprocessed file before compilation, and the brute force program I wrote you will see they are very similar. All in all writing programs that have a single point of maintenance, are compact, and are fully automated is very good. As an engineer this should be your goal, and to achieve it you should not shy away from the preprocessor it is your friend!

Appendix

a.) Preprocessor heavy program

```
/*
* Class: CSC 840
* By: Matthew Wishoff
* Date: 3/4/2016
* Description: This program multiplies two matricies 48 different ways.
*              It also steps the matrix size by 20 on each iteration through the 48 ways.
*/
#include <iostream>
#include <stdio.h>
#include <chrono>

using namespace std;

#define DO_MULTIPLY_1 c[i][j] += a[i][k]*b[k][j]
#define DO_MULTIPLY_2 c[i][j] += a[i][k]*b[j][k]
#define DO_MULTIPLY_3 c[i][j] += a[k][i]*b[k][j]
#define DO_MULTIPLY_4 c[i][j] += a[k][i]*b[j][k]
#define DO_MULTIPLY_5 c[j][i] += a[i][k]*b[k][j]
#define DO_MULTIPLY_6 c[j][i] += a[i][k]*b[j][k]
#define DO_MULTIPLY_7 c[j][i] += a[k][i]*b[k][j]
#define DO_MULTIPLY_8 c[j][i] += a[k][i]*b[j][k]

#define DO_MULTIPLY_IJK(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int i=0; i<n; i++)                                  \
                        for(int j=0; j<n; j++)                          \
                                for(int k=0; k<n; k++)                  \
                                        DO_MULTIPLY_##method;

#define DO_MULTIPLY_IKJ(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int i=0; i<n; i++)                                  \
                        for(int k=0; k<n; k++)                          \
                                for(int j=0; j<n; j++)                  \
                                        DO_MULTIPLY_##method;

#define DO_MULTIPLY_KJI(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int k=0; k<n; k++)                                  \
                        for(int j=0; j<n; j++)                          \
                                for(int i=0; i<n; i++)                  \
                                        DO_MULTIPLY_##method;

#define DO_MULTIPLY_KIJ(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int k=0; k<n; k++)                                  \
                        for(int i=0; i<n; i++)                          \
                                for(int j=0; j<n; j++)                  \
                                        DO_MULTIPLY_##method;
```

```
#define DO_MULTIPLY_JKI(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int j=0; j<n; j++)                                  \
                        for(int k=0; k<n; k++)                          \
                                for(int i=0; i<n; i++)                  \
                                        DO_MULTIPLY_##method;

#define DO_MULTIPLY_JIK(method)                                         \
        for(int m=0; m<M; m++)                                          \
                for(int j=0; j<n; j++)                                  \
                        for(int i=0; i<n; i++)                          \
                                for(int k=0; k<n; k++)                  \
                                        DO_MULTIPLY_##method;

#define MULTIPLY(method)                                                                        \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_IJK(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;      \
                                                                                                \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_IKJ(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                        RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;      \
                                                                                                \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_KJI(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;      \
                                                                                                \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_KIJ(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                        RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;      \
                                                                                                \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_JKI(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                        RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;      \
                                                                                                \
                startTime = chrono::steady_clock::now();                                        \
                        DO_MULTIPLY_JIK(method);                                                \
                stopTime = chrono::steady_clock::now();                                         \
                        RUN_TIME[TEST_NUM++ % NUM_OF_TESTS][col] =
                chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

#define NUM_OF_TESTS (8*6)
#define NMAX 500
```

```
#define NMIN 100
#define STEP 20

int TEST_NUM = 0;
double RUN_TIME[NUM_OF_TESTS][((NMAX - NMIN) / STEP) + 1];

int main(int argc, char**argv)
{
	chrono::steady_clock::time_point startTime, stopTime;
	int col = 0;
	int M ;

	// Initialize
	for(int n=NMIN; n <= NMAX; n+=STEP, col++) //increase size of matrix
	{
		M =(NMAX*NMAX*NMAX)/(n*n*n);
		if(M < 4)
		{
			M = 4;
		}
		double** a = new double*[n]; //Initialize to the min, the next iteration will step by 20
		double** b = new double*[n]; //Initialize to the min, the next iteration will step by 20
		double** c = new double*[n]; //Initialize to the min, the next iteration will step by 20

		cout << "before init" << endl;
		for(int i = 0; i < n; i++)
		{
			a[i] = new double[n];
			b[i] = new double[n];
			c[i] = new double[n];
		}

		// Matrix a[ ][ ], b[ ][ ], and c[ ][ ] initialization
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
			{
				//Initialize diagonal to 2.0002, and everywhere else to 1.0001
				if(i == j)
				{
					a[i][j] = 2.0002;
					b[i][j] = 2.0002;
				}
				else
				{
					a[i][j] = 1.0001;
					b[i][j] = 1.0001;
				}
				c[i][j]=0.0;
			}

		//multiply 6 different matricies and save their run times to the RUN_TIME 2d array
		MULTIPLY(1);
		MULTIPLY(2);
		MULTIPLY(3);
		MULTIPLY(4);
		MULTIPLY(5);
		MULTIPLY(6);
		MULTIPLY(7);
```

```
		MULTIPLY(8);



		//Delete dynamic array of pointers.
		for(int i = 0; i < n; i++)
		{
				delete[] a[i];
				delete[] b[i];
				delete[] c[i];
		}
		delete[] a;
		delete[] b;
		delete[] c;
	}

	//Pipe the RUN_TIME 2D array out into a text file.
	freopen ("MatrixMultiplicationWithBatteryPower5Iterations.txt","w",stdout);

	for(int i = 0; i < NUM_OF_TESTS; i++) // this is correct
	{
		for(int j = 0; j <= (NMAX - NMIN) / STEP; j++)
		{
			if(j == (NMAX - NMIN) / STEP)
			{
				cout << RUN_TIME[i][j];
			}
			else
			{
				cout << RUN_TIME[i][j] << ", "; //Print a row of data
			}
		}
		cout << endl;
	}
	cout << endl;
	fclose (stdout);
}
```

b.) Brute force without preprocessor

```
/*
* Class: CSC 840
* By: Matthew Wishoff
* Date: 3/4/2016
* Description: Standard way of multiplying matricies
*/

#include <iostream>
#include <stdio.h>
#include <chrono>

#define NUM_OF_TESTS 48
#define NMAX 500
#define NMIN 100
#define STEP 20

//Create 2d Array RUN_TIME and store values then you might be done.

int TEST_NUM = 0;
double RUN_TIME[NUM_OF_TESTS][((NMAX - NMIN) / STEP) + 1];

using namespace std;

int main(int argc, char**argv)
{
	chrono::steady_clock::time_point startTime, stopTime;
	int col = 0;
	int M;

	// Initialize
	for(int n=NMIN; n <= NMAX; n+=STEP, col++) //increase size of matrix
	{
		M =(NMAX*NMAX*NMAX)/(n*n*n);
		if(M < 4)
		{
			M = 4;
		}
		double** a = new double*[n]; //Initialize to the min, the next iteration will step by 20
		double** b = new double*[n]; //Initialize to the min, the next iteration will step by 20
		double** c = new double*[n]; //Initialize to the min, the next iteration will step by 20

		cout << "before init" << endl;
		for(int i = 0; i < n; i++)
		{
			a[i] = new double[n];
			b[i] = new double[n];
			c[i] = new double[n];
		}

		// Matrix a[ ][ ], b[ ][ ], and c[ ][ ] initialization
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
			{
				//Initialize diagonal to 2.0002, and everywhere else to 1.0001
				if(i == j)
				{
```

```
                    a[i][j] = 2.0002;
                    b[i][j] = 2.0002;
                }
                else
                {
                    a[i][j] = 1.0001;
                    b[i][j] = 1.0001;
                }
                c[i][j]=0.0;
            }


//*************************************************************************************************
//*************************************************IJKLOOPSSTART*********************************
            startTime = chrono::steady_clock::now();
            for(int m=0; m<M; m++)
                for(int i=0; i<n; i++)
                    for(int j=0; j<n; j++)
                        for(int k=0; k<n; k++)
                        {
                            c[i][j] += a[i][k]*b[k][j];
                        }
            stopTime = chrono::steady_clock::now();
        RUN_TIME[0][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

            startTime = chrono::steady_clock::now();
            for(int m=0; m<M; m++)
                for(int i=0; i<n; i++)
                    for(int j=0; j<n; j++)
                        for(int k=0; k<n; k++)
                        {
                            c[i][j] += a[i][k]*b[j][k];
                        }
            stopTime = chrono::steady_clock::now();
        RUN_TIME[1][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

            startTime = chrono::steady_clock::now();
            for(int m=0; m<M; m++)
                for(int i=0; i<n; i++)
                    for(int j=0; j<n; j++)
                        for(int k=0; k<n; k++)
                        {
                            c[i][j] += a[k][i]*b[k][j];
                        }
            stopTime = chrono::steady_clock::now();
        RUN_TIME[2][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
	startTime = chrono::steady_clock::now();

	for(int m=0; m<M; m++)
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
				{
					c[i][j] += a[k][i]*b[j][k];
				}
	stopTime = chrono::steady_clock::now();
RUN_TIME[3][col] = chrono::duration_cast<chrono::duration<double>>(stopTimestartTime).count() / M;

	startTime = chrono::steady_clock::now();
	for(int m=0; m<M; m++)
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
				{
					c[j][i] += a[i][k]*b[k][j];
				}
	stopTime = chrono::steady_clock::now();
RUN_TIME[4][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

	startTime = chrono::steady_clock::now();
	for(int m=0; m<M; m++)
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
				{
					c[j][i] += a[i][k]*b[j][k];
				}
	stopTime = chrono::steady_clock::now();
RUN_TIME[5][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

	startTime = chrono::steady_clock::now();
	for(int m=0; m<M; m++)
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
				{
					c[j][i] += a[k][i]*b[k][j];
				}
	stopTime = chrono::steady_clock::now();
RUN_TIME[6][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

	startTime = chrono::steady_clock::now();
	for(int m=0; m<M; m++)
		for(int i=0; i<n; i++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
				{
					c[j][i] += a[k][i]*b[j][k];
				}
	stopTime = chrono::steady_clock::now();
RUN_TIME[7][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
//*****************************************************************************************
//*********************************************IJKLOOPSEND*********************************


//*****************************************************************************************
//*********************************************IKJLOOPSSTART*******************************
                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int i=0; i<n; i++)
                                for(int k=0; k<n; k++)
                                        for(int j=0; j<n; j++)
                                        {
                                                c[i][j] += a[i][k]*b[k][j];
                                        }
                stopTime = chrono::steady_clock::now();
        RUN_TIME[8][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;


                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int i=0; i<n; i++)
                                for(int k=0; k<n; k++)
                                        for(int j=0; j<n; j++)
                                        {
                                                c[i][j] += a[i][k]*b[j][k];
                                        }
                stopTime = chrono::steady_clock::now();
        RUN_TIME[9][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int i=0; i<n; i++)
                                for(int k=0; k<n; k++)
                                        for(int j=0; j<n; j++)
                                        {
                                                c[i][j] += a[k][i]*b[k][j];
                                        }
                stopTime = chrono::steady_clock::now();
        RUN_TIME[10][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int i=0; i<n; i++)
                                for(int k=0; k<n; k++)
                                        for(int j=0; j<n; j++)
                                        {
                                                c[i][j] += a[k][i]*b[j][k];
                                        }
                stopTime = chrono::steady_clock::now();
        RUN_TIME[11][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int i=0; i<n; i++)
				for(int k=0; k<n; k++)
					for(int j=0; j<n; j++)
					{
						c[j][i] += a[i][k]*b[k][j];
					}
		stopTime = chrono::steady_clock::now();
	RUN_TIME[12][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int i=0; i<n; i++)
				for(int k=0; k<n; k++)
					for(int j=0; j<n; j++)
					{
						c[j][i] += a[i][k]*b[j][k];
					}
		stopTime = chrono::steady_clock::now();
	RUN_TIME[13][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int i=0; i<n; i++)
				for(int k=0; k<n; k++)
					for(int j=0; j<n; j++)
					{
						c[j][i] += a[k][i]*b[k][j];
					}
		stopTime = chrono::steady_clock::now();
	RUN_TIME[14][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int i=0; i<n; i++)
				for(int k=0; k<n; k++)
					for(int j=0; j<n; j++)
					{
						c[j][i] += a[k][i]*b[j][k];
					}
		stopTime = chrono::steady_clock::now();
	RUN_TIME[15][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;


//*****************************************************************************************
//*********************************************IKJLOOPSEND*********************************
```

```
//*****************************************************************************************
//*********************************************KJILOOPSSTART*********************************
                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[i][k]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[16][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[i][k]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[17][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[k][i]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[18][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[k][i]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[19][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[j][i] += a[i][k]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[20][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int k=0; k<n; k++)
                                for(int j=0; j<n; j++)
                                        for(int i=0; i<n; i++)
                                                c[j][i] += a[i][k]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[21][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int j=0; j<n; j++)
					for(int i=0; i<n; i++)
						c[j][i] += a[k][i]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[22][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int j=0; j<n; j++)
					for(int i=0; i<n; i++)
						c[j][i] += a[k][i]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[23][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;



//*********************************************************************************************
//*********************************************KJILOOPSEND*************************************


//*********************************************************************************************
//*********************************************KIJLOOPSSTART***********************************
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[i][j] += a[i][k]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[24][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[i][j] += a[i][k]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[25][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[i][j] += a[k][i]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[26][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[i][j] += a[k][i]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[27][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[j][i] += a[i][k]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[28][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[j][i] += a[i][k]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[29][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[j][i] += a[k][i]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[30][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int k=0; k<n; k++)
				for(int i=0; i<n; i++)
					for(int j=0; j<n; j++)
						c[j][i] += a[k][i]*b[j][k];


		stopTime = chrono::steady_clock::now();
	RUN_TIME[31][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;


//*****************************************************************************************
//*********************************************KIJLOOPSSTART*********************************
```

```
//*****************************************************************************************
//*********************************************JKILOOPSSTART*********************************
                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[i][k]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[32][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[i][k]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[33][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[k][i]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[34][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[i][j] += a[k][i]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[35][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[j][i] += a[i][k]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[36][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int k=0; k<n; k++)
                                        for(int i=0; i<n; i++)
                                                c[j][i] += a[i][k]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[37][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
					for(int i=0; i<n; i++)
						c[j][i] += a[k][i]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[38][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int k=0; k<n; k++)
					for(int i=0; i<n; i++)
						c[j][i] += a[k][i]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[39][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;



//*************************************************************************************************
//*********************************************JKILOOPSEND*****************************************


//*************************************************************************************************
//*********************************************JIKLOOPSSTART***************************************
		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int i=0; i<n; i++)
					for(int k=0; k<n; k++)
						c[i][j] += a[i][k]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[40][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int i=0; i<n; i++)
					for(int k=0; k<n; k++)
						c[i][j] += a[i][k]*b[j][k];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[41][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int i=0; i<n; i++)
					for(int k=0; k<n; k++)
						c[i][j] += a[k][i]*b[k][j];
		stopTime = chrono::steady_clock::now();
	RUN_TIME[42][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

		startTime = chrono::steady_clock::now();
		for(int m=0; m<M; m++)
			for(int j=0; j<n; j++)
				for(int i=0; i<n; i++)
					for(int k=0; k<n; k++)
```

```
                                                c[i][j] += a[k][i]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[43][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int i=0; i<n; i++)
                                        for(int k=0; k<n; k++)
                                                c[j][i] += a[i][k]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[44][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int i=0; i<n; i++)
                                        for(int k=0; k<n; k++)
                                                c[j][i] += a[i][k]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[45][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int i=0; i<n; i++)
                                        for(int k=0; k<n; k++)
                                                c[j][i] += a[k][i]*b[k][j];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[46][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

                startTime = chrono::steady_clock::now();
                for(int m=0; m<M; m++)
                        for(int j=0; j<n; j++)
                                for(int i=0; i<n; i++)
                                        for(int k=0; k<n; k++)
                                                c[j][i] += a[k][i]*b[j][k];
                stopTime = chrono::steady_clock::now();
        RUN_TIME[47][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

//*********************************************************************************************
//*********************************************JIKLOOPSEND*************************************

                for(int i = 0; i < n; i++)
                {
                        delete[] a[i];
                        delete[] b[i];
                        delete[] c[i];
                }
                delete[] a;
                delete[] b;
                delete[] c;
}
        //Pipe the RUN_TIME 2D array out into a text file.
freopen ("MatrixMultiplicationStandardBatteryPower5Iterations.txt","w",stdout);
```

```
	for(int i = 0; i < NUM_OF_TESTS; i++) // this is correct
	{
		for(int j = 0; j <= (NMAX - NMIN) / STEP; j++)
		{
			if(j == (NMAX - NMIN) / STEP)
			{
				cout << RUN_TIME[i][j];
			}
			else
			{
				cout << RUN_TIME[i][j] << ", "; //Print a row of data
			}
		}
		cout << endl;
	}
	cout << endl;
	fclose (stdout);
}
```

c.) Preprocessed .i File

```
//New lines added to increase readability at some points.

int TEST_NUM = 0;
double RUN_TIME[(8*6)][((500 - 100) / 20) + 1];

int main(int argc, char**argv)
{
	chrono::steady_clock::time_point startTime, stopTime;
	int col = 0;
	int M;


	for (int n = 100; n <= 500; n += 20, col++)
	{
		M = (500*500*500) / (n*n*n);
		if (M < 4)
		{
			M = 4;
		}
		double** a = new double*[n];
		double** b = new double*[n];
		double** c = new double*[n];

		cout << "before init" << endl;
		for (int i = 0; i < n; i++)
		{
			a[i] = new double[n];
			b[i] = new double[n];
			c[i] = new double[n];
		}


		for (int i = 0; i<n; i++)
			for (int j = 0; j<n; j++)
			{

				if (i == j)
				{
					a[i][j] = 2.0002;
					b[i][j] = 2.0002;
				}
				else
				{
					a[i][j] = 1.0001;
					b[i][j] = 1.0001;
				}
				c[i][j] = 0.0;
			}

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++)
c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++)
c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[i][j] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[i][j] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[i][j] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[i][j] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[j][i] += a[i][k]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;
```

```
 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;



startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[j][i] += a[i][k]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) c[j][i] += a[k][i]*b[k][j];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int i=0; i<n; i++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] = chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;
```

```
startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int j=0; j<n; j++) for(int i=0; i<n; i++)
c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;


startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int k=0; k<n; k++) for(int i=0; i<n; i++) for(int j=0; j<n; j++)
c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

 startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int k=0; k<n; k++) for(int i=0; i<n; i++)
c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;

startTime = chrono::steady_clock::now(); for(int m=0; m<M; m++) for(int j=0; j<n; j++) for(int i=0; i<n; i++) for(int k=0; k<n; k++)
c[j][i] += a[k][i]*b[j][k];; stopTime = chrono::steady_clock::now(); RUN_TIME[TEST_NUM++ % (8*6)][col] =
chrono::duration_cast<chrono::duration<double>>(stopTime - startTime).count() / M;;


        for (int i = 0; i < n; i++)
        {
            delete[] a[i];
            delete[] b[i];
            delete[] c[i];
        }
        delete[] a;
        delete[] b;
        delete[] c;
    }


    freopen("MatrixMultiplicationWithBatteryPower5Iterations.txt", "w", (__acrt_iob_func(1)));

    for (int i = 0; i < (8*6); i++)
    {
        for (int j = 0; j <= (500 - 100) / 20; j++)
        {
            if (j == (500 - 100) / 20)
            {
                cout << RUN_TIME[i][j];
            }
            else
            {
                cout << RUN_TIME[i][j] << ", ";
            }
        }
        cout << endl;
    }
    cout << endl;
    fclose((__acrt_iob_func(1)));
}
```