CSCE735 Final Project

Chonglin Zhang
833003072

1. Develop a shared-memory code using OpenMP. The below image shows simple test pass output.

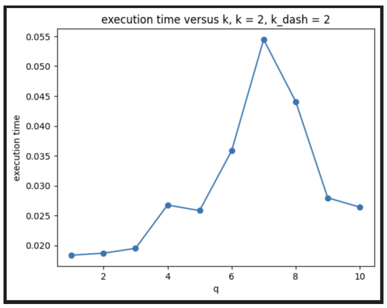
```
test,k',k,q
k': 2, Matrix Size: 2 x 2, Threads: 256, Time:  0.05136 sec
k': 2, Matrix Size: 4 x 4, Threads: 256, Time:  0.04404 sec
k': 2, Matrix Size: 16 x 16, Threads: 256, Time:  0.01811 sec
k': 2, Matrix Size: 64 x 64, Threads: 256, Time:  0.03230 sec
k': 2, Matrix Size: 256 x 256, Threads: 256, Time:  0.46774 sec
```

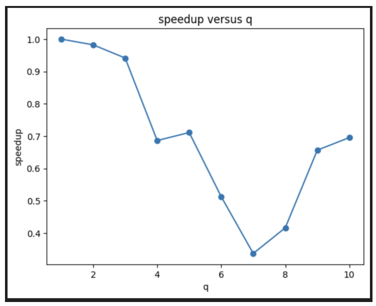
2. Below graphs are going to show datasets, and comparing different k and k' value which will plot graphs for execution time vs q, speedup vs q, and efficiency vs q.

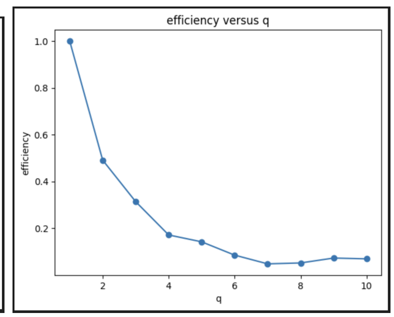
```
k' = 2, k = 2,q = 0,1,2,3,4,5,6,7,8,9,10
k': 2, Matrix Size: 4 x 4, Threads: 2, Time:  0.01837 sec
k': 2, Matrix Size: 4 x 4, Threads: 4, Time:  0.01870 sec
k': 2, Matrix Size: 4 x 4, Threads: 8, Time:  0.01951 sec
k': 2, Matrix Size: 4 x 4, Threads: 16, Time:  0.02676 sec
k': 2, Matrix Size: 4 x 4, Threads: 32, Time:  0.02583 sec
k': 2, Matrix Size: 4 x 4, Threads: 64, Time:  0.03582 sec
k': 2, Matrix Size: 4 x 4, Threads: 128, Time:  0.05446 sec
k': 2, Matrix Size: 4 x 4, Threads: 256, Time:  0.04408 sec
k': 2, Matrix Size: 4 x 4, Threads: 512, Time:  0.02797 sec
k': 2, Matrix Size: 4 x 4, Threads: 1024, Time:  0.02641 sec
k' = 2, k= 4, q = 0,1,2,3,4,5,6,7,8,9,10
k': 2, Matrix Size: 16 x 16, Threads: 2, Time:  0.01641 sec
k': 2, Matrix Size: 16 x 16, Threads: 4, Time:  0.01673 sec
k': 2, Matrix Size: 16 x 16, Threads: 8, Time:  0.01857 sec
k': 2, Matrix Size: 16 x 16, Threads: 16, Time:  0.02636 sec
k': 2, Matrix Size: 16 x 16, Threads: 32, Time:  0.02174 sec
k': 2, Matrix Size: 16 x 16, Threads: 64, Time:  0.02728 sec
k': 2, Matrix Size: 16 x 16, Threads: 128, Time:  0.02850 sec
k': 2, Matrix Size: 16 x 16, Threads: 256, Time:  0.02677 sec
k': 2, Matrix Size: 16 x 16, Threads: 512, Time:  0.02802 sec
k': 2, Matrix Size: 16 x 16, Threads: 1024, Time:  0.02616 sec
k'=4, k = 8, q = 0,1,2,3,4,5,6,7,8,9,10
k': 4, Matrix Size: 256 x 256, Threads: 2, Time:  0.19499 sec
k': 4, Matrix Size: 256 x 256, Threads: 4, Time:  0.22508 sec
k': 4, Matrix Size: 256 x 256, Threads: 8, Time:  0.23352 sec
k': 4, Matrix Size: 256 x 256, Threads: 16, Time:  0.23331 sec
k': 4, Matrix Size: 256 x 256, Threads: 32, Time:  0.22918 sec
k': 4, Matrix Size: 256 x 256, Threads: 64, Time:  0.23039 sec
k': 4, Matrix Size: 256 x 256, Threads: 128, Time:  0.21473 sec
k': 4, Matrix Size: 256 x 256, Threads: 256, Time:  0.20310 sec
k': 4, Matrix Size: 256 x 256, Threads: 512, Time:  0.20281 sec
k': 4, Matrix Size: 256 x 256, Threads: 1024, Time:  0.21711 sec
```

```
k'=6, k = 8, q = 0,1,2,3,4,5,6,7,8,9,10
k': 6, Matrix Size: 256 x 256, Threads: 2, Time:  0.17084 sec
k': 6, Matrix Size: 256 x 256, Threads: 4, Time:  0.22763 sec
k': 6, Matrix Size: 256 x 256, Threads: 8, Time:  0.24495 sec
k': 6, Matrix Size: 256 x 256, Threads: 16, Time:  0.23428 sec
k': 6, Matrix Size: 256 x 256, Threads: 32, Time:  0.19921 sec
k': 6, Matrix Size: 256 x 256, Threads: 64, Time:  0.23176 sec
k': 6, Matrix Size: 256 x 256, Threads: 128, Time:  0.21033 sec
k': 6, Matrix Size: 256 x 256, Threads: 256, Time:  0.21670 sec
k': 6, Matrix Size: 256 x 256, Threads: 512, Time:  0.20445 sec
k': 6, Matrix Size: 256 x 256, Threads: 1024, Time:  0.22866 sec
```

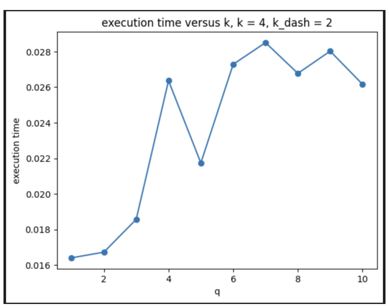
For k’ = 2, and k = 2, and q = 1,2,3,4,5,6,7,8,9,10 when p = 2^q

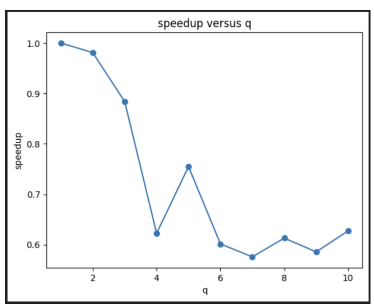


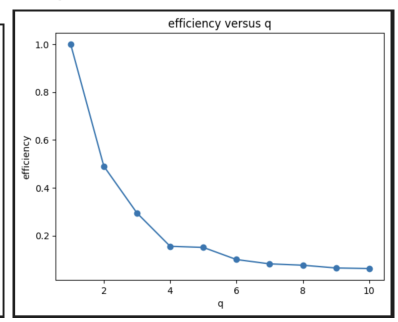




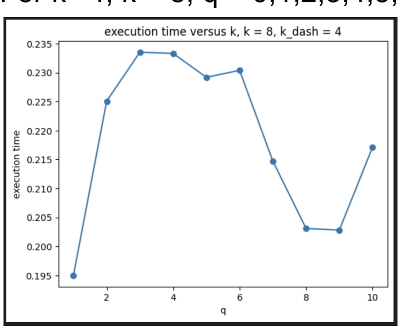
For k’ = 2, and k = 4, and q = 1,2,3,4,5,6,7,8,9,10 when p = 2^q

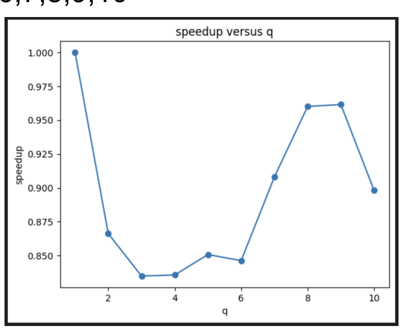


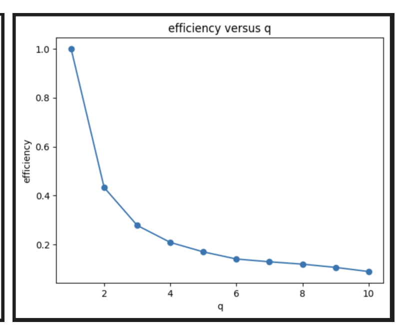




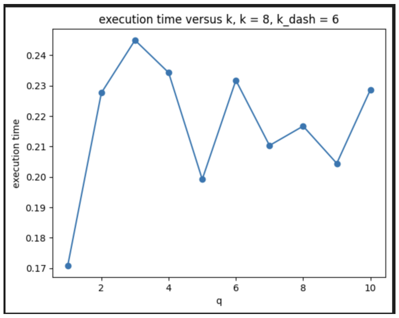
For k'=4, k = 8, q = 0,1,2,3,4,5,6,7,8,9,10

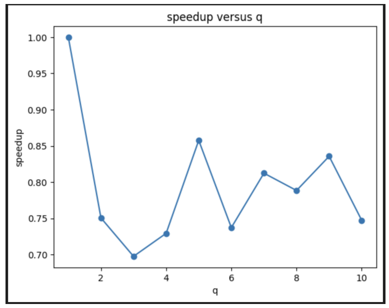


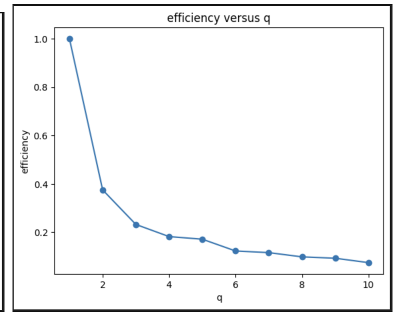




k'=6, k = 8, q = 0,1,2,3,4,5,6,7,8,9,10

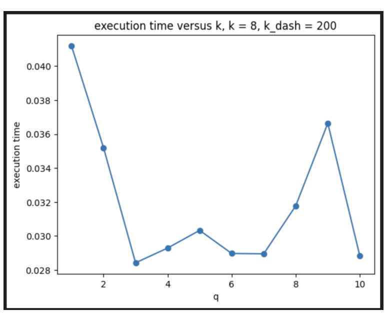


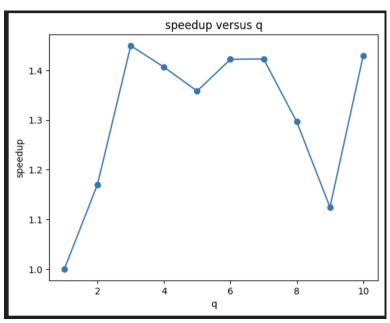


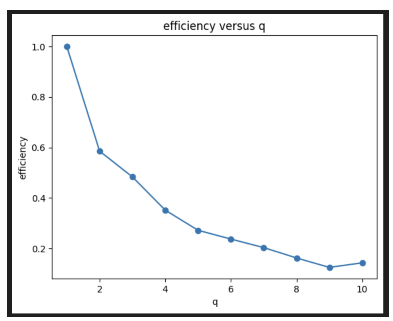


I used 4 different datasets to plot execution time, speedup, and efficiency for different k' value with the same k and same k' with different k which k > k'. We can see all the efficiency graphs show that it starts to drop down from the beginning. Base on the efficiency graph, when k is greater and higher, it required more time to finish the task. I also do the experiment which will show the graph and dataset below. Below the data set, I I use the same k value, but I increase the k' which is very close to k and smaller than k. We can observe that high k' will have lower execution time, and better performance for speedup base on what the speedup graph shows. Also, we can see the efficiency was increasing. Therefore, when we have larger k' which close to k but smaller than k, we will have better speedup and efficiency when q = 4 to 6.

```
k'=200, k = 8, q = 0,1,2,3,4,5,6,7,8,9,10
k': 200, Matrix Size: 256 x 256, Threads: 2, Time:  0.04120 sec
k': 200, Matrix Size: 256 x 256, Threads: 4, Time:  0.03520 sec
k': 200, Matrix Size: 256 x 256, Threads: 8, Time:  0.02842 sec
k': 200, Matrix Size: 256 x 256, Threads: 16, Time:  0.02929 sec
k': 200, Matrix Size: 256 x 256, Threads: 32, Time:  0.03033 sec
k': 200, Matrix Size: 256 x 256, Threads: 64, Time:  0.02897 sec
k': 200, Matrix Size: 256 x 256, Threads: 128, Time:  0.02895 sec
k': 200, Matrix Size: 256 x 256, Threads: 256, Time:  0.03178 sec
k': 200, Matrix Size: 256 x 256, Threads: 512, Time:  0.03664 sec
k': 200, Matrix Size: 256 x 256, Threads: 1024, Time:  0.02882 sec
```







For this code performance, I use divide and conquer method because it is a large multiplication which the problem or matrix need to be split in order to get a good performance. Also, implementing the parallel computing OpenMP which will let the code performance better and faster.

Lastly, here is how I compile the code. We need to load below command first.
Module load intel

Second, I use cpp for this final project. Using the command line below to run and execute the major.cpp file. Also, need to add -qopenmp for command line because it is for OpenMP. After all, we will generate and get the major.exe
icc -qopenmp -o major.exe major.cpp

Third, we can use below command line to run the code and fill in k’,k, and q values. Also, need to know that n = 2^k and p = 2^q.
./major.exe <k’> <k> <q>
EX: ./major.exe 2 6 8