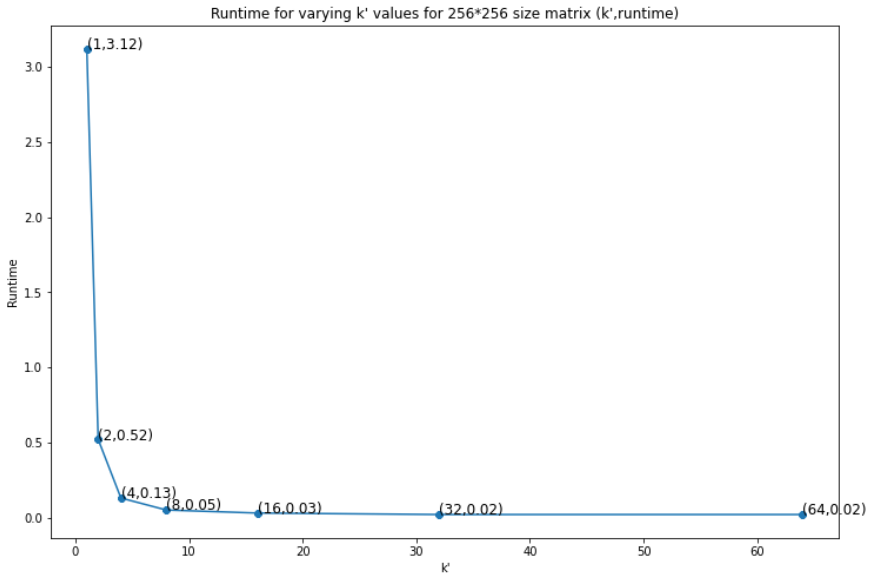
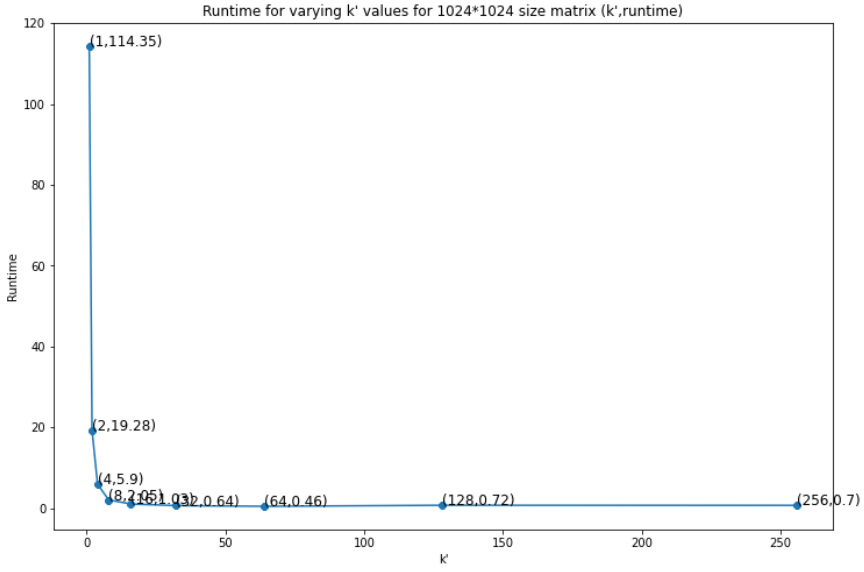
Major Project Report
CSCE 743
Reid OBoyle

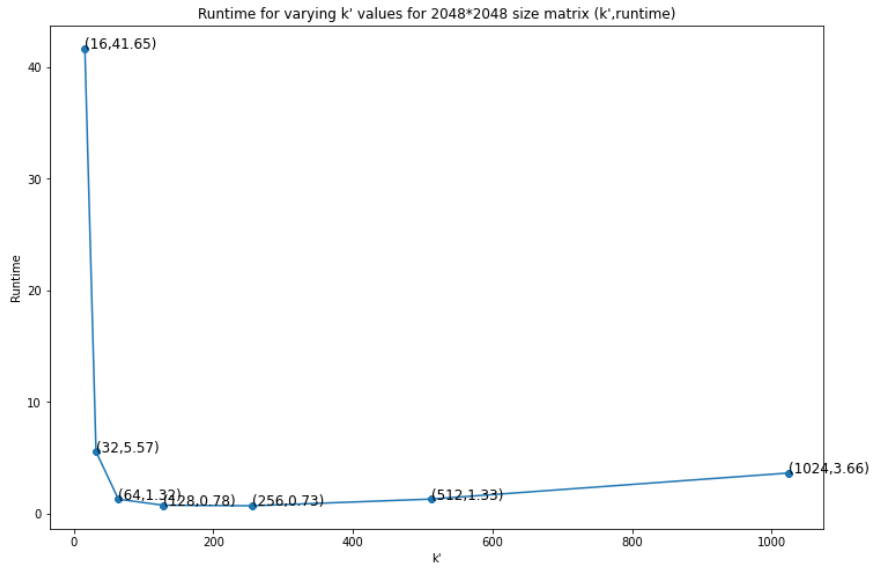
I chose to implement Strassen's algorithm with OpenMP. The first thing I measured is the runtime of different matrix sizes with varying k' sizes. This is shown below.

Performance on different matrix sizes with varying k' values





Major Project Report
CSCE 743
Reid OBoyle



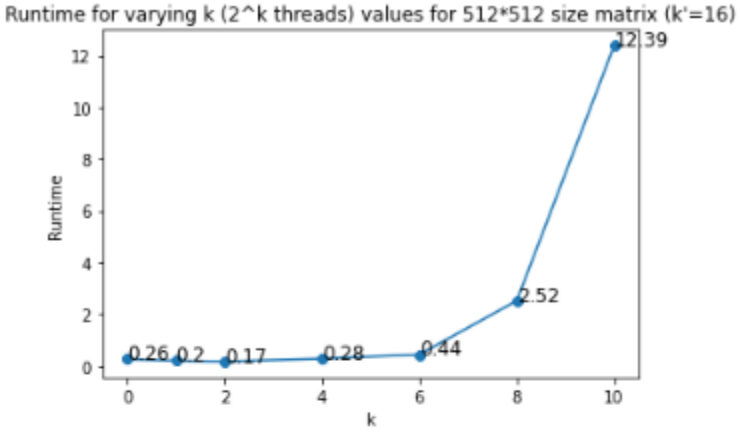
(To clarify: k' is the actual width of the matrix when Strassen does naïve matrix multiplication. The width is not 2^k' but is rather k')

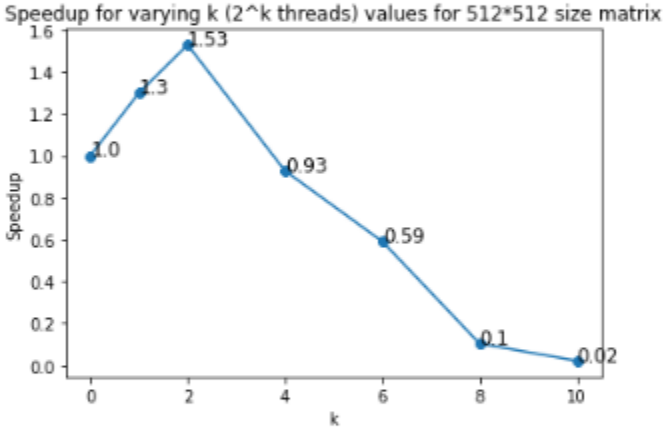
I ran varying k' values on matrices with dimensions of 256*256,1024*1024,2048*2048. For the experiment on the 2048*2048 sized matrix, running with k' of 1,2,4,8 was way too slow, so I did not include them in the figure. All three experiments showed the same result. I found that increasing k' drastically increases performance of Strassen's algorithm when k' is small. However, performance increase leveled off and got worse as k' got to larger numbers. This is likely because the bigger you make k' the closer you are to just running the naïve matrix multiplication. I expect that the bad performance for small k' values is because of the extra memory allocation in the deep recursions that come with a small k'. I found that the best performance for each matrix size was when k' was 1/32nd

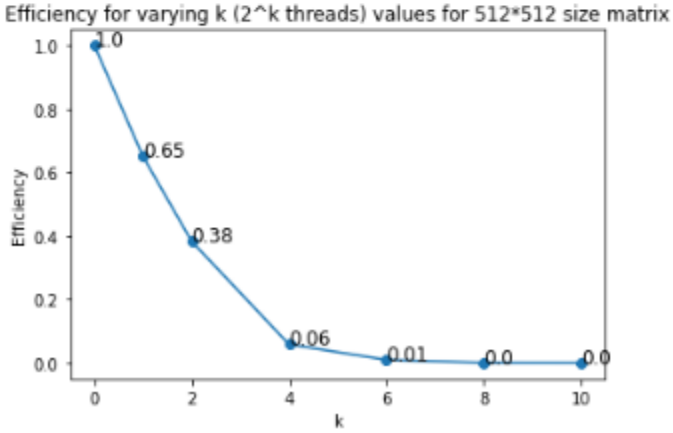
or 1/16th of the actual matrix size. When testing with different thread amounts, this stayed relatively consistent. For the following experiment we will use a k' 1/32 of the original matrix width to get the best performance.

Now that we have established the best k' value to use, we will conduct experiments on different values of k. We will also test each k value with different amounts of threads to measure the speedup and efficiency of the parallel algorithm. We will use k values 9,10,11 and threads of 2^n for n=0,1,2,4,6,8,10.

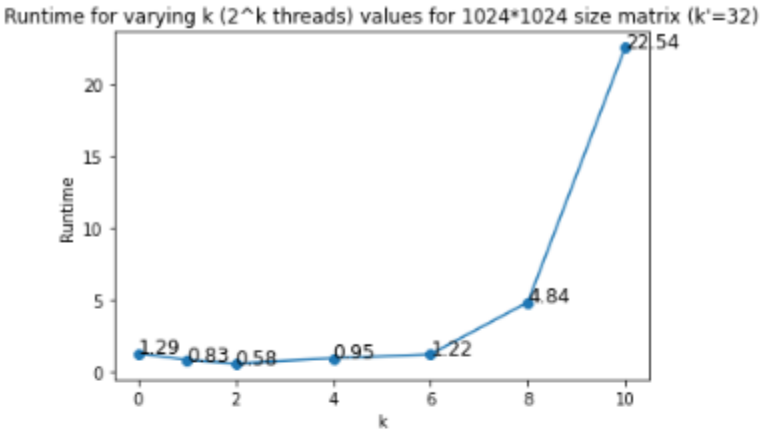
Major Project Report
CSCE 743
Reid OBoyle

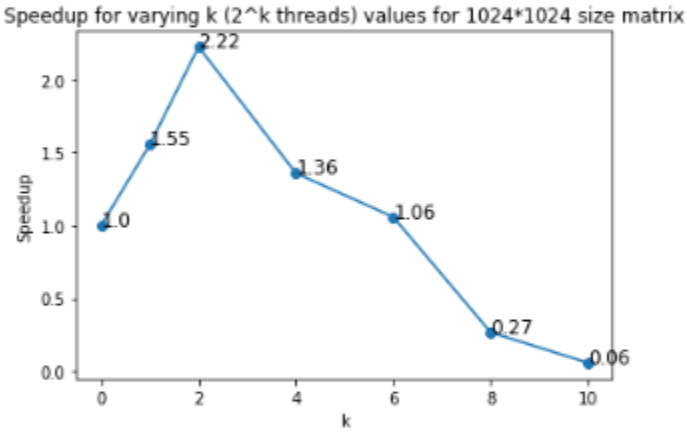


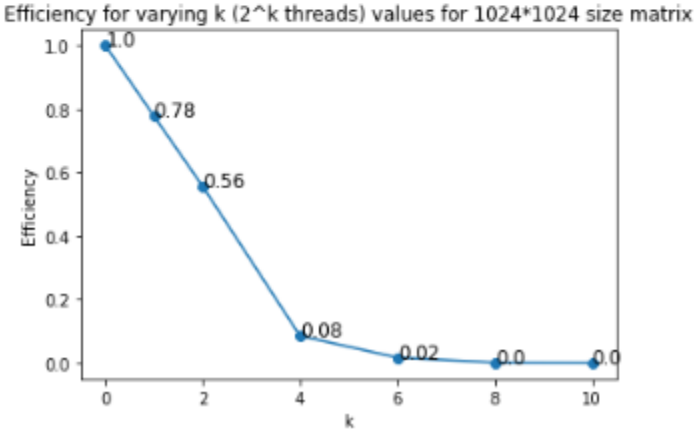




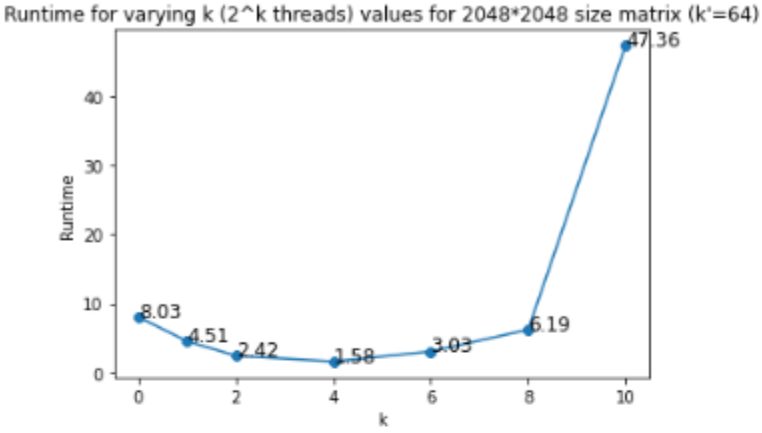
Major Project Report
CSCE 743
Reid OBoyle

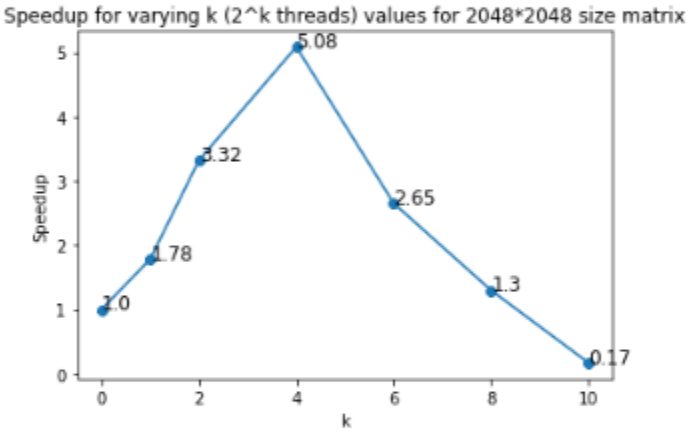


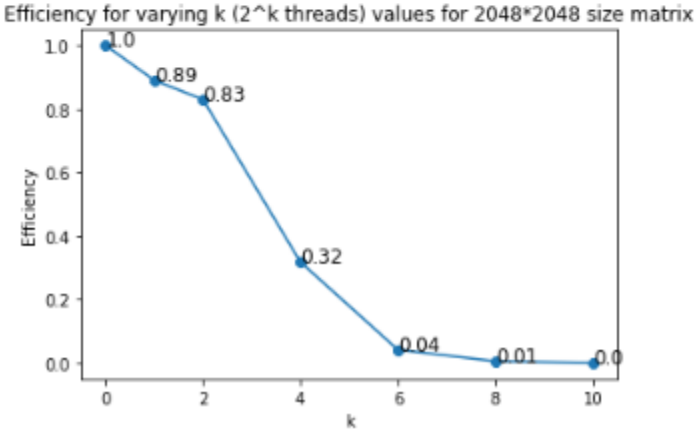




Major Project Report
CSCE 743
Reid OBoyle







We see similar results for k=9,10,11. In each experiment adding threads decreases the runtime by some degree but after about 2^6 threads we don't see any more improvement. Performance decreases as we add very large numbers of threads. I expect this is because when there are more threads there is a lot more overhead in creating the threads and destroying them at the end. Additionally, the threads are just

Major Project Report
CSCE 743
Reid OBoyle

doing the matrix multiplications at terminal matrix size and creating tasks. The small matrices they are working on are close in memory and are likely cached to the same block. They are likely dealing with a lot of extra time spent on the cache coherency. I expect that CUDA would yield better performance than OpenMP because GPUs work better on matrices. Strassen had a big improvement on naïve Matrix Multiplication for large matrix sizes. It seemed like the best thread count to use was probably 2^4 threads. This limits Strassen to one initial thread and 15 threads for each recursive call. This probably works best because those 15 threads each create all the tasks before anything. Once all the tasks are created, the threads go about executing the tasks that are doing regular matrix multiplication. I expect that a larger the terminal matrix improves performance because the threads can utilize more time executing the parallel matrix multiplication and the overhead is not as much of an issue.

Running the code

```
$ module load intel

$ icpc -qopenmp -o matrix.exe matrix.cpp
```

```
$ ./matrix [k] [k'] (optional) [n] (optional). Where matrices are size 2^k and k' is the terminal matrix size.\n2^n is the number of threads. Default k' is 2. Default value of n is 3\n"

Example running on 1024*1024 matrix with 32*32 terminal matrix size and 2^4 threads

$./matrix 10 32 4
```