Jasen Hall
CS 240A, Winter 2014
Homework 1B

Code

My code is based on the matrix multiplication code provided for the assignment. I modified it by added a total of 6 *algorithms*, or variations on the order in which the matrix elements are addresses, and a 7th algorithm of my own design. I will describe why order matters and the results of my own algorithm in the **Results** section.

In addition to the provided code, I also wrote a Python script that ran the program for each algorithm using increasing matrix sizes to determine each algorithm's scalability. The script timed a multiplication of two matrices for each matrix size. Further, it did this 4 times and averaged the time to compute the product. It then increased the size of the matrix by a factor of 2 for the next set of multiplications. The scaling of the factors increased until a multiplication took longer than 10 minutes.

Results

The fastest ordering of the operations was **jki** by a slim margin over **kji**. The order of **ikj** roughly tied with the ordering **kij** for the distinction of the worst ordering of operations. See Fig. 1 and the raw data at the end of the paper for details of each algorithm.

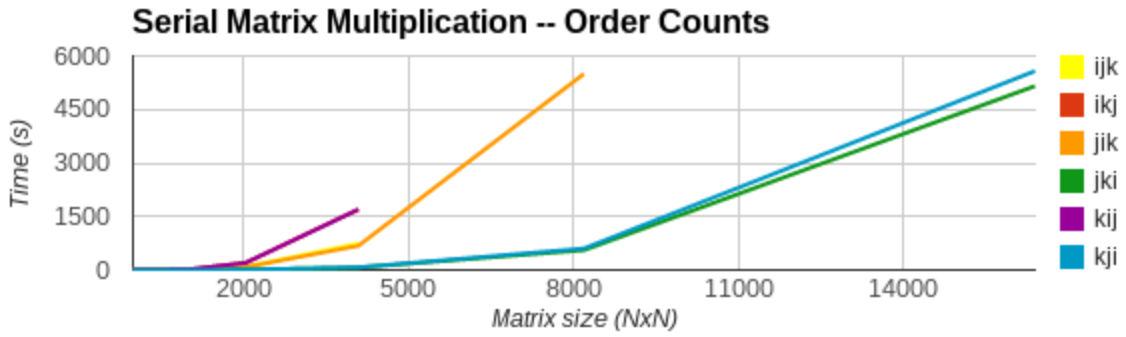

Fig. 1: Timing of multiplication algorithms

When considering the number of floating operations per second each algorithm was able to manage, the most- and least-efficient maintain their titles. Both the **jki** and **kji** orderings show initial high rates of calculations, followed immediately by sharp increases in efficiency and an eventual platueing as the matrix sizes increase. The two least efficient algorithms start at about half the rate of the leaders and decrease quickly as the sizes increase. See Fig. 2.

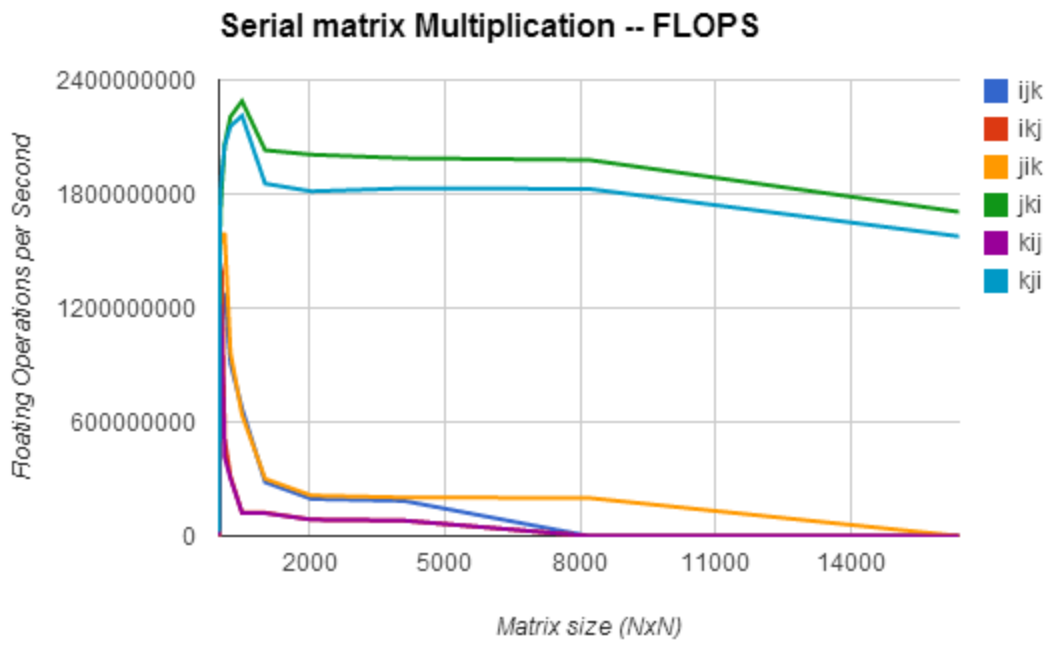

Fig. 2: Multiplication rate in floating operations per second

Conclusion

When we analyze the algorithms that performed the best we find that the key is in how the data is accessed. When matrix elements are accessed in the same order in which it they are laid out, specifically **column-major** order, we are able to leverage memory locality to our advantage. We get the benefit of temporal locality when we repeatedly access the same memory locations in consecutive operations. We are also able to benefit from spatial locality since data is read from main memory into cache not at the byte level but as blocks. This means that not only the element needed for the current operation will be cached, but it is very likely that the adjacent elements that we get as well will be needed for calculations in the immediate future.

For my own algorithm, I attempted to come up with a way to better take advantage of the above localities. After much whiteboard work and experimentation, I came up with an algorithm that worked in times almost identical to the **kji** ordering. Upon comparison I realised that the reason for this was that they *were* the same algorithm. So short of using a faster processor with more cache, I think the best algorithms determined through experimentation are near-ideal for a serial implementation.

The Python script was helpful for performing the analysis of the algorithms. It allowed me to run about 7 hours of timing runs on a CSIL machine overnight while it was unburdened. Without it I would have been less accurate in my record keeping and certainly less rested.

Data

	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
ijk	0	0.0000075	0.0000015	0.0000725	0.000053	0.0004085	0.0033102	0.0369412	0.399823	7.583983	88.155881	743.84158825		
ikj	0	0.0000025	0.0000125	0.0000575	0.0004575	0.0003755	0.0081757	0.1067945	2.1943957	17.5406335	199.354981	1700.14136775		
jik	0	0.0000005	0.0000015	0.0000625	0.0004275	0.000335	0.002628	0.034741	0.4203455	7.13964125	80.698704	678.09673075	5498.862167	
jki	0.0000005	0.000001	0.000002	0.0000675	0.0003875	0.0002865	0.0020365	0.0152055	0.1172682	1.0576595	8.560155	69.12302775	555.556624	5156.657473
kij	0	0.0000005	0.0000015	0.000007	0.0005375	0.0004975	0.0101265	0.1104412	2.1985845	17.8662697	199.760307	1695.7413732		
kji	0.0000075	0.0000125	0.000002	0.0000625	0.0003625	0.0002755	0.0020417	0.0155615	0.121378	1.15806975	9.47127825	75.1259675	601.91240225	5577.82604325