COL 380 A0 Report

Sarthak Panda(2022CS51217)

January 2025

1 Experiment - I

1.1 Plot of Execution Time v/s Matrix size

A plot was created for each of the permutations (IJK, IKJ, JIK, JKI, KIJ, KJI) by varying the matrix size (shown on the x-axis) and recording the corresponding execution time (in seconds, as measured by the driver script provided) on the y axis.

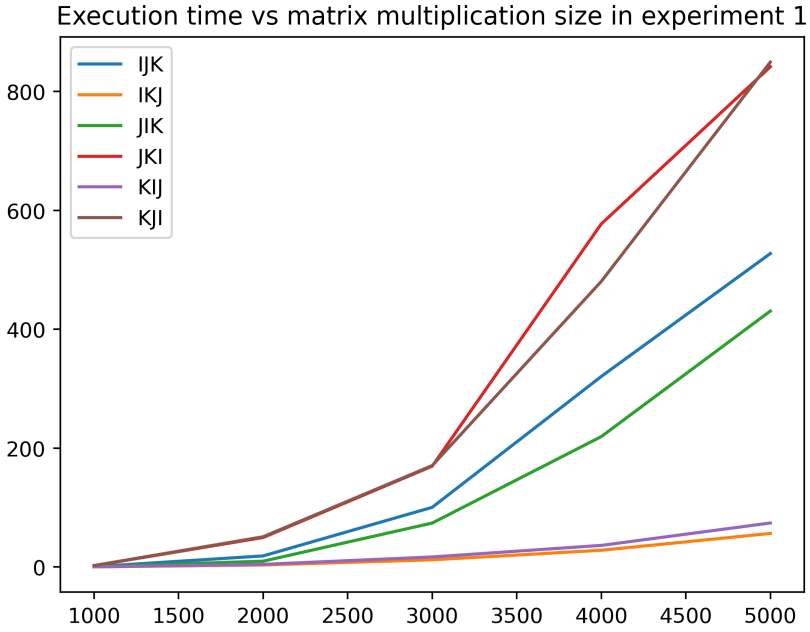

Figure 1: Experiment - I plot.

2 Experiment - II

2.1 Plot of User Time v/s Matrix size

Similar to the above plot but Execution time along the y-axis replaced by the user time provided by perf.

Comparsion with plot in section 1

Before comparing the plots, let us define the Execution time and User time.

- **Execution Time:** It represents the total time a program takes to execute from start to finish. It includes User Time, System Time, and Waiting Time.
- **User Time:** It is the time spent executing instructions in user mode, which is the non-privileged mode where application-level code runs.

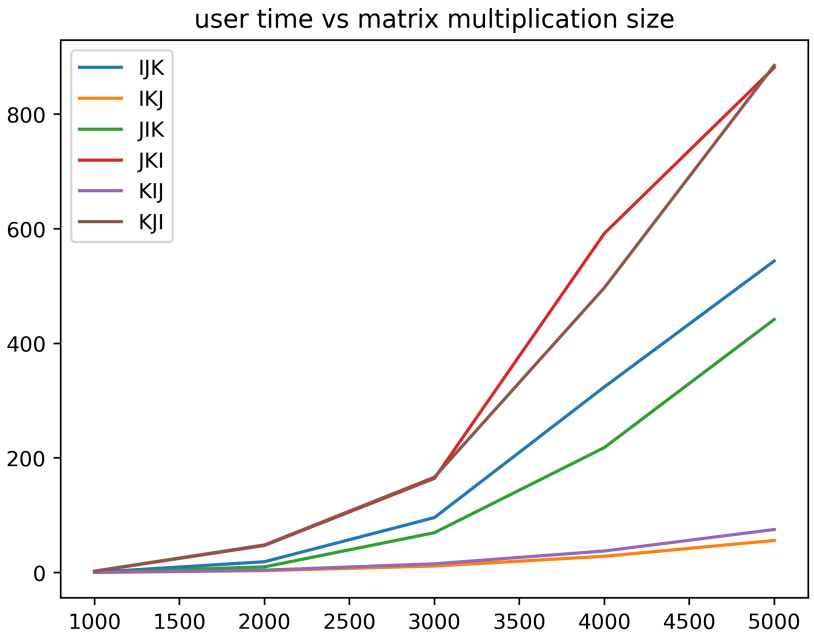

Figure 2: Experiment - II(a) plot.

- **System Time:** It is time spent in the kernel mode handling system calls (e.g., I/O operations).
- **Waiting Time:** It is time spent waiting for other processes, such as disk I/O or network responses.

By noticing the output from the perf command, it was evident that the system time and waiting time vary from a few milliseconds to seconds. Following this, we can say most of the execution time is indeed the user time, which is the reason the graphs have similar trends.

2.2 Plot of User Time for Matrix Multiplication v/s Matrix size

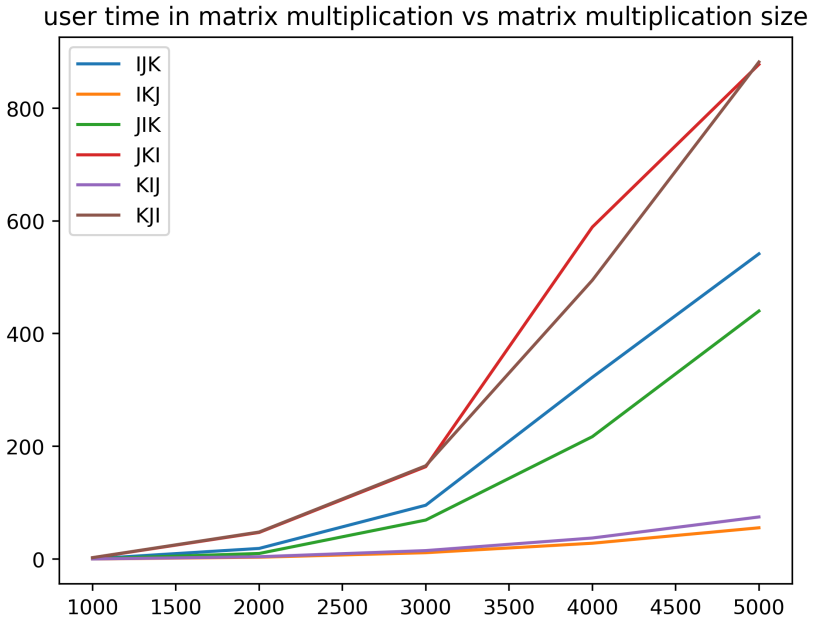

Figure 3: Experiment - II(b) plot.

Here we have considered plotting the self-time of matrix multiplication from the perf report against the matrix size. As the readMatrix and writeMatrix functions take a very small percentage of time in comparison to the matrixMultiplication function, which consists of 95% to 99% of userTime, this is the reason we observe a similar graph to the above one.

Understanding the Trends of Execution/User Time

To understand the trends we first need to look into memory access patterns of each of these permutations; based on this, we can classify them into the following types:

- **Type: IJK & JIK** In both of these cases we access the elements of A in row-major order and that of B in column-major order. It has moderate performance, by understanding below two categories it will be clear why this trend falls in between.
- **Type: KIJ & IKJ** In these cases we access the elements of both A & B in row major order. The result in this case is also stored in row-major order. It is the most favorable case in terms of performance because of proximity in memory accesses due to which we observe their trends to be at the bottom of the graph.
- **Type: JKI & KJI** In these cases the elements of A are accessed in row-major order. But the thing that makes it worst performing among all the three types in terms of performance is the fact the elements of C are being stored in column major order due to which there are frequent write backs to update the dirty cache lines while replacing the cache line with new data from next column ordered element (which is farther away in memory). This is the reason observe these permutations' curve lie above all other curves in the graph.

2.3 Plot of Cache Hit Rate v/s Matrix size

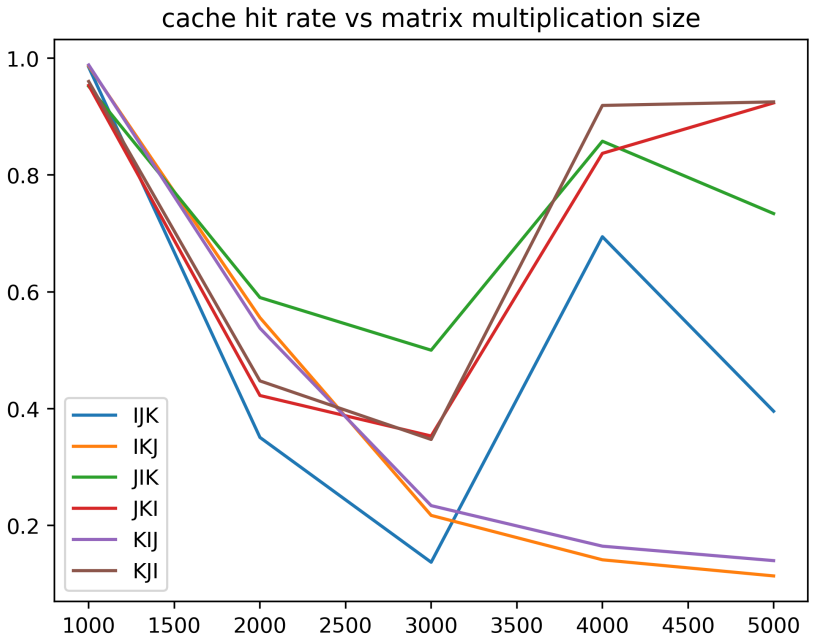

Figure 4: Experiment - II(c) plot.

Understanding the Trends of Cache Hit Rate

From the above discussion, it was clear that the IKJ and KIJ groups had optimal cache access patterns. However, we observed these to be at the bottom of the cache hit rate graphs. The reason could be the spatial proximity of data in the IKJ, causing the data from the cache to be moved to registers in a parallelized way, reducing the total number of cache accesses and hence giving a lower cache hit rate.

Permutation	Avg cache hit rate across matrix sizes
IJK	0.5124
IKJ	0.4029
JIK	0.7266
JKI	0.6978
KIJ	0.4129
KJI	0.7197

Table 1: Average cache hit rate for each permutation.

Permutation	Avg execution time across matrix sizes (sec)
IJK	196.66
IKJ	19.9376
JIK	148.0268
JKI	337.4176
KIJ	26.5836
KJI	319.879

Table 2: Average execution time for each permutation.

2.4 Optimal Permutation and Disscussions

By analyzing Table-2, it is clear that IKJ is the optimal permutation in terms of average execution time across various sizes. This aligns with our explanation mentioned under the section 'Understanding the Trends of Execution/User Time'. If we look at Table-1, we can see that the permutations JIK, KJI, and JKI have one of the highest cache hit rates. The reason for this is not very clear and seems to have some relation with cache register interaction (as mentioned under section 'Understanding the Trends of Cache Hit Rate') and the sequential nature of the test we conducted.

2.5 Comparison of Reports from perf and gprof for time spent by each individual function

The table on the next page shows that both perf and gprof indicate a similar distribution of user time spent in individual functions (namely main, readMatrix, writeMatrix, and multiplyMatrix). However, perf provides a more realistic and accurate profiling. This was expected because, while gprof can show the exact function call count for a single program, it cannot resolve the call stack accurately and only provides an approximation. In contrast, perf is a modern, Linux-only tool for statistical profiling. It leverages hardware performance monitoring units and supports both single-program profiling and system-wide profiling. [Note that in the table below, for the performance data, we provided the percentages of user time for each function, including its child processes. However, in the raw data csv, we provided the self-time for the function involved.]

		1000, IJK		1000, IKJ		1000, JIK		1000, JKI		1000, KIJ		1000, KJI	
		perf	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**
%age of time spent in	matrixMultiply	97.95	100.0	93.23	100.0	97.62	100.0	99.45	100.0	95.36	100.0	99.51	100.0
	readMatrix	0.34	0.0	1.04	0.0	0.4	0.0	0.04	0.0	0.71	0.0	0.07	0.0
	writeMatrix	0.34	0.0	2.08	0.0	0.79	0.0	0.17	0.0	1.07	0.0	0.11	0.0
	main	98.81	N/A	96.88	N/A	98.81	N/A	99.66	N/A	97.5	N/A	99.74	N/A
		2000, IJK		2000, IKJ		2000, JIK		2000, JKI		2000, KIJ		2000, KJI	
		perf	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**
%age of time spent in	matrixMultiply	99.76	100	98.31	100	99.46	100	99.88	100	98.73	100	99.89	100
	readMatrix	0.04	0	0.2	0	0.07	0	0.02	0	0.16	0	0.01	0
	writeMatrix	0.06	0	0.5	0	0.16	0	0.04	0	0.39	0	0.03	0
	main	99.87	N/A	99.21	N/A	99.72	N/A	99.94	N/A	99.38	N/A	99.94	N/A
		3000, IJK		3000, IKJ		3000, JIK		3000, JKI		3000, KIJ		3000, KJI	
		perf	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**
%age of time spent in	matrixMultiply	99.89	100	98.92	100	99.80	100	99.93	100	99.13	100	99.92	100
	readMatrix	0.02	0	0.16	0	0.03	0	0.01	0	0.12	0	0.01	0
	writeMatrix	0.02	0	0.31	0	0.05	0	0.02	0	0.25	0	0.02	0
	main	99.94	N/A	99.44	N/A	99.92	N/A	99.96	N/A	99.59	N/A	99.96	N/A
		4000, IJK		4000, IKJ		4000, JIK		4000, JKI		4000, KIJ		4000, KJI	
		perf	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**
%age of time spent in	matrixMultiply	99.93	100	99.24	100	99.89	100	99.95	100	99.32	100	99.94	100
	readMatrix	0.01	0	0.11	0	0.01	0	0	0	0.08	0	0.01	0
	writeMatrix	0.01	0	0.24	0	0.03	0	0.01	0	0.21	0	0.01	0
	main	99.97	N/A	99.63	N/A	99.95	N/A	99.98	N/A	99.71	N/A	99.98	N/A
		5000, IJK		5000, IKJ		5000, JIK		5000, JKI		5000, KIJ		5000, KJI	
		perf	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**	**perf**	**gprof**
%age of time spent in	matrixMultiply	99.94	100	99.41	100	99.9	100	99.95	100	99.54	100	99.95	100
	readMatrix	0.01	0	0.09	0	0.01	0	0.01	0	0.06	0	0.01	0
	writeMatrix	0.01	0	0.17	0	0.02	0	0.01	0	0.14	0	0.01	0
	main	99.97	N/A	99.71	N/A	99.96	N/A	99.98	N/A	99.77	N/A	99.98	N/A

Table 3: Comparison of Perf and Gprof data for matrix sizes 1000 to 5000.