CS 4552
Assignment 1

Matrix Multiplication Performance

180449H
K.P.D.T. Pathirana
Computer Science and Engineering

Table of Content

1.1 Results of the Experiments

The experiment rounds were run 20 times for each loop order and for each N (10, 100, 500, 1000). Finally, the average execution time was calculated with its variance.

Since the three matrices A, B, and C represent most of the memory utilization, estimated memory utilization was calculated as follows.

Estimated Memory Utilization = No. of matrices * N * N * sizeof(int *or* float)

No, of matrices = 3
sizeof(int) = 4 bytes
sizeof(float) = 4 bytes

N = 10

Floats			**Integers**		
Main Mem Available	**Est Mem Utilization**		**Main Mem Available**	**Est Mem Utilization**	
127039784 bytes	3*10*10*4 = 1200 bytes		127039280 bytes	3*10*10*4 = 1200 bytes	
Order	**Avg Time**	**Variance**	**Order**	**Avg Time**	**Variance**
IJK	0.000017	0.000000	IJK	0.000016	0.000000
IKJ	0.000016	0.000000	IKJ	0.000016	0.000000
JIK	0.000017	0.000000	JIK	0.000016	0.000000
JKI	0.000016	0.000000	JKI	0.000016	0.000000
KIJ	0.000016	0.000000	KIJ	0.000017	0.000000
KJI	0.000016	0.000000	KJI	0.000016	0.000000

N = 100

Floats			**Integers**		
Main Mem Available	**Est Mem Utilization**		**Main Mem Available**	**Est Mem Utilization**	
127039280 bytes	3*100*100*4 = 120000 bytes		127039280 bytes	3*100*100*4 = 120000 bytes	
Order	**Avg Time**	**Variance**	**Order**	**Avg Time**	**Variance**
IJK	0.006380	0.000003	IJK	0.006237	0.000003
IKJ	0.005541	0.000000	IKJ	0.005571	0.000000

JIK	0.005821	0.000000	JIK	0.005617	0.000000
JKI	0.005494	0.000000	JKI	0.005511	0.000000
KIJ	0.005529	0.000000	KIJ	0.005558	0.000000
KJI	0.005496	0.000000	KJI	0.005513	0.000000

N = 500

Floats			**Integers**		
Main Mem Available	**Est Mem Utilization**		**Main Mem Available**	**Est Mem Utilization**	
127038208 bytes	3*500*500*4 = 3000000 bytes		127037748 bytes	3*500*500*4 = 3000000 bytes	
Order	**Avg Time**	**Variance**	**Order**	**Avg Time**	**Variance**
IJK	0.477692	0.004017	IJK	0.459795	0.000000
IKJ	0.578760	0.000001	IKJ	0.577875	0.000001
JIK	0.463647	0.000000	JIK	0.466737	0.000000
JKI	0.422671	0.000000	JKI	0.422956	0.000000
KIJ	0.577101	0.000000	KIJ	0.578653	0.000000
KJI	0.423421	0.000000	KJI	0.423311	0.000000

N = 1000

Floats			**Integers**		
Main Mem Available	**Est Mem Utilization**		**Main Mem Available**	**Est Mem Utilization**	
127037812 bytes	3*1000*1000*4 = 12000000 bytes		127035528 bytes	3*1000*1000*4 = 12000000 bytes	
Order	**Avg Time**	**Variance**	**Order**	**Avg Time**	**Variance**
IJK	3.961426	0.000003	IJK	3.958236	0.000025
IKJ	4.954249	0.006279	IKJ	4.904521	0.000691
JIK	4.177223	0.000025	JIK	4.170228	0.000001
JKI	3.379006	0.000063	JKI	3.374859	0.000000
KIJ	5.023502	0.000011	KIJ	4.846281	0.000030
KJI	3.393048	0.000055	KJI	3.393101	0.000102

As mentioned above, the experiment was conducted 20 times for each iteration. With the given results, it is clear that the variance tends to increase with the value of N. This can be the case for several reasons.

- When the matrix size increases, it's more likely to get cache misses and the number of cache misses and cache misses penalties can introduce more variability in execution time.
- When the matrix size increases, the processors can be fully utilized. At full utilization, there can be competition for resources such as the memory bus or arithmetic logic unit (ALU), which can lead to contention and cause more variability in the execution time.

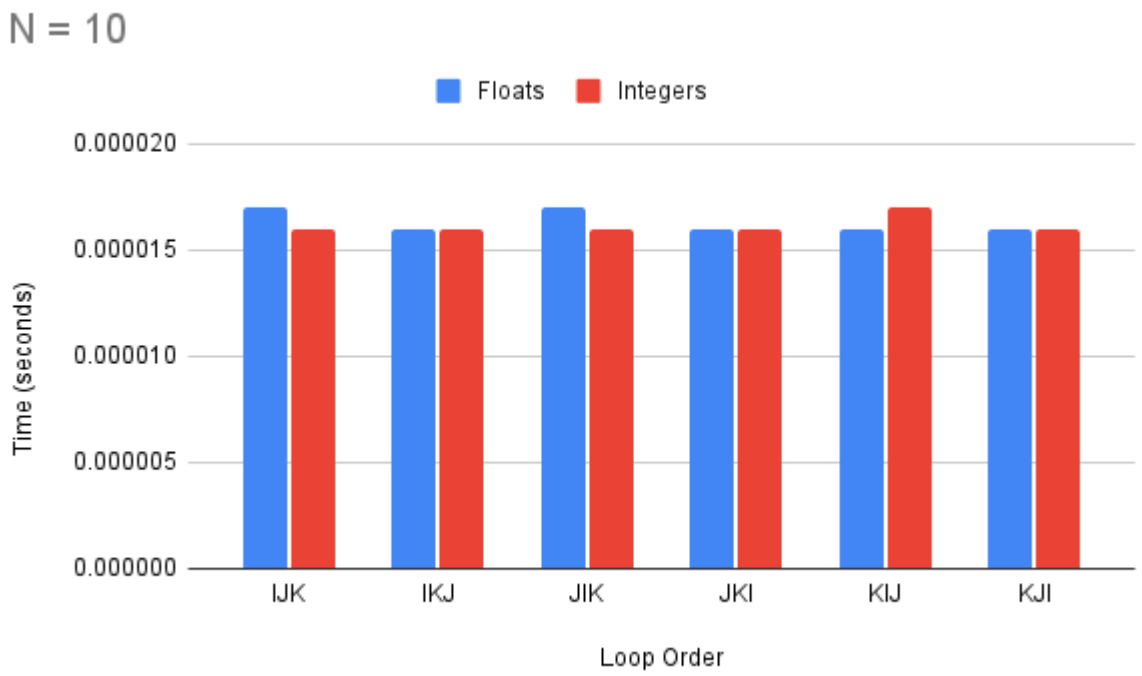
Valgrind Run

The written code for matrix multiplication was verified with the ***'valgrind'*** tool for memory leaks and it was found that there aren't any leaks possible. A sample log of a valgrind run is as follows.

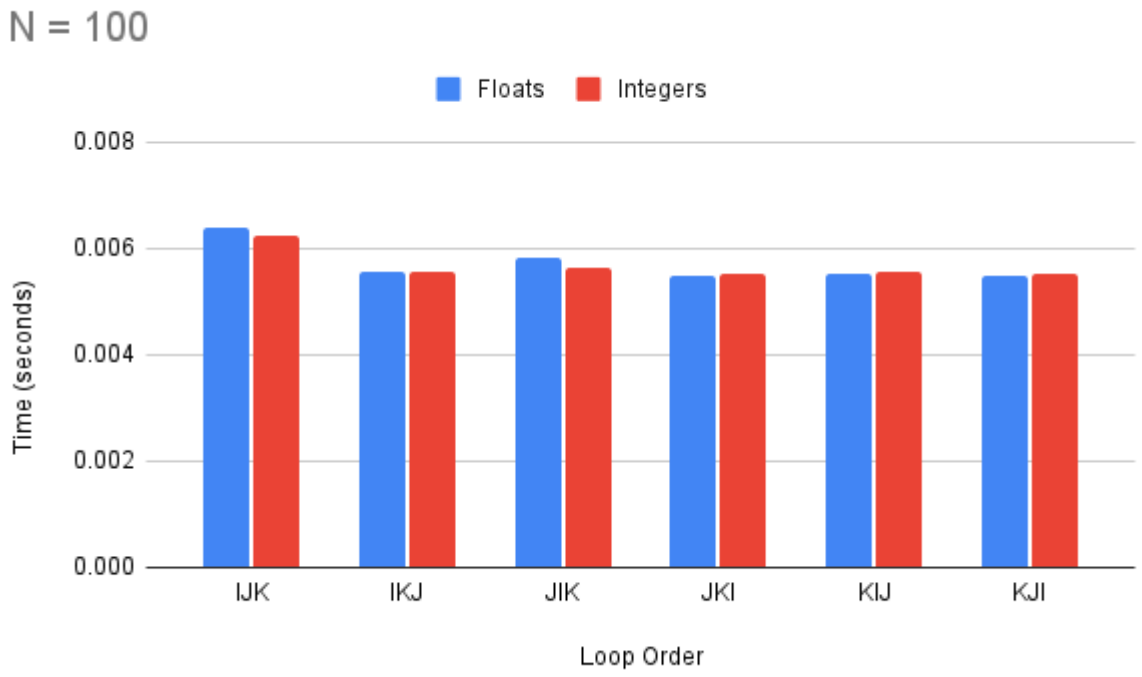
```
==3552083== Memcheck, a memory error detector
==3552083== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==3552083== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info
==3552083== Command: ./matMulFloat 100 20
==3552083== Parent PID: 3551992
==3552083==
==3552083==
==3552083== HEAP SUMMARY:
==3552083==     in use at exit: 0 bytes in 0 blocks
==3552083==   total heap usage: 5 allocs, 5 frees, 121,184 bytes
allocated
==3552083==
==3552083== All heap blocks were freed -- no leaks are possible
==3552083==
==3552083== For lists of detected and suppressed errors, rerun
with: -s
==3552083== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

1.2 Results Analysis

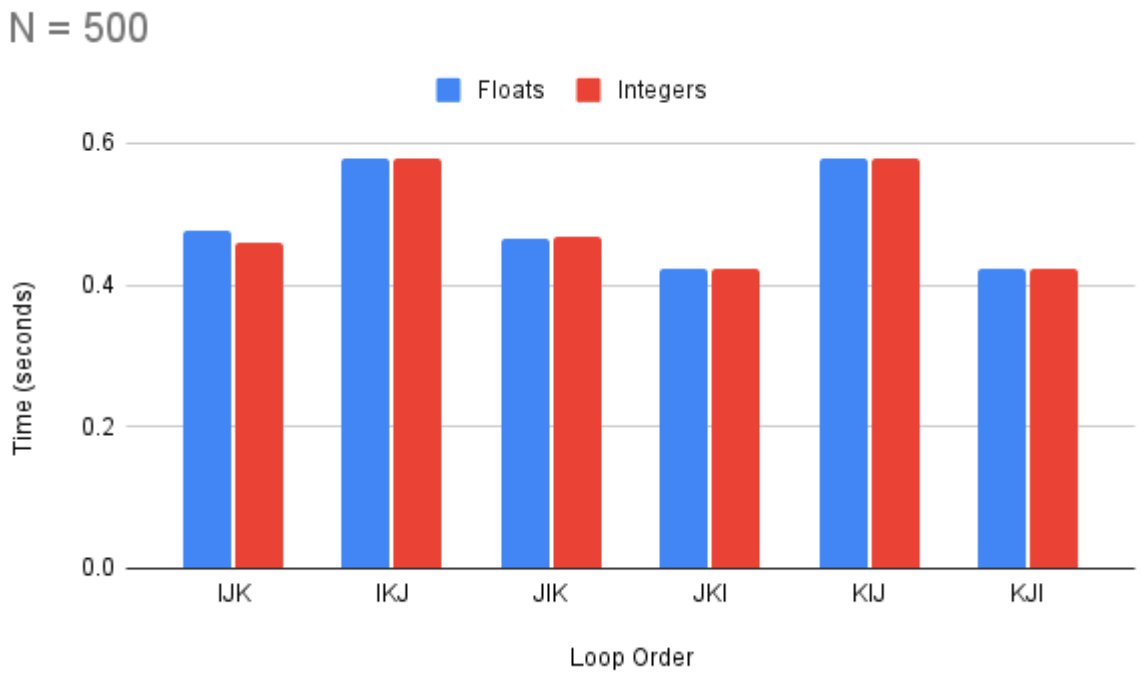
N = 10



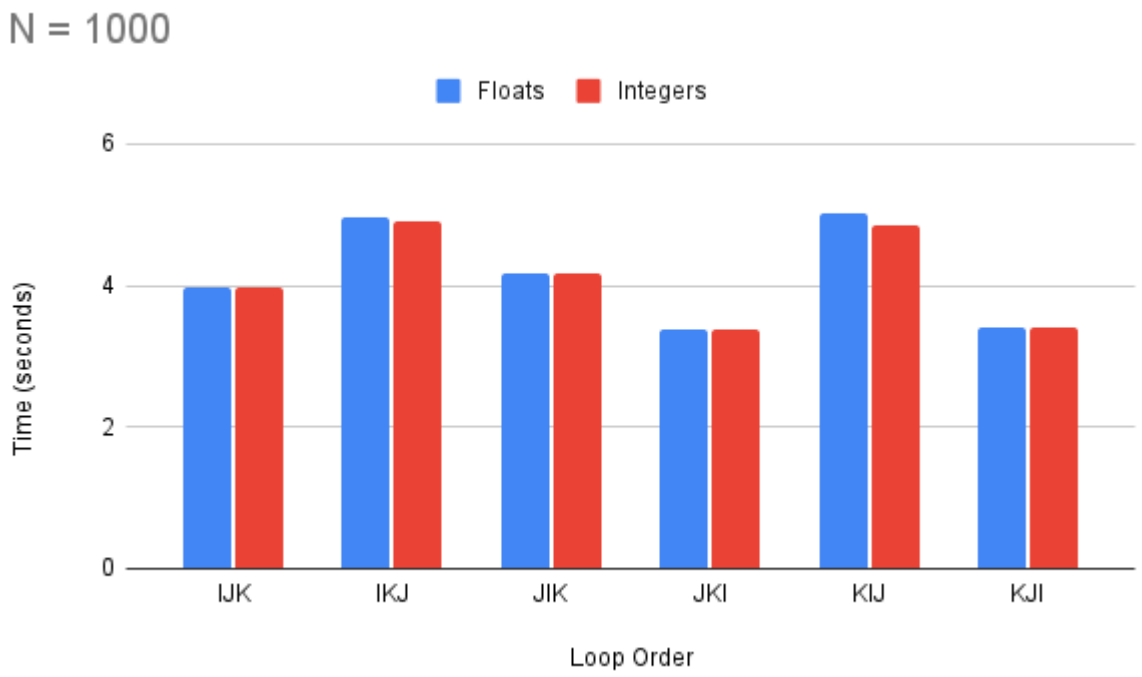
N = 100



N = 500



N = 1000



For small N (10, 100) values, it is clear that the difference between loop orders is negligible and every loop order performs the same for all the loop orders. But for large N (500, 1000) values, we can see that **JKI** and **KJI** significantly outperform all the other loop orderings.

Furthermore, the execution times for integers are a bit lesser than of floats but it is almost negligible. Even though, the size of floats and ints are the same in C, this time difference can be caused due to the fact that the floating-point arithmetic, which is used to perform operations on float data types, is typically more computationally expensive than integer arithmetic, which is used to perform operations on int data types.

1.3 Best Loop Orderings

JKI and **KJI** loop orders performed best out of all 6 six loop orders for N = 1000. Furthermore, JKI is slightly better than KJI and this is the case for both floats and integers. Thus, it is clear that loop orderings with ***'i'*** within the inner loop give the best performance overall. The way the matrices are implemented on memory and how they are cached while executing the operations are the two factors that affect the performance of different loop orderings.

The JKI and KJI loop orderings are generally faster because, for each iteration of the inner loop, the memory access point of the A iterates one by one [*A[i+k*n]*] (Since matrix A is represented by a contiguous block of 1D memory and it's in column-major). Thus, accessing a whole column of A at once, make it faster because the said memory locations are already cached due to being closer in the same contiguous memory block.

1.4 Worst Loop Orderings

IKJ and **KIJ** loop orders performed worst out of all 6 six loop orders for N = 1000. Furthermore, IKJ is slightly better than KJI and this is the case for both floats and integers. Thus, it is clear that loop orderings with ***'j'*** within the inner loop give the worst performance overall.

The IKJ and KIJ loop orderings are generally slower because, for each iteration of the inner loop, the memory access point of the B iterates n by n [*B[k+j*n]*] (Therefore, it jumps n by n and the cached memory block is useless. So, the memory should be loaded again). Thus, accessing a whole row of B at once, make it slower.

1.5 Available CPU Instruction Set Extensions

Using the `'lscpu'`, the following CPU instruction set extensions can be found in the “Flags” field.

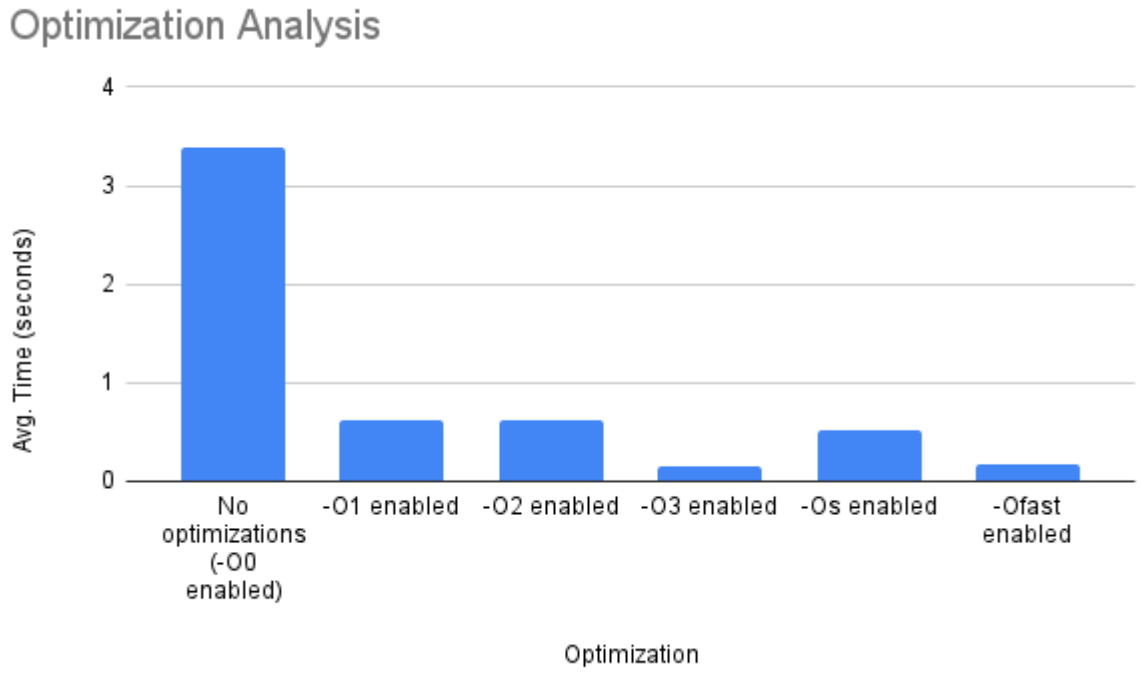
- sse
- sse2
- pclmulqdq
- vmx
- smx
- ssse3
- fma
- sse4_1
- sse4_2
- popcnt
- aes
- xsave
- avx
- f16c
- sha
- bmi1
- avx2
- bmi2
- avx512f
- avx512dq
- rdseed
- adx
- avx512ifma
- avx512cd
- sha
- avx512bw
- avx512vl
- xsaveopt
- xsave
- xsave
- avx512vbmi
- avx
- bmi2
- aes
- pclmulqdq
- avx
- avx
- avx
- popcnt

1.6 Alternative Further Optimizations

The JKI loop was run for N = 1000 for 20 times for each of the following optimizations. (Only the matrix multiplication of floats was considered.)

- *No optimizations (-O0 enabled)* - default
- *-O1 enabled* - tries to reduce code size and execution time
- *-O2 enabled* - all supported optimizations that do not involve a space-speed tradeoff
- *-O3 enabled* - turns on all optimizations specified by -O2 and a few more
- *-Os enabled* - Optimize for size
- *-Ofast enabled* - enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs

Optimization	Avg. Time (seconds)	Variance (seconds)
No optimizations (-O0 enabled)	3.379006	0.000063
-O1 enabled	0.62465	0.007686
-O2 enabled	0.623017	0.007565
-O3 enabled	0.154659	0.000860
-Os enabled	0.526042	0.007237
-Ofast enabled	0.171403	0.000453



According to the above graph, it is clear that **-O3 enabled** and **-Ofast** gives the vest performance and it is almost **20** times better than -O0 enabled (no optimizations). This is due to O3 enabling all the standard optimization parameters for a given compilation and Ofast enabling even non-standard optimizations.