MEMBENCH:

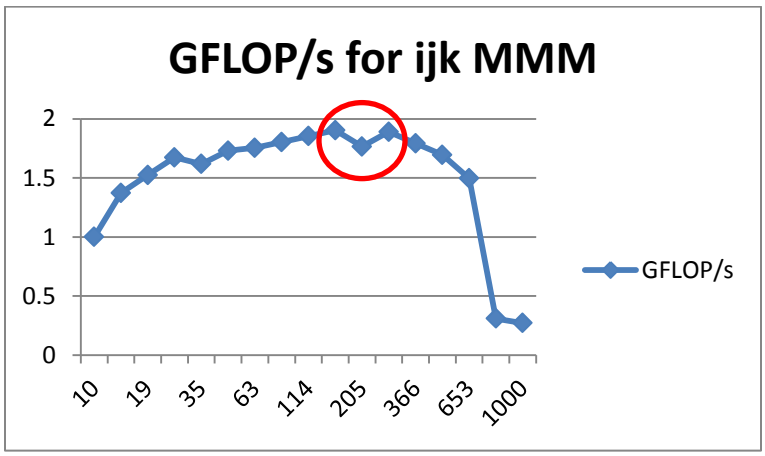
- For small strides, cache line reuse and prefetching keeps the processor well fed, so the memory overheads are relatively low. For larger strides, the cost starts to go up. When the stride is large enough, every data item we visit fits in L1 cache, and costs are again low.
- At stride 64, we can clearly see the four basic memory access times for a hit in L1, a hit in L2 , a hit in L3, and an access to main memory.
- So the line size of L1,L2 and L3 is 64 bytes.
- Till 64 KB, the access time remains at 1 ns for stride of 64. So L1 cache size = 32 KB.
- From 128 KB till 512 KB access time remains at 5 ns. So L2 cache size = 512 KB.
- Till 8 MB access time remains at 7 ns and for 16MB jumps to 13 ns.
- So L3 cache size should be around 8 to 10 MB. L3 cache size = 10 MB.
- At strides of 4k, there is a jump in access time across all matrix sizes. This is due to page size being 4 KB, thus every access results in a TLB lookup.
- Number of TLB entries can be calculated by observing a sudden spike at 4k stride.
- From 128 KB to 256 KB there is a sudden spike.
- Page size = 4 KB. 128 KB = 32 pages. 256 KB = 64 pages. [Either 32 entries or between 32 and 64]
- So our TLB size = 48 entries.
- At 32 M stride for 64 M matrix size, access time drops down again to 1 ns.
- So Associativity of L1 cache = 2 [64/32].

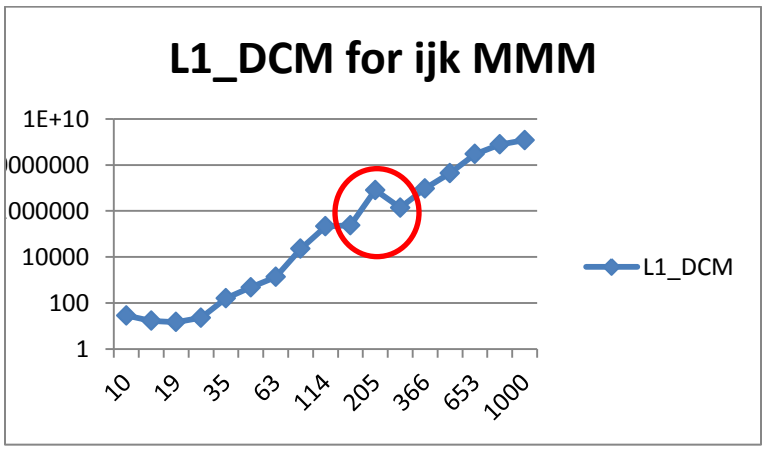
MATRIX-MATRIX MULTIPLICATION:

PART A: [ijk]

- By observing the plot for GFLOP/s for various values of n from 10 to 1000, performance increases and after some time it starts decreasing resulting in a “fly-over” shape.
- Initially though the L1 and L2 misses are low since the caches are not fully populated the computation to communication overhead is the bottleneck for low performance.
- With increasing n, performance increases hitting the max at n=153.
- Then it decreases from there on.
- But a **strange observation** at **n=206** is observed where a **‘v’ shape graph** is observed.
- The reason is **sudden increase in L1-Data cache misses for n=206** compared to the gradual increase which decreases at n=274 and increases gradually from there on.
- For large values of n, from n=366 till n=1000 the performance decreases due to conflict misses in L1 and L2 there by replacing the elements of the **A[I + k*n]** which has temporal locality across iterations and more **TLB Misses** as accessing every element with a larger stride results in a miss.

N	GFLOP/s	L1_DCM	L2_DCM	TLB_DM
10	1	29	10	0
14	1.372	17	4	0
19	1.524	15	4	0
26	1.674	23	0	0
35	1.618	163	18	0
47	1.73	485	29	0
63	1.755	1362	60	8
85	1.804	22735	152	0
114	1.853	214125	375	1
153	**1.903**	**238343**	5900	2318
205	**1.766**	**8124117**	49975	0
274	**1.89**	**1380239**	132653	184
366	1.793	9438253	343727	67214
489	1.695	43748609	857773	231915
653	1.496	2.97E+08	2232013	1272531
871	0.31	7.74E+08	3490469	**5.63E+08**
1000	0.272	1.19E+09	5189071	**9.79E+08**



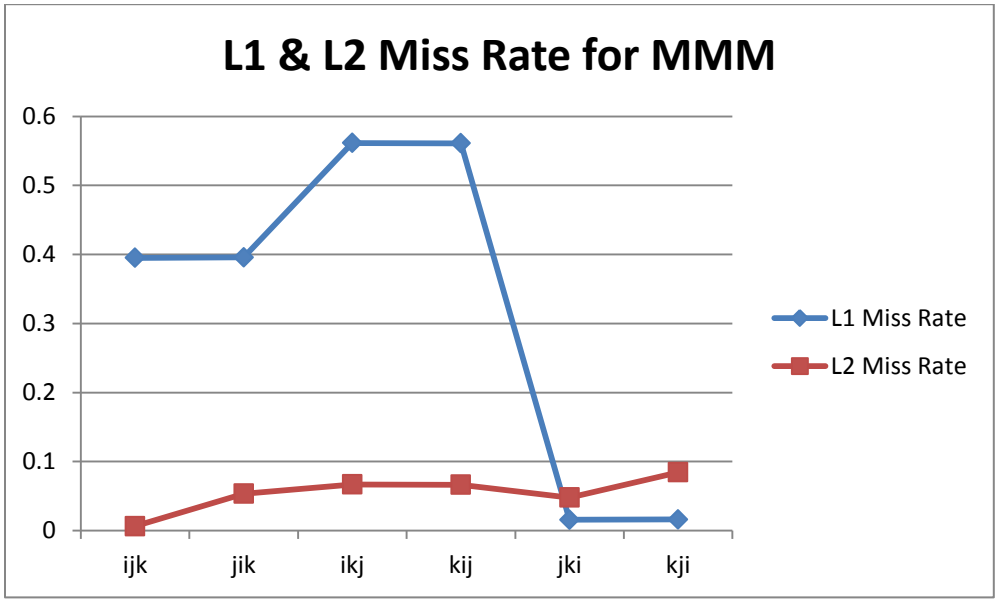


PART B:

- **'I'** as the innermost loop, '**jki**' and '**kji**' performs best.
- **'j'** as the innermost loop, '**kij**' and '**ikj**' performs worst.
- As we stride with increasing 'n' it results in "**KILLER STRIDES**".
- Though cache is not full, it results in conflict misses if the stride of access is a multiple of cache size resulting in accessing only one set of cache thereby replacing the same cache lines and resulting in a miss for every access. So **TEMPORAL/SPATIAL LOCALITY is not exploited**!!

MISS RATIO:[L1 and L2]:

LOOP	L1 LOADS	L1 STORES	L1 MISSES	L2 MISSES	L1 MR	L2 MR
ijk	2001802568	1000000032	1185702652	7612377	**0.39499688**	**0.00642**
jik	2001804412	1000000033	1187886212	63483037	**0.39572405**	**0.053442**
ikj	3001817534	1000000031	2247025673	150458043	**0.56150128**	**0.066959**
kij	3001855260	1000000032	2244667460	149092611	**0.5609067**	**0.066421**
jki	3000088180	1000000033	62759678	3000649	**0.01568957**	**0.047812**
kji	3000089525	1000000032	64772892	5466002	**0.01619286**	**0.084387**



LOOP ORDERING	C	A	B
Jki	**S & T**	**S**	**S & T**
Kji	S	S & T	S & T
Kij	No	S & T	No
Ikj	No	T	No
Ijk	S & T	No	S
Jik	T	No	S & T

3. MATRIX TRANSPOSITION:

A) Naive transposition for matrix of size 2000*2000: 34.702 milliseconds.

B) After blocking: For matrix size 2000*2000:

BLOCKSIZES	TIME (ms)
16	14.655
20	7.447
30	7.711
62	8.881
64	8.53
100	7.958
120	9.073
200	6.652
400	6.045
401	10.421
420	9.867
500	9.814
1000	32.963

C) For 2000*2000 Matrix, block size of 400 performs best and block size 1000 performs worst.

For matrix size 2048*2048:

BLOCKSIZE	TIME (ms)
22	16.831
30	18.507
32	19.485
64	21.66
128	21.56
256	21.419
400	32.394
512	32.392

- Block size 22 performs the best.
- The reason for smaller block size of 22 * 22 for 2048 * 2048 matrix size is due to conflict misses.
- This is because size of every line of the matrix is a multiple of the cache size.

D) Our algorithm works for block size of 30-by-30.