L1 and L2 miss rate, **total load and store instructions**, and **the number of committed floating point instructions**

	50x 50	100x 100	200x 200	400x 400	800x 800
i-j-k	L1 = 0.47% L2 = 15.40 % Load and Store = 380255 Total Floating Pt = 250031	L1 = 4.37 % L2 = 1.32 % Load and Store = 3020357 Total Floating Pt = 2000056	L1 = 4.42 % L2 = 69.20 % Load and Store = 24080563 Total Floating Pt = 16002949	L1 = 11.63 % L2 = 36.10 % Load and Store = 192321015 Total Floating Pt = 128124633	L1 = 28.28 % L2 = 9.42 % Load and Store = 1537284321 Total Floating Pt = 1076876194
j-i-k	L1 = 1.87 % L2 = 2.75 % Load and Store = 389565 Total Floating Pt = 250233	L1 = 4.58 % L2 = 1.84 % Load and Store = 3048783 Total Floating Pt = 2000046	L1 = 4.43 % L2 = 8.04 % Load and Store = 24184602 Total Floating Pt = 16004940	L1 = 10.55 % L2 = 3.29 % Load and Store = 192726270 Total Floating Pt = 128203858	L1 = 28.17 % L2 = 0.38 % Load and Store = 1538891993 Total Floating Pt = 1061902218
j-k-i	L1 = 0.86 % L2 = 1.83 % Load and Store = 634216 Total Floating Pt = 254012	L1 = 5.56 % L2 = 0.56 % Load and Store = 5028434 Total Floating Pt = 2049128	L1 = 10.99 % L2 = 19.57 % Load and Store = 40104259 Total Floating Pt = 16522886	L1 = 34.83 % L2 = 5.50 % Load and Store = 320406237 Total Floating Pt = 142445529	L1 = 32.75 % L2 = 8.98 % Load and Store = 2561614466 Total Floating Pt = 1137709712
k-j-i	L1 = 0.33 % L2 = 1.04 % Load and Store = 631775 Total Floating Pt = 254708	L1 = 5.12 % L2 = 0.56 % Load and Store = 5018545 Total Floating Pt = 2056176	L1 = 10.83 % L2 = 21.31 % Load and Store = 40064465 Total Floating Pt = 16502609	L1 = 32.88 % L2 = 5.55 % Load and Store = 320246642 Total Floating Pt = 142582384	L1 = 37.73 % L2 = 8.95 % Load and Store = 2560975293 Total Floating Pt = 1150567056
i-k-j	L1 = 0.81 % L2 = 8.32 % Load and Store = 199683 Total Floating Pt = 125001	L1 = 7.10 % L2 = 0.25 % Load and Store = 1539351 Total Floating Pt = 1000029	L1 = 7.86 % L2 = 2.02 % Load and Store = 12146067 Total Floating Pt = 8000037	L1 = 7.84 % L2 = 2.14 % Load and Store = 96569508 Total Floating Pt = 64000288	L1 = 7.72 % L2 = 3.50 % Load and Store = 770256516 Total Floating Pt = 512037782
k-i-j	L1 = 0.73 % L2 = 3.62 % Load and Store = 201822 Total Floating Pt = 125000	L1 = 7.64 % L2 = 0.66 % Load and Store = 1548640 Total Floating Pt = 1000860	L1 = 7.48 % L2 = 16.35 % Load and Store = 12184654 Total Floating Pt = 8000613	L1 = 7.12 % L2 = 19.29 % Load and Store = 96726703 Total Floating Pt = 64007498	L1 = 7.03 % L2 = 19.15 % Load and Store = 770890915 Total Floating Pt = 512080943

	1200x 1200	1600x 1600	2000 x2000
i-j-k	L1 = 38.55 % L2 = 9.44 % Load and Store = 5186891353 Total Floating Pt = 3543035715	L1 = 40.94 % L2 = 15.43 % Load and Store = 12293145185 Total Floating Pt = 8416377870	L1 = 33.26 % L2 = 28.64 % Load and Store = 24008048745 Total Floating Pt =16911616271
j-i-k	L1 = 38.23 % L2 = 2.31 % Load and Store = 5190501493 Total Floating Pt = 3563354487	L1 = 40.49 % L2 = 9.42 % Load and Store = 12299588171 Total Floating Pt = 8670921471	L1 = 34.86 % L2 = 23.87 % Load and Store = 24018063887 Total Floating Pt = 17420873604
j-k-i	L1 = 40.39 % L2 = 24.21 % Load and Store = 8643631440 Total Floating Pt =3981665583	L1 = 42.85 % L2 = 51.94 % Load and Store = 20486464419 Total Floating Pt = 9641500900	L1 = 39.81 % L2 = 69.75 % Load and Store = 40010119030 Total Floating Pt = 19221881386
k-j-i	L1 = 40.19 % L2 = 24.90 % Load and Store = 8642192739 Total Floating Pt = 3964248558	L1 = 42.55 % L2 = 52.50 % Load and Store = 20483905955 Total Floating Pt = 9553691467	L1 = 39.47 % L2 = 69.98 % Load and Store = 40006121245 Total Floating Pt = 18987827231
i-k-j	L1 = 6.26 % L2 = 3.73 % Load and Store = 2597064018 Total Floating Pt = 1728768387	L1 = 6.13 % L2 = 4.24 % Load and Store = 6152992251 Total Floating Pt = 4097726601	L1 = 6.43 % L2 = 3.57 % Load and Store = 12014041281 Total Floating Pt = 8003903911
k-i-j	L1 = 6.57 % L2 = 19.65 % Load and Store = 2598496419 Total Floating Pt = 1731108451	L1 = 6.25 % L2 = 22.06 % Load and Store = 6155543409 Total Floating Pt = 6155543409	L1 = 6.96 % L2 = 22.26 % Load and Store = 12018031923 Total Floating Pt = 8014293705

Part 1

- For the smallest matrix size, do the L1 and L2 miss rates vary for the different loop-order variants? Do they vary for the larger matrix sizes? Is there any difference in behavior between the different problem sizes? Can you explain intuitively the reasons for this behavior?
- **Answer:**
    - **The miss rates do vary between the different loop invariants. IKJ has the lowest overall L1 and L2 miss rates and KJI and JKI have the highest miss rates among all the invariants. Additionally, for larger sized matrices, regardless of the invariant the L1 and L2 cache miss rate increases with size.**
    - **One reason for this could be the fact that in KJI and JKI the accesses to the result matrix "result[i][j]" are made in column major order. This means that it is traversing an entire column and moving on to the next. However, in C matrices are stored in row major order, which means that iterating over the rows will naturally cause fewer cache misses than iterating over the columns.**
    - **Additionally, accessing the matrix in column order does not take advantage of spatial locality which makes a lot of difference when comparing it to the variants that do.**
- Re-instrument your code by removing PAPI calls, and using clock_gettime with CLOCK_THREAD_CPUTIME_ID to measure the execution times for the six versions of MMM and the eight matrix sizes specified above. How do your timing measurements compare to the execution times you obtained from using PAPI? Repeat this study using CLOCK_REALTIME. Explain your results briefly.
- **Answer:**
    - **CLOCK_THREAD_CPU_ID results with PAPI vs without PAPI: The difference in measurements with PAPI and without PAPI are the same for every invariant and matrix size. The important thing is the fact that these differences make up a larger portion of the time for the smaller matrix sizes like 50 or 100 than the larger ones like 1600 and 2000.**
    - **This is the case since PAPI always takes the same time to run regardless of the size of the matrix and the invariant it is running on, the only thing that might change is the time taken for measuring different PAPI values such as PAPI_L1_DCM vs PAPI_TOT_CYC.**

- **CLOCK_THREAD_CPUTIME_ID vs CLOCK_REALTIME : CLOCK_REALTIME leads to lower time taken than CLOCK_THREAD_CPUTIME_ID and this difference is more pronounced for larger matrix sizes.**
- **The reason is because when the matrix gets larger the multiplication takes more time which provides more opportunities for a context switch event, since the process will run for its scheduling quantum and if the process takes a lot of time then it takes more quantums for the process to run to completion, which means more quantums where it isn't running but being measured as well , hence the difference in real time and thread time increases .**
- **Additionally, processes that do not interact with the user for input regularyt are scheduled with lower priority by scheduling systems like multilevel feedback queues, which could be another reason for higher differences when the program takes a long time.**

Part 2

- What are *data and control dependencies?* Give simple examples to illustrate these concepts.
- **Answer:**
  - **Data Dependencies**
    - **Data dependencies occur when there exist two instructions such that one executes after the other, they both access the same data and one of the accesses is a write.**
    - **An example of a data dependence could be the instructions i1 : movq %r1 %r2 i2 : addq %r2 %r3/. In These instructions both access r2 and write to r2 , if they execute out of the order that they were written in then there would be a logical error to the code and it won't do what the programmer expects.**
  - **Control Dependence**
    - **Control dependencies occur in situations where an instruction(i1) will only execute if a previous instruction(i2) evaluates to a value that allows i1 to execute.**
    - **So in the following example, the instruction "i = 5" only executes conditionally if i > 40 evaluates to true. So regarding the definition "i=5" is i1 and "i > 40" is i2.**
- Explain *out-of-order execution* and *in-order retirement/commit*. Why do high-performance processors execute instructions out of order but retire them in order?
- **Answer:**

    - **Out of order execution is a good strategy for processors to pipeline instructions and be able to execute the program much faster than if it was in order and sequential. The processors will pipeline instructions together and due to optimizations taken some instructions that were written in the program to come after another may come before and vice versa.**
    - **However the reason there is out-of-order execution by in order retirement is to maintain an architectural state. If the program has an error or something goes wrong to stop the program, the programmer should be able to take a look at the registers and debug what went wrong. But if instructions are being committed out of order, then the programmer will not be able to debug since the state of the program cannot be determined.**
    - **Another thing to keep in mind with out of order execution is dependences, if certain instructions depend on the return value of others(data and control dependencies), then, they cannot be executed before them.**
    - **The reason they execute instructions out of order is to pipeline instructions together and increase the performance of a program, however in-order retirement/commit is necessary to maintain architectural state and allow for debugging of the program.**
- What hardware structure(s) are used to implement in-order retirement?
- **Answer:**
    - **Out of order execution is helped by a few things which allow them to execute out of order but end up actually committing the instructions in the correct order, these begin the ROB (reorder buffer), register renaming and branch prediction.**
    - **The ROB is a buffer of instructions where instructions enter the buffer and leave in the correct order of execution. Destination registers are only updated when instructions are retired from the ROB, instructions in the ROB only execute when the operands are updated from previous instructions. After writing to a register the instructions will signal other ROB instructions that the register had been updated. These invariants allow the ROB to assist in out of order execution and in order retirement.**
    - **Register renaming involves the use of two register files, one is the architecture which is visible to programmers and ISA, what should be happening and what is checked for debug purposes. It also has the physical register file, which is a larger set of registers that hold values for inflight instructions, the hardware manages this and these registers are the ones used by the instructions in the ROB in the speculative state. The architectured registers are committed to when the instructions leave the ROB but when they are still in the ROB they only modify the physical registers which allows them to execute instructions out of order, but still end up with in order retirement and maintenance of architectural state.**
    - **Lastly, branch prediction is a hardware rule that when a branch is fetched in an if statement, change the program counter to that branch and only switch if the prediction is wrong. This required hardware support that can**

predict if a branch exists, if it is taken and if it is the right one. This is done by the BPU.