Assignment 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?

- For the smallest matrix size (50×50), L1 and L2 cache miss rates show some variation among different loop orders, though differences are relatively minor. The `ijk` and `jik` orders generally have lower L1 miss rates, while L2 miss rates vary significantly, with `ijk` often showing much higher misses compared to `jik` and `ikj`. As matrix sizes increase, L1 and L2 miss rates rise for all loop orders. The increase in miss rates is due to the data size exceeding cache capacity, which leads to more frequent cache evictions and reloads. Loop orders like `ijk` show consistently higher miss rates, while more efficient orders like `ikj` demonstrate better performance.
- The differences in cache miss rates between various loop orders become more pronounced with larger matrices. Smaller matrices often fit within or close to the cache's capacity, resulting in less noticeable variations in cache performance. However, as matrices grow, inefficient access patterns become more problematic, causing greater variations in miss rates. This behavior is primarily due to the larger matrices exceeding cache capacity and the impact of different access patterns on cache efficiency. Orders that access memory contiguously, such as `ikj`, are better at utilizing cache and thus show lower miss rates.

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.

- When I re-instrumented my code to use `clock_gettime` with `CLOCK_THREAD_CPUTIME_ID` and `CLOCK_REALTIME` instead of PAPI for measuring execution times, I noticed that the results were generally consistent with only minor differences. For example, the execution times I measured with `CLOCK_THREAD_CPUTIME_ID` were very close to those from PAPI, with only slight variations (like PAPI's 269 microseconds compared to `CLOCK_THREAD_CPUTIME_ID`'s 250 microseconds for the `ijk` variant with a 50×50 matrix). Similarly, the times recorded with `CLOCK_REALTIME` were also close to the PAPI timings, though there were small differences due to its inclusion of wall clock time (like 264 microseconds with `CLOCK_REALTIME` versus 269 microseconds with PAPI). Overall, these minor discrepancies showed that both `CLOCK_THREAD_CPUTIME_ID` and `CLOCK_REALTIME` provide reliable performance measurements, reflecting different aspects of execution time but remaining consistent with PAPI.

What are *data and control dependences?* Give simple examples to illustrate these concepts.

- Data dependence is when an instruction relies on the result of another instruction.
  - **True dependence (Read After Write - RAW)**: Instruction B reads a value written by instruction A.
  - **Anti-dependence (Write After Read - WAR)**: Instruction A reads a value, and instruction B writes to it later.
  - **Output dependence (Write After Write - WAW)**: Both instructions write to the same location.
  - ***For example,*** in `x = a + b; y = x + 1;`, the second instruction depends on the result of the first because `x` is used in the second.

- **Control dependence** occurs when the execution of an instruction depends on the outcome of a previous branch instruction. ***For example***, in an `if` statement, `if (x > 0){ y = 1;}` the assignment to `y` depends on the condition `x > 0`.

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? What hardware structure(s) are used to implement in-order retirement?

- Out-of-order execution lets a processor run instructions as soon as their data is ready, without waiting for previous instructions to finish. This boosts performance by keeping the CPU busy, especially when some instructions are delayed. However, the processor retires or commits instructions in the original program order to make sure everything works as expected. If instructions were retired out of order, the program could behave incorrectly, with later instructions overwriting results that earlier instructions hadn't finished producing yet.

- To handle this, high-performance processors use a **Reorder Buffer (ROB)**. The ROB tracks all the instructions that have been executed out of order and holds onto their results until all the prior instructions are done. Once it confirms that all previous instructions have finished, it commits the results in the right order. This system keeps the CPU fast while making sure the program's results are correct.

Consider the invariants for retirement in the OOO execution with renaming shown in

slide 28 of the lecture slides. Why do we need to check the condition "(R3.PR# = ROB[n].PR#") before updating R3.v ? Explain what would happen if we did not check this condition before updating R3.v?

- The condition `(R3.PR# = ROB[n].PR#)` ensures that the update to `R3.v` is based on the correct physical register associated with `R3`. In out-of-order execution, multiple physical registers may map to the same logical register, and we need to ensure that the value being written is from the most recent instruction. Without this check, we might overwrite `R3.v` with an old or incorrect value, leading to erroneous results in the program. This check maintains the accuracy and correctness of the program by ensuring that only the latest valid value is committed.