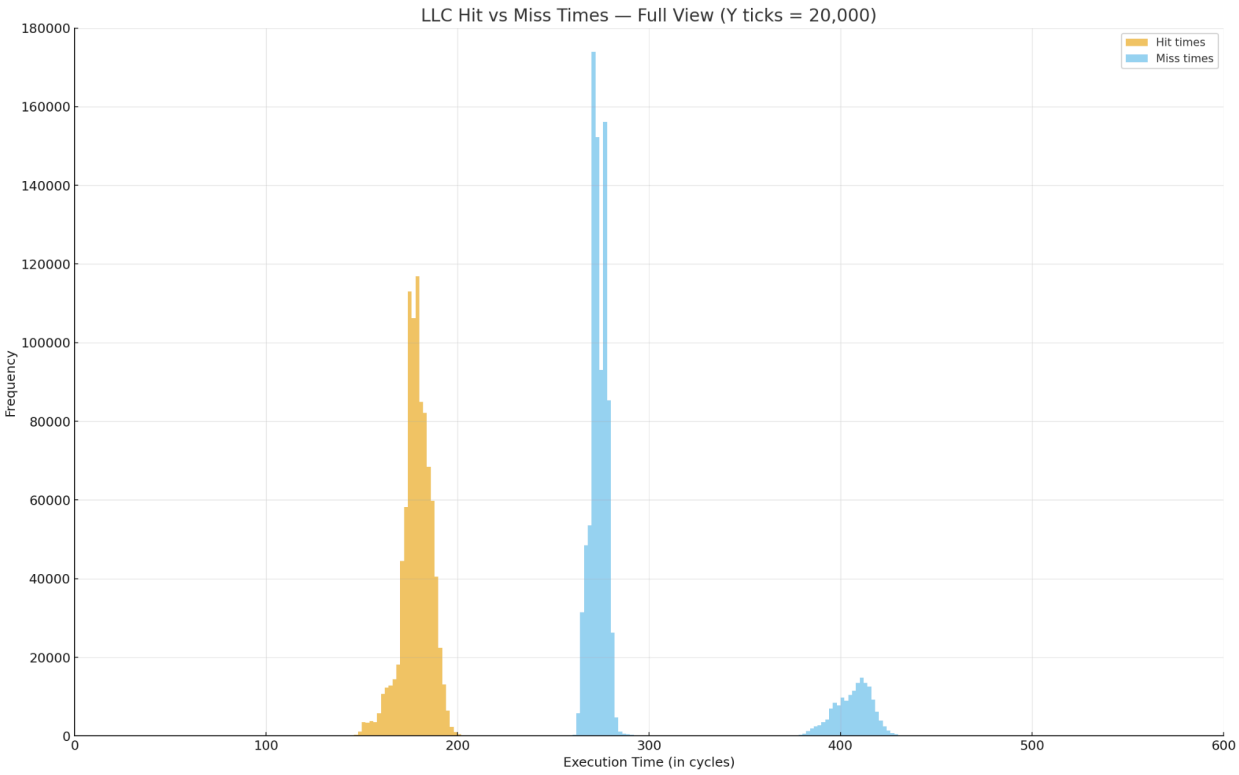
HOMEWORK 1 REPORT

Name: Sowmya Macheri Balaji **Student ID:** 200601527

Exercise - 1: Time a cache hit and miss, report the results. Repeat 1M times and plot a distribution of data collected for misses and hits.



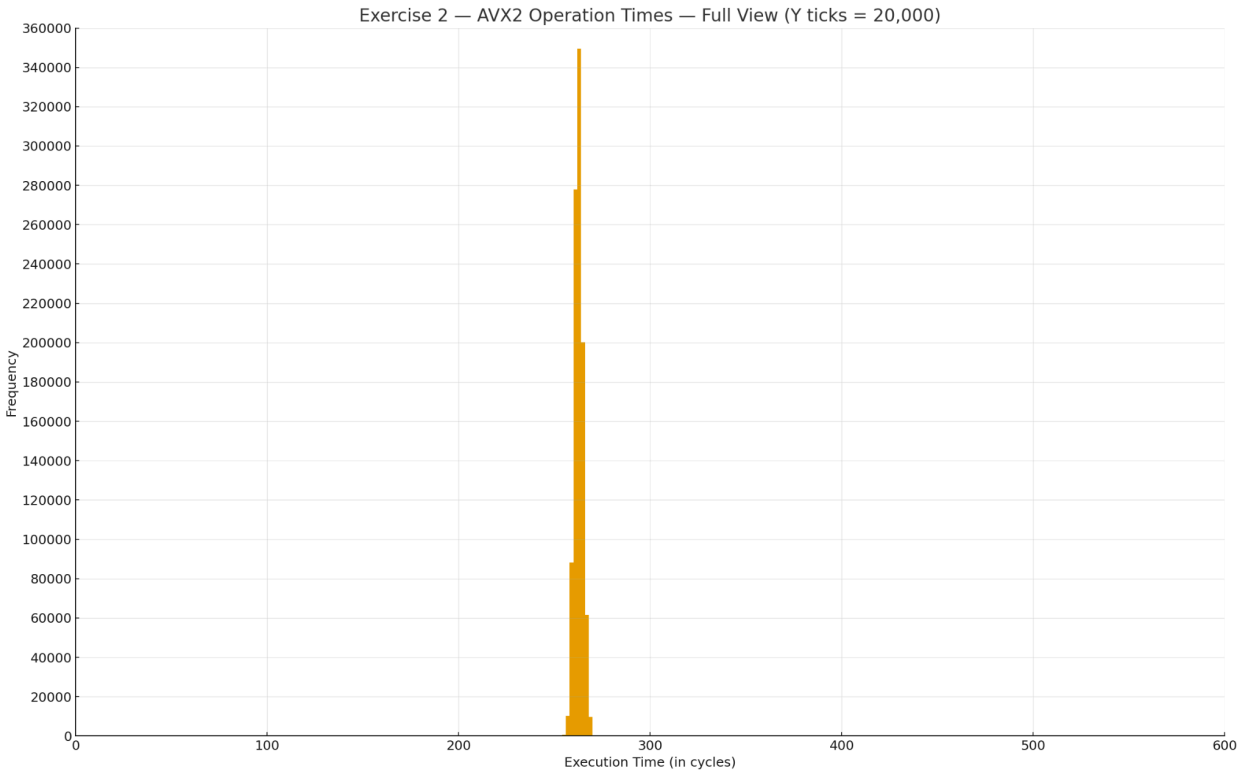
The program times a single memory access inside a tiny fenced window using RDTSCP before and after, so the result is in CPU cycles. It prepares two working sets: a small array that fits in the last-level cache (hit case) and a large array that exceeds it (miss case). Data are cache-aligned, and the access result is written to a volatile sink so the compiler can't delete or move it. Each iteration records `(t1-t0)` to CSV (one column, `cycles`) for the hit file and for the miss file.

Hit: mean ≈ **178.28** cycles

Miss: mean ≈ **295.43** cycles

Expectation: hits ≪ misses. **Observed:** exactly that; large gap, miss multimodality.

Exercise - 2:Time any AVX2 operation.



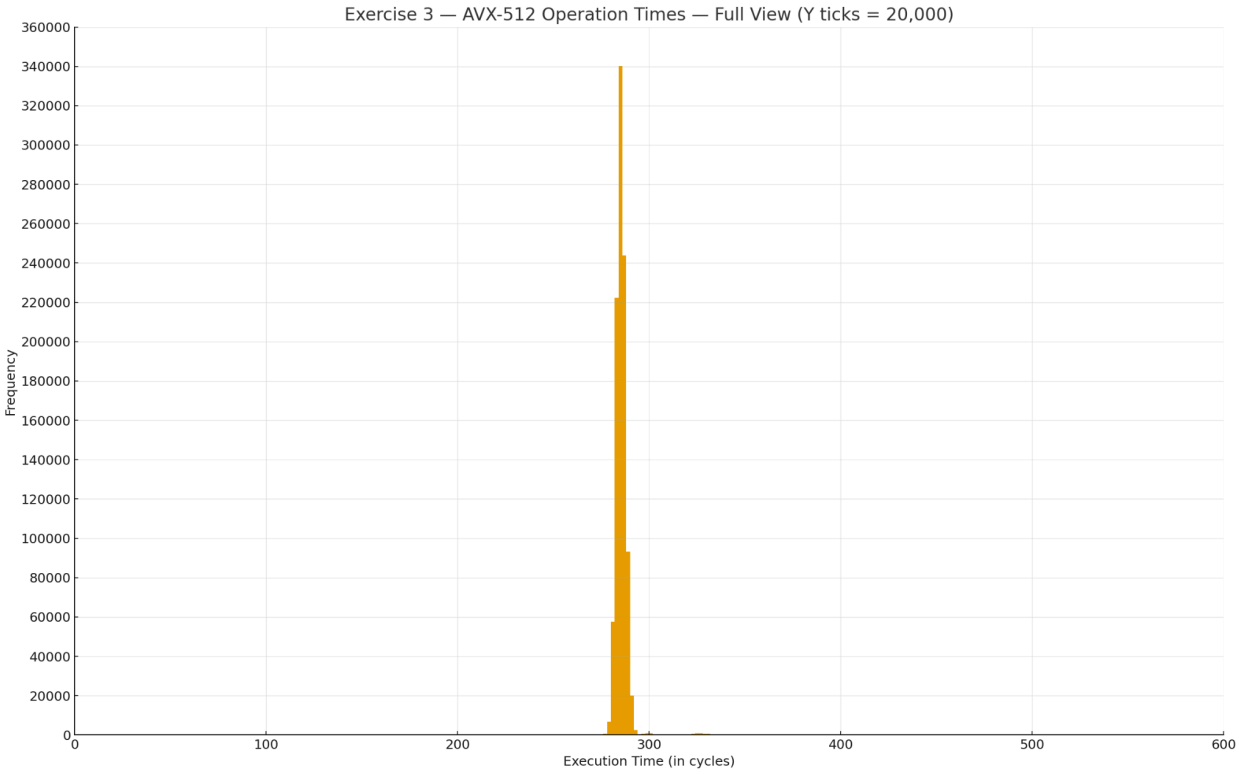
After checking OS/CPU support for AVX state via CPUID + `_xgetbv`, the code loads two aligned `__m256` vectors and measures exactly one `_mm256_mul_ps` inside the RDTSCP window with an `_mm_mfence()` to reduce reordering. The product is stored and a volatile read prevents dead-code elimination. Each loop writes one cycle count to `avx2_ex2.csv` (`cycles`). This yields a clean steady-state per-operation latency for 256-bit SIMD on your CPU.

Mean: ≈ 262.77 cycles

Expectation: could be longer under noise

Observed: smaller/steady latency—tight cluster, low variance.

Exercise - 3:Time any AVX512 operation.

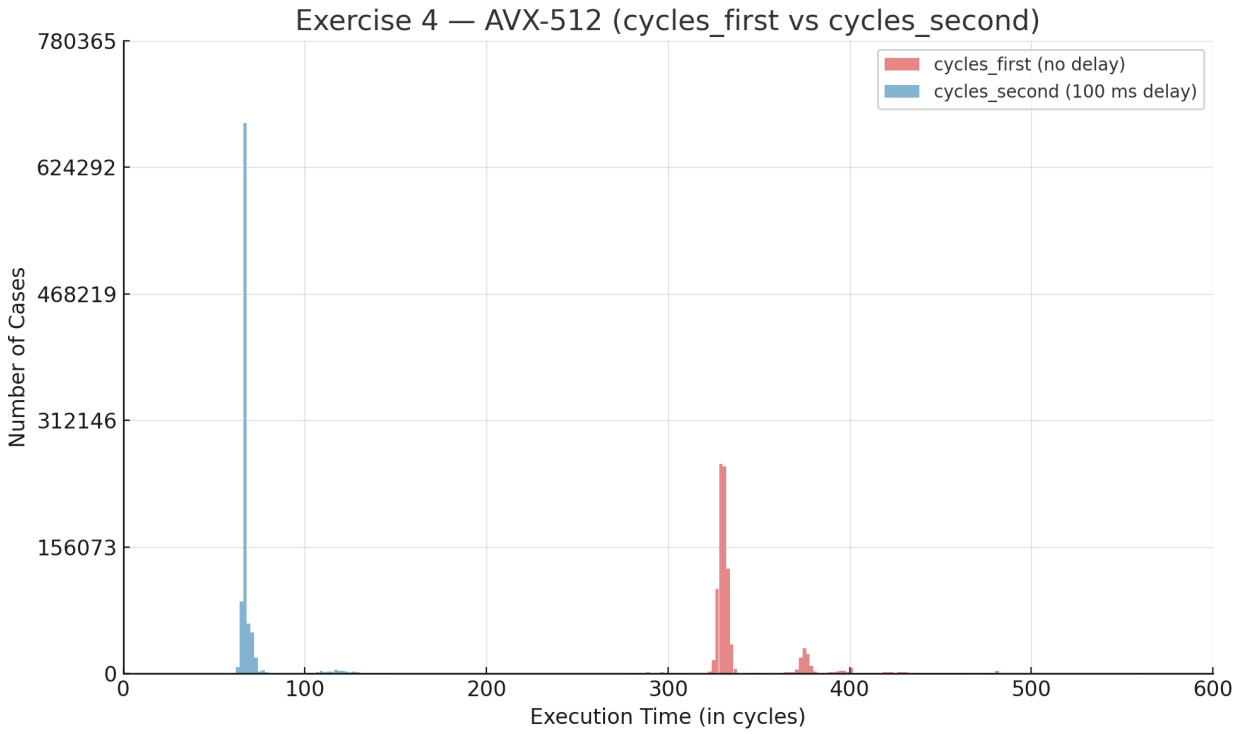


This mirrors Exercise 2 but for 512-bit SIMD: it verifies AVX-512F (and ZMM state) with CPUID/XCR0, aligns data to 64 bytes, loads `__m512` vectors, and times one `_mm512_mul_ps` between RDTSCP reads with fencing. The result is stored and touched to keep the compiler honest, and each iteration appends a single `cycle` value to `avx512_ex3.csv`. The outcome is a steady-state baseline for one 512-bit multiply.

Mean: ≈ 285.78 cycles

Expectation: a bit higher than AVX2; **Observed:** slightly longer than AVX2, still tight.

Exercise - 4:Time an AVX512 multiplication, wait for 100 ms (for example with usleep), and time another AVX512 multiplication. Is there a timing difference? Repeat 1M times and plot a distribution of data.

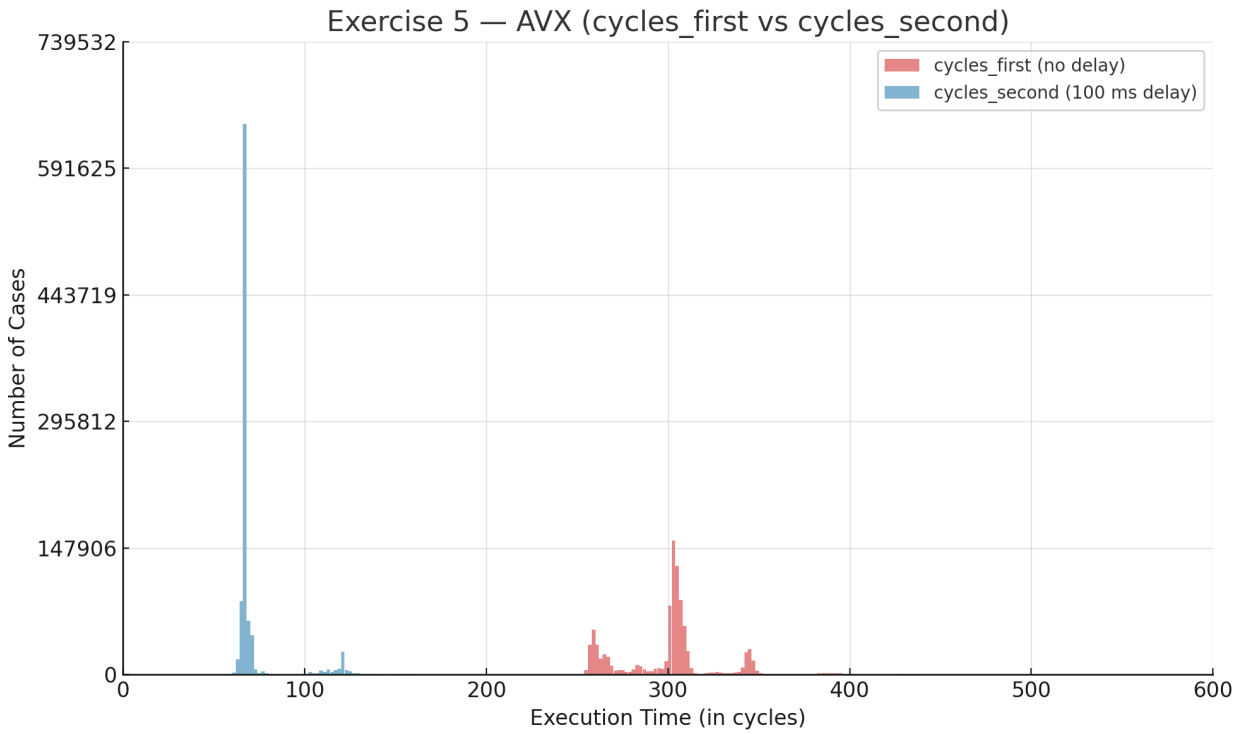


Using the same AVX-512 setup, each iteration performs two measured multiplies. First, it times one `_mm512_mul_ps` immediately (no delay) and saves `cycles_first`. Then it calls `usleep(100000)` (100 ms) to let the core's frequency/power state recover, and times the same multiply again, saving `cycles_second`. Both are written as a CSV row (`cycles_first,cycles_second`) to `avx512_rest_ex4.csv`, exposing how a rest period changes AVX-512 latency on your CPU.

- **No-delay (first):** mean ≈ **354.58**
- **After 100 ms (second):** mean ≈ **76.11**

- **Expectation:** after-rest faster. **Observed: much smaller** latency after rest (big separation).

Exercise - 5: Time an AVX multiplication, wait for 100 ms, and time another AVX multiplication. Is there a timing difference? Repeat 1M times and plot a distribution of data.



Identical to Exercise 4 but using `_mm256_mul_ps`. Each iteration records `cycles_first` (no delay) and `cycles_second` (after 100 ms) to `avx_rest_ex5.csv`. Comparing these two columns shows the (typically smaller) state/frequency effect for 256-bit AVX relative to AVX-512.

- **No-delay (first):** mean ≈ **299.18**
- **After 100 ms (second):** mean ≈ **69.40**

- **Expectation:** after-rest faster. **Observed: smaller** latency after rest (gap smaller than Ex4).

Github link: https://github.com/SowmyaMB30/ECE592-SMACHER