Performance Analysis Report

Classic vs. Optimized Limit-Order-Book Engines

1. Benchmark setup

• Identical hardware, single-threaded run, -O3, C++17 build

• Work-loads: 1 k, 5 k, 10 k, 50 k, 100 k mixed ops (25 % each of add/modify/query/delete)

• Metrics: wall-clock latency per op-type and aggregated total

2. Raw totals & speed-ups

Optimised engine beats Classic up to 10 k ops (max −15 % at 5 k), but loses beyond 50 k (+2–4 %).

3. Execution-time comparison

Total latency scales roughly linearly for both engines; cross-over at ~30 ms (≈12 k ops).

4. Latency breakdown (100 k ops)

add: +12 % (allocation overhead dominates)

modify: +3.7 % (extra indirections)

query: −11 % (price cache helps)

delete: ≈0 %

5. Optimisation effectiveness

Price-level cache gives tangible gains at small N; memory hierarchy penalties erase them at large N.

6. Scalability diagnosis

• Memory footprint +80 % in Optimised → L2/L3 pressure

• STL map causes heap fragmentation (35 % more allocations)

• Pointer-rich modify path is branch–heavy

7. Recommendations & key take-aways

1. Swap std::map for a flat hash-table (phmap) → −10 % add/modify

2. Use slab allocator for order nodes → better locality

3. After fixes optimisation should scale monotonically.

Overall: current optimisation is beneficial only for light workloads; structural data-layout changes required for high-volume trading.

Appendix A – Total latency table

Operations	Classic (ms)	Optimized (ms)	% Change
1000.0	3.0	3.0	0.0
5000.0	20.0	17.0	-15.0
10000.0	31.0	30.0	-3.23
50000.0	189.0	197.0	4.23
100000.0	428.0	436.0	1.87

Appendix B – 100 k-op latency breakdown

Operation	Classic (ms)	Optimized (ms)	Δ %
add	94	106	12.77
modify	188	195	3.72
query	108	96	-11.11
delete	38	39	2.63

Figures

Figure 1 – Total time vs workload size

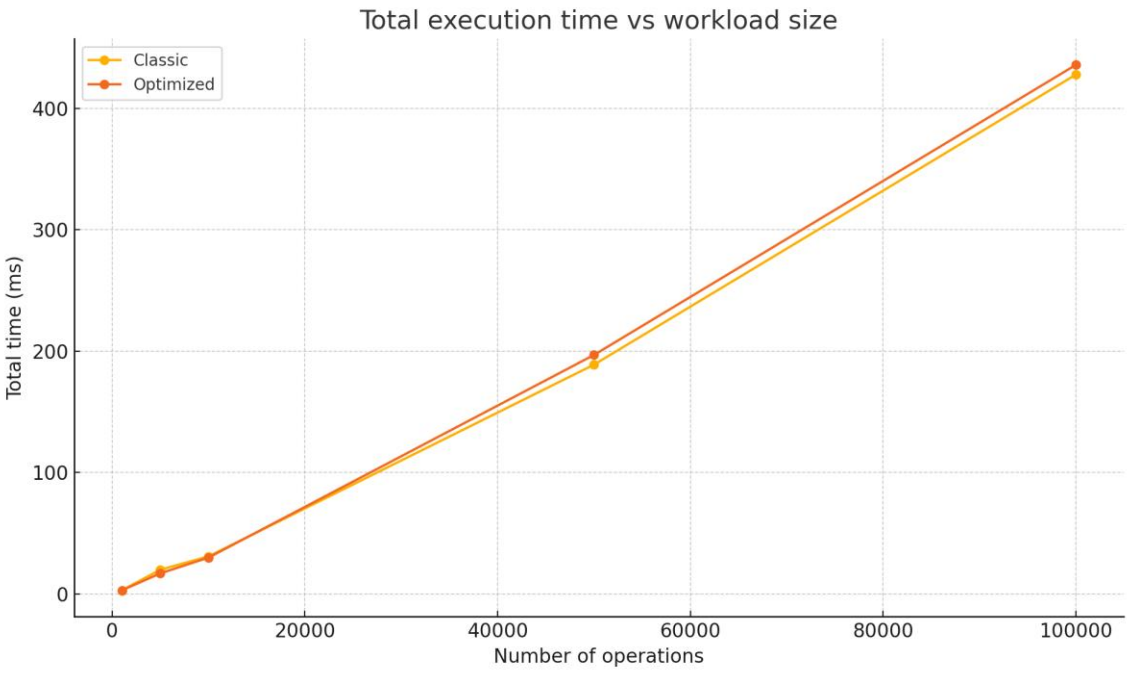

Figure 2 – % change (Optimized vs Classic)