
Programming Assignment-1

Task 1

Matrix Multiplication — The Rancho Way

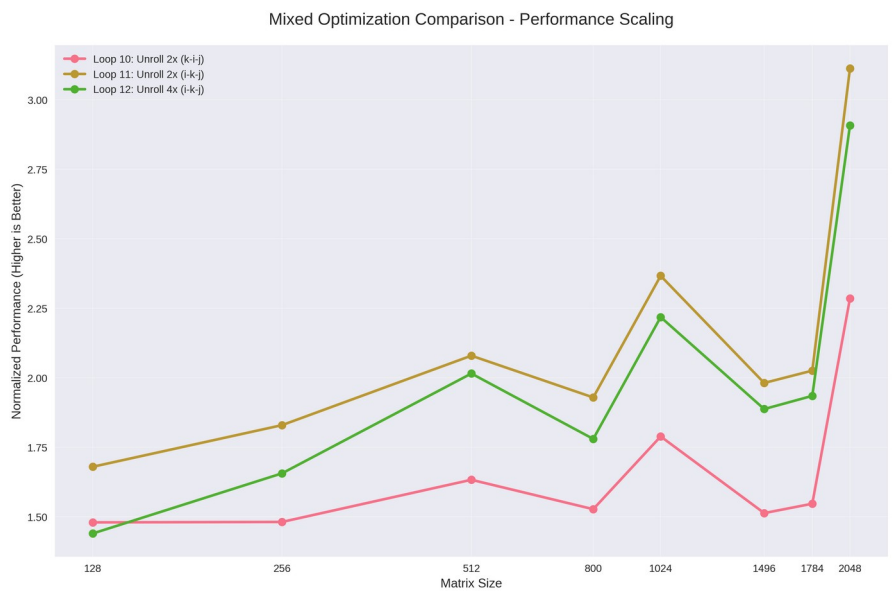
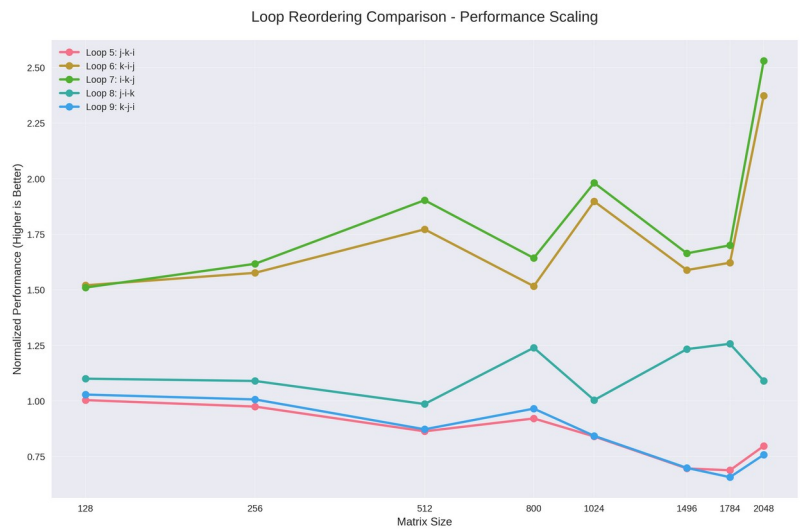
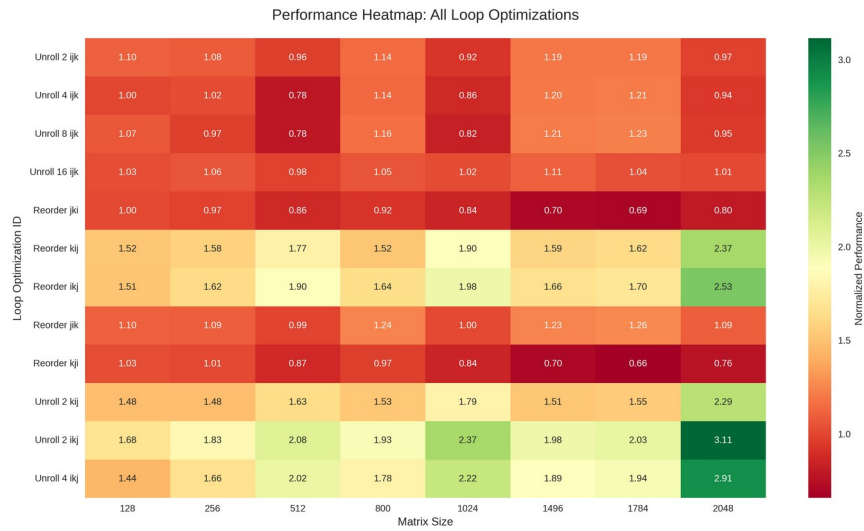
Task 1A: Unroll Baba Unroll

We explored 12 variations of matrix multiplication using different combinations of loop reordering and loop unrolling.

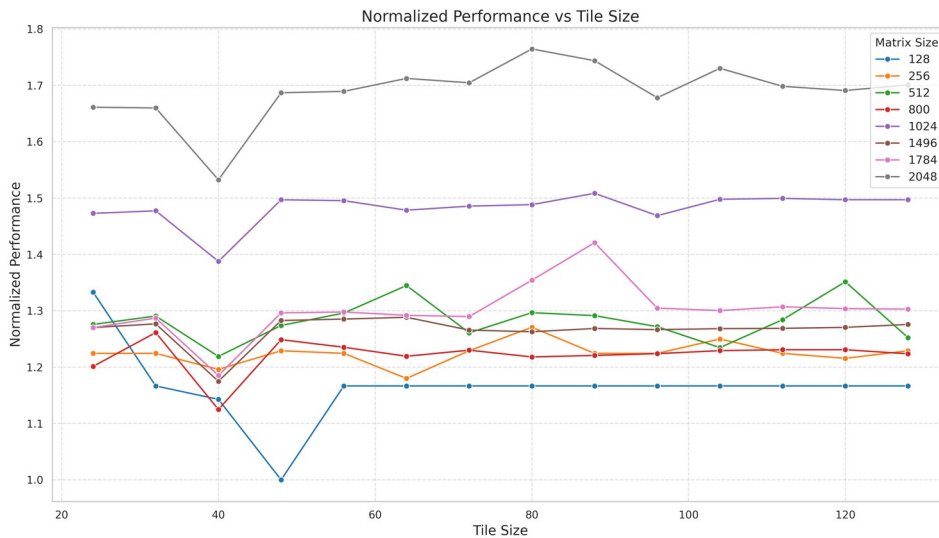
Overall Takeaway:

- **Pure unrolling (Loops 1–4):** Unrolling the naive i-j-k loop reduced loop overhead and gave some instruction-level parallelism (ILP). This offered small gains but didn't solve the main issue: poor cache locality. Beyond a point, high unroll factors hurt performance due to register pressure and instruction cache stress — an example of “too much unrolling.”
- **Loop reordering (Loops 5–9):** We tested all permutations of i, j, k. Only k-i-j and i-k-j improved performance because they improved data reuse by accessing A and B in a cache-friendly way. Others, especially those writing C column-wise, increased cache misses and in some cases performed worse than the naive version.
- **Reordering + unrolling (Loops 10–12):** Combining cache-friendly loop orders with light unrolling (especially i-k-j with unroll-by-2 or unroll-by-4) gave the best results. These versions reused loaded A values across multiple C updates, reduced memory traffic, and exploited ILP without overwhelming the CPU.

In short, the biggest gains came from fixing memory access patterns. Cache-aware loop orders like i-k-j or k-i-j are essential, and small, careful unrolling on top of them can squeeze out a bit more performance. But aggressive unrolling or reordering without cache considerations either helps very little or actively harms performance.



Task 1B: Divide Karo, Rule Karo



1) L1-D MPKI drops when moving from naive to tiled multiplication.

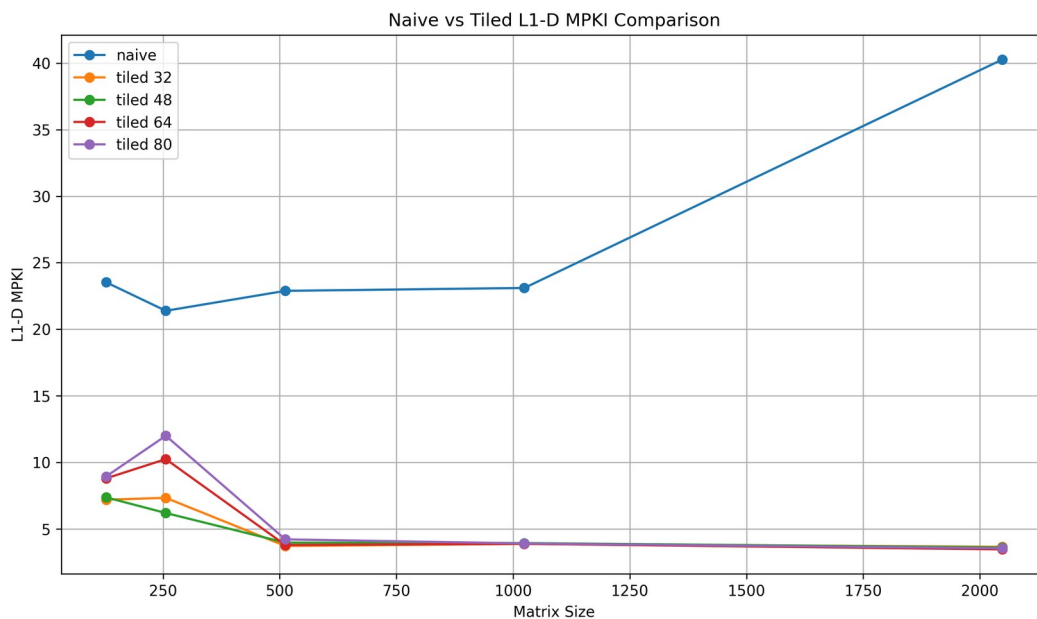
- Naive has poor access for B, causing frequent cache misses, especially as n grows.
- Tiling improves spatial and temporal locality by reusing tiles of A and B before eviction.
- The decrease is most noticeable for large matrices, where naive thrashes the cache.
- Tiling keeps A, B, and C blocks in cache, reducing loads/stores and lowering MPKI.

2) L1-D MPKI varies with matrix size and tile size.

- Small matrices (128, 256): fit in cache, naturally low MPKI, little gain from tiling.
- Medium matrices (512–1024): exceed L1, fit in L2; tiling reduces L1 misses and helps.
- Large matrices: exceed L1 and L2; theoretical MPKI depends on tile size, but in practice, little variation observed.
- Tile size theory predicts a U-shaped MPKI curve; practically, only slight variations at small sizes, no big effect at large sizes.
- Optimal tile size should fit A, B, and C tiles in L1 ($3 \cdot B^2 < 32\text{KB}$); large tiles break this advantage.

3) Speedup achieved.

- Execution time improves due to fewer cache misses, better spatial locality, and improved temporal reuse.
- Medium to large matrices (512–2048): $\sim 1.6\times$ – $2\times$ faster than naive.
- Small matrices (128, 256): $\sim 1.2\times$ – $1.4\times$ faster, limited by already good cache use.
- Gains come from reduced MPKI in L1/L2, fewer memory stalls, and higher arithmetic intensity (more FLOPs per cache miss).



Task 1C: Data Ko Line Mein Lagao

=> Change in Number of Instructions (Naive → SIMD)

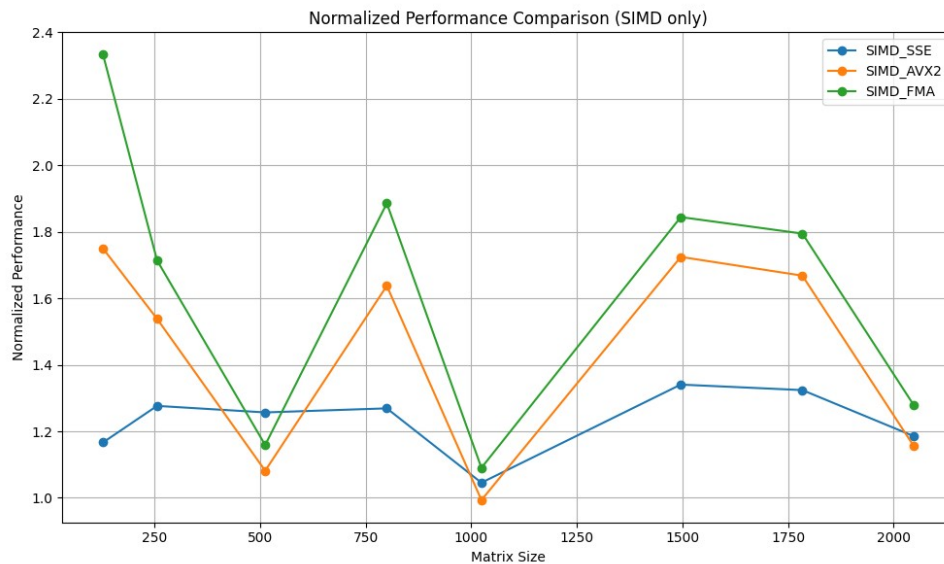
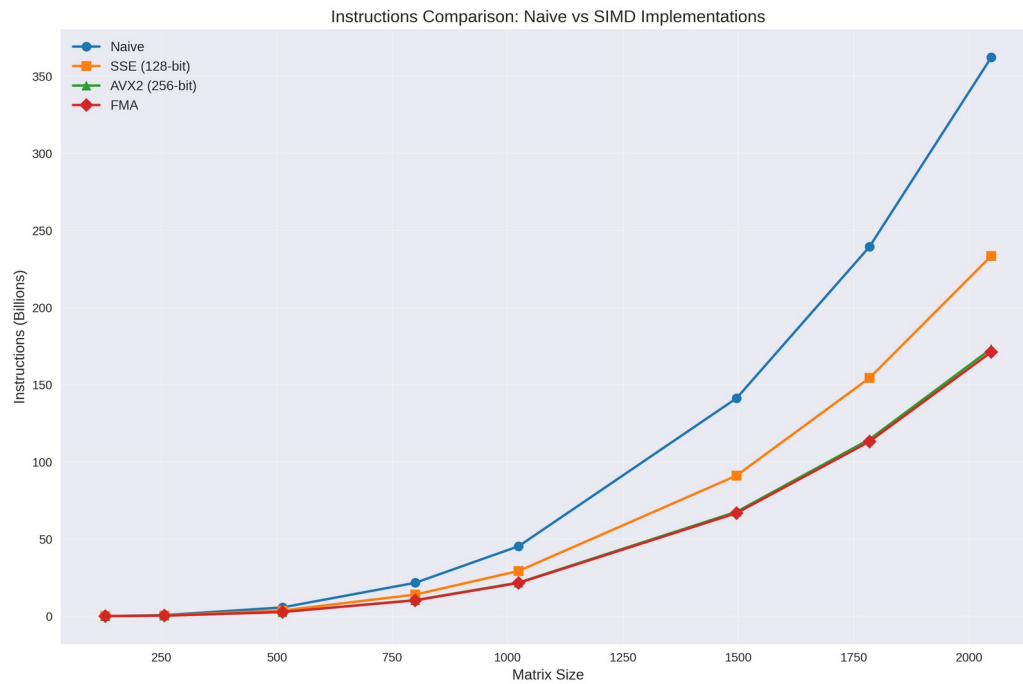
- Naive handles one element at a time. SIMD processes 2–4 elements per instruction, reducing total instructions.
- Vector multiply-add replaces scalar mul + add for chunks of data.
- With AVX (256-bit), ~4 doubles per instruction → ~4× fewer instructions (not exact due to alignment, load/store, and unrolling overhead).

Speedup Achieved

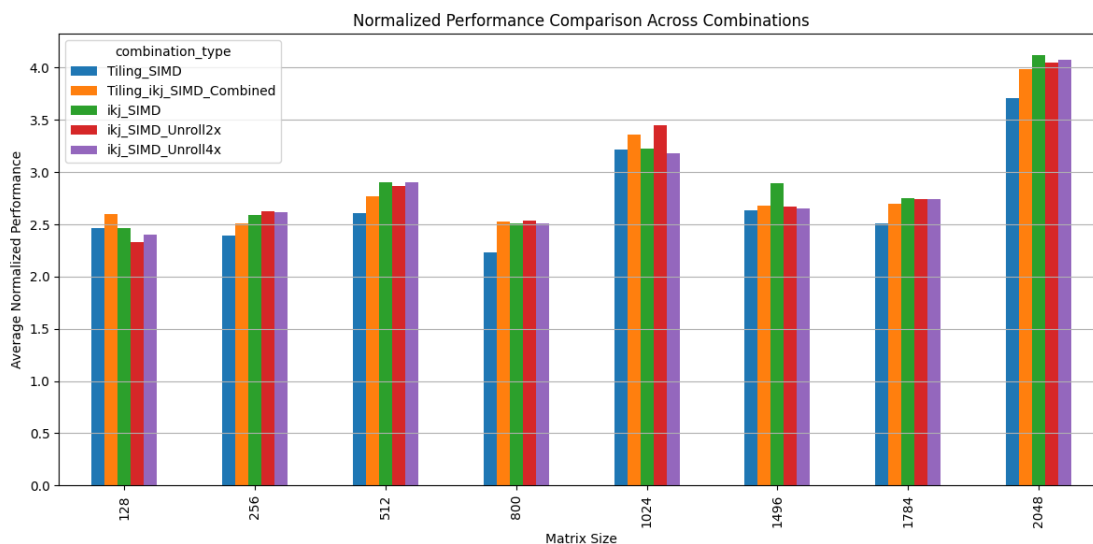
- SIMD gave clear speedups:
 - 128-bit: ~1.2× faster
 - 256-bit: ~1.8× faster
- Gains from parallel arithmetic, efficient loads/stores, and ILP via unrolling.
- Speedup drops after size 512 due to memory latency, cache misses, bandwidth saturation, and possibly less optimization on AMD hardware.

SIMD Intrinsics Used (AMD Machine)

- Types: `__m128d`, `__m256d`
- Load/Store: `_mm_loadu_pd`, `_mm_store_sd`, `_mm256_loadu_pd`, `_mm256_storeu_pd`
- Set/Init: `_mm_setzero_pd`, `_mm_set_pd`, `_mm256_setzero_pd`, `_mm256_set_pd`
- Multiply/Add: `_mm_mul_pd`, `_mm_add_pd`, `_mm256_mul_pd`, `_mm256_add_pd`
- Horizontal Add: `_mm_hadd_pd`
- FMA (AVX2): `_mm256_fmadd_pd`

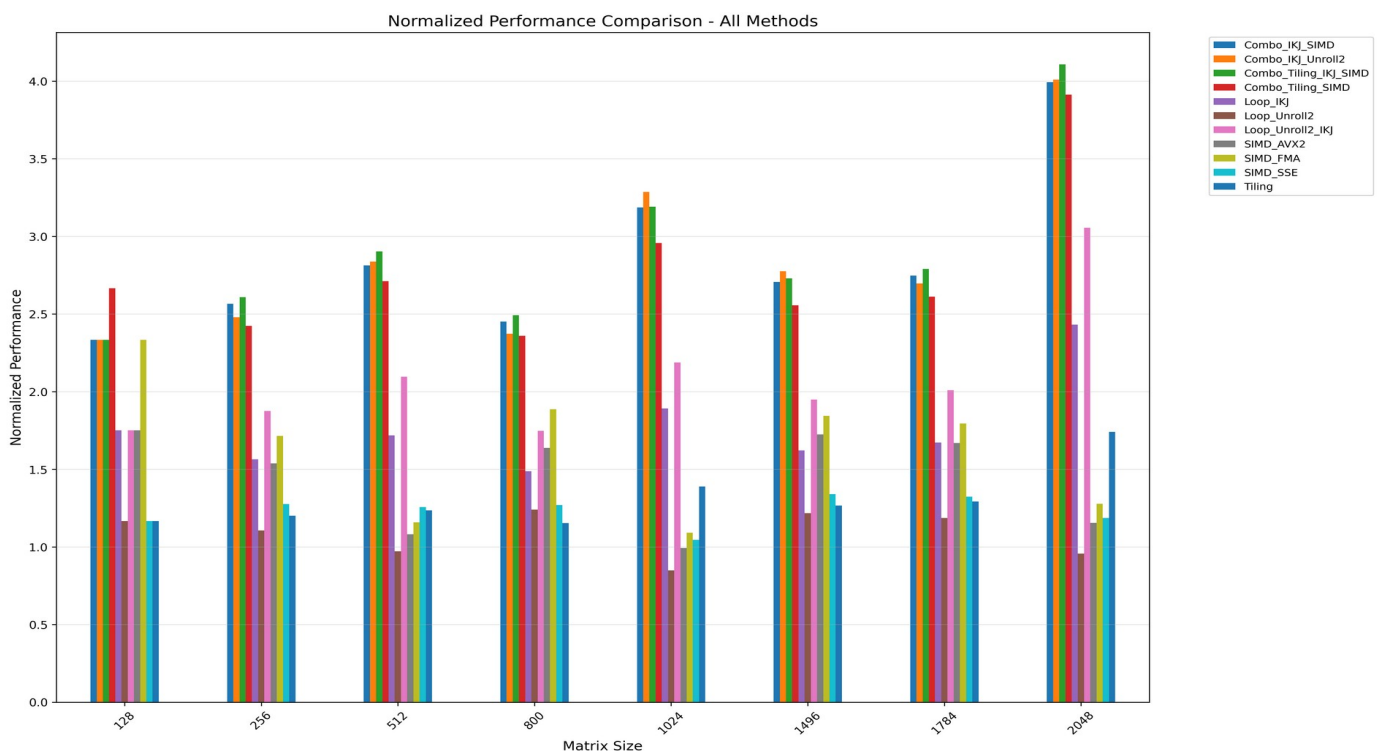


Task 1D: Rancho's Final Year Project



Combinations:

- Case 1 (ikj + SIMD) — Basic vectorization with ikj loop order, no tiling or unrolling; improves instruction-level parallelism.
- Case 2 (Tiling + SIMD) — Adds cache tiling to keep data longer in L1/L2; SIMD used inside tiles.
- Case 3 (ikj + SIMD + unrolling $\times 2$) — Same as Case 1 with inner loop unrolled $\times 2$ to boost instruction throughput.
- Case 4 (ikj + SIMD + unrolling $\times 4$) — Aggressive unrolling to reduce branch overhead and increase ILP.
- Case 5 (Tiling + ikj + SIMD) — Full combination: tiling, SIMD, and ikj; theoretically best for large matrices.



Summary:

- Small matrices (≤ 128) — All run mostly in cache; tiling/unrolling adds overhead; basic SIMD suffices.
- Medium matrices (256–512) — Unrolling $\times 2/\times 4$ improves performance via lower loop overhead and better ILP.
- Large matrices (≥ 800) — Tiling benefits offset by overhead; SIMD + unroll $\times 2$ (Case 3) is strong without high register pressure.
- Aggressive unrolling $\times 4$ can hurt at very large sizes due to register spilling and cache pressure.