Programming Assignment-1

Task 1

Matrix Multiplication — The Rancho Way

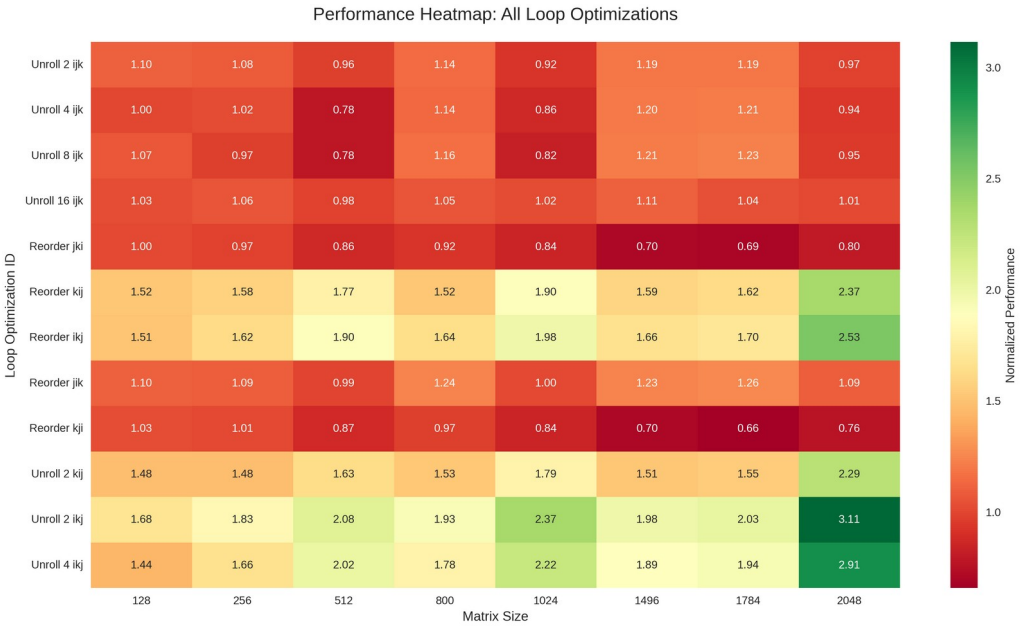
Task 1A: Unroll Baba Unroll

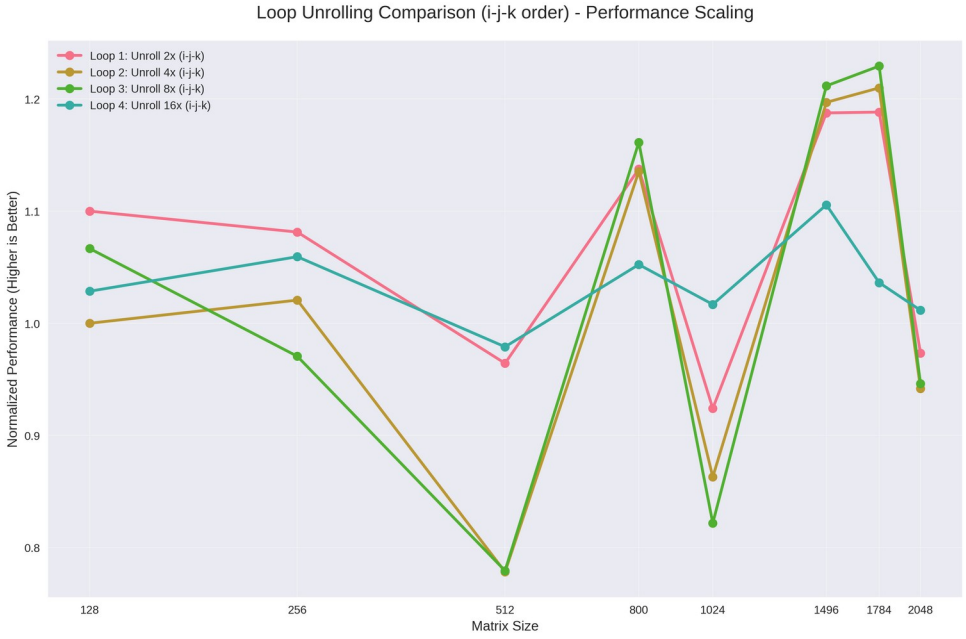
We explored 12 variations of matrix multiplication using different combinations of loop reordering and loop unrolling.

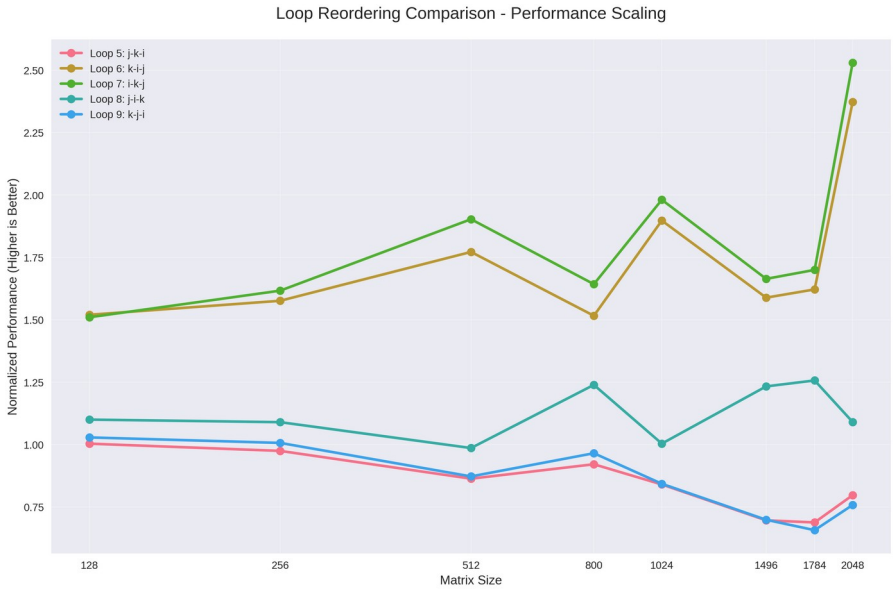
Overall Takeaway:

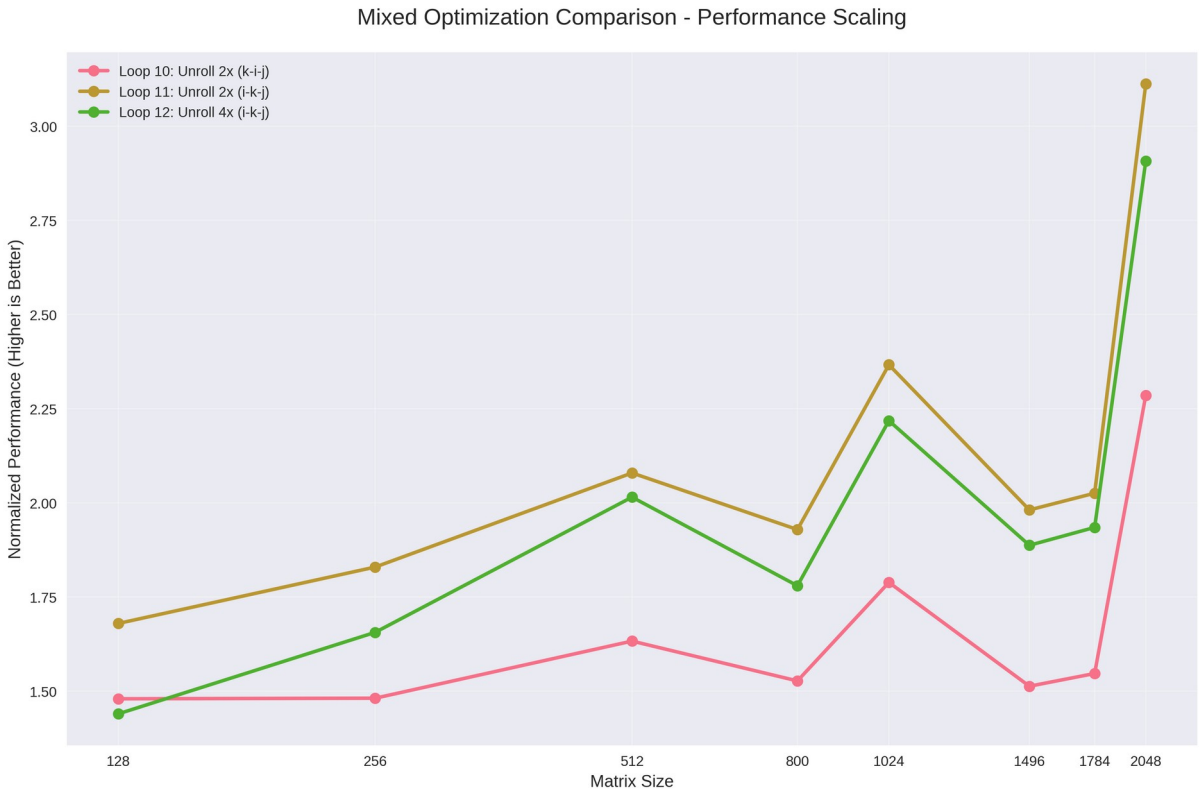
- **Pure unrolling (Loops 1–4):** Unrolling the naive i-j-k loop reduced loop overhead and gave some instruction-level parallelism (ILP). This offered small gains but didn't solve the main issue: poor cache locality. Beyond a point, high unroll factors hurt performance due to register pressure and instruction 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.


Performance Heatmap: All Loop Optimizations
Loop Optimization ID
Unroll 2 ijk 1.10 1.08 0.96 1.14 0.92 1.19 1.19 0.97
Unroll 4 ijk 1.00 1.02 0.78 1.14 0.86 1.20 1.21 0.94
Unroll 8 ijk 1.07 0.97 0.78 1.16 0.82 1.21 1.23 0.95
Unroll 16 ijk 1.03 1.06 0.98 1.05 1.02 1.11 1.04 1.01
Reorder jki 1.00 0.97 0.86 0.92 0.84 0.70 0.69 0.80
Reorder kij 1.52 1.58 1.77 1.52 1.90 1.59 1.62 2.37
Reorder ikj 1.51 1.62 1.90 1.64 1.98 1.66 1.70 2.53
Reorder jik 1.10 1.09 0.99 1.24 1.00 1.23 1.26 1.09
Reorder kji 1.03 1.01 0.87 0.97 0.84 0.70 0.66 0.76
Unroll 2 kij 1.48 1.48 1.63 1.53 1.79 1.51 1.55 2.29
Unroll 2 ikj 1.68 1.83 2.08 1.93 2.37 1.98 2.03 3.11
Unroll 4 ikj 1.44 1.66 2.02 1.78 2.22 1.89 1.94 2.91
128 256 512 800 1024 1496 1784 2048
Matrix Size
Normalized Performance
3.0 2.5 2.0 1.5 1.0


Loop Unrolling Comparison (i-j-k order) - Performance Scaling
Loop 1: Unroll 2x (i-j-k)
Loop 2: Unroll 4x (i-j-k)
Loop 3: Unroll 8x (i-j-k)
Loop 4: Unroll 16x (i-j-k)
Normalized Performance (Higher is Better)
1.2 1.1 1.0 0.9 0.8
128 256 512 800 1024 1496 1784 2048
Matrix Size


Loop Reordering Comparison - Performance Scaling
Loop 5: j-k-i
Loop 6: k-i-j
Loop 7: i-k-j
Loop 8: j-i-k
Loop 9: k-j-i
Normalized Performance (Higher is Better)
2.50 2.25 2.00 1.75 1.50 1.25 1.00 0.75
128 256 512 800 1024 1496 1784 2048
Matrix Size

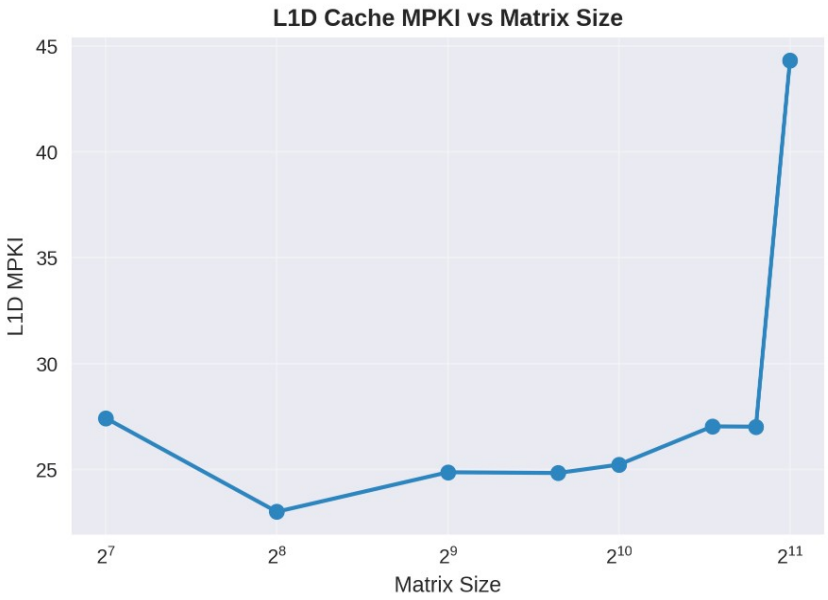


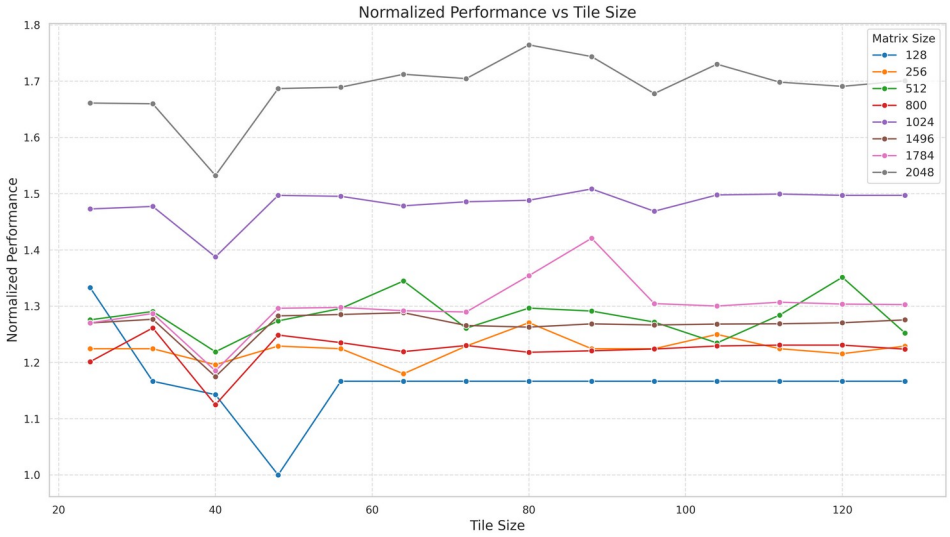
Task 1B: Divide Karo, Rule Karo

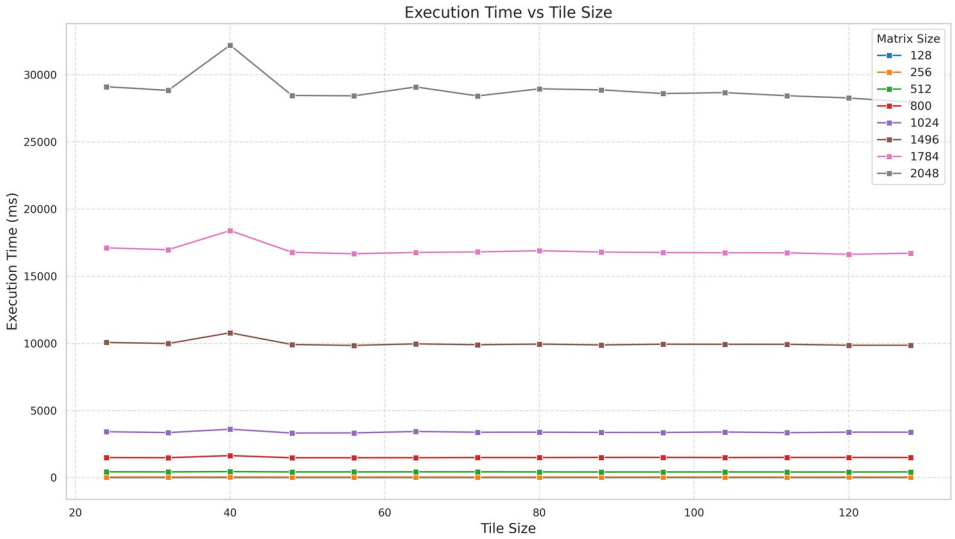
=> Ran “lscpu” command to get the L1d cache size which is 192 KiB across 6 instancs so single cache size is 32KiB

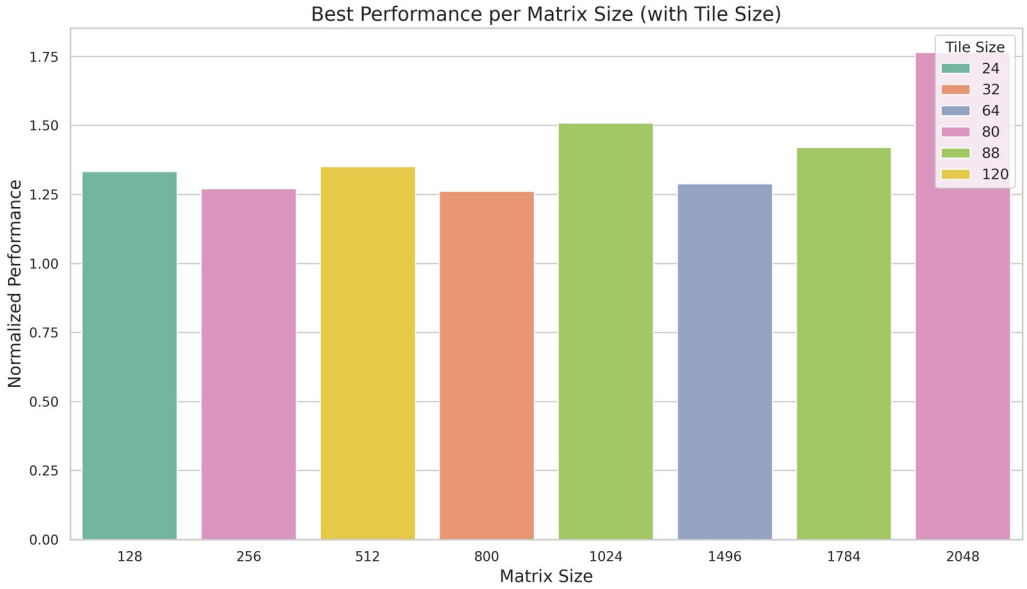
```
Caches (sum of all):
  L1d:                    192 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     3 MiB (6 instances)
  L3:                     16 MiB (1 instance)
NUMA:
```

=>




Normalized Performance vs Tile Size
Normalized Performance
Tile Size
Matrix Size
128
256
512
800
1024
1496
1784
2048


Execution Time vs Tile Size
Execution Time (ms)
Tile Size
Matrix Size
128
256
512
800
1024
1496
1784
2048


Best Performance per Matrix Size (with Tile Size)
Normalized Performance
Matrix Size
Tile Size
24
32
64
80
88
120

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 ($3*B^2 < 32KB$); large tiles break this advantage.

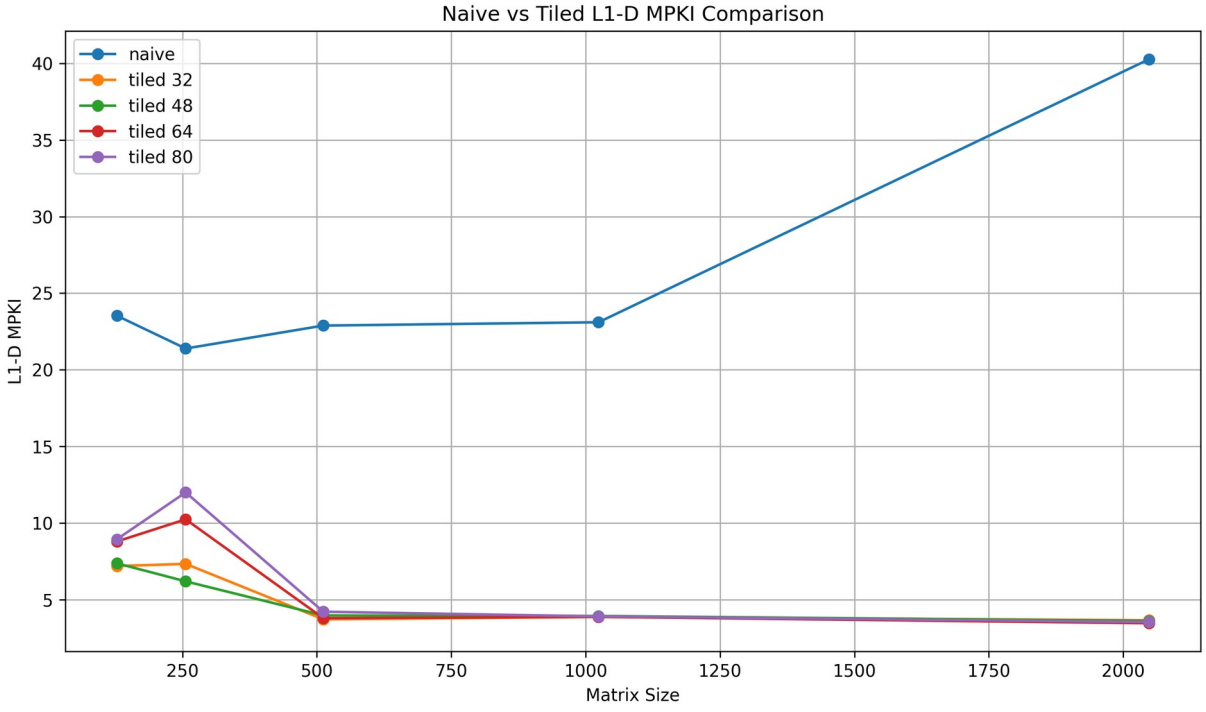
3) Speedup achieved.

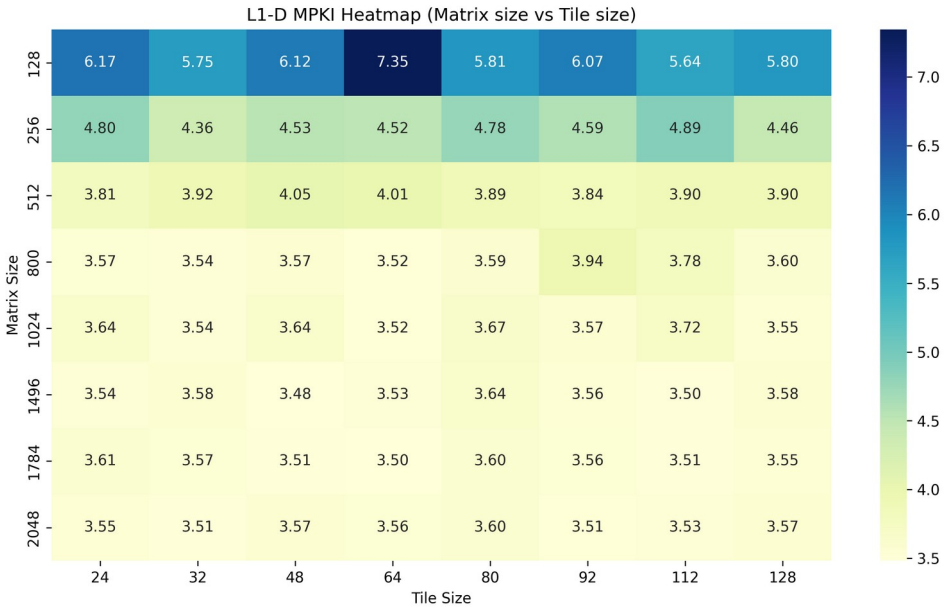
• Execution time improves due to fewer cache misses, better spatial locality, and improved temporal reuse.

• Medium to large matrices (512–2048): ~1.6×–2× faster than naive.

• Small matrices (128, 256): ~1.2×–1.4× 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

=> The number of instructions executed for the naive matrix multiplications are as follows

Size --- Instructions

128 --- 98055023

256 --- 725762252

512 --- 5711754785

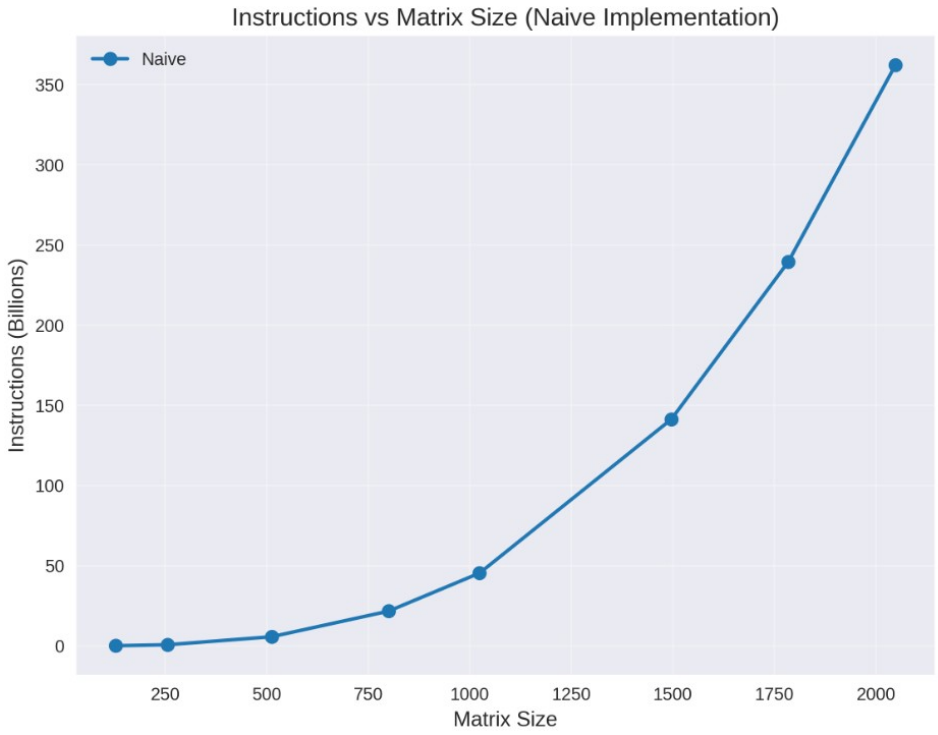
800 --- 21681855967

1024 --- 45413128138

1496 --- 141293823121

1784 --- 239458853080

2048 --- 362271101612



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

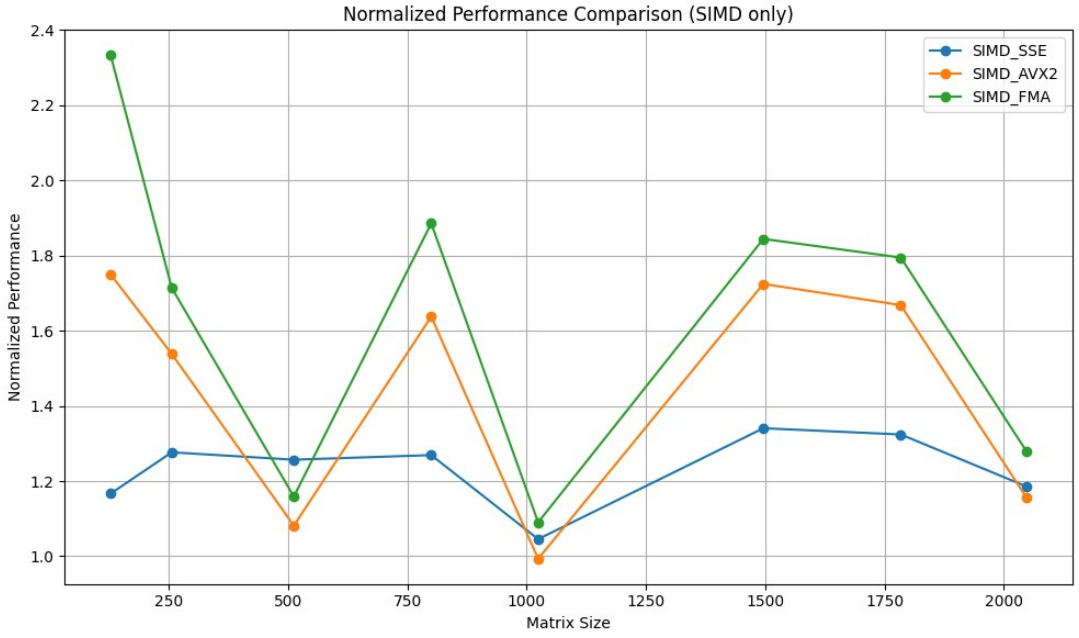
• Load/Store: _mm_loadu_pd, _mm_store_sd, _mm256_loadu_pd, _mm256_storeu_pd

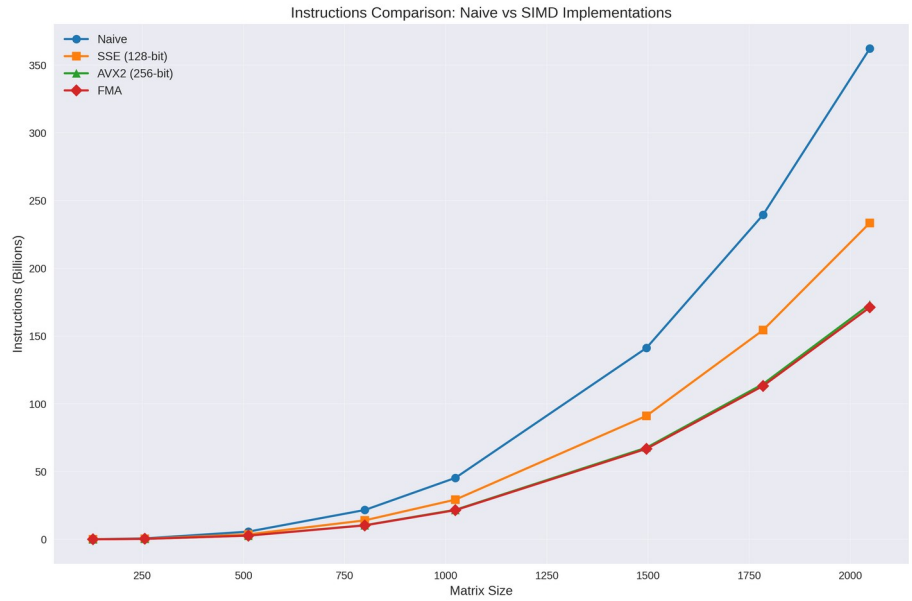
• Set/Init: _mm_setzero_pd, _mm_set_pd, _mm256_setzero_pd, _mm256_set_pd

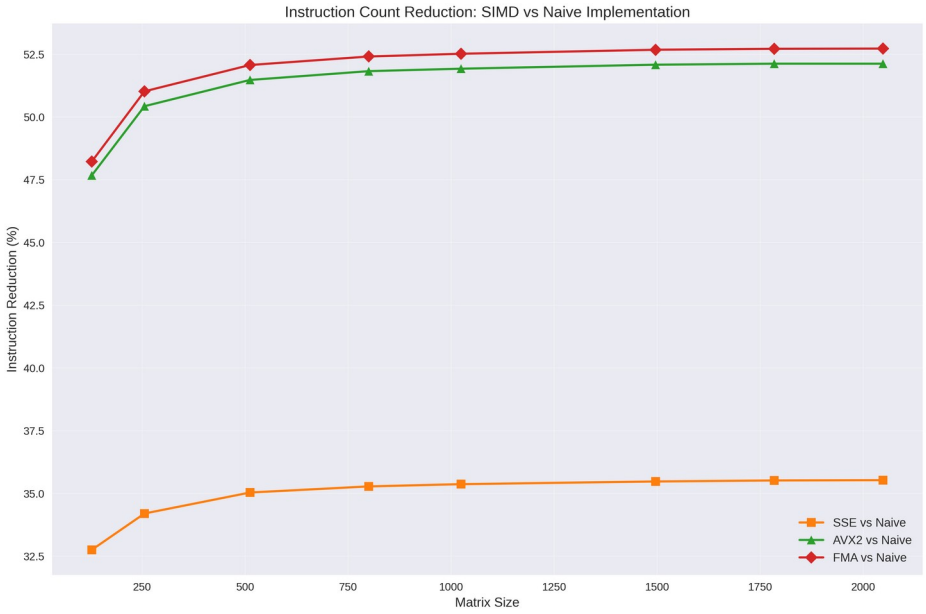
• Multiply/Add: _mm_mul_pd, _mm_add_pd, _mm256_mul_pd, _mm256_add_pd

• Horizontal Add: _mm_hadd_pd

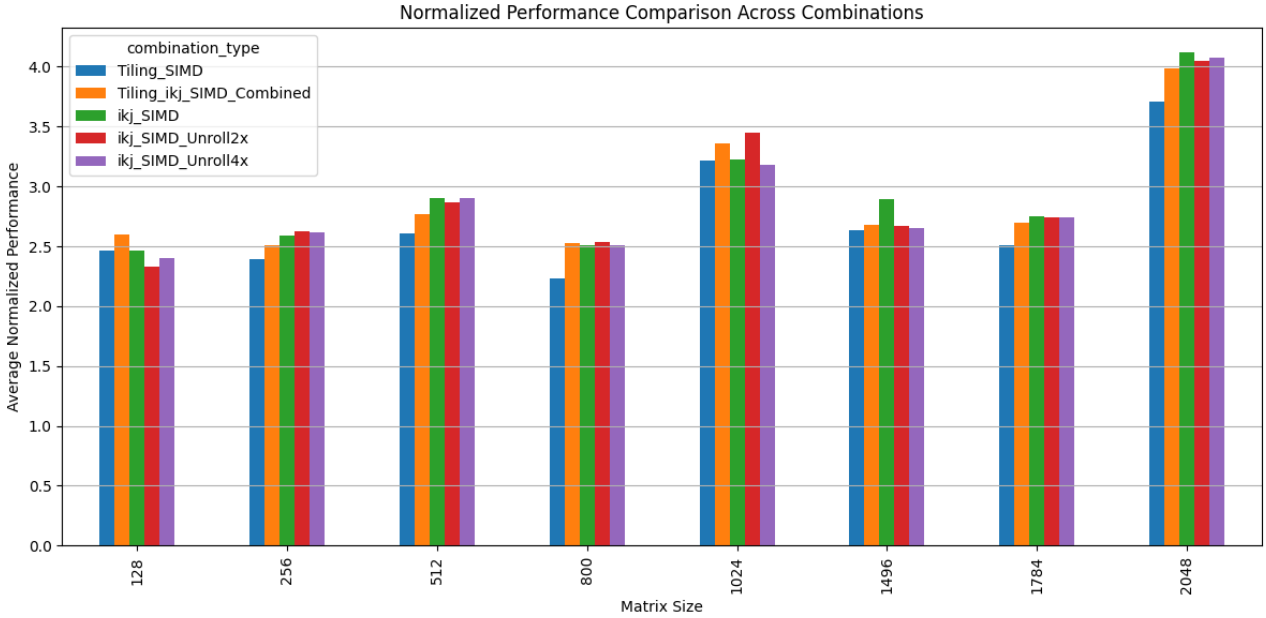
• FMA (AVX2): _mm256_fmadd_pd





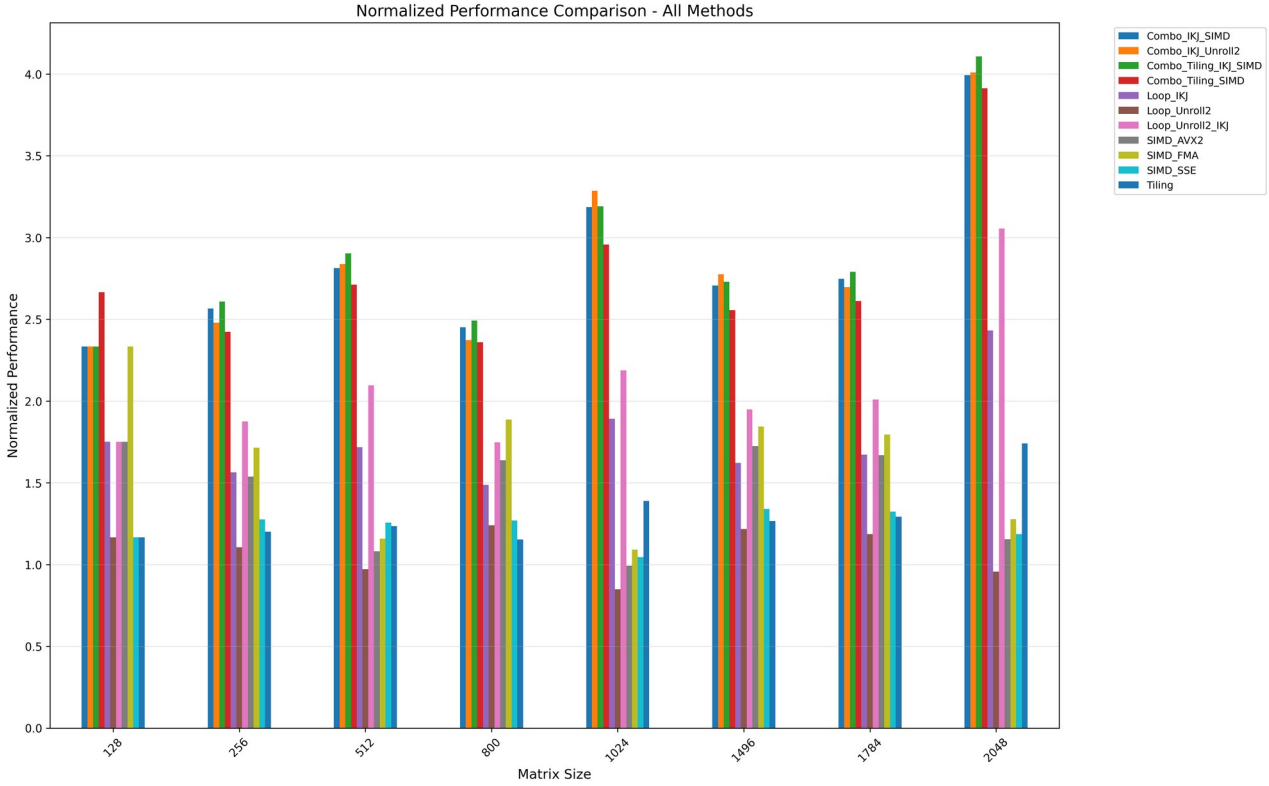


Task 1D: Rancho's Final Year Project



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 ×2) — Same as Case 1 with inner loop unrolled ×2 to boost instruction throughput.
• Case 4 (ikj + SIMD + unrolling ×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 ×2/×4 improves performance via lower loop overhead and better ILP.
• Large matrices (≥800) — Tiling benefits offset by overhead; SIMD + unroll ×2 (Case 3) is strong without high register pressure.