Programming Assignment-1

Task 1

Matrix Multiplication — The Rancho Way

Task 1A: Unroll Baba Unroll

=> We have tried multiple variations of matrix multiplications using loop unrolling and loop reordering, below are the summary of all the implementations that we have tried

Summary of Implementations (Loop 1 -> 12)

1. Loop Unrolling @2, i-j-k

- Uses the **classic i-j-k ordering** but unrolls the innermost `k` loop by a factor of 2.
- This cuts the loop control overhead in half and exposes a bit of **instruction-level parallelism (ILP)**.
- Still suffers from poor cache reuse because `B` is accessed column-wise.

2. Loop Unrolling @4, i-j-k

- Similar to Loop 1 but unrolled by factor 4.
- Further reduces branch instructions and increases ILP.
- Gains more speedup, but cache locality remains a bottleneck.

3. Loop Unrolling @8, i-j-k

- Similar to above two but unroll factor increased to 8.
- Performance improves up to the point where register pressure and instruction cache (I-cache) overhead kick in.
- Diminishing returns start here because not all CPUs can handle that many simultaneous in-flight instructions efficiently.

4. Loop Unrolling @16, i-j-k

- Unroll factor 16.
- Now the loop body is very large, now we are putting too much stress on the instruction cache (L1I), and now bandwidth will become the bottleneck.
- This is an example of “too much unrolling.”

5. Loop Reordering j-k-i

- Changes nesting to **j-k-i order**.
- This reordering makes writes to `C` less cache-friendly since memory is stored in row-major order, causing more cache misses.
- Normalised performance is less than 1;

6. Loop Reordering k-i-j

- Similar to above this is also not cache friendly, so speedup is not there and performace decreased;

7. Loop Reordering i-k-j

- This is the **most cache-friendly** loop order.
- As we see in thoery this has better spatial locality compared to i-j-k.
- As expected this loop order gave the best normalized performace on all matrix sizes

8. Loop Reordering j-i-k

- Access pattern to `C` is again column-major, which is suboptimal for row-major storage.
- Suffers from high L1 data cache (L1D) misses.
- Gave a slight speed up as compare to other non-friendly ones which gave below 1 performances..!

9. Loop Reordering k-j-i

- Updates column-by-column in `C` for each `k`.
- Similar memory access inefficiencies as j-i-k but performed worst than j-i-k, Reason can be higher cahce line thrashing.

10. Loop Unrolling @2, k-i-j

- Combines k-i-j order (cache friendly) with unrolling factor 2 along the `j` loop.
- Slight improvement over plain k-i-j, since two `C` elements are updated per iteration.

11. Loop Unrolling @2, i-k-j

- Starts with the most cache-friendly i-k-j order and unrolls inner `j` by 2.

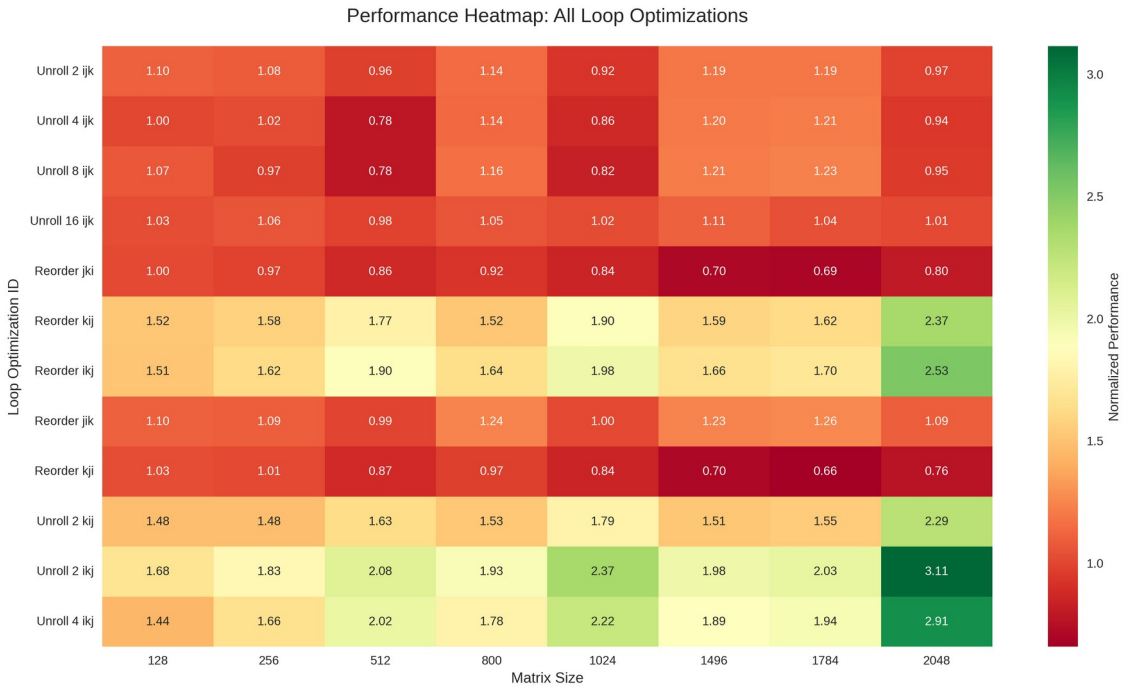
- Adds explicit variable reuse (`r = A[i*n + k]`) so that one `A` element is loaded once and reused.
- Cuts memory traffic and improves register-level optimization, noticeable performance boost with low overhead.

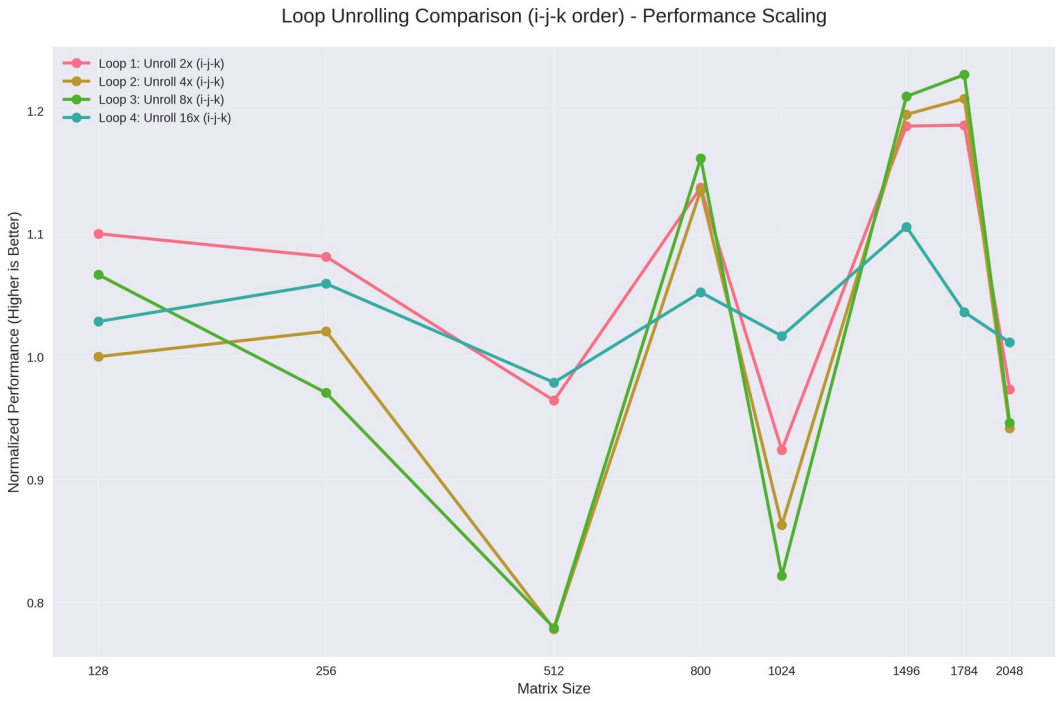
12. Loop Unrolling @4, with loop order i-k-j

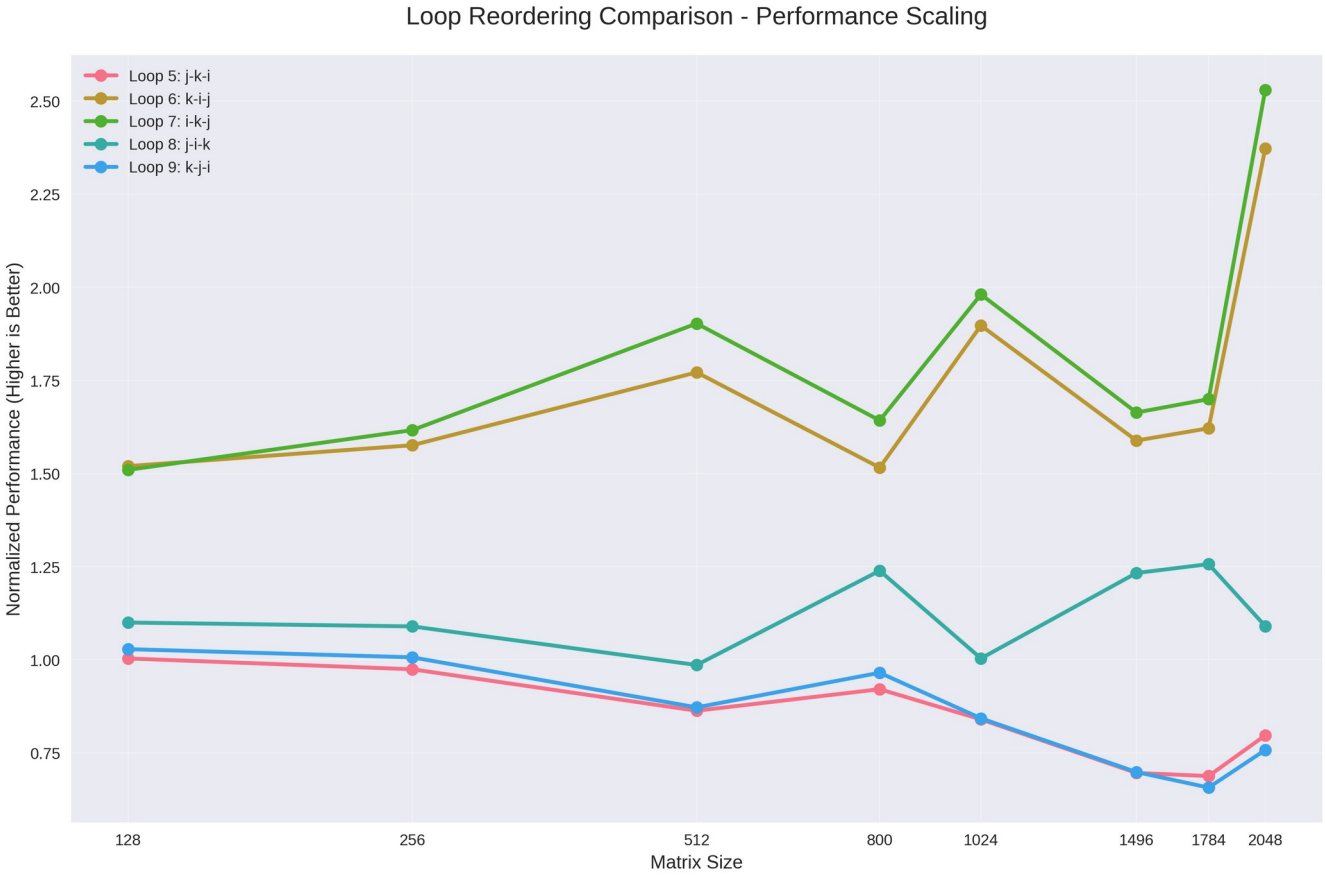
- Builds on Loop 11 with a higher unroll factor of 4.
- Processes 4 elements of `C` per iteration while keeping `A[i,k]` in a register.
- Tried to combines cache efficiency, unrolling, reduced memory traffic and ILP via loop unrolling.
- This performed almost similar to the Loop 11

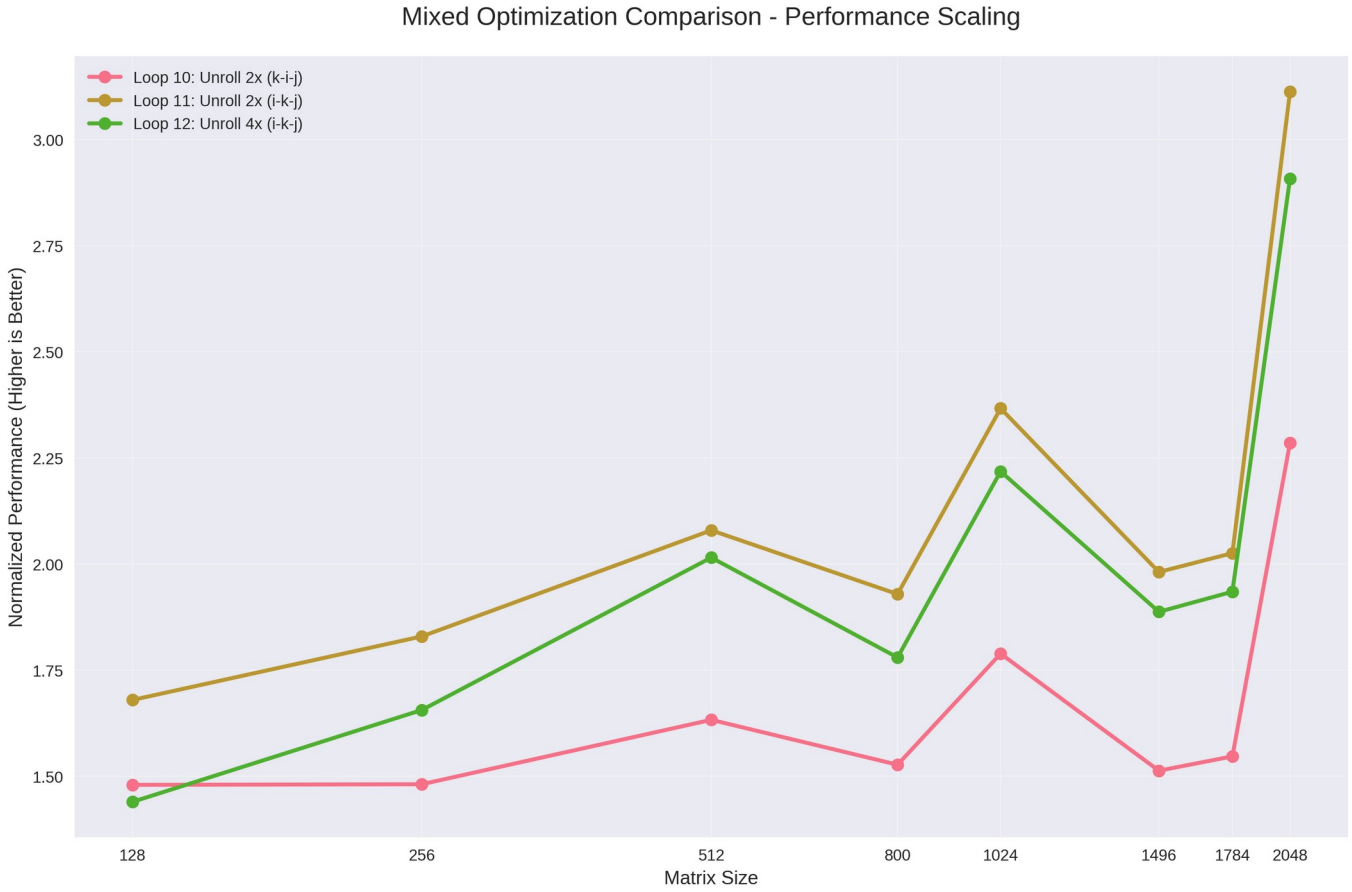
Overall Takeway---

- **Loops 1–4**: Pure unrolling of the naive i-j-k → brings instructions level parallelism(ILP), which which reduced L1i cache misses, but doesn't fixed cache inefficiency.
- **Loops 5–9**: We try all the possible permutations of i-j-k (except the naive i-j-k), k-i-j and i-k-j had the cache friendly access patterns so they performed good, and rest combinations did worst..!
- **Loops 10–12**: Tried the combination of reordering + unrolling → best performance, especially Loop 11 and 12.




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

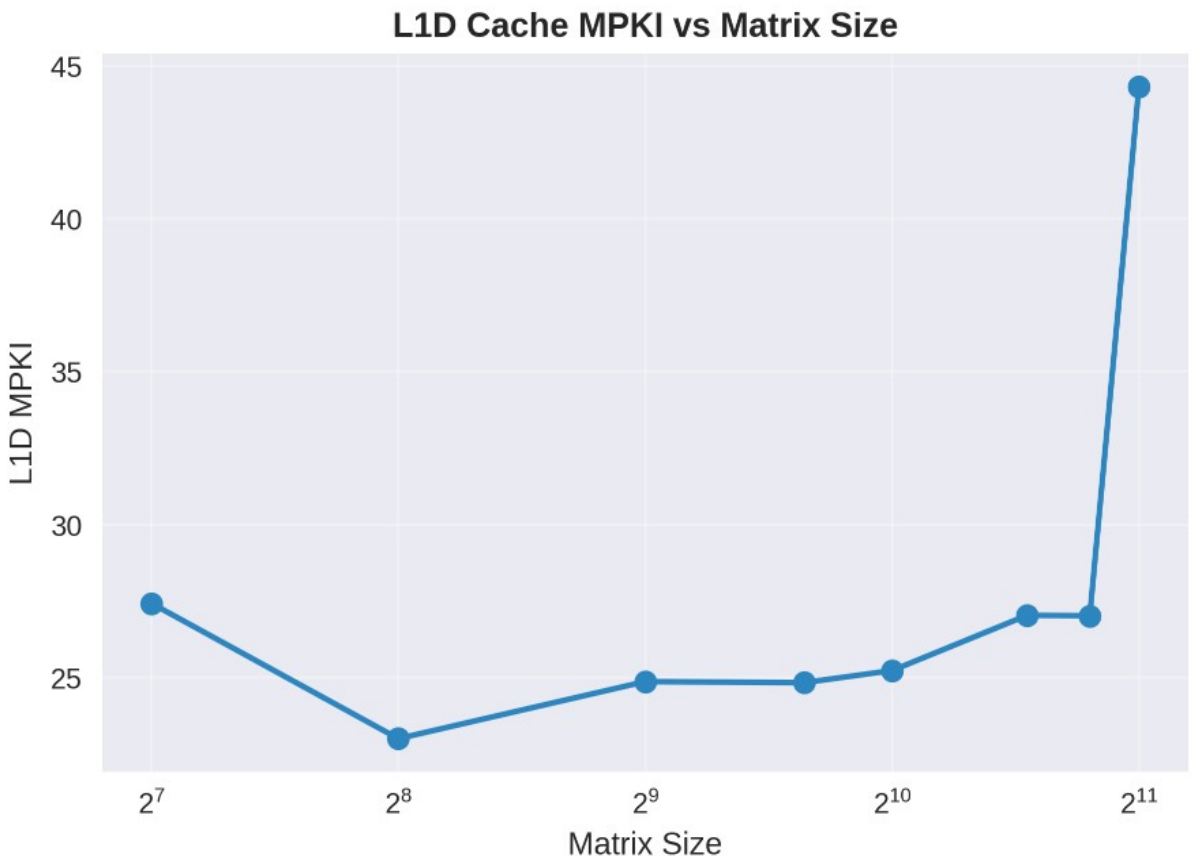

Mixed Optimization Comparison - Performance Scaling
Loop 10: Unroll 2x (k-i-j)
Loop 11: Unroll 2x (i-k-j)
Loop 12: Unroll 4x (i-k-j)
Normalized Performance (Higher is Better)
3.00
2.75
2.50
2.25
2.00
1.75
1.50
128
256
512
800
1024
1496
1784
2048
Matrix Size

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

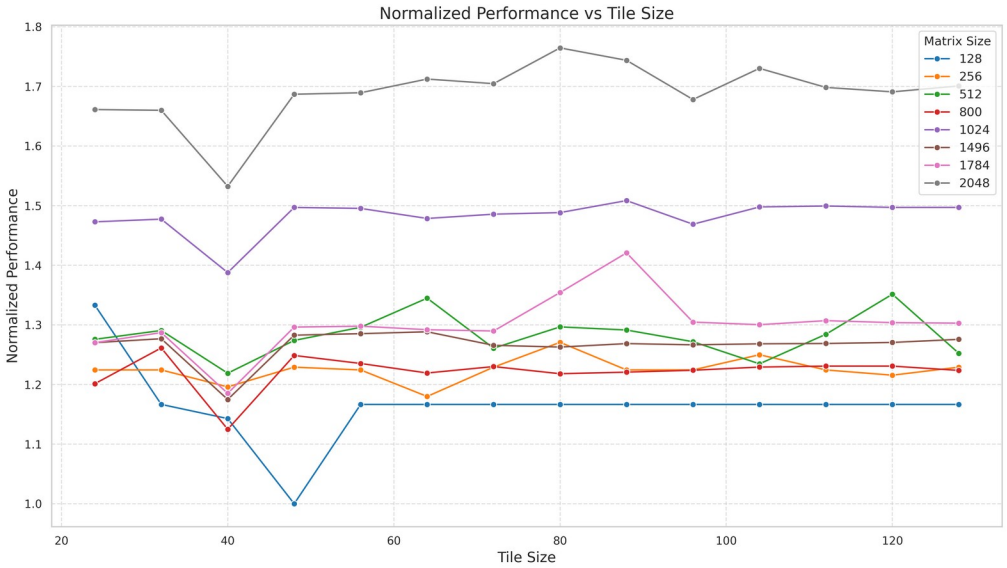
```
Caches (sum of all):
  L1d:                    192 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     3 MiB (6 instances)
  L3:                     16 MiB (1 instance)
NUMA:
```

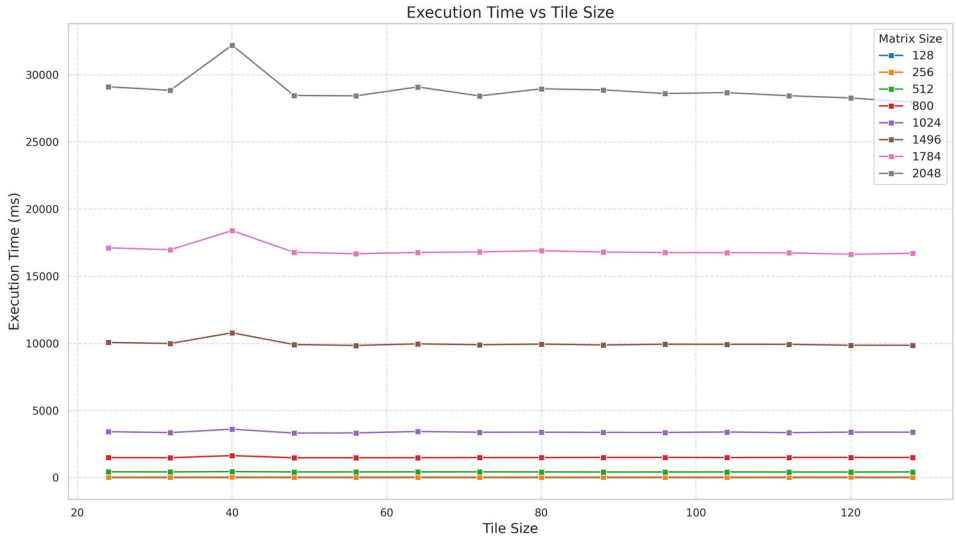
=> Ran perf tool to extract the L1d_cahce_misses and total instructions for each of the matrix sizes and calcualte MPKI..!

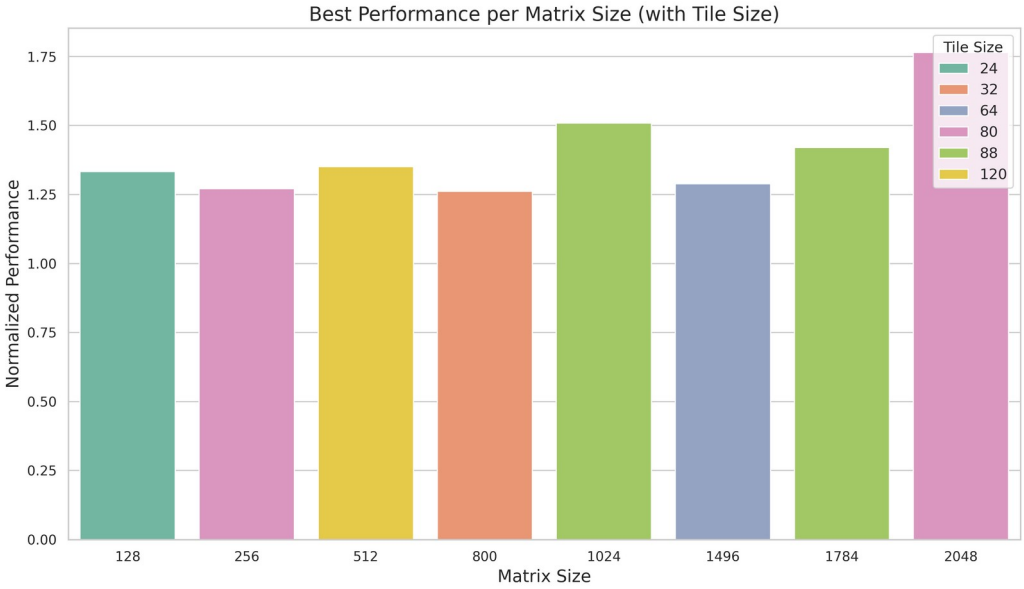


MATRIX SIZES = [128, 256, 512, 800, 1024, 1496, 1784, 2048]

=>


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. Report the changes in L1-D MPKI observed when moving from naive to tiled matrix multiplication. Justify your observations.

- In **naive matrix multiplication** (`i-j-k` triple loop without tiling):
    - Access pattern is poor for **matrix B** because its elements are accessed in a **strided manner** (`B[k*n + j]`), which leads to frequent cache misses.
    - As `n` grows, the working set easily exceeds L1 cache capacity, so L1-D MPKI is high.
- In **tiled matrix multiplication**:
    - By introducing tiles (sub-blocks of matrices), data reuse within the cache improves.
    - Once a tile of `A` and a tile of `B` are loaded, multiple multiplications occur before those cache lines are evicted.
    - This improves **spatial locality** and **temporal locality**, reducing cache misses significantly.

Observation:

- **L1-D MPKI decreases substantially when moving from naive to tiled implementation.**
- The reduction is most noticeable at **larger matrix sizes**, where naive multiplication suffers the most from cache thrashing.

Justification:

- Tiling ensures that portions of `A`, `B`, and `C` fit in cache and get reused before being replaced.
- This reuse reduces the number of loads/stores from memory, which directly lowers MPKI.

---

2. How did L1-D MPKI vary across different matrix sizes and tile sizes? Explain in terms of cache hierarchy and working set sizes.

- **Variation with Matrix Size:**
    - For **small matrices (128, 256)**:
        - Even the naive approach fits reasonably well in L1/L2 cache.
        - MPKI is naturally low, and tiling doesn’t offer much additional benefit.

    - For **medium and matrices (512 – 1024)**:
        - In medium sized matrices, working set exceeds L1 capacity but fits in L2.
        - Tiling shows noticeable improvement, as it reduces L1 misses by keeping reused blocks in cache.
        - Working set exceeds both L1 and L2 caches in the case of larger matrices.
        - According to theory MPKI must be dependent on the tile size we choose but in practical it showed very little variations
- **Variation with Tile Size:**
    - According to theory we must get a U-shaped curve for Tile size vs L1-D MPKI,
    - But in our practical implementation there was only slight variation seen in smaller matrices e.g. 128 and 256 but there wasn't any significant variation for larger matrices..

Cache Hierarchy Theory:

- L1 cache is small (~32KB on our CPUs), so optimal tile size is chosen such that `A`, `B`, and `C` tiles fit simultaneously in L1. (3*B^2 < 32KB).
- Larger matrices benefit more from tiling because they otherwise blow past L1/L2 caches.
- If tiles are too large, they no longer fit in L1, defeating the purpose.

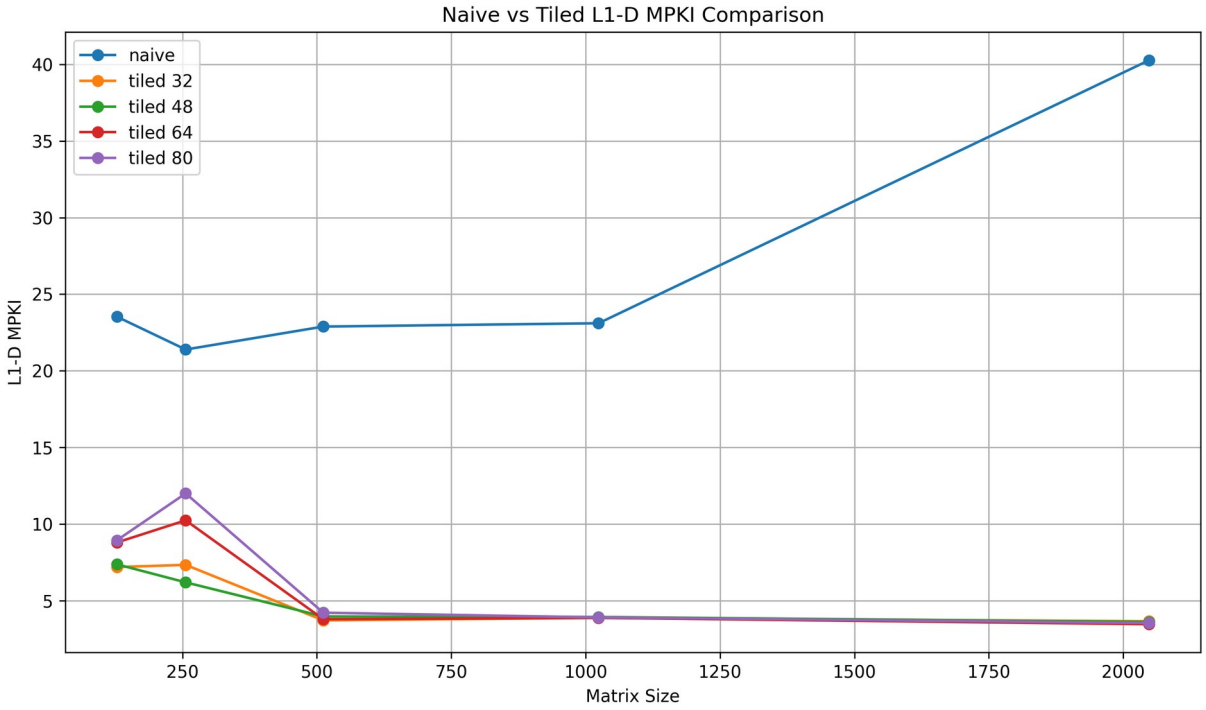
---

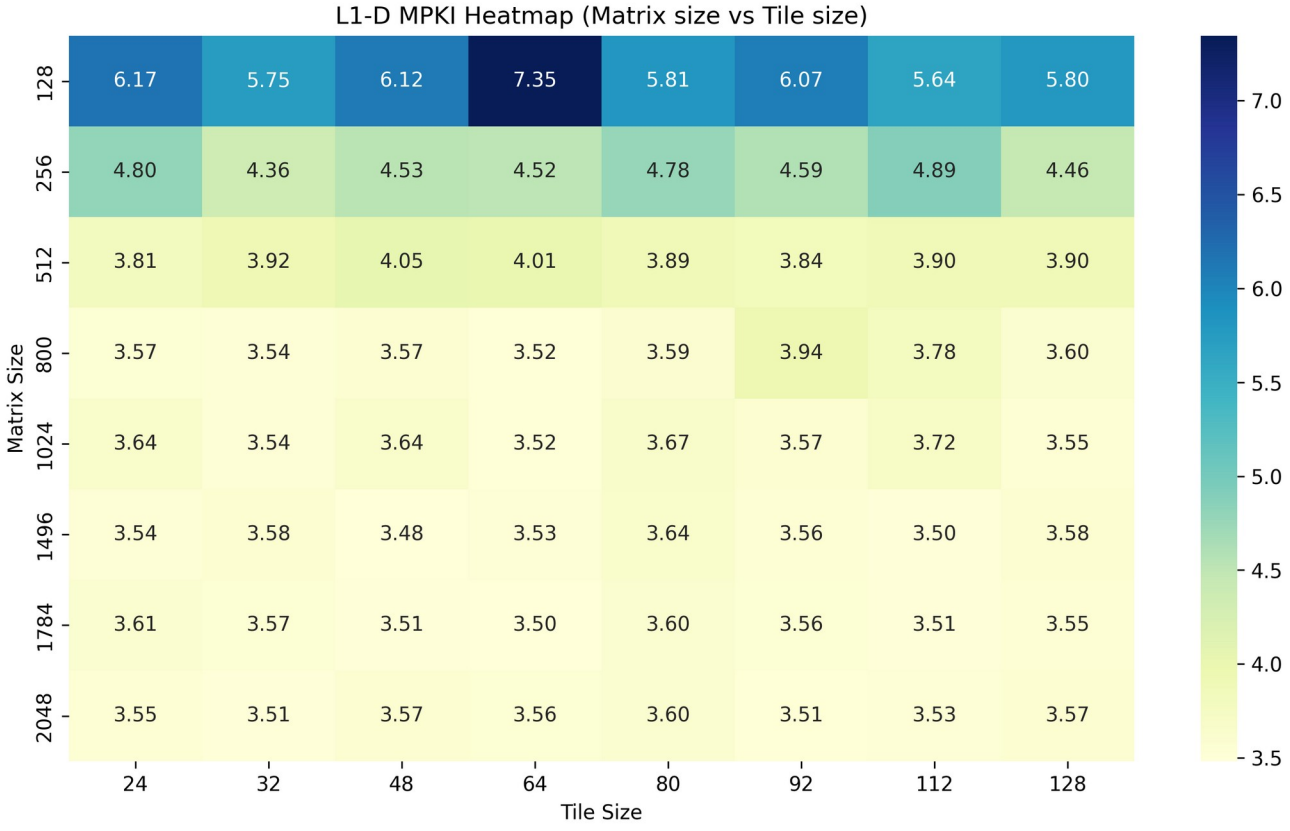
3. Did you achieve a speedup? If yes, quantify the improvement and identify contributing factors. If not, analyze limiting factors and propose solutions.

- **Yes, speedup is typically observed.**
- Execution time improves because:
    1. **Reduced cache misses (lower MPKI):** Less time spent waiting on memory.
    2. **Better spatial locality:** Consecutive accesses map well to cache lines.
    3. **Improved temporal reuse:** Each loaded tile is reused multiple times before eviction.
- **Quantification:**
    1. For **medium to large matrices (512 – 2048)**, speedups of **1.6x - 2x** are common compared to naive implementation.
    2. For **small matrices (128, 256)**, speedup is minimal around **1.2x-1.4x** since cache already handles them well.
- **Contributing factors to speedup:**
    1. Reduced MPKI in L1 and L2.

2. Fewer memory stalls.
3. Increased 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

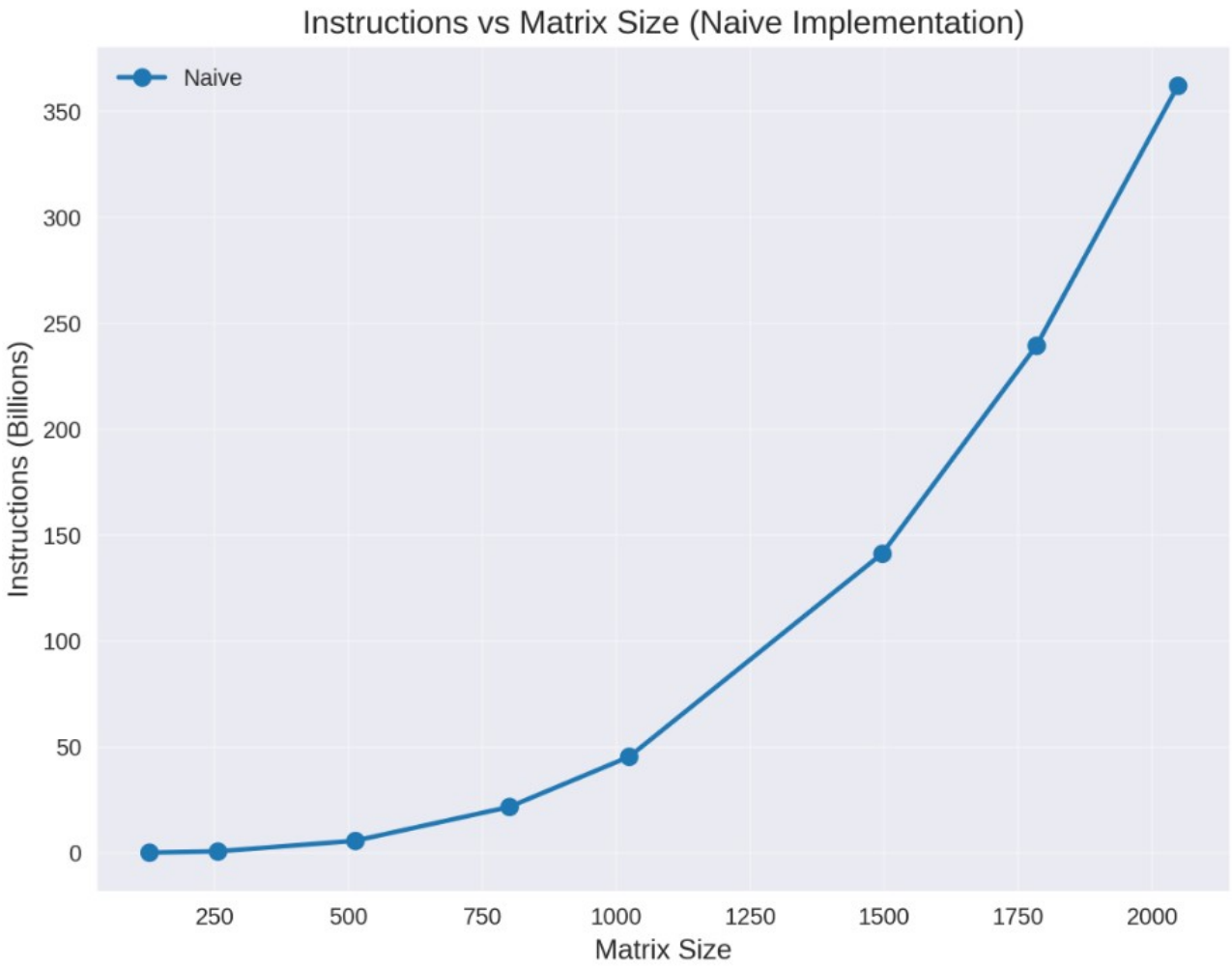
800 --- 21681855967

1024 --- 45413128138

1496 --- 141293823121

1784 --- 239458853080

2048 --- 362271101612



1. Change in Number of Instructions (Naive → SIMD)

- **Observation:**
  In the naive implementation, each scalar multiplication and addition was handled one element at a time. With SIMD (128-bit / 256-bit), multiple elements (2 or 4) were processed per instruction, significantly reducing the *total number of executed instructions*.
- **Justification:**
  - SIMD vectorizes the inner loop, meaning instead of one `mul` + `add` per element, a single vector multiply-add instruction operates on a chunk of elements.
  - For example, with AVX (256-bit), 4 doubles are processed per instruction, so the number of executed instructions is reduced roughly by a factor of 4 compared to scalar operations.
  - However, overhead from **memory alignment, loads/stores, and loop unrolling** means the reduction is not exactly proportional to the SIMD width.

---

2. Speedup Achieved

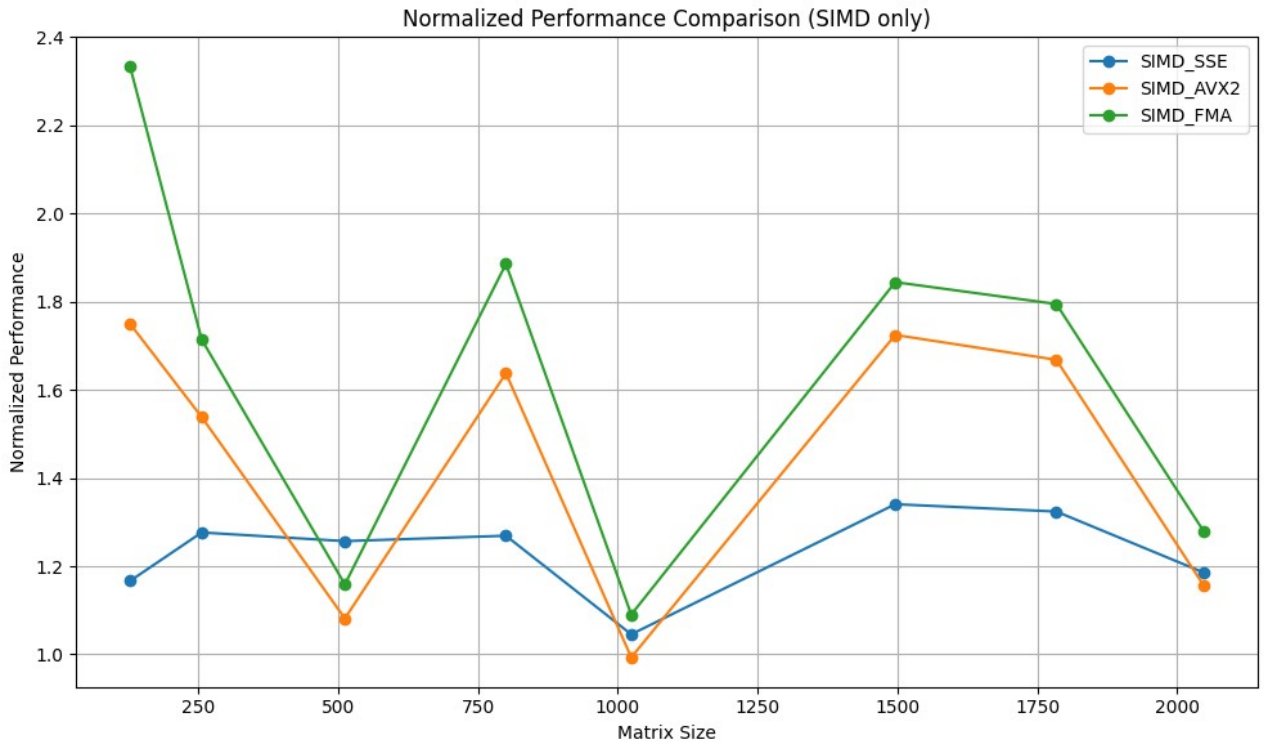
- **Observation:**
  - Yes, speedup was observed.
  - Speedup increased with SIMD register width:
    - 128-bit: ~1.2× faster than naive
    - 256-bit: ~1.8× faster
- **Reason for Speedup:**
  - SIMD parallelism reduces the number of executed arithmetic instructions.
  - Memory bandwidth and cache usage were more efficient due to aligned loads/stores.
  - Instruction-level parallelism (ILP) was exploited by unrolling the innermost loop.
- **Speedup decreased after size 512:**
  - Bottlenecks like **memory access latency, cache misses, or unaligned loads**.
  - Performance may flatten out at large sizes due to **memory bandwidth saturation**.
  - It can be the case that intel intrinsic are not much optimized for the AMD machine..!

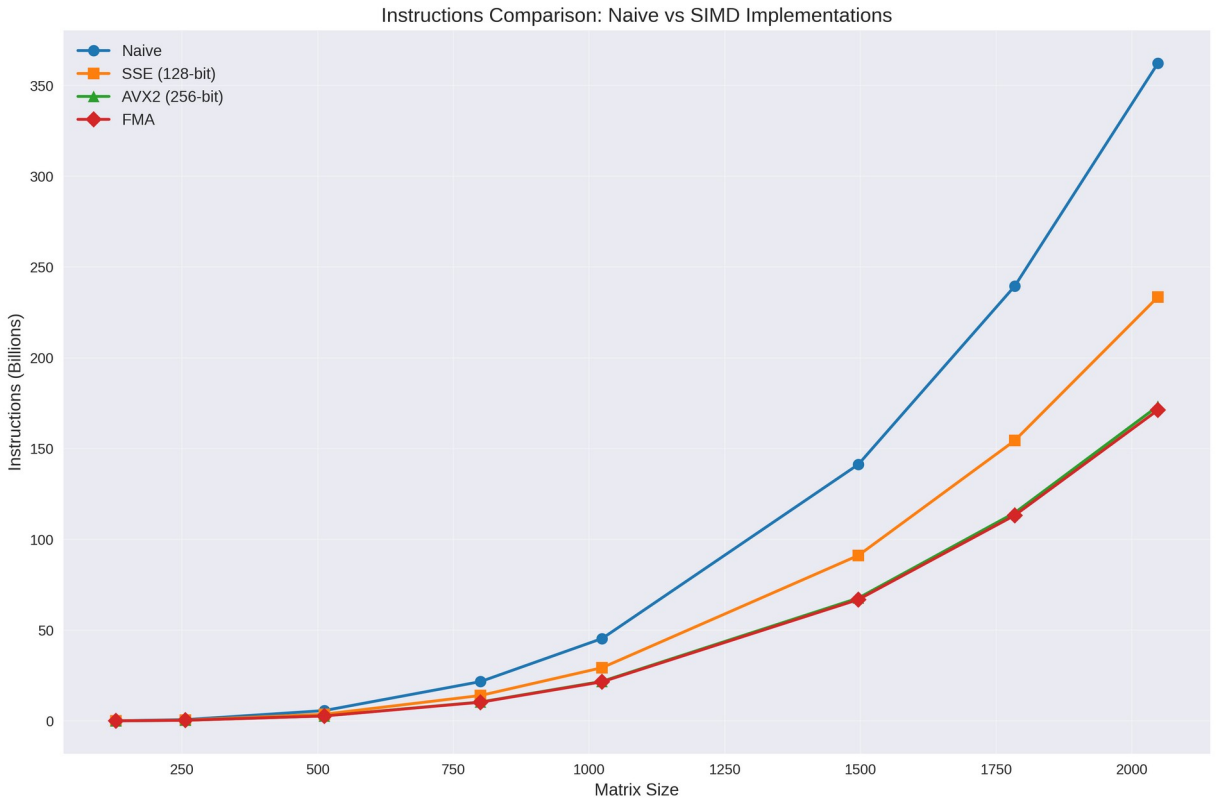
---

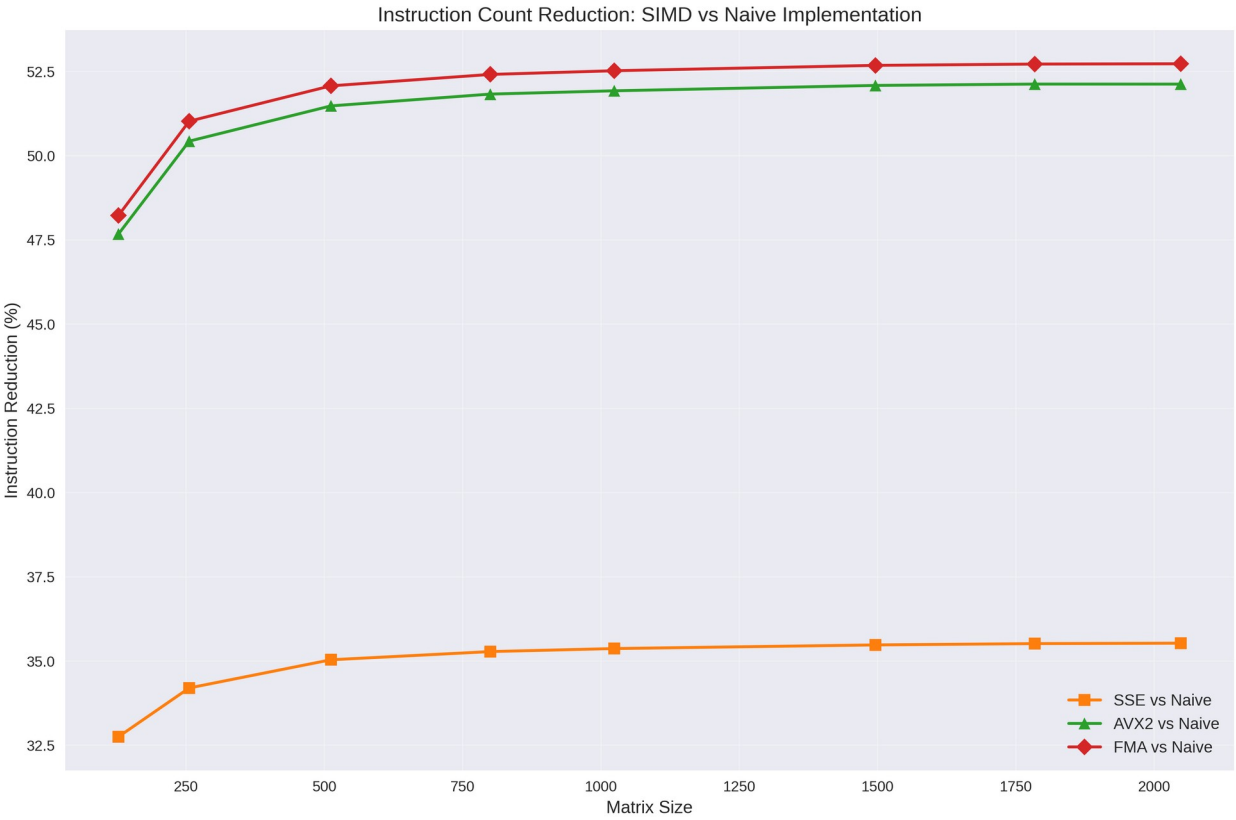
3. SIMD Intrinsics Used

- **Chosen Intrinsics(AMD Machine):**

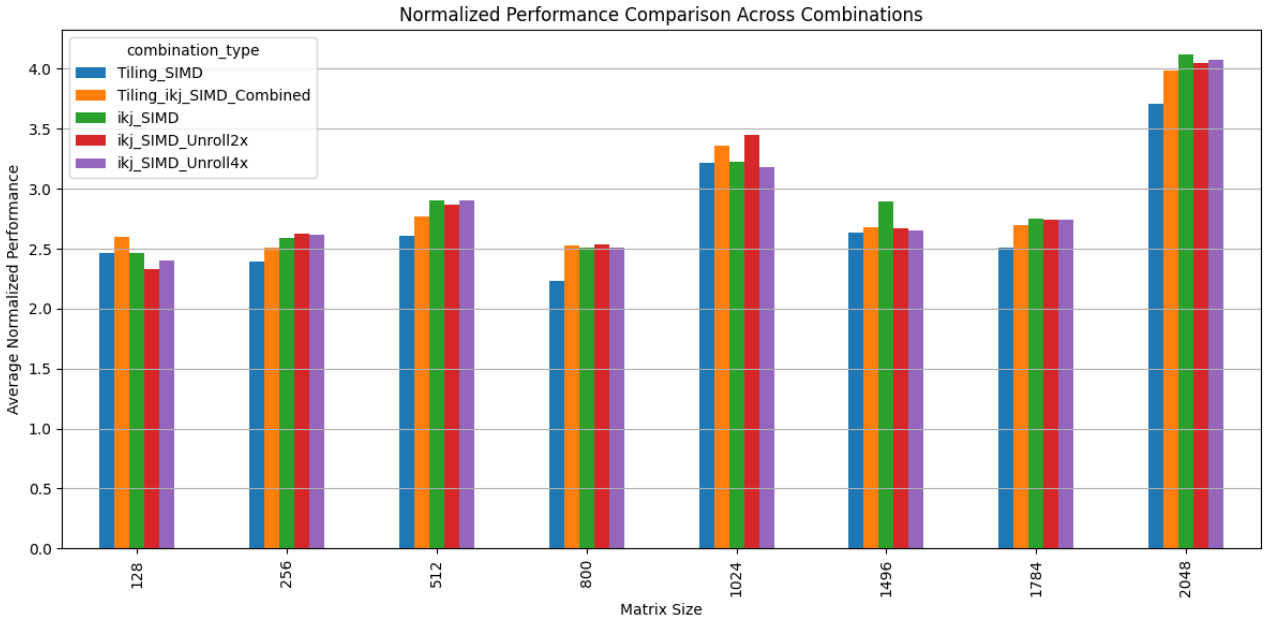
- `__m128`, `__m256`, `__m512` types for 128/256/512-bit registers.
- Load/Store: `_mm_load_ps`, `_mm256_load_ps`, `_mm512_load_ps`.
- Multiply: `_mm_mul_ps`, `_mm256_mul_ps`, `_mm512_mul_ps`.
- Add (accumulation): `_mm_add_ps`, `_mm256_add_ps`, `_mm512_add_ps`.
- Fused Multiply-Add (if available): `_mm_fmadd_ps`, `_mm256_fmadd_ps`, `_mm512_fmadd_ps`.

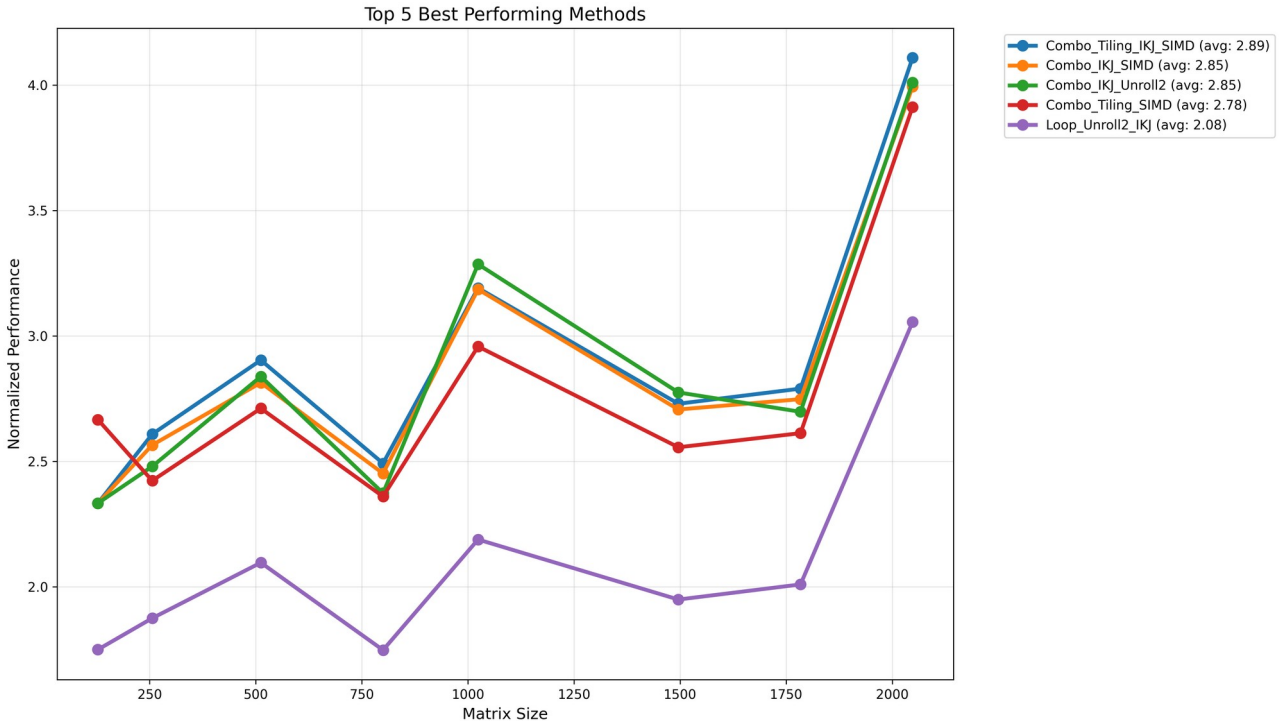


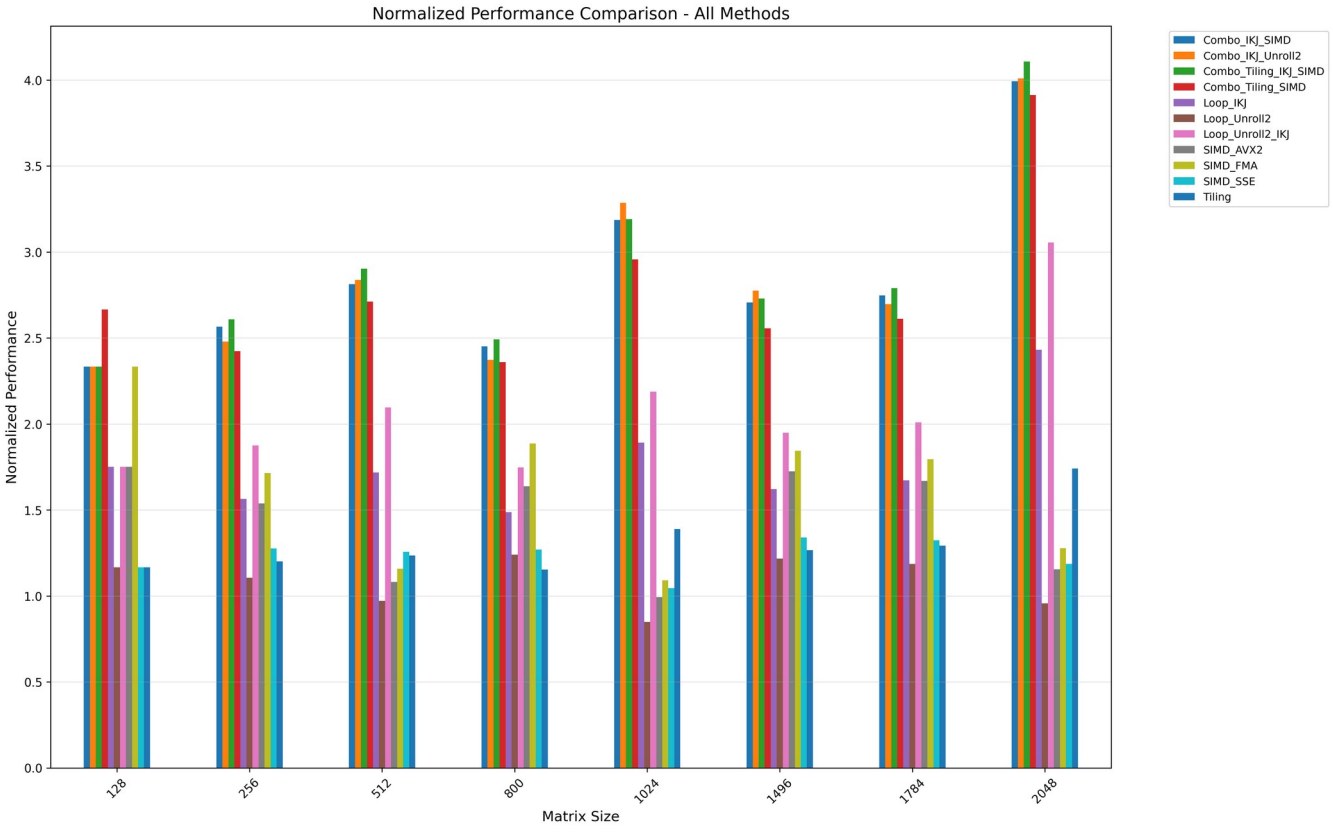

Instructions Comparison: Naive vs SIMD Implementations
Naive
SSE (128-bit)
AVX2 (256-bit)
FMA
Instructions (Billions)
0
50
100
150
200
250
300
350
250
500
750
1000
1250
1500
1750
2000
Matrix Size


Instruction Count Reduction: SIMD vs Naive Implementation
Instruction Reduction (%)
32.5
35.0
37.5
40.0
42.5
45.0
47.5
50.0
52.5
250
500
750
1000
1250
1500
1750
2000
Matrix Size
SSE vs Naive
AVX2 vs Naive
FMA vs Naive

Task 1D: Rancho’s Final Year Project







- **Case 1 (ikj + SIMD)**
  Basic vectorization, no explicit cache tiling or loop unrolling. Optimizes instruction-level parallelism via SIMD. Made the loop cache friendly via ikj.
- **Case 2 (Tiling + SIMD)**
  Adds cache tiling to keep data in L1/L2 longer, reducing cache misses. SIMD used inside tiles.
- **Case 3 (ikj + SIMD + unrolling ×2)**
  Same as Case 1 but tries to increase instruction throughput by unrolling the inner loop ×2.
- **Case 4 (ikj + SIMD + unrolling ×4)**
  Aggressive unrolling to reduce branch overhead and tried to increase ILP .
- **Case 5 (Tiling + ikj + SIMD combined)**
  Full combination: cache-aware (tiling), SIMD, and ikj ordering. This is theoretically the most balanced for large matrices.

Summary:

- **Small matrices (≤128):** All combinations run mostly within cache. Tiling and unrolling add overhead without much benefit. Basic SIMD is fine.
- **Medium matrices (256–512):** Loop unrolling (×2 or ×4) adds noticeable performance; less loop control overhead and better ILP help here.
- **Large matrices (≥800):** Tiling theoretically should help, but in our runs, tiling overhead (extra loops, bounds checking) and the theoritical optimal tile size (32) seem to cancel the cache benefits. The SIMD + unroll ×2 path (Case 3) emerges as consistently strong without increasing register pressure too much.
- **Aggressive unrolling (×4) can backfire** at very large sizes — register spilling and cache pressure can nullify gains.