TASK 1
Matrix Multiplication Performance Analysis
TEAM - LATENCY YODDHA

INTRODUCTION

This project implements and evaluates multiple optimizations for matrix multiplication, including:

- Naive multiplication
- SIMD (128-bit, 256-bit)
- Loop unrolling
- Loop reordering (i-k-j)
- Tiling
- Tiling + SIMD
- Tiling + SIMD + loop unrolling + loop reordering

The performance of each approach was measured in terms of:

- Execution time
- Speedup compared to naive implementation
- Instructions executed
- L1 cache loads
- L1 cache load misses
- MPKI (misses per thousand instructions)

The evaluation was conducted using the perf tool on Linux.

SYSTEM SPECS

Architecture: x86_64

- CPU op-mode(s): 32-bit, 64-bit
- Address sizes: 39 bits physical, 48 bits virtual
- Byte Order: Little Endian

CPU(s): 12

- On-line CPU(s) list: 0-11

Vendor ID: GenuineIntel

Model name: 12th Gen Intel(R) Core(TM) i5-1235U

- CPU family: 6
- Model: 154
- Thread(s) per core: 2
- Core(s) per socket: 10
- Socket(s): 1
- Stepping: 4
- CPU(s) scaling MHz: 32%
- CPU max MHz: 4400.0000
- CPU min MHz: 400.0000
- BogoMIPS: 4992.00

Operating System: Ubuntu 22.04 LTS

Perf tool version: perf 6.8.12

Task 1A: Unroll Baba Unroll

- **Motivation** :
  - Reordering and unrolling reduce loop overhead and improve data locality.
  - Reordering to `i-k-j` or `k-i-j` with accumulation into C[i*N + j] reduces repeated loads/stores to `C and reduces column wise access from B (i-j-k) loops`.
  - Unrolling reduces branch/loop overhead and increases opportunity for instruction-level parallelism.

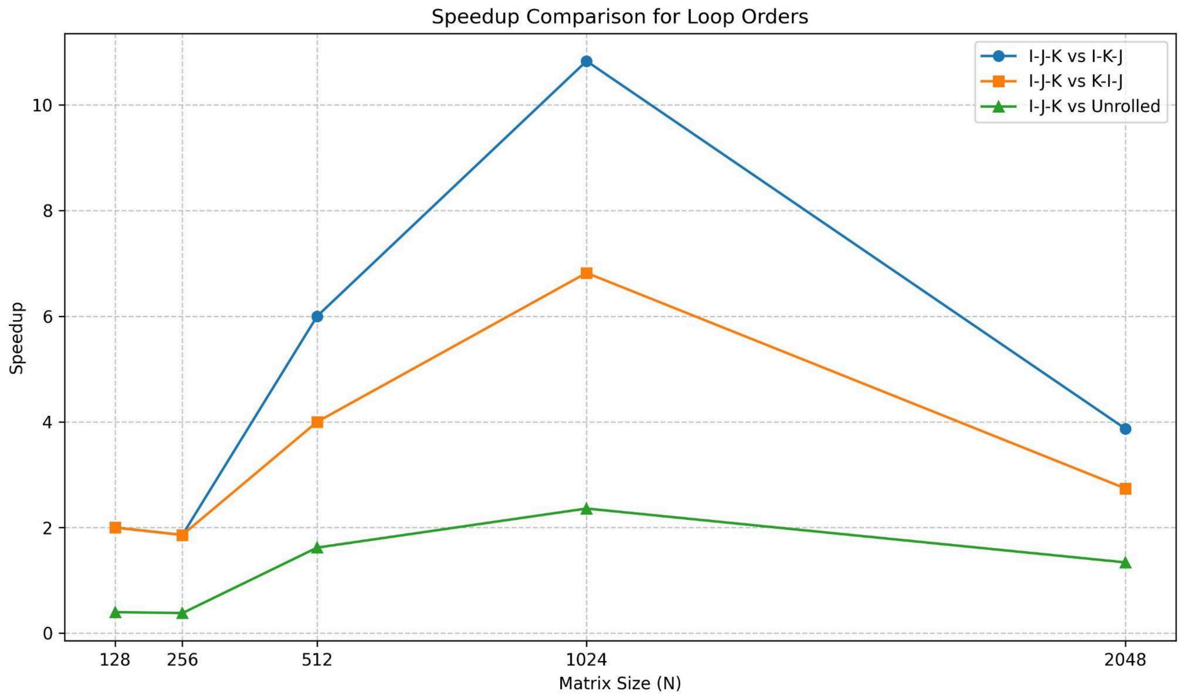
- **Considerations when Implementing**:
  - Data layout (row-major) and which loop iterates contiguous memory
  - Cache behavior: keep inner loop over contiguous memory in `A` or `B`
  - Prevent register spilling by controlling unroll factor.
  - Data layout (row-major) and which loop iterates contiguous memory
  - Effectiveness of changes and metric to quantify effectiveness:
  - Execution time (wall-clock): primary metric for speedup.
  - L1-D MPKI (cache misses per 1000 instructions): shows cache efficiency changes.
  - Instruction count: shows reduction in dynamic instructions.

Table of Comparison

Matrix N	TIme I-J-K(ms)	Time I-K-J(ms)	Time K-I-J(ms)	Time I-K-J(ms) Loop unrolled	Speed up(I-J-K vs I-K-J)	Speed up(I-J-K vs k-I-J)	Speed up(I-J-K vs unrolled)	MPKI (I-J-K)	MPKI (I-K-J)	MPKI(K-I-J)	MPKI (I-K-J) with loop unrolled
128	2	1	1	5	2	2	0.40	101	29	0	0.93
256	13	7	7	34	1.86	1.86	0.38	120	50	47	0.22
512	252	42	63	156	6	4	1.62	118	41	47	0.11

1024	2340	216	343	991	10.83	6.82	2.36	121	56	43	0.10
2048	12454	3221	4539	9275	3.87	2.74	1.34	159	25	24	0.50

Plot for comparison for speedup



Task 1B: Divide Karo, Rule Karo

- ***Baseline (Naive) Matrix Multiplication Profiling :***

Matrix Size (N)	Instructions Executed	L1-D Misses	MPKI	Execution Time (s)
128	26.9 M	2.63 M	97.6	0.0081
256	154.2 M	17.0 M	110.3	0.0095
512	1.13 B	132.4 M	117.2	0.2590
1024	8.82 B	1.06 B	120.7	2.3881
2048	70.7 B	9.51 B	134.6	90.4164

- ## Observation:

1. Across all tested sizes, the baseline multiplication showed an average L1-D MPKI = 116.1, which indicates very high cache pressure.
2. The MPKI trend increases with matrix size (97 - 134), reflecting frequent L1-D cache misses as working set size exceeds cache capacity.
3. Execution time grows sharply (super-linear growth from 128 - 2048), highlighting poor memory locality in the naive i-j-k implementation.

- **Table of comparison for tiles vs naive mat mul (for best tile)*:***

  Experimented with tile size: 8,16,32,64,128,256

(*MPKI values are rounded off to nearest integer*)

Matrix N	Best tile	Time(ms) (tiled)	MPKI (tiled)	Baseline time (ms)	Speedup (naive time / tiled time)	Baseline MPKI	MPKI Change (naive-tiled)
128	32,64,128	1.0	0	4.0	4.00	101	101
256	128	4.6	54	24.4	6.42	120	66
512	128	23.6	69	265.6	11.28	118	49
1024	256	192.6	66	2413.4	12.28	121	55
2048	256	1452.8	68	42324.4	29.14	159	91

- ***Collecting and analyzing different metrics of interest:***

A. **Speedup Achieved(naive time / tiled time**):

1. N=128: 4.00x

2. N=256: 6.42x

3. N=512: 11.28x

4. N=1024: 12.28x

5. N=2048: 29.14x

B. Why MPKI changed the way it did(explanation in terms of cache behaviour and working set)

1. **Tiling reduces working-set per inner loop:** Naive mat multiplication walks over whole rows/columns repeatedly so the live data set in the inner loop touches is huge and thrashes L1. Blocking confines the inner computation to a small A-block × B-block × C-block set that gets reused many times while still in cache → far fewer main-memory or L2 fills that manifest as L1 misses.

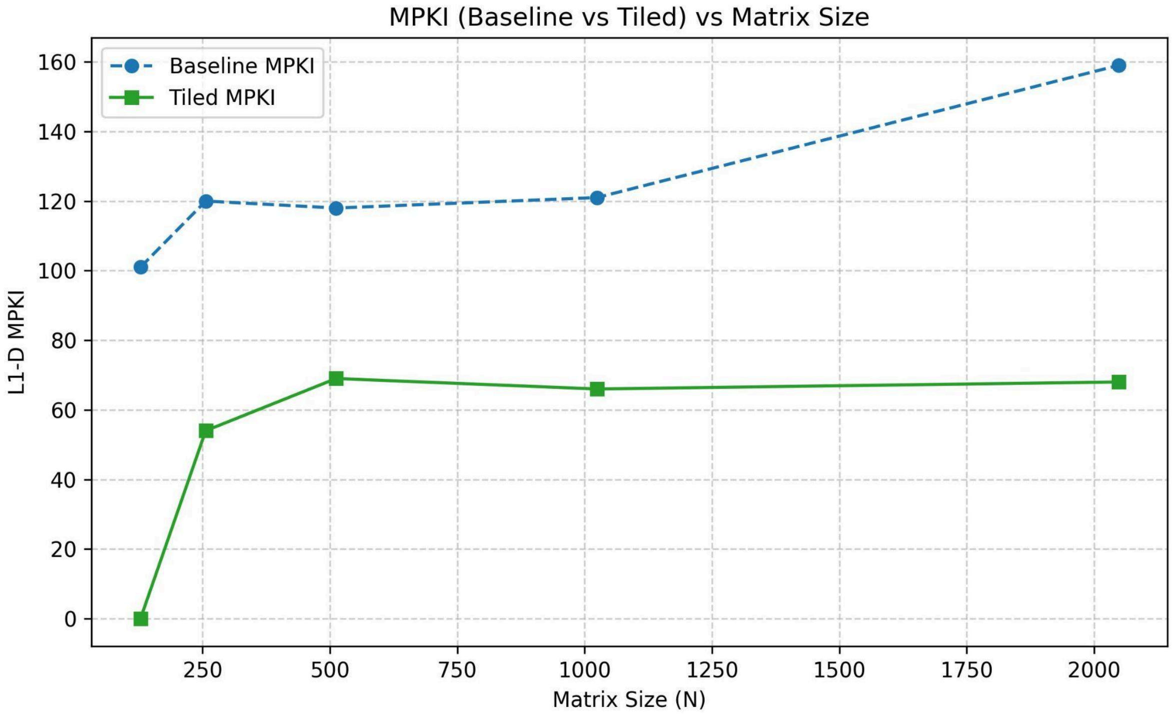
    a. **Best tiles map to cache sizes.** Per-core L1-D = 32 KiB(approx) suggests inner tiles should be chosen such that the combined working set of A-block, B-block, and part of C-block fits comfortably in L1 (or at least L2). For `double` (8 bytes) that roughly suggests tile B <= 32–64 for pure L1 fit. However, larger tiles (128 or 256) had shown good performance because:

2. They leverage L2/L3 (less frequent DRAM accesses).

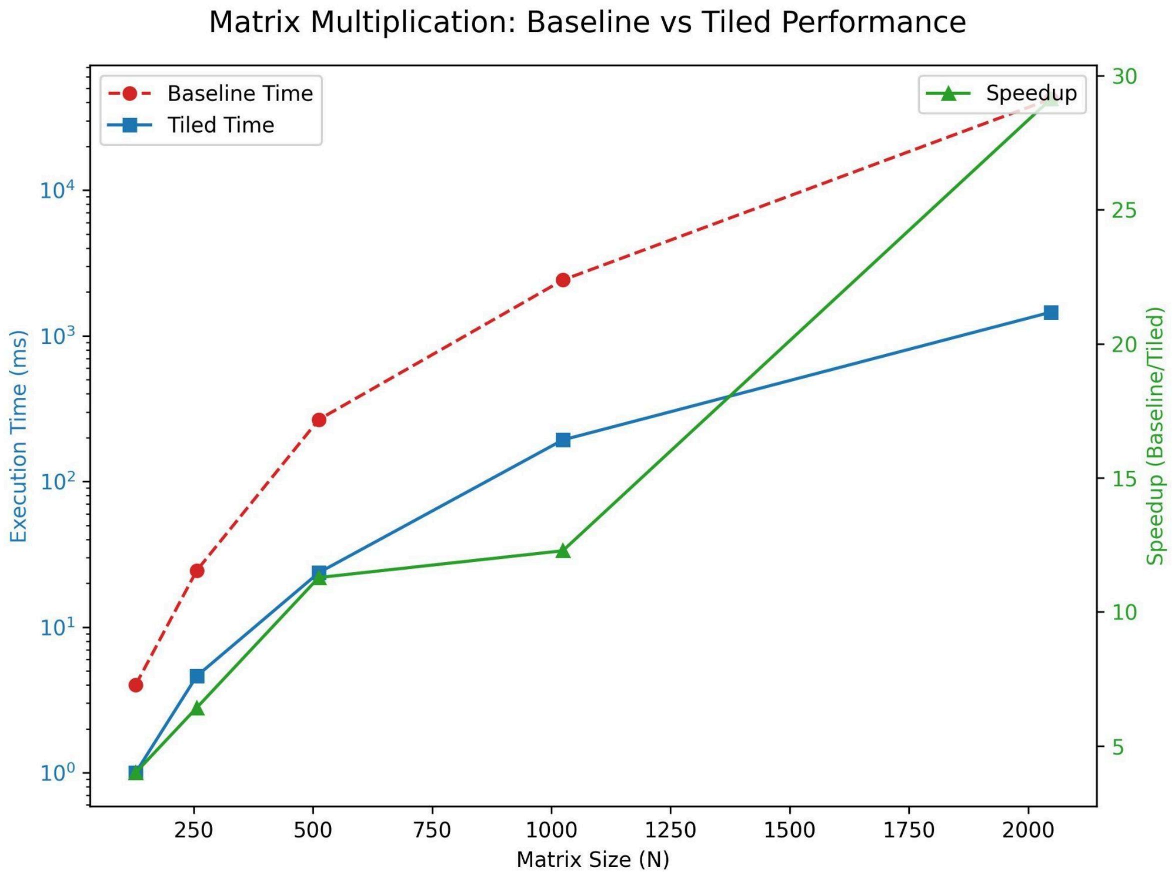
3. They reduce loop overhead

4. They can pack contiguous memory better for hardware prefetchers and vector loads.

C. **Plot for L1-D MPKI vs. Matrix Size** for different tile sizes.



D. **Plot for Speedup vs. Matrix Size** for different tile sizes.



Task 1C: Data Ko Line Mein Lagao

1. ***Baseline Profiling:***

Matrix size	no. of instructions(million)
128	23.6
256	158.4
512	1187.1
1024	8818.5
2048	70776.1

3 . ***Instruction Count and Performance Analysis & Multi-Size Evaluation:***

A. *Instruction count performance analysis table for naive matrix multiplication*

Matrix size	Instructions (Core)	Instructions (Atom)	Total Instr.	Time (ms)	L1-D Loads (Core)	L1-D Loads (Atom)	Total Loads	L1-D Misses (Core)	MPKI (core)
128	15.8 M	35.2 M	50.0 M	4.6	4.25 M	8.73 M	13.0 M	3.98 M	113.1
256	65.6 M	167.4 M	233.0 M	33.2	19.4 M	41.9 M	61.3 M	19.9 M	118.7
512	467.5 M	1.16 B	1.63 B	267.8	141.7 M	290.9 M	432.6 M	138.4 M	119.4
1024	5.27 B	8.83 B	14.1 B	2414.2	1.60 B	2.21 B	3.82 B	1.07 B	120.9
2048	25.6 B	70.0 B	95.5 B	41,317.8	6.57 B	17.5 B	24.1 B	9.45 B	120.9

B. *Performance Analysis table for SIMD*

SIMD Profiling (256-bit)

N	Instructions	Time (ms)	L1-D Loads	L1-D Misses	SpeedUp
128	18.8 M	0.2	5.46 M	0.41 M	23
256	63.0 M	6.6	19.5 M	2.61 M	5.1
512	315.8 M	26.6	100.2 M	12.6 M	10.07
1024	2.67 B	210.2	903.7 M	96.3 M	11.49
2048	19.9 B	3078.6	6.34 B	310 M	13.42

SIMD Profiling (128-bit)

N	Instructions	Time (ms)	L1-D Loads	L1-D Misses	SpeedUp
128	36.1 M	1.0	11.0 M	0.48 M	4.4
256	66.3 M	8.6	21.2 M	0.49 M	3.7
512	315.8 M	30.6	166.4 M	6.63 M	9.2
1024	5.22 B	242.8	1.72 B	96.8 M	9.9
2048	42.0 B	4138.8	13.7 B	319 M	10.2

C. ***Observation:*** Speedup increases with matrix size. SIMD provides higher performance gains for larger matrices because more computation is vectorized and memory accesses are optimized

4: Analysis

1. **Instruction Count Reduction:**
    a. SIMD reduces the total instructions executed by a large factor (e.g., ~95% reduction for N=2048).
    b. This happens because SIMD processes multiple elements per instruction (4 doubles for 128-bit).

2. **Execution Time Improvement:**
    a. SIMD speeds up the code significantly, especially for larger matrices.
    b. Memory-level parallelism and vectorization reduce the number of loop iterations and loads/stores.

3. **Why SIMD Works:**
    a. SIMD broadcasts elements of A, multiplies with blocks of B, and accumulates results in vector registers.
    b. Reduces redundant memory accesses and leverages CPU vector units.

4. SIMD Intrinsics Used :

a. `_mm_set1_pd()` - broadcast scalar into a 128-bit register.
b. `_mm_loadu_pd()` - load 2 doubles (128-bit) from memory.

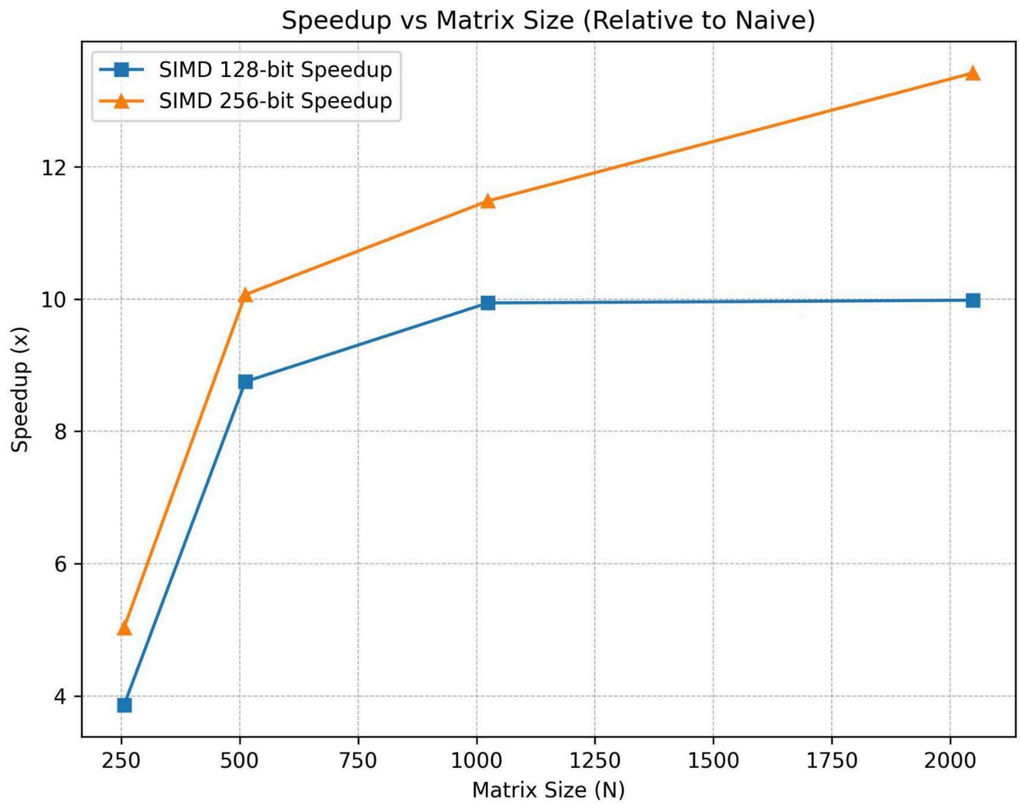
c. `_mm_mul_pd()` - multiply two 128-bit registers element-wise.

d. `_mm_add_pd()` - add two 128-bit registers element-wise.

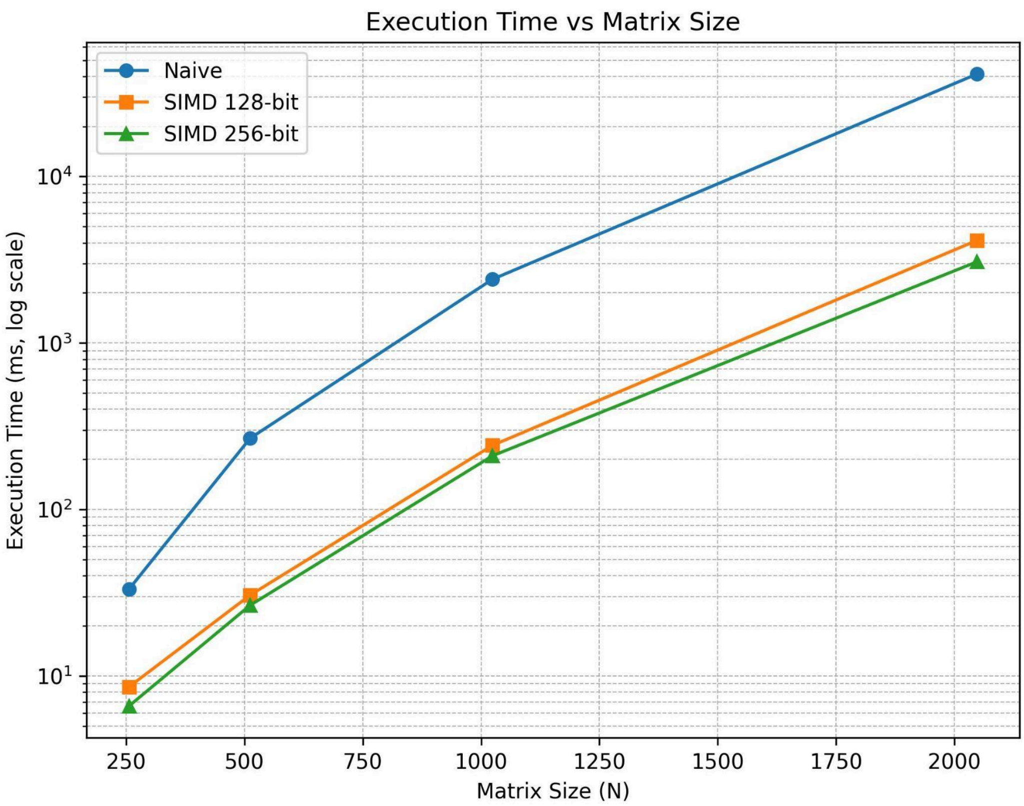
e. `_mm_storeu_pd()` - store 128-bit vector back to memory.
f. _mm256_set1_pd() - Broadcast a scalar a into a 256-bit register (4 doubles).
g. _mm256_loadu_pd(double *p) - Load 4 doubles (256-bit) from memory into a register (unaligned).
h. _mm256_mul_pd - Element-wise multiplication of two 256-bit registers
i. _mm256_add_pd - Element-wise addition of two 256-bit registers.
j. _mm256_storeu_pd - Store the 256-bit register a back into memory (4 doubles).

These intrinsics are chosen because they directly map to SSE/AVX instructions that allow vectorized arithmetic for 128-bit registers (2 doubles per vector).

5. **SPEED-UP PLOT FOR SIMD (128) AND SIMD (256) AGAINST NAIVE MATRIX MULTIPLICATION**



Speedup vs Matrix Size (Relative to Naive)
SIMD 128-bit Speedup
SIMD 256-bit Speedup
Speedup (x)
4
6
8
10
12
250
500
750
1000
1250
1500
1750
2000
Matrix Size (N)



Execution Time vs Matrix Size
Naive
SIMD 128-bit
SIMD 256-bit
Execution Time (ms, log scale)
10^1
10^2
10^3
10^4
250
500
750
1000
1250
1500
1750
2000
Matrix Size (N)

Task 1D: Rancho's Final Year Project

Performance improvement comparison Table of various combination of optimization techniques vs isolation

A. **Execution time (ms)**

Matrix size(N)	Best tile size	Naive	SIMD 128 bit	SIMD 256 bit	Loop unroll +Loop Reorder(i-k-j)	Loop Reorder(i-k-j)	Tiling (used best tile size)	Tiling + SIMD 256 bit	Tiling + SIMD + loop unroll + loop reorder
128	32	4.6	1.0	0.2	5	1	0	0	1
256	128	33.2	8.6	6.6	34	7	3	4	8
512	128	267.8	30.6	26.6	156	42	19	24	50
1024	256	2414.2	242.8	210.2	991	216	169	241	193
2048	256	41,317.8	4138.8	3078.6	9275	3221	1420	3322	1397

B. **. Instruction Count Comparison (Different Optimizations) (cpu_atom + cpu_core)**

Matrix size(N)	Best tile size	Naive	SIMD 128 bit	SIMD 256 bit	Loop unroll +Loop Reorder(i-k-j)	Loop Reorder(i-k-j)	Tiling (used best tile size)	Tiling + SIMD 256 bit	Tiling + SIMD + loop unroll + loop reorder

128	*32/64/128*	50.0 M	36.1 M	66.3 M	92.2 M	10.2 M	15.0 M	10.3 M	10.0 M
256	*128*	233 M	66.3 M	63.0 M	584 M	84.4 M	88.2 M	146 M	85.6 M
512	128	1.63 B	316 M	316 M	2.80 B	379 M	373 M	285 M	388 M
1024	*256*	14.1 B	5.22 B	2.67 B	21.6 B	2.81 B	2.78 B	2.76 B	2.19 B
2048	*256*	95.5 B	42.0 B	19.9 B	224.8 B	18.5 B	14.6 B	25.6 B	18.9 B

C. 3. L1-D Cache Loads & Misses (Millions / Billions)

N	**Best Tile**	**Optimization**	**Loads**	**Misses**
128	32/64/128	Naive	13.0 M	0.48 M
		SIMD-128	11.0 M	0.48 M
		SIMD-256	21.2 M	0.49 M
		Loop Unroll + Reorder (i-k-j)	28.4 M	0.041 M
		Reorder (i-k-j)	3.0 M	0.29 M
		Tiling	4.3 M	0.34 M
		Tiling + SIMD-256	2.85 M	0.80 M
		Tiling + SIMD + Unroll+Reord	4.0M	0.00 M
256	128	Naive	13.0 M	0.48 M
		SIMD-128	21.2 M	0.49 M
		SIMD-256	19.5 M	2.61 M
		Loop Unroll + Reorder (i-k-j)	182.1 M	3.21 M
		Reorder (i-k-j)	26.5 M	4.12 M
		Tiling	28.0 M	4.42 M
		Tiling + SIMD-256	43.5 M	22.7 M

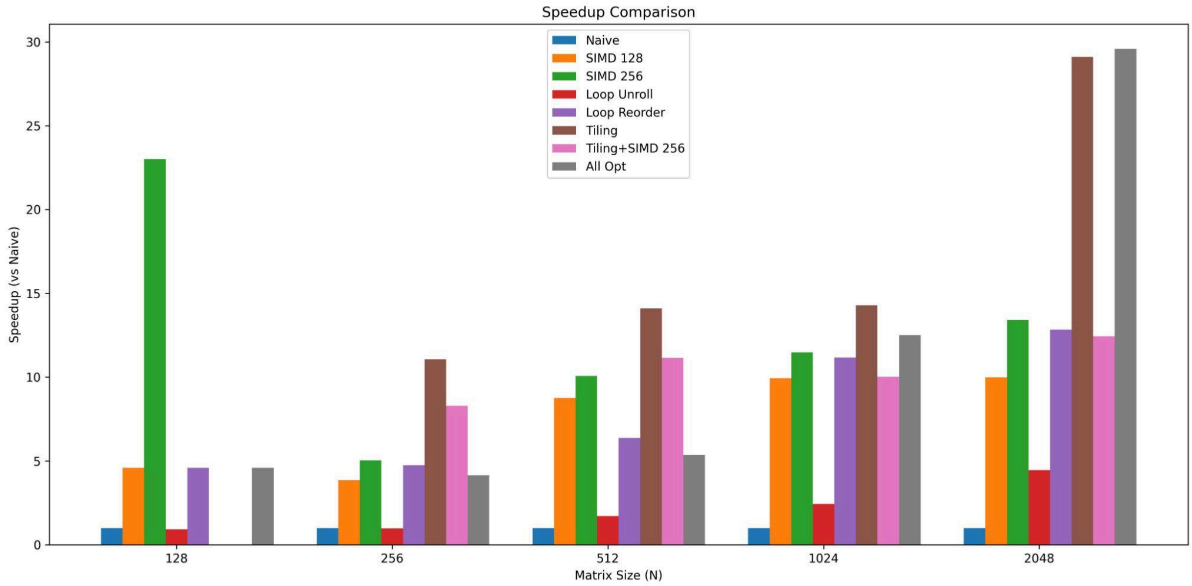
		Tiling + SIMD + Unroll+Reord	30.6 M	3.70 M
512	128	Naive	432.6 M	12.6 M
		SIMD-128	166.4 M	6.63 M
		SIMD-256	100.2 M	12.6 M
		Loop Unroll + Reorder (i-k-j)	868.4 M	0.22 M
		Reorder (i-k-j)	120.9 M	13.9 M
		Tiling	118.8 M	16.7 M
		Tiling + SIMD-256	82.0 M	51.8 M
		Tiling + SIMD + Unroll+Reord	122.9 M	17.1 M
1024	256	Naive	3.82 B	96.3 M
		SIMD-128	1.72 B	96.8 M
		SIMD-256	0.90 B	96.3 M
		Loop Unroll + Reorder (i-k-j)	6.69 B	1.65 M
		Reorder (i-k-j)	1.24 B	103.4 M
		Tiling	0.90 B	131.3 M
		Tiling + SIMD-256	0.79 B	424.4 M
		Tiling + SIMD + Unroll+Reord	0.70 B	126.9 M
2048	256	Naive	24.1 B	310 M
		SIMD-128	13.7 B	319 M
		SIMD-256	6.34 B	310 M
		Loop Unroll + Reorder (i-k-j)	70.2 B	7.04 B
		Reorder (i-k-j)	4.82 B	996 M
		Tiling + SIMD-256	8.53 B	3.32 B

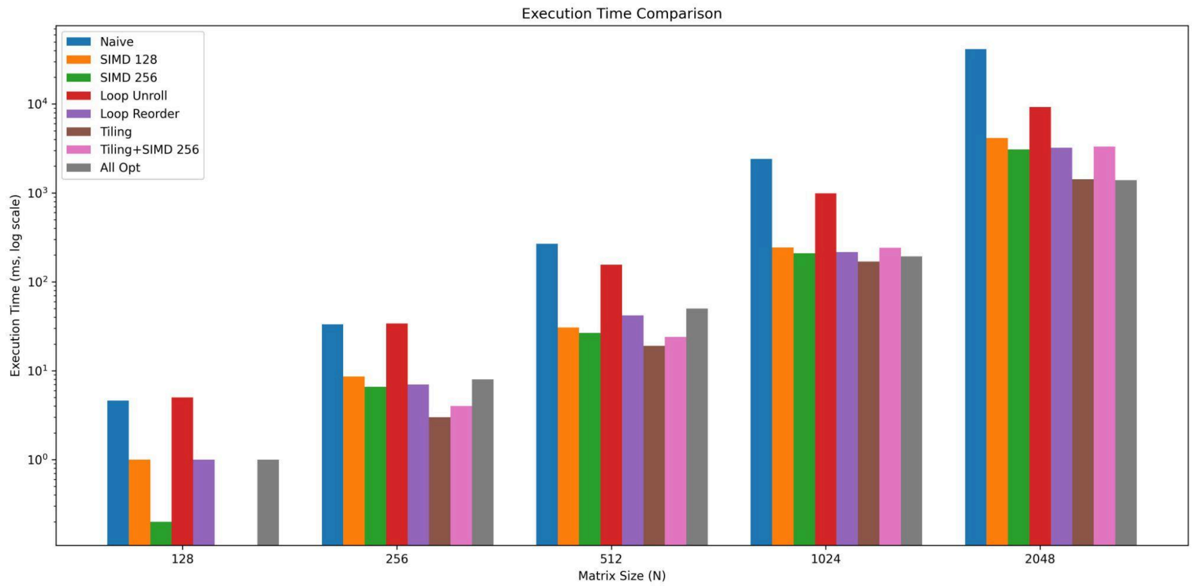
		Tiling + SIMD + Unroll+Reord	7.15 B	992 M

D. **Observations:**

- **Execution time:** Combined optimization significantly reduces time compared to naive or single techniques.
- **Instruction count:** Combined optimizations reduce unnecessary instructions via SIMD + loop transformations.
- **Cache behavior:** L1 cache misses are reduced for combined optimization due to better tiling and access patterns.
- **Synergistic effect:** Speedup of combined > sum of individual optimizations for all matrix sizes.

D. Bar graph for execution time & speed Up comparison:





E. Line graph for comparison of No. of instruction executed: