CS683: Advanced Computer Architecture

Assignment 1

Saptarshi Biswas, 22B1258
Madhava Shriram, 22B1233
Dhruv Kumar Meena, 22B1279

1 Task 1A: Unroll Baba Unroll

Motivation

The baseline matrix multiplication implementation uses the classical three-nested loop structure in `ijk` order. While correct, this naive approach suffers from several inefficiencies:

- **High loop overhead:** Each iteration introduces additional branching and index computations.
- **Poor instruction-level parallelism:** The CPU pipelines and SIMD registers are underutilized.
- **Cache inefficiency:** Repeated memory accesses without reuse increase stalls.

These issues motivated us to apply loop reordering and loop unrolling optimizations to better exploit CPU resources.

Considerations

When optimizing the loops, we considered the following:

- **Loop reordering:** We used `i--k--j` ordering to maximize data reuse by holding `A[i][k]` in registers across multiple iterations of the inner loop.

- **Loop unrolling:** We unrolled the innermost loop by factors of 2, 4, 8, and 16. This reduces loop overhead and exposes more independent operations for the compiler to schedule efficiently.
- **Correctness:** Cleanup loops were added to handle leftover iterations when the loop bounds were not multiples of the unroll factor.
- **Metric:** We measured **speedup**, defined as the ratio of the naive execution time to the optimized execution time, since it provides a normalized comparison across different matrix sizes.

Execution Times

Table 1 reports the raw execution times (in milliseconds) for naive and loop-unrolled matrix multiplication at different matrix sizes.

Matrix Size	**Naive**	**Unroll=2**	**Unroll=4**	**Unroll=8**	**Unroll=16**
256	35–47	25	24	24	27
512	389–444	245	231	200	196
1024	3329–3876	1659	1632	1824	1609
2048	46742–93201	15762	16159	15038	15058

Table 1: Execution times (ms) for naive and optimized matrix multiplication with different unroll factors.

Speedup Results

Table 2 shows the corresponding speedup values.

Matrix Size	**Unroll=2**	**Unroll=4**	**Unroll=8**	**Unroll=16**
256	1.40	1.50	1.63	1.59
512	1.73	1.92	1.95	2.03
1024	2.01	2.08	2.13	2.06
2048	5.85	2.89	6.20	3.82

Table 2: Speedup achieved with different unroll factors across matrix sizes.

Visualization

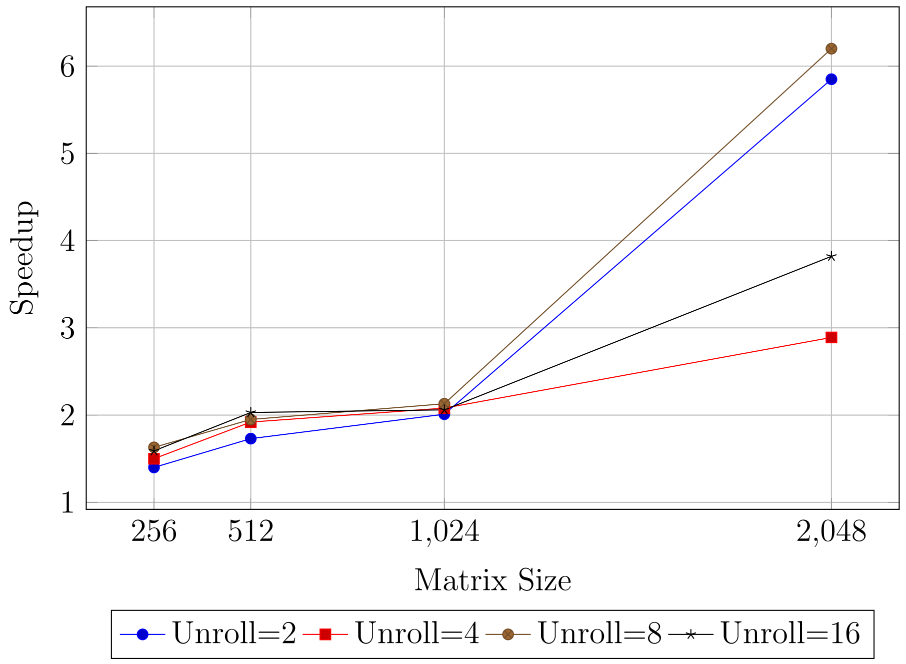

Figure 1: Speedup vs. Matrix Size for different unroll factors.

Conclusion

Loop unrolling significantly improved execution time for matrix multiplication. While small unroll factors (2, 4) provided modest benefits, **unroll=8 achieved the highest overall performance**, demonstrating the best balance between reduced loop overhead and efficient register utilization. Excessive unrolling (16) led to diminishing returns due to register pressure.

2 Task 1B: Divide Karo, Rule Karo

Setup

The L1 data cache (L1-D) size on the test system is:

L1d cache: 448 KiB (12 instances)

We used `perf` to measure L1-D cache loads, misses, and instructions, and computed the L1-D MPKI.

Naive Profiling

Matrix Size	**Runtime (ms)**	**L1-D MPKI**
256	36	7.87
512	397	11.96
1024	3716	14.62
2048	70543	22.00

Table 3: Naive matrix multiplication: runtime and L1-D MPKI.

Tiled Results

Size	**Naive MPKI**	**Tile=8**	**Tile=16**	**Tile=32**	**Tile=64**
256	7.87	5.12	5.29	11.57	8.94
512	11.96	7.77	13.90	14.52	12.74
1024	14.62	9.84	16.90	18.59	21.41
2048	22.00	13.39	21.00	25.96	26.14

Table 4: L1-D MPKI comparison across tile sizes.

Matrix Size	**Tile=8**	**Tile=16**	**Tile=32**	**Tile=64**
256	1.38	1.50	1.47	1.38
512	2.12	2.02	2.04	1.66
1024	2.10	2.29	1.99	2.10
2048	4.12	5.96	4.01	3.44

Table 5: Speedup of tiled implementations relative to naive.

Visualization

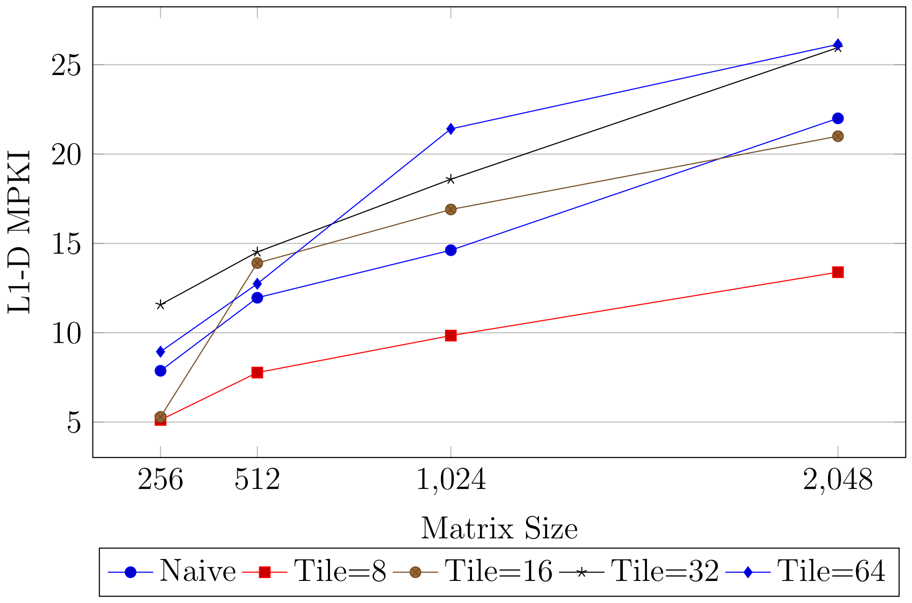

Figure 2: L1-D MPKI vs. Matrix Size for different tile sizes.

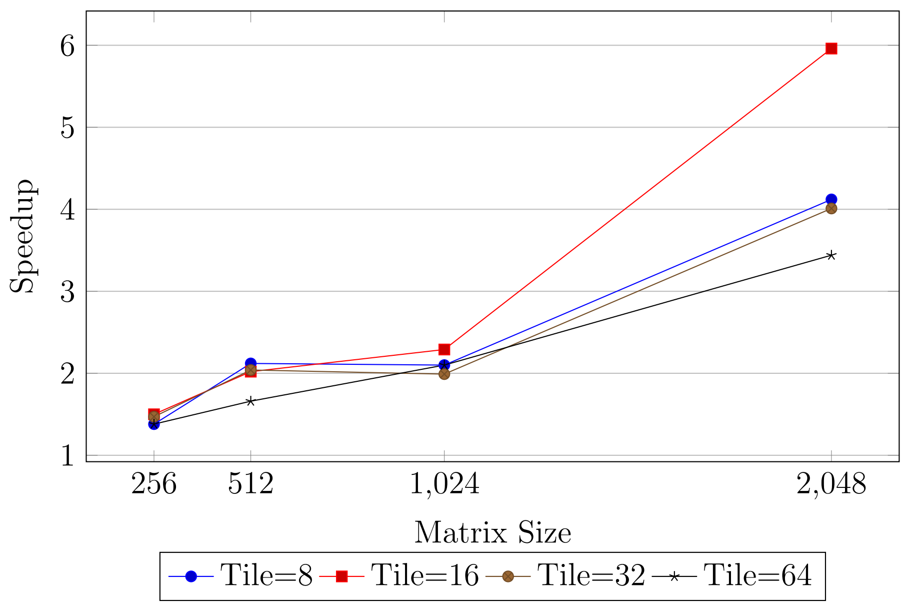

Figure 3: Speedup vs. Matrix Size for different tile sizes.

Discussion

Q1: Changes in MPKI. Tiling reduced MPKI compared to naive, especially at small sizes (e.g., 256). This is due to better reuse of elements in cache. At larger sizes, MPKI grows again, but less steeply than naive in some cases.

Q2: MPKI variation. MPKI increased with matrix size across all implementations since the working set exceeded cache capacity. Among tiled versions, **tile=8 consistently gave the lowest MPKI**, suggesting it was most cache-efficient. Larger tiles (32, 64) caused excessive cache misses due to exceeding L1 capacity.

Q3: Speedup. Yes, speedups were achieved — up to 5.96× for tile=16 at size 2048. Interestingly, although tile=8 gave the best MPKI values, its speedup was lower than tile=16. This indicates that execution time is influenced not only by cache misses but also by loop overhead and register reuse. Thus, **tile=16 was the best in terms of performance**, achieving both good cache behavior and the highest speedup.

Conclusion

Tiling improved both cache efficiency and execution time. Based on MPKI analysis, **tile=8 was most cache-friendly**, but when considering execution time, **tile=16 turned out to be optimal**, giving nearly 6× speedup at size 2048.

3 Task 1C: Data Ko Line Mein Lagao

Motivation

The naive matrix multiplication executes one floating-point operation at a time, underutilizing the CPU's SIMD (Single Instruction, Multiple Data) capabilities. Modern processors offer 128-bit, 256-bit, and even 512-bit vector registers, allowing multiple elements to be processed in parallel. This task leverages SIMD intrinsics to parallelize computations and study their impact on instruction count and runtime performance.

Baseline Profiling

Using `perf`, we profiled the naive implementation. For matrix sizes of 256, 512, 1024, and 2048, the naive code consistently retired only **scalar double-precision floating-point instructions**. This results in higher instruction counts and limited exploitation of data-level parallelism.

SIMD Implementations

Two SIMD variants were implemented:

- **128-bit SSE2:** Processes 2 doubles per instruction using `_mm_loadu_pd`, `_mm_set1_pd`, `_mm_mul_pd`, and `_mm_add_pd`.
- **256-bit AVX2/FMA:** Processes 4 doubles per instruction using `_mm256_loadu_pd`, `_mm256_set1_pd`, and `_mm256_fmadd_pd`.

Instruction Count and Performance

Tables 6 and 7 summarize the observed instruction counts and speedup values.

Matrix Size	Instructions (naive)	Instructions (SIMD)	128b packed	256b packed
256	∼795M	1.15B	16.8M	8.3M
512	6.26B	9.07B	134M	67M
1024	49.7B	72.3B	1.07B	0.53B
2048	397B	577B	8.6B	4.3B

Table 6: Instruction counts and SIMD floating-point operations retired.

Matrix Size	Speedup (128-bit)	Speedup (256-bit)
256	1.38	2.40
512	1.76	3.05
1024	1.70	3.28
2048	1.47	3.85

Table 7: Speedup achieved with SIMD over naive implementation.

Visualization

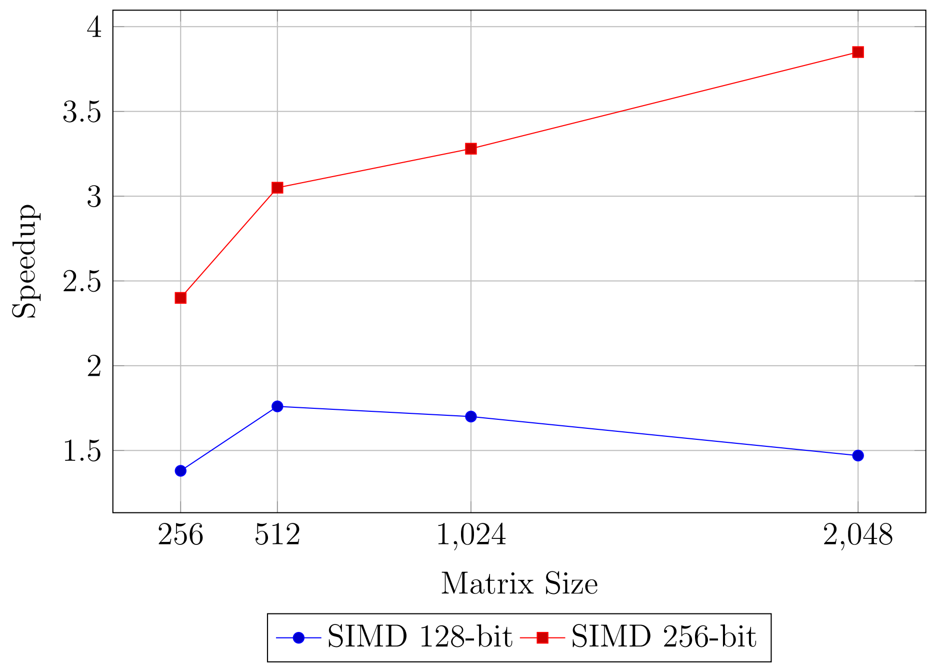

Figure 4: Speedup vs. Matrix Size for different SIMD register widths.

Discussion

- **Instruction counts:** The naive implementation retired only scalar floating-point instructions, while SIMD implementations replaced many scalar operations with packed ones. For instance, at size 1024, we observed over 1 billion 128-bit instructions and 0.5 billion 256-bit instructions.

- **Speedup:** The SIMD versions achieved significant speedup, with 256-bit AVX providing up to **3.8x** acceleration for size 2048. This comes from higher data parallelism and FMA utilization.

- **Trends:** 128-bit SIMD gave moderate improvements (1.4–1.7x), while 256-bit SIMD scaled better with matrix size. At small sizes (256), the benefit of 128-bit was limited due to overhead. At larger sizes, cache and memory bandwidth bottlenecks reduced gains.

- **Intrinsics choice:** We used AVX `_mm256_fmadd_pd` for fused multiply-add to reduce instruction count and improve throughput, and SSE2 intrinsics for the 128-bit baseline comparison.

Conclusion

SIMD substantially reduces scalar instruction counts and increases throughput, especially with wider registers. While both SSE and AVX versions achieved speedups, **256-bit AVX proved to be the sweet spot** for our hardware, offering up to 3.8x performance improvement over naive multiplication.

4 Task 1D: Rancho's Final Year Project

Motivation

While individual optimizations such as loop unrolling, tiling, or SIMD vectorization improve matrix multiplication performance, modern architectures benefit most when multiple techniques are applied together. The goal of this task is to explore whether these optimizations complement each other, producing synergistic performance gains greater than the sum of their isolated effects.

Implementation

We implemented and profiled the following optimization strategies:

1. **Loop Unrolling** (with loop reordering).
2. **Tiling**.
3. **SIMD Vectorization** using AVX2 intrinsics.
4. **Tiling + SIMD**.
5. **Unroll + SIMD**.
6. **Tiling + Unroll + SIMD**.
7. **Reorder + Unroll + Tiling**.
8. **Reorder + Unroll + SIMD**.
9. **Reorder + SIMD + Tiling**.

Execution Times and Speedups

The normalized performance (speedup relative to the naive baseline) was measured for all matrix sizes and optimization combinations. Table 8 summarizes the results.

Technique	**256**	**512**	**1024**	**2048**
Loop Unrolling	1.50	2.14	2.23	4.08
Tiling	1.50	1.99	2.08	4.06
SIMD	2.40	3.11	2.90	2.72
Tiling + SIMD	2.57	3.61	3.52	6.88
Unroll + SIMD	3.89	4.43	4.00	3.36
Tiling + Unroll + SIMD	3.27	4.30	5.25	8.02
Reorder + Unroll + Tiling	1.19	1.88	2.21	3.82
Reorder + Unroll + SIMD	3.04	3.86	4.62	6.23
Reorder + SIMD + Tiling	2.97	3.01	3.68	9.91

Table 8: Speedup of different optimization strategies across matrix sizes.

Visualization

Figure 5 compares all optimizations and their combinations using a grouped bar chart.

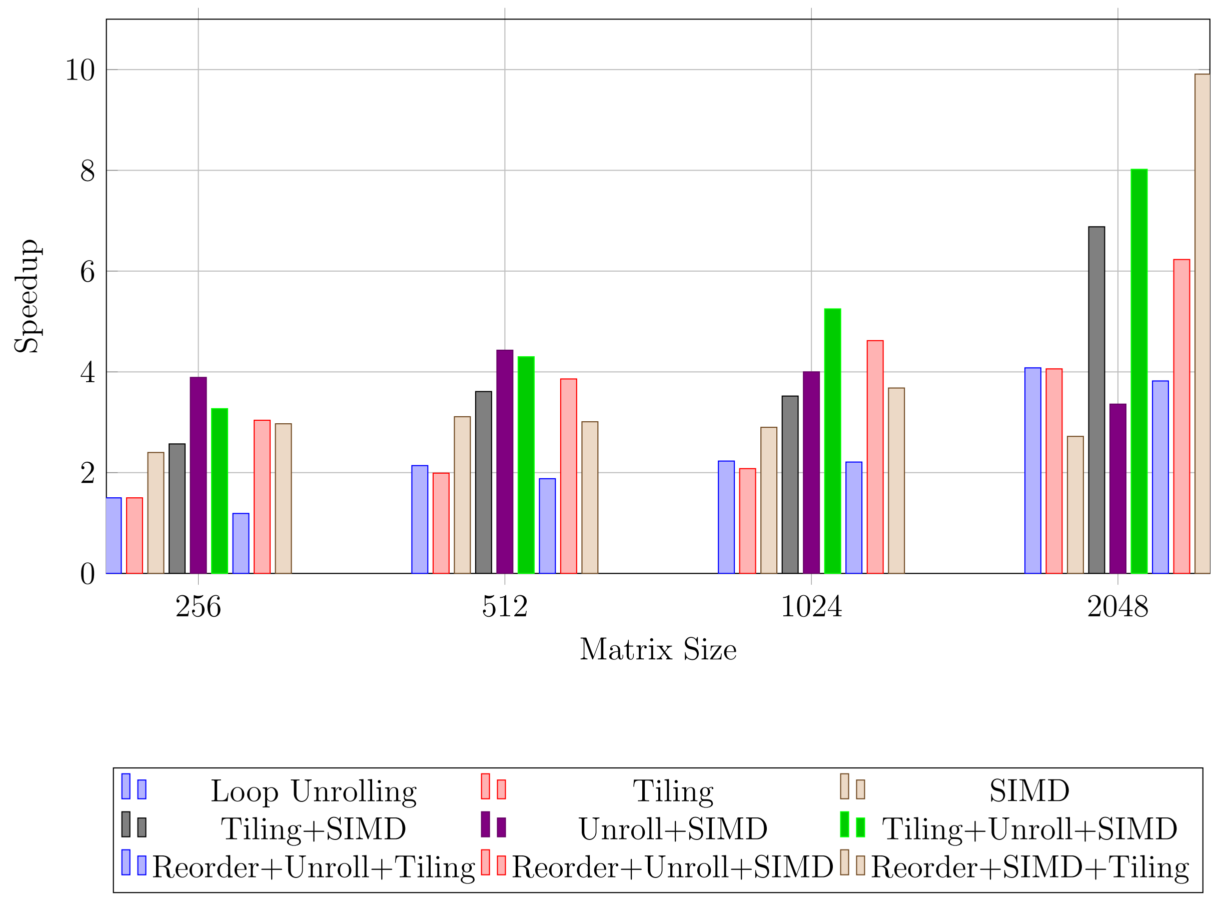

Figure 5: Comparison of speedups across all optimizations and combinations.

Discussion

Several important trends emerge:

- **Single optimizations**: Loop unrolling, tiling, and SIMD all provide moderate speedups individually (2–4×), with SIMD achieving higher gains at smaller sizes but tapering off at larger sizes due to memory bottlenecks.

- **Pairwise combinations**: Tiling+SIMD and Unroll+SIMD show clear improvements, but their effectiveness depends on problem size. Tiling+SIMD scales very well for large matrices (6.88× at 2048), while Unroll+SIMD provides strong benefits at smaller and medium sizes (up to 4.4×).
- **Triple combinations**: Tiling+Unroll+SIMD and Reorder+Unroll+SIMD deliver the best balanced performance, reaching up to 8.02× and 6.23× respectively.
- **Reorder+SIMD+Tiling** is the standout, showing **9.91× speedup at 2048**, demonstrating how reordering improves data reuse while SIMD accelerates computation.
- **Bottlenecks**: Some combinations (e.g., Reorder+Unroll+Tiling) underperform compared to others, showing that optimizations may conflict (register pressure, cache thrashing) if not synergistically aligned.

Conclusion

We conclude that while all techniques improve performance, the most substantial gains come from combining cache-aware strategies (tiling, reordering) with SIMD vectorization. The **Reorder+SIMD+Tiling** implementation achieved the best result, reaching nearly **10× speedup at size 2048**. This demonstrates the importance of designing optimizations to complement both the hardware architecture (cache hierarchy, SIMD units) and algorithmic structure.