Task 1A: Unroll Baba Unroll

1. *What motivated you to make changes to your code?*

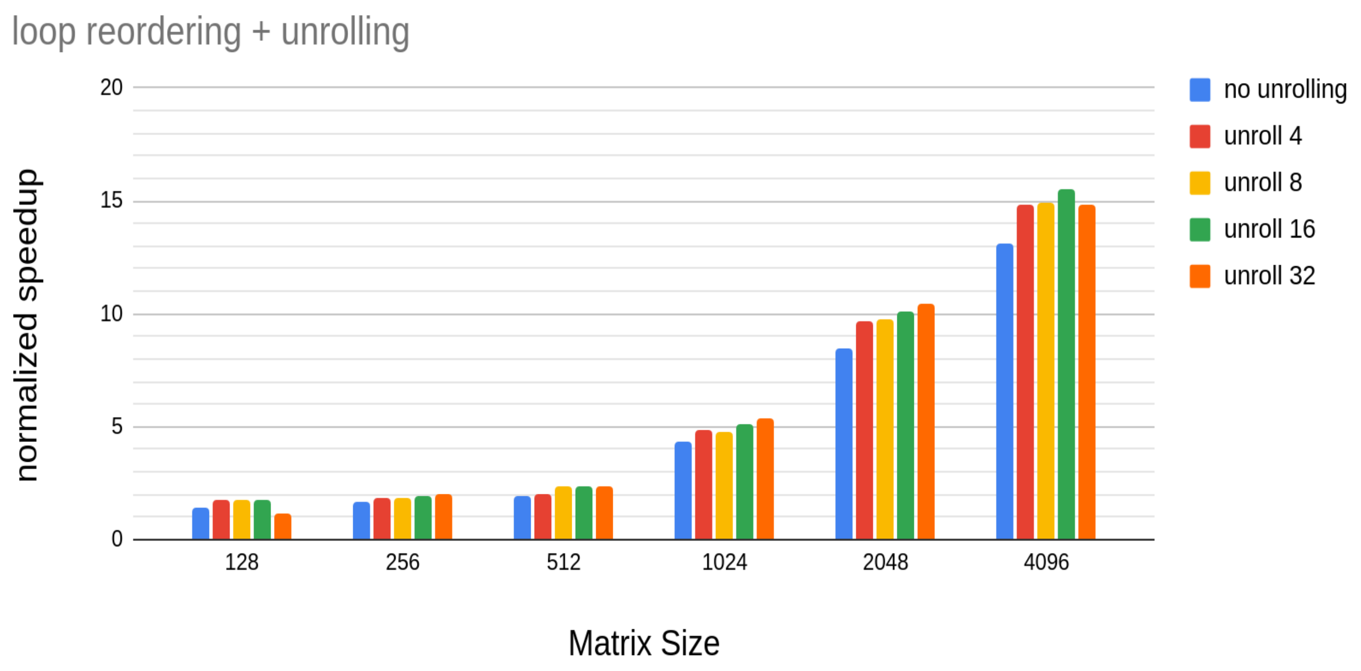
Ans: To implement what was taught in class and see the improvements in performance by ourselves.

2. *What considerations did you take into account when implementing those changes?*

Ans: We considered the cache block size, the size of individual elements (doubles in this case) and the cache size when implementing the changes.

3. *How effective are your changes? (Which metric will you use to quantify effectiveness and why?)*

Ans: There is a great improvement in performance after implementing loop reordering, although unrolling further does not bring much improvement. We compared the performance using normalized speedup and MPKI.
Normalized speedup gives us a comparative analysis with respect to a baseline, and MPKI gives a clearer picture of the role of cache's influence in improving performance.



Task 1B: Divide Karo, Rule Karo

1. ***Profiling baseline matrix multiplication code:***

   - *Find out the size of the **L1 Data (L1-D) cache** on your system.*
     - L1d: 288 KiB (6 instances), So each instance is **48KiB**

   - *Profile the baseline (naive) matrix multiplication implementation and report the **L1-D cache misses per kilo instructions (MPKI)**.*

Ans:

Matrix Size	Instructions	Cache Misses	MPKI
8	3833787	44191	11.526
16	4067005	43671	10.737
32	5549175	43422	7.824

64	16892871	50376	2.982
128	104700466	2556486	24.417
256	793255428	19536210	24.627
512	6249789303	135257770	21.641
1024	49771038235	1081762047	21.734
2048	397422544852	10298849175	25.914

2. ***Implementing Tiled matrix multiplication***

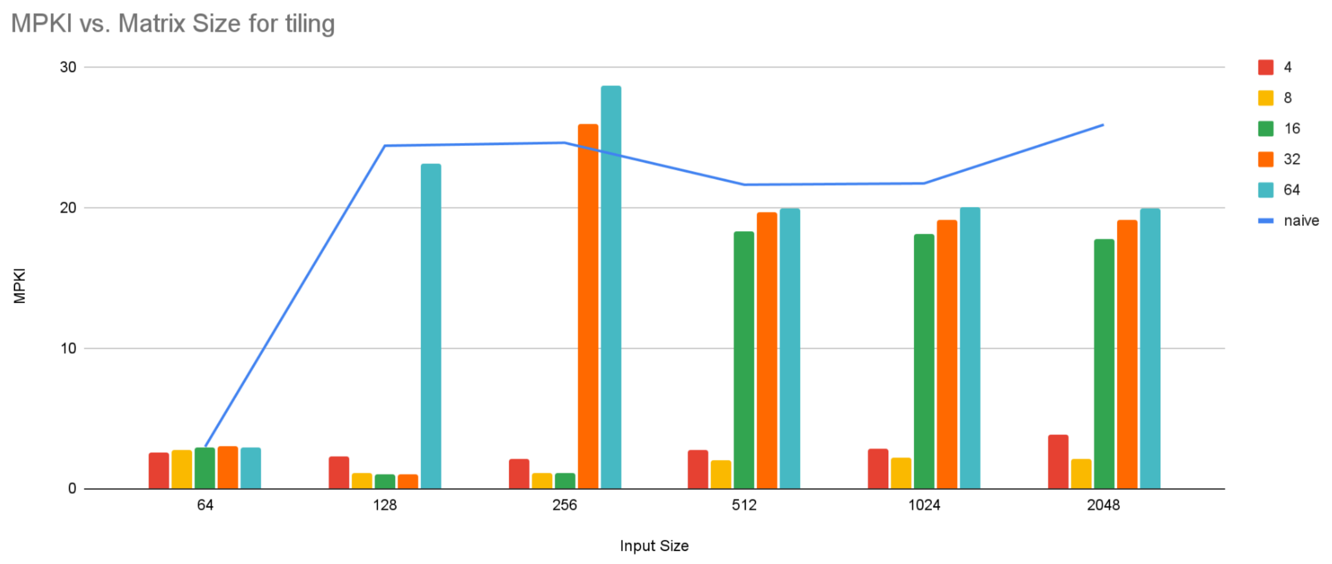
- *Identify the* ***tile size*** *that gives the best performance on your hardware in terms of cache efficiency and execution time.*

Ans: In terms of execution time, a tile size of 64 gave the best speedup across all matrix sizes, although there was an inconsistent trend in respective MPKIs.

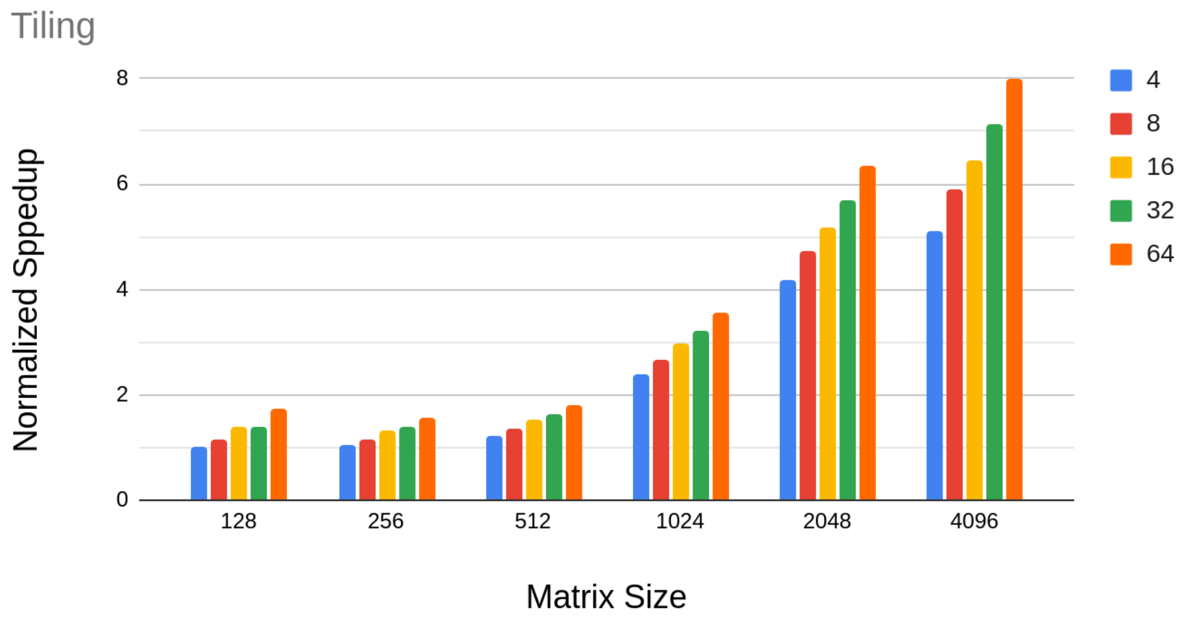
3. ***Collecting and analyzing different metrics of interest:***

- *Test your tiled implementation on matrices of various sizes.*
- *Calculate the* ***speedup*** *achieved by the tiled version compared to the baseline implementation for each matrix size.*
- *Create plots to visualize the performance trends:*

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



- **Speedup vs. Matrix Size** for different tile sizes.



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

Ans: There was a decrease in MPKI in most cases when moving from naive to tiled matrix multiplication. This is because tiling brings a more cache friendly access pattern, resulting in more hits.

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

Ans: MPKI significantly decreased with smaller tile sizes, showing a drastic change between tile sizes 8 and 16. This suggests that till tile size 8, optimal data reuse occurs. However, increasing the tile size to 16 likely necessitates accessing newer blocks in each interaction, leading to more misses. Also, for smaller tile sizes the total number of instructions increases further decreasing the MPKI.

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

Ans: There was a speedup observed that increased with increasing tile matrix sizes and respective tile sizes. This is because tiling makes access patterns high in spatial and temporal locality, thus using the cache more efficiently.

Task 1C: Data Ko Line Mein Lagao

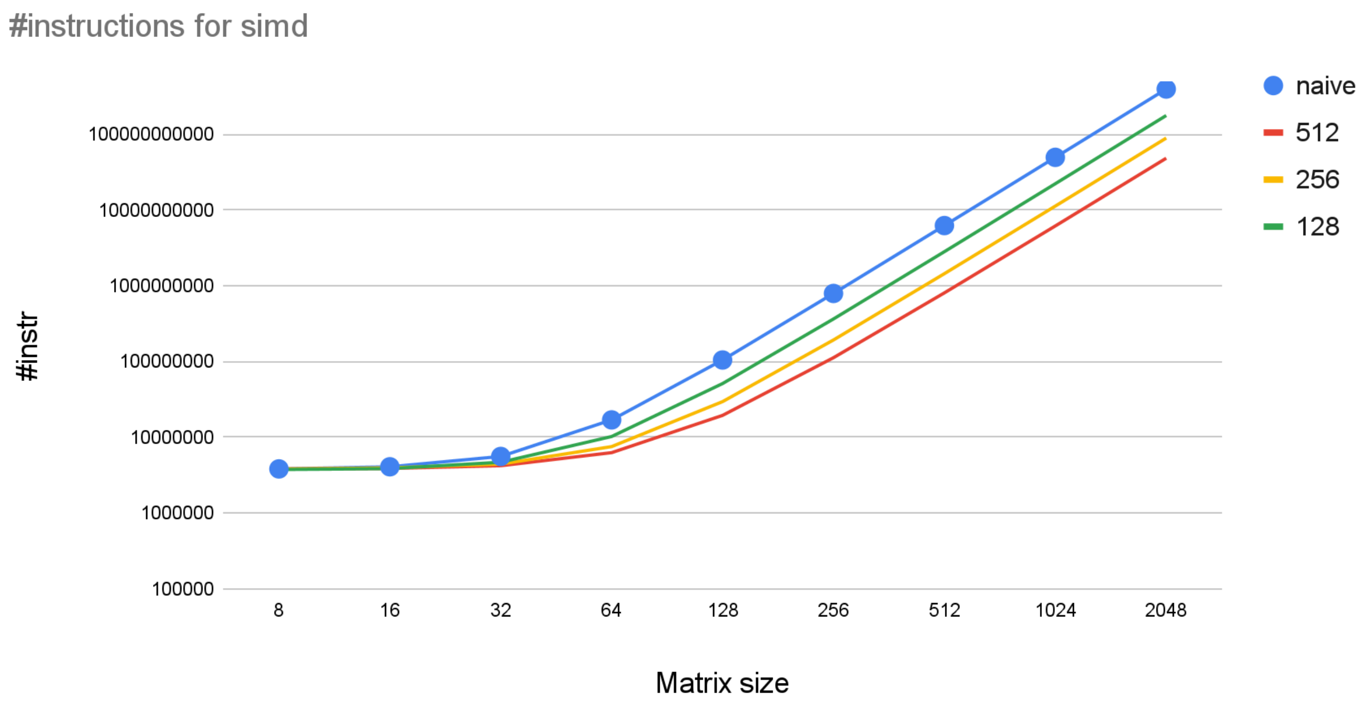
1. ***Baseline Profiling:***

   - Report the **number of instructions** executed for the **naive** matrix multiplication implementation.

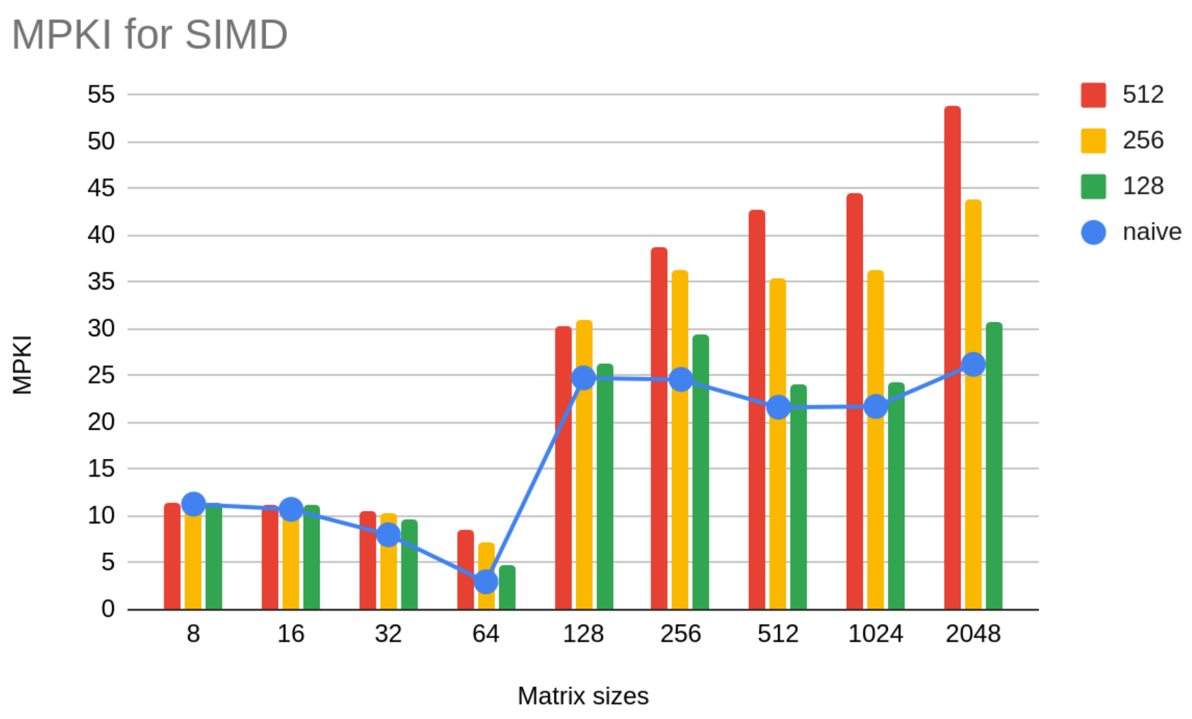
Matrix Size	Instructions	Cache Misses	MPKI naive
8	**3816535**	43055	11.281
16	**4071676**	43762	10.747
32	**5578019**	44770	8.026
64	**16950240**	50619	2.986
128	**104787479**	2596593	24.779
256	**793203708**	19522420	24.612
512	**6250991002**	135309646	21.646
1024	**49757209326**	1081662507	21.738
2048	**397373533397**	10429257092	26.245

Performance Analysis:

1. *Report the change in the number of instructions you observed when moving from the naive to the SIMD implementation. Justify your observations.*

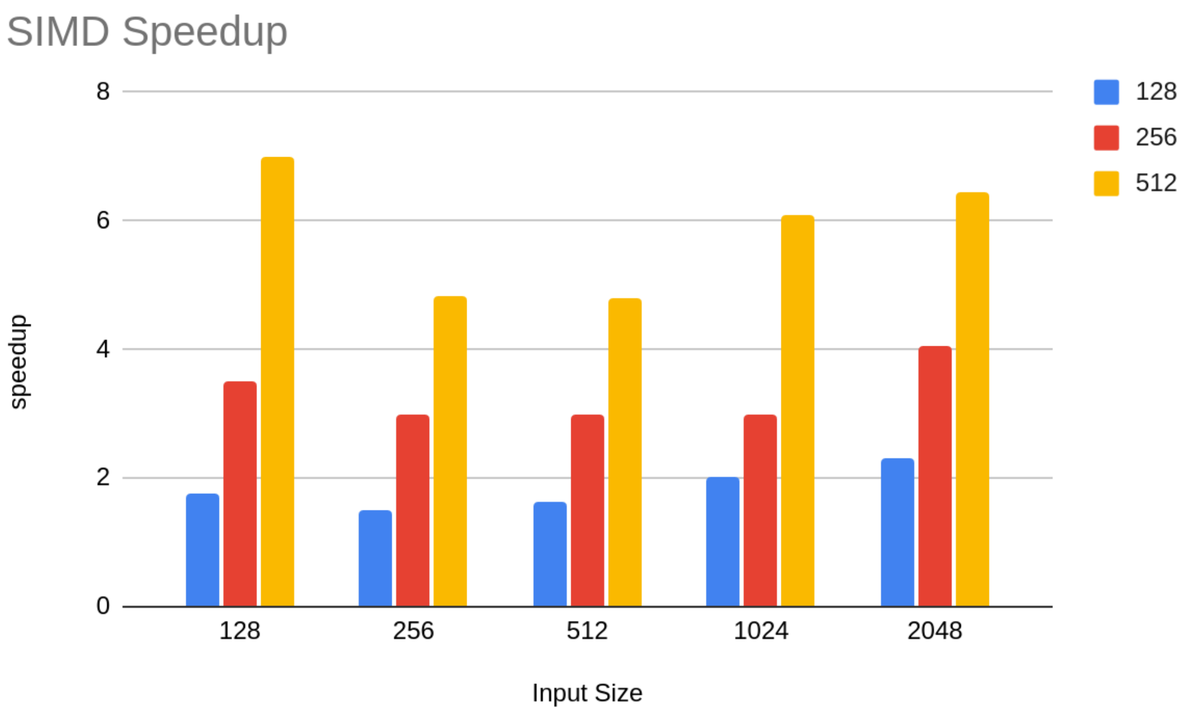


As SIMD width increases, the number of instructions decreases. This is an anticipated outcome, as SIMD allows for the execution of operations involving multiple data points within a single instruction.



Analysis of cache efficiency using MPKI reveals an unexpected trend: the MPKI in the SIMD implementation is significantly higher than in the naive implementation. This happens because the reduction in the number of instructions is proportionally greater than the reduction in cache misses, leading to a higher ratio.

2. *Did you achieve any speedup? If so, how much, and what contributed to it? If not, what were the reasons?*

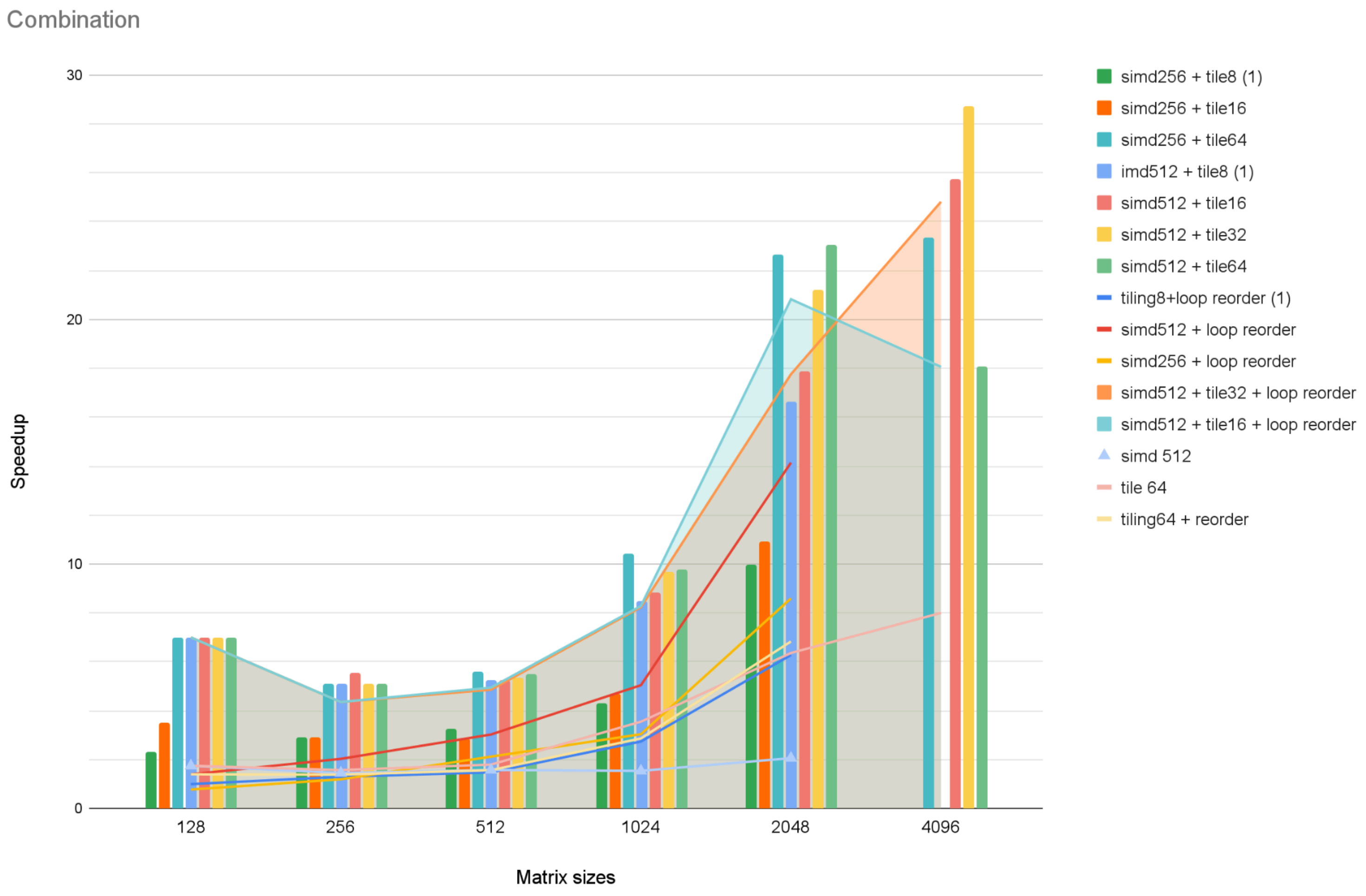


There was a significant speedup that increased with SIMD width. This speedup is the result of parallelization we achieve after performing computation on multiple data points simultaneously.

3. *Which SIMD intrinsics did you use? Justify your choice of functions.*

Ans: We used `mmxxx_loadu_pd`, `_mmxxx_storeu_pd`, `_mmxxx_fmadd_pd` for load, store and MAC operations. The `_mmxxx_fmadd_pd` function was the most efficient for execution, as it performs MAC operation with multiple data at the same time.

Task 1D: Rancho's Final Year Project



Observations:

- Simd+tiling combinations gave the best speedup. Tile size 64 had the best speedup with both simd widths. Simd width 512 had (in general) better performance than width 256.
- Combinations gave better performance than individual optimizations.
- When reordering was introduced in simd+tiling, speedup dropped slightly. It could be because of increased access to the result matrix in the inner loops since we need to compute partial sum in reordered implementation.