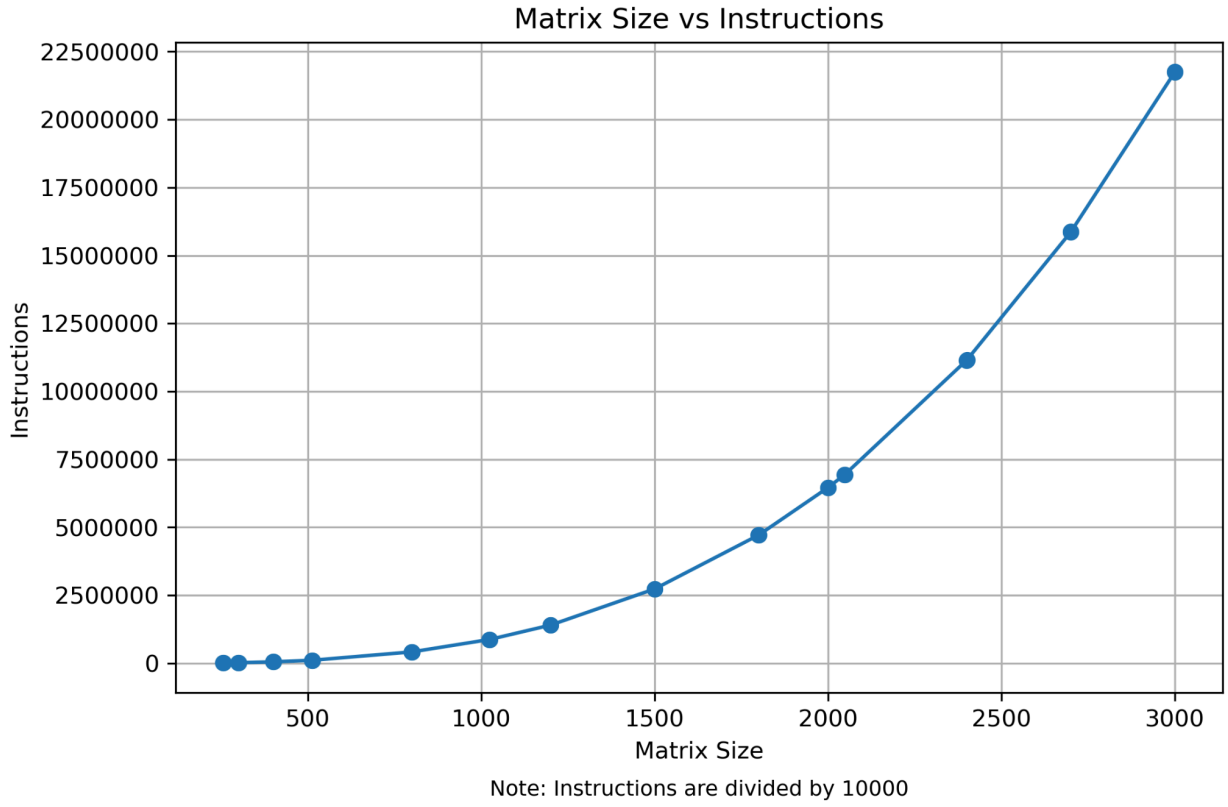
Task 1C :

1. *Baseline Profiling:*

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



MatrixSize	Instructions
256	146477227
257	148132159
300	231880051
301	234136200

400	538238994
401	542207136
512	1115095669
513	1121551044
800	4193261317
801	4208877400
1024	8747646703
1025	8773139705
1200	14039623576
1201	14074567617
1500	27335462006
1501	27389942592
1800	47137929882
1801	47216268413
2000	64594368882
2001	64691010939
2048	69342609260
2049	69443912788
2400	111446757000
2401	111585765285
2700	158545109185
2701	158720938755
3000	217334094045
3001	217551061153

2. ***SIMD Implementation:***

   - Modify your matrix multiplication to leverage **SIMD instructions**.
   - Implement versions using **128-bit**, **256-bit**, and (if available) **512-bit** SIMD registers.

   Done, check the uploaded source code

3. ***Instruction Count and Performance Analysis:***

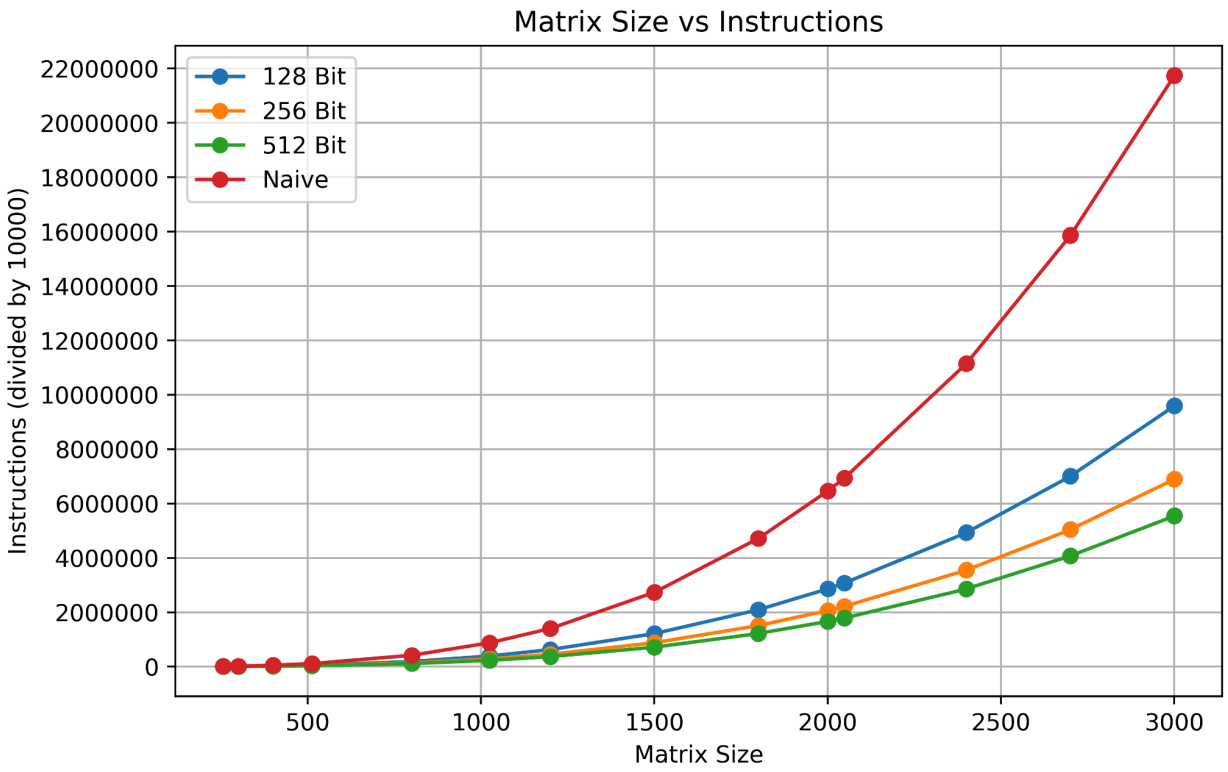
   - Report the **number of instructions** executed when running only the **SIMD version**.
   - **Compare performance** between the naive and SIMD implementations by calculating the **speedup** achieved.

4. ***Multi-Size Evaluation:***

   - Run your implementations on matrices of various sizes.
   - Analyze how performance and speedup vary with **matrix size** and **SIMD register width**.

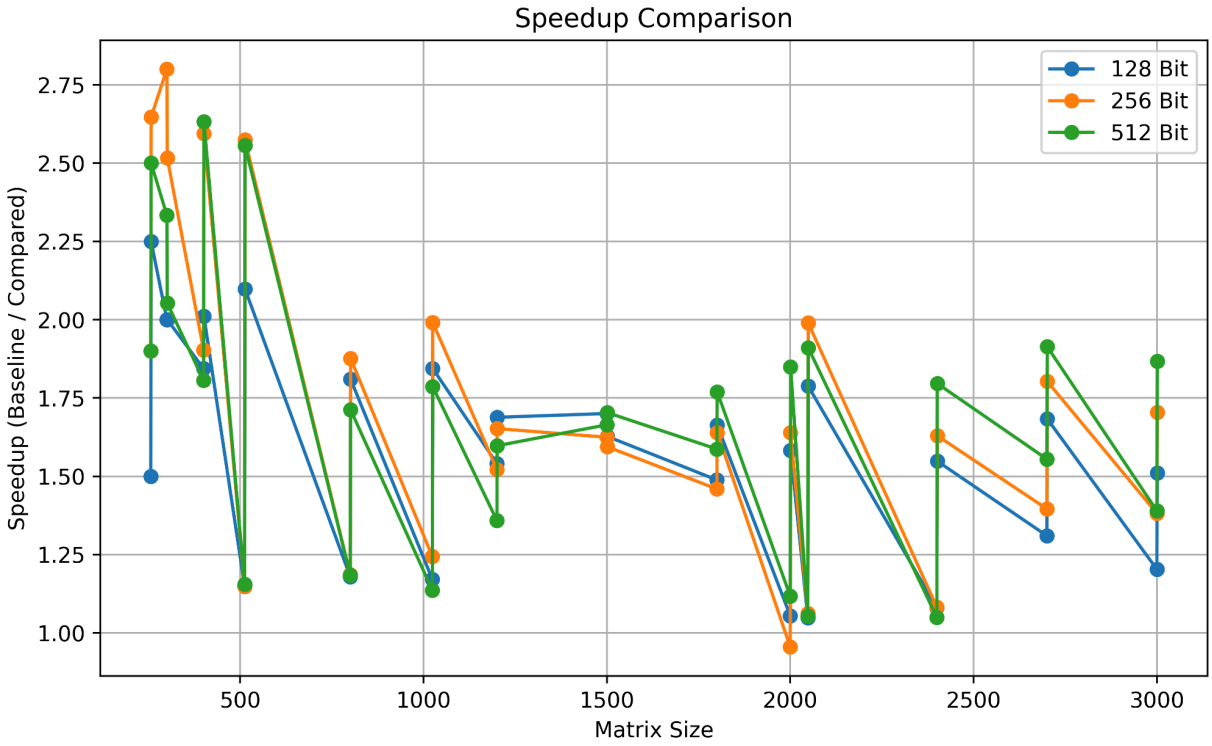
5. ***Visualization:***

   - Create plots showing the **speedup** for different matrix sizes and SIMD widths.
   - Choose matrix sizes that allow you to **clearly observe trends and draw meaningful conclusions**.



MatrixSize	Instructions (Naive)	Instructions (128 Bit SIMD)	Instructions (256 Bit SIMD)	Instructions (512 Bit SIMD)
256	146477227	71,220,232	54,770,964	46,384,418
257	148132159	72,651,845	56,139,604	47,753,711
300	231880051	110,666,721	84,117,023	73,228,245
301	234136200	112,655,768	86,019,081	74,334,282
400	538238994	250,740,123	187,540,531	155,543,748
401	542207136	254,232,110	190,876,525	158,880,748
512	1115095669	511,928,411	379,021,914	311,917,161
513	1121551044	517,602,235	384,438,729	317,335,268
800	4193261317	1,891,219,721	1,382,420,533	1,126,426,945
801	4208877400	1,904,923,583	1,395,492,395	1,139,500,487

1024	8747646703	3,919,002,218	2,850,504,268	2,313,641,819
1025	8773139705	3,941,359,388	2,871,823,134	2,334,963,018
1200	14039623576	6,267,998,728	4,547,199,866	3,683,209,344
1201	14074567617	6,298,634,202	4,576,407,597	3,712,419,729
1500	27335462006	12,154,779,111	8,791,030,659	7,168,786,654
1501	27389942592	12,202,523,845	8,836,540,349	7,194,116,067
1800	47137929882	20,903,728,686	15,087,930,073	12,171,945,174
1801	47216268413	20,972,362,254	15,153,342,270	12,237,360,847
2000	64594368882	28,606,456,022	20,626,458,276	16,626,473,815
2001	64691010939	28,691,115,289	20,707,137,196	16,707,157,840
2048	69342609260	30,700,574,188	22,131,612,293	17,836,661,565
2049	69443912788	30,789,313,158	22,216,178,350	17,921,232,356
2400	111446757000	49,256,135,873	35,460,937,861	28,548,957,159
2401	111585765285	49,377,888,303	35,576,952,880	28,664,977,763
2700	158545109185	69,993,591,165	50,347,042,229	40,716,962,242
2701	158720938755	70,147,571,806	50,493,760,584	40,798,196,503
3000	217334094045	95,861,216,873	68,906,216,356	55,406,240,325
3001	217551061153	96,051,207,776	69,087,238,024	55,587,268,566



MatrixSize	Speedup (128 Bit SIMD)	Speedup (256 Bit SIMD)	Speedup s (512 Bit SIMD)
256	1.297	1.47	1.455
257	1.75	1.975	1.923
300	1.755	2.248	1.867
301	1.702	2.068	1.633
400	1.694	1.725	1.588
401	1.874	2.126	2.174
512	1.139	1.141	1.132
513	1.897	2.142	2.154
800	1.172	1.177	1.17

801	1.727	1.778	1.634
1024	1.168	1.239	1.133
1025	1.788	1.928	1.741
1200	1.528	1.503	1.347
1201	1.667	1.633	1.577
1500	1.682	1.608	1.647
1501	1.616	1.584	1.69
1800	1.476	1.45	1.574
1801	1.654	1.631	1.759
2000	1.054	0.955	1.116
2001	1.576	1.631	1.838
2048	1.048	1.062	1.053
2049	1.779	1.977	1.899
2400	1.079	1.08	1.048
2401	1.544	1.624	1.789
2700	1.307	1.391	1.548
2701	1.678	1.796	1.906
3000	1.201	1.377	1.385
3001	1.508	1.7	1.861

Questions:

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

Ans.

- When using 128 Bit registers, the no. of instructions is reduced to half of naive, which is as expected as 128 bit registers can store 2 doubles at a time. So we can perform 2 multiplication operations in 1 instruction.
    - On Average, the no. of instructions in 128 Bit implementation were ***~2.2*** *times* less compared to Naive implementation
- Similar case goes for
    - 256 Bit register, Stores 4 doubles.
      Avg : ***~3.02*** *times* less inst. than Naive impl.
    - 512 Bit register, Stores 8 doubles.
      Avg : ***~3.7*** *times* less inst. than Naive impl.

The number of instructions executed in 256 bit & 512 Bit implementation should be way less compared to 128 bit but it's not

The no. of instructions executed also increases due to 2 operations

- Getting column's data
    - 128 bit : _mm128_set_pd(arg1…. arg2) translates to ~2 instructions
    - 256 bit : _mm256_set_pd(arg1…. arg4) translates to ~4 instructions
    - 512 bit : _mm512_set_pd(arg1…. arg8) translates to ~8 instructions
- Summing the result after multiplication
    - 128 bit : `sum = C_r[0] + C_r[1];`
    - 256 bit : `sum = C_r[0] + C_r[1] + C_r[2] + C_r[3];`
        - +2 instruction more compared to 128 bit
    - 512 bit : `sum = _mm512_reduce_add_pd(C_r);`
        - -1 instruction less compared to 128 bit

Conclusion:

- 256 bit does ~4 instructions more compared to 128 bit

- 512 bit does ~5 instructions more compared to 128 bit
- Since we are using the -O2 optimization flag during compilation , it might have also optimized no. of inst. Executed in Naive Impl. which is why you don't see ~4x , ~8x difference in no. of inst. Executed in 256 bit, 512 bit Impl. respectively.

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

Ans.

- Yes, the Average speedup achieved in
  - 128 Bit: **1.513**x
  - 256 Bit: **1.608**x
  - 512 Bit: **1.594**x

- The things which contributed to speedup was
  - SIMD intrinsic: _mm_fmadd_pd(A, B, C)
    - It parallely multiplies values in register A & B and adds the result from register C
  - SIMD intrinsic: _mm_loadu_pd(addr)
    - It tries to load the row values parallely from the memory to directly in the SIMD register. It takes advantage of Memory Level Parallelism (MLP)

- The things which hindered speedup was
  - SIMD intrinsic: _mm_set_pd(arg1… argX)
    - Loading column values of matrix B into the register. Since memory is stored row wise, accessing column wise results in more number of cache misses, hence taking more time
  - Not using Cache friendly Loop ordering

        - If we use Loop Re-ordering optimization, we can improve the execution speed by a lot.

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

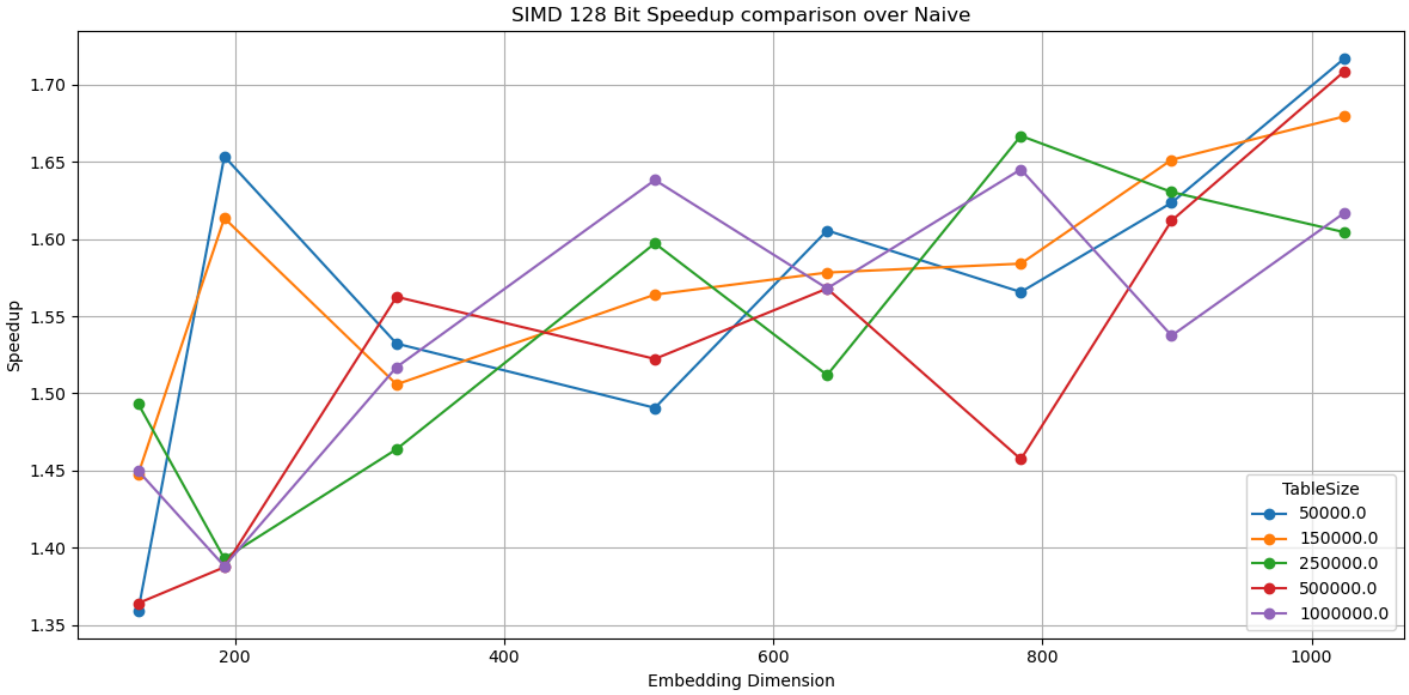
Ans.

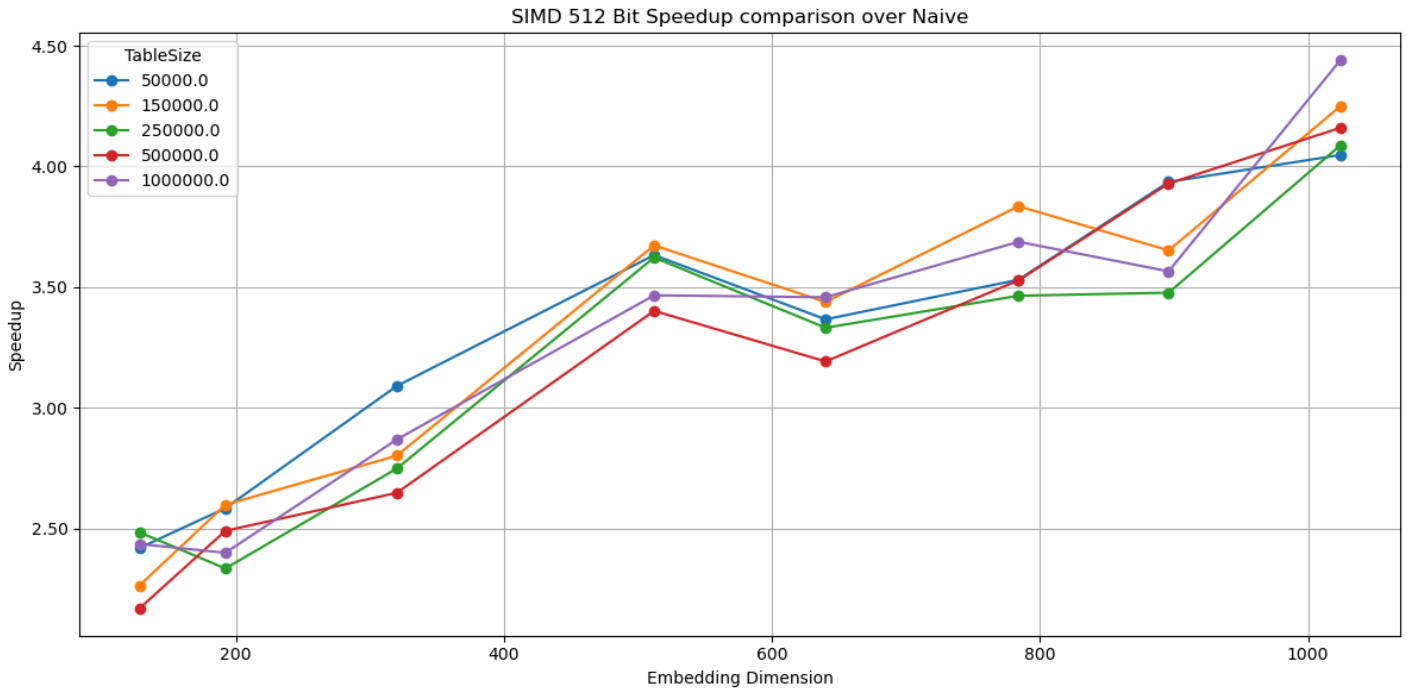
- **_mm_loadu_pd()**
    - _mm256_loadu_pd(), _mm512_loadu_pd()
    - To load the row values from matrix A into the SIMD register. Since the matrix is stored row wise. It makes sense to use this because it results in better performance than **_mm_set_pd()**
- **_mm_set_pd()**
    - _mm256_set_pd(), _mm512_set_pd()
    - It sets the values in the SIMD register. We are using this to store the column values of the matrix. We cannot use _mm_loadu_pd() because memory is stored row-wise rather than column-wise.
- **_mm_fmadd_pd(A, B, C)**
    - _mm256_fmadd_pd(), _mm512_fmadd_pd()
    - It is a FMA (Fused Multiply Add) instruction. It provides better performance compared as it multiplies & adds in **single instruction** compared to using 2 seperate instruction _mm_mul_pd(A,B) & summing the values of result
- **_mm512_reduce_add_pd(C)**
    - It performs a **horizontal addition** on a 512-bit register. It provides better performance as it performs addition using **single instruction** compared to doing
      sum = C[0] + C[1] + C[2] + C[3] + C[4] + C[5] + C[6] + C[7]

Task 2B:

1. *Analyze the Impact of SIMD Instruction Width*:

Experiment with different SIMD instruction widths provided by Intel intrinsics (e.g., 64-bit, 128-bit, 256-bit), depending on your machine's capabilities, and observe how each affects performance.



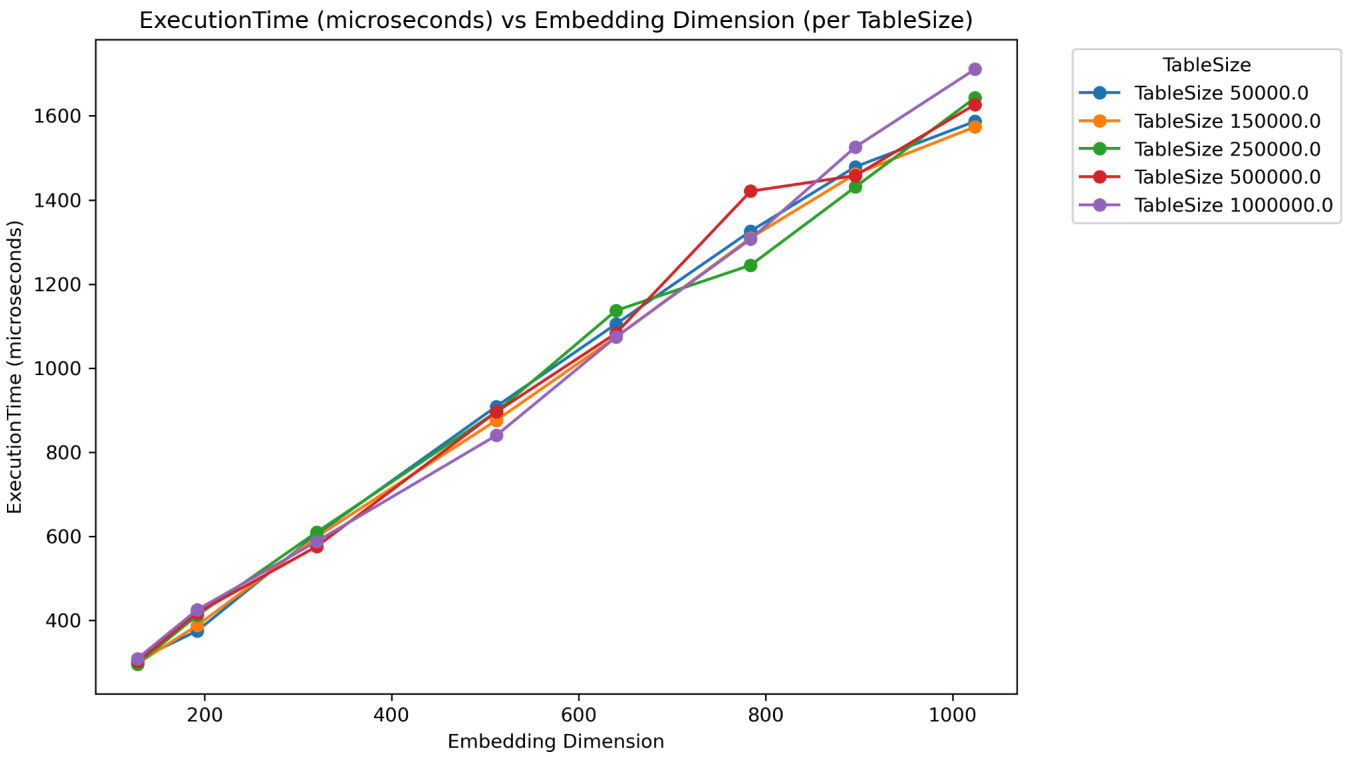


- For the analysis part of speedup, check out the answer from "*What trends do you observe in speedup for different combinations of embedding dimensions and SIMD widths?*"

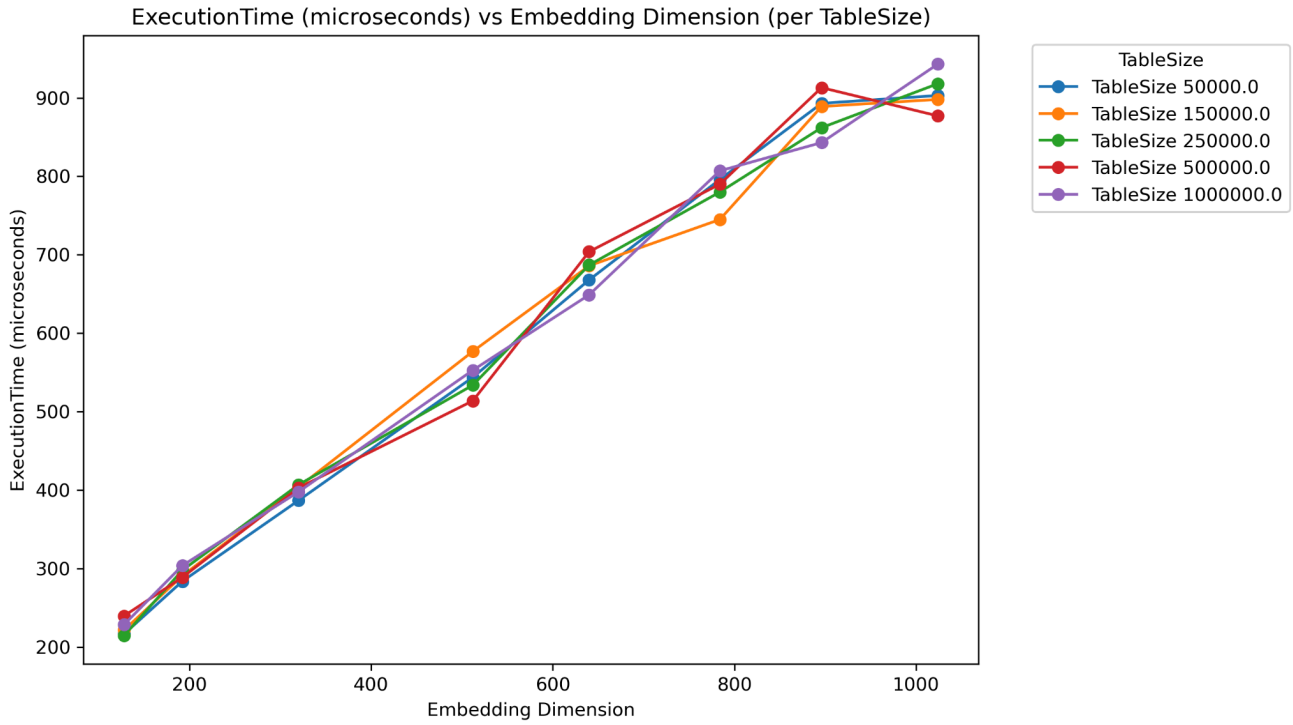
2. *Analyze the Impact of Embedding Dimension*:

Try different embedding dimensions (i.e., number of elements per row) in combination with different SIMD widths.

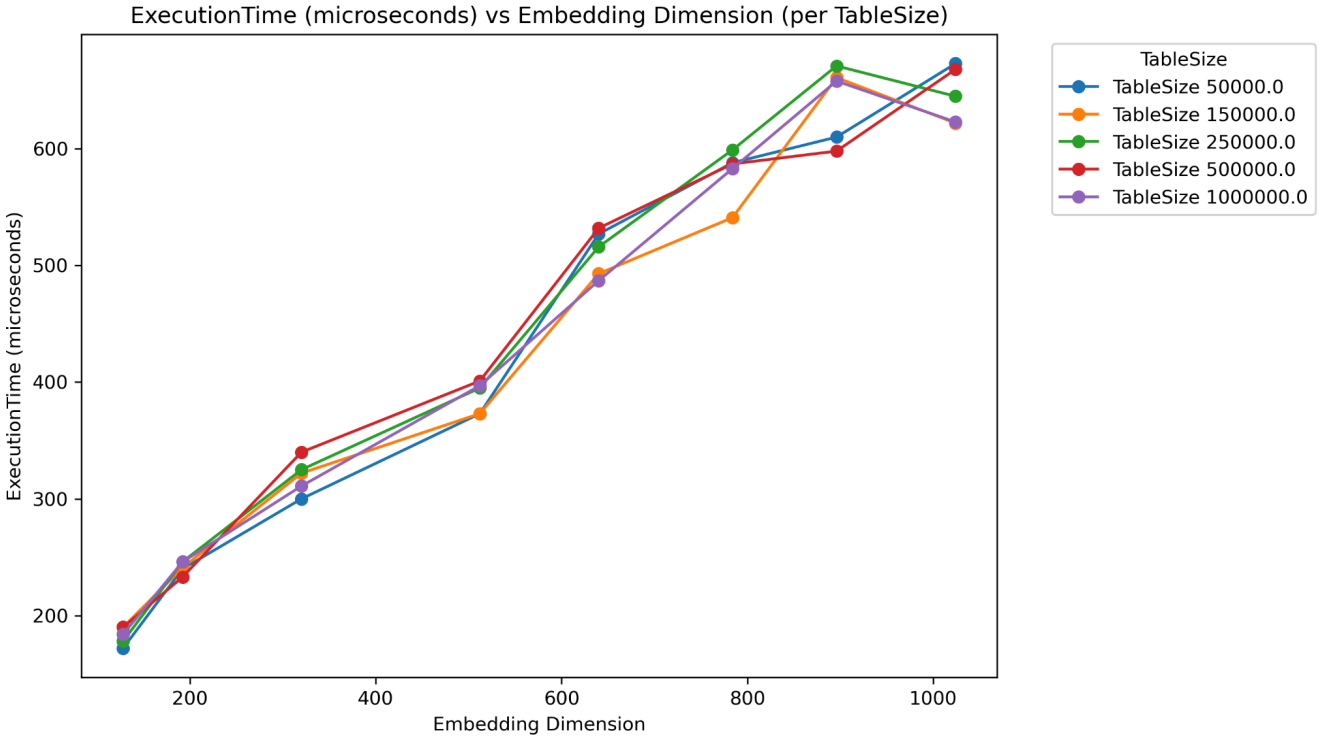
SIMD 128



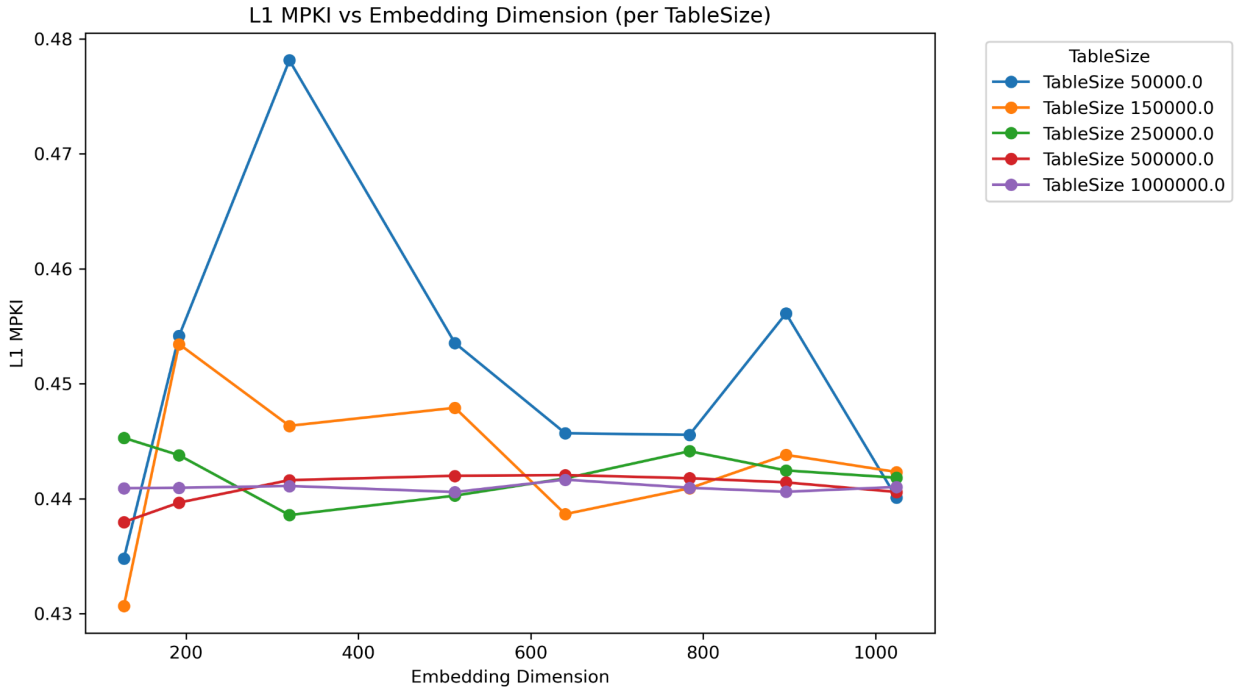
SIMD 256



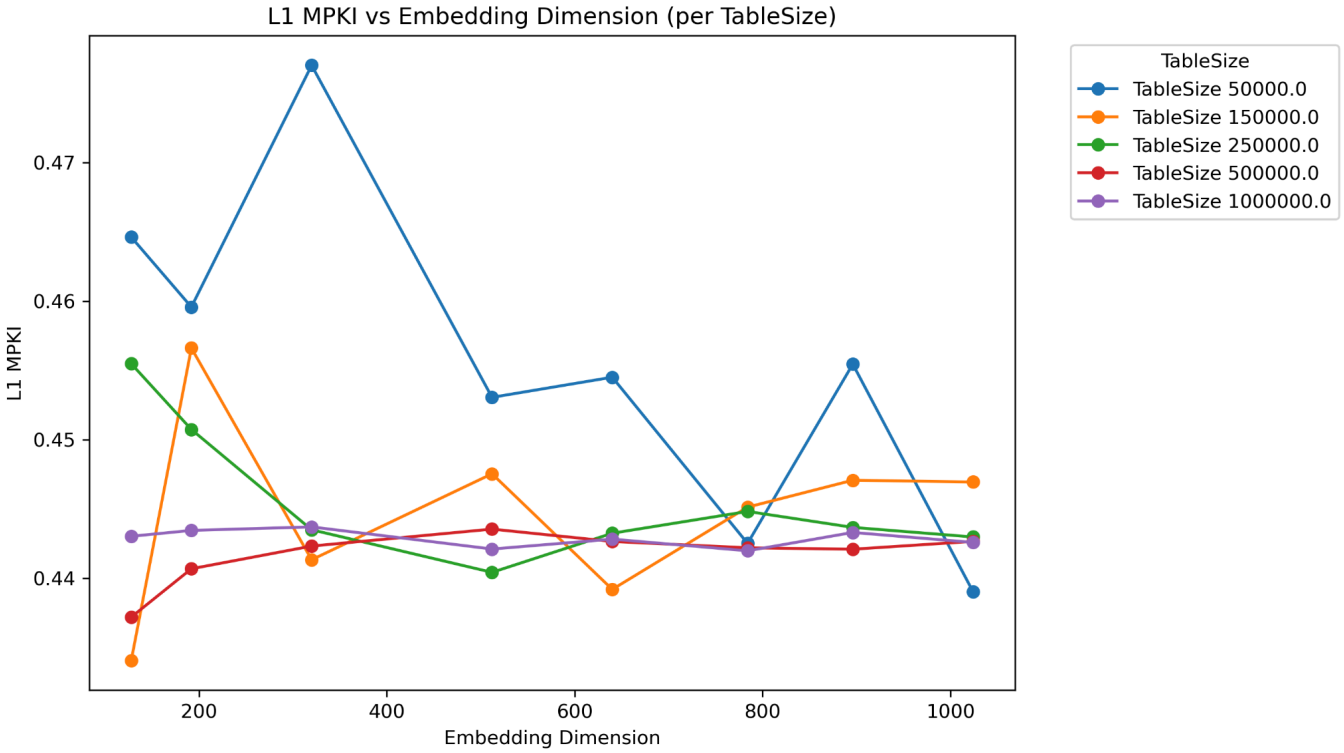
SIMD 512



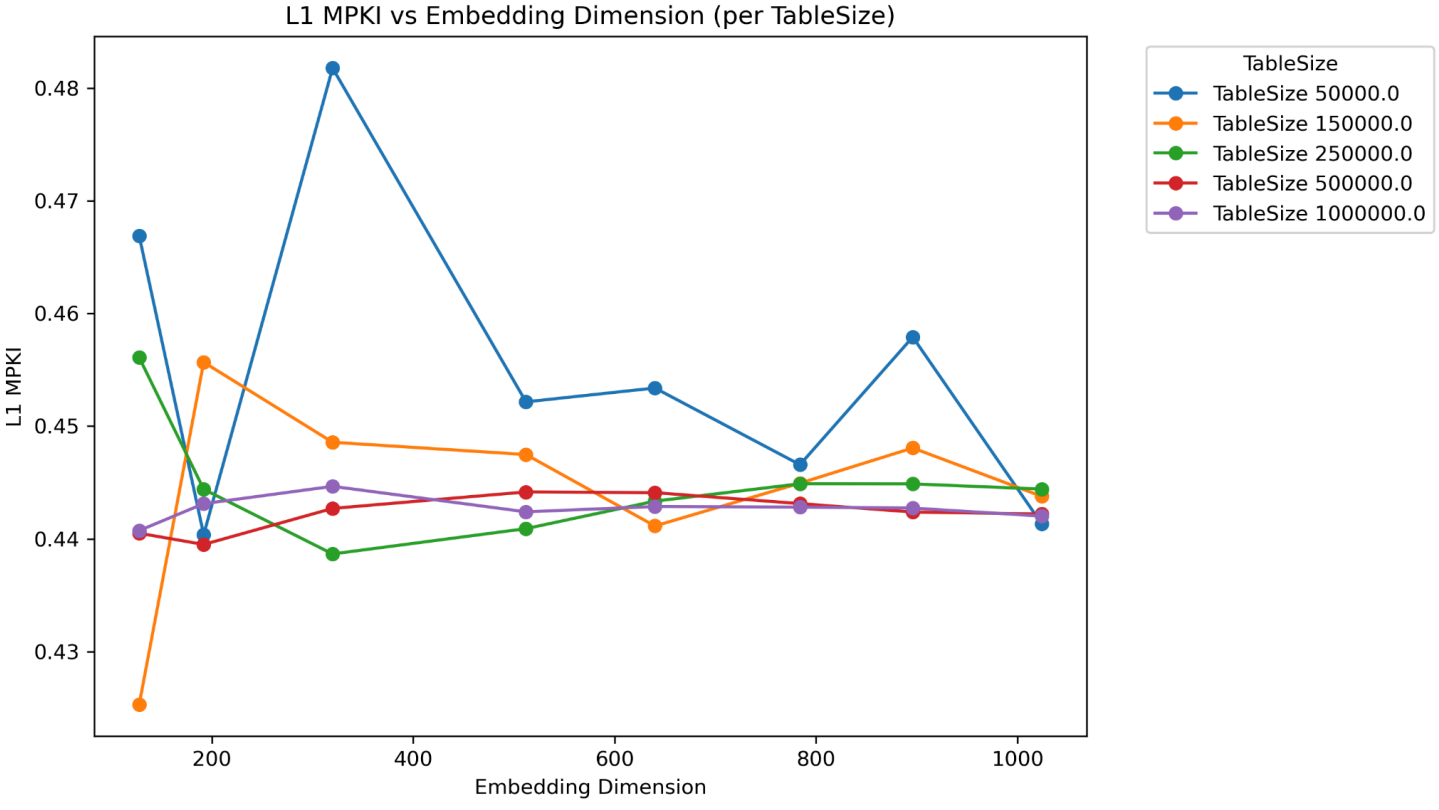
SIMD 128



SIMD 256

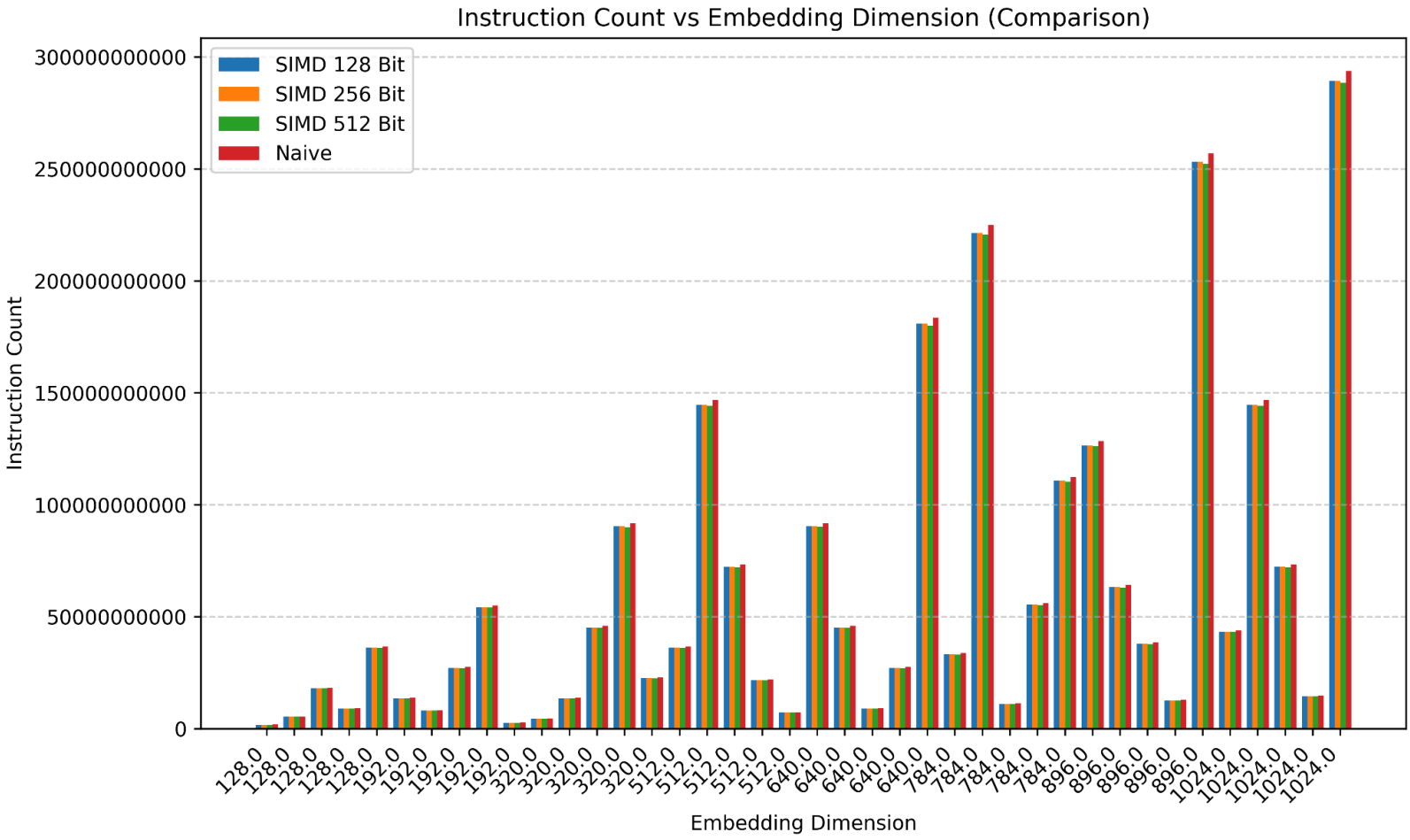


SIMD 512



3. *Analyze Instruction Count*:

Use the perf tool (or any suitable performance analysis tool) to measure the instruction count for different SIMD widths and embedding dimensions.



Note: On X axis, entries are for different table sizes

- The instruction count in SIMD implementations is almost as similar to Naive. It's because there are 3 loops nested (i, j , d). The SIMD implementation only reduces the inner most summation loop (d) but the outer loop i & j are unaffected by it. Due to this we don't see much improvement in the number of instructions affected.

4. *Collect Execution Time and Compute Speedup*:

- Compute the speedup as follows:

$$Speedup = \frac{Execution\ Time\ without\ SIMD}{Execution\ Time\ with\ SIMD}$$

- Analyze how the speedup varies with different embedding table sizes.
    - Refer to "*Analyze the Impact of SIMD Instruction Width*" Section

Table Size : 1000000

SIMD 128 Bits:

Metrics		**embedding dimension (elements)**							
		128	192	320	512	640	784	896	1024
No SIMD	Instructions	36778927002	55099616196	91815466941	146901319090	1836167360 04	224967476966	2570413360 88	293773428179
	Execution time	448	590	892	1376	1684	2150	2346	2766
SIMD	Instructions	36222188093	5427374114 1	9043271725 5	144692298027	1808289412 76	221490544062	2531816904 96	289399276659
	Execution time	309	425	588	840	1074	1307	1526	1711
Speedup		1.449	1.388	1.517	1.638	1.567	1.644	1.537	1.616

SIMD 256 Bits:

Metrics		embedding dimension (elements)							
		128	192	320	512	640	784	896	1024
No SIMD	Instructions	36778927002	55099616196	91815466941	146901319090	1836167360 04	224967476966	2570413360 88	293773428179
	Execution time	448	590	892	1376	1684	2150	2346	2766
SIMD	Instructions	36060311126	5407459464 1	9008478964 5	144196198875	1802135497 81	220790694835	2522427409 60	288360348525
	Execution time	229	304	398	553	649	807	843	943
Speedup		1.956	1.940	2.241	2.488	2.594	2.664	2.782	2.933

SIMD 512 Bits:

Metrics		embedding dimension (elements)							
		128	192	320	512	640	784	896	1024
No SIMD	Instructions	36778927002	55099616196	91815466941	146901319090	1836167360 04	224967476966	2570413360 88	293773428179
	Execution time	448	590	892	1376	1684	2150	2346	2766
SIMD	Instructions	36068233173	5408656055 4	9008539722 0	144201158053	1801728844 41	220714423852	2522608261 21	288403146558
	Execution time	184	246	311	397	487	583	658	623
Speedup		2.434	2.398	2.868	3.465	3.457	3.687	3.565	4.439

Questions

1. What trends do you observe in speedup for different combinations of embedding dimensions and SIMD widths?

Ans.

- In all SIMD widths, there's a clear trend that the speedup increases as we increase the embedding dimension. This is happening because we have applied the SIMD optimization to the summation loop. The no. of iterations of that loop depends on the SIMD register width.
    - So, SIMD width is directly proportional to the speedup

2. For which SIMD width do you achieve the maximum speedup?

Ans

- 512 Bit