Programming Assignment 1
THREE GENIUSES

CS683: Advanced Computer Architecture, Autumn 2025
Computer Science and Engineering
Indian Institute of Technology, Bombay
CASPER group: https://casper-iitb.github.io/

* *Disclaimer: The memes included in this document are intended solely for fun learning. They are not meant to offend, mislead, or be taken as factual information. Please enjoy them in the spirit of fun learning.*

Most programs that a programmer writes are driven primarily by functionality. When it comes to optimizing performance, they often experiment with various algorithms in an effort to achieve the best possible execution time. However, the underlying hardware that runs the code is frequently overlooked, leaving a significant portion of potential optimizations untapped. To truly optimize performance, a programmer must understand how the software interacts with the hardware. Only by gaining insight into both the code and the architecture can they meaningfully improve the performance of their programs.

This assignment aims to give you a taste of what happens when you optimize code with the underlying architecture in mind. In particular, we will explore how to make better use of the cache by improving **data reuse**.

NOTE:

Just a friendly reminder: if you think copying code is a clever shortcut, think again. It's not only easily spotted but also a great way to miss out on the chance to actually learn something. Why not impress us with your own work?

- You need to have Intel-based x86 machines to implement software prefetching.
- Make changes to the base code only. Do not use a different implementation of the base code.
- No compiler or any other optimizations should be used. Points will be deducted for using any additional optimization techniques other than the ones mentioned in the assignment.

The assignment is divided into two tasks, each having its subparts. The task structure and its respective points are shown below:

Main Tasks		
Task 1	The Matrix	7 points
Task 2	Embed it	8 points
Total		15 points

Task 1

Matrix Multiplication — The Rancho Way

This assignment focuses on optimizing the performance of matrix multiplication using various optimization techniques. You will implement and analyze multiple optimisation methods, measure their impact on performance, and gain a deeper understanding of the underlying principles that drive performance improvements.

Finally, applying the knowledge gained, you will optimize a neural network whose performance is significantly influenced by matrix multiplication.

Note: Please refrain from modifying any part of the code other than the matrix multiplication function.

List of Tasks (7 points)		
1A	Unroll Baba Unroll	1 point
1B	Divide Karo, Rule Karo	2 points
1C	Data Ko Line Mein Lagao	2 points
1D	Rancho's Final Year Project	1 point
1E	Confusion hi confusion hai !!	1 point

Task 1A: Unroll Baba Unroll

Analyze the provided matrix multiplication code and implement loop reordering and unrolling optimizations to improve execution time.

By the end of this task, you should be able to answer the following questions:

1. What motivated you to make changes to your code?
2. What considerations did you take into account when implementing those changes?
3. How effective are your changes? (Which metric will you use to quantify effectiveness and why?)

Task 1B: Divide Karo, Rule Karo

Deliverables

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

   - Find out the size of the **L1 Data (L1-D) cache** on your system.
   - Profile the baseline (naive) matrix multiplication implementation and report the **L1-D cache misses per kilo instructions (MPKI)**. *Hint: use a tool called perf*

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

   - Implement a **tiled** version of matrix multiplication.
   - Experiment with different **tile sizes**.

- For each tile size, measure the **L1-D cache MPKI** to evaluate cache behavior.
- Identify the **tile size** that gives the best performance on your hardware in terms of cache efficiency and execution time.

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.

Answer the following questions:

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

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.
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.

Task 1C: Data Ko Line Mein Lagao

In this task, you will utilize SIMD (Single Instruction, Multiple Data) instructions to parallelize matrix multiplication and analyze their impact on instruction count and overall performance.

Deliverables

1. ***Baseline Profiling:***
   - Report the **number of instructions** executed for the **naive** matrix multiplication implementation.
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.
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**.

Answer the following:

1. Report the change in the number of instructions you observed when moving from the naive to the SIMD implementation. Justify your observations.
2. Did you achieve any speedup? If so, how much, and what contributed to it? If not, what were the reasons?
3. Which SIMD intrinsics did you use? Justify your choice of functions.

Task 1D: Rancho's Final Year Project

In this part of the assignment, you are expected to demonstrate how combining multiple optimization techniques, such as tiling, SIMD vectorization, loop unrolling, and cache-aware programming, can lead to greater performance improvements than applying each technique in isolation.

Final performance summary for Matrix Multiplication:

Implement a version of matrix multiplication that combines two or more of the previously explored optimization techniques.

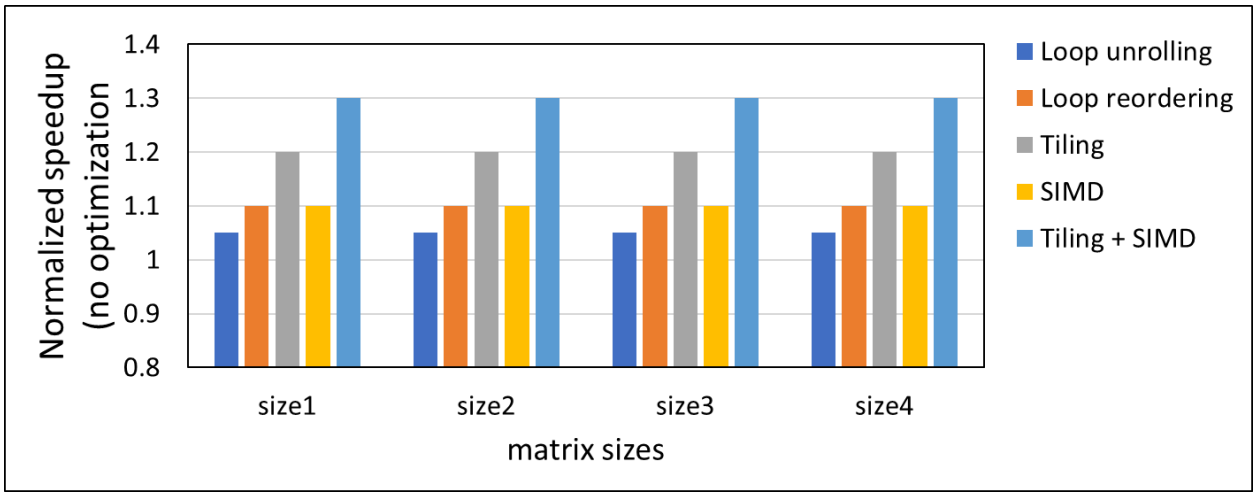
Demonstrate how these techniques complement each other to provide synergistic performance gains, i.e., the combined benefit is greater than the sum of individual optimizations.

Include a plot that compares the best performance achieved by each optimization technique —

- Loop unrolling
- Loop reordering
- Tiling
- SIMD
- Tiling + SIMD

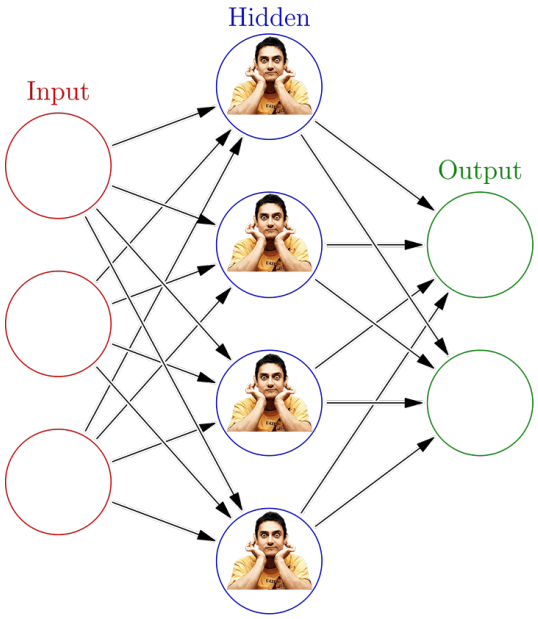
Plot this comparison against varying matrix sizes. Refer to **Figure 1.1** for reference.

Figure 1.1:



Note: This is an example plot, so we have only shown five optimization techniques. You can try different combinations and expand the plot accordingly.

Task 1E: Confusion hi confusion hai !!



You are provided with a custom implementation of a neural network written in C++. As discussed in the background section, neural networks rely heavily on matrix multiplication for their computations.

In this task, you are required to optimize the performance of the neural network by applying the matrix multiplication techniques you have implemented earlier (e.g., tiling, SIMD, etc.). You are allowed to modify only the matrix multiplication and transpose functions.

Your goal is to incorporate one or more optimization techniques and evaluate their effect on the execution time of the neural network. Clearly report the performance improvements observed, and support your findings with appropriate measurements and analysis.

Task 2

Embed it

Embedding Operation is a technique used to convert words, items, or categories into meaningful numeric vectors and is widely applied in areas like NLP, recommendation systems, and search engines.

In this section, you will optimize the embedding operation using optimization techniques such as software prefetching and SIMD.

List of Tasks (8 points)		
2A	Software prefetching	5 points
2B	SIMD	1 points
2C	Software prefetching + SIMD	2 points

Task 2A: Software prefetching

In this part of the assignment, you will be optimizing the embedding operation code using the software prefetching technique. You will be tuning various parameters of software prefetching for the embedding operation by analyzing trends in different CPU performance metrics. Additionally, you will study the impact of hardware prefetchers on the performance of the embedding operation.

You will be tuning two key parameters of software prefetching:

1. Prefetch Distance
2. Cache Fill Level

Both **_builtin_prefetch** and **_mm_prefetch** allow you to control these parameters during software prefetching.

Additionally, you will use the **sw_prefetch_access** event in *perf* to track the number of software prefetch requests issued during the execution.

Deliverables

1. *Analyze the Impact of Embedding Table Size*:

- Increase the size of the embedding table and observe how software prefetching performance changes.
- Optionally, you can also experiment with varying the sizes of the input array and offset array.

2. *Analyze CPU Metrics*:

For different embedding table sizes and different software prefetching parameters, analyze the trends in key CPU metrics (use the *perf* tool):

- L1D Cache Misses
- L2 Cache Misses
- LLC (Last Level Cache) Misses

3. *Collect Execution Time and Compute Speedup*:

- Measure the execution time of the embedding operation with and without software prefetching.
- Compute the speedup as follows:

$$Speedup = \frac{Execution\ Time\ without\ Software\ Prefetching}{Execution\ Time\ with\ Software\ Prefetching}$$

- Analyze how the speedup varies with different embedding table sizes and different software prefetching parameters.

4. *Identify Optimal Parameters*:

Determine the optimal prefetch distance and optimal cache fill level based on your observations.

5. *Study the Effect of Hardware Prefetchers*:

Enable and disable hardware prefetchers and analyze their impact on the embedding operation's performance.

Consider **Table 2.1** for reference. Identify the values of different metrics for different embedding table sizes, prefetching distances, and cache fill levels. Analyze them carefully and state your insights for the question below.

1. What trend do you observe in speedup with different embedding table sizes?
2. What is the best prefetch distance?
3. At what cache fill level do you achieve the maximum speedup

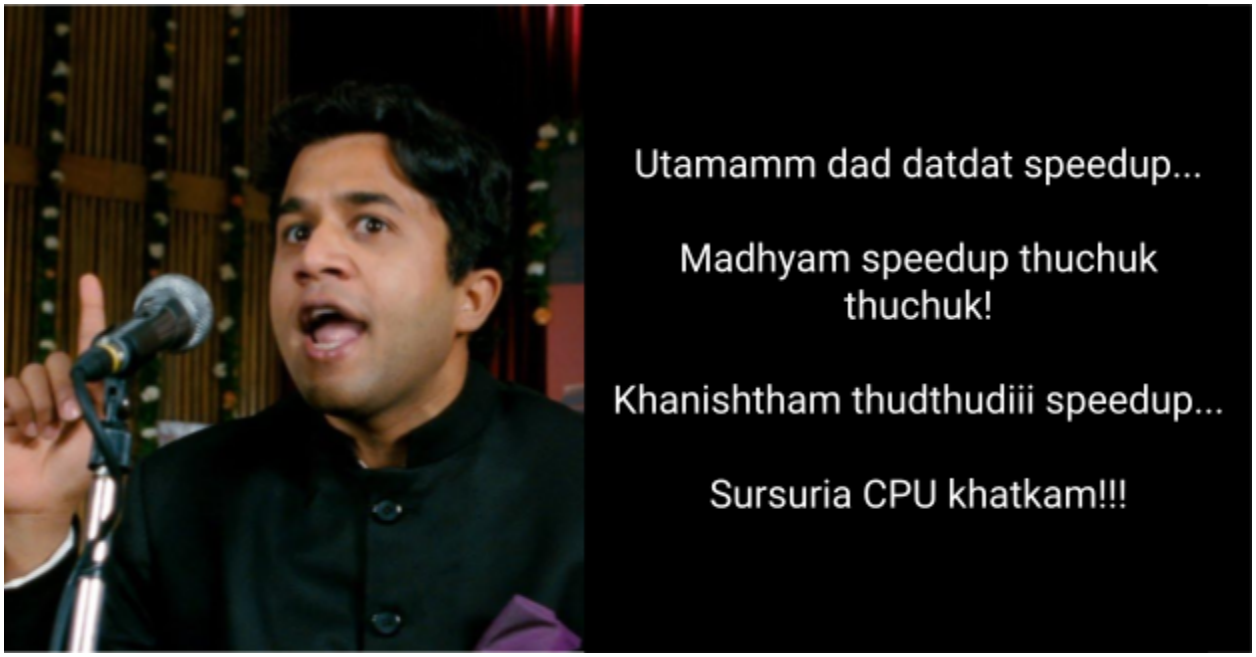


Table 2.1:

Metrics		embedding table size		prefetch distance		cache fill level		
		size1	size2	pd1	pd2	L1	L2	LLC
No software prefetching	L1D misses							
	L2 misses							
	LLC misses							
	Software prefetch requests							
	Execution time							
software prefetching	L1D misses							
	L2 misses							
	LLC misses							
	Software prefetch requests							
	Execution time							
Speedup (normalized to no software prefetching)								

Note: This is just an example table, so we have provided entries for only two embedding table sizes and two prefetch distances. You should expand and modify it as needed based on your analysis.

Task 2B: SIMD (Single Instructor Multiple Deadline)

Single *Instructor*, Multiple *Deadlines* (SIMD)

Utilize SIMD (Single *Instruction*, Multiple Data) instructions to parallelize the summation of elements within embedding rows, and analyze their impact on both instruction count and overall performance.

Deliverables

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.

2. *Analyze the Impact of Embedding Dimension*:

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

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.

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 and different software prefetching parameters.

Consider **Table 2.2** for reference. Identify the values of different metrics for different embedding dimensions and SIMD widths. Analyze them carefully and state your insights for the question below.

1. What trends do you observe in speedup for different combinations of embedding dimensions and SIMD widths?
2. For which SIMD width do you achieve the maximum speedup?

Table 2.2:

Metrics		**embedding dimension (elements)**				**SIMD width (bits)**		
		dim1	dim2	dim3	dim4	64	128	256
No SIMD	Instructions							
	Execution time							
SIMD	Instructions							
	Execution time							
Speedup (normalized to no SIMD)								

Note: This is just an example table, so we have provided entries for only a few embedding dimensions and SIMD widths. You should expand and modify it as needed based on your analysis.

Task 2C: Software prefetching + SIMD

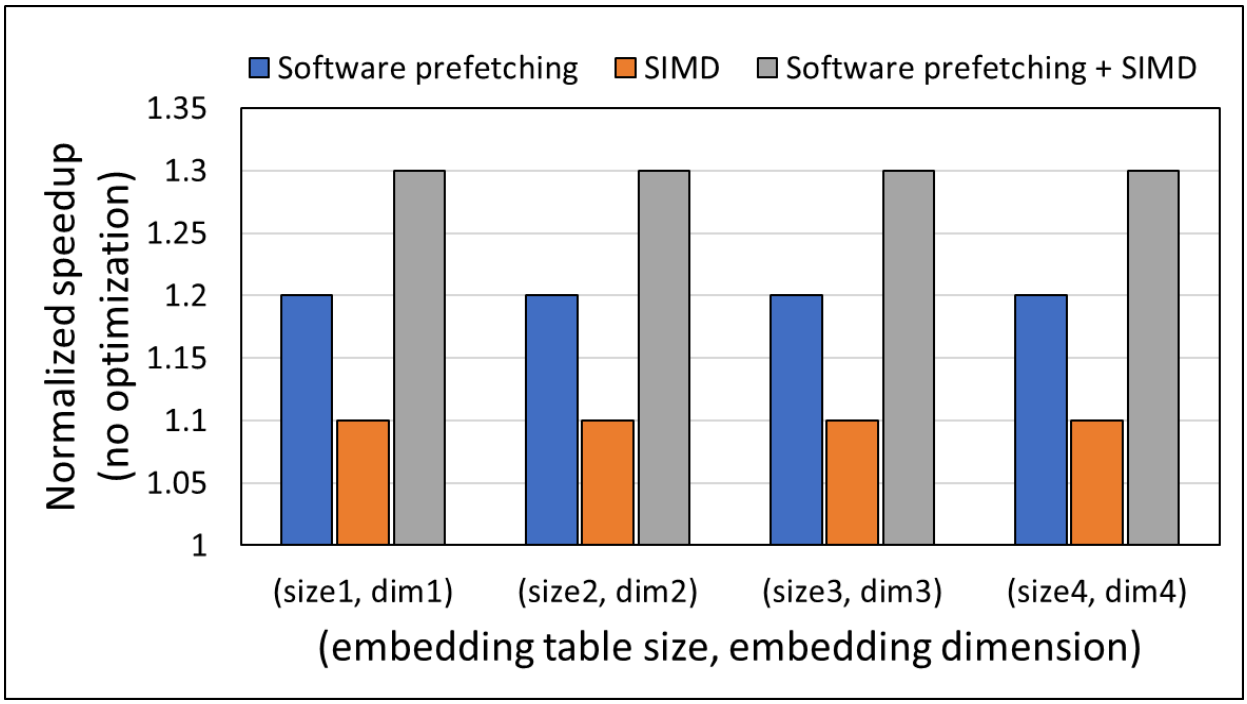
In this task, you will combine both software prefetching and SIMD techniques to optimize the embedding operation. You are required to repeat all the analyses you performed individually in Task 2A (Software Prefetching) and Task 2B (SIMD), but now with both optimizations applied together.

Deliverables

1. *All the objectives from Task 2A and Task 2B*.

2. *Combine Table 2.1 and Table 2.2.*

Figure 2.1:



Final performance summary for Embedding Operation:

Include a plot that compares the best performance achieved by each optimization technique —

- Software Prefetching
- SIMD
- Software Prefetching + SIMD

Plot this comparison against varying embedding table sizes and embedding dimension sizes. Refer to **Figure 2.1** for reference.

APPENDIX

Content	
1	*perf* tool
2	Software prefetching functions
3	SIMD functions

1. *perf* tool

- The following are the links where you can find details about **perf**: https://perfwiki.github.io/main/
- Demo slides: https://docs.google.com/presentation/d/1w_acNFVTud5YkASOEUEnuKXHX_4-nhftENyV0Jji3og/edit?usp=sharing

2. Software prefetching functions

2.1. *mm_prefetch*

- To implement software prefetching, you will use '**_mm_prefetch**.'
- **_mm_prefetch** is an intrinsic function provided by Intel that prefetches data into the cache to enhance memory access efficiency. It enables programmers to give the processor advanced notice about

which memory locations will be accessed soon, reducing cache misses and improving performance.

- The function is part of Intel's SSE (Streaming SIMD Extensions) and is highly optimized for Intel architectures. It is especially effective when used with SIMD (Single Instruction, Multiple Data) operations.
- The following are the links where you can find details about **_mm_prefetch**:
    - https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=5152&text=prefetch
    - https://stackoverflow.com/questions/46521694/what-are-mm-prefetch-locality-hints

2.2. *builtin_prefetch*

3. SIMD functions

Here are a few basic points to get started with using SIMD (Single Instruction, Multiple Data) instructions.

3.1. *Understanding SIMD Registers:*

- SIMD (Single Instruction, Multiple Data) allows the same operation to be performed on multiple data points simultaneously, which can significantly speed up computations.
- _m128d and _m256d represent 128-bit and 256-bit SIMD registers, respectively. These registers can hold multiple double-precision (64-bit) floating-point numbers.
- _m128d can hold two double-precision floating-point numbers.
- _m256d can hold four double-precision floating-point numbers.
- There is also _m512d, which can hold eight double-precision floating-point numbers. You can check if your system supports this by doing:

```
lscpu | grep avx512
```

If this yields some flags like avx512*, there you go!

3.2. *Loading Data into SIMD Registers:*

- Before performing any SIMD operations, data must be loaded into these registers.
- One of the ways to load data is using functions like _mm256_loadu_pd (for _m256d registers), which loads four double-precision floating-point values from memory into a SIMD register. The "u" in loadu stands for "unaligned," meaning the data does not need to be aligned to a specific boundary in memory.

3.3. *Performing SIMD Operations:*

- Once the data is loaded into SIMD registers, you can perform arithmetic operations on these registers.
- Functions like _mm256_add_pd and _mm256_mul_pd allow you to perform addition and multiplication, respectively, on the elements in the registers.

You can refer to all these functions here: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Submission Instructions

- Download the source code.
- You should submit a single tar.gz file with the name **rollno1_rollno2_pa1.tar.gz** on Moodle. (**NOTE**: rollno1 **<** rollno2 **lexicographically**)
- Only one member should submit it on Moodle. (**negative marks** for dual submission)
- The folder structure within tar.gz should be in the format below. Place the files in the appropriate folders for all the tasks.

```
pa1-three-geniuses/
├── part1/                              # Part 1: Matrix multiplication tasks
│   ├── mat_mul/                        # Contains various matmul optimizations
│   │   ├── matrix.c                    # Naive and optimized matmul code
│   │   ├── Makefile                    # Build rules for matmul
│   │   └── mat_mul_analysis.pdf        # Report with plots and analysis
│   └── neural_net/             # neural net version with optimized matmul
│       ├── <all relevant source files>
│       └── neural_net.pdf
├── part2/                              # Part 2: Embedding or related work
│   ├── emb.cpp                         # Code for part2
├── plots/                              # Plots and charts used in the report
│   ├── speedup_comparison.png
│   ├── cache_effect.pdf
│   └── ...
├── README.md                           # Assignment overview and instructions
```

Deadline

Deadline: 1st September, 5 PM

Vivas: TBD