Programming Assignment 1 - Chak De Microarchitecture!
CS 683: Advanced Computer Architecture, Autumn 2024

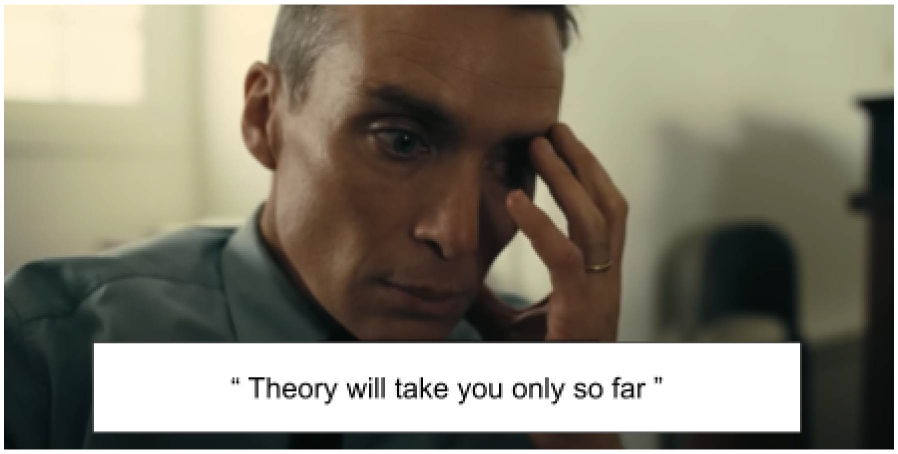
Computer Science and Engineering
Indian Institute of Technology, Bombay
CASPER group: https://casper-iitb.github.io/

---

* 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.*

Invite to the assignment: https://classroom.github.com/a/4u66Tjkg

We write a lot of programs, keeping the algorithm our primary concern, but while we do, we often forget about the underlying hardware on which our programs will run. A wise man once said,



In this programming assignment, we'll explore how we can exploit the architecture to gain an advantage beyond theory. We'll explore how we can better utilize the cache by improving the locality of programs. We'll see how we can use special hardware capabilities to speed up vector operations.

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?

NOTE:

- You need to have intel-based x86 machines to implement software prefetching.
- Make changes to the base codes only. Do not use a different implementation of the base codes.
- 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 own subparts. The task structure and their respective points are shown below:

Task 1 (Matrix transpose)		
1A	Tile it to see it	2.5 points
1B	Fetch it but with soft corner (software prefetching)	2.5 points
1C	Tiling + Prefetching	2 points
Task 2 (2D convolution)		
2A	Shhh SIMD in action	2 points
2B	Tile it again	2 points
2C	Software Prefetching	2 points
2D	Hum saath saath hain	2 points

Bonus Points: A bonus of 5 points will be given to the top 10 teams getting the best speedups**.**

Task 1A: Tile it to See it

One of the most common matrix operations is getting the transpose. This operation is particularly brutal on the cache if the matrix size is huge, as we access the matrix in column-major order. What if we can divide the matrix into tiles and get the transpose per tile? Maybe we'll be gentle on the cache then…

TODOs

1. Report the **L1-D cache MPKI** when executing only the naive matrix transpose.
2. Implement the tiled matrix transpose.
3. Report the **L1-D cache MPKI** when executing only the tiled matrix transpose.
4. Compare the performance by calculating the **speedup**.
5. Do this for multiple matrix sizes and tile sizes and analyze the results.
6. Plot the MPKI and speedup for different matrix sizes and tile sizes of your choice. Select the sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the changes in L1-D MPKI that you observed while moving from the naive to the tiled matrix transpose, and argue.
2. How much did the L1-D MPKI change for different matrix sizes and tile sizes? Explain the findings.
3. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Task 1B: Using Software Prefetching

" Don't let PROCRASTINATION take over your life. Be brave and take risks. Start PREFETCHING "

Software prefetching is a technique that aims to reduce cache misses by fetching data into the cache before it is needed. In this section, you will optimize the matrix transpose code using software prefetching techniques. Explain the concept of software prefetching and the different strategies that can be employed. Strategies like temporal locality of fetched data, fetching a variable number of addresses at a time, etc., can be considered.

Note:

1. You will have to turn off hardware prefetching to see the effects of software prefetching. How?

TODOs

1. Report the **number of instructions** and **L1-D cache MPKI** when executing only the naive matrix transpose.
2. Implement the software-prefetched matrix transpose.
3. Report the **number of instructions** and **L1-D cache MPKI** when executing only the software-prefetched matrix transpose.
4. Compare the performance by calculating the **speedup**.
5. Do this for multiple matrix sizes and analyze the results.
6. Plot the MPKI and speedup for different matrix sizes of your choice. Select matrix sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the change in the number of instructions that you observed while moving from naive to the software-prefetched matrix transpose, and argue.

2. Report the change in L1-D MPKI that you observed while moving from the naive to the software-prefetched matrix transpose, and argue.
3. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Resources:

1. To implement software prefetching, you will use '**_mm_prefetch**.'
2. **_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 advance notice about which memory locations will be accessed soon, reducing cache misses and improving performance.
3. 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.
4. 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

Task 1C: Tiling + Prefetching

You need to optimize the matrix transpose further using a combination of tiling and software prefetching.
Let T(**x**) be the time taken while executing matrix transpose with technique **x**.
So the **goal** of this task is as follows:
T(tiling + prefetching) < *min*(T(tiling), T(prefetching))

TODOs

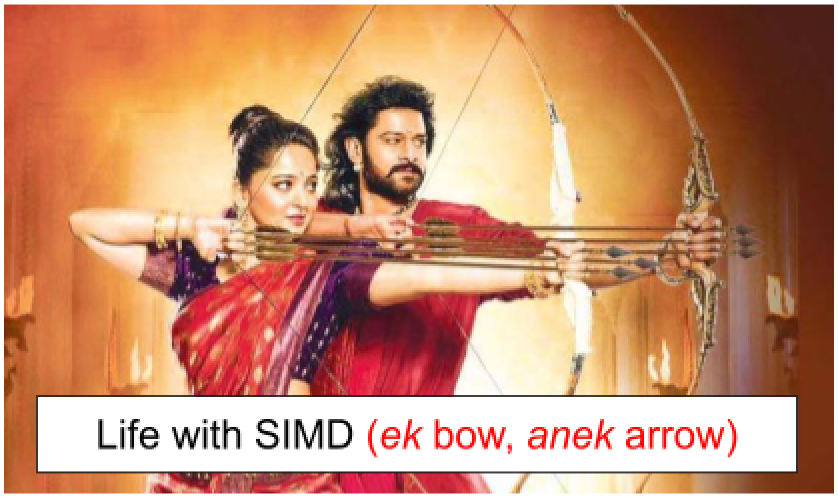
1. Report the **number of instructions** and **L1-D cache MPKI** when executing only the naive matrix transpose.
2. Implement the tiled + software-prefetched matrix transpose.
3. Report the **number of instructions** and **L1-D cache MPKI** when executing only the tiled + software-prefetched matrix transpose.
4. Compare the performance by calculating the **speedup**.
5. Do this for multiple matrix sizes and analyze the results.
6. Plot the MPKI and speedup for different matrix sizes and tile sizes of your choice. Select sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the change in the number of instructions that you observed while moving from the naive to the software-prefetched matrix transpose, and argue.

2. Report the change in L1-D MPKI that you observed while moving from the naive to the software-prefetched matrix transpose, and argue.
3. How much did the L1-D MPKI change for different matrix sizes and tile sizes? Explain the findings.
4. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Task 2A: Shhh, SIMD in action 🏹



Let's move on to another common operation used in image processing: **2D convolution**. Here, we have vector operations that can be optimized using special registers that modern processors have.

SIMD (Single Instruction, Multiple Data) instructions are specialized instructions that simultaneously perform operations on multiple data elements. These instructions can significantly speed up 2D convolution. In this section, you will explore SIMD instructions, such as SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions), depending on the available hardware.

TODOs

1. Report the **number of instructions** when executing only the naive convolution algorithm.
2. Implement the SIMD 2D convolution algorithm.
3. Report the **number of instructions** when executing only the SIMD 2D convolution algorithm.
4. Compare the performance by calculating the **speedup**.
5. Do this using SIMD registers of width 128 bits and 256 bits (and 512 bits if available) and compare the speedups.
6. Do this for multiple matrix sizes and kernel sizes and analyze the results.
7. Plot the speedup for the different matrix sizes and kernel sizes of your choice. Select the sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the change in the number of instructions that you observed while moving from the naive to the SIMD 2D convolution algorithm, and argue.
2. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Resources:

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

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!

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. **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.

Task 2B: Tile it again

TODOs

1. Report the **L1-D cache MPKI** when executing only the naive 2D convolution.
2. Implement the tiled 2D convolution.
3. Report the **L1-D cache MPKI** when executing only the tiled 2D convolution.
4. Compare the performance by calculating the **speedup**.
5. Do this for multiple matrix sizes and kernel sizes and analyze the results.
6. Plot the MPKI and speedup for different matrix sizes, kernel sizes, and tile sizes of your choice. Select the sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the change in L1-D MPKI that you observed while moving from the naive to the tiled 2D convolution, and argue.
2. How much did the L1-D MPKI change for different matrix sizes, kernel sizes, and tile sizes? Explain the findings.
3. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Task 2C: Software prefetching

Note:

1. You will have to turn off hardware prefetching to see the effects of software prefetching.

TODOs

1. Report the **number of instructions** and **L1-D cache MPKI** when executing only the naive 2D convolution.
2. Implement the software-prefetched 2D convolution.
3. Report the **number of instructions** and **L1-D cache MPKI** when executing only the software-prefetched 2D convolution.
4. Compare the performance by calculating the **speedup**.
5. Do this for multiple matrix sizes and kernel sizes and analyze the results.
6. Plot the MPKI and speedup for different matrix sizes and kernel sizes of your choice. Select the sizes such that you can draw clear conclusions from the results.

Answer the following:

1. Report the change in the number of instructions that you observed while moving from the naive to the software-prefetched matrix multiplication algorithm, and argue.
2. Report the change in L1-D MPKI that you observed while moving from the naive to the software-prefetched matrix multiplication algorithm, and argue.
3. Did you achieve any speedup? If so, how much and what contributed to it? If not, what were the reasons?

Task 2D: Hum Saath Saath Hain

You need to optimize the 2D convolution further using a combination of all the techniques in place. You need to understand how each technique optimizes the 2D convolution and how they can interact synergistically to improve performance further. Complete the following functions in the provided template code to calculate the speedup against baseline naive implementation.
Let T(**x**) be the time taken while executing matrix convolution with technique **x**.
So the **goal** of this task is as follows:
T(technique 1 + technique 2 + ...) < *min*(T(technique 1), T(technique 2), ...)

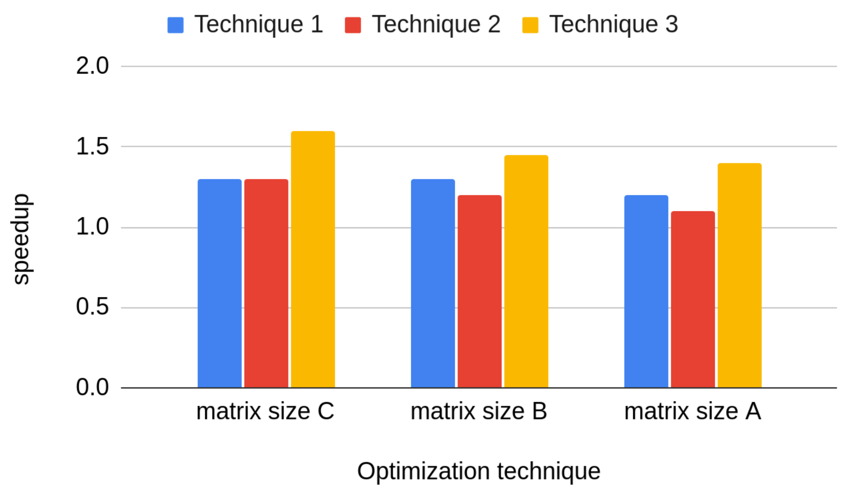
Techniques	Concerned function
Tiling + SIMD	tiled_simd_convolution
Tiling + prefetching	tiled_prefetch_convolution
SIMD + prefetching	simd_prefetch_convolution
Tiling + SIMD + prefetching	simd_tiled_prefetch_convolution

1. Measure and report the execution time of all the combinations.
2. Compare the performance improvement achieved by each technique against the baseline implementation.

Deliverables

1. Source code for all the tasks in ***transpose.c*** and ***convolution.c*** files.
2. README.md summarizing all the tasks and their respective todos. Describe what you did, why you did it, and how much it benefited you. Compare and analyze the performance improvements achieved by each of the tasks. Discuss any trade-offs or limitations associated with each optimization technique. Reflect on the importance of understanding hardware architecture and the impact it has on performance.
3. All the plots should be included in the README.md file along with their summary.

4. Include a plot showing the comparison of the performance of each technique along with the combination of the techniques against the different matrix sizes in the README.md file. There will be two different plots for transpose and convolution operations, respectively. This is what your plot should look like



Submission

- You should submit a single tar.gz file with the name **roll_number_pa1.tar.gz** on Moodle.
- The folder structure within tar.gz should be in the below format. Place the files in the appropriate folders for all the tasks.

```
---- pa1-chak-de-microarchitecture-template
      |----- part1
                  |---- Makefile
                  |---- transpose.c
      |----- part2
                 |---- Makefile
                 |---- convolution.c
      |----- README.md
```

- Kindly read the document at this link to create a private repository for the assignment. Do not push everything at the last moment. Maintain a proper commit history.

Appendix

Instructions to build and run the project:

1. **For part1 (cd to the part1 directory)**

   There are various sections to run for part 1:

    1. `naive`
    2. `tiling`
    3. `prefetch`
    4. `tiling-prefetch`
    5. `all`

   To execute a section in part1, you can run:

   ```
   make <section-name>
   ```

   This will create an executable file in the build directory; to run the executable, just do the following:

   ```
   ./build/<section-name> <matrix-size> <tile-size>
   ```

   The `<block-size>` can be a random number for the sections where tiling is not performed.

2. **For part2 (cd to the part2 directory)**

   There are various sections to run for part 2:

    1. `naive`
    2. `tiling`
    3. `prefetch`
    4. `simd`
    5. `tiling-prefetch`
    6. `tiling-simd`
    7. `simd-prefetch`
    8. `tiling-simd-prefetch`
    9. `all`

   To execute a section in part2, you can run:

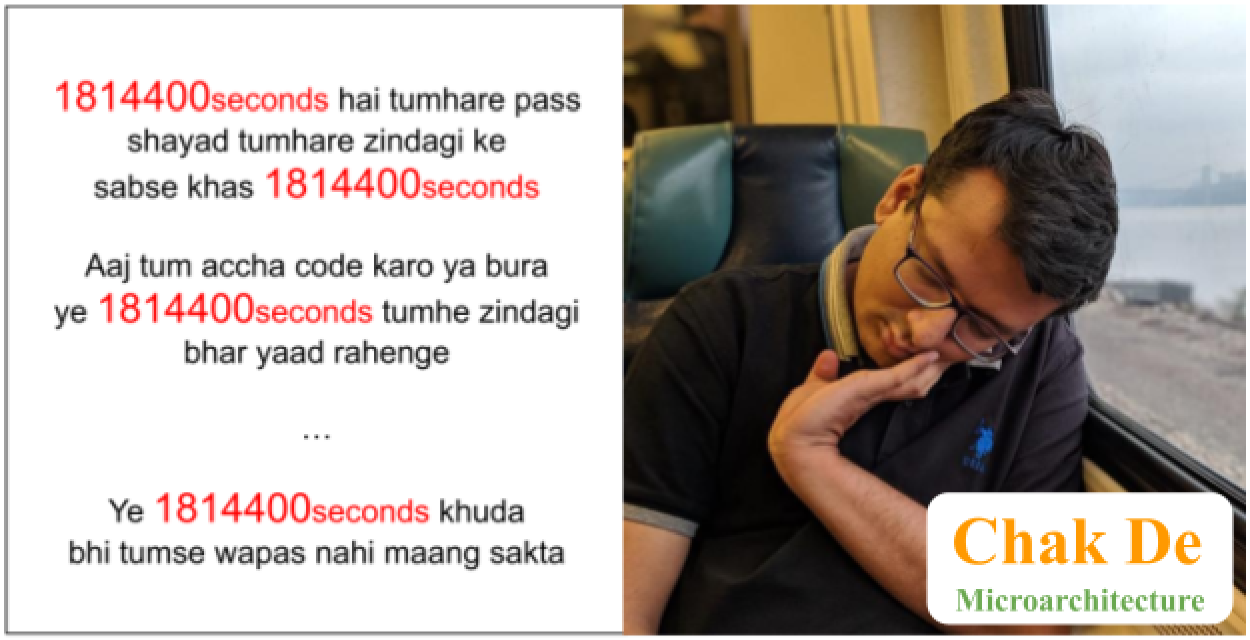
   ```
   make <section-name>
   ```

   This will create an executable file in the build directory; to run the executable, just do the following:

   ```
   ./build/<section-name> <matrix-size> <kernel-size>
   ```

The block size for the tiling tasks can be defined in the program itself.

Deadline


1814400seconds hai tumhare pass
shayad tumhare zindagi ke
sabse khas 1814400seconds
Aaj tum accha code karo ya bura
ye 1814400seconds tumhe zindagi
bhar yaad rahenge
...
Ye 1814400seconds khuda
bhi tumse wapas nahi maang sakta
Chak De
Microarchitecture

Best wishes