Aditya Pushkar
Roll No. 251110004

CS610 Assignment 1

Problem 1

Given

- Cache size = 128KB
- Block size = 128B
- Number of cache lines = 128KB / 128B = 1024 (0-1023)
- Word size = 4B
- Number of words in a cache line = 128 / 4 = 32
- Number of cache blocks in a set = 1024 / 8 = 128
- Address of A = 0x12345678
- Address of B = 0xabcd5678

Stride Analysis

Stride = 1

Clearly for the first iteration, A[0] will be a miss (cold miss) so the block A[0] to A[31] will be loaded into set 0 from memory to cache. After that, B[0] will be searched and it will also be a cold miss, so the block B[0] to B[31] will be brought from memory to set 0, evicting the block of A[0] to A[31]. Now for A[1], this will also be a miss, but this time it will be a conflict miss because the block was evicted by B[0] to B[31]. Again, the block A[0] to A[31] will be cached to set 0, evicting B[0] to B[31]. The same will happen for A[2] to A[31] of the first block of A. When we search for A[32], it will be a cold miss, as the block A[32] to A[63] has not been loaded into cache before. Then B[32] will cause a cold miss and its block will evict the block of A[32] to A[63] from set 0, resulting in conflict misses for A[33] to A[63]. This pattern continues for every 32-word block of A. Therefore, for every 32 words of A, there is 1 cold miss and 31 conflict misses. For the first iteration of the outer loop, the number of cold misses = 32K / 32 = 1024, and the remaining iterations of the outer loop cause conflict misses = 999 * 1024. The total number of misses = **1024 + 999*1024 = 1,024,000**.

Stride = 16

For stride = 16, each access jumps 16 words (64 bytes). The first access A[0] will be a cold miss and the block A[0] to A[31] will be cached in set 0. Then B[0] will also be a cold miss, bringing B[0] to B[31] into set 0, evicting the A block. Now A[16] falls within the same block of A[0] to A[31], but since it was evicted by B[0] to B[31], it will be a conflict miss. This block is reloaded into set 0, evicting the B block. Similarly, A[32] will be a cold miss for the block A[32] to A[63], followed by B[32] cold miss that evicts this block, leading to conflict misses for subsequent accesses A[48] etc. This alternation continues for all blocks of A, resulting in 1,024 cold misses and 999,000 conflict misses, summing to **1,024,000** total misses.

Stride = 32

For stride = 32, each access jumps exactly 32 words (128 bytes), which corresponds to the size of one cache block. Initially, A[0] will be a cold miss, so the block A[0] to A[31] is brought from memory to set 0. Immediately after, B[0] will also be a cold miss, bringing the block B[0] to B[31] into set 0. This evicts the block of A[0] to A[31] from the cache. Now, when we access A[32], this will be a cold miss for the new block A[32] to A[63], which is loaded into set 0. Accessing B[32] will also be a cold miss, and its block will again evict the A[32] to A[63] block from set 0, causing a conflict. The same pattern repeats for every subsequent block of A and B throughout the entire array. For the first iteration of the outer loop, each new block of A causes a cold miss, and the corresponding B access immediately evicts it, creating conflict misses for all remaining accesses to that block during the inner loop. For the remaining 999 iterations of the outer loop, all accesses to A result in conflict misses because each block of A keeps getting evicted by the corresponding B block that maps to the same set 0. This results in a total of 1,024 cold misses for the first iteration and $999 \times 1{,}024$ conflict misses for the remaining iterations, giving **1,024,000** total misses.

Stride = 64

For stride = 64, each access jumps 64 words (256 bytes), which corresponds to two cache blocks per access. Initially, A[0] will be a cold miss, bringing the block A[0] to A[63] into set 0. Then B[0] will also be a cold miss, and its block B[0] to B[63] will be brought into set 0, evicting the A[0] to A[63] block. When we access A[64], this will be a cold miss for the block A[64] to A[127], which will again be loaded into set 0. B[64] will evict this block, causing a conflict miss for subsequent accesses in this block. Because each access covers larger blocks, fewer blocks compete for the cache simultaneously, so the total number of misses is reduced compared to stride = 32. This pattern continues through all iterations of the inner and outer loops, and every new block of A gets evicted by the corresponding B block in set 0, leading to **512,000** total misses.

Stride = 2K

For stride = 2K (2048 words or 8,192 bytes per access), each access jumps much farther, so consecutive accesses of A map to different blocks that occupy the same set 0 at intervals. Initially, A[0] will be a cold miss, loading the block A[0] to A[2047] into set 0. B[0] will also be a cold miss, and its corresponding block will evict this A block from set 0, resulting in a conflict miss for subsequent accesses to that block in the inner loop. Next, A[2048] will be a cold miss for the new block, and B[2048] will evict it again. Because the stride is large, only a small number of blocks of A map to the same cache set at the same time, so the total number of misses is significantly lower. This pattern continues across all iterations of the outer loop, producing **16,000** total misses, mostly from cold misses for the first access to each far-apart block and a few conflict misses due to repeated eviction by B blocks in set 0.

Stride = 8K

For stride = 8K (32,768 bytes per access), each access jumps so far in memory that very few blocks of A and B ever map to the same set 0 simultaneously. Initially, A[0] causes a cold miss, loading its block into set 0. B[0] will also cause a cold miss, evicting this block. The next access, A[8192], will again be a cold miss for a new block, and B[8192] will evict it. Because the accesses are so widely spaced, almost no further conflicts occur, and each iteration of the outer loop does not significantly increase misses. Consequently, only 4 cold misses occur in total, corresponding to the first access to each of the four far-apart blocks, with almost no conflict misses. Therefore, the total number of misses = **4**.

Stride vs Total Misses

Stride	Total Misses (A)
1	1,024,000
16	1,024,000
32	1,024,000
64	512,000
2K	16,000
8K	4

Problem 2

Let BL denote the cache block size, i.e., the number of words (doubles here) that can be stored in a single cache block. Here, $BL = 16$. The matrix dimensions are $N \times N$ with $N = 1024$. Thus:

$$\frac{N}{BL} = \frac{1024}{16} = 64$$

Since the matrices are much larger than the cache capacity, the entire data set cannot fit in cache. Each cache line can store 16 words.

Direct-Mapped Cache

Array C: Accesses to C in the i and j loops proceed in row-major order. This means within the innermost loop, every 16th access (one block's worth) generates a cache miss. Hence, the j loop contributes N/BL, and this pattern repeats for each i, so i contributes N. Finally, since the entire matrix is larger than the cache, advancing k also contributes N.

Array A: In this loop ordering, the j loop repeatedly uses the same element $A[i][k]$, so it adds only a factor of 1. But the i and k loops traverse the matrix column-wise, which conflicts with the cache mapping. To illustrate: $A[0][0] \dots A[0][15]$ sit in one block, but when we reach $A[64][0]$, it maps to the same line and overwrites the earlier block. Thus, for both i and k, the contribution is N, resulting in misses for nearly every access.

Array B: For each k, the j loop walks along row $B[k]$. Streaming across a row yields one miss per block, so j contributes N/BL. Since the same row is reused across all i iterations, i adds only 1. Each new k selects a different row, so k contributes N.

Fully Associative Cache

Arrays B and C: Their accesses follow row-major order, so the miss behavior is unaffected by associativity compared to the direct-mapped case.

Array A: Here the difference shows. With fully associative replacement, when we fetch $A[64][0]$, the earlier block $A[0][0] \dots A[0][15]$ does not need to be evicted just because of index conflicts. Therefore, for each i, only every 16th k iteration brings in a new block. In other words, the k loop's contribution drops from N to N/BL.

Case 2: kij form

(a) Direct-Mapped Cache

	A	B	C
i	$N = 1024$	1	$N = 1024$
j	1	$N/BL = 64$	$N/BL = 64$
k	$N = 1024$	$N = 1024$	$N = 1024$
Total	$N^2 = 2^{20}$	$N^2/BL = 2^{16}$	$N^3/BL = 2^{26}$

(b) Fully Associative Cache

	A	B	C
i	$N = 1024$	1	$N = 1024$
j	1	$N/BL = 64$	$N/BL = 64$
k	$N/BL = 64$	$N = 1024$	$N = 1024$
Total	$N^2/BL = 2^{16}$	$N^2/BL = 2^{16}$	$N^3/BL = 2^{26}$

Case 2: jki form

Direct-Mapped Cache

For A: The innermost i loop scans column-wise through $A[i][k]$, which gives poor locality. Because of the stride-1024 accesses and conflicts in a direct-mapped cache, almost every access is a miss. Thus i, k, and j all contribute N, giving N^3 total misses.

For B: For a given j, the k loop traverses $B[k][j]$ down a column. This is again column-major access, so each new access maps to a different block without reuse. The i loop does not add new misses since r is reused. Contributions are $k = N$, $j = N$, $i = 1$, so total N^2.

For C: The inner i loop again touches a column of $C[i][j]$. In direct mapping, each access misses. Since every k iteration updates the same column, conflicts persist, so i, j, and k each contribute N. Total misses are N^3.

Fully Associative Cache

For A: With full associativity, when walking down a column, the accessed blocks can be retained. Thus, only every BLth k access per i brings in a new block. Contributions: $i = N$, $k = N/BL$, $j = N$. Total: N^3/BL.

For B: For a fixed k, moving across j traverses rows, which gives row-major streaming. Therefore, j contributes N/BL. The i loop still contributes 1 (since r is reused) and k contributes N. Total: N^2/BL.

For C: For a fixed j, the i loop traverses a column, requiring N blocks. Since 1024 blocks fit easily within the 4096-line cache, these remain across different k. Thus only the first load per block costs a miss. Contributions: $i = N$, $j = N$, $k = 1$. Total: N^2.

—

(a) Direct-Mapped Cache

	A	B	C
i	$N = 1024$	1	$N = 1024$
k	$N = 1024$	$N = 1024$	$N = 1024$
j	$N = 1024$	$N = 1024$	$N = 1024$
Total	$N^3 = 2^{30}$	$N^2 = 2^{20}$	$N^3 = 2^{30}$

(b) Fully Associative Cache

	A	B	C
i	$N = 1024$	1	$N = 1024$
k	$N/BL = 64$	$N = 1024$	1
j	$N = 1024$	$N/BL = 64$	$N = 1024$
Total	$N^3/BL = 2^{26}$	$N^2/BL = 2^{16}$	$N^2 = 2^{20}$

Problem 3

Compilation and Execution: The program is compiled using the following command:

```
g++ -std=c++17 -pthread main.cpp -o myprogram
```

It can then be executed with arguments specifying the input file, the number of producer threads, the minimum and maximum number of lines each producer can read, the buffer size, and the output file. For example:

```
./myprogram input.txt 4 5 10 50 output.txt
```

This runs the program with 4 producers, each reading between 5 and 10 lines, using a shared buffer of 50 lines, and writing results to `output.txt`.

Synchronization: For synchronization, I used a mutex (`buffer_mtx`) so that only one thread modifies the shared buffer at a time. I also used two condition variables: `cv_full`, which makes producers wait whenever the buffer is already full, and `cv_empty`, which makes consumers wait if the buffer is empty. To prevent producers from overlapping on the same lines, I used an atomic variable (`currentLine`) to assign line numbers safely, and another atomic flag (`producersDone`) to indicate when all producers have finished their work. For the output, I added a separate mutex (`out_mtx`) so that each consumer writes its results to the output file without interference from others. With this combination of mutexes, condition variables, and atomic variables, the program avoids race conditions and ensures smooth coordination between producers and consumers.