CS610 Assignment 1

Pragati Agrawal (220779)

August 2025

Problem 1

Note: I have assumed that the addressed of A and B are not aligned to the beginning of the cache block.

Given Cache Parameters:
Word size $W = 4\ B$
Block size $BL = 128\ B \implies 1Block = 32$ words
Block offset bits $= \log_2(128) = 7$
Cache capacity $C = 128\ KB = 2^{17}\ B = 2^{15}$ words $= 2^{10}$ blocks
Associativity $A = 8$
Number of sets $S = C/A = 2^{10}/8 = 2^7 = 128$
Set index bits $= \log_2(128) = 7$
So, address bits: [13–7] are the set index bits, [6–0] are the block offset bits.

Array size $32K\ floats = 128KB = 2^{17}B$
Given starting addresses of A and B:
$A : 0x12345678 \Rightarrow s_A = 44.$
$B : 0xabcd5678 \Rightarrow s_B = 44.$
Thus, $A[i]$ and $B[i]$ always map to the same set, effectively reducing associativity to 4 (since during every addition operation, A[i] and B[i] will be requested, and both will sit in the same set of the cache side-by-side).
Also, elements A[0] and A[1] will fit in first cache block, i.e. at set index 44, while A[2] to A[33] will come in the next cache block, then A[34] to A[65] and so on. So the last block will be A[32738] to A[32767] and it will also map to the set 44. Same argument follows for B as the case is symmetric. So both arrays span $2^{10} + 1$ blocks, and since effective associativity for each array is 4, just half of the arrays would be able to fit in the cache at any point in time.

Stride = 1

On the first access of each block of 32 words, we get a miss; the remaining words in that block are hits. Since we have $2^{10}+1$ blocks effectively (A[0] gives an extra miss at the start), each iteration has 1025 misses. This is because blocks evict after one pass (4 blocks of each array in a set), the pattern repeats in every outer loop iteration, because the first blocks are no longer present in the cache. As an example, for set 44, the blocks of A will be: A[0]-A[1], A[4066]-A[4097], A[8162]-A[8193], A[12258]-A[12289], and then A[16354]-A[16385] will evict A[0]-A[1] (due to LRU), A[20450]-A[20481] will evict A[4066]-A[4097]), A[24546]-A[24577] will evict A[8162]-A[8193], A[28642]-A[28673] will evict

A[12258]-A[12289] and lastly A[32738]-A[32767] will evict A[16354]-A[16385]. The last miss will not be seen in the other sets, but only for set 44. Other misses will be seen in all the other sets for each iteration.

$$\text{Total misses} = 1000 \times 1025 = \mathbf{1025000}$$

Stride = 16

Now only two words per block are accessed, but all blocks are still required to be accessed (even the last block A[32738]-A[32767]). The first such access is always a miss, but the second is a hit. Thus, each block still causes exactly one miss, just like stride 1.

$$\text{Total misses} = 1000 \times 1025 = \mathbf{1025000}$$

Stride = 32

Here only one word per block is touched, so every access results in a miss. But here the last block to be accessed will be A[32736], which will map to set 43. So we will not see the last extra miss in set 44 every iteration (that we saw earlier). There are 1024 blocks in A, so we get 1024 misses per iteration.

$$\text{Total misses} = 1000 \times 1024 = \mathbf{1024000}$$

Stride = 64

Here, only every alternate block (alternate set block) is accessed, with one word per block. Thus, half the blocks of A are touched in each pass, and still the earlier blocks get replaced by the later ones after every iteration, so each iteration shows the same number of misses. That gives 512 misses per iteration.

$$\text{Total misses} = 1000 \times 512 = \mathbf{512000}$$

Stride = 2K

Each iteration now has only 16 accesses: $i = \{0, 2K, \ldots, 30K\}$. These map to just two sets - set 44 and set 108, and every new access evicts the earlier one from the same set (since A and B compete and we are accessing 8 blocks for each array in each set). Hence, all 16 accesses are misses in every iteration.

$$\text{Total misses} = 1000 \times 16 = \mathbf{16000}$$

Stride = 8K

Only 4 accesses occur: $i = \{0, 8K, 16K, 24K\}$. All of them fall into the same set. The first iteration brings them into cache (cold misses), and then they stay without eviction, for every iteration after this. So only the first four accesses miss, and all later iterations are hits.

$$\text{Total misses} = \mathbf{4}$$

Summary

Stride	Total Misses
1	1025000
16	1025000
32	1024000
64	512000
$2K$	16000
$8K$	4

Problem 2

kij

Direct Mapped Cache

Cache Parameters:
Cache Size = 64K = 2^{16} words
Cache Line = 16 words (Blk)
Number of sets (direct mapped) = $64K/16 = 4K = 4096$
Size of any matrix = 2^{20} words

	A	B	C
k	1024	1024	1024
i	1024	1	1024
j	1	64	64
	2^{20}	2^{16}	2^{26}

- A,j: Same element reused in inner loop ⇒ factor 1.
- A,i: All i map to different blocks (column-wise traversal) ⇒ N = 1024 misses.
- A,k: Each column has $N/Blk = 1024/16 = 64$ blocks. These will map to $(64 * k)^{th}$ set, so only $4096/64 = 64$ sets will be useful for any k. By next k, all old rows will be evicted ⇒ N= 1024.
- B,j: One miss per 16 elements (row-wise traversal) ⇒ $N/Blk = 1024/16 = 64$.
- B,i: After first pass, row stays in cache ⇒ 1.
- B,k: Each k accesses new row ⇒ N=1024.
- C,j: Same reasoning as B,j (row-wise traversal) ⇒ 64.
- C,i: Each i is a new row ⇒ N=1024.
- C,k: Full C cannot fit, and one iteration of i*j swaps the whole matrix C, so old rows evicted each k ⇒ N=1024.

	A	B	C
k	64	1024	1024
i	1024	1	1024
j	1	64	64
	2^{16}	2^{16}	2^{26}

Fully Associative Cache

Same as direct mapped, except for A,k: now full cache can be used. For k=0, all misses; then for k 1–15 hits; repeat for k=16 onwards. Hence factor $N/Blk = 1024/16 = 64$.

jki

Direct Mapped Cache

	A	B	C
j	1024	1024	1024
k	1024	1024	1024
i	1024	1	1024
	2^{30}	2^{20}	2^{30}

- A,i: All different blocks (column-wise traversal) ⇒ 1024.
- A,k: Same as kij direct mapped (misses due to conflicts in cache sets) ⇒ 1024.
- A,j: Full A cannot fit, so pattern repeats every j ⇒ 1024.
- B: Same as A in kij case (factors 1, 1024, 1024 for i,k,j).
- C,i: All distinct blocks (column-wise traversal) ⇒ 1024.
- C,k: Similar to kij, cannot hold all i's (only some particular cache sets will get utilized) ⇒ 1024.
- C,j: Old blocks evicted every j iteration ⇒ 1024.

Fully Associative Cache

	A	B	C
j	1024	64	64
k	64	1024	1
i	1024	1	1024
	2^{26}	2^{16}	2^{16}

- A,i: All distinct blocks (column-wise traversal) ⇒ 1024.

- A,k: Same as kij fully associative (row-wise traversal and row fits in cache) $\Rightarrow N/Blk = 1024/16 = 64$.
- A,j: A does not fit, so repeated misses $\Rightarrow$ 1024.
- B: Similar to A in kij case (factors 1, 1024, 64).
- C,i: Distinct blocks (column-wise traversal) $\Rightarrow$ 1024.
- C,k: For fixed j, all i's fit, so only first k misses, others hit $\Rightarrow$ 1.
- C,j: Blocks reused across j, misses only once every 16 iterations $\Rightarrow$ 64.

Problem 3

How to run

```
g++ q3.cpp -o q3

./q3 input.txt 3 2 5 3 output.txt
```

Implementation

The program uses C++ threads, mutexes, and condition variables to synchronize producers and consumers as they interact with a shared buffer.

- **Shared Buffer:** The buffer is a `std::vector<std::string>` with a fixed maximum size (`bufferSize`). Producers write lines to it, and consumers read lines from it.
- **Mutexes:**
  - `producerLock`: Ensures only one producer writes its batch of lines to the buffer at a time, maintaining atomicity.
  - `prodConsLock`: Protects the buffer during both producer and consumer operations, preventing race conditions.
  - `inputFileLock`: Ensures only one producer reads from the input file at a time.
  - `printLock`: Serializes debug output to avoid interleaved prints.
- **Condition Variable:**
  - `bufferStatus`: Used by producers to wait when the buffer is full, and by consumers to wait when the buffer is empty. Producers call `wait` until there is space in the buffer; consumers call `wait` until there is data to consume.
- **Producer Workflow:**
  - Each producer reads a random number of lines (`L`) from the input file, protected by `inputFileLock`.
  - Producers acquire `producerLock` and `prodConsLock` before writing to the buffer.

  - If the buffer is full, the producer waits on `bufferStatus` until space is available.
  - Producers write their batch atomically, possibly in chunks if `L` is greater than `bufferSize`.
  - After writing, producers notify consumers using `bufferStatus.notify_all()`.

- **Consumer Workflow:**
  - Consumers acquire `prodConsLock` before reading from the buffer.
  - If the buffer is empty and producers are not done, consumers wait on `bufferStatus`.
  - When data is available, consumers read all lines in the buffer and write them to the output file atomically.
  - After consuming, consumers notify producers using `bufferStatus.notify_all()`.
  - Consumers exit when the buffer is empty and all producers have finished.

- **Termination:**
  - The `taskCompleted` flag and `producersDone` counter are used to signal consumers when all producers have finished, allowing consumers to exit gracefully.