CS610 Assignment-1

Mahaarajan J -220600

August 20, 2025

1 Problem1

Formalisation

- Firstly we assume that the addresses of A and B are block aligned for our analysis
- Size of a cache block is 2^7 bytes, size of the cache is 2^{17} bytes hence there are a total of 2^{10} cache blocks and because of 8 way associativity we will have $2^{10}/8 = 2^7$ many sets
- We assume that A and B compete for cache space and from the addresses of A, B (after block-aligning) would map to the same set with different tags for same value of i (offset) as the last 14 bits would be the same.
- Size of A,B is 2^{17} bytes each hence each of A, B would take up the space of $2^{17}/2^7 = 2^{10}$ contigious cache blocks respectively
- Since the data items are floats of 4B we can see that 2^5 floats would fit into one cache block

Miss analysis for different strides

- Stride=1.
  - Look at the inner loop first, at the first access will be a miss and we will fill 2 cache blocks into set0 (one for A, one for B) and then the next 31 requests will be cache hits then this pattern would repeat
  - Now this pattern would continue till all 2^7 sets would have 2 cache blocks one from A and one from B, then it will wrap around and cache blocks would get filled in set0 again.
  - Since the cache is 8 way associative this pattern continues for 4 complete cycles (each cycle being 2^{12} iterations of the inner loop) before we get conflicts and cache blocks are evicted
  - Hence in the inner loop we will have a total of 2^{10} misses for A and B seperately. Now when we go to the next it iteration the cache would not contain any of the first 2^7 blocks of A or B hence cache miss pattern would repeat
  - Hence finally misses for A : 1 024 000 and misses for B: 1 024 000
- Stride=16,32 .
  - Similar analysis of stride=1, we will still need to pull 2^{10} cache blocks per array and when we go the next *it* iteration the initial required cache blocks would be evicted
  - Therefore the misses for A and B would be the same
- Stride=64 .
  - Here we only access half of the cache blocks (cache block contains 32 floats hence we skip alternate cache blocks) i.e we access a total of 2^9 cache blocks per array in the inner loop. However we will only be able to accomodate a maximum of $2^6 * 4 = 2^8$ cache blocks per array because it is a 8 way set associative cache and alternate sets would be empty because of the access pattern
  - Hence after filling 2^9 cache blocks (2^8 per array) we will have conflicts and will have to evict cache blocks from the already filled sets. Therefore when we get to the next iteration of *it* the required blocks would be evicted. Hence the miss pattern would repeat for each *it* iteration
  - Therefore misses for A: 512 000 and misses for B: 512 000
- Stride=2K .
  - As stride is 2K we would skip $2^{11}/2^5 = 2^6$ cache blocks for each access. Total cache blocks accessed per array would be 16 (as 16 elements and they are in different cache blocks)

- The sets used in the cache would be set0 and set2^6 and nothing else because it is an indexed cache and other sets would be empty
- Therefore after filling 4 cache blocks of A, 4 cache blocks of B in set0 and set2^6 we will get conflicts and would need to evict cache blocks. Hence on the next iteration of *it* we will have misses again and the pattern would repeat like the previous cases
- Therefore misses for A: 16 000 and misses for B:16 000

- Stride=8K .
  - As stride is 8K we would skip $2^{13}/2^5 = 2^9$ cache blocks. Hence the set0 will be the only set used $2^9 mod 2^7 = 0$. We have to access a total of 4 cache blocks per array
  - since the cache is 8 way associative these 8 cache blocks (4 for A and 4 for B) will miss on the first iteration of *it* but would hit thereafter and not be evicted. Hence we would have misses only for $it = 0$
  - Hence misses for A: 4 and misses for B: 4
- Summary Table:

Stride	**Misses for A**	**Misses for B**
1	1 024 000	1 024 000
16	1 024 000	1 024 000
32	1 024 000	1 024 000
64	512 000	512 000
2K	16 000	16 000
8K	4	4

Table 1: Total cache misses for arrays A and B across different strides

2 Problem2

Formalisation

- The size of the cache is 64K words and a cache block is 16 words. Hence the cache has a total of 2^{12} cache blocks
- The Matrix is 1024*1024 in size and each element is 1 word. Therefore 1 row would take up $2^{10}/16 = 2^6$ cache blocks and would hence fit in the cache, however the entire matrix 2^{16} cache blocks would not fit
- We assume that there is no interference and we would be analysing the cache misses for each matrix A,B,C indivudually without considering accesses to the other and different matrices don't compete for cache space

Analysis for kij

- Array A
  - Fully associative
    - Now inside the j loop (innermost) A is not accessed, In the next order loop (i loop) ith column of A is accessed. Now in the 1st cycle of the loop (k=0) we would bring in 2^{10} cache blocks which would fit in our cache (fully associative maximum 2^{12} cache blocks
    - This leads to hits for k=1,2,3...,15 then again we have a pattern of misses, the cache gets full eventually but this pattern of misses would continue as we try to access a new cache block
    - Hence the number of misses would be $N * (N/16)$ which is 2^{16} misses
  - Direct Mapped
    - Again in the i loop we would need to bring in 2^{10} cache blocks but this time we can't fit all of them in the cache due to sets. 1 row spans 2^6 cache blocks hence we would be able to use set0, set 2^6, set $2 * 2^6$ and so on due to the set indexing in the direct mapped cache. Hence during the 2^{10} misses some of them will be conflict misses and the initial cache blocks would be evicted
    - Hence during the next iteration $k = 1$ we will have misses again and the pattern continues of having misses for every access done due to eviction of the initial cache blocks
    - Hence the number of misses would be $N * N$ which is 2^{20} misses
- Array B

- Fully associative
  - In the innermost loop we access B row wise hence every 16th access would be a miss and as we saw an entire row of B would fit in the cache and take up 2^6 blocks. Therefore in the next loop of i since the row is already present all accesses would be hits.
  - Finally in the outermost loop, B is accessed in column major order and the elements are not cached, Therefore the pattern would continue for each iteration as it will be a fresh row accessed
  - Hence number of misses would be $N/16 * 1 * N$ which is 2^{16}
- Direct Mapped
  - In this case also the row would fit in consecutive sets as the cache blocks are contigiously accessed. Hence we will have all hits on the i loop after the first iteration, similar behaviour for the k loop with the fresh row
  - Here as we are accessing row wise sequentially no set of the cache remains empty and cache blocks evicted only when cache is full, Therefore we are able to use the complete cache hence giving the same misses as when it was fully associative
  - Hence number of misses would be $N/16 * 1 * N$ which is 2^{16}

- Array C
  - Fully associative
    - In this case in the innermost loop row wise accesses so every 16th access will be a miss. In the next loop of i we are accessing different rows and hence they will also be misses and this pattern would repeat
    - Finally in the outermost loop the initial cache blocks would have been evicted as the entire matrix does not fit in the cache. Therefore the pattern of misses would repeat for each iteration
    - Hence the number of misses would be $N/16 * N * N$ which is 2^{26}
  - Direct Mapped
    - This would be similar to Fully associative as innermost loop every 16th access would be a miss and this pattern would continue for loop i as a new row and loop k as the entire matrix wouldn't fit (if doesn't fit in Fully associative will not fit in Direct mapped)
    - Moreover all sets are utilized here hence the behaviour of cache misses wouldn't change
    - Hence the number of misses would be $N/16 * N * N$ which is 2^{26}

Analysis for jki

- Array A
  - Fully associative
    - Innermost loop accesses are column wise hence we would have 2^{10} misses and these 2^{10} blocks would be there in the cache for the next iterations of k. Hence k=1,2,...,15 will be all hits (as they would be pulled in with the same cache block) and again we would have a miss and so on
    - However the entire matrix would not fit in the cache hence during the next iteration of j the required cache blocks would've been evicted hence the miss pattern continues
    - Hence the number of misses would be $N * N/16 * N$ which is 2^{26}
  - Direct Mapped
    - The entire column won't fit in the cache due to it being set indexed as we saw earlier. Hence now every iteration of k would be a miss as the required cache block would've been evicted
    - Hence the number of misses would be $N * N * N$ which is 2^{30}

- Array B
  - Fully associative
    - Innermost loop no B accessed, in the k loop B is accessed column wise, which means we will have 2^{10} misses and the column would fit in the cache,
    - Therefore when we look at iterations of j only every 16th access would be a miss as 16 elements of the row will be in cache with the column
    - Hence the number of misses would be $N * N/16$ which is 2^{16}
  - Direct Mapped
    - Again similar to previous the entire column would not fit in cache therefore the consecutive iterations of j would lead to misses in the pattern that every access is a miss

    - Hence the number of misses would be $N * N$ which is 2^{20}

- Array C
  - Fully associative
    - Innermost loop C is accessed column wise and since the cache is fully associative the column would fit in the cache. Now in the k loop it is this column that is repeatedly accessed hence from $k = 1$ we will have all hits
    - Finally in the outer j loop column 1,2,3 ...15 would already be in cache hence we would have hits alone and we will have a miss pattern every 16th column as the cache block size is 16
    - Hence the number of misses would be $N * 1 * N/16$ which is 2^{16}
  - Direct Mapped
    - Innermost loop is accessed column wise and as we saw a column won't fit in the direct mapped cache. Hence in the k loop on accessing the column again we will have misses as the required cache blocks would be evicted
    - Further even in the j loop we would have misses only as the last part of the column is what will be present in the cache
    - Hence the number of misses would be $N * N * N$ which is 2^{30}

Tables

Loop	A	B	C
k	N/B	N	N
i	N	1	N
j	1	N/B	N/B

Table 2: (a) kij Fully Associative

Loop	A	B	C
k	N	N	N
i	N	1	N
j	1	N/B	N/B

Table 3: (b) kij Direct Mapped

Loop	A	B	C
j	N	N/B	N/B
k	N/B	N	1
i	N	1	N

Table 4: (c) jki Fully Associative

Loop	A	B	C
j	N	N	N
k	N	N	N
i	N	1	N

Table 5: (d) jki Direct Mapped

Note: N is 1024 the size of the row of the square matrix and B is the cache block size in words which is 16

3 Problem3

3.1 Assumptions and constraints

- I have assumed everything given in the question and enforced the correctness constraint of the lines read by a single thread appearing together in the output file.
- I have assumed that the input and output file paths are valid, The value of L_max is $\geq$ L_min and both are ≥ 0 with L_max not 0 and the program checks for these constraints.

3.2 Implementation details

- Synchronization and shared variables
  - Used a total of 5 locks, one for reading the input file,one for accessing the buffer, one for consumer to write to output file, one for producer to wait and write its complete L lines even if buffer is full (`buff_lock_prod_only`) and one for consumer to wait and read from buffer even if empty when the producer hasn't finished its chunk (`buff_lock_cons_only`)
  - Additionally used 2 condition variables one for producers to wait when buffer is full and one for consumers to wait when buffer is empty `buffer_not_full buffer_not_empty`
  - Used 3 boolean flags one to indicate that a single producer thread has completed its write to buffer, one to indicate that all producers are done and the last to denote that input file has a trailing newline or not
  - Used a shared buffer as a FIFO queue of lines `queue<string> shared_buffer` and variables to track number of lines read and written

- Main
  - The main function takes arguments as given in assignment and spawns T many producer threads and the $max(1, T/2)$ consumer threads and also notifies the consumers when the producer threads are finished by using a `producers_done` then does cleanup and closure of files
  - Main also checks if the last character in the input_file is newline and sets the flag to prevent unncessary newline in the output
- Producer
  - Firstly L is sampled randomly between L_min and L_max for each read
  - A producer takes `in_lock` for input_file and reads, it then releases that lock. If number of lines left to read is 0 then thread exits
  - Then it takes `buff_lock_prod_only` and writes to buffer repeatedly ,by taking `buff_lock` (condition variable used to ensure buffer is not full) and releasing it after one write, `buff_lock_prod_only` is released only on complete of writing all the lines it has read and sets `thread_done` flag
  - It then samples a different value of L and waits to read again
- Consumer
  - Takes the lock `buff_lock_cons_only` and loops till the producer sets `thread_done` and in each loop it empties the buffer by taking `buff_lock` (condition variable used to ensure buffer is not empty) and stores it locally in a vector
  - Then it writes all the lines it has read (possibly multiple times) to the output file after taking the `out_lock`
  - Then it waits to take the lock and read from buffer again

3.3 Running Instructions

To run the solution for Problem 3, follow the steps below:

1. Unzip the submission archive:

   ```
   tar -xzf 220600-assign1.tar.gz
   ```

2. Change to the source directory:

   ```
   cd 220600-assign1
   ```

3. Compile the program using the provided `Makefile`:

   ```
   make
   ```

4. Ensure that your input file is present in the same directory as the executable.
5. Run the program using:

   ```
   ./prob_3 <input_file> <T> <Lmin> <Lmax> <M> <output_file>
   ```

   Example:

   ```
   ./prob_3 input.txt 9 4 6 10 output.txt
   ```

Make sure that the input and output files specified in the command are located in the current working directory. The output file will be created (or overwritten) after successful execution.