CS610 : Programming For Performance

Assignment 1

Submitted By : Kushagra Srivastava, 220573

Problem 1

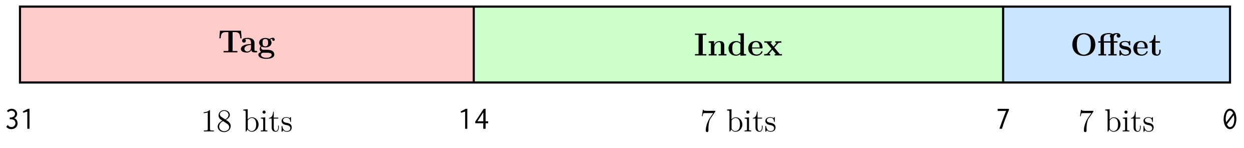
Consider the following loop.

```c
#define N (1000)
#define SIZE (1<<15) // 32K
float s = 0.0f, A[SIZE], B[SIZE];
int i, it, stride;
for (it = 0; it < N; it++) {
  for (i = 0; i < SIZE; i += stride) {
    s += (A[i] + B[i]);
  }
}
```

Assume an 8-way set-associative 128 KB cache, line size of 128 B, and word size of 4 B (for `float`). The cache is empty before execution and uses an LRU replacement policy. Determine the total number of cache misses on `A` for the following access strides: `1`, `16`, `32`, `64`, `2K`, and `8K`. Consider all the three kinds of misses, and arrays `A` and `B` compete with each other for cache space. The addresses of `A` and `B` are `0x12345678` and `0xabcd5678`.

Solution

Consider the addresses to be block-aligned. The physical byte address decomposes as the following for the 8-way set associative cache.



Two addresses map to the same *set* iff their index fields match i.e.,

$$\left\lfloor \frac{\text{address}}{128} \right\rfloor \bmod 128 \quad \text{is equal for both.}$$

Using this observe that, line L of A and line L of B (for the same L) always have *identical set indices*. With `SIZE` $= 2^{15}$ elements of `float` (4 B), each array occupies the entire cache, when competely filled.

$$|A| = |B| = 2^{15} \times 4\,\text{B} = 131{,}072\,\text{B} = 128\,\text{KB}.$$

Each cache line holds $128/4 = 32$ **floats**, so the number of cache lines per array is

$$\#\text{lines per array} = \frac{128\,\text{KB}}{128\,\text{B}} = 1024.$$

There are 128 sets total, so each array contributes

$$\frac{1024}{128} = 8 \text{ lines per set.}$$

Because A and B map to the *same* sets, each set sees

$$8 \text{ (from A)} \;+\; 8 \text{ (from B)} \;=\; 16 \text{ candidate lines per set,}$$

while the cache can hold only 8 lines per set (8-way associativity).
The 128 sets partition blocks such that blocks A_0, A_{128}, A_{256}, $\ldots$, A_{896} (where A_i denotes the i^{th} cache block of A) lie in set 0, and so on. When block A_512 is accessed, it leads to a conflict miss. This happens for other sets as well. B also competes for this set in the similar way, and hence, blocks get evicted by the LRU replacement policy before reuse.

Stride 1

- Each loop iteration accesses all 1024 lines of A. With A and B together, 2048 lines are accessed per pass, which exceeds the cache size, hence by the time we loop around to the first line, it would be evicted.
- We will have a miss every time a line is accessed, with the first 1024 misses being *cold* misses. Thus, $\mathsf{misses} = 10,24,000$

Stride 16

- Similar to the previous case, each loop iteration accesses all 1024 lines of A, as each line has 32 elements. With A and B together, 2048 lines are accessed per pass, which exceeds the cache size, hence by the time we loop around to the first line, it would be evicted.
- We will have a miss every time a line is accessed, with the first 1024 misses being *cold* misses. Thus, $\mathsf{misses} = 10,24,000$

Stride 32

- Similar to the previous cases, each loop iteration accesses all 1024 lines of A, as each line has 32 elements. With A and B together, 2048 lines are accessed per pass, which exceeds the cache size, hence by the time we loop around to the first line, it would be evicted.
- We will have a miss every time a line is accessed, with the first 1024 misses being *cold* misses. Thus, $\mathsf{misses} = 10,24,000$

Stride 64

- 512 distinct lines of A are accessed per pass, but half the sets remain empty throughout as alternate lines are skipped. This implies that both A and B content for the same 512 sets.
- We will have a miss every time a line is accessed, with the first 512 misses being *cold* misses. Thus, $\mathsf{misses} = 5,12,000$

Stride 2K

- Only 16 distinct lines of A are accessed per pass, but these 16 lines map to 2 unique sets. This implies that both A and B content for the same 2 sets.
- We will have a miss every time a line is accessed, with the first 16 misses being *cold* misses. Thus, $\mathsf{misses} = 16,000$

Stride 8K

- Only 4 distinct lines of A are accessed per pass, and these 4 lines map to one set. This implies that both A and B content for the same set. Since each set can hold 8 lines, there will be no conflicts.
- We will have a miss only for the first time a line is accessed. Thus, $\mathsf{misses} = 4$

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

Table 1: Cache miss classification on array A for different strides.

Problem 2

Consider a cache of size **64K words** and lines of size **16 words**. The matrix dimensions are **1024** $\times$ **1024**. Perform cache miss analysis for the `kij` and the `jki` forms of matrix multiplication (shown below) considering direct-mapped and fully associative caches. The arrays are stored in row-major order. To simplify the analysis, ignore misses from cross-interference between elements of different arrays (i.e., perform the analysis for each array, ignoring accesses to the other arrays).

Listing 1: `kij` form

```c
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    r = A[i][k];
    for (j = 0; j < N; j++)
      C[i][j] += r * B[k][j];
```

Listing 2: `jki` form

```c
for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
    r = B[k][j];
    for (i = 0; i < N; i++)
      C[i][j] += A[i][k] * r;
```

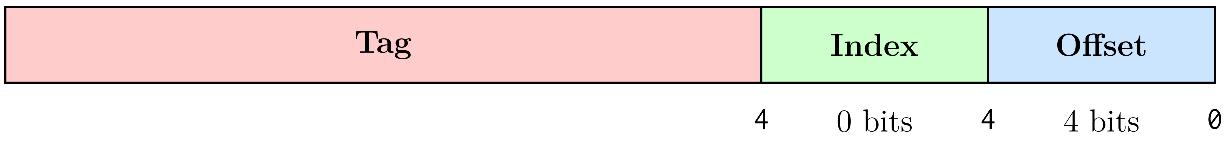
Your solution should have a table to summarize the total cache miss analysis for each loop nest variant and cache configuration, so there will be four tables in all. Briefly explain the result for each array for each loop nest (e.g., array `C` for variant `jki`).

Solution

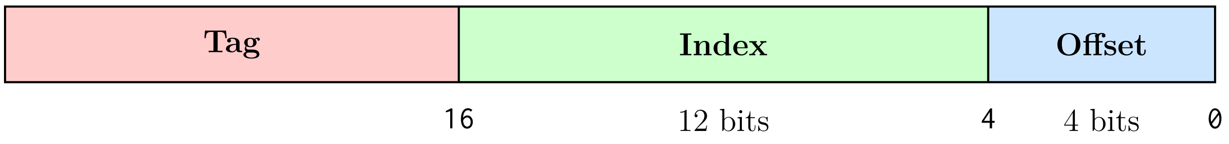
We are given a cache of capacity 64K *words* and a line size of 16 words. Thus,

$$\text{lines in cache} = \frac{64\text{K}}{16} = 4096 \text{ lines.}$$

- **Block offset bits**: $\log_2(16) = 4$.
- **Fully Associative Cache**: The physical byte address decomposes as the following.



- **Direct Mapped Cache**: The physical byte address decomposes as the following.



Matrices A, B, C are $N \times N$ with $N = 1024$, stored in row-major order.

$$|A| = |B| = |C| = N^2 \text{ words} = 1024^2 = 1{,}048{,}576 \text{ words.}$$

Each cache line holds $L = 16$ consecutive words. Hence:

$$\#\text{lines per row} = \frac{N}{L} = \frac{1024}{16} = 64, \qquad \#\text{lines per full matrix} = \frac{N^2}{L} = \frac{1024^2}{16} = 65{,}536.$$

`kij` order

Fully Associative Cache

- **`Array A`**
  - `Loop j` : There will be complete temporal reuse since the inner loop index extracts the same constant location from `A` stored in `r`, hence multiplier is 1.
  - `Loop i` : While traversing through this loop for a fixed k, an entire column of `A` is accessed. We load just one line for each row, there are misses every time a row is accessed, hence the multiplier is N
  - `Loop k` : While traversing through this loop, we can utilise spatial locality since one line per row is loaded into the cache which means BL elements per row are loaded, hence, the multiplier is $\frac{N}{BL}$
- **`Array B`**
  - `Loop j` : For a fixed k entire row is accessed and thus due to the array being stored in row major order, the multiplier is $\frac{N}{BL}$
  - `Loop i` : While iterating through this loop, we access the entire row of `B` for a fixed k, every time. Since this is a fully associative cache all the lines are still in memory, hence multiplier is 1.
  - `Loop k` : While traversing through this loop, we go through all columns of a row in `B`. Thus, there are misses every time, since an entirely new row is to be loaded, hence the multiplier is N
- **`Array C`**
  - `Loop j` : For a fixed i entire row is accessed and thus due to the array being stored in row major order, the multiplier is $\frac{N}{BL}$
  - `Loop i` : While traversing through this loop, we go through all columns of a row in `C`. Thus, there are misses every time, since an entirely new row is to be loaded, hence the multiplier is N
  - `Loop k` : There will no reuse since when k changes, the entire array needs to be traversed again, and since the cache can not contain the entire array, the multiplier is N.

Table 2: **kij** — Fully Associative

Loop	**A**	**B**	**C**
k	$\frac{N}{BL}$	N	N
i	N	1	N
j	1	$\frac{N}{BL}$	$\frac{N}{BL}$

Loop	**A**	**B**	**C**
k	64	1024	1024
i	1024	1	1024
j	1	64	64

Direct Mapped Cache

- **`Array A`**
  - `Loop j` : There will be complete temporal reuse since the inner loop index extracts the same constant location from `A` stored in `r`, hence multiplier is 1.
  - `Loop i` : While traversing through this loop for a fixed k, an entire column of `A` is accessed. We load just one line for each row, there are misses every time a row is accessed, hence the multiplier is N
  - `Loop k` : While traversing through this loop, we can not utilise spatial locality since the one line per row is that is loaded into the cache is evicted since we can only store 64 rows of `A` at a time, hence, the multiplier is N
- **`Array B`**
  - `Loop j` : For a fixed k entire row is accessed and thus due to the array being stored in row major order, the multiplier is $\frac{N}{BL}$
  - `Loop i` : While iterating through this loop, we access the entire row of `B` for a fixed k, every time. Since this is a fully associative cache all the lines are still in memory, hence multiplier is 1.
  - `Loop k` : While traversing through this loop, we go through all columns of a row in `BL`. Thus, there are misses every time, since an entirely new row is to be loaded, hence the multiplier is N
- **`Array C`**
  - `Loop j` : For a fixed i entire row is accessed and thus due to the array being stored in row major order, the multiplier is $\frac{N}{BL}$
  - `Loop i` : While traversing through this loop, we go through all columns of a row in `C`. Thus, there are misses every time, since an entirely new row is to be loaded, hence the multiplier is N
  - `Loop k` : There will no reuse since when k changes, the entire array needs to be traversed again, and since the cache can not contain the entire array, the multiplier is N.

Table 3: **kij** — Direct Mapped

Loop	**A**	**B**	**C**
k	N	N	N
i	N	1	N
j	1	$\frac{N}{BL}$	$\frac{N}{BL}$

Loop	**A**	**B**	**C**
k	1024	1024	1024
i	1024	1	1024
j	1	64	64

`jki` order

Fully Associative Cache

- **`Array A`**
  - `Loop i` : While traversing through this loop for a fixed j, k, an entire column of `A` is accessed. We load just one line for each row, there are misses every time, hence the multiplier is N
  - `Loop k` : While traversing through this loop, we can utilise spatial locality since one line per row is loaded into the cache which means BL elements per row are loaded, hence, the multiplier is $\frac{N}{BL}$
  - `Loop j` : There will no reuse since when j changes, the entire array needs to be traversed again, and since the cache can not fit the entire array, the multiplier is N.
- **`Array B`**
  - `Loop i` : There will be complete temporal reuse since the inner loop index extracts the same constant location from `B` stored in `r`, hence multiplier is 1.
  - `Loop k` : While traversing through this loop, a column of `B` is accessed. We load just one line for each row, there are misses every time, hence the multiplier is N
  - `Loop j` : While traversing through this loop, we can utilise spatial locality since one line per row is loaded into the cache which means BL elements per row are loaded, hence, the multiplier is $\frac{N}{BL}$
- **`Array C`**
  - `Loop i` : While traversing through this loop, we travel through the entire column. We load just one line for each row, there are misses every time, hence the multiplier is N
  - `Loop k` : While iterating through this loop, we access the entire row of `C` for a fixed k, every time. Since this is a fully associative cache all the lines are still in memory, hence multiplier is 1.
  - `Loop j` : While traversing through this loop, we can utilise spatial locality since one line per row is loaded into the cache which means BL elements per row are loaded, hence, the multiplier is $\frac{N}{BL}$

Table 4: **jki** — Fully Associative

Loop	**A**	**B**	**C**
j	N	$\frac{N}{BL}$	$\frac{N}{BL}$
k	$\frac{N}{BL}$	N	1
i	N	1	N

Loop	**A**	**B**	**C**
j	1024	64	64
k	64	1024	1
i	1024	1	1024

Direct Mapped Cache

- **`Array A`**
  - `Loop i` : While traversing through this loop, an entire column of `A` is accessed. We load just one line for each row, there are misses every time, hence the multiplier is N
  - `Loop k` : While traversing through this loop traversal needs to begin from the first row, however the cache will contain the last 64 rows of the column before. We need to load the data each time the row changes, thus there are misses every time, hence the multiplier is N
  - `Loop j` : There will no reuse since when k changes, the entire array needs to be traversed again, and since the cache can not fit the entire array, the multiplier is N.
- **`Array B`**
  - `Loop i` : There will be complete temporal reuse since the inner loop index extracts the same constant location from `B` stored in `r`, hence multiplier is 1.
  - `Loop k` : While traversing through this loop, a column of `B` is accessed. Thus, there are misses every time, hence the multiplier is N
  - `Loop j` : While traversing through this loop traversal needs to begin from the first row, however the cache can not contain all rows of one column. We need to load the data each time the row changes, thus there are misses every time, hence the multiplier is N
- **`Array C`**
  - `Loop i` : While traversing through this loop, an entire column of `C` is accessed. We load just one line for each row, there are misses every time, hence the multiplier is N
  - `Loop k` : While traversing through this loop traversal needs to begin from the first row, however the cache can not contain all rows of one column. We need to load the data each time the row changes, thus there are misses every time, hence the multiplier is N
  - `Loop j` : While traversing through this loop, a column of `C` is accessed. Thus, there are misses every time, hence the multiplier is N

Table 5: **jki** — Direct Mapped

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

Loop	**A**	**B**	**C**
j	1024	1024	1024
k	1024	1024	1024
i	1024	1	1024

Problem 3

Write a C++ program that takes the following arguments from the command line: the absolute path to an input file R, the number of producer threads T, the minimum number of lines each thread can read from the file $L_{\min}$, the maximum number of lines each thread can read from the file $L_{\max}$, the size of an intermediate shared buffer in lines M, and the path to an output file W.

Assume R contains N lines. The whole input file has to be read and can have more than $T \times L_{\max}$ lines. After all the T threads have been created, the threads will CONTEND for access to R to repeatedly read L consecutive lines, where L is random and $L_{\min} \leq L \leq L_{\max}$. Then, each thread will write its share of L consecutive lines ATOMICALLY to a FIFO shared buffer. There will be $\max\{1, \lfloor T/2 \rfloor\}$ consumer threads, where $T/2$ is integer division. The consumer threads will KEEP READING from the shared buffer and write their contents atomically to W.

Solution

Compilation

Run the following command from the root assignment submission folder.
`make problem3`

Usage

```
./pipeline <input_file R> <T> <Lmin> <Lmax> <M> <output_file W>
```

Data Structures

InputFile owns a shared `ifstream` and a mutex `input_file_lock` to serialize `getline` across producers.

OutputFile owns a shared `ofstream` and `output_file_lock`.

LineBuffer contains `queue<vector<string» q`, an integer `lines_in_buf` (current occupancy in lines), the fixed `capacity_lines = M`, flags, a mutex `buffer_lock`, and two condition variables:

- `cv_has_space`: producers wait until `lines_in_buf` + `want` $\leq M$.
- `cv_has_data`: consumers wait until `q` is non-empty, or all producers are done.

Counter holds `active_producers`, along with:

- `counter_lock`: protects `active_producers`.
- `producer_lock`: the *turn* mutex that enforces each producer to get L-block for the FIFO Buffer.

Synchronization Primitives & Invariants

- **Mutexes** provide mutual exclusion such that only one thread modifies a shared state.
  - `input_file_lock`: exclusive access to `ifstream::getline`.
  - `output_file_lock`: ensures chunk-level writes to W.
  - `buffer_lock`: protects `q`, `lines_in_buf`, and `producers_done`.
  - `producer_lock`: *turn* to preserve contiguity of each producer's L lines in the FIFO.
  - `counter_lock`: protects `active_producers`.
- **Condition variables** lets threads sleep efficiently while waiting for changes, and allows threads to notify when such changes occour.
  - `cv_has_space`: predicate (`lines_in_buf` + `want` $\leq M$).
  - `cv_has_data`: predicate (!`q.empty()`) $\vee$ `producers_done`.

Assumptions & Constraints

- Lines are delimited by `'\n'` and read with `std::getline`; blank lines are permitted.
- Buffer capacity is measured in *lines*; chunk size is at most M.
- Runtime parameters must satisfy: $T > 0$, $M > 0$, $L_{\text{min}} > 0$, and $L_{\text{max}} \geq L_{\text{min}}$.
- If EOF is hit mid-block, the partial block (non-empty) is still enqueued and written.