CS610 Assignment 2

Pragati Agrawal (220779)

September 2025

Problem 1

How to build and run

Either use the `Makefile` provided in `prob1-dir`, and run

```
make
```

to compile the code. Or from `prob1-dir/`:

```
mkdir -p bin
# Requires: sudo apt-get install libpapi-dev
g++ -O3 -std=c++17 220779-prob1.cpp -o ./bin/problem1.out -pthread -lpapi
```

To run,

```
./bin/problem1.out
```

(Wait for about 10 seconds for it to run completely, as there are **sleep(5)** instructions in between of papi measurements, to bring cache to fresh state and reduce interference of blocks before running blocked version of code.)

Implementation details

- **Blocked kernel** (`cc_3d_no_padding_blocked`): tile the output into BH × BW × BD tiles.
  - For each tile origin (i, j, k), iterate local coordinates (b_i, b_j, b_k) from $(0, 0, 0)$ to BH, BW, BD within the tile (edge tiles use `min` of remaining size and block config parameters to clip).
  - Access `input` and `result` using global offsets $(i + b_i, j + b_j, k + b_k)$ to perform convolution and store result.

  Both the naive and blocked versions of code are run 5 times, the average time is reported and the best performing blocking parameters are stored.
- **Autotuning**: explore BH, BW, BD $\in \{4, 8, 16, 32\}$ independently; for each candidate, ran the blocked kernel end-to-end and verified equality to the naive result to ensure correctness.
- **Timing hygiene**: before each timing, a cache flush routine is called `flush_cache()` that writes and then reads a 64 MiB buffer with a 64B stride (touching each cache block once) and issues a memory fence at the end to ensure all operations above it were completed. This ensures no elements are present in the cache due to previous runs.

- **PAPI integration**: after the best blocking configuration is found, PAPI measurements are performed for the naive and blocked versions. We create separate EventSets for naive and blocked kernels and measure `PAPI_L1_TCM` (L1 total cache misses) and `PAPI_L2_DCM` (L2 data cache misses) as only these were derivable in the machines. Flush caches and sleep briefly between measurements to avoid cross-run effects.

Results

I have used the machine `172.27.19.35` for this question. Across runs, the performance varies, specially in terms of speed and ideal blocking configuration. Below is the output of 2 runs:

Run 1 (q1.txt)

```
Average naive kernel time: 0.00279124 s
Average blocked kernel time: 0.00157056 s
Speedup: 1.77723x
Best block size : (4,16,32)
Starting PAPI measurements...
Naive L1 D-cache misses: 125799
Naive L2 D-cache misses: 157816
Blocked L1 D-cache misses: 115691
Blocked L2 D-cache misses: 162285
```

Run 2 (q11.txt)

```
Average naive kernel time: 0.00157148 s
Average blocked kernel time: 0.00157052 s
Speedup: 1.00061x
Best block size : (4,8,32)
Starting PAPI measurements...
Naive L1 D-cache misses: 125757
Naive L2 D-cache misses: 157597
Blocked L1 D-cache misses: 123112
Blocked L2 D-cache misses: 165122
```

Interpretation

The speedup observed varies across runs. These results are kindof 2 extremes. Blocking reduces L1 total cache misses by improving spatial/temporal locality within each 3D tile (inputs and corresponding outputs stay hot while computing the tile). The modest increase in L2 data misses is most probably due to write-allocate traffic for result tiles and capacity/conflict effects at tile boundaries from L1 to L2, specially if the L2 is exclusive. The best block size also varies across runs, but all these blocking configurations give almost the same time for all runs (just the min one may be different across runs).

Problem 2

Overview

Starting from the provided naive implementation (`prob2_naive.cpp`), I have systematically identified and fix these issues through two incremental versions: one that removes false sharing (`220779-prob2-false.cpp`) and another that additionally removes true sharing (`220779-prob2-true.cpp`).

How to build and run

Either use the `Makefile` provided in `prob2-dir`, and run

```
make
```

to compile the code. Or from `prob1-dir/`: From `source-files/`:

```
mkdir -p bin
g++ -O3 -std=c++17 generate_test_files.cpp -o ./bin/generate_test_files.out

# Compile versions
g++ -O3 -std=c++17 prob2_naive.cpp -o ./bin/prob2_naive.out -pthread
g++ -O3 -std=c++17 220779-prob2-false.cpp -o ./bin/prob2_false.out -pthread
g++ -O3 -std=c++17 220779-prob2-true.cpp -o ./bin/prob2_true.out -pthread
```

To run the codes and the perf c2c tool:

```
# Generate test files (5 files, ~4MB each)
./bin/generate_test_files.out

# Run performance analysis
./bin/prob2_naive.out 5 prob2-test1-files/input
./bin/prob2_false.out 5 prob2-test1-files/input
./bin/prob2_true.out 5 prob2-test1-files/input

# perf c2c analysis (if perf tools are available)
./run-prob2-script.sh ./bin/prob2_naive.out prob2-test1-files/input naive_c2c.out
./run-prob2-script.sh ./bin/prob2_false.out prob2-test1-files/input false_c2c.out
./run-prob2-script.sh ./bin/prob2_true.out prob2-test1-files/input optimized_c2c.out
```

Performance issues identified and fixes

Original naive version (`prob2_naive.cpp`)

The provided baseline implementation contained several critical performance bugs:

- **False sharing**: Per-thread counters stored in a compact array `word_count[4]` where adjacent elements share cache lines. When threads on different cores update their counters, they invalidate each other's cache lines even though they access different logical data.
- **True sharing with high contention**: Global totals (`total_words_processed`, `total_lines_processed`) updated under mutex protection *every iteration* in loops, creating severe lock contention is not required, can be done in a local variable and updated a bit later.

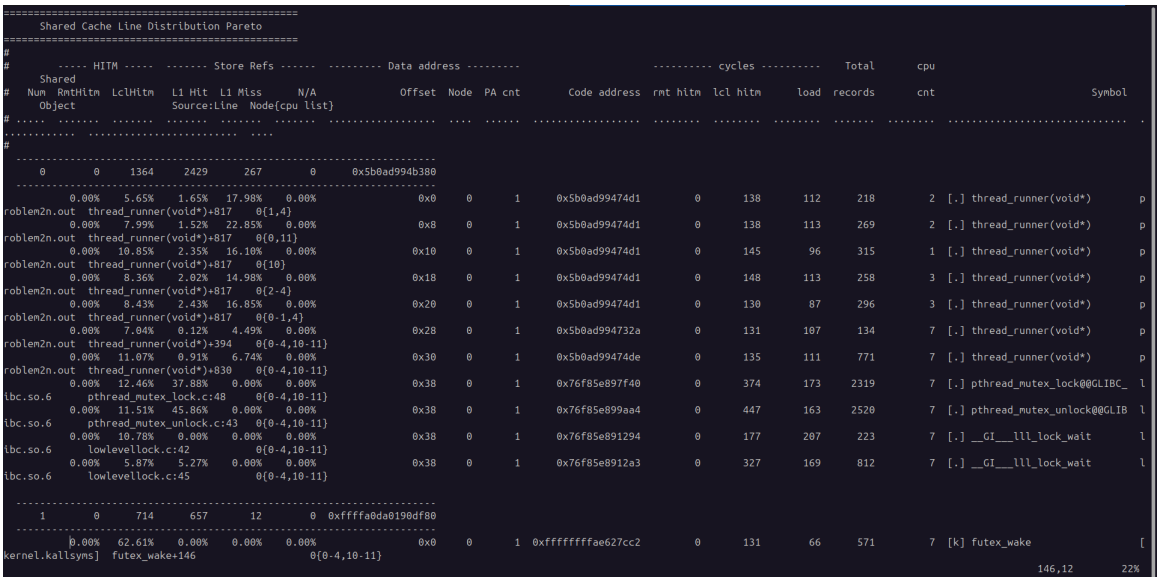

Figure 1: perf c2c report for naive implementation showing HITM events and cache line sharing

Fix 1: Remove false sharing (`220779-prob2-false.cpp`)

- **Cache-line padding**: Introduced `PaddedCounter` struct with `alignas(64)` to ensure each thread's counter occupies its own cache line:

```
struct alignas(CACHE_LINE) PaddedCounter {
  uint64_t value;
  char pad[CACHE_LINE - sizeof(uint64_t)];
};
```

- **Dynamic allocation**: Replaced fixed-size array with dynamically allocated `PaddedCounter*` sized to the actual thread count, avoiding global arrays sized by runtime parameters.
- **Separate alignment**: Used `alignas(CACHE_LINE)` for global totals to isolate them from other frequently accessed data.

Fix 2: Remove true sharing (`220779-prob2-true.cpp`)

- **Local aggregation**: Introduced per-thread local accumulators (`local_words_processed`, `local_lines_processed`) to eliminate frequent updates to shared global totals in hot loops.
- **Batch updates**: Moved global total updates outside the processing loops; each thread updates shared totals exactly once at completion under mutex protection. This also causes less contention to mutexes.

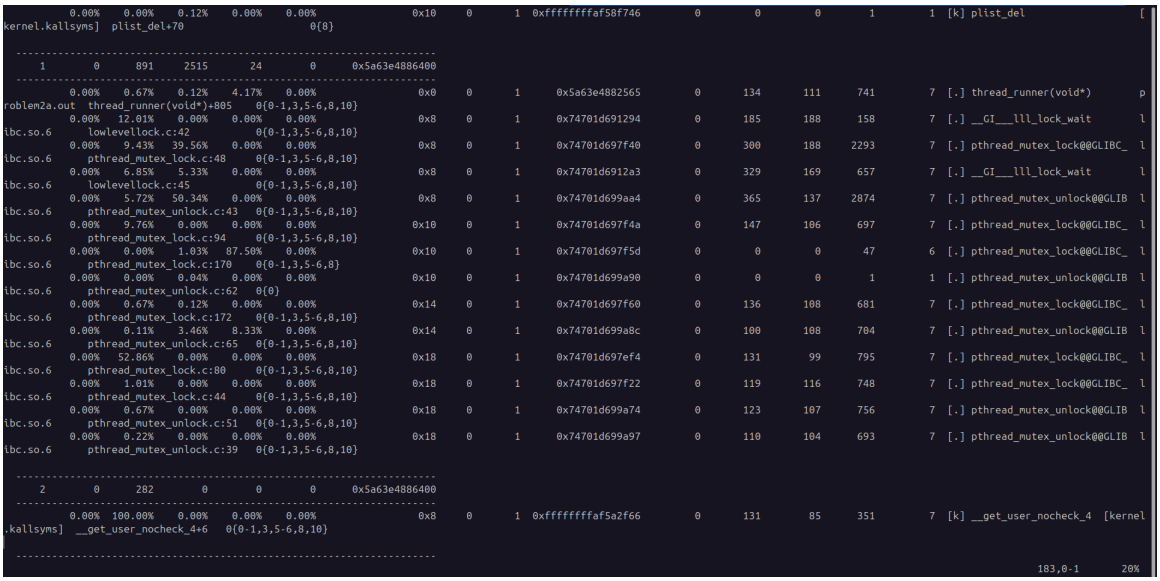

Figure 2: perf c2c report after fixing false sharing - reduced HITM events

Validation with perf c2c

- **Naive version**: Show significant HITM events in the `thread_runner` function, due to `word_count` array and global total variables, indicating frequent cache line bouncing between cores.
- **Optimized version**: Demonstrates reduced HITM events, confirming that the padding and local aggregation techniques successfully mitigated cache coherency overhead. The implementation that only mitigates false sharing (`220779-prob2-false.cpp`) has less HITMS, while the one also removing true sharing (`220779-prob2-true.cpp`) has 0 HITMS.

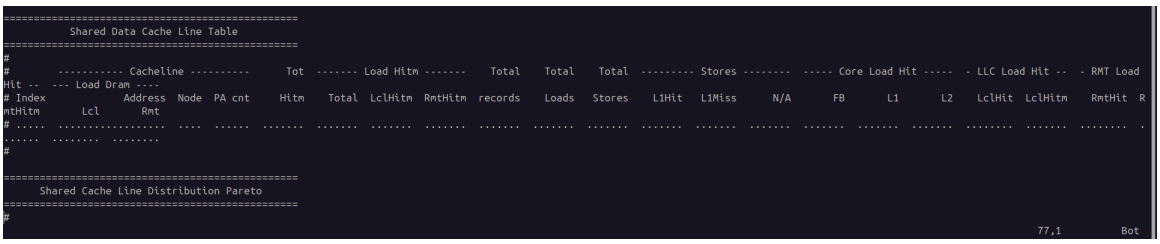

Figure 3: perf c2c report after fixing both false and true sharing - minimal/zero HITM events

Problem 3

How to build and run

All paths below are relative to `source-files/`.

- To compile and run for a given thread count T:

```
mkdir -p bin
g++ -O3 -std=c++17 -mavx -mavx2 220779-prob3.cpp -o ./bin/problem3.out -pthread
./bin/problem3.out T
```

- Batch runs for powers of two (1,2,4,8,16,32,64) using the helper script:

```
chmod +x run-prob3-script.sh
./run-prob3-script.sh output-prob3.txt
```

  The script builds `./bin/problem3.out` and appends results for each thread count to `output-prob3.txt`.

Machine Used

I have used machines `172.27.19.34` and `172.27.19.36` for running my programs (both have 12 cores). For larger number of threads (16,32,64), I could take measurements for smaller $N = 1e5$, as the program takes a very long time to run.

Implementation details

Filter lock Arrays `level[]` and `victim[]` (both padded) coordinate entry through levels $1..T-1$. A thread at level L announces itself as `victim[L]` and waits while any other has `level` $\geq L$ and it remains the victim. Starvation-free; higher per-acquire overhead and $O(T)$ reads in the wait condition.

Bakery lock (Lamport) Per-thread `choosing[]` and ticket `number[]` arrays (both padded). Acquire: set `choosing=true`, pick `number=1+max(number)`, set `choosing=false`, then wait for all lower lexicographic (number, id) pairs. Release sets `number=0`.

Spin lock A single volatile integer guarded by `cmpxchg`. Acquire repeatedly attempts to change $0 \to 1$; release writes 0. Uses `pause` in the spin loop to lower energy consumption and pressure on CPU.

Ticket lock Two padded counters: `next_ticket` and `now_serving`. Acquire does an atomic fetch-and-increment on `next_ticket` to get *my_ticket*, then spins until `now_serving==my_ticket`. Release increments `now_serving`.

Array-based Queue lock Each thread enqueues by atomically incrementing a padded `tail` and takes a slot index; threads spin on a per-slot flag in a preallocated, padded `flags[]` array, avoiding false sharing. Release clears its flag and sets the next slot's flag.

Performance Optimisations:

- Avoiding false sharing: per-thread metadata and shared variables use 64B cache-line alignment via `alignas(64)` and padded structs (e.g., `PaddedInt`, `PaddedUint32/64` etc).
- Atomic/CAS primitives: x86-64 inline `cmpxchg` is used to build a CAS-based `fetch_add`; spin loops use `pause` to reduce pipeline pressure.
- Fairness: Ticket and Array-Q locks provide FIFO progress; Filter and Bakery are starvation-free but higher-overhead; TAS spin lock is unfair.

Results

For 16 and 32 threads, result is for N=1e5, as this itself took more than 2 hours to run. For 64 threads, the result is for N=1e4, because at N=1e5 it run for more than 5 hours, and for 1e4 also for more than 2 hours.

```
=== Problem 3 Lock Performance Benchmark ===
=====================================
THREAD COUNT: 1
=====================================
Var1: 10000000 Var2: 1
Pthread mutex: Time taken (us): 123
Var1: 10000000 Var2: 1
Filter lock: Time taken (us): 181
Var1: 10000000 Var2: 1
Bakery lock: Time taken (us): 647
Var1: 10000000 Var2: 1
Spin lock: Time taken (us): 498
Var1: 10000000 Var2: 1
Ticket lock: Time taken (us): 392
Var1: 10000000 Var2: 1
Array Q lock: Time taken (us): 346

=====================================
THREAD COUNT: 2
=====================================
Var1: 20000000 Var2: 1
Pthread mutex: Time taken (us): 2018
Var1: 20000000 Var2: 1
Filter lock: Time taken (us): 5210
Var1: 20000000 Var2: 1
Bakery lock: Time taken (us): 5418
Var1: 20000000 Var2: 1
Spin lock: Time taken (us): 4542
Var1: 20000000 Var2: 1
Ticket lock: Time taken (us): 3206
Var1: 20000000 Var2: 1
Array Q lock: Time taken (us): 4320

=====================================
```

```
THREAD COUNT: 4
=====================================
Var1: 40000000 Var2: 1
Pthread mutex: Time taken (us): 6822
Var1: 40000000 Var2: 1
Filter lock: Time taken (us): 31993
Var1: 40000000 Var2: 1
Bakery lock: Time taken (us): 19714
Var1: 40000000 Var2: 1
Spin lock: Time taken (us): 31142
Var1: 40000000 Var2: 1
Ticket lock: Time taken (us): 13614
Var1: 40000000 Var2: 1
Array Q lock: Time taken (us): 16540

=====================================
THREAD COUNT: 8
=====================================
Var1: 80000000 Var2: 1
Pthread mutex: Time taken (us): 50767
Var1: 80000000 Var2: 1
Filter lock: Time taken (us): 234871
Var1: 80000000 Var2: 1
Bakery lock: Time taken (us): 87428
Var1: 80000000 Var2: 1
Spin lock: Time taken (us): 139055
Var1: 80000000 Var2: 1
Ticket lock: Time taken (us): 60128
Var1: 80000000 Var2: 1
Array Q lock: Time taken (us): 69184

=====================================
THREAD COUNT: 16
=====================================
Var1: 1600000 Var2: 1
Pthread mutex: Time taken (us): 3525
Var1: 1600000 Var2: 1
Filter lock: Time taken (us): 27199
Var1: 1600000 Var2: 1
Bakery lock: Time taken (us): 21166116
Var1: 1600000 Var2: 1
Spin lock: Time taken (us): 7275
Var1: 1600000 Var2: 1
Ticket lock: Time taken (us): 20750290
Var1: 1600000 Var2: 1
Array Q lock: Time taken (us): 21213110

=====================================
```

```
THREAD COUNT: 32
=====================================
Var1: 3200000 Var2: 1
Pthread mutex: Time taken (us): 12992
Var1: 3200000 Var2: 1
Filter lock: Time taken (us): 4039208
Var1: 3200000 Var2: 1
Bakery lock: Time taken (us): 273086618
Var1: 3200000 Var2: 1
Spin lock: Time taken (us): 74704
Var1: 3200000 Var2: 1
Ticket lock: Time taken (us): 270134510
Var1: 3200000 Var2: 1
Array Q lock: Time taken (us): 270577512

=====================================
THREAD COUNT: 64
=====================================
Var1: 640000 Var2: 1
Pthread mutex: Time taken (us): 5684
Var1: 640000 Var2: 1
Filter lock: Time taken (us): 45735383
Var1: 640000 Var2: 1
Bakery lock: Time taken (us): 287750225
Var1: 640000 Var2: 1
Spin lock: Time taken (us): 35738
Var1: 640000 Var2: 1
Ticket lock: Time taken (us): 288829060
Var1: 640000 Var2: 1
Array Q lock: Time taken (us): 522205283
=====================================
=====================================
```

The average time taken per thread (in minutes), is reported in the table:

	1	2	4	8	16	32	64
Pthread mutex	0.002050	0.016817	0.028425	0.105765	0.003672	0.006767	0.001480
Filter lock	0.003017	0.043417	0.133304	0.489315	0.028332	2.103754	11.910256
Bakery lock	0.010783	0.045150	0.082142	0.182142	22.048037	142.232614	74.934954
Spin lock	0.008300	0.037850	0.129758	0.289698	0.007578	0.038908	0.009307
Ticket lock	0.006533	0.026717	0.056725	0.125267	21.614885	140.695057	75.215901
Array Q lock	0.005767	0.036000	0.068917	0.144133	22.096990	140.925788	135.990959

Observations

- For a given thread count, the general trend of performance is `pthread > spin > filter > bakery = ticket = array queue`.
- For smaller thread counts (1–4, array size 10^7), all locks complete quickly, with `pthread mutex` giving the best performance. Filter and bakery locks already show higher overhead compared to mutex and spin/ticket locks.
- At 8 threads (array size 10^7), execution times rise significantly, especially for filter and spin locks, while `pthread mutex` remains the most efficient.
- For higher thread counts (16–32, array size 10^5), even with reduced input size, bakery, ticket, and array Q locks degrade drastically in performance, whereas mutex scales much better.
- At very high thread count (64, array size 10^4), custom locks (filter, bakery, ticket, array Q) take extremely long times despite reduced workload. Spin lock remains relatively stable, but mutex still provides the lowest per-thread overhead.
- Overall, `pthread mutex` consistently scales best, while software-based locks suffer from contention and coordination overhead, making them unsuitable beyond small thread counts.