CS610 : Programming For Performance

Assignment 1

Submitted By : Kushagra Srivastava, 220573

Problem 1

1. Write a C++ program to implement a naïve 3D convolution operation. The naïve function should implement the following computation:

$$O[i][j][k] = \sum_{u=0}^{K-1} \sum_{v=0}^{K-1} \sum_{w=0}^{K-1} I[i+u][j+v][k+w] \times F[u][v][w]$$

   where I is a $N \times N \times N$ input array and F is the $K \times K \times K$ filter (or kernel). O is the 3D output array (or feature map), and each dimension of O is of size $N - K + 1$ (no padding). Ignore flipping the filter.

2. Implement a blocked version of the convolution operation. You should focus on blocking the input and output arrays, and size the blocks for private L2 caches. Make the block size configurable because you will evaluate different sizes.

   (a) Experiment with different block sizes (individual dimensions can be different) and report the block size that gives the best performance for the blocked kernel. Automatically exploring different block sizes and zeroing in on the best setup is called **autotuning**.

   (b) Run the naïve and the blocked kernels at least five times and report the average times taken.

   (c) Use the **PAPI** library to estimate the number of cache misses for the data cache hierarchy on your target architecture to justify the performance that you observed.

Solution

We implement a 3D cross-correlation (convolution without flipping) on a cubic input $I \in R^{N \times N \times N}$ with a cubic filter $F \in R^{K \times K \times K}$ to produce $O \in R^{(N-K+1) \times (N-K+1) \times (N-K+1)}$.

Implementation

All arrays are stored as contiguous row-major 1D buffers. The fastest-varying index is the depth (k) so the innermost tap loop can auto-vectorize.

We tile the *output* space into blocks (B_i, B_j, B_k) and compute one output tile at a time. Each tile reuses an *input halo* of size $(B_i + K - 1) \times (B_j + K - 1) \times (B_k + K - 1)$. This improves locality.

Loop order keeps k (depth) innermost to maximize unit-stride memory access and help the compiler vectorize the small inner reduction.

Tuning

We explore a small candidate set $B_i, B_j, B_k \in \{4, 6, 8, 10, 12, 16, 20, 24, 28, 32\}$. The best block is the one with minimum measured time among feasible candidates. L2 size can be checked via `L2KB`.

Cache Flushing)

To ensure comparable *cold* timings, before every timed run we traverse a buffer larger than LLC (default 64MiB, configurable via `LLCKB`) with a read pass and a write pass. This reliably evicts prior data without requiring privileged instructions.

Setup

- Problem size used in code: $N = 128$, $K = 3$.
- L2 Size = $512KB$
- In order to compile the code

```
make
g++ -O3 -std=c++17 -DUSE_PAPI -mavx -mavx2 220573-prob1.cpp -lpapi -o 220573-prob1
```

Kernel	Time (ms)	GOPS	PAPI_TOT_CYC	PAPI_L1_DCM	PAPI_L2_DCM
Naïve	11.97 ms	9.54		1028662	2222323
Blocked	12.31 ms	9.60		1173531	1572311

Table 1: Values are averages over 5 runs.

As evident, on using blocking we are able to drastically reduce the no. of misses to L2 cache without severely impacting the performance at the L1 level.

Tuning

The best performance was achieved for a block size of $(6, 6, 32)$, which is smaller than the `L1 CACHE SIZE`. The time taken by this code is **`10.94 ms`**, which is even better than the naïve version

Problem 2

You are given a multithreaded program with N threads. The program reads N files, and should report the total number of words and lines processed by all the threads.

We provide a driver source code for the problem. Your task is to analyze the source code, and identify and report the performance bugs present in the source code, if any. If a performance bug is identified, provide a manually fixed version. Describe your modifications to the source code and report the performance gain.

Use the following commands to compile the attached driver and run the sample test case:

Listing 1: Compiling and running Problem 2 driver

```
make
./bin/problem2.out 5 ./prob2-test1/input
```

Solution

The bugs that I have fixed are,

- Count locally inside each thread (words lines), then do batched atomic adds. Removes the per-token critical section entirely.
- Remove false sharing on per-thread counters by putting each per-thread counter on its own cache line with alignas(64).
- Make per-thread counters sized at runtime. Replace `word_count[4]` with a cache-line-padded std::vector, sized to `thread_count`.

Table 2: Time in microseconds as on `csews29`

Threads	**2**		**4**		**8**		**16**	
	Original	Modified	Original	Modified	Original	Modified	Original	Modified
1	5742	3158	6674	3725	27082	7150	62482	3326
2	2284	3130	7088	3721	25084	3273	65149	4693
3	2572	3115	7438	3425	28404	4054	66980	3326
4	3156	3071	7207	3912	28588	3115	65281	7221
5	2831	3074	11318	4118	28051	5042	63370	3537
Avg	3317.0	3109.6	7945.0	3780.2	27441.8	4526.8	64652.4	4420.6

Problem 3

Implement the following locking strategies in C++: Filter lock, Bakery lock, Spin lock, Ticket lock, and Array-based Queue lock. You will evaluate the performance of your lock implementations using the following code snippet.

Listing 2: Parallel loop with lock acquisition

```
// The following loop nest will be executed in parallel
for (int i = 0; i < N; i++) {
    acquire(&lock);
    // The following variables are shared
    var1++;
    var2--;
    release(&lock);
}
```

You should use CAS-like APIs available on `x86_64` architectures to implement your version of the locks. Do not use lock definitions from existing libraries. Furthermore, you should try to avoid false sharing for implementations that make use of arrays.

Evaluate the above code snippet with 1–64 concurrent threads, in powers of two. Report the total time taken while using each lock type while varying the number of threads. Compare the performance of your lock implementations with `pthread_mutex_t` lock and unlock APIs as the baseline.

We have included a template C++ file that you will extend. Create separate classes for the above lock types, similar to `PthreadMutex`, and fill in the public `acquire()` and `release()` methods. Include all the definitions in the provided `problem3.cpp` file.

Fill in the following table with your results. Report any trends that you observe from the results.

Solution

Low-level primitives.

We avoid library atomics in the lock bodies and use x86 instructions with the `lock` prefix for atomicity:

- **CAS (compare-and-swap)**:
  - Implemented via `lock; cmpxchgb` (byte CAS) with `sete` to record success.
  - Semantics: atomically compare `*addr` with `expected`; if equal, write `desired` and return true, else return false. Used in the Spinlock.
- **FAI (fetch-and-increment)**:
  - Implemented via `lock; xaddq`. We pass an initial register value of 1; `xadd` returns the *old* value and increments memory by 1.
  - Used to obtain a FIFO ticket in the Ticket lock and to advance indices in the Array-Q lock.
- **Add-and-fetch (increment-and-read)**:
  - Also via `lock; xaddq`, but we add 1 and then return *old + 1* to get the new value. Used to advance the `owner` in the Ticket lock.

`cpu_relax()` (PAUSE).

Busy-wait loops call `cpu_relax()` which emits the x86 `pause` instruction. This reduces power consumption and pipeline pressure during wait. On non-x86 we fall back to a compiler-only barrier (`"" ::: "memory"`) to prevent the loop from being optimized away.

`compile_barrier()` Memory fence

This function issues **`mfence`**. Despite the name, this is a *full hardware memory fence*:

- The `"memory"` clobber prevents the compiler from reordering memory ops across the call.
- `mfence` enforces that all loads/stores before the fence become globally visible before any loads/stores after the fence.

Lock algorithms.

- **Pthread Mutex (baseline)**
  - Uses the standard `pthread_mutex_t` from POSIX.
- **Spin Lock**
  - Lock variable is a single boolean flag.
  - Threads spin locally, repeatedly reading the flag. Includes `cpu_relax()` to reduce pipeline pressure and improve fairness on hyperthreaded cores.
- **Ticket Lock**
  - Two counters: `next` (assigns a ticket using fetch-and-increment) and `owner`.
  - Each thread spins waiting for its ticket to match `owner`.
- **Array Q Lock**
  - Pre-allocates an array of boolean flags, one slot per thread.
  - Each arriving thread takes a slot index (via fetch-and-increment) and spins on its own flag.
- **Filter Lock**
  - Implements a k-ary tree of levels. Each thread moves through levels $1 \ldots (T-1)$.
  - At each level, thread declares itself as `victim[L]` and waits until no other thread at same/higher level is contending.
- **Bakery Lock (Lamport's algorithm)**
  - Each thread selects a ticket number: one greater than the maximum seen.
  - Uses lexicographic ordering (`number`, `tid`) for entry into the critical section.
  - Mutual exclusion without atomic RMWs — relies on coherent reads/writes.

Lock Type	1	2	4	8	16	32	64
Pthread mutex	5	24	87	261	3724	4242	25412
Filter lock	3	70	292	2376	37601	5298717	TIMEOUT
Bakery lock	8	52	263	1152	21355530	TIMEOUT	TIMEOUT
Spin lock	4	16	52	243	1918	6514	98740
Ticket lock	2	46	175	736	TIMEOUT	TIMEOUT	TIMEOUT
Array Q lock	3	64	260	976	3678	14425	56051

Table 3: Execution time (μs) of different lock implementations across varying thread counts for $N = 10^5$ on `csews38`

Lock Type	1	2	4	8	16	32	64
Pthread mutex	123	2265	7201	54715	242887	1587050	2310062 (csews29)
Filter lock	314	5970	28059	221167	117227980	TIMEOUT	TIMEOUT
Bakery lock	740	5430	25656	117087	TIMEOUT	TIMEOUT	TIMEOUT
Spin lock	359	1542	7236	25066	107547	639771	9323175 (csews26)
Ticket lock	242	4488	17992	71600	TIMEOUT	TIMEOUT	TIMEOUT
Array Q lock	349	6510	25896	100760	373907	1556860	5994053 (csews29)

Table 4: Execution time (μs) of different lock implementations across varying thread counts for $N = 10^7$ on `csews38`