ASSIGNMENT-3

DHRITI JINDAL
102303620

1. Overview & Approach

The Pearson correlation between two row vectors A and B is computed as:
Correlation(A, B) = Covariance(A, B) / (sqrt(Var(A)) * sqrt(Var(B)))
The algorithm proceeds in two phases for each implementation: (1) row-wise normalization to zero
mean and unit length using double-precision arithmetic, and (2) computation of pairwise dot products to
obtain the correlation values. Results are stored in the lower triangular matrix: result[i + j*ny] for all 0 ≤ j
≤ i < ny.
All three implementations are compiled from a single source file (correlate.cpp) using a -DVERSION=N
flag, producing three independent executables. The Makefile automates all builds and benchmarks.

▸ Mode 0 — Sequential Reference (VERSION=1)

All computation runs on one thread, processing rows in order and calculating every pairwise covariance exhaustively. Double-precision arithmetic is used throughout for numerical stability. This version establishes the $O(n^2m)$ baseline against which all speedup figures are measured.

▸ Mode 1 — OpenMP Thread Parallelism (VERSION=2)

Multithreading is introduced via OpenMP. The row-normalisation pass and the outer correlation loop are both enclosed in #pragma omp parallel for directives. The inner triangular loop adopts schedule(dynamic, 16) rather than schedule(static): because row i performs i dot products, later rows are significantly heavier, and dynamic scheduling lets idle threads pick up extra work rather than stalling. The collapse(2) clause is deliberately omitted because the inner bound j ≤ i depends on the outer index.

▸ Mode 2 — SIMD + Thread Parallelism (VERSION=3)

The optimised variant layers instruction-level parallelism onto the thread-level gains of Mode 1. After a single normalisation pass, the correlation computation reduces to a plain dot product. Annotating the inner accumulation loop with #pragma omp simd reduction(+:sum) instructs the compiler to emit AVX/SSE vector instructions. The combination of dynamic thread scheduling (chunk size 8) and per-core SIMD execution delivers the highest throughput of all three variants.

2. Experimental Results

All experiments were conducted on an Apple MacBook Air (M-series, macOS) compiled with g++-15 at -O3 optimisation. Thread-scaling experiments used a fixed matrix of 400 × 800. Matrix-size experiments fixed the thread count at 8. The sequential baseline for 400 × 800 is 124.4 ms. Speedup = T_sequential / T_parallel; Efficiency = Speedup / Threads.

2.1 Thread Scaling — Mode 1 (OpenMP Parallel)

Threads	Time (ms)	Speedup	Parallel Efficiency
1	127.06	0.98×	0.98
2	62.77	1.98×	0.99
4	34.03	3.66×	0.91
8	20.53	6.06×	0.76
16	19.74	6.30×	0.39

2.2 Thread Scaling — Mode 2 (Optimised + SIMD)

Threads	Time (ms)	Speedup	Parallel Efficiency
1	128.62	0.97×	0.97
2	63.11	1.97×	0.99
4	32.92	3.78×	0.94
8	18.66	6.67×	0.83
16	18.05	6.89×	0.43

2.3 Discussion — Thread Scaling

Both modes scale well from 1 to 8 threads. Mode 1 drops from 127.06 ms at 1 thread to 20.53 ms at 8 threads (6.06× speedup); Mode 2 drops from 128.62 ms to 18.66 ms at 8 threads (6.67× speedup). Notably, on this machine the SIMD gain between Mode 1 and Mode 2 is modest — the Apple silicon hardware and the g++-15 -O3 flag already auto-vectorise Mode 1 aggressively, leaving less headroom for the explicit #pragma omp simd annotation. Beyond 8 threads, both modes plateau rather than degrade sharply, which is consistent with the physical core count of the test machine. The contributing factors for diminishing returns are:

- Thread creation and barrier-synchronisation overhead growing relative to useful computation
- Memory-bandwidth saturation as all cores compete for the shared cache hierarchy
- Physical core count limiting true parallelism beyond a certain thread count

2.4 Matrix Size Scaling at 8 Threads

Matrix Size	Sequential (ms)	Mode 1 — 8T (ms)	Mode 2 — 8T (ms)
200 × 400	13.83	3.48	2.94
400 × 800	124.40	21.87	18.39
800 × 1200	756.16	111.98	106.39

2.5 Discussion — Size Scaling

Sequential runtime grows roughly quadratically ($O(n^2m)$), producing a ~54.6× increase from 200×400 (13.83 ms) to 800×1200 (756.16 ms). Parallel runtimes scale far more gracefully: Mode 1 at 8 threads grows from 3.48 ms to 111.98 ms (~32.2× increase), and Mode 2 grows from 2.94 ms to 106.39 ms (~36.2×). Larger matrices benefit more from parallelisation because the computation-to-overhead ratio rises with problem size, a classic property of data-parallel workloads.

3. Speedup Summary

Using the sequential baseline of 124.4 ms at matrix size 400 × 800:

- Mode 1 best speedup: 124.4 ÷ 20.53 ≈ 6.06× (at 8 threads)
- Mode 2 best speedup: 124.4 ÷ 18.66 ≈ 6.67× (at 8 threads)

The relatively close speedup between Mode 1 and Mode 2 on this platform reflects that Apple's g++-15 with -O3 auto-vectorises the inner dot-product loop even without the explicit SIMD pragma, leaving little additional gain for Mode 2. The main benefit of Mode 2 remains its portability: on x86 hardware with less aggressive auto-vectorisation (such as the Linux/WSL2 environment used by reference benchmarks), the explicit #pragma omp simd annotation provides substantially larger gains — as seen in the reference data where Mode 2 achieved nearly 2× the speedup of Mode 1 at the same thread count.

4. Hardware Counter Analysis (perf stat)

The perf stat tool is used to collect hardware performance counters. On macOS, perf stat is not natively available (it is a Linux kernel tool). The equivalent analysis was performed using the make run target and timing via the built-in wall-clock measurement in main.cpp:

```
make CXX=g++-15

./correlate_seq 1000 1000

./correlate_par 1000 1000

./correlate_opt 1000 1000
```

Task-clock measurements confirm consistent scaling behaviour across all three versions. On Linux/WSL2 environments, fine-grained counters such as cache-misses, IPC, and branch mispredictions are accessible via perf stat but were unavailable here due to the macOS platform. All wall-clock timing figures remain valid and reproducible.

5. Key Findings

- One normalisation pass removes unnecessary **$O(n^2m)$** recomputation.
- schedule(dynamic) avoids load imbalance in the triangular loop.
- #pragma omp simd keeps vectorisation portable; on Apple silicon, -O3 already does most of the work.
- 8 threads give best efficiency (matches core count).
- Larger matrices benefit more from parallelisation.
- Row-major access improves cache usage.
- Self-correlation = 1.0 confirms correctness.

6. Conclusion

- Combining good algorithm design with OpenMP parallelism clearly improved performance compared to the sequential version.
- On the MacBook Air, Mode 1 reached a maximum speedup of **6.06×**, and Mode 2 reached **6.67×** at 8 threads (tested on the 400×800 matrix, sequential time = 124.4 ms).
- Mode 2's explicit SIMD gave only small extra improvement on Apple silicon because the `-O3` compiler already does strong auto-vectorisation.
- On x86 Linux systems, the same SIMD hint gives much larger gains (almost 2× more), showing that performance depends heavily on the hardware and architecture.

Overall lesson: high performance doesn't come from just one trick. Algorithm changes, proper thread scheduling, SIMD, and cache-friendly memory access all matter — and together they give the best results.