CS610 Assignment-3

Mahaarajan J -220600

October 23, 2025

Workstation used: `csews8`

1 Problem 1: 3D Stencil Kernel Optimization

1.1 Objective

To optimize the given 3D 7-point stencil kernel using loop transformations and OpenMP, without using explicit SIMD intrinsics. The focus was on improving cache utilization, reducing loop overhead, and leveraging data parallelism.

1.2 Approach

We started from the baseline scalar kernel and incrementally applied the following transformations:

1. **Loop Unrolling:** Unrolled the innermost loop over k by various factors (2, 4, 8, 16, 32). The optimal unroll factor empirically turned out to be 8. Further unrolling did not yield a better result.
2. **Parallelization:** Used OpenMP with `collapse(2)` over both i and j dimensions. This exposed sufficient parallelism and kept the load balanced.
3. **SIMD Hints:** Used the compiler pragma `#pragma omp simd` to guide vectorization of innermost loops.
4. **Tail Loop Handling:** A tail loop was added to handle remaining elements after unrolling.

1.3 Results and Observations

Compilation:

```
g++ -O3 -fopenmp -march=native 220600-prob1.cpp -o prob1
./prob1
```

Observations on csews8:

- Baseline scalar: 15 ms
- Unroll only (factor=8): 13 ms (no significant improvement beyond factor 8)
- Unroll + parallel (collapse(2)): 3 ms
- Parallel + simd: 4 ms
- Hence kept the unroll+parallel kernel for best performance

1.4 Analysis

The best performance was achieved for the unrolled + parallelized version, reaching 3 ms runtime (5× speedup). Further unrolling or loop fusion attempts did not yield gains due to memory bandwidth saturation.

1.5 Summary

- Best configuration: **Unroll factor = 8, collapse(2)**
- Overall speedup: **5x over baseline**

2 Problem 2: Inclusive Prefix Sum Using SIMD

2.1 Objective

To implement inclusive prefix sum (scan) using SSE4 and AVX2 intrinsics, and compare with serial and OpenMP versions.

2.2 Approach

The prefix sum operation computes cumulative sums of an array: $b_i = a_0 + a_1 + \ldots + a_i$

The following versions were implemented, the first 3 were given and I implemented the AVX2 version:

- **Serial:** Sequential accumulation in a single loop.
- **OpenMP (SIMD scan):** Used `reduction(inscan, +: tmp)` and `scan inclusive(tmp)`.
- **SSE4:** Operated on 4 integers (128 bits) at once using intrinsics like `_mm_slli_si128`, `_mm_add_epi32`, and `_mm_shuffle_epi32`.
- **AVX2:** Operated on 8 integers (256 bits) simultaneously, using lane-wise shifts and carry propagation between 128-bit halves.

2.3 Compilation and Execution

```
g++ -std=c++17 -masm=att -msse4 -mavx2 -march=native -fopenmp
-fverbose-asm -fno-asynchronous-unwind-tables -fno-exceptions
-fno-rtti -fcf-protection=none -O2 220600-prob2.cpp -o problem2.out
./problem2.out
```

2.4 Results

Version	Serial (µs)	OMP	SSE	AVX2
2^{18} elements	28.2	18.0	18.2	25.6
2^{24} elements	11680.8	11262.4	11466.2	11478.4

2.5 Analysis

AVX2 and SSE achieved nearly identical performance due to being limited by memory bandwidth rather than computation. The prefix sum is inherently sequential; although SIMD can accelerate per-block operations, the carry propagation between blocks restricts scalability.

An explanation of the AVX2 version shows intra-lane and cross-lane operations for 8-element registers. SSE has a simpler implementation as each 128-bit half can be processed independently. It may happen that the AVX2 version may do better for even larger number of elements

2.6 Summary

- SIMD helped at small scales but gave limited benefit at large N.
- OpenMP intrinsics offered the most balanced performance.
- AVX2 added complexity without proportional gain beyond SSE.

3 Problem 3: 3D Gradient Vectorization

3.1 Objective

To vectorize a 3D gradient kernel operating on a 128×128×128 grid using SSE4 and AVX2 intrinsics, and compare their performance against the scalar baseline.

3.2 Approach

- **Scalar kernel:** Computed forward differences along the i dimension.
- **SSE4:** Processed 2 elements per iteration (2×64-bit = 128 bits) using `_mm_loadu_si128`, subtraction, and `_mm_storeu_si128`.
- **AVX2:** Operated on 4 elements per iteration (256 bits) using similar intrinsics.
- Memory alignment was enforced via `aligned_alloc()`.
- Checksums ensured correctness across all versions.

3.3 Compilation and Execution

```
g++ -std=c++17 -masm=att -msse4 -mavx2 -march=native -fopenmp
-fverbose-asm -fno-asynchronous-unwind-tables -fno-exceptions
-fno-rtti -fcf-protection=none -O2 220600-prob3.cpp -o problem3.out
./problem3.out
```

3.4 Results (averaged over 5 runs)

Grid Size	Scalar (ms)	SSE (ms)	AVX2 (ms)
128	277.0	263.8	286.6
256	2395.0	2354.8	2360.8
512	21845.4	21204.6	20948.2

3.5 Analysis

At smaller grid sizes, SIMD yielded modest gains (1.05x for SSE). At larger grids, performance was limited by cache reuse and bandwidth. AVX2 did not outperform SSE due to memory access overhead and lack of spatial locality.

Attempts at tiling and loop permutation did not improve results, as the access pattern was already sequential along the fastest-changing dimension.

3.6 Summary

- Speedup: 1.03–1.05x (memory-bound kernel)
- SSE slightly outperformed AVX2 due to smaller register load overhead.
- Further blocking/tiling was ineffective.

4 Problem 4: Grid Search Optimization and Parallelization

4.1 Objective

To optimize a nested grid search kernel (10 nested loops) originally taking over 360 seconds, using sequential optimizations and OpenMP parallelization.

4.2 Part (i): Sequential Optimization

The initial code (`problem4-v0.c`) had deeply nested loops with redundant computations. The following optimizations were applied:

- **Partial-Sum Reuse:** Eliminated repeated computation of cumulative products within nested loops by computing them only once.
- **Loop-Invariant Code Motion (LICM):** Moved invariant expressions outside inner loops.
- **Inlining and Function Simplification:** Merged function calls for reduced overhead.
- **Buffered Output:** Attempted write buffering for improved I/O performance but it did not give any significant further gain.
- **Loop Unrolling:** Attempted loop unrolling with a factor of 4 but did not give much further gain

- **Tiling:** Attempted Tiling with 16*16*16 for innermost 3 loops. Did not give significant improvement in performance

Compilation and Run:

```
gcc -std=c17 -O3 -masm=att -msse4 -mavx2 -march=native
-fopenmp -fverbose-asm -fno-asynchronous-unwind-tables
-fno-exceptions -fcf-protection=none -Wall -Wextra
220600-prob4_1.c -o problem4-1.out
./problem4-1.out
```

Results:

- Original : 362.94 s
- Partial-sum reuse + LICM : 258.40 s
- Buffered writes: 266.81 s (no gain)
- Loop tiling: degraded to 850 s
- Loop unrolling (unroll factor 8): 264.74s

Analysis: Partial-sum reuse and LICM offered significant gains by avoiding recomputation in 10 nested loops. Other measures did not affect the performance significantly in the positive direction

4.3 Part (ii): OpenMP Parallelization

The next version (`220600-prob4_2.c`) parallelized the outermost loop (`r0`) with deterministic file writes.

- **Ordered Output:** Ensured correct order of writes using `ordered` clause.
- **Reduction:** Used reduction on the point counter.
- **Per-thread buffers:** Each thread writes into a private buffer, then flushes deterministically.

Compilation and Run:

```
gcc -std=c17 -O3 -masm=att -msse4 -mavx2 -march=native -fopenmp
-fverbose-asm -fno-asynchronous-unwind-tables -fno-exceptions
-fcf-protection=none -Wall -Wextra problem4-2-v2.c
220600-prob4_2.c -o problem4-2.out -pthread
./problem4-2.out
```

Results:

- Naive: 362.94s
- Sequential: 258.40 s
- OpenMP: 73.43 s
- Points found: 11608 (correctness verified, also diff was done to check that the output files produced by all versions were same)

Analysis: Parallelization achieved a 3.5x improvement (over sequential) while maintaining output determinism. Sequential optimizations gave a speedup of about 1.4x over the naive implementation