CS610 Assignment 3

Pragati Agrawal (220779)

October 2025

How to run:

For each of the questions, just use the Makefile provided in the folder for running. It complies all the files and produces `.out` executables. To make a particular problem, say problem2, run `make problem2`. Problem4 has 3 versions, so run `make problem4-v0`, `make problem4-v1` and `make problem4-v2` to compile them individually.

Problem 1

In this question, we had to optimise a given loop nest using various loop transformations we have studied. Below is a summary of some of the different optimisations I have tried and their performances. The exact timings varied across runs, and I have quoted with respect to one run. But the overall performance of the optimisations was more or less similar. The runs were taken on KD ground floor lab machine **172.27.27.224**

Optimization Strategies

Version 1: Loop-Invariant Code Motion (LICM)

The loop computation involved many repeated index calculations. We can perform LICM and take them out and store them in temporary variables to avoid frequent computations in every iteration.

- Compute `i_offset = i * NY * NZ` once per i iteration
- Compute `jz_offset = j * NZ` once per j iteration
- Store neighbor offsets (`i_plus_offset`, `i_minus_offset`, `j_plus_offset`, and `j_minus_offset`) for other indices.

Impact: Gives a small amount of speedup upon naive (1.09x, 25ms → 23ms).

Version 2: Loop Unrolling (Factor 2)

Unrolled the innermost k-loop by a factor of 2.
Impact: Gives a modest speedup (1.14x, 25ms → 22ms), probably due to reduced loop overhead and enabling better instruction-level parallelism.

Version 3: Loop Unrolling (Factor 4)

Increased unrolling factor to 4 to try for more speedup.
Impact: Performs similar to factor-2 unrolling (1.09x, 25ms → 23ms), mostly because of increased code size.

Version 4-5: Loop Permutation

- **V4 (k-i-j):** Performs worse than baseline (0.83x, 30ms) due to poor spatial locality
- **V5 (j-k-i):** Performs even worse (0.66x, 38ms) with innermost stride being $NY \times NZ$

The original i-j-k ordering is optimal for this memory layout, where k has unit stride.

Version 6: Loop Tiling (Blocking)

The tile size: $8 \times 8 \times 16$ for (i, j, k) dimensions came out to be the best one after multiple runs for different configurations. The k-dimension has unit stride (favorable for cache), so we use a larger tile (16) compared to i and j (8 each).
Impact: Achieves modest improvement (1.14x, 25ms → 22ms), similar to unrolling.

Version 7: Loop Permutation + Tiling (k-i-j)

Combined permutation with tiling to see if blocking could help the permuted order.
Impact: Still performs worse than baseline (0.93x, 27ms), showing that permutation hurts performance even with tiling.

Version 8: OpenMP Parallelization

Added `#pragma omp parallel for schedule(static)` to the outermost i-loop, as there are no data dependencies between different i slices. Static scheduling provides balanced workload distribution. Used 32 threads.
Impact: Achieves significant speedup (3.57x, 25ms → 7ms).

Version 9: Loop Permutation + OpenMP (j-k-i)

Tried parallelizing the j-loop in the j-k-i permutation.
Impact: Achieves good speedup (4.17x, 25ms → 6ms), better than simple OpenMP.

Version 10: OpenMP + Unrolling

Combined OpenMP parallelization on the i-loop with 4x loop unrolling on innermost k-loop.
Impact: Achieves excellent speedup (6.25x, 25ms → 4ms).

Version 11: OpenMP + Tiling

Combined `#pragma omp parallel for collapse(2)` with loop tiling using $8 \times 8 \times 16$ tile sizes. The `collapse(2)` clause parallelizes both the outer tile loops.
Impact: Achieves the **best speedup of 8.33x** (25ms → 3ms).

Correctness Verification

All versions maintain correctness, verified through:

- Final center value: 0.0413429 (consistent across all versions)
- Total sum conservation: 100.0
- Tolerance: 10^{-9}

Analysis of Results

Version	Transformation	Time (ms)	Speedup
V0	Baseline	25	1.00×
V1	LICM	23	1.09×
V2	Unrolling (2x)	22	1.14×
V3	Unrolling (4x)	23	1.09×
V4	Loop Permutation (k-i-j)	30	0.83×
V5	Loop Permutation (j-k-i)	38	0.66×
V6	Tiling	22	1.14×
V7	Permutation + Tiling (k-i-j)	27	0.93×
V8	OpenMP	7	3.57×
V9	Permutation + OpenMP (j-k-i)	6	4.17×
V10	OpenMP + Unrolling (4x)	4	6.25×
V11	**OpenMP + Tiling**	**3**	**8.33×**

Key Observations:

- **Single-threaded optimizations (V1-V7):** Without parallelization, gains are limited (0.66x-1.14x). This suggests the code is memory-bound rather than compute-bound. Loop permutations (V4, V5) actually hurt performance by disrupting spatial locality.
- **Loop permutation:** The original i-j-k order is optimal because the innermost k-loop has unit stride, providing best cache line utilization. Permutations that move k to outer loops (V4: k-i-j, V5: j-k-i) degrade performance significantly in sequential execution.
- **OpenMP pragmas (V8):** Parallelization provides 3.57× speedup on naive version, which is the single most impactful optimization. With 32 threads available, we don't achieve perfect 32× scaling due to memory bandwidth limitations and synchronization overhead.
- **Combined optimizations (V10, V11):**
  - V10 (OpenMP + Unrolling): 6.25× speedup
  - V11 (OpenMP + Tiling): **8.33× speedup (BEST)**

  In some runs, **OpenMP+Unrolling** performed the best. We can say that both these outperform others (their difference being within 1ms in each run). This mostly explains effect of compound transformations, where OpenMP pragmas work on top of optimised serial codes, and give good speedup.

Problem 2

Approach

Understanding SSE4 Implementation

The SSE4 version provided in the assignment processes 4 integers (128 bits) at a time using a tree reduction approach:

1. Load 4 elements: `[d, c, b, a]`
2. Shift left by 4 bytes (1 element): `[c, b, a, 0]`
3. Add: `[c+d, b+c, a+b, a]`
4. Shift left by 8 bytes (2 elements): `[a+b, a, 0, 0]`
5. Add: `[a+b+c+d, a+b+c, a+b, a]`
6. Add offset from previous iteration
7. Broadcast the last element `(a+b+c+d)` as offset for next iteration

AVX2 Implementation

For AVX2 version, I extended this approach to process 8 integers (256 bits) at a time. However, I found in Intel Intrinsics that AVX2 operates on two independent 128-bit lanes. So, we needed to handle cross-lane data movement, for which I used `_mm256_permute2x128_si256` and `_mm256_blend_epi32` instructions.

Algorithm Steps:

1. Load 8 elements `[h, g, f, e | d, c, b, a]` (upper lane — lower lane)
2. Shift by 1 element and add
   - Shift within lanes: `[g, f, e, 0 | c, b, a, 0]`
   - Permute and blend to move element `d` to position 4: `[g, f, e, d | c, b, a, 0]`
   - Add: `[g+h, f+g, e+f, d+e | c+d, b+c, a+b, a]`
3. Shift by 2 elements and add
   - Shift within lanes by 8 bytes (2 elements): `[e+f, d+e, 0, 0 | a+b, a, 0, 0]`
   - Permute lower lane to upper: `[0, 0, 0, 0 | g+h, f+g, e+f, d+e]`
   - Blend at positions 4 and 5 (mask `0x30`): `[e+f, d+e, c+d, b+c | a+b, a, 0, 0]`
   - Add to `x`: `[(e+f)+(g+h), (d+e)+(f+g), (c+d)+(e+f), (b+c)+(d+e) | (a+b)+(c+d), (a)+(b+c), a+b, a]`
   - Simplified: `[e..h, d..g, c..f, b..e | a..d, a..c, a..b, a]`
4. Shift by 4 elements (cross-lane shift) and add
   - After permute, `shifted` = `[a..d, a..c, a..b, a | 0, 0, 0, 0]`

- Now add `shifted` to `x`:

Position	`x` value	`shifted` value	Result
7 (upper)	`e..h`	`a..d`	`a..h`
6 (upper)	`d..g`	`a..c`	`a..g`
5 (upper)	`c..f`	`a..b`	`a..f`
4 (upper)	`b..e`	`a`	`a..e`
3 (lower)	`a..d`	`0`	`a..d`
2 (lower)	`a..c`	`0`	`a..c`
1 (lower)	`a..b`	`0`	`a..b`
0 (lower)	`a`	`0`	`a`

5. Add cumulative sum from previous iterations
6. Extract and broadcast the last element for next iteration

Results

I used the KD ground floor lab machine **172.27.27.224** for collecting results. Also, I increased the array size to $N = 2^{24} = 16{,}777{,}216$ integers (64 MB) to balance out OpenMP threading overhead. Performance of SSE4 and AVX2 varied across runs, and it was any one fixed winner. They both performed similar mostly.

Run 1 Results

Version	Time (μs)	Speedup
Sequential	20,401	1.00×
OpenMP	16,377	1.25×
SSE4	7,693	2.65×
AVX2	9,105	2.24×

Run 2 Results

Version	Time (μs)	Speedup
Sequential	20,576	1.00×
OpenMP	18,393	1.12×
SSE4	7,990	2.58×
AVX2	7,800	2.64×

Analysis

Performance Observations

1. **OpenMP Performance:** The OpenMP SIMD version shows only modest improvement (1.18×), likely because it has less control over low-level optimizations compared to intrinsics SIMD instructions. It must rely on compiler auto-vectorization but the inherent sequential dependencies in prefix sum would limit optimization.
2. **AVX2 v/s SSE4:** AVX2 didn't outperform SSE4 in all runs, mostly because AVX2 requires additional permutation and blend operations for cross-lane data movement. Also, memory can be a bottleneck for AVX2 as it requires fetching 256 bits v/s 128 bits together.

Problem 3

Implementation Overview

The original scalar kernel computes a finite-difference gradient along the first dimension of a 3D grid. For each interior point at position (i, j, k) where $1 \leq i < NX - 1$, the kernel computes:

$$B[i, j, k] = A[i + 1, j, k] - A[i - 1, j, k]$$

SSE4 Vectorization

The SSE4 implementation uses 128-bit SIMD registers to process 2 elements (two 64-bit integers) simultaneously in the innermost loop:

- Uses `_mm_loadu_si128` to load two consecutive 64-bit values
- Uses `_mm_sub_epi64` to subtract two 128-bit vectors element-wise
- Uses `_mm_storeu_si128` to store the result
- Handles remaining elements with a scalar tail loop when NZ is not divisible by 2

AVX2 Vectorization

The AVX2 implementation uses 256-bit SIMD registers to process 4 elements (four 64-bit integers) simultaneously:

- Uses `_mm256_loadu_si256` to load four consecutive 64-bit values
- Uses `_mm256_sub_epi64` to subtract two 256-bit vectors element-wise
- Uses `_mm256_storeu_si256` to store the result
- Handles remaining elements with a scalar tail loop when NZ is not divisible by 4

Experimental Results

I used the KD ground floor lab machine **172.27.27.224** for collecting results. Correctness of the three versions was ensured by identical checksums (4128768).

Version	Time (ms)	Checksum	Speedup
Scalar	110	4128768	1.0x
SSE4	75	4128768	1.47x
AVX2	67	4128768	1.64x

Vectorization using SSE4 and AVX2 intrinsics successfully improved the performance of the 3D gradient kernel, achieving speedups of 1.47× and 1.64× respectively. The sub-linear speedups relative to SIMD width indicate that the kernel is memory-bound rather than compute-bound. The speedup is mostly limited by memory bandwidth. Other reasons for not achieving ideal speedup might be loop overhead and cache performance. There is also loop carried data dependence that limits extent of useful vectorisation.

$$\text{Speedup}_{\text{AVX2/SSE4}} = \frac{75}{67} = 1.12\times$$

AVX2 shows improvement over SSE4, which was expected because AVX2 used wider registers and can expose more parallel computation, requiring less time. But it is not twice as better mostly because of memory bandwidth and other factors like those listed above.

Problem 4

Part (i): Sequential Optimization using Loop Transformations

1. Loop Invariant Code Motion (LICM) Move loop-invariant computations outside all loops to avoid repeated computation of constraint thresholds and loop bounds. `const` qualifier enables compiler optimizations (registers allocation, constant propagation)

```
const double e1 = kk * ey1, e2 = kk * ey2, ..., e10 = kk * ey10;
const int s1 = (int)floor((dd2 - dd1) / dd3);
const int s2 = (int)floor((dd5 - dd4) / dd6);
// ... all loop bounds pre-computed
```

2. Incremental Variable Updates Replaced 10 multiplications per innermost iteration with 10 additions (which has lower latency).

```
double x1 = dd1;  // Initialize once
for (int r1 = 0; r1 < s1; ++r1, x1 += dd3) {  // Increment by addition
    double x2 = dd4;
    for (int r2 = 0; r2 < s2; ++r2, x2 += dd6) {
        // ... incremental updates for all variables
    }
}
```

3. Loop Fusion with Incremental Accumulation Restructured computation to compute `q_i` incrementally across loop levels, reusing computation from outer loops in inner loops. Instead of 100 multiplications per point, only 10 multiplications per dimension are needed.

```
double q1 = c11*x1, q2 = c21*x1, ..., q10 = c101*x1;

double r2q1 = q1 + c12*x2, r2q2 = q2 + c22*x2, ..., r2q10 = q10 + c102*x2;

double r3q1 = r2q1 + c13*x3, r3q2 = r2q2 + c23*x3, ...;

// Continue through all 10 levels...

// Innermost level (r10): Only add final term and evaluate
if (fabs(r9q1 + c110*x10 - d1) > e1) continue;
```

4. Check after each evaluation and leave the loop early Check if constraint fails, immediately after computation and continue without computing rest of the constraints.

```
if (fabs(r9q1 + c110*x10 - d1) > e1) continue;
...
if (fabs(r9q10 + c1010*x10 - d10) > e10) continue;

// print if all constraints pass
fprintf(...);
```

Transformations Attempted but Not Used:

- **Loop Tiling:** Attempted to tile outer loops for better cache utilization, but did not improve performance due to:
  - The arrays are small in size, already fits in cache
  - Added complexity of tile indexing offset increases time
- **Loop Permutation:** Tried reordering loops but observed no benefit since:
  - Memory access pattern is already optimal
  - All computations are scalar operations

Part (ii): Parallelisation using OpenMP

1. `#pragma omp parallel default(shared)`

Creates an initial pool of threads that execute the enclosed code block in parallel. Default shared clause is used because function parameters (grid bounds, coefficients, constraints) need to be accessible to all threads.

2. `#pragma omp for schedule(dynamic) collapse(3) nowait`

Dynamic scheduling is used because the workload varies significantly between iterations: some iterations find many valid points (satisfying all constraints) while others find few or none. Static scheduling would cause load imbalance. Then collapse(3) combines the first three nested loops into a single iteration space and increases granularity of parallelization. If only the outermost loop was parallelized and $s_1 <$ num_threads, some threads would be idle. Nowait clause removes the implicit barrier at the end of the parallel for loop. Due to dynamic scheduling, threads finish at different times and the computation doesn't require synchronisation at this point, we can continue in concurrent manner. We will synchronise at the end to update the global result from its locally computed values.

3. `#pragma omp critical`

Each thread maintains its own `local_results` buffer during computation. After completing assigned work, threads merge their results into the shared `results` array. To avoid data race, critical clause is used.

Results

I used the KD ground floor machine **172.27.27.224** for collecting results. All versions produce the same number of result points (11,608), confirming correctness of both sequential and parallel optimizations.

Version	**Time (seconds)**	**Speedup**	**Optimization**
Sequential (v0)	120.740664	1.00×	Baseline (no optimization)
Optimized (v1)	89.630421	1.35×	Part (i): Loop transformations
OpenMP (v2)	8.538000	14.14×	Part (ii): Parallelization (32 threads)