

CS610- Assignment3

Name: Nishant
Roll No: 251110053

Problem 1: 3D Stencil Optimization

Run commands:

```
make problem1  
./problem1.out
```

The original scalar code for this was optimized through three main strategies:

- **Manual unrolling:** The k loop was manually unrolled to reduce branching and boost instruction-level parallelism, allowing the processor to more effectively utilize execution units.
- **OpenMP parallelism:** The outer i and j loops were parallelized using `#pragma omp parallel for collapse(2)` to distribute grid planes across CPU cores.
- **Compound optimization:** The fastest kernel combined unrolling and parallelism.

Outputs from all versions were consistent, with matching sums (100) and center cell values (0.0413429).

Version	Mean Time (ms)
Scalar Only	37.5
Unrolled	31.0
OpenMP	6.5
OMP + Unroll	3.5

Table 1: Average execution times for 3D stencil kernel versions.(results from ccpc158)

Runtimes improved with each optimization layer, with parallel and unrolled execution achieving over $10\times$ speedup vs. the baseline.

Problem 2: SIMD Prefix Sum (Scan)

Run commands:

```
make problem3  
./problem3.out
```

The AVX2 implementation of prefix sum divides the array into blocks of 8 integers, each fitting perfectly into a 256-bit register. For each block, the algorithm loads eight consecutive elements

into a vector, and then constructs partial prefix sums by a sequence of left shifts and adds, cumulatively folding the vector values. After the in-register scan, an offset from the previous block's total sum is added across the current block, ensuring the correct prefix relationship for the whole array.

Version	Mean Time (μ s)
Serial	35483.5
OMP	34415.0
SSE	33956.5
AVX2	34148.0

Table 2: Mean execution times for prefix sum.(results from ccpc158)

All output sums matched the expected result.

Problem 3: 3D Gradient Kernel Vectorization

Run commands:

```
make problem3
./problem3.out
```

The gradient computation was vectorized over the innermost k loop, capitalizing on stride-1 memory access and maximizing efficiency. The implementation included three core versions:

- **Scalar baseline:** Used standard triple-nested loops for i , j , and k to compute differences along i . This served as the reference.
- **SSE4.1 kernel:** With 128-bit vectors, loaded and subtracted two 64-bit elements at a time. Utilized `_m128i` variables mapped to memory via aligned loads/stores.
- **AVX2 kernel:** Processed four 64-bit elements per loop using `_m256i`. For best performance, made sure input/output arrays were allocated using `aligned_alloc(32, ...)` for proper 32-byte alignment. Within each block, subtraction was performed directly in registers, and vector results were written back.

All arrays, output buffers, and checksums were validated at the end of each execution.

Version	Mean Time (ms)	Mean Speedup
Scalar	133.0	1.00
SSE4	107.7	1.24
AVX2	99.0	1.34

Table 3: Performance comparison for gradient kernel.(ccpc158)

Problem 4: Loop Transformation and Parallelization

Run commands:

```
make problem4-v1  
./problem4-v1.out  
make problem4-v2  
./problem4-v2.out
```

The directory for this problem contains files `problem4-v0.c`, `problem4-v1.c`, and `problem4-v2.c`. Output files are generated after compilation and execution. The solution proceeds in two stages:

- **Version 1 (Optimized Serial):** Redundant computations inside the main loop were moved outside using loop-invariant code motion (LICM), leading to about 2x faster execution. Loop unrolling and permutation were tested but did not improve performance. Only LICM was kept as it was the most effective optimization.
- **Version 2(Parallelized OpenMP):** Multiple for-loops were restructured for parallel execution. OpenMP `collapse` clause was used with empirically chosen parameter (best with collapse value = 8 for grid dimensions). Dynamic scheduling improved workload balance among threads. Atomic updates were used for shared variables. File writes, if present, were ordered using `omp ordered` to preserve output correctness.

All runs were validated to produce the correct output points (11608).

Version	Total Time (s)
v0	287.998
v1	148.581
v2	10.472

Table 4: Average execution times for Problem 4 versions(ccpc158)

Blocking and careful restructuring reduced computation time, while parallelization delivered the largest speedup.