CS610 : Programming For Performance

Assignment 3

Submitted By : Kushagra Srivastava, 220573

Problem 1

Apply valid loop transformations to the stencil computation given in the template file (problem1.cpp) to improve its performance. You can apply compound transformations to improve the kernel's memory access pattern and can use OpenMP to exploit parallelism. You should not write explicit SIMD code. Summarize the set of transformations (e.g., xx times unrolling + yy permutation) that gives you the best performance. Write different kernels to incrementally show the benefit of your transforma- tions. Ensure correctness of your transformed kernels and compare performance across versions.

Solution

Kernel Implementations

The following transformations were applied to optimize the 3D 7-point stencil.

1. **LICM**
   - Used `plane` $=$ `NY` $\times$ `NZ` and `row` $=$ `NZ` with a per-row base index `base` $= i \cdot$ `plane` $+ j \cdot$ `row` to minimize repeated multiplications across loops.
2. **OpenMP Parallelization**
   - Parallelize the outer spatial loops with `#pragma omp parallel for`.
   - Use `collapse(2)` on (i, j) to expose more iterations to the scheduler.
   - Use `schedule(static)` for low overhead given uniform work per iteration and to avoid repeated fork/join.
3. **SIMD vectorization with OpenMP `simd`**
   - Vectorized the innermost k-loop with `#pragma omp simd`, specifying `simdlen(...)`/`safelen(...)`, `aligned(curr,next:64)`, and `linear(k:1)` as hints.
   - Kept k unit-stride so that load/stores are contiguous.
4. **Loop unrolling (k-dimension)**
   - Applied $2\times$ manual unrolling of the k-loop (both in scalar and OpenMP variants) to trim loop-control overhead and increase instruction-level parallelism.
5. **Cache tiling for locality**
   - Introduced 3D blocking with typical tile sizes `TILE_I` $= 16$, `TILE_J` $= 16$, `TILE_K` $= 32$.
   - Issued `__builtin_prefetch` for rows/planes ahead of use within the same tile.

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

The following execution times were recorded for the these implementations

Implementation	**Time (ms)**	Speedup
Baseline	21	1.00×
OpenMP with `collapse(2)`	5	4.20×
OpenMP with `schedule(static)`	4	5.25×
Unrolled-only (no OpenMP)	18	1.16×
Tiled-only (no OpenMP)	19	1.10×
Unrolled + OpenMP	5	4.20×
OpenMP + SIMD	4	5.25×

Table 1: Execution time for 100 iterations with `NX = NY = NZ = 66`

Implementation	**Time (ms)**	Speedup
Baseline	164	1.00×
OpenMP with `collapse(2)`	16	10.25×
OpenMP with `schedule(static)`	15	10.93×
Unrolled-only (no OpenMP)	148	1.10×
Tiled-only (no OpenMP)	198	.82×
Unrolled + OpenMP	15	10.93×
OpenMP + SIMD	15	10.93×

Table 2: Execution time for 100 iterations with `NX = NY = NZ = 130`

Implementation	**Time (ms)**	Speedup
Baseline	1874	1.00×
OpenMP with `collapse(2)`	1919	.97×
OpenMP with `schedule(static)`	1916	.97×
Unrolled-only (no OpenMP)	1833	1.02×
Tiled-only (no OpenMP)	2363	.79×
Unrolled + OpenMP	1912	.98×
OpenMP + SIMD	1871	1.00×
Tiled + Unrolled + OpenMP + SIMD + Prefetch	1761	1.06

Table 3: Execution time for 100 iterations with `NX = NY = NZ = 258`

Problem 2

Implement an *inclusive* prefix sum function using AVX2 instrinsics. Compare the performance with the sequential, OpenMP, and SSE4 versions (discussed in class).

Solution

Implementation

Both SSE4 and AVX2 implementations use a tree reduction approach to compute prefix sums within each SIMD vector, then propagate the cumulative sum between vectors using an offset variable.

AVX2 Implementation

The AVX2 version processes 8 integers at a time. Although, *AVX2 shift operations work independently within each 128-bit lane*, not across the entire 256-bit register. The register is effectively two separate 128-bit lanes: elements [0-3] in the low lane and elements [4-7] in the high lane.

-

Steps 1-2: We perform the same tree reduction as SSE4, but the shifts only work within each lane. After two shift-and-add operations, we have correct prefix sums within each lane independently:

- Low lane: $[a, a+b, a+b+c, a+b+c+d]$
- High lane: $[e, e+f, e+f+g, e+f+g+h]$ (missing $a+b+c+d$)

Step 3 (Cross-lane correction – THE KEY DIFFERENCE): To fix the high lane, we must manually:

1. Extract the high 128-bit lane (elements [4-7])
2. Extract and broadcast the last element from the low lane (element [3], which contains $a+b+c+d$)
3. Add this value to all elements in the high lane
4. Insert the corrected high lane back into the 256-bit register

After this correction, the high lane becomes $[a+b+c+d+e, a+b+c+d+e+f, \ldots]$ which is correct.
Steps 4-5: Same as SSE4 – add offset, store, and broadcast the last element for the next iteration.

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

Baseline Comparison

Implementation	Time (μs)	Speedup
Serial	`91`	`1.00`×
OpenMP SIMD	`95`	`0.95`×
SSE4 (128-bit)	`10`	`9.1`×
AVX2 (256-bit)	`16`	`5.68`×

Table 4: Performance comparison for $N = 2^{16}$ with uniform input (all 1's), averaged over `100` runs

Implementation	Time (μs)	Speedup
Serial	`17549`	`1.00`×
OpenMP SIMD	`16017`	`1.09`×
SSE4 (128-bit)	`8440`	`2.06`×
AVX2 (256-bit)	`8310`	`2.11`×

Table 5: Performance comparison for $N = 2^{24}$ with uniform input (all 1's), averaged over `100` runs

Scalability Analysis

Size N	Execution Time (μs)			
	Serial	**OpenMP**	**SSE4**	**AVX2**
2^{16} (256K)	91	95	10	16
2^{18} (1M)	377	370	47	60
2^{20} (4M)	1362	1249	199	219
2^{22} (16M)	4467	3953	1380	1541
2^{24} (64M)	17549	16017	8440	8310
2^{26} (256M)	70704	65305	34485	34568

Table 6: Execution time averaged over 100 runs versus array size

Size N	Speedup vs. Serial			
	Serial	**OpenMP**	**SSE4**	**AVX2**
2^{16} (256K)	1.00×	0.95×	9.1×	5.68×
2^{18} (1M)	1.00×	1.01×	8.02×	6.28×
2^{20} (4M)	1.00×	1.09×	6.84×	6.21×
2^{22} (16M)	1.00×	1.13×	3.23×	2.89×
2^{24} (64M)	1.00×	1.09×	2.06×	2.11×
2^{26} (256M)	1.00×	1.08×	2.05×	2.04×

Table 7: Speedup averaged over 100 runs versus array size

Problem 3

Vectorize the scalar 3D gradient kernel given the template file (`problem3.cpp`) using SSE4 and AVX2 intrinsics. Compare the performance among the three versions, and report the speedups.

Solution

Implementation

The baseline scalar version of this kernel processes one element at a time with three nested loops. The implementations for the `SSE4` and `AVX2` versions are discusses below.

SSE4 Implementation
SSE4 provides 128-bit registers that can hold 2 `uint64_t` values. The vectorized loop processes 2 consecutive k values simultaneously.

1. **Load**: Use `_mm_loadu_si128` to load 2 consecutive elements from `A[i+1,j,k:k+1]` and `A[i-1,j,k:k+1]`
2. **Subtract**: Compute element-wise 64-bit subtraction. Since, SSE4 does not have a direct substraction function, this is done.

```
        __m128i neg_left = _mm_sub_epi64(_mm_setzero_si128(), left);
        __m128i result = _mm_add_epi64(right, neg_left);
```

3. **Store**: Use `_mm_storeu_si128` to write 2 results back to `B[i,j,k:k+1]`

AVX2 Implementation
AVX2 provides 256-bit registers that can hold 4 `uint64_t` values.

1. **Load**: Use `_mm256_loadu_si256` to load 4 consecutive elements
2. **Subtract**: AVX2 provides `_mm256_sub_epi64` for direct 64-bit subtraction across all 4 lanes

```
__m256i result = _mm256_sub_epi64(right, left);
```

3. **Store**: Use `_mm256_storeu_si256` to write 4 results

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

The following execution times were recorded for the two functions

Implementation	Time (ms)	Speedup	GOPS	Memory Bandwidth (GB/s)
Scalar	97	1.00×	2.13	34.05
SSE4	76	1.28×	2.72	43.46
AVX2	65	1.49×	3.18	50.82

Table 8: Execution time for 100 iterations ($N_{ITERATIONS} = 100$ and `NX = NY = NZ = 128`)

Implementation	Time (ms)	Speedup	GOPS	Memory Bandwidth (GB/s)
Scalar	884	1.00×	2.34	37.36
SSE4	694	1.27×	2.97	47.59
AVX2	631	1.40×	3.27	52.35

Table 9: Execution time for 1000 iterations ($N_{ITERATIONS} = 1000$ and `NX = NY = NZ = 128`)

Implementation	Time (ms)	Speedup	GOPS	Memory Bandwidth (GB/s)
Scalar	14115	1.00×	0.95	15.15
SSE4	13566	1.04×	0.99	15.77
AVX2	13646	1.03×	0.98	15.68

Table 10: Execution time for 100 iterations ($N_{ITERATIONS} = 100$ and NX = NY = NZ = 512)

Implementation	Time (ms)	Speedup	GOPS	Memory Bandwidth (GB/s)
Scalar	139999	1.00×	0.95	15.28
SSE4	135930	1.03×	0.98	15.74
AVX2	135969	1.03×	0.98	15.73

Table 11: Execution time for 1000 iterations ($N_{ITERATIONS} = 1000$ and NX = NY = NZ = 512)

Problem 4

Part (i)

Perform loop transformations to improve the performance of the attached C code (`prob4-v0.c`) for sequential execution on one core (i.e., no multithreading). You may use any loop transformation we have studied, e.g., loop permutation and loop tiling, but no array padding or layout transformation (e.g., 2D→1D) of arrays is allowed. You are free to apply any code optimization trick (e.g., LICM, function inlining, and changing function prototypes) on the attached version for improved performance.

Summarize the set of transformations (e.g., *xx times unrolling + yy permutation*) that gave you the best performance. Explain your optimizations and the improvements achieved. Some transformations you try may not improve the performance. You can still include a description in your report.

Solution

Optimizations

Several optimization strategies were tried to speed this up:

- LICM (Loop-Invariant Code Motion)

  Move computations whose values don't change across iterations outside the loop to avoid redundant work.

- Batching Writes

  Aggregate many small memory/I/O writes into fewer larger contiguous writes to cut per-operation overhead and improve locality.

- Collapsing Loops

  Flatten nested loops into a single loop over the combined iteration space to boost work per iteration and enable vectorization/parallelism.

- Vector Intrinsics

  Use architecture-specific SIMD functions that map directly to vector instructions to operate on multiple data elements per instruction.

- Loop Unrolling

  Duplicate the loop body multiple times per iteration to reduce loop-control overhead and expose instruction-level parallelism.

- Loop Re-ordering

  Moves the slowest loops (r9, r10) to the outside and builds constraint calculations step by step.

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

The following execution times were recorded for the these implementations

Implementation	Time (s)	Speedup
Baseline	99.03	1.00×
Loop Reordering **v1**	103.4	0.957×
LICM **v2**	108.11	0.916×
Tiling**v3**	170.05	0.58×
LICM at Level 8 **v4**	116.36	0.85×
LICM + Unroll **v5**	89.71	1.10×
Combine Inner Loops **v6**	45.81	2.16×

Table 12: Execution time averaged over 10 runs

Part (ii)

Parallelize the attached C code (`problem4-v0.c`) using OpenMP. Compare the performance of the OpenMP program with the sequential version. You may use any combination of valid OpenMP directives and clauses that you think will help.

Solution

Optimizations

Several optimization strategies were tried to speed this up:

- Parallelize outermost loop using SMID Directives: `parallel for`, `schedule(dynamic)`, `reduction`, `critical`
- Collapsing Loops using `collapse(3)` and `schedule(dynamic, 2)`
- Loop Re-ordering
  Moves the slowest loops (r9, r10) to the outside and builds constraint calculations step by step.

Performance Results

These functions were tested on `csews` with IP `172.27.28.135`, with the following hardware configurations,

- **Processor**: `Intel(R) Core(TM) i9-14900` with `32` cores
- **Cache**:
  - L1d: `896 KiB (24 instances)`
  - L1i: `1.3 MiB (24 instances)`
  - L2: `32 MiB (12 instances)`
  - L3: `36 MiB (1 instance)`

The following execution times were recorded for the these implementations

Implementation	Time (s)	Speedup
Sequential	99.03	1.00×
Outer Loop Parallelization **v1**	6.25	15.86×
Loop Collapsing **v2**	3.79	25.98×
Buffered **v3**	17.38	5.69×

Table 13: Execution time averaged over 10 runs