ECE415: High Performance Computing Systems
Lab1 - Sobel algorithm optimization

Contents

1 Introduction

The objective of this laboratory exercise is to optimize the sequential Sobel filter implementation on a CPU for detecting edges in grayscale images.

The Sobel operator computes the gradient magnitude of the image intensity at each pixel, highlighting regions of high spatial variation, which correspond to edges. Two 3×3 convolution kernels are used, one for horizontal changes and one for vertical changes. The final output image is obtained by combining the horizontal and vertical responses.

To verify correctness, the processed output is compared against a reference "golden" image using the Peak Signal-to-Noise Ratio (PSNR). Execution time is measured using the clock_gettime() function, allowing precise evaluation of the performance impact of subsequent optimizations.

2 Experimental Setup

- **Hardware:** Cpu: i5 1240p, 12 cores - 16 threads (Heterogeneous architecture- 4 performance cores - 8 effieciency cores), Alder Lake architecture
- **OS:** Ubuntu 24.04.3 LTS, kernel 6.14.0.
- **Compiler:** Intel oneAPI `icx`(2025.2.1)
- **Build flags without optimization :** `-O0 -g`
- **Build flags with optimization :** `-O3 -ffast-math -g` .
- **Timing:** Each binary is executed 12 times and the average and the standard deviation are computed excluding min and max runs.

3 Baseline Implementation (sobel_orig)

The provided baseline code applies the Sobel filter by calling a separate `convolution2D()` function for each pixel and for each kernel (horizontal and vertical). For each pixel the horizontal and vertical convolution results are squared using `pow(x, 2)`, summed, and the gradient magnitude is computed via `sqrt()`. The main computation is structured with nested loops with the outer loop iterating over columns (j), and the inner loop over rows (i). After computing the Sobel filtered image the PSNR is calculated by iterating over the output and reference "golden" image. Execution time is measured from the start of the nested loops through the PSNR calculation and it providea a baseline for evaluating the impact of subsequent optimizations.

4 Optimization Process under O0

The following optimizations were applied cumulatively, with each new modification building upon the previous version of the Sobel implementation. Only the changes that improved execution time (with one exception explained later) or at least did not degrade performance when compiled with the `-O0` flag (no compiler optimizations) were retained and included in the subsequent tests.

4.1 Test 1: Loop Interchange

Change: The order of the two nested loops was swapped from column-major traversal (`for (j) for (i)`) to row-major traversal (`for (i) for (j)`), aligning with the memory layout of C arrays. This ensures that consecutive memory locations are accessed within the inner loop.

Rationale: In C, arrays are stored in row-major order, meaning that consecutive elements in memory correspond to adjacent columns in the same row. By iterating over rows first, memory accesses become more sequential, improving spatial locality and cache utilization. This reduces cache misses significantly.

Results: The effects of this change on execution time and speedup are shown in Table 1.

Table 1: Performance of Test 1 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test1	-O0	1.445189	0.117923	1.40×

4.2 Test2: Replacing `pow()` with Multiplication

Change: Replaced calls to `pow(x, 2)` with direct multiplication $x * x$ in both the gradient magnitude ($p = p_1^2 + p_2^2$) and PSNR calculation.

Rationale: `pow(x, y)` is a general-purpose function that performs extra checks and uses costly operations such as logarithms and exponentials. For the fixed exponent 2, these operations are unnecessary using direct multiplication reduces instruction count and allows the compiler to compute the square inline, improving execution time.

Results: The effects of this change on execution time and speedup are summarized in Table 2.

Table 2: Performance of Test 1 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test2	-O0	0.881598	0.094736	2.30×

4.3 Test 3: Inlining `convolution2D()`

Change: The function `convolution2D()` was eliminated, and its 3×3 kernel computation was written directly inside the main pixel-processing loop. The convolution is now computed inline by iterating over the 3×3 neighborhood using two nested loops.

Rationale: Function calls introduce overhead due to stack frame creation, parameter passing, and return handling. By inlining the convolution operation, this overhead is removed and all arithmetic operations are kept within the main loop, allowing the compiler to optimize them more effectively.

Results: The effects of this change on execution time and speedup are shown in Table 2.

Table 3: Performance of Test 3 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test3	-O0	0.634364	0.011789	3.20×

4.4 Test 4: Loop Unrolling

Change: The two nested loops of the 3×3 convolution were fully unrolled. Each of the nine multiplications and accumulations corresponding to the kernel elements was written explicitly inside the main loop.

Rationale: Loop unrolling removes the overhead of loop counters, condition checks, and branching that occur for every iteration. Since the Sobel kernel size is fixed, all operations can be executed directly, allowing the compiler to schedule more instructions in parallel and improve instruction level parallelism.

Results: The effects of this change on execution time and speedup are shown in Table 3.

Table 4: Performance of Test 4 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test4	-O0	0.266311	0.006000	7.62×

4.5 Test 5: Common Subexpression Elimination

Change: Repeated address computations such as `i * SIZE` were replaced with precomputed variables that are reused within the loop. The row base addresses are calculated once per iteration:

```
for (i=1; i<SIZE-1; i++) {
    for ( j=1; j<SIZE-1; j++ ) {
        // common subexpression elimination
        row0   = i * SIZE;
        row_m1 = row0 - SIZE;
        row_p1 = row0 + SIZE;
    }
}
```

These are then used for all neighboring pixel accesses (`row_m1 + j - 1`, `row0 + j`, `row_p1 + j + 1`, etc.). The same approach was also applied in the PSNR computation loop to reuse the index `row0 + j` instead of recalculating it each time.

```
for (i=1; i<SIZE-1; i++) {
    for ( j=1; j<SIZE-1; j++ ) {
        // common subexpression elimination
        row0 = i*SIZE;
        temp = row0+j;
        t1 = output[temp] - golden[temp];
        t = t1*t1;
        PSNR += t;
    }
}
```

Rationale: This optimization removes redundant arithmetic operations from the inner loops. By avoiding repeated multiplications (`i * SIZE`) and reusing computed indices, ALU pressure and instruction count are reduced.

Results: The effects of this change on execution time and speedup are shown in Table 4.

Table 5: Performance of Test 5 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test5	-O0	0.276984	0.017776	7.33×

4.6 Test 6: Loop Invariant Code Motion

Change: The computation of row-dependent addresses (`row0` = (`i*SIZE`), `row0 - SIZE`, `row0 + SIZE`) was moved outside the inner pixel loop. Both the Sobel computation and the PSNR calculation now reuse these precomputed values, reducing repeated arithmetic inside the inner loop:

```
for (i = 1; i < SIZE-1; i++) {
    //Common Subexpression elimination + Loop Invariant code motion
    row0   = i * SIZE;
    row_m1 = row0 - SIZE;
    row_p1 = row0 + SIZE;
    for (j = 1; j < SIZE-1; j++) {
        // use precomputed row addresses for all neighbor accesses
    }
}
```

Rationale: By moving computations that are invariant across the inner loop outside of it we reduce redundant arithmetic for each pixel. This decreases ALU usage and improves instruction level efficiency.

Results: The effects of this change on execution time and speedup are shown in Table 6.

Table 6: Performance of Test 6 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test6	-O0	0.271080	0.016552	7.49×

4.7 Test 7: Simplifying multiplications with horizontal and vertical operators

Change: Algebraic simplifications were applied to the Sobel kernel operations. Multiplications by 0 were removed, multiplications by ±1 were replaced with simple addition or subtraction and multiplications by 2 were replaced with left shift operations:

```
// Examples of simplifications
p1 += -val;         // -1
p2 += val;          // 1

p2 += val << 1;     // 2
```

Rationale: Multiplications are costlier than additions or bit shifts. Replacing ±1 multiplications with add/sub, removing those by 0 entirely and replacing ×2 with a shift reduces instruction count and lowers register pressure.

Results: The effects of this change on execution time and speedup are shown in Table 7.

Table 7: Performance of Test 7 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test7	-O0	0.206507	0.002645	9.83×

4.8 Test8: Pointer Based Row Access

Change: Use pointers to access the current and neighboring rows as well as the output row in both the Sobel computation and PSNR calculation:

```
temp = i * SIZE;
unsigned char *row_m1 = input + temp - SIZE;
unsigned char *row0   = input + temp;
unsigned char *row_p1 = input + temp + SIZE;
unsigned char *out_row = output + temp;
```

Neighbors are then accessed as `row_m1[j-1]`, `row0[j+1]`, etc.

Rationale: Using pointers to access entire rows simplifies memory addressing beyond precomputed indices as it may and allow the compiler to keep row base addresses in registers reduce instruction dependencies and improve spatial locality.

Results: The effects of this change on execution time and speedup are shown in Table 8.

Table 8: Performance of Test 8 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test8	-O0	0.185617	0.001414	10.94×

4.9 Test9: Sequential Pointer Access

Change: Row pointers are initialized at column 1 and incremented directly in the inner loop instead of using the column index for array access. Neighbor accesses use relative offsets:

```
temp = i * SIZE + 1;
unsigned char *row_m1 = input + temp - SIZE;
unsigned char *row0   = input + temp;
unsigned char *row_p1 = input + temp + SIZE;
unsigned char *out_row = output + temp;

for (j = 1; j < SIZE-1; j++) {
    p1 = -row_m1[-1] + row_m1[1]
         - (row0[-1]<<1) + (row0[1]<<1)
         - row_p1[-1] + row_p1[1];

    p2 = row_m1[-1] + (row_m1[0]<<1) + row_m1[1]
         - row_p1[-1] - (row_p1[0]<<1) - row_p1[1];

    ......

    // Increment pointers for next column
    row_m1++; row0++; row_p1++; out_row++;
}
```

Rationale: This builds on previous pointer based optimizations. Incrementing pointers directly removes dependence on the column index, reducing arithmetic inside the inner loop. Sequential pointer traversal allows the compiler to generate simpler instructions and improves instruction level efficiency.

Results: The effect on execution time and speedup is shown in Table 9.

Table 9: Performance of Test 9 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test9	-O0	0.186470	0.001414	10.89×

4.10 Test10: Loop Fusion

Change: Combined the Sobel filter computation and PSNR calculation into a single nested loop:

```
temp = i * SIZE + 1;
unsigned char *row_m1 = input + temp - SIZE;
unsigned char *row0 = input + temp;
unsigned char *row_p1 = input + temp + SIZE;
unsigned char *out_row = output + temp;
unsigned char *gold_row = golden + temp;

for (j = 1; j < SIZE-1; j++) {
    // Compute Sobel filter for current pixel

    .....

    // Compute PSNR for current pixel
    t1 = *(out_row-1) - *gold_row;
    PSNR += t1*t1;

    // Increment pointers for next column
    row_m1++; row0++; row_p1++; out_row++, gold_row++;
}
```

Rationale: Merging the filtering and PSNR loops reduces loop overhead and avoids repeated index calculations. Although this modification does not significantly improve execution time on its own, it is retained as a foundation for subsequent optimizations as experimental results showed that it consistently enhances the performance of later optimization steps.

Results: The effect on execution time and speedup is shown in Table 10.

Table 10: Performance of Test 10 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test10	-O0	0.222350	0.001000	9.13×

4.11 Test11: Lookup Table for Gradient Magnitudes

Change: Introduce a lookup table (LUT) for gradient magnitudes:

- Precompute `sqrt_LUT[p]` = `(unsigned char)`$\sqrt{p}$ for $p \in [0, 2{,}080{,}800]$.
- Replace `res = (int)sqrt(p1*p1 + p2*p2)` with `res = sqrt_LUT[p]`.

Rationale: Without the LUT, a square root is computed for every pixel of the image. For a 4096×4096 image, this amounts to 16,777,216 calls to `sqrt()`. With the LUT, the square root is computed only once for each possible value of $p = p1^2 + p2^2$, i.e., for $p \in [0, 2{,}080{,}800]$, which is 2,080,801 computations. All subsequent pixels perform a single memory load from the LUT instead of an expensive calculation. This reduces the total computational cost dramatically.

Notes: LUT size $\approx$ 2 MB, with all values clipped to 255. Initialization of the LUT is done once and included in the timing measurements.

Results: The effect on execution time and speedup is shown in Table 11.

Table 11: Performance of Test 11 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test11	-O0	0.098181	0.001000	20.68×

4.12 Test12: Inlined Lookup Table Initialization

Change: Inline the initialization of the gradient magnitude LUT directly in the Sobel function.

Rationale: Inlining avoids the function call reduces overhead and allows better instruction cache locality. The total number of square root calculations remains 2,080,801, but the LUT is immediately available for processing.

Results: The effect on execution time and speedup is shown in Table 12.

Table 12: Performance of Test 12 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test12	-O0	0.099140	0.003741	20.48×

Table 13: Performance of Test 13 vs. baseline for -O0

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	2.030373	0.145282	1.00×
test13	-O0	0.098505	0.001414	20.61×

5 Optimization Process under `-ffast-math`

When compiled with the `-ffast-math` flag, the compiler already applies a broad set of optimizations. Consequently fewer manual modifications had an impact compared to the -O0 case. Only the changes that further improved execution time or at least did not degrade it were retained.

5.1 Test1: Replacing `pow()` with Multiplication

This optimization is identical to the its counterpart applied under `-O0`. The effects of this change on execution time and speedup are shown in Table 14.

Table 14: Performance of test1 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test1	-ffast-math	0.142447	0.007053	4.36×

5.2 Test2: Loop Interchange

This optimization is identical to the its counterpart applied under `-O0`. The effects of this change on execution time and speedup are shown in Table 15.

Table 15: Performance of test2 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test2	-ffast-math	0.120539	0.009229	5.15×

5.3 Test2_fast_1: Loop Unrolling in `convolution2D`

This optimization applies the same loop unrolling used in Test 4 under `-O0`, but without inlining since inlining was found to degrade performance when compiler optimizations were enabled. The effects of this change on execution time and speedup are shown in Table 16.

Table 16: Performance of test2_fast_1 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test2_fast_1	-ffast-math	0.09342	0.004145	6.4×

5.4 Test2_fast_2: Common Subexpression Elimination

This optimization applies a similar subexpression elimination as used in Test 5 under `-O0` but adapted as `convolution2D()` remains a separate function. The base addresses for each row in the 3×3 neighborhood are precomputed once per pixel avoiding repeated index calculations in the inner loop.

```
for (i = 1; i < SIZE-1; i++) {
    for (j = 1; j < SIZE-1; j++) {
        // Precompute base addresses for the three rows
        base_y0 = (i-1) * SIZE;
        base_y1 = i * SIZE;
        base_y2 = (i+1) * SIZE;

        ...
    }
}
```

The effects of this change on execution time and speedup are shown in Table 18.

Table 17: Performance of test2_fast_2 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test2_fast_2	-ffast-math	0.09312	0.001666	6.66×

5.5 Test2_fast_3: Separate convolution2D and Simplify Multiplications

This optimization separates `convolution2D` into two functions, horizontal and vertical, to simplify multiplications either by eliminating them (if they are with 0) or using shifts for multiplication by 2, similar to Test7. The effects of this change on execution time and speedup are shown in Table 18.

Table 18: Performance of test2_fast_3 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test2_fast_3	-ffast-math	0.33451	0.002476	17.98×

5.6 Test2_fast_4: Pointer based row access

This optimization uses row pointers in the horizontal and vertical `convolution2D` functions similar to Test8. The effects of this change on execution time and speedup are shown in Table 19.

Table 19: Performance of test2_fast_4 vs. baseline for `--ffast-math`

Version	Compiler Flags	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	-O0	0.620542	0.037342	1.00×
test2_fast_4	-ffast-math	0.033891	0.002476	18.31×

All other optimizations previously applied under `-O0` were also tested with `-ffast-math` but they either did not improve performance or resulted in slower execution and were therefore not retained in the final `-ffast-math` tests.

6 Failed or Ineffective Optimization Attempts

Several additional optimizations were tested during both the `-O0` and `-O3` compilation experiments but ultimately did not improve performance.

- **Loop unrolling in the main Sobel computation loop:** factors of 4, 8, 16, and 32 were attempted in the inner pixel loop over i and j, but all resulted in increased execution time.
- **Replacing `if` statements with ternary expression:** in the clipping operation

```
    if (res > 255)
        output[row0 + j] = 255;
    else
        output[row0 + j] = (unsigned char)res;

```

  was replaced with a ternary operator, which did not provide measurable speedup.

- **`restrict`, `const`, and `register` qualifiers:** applied in various parts of the Sobel function:
  - The input, output, and golden arrays (`unsigned char * restrict input/output/golden`)
  - Loop index and temporary variables (`register int p1, p2` and `register double t1`)
  - Row pointers inside the main loops (`const unsigned char * restrict row_m1/row0/row_p1`, `unsigned char * restrict out_row`, `const unsigned char * restrict gold_row`)

These optimizations were tested both individually and cumulatively with other optimizations. Overall they did not help execution time, although some marginal effects were observed with the compiler optimizations (restrict, const, registers, - as test13) and are included the following tables and graphs for completeness.

7 Final Experimental Results

This section presents the aggregated performance results for all optimization stages. Execution times, computed speedups and standard deviations are all summarized in tables and visualized in graphs for clarity. Each test corresponds to an additional optimization applied cumulatively on top of the previous ones, starting from the baseline Sobel implementation.

7.1 Results for O0 compilation

Table 20: Performance of Sobel without optimizations (-O0)

Test ID	Description	Average (s)	Std Dev (s)	Speedup
sobel_orig	Baseline	2,030373	0,145282	1,00×
test1	Loop interchange	1,445189	0,117923	1,40×
test2	Replace `pow()` with $x \times x$	0,881598	0,094736	2,30×
test3	Inlining `Convolution2D`	0,634364	0,011789	3,20×
test4	Loop unrolling	0,266311	0,006000	7,62×
test5	Common subexpression elimination	0,276984	0,017776	7,33×
test6	Loop-invariant code motion	0,271080	0,016552	7,49×
test7	Simplifying multiplications	0,206507	0,002645	9,83×
test8	Pointer-based access	0,185617	0,001414	10,94×
test9	Further pointer-based access	0,186470	0,001414	10,89×
test10	Loop fusion	0,222350	0,001000	9,13×
test11	Lookup table	0,098181	0,001000	20,68×
test12	Inlining lookup table	0,099140	0,003741	20,48×
test13	Added Restrict Const and Registers	0,098505	0,001414	20,61×

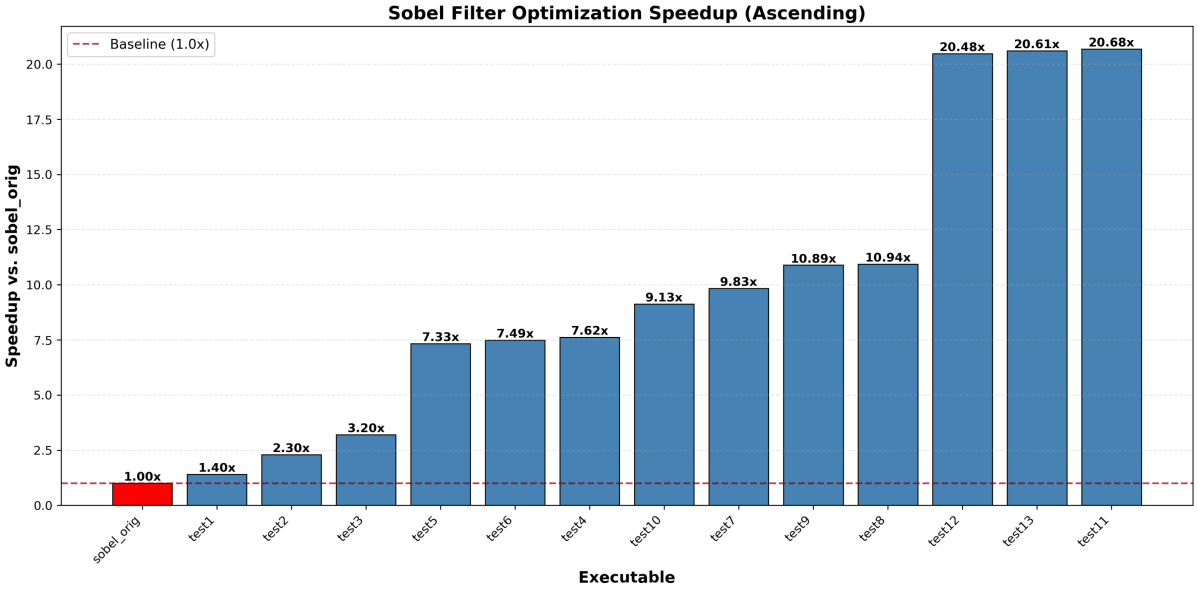

Figure 1: Speedup compared to original

Figure 2: Execution time compared to original

Figure 3: Standard deviation for every version

7.2 Results for -ffast-math compilation

Table 21: Performance of Sobel optimizations (-ffast-math)

Test ID	Description	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	Baseline	0.620542	0.035425	1.00×
test1	Loop interchange	0.142446	0.006633	4.36×
test2	Replace `pow()` with $x * x$	0.120539	0.008717	5.15×
test2_fast_1	Loop Unrolling	0.093389	0.005567	6.64×
test2_fast_2	Common Subexpression Elimination	0.093126	0.003872	6.66×
test2_fast_3	Separate convolution2D and Simplify Multiplications	0.034504	0.001414	17.98×
test2_fast_4	Pointer based row access	0.033891	0.002236	18.31×

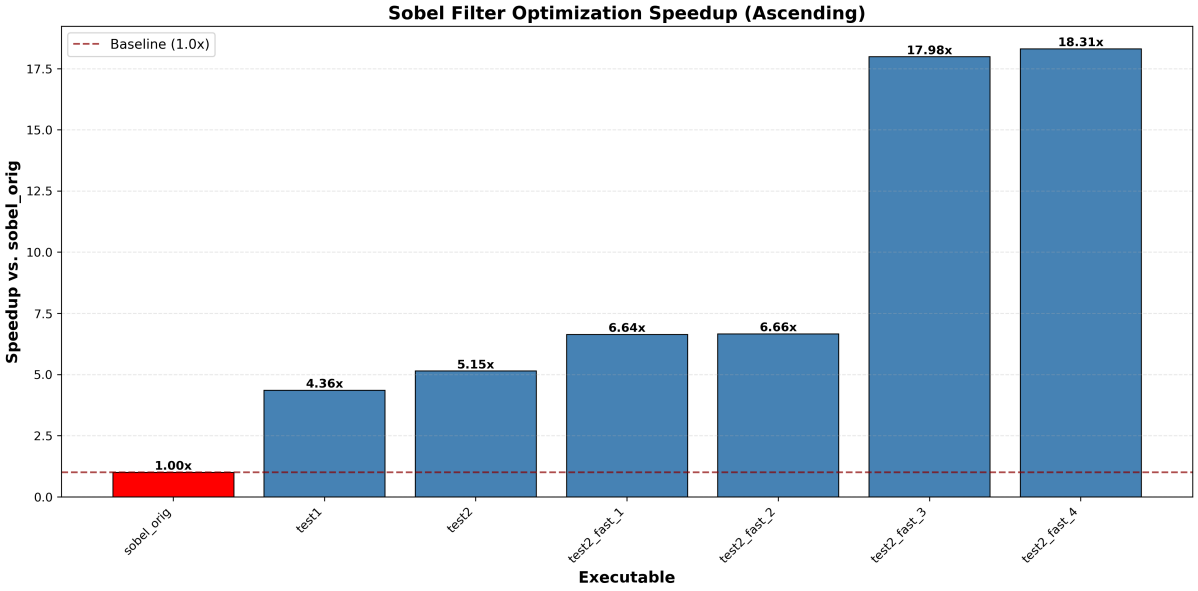

Figure 4: Compiler optimized speedup

Figure 5: Compiler optimized execution time

While the main focus of the `-ffast-math` experiments was on the set of optimizations above, for completeness we also incorporated the results of all the tests previously performed under `-O0`. These additional runs, were carried mainly out out of curiosity.

Table 22: Performance of Sobel optimizations for all(-ffast-math)

Test ID	Description	Avg Time (s)	Std Dev (s)	Speedup
sobel_orig	Baseline	0.620542	0.035425	1.00×
test1	Loop interchange	0.142446	0.006633	4.36×
test2	Replace `pow()` with $x * x$	0.120539	0.008717	5.15×
test2_fast_1	Loop Unrolling	0.093389	0.005567	6.64×
test2_fast_2	Common Subexpression Elimination	0.093126	0.003872	6.66×
test2_fast_3	Separate convolution2D and Simplify Multiplications	0.034504	0.001414	17.98×
test2_fast_4	Pointer based row access	0.033891	0.002236	18.31×
test3	Inlining `Convolution2D`	0.114456	0.001414	5.42×
test4	Loop unrolling	0.093215	0.002000	6.66×
test5	Common subexpression elimination	0.094136	0.003000	6.59×
test6	Loop-invariant code motion	0.092799	0.004242	6.69×
test7	Simplifying multiplications	0.034801	0.002449	17.83×
test8	Pointer-based access	0.036218	0.002236	17.13×
test9	Further pointer-based access	0.035226	0.003316	17.62×
test10	Loop fusion	0.036118	0.001414	17.18×
test11	Lookup table	0.042997	0.002000	14.43×
test12	Inlining lookup table	0.042269	0.001414	14.68×
test13	Added restrict, const, and registers	0.035759	0.001000	17.35×

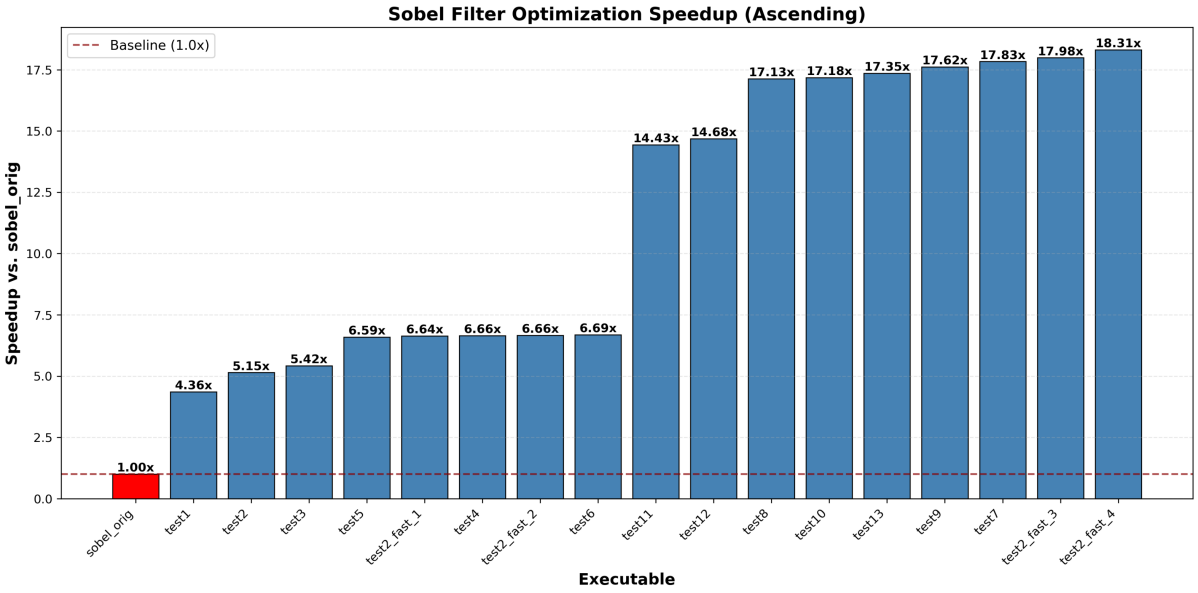

Figure 6: Speedup compared to original

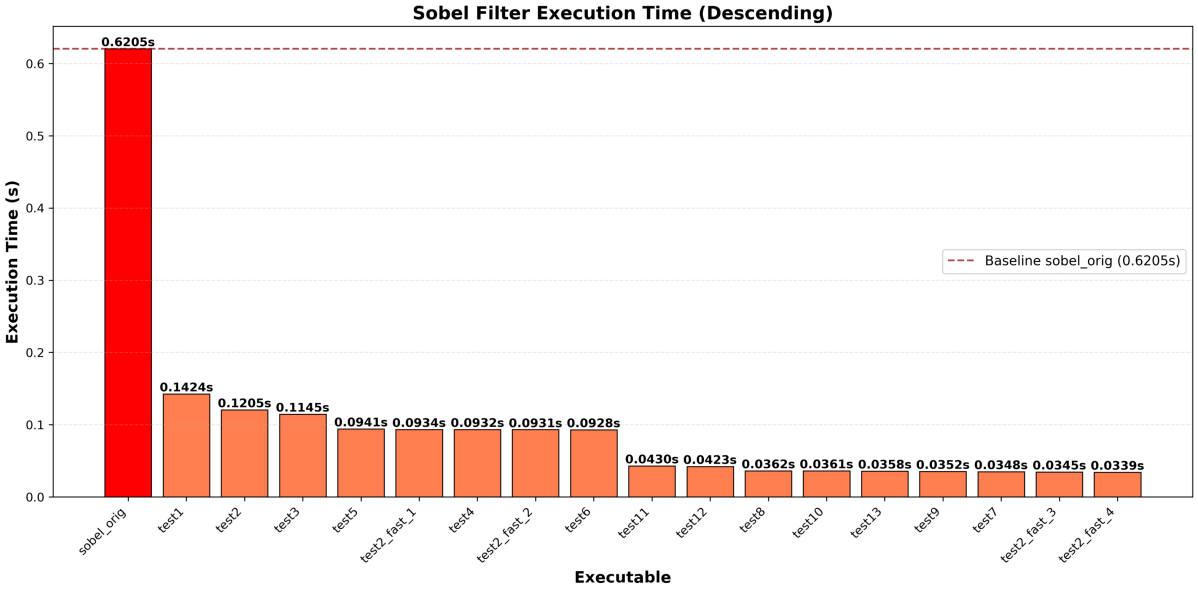

Figure 7: Execution time compared to original

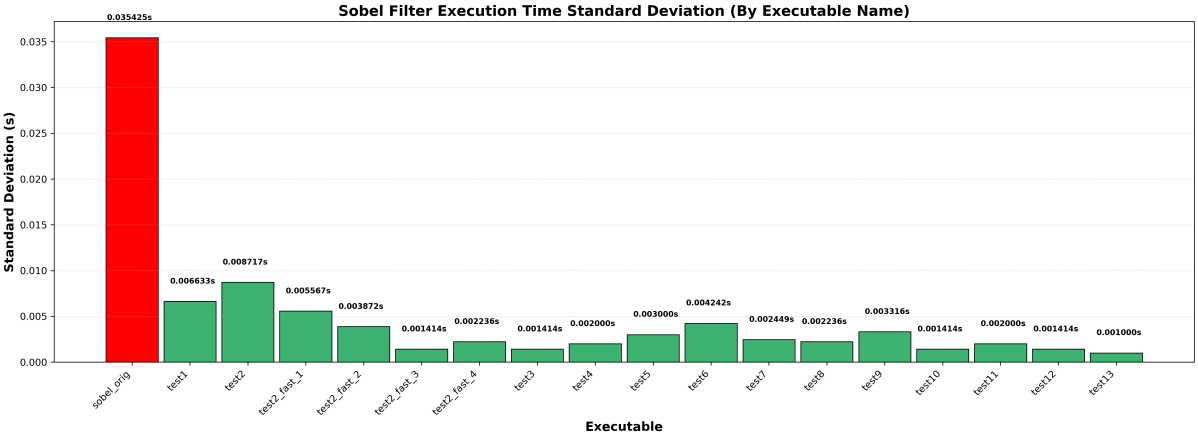

Figure 8: Standard deviation for every version

8 Conclusion

The results for both `-O0` and `-ffast-math` compilations show clear patterns of performance improvement as optimizations were applied cumulatively.

For the `-O0` compilation, the initial baseline execution time of 2.03 s was progressively reduced through successive optimizations. The most impactful contributions came from:

- **Replacing `pow()` with simple multiplication (test2)**: 64.3 speedup compared to test1 and 2 times faster than the original code
- **Inlining `convolution2D` and `Loop Unrolling` (test4)**: One of the most significant speedups, it halved the time compared to test2.
- **Loop unrolling, simplifying multiplications, and pointer-based row access (tests 7, 8)**: Collectively responsible for the largest cumulative speedup making it 10 times faster than the original code.

- **Lookup table implementation and inlining (tests 11, 12)**: Provided the final step to achieve over 20× speedup and the second most significant optimization after loop unrolling as we can observe 2 times the speedup compared to test10.

Standard deviations generally decreased with faster versions. Larger execution times showed higher variability, while heavily optimized tests were very stable across runs.

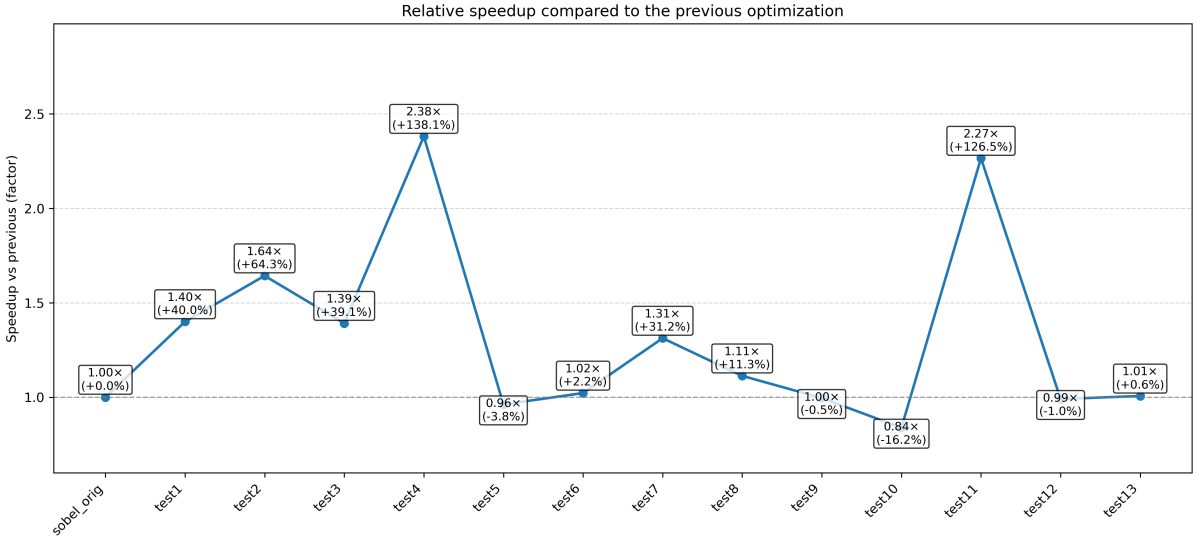

Figure 9: Relative Speedup

Under `-ffast-math`, the overall execution times were greatly reduced, with the baseline dropping to 0.62 s. The optimizations that yielded the highest speedups were:

- **Loop interchange (test1)**: the first major improvement, reducing execution time from 0.620 s to 0.142 s (4.36× speedup).
- **Separate convolution2D and simplification of multiplications (test2_fast_3)**: a later optimization producing another large incremental drop, from 0.093 s to 0.034 s (17.98× cumulative speedup) and 2.70x compared to convulution2D.

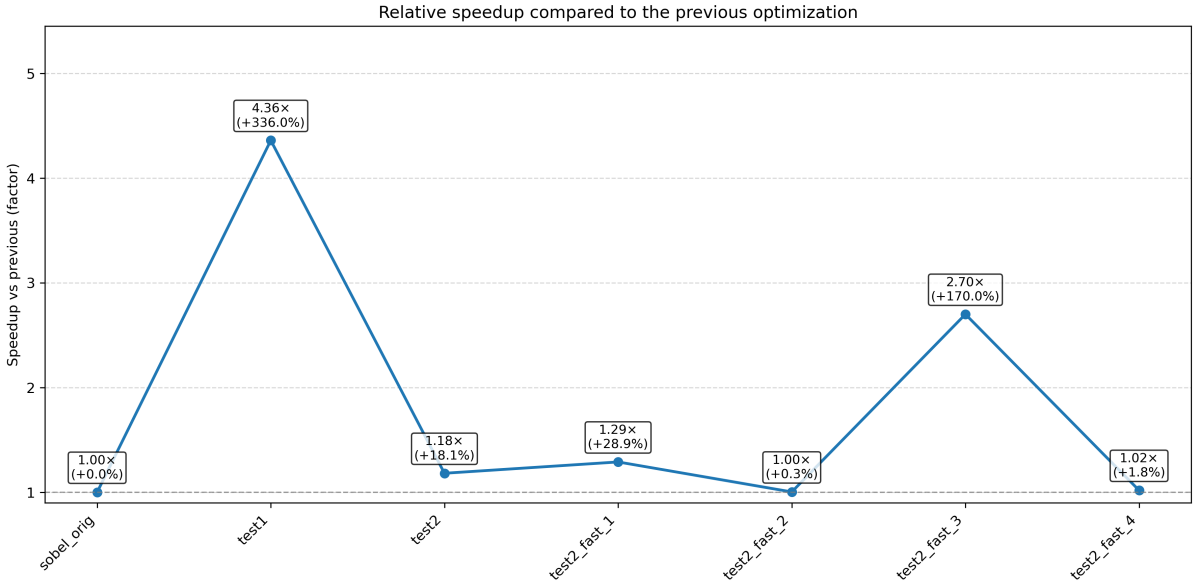

Figure 10: Relative Speedup

For `-ffast-math`, standard deviations were slightly higher compared to `-O0`, likely due to the compiler's aggressive optimizations and floating-point handling across the 12 measurement runs.

In summary, across both compilation modes, the optimizations that consistently produced the largest reductions in execution time were pointer-based memory access and arithmetic simplifications. Notably, optimizations that were highly effective under `-O0` did not necessarily yield improvements under `-ffast-math`.