# Lab1 Report: Optimization of Sobel Edge Detection

Christos Karagiannis 3544 - Ioannis Sideris 3418 - Athanasios Gerampinis 3466
University of Thessaly – Department of Electrical and Computer Engineering

October 21, 2025

## 1 System Information

The machine were the experiements took place have the following specifications:

- CPU: i7-12700H (24MB Cache)
- RAM: 16GB DDR5 4800MB/s
- OS: Ubuntu 24.04.3 LTS x86_64
- Kernel: 6.14.0-33-generic
- Compiler: Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1 (2025.2.0.20250806)

## 2 Optimization Techniques

The following optimizations were implemented to achieve full utilization of the available hardware and the compiler's resources.

### 2.1 Original Implementation (sobel_orig.c)

The baseline version represents the unoptimized Sobel filter implementation using nested loops for convolution. It serves as the performance reference.

### 2.2 Loop Interchange (sobel_loop_interchange.c)

Loop interchange changes the order of nested loops to improve cache locality by accessing memory in a sequential pattern, reducing cache misses.

### 2.3 Loop Unrolling (sobel_loop_unrolling.c)

This optimization expands the loop body to reduce loop overhead and improve instruction-level parallelism. It trades code size for speed.

### 2.4 Loop Fusion (sobel_loop_fusion.c)

Multiple adjacent loops operating over the same index range are fused into a single loop to minimize loop overhead and improve cache reuse.

### 2.5 Function Inlining (sobel_function_inlining.c)

Inline expansion replaces function calls with the function body, eliminating call overhead and enabling further compiler optimizations.

### 2.6 Loop Invariant Code Motion (sobel_loop_invariant.c)

Expressions inside loops that yield the same result in every iteration are moved outside the loop, avoiding redundant computations.

### 2.7 Common Subexpression Elimination (sobel_cse.c)

This technique identifies repeated expressions and stores their results once, reducing unnecessary recalculations.

### 2.8 Strength Reduction (sobel_strength_elimination.c)

Computationally expensive operations such as multiplications or divisions are replaced with cheaper alternatives like additions or bit-shifts.

### 2.9 Compiler-Assisted Optimization (sobel_compiler_assist.c)

This version uses compiler directives and flags (e.g. `-ffast-math`) to exploit hardware features and advanced compiler heuristics.

## 3 Experimental Results

All experiments were executed under identical system conditions. Each executable was run multiple times, and the mean and standard deviation of execution time were recorded.

### 3.1 Standard Execution Times

Table 1: Execution times of standard builds.

| **Executable** | **Mean Time (s)** | **Std. Dev. (s)** |
|---|---|---|
| sobel_orig | 1.5707 | 0.0200 |
| sobel_loop_interchange | 1.1545 | 0.0076 |
| sobel_loop_unrolling | 1.1877 | 0.0080 |
| sobel_loop_fusion | 1.2745 | 0.0052 |
| sobel_function_inlining | 0.7482 | 0.0093 |
| sobel_loop_invariant | 0.7649 | 0.0051 |
| sobel_cse | 0.8153 | 0.0115 |
| sobel_strength_elimination | 0.2244 | 0.0025 |
| sobel_compiler_assist | 0.2240 | 0.0022 |

Table 2: Execution times with aggressive compiler optimization flags.

| Executable (Fast) | Mean Time (s) | Std. Dev. (s) |
|---|---|---|
| sobel_orig_fast | 0.4381 | 0.0423 |
| sobel_loop_interchange_fast | 0.1020 | 0.0013 |
| sobel_loop_unrolling_fast | 0.1609 | 0.0011 |
| sobel_loop_fusion_fast | 0.1071 | 0.0012 |
| sobel_function_inlining_fast | 0.1663 | 0.0030 |
| sobel_loop_invariant_fast | 0.1663 | 0.0021 |
| sobel_cse_fast | 0.1556 | 0.0020 |
| sobel_strength_elimination_fast | 0.0912 | 0.0015 |
| sobel_compiler_assist_fast | 0.0320 | 0.0009 |

## 3.2 Optimized (Fast) Execution Times

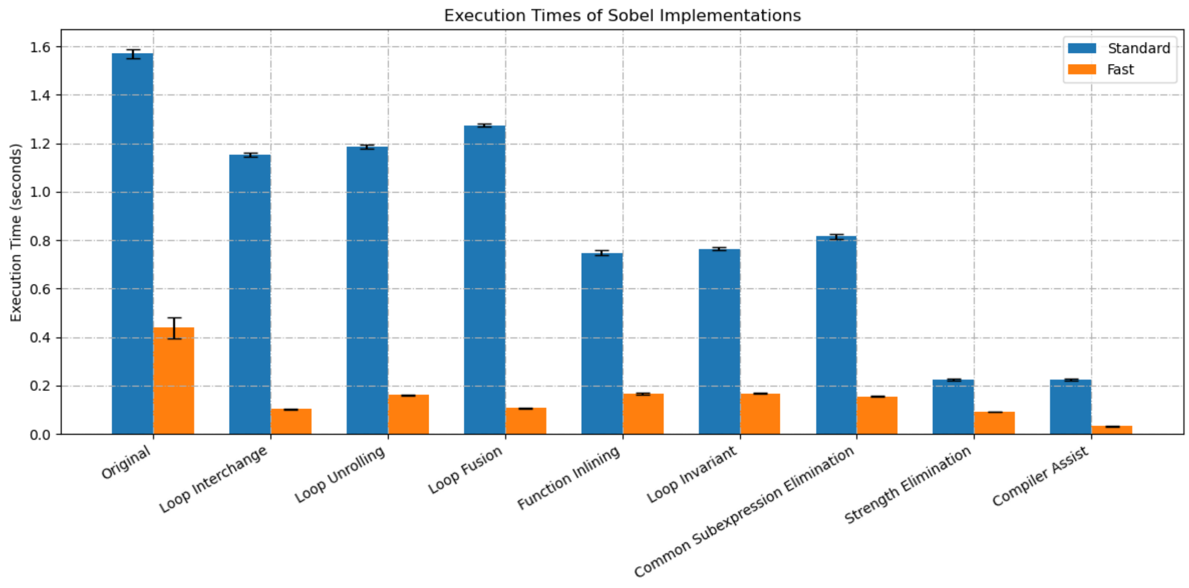

Figure 1:Execution Times of program with different optimizations applied.

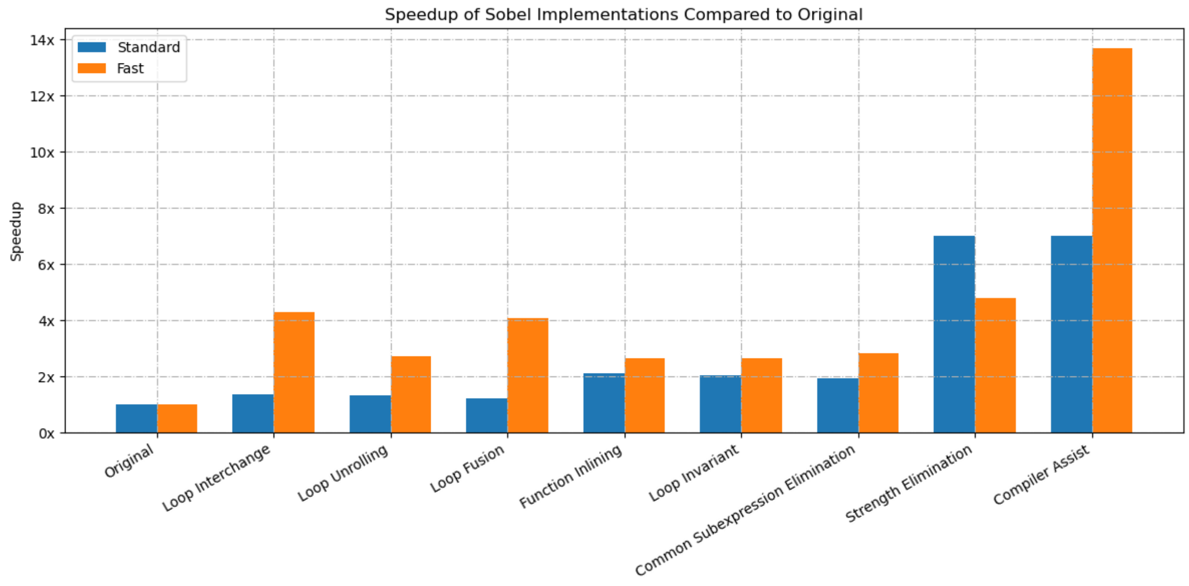

Figure 2: Speed Up achieved for each implemented optimization.

## 4 Performance Analysis

The results clearly show that, as more optimizations are applied, the overall performance of the program improves. The transition from the original to compiler-assisted versions demonstrates the power of both manual and automatic code transformations.

The most impactful techniques were:

- **Strength reduction** and **compiler-assisted optimization**, which achieved more than a $7\times$ speedup.
- **Loop interchange**, due to improved cache locality.
- **Function inlining**, which allowed compiler-level optimizations across function boundaries.

### 4.1 Speedup Comparison

Speedup relative to the original implementation:

$$S = \frac{T_{orig}}{T_{optimized}}$$

For instance, `sobel_compiler_assist_fast` achieved a speedup of approximately:

$$S = \frac{1.5707}{0.0320} \approx 49.1\times$$

This illustrates the dramatic performance gain achievable through combined manual and compiler optimizations.

## 5 Conclusion

This lab demonstrated the cumulative effects of various optimization techniques on the Sobel edge detection algorithm. While individual transformations such as loop interchange or inlining provide moderate gains, the combination of multiple strategies, alongside compiler optimizations, can achieve noteworthy performance improvements.