LAB 1

Christodoulos Zerdalis 03531 Charalampos Malakoudis 03516

October 20, 2025

1 Introduction

The objective of this lab was to optimize the execution of the Sobel filter running on a CPU. The optimizations were applied sequentially, following the order specified in the exercise instructions. In total, we implemented nine different versions, two of which employed approximate computing techniques.

2 System Specifications

- **CPU:** Intel Core i5-1035G1 @ 1.00GHz (4 Cores, 8 Threads)
- **Cache:** L1d: 192KB, L1i: 128KB, L2: 2MB, L3: 6MB (shared)
- **Memory:** 20GB @ 3200MHz
- **OS:** Ubuntu 24.04.2 LTS
- **Kernel:** 6.14.0-33-generic
- **Compiler:** oneAPI DPC++/C++ Compiler 2025.2.1

3 Optimizations

Here we describe the sequence of optimizations applied to the Sobel filter implementation. The performance metrics for each step are shown in a table directly following its description.

3.1 Loop Interchange (Version 1

For this optimization, there were three possible loops that could be interchanged one within the convolution function and two within the Sobel function. We first modified the loops in the convolution function; however, since the elements of the horiz and vert arrays fit entirely in cache, this change produced no noticeable performance improvement. We then interchanged the loops in the Sobel function, which resulted in a measurable performance gain.

Table 1: Performance: Loop Interchange

Flags	Time Before (s)	Time After (s)	St. Dev Before	St. Dev After
-O0	2.21	1.88	0.01	0.003
-Ofast	0.63	0.2	0.016	0.007

3.2 Loop Unrolling (Version 2)

For this optimization, we fully unrolled the double loop in the convolution function, which resulted in a noticeable performance improvement. We also attempted to unroll the inner loops of the Sobel function by factors, up to, four and eight. However, this led to worse performance. We believe this occurred because loop unrolling increased the number of calls by 4 to 8 times to the convolution, sqrt, and pow functions. These function calls are computationally expensive, and since most operations are performed within them, the compiler was unable to apply further optimizations.

Table 2: Performance: Loop Unrolling

Flags	**Time Before (s)**	**Time After (s)**	**St. Dev Before**	**St. Dev After**
-O0	1.88	1.27	0.003	0.002
-Ofast	0.2	0.13	0.007	0.006

3.3 Loop Fusion (Version 3

In this part we tried to fuse the two double loops inside the sobel function but it made no noticeable difference.

Table 3: Performance: Loop Fusion

Flags	**Time Before (s)**	**Time After (s)**	**St. Dev Before**	**St. Dev After**
-O0	1.27	1.27	0.002	0.002
-Ofast	0.13	0.13	0.006	0.0007

3.4 Function Inlining (Version 4)

For this optimization, we replaced the calls to the convolution2D() and pow() functions with their actual code. This change greatly improved performance. It might also be useful to inline the sqrt() function, which will be done in a later step.

Table 4: Performance: Function Inlining

Flags	**Time Before (s)**	**Time After (s)**	**St. Dev Before**	**St. Dev After**
-O0	1.27	0.38	0.002	0.0018
-Ofast	0.13	0.12	0.0007	0.0008

3.5 Loop Invariant Code Motion (Version 5)

In this part, we noticed that the calculation `(i - x) * SIZE` was being performed 18 times inside the j loop. To optimize this, we moved the computation to the i loop, so it is now executed once per row instead of 18 times per pixel. Interestingly, this change did not affect the execution time, which we believe is because the compiler automatically performs this optimization even though the optimization level was set to O0.

Table 5: Performance: Loop Invariant Code Motion

Flags	Time Before (s)	Time After (s)	St. Dev Before	St. Dev After
-O0	0.38	0.38	0.0018	0.003
-Ofast	0.12	0.12	0.0008	0.0006

3.6 Strength Reduction & Common Subexpression Elimination (Version 6)

We included both types of optimizations in this version because they are similar. In this version, we focused on simplifying every calculation inside the Sobel for loop. First, instead of performing the multiplication `(i - x) * SIZE` for every row, we initialized three variables `row_up`, `row`, and `row_down` with the appropriate starting values and then incremented them by SIZE in each iteration (e.g., `row += SIZE`). Second, instead of calculating `row_x + j + z` twice for every pixel, we stored the result in a variable and reused it. Finally, the most impactful optimization was replacing the table lookups `horiz_operator[x][y]` and `vert_operator[x][y]` with their actual constant values. This reduced the total number of calculations from 18 to 12 per pixel, since several of the table entries were zeros.

Table 6: Performance: Strength Reduction & CSE

Flags	Time Before (s)	Time After (s)	St. Dev Before	St. Dev After
-O0	0.38	0.26	0.003	0.001
-Ofast	0.12	0.039	0.0008	0.0005

3.7 Unrolling the Inner Loop (Version 7)

Up to this point, the improvements made in the O0-compiled version also resulted in similar gains for the Ofast version. However, in this case, we observed the opposite effect. In this version, we attempted to unroll the inner loop of the Sobel function by a factor of two. This optimization improved performance in the O0 version but decreased it in the Ofast version. We also reduced the total number of calculations from 12 to 9 after noticing that some `row_x + j + z` values overlapped, but this change did not produce any measurable difference, even in the O0 version.

Table 7: Performance: Inner Loop Unroll (x2)

Flags	Time Before (s)	Time After (s)	St. Dev Before	St. Dev After
-O0	0.26	0.189	0.001	0.001
-Ofast	0.039	0.077	0.0005	0.0006

3.8 Sqrt Approximation (Versions 6.1 & 7.1)

Lastly, we attempted to replace the `sqrt()` function with more efficient alternatives but found that the built in implementation was already very fast. Therefore, we tried approximating the function using a multi-level lookup table (LUT). This LUT used three separate tables one for the most significant bits, one for the middle bits, and one for the least significant bits to approximate the value of `sqrt()` linearly. This approach offers significantly lower memory usage but provides only an approximate result. We applied this method to both versions 6 and 7, and it improved performance in both cases. The resulting PSNR score was around 40, which indicates very good image quality.

Table 8: Performance: Sqrt Approx. on Versions 6.1 & 7.1

Version	Flags	Time Before (s)	Time After (s)	St. Dev Before	St. Dev After
6.1	-O0	0.26	0.25	0.0014	0.002
	-Ofast	0.039	0.067	0.0005	0.0007
7.1	-O0	0.189	0.152	0.001	0.0008
	-Ofast	0.077	0.067	0.0006	0.0006

4 Final results

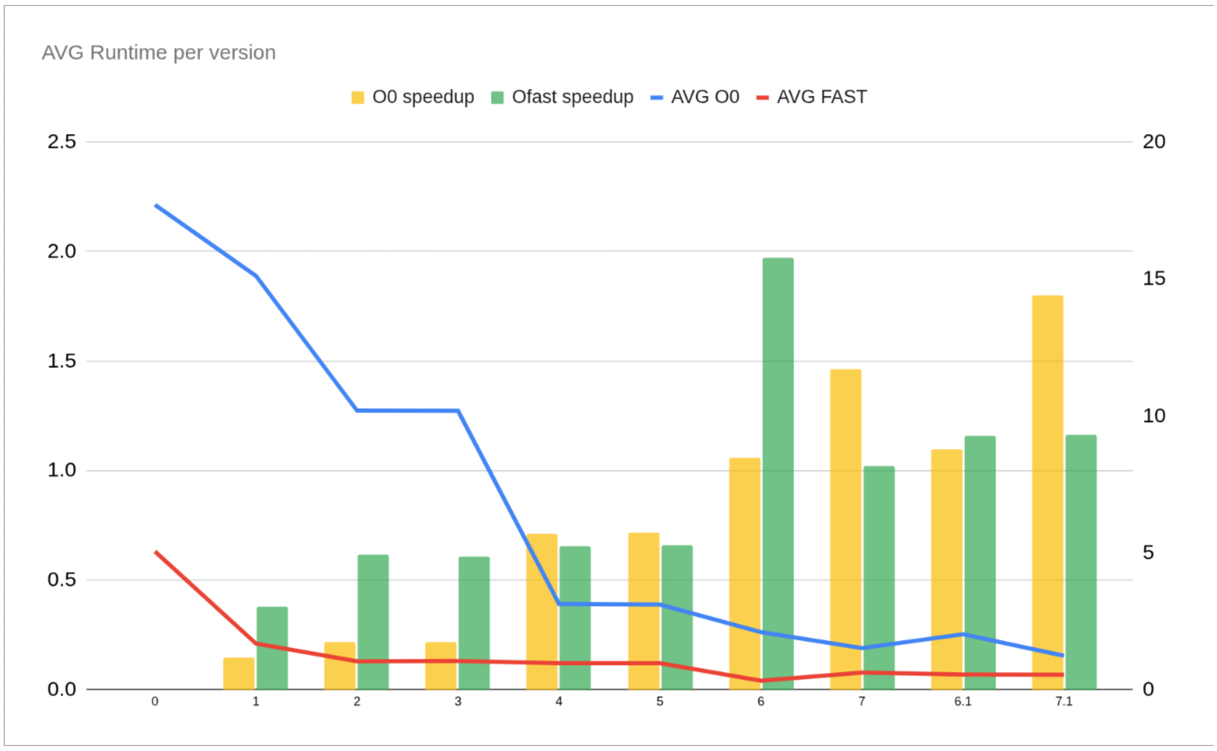

Figure 1: Performance and speed up per version