# ECE415 - High Performance Computing

**– Homework 1 –**

Code Optimizations on Sobel filter

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis

January 19, 2025

Aggeliki-Ostralis Vliora, Spyridon Liaskonis
avliora@uth.gr, sliaskonis@uth.gr
Student ID: 03140,03381

# Contents

# 1 Hardware specifications

## 1.1 CPU

The CPU used for the execution of the original as well as the optimised codes is an Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz.

## 1.2 Memory

### 1.2.1 Cache

As of the caches of the system, details are shown in Figure 1.

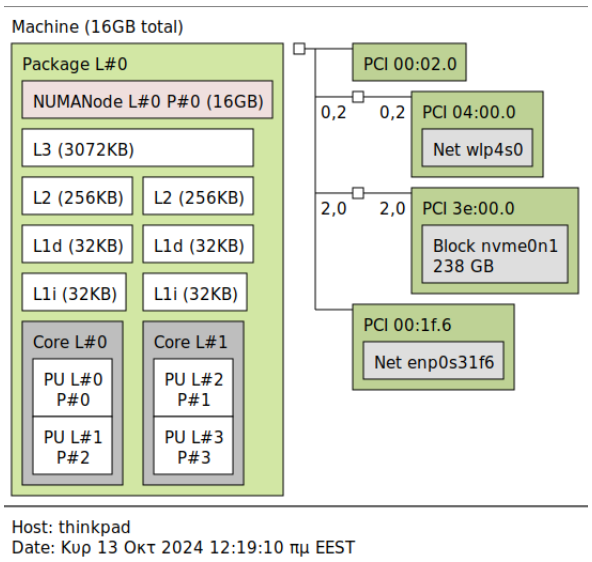


Figure 1: CPU Topology

### 1.2.2 RAM

The system is equipped with 16GB of SODIMM DDR4 RAM, split into two 8GB modules, both running at 2133 MT/s

### 1.3 System Specs

| Component | Details |
|---|---|
| OS | Ubuntu 22.04.5 LTS |
| Kernel | 6.8.0-45-generic |
| Compiler | ICX 2024.2.1 (Build 20240711) |

# 2 Code Optimizations

In the following subsections we analyze the optimizations applied on the original code for the Sobel filter. For each optimization we compare the execution times of the codes with the ones before the optimization. Each optimization aims to speedup the code. If a speedup is not gained, we discard the optimization and continue with the previous code.
The code is compiled using two flags: -O0 and -fast. The -O0 flag refers to no optimization level, whereas -fast applies aggressive optimizations in order to maximize the performance. We optimize the code based on the results of the code with no compiler optimizations since it better mirrors the original code, meaning that variables and function calls remain as they are written. However, in every result we compare the execution times of both -O0 and -fast compiled codes.

### 2.1 Loop-Interchange on sobel loop

The first optimization applied in the original code is a loop interchange. More specifically, the original code contains a nested loop inside the sobel() function that uses a column wise traversal. There, we apply a loop interchange in order to turn the loop traversal row wise, thus reducing the number of cache misses caused by reading/writing on the input and output arrays.
As expected, we get a performance speedup of 51.09% and 2355.4% for the -O0 and -fast compiled codes respectively.

### 2.2 Loop-Interchange on conv2d loop

In the same manner, we apply a loop interchange in the convolution2d loop which is responsible for computing the convolutions.

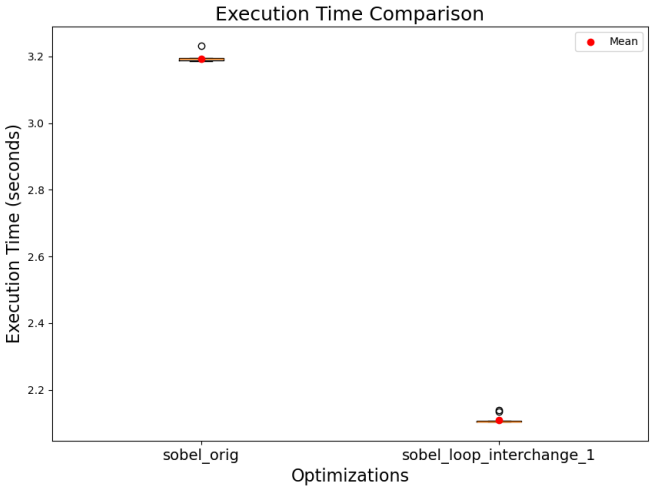


Figure 2: Loop Interchange Speedup - O0

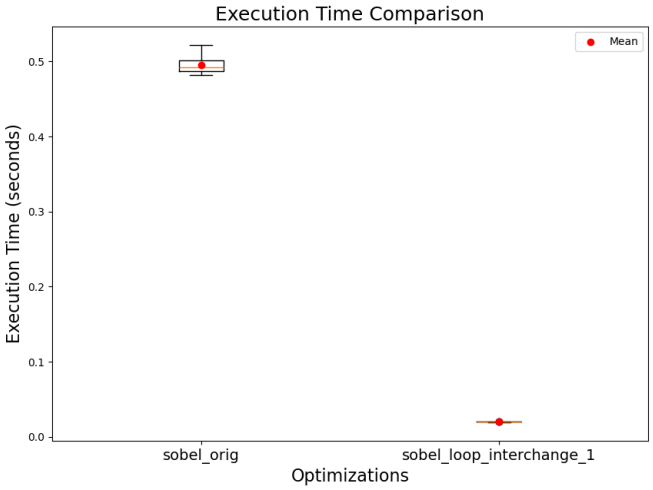


Figure 3: Loop interchange Speedup - fast

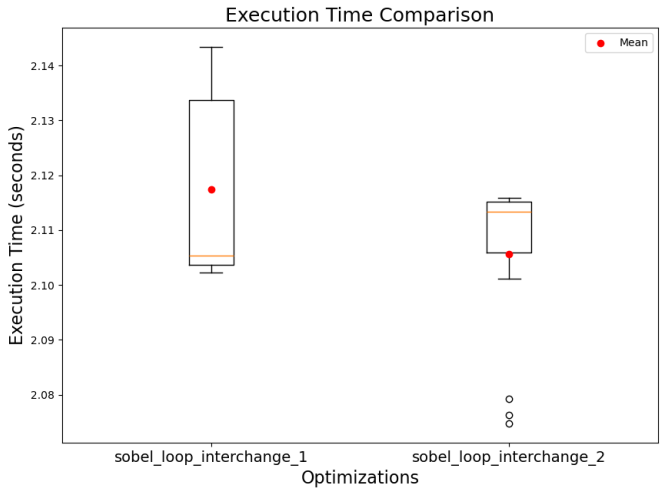


Figure 4: Loop Interchange 2 Speedup - O0

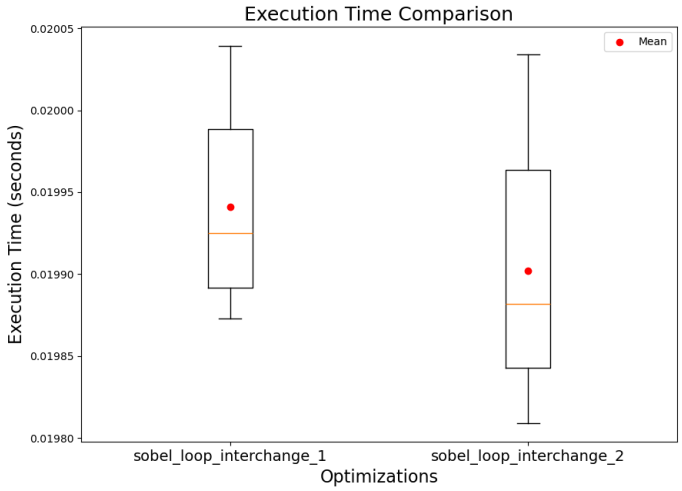


Figure 5: Loop Interchange 2 Speedup - fast

As seen in the box plots in figures 4 and 5, we get an additional speedup of 1.014947% and 1.00378% for both -O0 and -fast respectively. This is once more an expected result since that change helps exploit memory locality.

## 2.3 Loop-Unroll on conv2d loop

Since both of the previous opitimizations were successful, we keep them and we add one more on top of them. More specifically, we fully unroll the loop inside the convolution2d function. Doing so, we completely eliminate the loop control

overhead, which as seen in the figures bellow gives us a performance bump of 27.57% and 0.3735% for -O0 and -fast respectively.

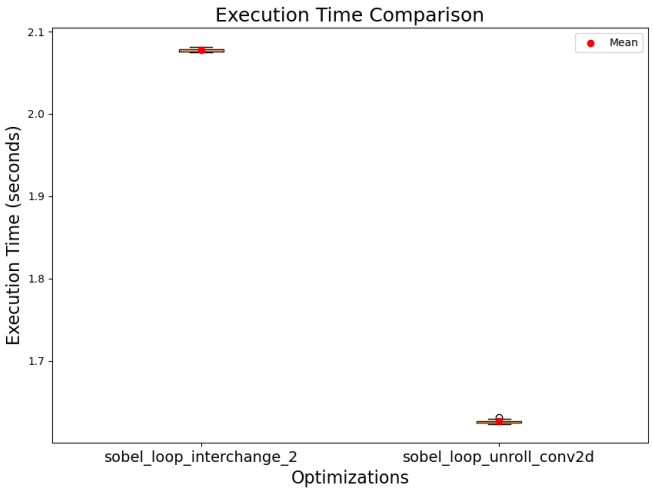


Figure 6: Loop Unroll Conv2d - O0

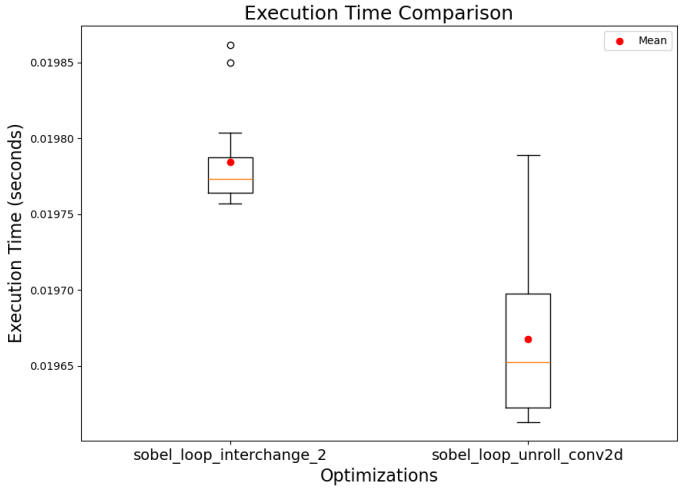


Figure 7: Loop unroll Conv2d - fast

## 2.4 Function inlining

Next, we eliminate the convolution2d function by applying function inlining, which reduces function call overhead and provides additional performance gains. More specifically we get a performance gain of 8.445% and -6.7% for -O0 and -fast respectively.

## 2.5 Strength Reduction - multiplication elimination

In the sobel function, we observe that the indexing is achieved using the following formula:

$$index = i \cdot \text{SIZE} + j. \tag{1}$$

At this point, we want to eliminate the multiplication and replace it with a cheaper operation. In our case, we replace the multiplication with one additional addition. More specifically:

1. Initialize index = SIZE - 2
2. Increase index as index += 2, on the outer loop
3. Increase index as index += 1, on the inner loop

This way we get the same results that we did with the multiplication, only now we use one much cheaper operation.

Doing so we get a performance gain of 0.65% for -O0. For -fast the performance gain is almost negligible. Figures 8 and 9 show the results for both optimizations.

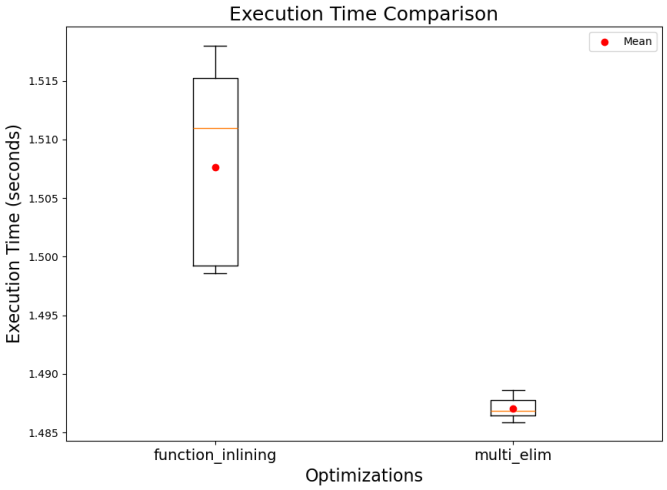


Figure 8: Strength Reduction - O0

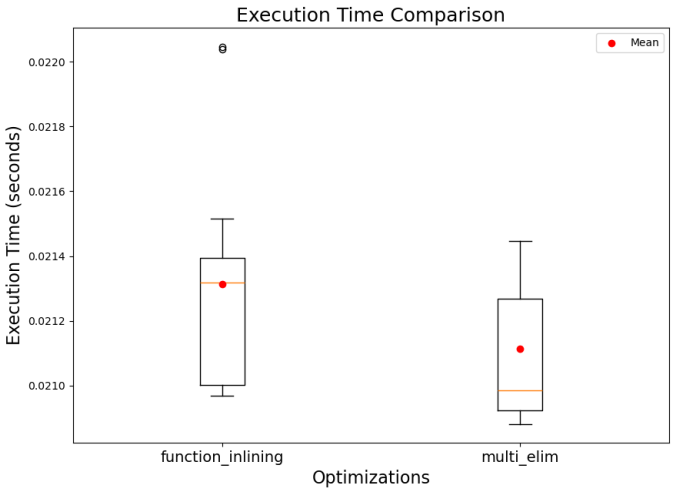


Figure 9: Strength Reduction - fast

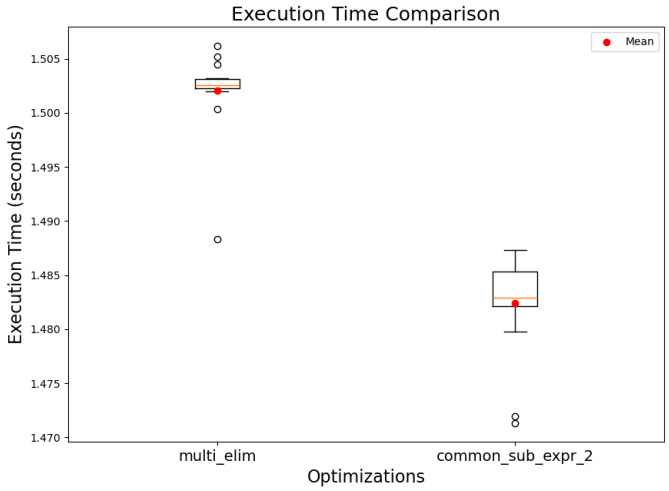


Figure 10: Common Subexpression Elimincation on Convolutions - O0

## 2.6 Common-Subexpression - Elimination on conv2d function

Then, we optimize the Sobel function by applying common subexpression elimination to the convolution calculations. During convolution operations, many computations are repeated multiple times. To reduce redundancy and overhead, we store these repeated operations in a temporary variable. This allows us to compute the common subexpressions once and reuse them, improving efficiency without recalculating the same values multiple times.

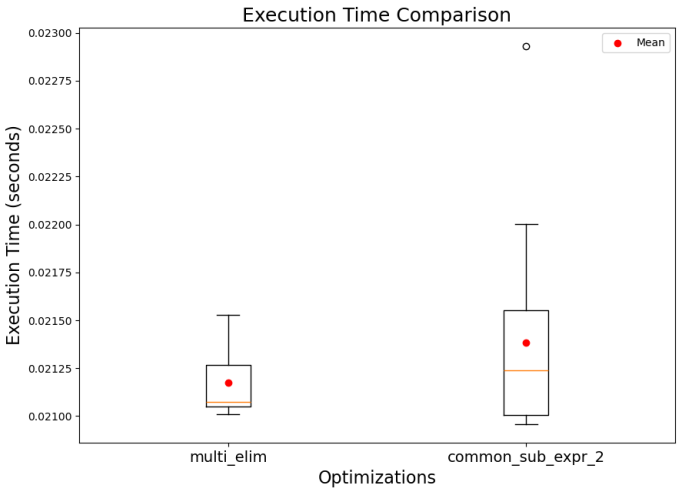


Figure 11: Common Subexpression Elimincation on Convolutions - fast

As shown in Figure 10, the -O0 code achieves a significant performance improvement, with an observed increase of 1.425%. However, the execution times for the -fast code increase. This can be attributed to the reuse of variables for common subexpressions, which introduces additional data dependencies between convolution calculations. These dependencies restrict the compiler's ability to fully leverage instruction-level parallelism, resulting in slower performance.

## 2.7 Replace sobel horizontal and vertical arrays

The sobel algorithm uses the following two matrices:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

These two matrices are applied to an image to compute the gradients in the x and y directions. In the original algorithm we create two 2D arrays where we store the values for each matrix. For each pixel in the image, we multiply the 3x3 surrounding pixels by the $G_x$ and $G_y$ matrix and sum up the results.

After loop unrolling, we avoid iterating through the arrays for multiplication because we already know the exact element needed for each operation. This allows us to completely eliminate the matrices and directly substitute their values into the calculations. For example, instead of writing:

$$res+ = input[temp - 1] * horiz_{operator}[0][0] \tag{2}$$

we replace it with

$$res+ = input[temp - 1] * (-1) \tag{3}$$

By doing this, we not only remove the memory overhead associated with these arrays but also skip computations involving zero values, further reducing the overall computational complexity.

With this optmization, we get an additional performance speedum of 4.744% and 3.5485% for -O0 and -fast respectively (shown in Figures 12 and 13).

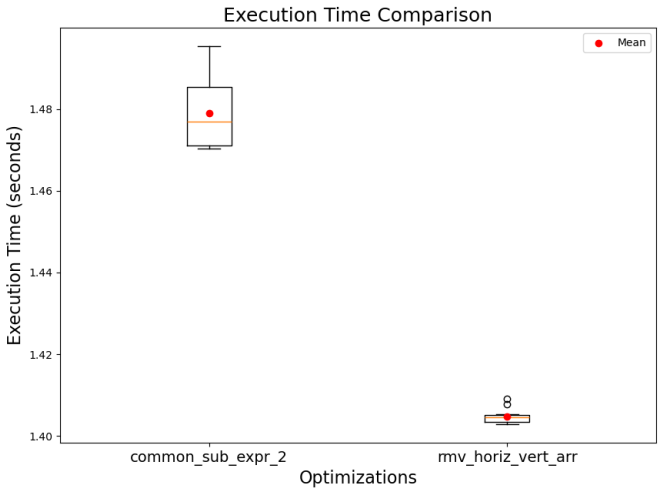


Figure 12: Substitute $G_x$ and $G_y$ arrays with its values - O0

## 2.8 Strength Reduction

Next, we can also substitute most of the multiplications applied with left shifts. For example, every multiplication such as $i \cdot SIZE$, can become $i \ll 2^{12}$ (since SIZE has a constant value of 4096). In this step, we eliminate all such multiplications, and replace them with left shifts.


Figure 13: Substitute $G_x$ and $G_y$ arrays with its values - fast

As expected, we again get additional performance gains of 0.387% for -O0, whereas for -fast the speedup is close to 0.


Figure 14: Replace multiplications with shifts - O0

## 2.9 Compiler Assist

One more way to optimize our code is by informing the compiler about certain optimizations. In this part, we use the following compiler optimizations on the previous code:

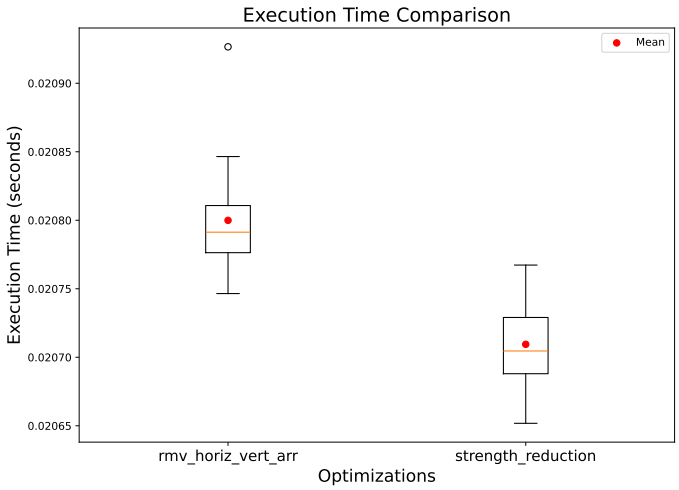


Figure 15: Replace multiplications with shifts - fast

1. Restrict keyword: we use the restrict keyword on pointers such as input, output and golden, since the memory locations pointed to by these pointers will only be accessed by them throughout the whole code.

2. Registers: we hint the compiler that some variables such as **i**, **j** iterators and **res** (variable used on the convolution calculations) should be kept in registers due to their extensive use throughout the code.

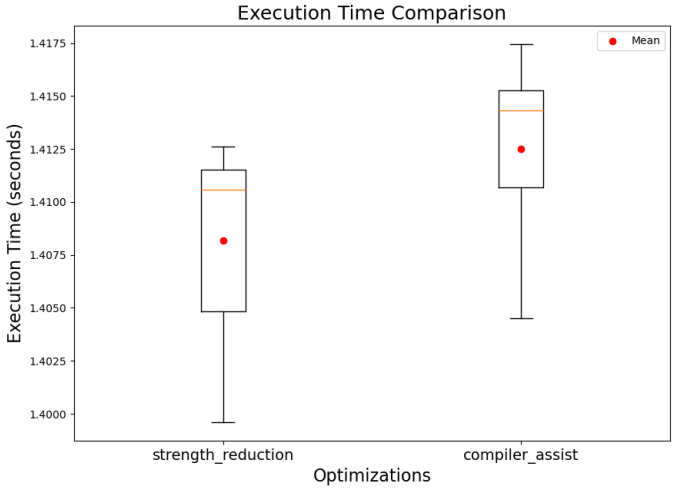


Figure 16: Compiler assist - O0

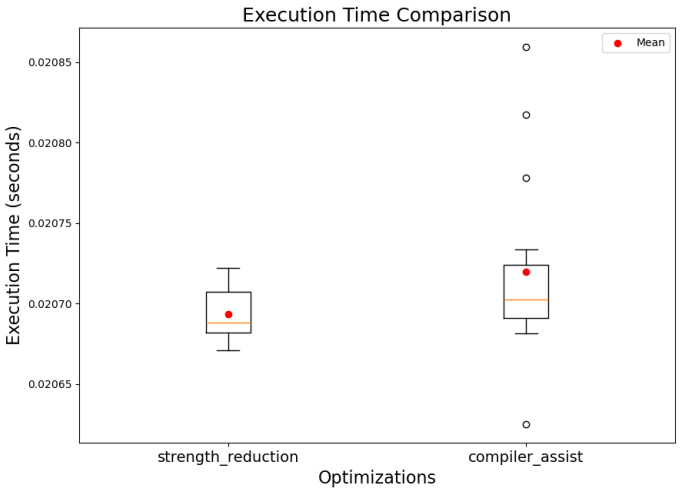


Figure 17: Compiler assist - fast

From Figures 16 and 17 we observe that the compiler optimizations here didn't give us any additional speedup, in contrast we got a small performance hit of -0.4% which is however negligible.

## 2.10 Strength Reduction - pow function replacement

Inside the code, we have three calls of the pow function. More specifically, we call the pow function as pow(x, 2) in order to square a number. According to man page, the pow function uses internally floating-point values which in our case is an extra overhead since for 2/3 times that we call the pow function, we call it for integer values.
In order to get rid of this extra overhead (plus the extra function calls), we replace the pow function with a multiplication operation.
This significantly boosts our performance in both -O0 and -fast with a speedup of 238.56% and 20.6216%.


Figure 18: Strength Reduction, Pow function replacement - O0

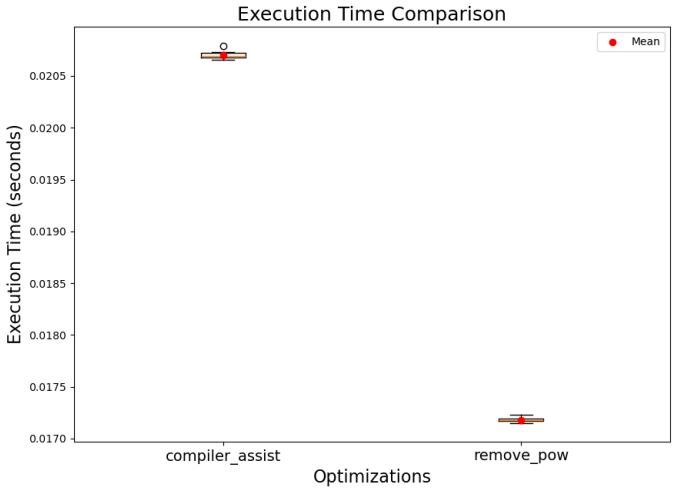


Figure 19: Strength Reduction, Pow function replacement - O0

## 2.11 Loop Invariant Code Motion

Loop invariant code motion, is an optimization where we move part of the code outside the loop, for operations that aren't affected by the loop's iterations.
In our case, we apply a similar optimization, but instead of moving the code outside the two loops, we move it inside some if statements, since that code doesn't need to be computed all the time.
More specifically in the original code:

```
1 res = (int)sqrt(p);
```

```
if (res > 255)
    output[it] = 255;
else
    output[it] = (unsigned char)res;
```

the sqrt() operation needs to be computed only when res is greater than 255, and when this is not the case, we simply clip the value 255 directly in the output. Thus, we can move the sqrt operation inside the else statement. Doing so, we limit this operations to only execute when the code joins the else statement and not throughout all the loops.

```
if (p > 65025)
    output[it] = 255;
else {
    output[it] = (unsigned char)sqrt(p);
}
```

This results in our performance gain of 2.442% and 3.2471% for -O0 and -fast respectively.

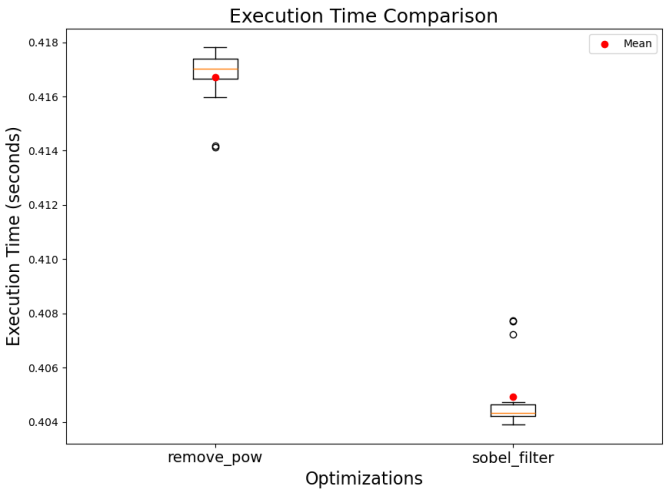


Figure 20: Loop invariant code motion - O0

## 2.12 Loop fusion

Currently, we have two loops that iterate through the same ranges. One is the loop responsible for the convolutions and the second one is responsible for calculating the PSNR based on the comparison between our output and the golden output.

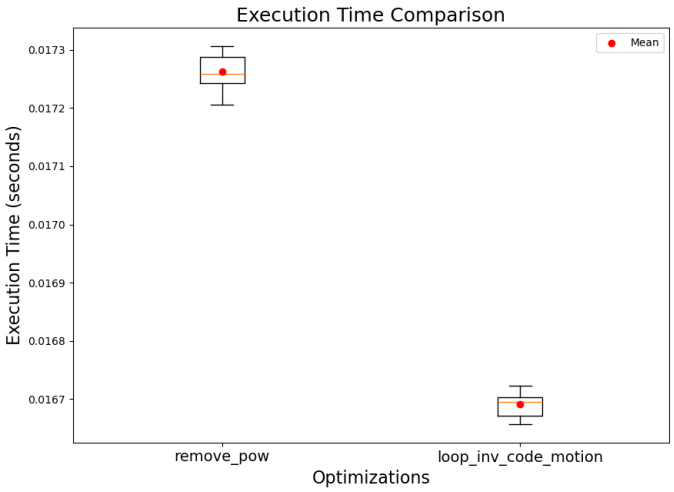


Figure 21: Loop invariant code motion - O0

We can fuse these two loops into one, thus completely eliminating the loop overhead of the second loop (increasing however the loop complexity on the new main loop).

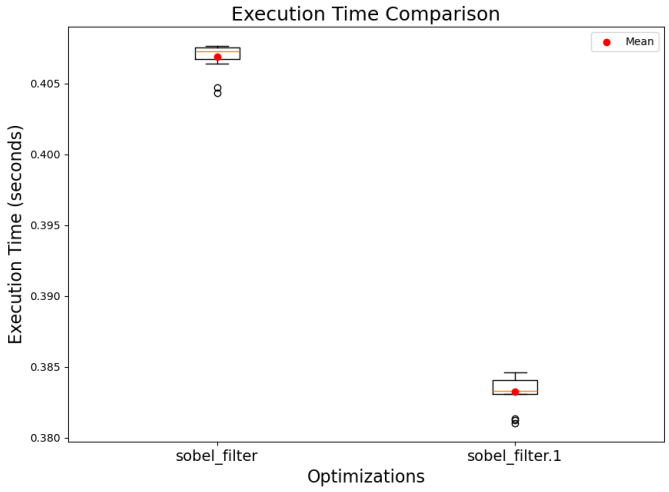


Figure 22: Loop fusion - O0

From Figure 22 we observe the the loop fusion gives us an additional performance speedup of 6.115% for the -O0 executions. In contrast, the -fast execution didn't have any significant performance gain or hit as seen in Figure 23.

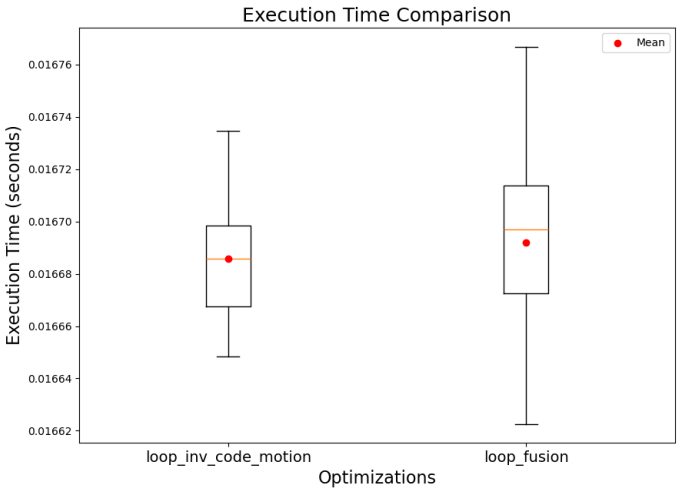


Figure 23: Loop fusion - O0

## 2.13 Final Minor Optimizations (strength reduction, common subexpression elimination)

Finally, we apply some more minor optimizations on the final code. More specifically we apply the following:

1. Common-subexpression-elimination: at the end of the fused loop we calculate the PSNR as:

$$sub = output[(i << 12) + j] - golden[(i << 12) + j].$$

   Here, $(i << 12) + j$ is the same as the it (iterator) variable we use throughout this loop for indexing. Thus we substitute this operation with the it variable.

2. Strength reduction: at the end of the code we use the following operation in order to calculate the final PSNR:

$$PSNR = 10 \cdot log_{10}\left(\frac{65536}{PSNR}\right).$$

   Using logarithmic properties we can turn this into:

$$PSNR = 10 \cdot log_{10}(65536) - log_{10}(PSNR),$$

   where for $10 \cdot log_{10}(65536)$ we substitute the value 48.164799.

With these final optimizations we get an additional speedup of 2.434% for the -O0 execution (Figure 24). As of the -fast execution (Figure 25 the final times are almost identical to the previous ones, thus no additional performance gains were achieved here.

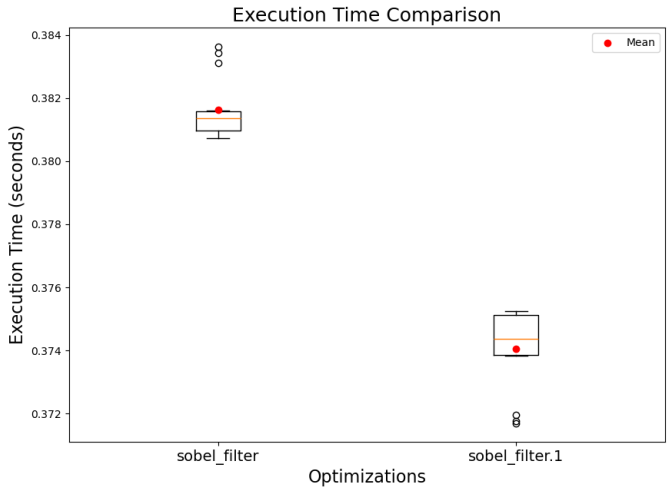


Figure 24: Final optimizations - O0

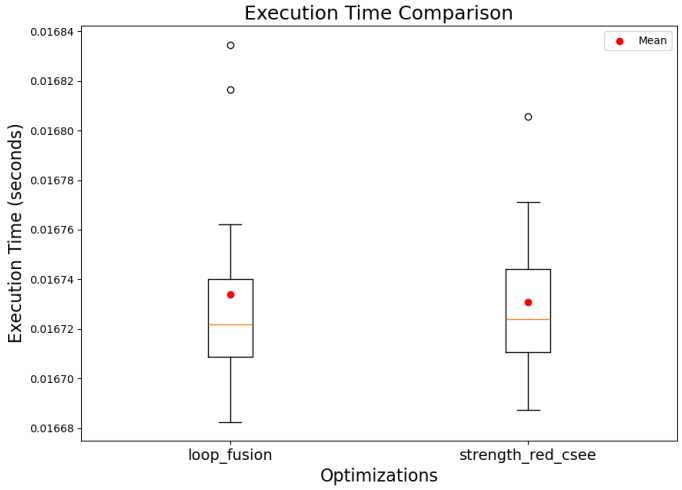


Figure 25: Final optimizations - fast

# 3 Final Results - Comparison

Figures 26 and 27 compare all optimizations, presented in the order they were applied.
A significant improvement is observed for the -O0 execution, with an overall speedup of 755.21%.
As of the -fast execution, we still get a performance gain of 2869.9%, which is mostly observed from the first optimization. After this optimization, the next ones had small to no effects on the overall average execution time.

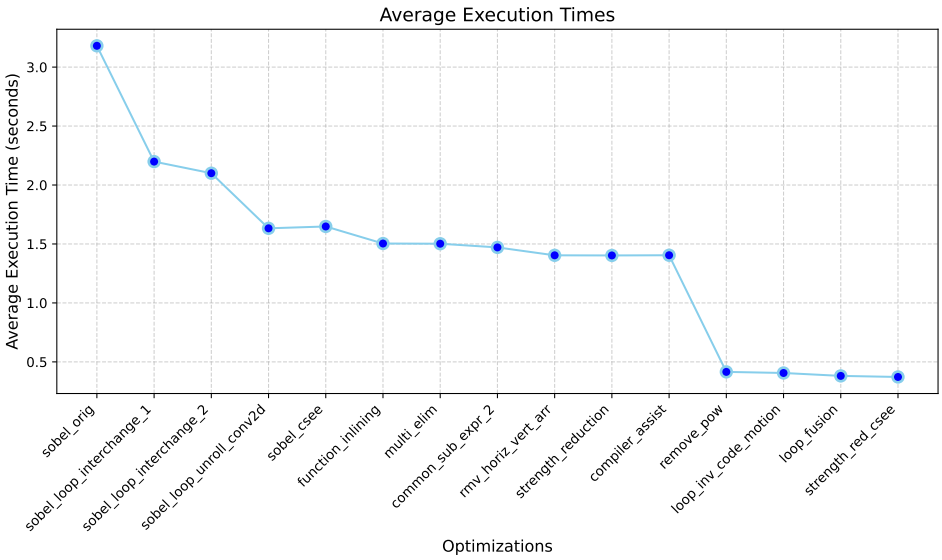


Figure 26: Average execution times for each optimization - O0

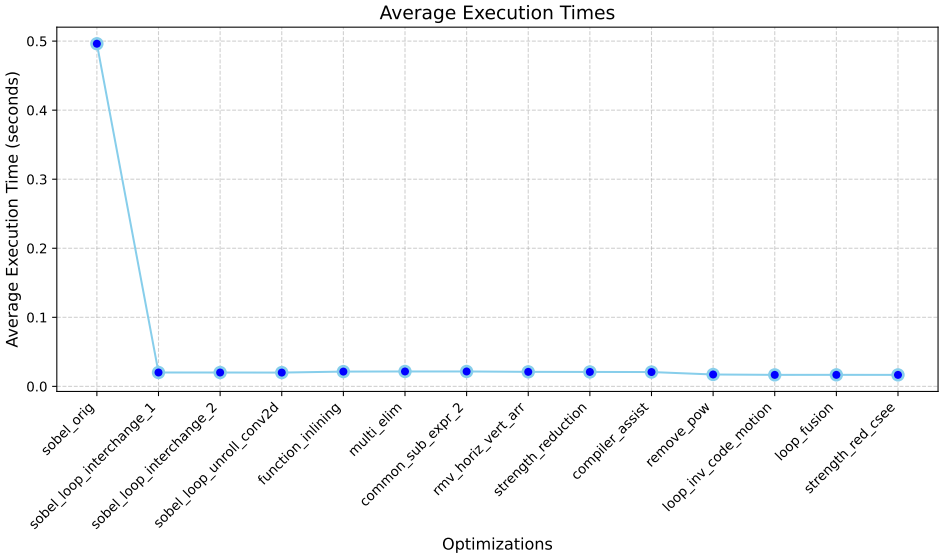


Figure 27: Average execution times for each optimization - fast