Department of Electrical and Computer Engineering

University of Thessaly


ECE415: High Performance Computing (HPC)

# Lab 1: Code optimizations on Sobel Filter

**Ioanna-Maria Panagou, 2962**

**Nikolaos Chatzivangelis, 2881**

Fall Semester 2022-2023

# Contents

# 1 Summary

In this lab, we were tasked with performing standard optimizations on an implementation of the sobel filter that detects edges in grayscale images. We progressively incorporated the optimizations that improved the execution time and reached a speedup of x9.56 compiled with -O0 from the baseline when compiling with -O0, x23.77 from the baseline when compiling with -fast and x123.34 when going from the baseline -O0 implementation to the fastest -fast implementation.

# 2 Introduction

The heart of the sobel filter is the sobel operator which is a discrete differentiation operator that approximates the gradient of the image intensity. After reading the input image, the sobel operator is applied to every pixel and the result is stored in the output image. The sobel operator uses 2 3×3 arrays $O_x$ for edges in the horizontal plane and $O_y$ for edges in the vertical plane and performs two-dimensional convolution with the input image. More specifically, if $A$ is the input image and $Y$ the output image, then:

$$O_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & -2 \\ -1 & 0 & 1 \end{bmatrix},\ O_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix},\ G_x = O_x * A,\ G_y = O_y * A,\ Y = \sqrt{{G_x}^2 + {G_y}^2}$$

The output image is padded with zeros at the edges, because the filter would slide off the image at these locations, as shown in fig.1

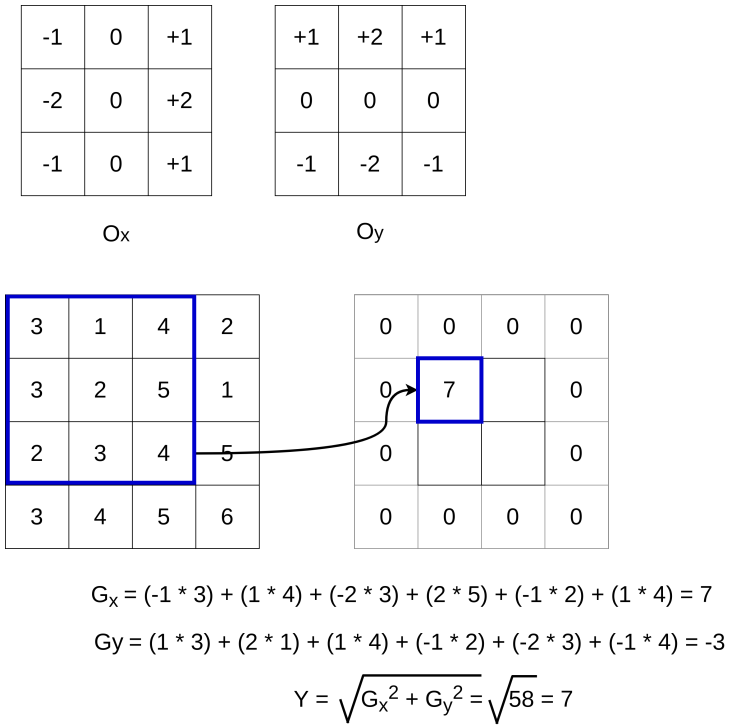


Figure 1: Update example for one output pixel

The implementation of the sobel filter handed to us consisted of two functions: the *sobel* function and the *convolution_2D* function. The *sobel* function is comprised of a nested loop that traverses the output image and, for each pixel, calls the *convolution_2D* function two times, one for the horizontal and one for the vertical operator, performs the computation for the value of the output pixel, which is the square root of the sum of the squared outputs of the *convolution_2D* function and clamps the result at 255, which is the maximum pixel value for grayscale images. After that, it traverses the image again to compute the MSE (mean squared error) between the output image and the golden output as:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [output(i,j) - golden(i,j)]^2$$

, where $m \times n$ is the size of the images.

Finally, the PSNR (peak signal to noise ratio) is used to compare the quality of the output and the golden image and is defined as:

$$PSNR = 10 \log_{10} \left( \frac{MAX^2}{MSE} \right)$$

, where $MAX$ is the maximum value of a pixel (255 for grayscale images). A correct implementation should have an MSE equal to zero and an infinite PSNR.

The remainder of our report is structured as follows: section 3 describes the process we followed to evaluate our optimizations, section 4 lists the specifications of the system we performed our experiments, section 5 presents our proposed optimizations steps and section 6 concludes

the report.

# 3 Evaluation

We cumulatively apply optimizations on the original source code. After each optimization, we stop and assess the correctness of our output, making sure that our PSNR is infinite, and the efficacy of our implementation. If a modification slows down the code, we do not include it in the next configuration. We run our algorithm 25 times, we calculate the first ($Q_1$) and third quartile ($Q_3$) of our measurements, the inter-quartile range as $IQR = Q_3 - Q_1$ and compute the mean (which corresponds to the median for our method) and the standard deviation for the observations that are not outliers, meaning they are within the interval ($Q_1 - 1.5 \cdot IQR$, $Q_3 + 1.5 \cdot IQR$).

# 4 System Specs

Here is an overview of our system as outputted by the *lstopo* command:

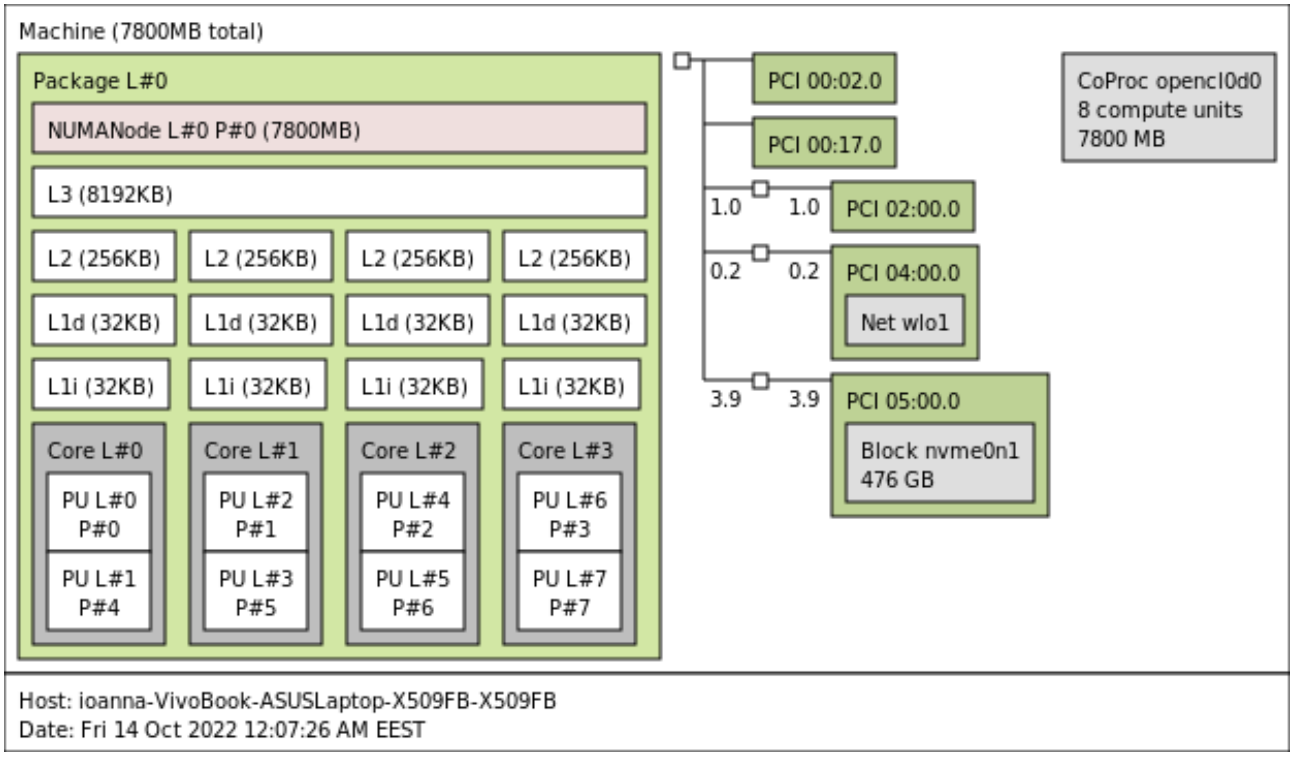


Figure 2: *lstopo* output

## 4.1 CPU

| **Vendor** | Intel |
|---|---|
| **Code Name** | Whiskey Lake-U (Core i7) |
| **Architecture** | x86_64 |
| **Specification** | Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz |

Table 1: CPU information

## 4.2 Caches

| **L1 Cache** | 4 x 32 kB, 8-way associative, 64-bytes line size |
|---|---|
| **L2 Cache** | 4 x 256 kB, 4-way associative, 64-bytes line size |
| **L3 Cache** | 8 MB, 16-way associative, 64-bytes line size |

Table 2: Cache information

## 4.3 Memory

| **Total** | 8GB |
|---|---|
| **Bank 0** | 4GB @ 2400 MHz (SODIMM DDR4) |
| **Bank 1** | 4GB @ 2667 MHz (SODIMM DDR4) |

Table 3: Memory Information

## 4.4 System

| **Kernel** | Linux 5.15.0-50-generic |
|---|---|
| **Distribution** | Ubuntu 22.04.1 LTS |

Table 4: System information

For our experiments, we used the Intel compiler **icc (ICC) 2021.7.0 20220726**

# 5 Optimization Steps

## 5.1 Loop Interchange

The most obvious enhancement to perform first was the loop interchange. In the original code, there were two spots where we could change the nested loops ordering, intending to reduce the number of cache misses, exploiting memory locality.

**`Convolution_2D`**

First, we converted the column wise traversal of the loop, in function `convolution_2D`, to row wise. To our surprise, this dramatically reduced the code performance for both -O0 and -fast flags as we can see from 3. This function obtains the index of the input image and, using the pixels around it, performs a convolution, as shown in fig 4.


Figure 3: Execution Time for Original vs Loop Interchange (Convolution_2D)

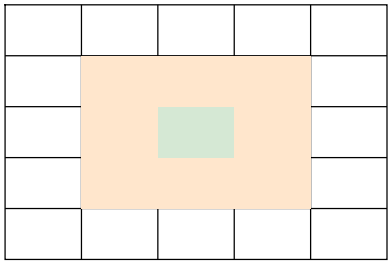

Figure 4: Given the green pixel, `convolution_2D` will traverse the orange pixels too.

**`Sobel`**

After our unsuccessful try at `convolution 2D`, we switched to the `sobel` function, where we rearranged the loops to read the input image row by row. This time, we significantly outperformed the original in terms of speed, which makes sense given that we significantly reduced the frequency of cache misses. We observed a speedup of 1.27 for -O0 and 8.65 for -fast.

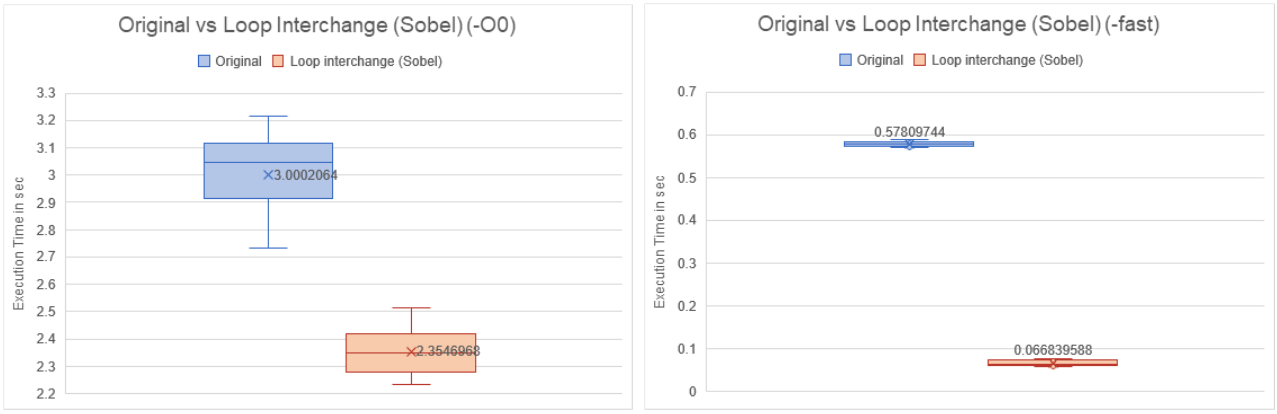


Figure 5: Execution Time for Original vs Loop Interchange (Sobel)

**`Convolution_2D + Sobel`**

Interestingly enough, if we combined the two approaches above, we gained an even better performance as shown in fig. **??**

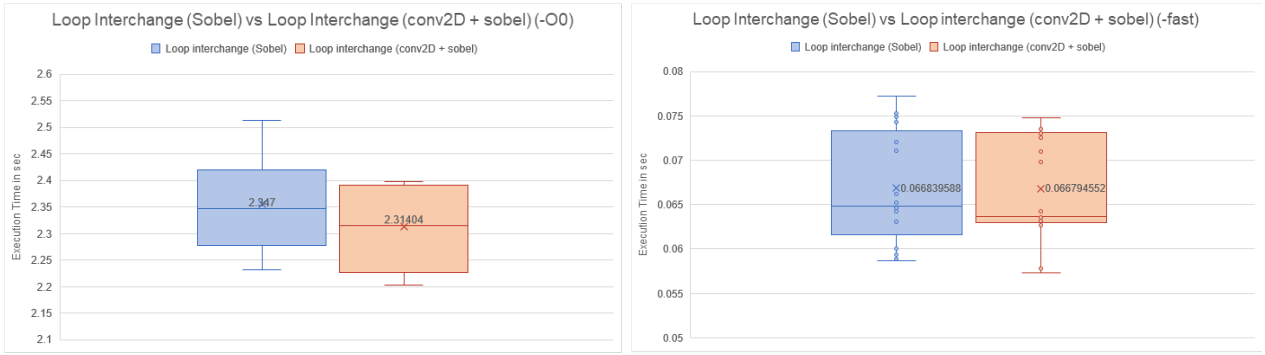


Figure 6: Execution Time for Loop Interchange (Sobel) vs Loop Interchange (conv2D + sobel)

## 5.2 Loop Unrolling *(convolution_2D)*

We noticed that the convolution 2D function had very few loop iterations, which meant that the loop control logic added a significant amount of overhead to the execution time. As a result, we made the decision to fully unroll the loop iterations, which resulted in an additional speedup of x1.28 (-O0) and x1.03 (-fast).

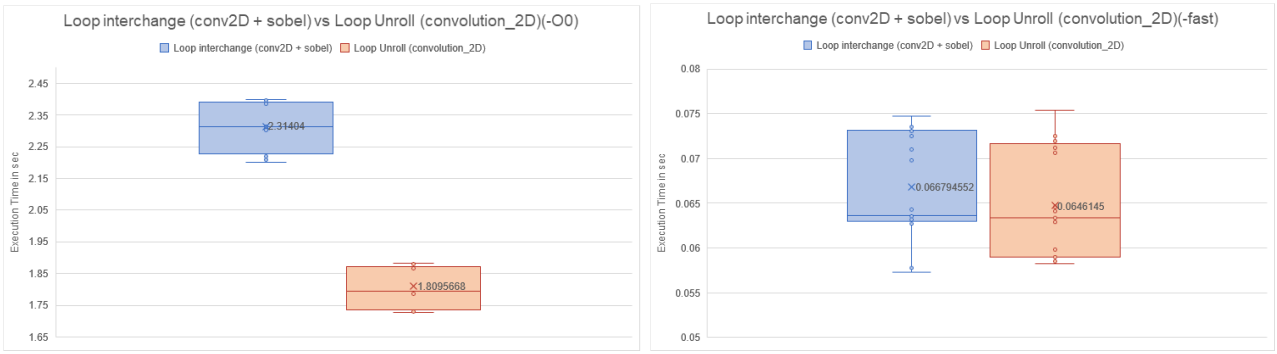


Figure 7: Execution Time for Loop Interchange (conv2D + Sobel) vs Loop Unrolling (conv2D)

## 5.3 Loop Fusion

The two loops in the sobel function—the one that traverses the output and golden picture to determine the PSNR and the one that calculates the output value for each pixel—could be merged. Fusing them should, at least in theory, yield better results due to the absence of loop control computations.However, this modification results in a lot of cache misses, since we access the same indexes of the input, output and golden matrices in the same loop, whereas before only the input and output matrices were accessed in the first loop and the output and golden matrices in the second, maintaining a better locality of reference. As seen in fig.8, loop fusion is slower than the previous optimization for both -O0 and -fast flag, so we omitted it for the rest of our optimizations.

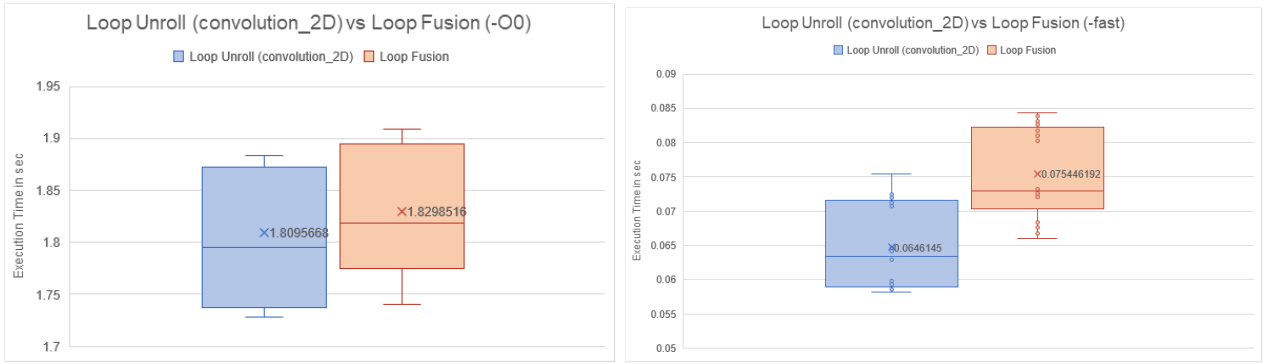


Figure 8: Execution Time for Loop Unrolling (convolution_2D) vs Loop Fusion

## 5.4 Function Inlining

To avoid the overhead of calling the *convolution_2d* function, we split it into two functions, *convolution_hor* and *convolution_ver* and used the keyword *inline*, so that the compiler automatically substitutes the function call.

Now that we have split the convolution function into two to account for the different sobel operators, we can omit the multiplications with the elements of the sobel operators that are

zero, replace the multiplications with -1 and 1 with subtracting and adding the pixel value in the final result and replace the multiplications with -2 and 2 with shifting left the pixel value and subtracting or adding it to convolution result.

Function inlining provided significant boost to our performance yielding a speedup of x1.2 for -O0 and and x1.4 for -fast.

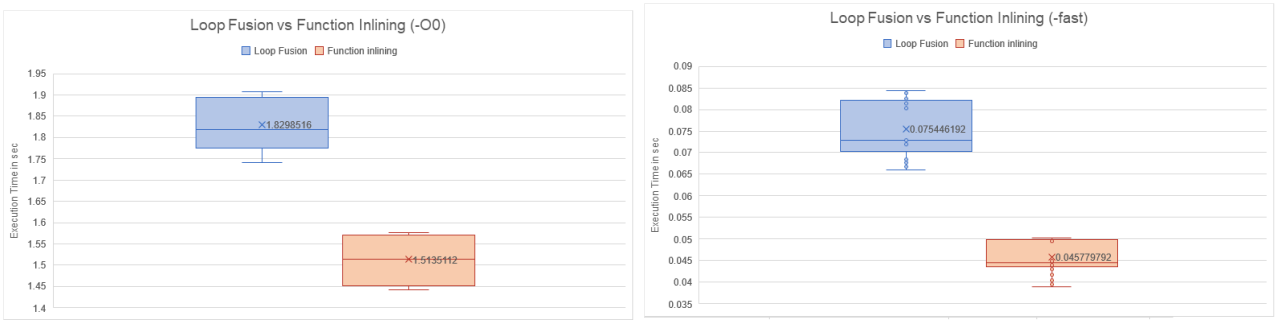


Figure 9: Execution Time for Loop Fusion vs Function Inlining

## 5.5 Strength Reduction

As our next optimization, we replaced heavy operations with lighter alternatives. More concretely:

1. We substituted every multiplication by *SIZE* (which is equal to 4096) with a left shift by 12 positions. Note that a change in the value of *SIZE* requires a change in source code and is possible if *SIZE* is a power of 2.

2. We moved the square root function call that was before the if clause in the *sobel* function loop inside the else part of the if clause to avoid performing this heavy operation unnecessarily.

3. We performed a multiplication of each item on our own rather than invoking the pow function for each number we wanted to square.

4. We also replaced the division in the calculation of the PSNR at the end, using log properties as:
$$PSNR = 10 \cdot \log\left(\frac{65536}{PSNR}\right) = 10 \cdot (\log(65536) - \log PSNR)$$
where log(65536) is a constant and can be hard coded. This operation takes place only once in the end, so no significant speedup is expected.

Strength Reduction proved our most significant optimization, with an additional speedup of x2.8 for -O0 and x1.6 for -fast.

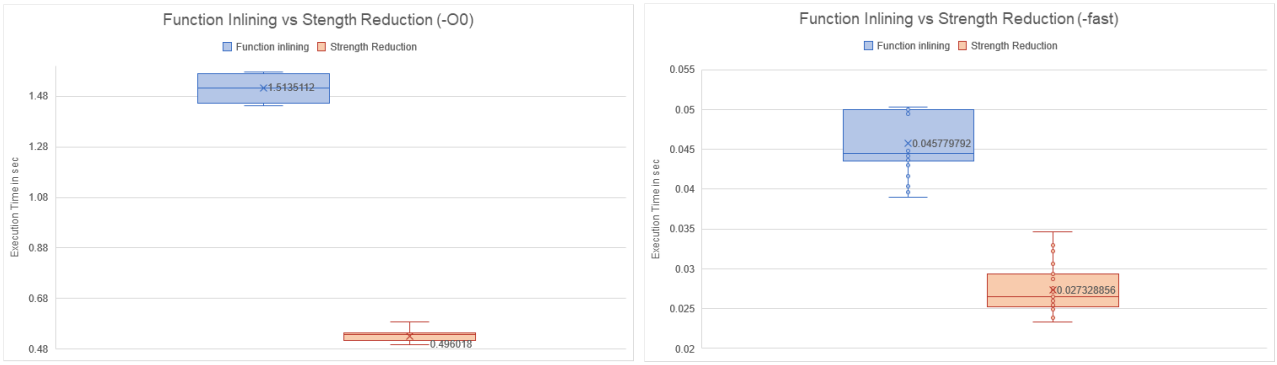


Figure 10: Execution Time for Function Inlining vs Strength Reduction

## 5.6 Loop Invariant Code Motion

To increase performance, loop-invariant code sections were moved outside of loops. We specifically achieved a minor speedup of x1.003 for -O0 and x1.09 for -fast by moving the multiplication $i \cdot SIZE$ outside of the inner for loops and doing it only once because it only depends on the row index.

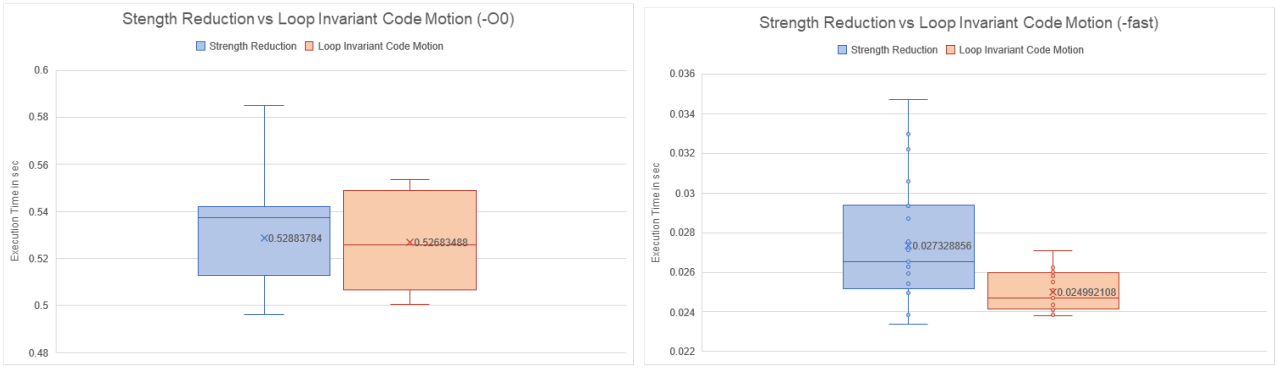


Figure 11: Execution Time for Strength Reduction vs Loop Invariant Code motion

## 5.7 Subexpression Elimination

In order to calculate the row and column indexes, respectively, just once for each unique row or column, we defined additional local variables in the *convolution_hor* and *convolution_ver* functions. This did not turn out to be much of an optimization, however, as it caused our code to run more slowly (fig. 12)

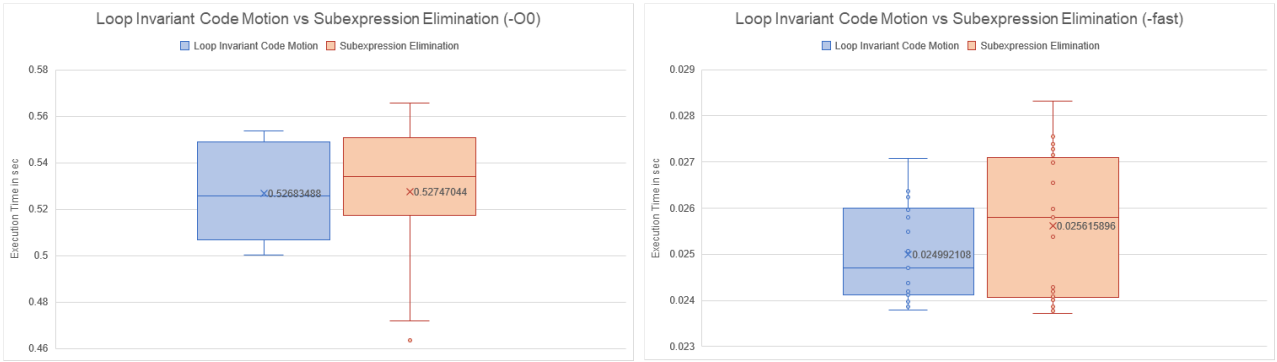


Figure 12: Execution Time for Loop Invariant Code motion vs Subexpression Elimination

## 5.8 Single Loop

We noticed that the nested loop that iterates over the image in the *sobel* function could essentially be replaced by one loop that just moves one pixel at a time, thus eliminating the need for computing the index of the pixel currently computed as $row\_index * SIZE + col\_index$, which can potentially reduce the amount of computations needed (both for the index calculation and loop control logic). Due to the fact that we now traverse the array in a single loop in a continuous manner, we perform some extra work, since we also compute values for the padding of the image, as shown in fig.14.

Avoiding this extra work demands that some extra checks are made, for example incrementing in each iteration a counter that wraps back to zero, wherever we reach the last column in order to skip 2 loop iterations. Oddly enough, this practise turned out to be less effective than just letting the extra work to be done, since it injected if statements. To maintain the same result as the golden output, we moved the loop that zeroed the first and the last columns at the end of the *sobel* function.

Despite the extra work at the edges of the image, our -O0 implementation achieved a speedup of x1.3. On the other hand, the -fast implementation performed worse, so we will not include this optimization in the next -fast optimization step.

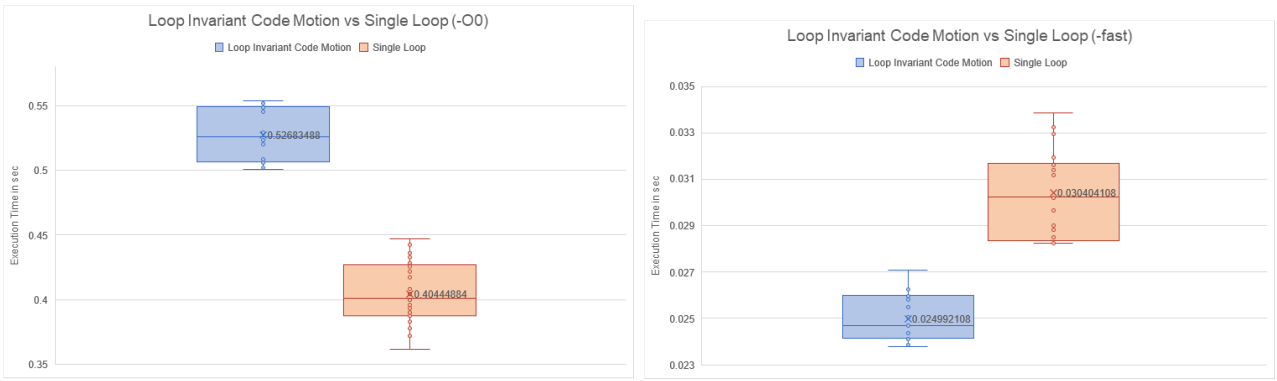


Figure 13: Execution Time for Loop Invariant Code Motion vs Single Loop

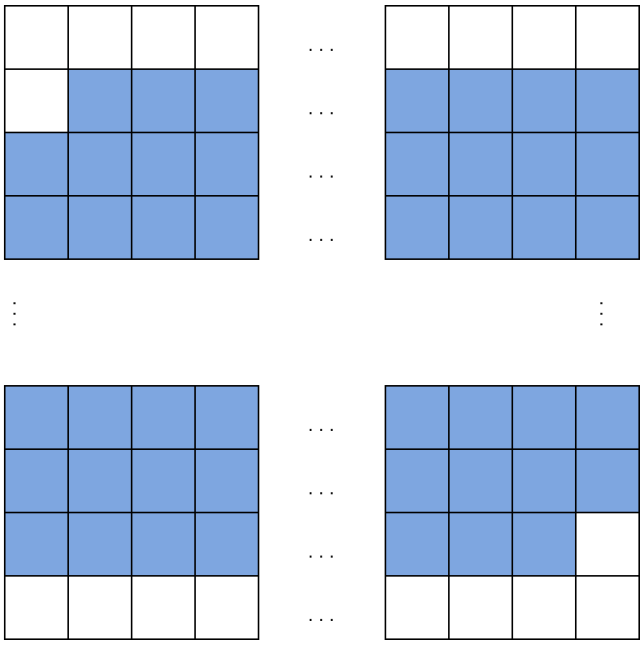

Figure 14: The blue pixels represent the pixels of the image for which we perform computations. Note that the pixels in the first and last row and in the first and last column are the padding and should be equal to zero.

## 5.9 Loop Unrolling *(sobel)*

In a previous optimization step, we fully unrolled the iterations of the for loop in the *convolution_2D* function. In a similar function, we attempted to partially unroll the loop in the *sobel* function, in order to cut down on loop control logic. We experimented with loop unrolling factors $f$ of 2, 4 and 8 and if the number of iterations $N$ were not divisible by $f$, we limited $N$ to the maximum multiple of our $f$ that is smaller than $f$ and performed the rest iterations outside the loop. We observed a slight increase in execution time for $f = \{2, 4\}$ (for -O0), but a slight decrease for $f = 8$, so we decided to keep the latter, even though it practically makes no discernible difference. In the case of -fast, loop unrolling performed even worse than single loop, so, again, we will not be including loop unrolling in the final version of -fast.


Figure 15: Execution Time for Single Loop vs Loop Unroll

## 5.10 Compiler Help

For this configuration, we aimed to aid the compiler by providing hints such as:

1. declaring some frequently used variables, such the loop counters and the results of the inlined convolution functions, as registers

2. declaring the *input, output* and *golden* pointers as restrict, since these memory locations are accessed only through the above pointers

For both -O0 and -fast flags, we observed a speedup of x1.03 and x1.83 from the previous successful optimization respectively.


Figure 16: Execution Time for Compiler help vs the previous successful optimization for -O0 and -fast

## 5.11 Square Root Look Up Table

A Vtune analysis at this point revealed that the *sqrt* function is now taking up the majority of the execution time. This function accepts a double as an input and outputs a double that is typecast to an integer. However, the actual parameter we pass to it is an integer, since it is the result of the convolution function and would like as output an integer, which is the floor of the actual square root (after typecasting). At first, we attempted to write an integer square root function, but our code became really slow. Our second thought was to create a look up table for the square root. Given that we must store the square roots of each of the 65536 distinct numbers (0–65535) that could be used as input, one might assume that this database would be enormous. Fortunately, since 255 is the greatest square root value we need to keep, just one byte is required for each entry, requiring a total of 64 KB of memory. Although that is still a sizeable amount of memory that does not even fit in the L1 cache, the fact that adjacent pixels have comparable input intensity values and, consequently, produce comparable results from the convolution function suggests that the computations for adjacent pixels will use nearby positions in the look up table, which means that cache misses are not as harmful as one might expect.

The -O0 version of our code ran about x1.28 times faster, whereas the -fast code ran x1.15 times slower than the compiler help version.

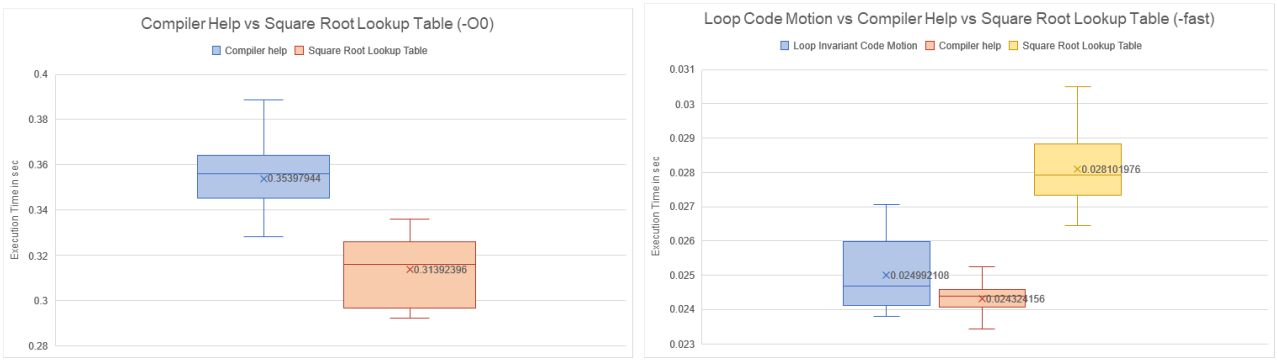


Figure 17: Execution Time for Square Root Look Up table vs the previous successful optimization for -O0 and -fast

# 6 Conclusions

In this Lab, we were assigned to perform optimizations on a sobel filter implementation. After applying the standard code optimization enhancements to the initial code, we managed to obtain a **123X speedup** compared to the provided implementation.

Through our experiments, we realized that whether an optimization technic works depends on various factors. For example, we realized that compiler optimization flags (*-o0* or *-fast*) sway the results. As we can see from figure 18, loop unrolling (factor 2) reduced the mean execution time when compiling with the -O0 compiler flag, whereas the mean execution time increased when compiling with the -fast flag. Furthermore, the sequence in which we apply optimizations also matters; applying loop interchange in the convolution_2D function after the sobel function positively impacted performance, whereas applying it before resulted in an increase in execution time.

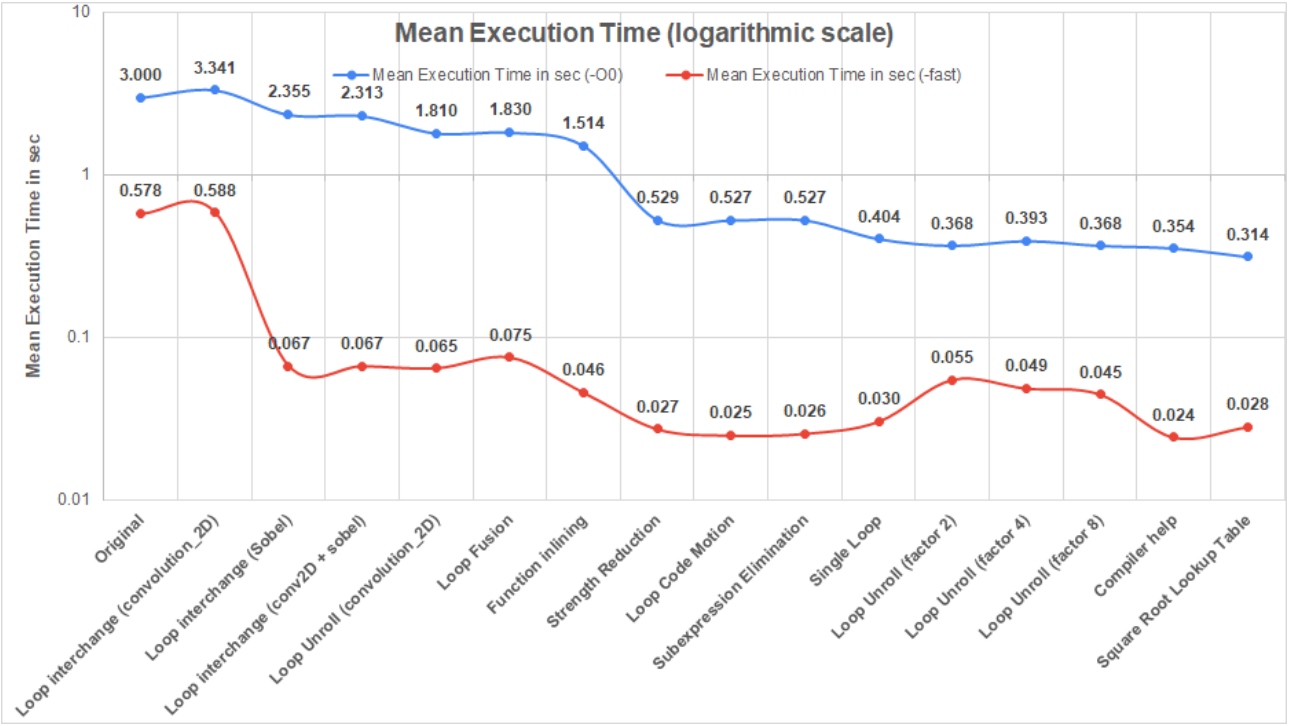


Figure 18: Mean Execution Time (logarithmic scale)

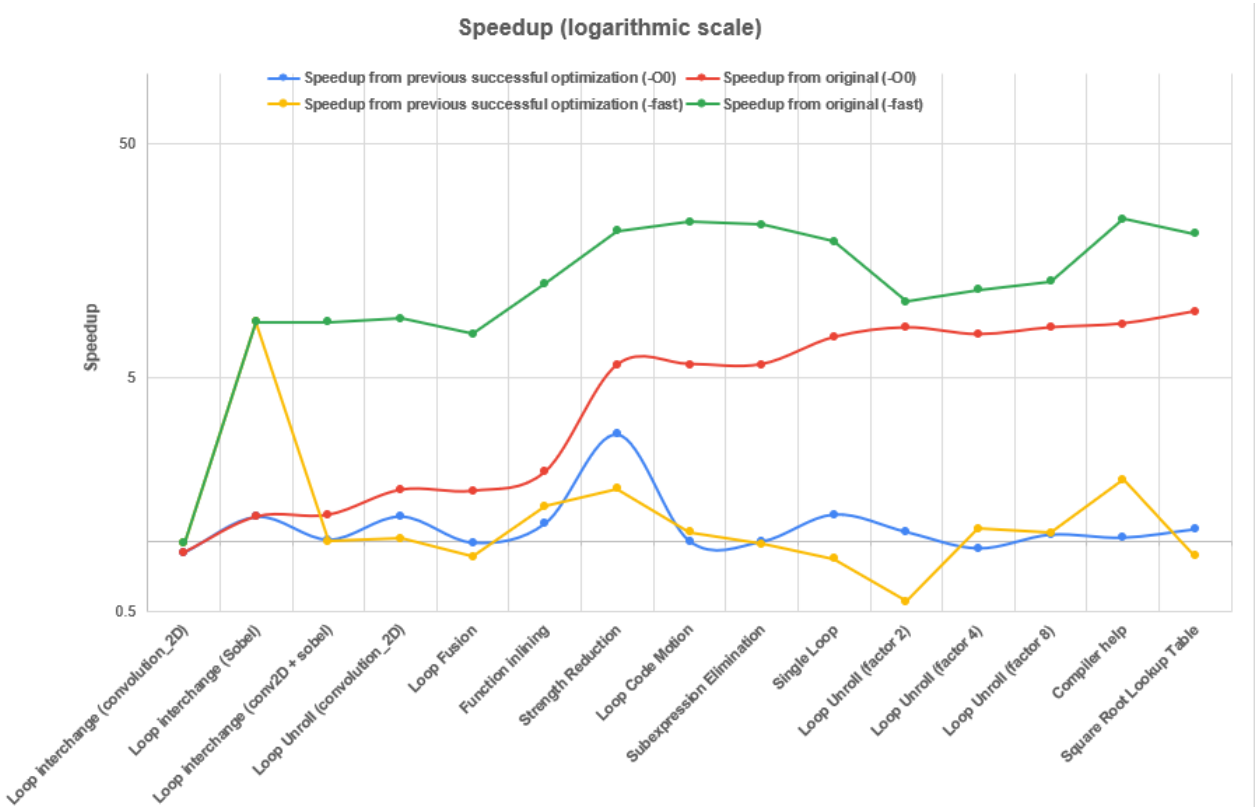


Figure 19: Speedup (logarithmic scale)