DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

ECE415 - HIGH PERFOMANCE COMPUTING SYSTEMS

Lab 1 - Optimizations in a sequential program

Students:
Karageorgos Nikolaos 02528, nkarageorgos@uth.gr
Lamprinos Isidoros 02551, ilamprinos@uth.gr

October 2021

Contents

1 Introduction

This assignment's subject matter was to apply optimizations in a sequential program that implements the Sobel operator in order to improve its performance. The Sobel operator is used in image analysis in order to detect edges of an image. Specifically, we had to apply step by step some common-used code optimizations so as to drop execution time and maintain the code efficient to produce the same results as the initial one. In every new optimized code version, the optimizations that worked were maintained and we compared each version with one another by taking the mean time and standard deviation of multiple executions' runtimes. In practice, we noticed a drop in execution time for -O0: from 3.27s to 0.27s and for -fast: from 0.75s to 0.031s.
The code implementations were tested both with the flag -O0 which signifies the compiler not to make any optimizations of its own and with the flag -fast which signifies the compiler to make any optimization considered that will improve time performance. For the purpose of this assignment we decided to keep a new optimization according to the execution times we extracted by running the executables that were compiled with the flag -O0.

2 System Specifications

System specifications of the computer we used for this assignment:

- Model : Dell Inspiron 15-3567
- CPU : Intel® Core™ i7-7500U CPU @ 2.70GHz × 4
- CPU clock speed : 2.70GHz
- Operating System : Ubuntu 20.04.2 LTS (64-bit)
- RAM : 4.4 GB
- L1 Cache : 128 KB
- L2 Cache : 512 KB
- L3 Cache : 4 MB
- Compiler : icc (ICC) 2021.4.0 20210910
- Kernel : 5.11.0-38-generic

3 Optimizations

We decided to apply the changes according to the order they were presented in the lab instructions. This decision was proven efficient since all the changes, with the exception of one (the red one in the block flow diagram) , managed to drop execution time consecutively, as it will be analysed in the following sections.

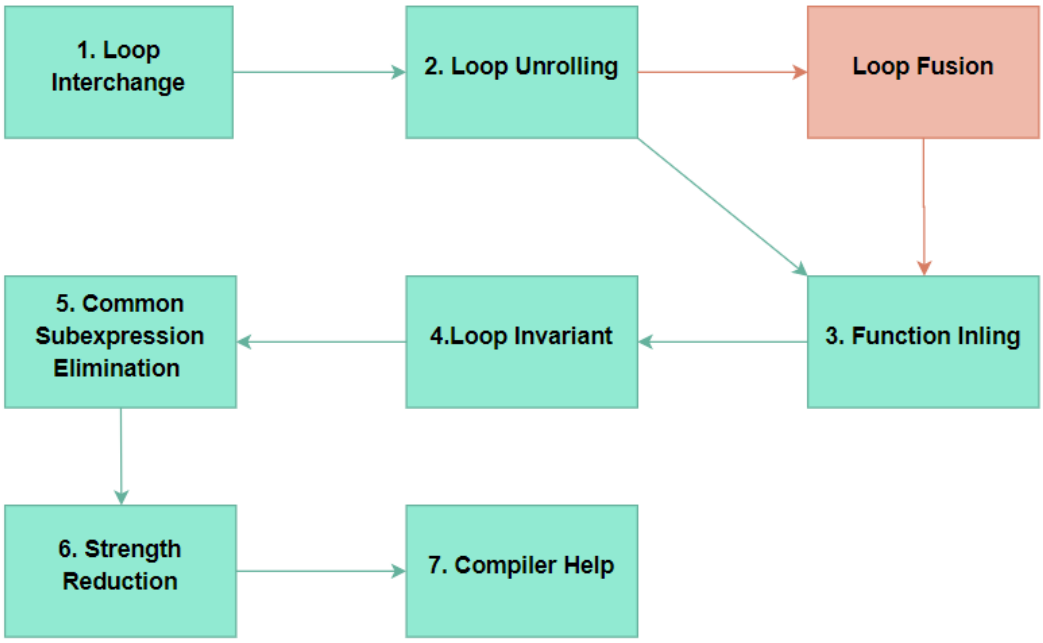

Figure 1: Order of the optimizations that were applied

3.1 Loop Interchange

- For the nested loops in the main Sobel function and in the convolution2d function we decided to change the order that the matrices were accessed. The initial code accessed the matrices: input, grey and golden by columns, though these matrices are declared as one-dimensional which means that are stored in memory consecutively by rows. By changing the way of accessing the matrices we manage to use a space of them that has come to memory, taking advantage of the spatial locality. In this way the number of misses is reduced which leads to execution time being dropped.

In practice, we noticed a drop in execution time for -O0: from 3.27s to 2.25s and for -fast: from 0.75s to 0.05s. By the significant time reduction in -fast compilation we understand that this change was difficult for the compiler to figure out without our intervention.

3.2 Loop Unrolling

- Initially, we tried to unroll the inside 'for' loop in the convolution2d function and after that we also unrolled the outside 'for' loop. The most optimized result came after unrolling both loops as the number of iterations was very small for both of them (3 iterations, 3x3 matrix operations)
- We also tried to unroll the double 'for' loop of the PSNR calculation by step 4, 8 and 16. We noticed a small drop in mean execution time for the unrollment by step 4. However, we decided not to keep this change as the difference was insignificant and this code version would not be convenient for trying the Loop fusion where we fuse the 2 'for' loops in the Sobel function.

By unrolling 'for' loops we avoid some unnecessary branch instructions. Practically, this optimization improved performance, making time for -O0: 1,74s and for -fast: 0,0045s

3.3 Function Inlining

- We placed the convolution2d function inside the Sobel function. This optimization worked as the convolution2d function is called multiple times inside the Sobel function.
- Also, we tried moving the Sobel function in the main function. However, this did not improve performance as the Sobel function is called once, so it was decided not to keep this change.

With function inlining we save time not calling the function and avoid a possible overhead. This optimization slightly reduced execution time for -O0 to 1.72s and for fast to 0.0044s.

3.4 Loop Invariant

- It was noticed that the calculations of (i-1)*SIZE, i*SIZE and (i+1)*SIZE were made repeatedly inside the 'for' loop of j operator, but this did not seem reasonable since they were not dependent on j iterator. So we moved them outside the j iterator 'for' loop and replaced them with the variables tmp1, tmp2, tmp3. This change applied not only in the double 'for' loop where the Euclidean distance is calculated but also in the one where the PSNR is found.

In this way we avoided many unnecessary calculations of multiplications that we had already calculated. This change improved by little the performance, making time 1.68s for -O0 and 0.0043s for -fast.

3.5 Common Subexpression Elimination

- Firstly we used tmp2 for the calculation of tmp1 and tmp3. In this way we avoid many multiplications.
- We also observed that the j+1, j-1 expressions are used a lot in the inside for loop where the Euclidean distance is calculated, so we moved the in the start of this for loop and replaced them with the variables tmp4, tmp5.
- Last, we replaced SIZE-1 with the variable forsize, since this calculation was used in the conditions of every for loop in the code.

This optimization dropped slightly execution time to 1.669s for -O0 and 0.0043s for -fast.

3.6 Strength Reduction

- Firstly, we replaced the function 'pow' with a multiplication. This was proven efficient as the call of a library function is costly.
- In the convolution calculation we replaced the horiz_operator and vert_operator with numbers. This removed some unnecessary multiplications with a zero element.

- We replaced all multiplications of 2 with shift operations which is way less costly.
- The call of costly function 'sqrt' was replaced by finding the value of a square root in a lookup table which was calculated once in the start of the Sobel function.

These changes proved strength reduction as a very efficient optimization as it managed to improve performance impressively, giving us 0,3s for -O0 and 0.0031s for -fast.

3.7 Compiler Help

- We declared the variables t, PSNR, p1, p2 as register ints and i, j, p, tmp1, tmp2, tmp3, tmp4, tmp5 as register unsigned ints. All these variables are accessed multiple times in the program.

This optimization was not the most beneficial as it only managed to drop execution time to 0.29s for -O0 and for -fast the time almost remained the same: 0.0031s but we decided to keep the change anyway.

3.8 Loop Fusion (no optimization)

- We tried to fuse into one the two nested 'for' loops as they were of the same size and after the Loop Interchange Optimization were exactly the same in size and in order.

Loop Fusion was a technique we tried using in the order the assignment suggested, but we noticed that the runtime increased instead of the expected decrease. That's why it was not used in that order. We also tried using it after each of the optimizations that followed, but still the time did not improve, it only got worse. So we ended up not using this technique at all. The part of the code we tried using it was in the Sobel function.

4 Performance diagrams and Conclusions

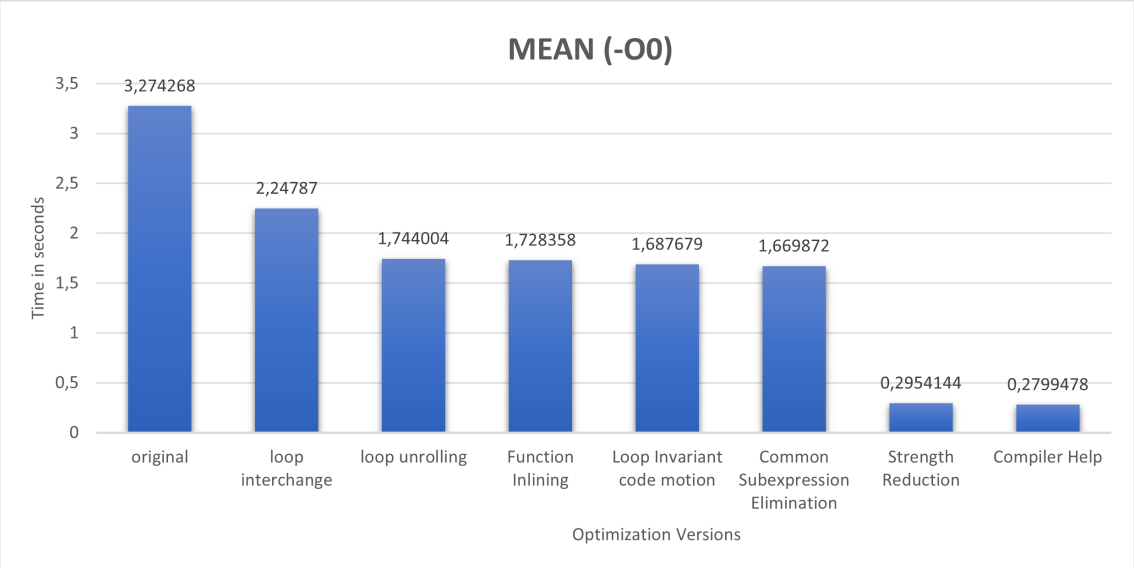

Figure 2: Mean time for -O0

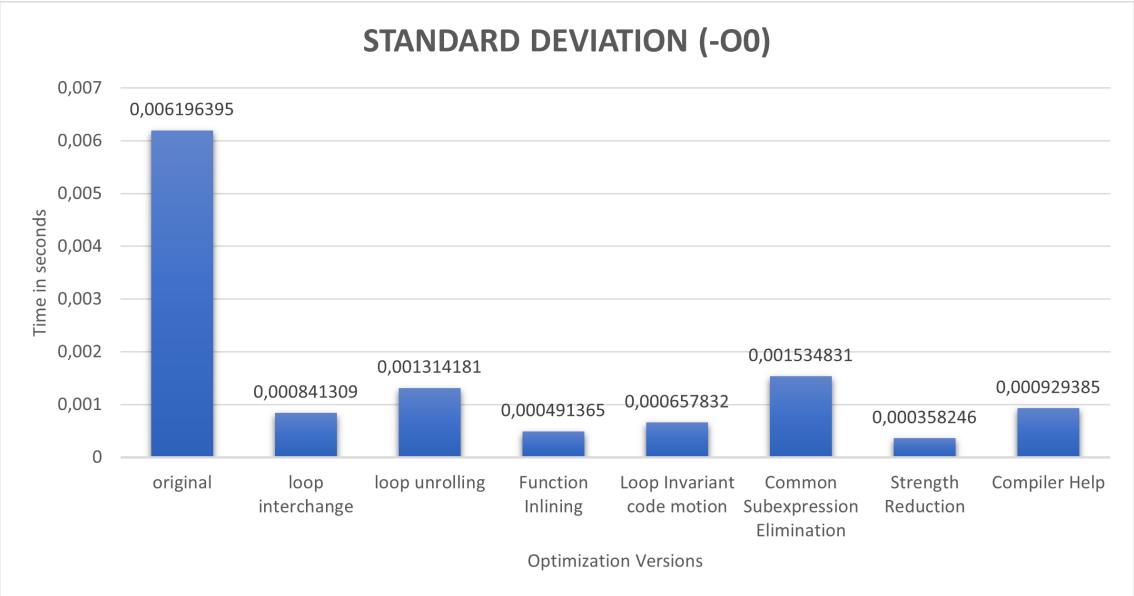

Figure 3: Standard Deviation for -O0

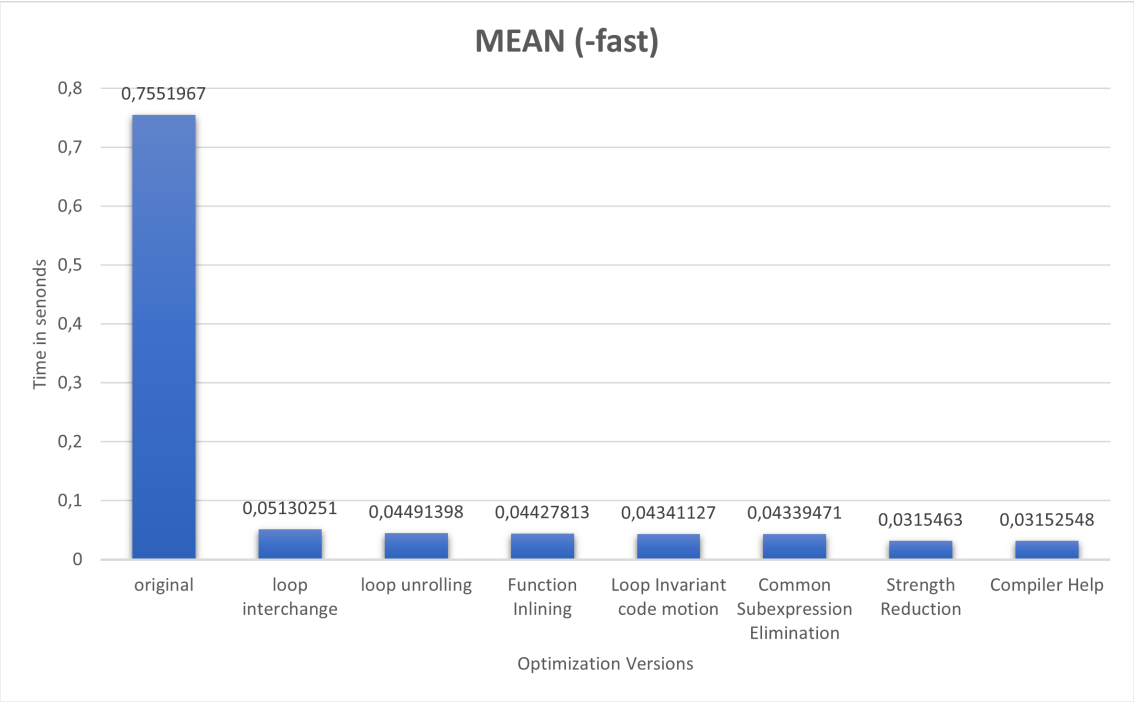

Figure 4: Mean time for -fast

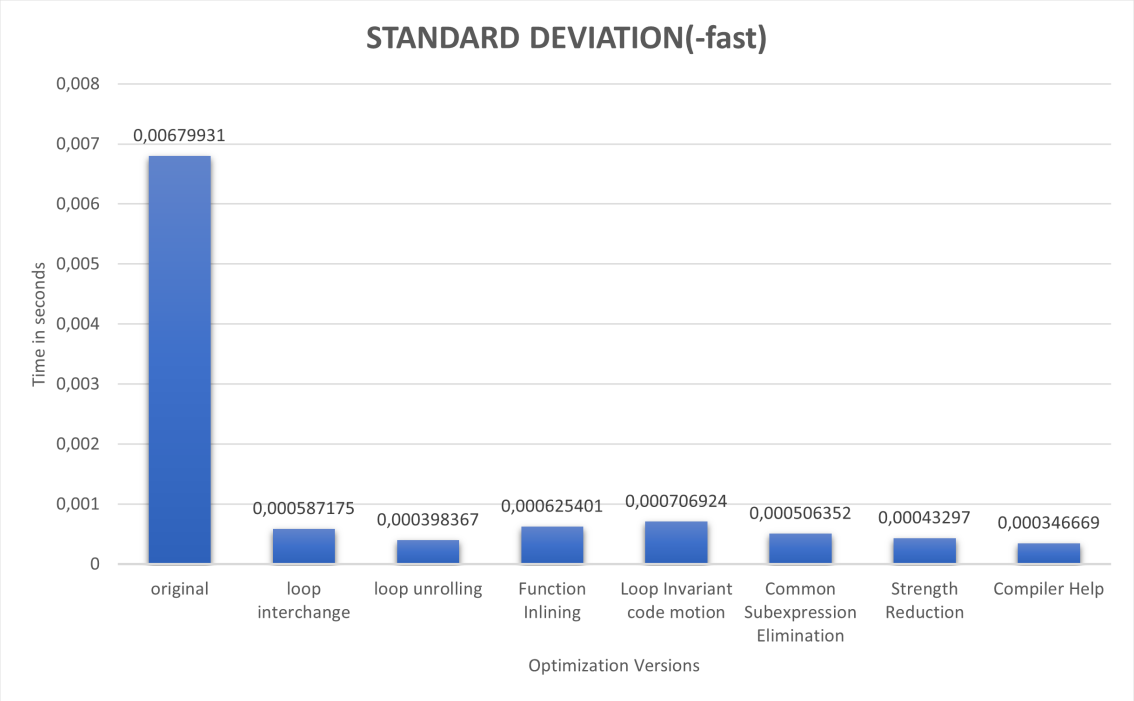

Figure 5: Standard Deviation for -fast

As we observe in the graphs above, the different optimization techniques drop the run time of the Sobel filter to 8.5%(with -O0 flag) and to 4% (with flag -fast) of the original runtime when used simultaneously. Of course different techniques have different impact. We notice that with flag(-O0) the Strength Reduction had the biggest impact, dropping the runtime to 17% of the previous optimization time. Also the Loop Unrolling affected the time in a significant manner dropping it to 68%. On the other hand with the -fast flag the biggest difference was made by the Loop Unrolling technique, probably because the compiler cannot take the initiative to change this part of the code as it does not know the algorithm we use and what exactly we want to do there. After changing that, the runtime drops to 6.8% of the original and it only goes down another $\sim$ 2.8%.. The rest of optimizations were not as significant as the ones mentioned above since they slightly improved perfomance.