ECE415 High Performance Computing

Lab Assignment 1

Optimization of a greyscale Sobel Filter

Kyritsis Spyridon 2697

Karaiskos Charalampos 2765

(a) Input RGB image I of the Sobel Filter (b) Application of the Sobel Filter onto I

Project Description

The goal of this project is to optimize the C code of a Sobel filter. The Sobel operator, sometimes called the Sobel–Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasising edges.

For the purposes of this lab assignment the Sobel filter is applied to a grayscale picture with a fixed size of 4096 by 4096 pixels. The pixels of the image in question are represented by 1-byte characters and thus take values from 0 to 255.

All experiments described above are run in the same machine a Dell laptop with the following specs:

- CPU: Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz, 2701 Mhz, 4 Core(s), 8 Logical Processor(s)
- Installed Physical Memory (RAM): 8.00 GB DDR3
- GPU: NVIDIA Quadro K1100
- Disk: Crucial MX500 500GB SSD
- Operating System: Ubuntu 20.04 LTS

As per the assignment's requirements all our basic tests were performed using the Intel compiler icc. In the Extras section we will talk about the results we obtained by using a different version of icc as well as another compiler.

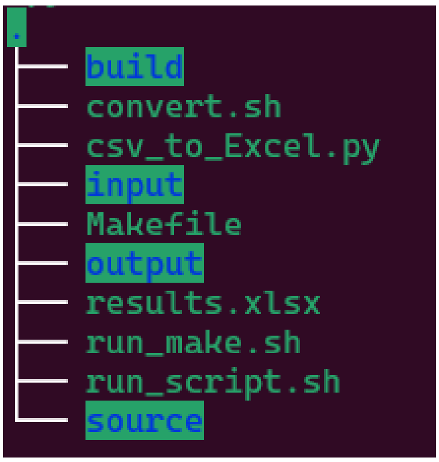
Deliverable Directory Structure

To make sure our deliverable is as clean as possible we organized our directory structure as follows:

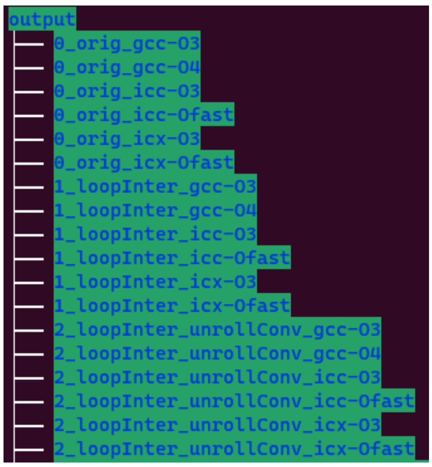
- input/
    - The input directory contains all necessary image files for the tests to run.
- source/
    - The source directory contains subdirectories each containing a C file with a different set of optimizations performed. Each subdirectory name begins with the number signifying the order in which the optimizations were performed. The rest of the

subdirectory name is a brief description of the optimization performed.

- build/
  - The build subdirectory contains all binary files produced during the make process. The executable files names are the name of the build directory their C source code originated from, with the name of the compiler and the optimization flag used appended in the end.
- output/
  - The output subdirectory contains a subdirectory for each executable file present in build/. Each subdirectory contains the .grey image available for reconversion to grayscale to verify the test results, as well as .runlog file. The .runlog file contains information about the tests execution time and PSNR value across multiple runs of the same test in a CSV format.
- The top-level directory contains a suite of scripts we will be discussing later as well as the makefile for the project.



Here can observe a snapshot of the directory structure



Experimental Methodology

To ensure our results reveal the actual performance of our application we performed each test a total of 22 times. We then proceeded to remove the best and worst result we got to make sure random events such as page faults and interrupts, due to operating system processes do not skew our measurements, leaving us with 20 runs of each experiment.

We also made sure that our system was as unoccupied as possible at the time the experiments were performed.

In order to streamline our experimental methodology, a suite of scripts was created in both Bash and Python.

The given Makefile was expanded to include extra functionality and functions as a master script. This means that we can call all other created scripts through the make <command> syntax.

The commands supported by the Makefile are:

- make clean
  - Removes all created executables and output files (images and logs)
- make all
  - Calls the run_make.sh Bash script. This script takes all source C files and creates executables for all available compilers and optimization levels. The script is configurable to compile the C files with an arbitrary number of compilers and optimization flags, as long as they have the appropriate flags set.
- make run
  - Calls the run_script.sh Bash script. This script runs 22 experiments for each executable in build/, created by the make all command. The script is configurable to run any number of experiments specified by the user. The script is configured to create all necessary directories under output/ and log files from the experiments. The output of the executables is redirected to a log file, where it is stored in a CSV format for easier data processing.
- make image
  - Calls the convert.sh Bash script. This script uses the convert tool, present in Linux installations, to convert the output .grey file to a .jpg image in order to visualize the result.
  - **Note:** To use the convert.sh script we needed to install the following in our Ubuntu installation:
    - `sudo apt-get install ImageMagick`
- make excel
  - Calls the csv_to_Excel.py Python script. This script takes as input the input, build and source directories as well as an optional parameter with number of test runs, if a value different than 22 was specified during the make run command. The script creates an Excel

spreadsheet with all the results sorted across compilers and optimization levels. The Python script reads all CSV log files finds and drops the best and worst rows, so as not to skew our results. Furthermore, the script writes the Excel formulas to calculate the average execution time of each test and the standard deviation of the results.

- **Note:** For the Python script to work we needed to install the following on our Ubuntu machine
  - `apt-get install python3.9`
  - `pip install openpyxl`
  - `pip install pandas`
- **Note:** We encountered a problem with the openpyxl framework where the formula for the standard deviation did not work in the actual Excel spreadsheet. This happened due to the openpyxl framework internally using a deprecated version of the STDDEV.P formula which included an extra character. This had to be removed by hand in the Excel spreadsheets via find and replace.

Optimizations

To perform our optimizations, we used the given code for the Sobel filter as the control version, and optimized the code based on that. Below we can find a list with all performed optimizations. The optimizations were applied in the order they are presented and in an additive manner, meaning that each test builds on top of each predecessor.

- **test1**
  - Interchanged the main double loop iterators in the sobel() function and the convolution2d() functions, making the i loop contain the j loop in both, so that the arrays are parsed efficiently.
  - The results show a massive performance increase, saving on average around a whole second of runtime with -O0 enabled, and 0.7 seconds with -fast enabled.
- **test2:**
  - Unrolled the convolution2D loop, removing the loop entirely and replacing it with a single sum of all the values needed.

- Split the function into convolution2D_ver and convolution2D_hor to split which operator is used, because we can then skip the 0 values of the operator and not perform redundant operations.
- The results show a sizable performance increase, saving on average around 0.8 seconds with -O0 enabled and about 0.005 seconds with -fast enabled.

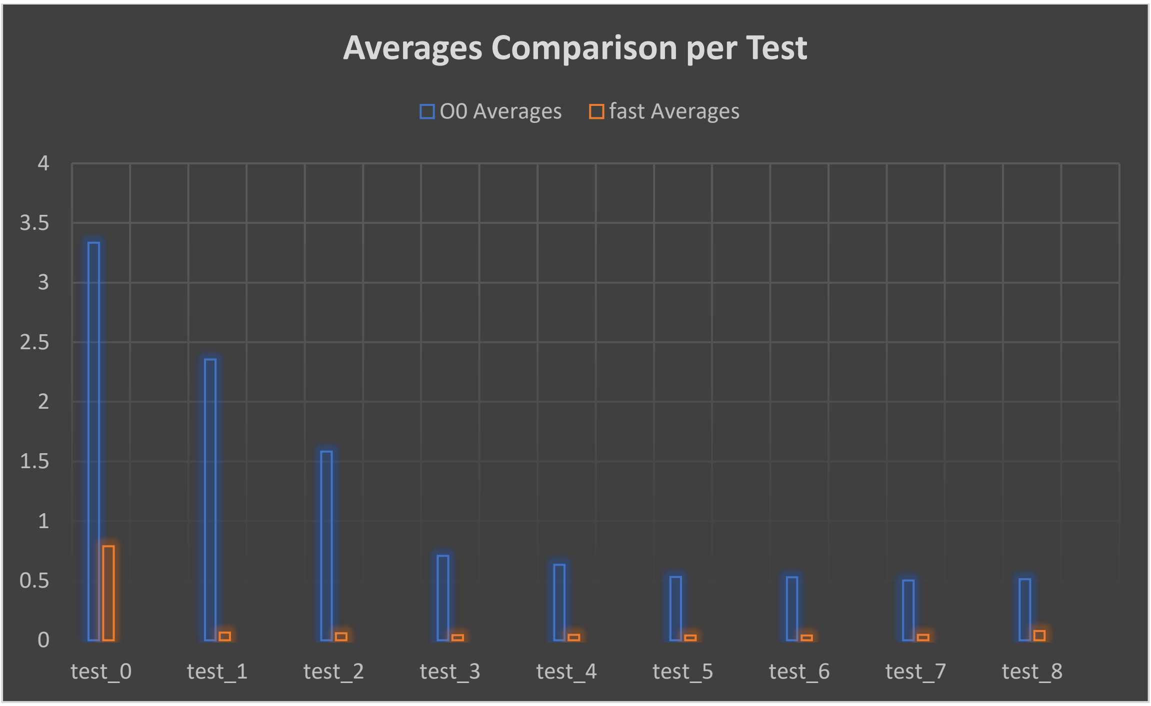
- **test3:**
  - Replaced all the pow() calls with the multiplications of the values with themselves to reduce the number of function calls during the algorithm.
  - The results show a sizable performance increase, saving on average almost 0.9 seconds with -O0 enabled and around 0.016 seconds with -fast enabled.
- **test4:**
  - We removed several operations that seemed to occur multiple times, performing either Loop Invariant code motion or Common Subexpression Elimination, whichever was appropriate for each case. Also replaced the multiplications with SIZE with the operation x << 12 since SIZE is 4096 == 2^12, thus replacing a costly operation with a much cheaper one.
  - The results show an increase in performance with -O0 enabled saving around 0.07 seconds but a slightly worse performance with -fast enabled, losing around 0.0016 seconds, which suggests that the internal compiler optimizations yield a better result.
- **test5:**
  - We inlined convolution2D_ver and convolution2D_hor to save time from the function calls. This was done by replacing the functions with preprocessor macros to also avoid the cost of calling functions in the main loops.
  - The results show a sizable performance increase, saving around 0.11 seconds with -O0 enabled and 0.005 seconds with -fast enabled.
- **test6:**
  - In test 6 we made heavy use of compiler directives in order to force the compiler to keep critical data close to the processor so that we avoid costly cache misses. However, this did not have the intended effect as observed by the execution times. The cause of this may be the fact that our data compete for the CPU registers leading to

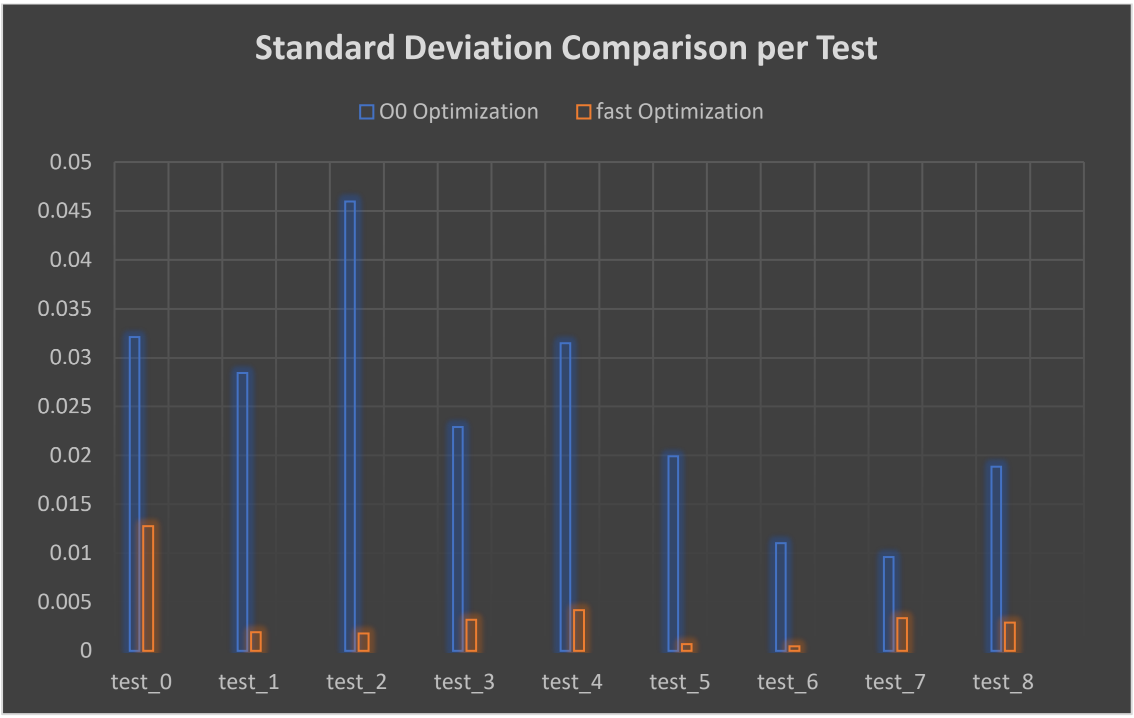
races and avoidable cache evictions for the register places, that have the exact opposite effect we wanted.
  - The results show a slight increase in performance, saving around 0.002 seconds on average with -O0 enabled and around 0.0005 seconds with -fast enabled.

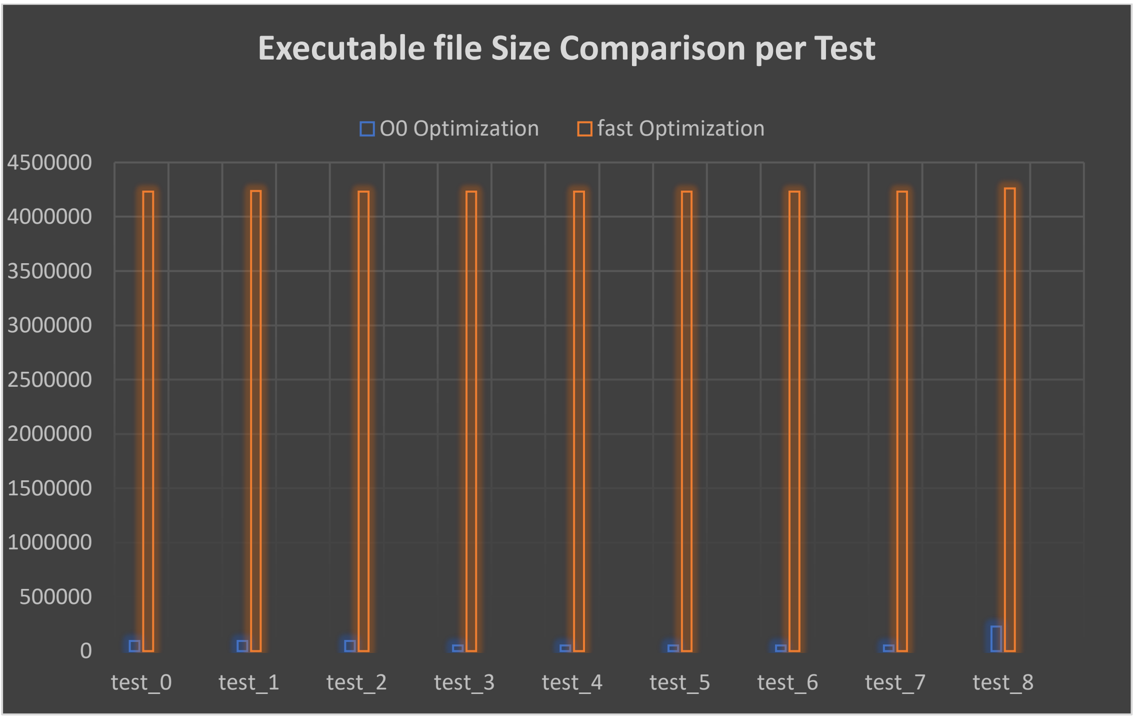
- **test7:**
  - For test 7 we initially tried to fuse the two loops in the sobel() function, but that proved to worsen performance.
  - We estimate that was because there were too many variables that had to be placed in the cache, resulting in more cache misses, so much that it outdid the gain from fusing the loops.
  - What we actually did for test 7 was replacing the if else in the first loop with a switch case, to save time from the failed ifs, and also placed the sqrt() call inside the block that will make use of it, thus saving time from both unnecessary function calls and unnecessary performance of the costly operations needed to find the square root.
  - The results show an increase in performance with -O0 enabled, saving on average about 0.02 seconds, but a loss of around 0.005 seconds with -fast enabled, once again suggesting that the internal compiler optimizations yield a better result.
- **test8:**
  - For test 8 we attempted to unroll the loops in the sobel() function, unrolling them into batches of 178 iterations per loop in the file present in the delivered files, but attempts of unroll of 8, 23 and 46 iterations have been attempted. In all of them a performance loss occurred, which we estimate comes from the instruction cache getting filled up and causing an increase of **instruction** cache misses with outdoes the gain from the unrolling of the loops.
  - The results show a significant performance loss with -O0 enabled, losing on average about 0.01 seconds, but a loss of around 0.034 seconds with -fast enabled, once again suggesting that the internal compiler optimizations yield a better result.

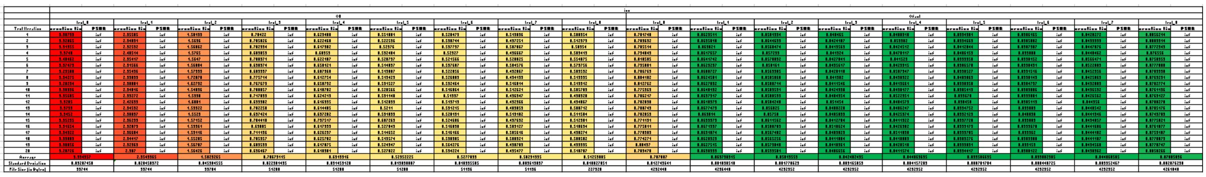
Graphical Result Representation

Below we can observe the graphs we obtained by plotting our results, as well as heatmap of our Excel spreadsheet that shows the gradual improvement in execution time across different optimization levels. Apart from the performance difference, which was expected, it is also noteworthy that in all -O0 experiments we observe considerably larger deviation (more than triple) than in the -fast experiments. Furthermore, we observe that all executables compiled with the -fast optimization flag are 1 to 2 orders of magnitude larger than the -O0 ones, indicating that the compiler manually unrolls the code in search of performance, sacrificing the size of the executable in the process.




Standard Deviation Comparison per Test
O0 Optimization
fast Optimization
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
test_0
test_1
test_2
test_3
test_4
test_5
test_6
test_7
test_8


Executable file Size Comparison per Test
O0 Optimization
fast Optimization
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
test_0
test_1
test_2
test_3
test_4
test_5
test_6
test_7
test_8



Extras

We also took advantage of our scripts to explore how different compilers react to different versions of the code.

For the Extras section, we experimented with:

- gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0,
- icc (ICC) 2021.7.0 20220726 and
- Intel(R) oneAPI DPC++/C++ Compiler 2022.2.0 (2022.2.0.20220730).

For each of the compilers above we produced executables with all available optimization flags {-O0, -O1, -O2, -O3, -O4 and -fast} (where applicable).

Due to the sheer volume of data, we did not have the time to properly examine all produced data present in the extras.xlsx file.

However, some notable observations could be made.

First and foremost, is the utter dominance of the Intel compiler versus gcc, although it should be noted that we are running on Intel hardware.

Secondly, icx seems to perform slightly worse than its older sibling icc which is something we did not really expect.

Lastly, we should look in the executable size. We should mention the remarkable consistency that gcc possesses in the size department. What is also notable is the improvement that icx shows versus icc.

Future Work

Due to timing constraints, we were not able to extract more statistical results from our experiments, especially in the Extras section were we could obtain some very interesting knowledge on how different compilers behave in different scenarios.

Another victim of limited time was the optimization process. We did not manage to implement any optimizations that would take advantage of the problems inherent and obvious parallelism, both at a CPU and a GPU level.

Closing, in the future we would like to revisit the optimization of the Sobel filter by using a more elaborate solution to avoid calculating the Euclidean distance in each iteration.

Literature

https://en.wikipedia.org/wiki/Supercomputer#/media/File:IBM_Blue_Gene_P_supercomputer.jpg

https://github.com/DanyEle/Sobel_Filter/blob/master/sobel_filter_picture.PNG