Axelou Olympia, 2161

oaxelou@uth.gr

HPC : ce421

14/10/2018

Homework 1: Optimization of a sequential program

This assignment is about optimizing a sequential program which uses the sobel operator to detect the edges of an image. In particular, we had to perform a series of common optimizations and in every stage we had to compare the result of multiple executions to the previous one (by calculating the average and the standard deviation of the data).

For the differences of the code and the management of the files I used Git. I have created multiple branches (each for every time an optimization went wrong and had to be excluded from the measurement). The final branch for -O0 is the one with the name "5th path" and the one for the -fast flag is "2nd path". The bundle files of the full repositories are repo_O0.bundle and repo_fast.bundle. To create the full repositories:

- $ git clone repo.bundle

I changed the makefile and added the targets "test" and "data_calculator". The first one executes the program 12 times and saves the outputs in a .txt file and then executes the data_calculator program to calculate average time and standard deviation (discards min and max). The second one is just a dependency for target "test". For the standard deviation I used the Population Standard Deviation method.

System Specifications of the computer I used:

- CPU: Intel Core i7-5500U 2.4GHz x 4
- Memory: 4GB
- Operating System: Ubuntu 16.04 LTS
- Kernel: 4.15.0-36-generic
- Compiler: ICC 19.0.0.117

-O0 flag Analysis

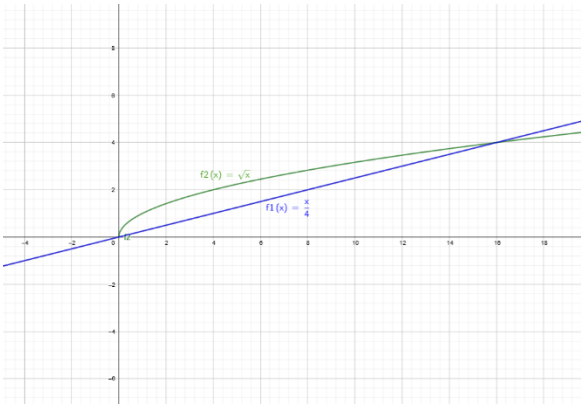
The steps of the optimization:

1) I began with the most basic optimization: Loop interchange. In the main *for* of the sobel function and in the *for* of the convolution2D function I changed the order of the iterators and I noticed a difference in the execution times: the original execution time was 6.982106sec and became 5.643562sec.
   It seems reasonable since inside the loops the iterator i is multiplied by the size of the row, so it accesses the table in a column-major order. However, in order to take advantage of the spatial locality the best way is the row-major order.
   I didn't alter the 3rd *for* which is for the PSNR (under the main *for* of the sobel function) because I believe that it's fine as it is.
2) Then I tried to replace the function call of the convolution2D with the block of code of the function inside the nested for of sobel(..). But since the code is "packed" inside a *for* loop there wasn't any opportunity to make a simpler code. So, that wasn't enough and the execution time did not improve so I threw away the changes.
   Instead of Function Inlining I focused on the convolution2D function: I moved the for-loop-iterator-invariant code out the loop and since we are editing pictures (2D arrays → nested for loops) the difference was noticeable, from 5.643562sec it dropped to 5.581059sec.
3) The next optimization was to unroll the *for* loop of the convolution2D function as each of them is only for 3 iterations, 9 in total. The results are satisfactory and in combination with the next step it will be very useful.
4) Function Inlining: Replacing the function with its body is faster not only because we don't waste any time for the function call (stack frame creation etc.) but also because it actually performs less operations since the two 3by3 arrays for the mask have all zeroes in one of the columns and in one of the rows respectively. Also, all the other elements are between -2 and 2 so the multiplications can be later transformed to bit shifts and additions.
5) Then I tried Loop Fusion between the two *for* loops in sobel(…). Although we don't have the overhead of the 2nd *for* anymore, it ends up taking part of the memory that would be later used by the next iteration of the 1st *for* so this part of the code becomes 0.1sec slower.
   At this stage I had almost exhausted all the optimizations of the assignment and I tried to think of others or to alter them, e.g. Function Inlining but changing the algorithm of the function, which leads us to the next step.
   The three pow(x, 2) are not mandatory. They can be replaced with a multiplication without the result being altered. As expected, the difference in the execution times is enormous. In particular, from 4.677227sec it dropped to 0.528007sec.
6) At this point I noticed that the code of the convolution2D function inlining had not the best performance available. Both the vertical and the horizontal convolution use the same elements of the input array. So, to take advantage of the spatial locality of the data I changed the order of the computation: The two convolutions are now computed together so that the index of the input array takes a value only once. I think this is best categorized as Common Expression Elimination. Difference in execution time: from 0.528007sec to 0.462162sec.

This is the end of the successful optimizations. The following are some of the attempts that failed:

1) sqrt: As we don't need the digits after the decimal point, I tried to implement a faster version of *res = (int) sqrt(p);* I thought of two ways (the first one has become a commit in Git) and the other one is:

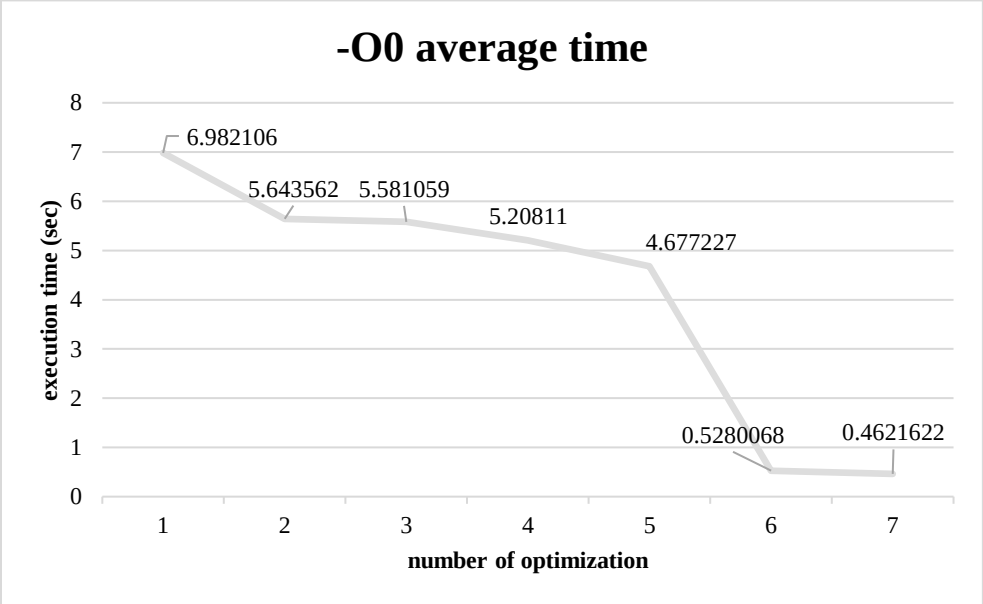
```
...
if(p < 16){
   res = 0;
   while(((res + 1) * (res + 1)) <= p)  res++;
}
else{
   res = p / 4;
   while((res * res) > p)  res--;
}
...
```



The goal of this is to provide the while loop with a better initial value and is based on the relation between f1(x) = x/4 & f2(x) = sqrt(x) : for x < 16: f1(x) < f2(x)
and for x > 16: f1(x) > f2(x)
However, the time results of both tries were bigger than the one of math.h's sqrt and especially the second one (approximately 17sec execution time).

2) I also used the register keyword for the iterators i, j but the result in execution time was not satisfactory.
3) I didn't use *const* or *restrict* keywords. *Const* could be used in convolution2D but it would have been useless with Function Inlining. The same applies for *restrict*.

The following line chart is only about the first 6 optimizations (the ones that actually reduced execution time).



-fast flag Analysis

The steps of the optimization:

1) I began with the Loop interchange, like with the -O0 flag. However, the execution time was reduced even more (proportionally to the scale of the time for -fast) compared to the one of the -O0. The initial time was 0.839272sec and it dropped to 0.063731sec.
2) The next optimization I performed was the "modified" Function Inlining for the pow functions. As it was expected, the time reduced noticeably: from 0.063731sec to 0.050362sec.
3) Loop Invariant Code Motion: In the *for* loops of Sobel convolution2D functions the index remains (partially) the same. So, we can move the computation outside the nested *for* for less operations. This change affected positively the result: time reduction by ~0.005sec which is important for our time scale.
4) As preparation for the function inlining of convolution2D, at this point I unrolled the *for* loop of the function. The execution time reduced by ~0.01sec.
5) Function Inlining and at the same time same small optimizations like omitting trivial operations (like multiplying with zero).
6) Then, I used Common Subexpression Elimination for the block of commands of the convolution2D function and I put in comments its implementation and its declaration. The execution time dropped to 0.038141sec.
7) Lastly, I used the keyword register for the iterators i, j and the execution time touched the 0.03784522sec.

Some of the attempts that failed:

1) PSNR variables (PSNR and t) to registers.
2) Strength Reduction (e.g. for the PSNR the division). In the Linux Systems -fast includes -no-prec-div flag which improves the speed of the floating-point division. So, I guess that trying to improve it with the usual ways didn't have a better result than the option provided by the compiler. Source : https://software.intel.com/en-us/node/522804 & https://software.intel.com/en-us/node/522989

The following line chart is only about the first 7 optimizations (the ones that actually reduced execution time).