# Assignment 3a: Loop unrolling and program optimization, the hard way

Martijn Koedam

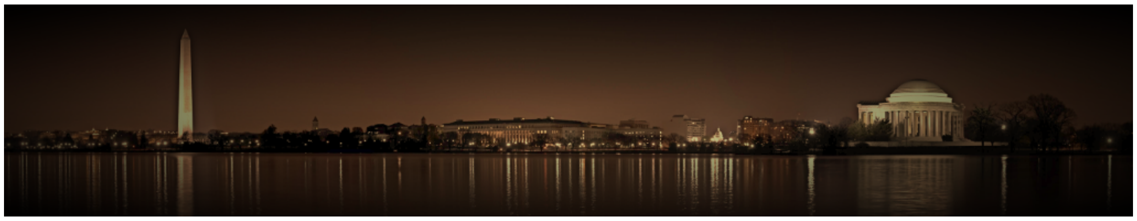
2016

## Organization

In the last two assignments we worked on the Atmel AVR microcontroller. In this assignment we are going to move up in performance and target the superscaler processor. Namely the Intel Core I7 and the ARM Cortex A15 and the application we target will, instead of blinking an LED, process a 6 megapixel image. In the first part of the assignment we are going to do optimizations by hand, we will turn off the compiler optimizations almost completely. ( Optimization level -O1). Later in the assignment we are going to see if we did better than the compiler could, if we can still help the compiler, and we take a look at auto-vectorization..

The application we are going to use is a type of application people use almost daily; they take a picture with a mobile phone, 'fix it' and then upload it to the web. So if you create a nice picture like this:

Then 'fix' (instagrammify) the image to make it look like this:



An application which does these enhancements normally exists of several filtering loops chained together. This application is not different. We start with a local color enhance algorithm [1], a blur, and a spotlight effect (darkening to the outside).

The goal of this assignment is to get a feeling about available optimizations, the impact they make, what the compiler can and cannot do, how to measure performance and what different performance indicators there are.

[1]`http://www.ipol.im/pub/art/2011/gl_lcc/`

## 1 Obtaining and running the application

**Exercise 1**

First we are going to clone the application:

```
$ git clone /home/pcp16/material/repos/assigment3.git
```

Inside the new directory you can type `make` to build the application. To run the application type the following:

```
$ ./filter in.png out.png
```

Run this application and look at the performance statistics on 3 different machines, `co6.ics.ele.tue.nl`, `co10.ics.ele.tue.nl` and `co17.ics.ele.tue.nl`. These machines have different generations of CPU's. (Pentium 4, i7 950, i7 6700).

**Note**

For the rest of this assignment run the code either on `co9.ics.ele.tue.nl` or on `co10.ics.ele.tue.nl`.

### 1.1 Measuring performance

As you saw in the previous exercise, the program itself already prints out some numbers for each step. Namely the time spend at that point. However time might not be the best measurement to see if the application is optimized, given between these 3 machines there is already a factor 4 difference.

Under linux there is a nice performance profiling application `perf`[2]. This gives more detailed information about the execution of the application, below is example output of `perf stat ./filter in.png out.png`[3]:

```
  25078.767142 task-clock (msec)    #   0.997 CPUs utilized
         2,916 context-switches     #   0.116 K/sec
            16 cpu-migrations       #   0.001 K/sec
        11,057 page-faults          #   0.441 K/sec
81,356,583,982 cycles               #   3.244 GHz
48,965,949,149 stalled-cycles-frontend # 60.19% frontend cycles idle
17,251,327,876 stalled-cycles-backend # 21.20% backend cycles idle
93,871,655,420 instructions         #   1.15  insns per cycle
                                    #   0.52  stalled cycles per insn
12,298,556,810 branches             # 490.397 M/sec
    73,014,313 branch-misses        #   0.59% of all branches

  25.150977590 seconds time elapsed
```

**Exercise 2**

Try to explain the different fields in the above statistics, see how they relate to each other and what field(s) is/are the most important to see if the program is optimized. Explain why the results can differ between multiple runs and propose a possible solution (take a look at `man perf stat`).

[2]https://perf.wiki.kernel.org/index.php/Main_Page

[3]Perf talks about the CPU frontend and backend.

`Perf` can display more interesting information about the execution of the application, to get a list of events it can monitor for check `perf list`. To get a more detailed breakdown of time spend in your application check `man perf-record` and `man perf-report`.

**Exercise 3**
Use `perf record` with the right flags to determine the part of the application that takes the most time.

## 2 Optimizations

First thing we do, is create a reference output, this is a simple (but not complete) test to see if we did not change the program when applying the optimizations.

We provide a small tool `imgdiff` that can compare two images, returns 0 if there are no differences and 1 otherwise and indicate the number differences and the maximal distance[4].

Why would we want to accept differences in outputs? The following exercise demonstrates how a small change, while correct, can result in a different value.

**Exercise 4**
Some optimizations, even though the code code has not changed might result in different output. For example:

```
float v1= -1.9513026f, v2= 0.31476471f, v3= 3.1415927f;
float b = v1-v2+v3;
```

versus

```
float v1= -1.9513026f, v2= 0.31476471f, v3= 3.1415927f;
float b = v1+v3-v2;
```

Can you reproduce this? and if so explain why the output is different.

To optimize the application we are going to apply some of the techniques we have seen during the lecture. It is important to keep an eye on the output of the `perf stat` tool, to see the performance and where the possible bottleneck is. This whole exercise is not as trivial as you might think, because you can reduce cycles, but cause more stalls in the fontend giving you the same execution time in the end.

**Exercise 5**
Start by applying some of the techniques seen during the lecture to main.c filter loops and localcorrection.c. Try to apply the following techniques at least once:

1. Loop interchange
2. Loop invariant code motion
3. Common subexpression elimination
4. If conversion
5. Loop unrolling
6. Loop fusion

[4] A common way for a program to indicate that it finished successful is using the return value of the main function. Returning 0 is the standard for indicating no error.

**Note**

In the quiz we are going to ask detailed questions about the effect of the above transformation techniques on the different performance measurements reported by perf. Try to investigate each of them.

We are not going to optimize the png generation code. This is just required code to get the image into the memory and write it out again in a readable format. You might have seen in the Makefile we always compiled this code with *-O3*.

## 3 Vectorization

The last manual optimization we are going to do is to rewrite one of the hot loops (code that gets executed the most) using SSE3 vector operations. Below is the loop from *localcorrection.c* that we are going to rewrite.

```
float sum = 0.0f;
for ( t = -radius; t <= radius; t++ ) {
    if ( nx + t >= w ) {
        sum = sum + Mask[2 * w - 2 - nx - t + ny * w] * Kernel[t + radius];
    }
    else{
        sum = sum + Mask[abs ( nx + t ) + ny * w] * Kernel[t + radius];
    }
}
Mask2[nx + ny * w] = (float) sum;
```

**Exercise 6**

Rewrite the above loop using intrinsics[a] or inline assembly using SSE2 and/or SSE3. Below are some hints to get you started:

1. Move the control (if/else) out of the loop.
2. Remove abs from the 'hot' part of the code.
3. unroll the loop.

After this you should have 2 loops remaining; one to handle the corner cases and one to handle the bulk of the work. The last loop is suitable for vectorization.

[a]Intel has a nice website with an overview of the available intrinsics: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

In the next assignment we are going to take a look how the compiler would have managed with this code, and how we can help the compiler along.