All elemental steps that will get you started for your new life as a computer science programmer.
Week F - Optimizations

Memory leaks
  - Finding leaks with valgrind

Performance optimization
  - Profiling with gprof
  - Profiling with callgrind

Week F – Optimizations
This week you will learn how to optimize your code. Even if your program is virtually error-free during compilation and runs smoothly when you test it, it may not perform well enough in certain cases. One reason might be that you implemented it inefficiently, so that the run-time is far longer than necessary. Another reason might be that there are so-called memory leaks, i.e. your program allocates memory without setting it free again.

You will see how you can find memory leaks under Linux with a tool called valgrind and how you can do profiling (analysis of your program’s performance) with gprof and with callgrind.

Valgrind is a collection of Linux tools and, unfortunately, you cannot use these tools for Windows. However, there are similar substitutes:

http://code.google.com/p/drmemory/
http://www.codersnotes.com/sleepy
Memory Leaks

Valgrind

- Virtual CPU
- Detects areas of memory that are lost
- Several other tools in valgrind suite
- Linux only

Simulation approach

- Quite accurate
- Very slow
- Architecture dependent

Finding memory leaks with valgrind

Finding memory leaks is quite comfortable with valgrind. It takes a program and runs it on a simulated processor. It stores, when memory is accessed and if it’s still in use. If something seems to be wrong with your memory accesses, valgrind might help you. There are also a lot of other tools that come with valgrind, e.g. callgrind.

The drawback of valgrind is that it can be very slow. Especially when you work with large datasets, you should try to run valgrind only if you have some kind of small example program. If your application usually needs minutes to reach the critical point, it may take hours (or days?) when you evaluate it with valgrind.
You should compile your program with -g -O0 flags

- g Produces debug information
- O0 means no optimizations

2. Run
valgrind --leak-check=yes ./assignment_06

3. Wait ...

4. Decipher leak summary

You should compile your program with -g -O0, but you do not have to. In fact, you can use valgrind with any software, even if you did not compile it yourself.

Once you start valgrind (see above), just wait until the program stops. All you have to do now is to decipher the leak summary.
Here you, that quite a large amount of data (almost 10 MB) is lost, further up you see that this is caused by a “cvCreateImage” in the called in the main.cpp in line 13. Someone probably forget to deallocate the memory with a cvReleaseImage().
Profiling with gprof

1. Compile with -g -pg flags
   - -g Produces debug information
   - -pg Generates extra code for analysis with gprof

2. Run program
   ./assignment_06

3. Program executes, gmon.out is created

4. Run
   gprof -p ./assignment_06 gmon.out

5. Interpret the “flat profile”

Profiling with gprof

Profiling with gprof is a little bit different. Instead of using a virtual machine, your program runs at normal speed, every few milliseconds, the current frame stack is analyzed and the statistics are saved to a file (gmon.out). There are several ways to inspect this file:
Profiling with gprof

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.88</td>
<td>2.11</td>
<td>2.11</td>
<td>100</td>
<td>21.13</td>
</tr>
<tr>
<td>36.62</td>
<td>4.16</td>
<td>2.04</td>
<td>100</td>
<td>20.43</td>
</tr>
<tr>
<td>25.67</td>
<td>5.59</td>
<td>1.43</td>
<td>100</td>
<td>14.32</td>
</tr>
</tbody>
</table>

cgColorsteps(_IplImage const*, _IplImage*, int)

cgColorsteps2(_IplImage const*, _IplImage*, int)

cgColorsteps3(_IplImage const*, _IplImage*, int)

[...]

Hint: Note that each function is called multiple times (here 100 times) to get statistically valid timings!

When you expect the basic output of gprof (the “flat profile”), you see where most of the time is spent. Please note that gprof suffers from the fact that it only samples the code execution with a certain frequency. You can circumvent this by calling your program dozens or hundreds of times.
Profiling with callgrind

profiling is integrated in the valgrind suite

1. Compile with -g -O0 flags
2. Run
   
   valgrind -tool=callgrind ./assignment_06

3. Program executes, callgrind.out.* is created
4. Run to generate textual overview
   
   callgrind_annotate callgrind.out.x
   
   (Alternative for graphical overview)
   
   kcachegrind callgrind.out.x

Profiling with callgrind

Another option is to use the valgrind suite for profiling. You run valgrind with the –tool=callgrind option to do this. A file callgrind.out.* is created. The * stands for the actual process ID.

Now you can run callgrind_annotate or kcachegrind to get some human-readable output. Callgrind_annotate produces a simple ASCII file, while kcachegrind gives you a graphical overview.
This is the output of kcachegrind, which displays graphically where your program spends its time. You see that cgColorsteps3() takes most of the time, but only because it saves an image to disk with cvSaveImage().
OpenCV Image Manipulation

cvCreateImage(size, depth, channels)
cvCloneImage
CvScalar cvGet2D(image*, y, x)
  ▪ A CvScalar ist just a container for 4 floats
  ▪ Access via scalar.val[i]
cvSet2D(image*, y, x, scalar)

There are more efficient ways to access image content ...
OpenCV Image Manipulation

IplImage* load_image = cvLoadImage("./test.png");
IplImage* base_image = cvCreateImage(cvSize(load_image->width, load_image->height), IPL_DEPTH_8U, 3);

for(int y=0; y < base_image->height; y++)
{
    for(int x=0; x < base_image->width; x++)
    {
        CvScalar base_px = cvGet2D(load_image, y, x);
        CvScalar ramp_px = cvScalar(128, x%255, y%255);
        CvScalar px;
        px.val[0] = base_px.val[0]*0.5+ramp_px.val[0]*0.5;
        px.val[1] = base_px.val[1]*0.5+ramp_px.val[1]*0.5;
        px.val[2] = base_px.val[2]*0.5+ramp_px.val[2]*0.5;
        cvSet2D(base_image, y, x, base_px);
    }
}

This is just a very simple example. It iterates over all rows and columns of the image and for each pixel, it accesses the pixel value of the load_image instance, performs an operation on the pixel and writes the result to the base_image instance.
Assignment F: Optimize your Code

This week we will show you how to avoid memory leaks and slow code.

Task 1: Speedup [70 points]

Have a look at the code in assignment_stub. I wrote a function cgShadowframe that does the following:

Create a crisp shadow of a given rectangle (specified by \((x_0,y_0)\) and \((x_1,y_1)\)). The shadow of the rectangle is displaced by the amount defined in \text{mean\_kernel}. Smooth that rectangle (by iterating over the neighborhood and taking the mean value, by “neighborhood” I mean all pixels whose \(x\) and \(y\) coordinate differ up to the value of \text{mean\_kernel}). According to the rectangle, darken the original image. Paint a white rectangle over the shadow. The result should be a white rectangle with a shadow border at the bottom and at the right-hand side.

Your task is to modify the function cgShadowframe so that it runs significantly faster. The important thing is that the result should stay the same. You can change the way it’s created completely. You may use valgrind –callgrind (Linux only) or gprof (all systems) for this task. If you like to explore your systems toolset, you may also try any profiler you can find on your system. When using valgrind, you have to compare the total amount of assembler construction fetches to the original version of the function. On my machine, it took about 15 000 000 000 fetches to execute cgShadowframe. If gprof is your tool of choice, you have to compare against the “total ms/call” value in the flat profile. My machine spent 610 ms on the original function. Please determine these values for your own system before you start optimizing. Please always turn off automatic compiler optimizations for this comparison, you can do this by providing the compiler flag \(-O0\).

Depending on the speedup you achieve, you’ll get more or less points

<table>
<thead>
<tr>
<th>Speedup Factor</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x</td>
<td>20</td>
</tr>
<tr>
<td>100x</td>
<td>40</td>
</tr>
<tr>
<td>1000x</td>
<td>50</td>
</tr>
<tr>
<td>4000x</td>
<td>60</td>
</tr>
<tr>
<td>5000x</td>
<td>70</td>
</tr>
</tbody>
</table>

On my machine, the code came down to about 2 500 000 instruction fetches and 0.082 ms per call.

Helpful Hints on performance optimization
In the sample code (01_profiling), I used a more efficient way of accessing image elements in an IplImage*. Instead of cvGet2D and cvSet2D you can access the pixel values directly. Avoid casting data types too often. Always think if accessing memory (in this case: pixels) is really necessary. Sometimes you do not need to iterate over the whole image for an operation. In some cases you can determine a value analytically instead of empirically (think about the blur operation).

You may use operations defined in OpenCV (and other libraries). Most of the time, these operations are implemented in a very efficient way (otherwise, these libraries wouldn’t probably be so popular).

**Task 2: Memory leaks [30 points]**

There are memory leaks in my implementation. Fix them!