CSCI 2400, Summer 2014
Lab Assignment L4: Code Optimization

1 Introduction

This assignment deals with optimizing memory intensive code. Image processing offers many examples of functions that can benefit from optimization. In this lab, you'll be improving the overall performance of an "image processing" application by a factor of about 25 – if you can increase the speed by a factor of ≈ 50, you'll get extra credit.

The application you'll be modifying reads in an "image" (a picture) and a "filter". An image is represented as a three-dimensional array, described in `cs1300bmp.h`. Each pixel is represented as a combination of (red, green, blue) values – when coded in the BMP picture format, the individual (R,G,B) values can taken on the values $0 \ldots 255$, but the `cs1300bmp.h` code is designed to handle larger pixel values. The code in `cs1300bmp.cpp` provides routines for reading and writing images in the BMP format. You are free to modify the format or the layout of the data structures in that code.

The "filter" is an $n \times n$ array of numbers. We'll go through the logistics of how a "filter" works in recitation and in class and briefly summarize it here. Basically, you an cause a number of visual affects by applying a filter to an image. The filter is implemented as a "convolution", which means that elements of the filter matrix are multiplied by the image matrix to compute a new value for the image. Pictorially, this is represented as:

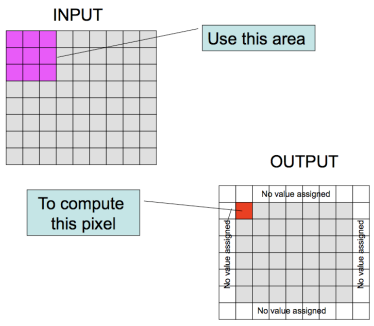

Figure 1: Filter Operation

Computationally, this is structured as five nested `for` loops (three to go over the colors, row and columns and two more to apply the filter). In the solution provided to you, filters are represented using the `Filter` class, implemented in `Filter.h` and `Filter.cpp`.

The majority of the work performed by the filter application is in the routine shown in Figure 2.

```
long long cycStart, cycStop;
rdtscll(cycStart);

output -> width = input -> width;
output -> height = input -> height;


for(int col = 1; col < (input -> width) - 1; col = col + 1) {
  for(int row = 1; row < (input -> height) - 1 ; row = row + 1) {
    for(int plane = 0; plane < 3; plane++) {
      output -> color[plane][row][col] = 0;
      for (int j = 0; j < filter -> getSize(); j++) {
        for (int i = 0; i < filter -> getSize(); i++) {
         output -> color[plane][row][col] = output -> color[plane][row][col]
          +  input -> color[plane][row + i - 1][col + j - 1] * filter -> get(i, j);
        }
      }
      output -> color[plane][row][col]
       = output -> color[plane][row][col] / filter -> getDivisor();
      if ( output -> color[plane][row][col]  < 0 ) { output -> color[plane][row][col] = 0; }
      if ( output -> color[plane][row][col]  > 255 ) { output -> color[plane][row][col] = 255; }
      output -> color[plane][row][col] = output -> color[plane][row][col];
    }
  }
}
rdtscll(cycStop);
double diff = cycStop - cycStart;
fprintf(stderr, "Took %f cycles to process, or %f cycles per pixel\n",
        diff, diff / (output -> width * output -> height));
```

Figure 2: Core of filter code

This routine is "instrumented" using the `rdscll` function. This inline function records the starting and finish times in terms of CPU cycles. We use this to determine the "cycles per element" needed to apply the filter to a given image. A sample of the output for the provided setup code looks like the following when run on the machine `perf-01.cs.colorado.edu`.

```
Took 170624720.000000 cycles to process, or 3229.816007 cycles per pixel
```

The cycle counter measures times using the CPU clock of your computer. It is fairly accurate, but many things (such as other running programs) influence the reported time. Thus, we will use the *median* of a number of runs to determine the time for a particular implementation of the program for a number of different filters.

Your job is going to be to improve the performance of this application using techniques detailed in Chapter 5 of the text. You'll be using two images (`boats.bmp` and `blocks-small.bmp`) The first image is fairly small (useful for quickly testing ideas) and the second image is larger (and used for grading / evaluation). You can run your program using a command line similar to this:

```
$ filter hline.filter boats.bmp
```

This invocation will leave the output image in `filter-hline-boats.bmp` and will report the time taken, as in the examples above. You can repeat the image name (or use different images) on the same command line to run the filter multiple times & return the average. For example:

```
$ ./filter hline.filter boats.bmp boats.bmp
Took 139255936.000000 cycles to process, or 2636.025138 cycles per pixel
Took 137527184.000000 cycles to process, or 2603.300977 cycles per pixel
Average cycles per sample is 2619.663057
```

We'll be using a script called `Judge` to score your program. The `Judge` program takes three optional argument. The `-n` argument specifies the number of times each filter should be executed, the `-i` specifies the image file and `-p` specifies the program name. The default options are shown explicitly in the following example.

```
% ./Judge -p filter -n 6 -i blocks-small.bmp
gauss: 2852.515711..2814.703339..2791.057323..2808.909958..2809.439831..2867.625660
avg: 2781.929192..2843.245386..2825.327816..2807.778870..2862.825325..2836.457027..
hline: 2923.815990..2881.138340..2889.208279..2856.241310..2899.939484..2866.064194
emboss: 2843.983543..2873.607147..2819.075932..2836.724133..2891.809189..2851.48949
Scores are 2781 2791 2807 2808 2809 2814 2819 2825 2836 2836 2843 2843 2851 2852 28
median CPE is 2851
Resulting score is 37
```

This would result in an average of 2851 cycles per second. Scoring is based on the function $log_2(CPE) * -17 + 173$ for the `blocks-small.bmp` image running on the `perf` machines. There are four machines used for scoring performance, named `perf-01.cs.colorado.edu`, `perf-02.cs.colorado.edu`, `perf-03.cs.colorado.edu` and `perf-04.cs.colorado.edu`. We use those machines to provide an "even playing field" for evaluation. You should be able to SSH into any of those systems - they don't have any of your files from the CSEL on them, and are only used for this lab.

To copy your performance lab to those machines, you can run the comman

```
% rsync -rvaP perflab-setup yourusername@perf-01.cs.colorado.edu:
```

This will prompt you for your IdentiKey password (the same one you use to log into the CSEL machines). The `rsync` command is nice because it only transfers files that differ rather than all files.

The provided Makefile compiles your program; you may need to modify it to change compiler options or the like. You can also compile your program "by hand", but you should remember what you did. We assume you are compiling your program on your virtual machine. We produce a "static" 32-bit binary that can run on the 'perf' machines. With the makefile as it is written, you can not compile on the 'perf' machines since they don't support compiling 32-bit binaries. You are free to use 64-bit binaries if you think that will help and know how.

The Makefile also provides a `make judge` rule that runs the test images and filters. Lastly, it provides a `make clean` rule to delete any temporary files or images.

Your measured time needs to include any active processing you do to the image after it is read in and before it is written out. You're free to go "whole hog" on any optimization that might work, as long **as it works with all the test images and filters included in the assignment**. Your modified code **does not** need to handle any filters or images not included in the evaluation suite. This means you can go ahead and *e.g* change the matrix layout in the BMP image library, replace the Filter library, *etc.* However, you'd be well advised to make certain those changes are important and effective before you sink a lot of time.

The most "extreme" solution is to use SIMD extensions such as MMX, SSE, SSE2 or SSE3. The cpu's on the 'perf' machines support SSE2 extensions. This lets you operate on up to 16 items in parallel at once time; equally important, it provides more registers for you to work with, which is huge on the x86. The `perf` machines also each have two CPU's or cores and you could use the OpenMP language extensions to try to use both cores. Some students have used these methods in the past, but you're well advised to go for the easy low-hanging fruit before tackling the most aggressive optimization.

2 Logistics

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

Different computers will run this code at different speeds (even measured in "cycles per second"). In order to have a level playing field, everyone must use the machines `perf-01` through `perf-04` to time their projects. When you use those machines to compare your performance, make certain no one else is using it (the "uptime" command will indicate that - we'll go over that in class).

You should do your development on your virtual machine. Since it's usually true that making something run faster on one machine makes it run faster on another, you might be able to do your development on one machine and then use the "perf" machines simply to validate your measurements or improvements.

Download the `perflab-setup.zip` file right away and unzip it. Make certain you can compile and run the distributed version and achieve times comparable to those above.

3 Grading

You should ZIP your working directory & upload that ZIP to the moodle by the due date.

You need to be able to explain *why* your code modifications improve the running time for the program and explain what would happen with minor modifications.

Again, you must be able to explain *why* you got the performance you did, so you should take notes for why you made each modification to bring to your grading meetings.