CIS 450 -- Computer Organization and Architecture – Spring 2018

Programming Lab Assignment 4: Performance Optimization Lab

Lab Dates: Mar. 16, Mar. 30, Due: Monday, Apr. 9, 2018, 11:59 pm

(50 points)

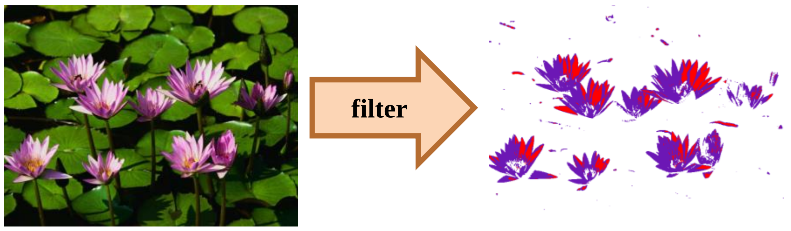
Problem Statement

This assignment deals with optimizing some memory intensive code. Image processing offers many examples of functions that can benefit from optimization. In this lab, we will consider one typical array operation, **multiply**, and two image processing operations: **filter**, which filters an image by setting each pixel in a given color range to be a fixed color and colors outside of the range to other colors, and **emboss**, which uses a convolution matrix to emboss an image.

For this lab, we will consider an image to be represented as a three-dimensional matrix M, where $M_{i,j,k}$ denotes the value of the kth color of the (i, j)-th pixel of the image denoted by M. Pixel values are triples of red, green, and blue (RGB) values (k=0 for blue, k=1 for green, and k=2 for red). Let **m** denote the number of rows, and **n** the number of columns in an image. Rows and columns are numbered, in C-style starting at 0, from 0 to m − 1, and 0 to n - 1, respectively.

Given this representation, the **filter** operation is implemented by setting pixels with any color below a lower threshold range to a given color (white in the example below), pixels within the range to a fixed foreground color specified in a file filter.txt (purple in the example RGB = (108,23,181)), and the remaining pixels, with at least one color above the upper threshold, are set to the above threshold color (red in the example RGB=(255,0,0)). The Blue, Green, Red values for the lower bound, upper bound, and filter colors are given in an input file, filter.txt:

```
64   32   32  ← lower bound (B, G, R)
245  197  250 ← upper bound (B, G, R)
255  255  255 ← below threshold color (B, G, R) = white
181  23   108 ← in range color (B, G, R)
0    0    255 ← above threshold color (B, G, R)
```



The **emboss** operation is implemented by using convolution of the image with a mask to compute rates of change (approximate derivatives) between adjacent pixels. An article by Wesley Faler in The C Users Journal discussed this technique for edge detection and embossing. The idea is to take a 3 x 3 array of numbers (called a ***kernel***) and multiply it point by point with a 3 x 3 section of the image. Then, sum the products and place the result in the center point of the given section of the image. The question in this operation is how to choose the 3 x 3 mask. Faler used several masks including:

```
-1 -1 -1
-1  8 -1
-1 -1 -1
```

For example, if all of the pixels in the 3 x 3 neighborhood have the same color, say pure red = (255,0,0), then the sum of the products would be -1*(255,0,0) + … + 8*(255,0,0) + … = (0,0,0) which is black. To make the edges show up as black, we can invert the final result by subtracting from white = (255,255,255). On the other hand, if the center pixel is red = (255,0,0), and all others are green = (0,255,0), then the sum of the products would be -1*(0,255,0) + … + 8*(255,0,0) + … = (8*255,-8*255,0), if the resulting value is greater than 255, we set the value to 255; likewise, if the value is below zero, we set the value to 0. So, the value stored would be (255,0,0) indicating that this is an edge because the color is different than its neighbors.

Many different masks have been proposed

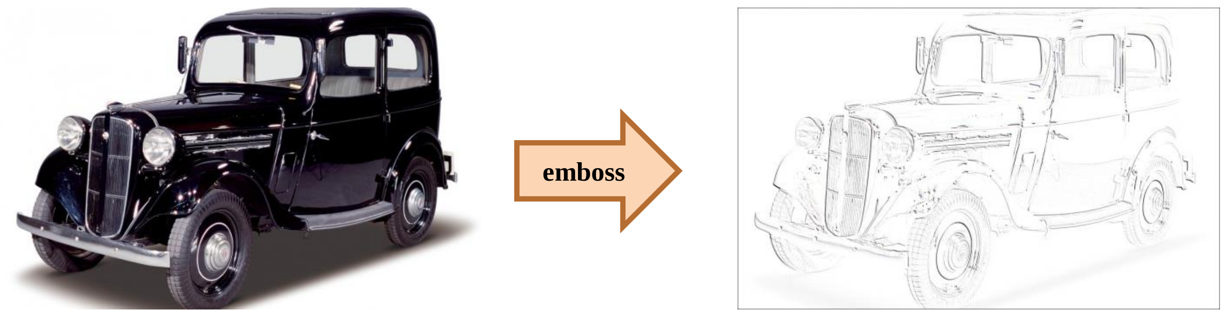
```
Kirsch              Prewitt             Sobel               Quick mask*
 5  5  5             1  1  1             1  2  1            -1  0 -1
-3  0 -3             1 -2  1             0  0  0             0  4  0
-3 -3 -3            -1 -1 -1            -1 -2 -1            -1  0 -1

-3  5  5             1  1  1             2  1  0
-3  0  5             1 -2 -1             1  0 -1
-3 -3 -3             1 -1 -1             0 -2 -2
```

*notes on convolution operators and the Quick mask is from Image Processing In C by Dwayne Phillips (the pdf file is available via K-State OnLine). In addition to edge detection, we can also emboss an image by using a simple mask with just -1 1, or combine edge detection and embossing with the second mask -- we just invented ourselves ;-):

```
 0  0  0          1  0  0
-1  1  0         -1  2 -1
 0  0  0          0  0 -1
```

Emboss **Emboss+Detect**

The image that results by the using the new Emboss+Detect mask (stored in emboss.txt) on the image datsun.bmp is shown below:



2 Logistics

You may work in a group of **up to two people** in solving the problems for this assignment. The only files that must be submitted via K-State OnLine will be your best solutions for preforming matrix multiplication, multiply.c, and image processing, filter.c and emboss.c, along with a brief summary (summary.txt or summary.pdf) of your performance evaluation comparing your solution with the given naïve solutions. Be sure to include all team members' names in the summary. Only one team member needs to upload the solutions, but both may if you don't trust your teammate ;-). Any clarifications or revisions to the assignment will be posted on the course web page. **Include the names of all group members in the summary and submit a single archive file with all three solutions and optionally the image data files created.**

3 Instructions

The code for this lab is available in the public directory /pub/cis450/programs/ as Lab4.tgz. Copy this gzipped tar file to your own working directory where you plan to work, and unzip and extract the files using the command: `tar xvzf Lab4.tgz`. This will cause a number of files to be unpacked into the working directory. The only files you will be modifying are **multiply.c, filter.c**, and **emboss.c**.

Code to evaluate the performance of the naïve and optimized routines, using a cycle timer, is embedded in the code. After computing the total time taken, the routines also compute **speedup** = time taken by naïve routine / time taken by your optimized code. To receive full credit, your code should achieve a speedup of at least 2.0; that is, it should be over twice as fast as the naïve code. To receive 80%, your code should achieve a speedup of at least 1.8. We'll have a little contest to see which code achieves the maximum speed-up. In most cases, it won't be too hard to achieve a speedup of at least 2. No compiler optimization is allowed (e.g., don't compile your code with gcc –O3, etc., although it might be fun to compare your code against the optimizing compiler on the naïve code :-), and no native operating system routines to copy memory, etc., can be used to skirt the intent of the lab which is for YOU to be the one optimizing the code; e.g., you can't just copy the naïve solution to the optimized one; e.g., c[i][j][k] = b[i][j][k] for all i, j, and k.

Optimization Techniques

Remember some of the optimization techniques that were discussed in class. The goal is to take advantage of caching and exploit both spatial and temporal locality.

1. **Blocking** – one way to optimize array operations is to block the computation so that instead of treating the entire array as a single block, the operations work on a subsection or block at a time. The loops must be adjusted to have both an outer loop and an inner loop for each index; for example, with 4x4 blocking, the outer loops:

   ```
   for (i=0; i<height; i++)
     for (j=0; j<width; j++)
   ```

   might become

   ```
   for (ii=0; ii<height; ii+=4)
     for (jj=0; jj<width; jj+=4)
     for (i=ii; i<4+ii; i++)
       for (j=jj; j<4+jj; j++)
   ```

2. **Loop Unrolling** – another way to optimize the code is to unroll part of the loop inside of a loop. This reduces the number of comparison operations that are required, and consequently speeds up the code. Remember we can pipeline the statements that are simple arithmetic operations. For example, a simple inner loop:

   ```
   for (k=0; k<3; k++)
     x[i][m-j-1][k]=x[j][i][k];
   ```

   might become:

   ```
   x[i][m-j-1][0]=x[j][i][0];
   x[i][m-j-1][1]=x[j][i][1];
   x[i][m-j-1][2]=x[j][i][2];
   ```

   So, now you're asking, why did they take off points when I did my own loop unrolling in CIS 300. Note that normally we would just ask the compiler to optimize the code for us, optimized code isn't necessarily the most readable – it just runs faster. But, for this lab you are the optimizer!

3. **Inline Procedures** – recall how the repeated call to strlen(s) doubled the running time of the code in class used to convert a string to lower-case. Instead of repeated function calls, always store the temporary results in local variables when possible, and inline the procedures within the code.

4. **Code Motion** – move operations out of the loop to the extent possible.

5. **Pointers** – use pointers to minimize address computations.

6. **Common Subexpressions** – use common subexpressions when referencing array elements.

7. **Others** – there are many more beyond the scope of this course. Consider taking a compiler course to learn more.

Good luck!

Notes: A Makefile is available in the Lab4 directory to make your life easier. The bmp file access code is available in bmp.c, the timing code is available in clock.c. These files won't need to be modified. To compile filter.c, the command gcc –g –o filter.c bmp.c clock.c is executed when you type: **make filter**. And a test is performed when you type: **make filterTest** which executes the command

```
./filter filter.txt flowers.bmp flowersBASE.bmp flowersOPT.bmp
Processor Clock Rate ~= 2533.6 MHz
Low Threshold: Blue = 64, Green = 32, Red = 32
High Threshold: Blue = 245, Green = 197, Red = 250
..
Naive CPE = 77.098396 cycles
cnt = 1
Optimized cnt = 2
Optimized cycles = 13049754.000000, MHz = 2533.599555, cycles/Mhz =
5150.677412
Optimized elapsed time per image = 0.002575 seconds
Optimized CPE = 13.593494 cycles
Speedup = 5.671713
diff flowersBASEfilter.bmp flowersOPTfilter.bmp
```

The first line is to invoke "filter" to filter out all of the pixels in flowers.bmp which are outside of the specified range and set them to white. All of the foreground pixels are set to a fixed color. The output from the naïve version is stored in flowersBASE.bmp, and the output from the optimized version is stored in the file flowersOPT.bmp. The second line verifies that the outputs are the same by using the "diff" – difference – command. Likewise, **multiply** and **emboss** can be compiled using **make** to make all 3, and **make tests** to run all of the tests. When the image file, such as flowers.bmp is loaded into memory, the original image is stored in array **a**. The naïve output is stored in array **b**, and the optimized output is stored in array **c**. Remember to update array **c** in your optimized function. In the initial version, the "optimized" version is just the same as the naïve version. You should only modify the "optimized" function. For timing purposes, the transformation is performed enough times to make an accurate reading of how long it takes to perform each transformation. Also, the cache is "warmed-up" by running the code one time in both cases – just to be fair. Your optimized code should be at least twice as fast as the naïve version of the code; i.e., speedup >= 2.0. To test the code, you can just use the simple make commands: **make filterTest**, **make multiplyTest**, or **make embossTest**.

To Turn In: Create a compressed, tar archive of the files in Lab4: (a) go to the Lab4 directory with your solution and issue the command: **make clean** – this will reduce the size of the output, (b) then issue the command: **cd ..** – go up one level, and (c) issue the command: **tar cvzf Lab4.tgz Lab4**, and upload the file Lab4.tgz via K-State OnLine.

Grading Rubric:

Multiply:	**Speedup > 1.3: +4, > 1.5: +6, > 1.8: +8, > 2.0: +10**
Filter:	**Speedup > 1.5: +8, > 2.0: +12, > 3.0: +16, > 4.0: +20**
Detect:	**Speedup > 1.3: +8, > 1.5: +12, > 1.8: +16, > 2.0: +20**