CS 2461
Project 6: Code Optimization

This is the final project and is in lieu of a required final exam. It requires both coding and a report – we expect you will spend enough time to provide a report with detailed analysis (your grade will depend on both the report and the code performance).

1 Introduction

This assignment deals with optimizing memory intensive code. Image processing offers many examples of functions that can benefit from optimization. In this project, we will consider two image processing operations: `rotate`, which rotates an image counter-clockwise by $90°$, and `smooth`, which "smooths" or "blurs" an image. You have probably used these operations when dealing with images (in Photoshop, Paint, etc.).

For this project, we will consider an image to be represented as a two-dimensional matrix M, where $M_{i,j}$ denotes the value of (i, j)th pixel of M. Pixel values are triples of red, green, and blue (RGB) values. We will only consider square images. Let N denote the number of rows (or columns) of an image. Rows and columns are numbered, in C-style, from 0 to $N - 1$.

Given this representation, the `rotate` operation can be implemented quite simply as the combination of the following two matrix operations:

- *Transpose*: For each (i, j) pair, $M_{i,j}$ and $M_{j,i}$ are interchanged.
- *Exchange rows*: Row i is exchanged with row $N - 1 - i$.

This combination is illustrated in Figure 1.

The `smooth` operation is implemented by replacing every pixel value with the average of all the pixels around it (in a maximum of 3×3 window centered at that pixel). Consider Figure 2. The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below:

$$\texttt{M2[1][1]} = \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[i][j]}}{9}$$

$$\texttt{M2[N-1][N-1]} = \frac{\sum_{\texttt{i}=N-2}^{N-1} \sum_{\texttt{j}=N-2}^{N-1} \texttt{M1[i][j]}}{4}$$

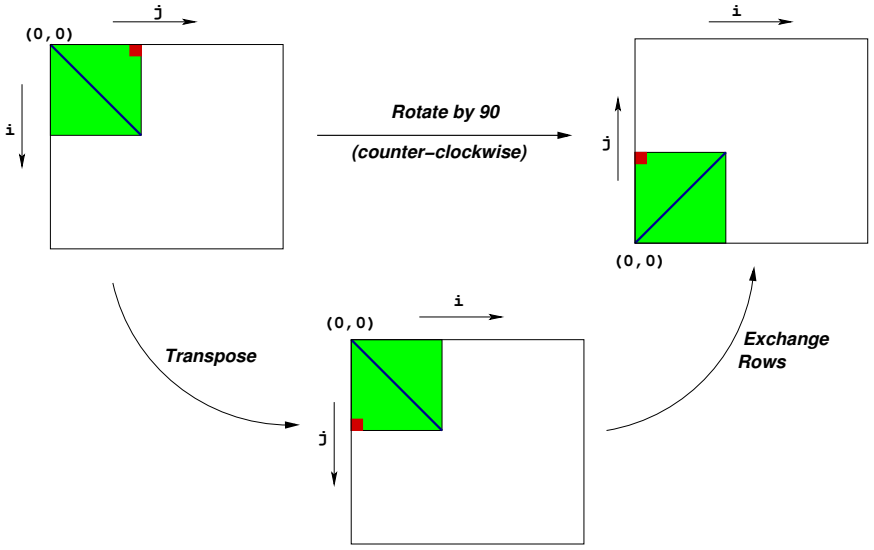

Figure 1: Rotation of an image by 90° counterclockwise

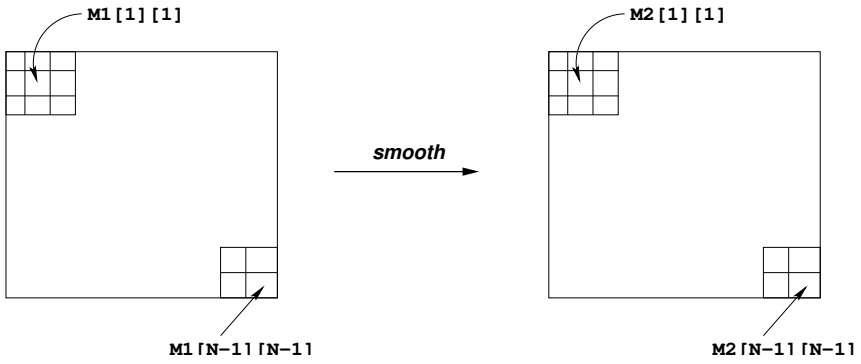

Figure 2: Smoothing an image

2 Logistics

You **must** work alone in solving the problems for this project. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page. There is NO collaboration of any sort allowed on this project – this includes outside sources (online or otherwise) and **you cannot discuss solution strategies with other students.** However, you can use any/all of the notes (and solutions) from the in-class group discussions. You are allowed to post questions related to compiler errors, C, and shell issues on Piazza - but do not post questions related to specific solutions you are exploring.

The project source files and installation are provided. The source code and project are based off a project provided by Carnegie Mellon University.

3 Instructions

Start by copying the files in project6 assignment to your protected directory on shell.seas in which you plan to do your work. There are a number of files in this directory. **The only file you will be modifying and handing in is** `kernels.c`. The `driver.c` program is a driver program that allows you to evaluate the performance of your solutions. It will also print out error messages if your modified code is not equivalent to the original algorithm/code. Use the command `make driver` to generate the driver code and run it with the command `./driver`.

Looking at the file `kernels.c` you'll notice a C structure `team` into which you should insert the requested identifying information about yourself. **Do this right away so you don't forget.**

4 Implementation Overview

Data Structures

The core data structure deals with image representation. A `pixel` is a struct as shown below:

```
typedef struct {
   unsigned short red;   /* R value */
   unsigned short green; /* G value */
   unsigned short blue;  /* B value */
} pixel;
```

As can be seen, RGB values have 16-bit representations ("16-bit color"). An image `I` is represented as a one-dimensional array of `pixel`s, where the (i, j)th pixel is `I[RIDX(i,j,n)]`. Here `n` is the dimension of the image matrix, and `RIDX` is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

See the file `defs.h` for this code.

Rotate

The following C function computes the result of rotating the source image `src` by 90° and stores the result in destination image `dst`. `dim` is the dimension of the image.

```
void naive_rotate(int dim, pixel *src, pixel *dst) {
  int i, j;
   for(j=0; j < dim; j++)
     for(i=0; i < dim; i++)
      dst[RIDX(dim-1-j,i,dim)] = src[RIDX(i,j,dim)];

  return;
}
```

The above code scans the rows of the source image matrix, copying to the columns of the destination image matrix. Your task is to rewrite this code to make it run as fast as possible using some of the techniques we discussed in the lecture.

See the file `kernels.c` for this code.

Smooth

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. Here is part of an implementation:

```
void naive_smooth(int dim, pixel *src, pixel *dst) {
  int i, j;

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(i,j,dim)] = avg(dim, i, j, src); /* Smooth the (i,j)th pixel */

  return;
}
```

The function `avg` returns the average of all the pixels around the `(i,j)` th pixel. Your task is to optimize `smooth` (and `avg`) to run as fast as possible. (*Note:* The function `avg` is a local function and you can get rid of it altogether to implement `smooth` in some other way.)

This code (and an implementation of `avg`) is in the file `kernels.c`. See the file `kernels.c` for this code.

Performance measures

Our main performance measure is the speedup of optimized code (i.e., your versions) to the naive (original) version. The driver will output the performance of both the naive method and your method for each of the algorithms (smooth and rotate). We are using two system utilities to measure the performance – (1) the number of (processor) cycles it takes to run for an image of size $N \times N$ for different values of N and (2) the execution time in milliseconds. If you examine the kernels.c code you will find that I am using the timestamp counter (rdtsc) and usage time. Both of these have problems when it comes to accurate measurements – specifically, they are affected by other system tasks. (So

you will may get weird times sometimes; including differing times each time you run the project. THIS is why the report is very important.)

The improvement in performance is usually measured one of two ways: (a) the difference in the old and new times as a percentage of the old time or (b) the ratio of the old execution time to the new execution time (i.e, the speedup which signified how much faster). While metric (a) is the more classical approach, in this project you can use the simpler metric of ratio.

- Speedup $S = T_{naive}/T_{opt}$ where T_{naive} is the original time and T_{opt} is the optimized time. For example if $T_{naive} = 600$ and $T_{opt} = 400$ then $S = 600/400 = 1.5$ which we refer to (in this project) as a 50% improvement. So a $x\%$ improvement in this context means $T_{naive}/T_{opt} = 1.x$.
- The ratios (speedups) of the execution time of the optimized implementation over the naive one will be used to compute your grade for your implementation.

(The classic method of measuring percentage improvement would be $\sigma = (T_{naive} - T_{opt})/T_{naive}$.)

Assumptions

To make life easier, you can assume that N is a multiple of 32. Your code must run correctly for all such values of N, but we will measure its performance only for large sizes (of 512 or greater).

5 Infrastructure

We have provided support code to help you test the correctness of your implementations and measure their performance. This section describes how to use this infrastructure. The exact details of each part of the assignment is described in the following section. The code handed out will test and measure the performance of your solutions. We used the system `clock` utility to measure the time and the timestamp counter `rdtsc` to count clock cycles – as a result, you can get different times and/or cycles each time you run the code. Because the system time is being included in these counts (process swapping time, memory fetch time, etc.), don't be surprised if the numbers don't always match with what you expect. **Your final code MUST run on shell.seas.gwu.edu without any errors – so make sure you test it on shell (and check the measrurements on shell) before submitting your project.**

To get a better measurement (of how well your optimized code performs), we recommend that you run the code on your machine with no other processes running on the machine. (This will minimize the interference from other process and give you a more accurate measurement.) You can include these measurements (i.e, running the code on your standalone machine) in your report. (For the adventurous: you can get more accurate measurements by specifying a higher priority for your process/program. Look up " sudo nice" as a starting point - you need to have super user privilege to execute a sudo command, so do not try this on shell).

The performance measurement also takes the average of running your code 10 times. So you do not need to test multiple times (although you are not discouraged from doing so).

Note: The only source file you will be modifying is `kernels.c`; in fact **you will only be writing the code for my_rotate and my_smooth**.

Driver

The source code you will write will be linked with object code that we supply into a `driver` binary. To create this binary, you will need to execute the command

```
unix> make driver
```

Note: You will need to re-make (i.e., `make clean`) driver each time you change the code in `kernels.c`. It is possible that when you run make, the compiler may throw up a few warnings – ignore them and run the code.

To test your implementations, you can then run the command:

```
unix> ./driver
```

System Details

Your solutions will be evaluated on the SEAS shell server (shell.seas.gwu.edu or shell01.seas.gwu.edu). You MUST use these machines to measure the performance of your solutions. You MUST use the gcc compiler, with no compiler optimization options. It is important to note that we will only be using the shell system to measure the performance of your solutions – improvements made on any other system will not be taken into account during our grading. However, you are free to use your preferred OS during development.

Your Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about your name etc.

6 Project Requirements

The goal of this project is for you to explore applying different code optimization techniques with the objective of getting the two programs to run faster. We require that you try out different techniques, and then describe which techniques were effective, or not effective, and why or why not. This analysis, which should be provided in the report, is **as important** as getting the code to run - and your grade will be based off **both** the report and the code. The report will play a very large role in your grade – if you have not explained your logic correctly, your grade will suffer even if your code works. You must use the optimization techniques covered in the lectures – any other techniques you used must be explained (i.e., what they are, why they work and proof of correctness for why they work – if you are not clear about what a proof of correctness is, then you should not even be considering any optimizations not covered in class). in addition to explaining why they improve this specific code. (The objective is to show how program performance can be improved by using only a small menu of optimization techniques.) The grade on this project will be based on both the report correctness (and depth) and the improvement (i.e., how much faster your version runs compared with the naive version). The minimum improvement we expect (for a passing grade) is 30% – you will find that you can get much higher improvements by trying out different optimizations. The report will be used to gauge how well you were able to analyze why the optimizations worked (for example, simply stating "improved locality" is not an in-depth analysis and will be considered as no explanation – you will need to explain how or why the locality was improved).

Optimizing Rotate

In this part, you will optimize `rotate` to achieve as low a time (cycles) as possible. You should compile `driver` and then run it with the appropriate arguments to test your implementations.

Optimizing Smooth

In this part, you will optimize `smooth` to achieve as low a time (cycles) as possible. You should compile `driver` and then run it with the appropriate arguments to test your implementations.

Comments: The two functions have different computational requirements. The `smooth` is more compute-intensive and less memory-sensitive than the `rotate` function, so the optimizations are of somewhat different flavors.

Coding Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.
- You can use **ONLY** the code optimization techniques we discussed in the lectures/course. In the report, you should describe which techniques worked well and why, and if applicable then which techniques did not work well.

You can only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in this file.

Grading

This project will be graded out of 60 points. Your grade will depend on the number of optimizations you have tried, produce significant speedups over the naive version, **and** the report explaining your experiments. If you choose to optimize only one of the two functions (Rotate and Smooth), then you can earn a maximum of upto 35 points depending on how well you have optimized the code and your report.

Your grade (for each algorithm) will be based on the following:

- **Correctness:** You will get NO CREDIT for buggy code that causes the driver to complain! This includes code that correctly operates on the test sizes, but incorrectly on image matrices of other sizes. As mentioned earlier, you may assume that the image dimension is a multiple of 32.
- **Performance:** in terms of the number of processor cycles, and/or time, taken by your code. The minimum performance improvement (speedup) that your code must achieve for each function was specified earlier in this document. Your solutions must provide at least this speedup to get a passing grade on this project.
- **Report:** Your report must (for each of the two functions `smooth` and `rotate`), describe the optimizations you used and why you think they will improve the performance. Remember that this final project is in lieu of a required final exam – so we expect you to spend enough time preparing a report with a thorough analysis. At a minimum, your report must describe:
  - (i) the optimization techniques you tried (you may end up using only a subset of these in the code that performs best) for each algorithm (smooth and rotate)
  - (ii) the set of optimizations you tried and how effective they were.
  - (iii) provide a complete table of your performance results (i.e., a table showing how much better you did compared to the naive implementations – the performance improvement in speed). Simply including the output of the program does NOT constitute a report!

- (iv) Which optimizations gave you the best performance and why you think they improved the performance. If some of the optimizations did not provide much of an improvement then explain why they did not.
- If you chose a specific technique, or a value for a parameter, over another technique your report should provide your rationale (or experimental results, if that was how you reached your decision) for these choices. We will give you (significant) partial credit if your report is correct but your implementations do not provide the minimum speedups we require.
- **Failure to submit the report will result in a grade of zero points regardless of how well your code does. You must provide justifications and MUST submit the report – failure to submit this report will result in a grade of zero for the entire project, regardless of how well your code performed; i.e., if you cannot explain and justify why your code ran faster then you get no credit for the project.**

7 Hand In Instructions

When you have completed the project, you will hand in (1) a report (in PDF or MS-word) describing your solutions and (2) your optimized `kernels.c`, that contains your solution.

Here is how to hand in your solution:

- You will submit your report and your kernels.c code only (i.e., do not change the driver.c or defs.h or makefile). Your report must be in PDF format, and must be titled First Initial Lastname-report6 (For example, if your name is Jane Smith then your filename is JSmith-report6.pdf).
- Make sure you have included your identifying information in the team struct in `kernels.c`.
- Make sure that the `rotate()` and `smooth()` functions correspond to your fastest implementations, as these are the only functions that will be tested when we use our driver to grade your assignment.
- Remove any extraneous print statements.
- Create a team name of the form:
  - "*ID*" where *ID* is your ID (you can use your GW ID)