# 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, we will consider an image processing operation `smooth`, which "smooths" or "blurs" an image.

For this lab, 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 $N-1$.

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

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

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

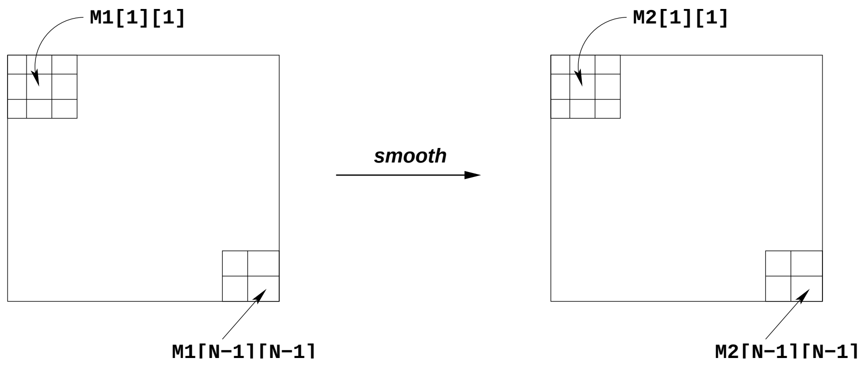

Figure 1: Smoothing an image

## 2 Logistics

The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

## 3 Hand Out Instructions

Start by copying `perflab-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf perflab-handout.tar`. This will cause a number of files to be unpacked into the 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. 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 the one or two individuals comprising your programming team. **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.

### 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;
```

| Test case | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Method N | 32 | 64 | 128 | 256 | 512 | Geom. Mean |
| Naive `smooth` (CPE) | 695 | 698 | 702 | 717 | 722 | |
| Optimized `smooth` (CPE) | 41.5 | 41.6 | 41.2 | 53.5 | 56.4 | |
| Speedup (naive/opt) | 16.8 | 16.8 | 17.0 | 13.4 | 12.8 | 15.2 |

Table 1: CPEs and Ratios for Optimized vs. Naive Implementations

```
  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`.

### Performance measures

Our main performance measure is *CPE* or *Cycles per Element.* If a function takes $C$ cycles to run for an image of size $N \times N$, the CPE value is $C/N^2$. Table 1 summarizes the performance of the naive implementations shown above and compares it against an optimized implementation. Performance is shown for 5 different values of $N$. All measurements were made on a Pentium III machines, and hence are not likely to reflect the values on your current machine.

The ratios (speedups) of the optimized implementation over the naive one will constitute a *score* of your implementation. To summarize the overall effect over different values of $N$, we will compute the *geometric mean* of the results for these 5 values. That is, if the measured speedups for $N = \{32, 64, 128, 256, 512\}$ are $R_{32}$, $R_{64}$, $R_{128}$, $R_{256}$, and $R_{512}$ then we compute the overall performance as

$$R \quad = \quad \sqrt[5]{R_{32} \times R_{64} \times R_{128} \times R_{256} \times R_{512}}$$

### 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 the 5 values shown in Table 1.

## 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.

**Note:** The only source file you will be modifying is `kernels.c`.

## Versioning

You will be writing many versions of the `smooth` routine. To help you compare the performance of all the different versions you've written, we provide a way of "registering" functions.

For example, the file `kernels.c` that we have provided you contains the following function:

```
void register_smooth_functions() {
   add_smooth_function(&smooth, smooth_descr);
}
```

This function contains one or more calls to `add_smooth_function`. In the above example, `add_smooth_function` registers the function `smooth` along with a string `smooth_descr` which is an ASCII description of what the function does. See the file `kernels.c` to see how to create the string descriptions. This string can be at most 256 characters long.

## 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
```

You will need to re-make driver each time you change the code in `kernels.c`. To test your implementations, you can then run the command:

```
unix> ./driver
```

The `driver` can be run in four different modes:

- *Default mode*, in which all versions of your implementation are run.
- *Autograder mode*, in which only the `smooth()` functions are run. This is the mode we will run in when we use the driver to grade your handin.
- *File mode*, in which only versions that are mentioned in an input file are run.
- *Dump mode*, in which a one-line description of each version is dumped to a text file. You can then edit this text file to keep only those versions that you'd like to test using the *file mode*. You can specify whether to quit after dumping the file or if your implementations are to be run.

If run without any arguments, `driver` will run all of your versions (*default mode*). Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-g` : Run only `smooth()` functions (*autograder mode*).

`-f <funcfile>` : Execute only those versions specified in `<funcfile>` (*file mode*).

`-d <dumpfile>` : Dump the names of all versions to a dump file called `<dumpfile>`, *one line* to a version (*dump mode*).

`-q` : Quit after dumping version names to a dump file. To be used in tandem with `-d`. For example, to quit immediately after printing the dump file, type `./driver -qd dumpfile`.

`-h` : Print the command line usage.

### Team Information

**Important:** Before you start, you should fill in the struct in `kernels.c` with information about your team (group name, team member names and email addresses). This information is just like the one for the Data Lab.

## 6 Assignment Details

### Optimizing Smooth

In this part, you will optimize `smooth` to achieve as low a CPE as possible.

For example, running driver with the supplied naive version (for `smooth`) generates the output shown below:

```
unix> ./driver

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       695.8   698.5   703.8   720.3   722.7
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

**Some advice.** Look at the assembly code generated for `smooth`. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class.

### 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 only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in these files.

### Evaluation

The score 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.

- CPE: You will get full credit for your implementations of `smooth` if it is correct and achieve mean CPEs above threshold $S_s$ (to be determined) respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.

## 7 Hand In Instructions

When you have completed the lab, you will hand in one file, `kernels.c`, that contains your solution. Here is how to hand in your solution:

- Make sure you have included your identifying information in the team struct in `kernels.c`.
- Make sure that the `smooth()` function correspond to your fastest implementation, as it is the only function that will be tested when we use the driver to grade your assignment.
- Remove any extraneous print statements.
- Create a team name of the form:
  - "*ID*" where *ID* is your Andrew ID, if you are working alone, or
  - "$ID_1$+$ID_2$" where $ID_1$ is the Andrew ID of the first team member and $ID_2$ is the Andrew ID of the second team member.

  This should be the same as the team name you entered in the structure in `kernels.c`.
- To handin your `kernels.c` file, type:

  ```
  make handin TEAM=teamname
  ```

  where `teamname` is the team name described above.

Good luck!