CS 33, Winter 2017
Parallel Performance Lab
Assigned: February 28th
Due: March 15th, 11:55PM

1 Introduction

This assignment deals with applying performance and parallelism optimizations to data-intensive code. Image processing offers many examples of functions that can benefit from optimization. In this lab, we will consider the `smooth` kernel, 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 3×3 window centered at that pixel). Consider Figure 1. The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below:

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

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

2 Lab Setup

Please work individually, you will only hand in the kernels.c file to CCLE, similar to lab 1.

The lab will be here: /w/class.1/cs/cs33/cs33w17/perflab-handout.tar

Start by copying `perflab-handout.tar` to a 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`.

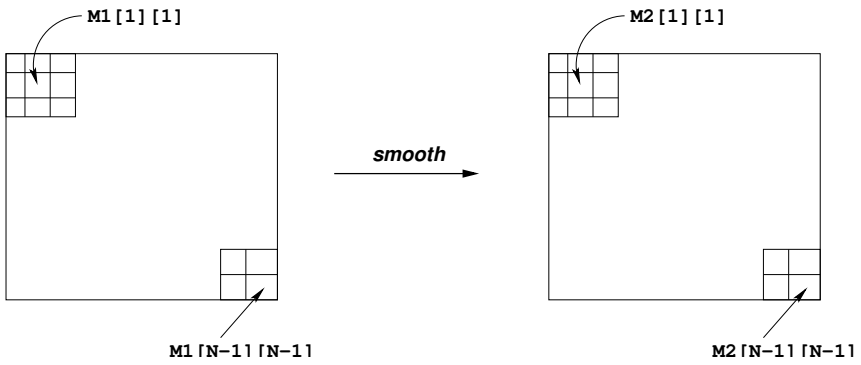

Figure 1: Smoothing an image

Looking at the file `kernels.c` you'll notice a C structure `team` into which you should insert the requested identifying information. **Do this right away so you don't forget.** (Even though the structure says team, you'll be working alone).

3 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 `pixels`, 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;

  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`.

Ignore everything related to `rotate`, you will not be implementing this function.

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.

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 $N = \{64, 128, 256, 512, 1024\}$ are S_{64}, S_{128}, S_{256}, S_{512}, and S_{1024} then we compute the overall performance as

$$S \quad = \quad \sqrt[5]{S_{64} \times S_{128} \times S_{256} \times S_{512} \times S_{1024}}$$

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: 64, 128, 256, 512, 1024.

4 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 likely be writing many versions of the code. 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.

A similar function for your smooth kernels is provided in the file `kernels.c`.

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 final version (`smooth()`) is 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.

Identity Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about yourself. This information is just like the one for the Data Lab.

5 Assignment Details

Optimizing Smooth (50 points)

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             64      128     256     512     1024    Mean
Your CPEs       76.3    78.8    95.1    115.3   113.7
Baseline CPEs   77.0    79.0    96.0    115.0   115.0
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Some advice. In order to get good performance, you will need to apply both traditional single-threaded code optimizations, as well as to multithread(!!!) your code.

For the traditional optimizations, remember to focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class. That means loop unrolling, function inlining, cache locality (loop-reordering), etc. Think about it like your enemy is function calls and "if" statements. An important tip here is that you may want to create multiple loops for handling "corner cases"[1], as this will help reduce the number of branching statements. You may want to look at the assembly code generated, though it is not required.

To multi-thread the code, we recommend using pthreads. The pthread libraries are already included, so you should be able to use them without modifying the makefiles or code. Remember, you'll need to partition the work across different threads – there are lots of choices for how to do this, some are more cache friendly than others.

The basic strategy then is:

- Create several threads, store their ids in an array for later.
- Pass each thread an "id" indicating which part of the matrix to work on.
- (do the work in parallel)
- Join the threads using the saved ids to wait until they are complete.

Keep in mind that the optimal number of threads for any given matrix size might be different. If your matrix is small, the overheads of starting many threads might actually completely outweigh the benefits of multi-threading!

[1] Here I mean corner quite literally.

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 must NOT DEADLOCK. You will get 0 points if you deadlock. (if your code never finishes)

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

Evaluation

- 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.
- Speed: Your goal will be to get a CPE which is six times lower than the naive version (geomean speedup of $6\times$). If you do not reach 6, your grade will be linearly interpolated from there. Grade = (speedup -1)/5. We may adjust the grading to be easier if the average score is too low. Since there is some variability in execution time when using threads, try to make sure your CPE is consistently above 6. Also, we will do the grading on `lnxsrv06`, so you may want to eventually test there – though it will be a bad idea for everyone to overload that machine, as you may end up with pessimistic timing results. :)
- Bonus: The top 10 solutions will get 10% bonus points, and the top 3 will get 20% bonus points.

6 Hand In Instructions

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

- Make sure you have included your identifying information in the struct in `kernels.c`.
- Make sure that the `smooth()` function corresponds to your fastest implementation – we will grade this only.
- Remove any extraneous print statements.

Good luck!