CS 426/525
Fall 2022
Project 3
Due 18/12/2022

1. Problem Definition

In this project, you will denoise image data by averaging and experimenting with a motion blur filter. You will first implement the serial version, then parallelize it with OpenMP API. You can use C or C++ in this project.

2. Denoising Images with Average Filtering

Average (or mean) filtering is a method of 'smoothing' images by reducing the intensity variation between neighboring pixels. The average filter moves through the image *pixel by pixel*, replacing each value with the average value of adjacent pixels, including itself.

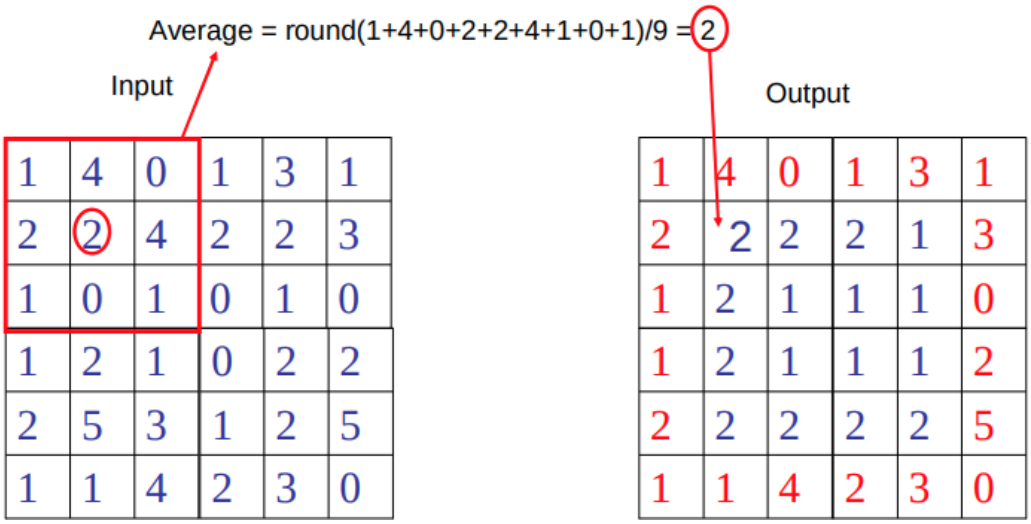

Figure 1: *2D Average filtering example using a 3 x 3 sampling window, keeping border values unchanged.*

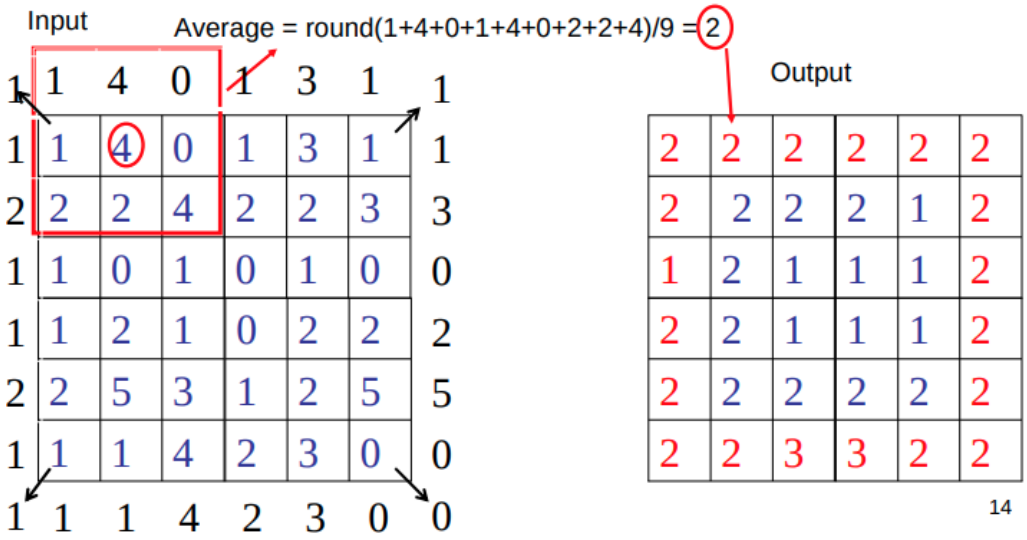

Figure 2: *Extending the borders to make it smoother.*

We define a window to slide through pixels and replace their intensity values with the average intensity. We define a window to slide through pixels and replace their intensity values with the average intensity calculated inside that window. However, there is a slight problem for pixels at the edge of the matrix, as some values to be calculated within the window would be out of the bounds of the input matrix. To handle the edge pixels, we will apply the edge-extend method, where the edge pixels are extended as far as necessary to provide values for the averaging.

You may assume that the window is a square one with an odd value of pxp, **the extension amount** for each side of the input matrix can be calculated as **floor(p/2)**.

There are better options than averaging for denoising, but it will be enough for our context. Moreover, it becomes more effective if the noise type is salt and pepper. You can generate synthetic denoised images and test them out.

3. Adding Motion Blur on Image

Before explaining the motion blur filter, reviewing the convolution operation is a good idea. Convolution is a widely used technique in the signal-processing domain that operates on linear time-invariant systems. Discrete 2D convolution is very useful for image manipulation in the image processing field and feature extraction in deep learning applications with the advances in Convolutional Neural Networks(CNN). In this context, a grayscale image is given as an integer matrix (2D array) with values between 0 and 255. A kernel is also given as a small matrix, usually 3x3,5x5,7x7, etc. We “convolve” the image matrix with the kernel matrix and get an output matrix of the same size as the image matrix. In general, for an input matrix A of mxn and kernel of pxq, the output matrix has m+p-1 rows and n+q-1 columns, but we apply the convolution in a slightly different way to keep the output image with the same size as the input image.

When we convolve an input matrix with the kernel, we place the center of the kernel matrix onto each location in the input matrix, then perform a multiply-accumulate (MAC) operation with the kernel and overlap pixel values in the input image. In this MAC operation, we multiply the overlapping values in the kernel and image pixels; then, we sum up all these multiplied values. In Figure 3, we placed the center of the kernel onto the location [1,1] in the input matrix where pixel value 99 lies, then did the following computation:

105 x 0 + 102 x (-1) + 100 x 0 +
103 x (-1) + 99 x 5 + 103 x (-1) 0 +
101 x 0 + 98 x (-1) + 104 x 0
= **89**

So we put the value 89 at the same location [1,1] in the output matrix. We slide the kernel over each pixel value to repeat the operation to fill the output matrix.

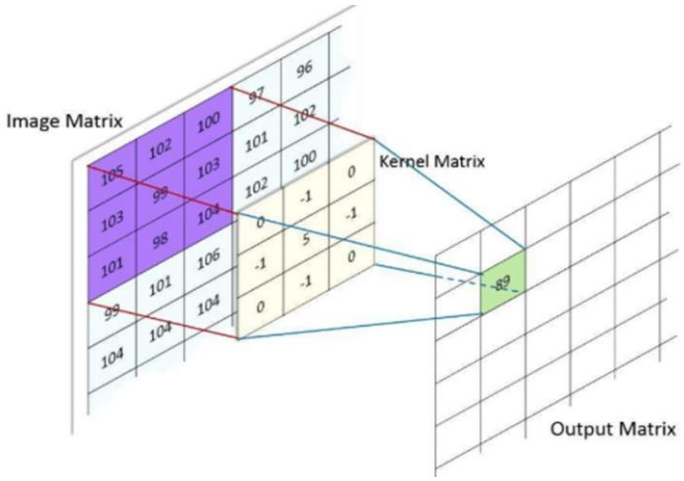

Figure 3: *Convolution operation.*

In motion blur, we apply blur in one direction, such that we will use a square identity matrix filter as our kernel and do the convolution operation. Doing such causes the pixels along the diagonal become more prominent.

$$\begin{pmatrix} 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \dots & \dots & \dots & 1 \end{pmatrix}$$

Lastly, we normalize our pixel values by dividing each pixel by the sum of the values in our filter, i.e., the identity matrix. For example, if our filter is a 7x7 identity matrix, the normalization amount will be simply 7*1=7.

You can play around with filter size, magnitude, and orientation to see how your blurred image will look. For example, if we use an exchange matrix instead of an identity matrix, we will get the image blurred in the opposite direction.

NOTE: You can consider denoising operation as the convolution of a matrix of ones (in the size of the window) on a noisy image. Since we are already averaging the pixels, normalizing is not needed.

4. Implementation Details

- First, you will implement the sequential version of image operations, then copy it into another file and parallelize it with OpenMP API. 5 functions will be implemented. You can change their return types as it fits your code.

- void **extend_edges** (int **img, int row_index, int col_index, int extend_amount)
  Extend the edges of the image by re-allocationg the 2d array **img given extend_amount.

- int **average_operation** (int window_size, int **img, int row_index, int col_index)
  Average operation at given location of the image.

- int ****denoise_image** (int window_size, int **img, int num_rows, int num_cols)
  This method expected to call average operation on whole image. Finally returns denoised image matrix.

- int **blur_operation** (int **kernel, int kernel_size, int **img, int num_rows, int num_cols)
  Convolution operation with blur filter at given location of the image.

- int **blur_image** (int **kernel, int kernel_size, int **img, int num_rows, int num_cols)
  This method expected to call blur operation on whole image. Finally returns normalized and blurred image matrix.

- You will profile your code using gprof and try to find the functions that are needed to be parallelized.
  - For gprof details, there are many online tutorials.
- Second, you will implement parallel version of image operations.
  - You will insert OpenMP pragmas to parallelize your code.
- You will profile your code again using gprof.
- You can download and use the util.h and util.c files. In these files, 2D array allocate and free functions, file reading functions are implemented.

- Your program will take two single command line inputs image and kernel (filter/window) which are stored in ASCII files whose first two lines gives the number of rows *m* and the number of columns *n* for the matrix (image), and the following *m* lines gives a lists of space-separated integer *n* elements of each row. In our case, kernel will be a square matrix, so *m* and *n* will be equal. **Assume size of the window in denoising, that is part A, is equal to the size of blur filter in part B.** So that you won't need aditional input argument for the size of the window.
- You are welcome to use **read_pgm_file** (char * file_name, int *num_rows, int *num_columns) to read the input matrices.
- The program generates two output files; first one is the denoised image matrix and the second one is blurred image matrix, in the same format as the input.
- You will also print parallel execution time and sequential execution time spent on the *blur_image* and *denoise_image* functions.

```
~$cat my_image.txt
3
5
51 53 53 3 52
89 2 3 127 167
21 44 2 78 84

~$cat kernel.txt
3
3
1 0 0
0 1 0
0 0 1

~$./filtering_omp my_image.txt kernel.txt denoise.txt blur.txt

Parallel time, denoise: XX.XX ms
Sequential time, denoise : YY.YY ms
Parallel time, blur: XX.XX ms
Sequential time, blur : YY.YY ms

~$cat denoise.txt
3
5
54 45 38 57 75
46 35 40 63 90
39 25 42 69 105
```

```
~$cat blur.txt
3
5
34 35 77 74 74
61 18 44 88 84
51 45 27 55 98
```

Run your code with various images, and kernels (use symmetric matrices for kernel).

5. But How Do My Results Look Like, I Tought They Were Images

If you want to see how your program works and what do your input images look like you can use the python script show_images.py we shared, you should just give the filename having same file format with input files as an argument to the program. It should generate a jpeg file for you, you need to install NumPy and Pillow libraries for your python environment to run the script. So to run, you should call it as:

```
~$python show_images.py my_image.txt
```

6. Grading

- filtering_seq: 20 points
- filtering_omp: 40 points
- gprof profiling results: 10 points
- Report: 30 points

7. Submission

Put everything under same directory, do not structure your project under directories like parallel, sequential etc. Put all of them in the same directory, one called **yourname_lastname_p3**. You will zip this directory and submit. When it is unzipped, it should provide the directory and files inside.

1. Your code:
   - filtering_seq.c, filtering_omp.c file and any other file that you have implemented
     a. If you used given files util.h util.c, also include them in your submission
     b. filtering_seq.c will include your sequential implementation and filtering_omp.c will include your parallel implementation.
   - A compile script that can compile your code
     a. Name this script as compile.sh
     b. It will produce 2 executables, filtering_seq and filtering_omp
2. Profiling results, where you will submit gprof profiling results
   - prof_sequential.txt file for sequential implementation's profiling results
   - prof_omp.txt file for parallel implementation's profiling results
     - It will include profiling results for several runs with different number of threads
3. Your report:

- **Reports should be in .pdf format.**
  - Submissions with wrong format will get 0.
- Detailed description of gprof's profiling outputs.
- Detailed description of your implementation details
  - Explain pragmas that you have used
    - Why did you insert this pragma to this specific region?
    - What does this pragma do?
    - What are the possible options that can be used with this pragma?
    - What are the options that you have used?
- Plot for accuracy results.
- Plot for execution times with different threads.
- Discussion of your results
  - Don't forget to use gprof's profiling outputs in your discussion.

- **Zip File name to upload:** yourname_lastname_p3.zip
- **No Late Submission Allowed!**