CS 426/525
Fall 2020
Project 3
Due 14/12/2020

Problem Definition:

In this project, you will implement a parallel 2D convolution on image data. You will first implement the serial version, then parallelize it with OpenMP API.

2D Convolution in Images:

Convolution is a widely used technique in signal processing domain that operates on linear time-invariant systems. Discrete 2D convolution is very useful for image maniulation in image processing field and for feature extraction in deep learning applications with the advances on Convolutional Neural Networks(CNN). In this context, a grayscale image is given as an integer matrix(2D array) whose values lie between 0 and 255. A kernel is also given as a small matrix, usually of size 3x3,5x5,7x7 etc. We “convolve” image matrix with kernel matrix and get an output matrix of same size with 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 input image). When we convolve an input matrix with the kernel, we place the center of kernel matrix onto each location in input matrix, then perform a multiply-accumulate(MAC) operation with kernel and overlapping pixel values in input image. In this MAC operation, we basically multiply the overlapping values in kernel and image pixels, then we sum up all these multiplied values. In Figure 1, we placed the center of kernel onto the location [1,1] in the input matrix where pixel value 99 lies, then do the following computation:

105x0 + 102x(-1) + 100x0 + 103x(-1) + 99x5 + 103x(-1)0 + 101x0 + 98x(-1) + 104x = 89

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

However, there is a slight problem for pixels at the edge of the matrix, as some values to be multiplied with kernel would be out of bounds of the input matrix. To handle the edge pixels, we are going to apply edge-extend method, the edge pixels are extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. You may assume that kernel matrix a square one of an odd value pxp, then extension amount for each side of the input matrix can be calculated as floor(p/2). For given input matrix in Figure 2, the edge-extend method for a kernel matrix of size 5x5 is done as in Figure 3. Kernel still slides over the real pixel values, the extended values are just for MAC operations.

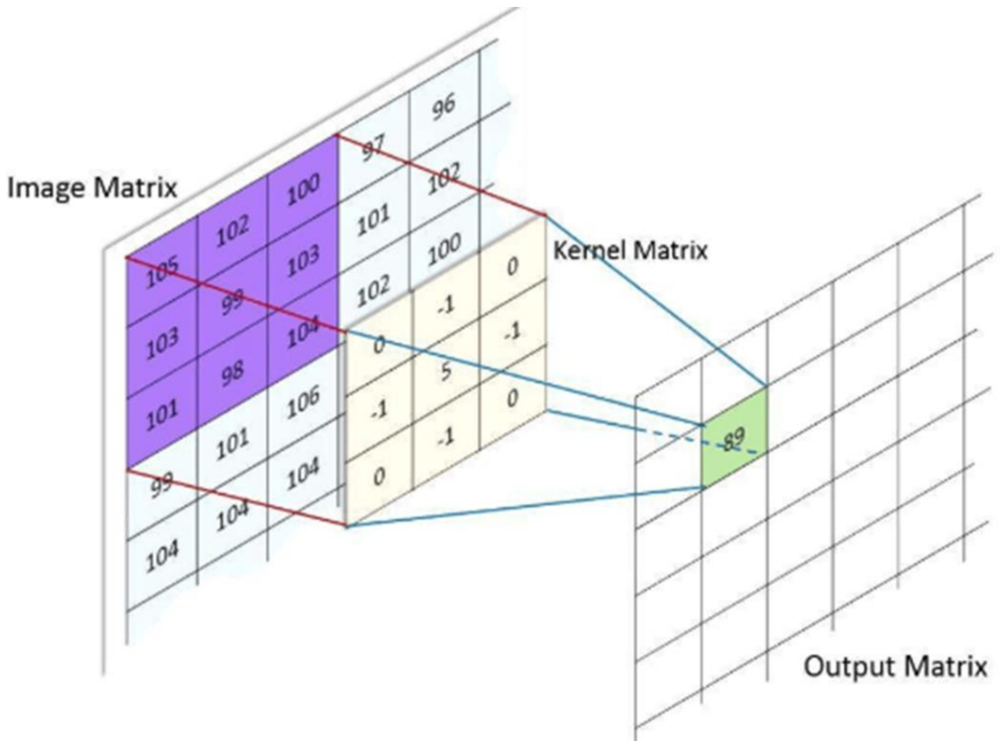
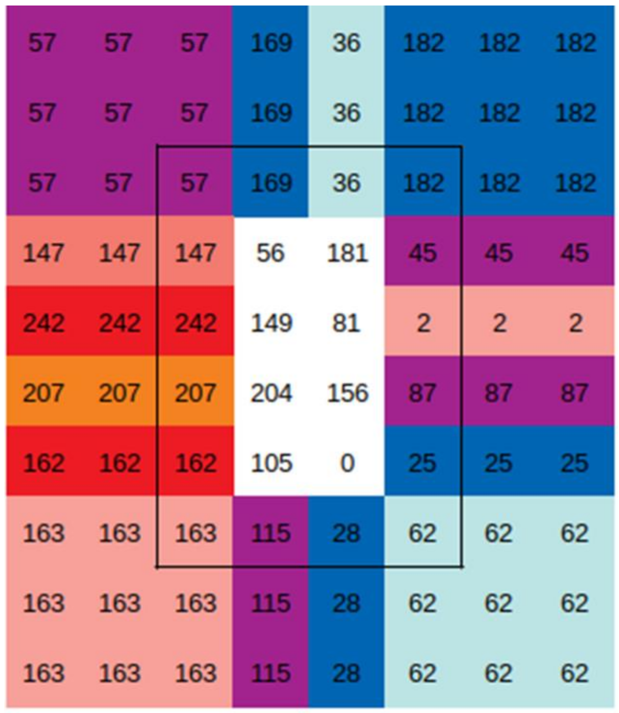


Figure 1: An example MAC operation

57	169	36	182
147	56	181	45
242	149	81	2
207	204	156	87
162	105	0	25
163	115	28	62

Figure 2: Input Matrix



57	57	57	169	36	182	182	182
57	57	57	169	36	182	182	182
57	57	57	169	36	182	182	182
147	147	147	56	181	45	45	45
242	242	242	149	81	2	2	2
207	207	207	204	156	87	87	87
162	162	162	105	0	25	25	25
163	163	163	115	28	62	62	62
163	163	163	115	28	62	62	62
163	163	163	115	28	62	62	62

Figure 3: Edge-extends on input matrix for kernel size 5x5

Normalization:

There is also a little detail such that we normally just flip the kernel matrix both vertically, and horizontally but we will ignore this as we assume that kernel matrices are all symmetric. But we should still normalize the kernel so that sum of the elements of the kernel are equal to one to ensure the output image has the same brightness as the input image. For normalization, we should find the sum of elements in the kernel, and after getting the output image, divide the each element with this sum (integer division is OK). Also, another paranthesis here is that kernel normalization should actually occur in kernel matrix but not the output image.Yet, we do it like that because we use int** matrices for both input image and kernel.

Implementation Details:

- First, you will implement the sequential version of 2D Convolution, then copy it into another file and parallelize it with OpenMP API.
    - 5 functions will be implemented
        - normalize_output(int **img,int normalize_amount, int num_rows, int num_cols, int **output_img)
            - Divides the each pixel value by normalize_amount in 2d array **img to normalize pixel values. Integer division is OK.
        - 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 kernel_sum(int **kernel, int kernel_size)
            - Calculates the normalize index by simply summing up all elements in the kernel. If sum is 0, return 1. Otherwise, return the sum.
        - int pixel_operation(int **kernel, int kernel_size, int **img, int row_index, int col_index)
            - Core multiply-accumulate operation at given location of the image.
        - void convolve_image(int **kernel, int kernel_size, int **img, int num_rows, int num_cols, int **output_img)
            - Convolution method that convolves the image with given kernel. This method expected to call all other methods above.
- 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 2D Convolution.
    - Basically, 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, they are in the link provided at the beginning of the file. 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 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. You are welcome to use read_pgm_file(char * file_name, int *num_rows, int *num_columns) to read the input matrices. The program writes the matrix that is the convolution of the two input matrices to a file, in the same format as the input. You will also print parallel execution time and sequential execution time spent of the current run. So the following below, the program convolution_omp spent XX.XX ms of its run time in sequential operations, and YY.YY ms of its run time in parallel operations:

```
~$cat my_image.txt
3
5
51 53 53 3 52
89 2 3 127 167
21 44 2 78 84
~$cat kernel.txt
3
3
0 -1 0
-1 4 -1
0 -1 0
~$./convolution_omp my_image.txt kernel.txt my_output.txt
Parallel time: XX.XX ms
Sequential time: YY.YY ms
~$cat my_output.txt
3
5
-40 53 100 -223 -66
193 -181 -172 257 238
-91 107 -119 21 -77
```

Run your code with various images, and kernels(use symmetric matrices for kernel). You may get negative values, or values that does not lie between 0 and 255 in the output image for kernels having negative values. So it is a normal occasion even if you normalize the kernel.

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
```

Grading:

- convolution_seq: 20 points
- convolution_omp: 40 points
- gprof profiling results: 10 points
- Report: 30 points

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 send. When it is unzipped, it should provide the directory and files inside.

1. Your code:
1. convolution_seq.c, convolution_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. convolution_seq.c will include your sequential implementation and convolution_omp.c will include your parallel implementation.
2. A compile script that can compile your code
    a. Name this script as compile.sh
    b. It will produce 2 executables, convolution_seq and convolution_omp
2. Profiling results
- 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.
- Submit to the respective assignment on **Moodle**.
- **Zip File name:** yourname_lastname_p3.zip (Without this name, will not be evaluated).
- **No Late Submission Allowed!**