# Assignment 6 : Image Processing

## 210010020 & 210010006

## 1. Transformations:

### a) Horizontal Blur:

i. Horizontal blur image transformation is a technique commonly used in image processing to reduce noise and enhance visual aesthetics. It works by averaging the color values of neighboring pixels along horizontal lines in an image. This averaging process smooths out abrupt changes in color intensity, effectively blurring the image horizontally.

ii. For each pixel, the code calculates the average color values of the pixel itself and its 10 adjacent pixels on both sides horizontally. These color values include the red, green, and blue components.

iii. After computing the average color values, they are assigned to the corresponding pixel in the output image. This process effectively smoothes out abrupt changes in color intensity along horizontal lines, resulting in a blurred appearance. Overall, the transformation aims to reduce high-frequency details in the image, creating a smoother and more aesthetically pleasing visual effect.

### b) RGB To Grayscale:

i. Grayscale images represent each pixel with a single intensity value, typically ranging from 0 (black) to 255 (white), with shades of gray in between. This transformation is achieved by calculating the average of the red, green, and blue color channels for each pixel in the input image and assigning this average value to all three channels in the output image.

ii. The **rgb_to_gray** function calculates the grayscale intensity value using a simple formula that takes the average of the red, green, and blue color values: **(r + g + b) / 3**.

iii. The **convert_to_grayscale** function applies this grayscale conversion to every pixel in the input image, replacing the original color values with the computed grayscale intensity value. This process effectively removes color information from the image, resulting in a grayscale representation where variations in intensity represent changes in brightness rather than color.

2. method to prove in each case that the pixels were received as sent,in the sent order.

   We have implemented a row wise transformation for the images, printing the row number in transformation 1 and transformation 2 should give a sequential increase in the row numbers, without skipping any row. Also the the row numbers should match.
   This helps us to verify that the pixels in the row are received in the same order in which they are sent to the function. This is the proof of correctness for the working of code.

3. Study the run-time and speed-up of each of the approaches and discuss.

| Method | 5mb(ms) | 8mb(ms) | 25mb(ms) | 73mb(ms) |
|---|---|---|---|---|
| Part1 (Sequential) | 579 | 378 | 1332 | 4486 |
| Part2_1a (Threads+Atomic) | 262 | 372 | 1404 | 4481 |
| Part2_1b(Threads+Semaphores) | 242 | 345 | 1283 | 4345 |
| Part2_2 (Processes+Shared Mem) | 57 | 81 | 255 | 815 |
| Part2_3(Proceses+Pipes) | 434 | 629 | 1969 | 4686 |

i. <u>Sequential :</u> The runtime is highest for Part1 and Part2 (iii) as both are being implemented in a sequential manner, greater overhead can be observed for Part iii because of higher IPC overhead.

iii. <u>Threading</u> methods likely involve parallel processing using threads. While threads can improve processing speed, they might contend for shared resources or need synchronization, leading to overhead. As the image size increases, the overhead becomes more pronounced, resulting in slightly increased execution times, but still generally faster than sequential processing.By using atomic operation we keep checking condition in a while loop but using semaphores we are checking the condition only once, this way the overhead of threading with semaphore is less. Thus using Semaphore for synchronization improves the performance.

iv. <u>Threading+ shared Memory</u>: method utilizes processes and shared memory for parallelism. Processes generally have higher overhead than threads due to separate address spaces. However, shared memory allows processes to communicate efficiently. As the image size increases, the overhead remains relatively low, resulting in relatively stable execution times compared to other methods.

v. <u>Two Processes with Pipes</u> : method employs processes and inter-process communication (IPC) via pipes. Processes with IPC typically incur higher overhead due to context switching and data transfer between processes. As the image size increases, the overhead becomes more significant, leading to notable increases in execution times, especially compared to shared memory-based approaches.

4. Explanation & the relative ease/ difficulty for each part:
    1. Sequential: Creating two functions one for horizontal blur and other for grayscale of images and then simply calling them in sequential order, waiting until first one completes it's execution before the second one begins executing. This is the easiest to implement, by just creating two functions.

    2. Threads & Atomic: Creating two threads and a global shared variable (vector<vector<Pixel>> buffer) that acts as a buffer as in the case of producer and consumer problem. Whenever the first Transformation is gets completed on a row then the row index is pushed into the buffer, this is read by the second thread after applying T2 it then pops the row from buffer. For the first time when the buffer is empty T2 waits. The synchronization is handled by using two atomic locks for two processes. The synchronization is relatively easy to handle with locks that are used for making the operations atomic.

    3. Thread and Semaphores: Two threads are created in the same way as above but in this case the semaphore is used for their synchronization. The semaphore is initialized to zero and incremented whenever T1 completes execution of a new row and decremented in T2 whenever T2 tries to access a new row for execution. Since we are processing each row at a time the loop counter ensures that the rows are processed in sequential order. Another semaphore is used to handle the atomic operation for buffer. This is of same difficulty as using atomic locks.

    4. Process communicating through shared memory: There are two processes created by using of fork() command. Two shared memory are created in this case one for passing the entire image matrix and other two pass the semaphore. With a single semaphore we are achieving the synchronization. Whenever a thread completes it's execution on a row and increments the semaphore . Then both the semaphore and the row are passed into the shared memory allocated respectively.T2 reads the rows from the shared memory , decrements the semaphore and then works on that row. The loop counter ensures that the rows are received in the same order as they are sent, thus there is no need to maintain another semaphore/atomic lock. This approach is most difficult to implement and also tedious for debugging as we need to handle two shared memory.

    5. Process and communication through Pipes: There are two processes created by the use of fork() command. These processes communicate through the pipes. Intially we want

the child process T1 to complete execution on the entire matrix, after completing the HorizontalblurImage transformation it sends the modified matrix to the parent. The pipe waits if there is nothing to read from it for the process T2. So even if the parent goes scheduled earlier it won't throw an error because of empty pipe but the pipe commands call handles it. At the end we need to add wait(NULL) in parent so that parent waits for the child to complete it's execution thus avoiding the case of a zombie process.  Though this approach is easier to implement as we just need to pass the entire matrix into the pipe without any need to implement synchronization.