CS314: Lab Report
Assignment 6 - Image Processing and Synchronization Principles

B Siddharth Prabhu
200010003@iitdh.ac.in

Devdatt N
200010012@iitdh.ac.in

19 February 2023

1 Part I - Simple Image Processing Application

Task: Write a simple image processing application in C/C++.

- The input to the application should be a ppm image file. There are many free utilities to convert from jpeg images to ppm ones.
- Your application must read this file and store the pixel information in a matrix.
- Then, it must perform two transformations to the image, one after the other. Choose some simple transformations such as "RGB to grayscale", "edge detection", "image blur", etc. Let us call the two transformations T1 and T2.
- Write the resultant pixel matrix to a new ppm file.
- Usage: `./a.out <path-to-original-image> <path-to-transformed-image>`

1.1 Description of Image Transformations Used

(1) RGB to Grayscale

There are three different approaches to convert a given colour image to a grayscale images. The approaches, along with the formulae used by them to compute the grayscale value of each pixel having (R, G, B) values given, are as follows:

- **Lightness Method**: Grayscale value = $(max(R, G, B) + min(R, G, B))/2$
- **Average Method**: Grayscale value = $(R + G + B)/3$
- **Luminosity Method**: Grayscale value = $0.21 \times R + 0.72 \times G + 0.07 \times B$

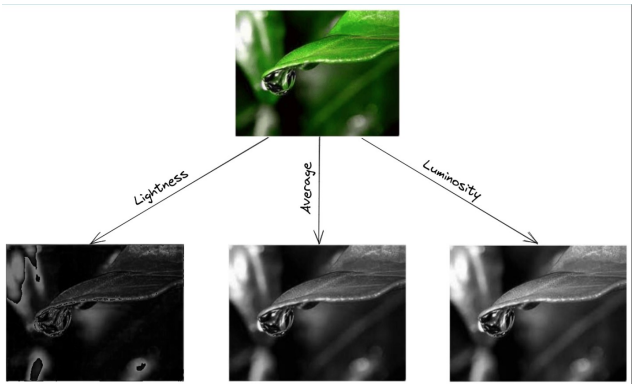


Figure 1: Methods of RGB to Grayscale Conversion

In our implementation, we used the **Luminosity Method**, in which the green channel has the largest coefficient because the human eye is most sensitive to green light. This method produces grayscale images that are visually pleasing to the human eye. This is often considered the most accurate and produces grayscale images that are perceived as the closest match to the original color image. (The Lightness method tends to produce grayscale images with higher contrast, while the Average method can result in images that appear more flat or dull.)

(2) Image Blur

In this transformation, we used a technique called **Convolution** to blur images using kernels. Convolution is a common image-processing technique that changes the value of a pixel according to the values of its surrounding pixels. Many common image filters, such as blurring, detecting edges, and sharpening, derive from convolution. **Kernels** are 1D or 2D grids of numbers that indicate the influence of a pixel's neighbors on its final value. To calculate the value of each transformed pixel, add the products of each surrounding pixel value with the corresponding kernel value. During a convolution operation, the kernel passes over every pixel in the image, repeating this procedure, and then applies the effect to the entire image.
Here, we used the kernel given below, and since just 1 run of it doesn't blur larger image to a noticeable extent, we made it run 50 times. This number could also be adjusted based on the image size/resolution. (Note: We're not using this transformation in the later parts of the assignment; the other transformations should suffice.)

$$K = \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

A more complex blurring kernel varies the influence of pixels according to their distance from the center of the kernel, and yields a smoother blurring effect. (Note that the sum of the values in the convolution kernel above is 1, i.e., the kernel is normalized. If the sum of the values is greater than 1, the resulting image is brighter than the source. If the sum is less than 1, the resulting image is darker than the source.)

(3) Edge Detection

For edge detection, we made use of the Sobel operator. It performs a 2-D spatial gradient measurement on an image, and hence emphasizes regions of high spatial frequency (that correspond to edges). Typically, it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The operator consists of a pair of 3×3 kernels, which are given by the following:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \textit{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & -1 \end{bmatrix}$$

These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The results obtained by each of the kernels are combined to obtain the overall result of the operator.

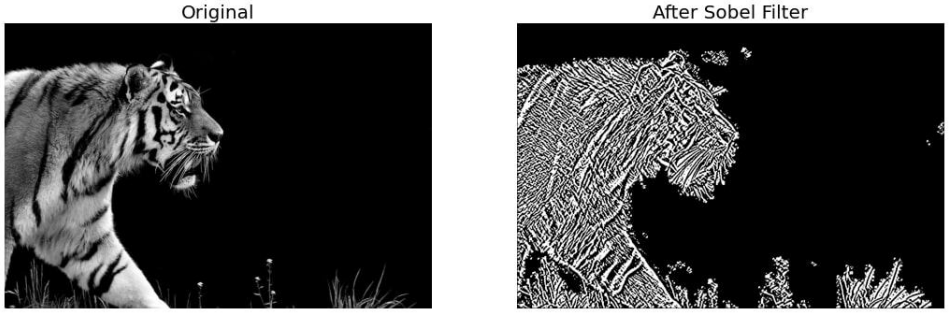


Figure 2: Effect of Sobel Operator

2 Part II - Implementing Synchronization

Task: Now suppose you have a processor with two cores. You want your application to finish faster. You can do this by having the file read and T1 done on the first core, passing the transformed pixels to the other core, where T2 is performed on them, and then written to the output image file. Do this in the following ways:

1. T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself.
   (a) Synchronization using atomic operations
   (b) Synchronization using semaphores
2. T1 and T2 are performed by 2 different processes that communicate via shared memory. Synchronization using semaphores. (Single source file)
3. T1 and T2 are performed by 2 different processes that communicate via pipes. (Single source file)

2.1 Inter-Thread Communication

2.1.1 Synchronization using Atomic Operations

In this part, as coded in `part2_1a.cpp` , two transformations are performed by two threads of the same process, and communicate through the process' address space itself. Some things to be noted about the code are as follows:

- The program uses the `atomic_flag` type to synchronize access to shared variables in multi-threaded code.
- The atomic flag `molecule` is used to ensure that only one thread modifies the pixels of the input image at a time.
- The global variable `position` is an integer array that are initialized to check if the bit is correct.

2.1.2 Synchronization using Semaphores

In this part, as coded in `part2_1b.cpp` , two transformations are performed by two threads of the same process, and communicate through the process' address space itself.

- The sem_t variable `sema` is used as the semaphore. This semaphore is shared between the threads and is used to synchronize access to the position and image variables, which are shared resources between the threads.
- The semaphore ensures that the shared pixel array is accessed by only one thread at a time, and this is done by using the `sem_wait()` and `sem_post()` functions to control access to the shared resource.

2.2 Inter-Process Communication: Shared Memory & Semaphores

In this part, the process uses shared shared memory to pass the image data between the two threads T1 and T3, and a semaphore to synchronize access to the shared memory. Before accessing the shared memory, the process waits on the semaphore. After updating the shared memory, the process releases the semaphore. (More details on runtime are explained later.)

2.3 Inter-Process Communication: Pipes

In this part, the process uses pipes to pass the image data between threads. One thread writes to the pipe, and another reads from the pipe. (More details on runtime are explained later.)

3 Proof of Correctness

When running the two threads, the goal is to evaluate T_1A *and* T_3B without any common elements between A and B, since T_1 and T_3 should not be operating on the same pixels at any given time. If permitted, this may lead to issues where one thread hasn't finished its operations on a pixel, but the other thread attempts to operate on it. To ensure that the threads don't allow such a thing to happen, an array `position` was declared. In each of the parts, it serves its purpose in slightly different ways, such as:

- In `part2_1a.cpp` and `part2_1b.cpp` , it is a global variable, which is directly accessible to all threads.
- In `part2_2.cpp` , it is a part of the shared memory.
- In `part2_3.cpp` , it is pushed to a separate pipe.

4 Analysis and Results

Below is a tabulation of our observations of the time elapsed when running the code.

Image Size ⟶	`r7.ppm` (496 kB)	`r9.ppm` (1203 kB)	`r11.ppm` (2269 kB)
`part1.cpp` (Sequential)	16030	37727	75441
`part2_1a.cpp` (Atomic Operations)	23699	56677	106174
`part2_1b.cpp` (Semaphores)	23965	54443	108535
`part2_2.cpp` (Shared Memory)	84299	203540	422580
`part2_3.cpp` (Pipes)	-	-	-

Table 1: Time taken for each approach with varying file sizes (in microseconds)

Points to be noted in our analysis are as follows:

- The sequential approach should have taken more time than the other approaches, since the other approaches are trying to implement concurrency-based approaches, that, in theory, should reduce the time taken. However, this is not what is observed. A plausible explanation for this is that the critical section code is quite small, to the point where the inter-process communication becomes an increased overhead.
- From our table, we can observe that the time taken by Shared Memory and Pipes are higher than that of the sequential approach, since they involve reading from and writing to values in the memory.
- Screenshots for the transformations are included on the next page.

5 Relative Difficulties of Implementation

- The sequential approach was the easiest to implement; it's the most straightforward task, and doesn't use any synchronization primitives.
- The atomic operations and semaphores were also relatively easy to implement since they involved threads directly.
- Finally, the Shared Memory and Pipes were relatively harder to implement due to the need to check if values were properly passed between the different processes. In pipes, there is less maintenance to be done by us, as compared to shared memory, where the whole structure had to be set by us.

6 Demonstration of Transformations Used

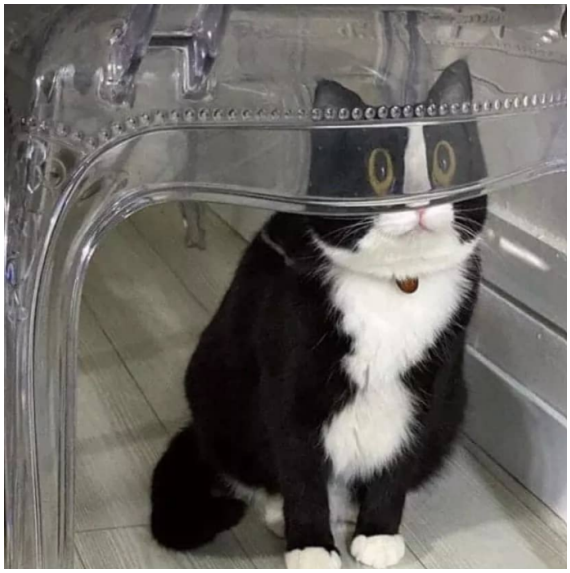
6.1 T1 : RGB to Grayscale

(a) Original image `r11.png`

(b) Resultant Image `result-r11.png`

Figure 3: Transformation T1 (Grayscale)

6.2 T2 : Image Blur

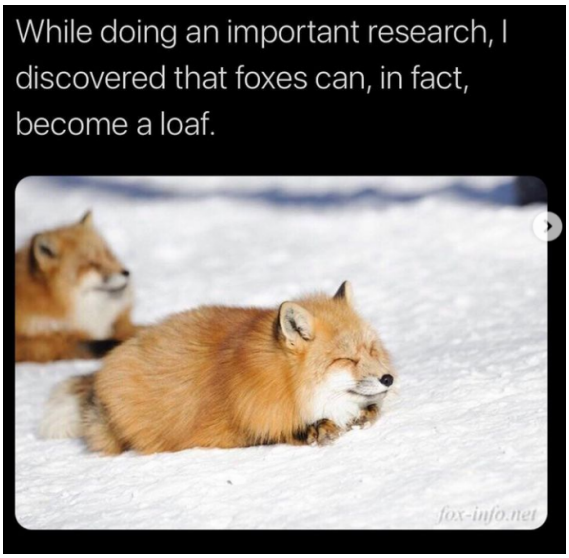


(a) Original image `r12.png`

(b) Resultant Image `result-r12.png`

Figure 4: Transformation T2 (Blur)

6.3 T3 : Edge Detection



(a) Original image `r6.png`

(b) Resultant Image `result-r6.png`

Figure 5: Transformation T3 (Edge detection)