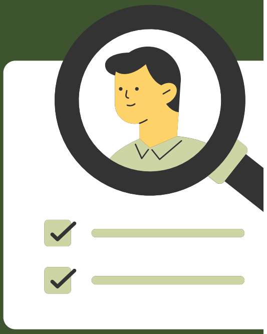
Operating System Laboratory Project

Image Processing



Overview

The goal of this project is to develop a multithreaded image processing application that can apply filters to large images efficiently. The application will divide an image into sub-matrices, process each sub-matrix in parallel using multiple threads, and then reassemble the processed sub-matrices into the final image. This approach leverages multithreading to improve the performance of image processing tasks.

Objectives

1. **Load an Image**:
   - Read an image from disk into memory. Only one copy of the image should exist in the main thread. Other threads responsible for applying filters should read values from this main thread, treating it as a shared resource.
   - Store your image as a matrix in memory.
2. **Divide the Image**:
   - Split the image into smaller, manageable sub-matrices (chunks) for easier processing.
   - Assign a specific range of the main matrix to each child thread.
3. **Apply Filters**:
   - Implement various image filters that can be applied to each sub-matrix.
   - Each thread creates its own result matrix for storing filter values for each pixel.
4. **Multithreading**:
   - Process each sub-matrix in parallel using multiple threads. The number of threads depends on the image size and should vary accordingly (at least 9 threads).
5. **Synchronization**:
   - Due to memory constraints, you cannot use a clone matrix for your results. Instead, you should directly modify the main image matrix and save your result sub-matrix in the range assigned to you initially.
   - Ensure that neighboring sub-matrices are processed correctly without interference. When applying filters, make sure that at the borders, the corresponding neighbor thread has finished calculating the filter for its sub-matrix before saving changes to the image.
6. **Reassemble and Save the Image**:

- Combine the processed sub-matrices and save the final image to disk.

Edge Detection Filter

- **Purpose**: Identify edges in the image by highlighting areas with high contrast or intensity change, which is useful for detecting boundaries and shapes within the image.

Steps:

1. RGB to Grayscale Formula: The formula for converting a color image to grayscale is typically:

$$\text{Gray} = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

Where R, G and B are the red, green, and blue color channels of a pixel, respectively.

2. You should multiply the image in both kernel matrix below:

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

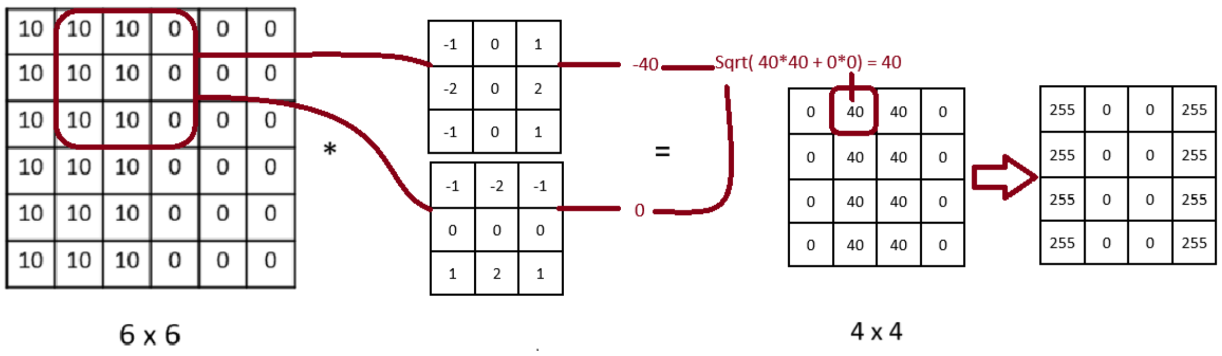
-1	-2	-1
0	0	0
1	2	1

for each window with the size of 3 * 3 in your picture you perform a **Hadamard product** with two kernels above resulting to integers.

if we call the result of X-Direction Kernel, Gx and Y-Direction Kernel, Gy you will calculate the gradient magnitude for each pixel using the following formula:

$$\sqrt{Gx^2 + Gy^2}$$

This magnitude will replace the pixel value in the corresponding location in the resulting edge-detected image.

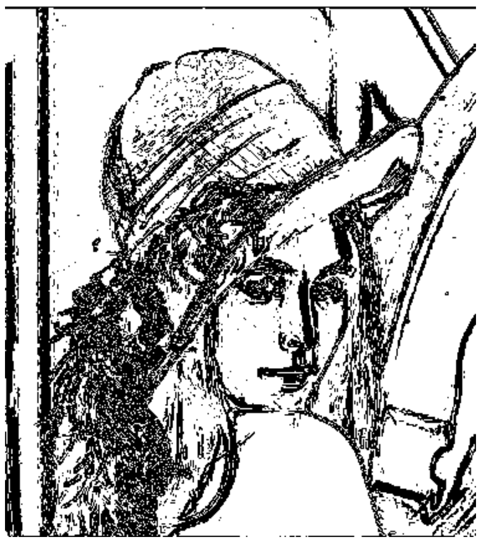


after applying the filters and obtaining the result matrix:

i. **Calculate the Mean**:
   - Find the average (mean) value of all the elements in the resulting matrix. For instance, if the mean is 20, this value will be used as a threshold.

ii. **Thresholding**:
   - Iterate over each element in the matrix:
     - If the value of an element is more than the mean, set it to 0 (black).
     - If the value is less than or equal to the mean, set it to 255 (white).

This post-processing step effectively converts the image into a binary (black and white) image based on the calculated mean, enhancing the edges and features detected by the filter.



Bonus: You can implement up to 2 additional filters for extra credit.

Helping Link:

1. [(256) SOBEL EDGE DETECTION IN DIGITAL IMAGE PROCESSING SOLVED EXAMPLE - YouTube]

TA: Soroush Eskandari
Good Luck to you all