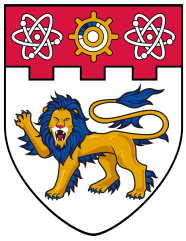


NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

*Nanyang Technological University*

*College of Computing and Data Science*

---

# Assignment I: Histogram Equalization

---


***Author:***

| | |
|---|---|
| Deng Jie | G2403273K |
| Liu Zhimeng | G2406060D |
| Wen Beichen | G2403469J |
| Yan Xinyu | G2405856A |
| Zhao Yiyang | G2402450B |

***Supervisor:***
Lu Shijian


An Assignment Submitted for NTU's Course:

**[AI6121] Computer Vision**

September 17, 2024

# Contents

# 1 Introduction of Histogram Equalization

Image enhancement can be considered as one of the fundamental processes in image analysis. The goal of contrast enhancement is to improve the quality of an image to become more suitable for a particular application. Till today, numerous image enhancement methods have been proposed for various applications, and efforts have been directed to increase the quality of the enhancement results further and minimize the computational complexity and memory usage [1].

Histogram equalization(HE) is a method in image processing of contrast adjustment using the image's histogram [2]. This method is usually used to enhance the global contrast of the image. Through this adjustment, the intensities can be better distributed evenly on the histogram, utilizing the full range of intensities. This allows for areas of lower local contrast to gain a higher contrast. Next, we will deduce the formula of the HE algorithm in continuous and discrete cases step by step.

The source code of the project has also been uploaded to GitHub, and readers can access it via GitHub. Meanwhile, the complete code for this assignment is also included in Appendix A.

## 1.1 Continuous Case

In the case of contiguous, histogram equalization is applied to images with pixel values of a continuous range (e.g., floating-point number) and is typically used in theoretical models. From the perspective of mathematical derivation, the goal is to achieve a uniform probability density function (PDF) distribution, as shown below:

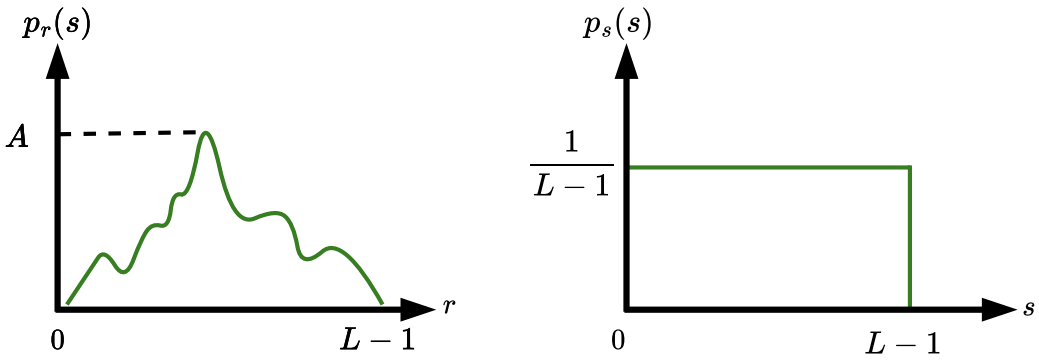



Figure 1: Histograms of an Image Before and After Equalization

Assuming that an image to be processed is a grayscale image where $r$ represents its original grayscale value and $s$ represents its grayscale value after processing, with a value range $[0, L-1]$. Transfer function $T$ shown below converts $r$ to $s$:

$$s = T(r), 0 \leq s \leq L-1 \tag{1}$$

$p_r(r)$ is the probability density function (PDF) of the image before equalization. $p_s(s)$ is the PDF of the image after performing equalization and is an equalized histogram that is uniformly distributed among all the possible values. Through the geometric meaning of the HE algorithm on the histogram, we can construct the following formula:

$$s = T(r) = (L-1)\int_0^r p_r(w)\,dw \tag{2}$$

Now, the differentiation of s with respect to $r$ is:

$$\begin{aligned}\frac{ds}{dr} &= \frac{d}{dr}T(r) \\ &= \frac{d}{dr}\,(L-1)\int_0^r p_r(w)\,dw \\ &= (L-1)p_r(r)\end{aligned} \tag{3}$$

Then, substitute formula(2) into formula(3). Relation between $p_r(r)$ and $p_s(s)$ can be achieved as:

$$\begin{aligned}p_s(s) &= p_r(r)\left|\frac{d_s}{d_r}\right| \\ &= p_r(r)\left|\frac{1}{(L-1)p_r(r)}\right| \\ &= \frac{1}{(L-1)}; (0 \le L-1, 0 \le s \le L-1)\end{aligned} \tag{4}$$

Therefore, the new variable $p_s(s)$ is uniformly distributed. We can say that equalization of the histogram can be achieved by the above derivation.

## 1.2 Discrete Case

The above derivation assumes that the values of pixels are continuous. In actual processing, the values of pixels are generally discrete (limited pixel values, such as grayscale values from 0 to 255). In this case, histogram equalization is typically applied to a pixel grayscale image with integer values.

Suppose we have a grayscale image with $M$ pixels in a row and $N$ pixels in a column. Thus, the total number of pixels, $S$, can be calculated using the following formula:

$$S = M \times N \tag{5}$$

Assuming that each pixel in the image corresponds to a gray value, with a range $[0, L-1]$. We divide grayscale into $k$ levels according to the intensity and assume that there are $n_j$ pixels in gray level $j$. Add pixels in all levels together according to the following formula:

$$S = \sum_{j=0}^{k} n_j \tag{6}$$

Then, we denote the probability of occurrence of pixels in $j$ level by $p_j$, so the following formula will construct the probability density function:

$$p_j = \frac{n_j}{S} \tag{7}$$

In the discrete case, we use a probability histogram instead of a probability density function and summation instead of an integral operation, respectively. Substitute formula(5), formula(6) and formula(7) into formula(2), so we construct the following formula to adjust the value of each pixel:

$$\begin{aligned}s_k &= T(r_k) \\ &= (L-1)\sum_{j=0}^{k} p_r(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j\end{aligned} \tag{8}$$

Because grayscale values are discrete, such that each pixel value is an integer. After applying the formula(8) for processing, the resulting value may be a rational number. Consequently, it is necessary to round $s_k$ to the nearest integer.

## 1.3 Example

Table 1 below provides an example of histogram equalization, where $r_k$ ranges from 0 to 7. Meanwhile, $L = 8$ in this case, indicating that there are 8 bars in total representing the pixel values in a bar chart.

Table 1: Example of $r_k$ Ranging from 0 to 7

| $r_k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $n_k$ | 790 | 1023 | 850 | 656 | 329 | 245 | 122 | 81 |
| $p_k$ | 0.19 | 0.25 | 0.21 | 0.16 | 0.08 | 0.06 | 0.03 | 0.02 |
| $s_k$ | 1.33 | 3.08 | 4.55 | 5.67 | 6.23 | 6.65 | 6.68 | 7.00 |
| Rounded $s_k$ | 1 | 3 | 5 | 6 | 6 | 7 | 7 | 7 |

By comparing the original histogram and the one after equalization in Figure 2, we can intuitively feel that the second figure has a better value distribution.

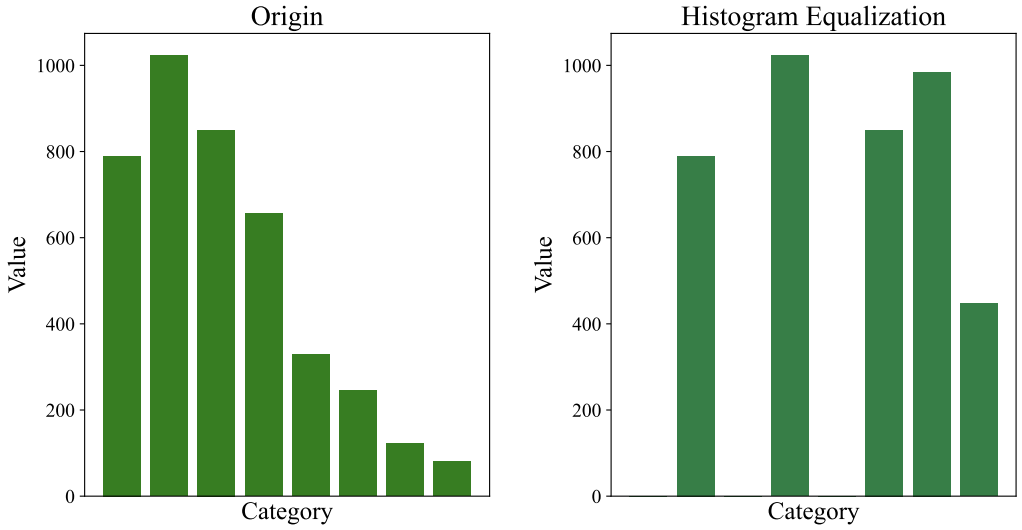



Figure 2: Original Histogram and Equalized Histogram

# 2 Task 1: HE Algorithm Implement

## 2.1 Histogram Equalization Function

After we deduce the HE algorithm using mathematics, the next step is to ascertain the accuracy of the formulas above. The pseudo-code of histogram equalization is shown as follows.

**Algorithm 1** Cumulative Histogram

**Input:** Histogram $H$
**Output:** Cumulative Distribution Function $CDF$
Initialize $CDF$ with zeros
**for** $i = 1$ **to** 255 **do**
    $CDF[i] \leftarrow CDF[i-1] + H[i]$
**end for**
**return** $CDF$

**Algorithm 2** Make Histogram

**Input:** Image $I$ with pixel values $I(x,y)$
**Output:** Histogram $H$
Initialize histogram $H$ with zeros
Separate the image into its three channels: hue ($h$), saturation ($s$), and value ($v$)
**for** row in $I.rows$ **do**
    **for** col in $I.cols$ **do**
        $V \leftarrow CDF[v][[row, col]]$
    **end for**
**end for**
Merge $h$, $s$, and $V$ channels to form $H$
**return** $H$

Code 1 presents the core implementation of histogram equalization.

**Note: Cumulative_distribution[0][i], Cumulative_distribution[1][i], Cumulative _distribution[2][i] represents three layers in a color image. If the image is grayscale, there is only Cumulative_distribution[0][i].**

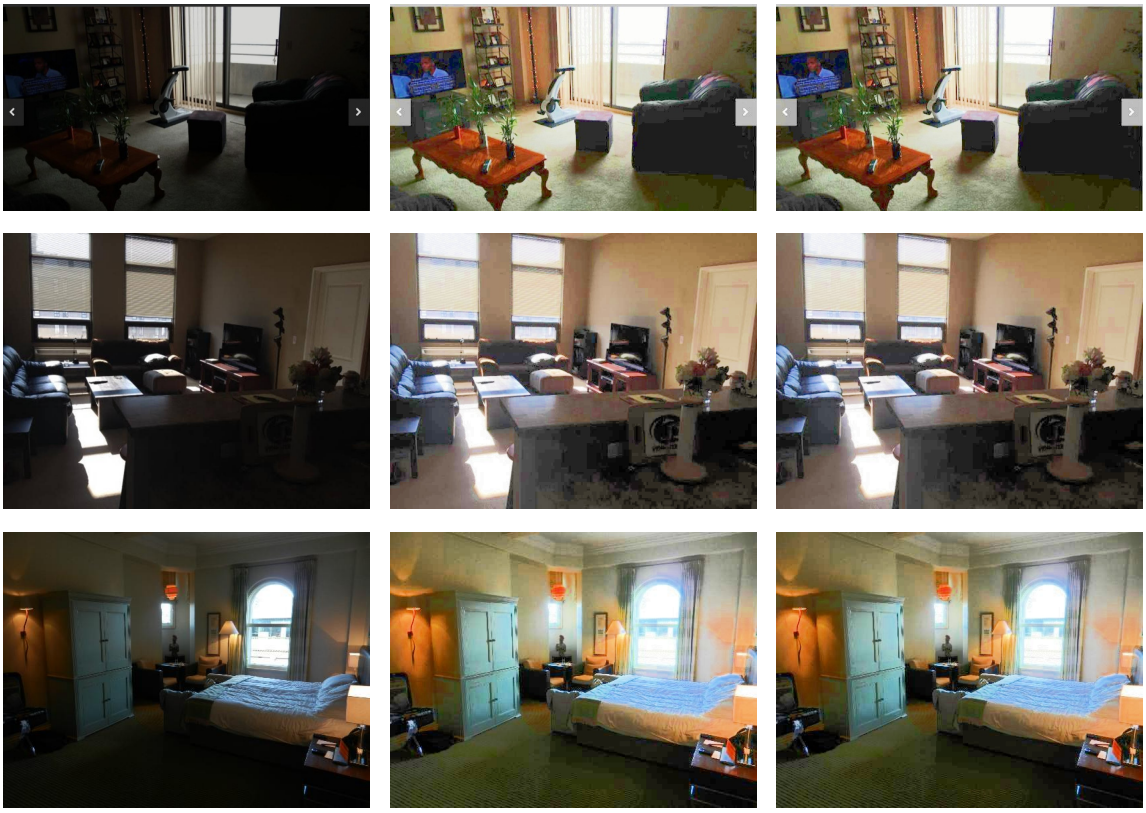
Code 1: Example Code for Histogram Equalization

```
height = self.OriginalImg.shape[0]
width = self.OriginalImg.shape[1]
N = height * width
for i in range(256):
    if i == 0:
        self.Cumulative_distribution[0][i] = self.Histogram[0][i] / N
        self.Cumulative_distribution[1][i] = self.Histogram[1][i] / N
        self.Cumulative_distribution[2][i] = self.Histogram[2][i] / N
    elif i == 255:
        self.Cumulative_distribution[0][i] = 1.0
        self.Cumulative_distribution[1][i] = 1.0
```

```
12            self.Cumulative_distribution[2][i] = 1.0
13        else:
14            self.Cumulative_distribution[0][i] = self.Histogram[0][i] / N
                  + self.Cumulative_distribution[0][i - 1]
15            self.Cumulative_distribution[1][i] = self.Histogram[1][i] / N
                  + self.Cumulative_distribution[1][i - 1]
16            self.Cumulative_distribution[2][i] = self.Histogram[2][i] / N
                  + self.Cumulative_distribution[2][i - 1]
17
18    self.Cumulative_distribution = self.Cumulative_distribution * 255
```

The Python interface of OpenCV is accessed through `cv2` module, allowing Python developers to leverage OpenCVs capabilities easily. So we could check the correctness using the function `cv2.equalizeHist()` . To reduce the experimental error, we choose three color images in the samples package and then test the implemented HE algorithm on the HSV channel and function `cv2.equalizeHist()` respectively.

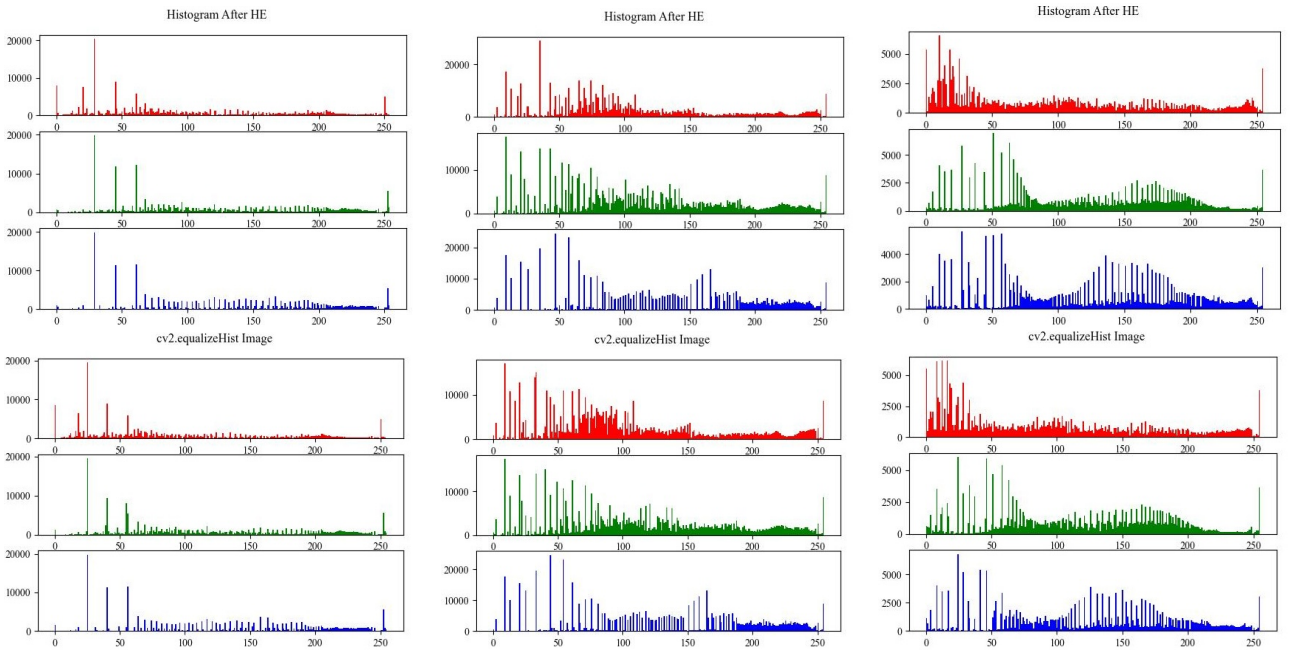
The following picture group is the result of the experiment. The first column contains the original pictures, the second column contains the pictures obtained after calculation using the above formulas, and the last column contains the picture obtained by calling the OpenCV library's function.



(a) Original Image (b) Our HE (Using HSV Model) (c) cv2.equalizeHist()

Figure 3: Images Comparison Between Implemented HSV Function and OpenCV's HE Function

The HE algorithm implemented in this assignment closely resembles the HE function in `cv2` library. To draw a more intuitive comparison, we analyze their histograms. The following images display the histograms of pixel values for the three samples: the top images are based on our deduced formulas. In contrast, the bottom images are generated using the OpenCV library's function.




(a) Image 1 (b) Image 2 (c) Image 3

Figure 4: Histogram Comparison Between Implemented HSV Function and OpenCV's HE Function

## 2.2 Color Models

### 2.2.1 RGB

When it comes to a color image, it's natural to apply Histogram Equalization separately to the Red, Green, and Blue channels of the RGB color values of the image. The image is then reconstructed from the processed channels.

Code 2: Histogram Equalization of RGB Image

```
for row in range(height):
    for col in range(width):
        r = self.OriginalImg[row][col][0]
        self.Histogram[0][r] += 1
        g = self.OriginalImg[row][col][1]
        self.Histogram[1][g] += 1
        b = self.OriginalImg[row][col][2]
        self.Histogram[2][b] += 1
for row in range(height):
    for col in range(width):
        Newvalue = self.Cumulative_distribution[0][self.NewImg[row][
            col][0]]
```

```
12          self.NewImg[row][col][0] = Newvalue
13          Newvalue = self.Cumulative_distribution[1][self.NewImg[row][
                col][1]]
14          self.NewImg[row][col][1] = Newvalue
15          Newvalue = self.Cumulative_distribution[2][self.NewImg[row][
                col][2]]
16          self.NewImg[row][col][2] = Newvalue
```

### 2.2.2 Weighted RGB

However, applying the same method to the Red, Green, and Blue components of an RGB image separately will result in dramatic changes in the image's color balance, as the relative distributions of the color channels change due to the algorithm being applied. The histograms of the R, G, and B channels are completely different, and they don't contribute equally to the image. So, we may need to give each channel a different weight.

When we convert an RGB image to grayscale, we use the linear combination of RGB channels. This can be a form of weighted sum of RGB channels, indicating the different contributions of RGB channels to an image.

$$I = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{9}$$

Weighted RGB histogram utilizes the grayscale image histogram as a weighted histogram of the RGB image, then applies the processed histogram to each channel separately.

Code 3: Histogram Equalization of Weighted RGB Image

```
1  for row in range(height):
2      for col in range(width):
3          I = GrayImg[row][col]
4          self.Histogram[0][I] += 1
5  for row in range(height):
6      for col in range(width):
7          Newvalue = self.Cumulative_distribution[0][self.NewImg[row][
               col][0]]
8          self.NewImg[row][col][0] = Newvalue
9          Newvalue = self.Cumulative_distribution[0][self.NewImg[row][
               col][1]]
10         self.NewImg[row][col][1] = Newvalue
11         Newvalue = self.Cumulative_distribution[0][self.NewImg[row][
               col][2]]
12         self.NewImg[row][col][2] = Newvalue
```

### 2.2.3 YCbCr

YCbCr or YUV originated in the signal television era. At this stage, engineers needed to find a color representation method to make the transmitted color television signal compatible with black-

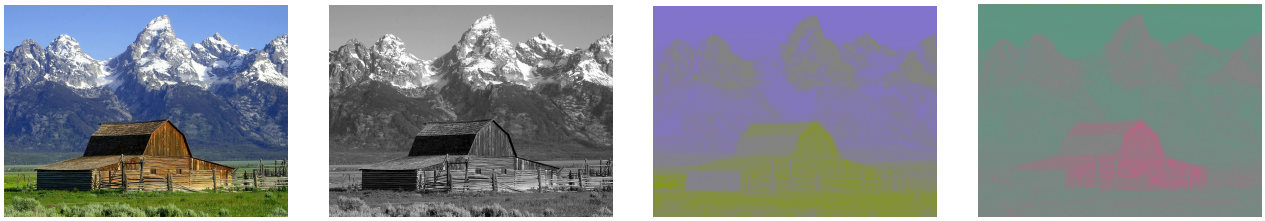
and-white television [3]. Ordinary RGB cannot do this, so engineers use different proportions of RGB colors to divide the signal into Y (Luma), U (Cb), and V (Cr).

$$Y = 0.299R + 0.587G + 0.114B \tag{10}$$

$$C_b = -0.1687R - 0.3313G + 0.5B + 128 \tag{11}$$

$$C_r = 0.5R - 0.4187G - 0.0813B + 128 \tag{12}$$

This calculation method perfectly achieves the expected effect. A single Y channel can display black-and-white images, and combining three channels can also synthesize a complete color image.



(a) Original (b) Y Channel (c) Cb Channel (d) Cr Channel

Figure 5: YCbCr Channel [4]

According to the above formula, we reorganize the original RGB image for this HE task into three color channels: Y, Cb, and Cr. Among them, the Y channel represents Luma alone, so only the Y channel needs to be adjusted by HE alone to adjust the brightness of the picture evenly, which will not affect the color like the RGB channel because the other two channels represent the color do not need to be adjusted by HE. The result is the best one among multi-channel HE.

In the code implementation, the Channel attribute in the class is first determined. If it belongs to YCbCr, the subsequent code will be executed. We use the `cv2.cvtColor` method in the OpenCV library to convert the image channels, where `cv2.COLOR_BGR2YCrCb` means converting the image from the BGR color space to the YCrCb color space. The image is then divided into three separate channels using the split method because we only need to perform HE on the Y channel to change the overall brightness of the image. Then, use the previously calculated `self.Cumulative_distribution` to perform HE processing on the Y channel and merge the new Y channel after completion. Finally, it is converted to RGB channels and outputs a standard image.

Code 4: Histogram Equalization of YCbCr Image

```
elif self.channel == "YCbCr":
    ycrcb_image = cv2.cvtColor(self.OriginalImg, cv2.COLOR_BGR2YCrCb)
    y, cr, cb = cv2.split(ycrcb_image)
    for row in range(height):
        for col in range(width):
            y_new_value = self.Cumulative_distribution[0][y[row, col
                ]]
            y[row, col] = y_new_value
    ycrcb_image_eq = cv2.merge([y, cr, cb])
    self.NewImg = cv2.cvtColor(ycrcb_image_eq, cv2.COLOR_YCrCb2BGR)
```

### 2.2.4 HSI and HSV

The HSI and HSV color spaces are entirely different from RGB. RGB's color space can be represented as a cube, in which each combination represents a coordinate. HSI and HSV represent the entire color through a pointer pointing to a point in two combined cones.

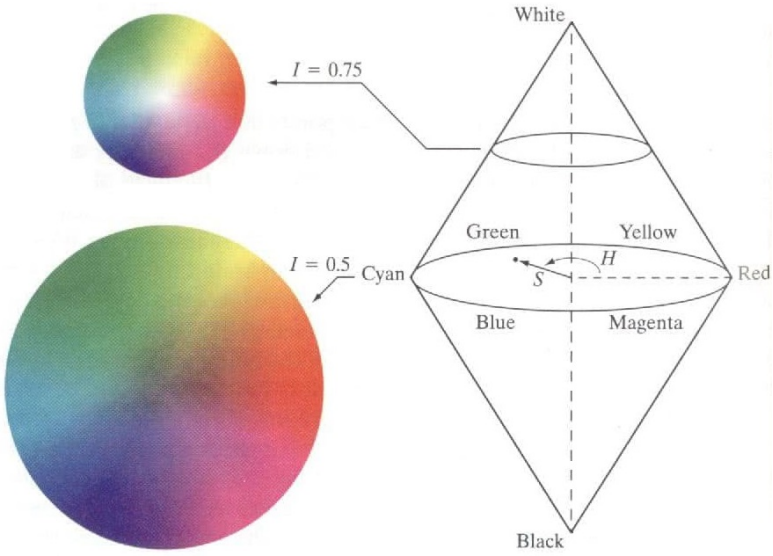



Figure 6: HSI Color Model [5].

HSI and HSV can be divided into three channels. Their first two channels are almost the same. H stands for Hue, and the color can be represented by angle, such as red at 0 degrees, yellow at 60 degrees, and green at 120 degrees. S stands for Saturation, where the bigger the value of S, the brighter the color, and the smaller the color, the closer to grey. The last channel is the main difference between the two color spaces, and this channel is also the channel we need to perform HE. The last channel, I (intensity) and V (value), both represent the brightness of the image. However, I (intensity) represents the average value of the three channels (the average of the RGB values), which more directly represents the total amount of light. However, V (value) is the value of the brightest channel in color, indicating the maximum brightness value of the color [6].

The conversion between RGB and HSI is as follows:

$$\theta = \begin{cases} \cos^{-1}\left(\frac{\frac{1}{2}[(R-G)+(R-B)]}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right) & \text{if } B \leq G, \\ 360^\circ - \cos^{-1}\left(\frac{\frac{1}{2}[(R-G)+(R-B)]}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right) & \text{if } B > G \end{cases} \tag{13}$$

$$S = 1 - \frac{3}{R+G+B}\min(R,G,B) \tag{14}$$

$$I = \frac{1}{3}(R+G+B) \tag{15}$$

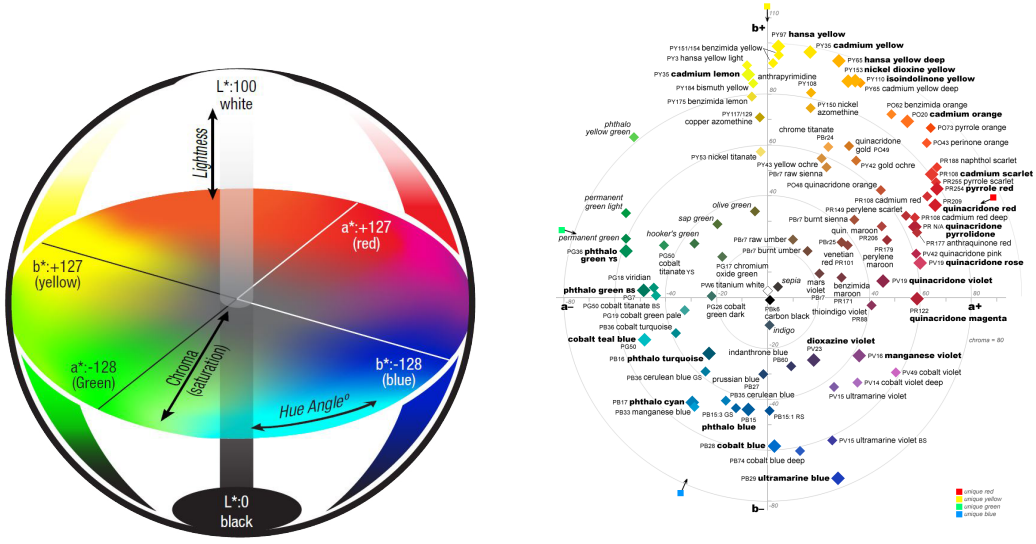
The code implementation is nearly the same as the YCbCr Channel. Initial, check whether the channel attribution corresponds. Then, Use the `cv2.cvtColor` method in the OpenCV library to convert the image channels. Next, split the channels into three individual channels and find the channel that controls the light (I or V). Use the `self.Cumulative_distribution` to process the Histogram Equalization. Finally, merge the calculated channels and transfer them to an RGB image.

Code 5: Histogram Equalization of HLS Image

```
elif self.channel == "HLS":
    hls_image = cv2.cvtColor(self.OriginalImg, cv2.COLOR_BGR2HLS)
    h, l, s = cv2.split(hls_image)
    for row in range(height):
        for col in range(width):
            l_new_value = self.Cumulative_distribution[1][l[row, col
                ]]
            l[row, col] = l_new_value
    hls_image_eq = cv2.merge([h, l, s])
    self.NewImg = cv2.cvtColor(hls_image_eq, cv2.COLOR_HLS2BGR)
```

### 2.2.5 LAB

LAB was formally defined by the International Commission on Illumination (CIE) in 1978. Its biggest feature is that it is designed based on human visual perception. It is not affected by devices such as monitors or printers like RGB. As a device-independent model, LAB can describe a color more accurately and is therefore favored by many design software such as PhotoShop and LightRoom.




(a) 3D LAB Color Model [7].

(b) CIELAB LAB Plane [8]

Figure 7: LAB Color Model

The color gamut space of LAB can be regarded as a sphere, which is also composed of three channels. Among them, L represents the brightness value range of [0, 100], which reflects the brightness of the color and does not depend on the color itself. The higher the value, the brighter the brightness. We will perform HE operation on this channel to keep the color information unchanged and only change the brightness of the image. The channel represents the red and green component value range between [-128, 127]. When the value is smaller, the color is closer to green, and when the value is larger, the color is closer to red. The b channel represents the blue and yellow component value range, which is the same as the a channel, [-128, 127]. The smaller the value, the closer to blue; the larger the value, the closer to yellow.

The conversion from RGB to LAB color space is more complicated than the previous. Firstly, we convert the RGB values to the CIE 1931 XYZ color space. Assuming that the RGB values are normalized (i.e., scaled to the range of 0 to 1), the transformation is done using linear matrix multiplication. The linear transformation formula for sRGB color space (with D65 reference white) is:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{16}$$

Once in the XYZ space, the next step is to convert it into the CIELAB (LAB) color space. This is done using a nonlinear transformation after normalizing the XYZ values with a reference white (commonly D65, the standard daylight illuminant, where $X_n = 95.047$, $Y_n = 100$, $Z_n = 108.883$).

The nonlinear transformation is defined as:

$$f(t) = \begin{cases} t^{\frac{1}{3}} & \text{if } t > \left(\frac{6}{29}\right)^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 t + \frac{4}{29} & \text{if } t \leq \left(\frac{6}{29}\right)^3 \end{cases} \tag{17}$$

After the transformation, the LAB color can be calculated by formula 18:

$$\begin{cases} L^* = 116 \cdot f\left(\dfrac{Y}{Y_n}\right) - 16 \\ a^* = 500 \cdot \left(f\left(\dfrac{X}{X_n}\right) - f\left(\dfrac{Y}{Y_n}\right)\right) \\ b^* = 200 \cdot \left(f\left(\dfrac{Y}{Y_n}\right) - f\left(\dfrac{Z}{Z_n}\right)\right) \end{cases} \tag{18}$$

The code implementation is the same as the YCbCr Channel. Initial, check whether the channel attribution corresponds. Then, Use the `cv2.cvtColor` method in the OpenCV library to convert the image channels.

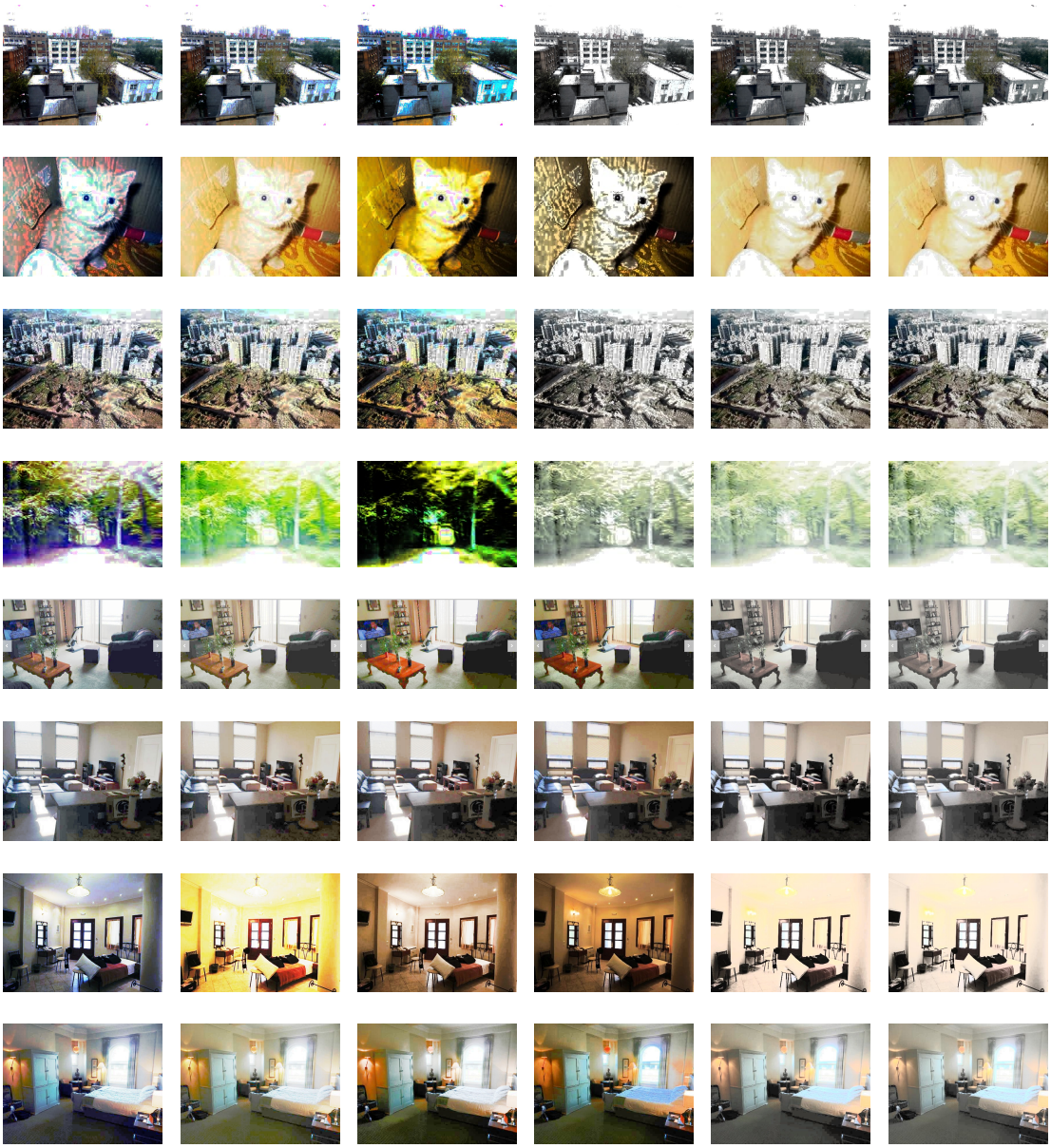
Next, split the channels into three individual channels and find the channel that controls the light(I or V). Use the `self.Cumulative_distribution` to process the Histogram Equalization. Finally, merge the calculated channels and transfer them to an RGB image.

Code 6: Histogram Equalization of LAB Image

```
elif self.channel == "LAB":
    lab_image = cv2.cvtColor(self.OriginalImg, cv2.COLOR_BGR2Lab)
    l, a, b = cv2.split(lab_image)
    for row in range(height):
        for col in range(width):
            l_new_value = self.Cumulative_distribution[0][l[row, col
                ]]
            l[row, col] = l_new_value
    lab_image_eq = cv2.merge([l, a, b])
    self.NewImg = cv2.cvtColor(lab_image_eq, cv2.COLOR_Lab2BGR)
```

## 2.3 Result

Figure 8 demonstrates the result images after histogram equalization. According to the enhanced sample images, histogram equalization can improve the contrast and brightness of an image, making objects and scenes more visible. Sample images 1-4 suffer from high brightness, while images 5-8 are too dark, making the structures and details of the images difficult to recognize. After histogram equalization, the contrast and brightness are improved, and the colors are balanced, restoring the images to good quality. In general, Weighted RGB and HSV outperform the other methods. HSI performs well in special cases, such as images 2 and 8.



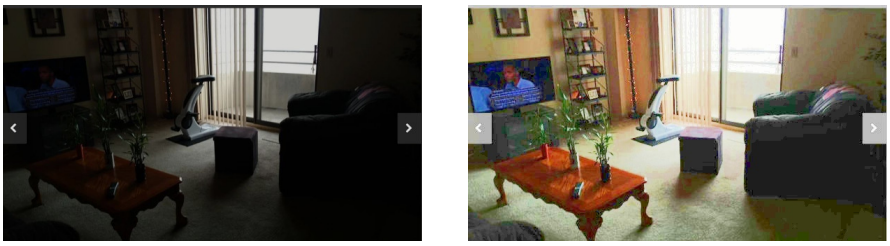
(a) RGB (b) Weighted (c) HSI (d) HSV (e) LAB (f) YCbCr

Figure 8: Resulting Pictures of All Color Models

# 3 Task 2: HE Algorithm Analysis

## 3.1 Advantage

- **Image Contrast Optimization:** Histogram equalization effectively optimizes the contrast of image data, which facilitates feature extraction, both from visual observation and algorithmic analysis. It optimizes the overall contrast of an image by making its histogram distribution more uniform. This method is particularly suitable for images with poor contrast due to less-than-ideal lighting conditions, such as images with uneven background or foreground lighting. Through histogram equalization, the distribution of pixels originally concentrated in a certain range of gray values is redistributed, resulting in a more even use of the entire gray range, thus improving the visual effect and recognizability of the image.
- **Improvement of Visual Effect:** Histogram equalization makes the darker areas of an image brighter and the brighter regions darker, which makes the details of the image more visible. For images where the gray values are unevenly distributed and concentrated in a narrow range, histogram equalization can make the gray distribution more even, thus making the visual effect of the image better. For example, for a blurred image, a lot of detailed things are not very clear, adjusting the visual effect of the image using histogram equalization can make the histogram distribution of the image more dispersed, which can cover the entire range of gray values, thus improving the visual effect of the image. In addition, the grayscale of the image after equalization is shifted to the right as a whole so that the range of gray levels of the image is widened or the grayscale is evenly distributed, thus achieving the purpose of image enhancement.



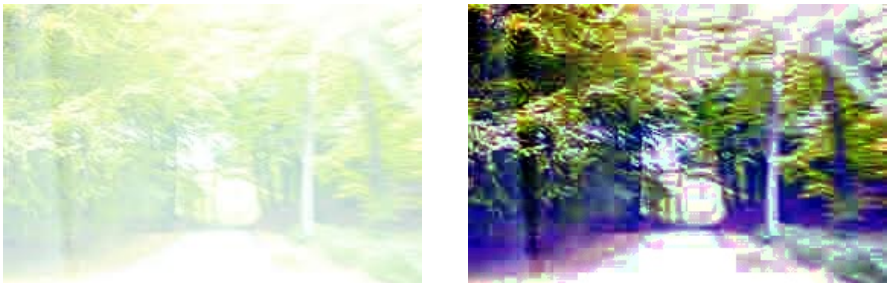
(a) Origin Image (b) HE Image

Figure 9: Images Comparison I

- **Automaticity:** The algorithm of Histogram Equalization needs no parameter to be inputted. It does not need manual feature selection or feature extraction. It also does not require adjustment of contrast or brightness, so no prior knowledge of the image is needed. The process is only based on the statistical information of intensities and is therefore fully automatic.
- **Invertibility:** It is a reasonably straightforward technique adaptive to the input image and an invertible operator. So, in theory, if the histogram equalization function is known, then the original histogram can be recovered. The calculation is not computationally intensive.

## 3.2 Disadvantage

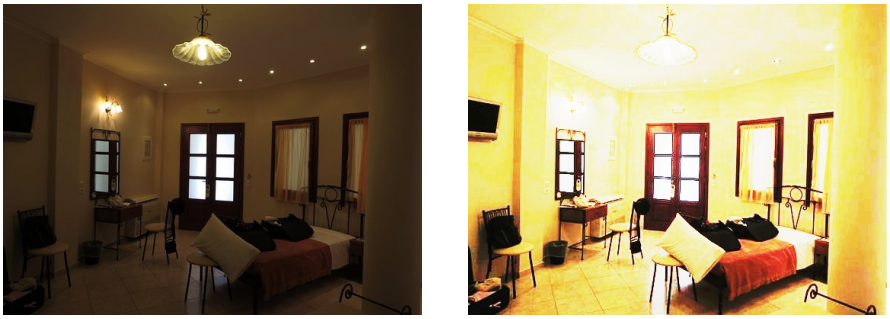
- **Local Detail Loss:** Traditional histogram equalization processes the entire image uniformly, regardless of how lighting and contrast are distributed across different regions of the image. For images with strong lighting differences or significant contrast variations, a single global processing approach cannot adapt to the complexity of the local areas. This can result in the loss of detail in the dark or bright parts of the image due to uneven contrast. For example, in an image with shaded and highlighted areas, histogram equalization may cause the shaded parts to be over-stretched, generating noise, while the bright areas may be compressed due to a lack of detail (like Figure 10).



(a) Origin Image (b) HE Image

Figure 10: Images Comparison II

- **Limited Ability of Handling Uneven Lighting:** In images with non-uniform illumination, traditional histogram equalization fails to distinguish between different regions of the image. All pixels are globally equalized, leading to poor results when dealing with uneven lighting. For instance, the darker areas of the image may be overly enhanced, while the brighter areas suffer from insufficient contrast. This results in the over-compression of highlight areas and additional noise and artifacts in the shadow regions when traditional equalization is applied to images with non-uniform lighting, leading to an unnatural overall effect. In sample image 4, Histogram Equalization is ineffective, making the picture's brightness wrong and distorted (like Figure 11).



(a) Origin Image (b) HE Image

Figure 11: Images Comparison III

- **Excessive Enhancement of Noise:** Global histogram equalization may amplify noise in the image, especially in areas with low contrast. Since histogram equalization stretches the range of pixel values, the initially vague or subtle noise is also enhanced, leading to a decline in image quality. This can result in noticeable noise in low-contrast areas of the image, making the processed image appear coarse, as seen in sample image 2 (like Figure 12).

(a) Origin Image (b) HE Image

Figure 12: Images Comparison IV

# 4 Task 3: Basic HE Algorithm Improvement

## 4.1 Contrast Limited Adaptive Histogram Equalization

CLAHE (contrast limited adaptive histogram equalization) is an enhancement technology widely used in image processing. Similarly to histogram equalization, it is especially suitable for improving image contrast. It is mainly optimized and improved based on standard adaptive histogram equalization (AHE). Through the processing of local areas, some errors and shortcomings caused by the overall histogram equalization can be prevented, such as excessive noise enhancement or loss of details in the picture.

Ordinary HE and AHE tend to magnify noise in images when enhancing contrast. It distorts, as seen in Chapter 2 when processing RGB channels. CLAHE effectively prevents this issue by setting contrast limits. He will set a maximum value for each change. If the grayscale value exceeds this value during the change, it will be evenly distributed to other places. This processing method effectively avoids the drawbacks of AHE's HE.

### 4.1.1 Algorithm

The Contrast-Limited Adaptive Histogram Equalization (CLAHE) algorithm enhances the local contrast of an image and effectively controls noise through a series of exemplary steps: first, the algorithm divides the input image into small blocks, called tiles, and each block is independently subjected to histogram equalization to enhance the local contrast. Next, the algorithm computes the histogram of each block and introduces the clip limit parameter to limit the contrast of the histogram to prevent over-enhancement of noise in uniform regions of the image.

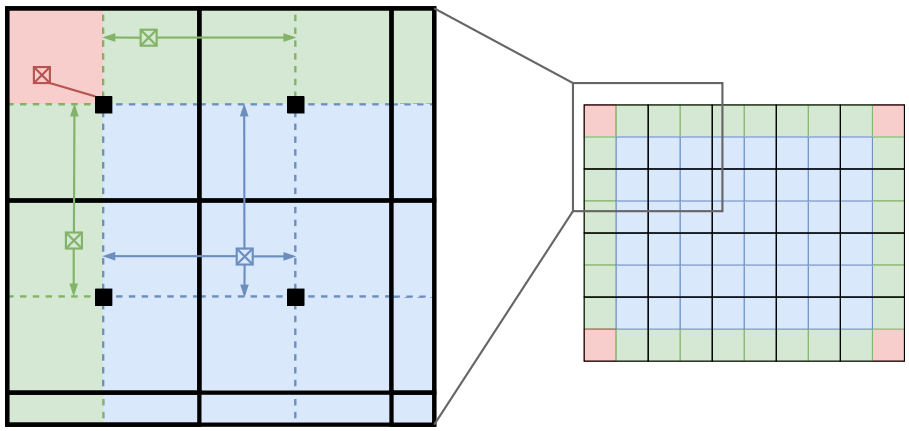


Figure 13: Sketch of the CLAHE

Then, the histogram is equalized after limiting and its cumulative distribution function (CDF) is computed to map the equalized histogram back to the pixel values; thereafter, the pixel values of each tidbit are mapped using CDF to generate the enhanced local image. To eliminate the unnatural transition of the boundary due to the independent processing of small blocks, the algorithm uses bilinear interpolation to smooth the results of the adjacent blocks; finally, all the processed blocks are recombined to form a complete image.

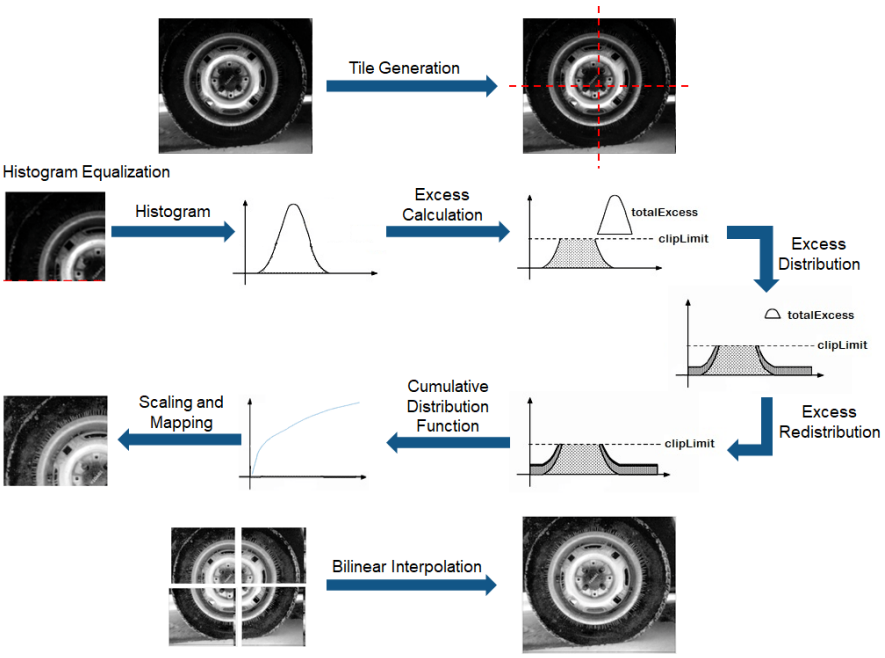



Figure 14: Flowchart of CLAHE

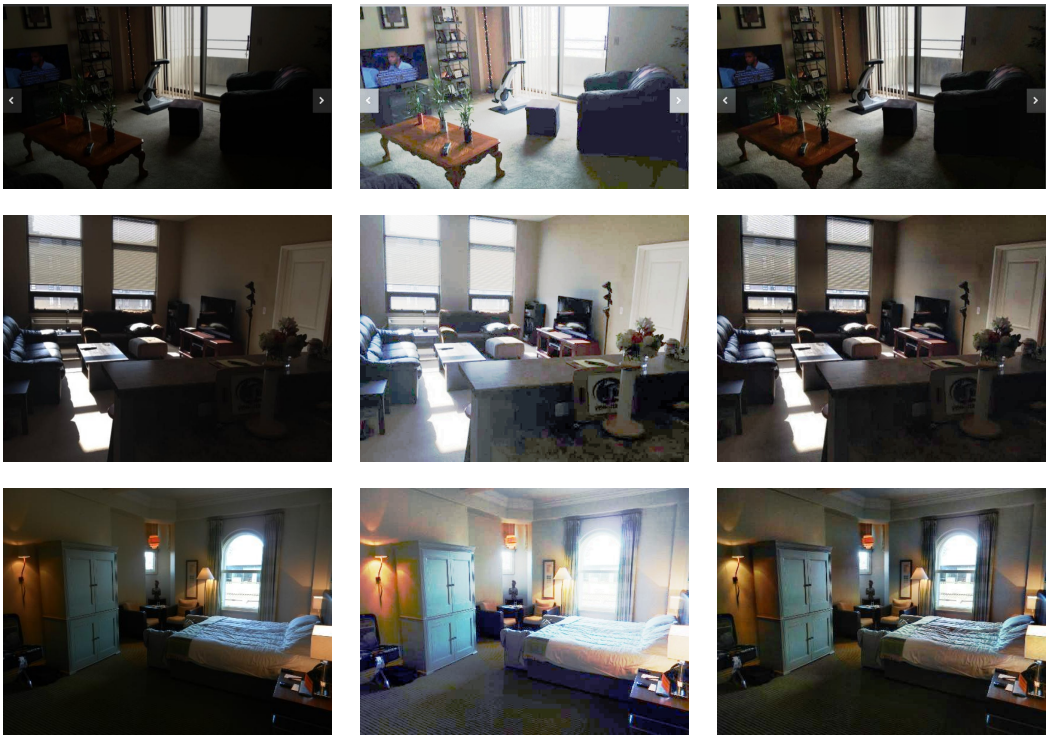
Next, we will explain how to implement the CLAHE algorithm using code: Its code divides the image into blocks according to `tile_size` and processes the contrast of each small block one by one. The code will perform equalization on each sub-block ( `tile` ) and use the `clahe_stile_tile` function to perform CLAHE operations on each sub-block. Subsequently, to avoid unnatural transitions between small block boundaries, we used bilinear interpolation to smooth adjacent boundaries. The processed sub-blocks are placed back in their original positions in the image. After merging all the small blocks, the enhanced image is output. This code ensures image contrast enhancement and smooth transitions between blocks.

Code 7: CLAHE

```
def clahe(image, clip_limit, tile_size):
    rows, cols = image.shape
    bins = 256
    output_image = np.zeros_like(image)
    for i in range(0, rows, tile_size):
        for j in range(0, cols, tile_size):
            tile = image[i:i+tile_size, j:j+tile_size]
            tile_padded = np.pad(tile, ((1, 1), (1, 1)), mode='edge')
            eq_tile = clahe_single_tile(tile_padded[1:-1, 1:-1],
                clip_limit, bins)
            for r in range(tile_size):
                for c in range(tile_size):
                    x = r + 0.5
                    y = c + 0.5
                    x1, x2 = int(x), int(x) + 1
                    y1, y2 = int(y), int(y) + 1
                    Q11 = output_image[i+r, j+c]
                    Q21 = output_image[i+r, j+c+1] if j+c+1 < cols
                        else Q11
```

```
18                      Q12 = output_image[i+r+1, j+c] if r+1 < rows else
                            Q11
19                      Q22 = output_image[i+r+1, j+c+1] if (r+1 < rows
                            and j+c+1 < cols) else Q11
20                      output_image[i+r, j+c] = bilinear_interpolate(Q12
                            , Q11, Q21, Q22, x - x1, y - y1)
21              output_image[i:i+tile_size, j:j+tile_size] = eq_tile
22      return output_image.astype(np.uint8)
```

### 4.1.2 Result



(a) Original Image (b) HE (c) CLAHE

Figure 15: Comparison of HE and CLAHE

As in Figure 15, the CLAHE algorithm enhances the contrast of the image while limiting the amplification of the noise, especially for unevenly lit images, and avoids the over-enhancement problem commonly seen in histogram equalization by contrast limitation. The bilinear interpolation technique reduces the artificial boundaries brought about by local processing and makes the final image more natural.

## 4.2 Region-based Histogram Equalization

In addition, we have independently designed and implemented a new histogram equalization method based on the above existing algorithms Region-based Histogram Equalization (RBHE). RBHE is an enhanced image processing technique designed to address the issue of uneven contrast enhancement in global histogram equalization. The core concept is to divide the image into multiple smaller

regions, perform histogram equalization on each region separately, and then merge the processed regions back into the original image. This way, local areas can achieve more refined contrast enhancement, making it particularly suitable for images with complex lighting or uneven contrast distribution.

To achieve this goal, we first divide the image into multiple regions based on different image characteristics, such as texture and brightness. Then, we perform histogram equalization or other enhancement processing on each region separately and finally recombine the processed regions into a complete image.

However, the results of following a simple approach are not satisfactory; the processed image has a very noticeable regional segmentation feature, meaning that the boundaries between different regions are particularly distinct. Researchers have attempted to address this issue by using sliding window operations and weighted fusion techniques.

### 4.2.1 Sliding Window

The Sliding Window operation is a widely utilized technique in image processing that addresses local areas of an image while preserving continuity with neighboring regions. It finds application in numerous computer vision tasks, such as object detection and feature extraction.

When processing each region in isolation, it can lead to abrupt transitions between the regions, particularly in images with significant variations in brightness and contrast. To mitigate this issue, an overlap can be introduced for each region, referred to as the Sliding Window area. Consequently, when processing each region, a portion of the pixels from the adjacent regions is also taken into account, facilitating a smooth transition between the regions. The size of the Sliding Window is typically governed by an "overlap" parameter, which indicates the proportion of pixels shared between neighboring regions. A higher overlap rate implies that more pixels are shared, resulting in a more natural transition between the processed regions. The specific operation is as follows:

Firstly, the image is divided into multiple non-overlapping core areas according to certain rules. Assuming the image size is $H \times W$, and it is divided into $m \times n$ areas, the size of each core area is:

$$\text{Area height} = \frac{H}{m} \tag{19}$$

$$\text{Area width} = \frac{W}{n} \tag{20}$$

These core areas are non-overlapping and are the basic processing units for the Sliding Window operation.

The key to introducing the Sliding Window is to have each area overlap with adjacent areas at the boundary. For example, assuming the size of each core area is $R \times C$, and the overlap rate is $overlap$, then the size of the overlapping part is:

$$\text{Overlapping height} = \text{Area height} \times \text{overlap} \tag{21}$$

$$\text{Overlapping width} = \text{Area width} \times \text{overlap} \tag{22}$$

In actual processing, the pixels in the overlapping area are processed by multiple adjacent areas. The processing range for each area becomes:

$$\text{Actual processing height} = R + 2 \times \text{Overlapping height} \tag{23}$$

$$\text{Actual processing width} = C + 2 \times \text{Overlapping width} \tag{24}$$

In this way, each area will have shared pixels with adjacent areas on all sides.

Code 8: Sliding Window Operation

```
def SlidingWindow(self, channel):
        height, width = channel.shape
        num_rows, num_cols = self.num_regions
        region_h = int(height // num_rows)
        region_w = int(width // num_cols)
        overlap_h = int(region_h * self.overlap)
        overlap_w = int(region_w * self.overlap)
        new_channel = np.zeros_like(channel)
        for i in range(num_rows):
            for j in range(num_cols):
                i_start = max(0, i * region_h - overlap_h)
                j_start = max(0, j * region_w - overlap_w)
                i_end = min(height, (i + 1) * region_h + overlap_h)
                j_end = min(width, (j + 1) * region_w + overlap_w)
                tile = channel[i_start:i_end, j_start:j_end]
                cdf = self.calculate_cdf(tile)
                region_center_i_start = i * region_h
                region_center_j_start = j * region_w
                region_center_i_end = (i + 1) * region_h
                region_center_j_end = (j + 1) * region_w
                new_channel[region_center_i_start:region_center_i_end
                    , region_center_j_start:region_center_j_end] =
                cdf[channel[region_center_i_start:region_center_i_end
                    ,region_center_j_start:region_center_j_end] ]
        return new_channel
```

### 4.2.2 Weights Fusion

A weighted fusion approach (similar to convolution operations) can perform a weighted average on the overlapping areas in the sliding window area. Pixels closer to the region's center have greater weight, while pixels farther from the edges have lesser weight. This weighting scheme ensures a smooth transition in the overlapping areas without producing noticeable boundaries. Specifically, to make the transition more natural, we assign weights to each pixel within the sliding window area. We use the concept of a convolution kernel to generate a weight matrix, where the center of the weight matrix has the highest weight and the edges have the lowest. For instance, a bilinear interpolation-based weight matrix can be created:

$$\text{Weight}(x,y) = 1 - \frac{\sqrt{(x - x_{\text{center}})^2 + (y - y_{\text{center}})^2}}{\text{Maximum Distance}} \tag{25}$$

This weight matrix is used to weigh each pixel within the sliding window area, such that pixels closer to the center of the region primarily rely on the results of that area. In contrast, pixels closer to the edge rely more on the processing results of adjacent areas. After multiple regions have processed the pixel values of each sliding window area, they are averaged with the weight matrix to obtain the final pixel values. In this way, the pixel values not only reflect the processing results of the current area but also incorporate the results of neighboring areas, achieving a smooth transition effect.

Code 9: Weight Fusion Operation

```
def WeightFusion(self, channel):
        ...
        weight_map = np.zeros_like(channel, dtype=np.float32)
        for i in range(num_rows):
            for j in range(num_cols):
                ...
                weight = self.create_weight_matrix(i_end - i_start,
                    j_end - j_start)
                new_channel[i_start:i_end, j_start:j_end] += weight *
                    cdf[tile]
                weight_map[i_start:i_end, j_start:j_end] += weight
        new_channel = new_channel / (weight_map + 1e-5)
        new_channel = new_channel.astype(np.uint8)
        return new_channel.astype(np.uint8)
```

### 4.2.3 Result

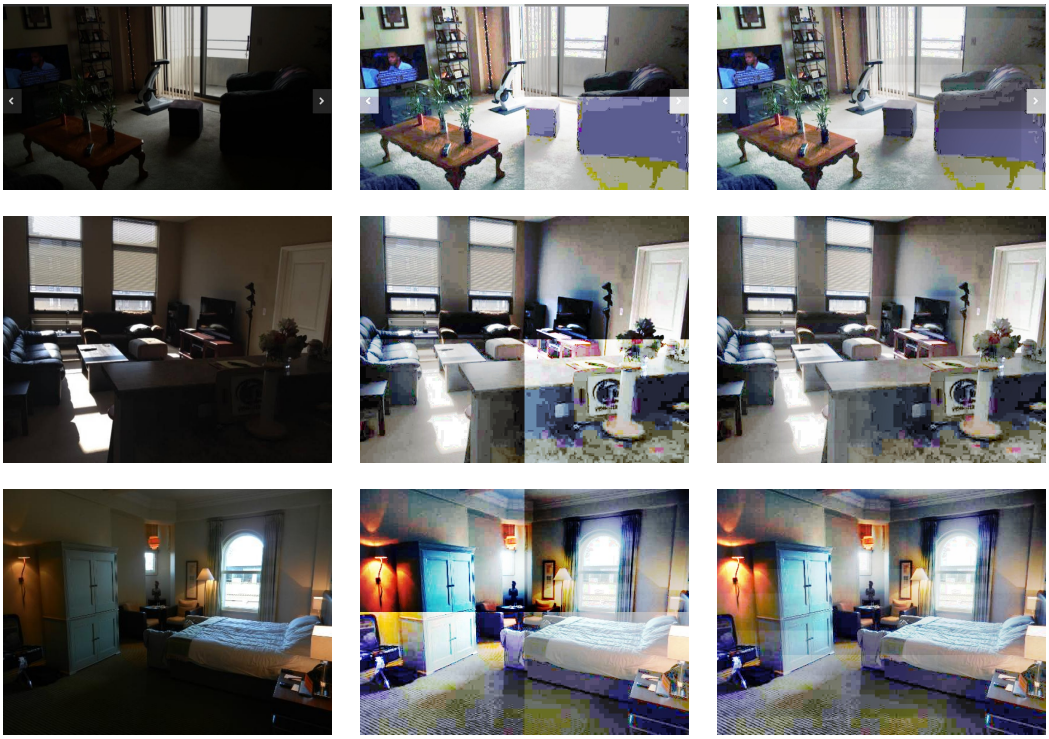
As shown in Figure 16, the regional pictures have an obvious demarcation line, i.e. the edges between different blocks are pronounced. However, it can still be seen that the equalization effect is better within each block. After weight fusion, the picture retains the sound characteristics of the different regions, but there are no apparent edges between the different blocks.

## 4.3 Parallel Histogram Equalization

Serial computing, or sequential computing, is a type of computing where instructions for solving computing problems are followed one at a time or sequentially. The fundamentals of serial computing require systems to use only one processor rather than distributing problems across multiple processors [9]. With computer science evolving, parallel computing is introduced because of the slow speeds of serial computing. Parallel computing, also known as parallel programming, is a process where large computing problems are broken down into smaller problems that can be solved simultaneously by multiple processors.

Various parallel application programming interfaces (API) are provided to simplify parallel programming, such as MPI (Message Passing Interface) for distributed memory architecture, OpenMP (Open Multi-Processing) for shared memory architecture, and CUDA (Compute Unified Device Architecture) for NVIDIA's graphics processing units (GPUs).

In this part, the main goal is to accelerate the speed of histogram equalization. We chose the threading module of the Python language and CUDA as an example.



(a) Original Image (b) Region HE (c) Improved Region HE

Figure 16: Comparison of Region HE and Improved Region HE

### 4.3.1 Python Threading Module

The first way to parallel the histogram equalization procedure is with Python's threading module. The threading module provides thread-based concurrency in Python. Technically, it is implemented on top of another lower-level module called `_thread`. The thread pool is also needed here to control the thread usage.

The parallel histogram equalization program using a threading module is divided into several functions. The `split_image` function is first defined to split the image into several blocks. Then, `parallel_histogram_equalization` calls the pre-defined equalization function, and the function `equalize_histogram_block` is to perform parallel histogram equalization on each image block. Next, `merge_blocks` merges the processed image blocks into a complete image. In addition, `run_benchmark` is used for benchmarking, recording the processing time under different thread counts.

Code 10: Parallel Histogram Equalization by Threading Module

```
def parallel_histogram_equalization(image, num_blocks, num_threads):
    blocks = split_image(image, num_blocks)
    # Use thread pool to process blocks in parallel
    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        results = executor.map(equalize_histogram_block, blocks)
    return merge_blocks(list(results))
```

### 4.3.2 CUDA

However, compared with other programming languages like C or C++, Python has a key limitation of threading. Python's thread for thread-based concurrency is that it is subject to the Global Interpreter Lock (GIL), which means that only one thread can run at a time in a Python process, unless the GIL is released, such as during I/O or explicitly in third-party libraries. So CUDA is introduced here to accelerate the speed of Histogram Equalization while meeting the picture with huge amounts of pixels. Meanwhile, the previous computing relies on the CPU, especially the ALU units inside, while the ALU's number is limited.

The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. Typically, we run the serial workload on the CPU and offload parallel computation to GPUs.

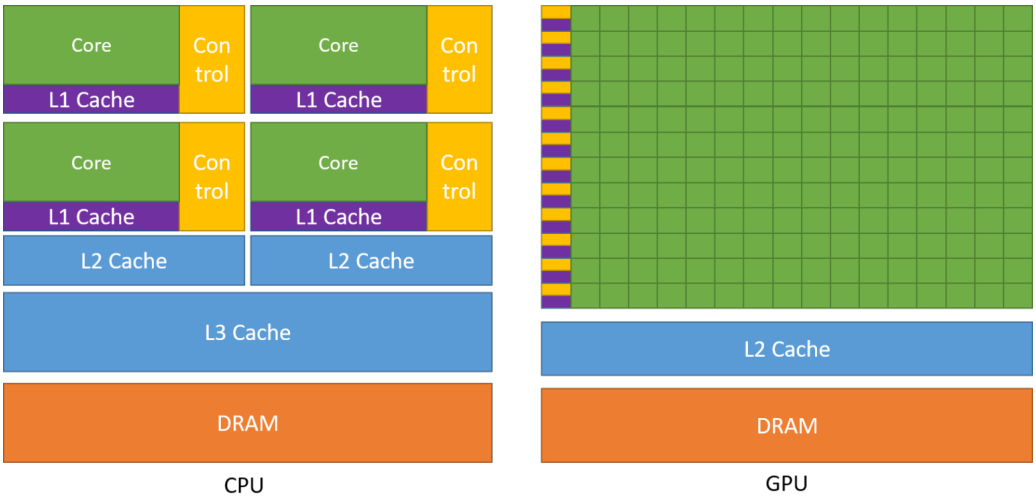



Figure 17: The GPU Devotes More Transistors to Data Processing [10]

The GPU is specialized for highly parallel computations and, therefore, designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic figure below shows an example distribution of chip resources for a CPU versus a GPU [10].

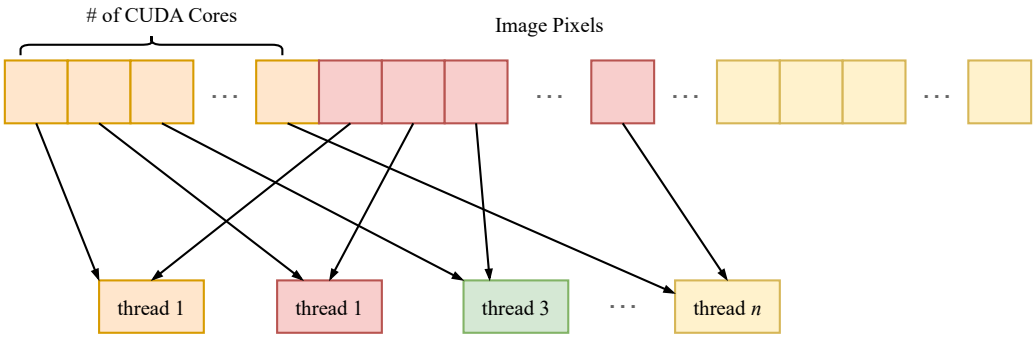



Figure 18: CUDA Image Management

The code and execution process of parallel histogram equalization using CUDA is nearly identical to that of Python threading. The main idea for the tasks parallelization has been the following: if

possible, depending on the number of CUDA cores, pixels of the image have been processed one per thread. Otherwise, what happens is illustrated in Figure18. However, compared with the execution on the CPU, we need to transfer data over the bus to the GPU and execute the CUDA kernel function on it for computation. After the calculation is complete, the data also need to be transferred back for image reconstruction.

Code 11: Parallel Histogram Equalization by CUDA

```
# CUDA kernel code
kernel_code = """
__global__ void histogram_equalization(unsigned char *img, unsigned
    char *out_img, int width, int height, int channels) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        int index = (y * width + x) * channels;
        for (int c = 0; c < channels; ++c) {
            out_img[index + c] = img[index + c];
        }
    }
}
"""

def histogram_equalization_cuda(image):
    height, width = image.shape[:2]
    channels = image.shape[2] if len(image.shape) == 3 else 1

    img_np = image.astype(np.uint8)
    out_img_np = np.zeros_like(img_np)
    img_gpu = cuda.mem_alloc(img_np.nbytes)
    out_img_gpu = cuda.mem_alloc(out_img_np.nbytes)
    cuda.memcpy_htod(img_gpu, img_np)
    mod = compiler.SourceModule(kernel_code)
    func = mod.get_function("histogram_equalization")
    block_size = (16, 16, 1)
    grid_size = (int(np.ceil(width / block_size[0])), int(np.ceil(
        height / block_size[1])), 1)

    # Create CUDA events
    start_event = cuda.Event()
    end_event = cuda.Event()
    # Record start event
    start_event.record()
    # Execute the kernel
    func(img_gpu, out_img_gpu, np.int32(width), np.int32(height), np.
        int32(channels), block=block_size, grid=grid_size)
    # Record end event and synchronize
    end_event.record()
    end_event.synchronize()
    # Compute elapsed time
    elapsed_time = start_event.time_till(end_event) / 1000.0  #
```

```
        Convert milliseconds to seconds

    cuda.memcpy_dtoh(out_img_np, out_img_gpu)
    return out_img_np, elapsed_time
```

Here is the source code and general introduction to the program flow for parallel acceleration of histogram equalization using CUDA:

- **Initialization:** Load and divide the image into smaller blocks.
- **Memory Allocation:** Allocate memory on the GPU for input and output image data.
- **Data Transfer:** Copy image data from CPU to GPU.
- **Kernel Execution:** Execute the CUDA kernel to process each image block.
- **Time Measurement:** Measure the execution time of the kernel.
- **Data Transfer:** Copy the processed data from GPU back to CPU.
- **Image Reconstruction:** Combine the processed blocks to form the final image.

### 4.3.3 Result

Tests have been performed using a larger picture [11], the scenery of the Hive, Nanyang Technological University, to demonstrate the acceleration effect.

We have tested the images above on Parallel and Distributed Computing Lab (PDCL)'s server, with 1, 2, 4, 6, and 12 threads of Intel Xeon W-2235 CPU and NVIDIA A5000 GPU. The baseline is using one thread of CPU.

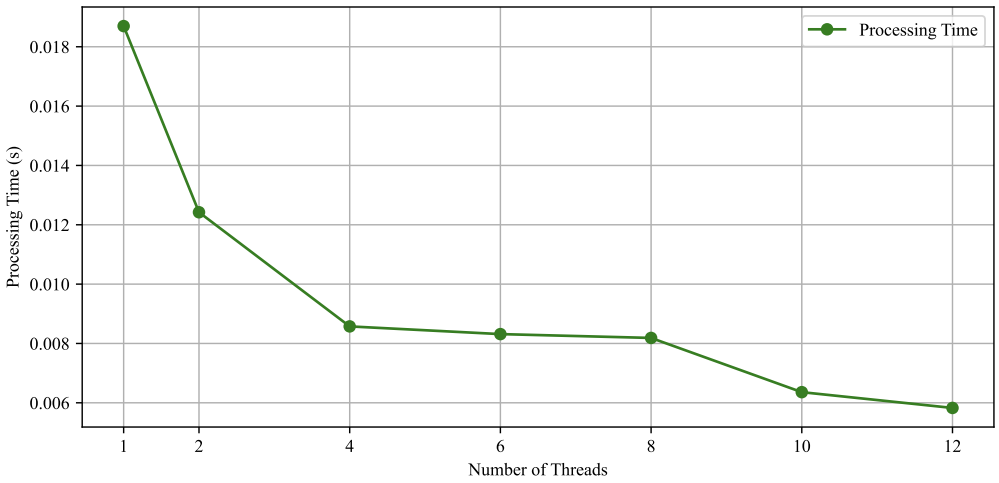



Figure 19: Result of Parallel HE Using Threading Module

We got the final result based on multiple rounds of testing and averaging the running time. The baseline is 0.0187 seconds. We can intuitively find that with the increase in the number of threads, the calculation time of histogram equalization is significantly reduced, with the minimization of 0.0058 seconds when using 12 threads.

Table 2: Time and Speedup of Parallel Histogram Equalization

| | Baseline | Python Thread (Fastest) | CUDA |
|---|---|---|---|
| Time (s) | 0.0187 | 0.0058 | 0.0176 |
| Speedup | 1 | 3.22 | 1.06 |

Meanwhile, parallel histogram equalization using CUDA was completed in 0.0176 seconds, slightly faster than the baseline method. However, it still falls short compared to the fastest results, which use the CPU's 12 threads. This performance discrepancy stems from the overhead of transferring data between the CPU and GPU via bus during computation. Consequently, for smaller datasets, like the cast that the single image example with a moderate pixel count in this study, the acceleration effect may not be significant enough.

# References

[1] W. A. Mustafa and M. M. M. Abdul Kader, "A review of histogram equalization techniques in image enhancement application," in *Journal of Physics: Conference Series*, vol. 1019, p. 012026, IOP Publishing, 2018.

[2] Wikipedia contributors, "Histogram equalization," 2024. Accessed: 2024-09-11.

[3] J. Maller, "Fxscript reference: Rgb and yuv color." Archive.org, 2002. Accessed: 2024-09-06.

[4] V. T. I. E2ESOFT, "Digital video introduction," July 21 2021. Accessed: 2024-09-06.

[5] Black Ice Software, *Black Ice Document Imaging OCX Online Manual*, 2024. Accessed: 2024-09-06.

[6] M. Carroll, "Hsi and hsv color space," 2015. Accessed: 2024-09-06.

[7] R. D. Warner, "Measurements of water-holding capacity and color: Objective and subjective," in *Encyclopedia of Meat Sciences (Third Edition)*, vol. 3, Academic Press, 2024. pp. 478-492.

[8] B. MacEvoy, "Handprint: Cielab ab plane." https://www.handprint.com/HP/WCL/labwheel.html, August 15 2005. Accessed: 2024-09-08.

[9] IBM, "Parallel computing." https://www.ibm.com/think/topics/parallel-computing, n.d. Accessed: 2024-09-13.

[10] NVIDIA Corporation, *CUDA C++ Programming Guide*. NVIDIA Corporation, release 12.6 ed., August 2024.

[11] Nanyang Technological University, "Hive banner," 2024. Accessed: 2024-09-13.

# A Source Code

Code 12: Main.py

```python
import os
from GrayscaleHE import GrayscaleHE
from ColorfulPictureHE import ColorPictureHE

if __name__ == '__main__':
    img_path = "input_images"
    img_file = os.listdir(img_path)

    #Processing Grayscale Historgram
    for img in img_file:
        HEG = GrayscaleHE(img_path,img)
        HEG.draw_histogram()
        HEG.equalization()
        HEG.draw_new_histogram()

    channels = ["YCrCb", "WeightedRGB", "RGB", "SharedRGB","HSV", "
       LAB", "HLS", "CALYCrCb","CALRGB"]
    for channel in channels:
        for img in img_file :
            HE = ColorPictureHE(img_path,img,channel)
            HE.draw_histogram()
            HE.equalization()
            HE.draw_new_histogram()
```

Code 13: Grayscale HE.py

```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib

class GrayscaleHE(object):
    def __init__(self, img_path, img_file):
        self.img_file = img_file
        self.OriginalImg = cv2.imread(os.path.join(img_path, img_file
           ))
        self.Histogram = np.zeros((1, 256), dtype=np.intp)
        self.NewHistogram = np.zeros((1, 256), dtype=np.intp)
        self.Cumulative_distribution = np.zeros((1, 256), dtype=np.
           float64)
        matplotlib.rcParams['font.sans-serif'] = ['SimHei']
        matplotlib.rcParams['axes.unicode_minus'] = False

    def resize(self, size: tuple = (64, 64)):
        self.OriginalImg = cv2.resize(self.OriginalImg, size)
```

```
19
20      def draw_histogram(self):
21          """
22          convert the image to grayscale
23          calculate histogram from pixels
24          draw histogram
25          :return: None
26          """
27          # convert the image to grayscale
28          self.OriginalImg = cv2.cvtColor(self.OriginalImg, cv2.
                COLOR_BGR2GRAY)
29          # calculate histogram from pixels
30          height = self.OriginalImg.shape[0]
31          width = self.OriginalImg.shape[1]
32          for row in range(height):
33              for col in range(width):
34                  I = self.OriginalImg[row][col]
35                  self.Histogram[0][I] += 1
36          # draw histogram
37          x = np.asarray(self.OriginalImg)
38          x.resize((height * width))
39          plt.figure()
40          plt.hist(x, bins=256, color='green')
41          plt.xlabel('gray level')
42          plt.ylabel('frequency')
43          plt.title('input image histogram')
44          os.makedirs(os.path.join("results", "GrayscaleHE", "
                origin_histogram"), exist_ok=True)
45          plt.savefig(os.path.join("results", "GrayscaleHE", "
                origin_histogram", self.img_file))
46          x.resize((height, width))
47          plt.close()
48
49      def equalization(self):
50          """
51          Histogram Equalization using cumulative distribution function
52          :return: None
53          """
54          # Scale I and c into [0, 255] after getting the cumulative
                distribution function
55          height = self.OriginalImg.shape[0]
56          width = self.OriginalImg.shape[1]
57          N = height * width # number of pixels
58          for i in range(256):
59              if i == 0:
60                  self.Cumulative_distribution[0][i] = self.Histogram
                        [0][i] / N
61              elif i == 255:
62                  self.Cumulative_distribution[0][i] = 1.0
63              else:
```

```
64                  self.Cumulative_distribution[0][i] = self.Histogram
                        [0][i] / N + \
65                      self.Cumulative_distribution[0][i - 1]
66
67          self.Cumulative_distribution = self.Cumulative_distribution *
                255
68          # draw new image
69          self.NewImg = self.OriginalImg.copy()
70          self.Cumulative_distribution = self.Cumulative_distribution.
                astype(np.intp)
71          for row in range(height):
72              for col in range(width):
73                  Newvalue = self.Cumulative_distribution[0][self.
                        NewImg[row][col]]
74                  self.NewImg[row][col] = Newvalue
75          os.makedirs(os.path.join("results", "GrayscaleHE", "
                result_images"), exist_ok=True)
76          cv2.imwrite(os.path.join("results", "GrayscaleHE", "
                result_images", self.img_file), self.NewImg)
77
78      def draw_new_histogram(self):
79          """draw histogram of the result image"""
80          # compute the histogram
81          height = self.OriginalImg.shape[0]
82          width = self.OriginalImg.shape[1]
83          for row in range(height):
84              for col in range(width):
85                  I = self.NewImg[row][col]
86                  self.NewHistogram[0][I] += 1
87          # draw histogram
88          self.NewImg.resize((height*width))
89          plt.figure()
90          plt.hist(self.NewImg, bins=256, color='green')
91          plt.xlabel('gray level')
92          plt.ylabel('frequency')
93          plt.title('histogram after HE')
94          os.makedirs(os.path.join("results", "GrayscaleHE", "
                new_histogram"), exist_ok=True)
95          plt.savefig(os.path.join("results", "GrayscaleHE", "
                new_histogram", self.img_file))
96          self.NewImg.resize((height, width))
97          plt.close()
```

Code 14: Colorful Picture HE.py

```
1  import os
2  import cv2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import matplotlib
```

```
6
7   class ColorPictureHE(object):
8       def __init__(self, img_path, img_file, channel = "WeightRGB"):
9           self.img_file = img_file
10          self.OriginalImg = cv2.imread(os.path.join(img_path, img_file
                ))
11          self.Histogram = np.zeros((3, 256), dtype=np.intp)
12          self.NewHistogram = np.zeros((3, 256), dtype=np.intp)
13          self.Cumulative_distribution = np.zeros((3, 256), dtype=np.
                float64)
14          self.channel = channel
15          matplotlib.rcParams['font.sans-serif'] = ['SimHei']
16          matplotlib.rcParams['axes.unicode_minus'] = False
17
18      def valification(self):
19          """
20          Using opencv functions to validate our implement of
                histogram_equalization
21          return: None
22          """
23          valification_img = self.OriginalImg.copy()
24          hsv = cv2.cvtColor(valification_img, cv2.COLOR_RGB2HSV)
25          channels = cv2.split(hsv)
26          cv2.equalizeHist(channels[2], channels[2])
27          cv2.merge(channels, hsv)
28          # convert HSV to RGB image
29          cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB, valification_img)
30          # save validation image
31          os.makedirs(os.path.join("results", "Validation", "
                validation_images"), exist_ok=True)
32          cv2.imwrite(os.path.join("results", "Validation", "
                validation_images", self.img_file), valification_img)
33          # draw histogram of the validation image
34          plt.figure()
35          plt.title('cv2.equalizeHist image')
36          plt.subplot(3, 1, 1)
37          plt.hist(valification_img[:,:,0].flatten(), bins=256, color='
                red')
38          plt.subplot(3, 1, 2)
39          plt.hist(valification_img[:,:,1].flatten(), bins=256, color='
                green')
40
41          plt.ylabel('frequency')
42          plt.subplot(3, 1, 3)
43          plt.hist(valification_img[:,:,2].flatten(), bins=256, color='
                blue')
44          plt.xlabel('gray level')
45          os.makedirs(os.path.join("results", "Validation", "
                validation_histogram"), exist_ok=True)
46          plt.savefig(os.path.join("results", "Validation", "
```

```
            validation_histogram", self.img_file))

    def draw_histogram(self):
        height = self.OriginalImg.shape[0]
        width = self.OriginalImg.shape[1]

        for row in range(height):
            for col in range(width):
                r = self.OriginalImg[row][col][0]
                self.Histogram[0][r] += 1
                g = self.OriginalImg[row][col][1]
                self.Histogram[1][g] += 1
                b = self.OriginalImg[row][col][2]
                self.Histogram[2][b] += 1

        plt.figure()
        plt.title('input image histogram')
        plt.subplot(3, 1, 1)
        plt.hist(self.OriginalImg[:, :, 0].flatten(), bins=256, color
            ='red')
        plt.subplot(3, 1, 2)
        plt.hist(self.OriginalImg[:, :, 1].flatten(), bins=256, color
            ='green')

        plt.ylabel('frequency')
        plt.subplot(3, 1, 3)
        plt.hist(self.OriginalImg[:, :, 2].flatten(), bins=256, color
            ='blue')

        plt.xlabel('gray level')
        os.makedirs(os.path.join("results", f"{self.channel}ChannelHE
            ", "origin_histogram"), exist_ok=True)
        plt.savefig(os.path.join("results", f"{self.channel}ChannelHE
            ", "origin_histogram", self.img_file))
        plt.close()

    def valification(self):
        """
        opencvhistogram_equalization
        return: None
        """
        valification_img = self.OriginalImg.copy()
        hsv = cv2.cvtColor(valification_img, cv2.COLOR_RGB2HSV)
        channels = cv2.split(hsv)
        cv2.equalizeHist(channels[2], channels[2])
        cv2.merge(channels, hsv)
        # hsvRGB
        cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB, valification_img)
        #
        os.makedirs(os.path.join("results", "Validation", "
```

```
            validation_images"), exist_ok=True)
91          cv2.imwrite(os.path.join("results", "Validation", "
            validation_images", self.img_file), valification_img)
92          #
93          plt.figure()
94          plt.title('cv2.equalizeHist image')
95          plt.subplot(3, 1, 1)
96          plt.hist(valification_img[:,:,0].flatten(), bins=256, color='
            red')
97          plt.subplot(3, 1, 2)
98          plt.hist(valification_img[:,:,1].flatten(), bins=256, color='
            green')
99
100         plt.ylabel('frequency')
101         plt.subplot(3, 1, 3)
102         plt.hist(valification_img[:,:,2].flatten(), bins=256, color='
            blue')
103         plt.xlabel('gray level')
104         os.makedirs(os.path.join("results", "Validation", "
            validation_histogram"), exist_ok=True)
105         plt.savefig(os.path.join("results", "Validation", "
            validation_histogram", self.img_file))
106
107     def equalization(self, CAL = False):
108         height = self.OriginalImg.shape[0]
109         width = self.OriginalImg.shape[1]
110         N = height * width
111         if self.channel == "WeightedRGB" or self.channel == "
            SharedRGB":
112             pass
113         else:
114             for i in range(256):
115                 if i == 0:
116                     self.Cumulative_distribution[0][i] = self.
                        Histogram[0][i] / N
117                     self.Cumulative_distribution[1][i] = self.
                        Histogram[1][i] / N
118                     self.Cumulative_distribution[2][i] = self.
                        Histogram[2][i] / N
119                 elif i == 255:
120                     self.Cumulative_distribution[0][i] = 1.0
121                     self.Cumulative_distribution[1][i] = 1.0
122                     self.Cumulative_distribution[2][i] = 1.0
123                 else:
124                     self.Cumulative_distribution[0][i] = self.
                        Histogram[0][i] / N + \
125                         self.Cumulative_distribution[0][i - 1]
126                     self.Cumulative_distribution[1][i] = self.
                        Histogram[1][i] / N + \
127                         self.Cumulative_distribution[1][i - 1]
```

```
128                     self.Cumulative_distribution[2][i] = self.
                            Histogram[2][i] / N + \
129                          self.Cumulative_distribution[2][i - 1]
130
131             self.Cumulative_distribution = self.
                    Cumulative_distribution * 255
132             self.NewImg = self.OriginalImg.copy()
133
134             self.Cumulative_distribution = self.
                    Cumulative_distribution.astype(np.intp)
135
136         if self.channel == "RGB":
137             for row in range(height):
138                 for col in range(width):
139                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][0]]
140                     self.NewImg[row][col][0] = Newvalue
141                     Newvalue = self.Cumulative_distribution[1][self.
                            NewImg[row][col][1]]
142                     self.NewImg[row][col][1] = Newvalue
143                     Newvalue = self.Cumulative_distribution[2][self.
                            NewImg[row][col][2]]
144                     self.NewImg[row][col][2] = Newvalue
145             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
146             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
147
148         elif self.channel == "WeightedRGB":
149             N = height * width
150             for i in range(256):
151                 if i == 0:
152                     self.Cumulative_distribution[0][i] = self.
                            Histogram[0][i] / N
153                 elif i == 255:
154                     self.Cumulative_distribution[0][i] = 1.0
155                 else:
156                     self.Cumulative_distribution[0][i] = self.
                            Histogram[0][i] / N + \
157                          self.Cumulative_distribution[0][i - 1]
158
159             self.Cumulative_distribution = self.
                    Cumulative_distribution * 255
160             #
161             self.NewImg = self.OriginalImg.copy()
162
163             self.Cumulative_distribution = self.
                    Cumulative_distribution.astype(np.intp)
164             for row in range(height):
```

```
165                 for col in range(width):
166                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][0]]
167                     self.NewImg[row][col][0] = Newvalue
168                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][1]]
169                     self.NewImg[row][col][1] = Newvalue
170                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][2]]
171                     self.NewImg[row][col][2] = Newvalue
172             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
173             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
174
175         elif self.channel == "SharedRGB":
176             for i in range(256):
177                 if i == 0:
178                     self.Cumulative_distribution[0][i] = self.
                            Histogram[0][i] / N
179                 elif i == 255:
180                     self.Cumulative_distribution[0][i] = 1.0
181                 else:
182                     self.Cumulative_distribution[0][i] = self.
                            Histogram[0][i] / N + \
183                         self.Cumulative_distribution[0][i - 1]
184
185             self.Cumulative_distribution = self.
                    Cumulative_distribution * 255 / 3
186             self.NewImg = self.OriginalImg.copy()
187
188             self.Cumulative_distribution = self.
                    Cumulative_distribution.astype(np.intp)
189             for row in range(height):
190                 for col in range(width):
191                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][0]]
192                     self.NewImg[row][col][0] = Newvalue
193                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][1]]
194                     self.NewImg[row][col][1] = Newvalue
195                     Newvalue = self.Cumulative_distribution[0][self.
                            NewImg[row][col][2]]
196                     self.NewImg[row][col][2] = Newvalue
197             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
198             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
```

```
199
200         elif self.channel == "HSV":
201             hsv_image = cv2.cvtColor(self.OriginalImg, cv2.
                COLOR_BGR2HSV)
202             h, s, v = cv2.split(hsv_image)
203             for row in range(height):
204                 for col in range(width):
205                     v_new_value = self.Cumulative_distribution[2][v[
                        row, col]]
206                     v[row, col] = v_new_value
207             hsv_image_eq = cv2.merge([h, s, v])
208             self.NewImg = cv2.cvtColor(hsv_image_eq, cv2.
                COLOR_HSV2BGR)
209             os.makedirs(os.path.join("results", f"{self.channel}
                ChannelHE", "result_images"), exist_ok=True)
210             cv2.imwrite(os.path.join("results", f"{self.channel}
                ChannelHE", "result_images", self.img_file), self.
                NewImg)
211
212         elif self.channel == "YCrCb":
213             ycrcb_image = cv2.cvtColor(self.OriginalImg, cv2.
                COLOR_BGR2YCrCb)
214             y, cr, cb = cv2.split(ycrcb_image)
215             for row in range(height):
216                 for col in range(width):
217                     y_new_value = self.Cumulative_distribution[0][y[
                        row, col]]
218                     y[row, col] = y_new_value
219             ycrcb_image_eq = cv2.merge([y, cr, cb])
220             self.NewImg = cv2.cvtColor(ycrcb_image_eq, cv2.
                COLOR_YCrCb2BGR)
221             os.makedirs(os.path.join("results", f"{self.channel}
                ChannelHE", "result_images"), exist_ok=True)
222             cv2.imwrite(os.path.join("results", f"{self.channel}
                ChannelHE", "result_images", self.img_file), self.
                NewImg)
223
224         elif self.channel == "LAB":
225             lab_image = cv2.cvtColor(self.OriginalImg, cv2.
                COLOR_BGR2Lab)
226             l, a, b = cv2.split(lab_image)
227             for row in range(height):
228                 for col in range(width):
229                     l_new_value = self.Cumulative_distribution[0][l[
                        row, col]]
230                     l[row, col] = l_new_value
231             lab_image_eq = cv2.merge([l, a, b])
232             self.NewImg = cv2.cvtColor(lab_image_eq, cv2.
                COLOR_Lab2BGR)
233             os.makedirs(os.path.join("results", f"{self.channel}
```

```
                ChannelHE", "result_images"), exist_ok=True)
234             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
235
236         elif self.channel == "HLS":
237             hls_image = cv2.cvtColor(self.OriginalImg, cv2.
                    COLOR_BGR2HLS)
238             h, l, s = cv2.split(hls_image)
239             for row in range(height):
240                 for col in range(width):
241                     l_new_value = self.Cumulative_distribution[1][l[
                            row, col]]
242                     l[row, col] = l_new_value
243             hls_image_eq = cv2.merge([h, l, s])
244             self.NewImg = cv2.cvtColor(hls_image_eq, cv2.
                    COLOR_HLS2BGR)
245             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
246             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
247
248         elif self.channel == "CALYCrCb":
249             clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(4,
                    4))
250             ycrcb_image = cv2.cvtColor(self.OriginalImg, cv2.
                    COLOR_BGR2YCrCb)
251             y, cr, cb = cv2.split(ycrcb_image)
252             y = clahe.apply(y)
253             ycrcb_image_eq = cv2.merge([y, cr, cb])
254             self.NewImg = cv2.cvtColor(ycrcb_image_eq, cv2.
                    COLOR_YCrCb2BGR)
255             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
256             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
257
258         elif self.channel == "CALRGB":
259             clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(4,
                    4))
260             for i in range(3):
261                 self.NewImg[:, :, i] = clahe.apply(self.OriginalImg
                        [:, :, i])
262             os.makedirs(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images"), exist_ok=True)
263             cv2.imwrite(os.path.join("results", f"{self.channel}
                    ChannelHE", "result_images", self.img_file), self.
                    NewImg)
```

```
264
265         else:
266             raise ValueError(f"Unknown Channel: {self.channel}.
                Available Channels are RGB, WeightedRGB, SharedRGB,
                HSV, YCrCb, LAB, HLS, CLAHE.")
267
268     def draw_new_histogram(self):
269         plt.figure()
270         plt.title('histogram after HE')
271         plt.subplot(3, 1, 1)
272         plt.hist(self.NewImg[:, :, 0].flatten(), bins=256, color='red
                ')
273         plt.subplot(3, 1, 2)
274         plt.hist(self.NewImg[:, :, 1].flatten(), bins=256, color='
                green')
275         plt.ylabel('frequency')
276         plt.subplot(3, 1, 3)
277         plt.hist(self.NewImg[:, :, 2].flatten(), bins=256, color='
                blue')
278         plt.xlabel('gray level')
279         os.makedirs(os.path.join("results", f"{self.channel}ChannelHE
                ", "new_histogram"), exist_ok=True)
280         plt.savefig(os.path.join("results", f"{self.channel}ChannelHE
                ", "new_histogram", self.img_file))
281         plt.close()
```

Code 15: Region Based HE.py

```
1  import os
2  import cv2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import matplotlib
6  class RegionHE(object):
7
8      def __init__(self, img_path, img_file, num_regions=(4, 4),
           overlap=0.7):
9          self.img_file = img_file
10         self.OriginalImg = cv2.imread(os.path.join(img_path, img_file
               ))
11         self.num_regions = num_regions
12         self.overlap = overlap
13         self.NewImg = np.zeros_like(self.OriginalImg)
14         matplotlib.rcParams['font.sans-serif'] = ['SimHei']
15         matplotlib.rcParams['axes.unicode_minus'] = False
16
17     def calculate_cdf(self, tile):
18
19         histogram = np.zeros(256, dtype=np.intp)
20         for pixel in tile.flatten():
```

```
21              histogram[pixel] += 1
22
23          cdf = histogram.cumsum()
24          cdf_normalized = (cdf / cdf[-1]) * 255
25          cdf_normalized = cdf_normalized.astype(np.uint8)
26          return cdf_normalized
27
28      def apply_he_to_channel(self, channel):
29          height, width = channel.shape
30          num_rows, num_cols = self.num_regions
31
32          region_h = int(height // num_rows)
33          region_w = int(width // num_cols)
34          overlap_h = int(region_h * self.overlap)
35          overlap_w = int(region_w * self.overlap)
36
37          new_channel = np.zeros_like(channel, dtype=np.float32)
38
39          weight_map = np.zeros_like(channel, dtype=np.float32)
40
41          for i in range(num_rows):
42              for j in range(num_cols):
43
44                  i_start = max(0, i * region_h - overlap_h)
45                  j_start = max(0, j * region_w - overlap_w)
46                  i_end = min(height, (i + 1) * region_h + overlap_h)
47                  j_end = min(width, (j + 1) * region_w + overlap_w)
48
49                  tile = channel[i_start:i_end, j_start:j_end]
50
51                  cdf = self.calculate_cdf(tile)
52
53                  region_center_i_start = i * region_h
54                  region_center_j_start = j * region_w
55                  region_center_i_end = (i + 1) * region_h
56                  region_center_j_end = (j + 1) * region_w
57
58                  weight = self.create_weight_matrix(i_end - i_start,
                        j_end - j_start)
59
60                  new_channel[i_start:i_end, j_start:j_end] += weight *
                        cdf[tile]
61
62                  weight_map[i_start:i_end, j_start:j_end] += weight
63
64          new_channel = new_channel / (weight_map + 1e-5)
65          new_channel = new_channel.astype(np.uint8)
66          return new_channel.astype(np.uint8)
67
68      def create_weight_matrix(self, height, width):
```

```
69
70          y, x = np.mgrid[0:height, 0:width]
71          center_y, center_x = height // 2, width // 2
72          distance = np.sqrt((y - center_y)**2 + (x - center_x)**2)
73          max_distance = np.sqrt(center_y**2 + center_x**2)
74          weight = 1 - distance / max_distance
75          return weight
76
77      def apply_region_based_he(self):
78
79          channels = cv2.split(self.OriginalImg)
80
81          new_channels = [self.apply_he_to_channel(ch) for ch in
                channels]
82
83          self.NewImg = cv2.merge(new_channels)
84
85      def draw_histogram(self, img, title, save_dir):
86          plt.figure()
87          plt.title(title)
88          for i, color in enumerate(['red', 'green', 'blue']):
89              plt.hist(img[:, :, i].flatten(), bins=256, color=color,
                    alpha=0.6)
90          plt.xlabel('gray level')
91          plt.ylabel('frequency')
92          os.makedirs(save_dir, exist_ok=True)
93          plt.savefig(os.path.join(save_dir, self.img_file))
94
95      def save_image(self, save_dir):
96          os.makedirs(save_dir, exist_ok=True)
97          cv2.imwrite(os.path.join(save_dir, self.img_file), self.
                NewImg)
98
99      def process(self):
100         self.apply_region_based_he()
101
102         self.draw_histogram(self.OriginalImg, 'Original Image
                Histogram', 'path')
103
104         self.save_image( 'path')
105
106         self.draw_histogram(self.NewImg, 'Histogram after Region-
                based HE', 'path')
```

Code 16: Threading Module.py

```
1  import time
2  import threading
3  import numpy as np
4  import matplotlib.pyplot as plt
```

```
5  import cv2
6  from concurrent.futures import ThreadPoolExecutor
7
8  # Configure matplotlib to use Times New Roman font
9  plt.rcParams['font.family'] = 'serif'
10 plt.rcParams['font.serif'] = 'Times New Roman'
11 plt.rcParams['text.usetex'] = False  # Set to True to use LaTeX
12
13 def equalize_histogram_block(block):
14     """Equalize histogram of a single block of the image"""
15     b_channel, g_channel, r_channel = cv2.split(block)
16     b_eq = cv2.equalizeHist(b_channel)
17     g_eq = cv2.equalizeHist(g_channel)
18     r_eq = cv2.equalizeHist(r_channel)
19     return cv2.merge([b_eq, g_eq, r_eq])
20
21 def split_image(image, num_blocks):
22     """Split the image into multiple blocks"""
23     height, width, _ = image.shape
24     block_size = height // num_blocks
25     blocks = []
26     for i in range(num_blocks):
27         start_row = i * block_size
28         end_row = (i + 1) * block_size if i != num_blocks - 1 else
               height
29         block = image[start_row:end_row, :]
30         blocks.append(block)
31     return blocks
32
33 def merge_blocks(blocks):
34     """Merge processed image blocks back into a single image"""
35     return np.vstack(blocks)
36
37 def parallel_histogram_equalization(image, num_blocks=16, num_threads
      =1):
38     """Perform histogram equalization in parallel"""
39     blocks = split_image(image, num_blocks)
40
41     # Use thread pool to process blocks in parallel
42     with ThreadPoolExecutor(max_workers=num_threads) as executor:
43         results = executor.map(equalize_histogram_block, blocks)
44
45     return merge_blocks(list(results))
46
47 def run_benchmark(image, num_blocks, thread_counts):
48     """Run the benchmark with different thread counts"""
49     processing_times = {}
50     for num_threads in thread_counts:
51         start_time = time.perf_counter()
52         equalized_image = parallel_histogram_equalization(image,
```

```
            num_blocks=num_blocks, num_threads=num_threads)
        end_time = time.perf_counter()
        processing_time = end_time - start_time
        processing_times[num_threads] = processing_time
        print(f"Thread count: {num_threads}, Total processing time: {
            processing_time:.4f} seconds")
    return processing_times

def plot_results(processing_times):
    """Plot processing times against thread counts"""
    plt.figure(figsize=(8, 4))  # Adjust figure size
    plt.plot(list(processing_times.keys()), list(processing_times.
        values()), marker='o', label='Processing Time', color='#377D22
        ')
    plt.xlabel('Number of Threads')
    plt.ylabel('Processing Time (s)')
    plt.grid(True)
    plt.xticks(list(processing_times.keys()))  # Ensure all thread
        counts are shown on x-axis
    plt.legend()  # Add legend
    plt.tight_layout()  # Adjust layout to ensure everything fits

    # Save the figure as a high-resolution PDF
    plt.savefig('processing_time_vs_threads.pdf', format='pdf', dpi
        =600)
    plt.show()

if __name__ == "__main__":
    image = cv2.imread('1.jpg')  # Load the image file
    num_blocks = 4  # Fixed number of blocks
    thread_counts = [1, 2, 4, 6, 8, 10, 12]  # Varying thread counts
    processing_times = run_benchmark(image, num_blocks, thread_counts
        )
    plot_results(processing_times)
```

Code 17: CUDA.py

```
import numpy as np
import cv2
import pycuda.autoinit
import pycuda.driver as cuda
import pycuda.compiler as compiler

# CUDA kernel code
kernel_code = """
__global__ void histogram_equalization(unsigned char *img, unsigned
    char *out_img, int width, int height, int channels) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

```

```
    if (x < width && y < height) {
        int index = (y * width + x) * channels;
        for (int c = 0; c < channels; ++c) {
            out_img[index + c] = img[index + c];
        }
    }
}
"""

def histogram_equalization_cuda(image):
    height, width = image.shape[:2]
    channels = image.shape[2] if len(image.shape) == 3 else 1

    img_np = image.astype(np.uint8)
    out_img_np = np.zeros_like(img_np)

    img_gpu = cuda.mem_alloc(img_np.nbytes)
    out_img_gpu = cuda.mem_alloc(out_img_np.nbytes)

    cuda.memcpy_htod(img_gpu, img_np)

    mod = compiler.SourceModule(kernel_code)
    func = mod.get_function("histogram_equalization")

    block_size = (16, 16, 1)
    grid_size = (int(np.ceil(width / block_size[0])), int(np.ceil(
        height / block_size[1])), 1)

    # Create CUDA events
    start_event = cuda.Event()
    end_event = cuda.Event()

    # Record start event
    start_event.record()

    # Execute the kernel
    func(img_gpu, out_img_gpu, np.int32(width), np.int32(height), np.
        int32(channels), block=block_size, grid=grid_size)

    # Record end event and synchronize
    end_event.record()
    end_event.synchronize()

    # Compute elapsed time
    elapsed_time = start_event.time_till(end_event) / 1000.0  #
        Convert milliseconds to seconds

    cuda.memcpy_dtoh(out_img_np, out_img_gpu)

    return out_img_np, elapsed_time
```

```
60
61  def process_image(image_path):
62      image = cv2.imread(image_path)
63
64      block_size = (32, 32)
65      height, width = image.shape[:2]
66
67      blocks = []
68      for y in range(0, height, block_size[1]):
69          for x in range(0, width, block_size[0]):
70              block = image[y:y+block_size[1], x:x+block_size[0]]
71              blocks.append(block)
72
73      processed_blocks = []
74      total_time = 0
75      for block in blocks:
76          result, time_taken = histogram_equalization_cuda(block)
77          processed_blocks.append(result)
78          total_time += time_taken
79
80      # Combine blocks
81      rows = [np.hstack(processed_blocks[i:i+(width//block_size[0])])
            for i in range(0, len(processed_blocks), width//block_size[0])
            ]
82      result_image = np.vstack(rows)
83
84      return result_image, total_time
85
86  # Example usage
87  image_path = 'test.jpg'
88  result_image, time_taken = process_image(image_path)
89  print(f"Time taken for histogram equalization: {time_taken:.4f}
        seconds")
90
91  # Save the result
92  cv2.imwrite('result_image.jpg', result_image)
```