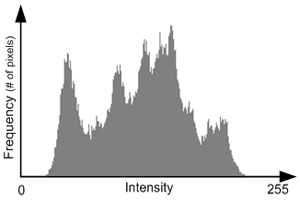
**ECTE331 Project- Question 2**

**Single-Threaded vs. Multithread implementation of Histogram Equalisation algorithm**

Histogram equalisation is a technique used to improve the visibility of image features by adjusting the image pixel intensities in order to enhance the image contrast. The technique aims to distribute the pixels evenly over the whole intensity range of the image.

A histogram is a one-dimensional array that contains the frequency (number of pixels) of each intensity value in an image.

For a b-bit image, the size of histogram is 2b, because the range of the intensity of b-bit image is from 0 (black) to 2b -1(white). Figure 1 depicts the histogram of an 8-bit image.



**Figure 1: histogram of an image**

The appendix gives the detailed steps of the histogram equalisation algorithm applied on a mono-chrome image. For a colour image; the same algorithm has to be applied on every component, i.e. Red, Green and Blue, of the image.

**Task:**

Implement the **Histogram Equalisation** algorithm using **single thread** and **multi-thread** implementations; then compare their respective ***execution times[[1]](#footnote-1)***. Furthermore, your program should implement the following requirements:

* Your code should make use of only the standard Java APIs; no other third-party API such OpenCV can be used.
* Your code should handle any image dimensions. However, the testing should be conducted using the provided image: ‘Rain\_Tree.jpg’
* For the multi-thread implementation of the algorithm, every thread should apply the algorithm on a subpart of the source image. The threads of your application should be defined in an array of length name: numOfThreads. The following three designs should be implemented:
  1. One Histogram array **shared** between all the threads with the **two** thread-memory access patterns shown in Figure 2.a and Figure 2.b. The shared histogram should be defined an **atomic array**[[2]](#footnote-2) to avoid race condition.
  2. numOfThreads **sub-histograms**, each updated by one assigned thread. The final histogram is the sum of the sub-histograms.

|  |  |
| --- | --- |
|  | **(b)** |

**Figure 2**

* Compare the execution times of the single thread and multi-thread implementation with different values of numOfThreads. The execution times should include the overhead of launching the threads.
* For fair comparison, calculate an average (~3) of the execution times for each test case. Either single- thread or multi-thread computation should be enabled in each test but not both.

**What to submit**

* Your java programs
* A report explaining the implementation of the single-thread and multi-thread implementation, with reference to the relevant lines of your code. The report should include evidence of successful implementation. The execution times should be given numerically and displayed graphically using charts with associated critical discussion. Maximum 3-4 pages.

**Assessment Criteria**

* Correctness and efficiency of the implementations code

**Appendix**

**Histogram Equalisation Algorithm of a mono-chrome 8-bit image**

Given an 8-bit image and the following data structures:

* **InpImage[**] : array that contains input image pixel values (assumed to be grey scale)
* **L**: the highest intensity value of the b-bit grey scale image, i.e. 2b -1
* **Histogram[]**: array to store the image histogram
* **CumulativeHist[]**: array to store the cumulative histogram of the image, where CumulativeHist[i] gives the number of pixels with a grey-level less than or equal to the grey level *i*
* **OutImage[]**: array to store the resulting image pixel values following histogram equalisation
* **Size**: the number of pixels in the input and output images

The three steps of **Histogram Equalisation** algorithm are:

* **compute the histogram of the input image**:

Initialise Histogram [0: 2b -1] to zeros

**For** i = 0 **to** size-1

Histogram[InpImage[i]] = Histogram[InpImage[i]] + 1

**End**

* **calculate the cumulative histogram of the image:**

Set the value of the CumulativeHist[] as follows:

CumulativeHist[0]= Histogram[0]

**For** i = 1 **to** L

CumulativeHist[i]= CumulativeHist[i-1]+ Histogram[i]

**End**

* **Image Mapping**:

This step aims to adjust the intensity values of pixels in the input image using the cumulative histogram such that the output image contains a uniform distribution of intensities.

**// Perform Histogram Equalisation**

**For** i = 0 **to** size-1

OutImage[i] = (CumulativeHist[InpImage[i]]/size)\*L ; (in coding, do the multiplication before division)

**End**

**Algorithm Computation Optimisation**

The computation in the above algorithm can be reduced because multiplication and division must be executed for every pixel in step 3 cited above. For faster computation, we can amend step 2 and 3 respectively as follows:

CumulativeHist[0]= Histogram[0]

**Step2**

**For** i = 1 **to** L

CumulativeHist[i]= CumulativeHist[i-1]+ Histogram[i];

**End**

**For** i = 0 **to** L

CumulativeHist[i]= (CumulativeHist[i]/size)\*L (in coding, do the multiplication before division)

**End**

**Step3**

**For** i = 0 **to** size-1

OutImage[i] = CumulativeHist[InpImage[i]] // Perform Histogram Equalisation

**End**

1. https://www.programiz.com/java-programming/examples/calculate-methods-execution-time [↑](#footnote-ref-1)
2. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html [↑](#footnote-ref-2)