Assignment #3 Part 2

CSD2170 Programming Massively Parallel Processors, Fall 2022

Due Date:	As specified on the moodle
Topics covered:	Shader programming, Atomic Operation, Shared Memory, Histogram, Scan, Reduction
Deliverables:	The submitted project files are the source code of your shader programs (histogram.comp, cdfscan.comp and applyHisto.comp) and C++ program (main.cpp). The files should be put in a folder and subsequently zipped according to the stipulations set out in the course syllabus.
Objectives:	Implementation of histogram equalization in shader program and Vulkan application. Learn how to use shared memory, atomic operation, histogram and scan pattern to write more efficient shader programs.

Objectives

This assignment is the implementation of image processing routines in shader programming and Vulkan application. For your reference, a C implementation are provided.

```
/*
* Copyright 2022. All rights reserved.
*
* Author: William Zheng
*
* Please refer to the end user license associated
* with this source code for terms and conditions that govern your use of
* this software. Any use, reproduction, disclosure, or distribution of
* this software and related documentation outside the terms
* is strictly prohibited.
*
*/

#define HISTOGRAM256_BIN_COUNT 256

#define mymin(a, b) (((a) < (b)) ? (a) : (b))
#define mymax(a, b) (((a) > (b)) ? (a) : (b))

#define PROB(x, width, height) (x) / ((width) * (height))
#define CLAMP(x, start, end) (mymin(mymax((x), (start)), (end)))
#define CORRECT_COLOR(cdfVal, cdfMin) CLAMP(255 * ((cdfVal) - (cdfMin)) / (1 - (cdfMin)), 0.0, 255.0)
typedef unsigned int uint;
typedef unsigned char uchar;
//assume inRGB is RGB model
// dataYUV is for YUV model
// inRGB, dataYUV, outRGB : non-interleaving layout
// histo, histoCdf : for Y component only
extern "C" void histogram256CPU(
  uint* histo,
  float* histoCdf,
  uchar* inRGB,
  uchar* dataYUV,
  uchar* outRGB,
  uint imgWidth,
  uint imgHeight,
  uint imgChannels
)
{
  //initialize histogram bin and cdf values, size = HISTOGRAM256_BIN_COUNT
  for (int i = 0; i < HISTOGRAM256_BIN_COUNT; i++) {
    histo[i] = 0;
    histoCdf[i] = 0;
  }
  int imgSz = imgWidth * imgHeight;
  //RGB to YUV
  for (int row = 0; row < imgHeight; row++) {
    for (int col = 0; col < imgWidth; col++) {
      int i = row * imgWidth + col;
      float y = 0.299f * inRGB[i] + 0.587f * inRGB[i+imgSz] + 0.114f * inRGB[i+2*imgSz];
      float u = -0.169f * inRGB[i] - 0.331f * inRGB[i+imgSz] + 0.499f * inRGB[i+2*imgSz] + 128.0f;
      float v = 0.499f * inRGB[i] - 0.418f * inRGB[i+imgSz] - 0.0813f * inRGB[i+2*imgSz] + 128.0f;
      dataYUV[i] = CLAMP(y, 0, 255);
      dataYUV[i+imgSz] = CLAMP(u, 0, 255);
      dataYUV[i+2*imgSz] = CLAMP(v, 0, 255);
    }
  }

  //histogramming on Y component
  for (int row = 0; row < imgHeight; row++) {
    for (int col = 0; col < imgWidth; col++) {
      int i = row * imgWidth + col;
      histo[dataYUV[i]]++;//histogram for Y
    }
  }

  //cdf scan: calculate cdf for histogram bins
  histoCdf[0] = PROB((float)histo[0], imgWidth, imgHeight);
  for (int i = 1; i < HISTOGRAM256_BIN_COUNT; i++) {
    float pdf = PROB((float)histo[i], imgWidth, imgHeight);
    histoCdf[i] = pdf + histoCdf[i - 1];
  }

  //get min cdf value
  float cdfMinY = histoCdf[0];

  //apply histogramming
  for (int row = 0; row < imgHeight; row++) {
    for (int col = 0; col < imgWidth; col++) {
      int i = row * imgWidth + col;

      float y = CORRECT_COLOR(histoCdf[dataYUV[i]], cdfMinY);
      float u = dataYUV[i+imgSz];
      float v = dataYUV[i+2*imgSz];

      u -= 128.0;
      v -= 128.0;

      outRGB[i] = (uchar)CLAMP(y+1.402*v, 0, 255);
      outRGB[i+imgSz] = (uchar)CLAMP(y-0.344*u-0.714*v, 0, 255);
      outRGB[i+2*imgSz]= (uchar)CLAMP(y+1.772*u, 0, 255);
    }
  }
}
```

histogram_cpu.cpp

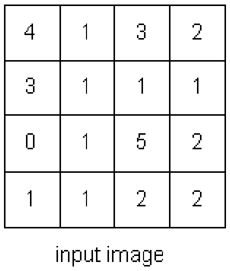
Implementation Requirement

Image Adjustment. Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

Images with skewed distributions can be helped with histogram equalization as shown in Figure 2.

Histogram equalization is a point process that redistributes the image's intensity distributions in order to obtain a uniform histogram for the image. Histogram equalization can be done in a few steps:

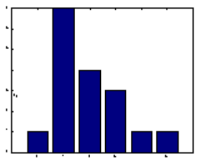
1. Create the histogram for the image.
2. Calculate the cumulative distribution function histogram (normalized sum).
3. Calculate the new values through the general histogram equalization formula.



4	1	3	2
3	1	1	1
0	1	5	2
1	1	2	2

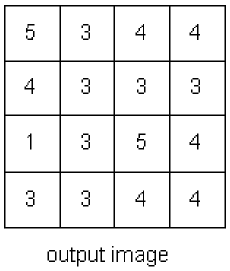
input image

(a) Image Pixel Value



(b) Histogram

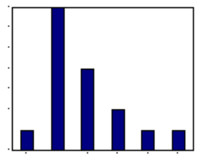
Figure 1: A sample image with a skewed histogram (poor intensity distribution)



5	3	4	4
4	3	3	3
1	3	5	4
3	3	4	4

output image

(a) Image Pixel Value



(b) Histogram

Figure 2: A histogram-equalized sample with histogram and normalized sum

4. Assign new values for each value in the image.

Table 1 shows the normalized sum of the image in Figure 1, where *sum* is the cumulative value for each intensity and normalized sum is obtained after dividing *sum* by total number of pixels, i.e. 16, and multiplying it with the maximum intensity value, i.e. 5, in this example. The histogram-equalized image, and its histogram are shown in Figure 2. Note that the resulting histogram is not truly uniform, but it is better distributed than before. Truly uniformed histograms for discrete images are difficult to obtained because of quantization.

In order to implement this on the GPU, an image histogram function is needed. Note that you have to use the 256 level method. In addition, the calculation of CDF (and therefore the minimum and maximum value of the pixel intensities) can be carried out on GPU.

intensity	sum	normalized sum	cdf
0	1	1/16*5=0.31255	1/16
1	8	8/16*5=2.5	8/16
2	12	12/16*5=3.75	12/16
3	14	14/16*5=4.375	14/16
4	15	15/16*5=4.6875	15/16
5	16	16/16*5=5.0	16/16

Table 1: Normalized Sum

To obtain the sum in Table 1, you need to perform a prefix-sum (i.e. scan pattern as taught in the class) for the pixel intensity (i.e. histogram). The actual assignment of new values for each value (pixel intensity) in the image, instead of using normalized sum as shown in Table 1, follows the following equation:

$$CLAMP(255 * ((cdfVal) - (cdfMin))/(1 - (cdfMin)), 0.0, 255.0) \tag{1}$$

where $cdfVal$ is the prefix-sum of histogram bin value divided by the total number of pixels. $cdfVal$ is the cumulative value (i.e. cdf in Table 1) for the original pixel value, and $cdfMin$ is the minimum cumulative value (e.g. the cdf value at the first row in Table 1) and $CLAMP(x, start, end)$ is defined as $min(max((x), (start)), (end))$. To obtain the new value for each pixel, the minimum value of pixel intensity $cdfMin$ is also required, as shown in Eq. 1. Please refer to the reference C code for the details.

YUV Model

Histogram equalization is a non-linear process. Channel splitting and equalizing each channel separately is incorrect. Equalization involves intensity values of the image, not the color components. So for a simple RGB color image, histogram equalization cannot be applied directly on the channels. It needs to be applied in such a way that the intensity values are equalized without disturbing the color balance of the image. So, the first step is to convert the color space of the image from RGB into one of the color spaces that separates intensity values from color components. Some of the possible options are HSV/HLS, YUV, YCbCr, etc. In this assignment, we assume that the conversion between YUV and RGB models are as follows:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.499 \\ 0.499 & -0.418 & -0.0813 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{2}$$

and

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{bmatrix} \times \begin{bmatrix} Y' \\ U - 128 \\ V - 128 \end{bmatrix} \tag{3}$$

In the shader program of `histogram.comp`, you need to convert the input of RGB model into YUV model based on Eq 2 before you start histogramming on the component of Y in YUV model. In the shader program of `applyhisto.comp`, you use the computed values of Y to index the CDF values generated by the shader program `cdfScan` to obtain the values of Y' and convert Y'UV to RGB model at the end using Eq. 3.

The following screen shot images are shown respectively before equalization (left) and after equalization (right).

Template of Image Processing Application

You should use the given template of image processing application for Vulkan. Only `main.cpp` is required to modify.

1. You should create 3 compute pipelines respectively for the compute shaders `histogram.comp`, `cdfScan.comp` and `applyhisto.comp`.

2. You can create a storage buffer as follows:

```
struct {
    unsigned int histoBin[256];
    float cdf[256];
} bufferHistoeq;
```

3. You may create the functions to build command buffers for compute pipelines of `histogram`, `cdfScan` and `applyhisto`, using `buildComputeCommandBuffer()` as the template. Please note that the workgroup size should be set correctly for each command buffer.

4. In `setupDescriptorPool()`, you should set the pool correctly for storage images and buffers.

5. Using `prepareCompute()` as the template, you can prepare for compute pipelines of `histogram`, `cdfScan` and `applyhisto`. You should set descriptor set layout binding and create the semaphore for synchronization between the compute pipelines as well as compute and graphics pipelines.

6. Create a function to prepare storage buffers that will be used in compute shaders and update/initialize the content before using.

7. In `draw()`, you can modify and submit the commands for compute pipelines of `histogram`, `cdfScan` and `applyhisto`. Create the dependencies between the compute pipelines as well as compute and graphics pipelines.

8. In `prepare()`, you can include the function calls to prepare storage buffers and compute shaders/pipelines.

9. Please remember to destroy the allocated resources such as pipeline layout, descriptor set layout, semaphore and command pool in `~VulkanExample()`, including but not limited to compute pipelines of `histogram`, `cdfScan` and `applyhisto`. You should destroy the storage buffer as well.

10. You should use the `saveScreenshot()` to generate screen shot image as your output, which will be used to compare with the reference. You may run the following in the command prompt:

```
c:> VulkanApp.exe -b -bw -br 1 -bf perf-result.tx -sf lena.ktx
```

Rubrics

This assignment will be graded over 100 points. Here is the breakdown:

- If your code fails to compile, zero is awarded immediately.
- If you did not use shared memory, atomic operation, histogram and scan to implement the histogram equalization, zero is awarded immediately.
- Comments/Code Readability. Comments are important for the code to enhance readability. Up to 10 points will be deducted for insufficient comments.
- Correctness. You must ensure that the code works for images of all sizes. At least pass the test for 512×512 picture. Pictures with different sizes could be used in the evaluation.
- Latency. Frame rate will be tested for the given samples. 30 points are allocated for performance evaluation.