Local Contrast Enhancement (Histogram Equalization)

0. Original Image

Continuing from the HDR image, we flattened the RGB image into gray scale and perform local contrast enhancement on such image.

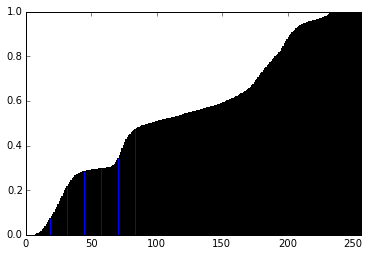
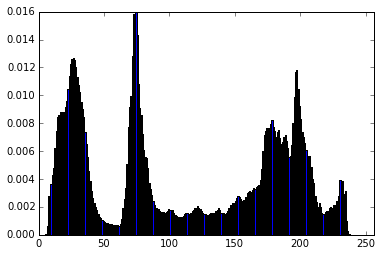
The picture below was taken on October 6, 2014, in Hiawatha national forest on a trip to upper Michigan. Clear in the original image, the sky and the lake are bright, backlighting the forest and the wood. As the image is already enhanced by HDR, the contrast has in fact been already balanced through combining multiple exposures. We want to further enhance the contrast, to show better the overexposed and underexposed details, or create artistic effects like sketching or engraving.

The upper part of the figure shows the original image (3648\*2432, about 9 megapixels) and small one shrunk to 1/12\*12 (304\*203, about 61k pixels) for benchmarking. The lower part shows the histogram of the original image.

We show here both the normalized histogram as well as cumulative histogram. Histogram has 256 bins ranging from 0 to 255. We wrote two methods to calculate and plot the histogram using for loops vs. calling numpy hist and interp, and we see the magic of using numpy increases the speed almost 10x.

Original (3648\*2432) vs small (304\*203):





Histogram plotting: 3100 ms

Histogram plotting numpy: 324 ms

Cumulative histogram plotting: 4760 s

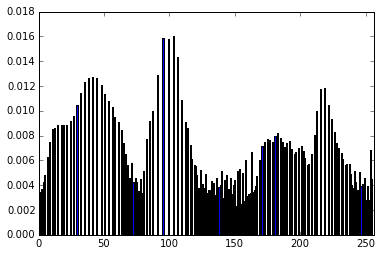
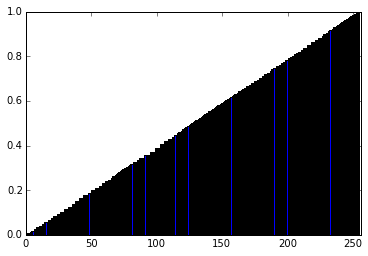
Cumulative histogram plotting numpy: 427 ms

1. HE serial (Histogram Equalization)

(Complexity: O(m\*n))

Histogram equalization (HE) is a commonly used technique to balance the pixel intensities in the image to enhance contrast. On the first stage, we loop over all pixels and obtain the cumulative distribution function of pixels in the picture. From that we create a lookup table through the inverse pixel intensity mapping. On the second stage, we loop over all the pixels again and map each pixel to its equalized value. Eventually, we see a much more balanced image with clear details of forest, lake, log and sky. After HE, cumulative distribution function shows a linear increase. The HE serial code takes 3.4 seconds.



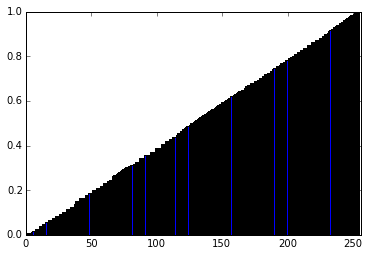
HE serial: 3400 ms

HE serial (on small picture): 39.4 ms

2. HE numpy

(Complexity: O(m\*n))

As a large part of numpy is written in Cython with no GIL, it is optimized in parallel. We tested similar code but calling numpy functions like interp. The time taken is 360 ms - 8.6 times faster than serial.

HE numpy: 360 ms

HE numpy (on small picture): 3.25 ms

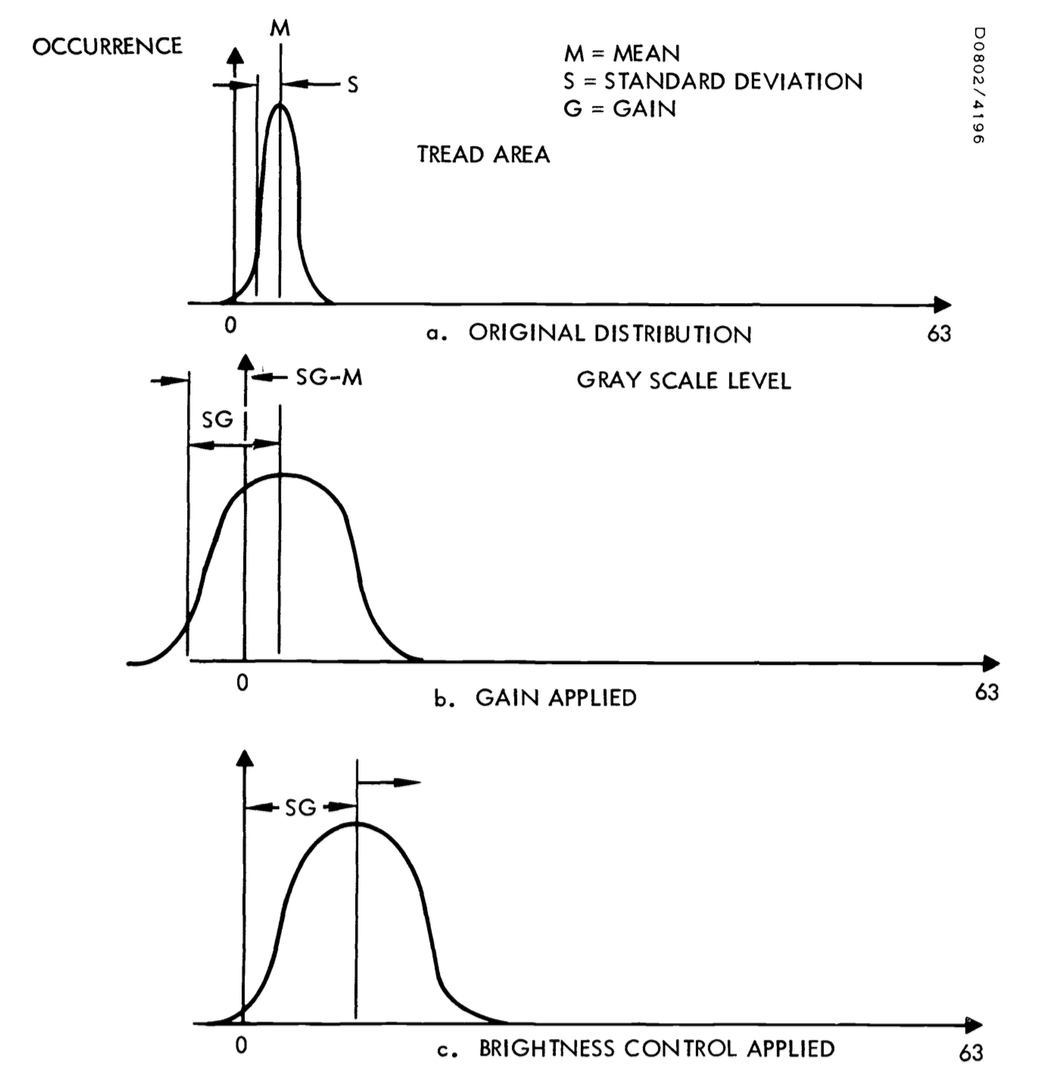
3. LHE serial (Local Histogram Equalization)

(Complexity: O(m\*n\*w\*w))

As we saw HE generated fairly satisfactory outcome of contrast enhancement. We moved on to Local Histogram Equalization (LHE) to emphasize the local contrast. We first looked at the method proposed in:

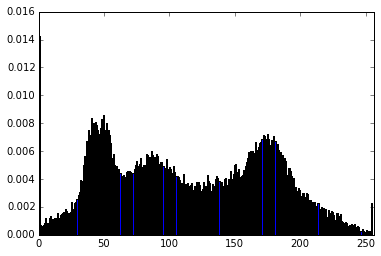
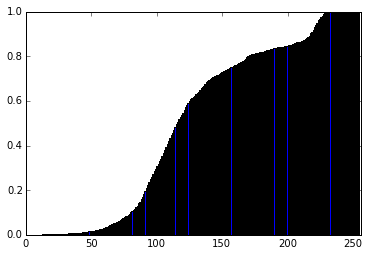
D. J. Ketcham, R. W. Lowe, and J. W. Weber, Real-time image enhancement techniques, in Seminar on Image Processing, Hughes Aircraft, Pacific Grove, California, 1976, pp. 1-6.

In this paper, the author proposed to focus on a window surrounding the pixel and calculate the mean and standard deviation of the pixel intensities. Then transformation (enlarging and shifting) is performed on the center pixel so as to yield a large distribution spread, as shown below.



At first look, it looks to be a sound way to enhance contrast. However, after realizing it on the image (small), the effect is not satisfactory: as much as some details are clearer (trees and clouds), there are apparently noises and some part of the image is unnatural. This is because the pixel histogram is not necessarily unimodal: at the edges there will be a large variance and shifting it either way may result in pure bright or dark pixels. Therefore, even though from the histogram it looks to be enhanced, we decided not to use this method. The time it takes is 2790 ms on the small picture, >5 minutes on the large.



LHE serial: > 5 min

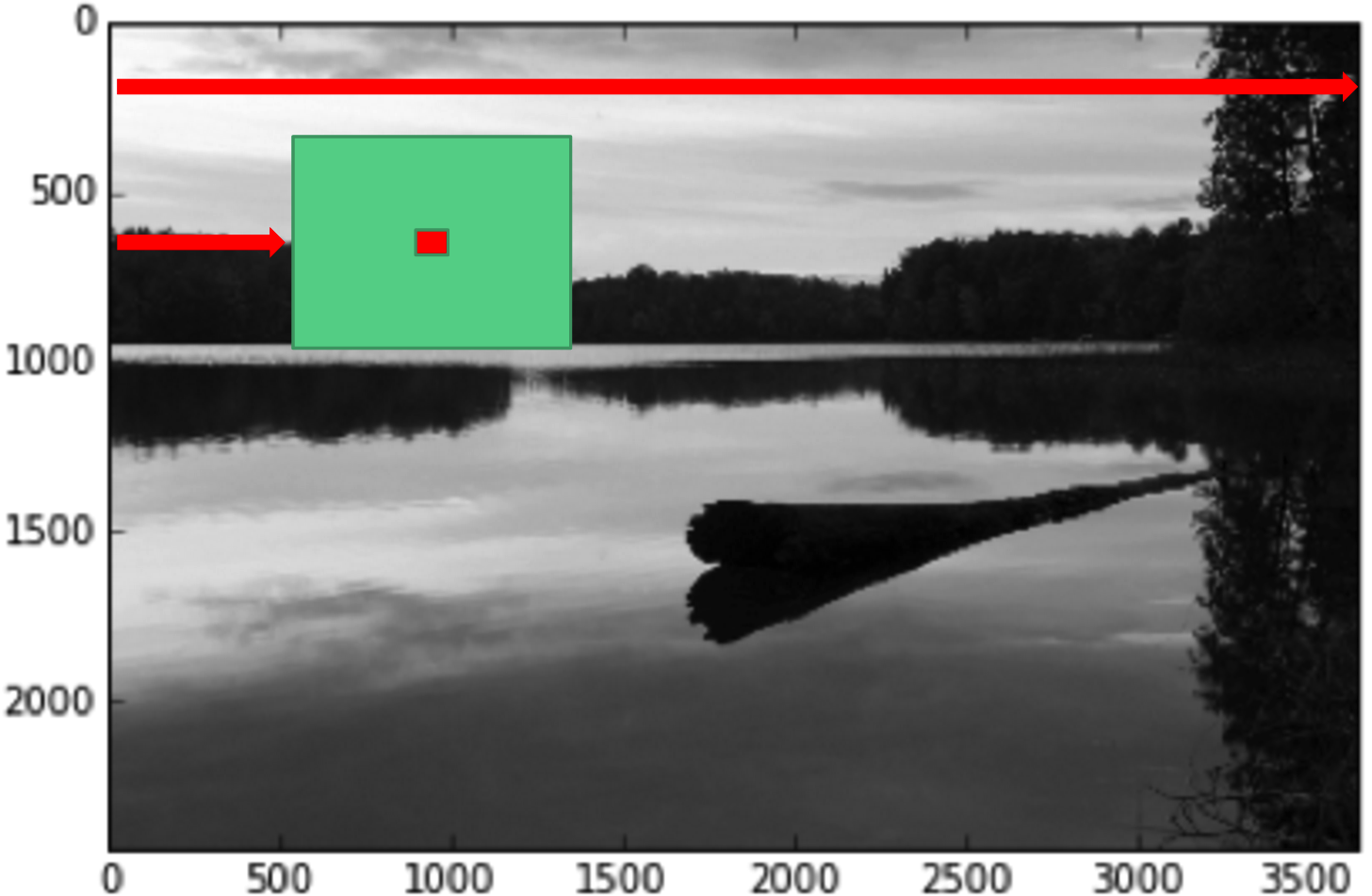
LHE serial (on small picture): 2790 ms

4. AHE serial (Adaptive Histogram Equalization)

(Complexity: O(m\*n\*w\*w))

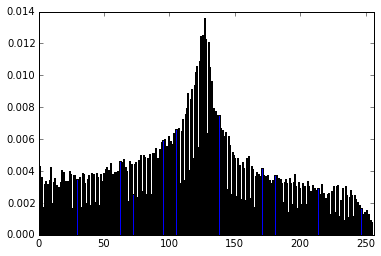
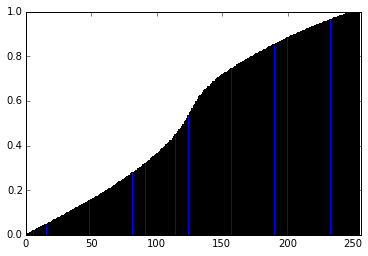
This paper coming one year after the LHE paper suggests an adaptive histogram equalization method. Looping over all pixels, for each, it looks at the window surrounding it, and calculate the rank of the center pixel intensity within the window and mapping it into the right level.

R. A. Hummel, Image enhancement by histogram transformation, Comput. GraphicsImage Process. 6, 1977, 184-195.



The resulting image shows very clear local contrast improvement and yields an artistic effect. The histogram is also satisfactory. However, this serial method runs slow – 10 seconds on a small image. This is also not efficient on large size images as large images ask for much larger window, increasing the time in quadratic.



AHE serial: > 5 min

AHE serial (on small picture , windowsize=21): 10500 ms

5. AHE opencl with buffer

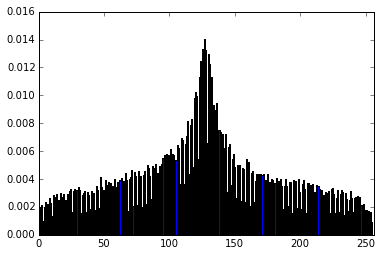
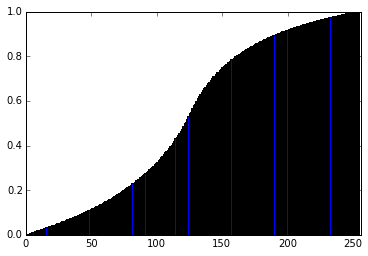
(Complexity: O(m\*n\*w\*w))

Migrating the serial AHE code into parallel, using opencl and buffering all pixels in the moving window into local memory, we are able to transform with the help of GPU parallelism.

The improvement seems to be drastic – over 2000 times!

On large image, the outcome image is not satisfying due to a small window size (same as mentioned above, we will optimize in next section). We performed this AHE on a small image so as to compare with section 4 of small image.



AHE opencl with buffer: 390.71936 ms

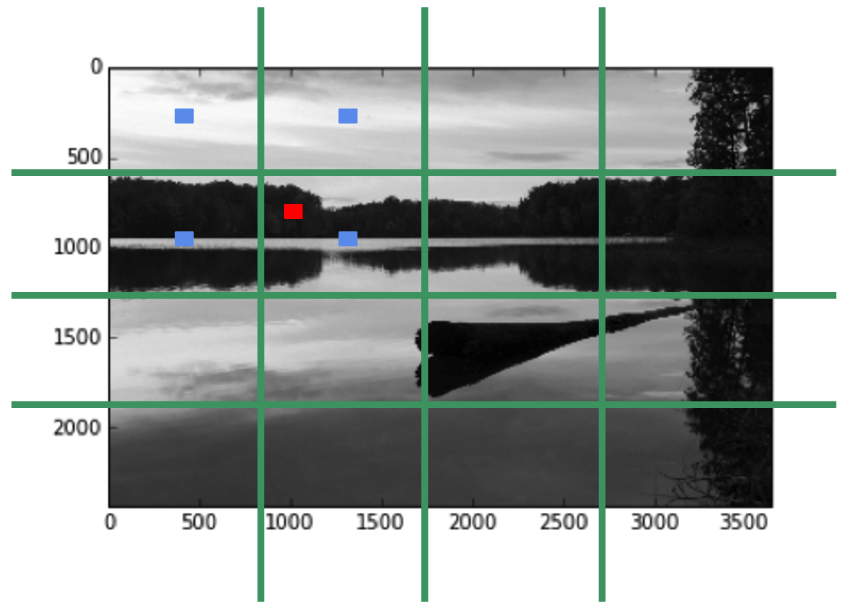
AHE opencl with buffer (on small picture with windowsize=21): 3.96824 ms

6. AHE opencl interpolation optimization

(Complexity: O(m\*n), atomic\_inc for histogram calculation)

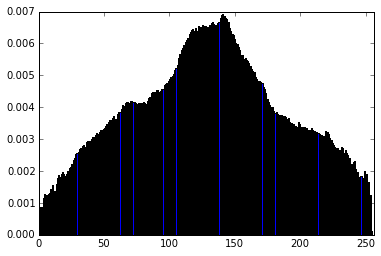
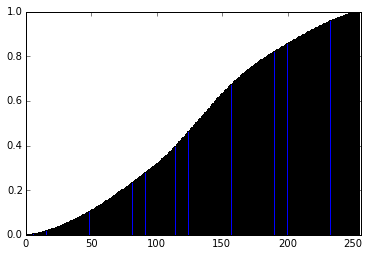
In 1987, Pizer *et al.* proposed an algorithm that improved the speed drastically through bilinear interpolation. The general idea is to first slice the image into segmentations and calculate the mapping histogram for all pixels in each segment, then loop over all pixels and find the mapping intensity in the four neighboring histograms and calculate the weighted average.

S. M. Pizer, E. P. Amburn, J. D. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. H. Romeny, J. B. Zimmerman, and K. Zuiderveld, Adaptive histogram equalization and its variations, *Comput. Vision Graphics Image Process.* **39**, 1987, 355–368.



The resulting image is very much like the one we obtained using not optimized AHE. The histogram also looks very satisfying. We timed the two stages individually – histogram calculation and pixel transformation and the total run time has decreased by factor of 8 on the small image.



AHE opencl interpolation optimization:

1. histogram calculation: 201.05504 ms

2. transformation: 7.4756 ms

3. total: 208.53064 ms

AHE opencl interpolation optimization (on small picture):

1. histogram calculation: 0.41576 ms

2. transformation: 0.07736 ms

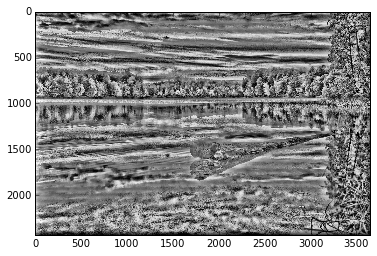
3. total: 0.49312 ms

We further explore the effects of changing how much we slice our image into, where segmentation size indicates the number of slices on each side:

Segmentation size = 1, 3, 6, 12, 24, 36:







Segmentation size =1 is simply global HE. As it gets larger, the effect of local contrast enhancement gets more drastic.

Another interesting observation is that as segmentation size increases, the speed actually becomes faster. This is because histogram calculation (the first stage) takes up the most time (a result of atomic increase, a serial function), and more segments means smaller histogram and more likely to parallelize this histogram calculation.

Total time:

Segmentation=2: 272.64904 ms (265.37424 + 7.552)

Segmentation=3: 264.61312 ms (255.14872 + 7.67632)

Segmentation=6: 248.44032 ms (235.45944 + 7.57376)

Segmentation=12: 208.53064 ms (202.96056 + 8.07872)

Segmentation=24: 140.31648 ms (135.84248 + 7.6752)

Segmentation=36: 121.5244 ms (110.9908 + 7.81456)

7. AHE opencl interpolation optimization with buffer

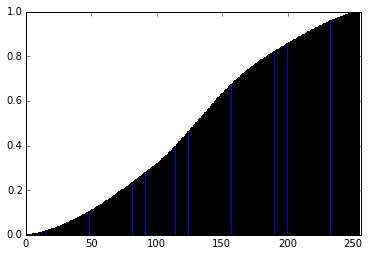
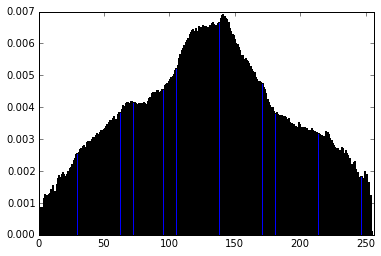
We further added buffer as a way to increase speed. We loaded a local buffer of size 256\*4 to store all the neighboring four histogram information of the work group, and expected an increase in the already fast transformation speed. The result is unfortunately not as we expected: the time used for transformation is 514.5 ms comparing to 7.6 ms (on a mac with intel iris). We think there are two reasons:

1) For every work group, we need to read from the global memory the huge histogram information and store it in our local buffer. This takes a lot of time accessing the global memory 1024 times for each work group.

2) For our 8\*8 local worker group, every pixel access the buffered histogram only four times grabbing the corresponding mapped values. The majority of the buffered values then remain untouched, let alone repetitive accessing.

Given large overhead buffering (both size and time) and low usage, no wonder it is much slower than directly accessing global memory.





AHE opencl interpolation optimization with buffer:

1. histogram calculation: 203.61952 ms

2. transformation: 514.46936 ms

3. total: 718.08888 ms

AHE opencl interpolation optimization with buffer (on small picture):

1. histogram calculation: 0.42904 ms

2. transformation: 3.30296 ms

3. total: 3.732 ms

8. AHE opencl interpolation optimization with buffer of data type uchar (256 bins vs 32 bins)

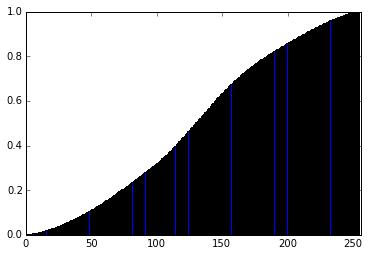
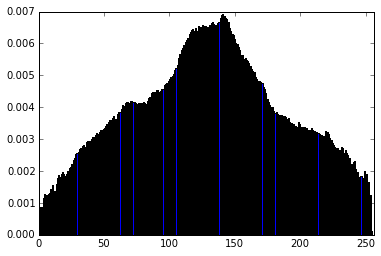
(Complexity: O(m\*n))

Building upon the “already much slower” buffer optimization, we further refine the buffer data structure from float to uchar (unsigned int 8 bytes) and this decreases the data storage by four time. Indeed, the transformation time decreases from 514 to 303 ms, by 40%.

If instead of having 256 bins, we coarse grain the bin size into 32, this further decreases the buffer size by 4 times, at the cost of loss of information and non-gradual pixel intensity range. As expected, the time now is only 166 ms, down by almost 50% from the 256 bins case.

Part 1: Still 256 bin histogram





AHE opencl interpolation optimization with buffer of uchar (256 bins):

1. histogram calculation: 191.29048 ms

2. transformation: 302.83328 ms

3. total: 494.12376 ms

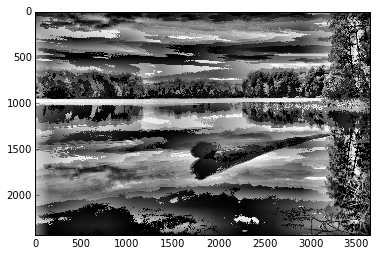
AHE opencl interpolation optimization with buffer of uchar (on small picture):

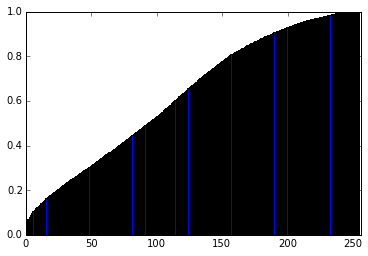
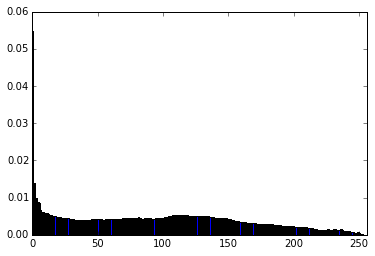
1. histogram calculation: 0.4096 ms

2. transformation: 2.1324 ms

3. total: 2.542 ms

Part 2: 32 bin histogram





AHE opencl interpolation optimization with buffer of uchar (32 bins):

1. histogram calculation: 197.19608 ms

2. transformation: 166.97552 ms

3. total: 364.1716 ms

AHE opencl interpolation optimization with buffer of uchar (32 bins, on small picture):

1. histogram calculation: 0.41928 ms

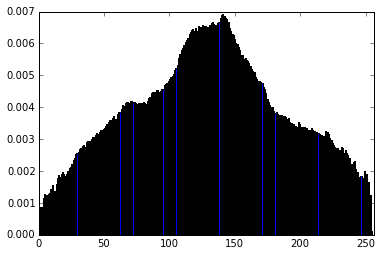
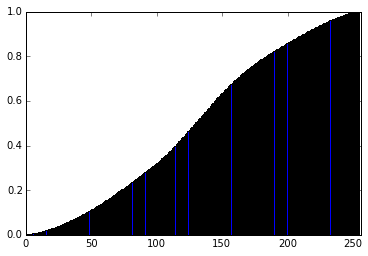
2. transformation: 1.60344 ms

3. total: 2.02272 ms

9. AHE opencl interpolation optimization with buffer of data type uchar avoiding bank conflict

Building upon the “already much slower” buffer optimization and improved data structure (256 bins), we improve our algorithm by avoiding bank conflict. We added a redundant column and thereby shifting the simultaneous access to the same bank. Again due to the small access times, the increase in speed is not significant – around 5%.



AHE opencl interpolation optimization with buffer of uchar (256 bins) avoiding bank conflict:

1. histogram calculation: 203.91328 ms

2. transformation: 289.16208 ms

3. total: 493.07536 ms

AHE opencl interpolation optimization with buffer of uchar avoiding bank conflict (on small picture):

1. histogram calculation: 0.43032 ms

2. transformation: 2.12488 ms

3. total: 2.5552 ms

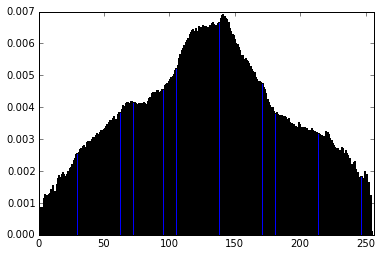
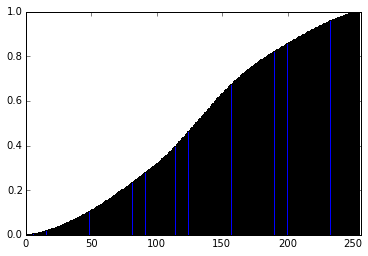
10. AHE opencl interpolation optimization with buffer of data type uchar avoiding bank conflict, modified:

As our last effort, building upon the “already much slower” buffer optimization and improved data structure (256 bins), we changed:

1) The local group size from 8 by 8 to 16 by 32 (max attainable on my machine), closer to a 3 by 4 image size and allowing more readings from the buffer.

2) Instead of reading all histogram into buffer when using the first thread, I distribute the work to all workers in the group. In a size 512 (16 by 32), each worker is responsible for two of the 1024 mapping histogram buffering. This greatly increases the speed by factor of 2.



AHE opencl interpolation optimization with buffer of uchar (256 bins) avoiding bank conflict, modified:

1. histogram calculation: 236.32184 ms

2. transformation: 163.53752 ms

3. total: 399.85936 ms

AHE opencl interpolation optimization with buffer of uchar avoiding bank conflict, modified (on small picture):

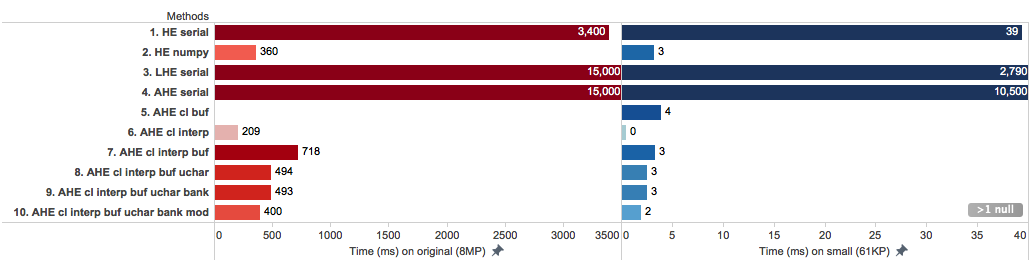
1. histogram calculation: 0.42768 ms

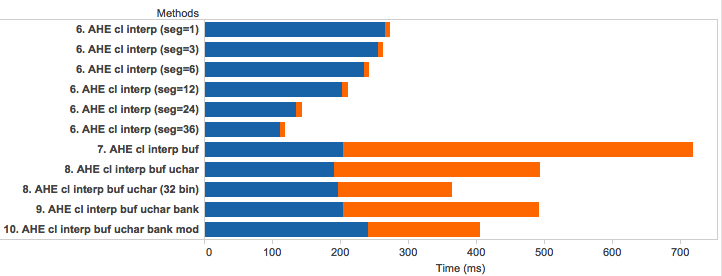
2. transformation: 1.51344 ms

3. total: 1.94112 ms

11. Summary

To enhance the local contrast, we used histogram equalization. We started from simple global HE, then speeded it up by calling numpy. Going from global to local (to have artistic effect), we first applied LHE, but the outcome image quality is not satisfactory. Therefore we moved on to AHE, pixel by pixel, but it runs too slow on a large image in serial. We ran it on opencl with buffer – the speed increased a lot. We further optimized the algorithm using interpolation (more than 10,000 times than serial), and found this to be the fastest. In an effort to make it faster, we added buffer and found it to be impeding the speed. Further changing the data structure and avoiding bank conflict and grabbing simultaneously to optimize the buffer, the speed has increased a lot comparing to the one with buffer.



Macintosh HD:Users:haosutang:Dropbox:CS205:project:working_notebooks:Tang:result:hist_trans_legend.png