Image Histogram along A Direction with CUDA

CS677 Final Project Report   Ying Liu

1. Problem description

When we have a 2D image of a measurement result, for example, pressure or temperature distribution on a surface or a CT image for medical purpose, other than the histogram of the whole image, sometimes we would like to see how the data is distributed horizontally or vertically, or even along a certain line we drew on the picture. Thus, getting histogram along a direction is useful in many applications.

For histogram along a line, the line can be decided by the coordinates of two points (Fig 1).

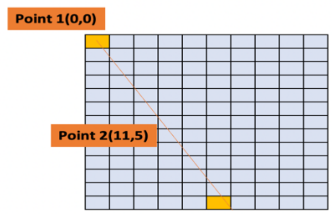


Fig. 1

To produce luminance histogram of an image along this line, we can rotate the image first with an angle, then calculate the histogram column wisely. The output result will be a 2D matrix with bin_Num * WIDTH elements. This WIDTH will be the width after rotation. This 2D matrix contains the position information along the given direction. The pseudo code is as follows:

for pixel along the col:

Histogram[luminance(pixel)] [position] ++

After some calculation, we can get the histogram at this line after rotation, which will be just a column of the final result.

2. Suitability for GPU acceleration

i. Image rotation is a natural fit for GPU acceleration. Each thread rotates one pixel, there is no data dependency among threads.
ii. Image histogram can also be paralleled processed by GPU either by each thread deals with one pixel or each thread gathers data for one bin.
iii. Amdahl's Law: close to 100%: After image padding, no extra processing on CPU.

3. Content

There are 2 steps for the GPU code implementation: rotation and histogram.

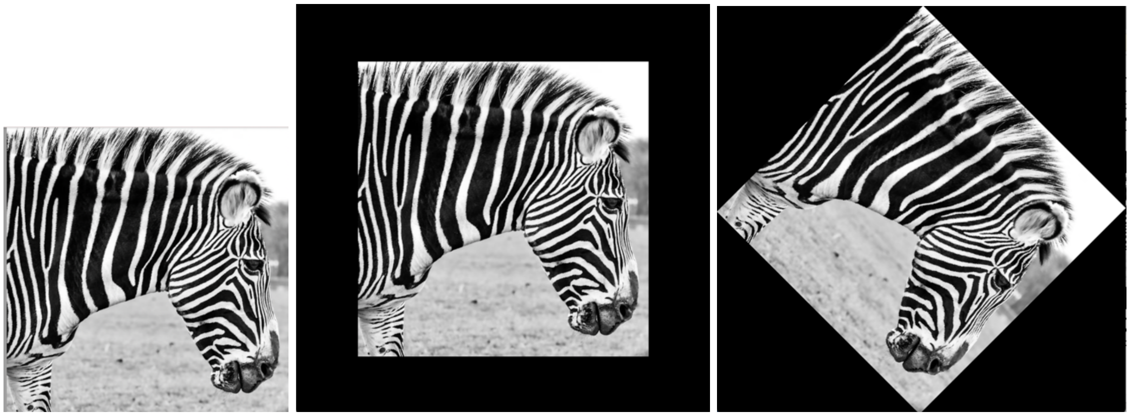
1) Image Rotation

The new coordinates (x2, y2) of (x1, y1) when rotated by an angle θ around origin point (x0, y0) is:

$$x2= \cos(\theta) * (x1 - x0) - \sin(\theta)*(y1 - y0) + x0$$

$$y2= \sin(\theta) * (x1 - x0) + \cos(\theta)*(y1 - y0) + x0$$

For example, if the given points are (0, 0) and (1, 1), the rotate θ is 45 degree. I choose to rotate along the center of the image as given angle, like a swirl. In order to keep the original image non-cropped, I paddled the image to the size of its diagonal size then rotate it. In this case, no matter rotating the image to any angle, it will keep all the original image safe and sound. Three diagrams in Fig. 2 show the steps to rotate an image 45°clockwise.



Original Paddled Rotated 45°

Fig. 2 (Original Image Source: Unsplash)

Since there is no data sharing or write conflicts between each pixels, in the kernel, it launches as many threads as long as the number of threads can cover the pixels number. Here are the block dimension and grid dimension of rotation kernel. The "diaLen" as mentioned before, is the width/height of the paddled image:

int blockSize = 32;
dim3 blockDim(blockSize,blockSize,1);
int gridSize = (diaLen + blockSize -1)/blockSize;
dim3 gridDim(gridSize,gridSize,1);

I implemented 3 versions of rotation kernel:

A. "rotation_kernel_naive": This kernel directly loads and stores data to the global memory.
B. "rotation_kernel_2": This kernel used more registers to save several precalculated values such as: xCenter, yCenter,sin(angle), cos(angle). These four values are repeatedly used by every thread.

```
//--------rotation kernel 2----------------------------------
__global__ void rotation_kernel_2(unsigned int *input,unsigned
  //TO DO
  double xCenter = (double)width / 2;
  double yCenter = (double)height / 2;
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int index = y * width + x;
  double sinA = sin(angle);
  double cosA = cos(angle);
  double shiftX = (double)x - xCenter;
  double shiftY = (double)y - yCenter;
  int orgX = 0;
  int orgY = 0;
  //boundary check
  if(x >=0 && x < width && y >=0 && y < height){
    orgX = (int)(cosA * shiftX - sinA * shiftY + xCenter);
    orgY = (int)(sinA * shiftX + cosA * shiftY + yCenter);
  }

  if(orgX>=0 && orgX < width && orgY>=0 && orgY<height){
    output[index] = input[ orgY * width + orgX];
  }

}
```

Fig. 3 Snapshot of "rotation_kernel_2"

C. "rotation_kernel_3": This kernel used constant memory to save those four values mentioned in "rotation_kernel_2".

```
double *P;
P = (double*) malloc(4 * sizeof(double));
P[0] = (double)diaLen / 2; //xCenter
P[1] = (double)diaLen / 2; //yCenter
P[2] = sin(angle);
P[3] = cos(angle);

//load data to constant memory
cudaMemcpyToSymbol(PARAMS, P, 4 * sizeof(double));
```

Fig. 4 Snapshot of Constant Memory Kernel Code

Comparison between CPU rotation and GPU naïve kernel on different size of Images is showed in Table 1. From the time consumption comparison, it is obvious that the GPU rotation speed up a lot more than the CPU. For the XLarge image, the GPU is almost 2000x faster than the CPU.

Time Unit: milliseconds	Small size (640x651 pixels)	Middle size (1920x1953 pixels)	Large Size (2400x2442 pixels)	Xlarge Size (3932 x 4000 pixels)
CPU-Rotation	57	473	777	2134
GPU-Rotation_Naive	0.033	0.278	0.4214	1.1178

Table 1

The GPU optimization comparison by different versions of kernels based on different image size is showed in Table 2 and Fig. 5. We can see that the best performance among these three kernels is using constant memory.

Time unit: milliseconds	**Naïve**	**Registers**	**Constant Memory**
Small size (640x651 pixels)	0.03351	0.01752	0.009472
Middle size (1920x1953 pixels)	0.2786	0.124	0.0853
Large Size (2400x2442 pixels)	0.4227	0.1884	0.1292
Xlarge Size (3932 x 4000 pixels)	1.0871	0.5	0.3313

Table 2

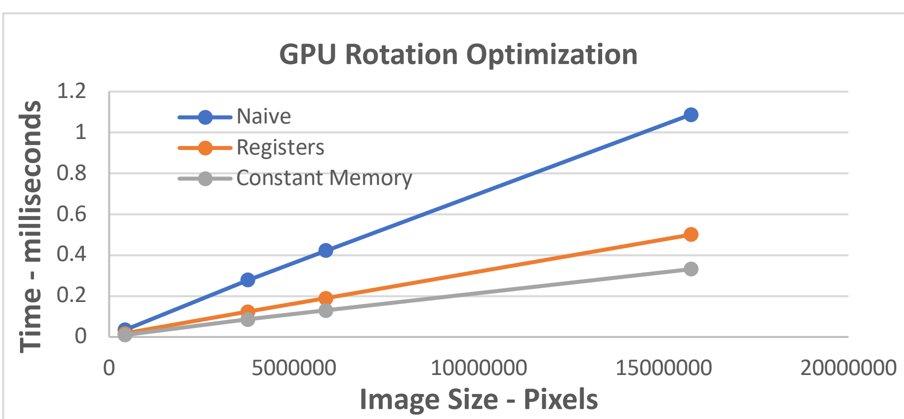


Fig. 5

2) Image Histogram

After paddling and rotation, we get a square image with width and height both equal to the original image's diagonal length, noted as "diaLen". Now by calculating the histogram of every column, we get a 2D matrix with height equals the number of bins and width equals to "diaLen". I implemented 2 versions of histogram kernel: naïve one and the one using shared memory.

A. "histo_kernel_naive": Here are the block dimension and grid dimension of the naïve histogram kernel:

int blockSize = 32;
dim3 blockDim(blockSize,blockSize,1);
int gridSize = (diaLen + blockSize -1)/blockSize;
dim3 gridDim(gridSize,gridSize,1);

The naïve version directly loads and store using global memory. It uses "atomicAdd" method to deal with race condition. The row of the "histo" will be the gray scale value of the input and the column will be same as the image column. Fig. 6 is a snapshot of kernel:

```
__global__ void histo_kernel_naive(unsigned int *input,unsigned int *histo,int wid

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

if(x >=0 && x < width && y >=0 && y < height){
    atomicAdd(&histo[input[y*width+x]*width+x],1);
  }

}
```

Fig. 6 Snapshot of histo_kernel_naive

B. *"histo_kernel_2":* In the naïve kernel, the "histo" global parameter has be accessed multiple times by each thread to do the accumulation, thus using shared memory can decrease the times of read and write global memory, which is a good choice to decrease the time consuming. So, in "histo_kernel_2", shared memory H[NUM_BINS] is used for storing one column of the final result. When the accumulation is done, each thread writes to global memory once. For this kernel the threads number launched equals to the histogram result element number:

histo_kernel_2<<<diaLen,256>>>(d_rotated,d_histo2,diaLen,diaLen);

Here is a snapshot of this kernel:

```
__global__ void histo_kernel_2(unsigned int *input,unsigned int *hist
 __shared__ unsigned int H[NUM_BINS];// one column of the output bin
 int tx = threadIdx.x;
 int col = blockIdx.x;//col number of input
 int bd = blockDim.x;
  H[tx] = 0;
 __syncthreads();
  for(int t =0;t<(height+bd-1)/bd;t++){ //when height is bigger than
     int row = t*bd+tx;
     if(row<height){
       int value = input[row*width+col];
        atomicAdd(&H[value],1);
     }
  }
  __syncthreads();
    histo[tx*width+col] = H[tx];
}
```

Fig. 7 Snapshot of histo_kernel_2

Since threads number in one block could be smaller than the height of the image, there is a loop in the kernel to read all the sessions of one column of the image and does the “atomicAdd”. When it is done, each block writes one column of the final result.

Comparison between CPU histogram and GPU naïve kernel is showed in Table 3. We can see that for the XLarge size image, even using GPU naïve version can get almost 190x faster speed.

(milliseconds)	Small (640x651)	Middle (1920x1953)	Large (2400x2442)	Xlarge (3932 x 4000)
CPU-Histogram	4.86	33	43	116
GPU-Histogram_Naive	0.0224	0.1471	0.2159	0.5909

Table 3

GPU optimization comparison on different version of kernels based on different size of Images is showed in Table 4 and Fig. 8.

Time unit: milliseconds	Naïve	shared memory
Small size (640x651 pixels)	0.0225	0.0164
Middle size (1920x1953 pixels)	0.1471	0.0732
Large Size (2400x2442 pixels)	0.2159	0.1066
Xlarge Size (3932 x 4000 pixels)	0.5908	0.2579

Table 4

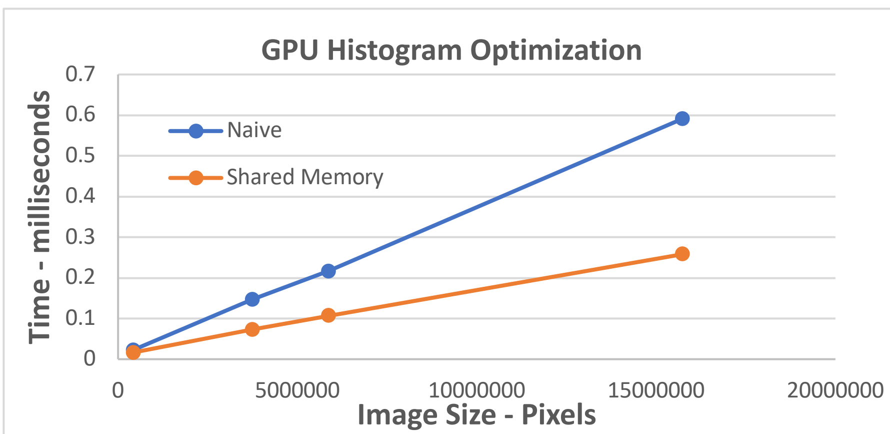


Fig. 8

4. Instructions on running the project

All the kernels and main function are in one file. When running the project without arguments:

```
!./histo
```

The main function will do default 45 degree rotation clockwise and the default image “ small-zibra-unsplash.ppm” will be used. If one argument – the rotation angle be supplied, it will rotate the default image using the given angle, for example, rotating the image to 30 degree.

```
!./histo 30
```

For 2 args, the first one will be angle and the second one will the image file. For example, the below command line will rotate the image “mid-zibra-unsplash.ppm” to 30 degree and then do the histogram. The histogram result will be in a “.csv” file.

```
!./histo 30 mid-zibra-unsplash.ppm
```

5. Track the line

Remember the arbitrary line we’d like to know the histogram along it? Now we have the histogram result, by using the method “getNewXY” in the project we can have the answer. For example, we have the start point (0,0) and end point (4,4) and the angle we get is 45 degree clockwise. After rotation we use this “getNewXY” function we get the (newX, newY) is (461,1). The 462nd column of the result is the histogram along this line. For other coordinates, just modify the 96th and 97th line of code to modify the coordinates.

6. Conclusion

This project produces histogram on an image along a direction by firstly rotating the image to desired angel and then do histogram column wisely on the rotated image. By applying parallel programming on CUDA, the program gained very satisfying speed up. Optimization methods used during the GPU implementation include using registers, using constant memory and using shared memory.