GPU-Accelerated Motion Blur Detection on Mobile Phone

Ronnachai Jaroensri
Department of Electrical Engineering
Stanford University
Stanford, CA
tiam@stanford.edu

***Abstract*—A technique to quickly detect motion blur from mobile phone camera and its implementation on an iPhone 4S were presented. GPU was utilized to accelerate the implementation. Direction of motion relative to the image space was used in addition to the image itself to detect motion blur. The accuracy of detection was 75%, and computation took on average 0.73 seconds for 8MP image on an iPhone 4S. Error in direction of motion and memory management for hi-resolution were the main unresolved issues for this implementation.**

Keywords-Motion Blur; Blur Detection; GPU filtering;

I. INTRODUCTION

Camera has been an integral part of mobile phone today. The quality of the pictures from mobile phone camera is now comparable to small compact camera too. However, due to its small size, it is hard to hold the camera steady. Most camera phones also only operate in automatic mode, preventing user's access to advanced controls of the camera. Therefore, it is relatively challenging for the user to get high quality image in low light situation. Moreover, the screen is usually too small to display the full resolution of the image, so the user usually do not notice the motion blur due to camera shake until they come back to the image later, and already miss the photographic opportunity.

Most existing techniques of motion blur detection relied on a single image, while many phones today are equipped with gyroscope and/or accelerometer. In this paper, we will demonstrate how we can use this additional information of the motion of the phone to help quickly determined if the image taken by the camera is blurred or not. To accelerate our calculation further, we utilize the GPU available on the phone to do calculation, which are mostly pixel-wise operation. Our implementation was based on iPhone 4S.

II. RELATED WORK

Motion blur detection has been studied to a great extent. There are many existing techniques that are designed to detect space-invariant and space-variant blur. The latter type of blur is usually caused by the motion of objects in the image frame during long exposure time, whereas the former is usually the result of camera shake, which shifts the whole scene.

Transparency information (alpha channel) has been used to detect and correct for motion blur. In [1], the detection of both global and local motion blur was presented. The technique made use of the observation that, the gradient of blending channel between foreground and background tended to distribute themselves into lines perpendicular to the direction of motion. In [3], the alpha channel was utilized to estimate the blur kernel and was used for a high-quality reconstruction of the unblurred image. Unfortunately, techniques employing alpha-channel segmentation tended to have high memory footprint, and involve long computation time. The alpha channel extraction alone took a few seconds on a laptop computer for 1 Megapixel image, while the deblurring using alpha channel could take up to 490 MB of memory for 0.5 Megapixel image [4].

There had been other techniques that relied on observation of the gradient of the image, which could be computed very efficiently using GPU. In [5], the gradient magnitude distribution was considered. Local autocorrelation of the color in the image was also considered. Although the technique was not tailored specifically for motion blur, the detection accuracy was only around 65%. [6] proposed a low cost scheme to detect the blur. The average magnitude of the gradient, and the variance of it were considered. Although the accuracy of this method is quite high (86%), it did not utilize additional information about the device's motion, which might be useful to improve the accuracy of detection.

III. THE ALGORITHM

The smearing of features in the image will serve as important visual cue in our method, which can be described in 4 processes as follow:

1. Capture image with motion information
2. Calculate gradient in the direction of motion and the orthogonal direction
3. Sum squared-magnitude of the gradient in each direction and find the ratio between the directions of motion to the orthogonal direction
4. Compare this ratio to the training data.

From 50 test images (8 Megapixel each), we simulate motion blur of different directions and different blur length. The distributions of ratios for the blurred and unblurred image are shown in Figure 1.

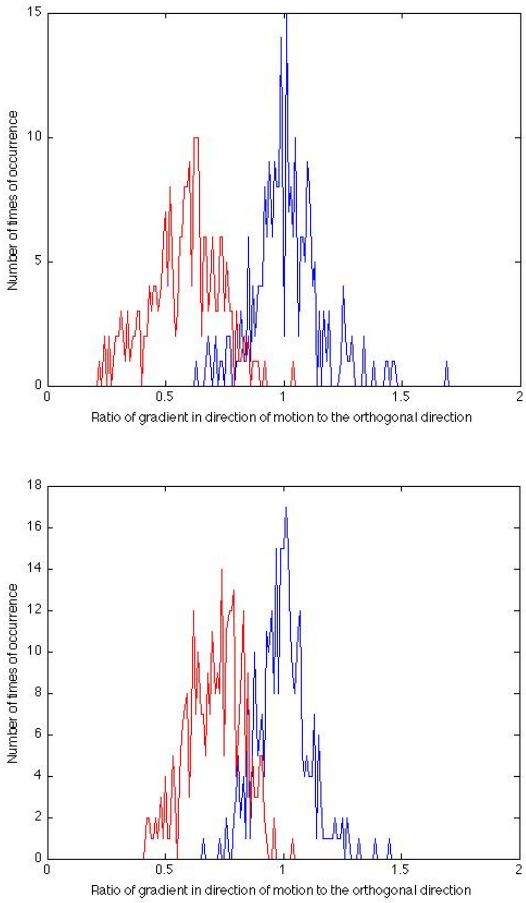

Figure 1 Training Data of more than 200 blurred images and 200 unblurred images. The red line represents the distribution of ratio from blurred images, and the blue line represents that of unblurred images. (top) simulated blur length randomly chosen between 15 and 150 pixels (bottom) the blur length is fixed to 15 pixels. The image size is 3456x2314. For the first case, the ratio for the two categories are quite exclusive. For the second case, where the blur is barely noticeable, the two peaks are closer together, but they are still distinguishable.

The blur distributions of the blurred and the unblurred images seem to be well-separated. If we can get an accurate motion information, this algorithm is likely to be accurate.

For specific details of the implementation, please see section IV below.

IV. The Implementation and Design Consideration

Because gradient calculation is a pixel-wise operation, we can use GPU to accelerate the calculation of gradient. We implemented our application on an Apple iPhone 4S, and we made use of a framework called GPUImage [7].

GPUImage is a programming framework that abstracts complicated OpenGL setup needed to perform calculation on GPU from the clients. The clients, however, will be able to customize the process by writing their own shader program using c-like Graphic Library Shader Language (GLSL). This shader program controls how GPU operates on each pixel. Note that GPU does have access to other pixels in the original image through texture look up, although there might be some performance issues to be considered when performing a texture look up [8]. Interested reader should find more information about GLSL in [9].

Figure 3 Gradient images from different direction. (a) the original blurred gray scale image (simulated), (b) the gradient image in the direction of motion, (c) the gradient image in the orthogonal direction. Note that the (c) is much brighter than (b). The calculate ratio of rms magnitude between them is 0.53.

GPUImage also provide a complete framework for capturing and exporting pictures in various format. Our implementation uses a `GPUImageStillCamera` class to capture the image as a `UIImage`, to be passed on to the calculation module. The four steps of our implementation are described on the next page.

A. Capture image with motion information

At start up, an instance of `CoreMotionManager` is initiated and set up in a push configuration. This instance of `CoreMotionManager` will push motion data to the application at 50 Hz. This motion information will be averaged when the user starts taking the picture, and the averaging stop immediately when the `GPUImageStillCamera` finishes taking picture (`GPUImageStillCamera` does some processing before its capture function returns, so we slightly modified its interface to allow it to send signal to `CoreMotionManager` to stop averaging data).

We approximate acceleration information as velocity as iPhone 4S does not have hardware to blindly detect velocity of the device[1]. Rotation data was not used due to limited time to do this project. Although rotation could directly give velocity information, we will need to calibrate the distance from the gyroscope to the camera to get an accurate calibration of contribution of rotation from each axis. This can be an important add-on in the future.

Since we will need the angle for gradient calculation, the angle is simply calculated by using principal arctangent:

$$\theta = \arctan(v_y/v_x). \quad (1)$$

Where v_y represents the averaged velocity in the y-direction, and v_x represents the averaged velocity in the x-direction. θ will fall in the range $[-\pi,\pi]$.

B. Calculate gradient in the direction of motion and the orthogonal direction

After the angle has been calculated, we calculate convolution kernel in each direction using the expression,

$$K_{//} = \cos(\theta)\, K_x + \sin(\theta)\, K_y \quad (2)$$

$$K_{\perp} = -\sin(\theta)\, K_x + \cos(\theta)\, K_y \quad (3)$$

Where $K_{//}$ and $K_{\perp}$ represents the convolution kernel in the direction of motion, and orthogonal direction respectively, and K_x and K_y represents convolution kernel for gradient in x-, and y-directions respectively, and are defined as:

$$K_x = \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, K_Y = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (4)$$

The factor of ¼ is to make sure that the pixel value do not saturate if the gradient is strong. Note that (2) and (3) are simply rotation of K_x and K_y, by the matrix:

$$R_\theta = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

By pre-computing the convolution kernel, our algorithm becomes more efficient as we do not have to perform rotation at each pixel again.

Since the output from the OpenGL frame buffer will always have 4 channels, we pack gradient calculations of both direction into one shader program, and encode the output in two of the color channels. This reduces the memory usage by half especially when we are calculating for the full 8MP image (we still are still wasting the other two channels, but GPUImage only allows this type of buffer for us). The gradient is squared on the GPU before the image is passed to the next step.

C. Sum squared-magnitude of the gradient in each direction and find the ratio between the directions of motion to the orthogonal direction

Since summation on the GPU is also faster than on CPU, we perform a summation using a 3x3 averaging convolution once before passing the information to the CPU. Note that it is possible to do the entire summation process on GPU by downsampling the image multiple times, but this feature is not yet supported by GPUImage. Due to the limited time constraint to do this project, we could not set up the appropriate OpenGL computing environment to perform this calculation, and have to rely on slower CPU.

D. Compare the ratio to the threshold

We then calculate the ratio between sums of the gradients in two directions, and compare it to the threshold calculated from our learning data, which is the probability distribution function of the ratio of ground truth blurred and unblurred images. The threshold is calculated in such a way to minimize the probability of error (the number of blurred images that are classified as not blur, plus the number of unblurred images that are classified as blur).

The speed and accuracy of the implementation will be discussed in the next section.

V. Experimental Result

To test this implementation, the application has a testing mode where it takes the picture, and predicts whether or not the image taken is blurred. The user then gives the application feedback whether its prediction is right or not.

From 57 test images taken on the device, the accuracy rate was approximately 75%. Most blur predictions were correct, but there seemed to be some structure that trigger false detection of blur. However, those detections seemed to have gradient ratio in the same range, therefore we believe that more training data on the device will improve the accuracy.

Moreover, the system seemed to be quite sensitive to direction of motion data. During the test, there were many instances where the device detected the direction wrong, resulting in false negative detection of blur. Any future implementation should consider improving the accuracy of direction of motion.

In addition to the iPhone implementation, the same algorithm was also implemented on MATLAB®. For an 8 MP image, MATLAB® takes 0.54 seconds on average to compute the gradient ratio on a 2.4 GHz Core2Duo laptop, while the iPhone 4S implementation takes approximately 0.73 second for 8MP image, and approximately 0.06 second for 640x480 frame, which allows it to run in real time. Table 1 summarizes the computing speed on various platforms.

[1] In fact, this seems rather impossible as one can only infer one's velocity by considering motion relative to other objects.

TABLE I. COMPUTATION SPEED VS PLATFORM

Platform	*Resolution*	*Average Computation Time (s)*
MATLAB® @2.4GHz Core2Duo	8MP	0.54
iPhone 4S	8MP	0.76
iPhone 4S	1080p	0.24
iPhone 4S	VGA	0.066

Table 2 shows a breakdown of computation time at each stage. The GPU computation seemed to be constant at all resolution: 20 – 40 ms, while at hi-resolution image, there was a time lag while the CPU was copying the image from the framebuffer of the GPU. If the summing step was implemented on the GPU, this implementation could be even faster.

TABLE II. COMPUTATION TIME BREAKDOWN ON IPHONE 4S

Resolution	*Stage*	*Average Time (s)*
8MP	GPU Processing	0.040
	Framebuffer Read	0.584
	CPU Processing	0.070
1080p	GPU Processing	0.027
	Framebuffer Read	0.124
	CPU Processing	0.018
VGA	GPU Processing	0.023
	Framebuffer Read	0.023
	CPU Processing	0.005

Even though the processing speed at 8MP was relatively fast, there were some memory problems that affected the stability of the application. This is because the output from the GPUImage buffer was always in the RGBA format, which took 4 bytes per pixel. For an 8 MP image, this will result in 32 MB of memory allocation.

VI. CONCLUSION

We have implemented a high-speed detection of motion blur using direction of motion information from the internal accelerometer and gyroscope. The accuracy of prediction was approximately 75%, while the computing speed was 0.73 seconds on average for an 8MP image. Error in direction of motion and memory management at high-resolution were the two main issues that need to be improved on in the future implementation.

ACKNOWLEDGMENT

We thank Derek Pang for all his guidance and advice through the project, both on image processing techniques and iOS-specific issues, Jeff Piersol for introducing us to GPUImage framework, which is the backbone of our implementation.

REFERENCES

[1] Shengyang Dai; Ying Wu; , "Motion from blur," *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on* , vol., no., pp.1-8, 23-28 June 2008
doi: 10.1109/CVPR.2008.4587582
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4587582&isnumber=4587335

[2] Levin, A.; Rav-Acha, A.; Lischinski, D.; , "Spectral Matting," *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on* , vol., no., pp.1-8, 17-22 June 2007
doi: 10.1109/CVPR.2007.383147
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4270172&isnumber=4269956

[3] Qi Shan, Jiaya Jia, and Aseem Agarwala. 2008. High-quality motion deblurring from a single image. In *ACM SIGGRAPH 2008 papers* (SIGGRAPH '08). ACM, New York, NY, USA, , Article 73 , 10 pages. DOI=10.1145/1399504.1360672 http://doi.acm.org/10.1145/1399504.1360672

[4] Qi Shan, Jiaya Jia, and Aseem Agarwala. (2012, June 1). *Single-image Deblurring (Motion PSF Estimation)* [Online] . Available: http://www.cse.cuhk.edu.hk/~leojia/programs/deblurring/deblurring.htm

[5] Renting Liu; Zhaorong Li; Jiaya Jia; , "Image partial blur detection and classification," *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on* , vol., no., pp.1-8, 23-28 June 2008
doi: 10.1109/CVPR.2008.4587465
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4587465&isnumber=4587335

[6] Jaeseung Ko; Changick Kim; , "Low cost blur image detection and estimation for mobile devices," *Advanced Communication Technology, 2009. ICACT 2009. 11th International Conference on* , vol.03, no., pp.1605-1610, 15-18 Feb. 2009
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4809380&isnumber=4809346

[7] Brad Larson. (2012, May 19). *Introducing the GPUImage Framework* [Online]. Available: http://www.sunsetlakesoftware.com/2012/02/12/introducing-gpuimage-framework

[8] (2012, May 20). *Best Practices for Shaders* [Online] Available: http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW3

[9] (2012, June 4). *OpenGL Shading Language* [Online] Available: http://www.opengl.org/documentation/glsl/