

GPU-ACCELERATED JOINT 1D AND 2D BARCODE LOCALIZATION ON SMARTPHONES

Gábor Sörös

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The built-in cameras and powerful processors have turned smartphones into ubiquitous barcode scanners. In smartphone-based barcode scanning, barcode localization is an important preprocessing step that quickly scans the entire camera image and passes barcode candidates to the actual decoder. This paper presents the implementation steps of a robust joint 1D and 2D barcode localization algorithm on the mobile GPU. The barcode probability maps are derived from the structure matrix and the color of the individual pixels. The different steps of the localization algorithm are formulated as OpenGL ES 2.0 fragment shaders and both 1D and 2D barcode saliency maps are computed directly on the graphics hardware. The presented method can detect barcodes at various scales and orientations at 6 frames per second in HD resolution images on current generation smartphones.

Index Terms— barcode, QR code, localization, smartphone, GPGPU

1. INTRODUCTION

Thanks to their built-in cameras and powerful processors, smartphones are increasingly used for barcode scanning in both consumer and enterprise domains. Example applications include comparing prices, reading product reviews, checking allergy information, assisting visually impaired users, or tracking assets within a company. As the resolution of mobile cameras increases, scanning smaller, distant, or even multiple codes might soon become possible which would lead to a better user experience. With the upcoming smart glasses having smartphone-equivalent hardware, even wearable barcode scanning could be realized in the close future. Recent decoder applications like RedLaser¹ and Scandit² can also decode fairly blurry codes before the camera has focused on the code. However, the computational complexity of blurry decoding limits practical decoder algorithms to a small search window and requires the user to align the code close to the camera. Most algorithms have to assume a limited scale range of codes that fit the search window, and often also a limited set of orientations. To enable barcode scanning in a wider

range and at any orientation, a barcode localization algorithm is necessary that quickly searches the entire image for possible barcode candidates and passes those cut-out regions to the actual decoder.

The fast and robust localization of barcodes in digital images has been an active research area for the past decade. Previous approaches can be categorized by their image processing techniques. 1D localization has been presented using simple image filters [1, 2, 3], orientation histograms [4], line detection [5], morphology operators [6, 7], Gabor filters [8], or harmonic analysis [9, 10]. 2D localization has been implemented via image thresholding and scanning, via orientation maps, or via combinations of line/corner/square detection [11, 12]. The localization algorithms usually need to trade accuracy and robustness for speed and make certain assumptions about code orientation, code scale, or code symbology. Existing algorithms often have difficulties with defocused or motion-blurred images because blur distorts the barcode structures. The author has recently published [3] a new, combined localization algorithm that addresses orientation, scale, symbology and blur invariance and runs close to real time on a smartphone CPU.

This paper presents the implementation steps of the above barcode localization algorithm on the embedded graphics hardware. The algorithm calculates both 1D and 2D barcode probability maps over the entire image using the structure matrix and the color of the pixels. By reformulating the distinct steps as fragment shader programs, one can compute the barcode probability maps directly on the graphics hardware which brings significant speedup (e.g., of factor 3.2 in our tests with 4 shader cores). The presented GPU-assisted approach allows real-time operation without the limiting assumptions of the concurrent methods.

The paper is organized as follows: Section 2 introduces image processing on the graphics hardware in general, Section 3 gives a brief overview of the selected barcode localization algorithm and Section 4 presents how to implement the algorithm on the GPU with various optimizations. Section 5 presents the experimental results while Section 6 concludes the paper.

¹<http://www.redlaser.com>

²<http://www.scandit.com>

2. IMAGE PROCESSING ON MOBILE GPU

OpenGL ES is the de facto standard software-hardware interface for rendering 3D graphics on embedded devices. Version 2.0 of OpenGL ES introduced the programmable graphics pipeline to mobile graphics hardware opening the doors for general purpose GPU computations on smartphones. There are certain limitations though compared to proprietary high-level parallel computing frameworks such as CUDA³, OpenCL⁴, or RenderScript⁵ because OpenGL was primarily designed for rendering 3D scenes. This means one has to reformulate computing problems as rendering problems and carefully tune the algorithms to graphics features. Nevertheless, OpenGL ES provides portability across all smartphone platforms and version 2.0 is widely available already in middle-class consumer devices.

The programmable graphics pipeline consists of several steps as shown in Figure 1. The 3D scene is defined as a set of triangles formed by vertices. The vertices have various attributes such as 3D coordinates, normal vectors, primary color, texture coordinates, etc., that are passed to a programmable SIMD processor called vertex shader. The vertex shader program calculates the screen coordinates of each vertex and passes those to the primitive assembly unit that determines the points, lines, and triangles to be rendered on the screen. The rasterizer decomposes the primitives into individual fragments and for each fragment it interpolates the attributes from those of the corresponding vertices. Each fragment is passed to another programmable SIMD processor called fragment shader which calculates the color of the fragment based on the attributes and optional texture inputs. Textures are general 2D data containers with hardware-accelerated interpolation. The individual fragments run through several tests until they are finally combined in a pixel of the frame buffer. When all primitives are processed, the content of the frame buffer gets presented on the screen. OpenGL ES 2.0 also features off-screen render targets called frame buffer objects (FBOs). Using an FBO, the output can be directed into a texture.

Image filtering can be reformulated as a multi-pass rendering problem in the following setting (see Figure 2): The input image is stored in a texture and an output FBO is set up to have the same size as the input texture (1-to-1 mapping between texels and pixels). The 3D scene consists of only two triangles (a quad) that together cover the whole screen (or FBO). The virtual camera looks fronto-parallel to this quad and orthographic projection is applied. Once the scene is drawn, the full-screen quad generates a single fragment for each output pixel. To determine the color value of a fragment, the fragment shader program is run which contains the actual image processing routine. The fragment shader can read from

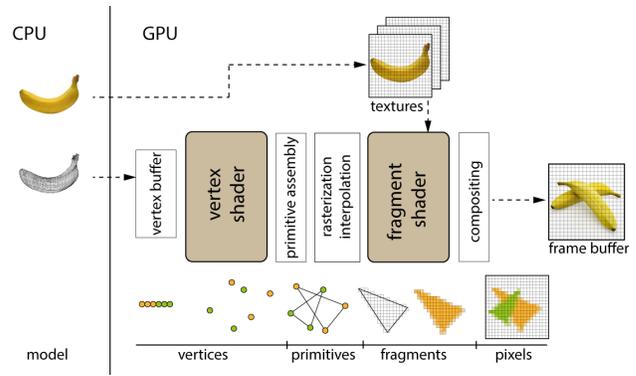


Fig. 1. The OpenGL ES 2.0 rendering pipeline.

any input texture position but can write its output only to the fragment index it is currently assigned to. Therefore, different shaders (filters) are loaded in subsequent rendering passes and the role of the input and output textures is always exchanged. This way a whole chain of image filters can be realized in the GPU. The resulting image is presented on the screen or read back to the CPU for further non-parallel processing. For further reading on GPU computing, please refer to the excellent textbooks GPU Gems [13] and GPU Pro [14].

2.1. Constraints on mobile platforms

OpenGL ES is a constrained subset of its desktop counterpart and hence additional considerations need to be taken in our algorithms when targeting mobile devices. Mobile GPUs have significantly less memory and lower clock speed to reduce energy consumption. Also, transferring data from GPU to CPU and back should be avoided during the algorithm due to low memory bandwidth. The texture fetch latency is significant so the number of texture reads should be minimized. The number of texture units varies but is at least 8 while the maximum texture size is 2048. Although the shaders can perform floating-point calculations, the results must be stored in low-precision fixed-point textures and there is only one color render target (with 4 color channels). However, for practical applications the wide availability compensates for the API's limitations.

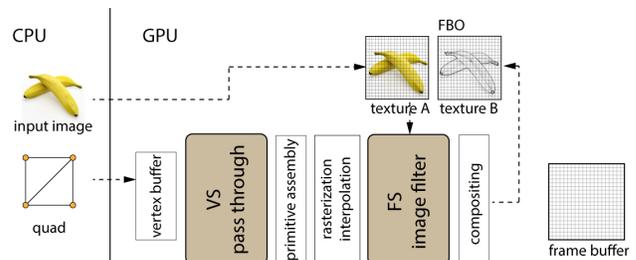


Fig. 2. Image filtering can be reformulated as a rendering task using textures for input as well as for output.

³http://www.nvidia.com/object/cuda_home_new.html

⁴<http://www.khronos.org/opencl/>

⁵<http://developer.android.com/guide/topics/renderscript/index.html>

3. ALGORITHM OUTLINE

The selected barcode localization algorithm [3] consists of multiple stages that are illustrated in Figure 3. The captured color camera image is first converted to grayscale and the chroma value $c = \max(R, G, B) - \min(R, G, B)$ of each pixel is also stored separately for later steps. Because 1D barcodes contain many edges and 2D codes contain many corners, the algorithm searches for areas with high concentration of edge structures as well as for areas with high concentration of corner structures. Assuming that the codes are printed black and white, the chroma mask can be used as a fast test to reject no-code areas in an early stage of the pipeline. The next step is to calculate the elements of the structure matrix $M = \begin{pmatrix} C_{xx} & C_{xy} \\ C_{xy} & C_{yy} \end{pmatrix}$ for each pixel. The C_{ij} entries of M are calculated from the image derivatives I_x and I_y over an image patch D around pixel p using a window w :

$$C_{i,j} = \sum_{(x,y) \in D} w(x,y) I_i(x,y) I_j(x,y)$$

Next, an edge map m_1 and a corner map m_2 are derived from the structure matrix according to the work of Ando [15].

$$m_1 = \frac{(C_{xx} - C_{yy})^2 + 4C_{xy}^2}{(C_{xx} + C_{yy})^2 + \epsilon}, \quad m_2 = \frac{4(C_{xx}C_{yy} - C_{xy}^2)}{(C_{xx} + C_{yy})^2 + \epsilon}$$

The two maps are then blurred with a big block filter to connect areas where there is high line density or high corner density and to remove background clutter. Finally, the two blurred maps are linearly combined to give the 1D and 2D barcode probability maps s_1 and s_2 , respectively. Note that all steps from color conversion through filtering to linear combination are data-parallel operations that can be efficiently implemented in fragment shader programs.

4. BARCODE LOCALIZATION ON MOBILE GPU

This section presents how the above algorithm can be reformulated and optimized for the mobile GPU.

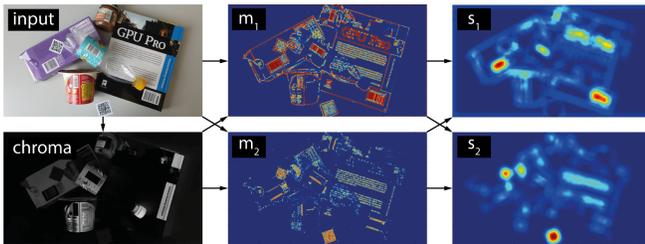


Fig. 3. Algorithm outline: input image and its chroma map, m_1 : edge density map, m_2 : corner density map, s_1 : 1D saliency, s_2 : 2D saliency. The color coding is normalized to $[0..1]$ in all figures. (Please zoom in for viewing)

4.1. Data formats

The inputs and outputs of the subsequent stages are stored in 32-bit RGBA textures. The colors are represented as 8-bit fixed-point values between 0 and 1. Each stage consists of one or two render passes in which a simple quad gets textured with the result of the previous step. Texture filtering is set to *NEAREST* to sample one pixel only and coordinate wrapping is set to *CLAMP_TO_EDGE*. We use non-power-of-two textures which prohibits mipmapping. Texture coordinates are measured in the $[0..1]$ range so the offset between neighboring texels ($1.0/\text{width}$) has to be supplied to all the shaders as a uniform variable. Calculating the coordinates of the neighbors can be shifted to the vertex shader letting the hardware interpolate them for all the fragments.

4.2. Streaming camera images directly to the GPU

The smartphone camera captures the preview frames in NV21 (YUV) format but the textures must be stored in RGBA format. The GLES2 extension *EGL_image_external* allows to bind an external image buffer to a texture unit. The stream output of the camera can be redirected to this image buffer and OpenGL automatically converts its content to RGBA at reading time. This way, the manual conversion of camera frames can be spared, so the frames do not need to be loaded into the CPU.

4.3. Calculating the structure matrix

In the first stage of the algorithm, the elements of the structure matrix $\sum_D I_x^2$, $\sum_D I_y^2$, and $\sum_D I_x I_y$ need to be calculated. The summation over the small neighborhood D can be postponed to the following step so that the calculations for each pixel can be decoupled and can run in parallel. Gray conversion is easily performed as a dot product of the input RGB vector with the constant vector $[0.3, 0.59, 0.11]$ while the chroma calculation involves only built-in maximum and minimum functions.

We calculate the gradients using the *derivative5* filters of Farid et al. [16] that improve the robustness to barcode orientation. The *derivative5x* consists of a 5-tap horizontal component d_1 and a 5-tap vertical component p while in *derivative5y* these two components change their roles. Thanks to the commutativity of convolution, we can perform all horizontal filtering in one pass and all vertical filtering in the second pass which requires only 5 texture fetches in each pass: $I_x = I \otimes d_1 \otimes p^T$ while $I_y = I \otimes d_1^T \otimes p = I \otimes p \otimes d_1^T$. In the first pass, we take 5 gray pixels horizontally and multiply-sum them once with d_1 and store the result in R, and once with p and store the result in G. In the second pass, we filter R with p^T and G with d_1^T vertically. The squares I_x^2 , I_y^2 , and $I_x I_y$ are then stored in the R,G,B channels of the output while the chroma value is always passed in the A channel.

To remove colorful non-barcode areas in this stage, we set all entries in M to zero if the chroma of the pixel is over 0.25. Conditional statements in shaders decrease performance, so we formulate the thresholding without a condition using the built-in *step* function. The line $c < 0.25 ? A = 1.0 : A = 0.0$; is equivalent to $A = 1.0 - \text{step}(0.25, c)$;

4.4. Fast Gaussian filtering

In the second stage, the derivatives are summed up in a small neighborhood to get the final entries of the structure matrix. We apply a 7×7 Gaussian window with standard deviation 2. The 2D Gaussian is the outer product of two 1D Gaussians so again the 2D filter can be substituted by a horizontal plus a vertical 1D convolution. The two passes require only $7 + 7 = 14$ texture fetches instead of $7 \times 7 = 49$. Furthermore, the GPU works parallel on the RGB channels of the input so the three different entries of the structure matrix are calculated at the same time.

4.5. Edge and corner maps

In the third stage, an “edgeness” and a “cornerness” map is generated over the image based on the entries of the structure matrix which involves only pixel-wise arithmetic operations. Intermediate steps of the algorithm are shown in Figure 4.

4.6. Box filtering and saliency maps

Box filtering with a large kernel would require a great number of texture fetches for each fragment. However, if we set the texture filtering to *LINEAR* and sample between the pixels, thanks to hardware interpolation we get the average of the two values in one texture fetch. We use a horizontal and a vertical 32-tap box filter with only 16 texture fetches each between texels (ie., 32 in total instead of 32^2).

In the final stage, the blurred m_1 and m_2 maps are combined to 1D and 2D barcodeness maps, respectively. Linear combination of fragment values can also be done very efficiently in GPU using the *mix* command.

4.7. Boundary detection

Finally, the barcode probability maps are transferred back to the CPU where barcode region candidates are selected. This step is identical to [2, 3].



Fig. 4. Intermediate results rendered on the screen: original image; x-derivatives; edges (red) and corners (green)

5. EXPERIMENTAL RESULTS

We tested our algorithm on three smartphones with different hardware: a *Galaxy Nexus* (1.2 GHz dual-core ARM Cortex-A9 CPU, Imagination PowerVR SGX540 GPU), a *Galaxy S3* (1.4 GHz quad-core ARM Cortex-A9 CPU and ARM Mali-400 MP4 GPU) and a *Galaxy S4* (1.9 GHz quad-core Qualcomm Krait 300 CPU and Qualcomm Adreno 320 GPU). The algorithm has been implemented using OpenCV 2.4.6 (native C++) for the CPU and using OpenGL ES 2.0 for the GPU. Table 1 summarizes the runtimes from image capture to the barcode maps including GPU-CPU transfer. The GPU implementation achieves a speedup of factor 3.2 on the Nexus phone with 4 shader cores and factor 2.0 on the S3 which we believe has only 2 shader cores. The S4 allows 6 frames per secundum even in HD resolution. In that case the 152ms(‡) runtime adds up as follows: derivatives and color conversion (22ms), Gaussian blur (23ms), edge/corner maps (7ms), box filter (41ms), reading from GPU (20ms), and rendering to the screen (39ms). We conclude that OpenGL ES 2.0 brings a significant speedup to data-parallel image processing algorithms and offers code portability across a wide range of hardware platforms.

Frame size	Galaxy Nexus	Galaxy S3	Galaxy S4
640x480	118ms (3.22x)	144ms (2.02x)	49ms (6.10x)
960x720	259ms (3.08x)	322ms (1.96x)	104ms (5.48x)
1280x720	349ms (3.23x)	414ms (2.03x)	152ms (5.41x)‡

Table 1. Average runtime of the GPU implementation and speedup compared to the CPU implementation on different smartphone models.

6. SUMMARY AND OUTLOOK

We have presented the implementation, optimization and evaluation of a robust barcode localization algorithm on embedded GPU using OpenGL ES 2.0. Our localization algorithm can be applied as a preprocessing step for existing barcode decoder algorithms pushing smartphones one step closer to enterprise-grade barcode scanning. We foresee further optimizations in the future once the OpenGL ES 3.0 standard becomes widely available in embedded graphics hardware. The new standard brings several additional features compared to v2.0 including multiple render targets, GPU-CPU synchronization, and floating point textures that allow more sophisticated image processing with less rendering passes. Furthermore, GPU-CPU parallelism would speed up barcode scanning even more; instead of simply waiting for the results, the CPU can process the barcode candidates of the previous frame. We leave this implementation for future work. The approach presented here can be extended to run on any OpenGL ES-compliant wearable computer with little modifications which is an exciting research topic to explore in the future.

7. REFERENCES

- [1] Z. Bai Y. Chen, Z. Yang and J. Wu, “Simultaneous real-time segmentation of diversified barcode symbols in complex background,” in *Proc. First International Conference on Intelligent Networks and Intelligent Systems*, 2008, ICINIS’08, pp. 527–530.
- [2] O. Gallo and R. Manduchi, “Reading 1D barcodes with mobile phones using deformable templates,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 9, pp. 1834–1843, 2011.
- [3] G. Sörös and C. Flörkemeier, “Blur-resistant joint 1D and 2D barcode localization for smartphones,” in *Proc. of ACM Mobile and Ubiquitous Multimedia*, 2013, MUM’13.
- [4] E. Tekin and J. Coughlan, “BLaDE: Barcode localization and decoding engine,” Tech. Rep. 2012-RERC.01, The Smith-Kettlewell Eye Research Institute, December 2012.
- [5] A. Herout I. Szentandrás and M. Dubská, “Fast detection and recognition of QR codes in high-resolution images,” in *Proc. 28th Spring Conference on Computer Graphics*. 2012, SCCG ’12, pp. 129–136, ACM.
- [6] D. Chai and F. Hock, “Locating and decoding EAN-13 barcodes from images captured by digital cameras,” in *Proc. Fifth International Conference on Information, Communications and Signal Processing*, 2005, pp. 1595–1599.
- [7] M. Katona and L. G. Nyúl, “Efficient 1D and 2D barcode detection using mathematical morphology,” in *Proc. International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, 2013, ISMM’13, pp. 464–475.
- [8] L. Li M. Wang and Z. Yang, “Gabor filtering-based scale and rotation invariance feature for 2D barcode region detection,” in *Proc. International Conference on Computer Application and System Modeling*, 2010, vol. 5 of *ICCASM’10*, pp. 34–37.
- [9] A. Tropsch and D. Chai, “Locating 1-D bar codes in DCT-domain,” in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, 2006, vol. 2 of *ICASSP’06*.
- [10] H. Wang K. Wang, Y. Zou, “1D barcode reading on camera phones,” *International Journal of Image and Graphics*, vol. 7, no. 3, pp. 529–550, 2007.
- [11] J. Alfthan, “Robust detection of two-dimensional barcodes in blurry images,” M.S. thesis, KTH Stockholm, Sweden, 2008.
- [12] A. Herout M. Dubská and J. Havel, “Real-time precise detection of regular grids and matrix codes,” *Journal of Real-Time Image Processing*, pp. 1–8, Feb 2013.
- [13] H. Nguyuen, Ed., *GPU Gems 3*, Addison Wesley Professional, 2008.
- [14] W. Engel, Ed., *GPU Pro, Advanced Rendering Techniques*, A K Peters, 2010.
- [15] S. Ando, “Image field categorization and edge/corner detection from gradient covariance,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 2, pp. 179–190, Feb. 2000.
- [16] H. Farid and E. P. Simoncelli, “Differentiation of discrete multidimensional signals,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 496–508, Apr. 2004.