Research Journal of Applied Sciences, Engineering and Technology 4(24): 5565-5571, 2012
ISSN: 2040-7467

Submitted: April 25, 2012 Accepted: May 13, 2012 Published: December 15, 2012

# A Fast GVF Snake Algorithm on the GPU

Zuoyong Zheng and Ruixia Zhang
Department of Information Engineering, North China University of Water Resources and Electric Power, Zhengzhou, China

**Abstract:** GVF Snake is one of the most widely-used edge detection algorithms, nevertheless subject to its slow computation. This study reveals the bottleneck and transfers the time-consuming part of this algorithm to the GPU for better performance. In detail, this algorithm is decomposed into three parts, (1) GVF Computation, (2) inversing a circulant matrix and (3) curve deformation. All of these parts are analyzed and designed to run on the GPU via suitable data structures and corresponding operations. With the help of parallel computational power of the GPU, our improved algorithm could be about 15 times as fast as is executed on the CPU.

**Keywords:** Circulant matrix, discrete fourier transform, fragment shader, GPU, GVF, snake

## INTRODUCTION

Active Contour Model (Snake) (Kass *et al*., 1987) is one of the most well-known edge detection technologies in image processing and computer vision field. It defines an energy function of a closed parametric curve, which can achieve a local minimum when this curve converges to the edge. In essence, it is an extremum of function which can be solved by variational calculus. To overcome its inherent limitation, Gradient Vector Flow (GVF) (Xu and Prince, 1998) was proposed to be a new external force, under which even points far away from the edge could also be pushed toward it. It is very distinct that GVF computation is added in addition to the original snake deformation. Therefore, GVF Snake is not suitable for time-sensitive applications due to its slow computational speed.

Aiming at this performance problem, a GPU-accelerated GVF Snake algorithm is proposed in this study, transferring the bottleneck component to the GPU. This transition toward real-time execution can provide wider application prospects.

## LITERATURE REVIEW

Graphics Processing Unit (GPU) evolves rapidly in the last decade, which possesses following five features: High-speed floating computation:

- Highly parallelism, including data and instruction parallelism
- I/O stream model
- High-speed on-board memory chip
- Flexible programmability

At present, GPUs have not been limited within the traditional graphic rendering any more, instead, they are going deep into general computing (NVIDIA Corp, 2011).

In image processing and computer vision, many fundamental algorithms are time-elapsing. Therefore, accelerating these algorithms using GPU-related technologies becomes significant and practical. Sinha *et al*. (2011) realized a basic SIFT and KLT algorithm. Chriot and Keriven (2008) implemented SIFT/SURF which can simultaneously track about 5,000 feature points at the speed of 50 FPS on a Geforce 8800 M card. Choudhary *et al*. (2010) provided the fundamental principles and implementation details for bundle adjustment executed on the GPU, which are based on the traditional Levenberg-Marquardt optimization.

Relevant to individual vision algorithms, (Tzevanidis *et al*., 2010) implemented a complete real-time 3D reconstruction system based on visual hull and executed on the GPU, from image segmentation, visual hull calculation, Marching Cube (MC), surface smoothness to texture mapping. Their work proved that almost all of the core algorithms in a 3D reconstruction system could be transplanted to the GPU.

In this study, the GPU will be used to accelerate the implementation of GVF Snake.

**GVF snake algorithm:** In the Snake algorithm (Kass *et al*., 1987), an initial closed curve $x(s) = (x(s), y(s))$ deforms under the internal and external force and

**Corresponding Author:** Zuoyong Zheng, Department of Information Engineering, North China University of Water Resources and Electric Power, Zhengzhou, China

converges gradually to the edge. The internal and external forces are derived from the corresponding energy functions. The internal energy function is defined as:

$$E_{\text{int}ernl} = \int_s \frac{1}{2}\left(\alpha(s)\left|x_s\right|^2 + \beta\left|x_{ss}\right|^2\right)ds \quad (1)$$

The first item in the integral represents the elastic energy which forces the curve to shrink; and the second term defines the bending energy which makes the curve tend to be smooth. $\alpha$ (*s*) and $\beta$ (*s*) are the piecewise weighting function, which are usually treated as constants. The external energy is from the image itself, defining the image energy along the curve:

$$E_{external} = \int_S E_{image}\left(x(s)\right)ds \quad (2)$$

The image energy is usually defined as either of the following two forms:

$$E_{image}\left(x,y\right) = -\left|\nabla I\left(x,y\right)\right|^2 \quad (3)$$

$$E_{image}\left(x,y\right) = -\left|\nabla\left[G_\sigma\left(x,y\right) * I\left(x,y\right)\right]\right|^2 \quad (4)$$

The total energy is the sum of the above two energy items:

$$\begin{aligned} E_{snake} &= E_{\text{int}ernal} + E_{external} \\ &= \int_S \left[\frac{1}{2}\left(\alpha(s)\left|x_s\right|^2 + \beta(s)\left|x_{ss}\right|^2\right) + E_{image}\left(x(s)\right)\right]ds \end{aligned} \quad (5)$$

The core idea of Snake is to find such a curve that makes the total energy $E_{snake}$ achieve a minimum. It is a functional extremum in mathematics, which can be solved by the following Euler-Lagrange equation corresponding to (5):

$$\alpha x_{ss} - \beta x_{ssss} - \nabla E_{image} = 0 \quad (6)$$

This equation can be understood as a force balance equation, in which the first two items correspond to the internal force relevant to the curve shape; and the third item corresponds to the external force based on the image structure. The curve would deform under the cooperative influence of these forces and close up at the boundary.
The Snake algorithm is employed in many applications, however, it depends on the initial curve location too much to guarantee its convergence to concave boundaries. To this problem, (Xu and Prince, 1998) proposed a new representation for the external force so as to replace the $-\nabla E_{image}$ in (6):

$$\alpha x_{ss} - \beta x_{ssss} + V = 0 \quad (7)$$

The V item is defined as a vector field, i.e., V (x, y) = (u (x, y), v (x, y) , which can be figured out through the following energy functional:

$$\Psi = \int\int \mu\left(u_x^2 + u_y^2 + v_x^2 + u_y^2\right) + \left|\nabla f\right|^2\left|V - \nabla f\right|^2 dxdy \quad (8)$$

$\mu$ is a adjustment parameter and *f* (*x*, *y*) is the edge map of the image taking the form in (3) or (4) (In fact, *f* (*x*, *y*) = $-E_{image}$ (*x*, *y*)). Therefore, *V* is determined by *f* (*x*, *y*), say, the image itself. The definition of (8) assures that the gradient at the edge can be scattered far away, therefore *V* is referred to as GVF. Minimizing the energy $\Psi$ is also a problem regarding functional extremum, solved by the following Euler-Lagrange Equation for multi-variable functions:

$$\begin{aligned} \mu\nabla^2 u - \left(u - f_x\right)\left(f_x^2 + f_y^2\right) &= 0 \\ \mu\nabla^2 v - \left(v - f_y\right)\left(f_x^2 + f_y^2\right) &= 0 \end{aligned} \quad (9)$$

where, $\nabla^2$ is Laplacian operator. So the Snake with this specially defined external force *V* becomes GVF Snake, which can force the curve to extend to concave regions. However, it brings slower computation because the introduction of GVF. We are about to run GVF Snake on the GPU, i.e., implement (7) and (9) via GPU acceleration. Before doing this, it is necessary to discretize these two equations and obtain their numerical solutions.

## NUMERICAL METHODOLOGY

**Computation of GVF:** If *u* and *v* in (9) are viewed as functions of time *t* and the right side of this equation are written as derivatives of *u* and *v* with respect to *t*, the following equations are obtained:

$$\begin{aligned} u_t\left(x,y,t\right) &= \mu\nabla^2 u\left(x,y,t\right) \\ &- \left[u\left(x,y,t\right) - f_x\left(x,y\right)\right]\cdot\left[f_x^2\left(x,y\right) + f_y^2\left(x,y\right)\right] \\ v_t\left(x,y,t\right) &= \mu\nabla^2 v\left(x,y,t\right) \\ &- \left[v\left(x,y,t\right) - f_y\left(x,y\right)\right]\cdot\left[f_x^2\left(x,y\right) + f_y^2\left(x,y\right)\right] \end{aligned} \quad (10)$$

Equation (10) establishes the relationship for *u* and *v* between neighboring time points, forming an iterative numerical scheme, whose convergent solution is the same as (9). Let $b(x, y) = f^2_x(x, y) + f^2_y(x, y)$, $c^1(x, y) = f_x(x, y)\, b(x, y)$ and $c^2(x, y) = f_y(x, y) b(x, y)$, Eq. (10) becomes (11):

$$\begin{aligned} u_t(x,y,t) &= \mu\nabla^2 u(x,y,t) - b(x,y)u(x,y,t) + c^1(x,y) \\ v_t(x,y,t) &= \mu\nabla^2 v(x,y,t) - b(x,y)v(x,y,t) + c^2(x,y) \end{aligned} \quad (11)$$

Replacing pixel location $(x, y)$ and time $t$ respectively with subscript $i$, $j$ and superscript $n$, the Laplacian operator in the above equation can be written as:

$$\begin{aligned} u_t &= u_{i,j}^{n+1} - u_{i,j}^n \\ v_t &= v_{i,j}^{n+1} - v_{i,j}^n \\ \nabla^2 u &= u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n \\ \nabla^2 v &= v_{i+1,j}^n + v_{i-1,j}^n + v_{i,j+1}^n + v_{i,j-1}^n - 4v_{i,j}^n \end{aligned} \quad (12)$$

Substitute (12) into (11), the iterative numerical method for GVF is finally obtained:

$$\begin{aligned} u_{i,j}^{n+1} &= (1 - b_{i,j} - 4\mu)u_{i,j}^n + \mu(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) + c_{i,j}^1 \\ v_{i,j}^{n+1} &= (1 - b_{i,j} - 4\mu)v_{i,j}^n + \mu(v_{i+1,j}^n + v_{i-1,j}^n + v_{i,j+1}^n + v_{i,j-1}^n) + v_{i,j}^2 \end{aligned} \quad (13)$$

We can see the result at the next iteration is just the sum of the convolution of the current result with a constant. How to implement (13) on the GPU will be explained in section VI.

**Deformation of snake:** In GVF Snake, the deformation of the curve is controlled by (7). If the parametric curve $x(s)$ is discretized into $k$ points with $i$ being its index, then for the $i^{th}$ point, its second-order and fourth-order derivative vector, $x_{ss}$ and $x_{ssss}$, respectively equal $x_{i+1}$-$2x_i$+$x_{i-1}$ and $x_{i+2}$-$4x_{i+1}$+$6x_i$-$4x_{i-1}$+$x_{i-2}$. Substituting them into (7), we can obtain:

$$\begin{aligned} &\beta x_{i+2} - (\alpha + 4\beta)x_{i+1} + (2\alpha + 6\beta) \\ &x_i - (\alpha + 4\beta)x_{i-1} + \beta x_{i-2} V(i) = 0 \end{aligned} \quad (14)$$

All of the $k$ points on the curve satisfy (14), therefore a sparse linear equation system is formulated and can be written into the following matrix form:

$$\begin{aligned} Ax - u(x, y) &= 0 \\ Ay - v(x, y) &= 0 \end{aligned} \quad (15)$$

Again, regarding point coordinate $(x, y)$ as a function of time $t$ and the right side of (15) as derivatives of $(x, y)$ with respect to time $t$, the following equation is achieved:

$$\begin{aligned} Ax^{n+1} - u(x^n, y^n) &= x^n - x^{n+1} \\ Ay^{n+1} - v(x^n, y^n) &= y^n - y^{n+1} \end{aligned} \quad (16)$$

Thus, an iterative scheme for the location of each point on the curve is derived:

$$\begin{aligned} x^{n+1} &= (A + I)^{-1}\left[x^n + u(x^n, y^n)\right] \\ y^{n+1} &= (A + I)^{-1}\left[y^n + v(x^n, y^n)\right] \end{aligned} \quad (17)$$

The polygon formed by these points is just the closed-form solution to the desired smooth edge.

**Inversing a circulant matrix:** It is obvious that (13) and (17) are the fundamental numerical methods for implementing GVF Snake. In (17), the inverse of $A+I$ is a non-negligible problem. In mathematics, $A$ and $I$ are all circulant, so their sum is too. An $n$-order circulant matrix $C$ is defined as the following form Xu *et al*. (1999):

$$C = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & \cdots & c_{n-2} \\ \cdots & \cdots & \ddots & \cdots \\ c_1 & c_2 & \cdots & c_0 \end{bmatrix} \quad (18)$$

Apparently, $C$ is determined by its first row vector, which is denoted by $C$ = circ ($c_0$, $c_1$, $c_{n-1}$). For $C$ inverse, there exists a fast algorithm which consists of three steps:

- Calculate all eigenvalues $\lambda_k$ of $C$, which is equivalent to execute 1D DFT (Discrete Fourier Transform) on the $C$'s first row vector:

$$\lambda_k = \sum_{j=0}^{n-1} c_j \omega^{kj} \quad (k = 0, 1, \cdots, n-1) \quad (19)$$

where,

$$\omega = e^{-\frac{2\pi}{n}i}, (i = \sqrt{-1})$$

- Inverse $\lambda_k$ to obtain $\mu_k$: $\mu_k = 1/\lambda_k$.
- Execute 1D IDFT (Inverse DFT) on $\mu_k$ to get the first row $\eta$ of $C^{-1}$:

$$\eta_j = \frac{1}{n}\sum_{k=0}^{n-1} \mu_k \omega^{-kj}, \quad j = 0, 1, \cdots, n-1 \quad (20)$$

Because $C^{-1}$ is also circulant, evaluating its first row vector $\eta$ is sufficient. If the matrix order $n$ is just the power of 2, the first and third step can be applied by FFT and IFFT, which means the time complexity can be reduced significantly. However, because this is not a general hypothesis for most applications (e.g., in this study, $n$ is the number of contour points), we still adopt the original definitions of DFT and IDFT.

**Implementation details:** From (13), (17), (19) and (20), it is clear that the kernel algorithm is just operations on

matrices and vectors, including addition and multiplication. We have implemented these basic linear algebra operations on the GPU.

We use OpenGL and its shading language to accelerate the GVF Snake algorithm. OpenGL is a well-known graphics library, supplying a lot of APIs and extensions to interact with the display driver and finally controlling the graphic rendering. In the rendering pipeline, the vertex and pixel shaders are two main parts, which are programmable via the OpenGL Shading Language (GLSL). OpenGL and its GLSL are principally used for fast rendering. However, they can be suitable for general-purpose computation by means of intricate alteration. Nowadays, applying the GPU for general-purpose computation has become a prevalent trend.

Taking our algorithm into account, its some critical ingredients correspond to GPU/OpenGL/GLSL concepts in the following ways:

- **Matrices:** They are arrays in the system memory and become textures after transferred to the video memory. Each *texel* (texture element) in a texture can store four 16- or 32-bit floats.
- **Delivery of matrices:** The OpenGL program is responsible for the delivery of matrices between the system memory and the video memory.
- **Function modules:** Implemented by the pixel shader, which is compiled and linked on the host and then transferred to the GPU. The pixel shader reads data from read-only textures and writes results into write-only textures.
- **Execution of function modules:** Triggered by the OpenGL rendering instructions. Here, the primitive rendered is a quad which is rasterized into internal pixels within it, each of them corresponding to a matrix element. The same copies of the pixel shader run simultaneously on all pixel processing units, handling each matrix element in a random order (SIMD, Single Instruction Multiple Data). This procedure is equivalent to evaluate each element in a matrix.

The OpenGL extensions we used include:

- **ARB_texutre_float:** Allowing 16-bit or 32-bit floating format for textures.
- **ARB_texture_rectangle:** Allowing texture size to be any integer within the permitted range rather than a power of 2 and allowing texture coordinates not to be normalized.
- **EXT_framebuffer_object:** Supporting one or more textures to be bound to a framebuffer object so as to write calculation results directly into textures, that is, Render-to-Texture (RTT). This functionality is very useful for storing intermediate or temporary data.
- **ARB_shader_object and ARB_fragment_shader:** Fragment program.

The pipeline for computing GVF is demonstrated in Fig. 1.$b_{i,j}$ $c^1_{i,j}$ $c^2_{i,j}$ (see (10) and (11) ) are fitted into a texture as its *x*, *y* and *z* components after they are precomputed on the CPU, while GVF is initialized as gradient of the edge map with its *u* and *v* component being stored in *x* and *y* channel of another texture. GVF calculation is composed of a loop that renders a quad. The sizes of this quad, two textures and the viewport are all same. Calculation results are rendered into a resultant texture via RTT. After one pass rendering, invert the input and resultant texture and launch the next pass rendering until GVF converges.

For solving the inverse of $A+I$, we incorporate its first and second steps (step 1 and 2) into a one (See section V), corresponding to rendering the first quad; while the last step corresponds to rendering the second quad (Fig. 2). Before the rendering process is initiated, the texture *signal* stores the first row of *C*. After the first quad is rendered, what the texture *inverse spectrum* contains is $\mu_k$. Because $\mu_k$ are complex numbers, their real and imaginary parts are stored respectively in *x* and *y* channels of texels. In the second pass, the texture *inverse spectrum* becomes read-only and *signal* becomes write-only for the sake of the forthcoming IDFT. Once the inverse is finished, the matrix $(A+I)^{-1}$ stays in the video memory as a readable texture for the subsequent computation.

The last time-consuming part of GVF Snake is the curve deformation (Fig. 3), which is described as follows:

- The initial boundary points are transferred to the GPU as ordinary textures.
- Similar to computing GVF, a quad with its height being 1 and width being the number of edge points is rendered; each fragment is sent to the fragment program and its color is calculated in terms of (17), which is just the new position of the corresponding contour point and is stored in the color buffer 1. In addition, Euclidean distances between the new points and their old counterparts are calculated and stored in color buffer 2, in which the *x*, *y* and *z* channel are respectively the new *x*, *y* coordinate and the distance map.
- After one pass rendering, exchange the input and output textures for contour points and continue the next pass (step 2). If the accumulated rendering count reaches a predefined number (e.g., 10), then go to step 4.
- In order to confirm the collection of discrete contour points is convergent to the true object edge, copy color buffer 2 (the distance map) to a PBO (Pixel Buffer Object) and use it as a VBO (Vertex Buffer Object) where its *x*, *y* and *z* channel are directly treated as conventional vertex coordinates. When rendering these vertices, enable the depth test and count how many vertices could pass the test, that is, how many vertices have their *z* coordinates being less

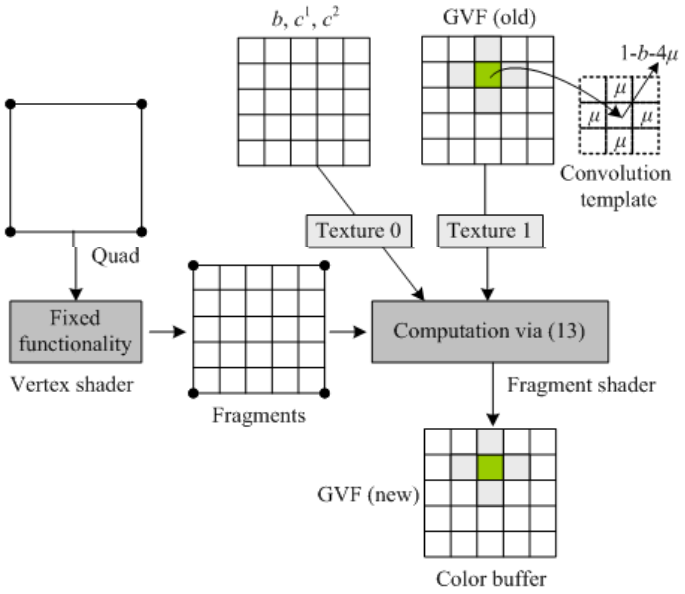


Fig. 1: Pipeline for computing GVF

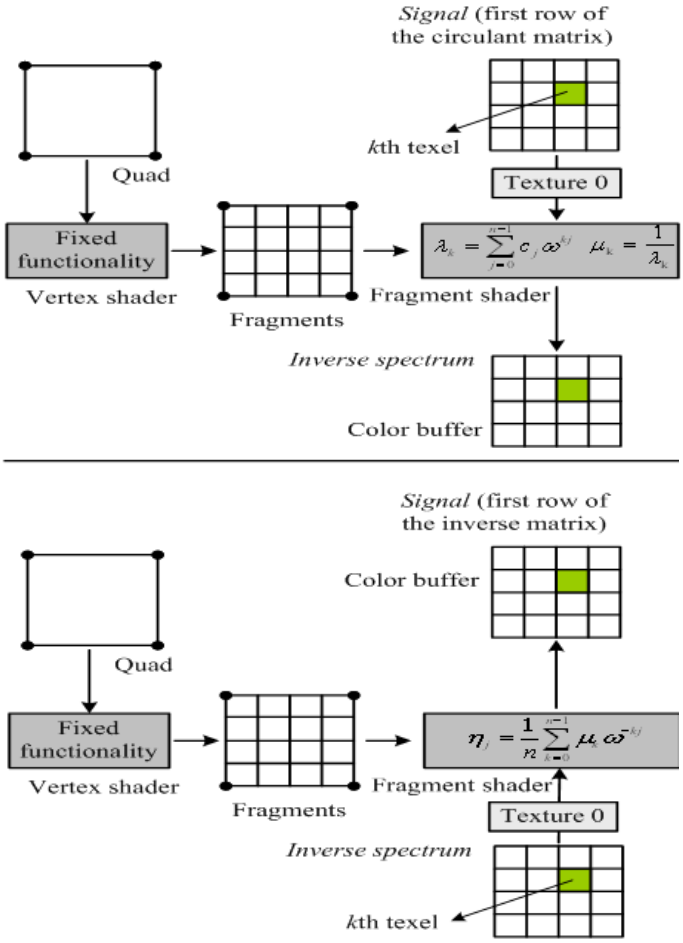


Fig. 2: Pipeline for inversing a circulant matrix

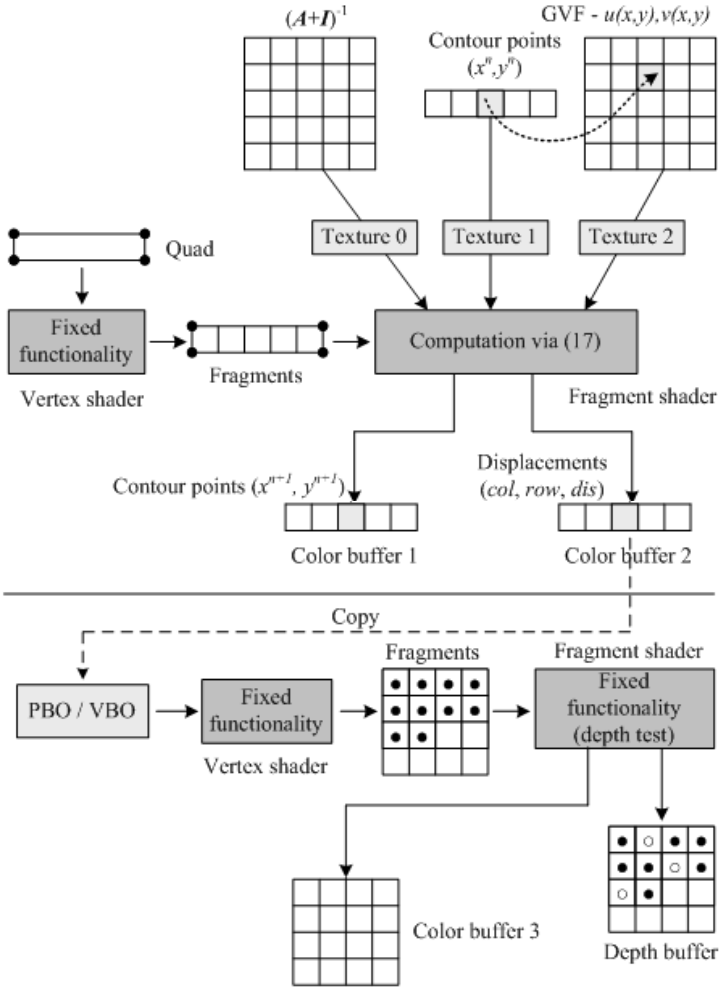


Fig. 3: Pipeline for snake deformation

than a predefined value. If such vertices occupy the majority of all vertices which means the detected edge is to be convergent, terminate this algorithm; otherwise go to step 5.

- Transfer the current collection of contour points back to the system memory and execute the culling and interpolation on the CPU to keep uniform and intensive distances between any two neighboring points. Use this new point set as a new initial boundary, then go to step 1.

## EXPERIMENTAL RESULTS

We run our GPU version of GVF Snake on Intel Core i3-530 with 2G DDR-1333 system memory. The graphic card used is Yeston GT240-TC512GD5 standard edition (core name GT215). Edge detection results on some synthesized and real images are displayed in Fig. 4, which are exactly same as the results by CPU computation.

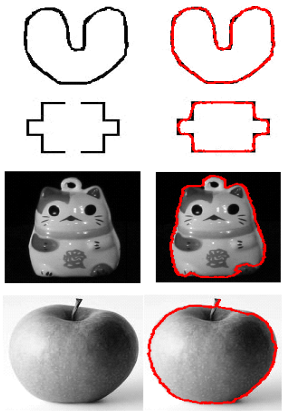

Fig. 4: Some edge detection results

From Table 1 we can see that the GPU version of GVF Snake is about 15 times faster than the CPU version.

Table 1: Performance comparison between CPU and GPU version

| Image | Size | CPU time (ms) | GPU time (ms) |
|---|---|---|---|
| U-shape | 300x300 | 8,936 | 672 |
| Room | 300x300 | 8,847 | 624 |
| Cat | 200x200 | 4,173 | 335 |
| Apple | 308x217 | 6,395 | 417 |

The number of deformation was preset to an estimated constant (see the step 3 of curve deformation in the previous section), which means that either CPU or GPU computation time could be influenced by this parameter. However, this does not hamper performance comparison between these two versions.

## CONCLUSION

In this study, bottleneck parts of GVF snake are analyzed, which are then transferred to the GPU to accelerate their computation by means of both powerful parallel capacity of GPU and adequate operations/data structures. In practical use (e.g., edge detection for moving objects in a video stream), the continuity existing either in GVF or in contour deformation can be utilized to reach real time tracking, concretely, by treating the result of the current frame as input for the next frame. Therefore, it may provide a mighty support for the real time application of GVF Snake.

## ACKNOWLEDGMENT

This study was supported by the research project 10PTGS507-3 of Zhengzhou Science and Technique Bureau.

## REFERENCES

Chriot, A. and R. Keriven, 2008. GPU-boosted online image matching. Proceeding of IEEE International Conference on Pattern Recognition (ICPR 08), Tampa, Florida, USA, pp: 1-4.

Choudhary, S., S. Gupta and P.J. Narayanan, 2010. Practical time bundle adjustment for 3D reconstruction on GPU. Proceeding of ECCV 2010 Workshop on Computer Vision on *GPUs* (CVGPU 2010), Crete, Greece.

Kass, M., A. Witkin and D. Terzopolous, 1987. Snake: Active contour models. Int. J. Comput. Vision, 1: 321-331NVIDIA Corp., 2011. Retrieved from:http: // www.nvidia.com/object/cuda-home-new.html.

Sinha, S.N., J.M. Frahm, M. Pollefeys and Y. Genc, 2011. Feature tracking and matching in video using programmable graphics hardware. Mach. Vision Appl., 22: 207-217.

Tzevanidis, K., X. Zabulis, T. Sarmis, P. Koutlemanis, N. Kyriazis and A.A. Argyros, 2010. From multiple views to textured 3D meshes: A GPU-powered approach. Proceeding of ECCV 2010 Workshop on Computer Vision on GPUs (CVGPU 2010), Crete, Greece.

Xu, C. and J.L. Prince, 1998. Snakes, shapes and gradient vector flow. IEEE T. Image Process., 7: 359-369.

Xu, Z., K. Zhang and Q. Lu, 1999. Fast Algorithms for Toeplitz Matrices. G. Leng, (Ed.), Xi'an: Publishing House of Northwest Industrial University, Chinese, pp: 31-34.