GPU-Based Active Contour Segmentation Using Gradient Vector Flow

Zhiyu He and Falko Kuester

Calit2 Center of GRAVITY
University of California, Irvine
zhe@uci.edu, fkuester@uci.edu

Abstract. One fundamental step for image-related research is to obtain an accurate segmentation. Among the available techniques, the active contour algorithm has emerged as an efficient approach towards image segmentation. By progressively adjusting a reference curve using combination of external and internal force computed from the image, feature edges can be identified. The Gradient Vector Flow (GVF) is one efficient external force calculation for the active contour and a GPU-centric implementation of the algorithm is presented in this paper. Since the internal SIMD architecture of the GPU enables parallel computing, General Purpose GPU (GPGPU) based processing can be applied to improve the speed of the GVF active contour for large images. Results of our experiments show the potential of GPGPU in the area of image segmentation and the potential of the GPU as a powerful co-processor to traditional CPU computational tasks.

1 Introduction

In the area of image based analysis and its related applications, segmentation is, in many cases, the starting point for further processing. The segmentation algorithm may provide the foundation for further processing, such as identifying features or objects that subsequently are used for the reconstruction of 3D models. Among many existing segmentation algorithms, the active contour technique or *snake* [1] is an algorithm that uses an *external force* and an *internal force* to progressively fit a closed curve to edges, boundaries or other features of interest specified via gradient. The snake has been widely used in areas such as biomedical image analysis and further enhanced for specific problem domains. For example, Xu and Prince [2] proposed a better way of calculating the external force of the curve. This improved snake algorithm is called *Gradient Vector Flow* (GVF) snake and has two advantages over the original snake algorithm: (1) it is less sensitive to initialization and (2) it can move into boundary concavities. This paper introduces a hardware accelerated technique for gradient vector flow computation, utilizing the vertex and fragment units on today's graphics processing units. Most mid-range GPUs now have a SIMD architecture and deep parallel processing capabilities on the vertex and fragment units [3], which can be used as a very efficient co-processor that can take over some of the computation

G. Bebis et al. (Eds.): ISVC 2006, LNCS 4291, pp. 191–201, 2006.

tasks otherwise handled by the CPU. These computation tasks are not limited to graphics and visualization but may also include general purpose computation. This paper is organized as follows: Section 2 introduces background and prior work done in related areas. Section 3 gives an overview of GPGPU based processing and how it is related to this research. Section 4 describes the GPU implementation of the GVF snake algorithm. Section 5 provides a performance and test results.

2 Related Work

The segmentation by active contour or snake algorithm can be found in combination with many image related applications. Kass et al. [1], introduced the snake algorithm, which uses an external force and an internal force to conform the contour to certain features in the image. The external force is calculated from the image and the internal force is derived from the contour itself. The corresponding curve is defined by:

$$X_t(s,t) = \alpha X''(s,t) - \beta X''''(s,t) - \nabla E_{ext} \tag{1}$$

where $X_t(s,t)$ is the curve that represents the snake at time t and $X(s) = [x(s), y(s)], s \in [0,1]$ is the parametric curve, X'' and X'''' are the second and fourth order derivatives, α and β are constants that defines the internal forces. ∇E_{ext} is the external force. The GVF snake introduced by Xu and Prince [2] improves the above by introducing a new external force. The revised dynamic snake function can then be formulated as:

$$X_t(s,t) = \alpha X''(s,t) - \beta X''''(s,t) + V \tag{2}$$

where V stands for the new static external force field called *gradient vector flow* (GVF). Zimmer et al. [4] applied the algorithm to video tracking for the quantitative analysis of cell dynamics. Ding et al. [5] described a volumetric CT data segmentation that is based on application of GVF snake to 2D CT slices. Vidholm et al. [6] introduced a virtual reality system for the visualization of volume data combined with force-feedback. GVF snake segmentation of the data was used for visual augmentation and control of the haptic device. Some of the GPU processing and bandwidth characteristics can outpace that of CPUs, which make it appealing to convert processing extensive algorithms to the GPGPU domain if their nature is compatible. For example, Rumpf et al. [7] introduced a level-set based segmentation that was leveraging GPU capabilities. Despite of the advantages of the level-set segmentation, the implementation was still limited by the graphics hardware available at that time and therefore is not completely GPU centric. Kondratieva et al. [8] described a real-time computing and visualization technique for diffusion tensor images, which achieves both visual and speed improvements over traditional CPU realization. Fan et al. [9] built a computing cluster based on GPU to achieve greater parallel processing power. Kipfer et al. [10] implemented a fluid dynamics simulation engine on the GPU, which leverages the GPU to avoid I/O bottlenecks and improves performance. Fatahalian

et al. [11] implemented an efficient matrix multiplication algorithm on the GPU. GPU based computation is not limited to the above mentioned areas and can be expand to many other areas compatible with the SIMD architecture.

3 GPU and GPGPU

Recent GPUs demonstrate enormous potential for scientific computing tasks in the form of General Purpose GPU-based processing (GPGPU). In particular, memory bandwidth and instructions per second highlight potential benefits. For instance, the Nvidia Geforce 6800 graphics chip can process 600 Million vertices/sec and has a fill rate of 6.4 billion pixels/sec, while the Geforce 7800 series can almost double that performance. Galoppo et al. [14] reported that the 6800 could achieve 2.5 billion instructions per second for division, which compare to 6.7 billion for a Pentium4 3.2GHZ CPU. Kilgariff and Fernando [3] demonstrated that the GPU Memory Interface of the Geforce 6800 series can reach 35 GB/sec, which compares well against the 6.4 GB/sec of the CPU Memory Interface for a 800 MHz Front-Side Bus. Besides these, the GPU has a very different architecture and processing stream than the CPU. The GPU processing model can be decomposed into several stages ([15]). Data goes from the CPU to GPU through system bus. On the GPU, it goes from vertex buffer, the vertex processor, rasterization and finally gets to the fragment processor. One important feature of GPU is its SIMD architecture that naturally supports parallel processing. Most computation tasks on GPU are parallelized as illustrated in Fig.1. For example, the Geforce 6800 supports 6 vertex units and 16 fragment units. And each unit can process 4 components (RGBA or xyzw) in parallel.

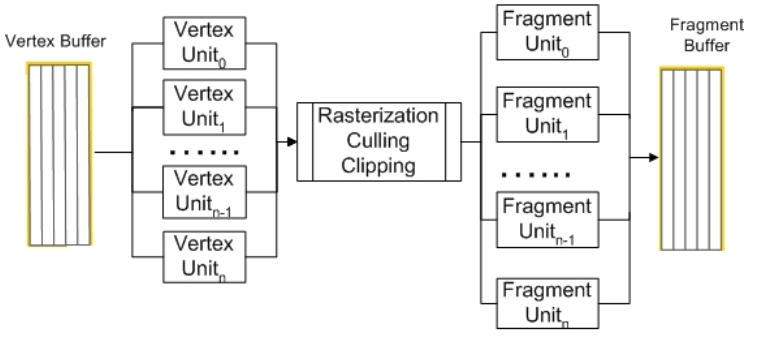


Fig. 1. The parallel nature of GPU

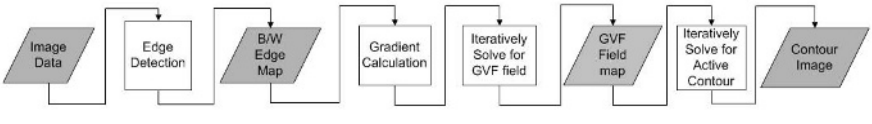


Fig. 2. The process of GVF Snake

However, it is important to carefully consider strengths and weaknesses of GPU-based techniques. First of all, although the GPU has excellent computational power, the majority of graphics cards are still limited to 16bit floating point precisions. This means, in many cases, the traditional implementations of algorithms will be subjected to a loss in precision when migrated directly onto the GPU. One solution is to use 2 components of texture unit to store one 32bit float number. With more graphics cards supporting the 32bit floating point textures, this problem will be reduced in the near future. However, it is still very important to find a balance between the precision and speed because the 32bit floating point data lead to nearly half the speed of the 16bit precision data as reported in [16].

Secondly, while the bandwidth on the CPU or on the GPU alone can be enormous, the bus I/O between CPU and GPU can sometimes become a bottleneck. On the Geforce 6800 card, the PCI Express×16 inteface provides 8 GB/sec throughput while the on-board bandwidth for the GPU is 35 GB/sec. Therefore, it is worthwhile to optimize the code for fewer I/O on the GPU.

Third, the data structure should fit to the platform architecture. When implementing an algorithm on the GPU, it is important to consider its SIMD architecture. The data should be independent from each other, and random access of data such as a linked-list should be avoided if all possible.

Shader Model 3.0 and the OpenGL 2.0 standard provide a means to resolve the problems mentioned this far. For example, multiple rendering target could save rendering passes by using a single input texture to generate multiple output textures. In addition, Frame Buffer Objects (FBOs) greatly improve the speed by saving I/O between GPU and CPU. The vertex texturing functionality allows the texture to be used as a data array. In support of hardware-based processing, different high-level languages were created, such as CG [17] and [18], which supports most features for Shader Model 3.0. HLSL [19] and the OpenGL Shading Language [20] are also such languages.

4 Gradient Vector Flow Snake Implementation on GPU

Equation (2) describes the GVF-based snake function, which introduced the V term for the gradient vector flow. V can be defined as a vector field $V(x, y) = [u(x, y), v(x, y)]$ that minimizes the energy function:

$$\varepsilon = \int\int \mu(u_x^2 + u_y^2 + v_x^2 + v_y^2) + |\nabla f|^2 \, |V - \nabla f|^2 \, dxdy \tag{3}$$

where $f(x, y)$ is an edge map of the original image, ∇f is its gradient map and μ is a constant that represents the level of noise. To solve Equation (3) for the $V(x, y)$, u and v need to be treated as functions of time by solving the following equations:

$$u_t(x, y, t) = \mu\nabla^2 u(x, y, t) - b(x, y)u(x, y, t) + c^1(x, y)$$
$$v_t(x, y, t) = \mu\nabla^2 v(x, y, t) - b(x, y)v(x, y, t) + c^2(x, y)$$

where:

$$b(x,y) = f_x(x,y)^2 + f_y(x,y)^2,\ c^1(x,y) = b(x,y)f_x(x,y),\ c^2(x,y) = b(x,y)f_y(x,y)$$

and ∇^2 is the laplacian operator. This can be numerically expressed as:

$$u_t = \frac{1}{\Delta t}(u_{i,j}^{n+1} - u_{i,j}^{n}), v_t = \frac{1}{\Delta t}(v_{i,j}^{n+1} - v_{i,j}^{n})$$

$$\nabla^2 u = \frac{1}{\Delta x \Delta y}(u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j})$$

$$\nabla^2 v = \frac{1}{\Delta x \Delta y}(v_{i+1,j} + v_{i,j+1} + v_{i-1,j} + v_{i,j-1} - 4v_{i,j})$$

By substituting the above variables into the equations for $u_t(x,y,t)$ and $v_t(x,y,t)$, an iterative solution to the GVF field can be obtained.

The GVF snake algorithm is composed of two parts: (1) the pre-computing of the GVF field and (2) the iterative solution of the snake function. Both parts has the temporal and spatial locality. At any single time step, the $u(x,y)$ and $v(x,y)$ only involves 4 of its neighboring points and 1 previous time step. The general flow for this algorithm is illustrated in Figure 2. First, the input image is converted to greyscale and an edge detection filter is applied to obtain the edge map. Subsequently, a second shader is used to obtain the gradient map and generates three contants for every pixel, namely $b(x,y)$,$c^1(x,y)$ and $c^2(x,y)$. The multiple rendering target technique is then used to generate and store the results in two seperate textures, one for the gradient and the other for the constants. Most current GPUs only support 16bit floating point precision with values clamped to the range of $[0.0, 1.0]$. Therefore, we store the data using a *packing* scheme. The gradient dx and dy are stored in R and G components and the B and A components save a flag number identifying how the dx and dy are stored. In this particular case, the dx is stored as is if $|dx| \geq 0.01$, otherwise, as $-1/\ln dx$. Similar *packing* is performed on the three constants.

After these preparation steps, the iterative GVF field calculation can start. The iterative computation on GPU can be mapped to a so-called *ping-pong* scheme using the FBO(*frame buffer objects*). Each FBO can be bound to four framebuffers, namely, COLOR0 through COLOR3. This is illustrated in following pseudo code:

```
Src_Buffer = COLOR0;  Dst_Buffer = COLOR1;
while(counter<Number)
{Attach Dst_Buffer as DrawBuffer; Src_Buffer = input for fragment shader;
 Draw the texture;
 Swap the Src_Buffer and Dst_Buffer;
 Increase counter;}
```

The resultant GVF field is stored in one framebuffer and is used as input parameter to the snake process fragment shader. The fragment shader for the snake process involves solving a linear system:

$$\overline{A} * \overline{X_t} = \gamma * \overline{X_{t-1}} + \kappa * \overline{V} \tag{4}$$

where $\overline{V}$ is the GVF field and $\overline{X_t}$ is the snake contour at t time, γ and κ are constants and $\overline{A}$ is a constant matrix. Note that solving the above linear system not only requires an inverse of the matrix, but also brings in the violation of the spatial locality. While the first problem can be addressed on the GPU [14], the second problem will dramatically decrease the efficiency of the GPU implementation because it results in extensive amount of I/O for texel fetch operations. However, $\overline{A}$ is very similar to a symmetric band matrix and is positive definite, with the exception that the upper right corner and lower left corner of the matrix is not zero. It can be expressed as Equation (5a).

$$\begin{bmatrix} c & b & a & 0 & 0 & 0 & \cdots & 0 & a & b \\ b & c & b & a & 0 & 0 & \cdots & 0 & 0 & a \\ a & b & c & b & a & 0 & \cdots & 0 & 0 & 0 \\ 0 & a & b & c & b & a & \cdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a & 0 & 0 & 0 & \cdots & 0 & a & b & c & b \\ b & a & 0 & 0 & 0 & \cdots & 0 & a & b & c \end{bmatrix} \quad [5a.] \qquad \begin{bmatrix} C & B & A & 0 & 0 & 0 & \cdots & 0 & A & B \\ B & C & B & A & 0 & 0 & 0 & \cdots & 0 & A \\ A & B & C & B & A & 0 & 0 & \cdots & 0 & 0 \\ 0 & A & B & C & B & A & 0 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ A & 0 & 0 & 0 & \cdots & 0 & A & B & C & B \\ B & A & 0 & 0 & 0 & \cdots & 0 & A & B & C \end{bmatrix} \quad [5b.] \quad (5)$$

where a, b are positive constant values that are much smaller than 1 and c is around 1. The major diagonal has the value of c, the second major diagonal has the value of b and the next one is a. Furthermore, its inverse matrix is of similar type. Because most of the numbers in $\tilde{A}$ are far smaller than 10^{-6} after the third diagonal, an approximation of $\tilde{A}$ can be given as Equation (5b). where:

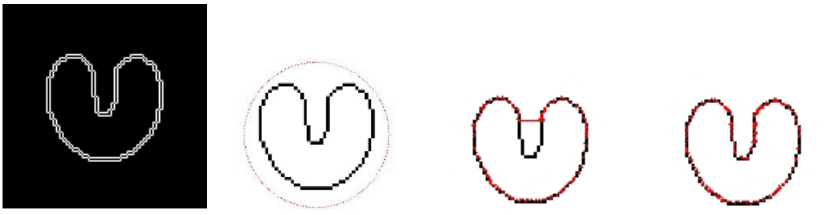
$$\begin{gathered} A = c(b^2 - ac)/\,[X]\,, \;\; B = \tfrac{-b[X] - bc(b^2-ac)(a-c)}{(c^2-b^2)[X]} \\ C = \tfrac{c[X] - c(b^2-ac)(a-b^2)}{(c^2-b^2)[X]}, \;\; [X] = (c^2-a^2)(c^2-b^2) - b^2(c-a)^2 \end{gathered} \quad (6)$$

By inserting the above equations to Equation (4), a discrete solution for the snake contour can be obtained and implemented as a shader program: $x_i^t = \sum_{k=i-2}^{i+2} M_{i,k}^{t-1} v_k$, $i \in [1, n]$, where n is the number of points in the snake contour and $M_{i,k}^{t-1}$ is the row i, column k element of the matrix multiplication result of $\tilde{A}$ and the column matrix of x_i^{t-1} and $v_k = \gamma x_k^{t-1} + \kappa f_k$.

5 Experiment Results

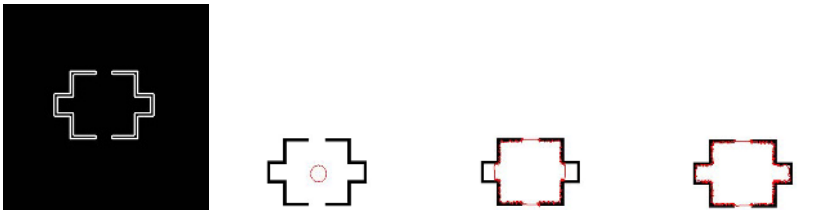
All tests were performed on a laptop PC with a 2.4GHZ Pentium processor, 1GB RAM and a Geforce Go6800 card with 128MB on-board graphics memory. Fig.3(a) shows the edge map for the U-shape image, followed by the initial curve in Fig.3(b), a partial result after 150 iterations in Fig.3(c) and the final result in Fig.3(d). The result shows that the GVF snake algorithm can contract to concave shapes where ordinary snake could not. Fig.4 shows a non-continuous room model results. These two data sets were modeled after the ones by Xu and Prince [2] to provide a better comparison. Fig.5 is an MRI brain scan which shows the algorithm can be applied to real-world data with non-uniform background and concave shape. A set of scans of human shoulder was studied at levels of image ranging from 128×128 to 1024×1024 pixel resolution in order to evaluate the scalability of the algorithm and pinpoint performance tradeoffs. The results for computations on the above data are provided in Fig.6.

In Fig.6, the GVF field calculation was performed both on the GPU and CPU, and it includes the edge detection stage for the GPU. For the snake algorithm,



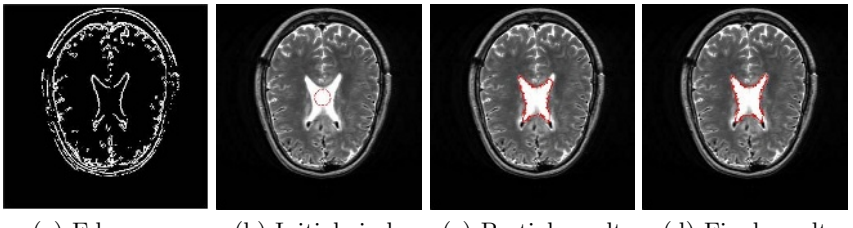
(a) Edge map (b) Initial circle (c) Partial result (d) Final result

Fig. 3. U-shape 256 × 256 pixel



(a) Edge map (b) Initial circle (c) Partial result (d) Final result

Fig. 4. Room 256 × 256 pixel



(a) Edge map (b) Initial circle (c) Partial result (d) Final result

Fig. 5. MRI 256 × 256 pixel

Data	Total time	Time for GVF	Time for Snake	Iterations for snake	CPU Time for GVF
U-shape 256	7857 ms	2677 ms	4963 ms	300	1993ms
Room 256	4022 ms	2461 ms	1368 ms	80	1862ms
MRI 256	5389 ms	2581 ms	2600 ms	160	1909ms
MRI 512	6234 ms	2503 ms	3223 ms	100	15462ms
shoulder 128	3132 ms	2527 ms	411 ms	70	446 ms
shoulder 256	5761 ms	3135 ms	1266 ms	480	1853 ms
shoulder 512	11493 ms	5961 ms	4695 ms	400	15357ms
Shoulder 1024	11596 ms	2638 ms	8958 ms	250	316073ms

Fig. 6. Benchmark on test images

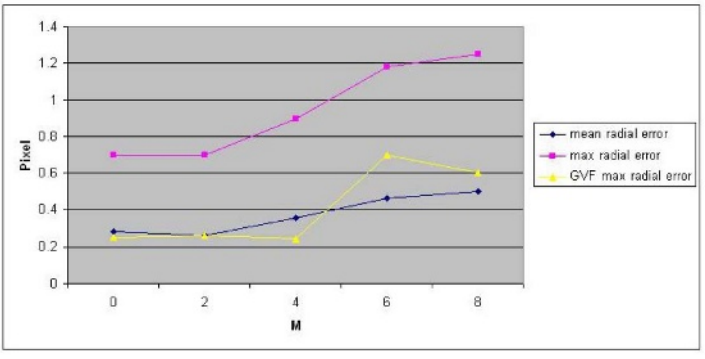
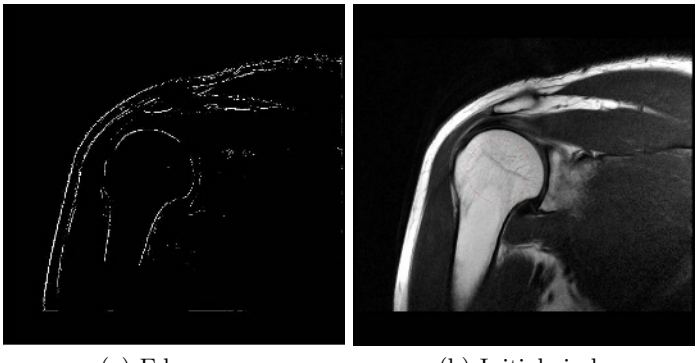
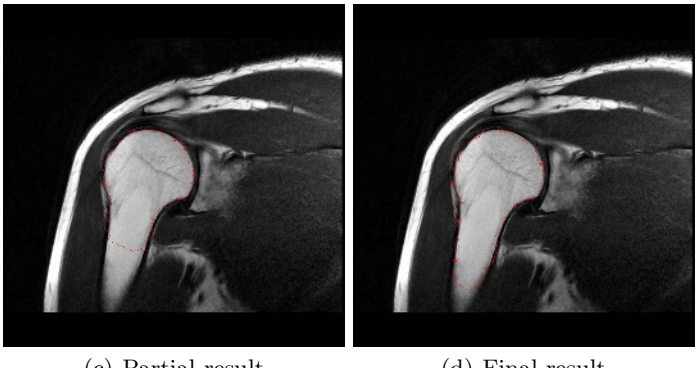


Fig. 7. MRE comparison



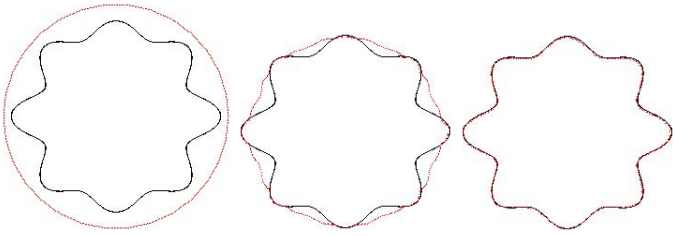
(a) Edge map (b) Initial circle



(c) Partial result (d) Final result

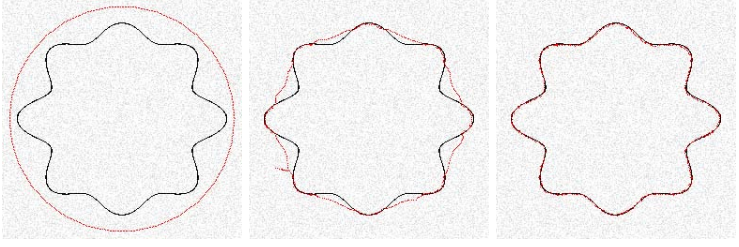
Fig. 8. Shoulder 1024×1024 pixel

it can be observed that the speed performance complexity can be expressed as $O(n^2k)$, where n^2 is the size of the texture and k is the number of iterations. This means that newer graphics cards with more texture memory will be able



(a) Initial circle (b) Partial result (c) Final result

Fig. 9. Spine



(a) Initial circle (b) Partial result (c) Final result

Fig. 10. Spine with Gaussian noise

to efficiently process larger images. For images of 256×256, the CPU is about 20% faster than the GPU. But for 512×512 image, the GPU technique starts to outperform the CPU by 4 times. Therefore, the parallel capability of GPU computing shows its advantages on larger images. Each individual GPU fragment or vertex processor is lower than the CPU. However, with the increase in data size, the GPU parallel pipeline becomes more efficient and greatly outpaced the CPU. Another observation is that texture I/O may become the bottleneck. For example, the texture fetch for the snake shader is more than two times that of the GVF shader and so the snake shader is 50% slower.

One test case is studied to analyze the accuracy of the GPU technique (Fig.9 and Fig.10). This test case uses simple harmonic curves given by: $r = a + b\cos m\theta + c$, where a, b, c are constant values and by varying the m, a set of curves can be obtained. Each image is 256×256 and we used $m = 0, 2, 4, 6, 8$. The measure of error is MRE(mean radial error), which is the mean distance in the radial direction between the final active contour and the harmonic curve. Fig.7 shows the MRE result. The blue line shows the MRE, the red line shows the maximum radial error as the worst case scenario and the yellow line shows the maximum radial error from a CPU implementation of improved GVF algorithm as stated in [21]. As we can see, the performance of CPU implementation generally has better accuracy. The reason for the performance gap is the difference in the precision of floating point data. Nonetheless, the GPU implementation still achieves a good overall accuracy and the mean errors are within sub-pixel level.

Fig.10 shows the robustness of the GPU technique with the addition of gaussian noise. The image with noise has an MRE of 0.5 while the clean image is 0.35.

6 Conclusion

A hardware accelerated gradient vector flow algorithm for image segmentation was presented. The algorithm utilizes the fragment and texture units of the GPU. A set of test cases was presented and evaluated comparing CPU and GPU results. In addition, some new features of GPGPU are exploited and some important issues involved in porting algorithms onto the GPU are specified, which provides a foundation for further exploration in this algorithm.

References

1. Kass, M., Witkin, A., Terzopoulos, D.: Snakes: Active contour models. International Journal of Computer Vision. **1** (1988) 321–331
2. Xu, C., Prince, J.L.: Snakes, shapes, and gradient vector flow. IEEE Transaction on Image Proccessing. **7** (1998) 359–369
3. Kilgariff, E., Fernando, R.: The geforce 6 series gpu architecture. In: GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation. (2005) 471–493
4. Zimmer, C., Labruyere, E., Meas-Yedid, V., Guillen, N., Olivo-Marin, J.: Segmentation and tracking of migrating cells in videomicroscopy with parametric active contours: A tool for cell-based drug testing. IEEE Transaction. on Medical Imaging **21** (2002) 1212–1221
5. Ding, F., Leow, W., Wang, S.: Segmentation of 3d ct volume images using a single 2d atlas. In: Lecture Notes in Computer Science. Volume 3765., Springer (2005) 459–468
6. Vidholm, E., Nystrom, I.: Haptic volume rendering based on gradient vector flow. In: Proceedings of Swedish symposium on image analysis (SSBA'05). (2005) 97–100
7. Rumpf, M., Strzodka, R.: Level set segmentation in graphics hardware. In: Proceedings of the 2001 International Conference on Image Processing. Volume 3. (2001) 1103–1106
8. Kondratieva, P., Krüger, J., Westermann, R.: The application of gpu particle tracing to diffusion tensor field visualization. In: Proceedings IEEE Visualization 2005(Vis'05). (2005)
9. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04). (2004) 47
10. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a gpu-based particle engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS'04). (2004) 115–122
11. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS'04). (2004) 133–137
12. Lefohn, A.E., Kniss, J., Hansen, C., Whitaker, R.: Interactive deformation and visualization of level set surfaces using graphics hardware. In: Proceedings of the 14th IEEE Visualization(VIS'03). (2003) 75–82

13. Yang, R., Welch, G., Bishop, G.: Real-time consensus-based scene reconstruction using commodity graphics hardware. In: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications (PG'02). (2002) 225
14. Galoppo, N., Govindaraju, N., Henson, M., Manocha, D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05). (2005) 3–3
15. Lefohn, A.: Gpu memory model overview. In: Proceedings of the ACM Siggraph 2004. (2004)
16. Govindaraju, N., Raghuvanshi, N., Henson, M., Manocha, D.: A cache-efficient sorting algorithm for database and data mining computations using graphics processors. In: UNC Tech. Report. (2005)
17. NVidia: The cg toolkit. In: http://developer.nvidia.com/object/cg_toolkit.html. NVidia Corp. (2005)
18. gpgpu.org: General-purpose computation on gpus. (2005)
19. Microsoft: Hlsl shaders. In: http://msdn.microsoft.com. Microsoft Inc. (2004)
20. OpenGL: Opengl shading language. In: http://www.opengl.org/documentation/oglsl.html. OpenGL.org (2005)
21. Xu, C., Prince, J.L.: Generalized gradient vector flow external forces for active contours. Signal Processing — An International Journal **71** (1998) 131–139