Fast Flow Visualization on CUDA Based on Texture Optimization

Ying Tang[1], Zhan Zhou[1], Xiao-Ying Shi[1,2], and Jing Fan[1,*]

[1] School of Computer Science and Technology,
Zhejiang University of Technology, Hangzhou,China
[2] School of Information Engineering,
Zhejiang University of Technology, Hangzhou,China
{Ytang,fanjing}@zjut.edu.cn, zhanzhanlove.hi@gmail.com,
shixiaoying8888@yahoo.com.cn

Abstract. Flow visualization plays an important role in many scientific visualization applications. It is effective to visualize flow fields with moving textures which vividly capture the properties of flow field through varying texture appearances.Texture-optimization-based (TOB) flow visuliaztion can produce excellent visualization results of flow fields. However, TOB flow visualization without acceleration is time-consuming. In this paper, we propose fast flow visualization based on the accelerated parallel TOB flow visualization which is entirely implemented on CUDA. High performance is achieved since most time-consuming computations are performed in parallel on GPU and data transmission between CPU and GPU are arranged properly. The experimental results show that our TOB flow visualization generates results with fast synthesis speed and high synthesis quality.

Keywords: CUDA, flow visualization, texture optimization, GPU parallel computing.

1 Introduction

Flow visualization is a very important branch of scientific visualization which can be applied to lots of fields, such as automotive design, meteorology, medical imaging and water conservancy.

In recent years, the rapidly developing methods of texture-based flow visualization [1-2], [6-7] not only capture the rich details of a static flow field, but also vividly show the movements of the dynamic flow field. LIC (Line Integral Convolution) [1] and spot noise [2] are two basic algorithms of texture-based flow visualization. Both methods can generate directional features of textures which are suitable for the flow field visualization. However the textures generated by these methods are blurry to some extent. In addition, most of them use random noise texture to visualize the flow field instead of the real-world pictures with rich visual patterns.

Kwatra *et al.* proposed a texture-optimization-based (TOB) flow visualization to solve the problem [3-4]. Flow fields are visualized and animated with the

* Corresponding author.

R. Huang et al. (Eds.): AMT 2012, LNCS 7669, pp. 257–267, 2012.

user-provided textures, and it produces good visualization effects. But this method involves much iterative optimization which makes it computationally expensive and limits its applicability to visualizations with the requirement of fast speed. Han *et al* [9] and Huang *et al* [19] accelerated the texture optimization by a GPU-based algorithms. Their algorithms are implemented on traditional GPU by specific GPU programming interface involving a lot of tricky GPU operations which makes them difficult and inefficient to realize. In this paper, we propose the CUDA (Compute Unified Device Architecture)-based flow visualization algorithm which is entirely implemented on GPU. CUDA is a parallel computing platform and programming model launched by NVIDIA [14]. It enables dramatic increases in computing performance by harnessing the power of GPU. It has several advantages over traditional general GPUs, such as shared memory, faster downloads and reads back to and from the GPU. Besides, the GPU program with CUDA is easier to understand and implement.

In this paper, our contribution is utilizing the powerful performance of CUDA architecture to implement a parallel algorithm of TOB flow visualization, which greatly increases the synthesis efficiency.

The rest of the paper is organized as follows: A brief overview of the related work is given in Section 2. Section 3 describes the TOB flow visualization in detail. In Section 4, we describe how to use CUDA to realize the parallel algorithm of TOB flow visualization. Experimental results are presented and discussed in Section 5. Section 6 concludes the paper with suggestions for future work.

2 Related Work

Texture-based flow visualization plays an important role in the flow visualization field. Most of previous methods were based on Line Integral Convolution (LIC) [1, 5]. Van Wijk [6] extended LIC-based visualization with texture animation to visualize the flow field. These methods adopt the random noise as visual texture patterns. It can depict rich details of the flow field effectively but is not suitable for visualize more complex flow field which comes from the real-world flow, like water, cloud and fire etc [9].

In recent years, some methods have been proposed to visualize flow field by using the real-world textures. Zhang *et al.* proposed the progressively-variant texture technique which synthesized the textures according to the user-specified vector fields [10]. Taponecco *et al.* presented the steerable texture synthesis technique [11] in which several samples with different rotations and scales are used to generate textures with the same vector filed. A similar approach was proposed by Lefebvre *et al* [12]. And Fisher *et al* .extended this method to three dimension surface flow field [23]. However, all of the above methods were applied only to static flow fields and were not extended to deal with the dynamic flow fields. Using the animated sequence of general textures to visualize the flow on 3D surfaces flow is a rising technique in recent years. It deals with both the static and dynamic flow field, and is more effective than the noise methods. The consecutive animated sequences of textures were generated by warping the texture image according to the surface flow field in [6-7]. However, the textures are made to stretch, compact or twist during the process, especially

in the vicinity of singularity (such as sink and source) in flow field. This method would produce the results which appear different from the sample texture. Lefebvre *et al* [4] solved the problem by re-synthesizing the large deformed texture region, which ensured the similarity between the synthesis results and sample texture but did not ensure the consistency between the consecutive frames.

Kwatra *et al* [3] presented a global optimization solution algorithm to address this issue. It added a correction step after warping the former frame to keep the consistency and similarity. The correction step adopted the energy minimization function. This technique is capable to generate good visual appearances, but with slow computation speed. Han *et al* [9] accelerated it by a GPU-based parallel algorithm. However, the algorithm based on traditional GPU involves many programming optimization tricks which make it difficult to implement. In this paper, we study and implement the TOB flow visualization algorithm based on CUDA architecture.

3 Texture Optimization-Based Flow Visualization

The texture sequences are synthesized to keep consistency and similarity, namely the textures not only flow in a coherent fashion (consistency) but also maintain their structural elements (similarity) under the control of flow field. Fig.1 shows two criteria to be satisfied the texture sequences are synthesized: 1) Flow Consistency: the motion of synthesized textures should be in accordance with the flow; 2) Texture Similarity: the visual appearance of target textures should be similar to the source.

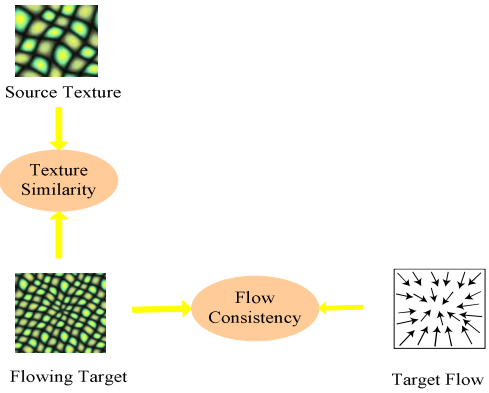


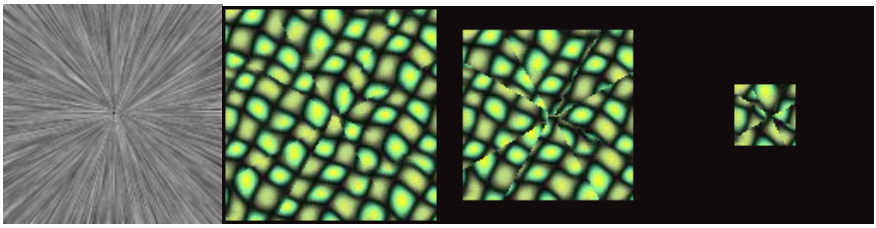
Fig. 1. Two criteria for texture optimization-based flow visualization

Kwatra *et al* [3] proposed that textures are synthesized in a frame-by-frame fashion to visualize the flow and two steps are performed when a new frame is synthesized. At first, the warp step warps the former frame according to the flow field. After the first step, a correction step is adopted to make the warped frame look similar to the exemplar. The two steps are described in the following subsections.

3.1 Warp Step

Let X_{i+1} denote the synthesized texture and X_i denote its former one. Let f_i denote the flow field function for frame X_i.

To get next frame X_{i+1}, X_i is first warped by flow function f_i and get the result $fi(Xi)$. In order to get the pixel value q of warped frame, we start from q backtracks along with the flow field to find p which means $q=f_i(p)$. Fig.2 shows the result of the warp step.



(a) (b) (c) (d)

Fig. 2. Result of warp step. From the pictures we can see that warp step can keep the flow consistency but can't keep the texture similarity. Indeed it will disappear at last along the sink flow.

3.2 Correction Step

Fig.2 demonstrates that when only use the warp step the synthesized sequences will not maintain the similarity of the texture just the flow consistency. So after warp step the correction step should be applied.

In correction step the synthesized frame needs to be close to the warped frame to ensure flow consistency and also be similar to the source texture to ensure texture similarity. Correction step is casted as an optimization problem where the energy is defined for maintaining each goal of correction step.

To keep the flow consistency, flow energy is defined to make sure the synthesized frame as close as the warped frame. And to keep the texture similarity, texture energy is defined to make sure the synthesized frame as similar as the source texture. Equation (1) (2) (3) shows the flow energy, texture energy and the energy sum of them.

$$E_f(X) = \| x - w \|^2 \tag{1}$$

$$E_t(X) = \sum_{p \in X^+} \| x_p - z_p \|^2 \tag{2}$$

$$E(X) = \sum_{p \in X^+} \| x_p - z_p \|^2 + \lambda \| x - w \|^2 \tag{3}$$

$E_f(X)$ is computed as the squared differences between the warped frame and the synthesized frame. And $E_t(X)$ is computed as the total of all individual neighborhood

energy where individual neighborhood energy refers to the squared difference between the neighborhood of one pixel in the subset of synthesized texture X^+ and its nearest neighborhood in the source texture. X refers to the synthesized texture, X^+ refers to the subset of X , x is the vectorized X, w is the vectorized warped texture, x_p is the neighborhood of p , z_p is the vectorized pixel neighborhood in Z (source texture) who is most similar to x_p,$\lambda$is a relative weighting coefficient.

The problem to visualize flow fields is formulated to minimize the energy $E(X)$. A EM-like algorithm is applied to solve the optimization problem.

EM-like iterative algorithm is divided into two steps: E-step and M-step. In E-step, according to the energy minimum requirement, the output pixels' neighborhoods $\{x_p\}$ are updated while neighborhoods $\{z_p\}$ are unchanged; in M-step, $\{x_p\}$ are fixed while $\{z_p\}$ are updated. E-step and M-step are processed in turn, and the optimized global solution is obtained until it is convergent or reaches the specific iterative number. Table 1 shows the procedure of correction step which are derived from [3].

Table 1. pseudo-code of correction step (Algorithm 1) and TOB flow visualization (Algorihthm 2)

Algorithm 1. Correction Step

$z_p^0 \leftarrow$ random neighborhood in Z $\forall p \in X^+$

for iteration n=0: M **do**

$x^{n+1} \leftarrow \text{argmin}_x [\sum_{p \in X^+} \| x_p - z_p^n \|^2 + \lambda \| x - w \|^2]$ //E step

$z_p^{n+1} \leftarrow \text{argmin}_v [\| x_p - v \|^2 + \lambda \| y - w \|^2]$

// M-step , v is neighborhood in Z and y is the same as x

//except for neighborhood x_p which is replaced with v

If $z_p^{n+1}=z_p^n \ \forall p \in X^+$ **then**

$x=x^{n+1}$ **break**

end if

end for

Algorithm 2. Texture Optimization-Based Flow Visualization

for i=1:N

$X_{i+1} \leftarrow f_i(X_i)$ //warp step

for j=1:K // K is the consecutive neighborhood number

CorrectionStep(X_{i+1},f_i) //correction step

end for

end for

3.3 Flow Visualization

Let $f = (f_1, f_2, ..., f_{N-1})$ denote the input flow fields and $(X_1, X_2, ..., X_N)$ denote the texture sequences being synthesized, X_1 is the initial frame which can be synthesized by texture optimization [3]. During synthesis of frame $i+1$, $f_i(X_i)$ is the warped texture

which is the result of Warp step, then the Correction step is used to synthesize frame i+1. Table 1 shows the algorithm of TOB flow visualization.

4 Fast TOB Flow Visualization on CUDA

In [3], the hierarchical tree search is utilized in M-step which makes the EM-like algorithm time-consuming and the least squares solver is used in E-step which makes the result blurry to some extent. To solve the problem, paper [9] incorporated k-coherence [8] into both E-step and M-step of the original EM solver in [3]. And in our implementation, we have used a CUDA-friendly *k*-coherence algorithm [21] to accelerate the flow visualization speed. The warped step and correction step are also implemented on CUDA architecture system.

In CUDA a thread hierarchical model is used which is: "thread" $\rightarrow$ "thread blocks" $\rightarrow$ "thread-block grid". Each thread block contains a certain number of threads. A thread block grid can be divided into multiple thread blocks.

Our method can be divided into two phrases: Warp and Correction step. In order to fully utilize CUDA, we design the algorithm deliberately, including data transfer between CPU and GPU, allocating graphic memory appropriately and designing the dimension of both block and grid carefully.

When synthesizing a new frame before Correction step, the warped frame and initial new frame are obtained by warping the former frame. Before warping, the number of blocks is set to be the size of X_{i+1}, and the number of threads in each block is 1, such that each block processes one pixel. X_i and the flow field are bounded with texture memory.

4.1 Correction Step

During correction step, the basic step is divided into E-step and M-step. Table 2 shows correction step which incorporates the parallel k-coherence.

Table 2. Pseudo-code of correction step with *k*-coherence

Algorithm 3. Correction Step with *k*-coherence

$z_p^0 \leftarrow$ random neighborhood in $Z \ \forall p \in X^+$

for iteration n=0: M **do**

$x^{n+1} \leftarrow \arg\min_{x, x(p) \in k(p)} [\sum_{p \in X^+} \| x_p - z_p^n \| + \lambda \| x - w \|^2]$ //E -step

$z_p^{n+1} \leftarrow \arg\min_{v, v \in k(p)} [\| x_p - v \|^2 + \lambda \| y - w \|^2]$ // M-step

if $z_p^{n+1} = z_p^n \ \forall p \in X^+$ **then**

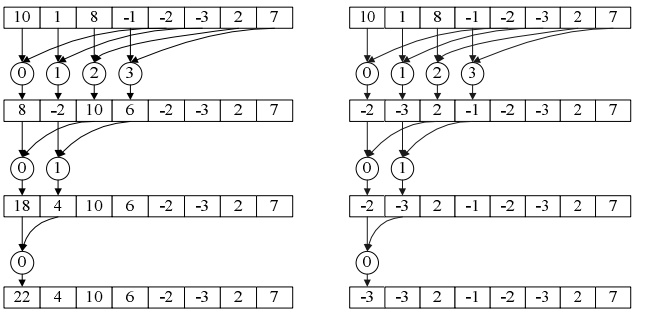
$x = x^{n+1}$ **break**

end if

end for

In M-step, the CUDA-friendly *k*-coherence search is used to solve the neighborhood searching problem. For each q in the synthesizing frame, candidate set $c(q)$ is built by the union of the neighborhood's similar set of q. And then search in $c(q)$ to find the neighborhood that closet to q's neighborhood using an improved parallel reduction algorithm. The searched nearest neighborhood is saved in the corresponding array. Since each output pixel is independent on any other output pixels, the number of blocks is equal to the size of X^+, and the number of threads in one block is equal to the number of candidates. We copy the sample texture, pre-computed similarity set, warped texture ($f_i(X_i)$) and X_{i+1} to texture memory (due to un-changing of them during the whole M step) , use shared memory and registers to complete parallel *k*-coherence search and copy result back to host, since access to shared memory is much faster than to global memory. In the kernel function of M-step kernel we pre-cache the *k*-coherence candidate set in shared memory.

The adoption of improved parallel reduction is to avoid time expensing of data transfer between CPU and GPU and improve the efficiency of the whole execution processing. The idea of parallel reduction is similar to the merging sort algorithm. Fig.3(a) shows the procedure of reduction algorithm in one block. In our reduction process, the plus operation is replaced by compare operation. In thread block, each thread processes the data respectively; the former half threads compare the first half data in data set with the second half, and the smaller ones are moved to the first half of data set. After one process, the number that needs to be compared is half down, and all of them are stored in the first half of data set. After certain iteration, the lowest one in data set is the first item .To all blocks and all threads in the blocks this process is parallel and the number we need to compare is constant, so the parallel reduction has constant time complexity. Fig.2 (b) shows the improved parallel reduction.



(a) Parallel reduction (b) improved parallel reduction

Fig. 3. Parallel reduction theory

After M-step, E-step follows. During E-step, in order to minimize the energy function (3), we copy the warped texture ($f_i(X_i)$) to the texture memory. In E-step we also use CUDA-based *k*-coherence search to find $\{x_p\}$. The number of blocks equals to the

size of X_{i+1} and the threads number in each block is equal to the candidate number. The allocated *uchar4* global array *cu_CorrectCoord* has the same size as that of X_{i+1}. Since in E-step the value of $\{z_p\}$ does not change, we copy *cu_MatchCoord*, pre-computed similarity set and $f_i(X_i)$ to texture memory.

The E-step kernel function executes as follow:

1. Declare an *int4* array *distance* and an *uchar2* array *kcoh* in the shared memory. Since when calculating the energy, an average value will be accessed by all threads, a *float4* type variable *ref_color* should be declared in the shared memory. And another *float4* type shared variable *ref_color_con* is needed as the warped color. Then according to the similarity set, compute candidate coordinates and store them in *kcoh*.
2. According to *blockIdx.x* and *blockIdx.y* get the position *p2* of the pixel being synthesized. According to *p2* the average value *ref_color* and the constrained color value *ref_color_con* is calculated. Then to the current thread we get the corresponding candidate *p1*.The energy value among *p1*, *ref_color* and *ref_color_con* is calculated, it is stored in *distance*.
3. Synchronize all operations above. After that we use the improved parallel reduction algorithm to search the candidate point with lowest energy. This candidate point will be stored to corresponding position in *cu_CorrectCoord*.

At last *cu_CorrectCoord* is copied back to host.

5 Results

We have implemented our algorithm with the following experimental hardware configuration: CPU: Intel Core™ 2Duo 2.83 GHz, Memory: 2GB, Graphics card: GeForce 8800 GTX. The algorithm is developed in Visual Studio 2008 with CUDA4.0.

In our CUDA implementation, we use successive neighborhood sizes of 16*16 and 8*8 pixels at each frame, and perform 1~2 iterations for each neighborhood.

We follow the method in [15-16] to define several user-specified flow fields.

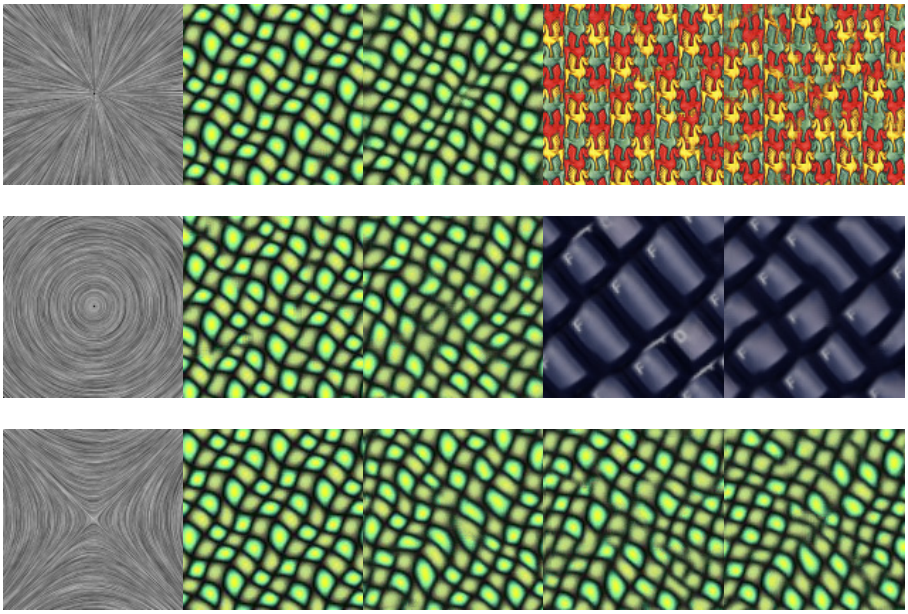
Similar to the pre-computation in [9], for *k*-coherence search, the similarity set is pre-computed for each neighborhood. During pre-processing, an improved CUDA-based KNN search [17, 21] is used.

Fig.4 presents the results of our TOB flow visualization on CUDA, which shows that our technique is able to achieve high quality flow visualization by keeping consistency and similarity.

From Table 3, we can see that our method is almost 200+ faster than the original method and 10 times faster than the discrete solver [9]. We utilize the powerful parallel computation capacity of GPU under CUDA to implement the accelerated E-step and M-step, which makes the execution time of TOB flow visualization reduce one order of magnitude compared with previous methods.

Table 3. Synthesis Time for Different Synthesis Schemes

	Flow-guided synthesis (256^2—>256^2) (each frame)
Original texture optimization[3]	20~60s
Discrete texture optimization[9]	>1.2s
Texture optimization-base flow visualization on CUDA	180ms~200ms



(a) (b) (c) (d) (e)

Fig. 4. The results of flow visualization. Pictures of column (a) are flow fields. For the first row column (b) gives frame1, and column (c) is frame 20. (d) is frame 1 and (e) is frame 20. The second row is similar to first. At the last row, from (b) to (e) are frame 1, frame 10, frame 50 and frame 100, respectively.

6 Conclusion

We adopt the *k*-coherence parallel algorithm to the CUDA-based algorithm, which greatly accelerates the Correction step. Experimental results show that our CUDA-based flow visualization algorithm can not only achieve the similarity and consistency goals but also produce texture sequence at high speed.

In our future work, we will continue to study CUDA parallel techniques, and further explore how to use the general texture to achieve efficient flow visualization which can vividly simulate the feature details of the flow and has a higher speed.

Acknowledgements. This work is supported by National Natural Science Foundation of China (61173097, 61003265), Zhejiang Natural Science Foundation of China (Z1090459), Zhejiang Science and Technology Planning Project of China (No. 2010C33046), and Tsinghua–Tencent Joint Laboratory for Internet Innovation Technology.

References

1. Falk, M., Weiskopf, D.: Output-Sensitive 3D Line Integral Convolution. IEEE Transactions on Visualization and Computer Graphics, 820–834 (2008)
2. Falk, M., Seizinger, A., Sadlo, F., Uffinger, M., Weiskopf, D.: Trajectory-Augmented Visualization of Lagrangian Coherent Structures in Unsteady Flow. In: 14th International Symposium on Flow Visualization (2010)
3. Kwatra, V., Essa, I., Bobick, A., Kwatra, N.: Texture Optimization for example-based synthesis. ACM Transactions on Graphic 24(3), 795–802 (2005)
4. Wei, L.Y., Lefebvre, S., Kwatra, V., Turk, G.: State of art in example-based texture synthesis. In: Eurographics 2009, State of the Art Report, EG Association (2009)
5. Cabral, B., Leedoml, C.: Imaging vector fields using line integral convolution. In: Proceedings of ACM SIGGRAPH 1993, p. 263. ACM, New York (1993)
6. VanWijk, J.J.: Imagebased flow visualization. ACM Transactions on Graphics 21(3), 7 (2002)
7. Rasmussen, N., Enright, D., Nguyen, D., Marino, S., Sumner, N., Geiger, W., Hoon, S., Fedkiw, R.: Directable photorealistic liquids. In: 2004 ACM SIGGRAPH. Eurographics Symposium on Computer Animation, pp. 193–202 (2004)
8. Lefebvre, S., Hoppe, H.: Parallel controllable texture synthesis. ACM Transactions on Graphic 24(3), 777–786 (2005)
9. Han, J., Zhou, K., Wei, L., Gong, M., Bao, H., Zhang, X., Guo, B.: Fast example-based surface texture synthesis via discrete optimization. The Visual Computer 22(9), 918–925 (2006)
10. Zhang, J., Zhou, K., Velho, L., Guo, B., Shum, B.Y.: Synthesis of progressively-variant textures on arbitrary surfaces. ACM Transactions on Graphics 22(3), 295–302 (2003)
11. Taponecco, F.: Steerable texturesynthesis. In: Proceedings of Eurographics, pp. 57–60 (2004)
12. Lefebvre, S., Hoppe, H.: Appearance-space texture synthesis. ACM Transactions on Graphics 25(3), 541–548 (2006)
13. Yu, Q., Neyret, F., Bruneton, E., Holzschuch, N.: Scalable real-time animation of rivers. Computer Graphics Forum (Proceedings of Eurographics 2009) 28(2) (March 2009)
14. CUDA Programming Guide ver. 1.0, NVIDIA (2007)
15. Zhang, E., Mischaikow, K., Turk, G.: Vector field design on surfaces. Tech. Rep. 04-16, Gerogia Institute of Technology (2004)
16. Chen, G., Kwatra, V., Wei, L.Y., Hansen, C.D., Zhang, E.: Design of 2D Time-Varying Vector Fields. IEEE Transactions on Visualization and Computer Graphics (2011)
17. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–6 (2008)
18. Huang, H., Tong, X., Wang, W.: Accelerated parallel texture optimization. Journals of Computer Science and Technology 22(5), 761–769 (2007)
19. CUDA C Best Practices Guide v4.0, Navidia (2011)

20. Laramee, R.S., Hauser, H., Doleisch, H., Vrolijk, B., Post, F.H., Weiskopf, D.: The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. Proc. Computer Graphics Forum 23(2), 203–221 (2004)
21. Tang, Y., Shi, X., Xiao, T., Fan, J.: An improved image analogy method based on adaptive CUDA-accelerated neighborhood matching framework. Vis. Comput. 28, 743–753 (2012)
22. Van Wijk, J.J.: Spot noise-Texture Synthesis for Data Visualization. Computer Graphics (Proceedings of ACM SIGGRAPH 1991) 25, 309–318 (1991)
23. Fisher, M., Schroder, P., Desbrun, M., Hoppe, H.: Design of tangent vector fields. ACM Transactions on Graphics 26(3), 56:1–56:9 (2007)