Hardware Accelerated HyperLIC

Robert Hero and Alex Pang
rghero@cse.ucsc.edu and pang@cse.ucsc.edu

Computer Science Department

July 2007
Technical Report No. UCSC-SOE-08-21
School of Engineering, University of California, Santa Cruz, CA 95064

This paper presents hardware acceleration techniques for both the LIC and HyperLIC algorithms using off-the shelf graphics hardware. The methods used in this paper take the burden of calculation from the CPU and place it on the graphics card through the use of programmable fragment shaders and texture maps. Frame rates increased almost 2 orders of magnitude over the corresponding software implementations of Fast LIC and HyperLIC algorithms. The methods presented here can also be extended to 3D with similar performance gains.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques;

Keywords: hyperstreamlines, LIC, hardware acceleration, fragment shader, pixel shader, symmetric tensors,

1 INTRODUCTION

Visualization of vector and tensor fields is a challenging task. Recently, a significant improvement in dense 2D flow visualization was realized using image based flow visualization (IBFV) [12]. This work was further extended to handle 3D flow fields and also took advantage of hardware acceleration [11]. On the other hand, tensor visualization remains a challenging task both in terms of showing myriad of relationships and features in a tensor field, as well as generating the visualizations reasonably fast to allow user interaction.

There are some basic approaches for visualizing tensor fields including the use of hyperstreamlines [3], glyphs [8, 6], volume rendering [7], and more recently with HyperLIC [16], as well as topological analysis [4, 17]. The glyph based approach shows the multifacet nature of tensor fields, but does not provide a continuous depiction. Hyperstreamlines provide a continuous representation of the underlying field, but cannot effectively show the relationships of the different eigenvector fields without too much clutter. Both volume rendering and HyperLIC approach shows a continuous representation but only for a few special parameters of the tensor field e.g. anisotropy. Topological analysis is a promising approach but is still relatively in its infancy and therefore not quite ready for optimization.

In this paper, we investigate how dense visualization such as LIC and HyperLIC can benefit from hardware acceleration. LIC [**?**] is an extensively researched technique for flow visualization which numerous software optimizations and enhancements e.g. [10, 9, 13]. Even with more advanced versions of LIC such as FastLIC [10], images are generated on the order of seconds for 2D images. The methods we present here are based on the traditional LIC algorithm, but implemented to take advantage of modern GPUs.

Two of the methods presented are identical to the software versions, but are adapted to run completely on the graphics hardware. The other methods presented are based on the LIC algorithm, and run completely on the graphics hardware as well. By storing the tensor field data on the graphics card, we were able to use the programmable shaders to perform the LIC calculations. Compared to the traditional LIC algorithm, the hardware accelerated method performed two orders of magnitude better than the software version. Hardware accelerated LIC achieved over 33 frames per second for 512 x 512 images. The hardware accelerated HyperLIC algorithm generated frame rates of 22 fps on an nVidia GeForceFX 5950 for 512 x 512 pixel images. These methods can be extended to 3D cases with almost no change to the 2D implementation, and with substantial performance gains over their corresponding software versions as well. The focus of this paper will be on the 2D methods, however a discussion of the 3D implementation will be presented.

2 RELATED WORK

The HyperLIC method [16] combined the idea of hyperstreamlines [3] and LIC [2] to visualize symmetric tensor fields. The basic LIC algorithm is well know and is given by the equation

$$I(x,y) = \int_{-L/2}^{+L/2} k(i)N(P_i)di$$

where L is the integration length, $N(P)$ is the noise texture at location P, P_i is a location along the streamline centered at x, y, $k(i)$ is the filter kernel, and $I(x,y)$ is the pixel in the output image at x, y. This integral is simply the streamline generated by the vector field in 2D from point at x, y. By convolving this streamline with a noise image, an image showing the structure of the vector field is generated. The output image is generated pixel by pixel with this convolution.

The 2D HyperLIC algorithm takes the formula used in the LIC algorithm and applies it to a 2D symmetric tensor field. The HyperLIC algorithm uses a modified formula of

$$I(x,y) = \int_{-L/2}^{+L/2} \int_{-L/2}^{L/2} k(i,j)N(P_{ij})didj$$

to generate the resulting image. The outer integral represents the convolution along the major eigenvector and the inner integral is along the minor eigenvector. The area spanned by this convolution of the major and minor eigenvectors highlights the anisotropy of the tensor. The performance of the 2D HyperLIC algorithm is improved by using a two pass method. Rather than computing the

double integral for each pixel in the target image, the algorithm can first generate a LIC image using the major eigenvector, then use the resulting image as the input noise image for another pass with the LIC algorithm using the minor eigenvector field. This approach generates a useful image for the visualization of anisotropy in symmetric tensor fields, but takes twice the amount of time to generate images compared to standard LIC.

HyperLIC is by no means the only technique that uses textures to visualize tensor fields. Other notable methods include transforming the tensor field into a positive definite metric tensor and then using LIC to independently render each eigenvector field, and then blending them together [5]. Two variations of volume rendering tensors have also been proposed. In [1], tensors are represented as 3D gaussians where they are volume rendered using texture mapping hardware. On the other hand, in [14], symmetric positive definite tensors are represented by flat, transparent, planar glyphs which are then volume rendered through a technique called splatting.

Since the publication of the original LIC algorithm, there have been numerous papers describing how to enhance and speed up the algorithm. There have also been efforts in using hardware techniques to accelerate the LIC algorithms. In [15], a hardware accelerated method for generating LIC images is presented. By storing LIC images generated by each step along the integral in the accumulation buffer and using blending operations to update a vector field texture they were able to achieve frame rates of approximately 3 fps. The difficulty with this method is that it requires constant swapping of texture memory on the graphics card which can hinder performance. Rather than using a multipass technique to achieve the integration along the streamline, we program the GPU to calculate the streamline for each pixel during the rendering. For tensor fields, we use a two pass rendering just as the 2D HyperLIC method uses two passes of the LIC algorithm to generate images.

Similar efforts in hardware acceleration of IBFV [12] have also been proposed. Examples include [11, 15]. These methods use graphics hardware to deform rendering primitives in the direction of the flow. Multipass rendering is used to achieve the final image, which is an accumulation of texture images.

The methods presented in this paper differ from previous hardware accelerated approaches as the calculations for each pixel in the output image are calculated on the graphics card. In contrast, other approaches have focused on using fewer calculations and blending the results with other intermediate images. We also present methods that blur the image along the streamlines local to each output pixel in an attempt to reduce the number of calculations needed to generate the streamlines. The image is blurred through several rendering passes. Each pass uses the previously generated image as the noise image, where the first pass uses a randomly generated white noise image.

3 METHODS

3.1 Hardware LIC

Adapting the LIC algorithm to run solely on a modern GPU requires the ability to program the GPU and to transfer the needed data to the GPU efficiently. Fragment shaders provide the ability to program the GPU and texture maps are used to quickly send the important data to the graphics card. Most modern graphics cards support these features.

The Hardware LIC algorithm is the same on the graphics card as it is in software. The main difficultly in adapting LIC to run on the GPU is how to transfer the data to the GPU in a way that it can efficiently access the vector field. Each pixel in the output image will need to calculate the streamline that originates from that position. The information is contained in the vector field and thus the entire vector field must be transferred to the GPU. A two dimensional vector field consists of an NxN grid of two scalar values that represent each vector component. The total size of each vector field used in this paper is 512x512x2x4 = 2 megabytes. The vector field is converted into an NxN texture map where the two components of the vector are stored in two of the color channels of the texture map. In this case, we used the red and green channels and left the blue and alpha channels free for other information. In Figures 3(b) and 4(b), we used those extra color channels to make color images. A colormap was stored in the texture memory and the extra color channels stored an offset into the colormap. For the LIC images the index was simply the magnitude of the vector centered at each pixel, while the HyperLIC images used the ratio between the major and minor eigenvectors. This ratio highlights regions of isotropy as green and anisotropy as cyan.

Texture maps are traditionally clamped to values between 0.0 and 1.0. There are two issues with this limitation, the first being that the vectors used ranged from -1.0 to 1.0. The solution to this was to store each value in the texture as

$$texturevalue = vectorvalue/2.0 + 0.5$$

Then in the fragment program one line of code reversed this transformation. The other limitation of the texture map was in the case of vectors outside the range of -1.0 to 1.0. This is not typically a problem, but if it is, the solution is to use two color channels per value. The first channel stores a value just as before, and the second channel stores a scaling factor. Again adding one line in the fragment shader can then invert this value back to its correct value. The actual vector value is calculated by

$$realvalue = texturevalue * 1.0/scalefactor$$

Another potential problem is the precision of the texture map values. Texture maps in OpenGL is typically limited to 8 bit precision per color channel. Precision has not been a large issue in our hardware LIC implementation. If it was, it can be solved by using DirectX, or vendor specific OpenGL extensions that allow more bits per channel.

Once the vector field has been copied to the GPU as a texture map, each pixel is able to access the vector field at any given location through a texture lookup operation. The output pixel value is calculated from the summation

$$\sum_{i=-L/2}^{L/2} N(P_i)k(i)$$

where

$$P_i = P(i-1) + V(P(i-1))\Delta t$$

and $V(P)$ is the vector defined at point P. Each pixel will perform $2L+1$ texture lookups, where L is the number of integration steps in each direction. For each step there is one lookup into the vector field texture and one lookup into the noise texture. The additional texture lookup is for the noise image at the center of the streamline.

3.2 Hardware HyperLIC

The Hardware HyperLIC algorithm is again the same algorithm as the software version. The algorithm performs the first pass of the algorithm in the same manner as in the Hardware LIC, then uses that image as the noise image and performs a second pass. During the first pass the noise image is randomly generated white noise. This image is convolved with the major eigenvector field of the tensor data. The output of this pass is then switched with the noise image and the vector field used is changed to be the minor eigenvector field. The second pass performs the LIC algorithm with the

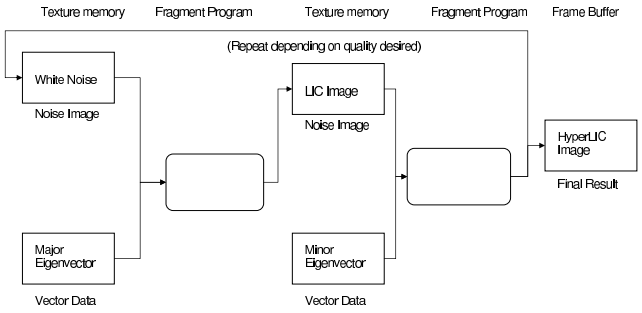


Figure 1: Overview of Hardware Accelerated HyperLIC

updated data to generate the output image. The ordering of vector fields does not matter and can be reversed without noticeably changing the final image.

However, the operation of rendering an image to the framebuffer, transferring data out of the framebuffer, and then back to texture memory is expensive. This transfer reduces the performance significantly, as there is no reason to transfer an image out of the graphics card, only to place it right back on the card. Therefore, the render-to-texture feature of the GPU was utilized to render the first pass directly to texture memory, and then switching the texture address in the fragment shader between the just calculated LIC texture and the noise texture. Using render-to-texture allows for a minimal use of the CPU in the calculation of the HyperLIC image. In fact, the only use of the CPU is to switch which texture maps are being used. Three textures need to be changed in between each pass- the one to render to, the noise texture, and the vector field itself, as it needs to use the minor eigenvector instead of the major eigenvector used in the first pass. All of these textures are stored on the graphics card at the beginning of the rendering to eliminate any performance loss caused by transferring data between the CPU and the GPU.

3.3 Neighboring pixel based LIC and HyperLIC

3.3.1 One Neighbor

The main cost of the LIC algorithm is from the calculation of streamlines. Using the idea of a multipass rendering and diffusion, we seek to eliminate the costly streamline calculation from the rendering. The idea is to calculate for each pixel in the output image, which neighboring pixels the streamline passes through. With these neighbors we blur the image along the streamline for each output pixel and repeat the process several times. This results in much less computation per pixel than the LIC algorithm, but at the cost of several intermediate renderings.

A streamline generated at any pixel will cross through two neighboring pixels. The location of these two pixels is calculated in the preprocessing step. A side effect is that time-varying vector fields will not benefit from this approach, since the neighbors will change after each time step. For the HyperLIC version, we just include the neighbor information for both the major and minor eigenvectors, resulting in four neighboring pixels. The number of neighbors that need to be calculated is reduced in half, as the two neighbors for each streamline are always separated by 180 degrees. The location of these neighboring pixels are loaded into a texture map and passed to the GPU instead of the actual vector field data. Next, for each pixel, the fragment shader retrieves the neighbor information with one texture lookup for both the LIC and HyperLIC versions. Only one lookup is needed as both the major and minor eigenvector neighbors can be stored in one texel. The shader performs three more texture lookups, five for the HyperLIC version, into the noise image and averages the values to get the resulting output value. Using render-to-texture, this method can achieve real-time frame rates. There are some serious limitations though. Since we only look at one neighbor, streamlines will have an angle of $0, 45,$ or 90 degrees as in Figure 5(b). This angular aliasing of the vectors results in poor images that ignore many fine detail in the vector/tensor field. We tried two possible solutions to this problem.

3.3.2 Weighted Neighbor

The first solution still only used the eight neighbors immediately around a given pixel. Rather than calculating which two of the eight neighboring pixels a streamline crosses, a weight is assigned to all eight of the neighbors. The weight is an indication of how close the streamline comes to crossing through a particular neighboring pixel. The idea being that if a streamline crossed directly through the center of a pixel, that pixel would be assigned a weight of 1.0, and the pixels next to that one would be assigned 0.0.

Weights are calculated by applying a function to the dot product of the streamline and the vector formed by the origin of the streamline and the center of each neighboring pixel. The function used is best chosen to be gaussian with a small standard deviation. Larger gaussians result in the image becoming very blurry. However if the standard deviation is too small there are two issues that can occur. The first problem is that no weights are assigned to any neighboring pixels. This occurs when the stream line passes in between the center of two pixels. The other issue is that the weighted neighbor looks much like the approach based on one neighbor. A good choice is a gaussian with a width of slightly more than $\sqrt{2}$. This results in most of the weight being assigned to no more than two neighboring pixels.

This method produced much better results than the one neighbor LIC and HyperLIC methods as shown in Figure 5 and 6. Since we are no longer limited to only one neighbor on each side of the streamline, we get smoother results for streamlines that curve, and less abrupt changes in direction. The downside is that most regions will appear more isotropic than they would if rendered using traditional HyperLIC.

This method is more costly than the single neighbor based LIC. Since each neighbor pixel has a weight associated with it, there are many more lookups into the noise texture map. For each pixel, two lookups into a texture storing the neighbor weights are needed to get the weights of the eight neighbors, along with nine lookups into the noise texture. The amount of texture lookups needed for this method greatly reduces the performance. Once the fragment shader has performed all the lookups for one pass it is a simple calculation to get the resulting output pixel value. This process is repeated just as in the one neighbor approach, to blur the images and show the streamlines in more detail.

3.3.3 Two Neighbor

The other approach to solve the aliasing issue with the one neighbor LIC method was to look at the next two neighbors in each direction along the streamline. This expands the neighborhood from nine pixels to twenty five pixels. The idea was to keep the amount of texture lookups low, and to reduce the aliasing.

The number of neighbors used with this approach required a small change to the neighbor lookup texture map. Now there was far more information needed than could be encoded in four color channels, so a three dimensional texture map was used. This increased the amount of data that could be used significantly. The idea is the same as the one neighbor approach. The neighbor information is looked up in the texture map, and then used as indices to lookup the values in the noise texture map. The only difference is the number of lookups. The fragment shader has to do a total of seven texture lookups for each output pixel in the LIC version and eighteen for the HyperLIC version. The amount of texture lookups

for the HyperLIC version is quite high, even when compared with the weighted neighbor approach, but compared to the number of lookups needed for weighted neighbor with a 5x5 neighborhood, it is much lower. Most of those lookups for the two neighbor approach are into the noise texture. The number of texture lookups in the HyperLIC version reduces performance substantially compared to the LIC version.

This method created slightly smoother streamlines, but the gain was small when comparing to the one neighbor based approach. The performance of this method is mainly influenced by the number of texture lookups. Since there are so many texture operations, the algorithm is only marginally faster than the hardware LIC and HyperLIC methods, with much lower image quality.

3.4 Extension to 3D

All of these methods can be extended to support a 3D tensor or vector space with only a few changes. The 3D version of the LIC and HyperLIC algorithm is similar to the software algorithm. The 3D versions of the neighbor lookup based methods only require more neighbors to be stored. The need to store more data on the graphics card is the limiting factor in these methods.

3.4.1 3D LIC

Extension to 3D LIC is the easiest method to implement. The only change in the LIC algorithm is to change the input data to be a 3D texture. Once this is done the calculations along the streamline are done in the same manner as the 2D case. This should result in a 3D volume, but the graphics cards are designed to only render to a 2D plane. The resulting 3D LIC volume must be calculated one slice at time and then composited into a 3D volume after all slices are completed. Once the 3D volume is created it can be used with a volume rendering technique that best suits the data to be visualized.

3.4.2 3D HyperLIC

HyperLIC requires a lot more information than the LIC algorithm. Instead of one vector field that must be loaded into memory at any time, 3D HyperLIC requires three different eigenvectors to generate the final volume. A 512^3 tensor field would require far more memory than is available on an off-the-shelf graphics card. One way to reduce the amount of memory needed at one time is to use the multipass approach. Only one eigenvector field is loaded at a time and an intermediate volume is generated using the 3D LIC method. This intermediate volume is then used as the noise volume for the next eigenvector field. Again the rendering capabilities of the graphics card will force the creation of individual slices of the 3D volume that will be composited after the completion of each slab.

3.4.3 Neighbor based methods

The 2D neighbor based methods are fairly simple to extend to 3D. The main change is the amount of neighbors that must be precalculated. In particular, the weighted neighbor method would require 27 lookups into the noise texture map for each pass. This is far worse than the 3D LIC method and would require more memory than storing the 3D tensor space in its entirety. The one and two neighbor methods in 3D would only require additional neighbor information for the third eigenvector field when approximating the HyperLIC method.

4 IMPLEMENTATION ISSUES

4.1 Precision

There are several issues with implementing the HyperLIC algorithm completely on hardware. The first concern is with transferring the tensor field to the GPU. When encoding the tensor information in a texture map there is an associated loss of precision. Typical texture maps are 32 bits per texel. Using a texture map of this depth resolution, a 2D tensor can be stored in one texel of the texture map with some loss of precision. Another property of texture maps is that when used to store floating point textures, the values of each color channel are clamped between zero and one. For tensors that vary outside of that range, this can be a problem that can be overcome by the use of an additional color channel. For each value in the tensor, two color channels are used. This means that a 2D tensor will require 2 texels of storage. The first channel stores the value V of the tensor and the second channel stores a scaling value S. The full tensor value is created in the fragment shader by V/S. This approach increases range of potential values for a texture map, but is no substitute for a texture map with a more bits per texel. Most graphics cards available now have that capability, which can be accessed through vendor specific OpenGL extensions, or through DirectX. Both the X800 card from ATI and the GeForce6800 card from nVidia support 64 and 128bit texels.

4.2 Non-Power-of-Two Textures

Texture dimension is another issue that must be addressed. Because texture dimensions need to be a power-of-two, both the resulting image from the hardware HyperLIC and the input vector field must have dimensions that are power-of-two as far as the graphics card is concerned. A non-power-of-two vector field can be used, by padding the input texture map with any value, so that the field is 2^N in size. For the resulting image, the undesired pixels can be ignored, with the only side effect being a small loss of performance, and more texture memory being used.

4.3 Memory usage

Memory limitation is not a concern for most 2D applications, but for 3D, it becomes the primary concern. Not only must the tensor field be stored on the graphics card, the output volume must be stored as well. Common graphics cards are limited to 256 megabytes of memory, which is not enough to store even just the input tensor field. A 512^3 3D tensor field will require more than 384 megabytes if each eigenvector could be stored as one byte each. In addition to the input texture, the output texture must be stored. Obviously, there is a large problem for extending any of these methods directly to 3D. A solution to this problem is to place a smaller subsection of the input data into memory at anytime. By placing a slab of the tensor field into memory, a slice of the output texture can be created.

A 3D texture can be thought of as nothing more than a stack of 2D textures. When slice z_i of the output volume is being created, only slices $i-(L/2)$ to $i+(L/2)$ need to be in memory, where L is the number of steps in the integration. This does create some overhead however. Depending on how much memory is available there will be constant swapping of memory between the graphics card and the main memory of the computer to load the tensor texture, in addition to the transfer of the output slices. Despite the overhead, this approach will still realize performance gains over the software version, and as the memory available on graphics cards increases, the overhead will drop to almost zero.

4.4 Image Boundary

The edges of images are also a problem. As a streamline leaves the defined region of the vector field , there is a question of how to handle the boundary. One solution is to generate an image that is smaller than the vector field. This solution provides quality images, but requires a larger dataset than needed. A simple solution for the LIC images is to change the settings for the noise texture lookup to wrap around the texture. If the texture coordinates are clamped between 0.0 and 1.0 a smearing appears on the borders, but with periodic wrap, the smearing disappears and the images maintain most of the information at that edges. This doesn't work as well for the HyperLIC images. The wrapping texture lookups work as well for LIC because the noise image is random. In the intermediate step of the HyperLIC algorithm, the noise image is no longer random.

5 RESULTS

The use of hardware acceleration produced images of the same quality as traditional software LIC. Figure 3 shows an example of a vector field visualized using the traditional LIC algorithm done in software and in hardware. The image on the left took 3.773 seconds to create, while the hardware image was generated in 0.02985 seconds. Using an nVidia GeForceFX 5950 GPU, frame rates of 33 fps were achieved for generating a LIC image. For HyperLIC, rates of 22 fps were achieved on the same graphics card. Figure 4(b) shows a HyperLIC image generated using 16 integration steps per streamline along both the major and minor eigenvectors. This is more than two orders of magnitude faster than two pass HyperLIC images using traditional LIC implemented in software, and more than an order of magnitude over FastLIC. The hardware implementations show great performance gains over the optimized software versions.

The images generated by the one neighbor based method for tensor visualization generated the same frame rates as the hardware accelerated HyperLIC when using 16 rendering passes. While the frame rate was acceptable, the image quality was much lower than that of the HyperLIC algorithm. The two neighbor based method images appeared more blurred in regions where the tensor field changed rapidly. Figures 5(b) and 5(c) show only the major eigenvector field rendered using one and two neighbors. Figure 6(b) shows both the major and the minor eigenvector fields rendered using the two neighbor method.

The weighted neighbor method produced the images shown in Figure 5(d) and 6(c). Again the images did not have the same quality as when using the traditional LIC and HyperLIC algorithms. However the picture quality is much better than that of the one and two neighbor based approaches, especially for the LIC version. If the number of passes is limited to 2, performance is on par with the hardware HyperLIC algorithm, but the image quality is lower. This method is limited in performance as it uses almost as many texture lookups as one pass in HyperLIC, and uses slightly less floating point calculations.

6 CONCLUSIONS

Fragment and Vertex shaders provide a very useful tool for scientific visualization. Many applications that can take advantage of parallel processing such as LIC and HyperLIC can be adapted to run on these GPUs. The performance gain from parallelization and the graphics hardware itself is remarkable. For HyperLIC, interactive frame rates for a 512x512 image are easily attainable. It is even possible to gain more performance improvements as new, more capable graphics cards come out.

Floating point operations are typically very expensive on most graphics cards. In addition to being expensive, the amount of bits available to represent a floating point number is limited. This results in slower frame rates and slightly lower image quality than can be obtained by a software implementation. Even with the loss of performance by using floating point calculations, hardware accelerated methods are vastly faster than software. The image quality issue is not readily apparent, but with new graphics cards not only will performance increase, but the quality of the images will be identical to software. By using 64 bit or 128 bit textures, there will be no perceptible loss of precision; and new graphics cards are just as fast with floating point numbers as integers.

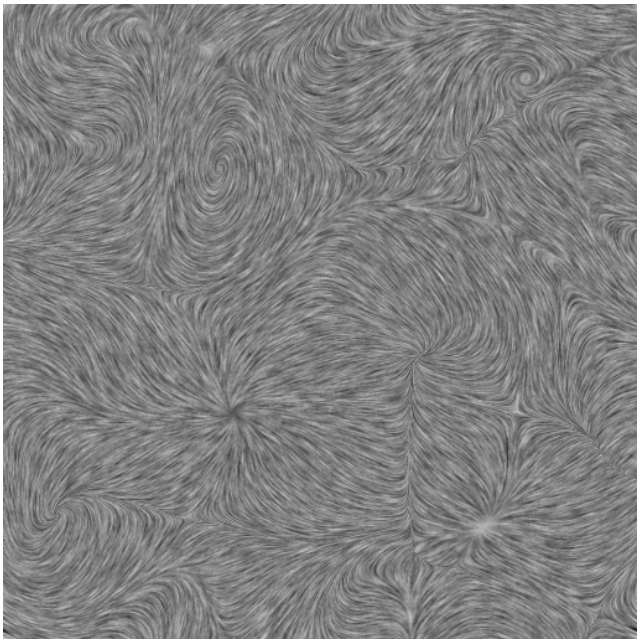


Figure 2: LIC image generated completely in hardware.

ACKNOWLEDGEMENTS

We would like to thank Xiaoqiang Zheng for insights and suggestions on HyperLIC, Jiwon Shin for earlier work on image based HyperLIC, and Yeon Gyoung Gwack for the FastLIC implementation of HyperLIC. We would also like to thank Craig Wittenbrink and NVidia for the graphics card used in the experiments.

REFERENCES

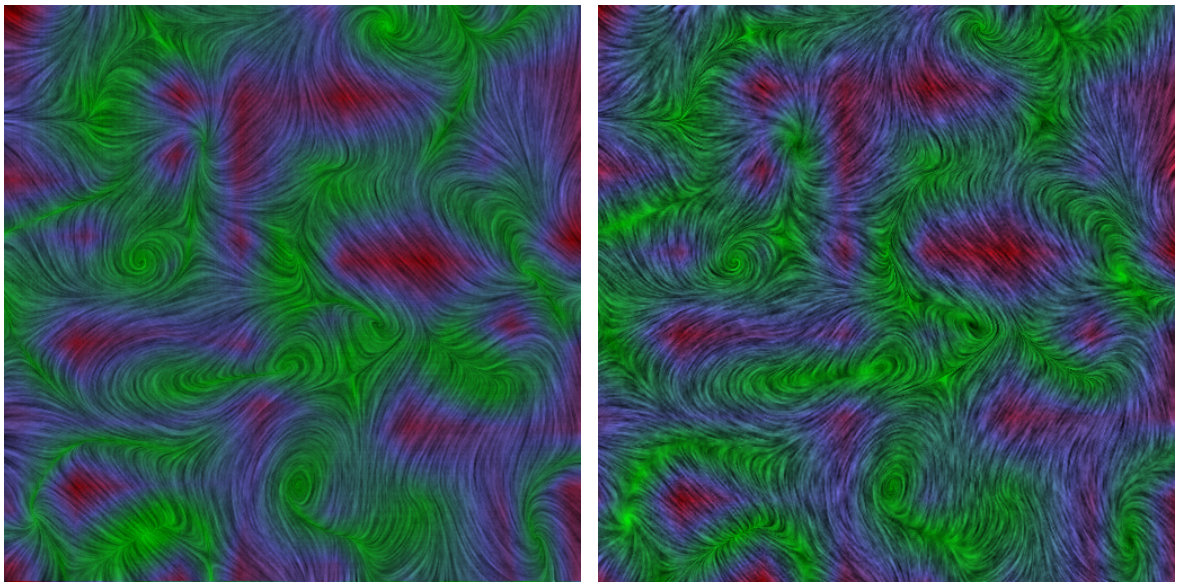
[1] A. Bhalerao and C.-F. Westin. Tensor splats: Visualising tensor fields by texture mapped volume rendering. In *Sixth International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI'03)*, pages 294–901, Montreal, Canada, November 2003.

[2] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. *Computer Graphics (SIGGRAPH Proceedings)*, 27(4):263–272, 1993.

[3] T. Delmarcelle and L. Hesselink. Visualizing second-order tensor fields with hyperstreamlines. *IEEE Computer Graphics and Applications*, 13(4):25–33, July 1993.

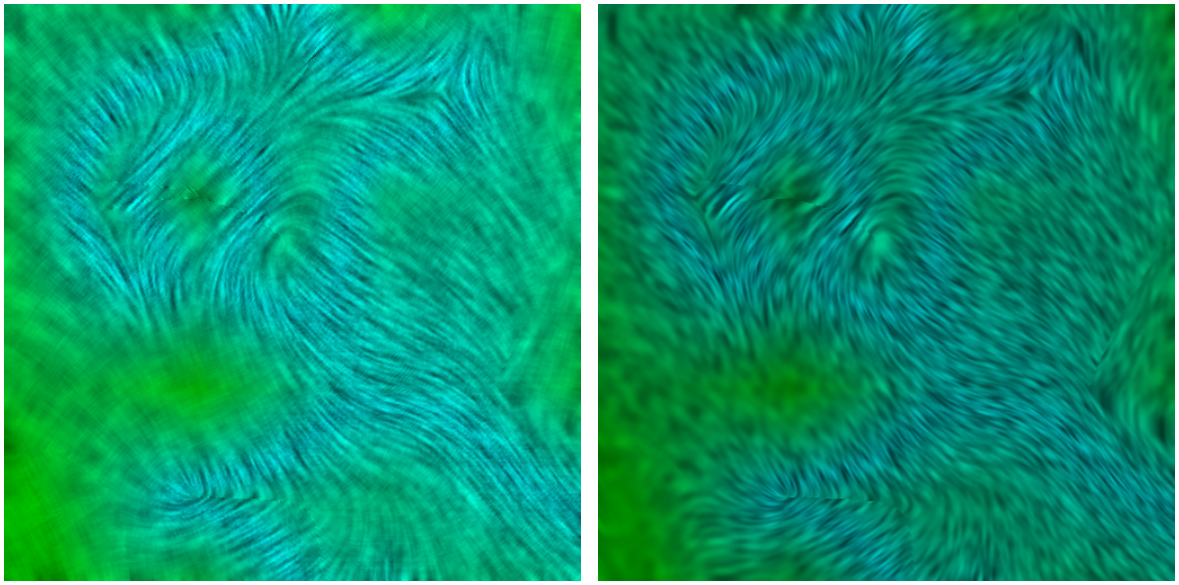
[4] L. Hesselink, T. Delmarcelle, and J.L. Helman. Topology of second-order tensor fields. *Computers in Physics*, 9(3):304–311, May-June 1995.

[5] I. Hotz, Z.X. Feng, H. Hagen, B. Hamann, K.I. Joy, and B. Jeremic. Physically based methods for tensor field visualization. In *Proceedings of Visualization '04*, pages 123–130, Austin, 2004.



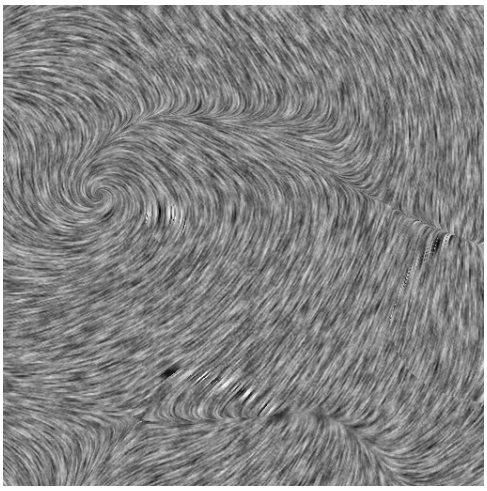
(a) Software (b) Hardware

Figure 3: Software and Hardware LIC images

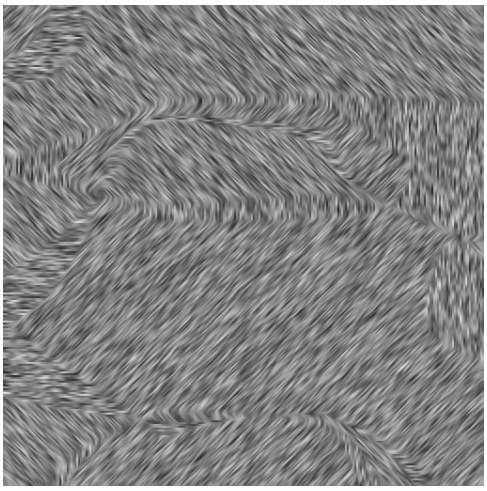


(a) Software (b) Hardware

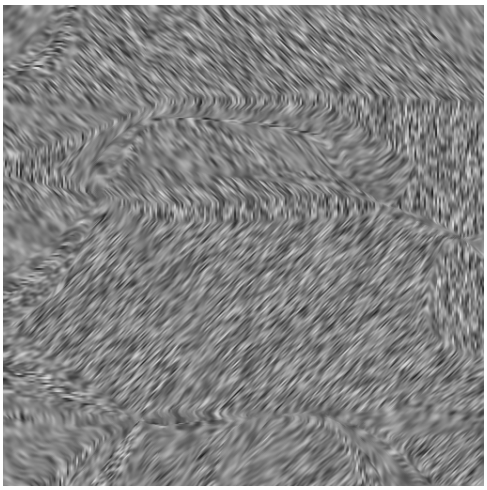
Figure 4: Software and Hardware HyperLIC images



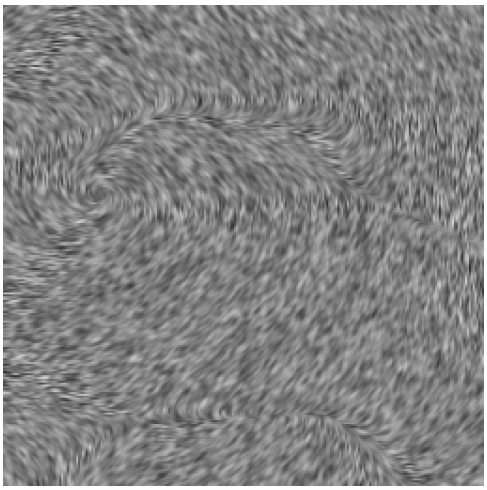
(a) Hardware LIC



(b) One Neighbor

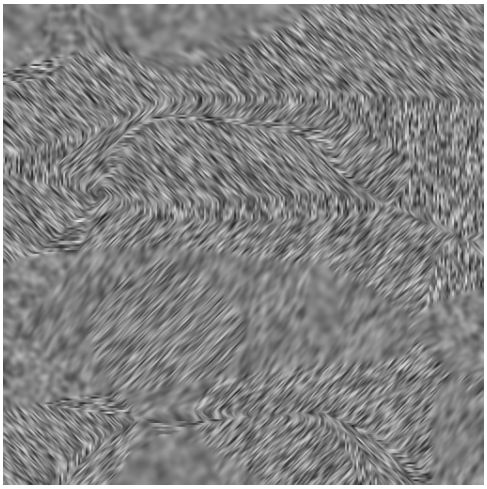


(c) Two Neighbor

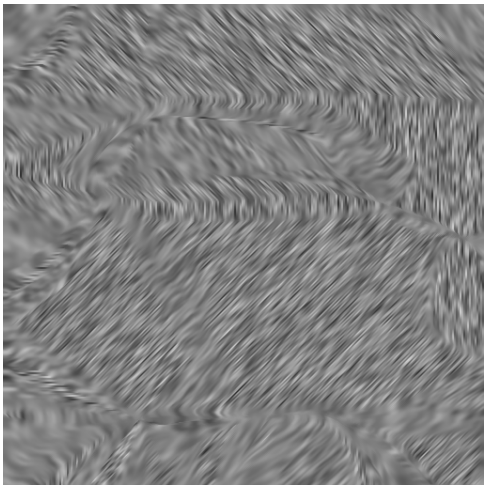


(d) Weighted Neighbor

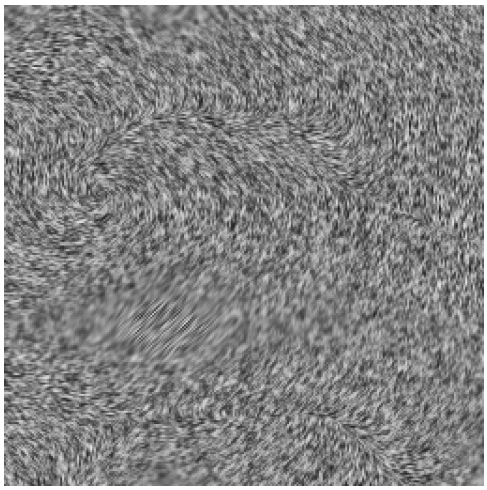
Figure 5: Neighbor Based LIC compared to Hardware LIC



(a) One Neighbor



(b) Two Neighbor



(c) Weighted Neighbor

Figure 6: Neighbor Based HyperLIC

[6] Gordon Kindlmann. Superquadric tensor glyph. In *Vissym'04*, pages 147–154, 2004.

[7] Gordon L. Kindlmann and David M. Weinstein. Hue-balls and lit-tensors for direct volume rendering of diffusion tensor fields. In *IEEE Visualization*, pages 183–189, 1999.

[8] David Laidlaw, Eric Ahrens, David Kremers, Matthew Avalos, Russell Jacobs, and Carol Readhead. Visualizing diffusion tensor images of the mouse spinal cord. In *Proceedings of Visualization '98*, pages 127–134, 1998.

[9] H.W. Shen and D.L. Kao. Uflic: A line integral convolution algorithm for visualizing unsteady flows. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 317–322. IEEE Computer Society Press, 1997.

[10] D. Stalling and H.-C. Hege. Fast and resolution independent line integral convolution. *Computer Graphics Siggraph Proceedings*, pages 249–256, 1995.

[11] Alexandru Telea and Jarke J. van Wijk. 3D IBFV: Hardware-accelerated 3d flow visualization. In *Proceedings of Visualization '03*, pages 233–240, 2003.

[12] Jarke J. van Wijk. Image based flow visualization. *Computer Graphics*, pages 745–754, 2002.

[13] V. Verma, D. Kao, and A. Pang. Plic: Bridging the gap between streamlines and lic. In D. Ebert, M. Gross, and B. Hamann, editors, *Proceedings IEEE Visualization '99*, pages 341–348. IEEE Computer Society Press, 1999.

[14] W.Benger and H.-C. Hege. Tensor splats. In *Visualization and Data Analysis*, pages 151–162, 2004.

[15] D. Weiskopf and T. Ertl. GPU-based 3D texture advection for the visualization of unsteady flow fields. In *Proceedings of WSCG 2004 Short Papers*, pages 259–266, 2004.

[16] Xiaoqiang Zheng and Alex Pang. HyperLIC. In *Proceedings of Visualization '03*, pages 249–256, Seattle, 2003.

[17] Xiaoqiang Zheng and Alex Pang. Topological lines in 3D tensor fields. In *Proceedings of Visualization '04*, pages 313–320, Austin, 2004.