GPU-PIV

Thomas Schiwietz, Rüdiger Westermann

Siemens Corporate Research, Inc.
755 College Road East, 08540 Princeton, NJ
Email: thomas.schiwietz@scr.siemens.com

Technische Universität München, Institut für Informatik
Computer Graphics & Visualization Group
Boltzmannstr. 3, 85748 Garching, Germany
Email: westerma@in.tum.de

Abstract

Digital Particle Image Velocimetry (PIV) is an optical technique used to measure the velocity of seeded particles in real flow. A CCD camera captures the flow field twice under exposure to a short duration laser flash. Recorded image pairs are cross-correlated to extract velocity information from these records. Time resolved PIV technology can capture images with some hundreds of frames per second.

In this paper, we present a PIV-system that implements vector field reconstruction and visualization on programmable graphics processing units (GPUs) thus providing a high-speed back-end for time resolved PIV technology. We propose an efficient FFT implementation on such hardware, which is used to cross-correlate multiple pairs of interrogation windows. To visualize extracted vector fields we employ functionality to create and to render geometry data on the GPU. In this way, not only can any data transfer between the CPU and the GPU be avoided, but spatial information derived from PIV as well as the time history of points in the flow can be combined instantaneously.

1 Introduction

Over the last decades, Particle Image Velocimetry (PIV) has positioned itself as a reliable technique for the measurement of particle velocities in real flow [5, 8, 9, 10]. In principle, PIV is a planar laser light sheet technique which records images of seeded tracer particles in these sheets on a video camera. The sheet is pulsed twice, and recorded image pairs are processed to determine the displacement of particles in this sheet. The displacement combined with the time delay between consecutive images give the velocity information for a small subregion of the flow area.

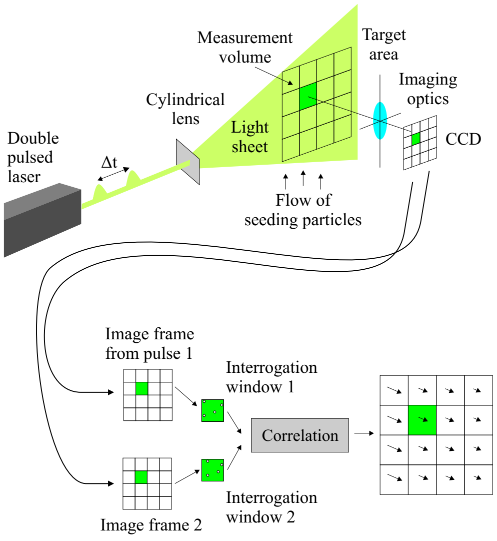

Figure 1: PIV Overview.

To compute particle displacements, the image plane is divided into small disjoint or overlapping interrogation windows, and corresponding window pairs in consecutive recordings are cross-correlated. The spatial displacement that produces the maximum cross-correlation statistically approximates the average displacement of particles in the interrogation window. This displacement, divided by the time between laser pulses, yields the velocity that is associated with each interrogation window (see Figure 1).

The cross-correlation function for two discretely sampled interrogation windows is defined as:

$$C_{fg}(m,n) = \sum_i \sum_j f_{i,j} \times g_{i+m,j+n} \qquad (1)$$

with $f_{i,j}$ and $g_{i,j}$ denoting the image intensity distribution of the first and second image, m and n the pixel offset between the two images and $C_{fg}(m,n)$ the two dimensional cross-correlation function. Given the size of a square interrogation area M, $O(M^4)$ operations have to be computed.

The cross-correlation between two image pairs can be normalized to prevent false correlation peaks arising from changes in the search area local means. In addition, any local additive bias differences can also be removed. This is achieved by removing the mean from both interrogation windows up front, and by dividing each correlation sample by

$$\sqrt{\sum_{ij}(f_{i,j} - \mu_F)^2} \times \sqrt{\sum_{ij}(g_{i+m,j+n} - \mu_{\tilde{G}})^2} \qquad (2)$$

where μ_G and μ_G are the mean of the first and the shifted second interrogation window, respectively.

To overcome the high numerical complexity of direct cross-correlation, in time-critical applications it is usually implemented by means of the discrete Fast-Fourier-Transform (FFT). The Wiener-Khinchin Theorem states that the cross-correlation $f \star g$ between two signals f and g can be computed in the frequency domain as

$$f \star g = F^{-1}(F_f F_g^*) \qquad (3)$$

where F_f and F_g^* denote the Fourier transform of the first interrogation window and the complex conjugate of the Fourier transform of the second interrogation window, respectively. F^{-1} denotes the inverse Fourier transform. The overall complexity now reduces to $O(M^2 lnM)$.

Mainly because of the periodic domain assumption, FFT based cross-correlation is supposed to produce less accurate results compared to the direct approach. On the other hand, due to its numerical efficiency it is suitable for high-speed applications as well as for multi-pass approaches, where less accurate velocity estimates are used to predict the search direction in upcoming passes.

Once the correlation function has been computed, the position of its maximum in the domain is used to estimate the movement of structures from the first to the second interrogation window. Adequate fitting procedures are used to determine the maximum within sub-pixel accuracy. The relative position of the maximum with respect to the center of the interrogation window is finally used to estimate the velocity.

Nowadays, high-speed CCD cameras allow for the recording of image pairs at frame rates of some hundred frames per second. In this respect, one of the challenges is to develop techniques for vector field reconstruction and visualization which can be integrated instantaneously into the recording process.

In this paper, we present the implementation of such a system on programmable graphics hardware. The system performs both the reconstruction of vector fields from image pairs and the visualization of these fields on the graphics chip. In this way, by directly connecting the CCD camera to a frame grabber on the graphics card, data transfer between the CPU and the GPU can be avoided entirely. We describe an efficient implementation of the FFT on programmable GPUs, and we exploit this implementation for simultaneous cross-correlation of multiple pairs of interrogation windows. Furthermore, a novel technique to visualize vector data is proposed, which exploits new functionality to create and to render line segments on the GPU.

The remainder of this paper is organized as follows: In the following sections we first review the basics of the discrete FFT, and we outline an efficient GPU implementation of this transform, both in 1D and 2D. We then describe how to use this technique to compute the cross-correlation between multiple interrogation windows, and we give additional information concerning the reconstruction of vector field data from 2D correlation signals on the GPU. Finally, we present a new technique for GPU-based visualization of vector field data.

2 FFT

The Fourier-Transform (FT) $F(n)$ of a discrete signal $f(k)$, with $k \in (0, N-1)$ represents the signal as a superposition of sinusoids of different frequencies:

$$F(n) = \sum_{k=0}^{N-1} f(k) e^{\frac{-i2\pi kn}{N}} \tag{4}$$

The FFT, as described in [2], reduces the numerical complexity of the discrete FT to $O(n\,log\,n)$. It is derived from the observation that equation 4 can be written as a matrix-vector product

$$\begin{pmatrix} F(0) \\ F(1) \\ F(2) \\ \vdots \\ F(n) \end{pmatrix} = \begin{pmatrix} r^0 & r^0 & r^0 & \cdots & r^0 \\ r^0 & r^1 & r^2 & \cdots & r^{n-1} \\ r^0 & r^2 & r^4 & \cdots & r^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r^0 & r^{n-1} & r^{2n-2} & \cdots & r^{(n-1)^2} \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(n) \end{pmatrix} \tag{5}$$

where $r^k = e^{\frac{-i2\pi k}{N}} = cos\left(\frac{2\pi k}{N}\right) + i \cdot sin\left(\frac{2\pi k}{N}\right)$.

By symmetry considerations, i.e. $r^{nk} = r^{nkmod(N)}$, the matrix can be split into a chain of $log\,N$ sparse matrices (FFT matrices), each of which contains exactly two non-zero entries. Note that one of these entries is always 1 and does not need to be stored explicitly. Below, this kind of factorization is illustrated for a four-component input signal. Finally, the output signal has to be rearranged to yield the Fourier coefficients in the right order.

$$\begin{pmatrix} F(0) \\ F(2) \\ F(1) \\ F(3) \end{pmatrix} = \begin{pmatrix} 1 & r^0 & 0 & 0 \\ 1 & r^2 & 0 & 0 \\ 0 & 0 & 1 & r^1 \\ 0 & 0 & 1 & r^3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & r^0 & 0 \\ 0 & 1 & 0 & r^0 \\ 1 & 0 & r^2 & 0 \\ 0 & 1 & 0 & r^2 \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{pmatrix} \tag{6}$$

The 2D discrete FFT can be computed by consecutive 1D discrete FFTs along the rows and the columns, respectively. Assuming a 2D signal of size $N \times N$, the discrete FFT is computed by means of $2 \cdot log\,N$ multiplications of a sparse-matrix and N 1D vector. For any possible N, the FFT matrices, or more precisely the non-zero elements in these matrices, can be pre-computed.

3 GPU-FFT

The implementation of the discrete FFT on graphics hardware exploits the fact that linear algebra operations can be performed very efficiently on this kind of parallel streaming architecture [1, 6]. On current GPUs, fully programmable parallel geometry and fragment units are available providing powerful instruction sets to perform arithmetic and logical operations on multi-component (RGBA) data. In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects. By representing matrices and vectors as texture maps, arbitrary operations between such object can be performed very efficiently.

$\begin{pmatrix} r^0_{real} \\ r^0_{imag} \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} r^2_{real} \\ r^2_{imag} \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} r^1_{real} \\ r^1_{imag} \\ 3 \\ 2 \end{pmatrix}$	$\begin{pmatrix} r^3_{real} \\ r^3_{imag} \\ 3 \\ 2 \end{pmatrix}$

Table 1: Texture layout for the pre-computed FFT table for the 1st matrix with $N = 4$

In regard to the particular structure of matrices in the FFT computation, we employ a special internal representation of these objects on the GPU. Every FFT matrix is represented as a 1D RGBA texture map, which contains in the i-th component the value of the complex non-zero entry in the i-th row of the matrix, and the absolute positions of both non-zero entries in this row (see table 1). These values are stored in the RG and BA components, respectively. The position of the vector component that has to be multiplied with the complex entry is stored in the B component.

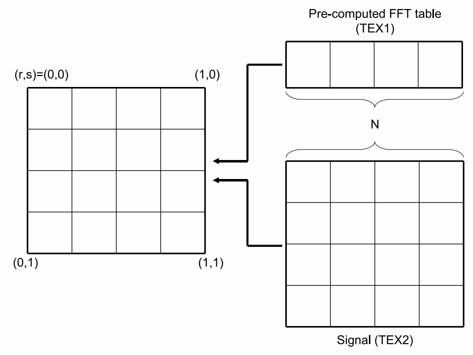

Figure 2: FFT shader inputs.

Without loss of generality, let us assume input images of size $N \times N$ throughout the remainder of this paper. The 2D signal to be transformed is stored as a 2D texture map. To perform the matrix-vector multiplication at a particular FFT stage, a quadrilateral covering $N \times N$ fragments is rendered. To this quadrilateral, both the 2D texture containing the signal (TEX2) and the 1D texture containing the FFT matrix (TEX1) are bound (see Figure 2). Then, a fragment shader program performs the following operations, where $\times$ and $+$ indicate complex multiplication and addition: To avoid binding a different

Column-Wise FFT-Matrix-Vector Operation

```
1  $OP1 = TEX1[s]$
2  $OP2 = TEX2[r, OP1_3]$
3  $OP3 = TEX2[r, OP1_4]$
4  output = $OP2 \times OP1_{1,2} + OP3$
```

1D texture (TEX1) in every FFT stage, all these textures are combined into one single 2D texture. The respective row to be accessed in each stage is specified in a constant parameter to the shader program. In every pass, results are written to a texture render target, which becomes TEX2 in the following pass. After $log\,N$ passes, the column-wise FFT has been performed, and results are going to be reordered using a reorder texture. This texture stores for every element the counter component to swap with. The same passes are then repeated, but now the quadrilateral is rendered with transposed texture coordinates to perform the FFT in row-wise order. At the end, the RG components of the most recent render target carry the Fourier coefficients.

The FFT performance can be doubled by storing pairs of consecutive columns in the RG and the BA components of TEX2, respectively. Furthermore, arithmetic operations can be calculated in parallel due to the internal RGBA-pipeline. This effectively halves the number of texture fetches to be performed during column-wise FFT. In an intermediate pass, the output texture is reorganized to store pairs of consecutive row entries in a single RGBA texture element. Then, the row-wise FFT can be accelerated by a factor of two as well.

3.1 Performance

To verify the effectiveness of the proposed FFT implementation, we investigate the performance for different image sizes. All our experiments were run under WindowsXP on a P4 3.0 GHz processor equipped with an ATI 9800 XT graphics card. In particular, we compare the performance of the GPU-FFT to the FFTW [3], an efficient CPU implementation of the discrete FFT leveraging various acceleration strategies like SSE parallelization, cache optimization and pre-computed FFT tables. To conduct a fair comparison, the FFTW_MEASURE setting was enabled and the code was run in 32-bit floating point precision. In all our experiments, the time it takes to perform the FFT of a discrete complex 2D signal is measured.

	128^2	256^2	512^2	1024^2
GPU-FFT	1083	296	67	16
FFTW	1500	448	65	15

Table 2: GPU-FFT performance measures fps.

Table 2 essentially shows the GPU-FFT to be able to process even high resolution images at interactive rates. The implementation, on the other hand, does not yield a significant speed up compared to the FFTW. This is due to the many floating point texture fetches that have to be performed to look up the FT table and the complex operands in each of the $2 \cdot log\,N$ transform stages. We should note here that we also measured the loss of performance introduced by the two dependent texture fetches in lines 2 and 3 of the pseudo code above. In the current setting, however, they only make about 5% of the overall time.

Compared to the GPU-FFT proposed by Moreland et. al [7], our implementation runs at significantly faster rates. From the timings presented in the aforementioned paper (including four forward transforms, a complex multiplication stage and backward transforms on a 1024^2 grid), we find our solution to be about a factor of 11 faster.

4 GPU-PIV

Based on the proposed GPU-FFT, we now outline a strategy to exploit this implementation in digital PIV. Note that compared to the FFTW, the results already reside in video memory and can be directly visualized.

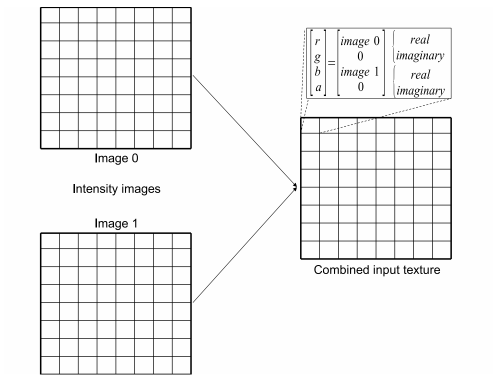

Figure 3: Pairs of input images are combined into one texture map.

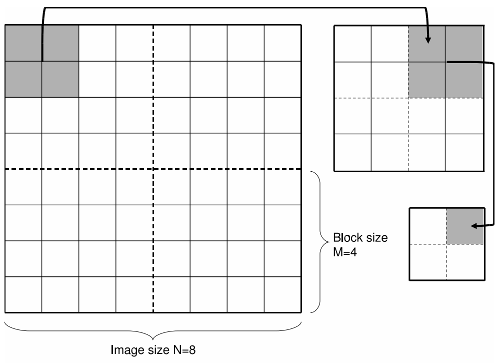

Figure 4: Reduce operation to calculate average intensities in each interrogation block

4.1 Image Combiner

Once the two images captured by the CCD camera have been received on the GPU, they are combined into one texture as depicted in figure 3. This allows upcoming operations to be performed on interrogation window pairs in parallel, and it essentially halves the number of texture fetches in these operations. The intensities of the first and the second image are interpreted as the real parts of two complex input signals. They are stored in the R and B color channel, respectively. Both the G and the A channel, which store the corresponding imaginery parts, are initially set to zero.

4.2 Block Average Removal

To normalize the intensity distribution in the interrogation windows, for each window average intensities have to be computed and subtracted from intensity values. If the difference gets negative, the value is set to zero.

To calculate block averages, we employ a reduce operation as proposed in [6], which recursively combines texture samples in multiple rendering passes. Starting with the 2D texture that is made of a set of interrogation windows of size $M \times M$, in $log\, M)$ steps s a quadrilateral covering $M/2^s$ pixels in screen space is rendered. The texture value that is mapped to the current fragment position is combined with the three adjacent texture elements in positive (u,v) texture space direction. The output is written to a new texture render target of a factor of two smaller in each dimension than the previous one. This procedure is repeated until for each block one single average value is left.

In a final pass, the block average texture is rendered over the original image texture as to allow the pixel shader to access average values. At every pixel the block average is subtracted and the resulting value is clamped to zero.

4.3 Block FFT

Once input images have been normalized, the GPU-FFT carries out the FFT stages for all interrogation windows at once. Therefore, a slightly different FFT table is built. At first, a table of size M is built, which is then extended periodically until its size is equal to N. Absolute positions that are stored in the table need to be changed accordingly. Images containing sets of interrogation windows can now be transformed using the core FFT implementation as described. The difference simply is that less FFT stages have to carried out compared to a transformation of the entire image. The table for $N = 8$ and $M = 4$ is illustrated in Figure 5.

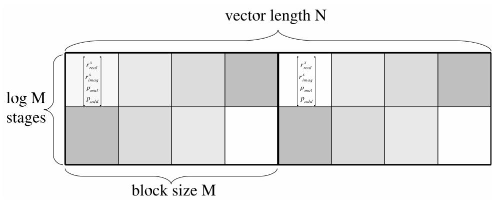

Figure 5: Table layout for Block-FFT.

4.4 Cross-Correlation

The cross-correlation in frequency domain is simply a complex conjugate multiplication of corresponding values in both images. Since both transformed images are stored in one texture map, a pixel shader program samples the texture and computes the multiplication using real and imaginary parts in the RB and the GA color components, respectively. As a result, a single complex value is computed and stored in the RG components of an output texture. In a final pass, pairs of interrogation windows are packed into RGBA samples of a smaller texture, thus enabling the inverse transformation of two blocks simultaneously. The inverse FFT transforms the content of each block into two real valued signals that are encoded in the RB components of a texture map.

4.5 Peak Finding

In each block the relative position of the maximum value, i.e. the correlation-peak, determines the velocity of that block within the current time interval.

In order to find that maximum, we exploit the reduce operation described in 4.2. Instead of computing averages, the pixel shader now combines adjacent samples by computing their maximum. In every stage, the position of this maximum is kept, and it is updated according the position of maxima that are found in upcoming passes.

4.6 Sub-Pixel Displacement

To determine the correlation peak at sub-pixel accuracy, a function is fitted to a set of samples including the one where the maximum was found as well as a ring of neighbors.

In this work, we have implemented the two estimators below (the 1D case is illustrated), where c_j is the correlation value at position j. Both can be evaluated straight forwardly in a pixel shader program.

- *Center-Of-Mass*: $\frac{c_{i-1}-c_{i+1}}{c_{i-1}+c_i+c_{i+1}}$
- *Gauss-Fit*: $\frac{\ln c_{i-1}-\ln c_{i+1}}{2\cdot\left(c_{i-1}-2\cdot c_i+c_{i+1}\right)}$

4.7 Outlier Removal

Outliers are vectors in the field which have an orientation or length that significantly differs from the values at adjacent grid points. They are removed by calculating the average vector within a certain support, and by replacing a vector by this average once it diverges too much from it. By replacing every vector by the average, the entire field can be smoothed. Since computing the average only requires a few adjacent texture samples to be fetched, it can be performed on a per-fragment basis very efficiently.

4.8 Overlap Sampling

Finally, to increase the resolution of the generated vector field we increase the number of interrogation windows by allowing for overlapping windows. In the implementation this is realized by performing multiple PIV passes, and by equally shifting the interrogation windows in each pass. The shift is simply issued as a constant parameter in the respective shader programs, leaving the core PIV implementation unchanged. Vector fields constructed in consecutive passes are finally merged to a field of higher resolution.

5 Performance

Below, timings are given for differently sized images and interrogation windows. About 50% of the computation time is consumed by the FFT. The remaining time is mostly required by mean removal and peak finding. Similar to the FFT implementation, both operations are also performed recursively in $log\,M$ rendering passes. All the other operations are carried out in one single pass. As can be seen, typical PIV images having resolution of about 800x600 pixels can be processed at interactive frame rates using our system.

window/image	256^2	512^2	1024^2
8^2	151	48	13
16^2	126	34	9
32^2	105	28	7

Table 3: GPU-PIV performance in fps.

6 Vector Field Visualization

To allow for the analysis of the reconstructed flow fields, we have implemented several GPU-based visualization techniques. In this way, any data transfer between the CPU and the GPU can be entirely avoided.

In general, derived flow quantities like vorticity, velocity magnitude, or divergence can be easily computed in appropriate fragment shader programs. These quantities are computed at every grid point of the reduced grid, which consists of the center points of the interrogation windows. Results are written to an additional render target, which is finally displayed as a background texture covering the entire domain. In this way, interactive visual analysis of the flow dynamics is possible, yet providing multiple visualization options. In the current scenario, the following quantities have been considered:

- Velocity magnitude is used to index into a user-specified color map. The color map is realized as a 1D texture map.
- In an analogous manner we display the magnitude of the rotation $\omega_z = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$, and the divergence $\eta = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$.

Figure 3 on the color page below shows different visualizations of a flow field generated by our system. The vector field was derived from moving micro-biological structures as seen figure 2.

6.1 Visualization Geometry

For the analysis of vector fields, vector plots are still the most popular visualization technique in the PIV community. Although easy to implement, this kind of technique usually requires the vector valued data to be read back to the CPU, to construct the arrow primitives and to send them to the GPU again for rendering purposes.

Especially for high resolution data sets, this approach puts the burden almost entirely on the bus connecting the CPU with the GPU. On the other hand, until recently it was not possible on any graphics hardware architecture to generate or to arbitrarily manipulate geometric primitives, thus prohibiting the use of vector plots without data transfer.

Nowadays, however, Shader 3.0 [4] available in DirectX on recent nVidia cards, i.e. the GeForce 6800, allows direct access to texture maps in the vertex units. In addition, recent ATI graphics hardware provides an extension to OpenGL called *SuperBuffers*. The interface allows the application to allocate graphics memory directly, and to specify how that memory is to be used. This information, in turn, is used by the driver to allocate memory in a format suitable for the requested uses. When the allocated memory is bound to an *attachment point* (a render target, texture, or vertex array), no copying takes place. The net effect for the application program therefore is a separation of raw GPU memory from OpenGLs semantic meaning of the data. In summary, both Shader 3.0 and OpenGL SuperBuffers provide an efficient mechanism for storing GPU computation results and later using those results for subsequent GPU computations.

As a consequence thereof, it is now possible to compute intermediate results in the fragment units on the GPU, drawing these results to invisible buffers, and then using them either directly as vertex information or as displacement vectors for visualization geometry.

6.2 Vector Plots

To construct vector plots on the GPU, we assume that every vector is represented as a polyline consisting of three line segments, i.e. the tail and the arrow head. Using Shader 3.0 functionality, a vertex array is built up front, which contains the vertex information necessary to render as many arrows as there are grid points in the reduced grid. In this array, vertex coordinates are specified as to produce vertically oriented arrows having unit length.

After the vector field has been derived from the cross-correlation between pairs of interrogation windows, an additional rendering pass is carried out. It produces as many fragments as there are grid points, and for each grid point it computes the scaling and the orientation of the respective vector with respect to the initial (vertically oriented) vector. Both values (scaling parameter and angle) are rendered into the RG color components of a texture render target. In a final rendering pass, the application program renders the pre-computed vertex array, and it enables the geometry units to access the render target. Now, every vertex can scale and rotate itself according to the stored values. The modified line segments are finally rendered to generate the vector plot.

Note that the same procedure can be performed using OpenGL SuperBuffers. In this case, the content of the pre-computed vertex array is stored in local video memory as a memory object. Semantically, this object can be interpreted as a texture map, allowing for the manipulation (scaling and rotation) of entries in the fragment units. Results of these operations are rendered into a copy of this memory object, which, semantically is interpreted as a vertex array. This object is then passed to the geometry processing unit to render the vector plots.

6.3 Performance Evaluation

In the following table we give timings statistics for the construction and rendering of differently sized vector plots using Shader 3.0 functionality and the OpenGL SuperBuffers. As can be seen, compared to the time needed for vector field reconstruction, visualization of the vector field does not impose any significant overhead. In particular, for typical PIV images between 256^2 and 1024^2 in combination with interrogation windows between 16^2 and 32^2, the visualization process requires less than 2% of the overall time.

	128^2	256^2	512^2	1024^2
Shader 3.0	1050	321	85	23
SuperBuffers	311	230	97	38

Table 4: Performance measures in frames per second for GPU-based construction and rendering of vector plots.

7 Conclusion

In this paper, we have presented the first digital PIV system that performs vector field reconstruction and visualization on programmable graphics hardware. By combining a GPU implementation of the discrete FFT with Shader 3.0 functionality or OpenGL SuperBuffers, the system provides an efficient back-end for time resolved PIV technology. Our timings have shown that the proposed system has the potential to directly process the output of high-speed CCD cameras. Two images of size 1024^2, which are split into 16^2 interrogation windows, can be cross-correlated and visualized by means of vector plots with about 10 fps.

In the future, we will directly connect our system to frame grabber hardware on the graphics card. In this way, any data transfer to and from the CPU can be avoided. Due to various visualization options, spatial information derived from PIV as well as the time history of points in the flow can be analyzed instantaneously.

8 Acknowledgement

This work was financially supported by the Deutsche Forschungsgemeinschaft (DFG) in the Schwerpunktprogramm 1147 ”Bildgebende Messverfahren für die Strömungsmechanik”.

References

[1] Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Computer Graphics (Proc. SIGGRAPH ’03)*

[2] Brigham E.O: The Fast Fourier Transform And Its Applications. *Prentice-Hall, Englewood Cliffs, NJ.*

[3] Frigo M., Johnson S.: FFTW: An adaptive software architecture for the FFT. *In Proc. Acoustics, Speech, and Signal Processing 3 (1998)*

[4] Gray K.: The Microsoft DirectX 9 Programmable Graphics Pipeline. *Microsoft Press 2003*

[5] Keane, R. D., Adrian, R. J.: Theory of cross-correlation analysis of PIV images. *In: Applied Scientific Research 49 (1992), pp. 191215 2*

[6] Krueger, J., Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Computer Graphics (Proc. SIGGRAPH ’03)*

[7] Moreland K., Angel E.: The FFT on a GPU. *In Proc.. SIGGRAPH/EG Conference on Graphics Hardware ’03(2003)*

[8] Raffel, M., Willert, C. E., Kompenhans, J.: Particle image velocimetry: a practical guide. *Berlin : Springer, 1998 2, 4.*

[9] Westerweel J.: Digital Particle Image Velocimetry -Theory and application. *Ph.D. Thesis, Technical University of Delft*

[10] Willert, C. E., Gharib, M. : Digital particle image velocimetry. *In: Experiments in Fluids 10 (1991), pp. 181193*