



# ON A GRAPHICS HARDWARE-BASED VORTEX DETECTION AND VISUALIZATION SYSTEM

**Simon Stegmaier, Thomas Ertl**  
**Institute for Visualization and Interactive Systems**  
**University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany**

**Keywords:** *Flow Visualization, Vortex Detection, Graphics Hardware*

## ABSTRACT

*We propose a system that provides interactive denoising, vortex detection, and visualization of vector data. All the major phases of the system are implemented on a GPU (graphics processing unit). Once the vector field has been loaded onto the graphics adapter, no intermediate results need to be read back by the application. By taking advantage of modern GPUs' parallel processing capabilities, our system is able to significantly outperform software implementations and thus provides a valuable tool for the interactive exploration of vector fields and the understanding of flow structures.*

## 1 INTRODUCTION

Physical and numerical flow simulations have become an important part of the research activities in both industry and academia. To gain an understanding of the simulated flow, it is necessary to perform some kind of data analysis. This is usually done by scientific flow visualization. A good survey of state-of-the-art flow visualization techniques can be found in [11].

While there is probably no single visualization technique that can be regarded the best one, there is no doubt that feature detection methods are among the most effective tools for understanding a flow field. Of these, in turn, the class of vortex detection algorithms has proven to be of special importance.

However, vortex detection is computationally much more expensive than drawing an isosurface of the velocity magnitude or a slice with any other scalar property mapped to a color. Accordingly, vortex detection currently cannot be done on an off-the-shelf PC at interactive frame rates—which may be desirable for tracking flow structures over several time-steps in unsteady flows or for analyzing flow data obtained by some measuring or simulation technique vulnerable to noise, e.g. particle image velocimetry (PIV) or direct numerical simulation (DNS).

In the latter case, some form of denoising should be applied to the raw data before one tries to detect vortices. If the frequency of the noise is known in advance, denoising is most effectively accomplished by designing a bandpass filter capable of removing the relevant frequencies.

However, if the noise cannot be exactly located in the frequency domain, an interactive cycle of filtering, vortex detection, visualization, and evaluation (based on the existing knowledge of the flow) must be entered and repeated until the optimal filter characteristics have been found and a visualization

of acceptable quality is obtained. Obviously, neither filtering nor visualization come for free, so handling the complete cycle is even more difficult than handling the vortex detection alone.

In this paper we demonstrate that by shifting the entire cycle from the CPU to the GPU and by exploiting the modern GPUs' parallel processing capabilities interactive work *is* possible. Our solution expects the vector field data to be made available in a texture which assumes the input data is defined on a uniform Cartesian grid. However, there are efficient techniques for the conversion between grids without sacrificing much accuracy (see e.g. [14]), thus, this poses no limitation. Once the vortices have been detected, an isosurface of the detected vortex regions is generated. If the user decides to adjust the filter support or the isovalue, he will get an instantaneous update of the visualization. At no instant is it necessary to pass any intermediate results back to the application.

The remainder of the paper is organized as follows: In Sec. 2 we discuss work of other researchers related to this paper. Sec. 3 gives an overview of vortex detection algorithms and a rating with regards to suitability for graphics hardware. Sec. 4 gives some background information required for a thorough understanding of the system architecture presented in Sec. 5 and Sec. 6. An evaluation of our system follows in Sec. 7. The paper concludes in Sec. 8.

## 2 RELATED WORK

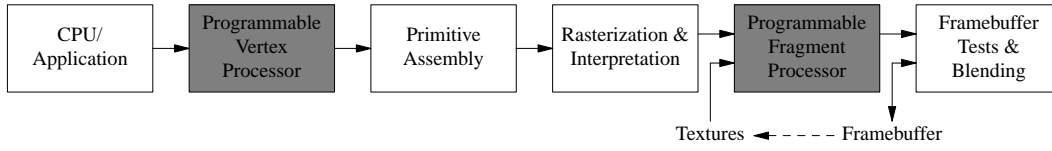
Much work has been invested into methods that are able to reliably detect vortices. The papers by Jiang et al. [5] and Post et al. [11] provide good overviews of the most popular techniques and give taxonomies of vortex detection algorithms. So-called *local* methods require only operations within the neighborhood of a cell; thus, all algorithms based on the Jacobian matrix (or the velocity gradient tensor) fall into this class. On the contrary, *global* methods examine many grid cells to detect a vortex. Typical representatives of this class are algorithms based on streamline tracing. It is obvious that global methods are harder to parallelize and in general more complex than local methods. For this reason, only local algorithms are suitable candidates for an implementation on a target platform as limited as a GPU.

Another desirable property of a vortex detection algorithm is Galilean invariance since this allows the detection of vortices not only in steady but also in time-varying flow fields. This further isolates the number of algorithms appropriate for an implementation on the GPU.

To our knowledge, none of the vortex detection algorithms found in the literature has ever been implemented on a GPU. However, many techniques have been developed to effectively visualize flow fields using a GPU. Especially texture-based techniques—though rather dated [2, 8]—are enjoying great popularity and are still a topic of research [15]. Since the basic procedure of GPU-based vortex detection can be expected to be the same as that of GPU-based flow visualization, these research results are nevertheless a valuable foundation for this work.

## 3 VORTEX DETECTION

As was explained in the previous section, a vortex detection algorithm suitable for a GPU-implementation should be both local and Galilean invariant. The most simple method that meets these criteria is vorticity  $\omega := \nabla \times \mathbf{u}$  with the vector field  $\mathbf{u}(\mathbf{x}) = (u_1, u_2, u_3)^T$ . Since  $\omega$  is a vector quantity, vorticity is not directly suitable for visualizations. Isosurfaces representing vorticity magnitude are, therefore, usually used to show vortical structures. Albeit vorticity gives a fairly good approximation of regions containing vortices, it is nevertheless inferior to more advanced vortex detection methods [1].



**Fig. 1** Simplified programming model for modern GPUs. The shaded boxes indicate units freely programmable by the user, the dashed path indicates multi-pass rendering.

A more sophisticated and widely used approach that also complies with the criteria defined for a GPU-based implementation is the  $\lambda_2$  method proposed by Jeong and Hussain [4, 9]. The  $\lambda_2$  method first decomposes the velocity gradient tensor  $\nabla \mathbf{u}$  (the Jacobian matrix of the vector field) into a symmetric part  $\mathbf{S}$  and an anti-symmetric part  $\mathbf{\Omega}$ :

$$\mathbf{S}_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right), \quad \mathbf{\Omega}_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i} \right).$$

From a physical point of view  $\mathbf{S}$  is the strain-rate tensor and  $\mathbf{\Omega}$  the spin tensor. Next, the eigenvalues of the matrix  $\mathbf{S}^2 + \mathbf{\Omega}^2$  need to be determined. Since this matrix is real and symmetric, there will be three eigenvalues denoted by  $\lambda_1 \geq \lambda_2 \geq \lambda_3$ . A vortex is then defined as a connected region where two of the eigenvalues are negative. The eigenvalue relevant for visualization is  $\lambda_2$ —hence the name of the method.

Obviously, both the most primitive and the advanced method employs information extracted from the Jacobian of the vector field. That is, vorticity-based vortex detection basically comes for free when a more advanced detection algorithm is to be implemented. It is therefore well worth a try to evaluate the vorticity approach first.

## 4 PROGRAMMING THE GPU

Until a few years ago, graphics hardware included only a fixed-function pipeline that was programmed by setting states and generating geometry. Starting with NVIDIA’s GeForce256 very limited programmability became possible with texture-combining modes. Nowadays, most of the graphics adapters shipped with off-the-shelf PCs include both programmable vertex and fragment processors (Fig. 1). Thus, the programmer is able to operate on each vertex provided by the application and each fragment generated by rasterization. Usually, these features are used to create advanced visual effects (per pixel lighting, shadows, refractions) in real-time. However, modern GPUs combine several geometry engines and rendering pipelines with a very high memory throughput, so what has entered the desks are actually quite powerful parallel processors. And programmability has made these accessible. This fact has been exploited by many researchers in various fields to create applications that outperform software-based solutions by up to two orders magnitude. Examples can be found not only in visualization but also in general numerical computing [7].

However, GPUs are not as easy to program as CPUs. This is *not* because of the programming environment since the advent of high-level shader languages like NVIDIA’s Cg [10] or Microsoft’s HLSL (virtually the same languages) has made it possible to write GPU programs with a C-like syntax. Rather, the reason is that the streaming model of GPUs does not support branching and loops with non-constant numbers of iterations. In addition, there are hard limits regarding the maximum number of instruction

slots<sup>1</sup> and variables that can be used by a program.

These limitations are highly vendor-dependent. E.g., on a NVIDIA GeForceFX the maximum number of instructions slots is 1,024 while there are only 64 arithmetic instruction slots and 32 texture lookups available on an ATI 9800.

For our implementation we have chosen the ATI 9800 graphics adapter and DirectX/D3D despite the lower instruction limit since on this platform multi-pass rendering (i.e. the use of previous computation results/pixel values in another rendering pass, see dashed path in Fig. 1) does involve only a very minor performance penalty. And as will be shown, multi-pass rendering is required if the number of redundant computations is to be reduced.

## 5 SYSTEM ARCHITECTURE

The system is divided into two parts: an initialization part that is executed once per dataset and the actual cycle that is entered each time the filter characteristics are adjusted by the user.

Prior to entering the cycle, the vector field (which is assumed to be defined on a Cartesian grid) must be loaded and preprocessed. The preprocessing consists of adding a one-cell border around the volume. The values of the border cells are chosen such that the gradients at the original cells can all be determined using central differences. This work-around relieves us from the burden of having to handle border cells differently from inner cells during gradient estimation. Thus, assuming original grid dimensions of  $I \times J \times K$ , a volume of the dimensions  $(I + 2) \times (J + 2) \times (K + 2)$  is obtained. This volume is then cut into  $K + 2$  slices each holding all the values of a constant Z-coordinate and each slice is copied to a 2D floating point RGBA-texture.

The cycle in turn can be further divided into three major parts: denoising of the raw vector data, computation of vorticity magnitudes, and rendering of the output of the vortex detection algorithm. Since vorticity magnitudes comprise scalar data, any volume rendering technique can be employed for visualization. Thus, part three can be titled the volume visualization phase. The following subsections describe these parts in detail.

### 5.1 Filtering

All measurements are subject to noise. To oppress this noise, a lowpass filter can be employed. Usually, a lowpass filter's ability to oppress noise greatly depends on its support, i.e. the number of neighbors incorporated into the calculation of the filtered value.

In a hardware implementation the neighbors need to be determined by lookups into textures filled by the application with the appropriate information. Thus, a filter of support  $N$  requires  $N^3$  texture lookups to obtain the neighbor information. As was said in Sec. 4, our target platform only supports 32 texture lookups per pass; thus, if the filter is not separable, the maximum filter support that can be implemented in a single pass is three. Of course, a filter of support three is very limited in its effect; thus, for our implementation we concentrated on separable filters requiring  $3 \times N$  texture lookups and a multi-pass implementation. Since the Gaussian lowpass filter is separable, isotropic (i.e. it well preserves oriented features) and easy to implement, this filter presents a suitable choice for our application.

---

<sup>1</sup>There is a one-to-many relation between *instructions* and *instruction slots* so differentiation between the two terms is inescapable.

The order of the passes of a separable filter is arbitrary and does not effect the result. However, the complexity of the implementation and the amount of temporary texture memory are very well effected. If the data is filtered first in X- and Y-direction, only a single input and output texture are required for the first two passes since the input data is stored as Z-aligned slices. However, the final Z-direction rendering pass requires  $N$  slices to have already been filtered in X- and Y-direction; thus,  $N + 1$  temporary floating point textures are required.

On the other hand, if the data is filtered first in Z-direction,  $N$  slices of the unfiltered vector data are used as input and the result is written to a single floating point texture. For the remaining passes, another floating point texture is required; thus, only two temporary textures are required (independent of the filter support) and no program logic is needed that caches the results of the XY-filtering for a final Z-filtering pass.

Therefore, the filtering is accomplished by rendering  $K + 2$  filled quadrilaterals of  $(I + 2) \times (J + 2)$  pixels. In the pixel shader, the current pixel's  $N$  neighbor values (including itself) are looked up. Since all neighbors are equidistantly spaced, the Gauss function is now evaluated at equidistantly spaced points to obtain the weights.

The passes consume  $N$  texture lookups each and 32 and 12 instructions slots for the filtering in X-/Y- and Z-direction, respectively. Since in either case at most  $N$  texture lookups are required, filter supports of up to 31 could theoretically be implemented with the number of texture lookups restricted to 32. However, on our platform the number of texture samplers is currently limited to 16 so the maximum filter support is 15.

### 5.2 Vortex Detection

Mapping a vorticity-based vortex detection to graphics hardware is straightforward when using HLSL. All that needs to done is calculating the Jacobian using central differences from the six surrounding neighbors of the volume element currently being processed:

```
// Determine first column of the Jacobian (x-direction)
forwardNeighbor = tex2D(slice1Sampler, IN.rightTexCoords);
backwardNeighbor = tex2D(slice1Sampler, IN.leftTexCoords);
gradientX = (forwardNeighbor - backwardNeighbor) * DOUBLE_DISTANCES_INV.x;

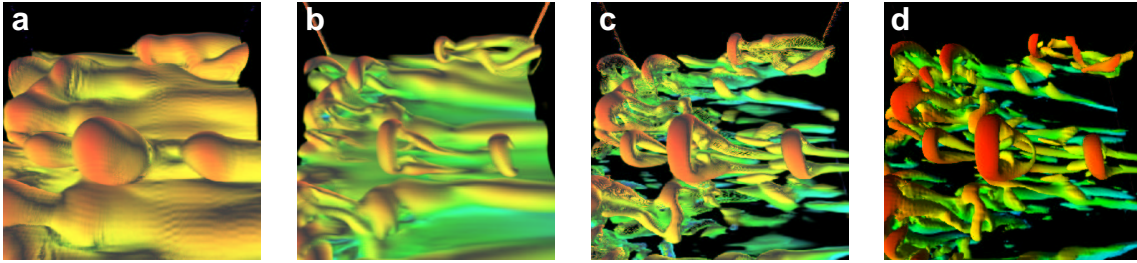
// Determine second column of the Jacobian (y-direction)
...

// Determine third column of the Jacobian (z-direction)
...

vorticity.x = gradientY.z - gradientZ.y;
vorticity.y = gradientZ.x - gradientX.z;
vorticity.z = gradientX.y - gradientY.x;

OUT.color = float4(length(vorticity), 0.0, 0.0, 0.0);
```

The resulting pixel shader code is—with six texture lookups and 16 arithmetic instructions—well within the limits of our hardware platform.



**Fig. 2** Isosurface visualizations of vortex structures in K-type transition simulation data ( $229 \times 116 \times 250$  voxels). **a)** Vorticity magnitude of original data **b)** Vorticity magnitude of filtered data **c)**  $\lambda_2$  values of filtered data **d)** Reference image generated by a commercial flow visualization tool (software implementation).

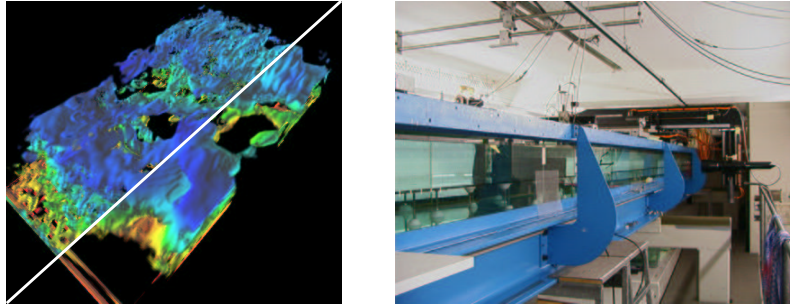
### 5.3 Volume Visualization

The input vector data is processed slice per slice and the resulting vorticity magnitudes again written to a stack of 2D textures since at the time of this writing rendering to 3D textures was not possible. To produce appealing visualizations using this stack of 2D textures, we adopted a solution proposed by Rezk-Salama et. al [12]. The basic idea is to determine the intersection polygons of viewport-aligned slices with the given stack of quadrilaterals and to interpolate color values on these polygons using the two neighboring textures.

A drawback of this approach is the large number of intersection polygons that have to be calculated each time the volume is rotated. Our implementation, therefore, employs a slightly different approach: Instead of determining viewport-aligned intersection polygons on-the-fly, we pre-compute sets of intersection polygons from the X- and Y-direction (both positive and negative), respectively, and switch between them (and the original stack) depending on the orientation of the volume’s bounding box to the viewer.

This approach not only enables us to pre-calculate the intersection polygons but also to send the geometry data of the slices only once to the graphics adapter as a vertex buffer. If the original stack’s slices are assumed to be equidistant, the amount of geometry stored in the vertex buffers can further be significantly reduced (by factors of  $I + 1$  and  $J + 1$ , respectively) by storing only a single stack of stripes and rendering the remaining ones with a suitable translation applied. Thus, the approach is able to accelerate the visualization without taking a noteworthy amount of memory.

Fig. 2 shows two visualizations of vortices detected with our system using a vorticity-based approach. Image a) shows the isosurface obtained from the original dataset, image b) the result when first applying a strong Gauss filter of kernel size 11. The dataset used for these screenshots was obtained by DNS of K-type transition experiments [6, 13]. This kind of data typically exhibits striking  $\Lambda$ - and  $\Omega$ -vortices. Obviously, these vortex structures are only found in the filtered data while the original data seems to cast a shroud over the vortex structures. However, the individual vortices are not even neatly separated when filtering the data—which comes at no surprise regarding the primitive vortex detection approach. On the other hand, this means that for a productive system a more sophisticated approach like the  $\lambda_2$ -method is required.



**Fig. 3** Left: Two  $\lambda_2$  isosurfaces extracted from experimental data with increasing filter width ( $200 \times 40 \times 110$  voxels). The right image shows the laminar water tunnel used in obtaining the dataset. Dataset and photograph courtesy IAG, University of Stuttgart.

## 6 $\lambda_2$ VORTEX DETECTION

A careful implementation of  $\lambda_2$  vortex detection requires almost twice as many instruction slots as are available on the ATI 9800. Since only four 32-bit floating point values can be passed from one pass to the next and expensive multiple render targets (MRTs) are to be avoided, this split-up must be chosen carefully.

Our solution is to calculate the coefficients of the characteristic polynomial in a first pass and to solve the characteristic equation in a second pass. Since the characteristic polynomial is a cubic, only four floating point numbers need to be passed between the passes. This data tightly fits into an RGBA value and thus allows us to abandon MRTs.

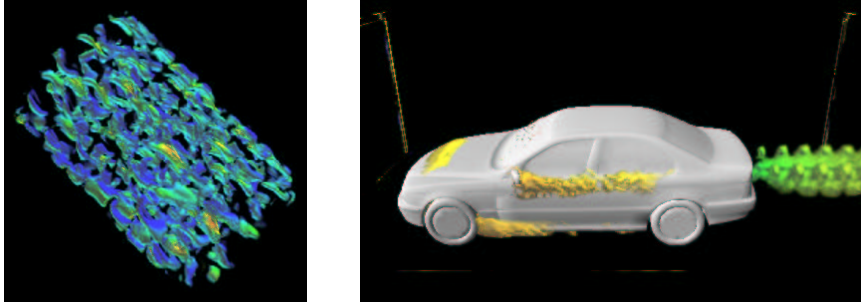
We will not go into detail on how the two passes are actually mapped to the graphics hardware since the basic approach is similar to what has already be described in Sec. 3 and fairly subtle and tedious details have to be coped with to stay within the instruction limit and to obtain an efficient implementation. Nevertheless, a GPU-based  $\lambda_2$  implementation *is* possible and a comparison with vorticity-based vortex detection (Fig. 2 b) and c)) well justifies the introduction of a more advanced—if also more time-consuming—detection algorithm.

## 7 RESULTS

### 7.1 Visual Evaluation

The presented system combines filtering and vortex detection capabilities. Thus, the system is particularly well suited to noisy data like, e.g., the datasets obtained by experiments. However, the system is also applicable to special types of simulation data.

Both Reynolds Averaged Navier-Stokes (RANS) and Large Eddy simulations (LES) make use of increased viscosity. Increased viscosity basically means that small-scale structures will effectively be eliminated through diffusion and dissipation. Neither RANS simulations nor LES will, therefore, result in very noisy data. On the contrary, DNS (Direct Numerical Simulation) should capture all scales that are relevant in the flow without any turbulence modeling. Large Reynolds number flows typically mean small viscosity, and—particularly for turbulent flows—very small length scales in comparison to a relevant geometric length scale. Very small length scales, in turn, means high grid resolution. If insufficient



**Fig. 4** Vortices extracted on the GPU. Left: DNS dataset of a flow through a cylinder filled with spheres ( $201 \times 152 \times 152$  voxels). Right: Simulated flow around a car body (RANS simulation,  $301 \times 131 \times 100$  voxels). Datasets courtesy LSTM, University of Erlangen, and BMW AG.

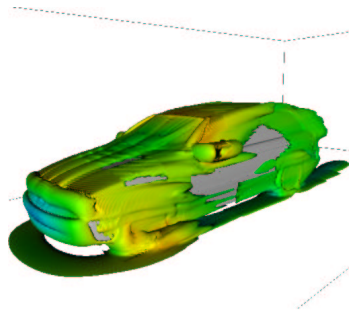
resolution is available, the smallest scales will be determined by the numerics, and not by the physics, which will result in noisy data. We have, therefore, taken care to include both experimental data and data resulting from DNS for the evaluation of our system.

Fig. 3 and 4 show screenshots of isosurfaces extracted from experimental data, DNS data, and RANS data. Albeit the generated visualizations are very satisfying, they do not make definitive statements about the quality of the visualized data. For this reason we have created reference images for our datasets using the commercial flow visualization software *PowerVIZ* [3]. Fig. 2 c) and d) contrast two  $\lambda_2$  visualizations of the K-type transition dataset. *PowerVIZ* implements geometry-based isosurface extraction, does not reveal all its internal settings used for producing the visualizations (perspective transformation, color tables, etc.) and uses higher-precision arithmetic than is currently available on the ATI 9800 (24 bit). Exactly matching the images is, therefore, virtually impossible. However, the visualizations are nevertheless almost identical and give evidence of the quality that can be obtained with hardware-accelerated implementations.

## 7.2 Storage Requirements

Our GPU-based implementation stores the input data as textures. However, the amount of texture memory is 256 MB on our platform; thus, memory efficiency is a crucial topic in the evaluation of graphics-hardware-accelerated implementations.

Our system stores the raw input data in a 128 bit RGBA texture. This texture must not be modified since it is needed each time the filter characteristics are adjusted. To store the filtered results, another 128 bit floating point texture is required. This texture is reused for storing the gradients of the  $\lambda_2$  scalar field required for lighting the isosurface. The remaining textures are independent of the size of the input data and of negligible size. Thus, if the input data comprises  $N$  nodes, about  $32N$  byte memory are consumed by the system. Assuming 256 MB texture memory, this means that the system is applicable to datasets of at most eight million grid points or equivalently a cube of the dimensions  $200 \times 200 \times 200$ . We consider this to be sufficient for experimentally obtained datasets and many simulation datasets. However, both the filtering and the  $\lambda_2$  vortex detection are local. Thus, if the system limits are surpassed, a bricking approach can be easily used to accommodate the system to larger datasets.



**Fig. 5** Shear layer visualization of the simulated flow around a car body. Dataset courtesy BMW AG.

### 7.3 Performance Evaluation

For evaluating the performance of the system a dataset of the dimensions  $135 \times 225 \times 129$  was visualized at a viewport size of  $512 \times 512$  pixels on an ATI 9800 XT graphics adapter (price at time of this writing: about \$400). Since the system is almost completely GPU-based, the configuration of the underlying PC/workstation is irrelevant for the benchmark.

We have found the filtering time to be 165 ms for a Gauss filter of support 11 and a  $\lambda_2$  computation time of 117 ms. The gradient calculation required for lighting the isosurface took 16 ms. These times are independent of the number of intermediate slices and the direction from which the volume is looked at. On the other hand, the rendering time depends strongly on the direction: For the Z-direction we measured a visualization time of 41 ms (24.4 fps), for the Y-direction a time of 229 ms (4.4 fps). As expected, the frame rates scale linearly with the amount of pixels generated by the application.

In order to get a reference value for the computation performance, we implemented the  $\lambda_2$  vortex detection algorithm on an Intel Pentium 4 processor using SSE2 vector instructions. Using this implementation, we measured 1,150 ms for the  $\lambda_2$  vortex detection on the  $135 \times 225 \times 129$  dataset—almost an order of magnitude slower compared to the hardware-based approach.

As expected, PowerVIZ, a tool well-known for its generally good performance is even significantly slower (about 8 s) than the optimized software implementation. This must probably be ascribed to more complex internal data structures since PowerVIZ works on a hierarchy of Cartesian Grids while our system requires a single Cartesian grid. Anyway, using an own optimized implementation for comparison is well justified.

## 8 CONCLUSIONS

We have described a system for filtering, vortex detection, and visualization of flow data. By employing modern graphics hardware for performing the calculations instead of the CPU, we were able to improve the system performance by almost an order of magnitude. For the first time, the cycle of filtering, vortex detection, and visualization can be handled interactively using low-cost off-the-shelf hardware readily available at the desks of many researchers.

In the future, we plan to evaluate further local vortex detection algorithms and more advanced filters in terms of portability to graphics hardware. Furthermore, we intend to investigate GPU-based shear layer computations (see Fig. 5 for an example calculated by PowerVIZ) to be able to provide the fluid dynamics engineer with a richer set of features to be explored interactively.

## Acknowledgements

We are grateful to the German Research Council (DFG) for financing this work as part of SPP 1147 and to Milosz Walter and Martin Schulz of science+computing AG for making available a prototype of PowerVIZ capable of calculating and visualizing  $\lambda_2$  values. Furthermore, we would like to thank all the people that provided us with experimental and simulated flow dataset, namely Ulrich Rist and Mark Linnick (IAG, University of Stuttgart), Oliver Theissen (BMW AG), and Thomas Zeiser (LSTM, University of Erlangen).

## REFERENCES

- [1] J. P. Bonnet. *Post-Processing of Experimental and Numerical Data*. Von Karman Institute for Fluid Dynamics, 2002.
- [2] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH*, pages 263–270. ACM Press, 1993.
- [3] Exa Corporation. PowerVIZ specifications, 2001. <http://www.exa.com/pdf/PowerVIZscreen.pdf>.
- [4] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, pages 69–94, 285 1995.
- [5] M. Jiang, R. Machiraju, and D. Thompson. Detection and visualization of vortices. In *Visualization Handbook*. Academic Press, 2003.
- [6] Y. S. Kachanov. Physical mechanisms of laminar-boundary-layer transition. *Annu. Rev. Fluid Mech.*, pages 411–482, 26 1994.
- [7] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.
- [8] N. Max and B. Becker. Flow visualization using moving textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data*, 1995.
- [9] K. Müller, U. Rist, and S. Wagner. Enhanced visualization of late-stage transitional structures using vortex identification and automatic feature extraction. In *Computational Fluid Dynamics*, pages 786 – 791. John Wiley & Sons, 1998.
- [10] NVIDIA Corp. Cg Language Specification, 2002. Available at <http://developer.nvidia.com/cg>.
- [11] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramée, and H. Doleisch. Eurographics 2002 STAR – State of The Art Report Feature Extraction and Visualisation of Flow Fields, 2002.
- [12] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118,147. Addison-Wesley Publishing Company, Inc., 2000.
- [13] U. Rist and Y. S. Kachanov. Numerical and experimental investigation of the K-regime of boundary-layer transition. *Laminar-Turbulent Transition*, pages 405–412, 1995.
- [14] S. Stegmaier, M. Schulz, and T. Ertl. Resampling of Large Datasets for Industrial Flow Visualization. In *Workshop on Vision, Modelling, and Visualization VMV '03*, pages 375–382. infix, 2003.
- [15] D. Weiskopf, G. Erlebacher, and T. Ertl. A Texture-Based Framework for Spacetime-Coherent Visualization of Time-Dependent Vector Fields. In *Proceedings of IEEE Visualization '03*, pages 107–114, 2003.