Numerical Simulations on PC Graphics Hardware

Jens Krüger, Thomas Schiwietz, Peter Kipfer, Rüdiger Westermann

Technische Universität München, Munich, Germany,
jens.krueger@in.tum.de, schiwiet@in.tum.de, kipfer@in.tum.de,
westermann@in.tum.de
WWW home page: http://wwwcg.in.tum.de

Abstract. On recent PC graphics cards, fully programmable parallel geometry and pixel units are available providing powerful instruction sets to perform arithmetic and logical operations. In addition to computational functionality, pixel (fragment) units also provide an efficient memory interface to local graphics data.
To take full advantage of this technology, considerable effort has been spent on the development of algorithms amenable to the intrinsic parallelism and efficient communication on such cards. In many examples, programmable graphics processing units (GPUs) have been explored to speed up algorithms previously run on the CPU. In this paper, we will demonstrate the benefits of commodity graphics hardware for the parallel implementation of general techniques of numerical computing.

1 Introduction

Over the last couple of years, the evolution of GPUs has followed a tripled Moores law, currently providing up to 222 million transistors compared to 50 million on an Intel P4 Northwood CPU. Recent GPUs can be thought of as stream processors, which operate on data streams consisting of an ordered sequence of attributed primitives like vertices or fragments. GPUs can also be thought of as SIMD computers, in which a number of processing units simultaneously execute the same instructions on stream primitives. At various stages in the rendering pipeline, GPUs provide parallel and fully programmable processing units that act on the data in a SIMD-like manner. Each of these units itself is a vectorized processor capable of computing up to 4 scalar values in parallel.

In recent years, a popular direction of research is leading towards the implementation of general techniques of numerical computing on such hardware. The results of these efforts have shown that for compute bound applications as well as for memory bandwidth bound applications the GPU has the potential to outperform software solutions [1–3, 6].

To initialize computations, the application program specifies polygons to be rendered by sending a vertex stream to the GPU. This stream is automatically distributed to the vertex units, where currently up to 6 of these units work in parallel on the specified vertex information. The result of this computation is

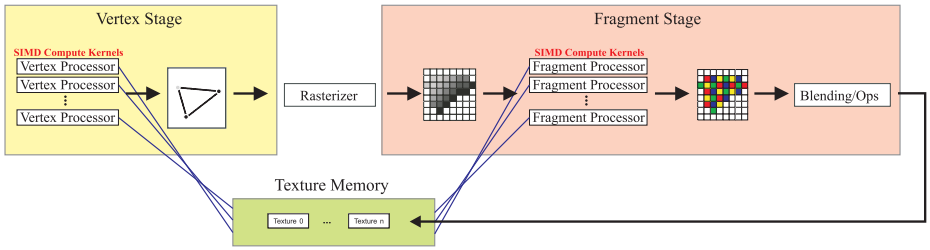

Fig. 1. This figure illustrates the rendering pipeline as it is realized on recent PC graphics cards.

sent to the rasterizer. The rasterizer maps the input vertex stream to a fragment stream by computing the coverage of polygons in screen space. For each covered pixel one fragment is generated thus increasing the number of stream primitives in general. The fragment stream is processed by currently up to 16 fragment units, which work in parallel on up to 4 scalars at a time. Vertex and fragment units can concurrently access shared DDR3 memory. Such fetches can be executed parallel to other operations, as long as these operations do not depend on the retrieved value. In this way, memory access latency can often be hidden. Figure 1 shows an overview of the basic architecture.

Both vertex and fragment processing units provide powerful instruction sets to perform arithmetic and logical operations. C-like high level shading languages [4, 5]in combination with optimizing compilers allow for easy and efficient access to the available functionality. In addition, GPUs also provide an efficient memory interface to local data, i.e. random access to texture maps and frame buffer objects. Not only can application data be encoded into such objects to allow for high performance access, but rendering results can also be written to such objects, thus providing an efficient means for the communication between successive rendering passes.

2 Numerical Simulation on GPUs

To realize numerical simulations on the GPU, we have implemented a general linear algebra framework that is based on the internal representation of vectors and matrices as 2D textures. On top of these representations, intrinsically parallel and streamable vector-vector and matrix-vector operations have been implemented very efficiently. To simulate computations on a 2D grid, only one quadrilateral covering as many fragments in screen space as there are grid cells has to be rendered. The rasterizer generates a stream of exactly this number of fragments, one for every vector/matrix component. In the fragment units, arithmetic operations between pairs of input values are performed, and the result is directly written to the output stream.

To combine the elements of one vector, we employ the well known reduce-operation as implemented on parallel architectures. Therefore, quadrilaterals at

ever smaller size are rendered in consecutive rendering passes. In every pass, each fragment reads four adjacent data values from the previous rendering result and combines them using the specified operation. This process is repeated until one single value is left, thus enabling the reduction of one vector in logarithmic time.

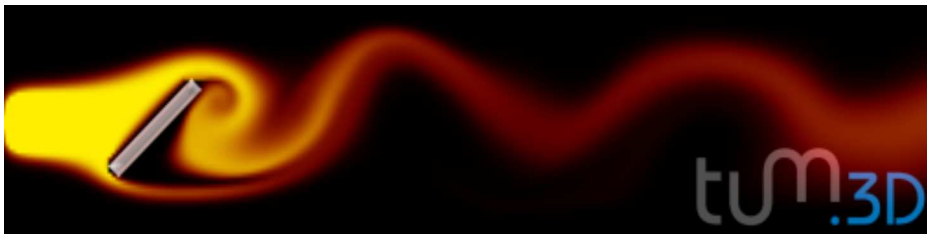

Fig. 2. Simulation of the Karman Vortex Street on a 256x64 Grid is shown. The simulation and rendering runs at 400 fps using our GPU framework on current ATI hardware

The described linear algebra operations have been encapsulated into a C++ class framework, on top of which we have implemented implicit solvers for systems of algebraic equations. Using this framework, we demonstrate the solution to the incompressible Navier-Stokes equations in 2D at quite impressive frame rates, including boundary conditions, obstacles and simulation of velocities on a staggered grid (see Figure 2).

Besides the fact that the GPU-based simulation framework outperforms CPU-based solvers of about a factor of 10-15, running the simulation on the GPU has another important advantage: Because simulation results already exist on the GPU, they can be rendered immediately. In the following example these results are rendered by injecting dye into the flow and by advecting the dye according to the simulated flow field. Having the data already in graphics memory avoids any data transfer between the CPU and the GPU.

Incompressible Navier-Stokes equations

Fluid phenomena like water and gaseous media can be simulated by the incompressible Navier-Stokes equations. We solve for the velocity $V = (u, v)^T$ governed by the these equations

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{1}{Re}\nabla^2 u - V \cdot \nabla u + f_x - \frac{\partial p}{\partial x} \\ \frac{\partial v}{\partial t} &= \frac{1}{Re}\nabla^2 v - V \cdot \nabla v + f_y - \frac{\partial p}{\partial y} \\ \nabla \cdot V &= 0 \end{aligned}$$

in two passes. First, by ignoring the pressure term an intermediate velocity is computed. The diffusion operator is discretized by means of central differences,

and, as proposed in [11], we solve for the advection part by tracing the velocity field backward in time. To make the resulting intermediate vector field free of divergence, pressure is used as a correction term. Mass conservation of incompressible media leads to a Poisson-Equation for updating this pressure term. This equation is solved using a Conjugate-Gradient method build upon the numerical framework described above.

In addition to the plain solution to the Navier Stokes equations as described above we have integrated the following extensions to improve the simulation.

- We have integrated a mechanism that allows one to arbitrarily specify inflow regions and characteristics. Therefore, the current inflow settings are stored in the color components of a 2D texture map, which is interpreted as external forces acting on the flow in every iteration of the simulation process.
- Instead of a collocated grid the entire simulation is run on a staggered grid as shown in Figure 3. In this way, centered space derivatives use successive points of the same variable, and the dispersion characteristics is improved because the effective grid length is halved. Furthermore, we can now handle arbitrarily positioned obstacles exhibiting special boundary conditions very effectively.
- To preserve vorticity on the regular staggered grid we have integrated vorticity confinement [12] into our simulation code. At each grid point, a fragment shader computes

  $$v_c = \tilde{n} \times (\nabla \times \omega)$$

  where $\tilde{n} = \nabla|\omega|/|\nabla|\omega||$ is a unit vector pointing towards the centroid of the vortical region, and ω is the vorticity vector. This vector is added to the velocity to convect vorticity towards the centroid.

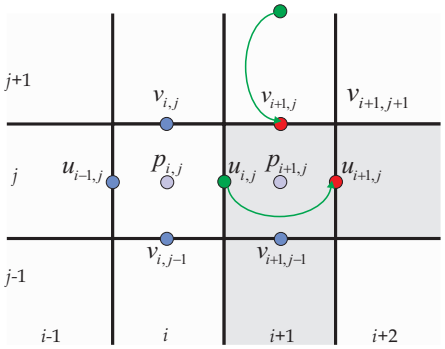

Fig. 3. The image shows a staggered grid with obstacles (grey) and how the velocity values at the obstacle borders are mirrored to match the border conditions.

In addition to the plain numerical simulation on the GPU we will now given an example how the results of this computation, which is stored in the graphics

memory can be reused for visualization. This visualization is done without the bus transfer bottleneck.

3 GPU Particle Tracing

In real-world fluid flow experiments, external materials such as dye, hydrogen bubbles, or heat energy are injected into the flow. The advection of these external materials can create stream lines, streak lines, or path lines to highlight the flow patterns. Analogies to these experimental techniques have been adopted by scientific visualization researchers. Numerical methods and three-dimensional computer graphics techniques have been used to produce graphical icons such as arrows, motion particles, stream lines, stream ribbons, and stream tubes that act as three-dimensional depth cues.

Over the last decade, such methods have been investigated intensively in terms of numerical accuracy and grid structures, as well as acceleration, implementation and perception issues. While these techniques are effective in revealing the flow fields local features, they lack in that they can usually not produce *and* display the amount of graphical icons needed to visually convey large amounts of three-dimensional directional information at interactive rates.

This is due to the following reasons: First, both numerical and memory bandwidth requirements imposed by accurate particle integration schemes are too high as to allow for the simultaneous processing of large particle sets. Second, even in case that the graphical icons to visualize the flow characteristics can be computed at sufficient rates, rendering of these icons includes the transfer of data to the graphics system and thus limits the performance significantly.

To overcome the limitations of classical particle based techniques and global imaging techniques we provide a system for real-time integration and rendering of large particle sets. Contrarily to topology or feature based techniques, which aim at extracting relevant flow structures thus reducing the information to be displayed at once [7–10], we attack the problem of 3D vector field visualization by means of interactivity. We provide the user with a mechanism to interactively guide the exploration of flow structures at arbitrary resolution. Our approach enables virtual exploration of high resolution flow fields in a way similar to real-world experiments.

Therefore, we have integrated numerical integration schemes into a GPU system for interactive exploration of flow fields [13]. It takes advantage of OpenGL memory objects (*SuperBuffers*) to store particle positions on the graphics card. Programmable fragment shaders implement interpolation methods up to order 3, and construct stream lines and stream bands. Since memory objects can either be interpreted as texture maps accessible in the fragment shader program or as vertex arrays used as input to the geometry units, particle tracing can be entirely performed on the GPU without any read back to application memory.

Our implementation exploits a feature of recent ATI graphics hardware that allows graphics memory to be treated as a render target, a texture, or vertex data. This feature is presented to the application through an extension to

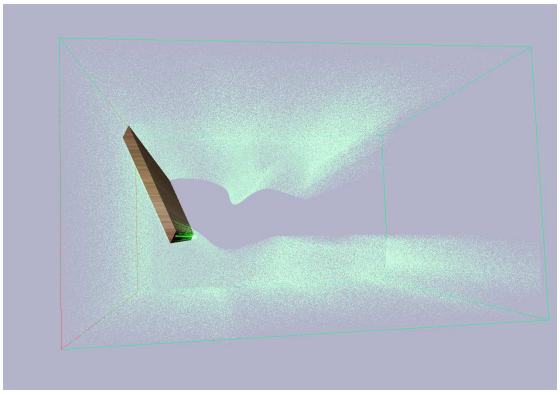

Fig. 4. High performance particle tracing on the GPU.

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 OpenGL's semantic meaning of the data. Thus, SuperBuffers provide an efficient mechanism for storing GPU computation results and later using those results for subsequent GPU computations.

In figure 4 we demonstrate the effectiveness of our approach for particle integration and rendering. About 42 millions of particles per second are traced through the flow using an Euler integration scheme. This number is reduced to 21 millions and 12 millions, respectively, using schemes of 2^{nd} and 3^{rd} order accuracy, respectively. It is interesting to note that a CPU version of particle tracing in 3D flow fields roughly takes about a factor of 30 longer than the GPU version. This is mainly due to the following reasons: First, the advection step extremely benefits from the numerical compute power and high memory bandwidth of the parallel fragment units on our target architecture. Second, the transfer of that many particles to the GPU in every frame of the animation imposes a serious limitation on the overall performance.

4 Future Work

In order to process large simulation domains, we can exploit distribution and parallelization functionality present in common multiprocessor architectures For interactive steering of simulations, we envision a distributed approach that makes use of the low-latency node interconnectors and high performance intra-node bus architectures to provide adequate user response times.

The whole system is initialized by computing a regular partition of the simulation domain in parallel that has the least edge-cut property. The obtained

information is processed by a program on each node, which assembles the data for all GPUs on this node. A dedicated thread is responsible for uploading the data to the GPUs in order to avoid synchronization delays. Now each GPU processes a part of the sub-domain that is obtained by a regular split. The boundary conditions are read-back and are redistributed to the adjacent regions in a hierarchical way by first updating the boundaries on each node and then between the nodes on a peer-to-peer basis. Note that this perfectly exploits the high bandwidth on the upcoming PCI-Express graphics bus and reduces at the same time the bandwidth needed on the node interconnect. The packets transferred at this level are therefore small and allow efficient inter-node transport across the low latency interconnector. After completing a configurable number of simulation time-steps, a visualization of the simulation result is computed on each client node and the partial results, i.e. images, are transferred back to the master node for display. The packet size of this final communication step is rather big, so we can efficiently maintain the high bandwidth PCI-Express path on the client nodes all through to the master node.

5 Conclusion

In this work, we have described a general framework for the implementation of numerical simulation techniques on graphics hardware. For this purpose, we have developed efficient internal layouts for vectors and matrices. By considering matrices as a set of diagonal or column vectors and by representing vectors as 2D texture maps, matrix-vector and vector-vector operations can be accelerated considerably compared to software based approaches.

In order to demonstrate the effectiveness and the efficiency of our approach, we have described a GPU implementation to numerically solve the incompressible Navier-Stokes equations. The results have shown that recent GPUs can not only be used for rendering purposes, but also for numerical simulation and integration. The combination of shader programs to be executed on the GPU and new concepts like memory objects allow one to carry out numerical simulations efficiently and to directly visualize the simulation results.

References

1. Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Computer Graphics (Proc. SIGGRAPH '03)*.
2. Bolz, J., Farmer, I., Grinspun, E., and Schroeder, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *Computer Graphics SIGGRAPH 03 Proceedings*.
3. Moreland, K and Angel, E. 2003. The FFT on a GPU. *SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*.
4. Microsoft, 2002. DirectX9 SDK. http://www.microsoft.com/DirectX.

5. Mark, W., Glanville, R., Akeley, K., and Kilgard, M. 2003. Cg: A system for programming graphics hardware in a C-like language. In *ACM Computer Graphics (Proc. SIGGRAPH '03)*.
6. N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary-value problems using programmable graphics hardware. In *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*
7. L. Hesselink and T. Delmarcelle. *Scientific visualization - advances and challenges*, chapter Visualization of vector and tensor data sets, pages 367–390. Academic Press, 1994.
8. R. Peikert and Roth. M. The 'parallel vectors' operator - a vector field visualization primitive. In *Proceedings IEEE Visualization 99*, pages 263–271, 1999.
9. A. Telea and J. Wijk. Simplified representation of vector fields. In *Proceedings IEEE Visualization 99*, pages 35–43, 1999.
10. Frits H. Post, Benjamin Vrolijk, Helwig Hauser, Robert S. Laramee, and Helmut Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.
11. J. Stam. Stable fluids. *Computer Graphics SIGGRAPH 99 Proceedings*, pages 121–128, 1999.
12. John Steinhoff and David Underhill. Modification of the euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluids Vol 6(8)*, pages 2738–2744, 1994.
13. Peter Kipfer, Mark Segal and Rüdiger Westermann. UberFlow: A GPU-Based Particle Engine To appear in *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware 2004*.