Navier-Stokes on Programmable Graphics Hardware using SMAC

CARLOS EDUARDO SCHEIDEGGER[1], JOÃO LUIZ DIHL COMBA[1], RUDNEI DIAS DA CUNHA[2]

[1]II/UFRGS–Instituto de Informática, Universidade Federal do Rio Grande do Sul - Porto Alegre, RS, Brasil
{carlossch, comba}@inf.ufrgs.br
[2]IM/UFRGS–Instituto de Matemática, Universidade Federal do Rio Grande do Sul - Porto Alegre, RS, Brasil
rcunha@mat.ufrgs.br

Abstract. Modern programmable graphics hardware offers sufficient computing power to suggest the implementation of traditional algorithms on the graphics processor. This paper describes a complete implementation of a standard technique to solve the incompressible Navier-Stokes fluid equations running entirely on the GPU: the SMAC (Simplified Marker And Cell) method. This method is widely used in engineering applications. The described implementation works with general rectangular domains, with or without obstacles, and with a variety of boundary conditions. Furthermore, we show that our implementation is about sixteen times faster than a reference CPU implementation running on similar cost hardware. Finally, we discuss simple extensions to the method to deal with more general situations, such as free boundary-value problems and three-dimensional domains.

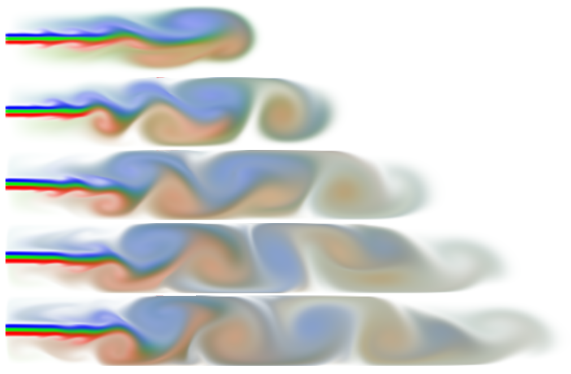
Figure 1: A 1024×128 Navier-Stokes simulation running in interactive rates, $Re = 10000$.

1 Introduction

Harnessing the modern programmable graphics hardware processing power for general computation is a very active area of research [1] [4] [6]. Although this is not a new idea [9] [12], it was only recently that the graphics hardware used in consumer-level personal computers reached interesting levels, both in terms of raw performance and programmability.

Nowadays, modern Graphics Processing Units (*GPU*s) have a full 32-bit floating-point pipeline, with programmable vertex and fragment shading units. This allows us to interpret the GPU as a *stream processor* [13] [3], where streams are defined as sets of independent uniform data. This is the main advantage that a GPU has over a CPU: since computation on pieces of the stream are independent from each other, it is possible to use multiple functional units to process the data efficiently, in parallel.

Of course, not every problem decomposes itself gracefully in independent pieces: one must find a way to adapt the algorithm to the restrictions that the GPU imposes. A GPU algorithm is a carefully constructed sequence of graphics API calls, with textures serving as storage for arrays and data structures, and vertex and fragment programs performing the computation. In this work, we use NVIDIA's NV35 and NV40 architectures. An implementation of this kind requires a thorough understanding of the interplay between the different parts of the graphics system, as, for example, the different pipeline stages and respective capabilities, CPU/GPU communication issues and driver and API quirks.

In this work we demonstrate how to cast SMAC [5], a computational fluid dynamics algorithm used in engineering applications, as one such carefully constructed sequence. We will see that in some cases, this GPU version outperforms a single-CPU reference implementation by as much as 21 times; on average our GPU implementation runs sixteen times faster.

2 Related Work

Stam's stable fluids [15] are a standard computer graphics technique for the simulation of fluid dynamics. Stam's solver relies on the Hodge decomposition principle and a projection operator. Being an unconditionally stable solver, it is able to use much larger timesteps than explicit solvers, that are typically stable only under certain conditions. Although Stam's solution to the Navier-Stokes equations produce visually pleasing fluids, the implicit solver creates too much numerical dissipation. This deteriorates the solution to the point where it has no more relation to fluids in real life. Since we are interested in using the GPU as a numerical co-processor, we must not allow experimental discrepancies in the simulations.

Stable fluids running on graphics hardware are abundant in the literature [1] [10]. Also related is Goodnight et al.'s multigrid solver [4], which is used to solve the stream portion of a stream-vorticity formulation of the Navier-Stokes equations. Harris et al. [8] show that a variety of natural phenomena can be simulated efficiently in graphics hardware. Harris et al. [7] also show a simulation of cloud dynamics running in graphics hardware, using Stam's stable fluids as the dynamics engine.

Recently, Buck et al. [14] developed a data-parallel programming language that allows the user to abstract away from the graphics API, letting the compiler convert the code to the specific calls. This is a major step towards the perception of the GPU as a viable computing platform by the general developer.

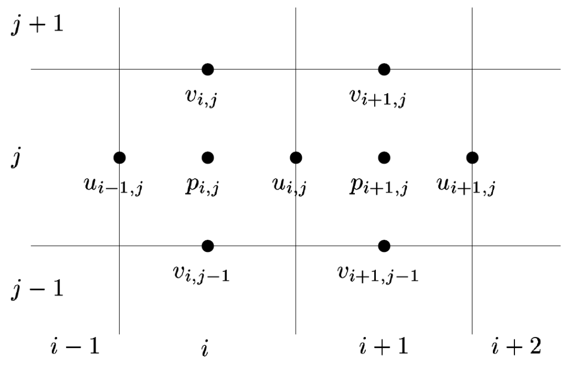

Figure 2: Staggered grid discretization.

3 The SMAC Method

3.1 The Navier-Stokes Equations

The Navier-Stokes equations are a standard tool for dealing with fluid dynamics, and the SMAC method relies on a discretization of these equations. The incompressible Navier-Stokes equations, in their vector form, are:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{1}{\rho}\nabla p + \nu \nabla^2 u + g, \tag{1}$$

$$\nabla \cdot u = 0 \tag{2}$$

where u is the velocity vector field and p is the pressure scalar field. ν and ρ are the viscosity and the density of the fluid, and g represents external forces acting on all of the fluid (gravity, for example). Our implementation uses the adimensional, two-component cartesian version of the equations:

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \tag{3}$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y, \tag{4}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{5}$$

where Re is the Reynolds number, relating viscous and dynamic forces.

3.2 Boundary Conditions and Domain Discretization

We assume a rectangular domain $[0, w] \times [0, h] \subset \mathbb{R}^2$ in which we restrict the simulation. This means we have to deal with the appropriate boundary conditions along the borders of the domain. We implemented boundary conditions that simulate walls, fluid inflow, and fluid outflow, which allow a variety of real-life problems to be modeled. The wall and inflow are Dirichlet boundary conditions: the velocity field has a certain fixed value at the boundary. The outflow condition is different. Since this condition is not a physical condition, we approximate the outflow condition by assuming that the fluid that leaves the domain is uninteresting, and behaves exactly as the neighborhood of the boundary that is inside the domain. This gives us Neumann conditions: the derivative of the velocity field is fixed across the boundary (in our case, at zero). In the following, we first explain the simpler case of domains without obstacles.

To solve the equations numerically, we approximate the rectangular subset of $\mathbb{R}^2$ with a regular grid, ie. the velocity and pressure scalar fields are sampled at regular intervals. We discretized the domain using a *staggered grid*, which means that different variables are sampled in different positions. This representation is used because of its better numerical properties [5]. The grid layout for our simulation is shown in Figure 2.

The boundary conditions in the grid are simulated by adding a *boundary strip*. The boundary strip is a line surrounding the grid cells that will be used to ensure that the desired boundary condition holds. In Figure 3, we show one corner of the boundary strip.

We can set the boundary strip appropriately to create the boundary conditions. Consider for example the wall boundary condition, where the velocity components must vanish. The values that are sampled directly on the boundary can simply be set to zero. For values that are not sampled directly on the boundary, we assume a linear interpolation of the fields, and set the boundary strip value so that the average of the two values becomes zero. This can be applied to all boundary conditions.

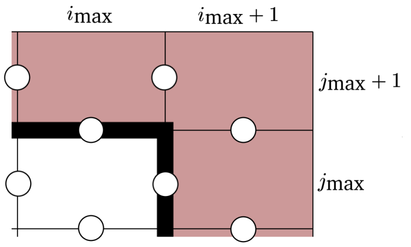

Figure 3: The thicker line is the boundary, and the shaded cells are the boundary strip. The circles represent the sampling positions. Notice that some points on the boundary are not sampled, hence the need for a boundary strip.

3.3 Discretization of the Equations

The Navier-Stokes equation will be solved by time-stepping: from known velocities at time t, we compute new values at time $t + \Delta t$. The values in the varying timesteps will be called $u^{(0)}, u^{(1)}, \ldots$. To discretize the Navier-Stokes equations, we first introduce the following equations:

$$F = u^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} - g_x \right] \quad (6)$$

$$G = v^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} - g_y \right] \quad (7)$$

Rearranging 3 and 4, and discretizing the time variable with forward differences, we have

$$u^{(n+1)} = F - \delta t \frac{\partial p}{\partial x}, v^{(n+1)} = G - \delta t \frac{\partial p}{\partial y} \quad (8)$$

This gives us a way to find the values for the velocity field in the next step. F and G, when discretized, will depend only on known values of u and v and can be computed directly. We use central differences and a hybrid donor cell scheme for the discretization of the quadratic terms, following the reference CPU solution [5]. We are left to determine the pressure values. To this end, we substitute the continuous version of Equations (8) into Equation (5) to obtain a Poisson equation:

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\partial t} \left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y} \right) \quad (9)$$

When discretizing the pressure values, we notice that we cannot compute them directly. We have to solve a system of linear equations, with as many unknowns as there are pressure samples in the grid. This can be solved with many different methods, such as Jacobi relaxation, SOR, conjugate gradients, etc. With the pressure values, we can determine the velocity values for the next timestep, using Equations 8. We then repeat the process for the next timestep.

3.4 Stability Conditions

SMAC is an explicit method, and, as most such methods, is not unconditionally stable. To guarantee stability, we have to make sure that these inequalities hold:

$$\frac{2\delta t}{Re} < \left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2} \right)^{-1} \quad (10)$$

$$|u_{max}| \delta t < \delta x \quad (11)$$

$$|v_{max}| \delta t < \delta y \quad (12)$$

Here, δt, δx and δy refer to the timestep sizes, and distance between horizontal and vertical grid lines, and u_{max} and v_{max} are the highest velocity components in the domain. During the course of the simulation, δx and δy are fixed, so we must change δt accordingly. In practice, one wants to use a safety multiplier $0 < s < 1$ to scale down δt.

4 The Implementation in a GPU

In this section we show the GPU implementation of the SMAC method using NVIDIA's NV35 and NV40 architectures. First we show how the data structures are stored into texture memory, followed by the presentation of all programs used to implement the algorithm.

4.1 Representation

We use a set of floating-point textures to store the values of the velocity fields and intermediate variables. These textures are more precisely called *pbuffers*, or *pixel buffers*, because they can be also the target of a rendering primitive, similar to writing to the frame buffer. We will have five *pbuffers*:

- **uv**: This will store the velocity field. One channel will store the u values, and the other will store v.
- **FG**: This *pbuffer* will be used to store the intermediate F and G values, each on one channel.
- **p**: This *pbuffer* will store the pressure values.
- **ink**: This *pbuffer* will store ink values, not used in the simulation but used for the visualization of the velocity field.
- **r**: This auxiliary buffer will be used in *reduction* operations described later.

We will have an additional status texture that will signal whether a cell is an obstacle or a fluid cell. For now, the only obstacles are the wall boundary conditions. It is important to mention that the NV35 and the NV40 do not allow simultaneous reads and writes to the same same surface [11], which are needed by some iterative algorithms. To circumvent this problem, we use a standard GPU technique called *ping-pong rendering* , which alternates between writing to texture a while reading from texture b and vice-versa. Therefore, the **r**, **uv** and **p** *pbuffers* have two surfaces, and take twice the amount of memory.

4.2 Setting the Boundary Conditions

The first step in the algorithm is to enforce the boundary conditions. A fragment program that reads the velocity values and the status texture gets the necessary texture offsets and determines the correct velocity components for the boundaries. We need to use the right offsets because boundaries in different directions are determined from different neighbors. All of our boundary conditions can be calculated with one fragment program when we notice that they share a common structure: for each component (in the 2D case, only u and v), we only need to sample a direct neighbor. Then, the boundary conditions are of the following form:

$$u_{ij} = \alpha u_{ij} + \beta u_{\text{neighbor}}$$

We store the appropriate α, β values, along with the offsets to determine the neighbor, in the status texture. If the cell happens not to be a boundary cell, we only set $\alpha = 1, \beta = 0$. This fragment program is used to render a domain-sized quadrilateral.

4.3 Computing FG

The velocity field with enforced boundary conditions is used to compute the **FG** buffer. The **FG** *pbuffer* is computed simply by rendering another domain-sized quad, using the **uv** *pbuffer* as input, and a fragment program that represents the discretization of Equations (6) and (7).

4.4 Determining Pressure Values

With the **FG** values, we can now determine the pressure value. As mentioned above, we must solve the equation system generated by the Poisson equation discretization. In CPUs, SOR is the classical method used to solve these systems, because of the low memory requirements and the good convergence properties. The main idea of SOR is to use, in iteration it, not only the values of the pressure in the iteration $it-1$, but the values in it that have just been calculated. In a GPU, unfortunately, we cannot do that efficiently: it would require reading and writing the same texture simultaneously.

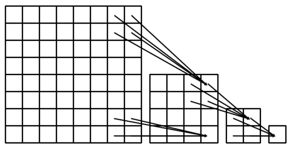

Figure 4: Combining all elements in a SIMD architecture through reductions.

The solution we adopted is to implement Jacobi relaxation as a fragment program. To check for convergence, we must see if the norm of the residual has gone below a user-specified threshold. The norm is a computation that combines all of the values in a texture, differently from every other fragment program described so far. The GPU is used to dealing with streams of independent data, so we must find a special way of doing the calculation.

What we implement is called a *reduction*. In each reduction pass, we combine values of a local neighborhood into a single cell, and recursively do this until we have but one cell. This cell will hold the result of the combination of all original cells. Figure 4 illustrates the process. Not only this computation is significantly more expensive than the relaxation step, there is a measurable overhead in switching between fragment programs and *pbuffers*. We use a more clever scheme to reduce the number of switches: instead of computing the residual at each relaxation step, we adaptively determine whether a residual calculation is necessary, based on previous results using an exponential backoff algorithm. That is, we calculate the residual for the i^{th} time only after 2^i relaxation steps. After the first pressure solution is determined, we use the number of relaxation steps that were necessary in the previous timestep as an estimate for the current one. This results in significantly better performance.

4.5 Computing the $t^{(n+1)}$ Velocity Field

After computing the pressure values, we can determine the velocity field for the next timestep using Equation (8). This is done by another fragment program that takes the appropriate textures and renders, again, a domain-sized quad. The final step is ensuring that the stability conditions (10), (11) and (12) hold.

The first condition is easy to determine, since it is constant for all timesteps and can be pre-calculated. The other ones, though, require the computation of the maximum velocity components. This is an operation that requires a combination of all the grid values, and again a reduction is needed. This time, though, we use the maximum of the neighbors instead of the sum as the reduction operation.

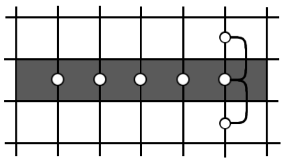

Figure 5: An ambiguous obstacle: should the boundary strip use the north or the south cell?

4.6 Obstacles

To implement obstacles, we simply extend the idea used in the wall boundary condition to general places inside the domain. The status texture will hold a special value to denote a wall for visualization purposes, but there is no need to change the boundary fragment program. The original formulation handles the walls seamlessly.

One must take into account, however, that not all domain configurations are valid. The main problem are thin lines, in which the boundary condition is underspecified, as can be seen in Figure 5. This can be easily fixed with a finer subdivision or with a thicker boundary, so it is not a real issue.

4.7 Visualization

Usually, the simulation of Navier-Stokes is not fast enough to allow interactivity, and so the results are simply stored in a file to be interpreted later. We instead take advantage of the fact that the simulation runs at interactive rates, and that the data is already in the graphics memory to implement interactive visualization tools.

We could visualize the velocity field directly as intensities in textures, but most interesting features would be missed in this way. We developed a visualization tool inspired on the use of colored smoke in real-life airflow visualization. We store, in addition to the velocity fields, an *ink field*, which is a passive field that does not affect the velocity in any way. The ink field is advected by the velocity field, and the motion of the ink is used to visualize features such as vortices. *Ink emitters* of different colors can be arbitrarily placed and moved around in the domain, allowing to investigate areas of flow mixture or separation.

The advection step occurs right after the boundary conditions are enforced. A first shot in an algorithm for the advection would be to get the current velocity at the center of the cell, and, using the timestep value, determine the position for this parcel of fluid. This has two issues: the first one is that we would have to write to different cells, because the timestep never takes an ink particle more than a grid width or height (consider the stability conditions for the discretization). Aside from that, there's a more serious problem: we don't know, prior to running the fragment program, what are the cells in which to write our results. This is known as a *scatter* operation [1], and is one that is missing from GPUs: the rasterization issues a fixed output place for each fragment. We need to change the scatter operation by a *gather* one: an operation in which we don't know, prior to running the program, what are the cells we will *read*. This is implemented in GPUs through the use of *dependent texturing* [8]. We can switch the scatter operation for a gather operation using the idea illustrated in Figure 6. Instead of determining the position that the ink in the present position will be, we will determine what portion of ink was in a past position. To do this, we assume that the velocity field is sufficiently smooth, and we use a step backward in time using the present velocity. We have to ensure that the velocity is sampled in the center of the grid cell, because that's where the ink is sampled. This requires an appropriate interpolation of the velocities.

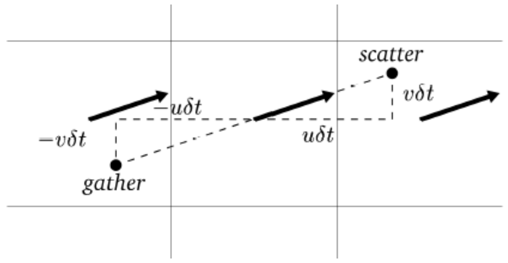

Figure 6: Stepping backward in time to avoid a scatter operation.

This is certainly not the only way of implementing visualizations of vector fields; see, for instance, [2] and [16].

5 Results

To judge the performance of the GPU implementation, we compared our solution to a CPU reference code provided by Griebel et al [5]. We used a classical CFD verification problem, the *lid-driven cavity*. The problem begins with the fluid in a stationary state, and the velocity of the fluid is created by the drag of a rotating lid. This is a *steady* problem, no matter what are the conditions such as Reynolds number and lid velocity: when t increases, the velocity field tends to stabilize. Knowing this, we run the simulations until the changes in the velocity field are negligible.

We conducted our tests using two different Reynolds numbers and three different grid sizes. The results can be seen in Table 1. Figure 7 shows the ratio of improvement of the GPU solution. The CPU is a Pentium IV running at 2 GHz, and the GPUs are a GeForce FX 5900(NV35) and GeForce 6800 Ultra(NV40). Both programs were compiled with all optimization options enabled, using Microsoft Visual Studio .NET 2003.

CPU	32×32	64×64	128×128
Re = 100	1.73s	35.71s	428.05s
Re = 1000	5.52s	122.47s	903.63s
NV35	32×32	64×64	128×128
Re = 100	3.36s	13.34s	60.29s
Re = 1000	6.14s	28.60s	110.36s
NV40	32×32	64×64	128×128
Re = 100	1.54s	5.29s	30.79s
Re = 1000	2.11s	9.15s	42.89s

Table 1: Timings for CPU, NV35 and NV40 solutions, respectively.

As we can see, the only case where the GPU was outperformed by the CPU was in very small grids with the NV35. This is a situation where convergence is very quick, and the overhead due to *pbuffer* switches [1] probably overshadowed the parallel work of the GPU. Also, the ratio between GPU computation and CPU-GPU communication was smallest in this case. In all the other situations, the GPU implementation was significantly faster, with the NV40 achieving an impressive speedup factor of 21 in large grids with large Reynolds numbers. In fact, all of the figures of this paper were generated by taking screenshots of the application running interactively.

5.1 Quality

To judge the quality of the GPU implementation when compared to the reference CPU implementation, we ran both programs with exactly the same problem specifications, and compared the velocity fields at each timestep. In our experiments, the difference between velocity components computed in the two programs was always less than 10^{-2}, and most of the time less than 10^{-3}. The problems had velocity ranges between 0 and 1. The largest differences were found in high pressure areas, probably due to the difference between the Jacobi and the SOR algorithms.

The reference CPU implementation didn't allow for general domains, so for that part of the implementation, we had to rely on qualitative measurements. For example, we expect vortices around corners with high speed fluid, and we can see this in Figure 10. Some well-known phenomena, such as the *Kárman vortex street* [5], were also experienced in our software, in accordance to experimental results. See Figure 13.

6 Analysis

The GPU achieves top performance when doing simple calculations on massive amounts of data, and this is the case in our algorithm. Most of the computation is done on the

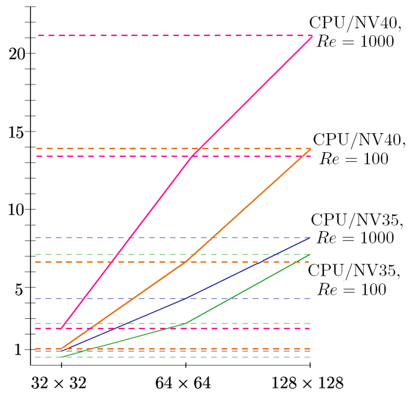

Figure 7: Ratio between GPU and CPU timings

GPU. The CPU is only used to orchestrate the different fragment programs and buffers, to adjust the timestep appropriately and to determine the convergence of the Poisson equation.

Measuring the amount of time taken in each part of our algorithm, we noticed that more than 95% percent of the time was spent solving the Poisson equation. This was the main motivation for the exponential backoff residual calculation step. This change doubled the overall performance. We could have implemented a multigrid solver for the Poisson equation, such as the one developed by Goodnight et al. [4]. This would have meant a very significant performance increase. We chose not to do so because we did not have a suitable CPU multigrid code to compare to, and we did not want to skew the results in either way. For a real-life application, a multigrid solver would probably have meant another order-of-magnitude performance increase.

In the simulation depicted in Figure 1, we have a 1024×128 grid, and the simulation runs at approximately 20 frames per second, allowing real-time visualization and interaction.

7 Future Work and Conclusion

The Navier-Stokes GPU solver shown here can be easily extended to three dimensions. The **uv** and **FG** *pbuffers* would have to hold an additional channel. Additionally, we can't use 3D textures as *pbuffers*, so the texture layout would probably follow [7]. The fragment programs would not fundamentally change, and the overall algorithm structure would stay the same.

A more ambitious change is to incorporate free boundary value problems to our solver. In this class of problems, we not only have to determine the velocity field for the fluid, but we have to determine also the interface between the fluid and the exterior (for sloshing fluid simulations, for example). The approach that is proposed in the SMAC algorithm is to, starting with a known fluid domain, place particles throughout the domain and then displace them according to the velocity field. At the next timestep, the algorithm checks whether any particles arrived in cells that had no fluid. These cells are then appropriately marked, and the simulation continues. We can't do that directly on the GPU, because that would require a scatter operation. A possible solution is to use the *volume-of-fluid* method [5]. The volume-of-fluid method keeps track of the fraction of the fluid that leave the cells through the edges. This way, all cells that are partially filled are marked as border cells, the ones completely filled are marked as fluid cells, and the ones without any fluid are marked as empty cells. Such a scheme could be implemented using GPUs, since the calculation of fluid transfer between cells can be done for each cell individually, without having to write to arbitrary locations. However, this remains to be implemented.

Nevertheless, we have shown that the GPU is a viable computing engine for the complete solution of the Navier-Stokes via a explicit solver, suitable for engineering contexts. Our solution takes advantage of the streaming nature of the GPU and minimizes the CPU/GPU interaction, resulting in the high performances reported. We hope that the fact that GPU performance growth is largely out-pacing the CPU will serve as an additional motivation for the implementation of other similar applications.

8 Acknowledgments

The authors would like to thank NVIDIA for providing the graphics hardware used in this paper, specially the NV40 reference board and drivers. The authors would like to specially thank Nick Triantos for a most informative talk at the University of Utah about the NVIDIA architectural issues.

References

[1] J. Bolz, I. Farmer, E. Grinspun, P. Schröder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003), 2003.

[2] B. Cabral, L. Leedom. *Imaging vector fields using line integral convolution*. Proceedings of SIGGRAPH 1993.

[3] J. Comba, C. Dietrich, C. Pagot, C. Scheidegger. *Computation on GPUs: From A Programmable Pipeline to an Efficient Stream Processor*. Revista de Informática Teórica e Aplicada, Volume X, Número 2, 2003.

[4] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys. *A Multigrid Solver for Boundary-Value Problems Using Programmable Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

[5] M. Griebel, T. Dornseifer, T. Neunhoffer, *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.

[6] N. Govindaraju, S. Redon, M. Lin, D. Manocha. *CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

[7] M. Harris, W. Baxter III, T. Scheuermann, A. Lastra. *Simulation of Cloud Dynamics on Graphics Hardware*. proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

[8] M. Harris, G. Coombe, T. Scheuermann, A. Lastra. *Physically-Based Visual Simulation on Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2002.

[9] G. Kedem, Y. Ishihara. *Brute Force Attack on UNIX passwords with SIMD Computer*. Proceedings of the 8th USENIX Security Symposium, 1999.

[10] J. Krüger, R. Westermann. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003), 2003.

[11] NVIDIA Corporation. *OpenGL Extension Specifications*. Web site last visited on May 17th, 2004. `http://developer.nvidia.com/object/nvidia_opengl_specs.html`

[12] E. Larsen, D. McCallister. *Fast Matrix Multiplies using Graphics Hardware*. Supercomputing 2001.

[13] T. Purcell, I. Buck, W. Mark, P. Hanrahan. *Ray Tracing on Programmable Graphics Hardware*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002), 2002.

[14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. To appear in ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), 2004.

[15] J. Stam. *Stable Fluids*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 1999), 1999.

[16] J. van Wijk. *Image-based Flow Visualization*. ACM Transaction on Graphics (Proceedings of SIGGRAPH 2002), 2002.

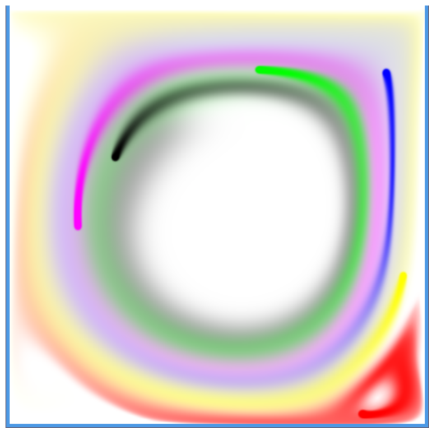

Figure 8: Lid-driven cavity, 256×256 grid, $Re = 10000$. Note the counter-eddies in the corners: these are experimentally confirmed for large Reynolds numbers.

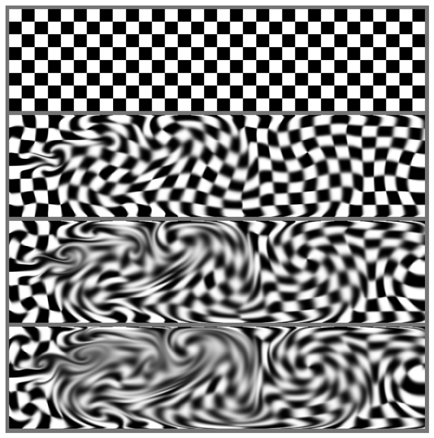

Figure 9: Using regular patterns to visualize the flow. 1024×256 grid, $Re = 1000$, small inflow in the west boundary, outflow throughout the east boundary

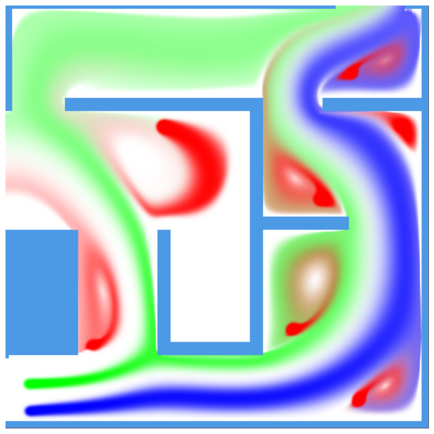

Figure 10: Domain with obstacles. 128×128 grid, $Re = 1000$, Inflow in the lower west boundary, outflow on the other exits.

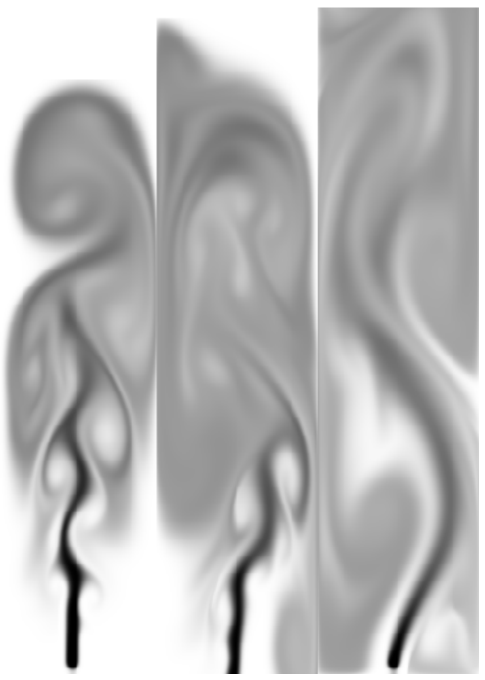

Figure 11: Smoke simulation with large reynolds numbers. 128×1024 grid, $Re = 10000$, small inflow in the south boundary, outflow throughout the north boundary

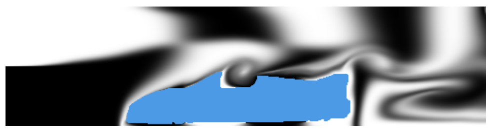

Figure 12: Wind tunnel mock-up. 256×64 grid, $Re = 100$, inflow throughout the west boundary, outflow throughout the east boundary.

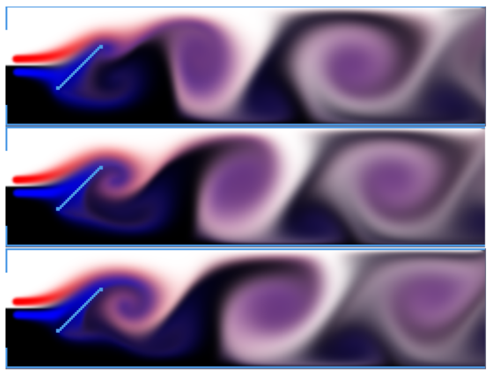

Figure 13: The *Kárman vortex street*. 256×64 grid, $Re = 1000$, inflow in the west boundary, outflow in the east boundary.