Linear Solvers for Stable Fluids: GPU vs CPU

Gonçalo Amador Abel Gomes
Instituto de Telecomunicações,
Departamento de Informática, Universidade da Beira Interior
Covilhã, Portugal
a14722@ubi.pt, agomes@di.ubi.pt

Abstract

Fluid simulation has been an active research field in computer graphics for the last 30 years. Stam's stable fluids method, among others, is used for solving equations that govern fluids. This method solves a sparse linear system during the diffusion and move steps, using either relaxation methods (Jacobi, Gauss-Seidel, etc), Conjugate Gradient (and its variants), or others (not subject of study in this paper). A comparative performance analysis between a parallel GPU-based (using CUDA) algorithm and a serial CPU-based algorithm, in both 2D and 3D, is given with the corresponding implementation of Jacobi (J), Gauss-Seidel (GS) and Conjugate Gradient (CG) solvers.

Keywords

Stable Fluids, CUDA, GPU, Sparse Linear Systems.

1. INTRODUCTION

The stable fluids method was introduced by Stam [Stam 99] to the field of computer graphics. It allows for unconditionally stable physics-based fluid simulations. During the study of this method, it became clear that during its diffusion and move steps a sparse linear system had to be solved. The performance and scalability (maximum grid size that can be used in real-time) of this method is directly related to the efficiency of the solver. Solvers that converge to better values give better visual results. However, the solver must converge quickly, but not at the cost of more computation, to allow real-time simulations. Interestingly, and in spite of existing more than one alternative to solve sparse linear systems (J, GS, CG, etc), an implementation (to the specific problem of stable fluids) and an comparative analysis of the various solvers on different architectures (GPU using CUDA and CPU) is hard to find, not to say that they do not exist.

Solvers have to iterate repeatedly and update elements of a grid. For each solver, the elements of the grid can be accessed and updated in an asynchronous way (with some minor changes in the GS solver). Therefore, this is a kind of problem where clearly we get performance gains in using parallel computing resources such as GPUs. Since the release of CUDA, i.e. an API for GPU processing, the subject of parallel computation on GPU has become more and more attractive.

This paper addresses to: how to code these solvers on GPU, evaluation of the gains and drawbacks of GPU implementations, and is it possible to improve the scalability of the stable fluids method using CUDA.

So, the main contributions of this paper are:

- CUDA-based implementation of stable fluids solvers in 3D. There is already a CPU-based implementation for stable fluids in 3D using a Gauss-Seidel solver, which is due to Ash [Ash 05]. There is also a Cg shading-based implementation using a Jacobi solver for 3D stable fluids [Keenan 07]. However, at our best knowledge, there is no CUDA-based implementation of 3D stable fluids.
- A comparative study of CUDA-based implementations of 3D stable fluids using different solvers, namely: Gauss-Seidel, Jacobi, and Conjugate Gradient.

This paper is organised as follows. Section 2 reviews the previous work. Section 3 briefly describes the NVIDIA Compute Unified Device Architecture (CUDA). Section 4 describes the method of stable fluids, including the Navier-Stokes equations. Section 5 deals with the implementation of the three sparse linear solvers mentioned above. Section 6 carries out a performance analysis of both CPU- and GPU-based implementations of the solvers. Finally, Section 7 draws relevant conclusions and points out new directions for future work.

2. PREVIOUS WORK

Since the appearance of the stable fluids method due to Stam [Stam 99], much research work has been done with

this method. To solve the sparse linear system for an 2D simulation, in the diffusion and move steps, both the Fast Fourier Transform (FFT) in Stam's [Stam 01] and the Gauss-Seidel relaxation in Stam's [Stam 03] were used (both implementations run on the CPU). The Gauss-Seidel version was the only that could support internal and moving boundaries (tips in how to implement internal and moving boundaries are given in Stam's [Stam 03]). Later on, in 2005, Stam's Gauss-Seidel version was extended to 3D, also for the CPU, by Ash [Ash 05]. In 2007, Ash's 3D version of stable fluids, was implemented for C'Nedra (open source virtual reality framework) by Bongart [Bongart 07]; this version also runs on the CPU. Recently, in 2008, Kim presented in [Kim 08] a full complexity and bandwidth analysis of Stam's stable fluids version [Stam 03].

In 2005, Stam's stable fluids version was implemented on the GPU by Harris [Harris 05] using the Cg language. This version supported internal and moving boundaries, but used Jacobi relaxation instead of the Gauss-Seidel relaxation. In 2007, an extension to 3D of Harris'work, was carried out by Keenan et al [Keenan 07]. Still in 2007, when CUDA (see [Nickolls 08] for a quick introduction to CUDA programming) was released, Goodnight's OpenGL-CUFFT version of Stam's [Stam 01] became available [Goodnight 07]. The code of this implementation is still part of the CUDA SDK examples.

The stable fluids method addressed the stability problem of previous solvers, namely Kass and Miller method in [Kass 90]. The Kass and Miller method could not be used for large time steps; consequently it was not possible to use it in real-time computer graphics applications. The reason behind this was related to the usage of explicit integration, done with the Jacobi solver, instead of stable fluids implicit integration. In spite of the limitations of Kass and Miller method, in 2004, there was a GPU Cg-based version of this method implemented by Noe [Noe 04]. This GPU version is used in real-time, thanks to the gains obtained from using the GPU.

As earlier mentioned, it is hard to find, not to say that it does not exist, an comparative analysis of the various solvers on different architectures (GPU and CPU). However, one might find performance analysis, comparing the CPU and GPU versions, of individual methods. In 2007, the CG method performance for the CPU and GPU was analysed by Wiggers et al. in [Wiggers 07]. In 2008, Amorim et al. [Amorim 08]) implemented the Jacobi solver on both CPU and GPU, having then carried out a performance analysis and comparison of both implementations.

Implementations of solvers, for the GPU using shading language APIs have already been addressed, namely: Gauss-Seidel and Conjugate Gradient (in [Kruger 03]), and Conjugate Gradient and multigrid solvers (in [Bolz 03]).

To understand the mathematics and algorithm of CG method (and variants) the reader is referred to Shewchuck [Shewchuk 94]. Carlson addressed the problem of applying the CG method to fluids simulations [Carlson 04]. The Preconditioned CG (PCG) was also overviewed during SIGGRAPH 2007 fluid simulation course [Bridson 07], where the source code for a C++ implementation of a sparse linear system solver was made available. SIGGRAPH version builds up and stores the non-null elements of the sparse matrix. When the sparse matrix is stored in, memory access efficiency is crucial for a good GPU implementation, which is a problem that was addressed by Bell and Garland [Bell 08].

An overview on GPU programming, including the GPU architecture, shading languages and recent APIs, such as CUDA, intended to allow the usage of the capabilities of GPUs parallel processing was addressed in [Owens 08].

In spite of the previous work described above, a CUDA implementation of Gauss-Seidel and CG solvers for the specific case of the stable fluids method seems to be non-existent. This paper just proposes such CUDA-based implementations. Besides, this paper also carries out a comparison between Jacobi, Gauss-Seidel and CG solvers, in their CPU and CUDA implementations.

3. NVIDIA COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

With the advent of the first graphics systems it became clear that the parallel work required by graphics computations should be delegated to another component other than the CPU. Thus, the first graphics cards arrived to alleviate graphics processing load of the CPU. However, the graphics programming was basically done using a kind of assembly language. With the arrival of the graphics programming APIs (such as OpenGL or DirectX) and latter the high-level shading languages (such as Cg or GLSL), programming graphics became easier.

When, in 2007, NVIDIA CUDA (Programming Guide [NVIDIA 08a] and Reference Manual [NVIDIA 08b]) was released, it was made possible to specify how and what work should be done in the NVIDIA graphics cards. It became possible to program directly the GPU, using the C\C++ programming language. CUDA is available for Linux and Windows operative systems, and includes the BLAS and the FFT libraries.

The CUDA programming logic separates the work (see Fig. 1) that should be performed by the CPU (host) from the work that should be performed by the GPU (device). The host runs its own code and launches kernels to be executed by the GPU. After a kernel call, the control returns immediately to the host, unless a lock is activated with `cudaThreadSynchronize`. The GPU and the CPU work simultaneously after a kernel call, each running its own code (unless a lock is activated in the host). The GPU runs the kernels by order of arrival ($kernel$ 1 then $kernel$ 2 as shown in Fig. 1), unless they are running on different streams, i.e. kernel execution is asynchronous. If a lock is activated the next kernel will be only called when the previously called kernels finish their jobs.

A kernel has a set of parameters, aside from the pointers

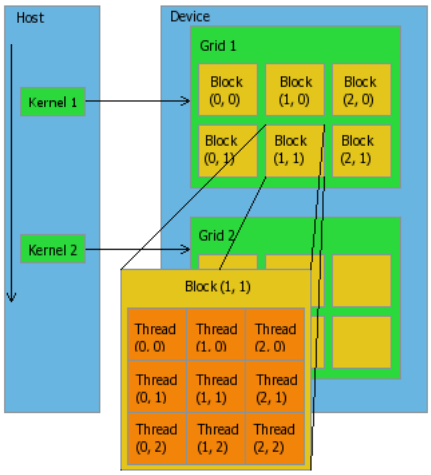

Figure 1. CUDA work flow model.

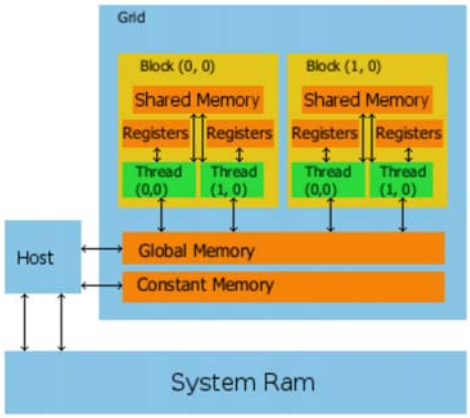

Figure 2. CUDA memory model.

to device memory variables or copies of CPU data. The parameters of a kernel specify the number of blocks in a grid (in 2D only), the number of threads (in x, y, z directions) of each grid block, the size of shared memory per block (0 by default) and the stream to use (0 by default). The maximum number of threads allowed by block is 512 ($x \times y \times z$ threads per block). All blocks of the same grid have the same number of threads. Blocks work in parallel either asynchronously or synchronously. With this information the kernel specifies how the work will be divided over a grid.

When talking about CUDA, four kinds of memory are considered (see Fig. 2). Host memory refers to the CPU-associated RAM, and can only be accessed by the host. The device has three kinds of memory: constant memory, global memory and shared memory. Constant and global memory are accessible by all threads in a grid. Global memory has read/write permissions from each thread of a grid. Constant memory only allows read permission from each thread of a grid. The host may transfer data from RAM to the device global or constant memory or vice-versa. Shared memory is the memory shared by all threads of a block. All threads within the same block have read-/write permissions to use the block shared memory.

4. STABLE FLUIDS

The motion of a viscous fluid can be described by the Navier-Stokes (NS) equations. They are differential equations that establish a relation between pressure, velocity and forces during a given time step. Most physically based fluid simulations use NS equations to describe the motion of fluids. These simulations are based on three NS equations. One equation just ensures mass conservation, and states that variation of the velocity field equals zero ($\bigtriangledown v = 0$). The other two equations describe the evolution of velocity (Eq. 1) and density (Eq. 2) over time as follows:

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla) u + v\nabla^2 u + f \tag{1}$$

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla) \rho + k\nabla^2 \rho + S \tag{2}$$

where u represents the velocity field, v is a scalar describing the viscosity of the fluid, f is the external force added to the velocity field, ρ is the density of the field, k is a scalar that describes the rate at which density diffuses, S is the external source added to the density field, and $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ is the gradient.

NS-based fluid simulators usually come with some sort of control user interface (CUI) to allow for the distinct users to interact with the simulation (see steps 2 and 3 in Algorithm 1). In order to solve the previous equations, NS-based fluid simulators work as follows:

Algorithm 1 NS fluid simulator.

Output: Updated fluid at each time-step

1: **while** simulating **do**
2: Get forces from UI
3: Get density source from UI
4: Update velocity (Add force, Diffuse, Move)
5: Update density (Add force, Advect, Diffuse)
6: Display density
7: **end while**

To better understand velocity and density updates, let us detail steps 4 and 5 in Algorithm 1.

4.1 Add force (f term in Eq. 1 and S term in Eq. 2)

In this step the influence of external forces to the field is considered. It consists in adding a force f to the velocity field or a source S to the density field. For each grid cell the new velocity u is equal to its previous value u_0 plus the result of multiplying the simulation time step Δt by the

force f to add, i.e. $u = u_0 + \Delta t \times f$. The same applies to the density, i.e. $\rho = \rho_0 + \Delta t \times S$, where ρ_0 is the density previous value, ρ is the new density value, Δt is the simulation time step, and S is the source to add to the density.

4.2 Advect ($-(u.\nabla)\,u$ term in Eq. 1 and $-(u.\nabla)\,\rho$ term in Eq 2)

The fluid moves according to the system velocity. When moving the fluid transports objects, densities, itself (self-advection) and other quantities. This is referred as advection. Note that advection of the velocity also exists during the move step.

4.3 Diffuse ($v\nabla^2 u$ term in Eq. 1 and $k\nabla^2$ term in Eq. 2)

Viscosity describes a fluid's internal resistance to flow. This resistance results in diffusion of the momentum (and therefore velocity). To diffuse, we need to solve, for the 3D case, the following equation for each grid cell.

$$D^{n+1}_{i,j,k} - \frac{kdt}{h^3}\left(D^{n+1}_{i-1,j,k} + D^{n+1}_{i,j-1,k} + D^{n+1}_{i,j,k-1} + D^{n+1}_{i+1,j,k} + D^{n+1}_{i,j+1,k} + D^{n+1}_{i,j,k+1} - 6D^{n+1}_{i,j,k}\right) = D^{n}_{i,j,k} \tag{3}$$

In both cases we will have to solve a sparse linear system in the form $Ax = b$.

4.4 Move ($-(u.\nabla)\,u$ term in Eq. 1) and $\bigtriangledown v = 0$

When the fluid moves, mass conservation must be ensured. This means that the flow leaving each cell (of the grid where the fluid is being simulated) must equal the flow coming in. But the previous steps (Add force and diffuse for velocity) violate the principle of mass conservation. Stam uses a Hodge decomposition of a vector field (the velocity vector field specifically) to address this issue. Hodge decomposition states that every vector field is the sum of a mass conserving field and a gradient field. To ensure mass conservation we simply subtract the gradient field from the vector field. In order to do this we must find the scalar function that defines the gradient field. Computing the gradient field is therefore a matter of solving, for the 3D case, the following Poisson equation for each grid cell.

$$\begin{aligned} &P_{i-1,j,k} + P_{i,j-1,k} + P_{i,j,k-1} + \\ &+P_{i+1,j,k} + P_{i,j+1,k} + P_{i,j,k+1} - 6P_{i,j,k} = \\ &= (U_{i+1,j,k} - U_{i-1,j,k} + V_{i,j+1,k} - \\ &-V_{i,j-1,k} + W_{i,j,k+1} - W_{i,j,k-1})h \end{aligned} \tag{4}$$

Solving this Poisson equation, for the 3D case, for each grid cell is the same as solving a sparse symmetrical linear system. This system can be solved with the solver used in the diffuse step (J, GS or CG) as described in the next section.

5. SOLVER ALGORITHMS

As previously mentioned, the density diffusion, the velocity diffusion, and move steps require for a sparse linear system to be solved. To best understand the kind of problem at hand let us assume we are going to simulate our fluid in a 2^2 grid domain (blue cells in Fig. 3 on the left). This means that our grid will actually be a 4^2 grid, where the fluid is inside a container. So the extra cells are the external boundaries of the simulation (red cells in Fig. 3 on the left). To allow a better memory usage, we represent the grid as a 1D array, with 4^2 elements (as shown in Fig. 3 on the left). For a 3D simulation eight 1D arrays are required. Velocity requires six 1D arrays, two for each of its components, i.e. current and previous values of velocity (vx, vx_0, vy, vy_0, vz, vz_0). Density will require the remaining two 1D arrays (from the eight), for its current and previous values (d, d_0).

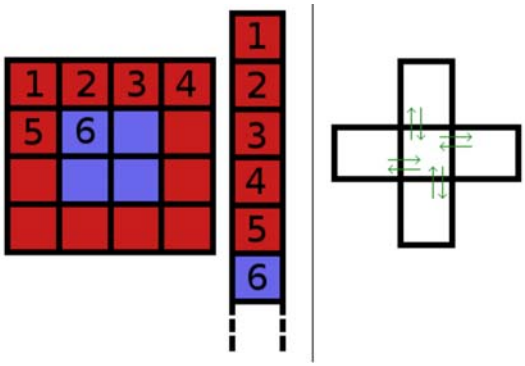

Figure 3. 2D Grid represented by a 1Darray (left), and grid cell interacting with its neighbours (right).

During the density diffusion, and the velocity diffusion and move steps, each cell in the grid interacts with its direct neighbours (see Fig. 3 on the right). In a 4^2 grid, there would be a total of 4^2 interactions between a cell and its neighbours. Let us consider one of the 1D array pairs, for example for the velocity y component previous (vy_0) and current values (vy). If we took the interactions for each fluid cell we would obtain a linear system in the form $Ax = b$ (see Fig. 4).

In this system A is a Laplacian matrix of size 16^2, and its empty cells are zero. For diffusion and move steps a system in this form has to be solved. To do so one can either build and store A in memory, using a 1D array or a data structure of some kind, or to use its values directly. In the second option, this means that the central cell value is multiplied by -4 in 2D or by -6 in 3D, and its direct neighbours are multiplied by 1.

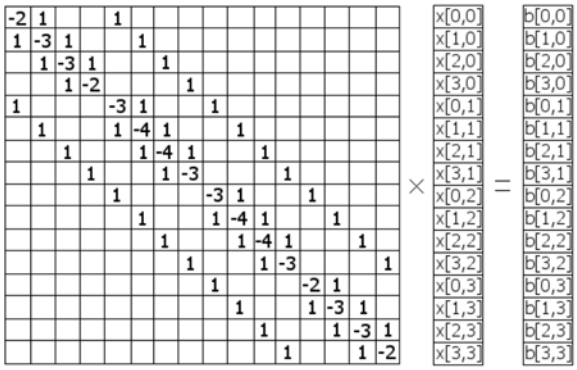

Figure 4. The sparse linear system to solve (for a 2^2 fluid simulation grid).

5.1. Jacobi and Gauss-Seidel Solvers

The Jacobi and Gauss-Seidel solvers, for a given number of iterations (line 1 in Algorithms 2 and 3) for each cell of the grid (line 2 in Algorithms 2 and 3) will calculate the cell value (line 3 in Algorithms 2 and 3). What distinguishes both solvers is that Gauss-Seidel uses the previously calculated values, but Jacobi does not. Therefore Jacobi convergence rate will be slower when compared to the Gauss-Seidel solver. Since the Jacobi solver does not use already updated cell values it requires the storage of the new values, in a temporary auxiliary 1D array (aux). When all new values of cells have been determined, the old values of cells will be replaced with the values stored in the auxiliary 1D array (lines 5 to 7 in Algorithm 2). After some maths (not addressed in this paper) the diffusion and move equations to solve can be made generic where only $iter$ and a will differ (line 3 in Algorithms 2 and 3).

Algorithm 2 CPU based Jacobi.

Input:
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
aux: auxiliary 1D array.
a: $\frac{kdt}{h^3}$ (see Eq. 3).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3).
max_iter: number of times to iterate.
Output:
x: 1D array with the grid new interpolated values.

1: **for** $iteration = 0$ to max_iter **do**
2: **for all** grid cells **do**
3: $aux_{i,j,k} = (x0_{i,j,k} + a \times (x_{i-1,j,k} + x_{i,j-1,k} + x_{i,j,k-1} + x_{i+1,j,k} + x_{i,j+1,k} + x_{i,j,k+1}))/iter$
4: **end for**
5: **for all** grid cells **do**
6: $x_{i,j,k} = aux_{i,j,k}$
7: **end for**
8: Enforce Boundary Conditions
9: **end for**

Algorithm 3 CPU based Gauss-Seidel.

Input:
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
a: $\frac{kdt}{h^3}$ (see Eq. 3).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3).
max_iter: number of times to iterate.
Output:
x: 1D array with the grid new interpolated values.

1: **for** $iteration = 0$ to max_iter **do**
2: **for all** grid cells **do**
3: $x_{i,j,k} = (x0_{i,j,k} + a \times (x_{i-1,j,k} + x_{i,j-1,k} + x_{i,j,k-1} + x_{i+1,j,k} + x_{i,j+1,k} + x_{i,j,k+1}))/iter$;
4: **end for**
5: Enforce Boundary Conditions
6: **end for**

In the GPU, for Jacobi and the Gauss-Seidel, we will have a call to a kernel (the kernels are Algorithms 4 and 5). A kernel will have two parameters: $grid$ stands for the number of blocks in X and Y axis, and $threads$ stands for the number of threads per block. The dimensions of the block are given with $BLOCK\_DIM\_X$ and $BLOCK\_DIM\_Y$. Each block treats all grid slices in z direction for the threads in x and y.

```
dim3 threads(BLOCK_DIM_X,BLOCK_DIM_Y);
dim3 grid(NX/BLOCK_DIM_Z,NY/BLOCK_DIM_Y);

// Jacobi kernel call
_jcb<<<grid,threads>>>(x,x0,a,iter,max\_iter);
CUT_CHECK_ERROR("Kernel execution failed");
cudaThreadSynchronize();

// or

// Gauss-Seidel red black kernel call
_gs_rb<<<grid,threads >>>(x,x0,a,iter,max\_iter);
CUT_CHECK_ERROR("Kernel execution failed");
cudaThreadSynchronize();
```

In the GPU implementations of Jacobi and Gauss-Seidel red black algorithms, the values of i and j (cell coordinates) will be obtained with the blocks, threads, and grid information (lines 1 to 2 in Algorithms 4 and 5). The Gauss-Seidel solver is a sequential algorithm since it requires previous values to be calculated. The GPU-based version of Gauss-Seidel has two interleaved passes, first it updates the red cells (line 7 in 5) and then the black cells (line 11 in Algorithm 5), according to the pattern shown in Fig. 5.

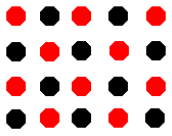

Figure 5. Gauss-Seidel red black pattern for a 2D grid.

Thus previous values are used as in the CPU-based version. The GPU-based implementation of Gauss-Seidel allows more iterations than the CPU-based implementation. Nevertheless, it also takes two times more iterations to converge to the same values as the CPU-based implementation.

The Jacobi GPU-based version requires to temporarily store each grid cell new value in a device global memory 1D array (aux). After each iteration the values stored in x are replaced by the new values, temporarily stored in axu (line 8 in Algorithm 4).

The GPU-based version of all solvers (J, GS, CG) suffer from global memory latency, which appears during successive runs of the solvers (an issue for real time purposes). However, only the Conjugate Gradient is affected to a level of degrading notoriously the solver performance.

Algorithm 4 Jacobi GPU kernel.

Input:
x: 1D device global memory array with the grid current values.
$x0$: 1D device global memory array with the grid previous values.
aux: auxiliary 1D device global memory array.
a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3), 1 for move (see Eq. 4).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3), 6 for Move (see Eq. 4).
max_iter: number of times to iterate.
Output:
x: new interpolated values of x.

1: $i = threadIdx.x + blockIdx.x \times blockDim.x$
2: $j = threadIdx.y + blockIdx.y \times blockDim.y$
3: **for** $iteration = 0$ to max_iter **do**
4: **for** $k = 0$ to NZ **do**
5: **if** $(i! = 0)\,\&\&\,(i! = NX-1)\,\&\&\,(j! = 0)\,\&\&\,(j! = NY-1)\&\&(k! = 0)\&\&(k! = NZ-1)$ **then**
6: $aux_{i,j,k} = (x0_{i,j,k} + a \times (x_{i-1,j,k} + x_{i+1,j,k} + x_{i,j-1,k} + x_{i,j+1,k} + x_{i,j,k-1} + x_{i,j,k+1}))/iter$
7: $\_\_syncthreads$
8: $x_{i,j,k} = aux_{i,j,k}$
9: **end if**
10: Enforce Boundary Conditions
11: **end for**
12: **end for**

5.2. Conjugate Gradient Solver

The Conjugate Gradient algorithm (see Algorithm 6) consists in a series of calls to functions, in the CPU-based version, or to a kernel call, in the GPU-based implementation.

```
_cg<<<1,NX>>>(r,p,q,x,b,alpha,beta,rho,rho0,rho_old,a↩
    ,iter,max_iter);
CUT_CHECK_ERROR("Kernel execution failed");
```

Before iterating, it is first required (lines 1 to 4 of Algorithm 6) to set the initial values of r and p, and of ρ_0 and ρ. After the initial values are set up we are ready to iterate.

Algorithm 5 Gauss-Seidel red black GPU kernel.

Input:
x: 1D device global memory array with the grid current values.
$x0$: 1D device global memory array with the grid previous values.
a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3), 1 for move (see Eq. 4).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3), 6 for Move (see Eq. 4).
max_iter: number of times to iterate.
Output:
x: new interpolated values of x.

1: $i = threadIdx.x + blockIdx.x \times blockDim.x$
2: $j = threadIdx.y + blockIdx.y \times blockDim.y$
3: **for** $iteration = 0$ to max_iter **do**
4: **for** $k = 0$ to NZ **do**
5: **if** $(i! = 0)\,\&\&\,(i! = NX-1)\,\&\&\,(j! = 0)\,\&\&\,(j! = NY-1)\&\&(k! = 0)\&\&(k! = NZ-1)$ **then**
6: **if** $(i+j)\%2 == 0$ **then**
7: $x_{i,j,k} = (x0_{i,j,k} + a \times (x_{i-1,j,k} + x_{i+1,j,k} + x_{i,j-1,k} + x_{i,j+1,k} + x_{i,j,k-1} + x_{i,j,k+1}))/iter$
8: **end if**
9: $\_\_syncthreads$
10: **if** $(i+j)\%2! = 0$ **then**
11: $x_{i,j,k} = (x0_{i,j,k} + a \times (x_{i-1,j,k} + x_{i+1,j,k} + x_{i,j-1,k} + x_{i,j+1,k} + x_{i,j,k-1} + x_{i,j,k+1}))/iter$
12: **end if**
13: **end if**
14: Enforce Boundary Conditions
15: **end for**
16: **end for**

We will iterate until all iterations are done or the stop criterion is achieved (lines 5 and 6 of Algorithm 6). For each iteration, the first step (line 7 of Algorithm 6) is to update q. After updating q, the next step (lines 8 to 12 of Algorithm 6) is to determine the new distance to travel along p, α. During the update of α, the dot product of p by q must be determined. After updating α, we need to determine the iterated values of x, and the new r residues (lines 9 and 10 of Algorithm 6. Before updating each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors, p (line 13 of Algorithm 6), $\rho_o ld$, ρ and β must be updated (lines 11 to 13 of Algorithm 6). After updating β the new search directions (p values) must be set.

The most intuitive way to migrate the Conjugate Gradient from a sequential to a parallel algorithm, is to perform its steps (i.e. dot products, update of grid positions, etc) by kernels, or using the CUDA BLAS library kernels. However, most of these kernels must be called for a certain number of iterations. Therefore, the successive invocation of kernels will result in timeouts in the simulation. The best solution found was to build up a massive kernel. However, this results in losing much of the CUDA performance gains. The reason is related with the parallel version of dot product, which forces the use of one block,

Algorithm 6 Conjugate Gradient method.

Input:
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
r, p, q: auxiliary 1D arrays.
a: $\frac{kdt}{h^3}$ (see Eq. 3).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3).
max_iter: number of times to iterate.
tol: tolerance after which is safe to state that the values of x converged optimally **Output:**
x 1D array with the grid new interpolated values.

1: $r = b - Ax$
2: $p = r$
3: $\rho = r^T \cdot r$
4: $\rho_0 = \rho$
5: **for** $iteration = 0$ to max_iter **do**
6: **if** $(\rho! = 0)$ and $\rho > tol^2 \times \rho_0$ **then**
7: $q = Ap$
8: $\alpha = \rho/(p^T \cdot q)$
9: $x+ = \alpha \times p$
10: $r- = \alpha \times q$
11: $\rho\_old = \rho$
12: $\rho = r^T \cdot r$
13: $\beta = \rho/\rho\_old$
14: $p = r + \beta \times p$
15: Enforce Boundary Conditions
16: **end if**
17: **end for**

with NX threads in x. Much of the steps of the Conjugate Gradient performance degrades with this restriction. Even worse this version has worst performance than the CPU-based version.

6. SOLVERS PERFORMANCE ANALYSIS

After implementing the solvers, tests of their overall performance were made (see Tables 1 and 2). The solvers, both for GPU and CPU, were tested on a Intel(R) Core(TM)2 Quad CPU Q6600@2.40GHz with 4096MBytes of DDR2 RAM and an NVIDIA GeForce 8800 GT graphics card. The CPU-based version is purely sequential, i.e. it runs in a single core and it is not multi-threaded. The following tables show the total time ('CPU time' and 'GPU time') that each solver takes, for a certain number of iterations ('#Iterations'), and a specific grid size ('Grid Size'). Each solver is invoked a number of times in a single step of the stable fluids method (5 for 2D and 6 for 3D). For 2D, we tested each solver using 10 iterations, for all grid sizes. In 3D, we used 4 iterations instead for each solver. In 2D 10 iterations suffice, while we need a minimum number of 4 iterations in 3D, to ensure some convergence of the results. More accurate converging values result in better visual quality. A total time superior to 33ms does not guarantee real-time performance, i.e. no frame rate greater than 30 frames per second is achieved. The time values were obtained from the average time of 10 tests for each solver implementation, independently of the grid size.

	Grid Size	# Iterations	CPU Time (ms)	GPU Time (ms)
J	32^2	10	$0,0$	$0,290219$
GS		10	$0,0$	$0,290032$
CG		10	$0,0$	$0,567346$
J	64^2	10	$2,0$	$0,290900$
GS		10	$3,0$	$0,295285$
CG		10	$3,0$	$0,571604$
J	128^2	10	$8,0$	$0,293459$
GS		10	$15,0$	$0,289818$
CG		10	$13,0$	$0,573089$
J	256^2	10	$35,0$	$0,296003$
GS		10	$60,0$	$0,289882$
CG		10	$56,0$	$0,579169$
J	512^2	10	$298,0$	$0,306887$
GS		10	$245,0$	$0,308024$
CG		10	$350,0$	$0,580205$

Table 1. Performance of 2D solvers for CPU and GPU, for distinct grid sizes.

	Grid Size	# Iterations	CPU Time (ms)	GPU Time (ms)
J	32^3	4	$10,0$	$0,408663$
GS		4	$15,0$	$0,343595$
CG		4	$15,0$	$0,665508$
J	64^3	4	$170,0$	$0,416022$
GS		4	$141,0$	$0,348235$
CG		4	$239,0$	$0,673512$
J	128^3	4	$1482,0$	$0,424283$
GS		4	$1208,0$	$0,360813$
CG		4	$2017,0$	$0,68272$

Table 2. Performance of 3D solvers for CPU and GPU, for distinct grid sizes.

From the previous tables we can draw some conclusions. When comparing the columns 'CPU time' and 'GPU time' it is clear that in the 2D and 3D versions the GPU-based implementation surpasses the CPU-based implementation. However, the previous tables only present the processing times, not including timeouts. When running the simulation, timeouts appear from device global memory latency in the GPU-based version. These timeouts are hidden (i.e. they exist but they do not significantly degrade the solver overall performance) in the Jacobi and Gauss-Seidel GPU implementations. Unfortunately, the Conjugate Gradient timeouts are so severe that the losses overcome the gains in time.

The Conjugate Gradient method converges faster than Jacobi and Gauss-Seidel in spite of involving more computa-

tions during each iteration, but this is not visible for a small number of iterations. Therefore, CPU-based implementation of stable fluids using the Conjugate Gradient solver is inadequate for real-time purposes when the grid size is over 128^2.

Except for the 32^2 grid, the GS and J solvers have significant gains on GPU. Comparing 'CPU time' and 'GPU time' for J and GS solvers it becomes clear that the GPU-based versions are faster. Besides, we can fit more solver iterations per second, using the GPU-based implementation. Unlike the CPU-based version, the GPU-based versions of J and GS solvers enable the usage of a 128^3 grid in real-time. Thus, the observation of the '#Iterations' and 'GPU time' columns leads us to conclude that GS is the best choice for 2D and 3D grid sizes. In the CPU-based versions, the best choice in 2D is the J solver, except for the 512^2 grid where GS is the best choice. In 3D, the CPU-based version of GS is a better choice for grid sizes superior to 32^3.

Another important consideration has to do with the time complexity of both CPU- and GPU-based implementations of stable fluids. Looking at Tables 1 and 2, we easily observe that the GPU-based solvers have constant complexity approximately. On the other hand, CPU-based solvers have quadratic complexity for small grids, but tend to cubic complexity (i.e. the worst case) for larger grids. However, computing the time complexity more accurately would require more exhaustive experiments, as well as a theoretical analysis.

Figs. 6 to 8 show a 128^2 fluid simulation with internal and moving boundaries (red dots). Rendering was done using OpenGL Vertex Buffer Objects. The CPU version is the one here shown. The frame rate includes the rendering time.

7. CONCLUSIONS AND FUTURE WORK

This paper has described CUDA-based implementations of Jacobi, Gauss-Seidel, and Conjugate Gradient solvers for 3D stable fluids on GPU. These solvers have been then compared to each other, including their CPU-based implementations. The most important result from this comparative study is that the GPU-based implementations have constant time complexity, which allows to have a more accurate control in real-time applications.

The 3D stable fluids method has significant memory requirements and time restrictions to solve the Navier-Stokes equations at each time step. It remains to prove that other alternatives (not addressed in this paper) to 3D fluid simulations such as Shallow Water Equations [Miklós 09], the Lattice Boltzmann Method [TJ08], the Smoothed Particle Hydrodynamics [Schlatter 99], or procedural methods [Jeschke 03] are better choices. We hope to explore other emerging solvers for sparse linear systems in a near future. In particular, we need a solver with a better convergence rate than relaxation techniques (J and GS), and with no significant extra computational effort such as the CG.

Figure 6. A CPU version of [Stam 03] fluid simulator with internal and moving boundaries (red dots), using the J solver.

References

[Amorim 08] Ronan Amorim, Gundolf Haase, Manfred Liebmann, and Rodrigo Weber. Comparing CUDA and OpenGL implementations for a Jacobi iteration. Technical Report 025, University of Graz, SFB, Dec. 2008.

[Ash 05] Michael Ash. Simulation and visualization of a 3d fluid. Master's thesis, Université d'Orléans, France, Sep. 2005.

[Bell 08] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[Bolz 03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[Bongart 07] Robert Bongart. Efficient simulation of fluid dynamics in a 3d game engine.

Figure 7. A CPU version of [Stam 03] fluid simulator with internal and moving boundaries (red dots), using the GS solver.

Figure 8. A CPU version of [Stam 03] fluid simulator with internal and moving boundaries (red dots), using the CG solver.

Master's thesis, KTH Computer Science and Communication, Stockholm Sweden, 2007.

[Bridson 07] Robert Bridson and Matthias F. Muller. Fluid simulation: SIGGRAPH 2007 course notes. In *ACM SIGGRAPH 2007 Course Notes (SIGGRAPH '07)*, pages 1–81, New York, NY, USA, 2007. ACM Press.

[Carlson 04] Mark Thomas Carlson. *Rigid, melting, and flowing fluid*. PhD thesis, Atlanta, GA, USA, Jul. 2004.

[Goodnight 07] Nolan Goodnight. CUDA/OpenGL fluid simulation. `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#postProcessGL`, 2007.

[Harris 05] Mark J. Harris. Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Course Notes (SIGGRAPH '05)*, number 220, New York, NY, USA, 2005. ACM Press.

[Jeschke 03] Stefan Jeschke, Hermann Birkholz, and Heidrun Schumann. A procedural model for interactive animation of breaking ocean waves. In *The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003 (WSCG '2003)*, 2003.

[Kass 90] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'90)*, pages 49–57. ACM Press, 1990.

[Keenan 07] Crane Keenan, Llamas Ignacio, and Tariq Sarah. Real-time simulation and rendering of 3d fluids. In Nguyen Hubert, editor, *GPU Gems 3*, chapter 30, pages 633–675. Addison Wesley Professional, Aug. 2007.

[Kim 08] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games (SI3D '08)*, pages 99–106, 2008.

[Kruger 03] Jens Kruger and Rudiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2003 Papers (SIGGRAPH '03)*, pages 908–916, New York, NY, USA, 2003. ACM Press.

[Miklós 09] Bálint Miklós. Real time fluid simulation using height fields semester thesis. `http:\www.balintmiklos.com/layered_water.pdf`, 2009.

[Nickolls 08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[Noe 04] Karsten Noe. Implementing rapid, stable fluid dynamics on the GPU. `http://projects.n-o-e.dk/?page=show&name=GPU%20water%20simulation`, 2004.

[NVIDIA 08a] NVIDIA. CUDA programming guide 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`, Jul. 2008.

[NVIDIA 08b] NVIDIA. CUDA reference manual 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf`, Jun. 2008.

[Owens 08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–89, 2008.

[Schlatter 99] Brian Schlatter. A pedagogical tool using smoothed particle hydrodynamics to model fluid flow past a system of cylinders. Technical report, Oregon State University, 1999.

[Shewchuk 94] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, Aug. 1994.

[Stam 99] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, pages 121–128, New York, NY, USA, Aug. 1999. ACM Press.

[Stam 01] Jos Stam. A simple fluid solver based on the FFT. *J. Graph. Tools*, 6(2):43–52, 2001.

[Stam 03] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, Mar. 2003.

[TJ08] Tölke-Jonas. Implementation of a Lattice-Boltzmann kernel using the compute unified device architecture developed by NVIDIA. *Computing and Visualization in Science*, Feb 2008.

[Wiggers 07] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler, and G.J.M. Smit. Implementing the conjugate gradient algorithm on multi-core systems. page 14, Nov. 2007.