Introduction to Fluid Simulation

Kristofer Schlachter*
New York University

Abstract

This is a short tutorial on Computational Fluid Dynamics, which is a challenging subject to learn. I will introduce all of the topics necessary to understand and build a fluid solver. Very little previous knowledge in the subject will be assumed and explanations will be given with the goal of being able to implement a solver on the CPU and GPU.

Keywords: CFD, PDE Solvers, stable solvers, Navier-Stokes, Euler Equations, Fluids, Semi-Lagrangian, OpenCL, GPU

Introduction

It is challenging to start with an example like Jos Stam's solver that he describes [Stam 2003] and [Stam 1999] and extend it. The math and notation is difficult especially for someone who has not taken vector calculus or differential equations. Recently there have been papers and a book that makes this task a little easier; In this project I used Robert Bridson's book, [Bridson 2009] which is in turn based upon the notes for a Siggraph course on Fluid Simulation[Bridson 2007]. I also used Ronald Fedkiw's paper [R. Fedkiw 2001]. David Cline et al. paper [David Cline] was also quite useful.

1 What does Semi-Lagrangian mean?

There a couple of common ways to approach simulating fluids, and among these they basically fall into two camps. The Lagrangian point of view treats the world like a particle system where particles have properties which are tracked as the particle moves. The other viewpoint is the Eulerian viewpoint where you have fixed points in space, usually placed in a grid layout, where you measure things as they go past. If you think of it as measuring the weather, the Lagrangian way would be using weather balloons floating with the wind. The Eulerian way would be to place sensors on the ground measuring the weather over time. The simulator described in this report falls somewhere in between. It uses a grid to store and track fluid properties but it uses virtual particles to help compute where things are going. This approach is categorized as Semi-Larangian, because it uses virtual particles to handle the advection.

2 Incompressible Navier Stokes

The governing equations of fluid flow that we will start with are called the incompressible Navier Stokes equations:

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

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

*e-mail:ks228@cs.nyu.edu

The equation is broken down as follows:

$\frac{\partial u}{\partial t}$	**The derivative of velocity with respect to time**. Calculated at each grid point each time step.
$-(u \cdot \nabla)u$	**The convection term**. This is the self advection term where the velocity field advances along itself. In the code we will use the backward particle trace for this term.
$-\frac{1}{\rho}\nabla p$	**The pressure term**. ρ is the density of the fluid and p is the pressure. p is whatever it takes to make the velocity field divergence free. The simulator will solve for a pressure that makes our fluid incompressible at each time step.
$v\nabla^2 u$	**The viscosity term**. The Euler equations that we are going to use drop this term. (3)
F	**External force**. Any external forces including gravity.

3 Incompressible Euler Equations

If you drop the viscosity term from the incompressible Navier Stokes equations we get:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u + \frac{1}{\rho}\nabla p = F \tag{4}$$

$$\nabla \cdot u = 0 \tag{5}$$

Such an ideal fluid with no viscosity is called *inviscid*. These are the equations we are going to use.

4 Mathematical Notation

Inspired by another paper [David Cline], here is a quick introduction to some Mathematical operators and what they turn into when you discretize them on a grid.

The partial derivative (∂). In this paper approximated with a central difference:

$$\frac{\partial f(x,y,z)}{\partial y} = \frac{f(x,y+h,z) - f(x,y-h,z)}{h}$$

on a MAC grid:

$$\frac{\partial f(x,y,z)}{\partial y} = f(x,y+1,z) - f(x,y,z)$$

The gradient operator (∇) is a vector of partial derivatives:

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$$

A 3D gradient on a MAC grid will look like:

$$\nabla f(x,y,z) = \begin{pmatrix} f(x+1,y,z) - f(x,y,z), \\ f(x,y+1,z) - f(x,y,z), \\ f(x,y,z+1) - f(x,y,z) \end{pmatrix}$$

The divergence of a vector field ($\nabla \cdot u$) produces the scalar field:

$$\nabla \cdot u = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$

A 3D divergence on a MAC grid will look like:

$$\begin{aligned} \nabla \cdot u(x,y,z) = &(u_x(x+1,y,z) - u_x(x,y,z)) + \\ &(u_y(x,y+1,z) - u_y(x,y,z)) + \\ &(u_z(x,y,z+1) - u_z(x,y,z)) \end{aligned}$$

The Laplacian operator (∇^2) is the dot product of two gradient operators:

$$\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

A 3D Laplacian on a MAC grid will look like:

$$\begin{aligned} \nabla^2 f(x,y,z) = &f(x+1,y,z) + f(x-1,y,z) + \\ &f(x,y+1,z) + f(x,y-1,z) + \\ &f(x,y,z+1) + f(x,y,z-1) - 6f(x,y,z) \end{aligned}$$

To apply ∇^2 to a vector field we apply the operator to each vector component separately:

$$\nabla^2 u(x,y,z) = \begin{pmatrix} \nabla^2 u_x(x,y,z), \\ \nabla^2 u_y(x,y,z), \\ \nabla^2 u_z(x,y,z) \end{pmatrix}$$

5 MAC Grid

In the simulation we store various values in grids (velocity, pressure, fluid concentration, etc) at various points in space. Unfortunately the obvious choice of a uniform grid isn't the best. There is a technique called the marker-and-cell (MAC) method [F H Harlow 1965] for discretizing incompressible flow problems. The main contribution that method made to modern CFD was the introduction of the *staggered grid*. The Mac grid method discretizes space into square or cubic cells with width h. Each cell has a pressure, p, defined at its center. It also has a velocity, $u = (u_x, u_y, u_z)$, but the components of the velocity are placed at the centers of the cell faces (for example u_x on the x-min face and so on as shown in Figures 1 and 2).

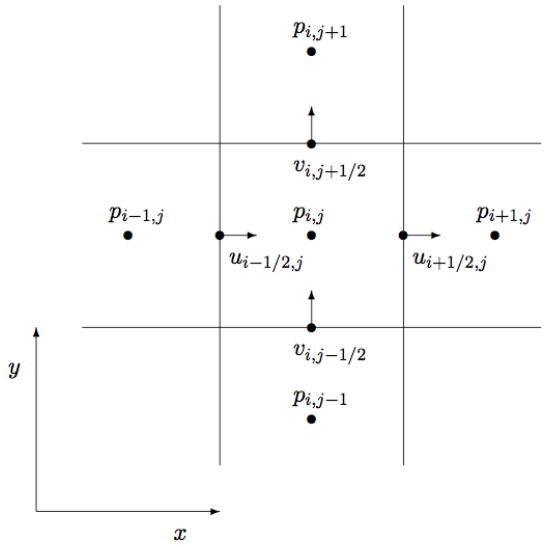


Figure 1: *2D MAC Cell*

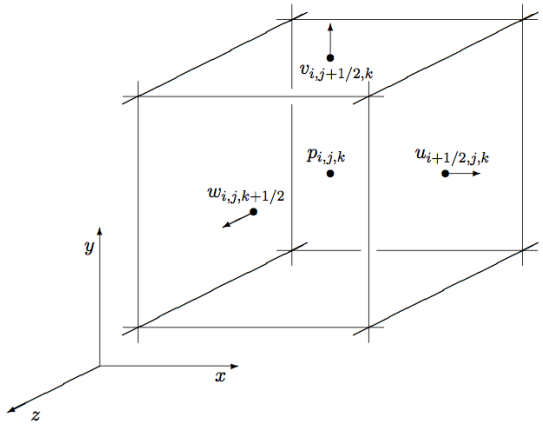


Figure 2: *3D MAC Cell*

The rationale for putting the velocity components on the faces is that we can use more accurate *central differences* for the pressure gradient and for the divergences without the disadvantages of a regular grid. Central differences (6) are $O\left(\Delta x^2\right)$ accurate but they have to be handled carefully:

$$\frac{\partial q}{\partial x} \approx \frac{q_{i+1} - q_{i-1}}{2\Delta x} \tag{6}$$

As described in the literature there are problems with the sampling pattern of (6).[1] It turns out that the MAC grid solves this problem by using a staggered grid, where we don't skip over any indices and it achieves an $O\left(\Delta x^2\right)$ accuracy. You can then estimate the derivative at grid point i in the staggered grid as:

$$\frac{\partial q}{\partial x} \approx \frac{q_{i+1/2} - q_{i-1/2}}{\Delta x} \tag{7}$$

It turns out that in the *Pressure Projection* stage the staggered grid has other useful properties. However there are problems with it. Specifically in order to evaluate velocity anywhere in the grid you

[1] See Bridson's book for an explanation

have to do a three trilinear interpolations, one for each component. In general this is made up for by the increased accuracy of the solver.

6 Algorithm

6.1 Simulation Loop

With the aforementioned MAC structure we can now present the components of the simulator. Starting with an initial divergence-free velocity field u:

- For each simulation step...
  - Choose a timestep Δt.
  - Compute the advection term $-(u \cdot \nabla)u \rightarrow u^A = \texttt{advect}(u^n, \Delta t, u^n)$
  - Add external forces $\rightarrow u^B = u^A + \Delta t F$
  - Solve for pressure so such that $\nabla \cdot u = 0 \rightarrow u^{n+1} = \texttt{project}(\Delta t, u^B)$

6.2 Choosing a Timestep

A great property of using a semi-Lagrangian solver is that it can handle large time steps without instability. While you can choose a Δt that suits your own accuracy, Bridson mentions that you can get strange results if you set Δt too high. So he recommend using a heuristic put forth by [N Foster 2001] where the strategy is to limit the time step so that the furthest particle trajectory is traced five cell widths:

$$\Delta t \leq \frac{5\Delta x}{u_{max}} \tag{8}$$

Bridson recommends calculating u_{max} in the following way:

$$u_{max} = \texttt{max}\left(|u|\right) + \sqrt{\Delta x |F|} \tag{9}$$

Where F are the body forces and Δx is a grid cell width. This is slightly more robust because it also takes into account the forces acting on the fluid, it is always positive and can't cause a divide by zero.

6.3 Advection

Advection (or convection) propagates according to the expression $-(u \cdot \nabla)u$. This term makes the Navier-Stokes and Euler Equations non-linear. Some methods use finite differencing [N Foster 1996] which is only stable when the time step is small enough to satisfy $\Delta t < \Delta h / u_{max}$ (h is the grid cell width) [2]. Bridson uses a technique made popular by Jos Stam [Stam 1999] that results in an "unconditionally" stable solver[3]. This method is referred to as a *backwards particle trace* and has several advantages; most importantly it is unconditionally stable. Stam argues that this can be seen because the new field is simply an interpolated sampling of the previous field and as a result the maximum value of the new field is never larger than the largest value of the previous field. It is also simple to implement. See Figure 3.

[2] Please see the references for the conditions that led to this inequality

[3] Stam based the method upon a technique to solve partial differential equations known as the *method of characteristics*. See appendix A of his paper [Stam 1999]

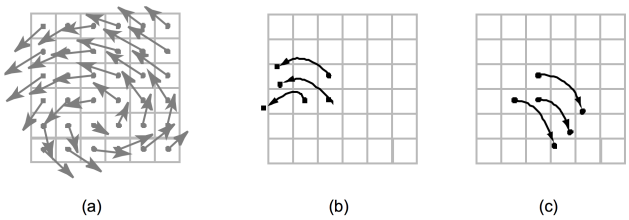


Figure 3: *Basic idea behind the advection step. Instead of moving the cell centers forward in time (b) through the velocity field shown in (a), we look for the particles which end up exactly at the cell centers by tracing backwards in time from the cell centers (c).* [Stam 2003]

6.3.1 Semi-Lagrangian Advection Method

The advection step can be encapsulated by the function:

$$q^{n+1} = \texttt{advect}(u, \Delta t, q^n)$$

where the value q represents the quantity that is being advected. Stam [Stam 1999] explains the method of *backward particle trace*: "At each time step all the fluid properties are moved by the flow field u. To obtain the velocity at point x_G at the new time $t + \Delta t$, we backtrace the point x_G through the flow field u over a time Δt. This traces backward partially along the streamlines of the flow field. The new velocity at the point x_G, had its previous location a time Δt ago." The tricky part of this method is how you calculate the previous location. You can use a *Forward Euler* step, which simply evaluates the velocity at x_G and updates the position of x_G based upon this velocity. You end up arriving at point x_p which is your back traced position:

$$x_p = x_G + \Delta t u(x_G)$$

Because the velocity is not constant this can be inaccurate, and therefore Bridson recommends using at least *Runge-Kutta* order two interpolation (RK2) instead:

$$x_{mid} = x_G - \frac{1}{2}\Delta t u\left(x_G\right)$$

$$x_p = x_G - \Delta t u\left(x_{mid}\right)$$

Again where $u(x_G)$ is evaluating the velocity field u at grid position x_G to get a position half a time step away at x_{mid}. Use x_{mid} to sample the velocity field again $u(x_{mid})$ which is the velocity which will be used for the whole time step.

The above method plus the RK2 interpolation is encapsulated by:

$$q_G^{n+1} = \texttt{interpolate}\left(q^n, x_p\right)$$

Where q_G^{n+1} is the new value of the quantity you are advecting (velocity, density, temperature, etc.) at a grid point G, q^n is the field of the current values for that quantity and x_p is the back traced position using the flow field.

Many people get confused by the backwards particle trace. The listing below shows how it translated into code for my simulator:

```
void advect_velocity_RK2(float delta_time, float *u, float *v, float *w,
  float * u_prev, float * v_prev, float * w_prev)
{
   //The NX scaling factor was found by looking at Stam's code. I can't find
   //a reason in his paper or anywhere else why you would scale this way.
   //But without it the backtrace never makes it out of the source cell.
  float dt = -delta_time*NX;
  int3 dims = {NX,NY,NZ};
  FOR_EACH_FACE
  {

    float3 pos = {i*H,j*H,k*H};
    float3 orig_vel = {u_prev[IX(i,j,k)],v_prev[IX(i,j,k)],w_prev[IX(i,j,k)]};

    //RK2
    // What is u(x)?  value(such as velocity) at x
    //y = x + dt*u(x + (dt/2)*u(x))

    //backtrace based upon current velocity at cell center.
    float halfDT = 0.5*dt;
    float3 halfway_position = {
      pos.x+(halfDT*orig_vel.x),
      pos.y+(halfDT*orig_vel.y),
      pos.z+(halfDT*orig_vel.z)
    };

    float3 halfway_vel;
    halfway_vel.x = get_interpolated_value(u_prev, halfway_position,H,dims);
    halfway_vel.y = get_interpolated_value(v_prev, halfway_position,H,dims);
    halfway_vel.z = get_interpolated_value(w_prev, halfway_position,H,dims);

    float3 backtraced_position;
    backtraced_position.x  = pos.x + dt*halfway_vel.x;
    backtraced_position.y  = pos.y + dt*halfway_vel.y;
    backtraced_position.z  = pos.z + dt*halfway_vel.z;

    //Have to interpolate at new point
    float3 traced_velocity;
    traced_velocity.x = get_interpolated_value(u_prev, backtraced_position,H,
          dims);
    traced_velocity.y = get_interpolated_value(v_prev, backtraced_position,H,
          dims);
    traced_velocity.z = get_interpolated_value(w_prev, backtraced_position,H,
          dims);

    //Has to be set on u
    u[IX(i,j,k)] = traced_velocity.x;
    v[IX(i,j,k)] = traced_velocity.y;
    w[IX(i,j,k)] = traced_velocity.z;
  }
}
```

Listing 1: *Backwards particle trace based advection using RK2*

6.4 Add Body Forces

At this point you would add any external forces, such as gravity or buoyancy to the flow field u. This is also the correct point to add any forces a user might want to add during an interactive simulation.

6.5 Projection / Pressure Solve

The `project`$(\Delta t, u)$ routine does the following:

- Calculate the divergence b (the right-hand side)
- Set the entries of A (see below)
- Solve $Ap = b$ with an appropriate linear solver. (I used Jacobi and Conjugate Gradient methods)
- Using the p, compute the new velocities u^{n+1} by subtracting the pressure-gradient from the velocity field u^n.

6.6 Calculating The Pressure

After advection we have a velocity field that does not satisfy the incompressibility constraint in equation 5 but we still have to apply the pressure. What we need to do is set the pressures in the fluid cells so that the divergence of the entire flow field will be zero. We can't iterate through each cell and satisfy $\nabla \cdot u = 0$, as this would change the divergence of the neighboring cells. What we have to do is solve the constraint for all the cells at once. This gives rise to a large sparse (lots of zero entries) matrix.

We can create a linear equation for the new pressure in every grid cell. We then combine these divergence and pressure equations into matrix form and we end up with a system of equations:

$$Ax = b \tag{10}$$

Remember that divergence looks like:

$$\nabla \cdot u = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$

Using central differences it will look like:

$$\begin{aligned} D_i = (\nabla \cdot u)_{i,j,k} \approx & \frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta h} + \\ & \frac{u_{i,j+\frac{1}{2},k} - u_{i,j-\frac{1}{2},k}}{\Delta h} + \\ & \frac{u_{i,j,k+\frac{1}{2}} - u_{i,j,k-\frac{1}{2}}}{\Delta h} \end{aligned} \tag{11}$$

Every row of A corresponds to one equation for one fluid cell. In this formulation we will setup our matrix such that b is simply our divergence for every fluid cell. When written out our linear system takes the following form:

$$\begin{bmatrix} -O_1 & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & -O_2 & & \vdots \\ \vdots & & -\ddots & C_{n-1,n-1} \\ C_{n,1} & \cdots & C_{n,n-1} & -O_n \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} -D_1 \\ -D_2 \\ \vdots \\ -D_n \end{bmatrix} \tag{12}$$

Where D_i corresponds to the divergences through cell i (11). O_i is the number of non-solid neighbors of cell i[4], and $C_{i,j}$ takes values based upon:

$$C_{i,j} = \begin{cases} 1 & \text{if cell } i \text{ is a neighbor of cell } j \\ 0 & \text{otherwise} \end{cases}$$

A is symmetric and sparse and it is also a very well studied matrix. In 2D it is called the *5 point Laplacian Matrix* and in 3D it is called the *7 Point Laplacian Matrix*. Bridson recommends using the *Modified Incomplete Cholesky Conjugate Gradient Level 0* (CG MIC(0)) algorithm. This method incorporates a pre-conditioner specially chosen for this matrix form to improve conjugate gradient convergence. The implemented simulator includes both jacobi and conjugate gradient solvers. I based my implementation of the conjugate gradient method on the pseudo code in the back of the tutorial written by Shewchuk [Shewchuk 1994]. Another resource I referred to which covered both solvers but not in a form I used was [Gonzalo Amador 2010].

6.7 Pressure Update

6.7.1 Applying the pressure gradient to the velocity field

We calculate the pressure gradient and subtract it from the the velocity in each cell to ensure that the flow field is divergence free. Below are formulas for the 3D case.

$$u^{n+1}_{i+\frac{1}{2},j,k} = u^n_{i+\frac{1}{2},j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \tag{13}$$

[4] in our case for no internal periodic boundaries it is always 4 for 2D or 6 for 3D

$$v^{n+1}_{i,j+\frac{1}{2},k} = v^{n}_{i,j+\frac{1}{2},k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} \quad (14)$$

$$w^{n+1}_{i,j,k+\frac{1}{2}} = w^{n}_{i,j,k+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+} - p_{i,j,k}}{\Delta x} \quad (15)$$

Remember in a MAC grid that pressures are stored in the center so there are no 1/2 indices.[5]

7 Implementation Details and Speed

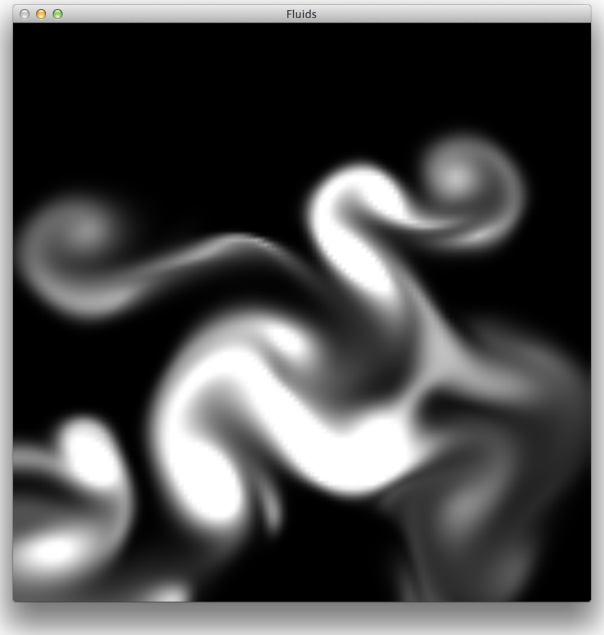


Figure 4: *Screen shot of smoke/density in the interactive simulation of a 128x128x1 grid*

7.1 Implementation Notes

A couple of things that I implemented but didn't discuss in this writeup are the MacCormack method [A Selle 2008] for advection, "Vorticity Confinement" which was discussed in the paper [R. Fedkiw 2001]. Please refer to the papers and to my source code for the details of these methods. Because of time-pressure the grid was eventually implemented as uniform and not staggered. Also a fixed time step of 0.01 was used and h was set to one in order to simplify all of the code. Eventually I will correct these simplifications.

7.2 Scale of the Problem

3D fluids on a grid computations scale at $O(n^3)$. The most expensive part is the projection step where a matrix of size $\left(O\left(N^3\right) \times O\left(N^3\right)\right)$ must be solved[6]. So the choice and implementation of the linear solver is very important for performance. The problem target sizes were a modest 256x256x1 in realtime with

[5]Implementation note: See Bridson Figure 4.1. Instead of looping over velocity locations and updating them with pressure differences, loop over pressure values and update the velocities they affect for greater efficiency.

[6]NX, NY & NZ are the dimensions of the simulation grid

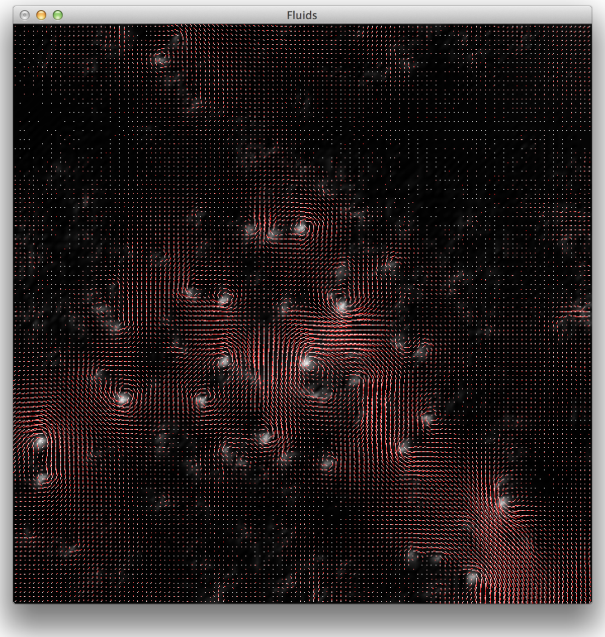


Figure 5: *Screen shot of velocity view in an interactive simulation of a 128x128x1 grid*

$GridSize$	$Kernel$	$Time(ms)$	Throughput
128x128x64	Advection Velocity	4.467	234
128x128x64	Advection Density	2.828	370
128x128x64	Divergence	0.871	1203
128x128x64	Projection Jacobi	15.519	68
128x128x64	Projection CG	14.205	74
128x128x64	Pressure Apply	1.139	920

Table 1: *Throughput is in MegaCells/sec and only one projection is active at once*

aspirations to do 128x128x64 in real time as well. Both are reasonable and were achieved (with a caveat for copy down of buffers for rendering[7]). When all of the buffers are kept on the GPU and don't need to be copied over the bus multiple times, performance was acceptable. Table 1 shows performance for the target grid size of 128x128x64. A 128x128x64 grid has roughly 1 million cells which is 1 MegaCell per second in our measurements. The slowest kernel is the projection which has a throughput of 68 MegaCells per second for that grid size and that translates to less than 15 milliseconds of run time. To run all of the kernels at that size takes 25 milliseconds which corresponds to 40 FPS. Since 30 FPS is considered a minimum for interactivity the performance goal was achieved.

7.3 Other Peoples Work and Code

Jos Stam's Real-time fluid dynamics for games code [Stam 2003] was reviewed. It helped in understanding some problems I had with my advection implementation. In code Listing 1 you can see that I used Stam's scaling factor of the grid size to get the backwards par-

[7]OpenCL allows sharing of buffer data with OpenGL. Using that feature would eliminate the need for the buffer copy to the host for rendering. This is future work and will be checked into the repository sometime in the near future.

ticle trace to work. The most useful resources I found were Bridson's book [Bridson 2009] and a much more concise paper from Ron Fedkiw [R. Fedkiw 2001]. I also found that using the pseudo-code at the end of [Shewchuk 1994] to implement CG was the best way to go.

7.3.1 Parallelization

The simulation has a series of steps which are written as straight C functions and equivalent OpenCL kernels. Porting to OpenCL was trivial as it was mostly copying the C version of the code and adding a couple of boilerplate lines to calculate i, j, k[8] from the global_id and removing the surrounding triple loops. I did not have time to optimize the OpenCL code by using local memory and doing block calculations for all my kernels. The one kernel that was enhanced with local memory copies of data was the apply pressure step. The performance of that kernel is discussed in the section titled "Fell short of expectations" in the Performance Measurements section.

8 Performance Measurements

Note that all of the performance graphs are at the end of the paper. The performance tests were run on a Macbook Pro with an Intel Core i7-3720QM CPU @ 2.60GHz, which has 4 cores and 8 logical threads. The GPU was a GeForce GT 650M. Performance was measured in MegaCells per second. Since each cell takes set number of floating point calculations it seemed like a reasonable way to approximate a GigaFlops without having to actually count the number of floating point operations for an entire time step in the simulation.

8.1 Performance Expectations

I expected the throughput serial code and the OpenCL on the CPU to remain constant regardless of problem size, which turned out to be true. I also expected that on the GPU, throughput would ramp up with larger grid sizes and workgroups, which turned out to be true with diminishing returns at the largest grid sizes. I expected the most time consuming part to be the projection step. Table 1 shows that projection is roughly 15 milliseconds out of an entire simulation time of 25 milliseconds. This confirms that projection calculations are more than half (60%) of the entire simulation. However, buffers being sent and retrieved from the GPU took a long time and any technique that can minimize them has an advantage in a interactive simulation. So while Conjugate Gradient converged in much fewer iterations and is a much better solver in general, the Jacobi iterations were faster in an interactive setting where just getting a valid result was more important than the most accurate. CG had to send and receive two vectors of size $O\left(n^3\right)$ each iteration. With Jacobi no buffers had to be sent or received as the data could stay on the GPU[9]. This caused a big difference in the total time to calculate one time step. However, when OpenCL was targeting the CPU the buffer transfer costs were eliminated and CG was much more efficient as it could converge faster and detect convergence after only a few iterations, while my implementation of Jacobi was always doing a fixed number of iterations since it lacked a residual calculation necessary to detect convergence.

8.1.1 Exceeded expectations

One thing that I was surprised to see was that for most simulation step I didn't max out the performance of the GPU. The throughput grew on the GPU with problem size. See Figures 6, 9 and 10. The kernels that behaved this way were all very simple lookups of 7 values and a few multiplies and adds. The kernels that were more complex hit their maximum throughput or started to see a drop off in gains when the grid sizes grew.

I was very surprised by the performance of the OpenCL code running on the CPU. The auto-vectorization and threading of the code caused a big speedup from the serial version, and was fast enough for real-time. The buffer copies to and from the device were extremely fast, much faster than to the GPU which had to go over PCI express. If you added the time to simulate multiple frames, the buffer copying to the device cut down the performance of the total simulation. This isn't shown in the performance charts because I chose to focus on the computational performance. However, the buffer transfer speed gain can be seen in figure 8 where the performance of the Projection step using a conjugate gradient solver is used. My conjugate gradient solver is mostly doing dot products and matrix-vector multiplies but the results of each one was brought back to the CPU for proper synchronization for further calculation. Figure 8 shows how the OpenCL version using the CPU is actually faster for because the buffer transfers dominated the total time for the simulation step. If the grid sizes went up this advantage would disappear and the GPU version would be faster as the time would be bounded by computations and not data transfer.

8.1.2 Fell short of expectations

The simulation did not benefit from using local memory at the block size I chose or at the simulation grid sizes tested. See Figure 10 for a throughput comparison on a simple kernel that simply applied the pressure gradient to the velocity field. I was a little surprised by this. Looking closer at the kernel and how it loaded values into local memory as well as how it utilized the local memory revealed the cause of this behavior. The local memory only had each entry reused 6 times (7 accesses) and the code that transferred the appropriate global memory data into the local memory involved 5 branches to handle the edge cases. This apparently negated the gains by locally accessing the data in the main calculation part of the kernel.

8.2 Source Code Links and Instructions

To get the code:

- Go to http://github.com/kristofe/OpenCLFluids
- Clone or download the code.
- Switch to branch `code_for_paper`. [10]
- On a Mac run: `make mac`
- On a linux run: `make linux`[11]
- Then run: `./fluidsim`

OSX has all the libraries necessary. I tested on 10.7.5 on a Macbook Pro, Macbook Air and a Mac Pro. Everything ran well and should run on any mac running OS X 10.7.5. I tested the linux build on the

[8] i, j, k represent the indices of the grid cell being computed

[9] The kernel needed to sync all global memory and therefore could not use barriers as they only sync workgroups. So the whole kernel had to finish and be relaunched for each iteration.

[10] This branch was made to preserve the state of the code at the time of the writing of this paper.

[11] On linux make sure you install freeglut-dev with your package manager to compile and link the code.

class VM in virtual box, but I only was able to compile the code. Running it didn't work because it looked like it couldn't run the OpenGL code.

References

A SELLE, R. F. 2008. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 350–371.

BRIDSON, R., 2007. Fluid simulation: Siggraph 2007 course notes. http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf.

BRIDSON, R. 2009. *Fluid Simulation For Computer Graphics*. A.K. Peters.

DAVID CLINE, DAVID CORDON, P. K. E. Fluid flow for the rest of us: Tutorial of the marker and cell method in computer graphics. http://people.sc.fsu.edu/~jburkardt/pdf/fluid_flow_for_the_rest_of_us.pdf.

F H HARLOW, J. E. W. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *The Physics of Fluids 8*, 2182–2189.

GONZALO AMADOR, A. G. 2010. Linear solver for stable fluids: Gpu vs cpu. *International Conference on Information Science and Applications (ICISA)*.

N FOSTER, D. M. 1996. Realistic animation of liquids. *Graphical Models and Image Processing*, 471–483.

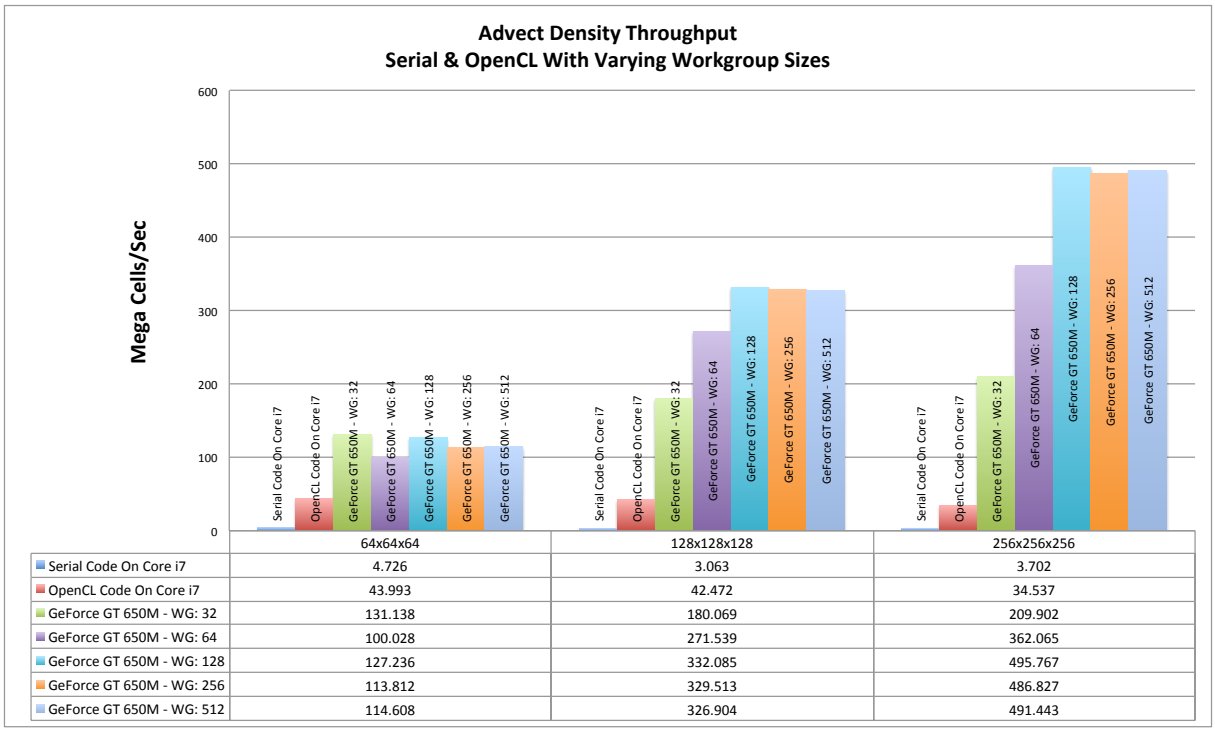
N FOSTER, R. F. 2001. Practical animation of liquids. *Proceedings of the 28th annual conference on Computer Graphics and interactive techniques*, 23–30.

R. FEDKIW, J. STAM, H. W. J. 2001. Visual simulation of smoke. *Proceedings of SIGGRAPH 2001*, 15–22.

SHEWCHUK, J. R., 1994. An introduction to the conjugate gradient method without the agonizing pain. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

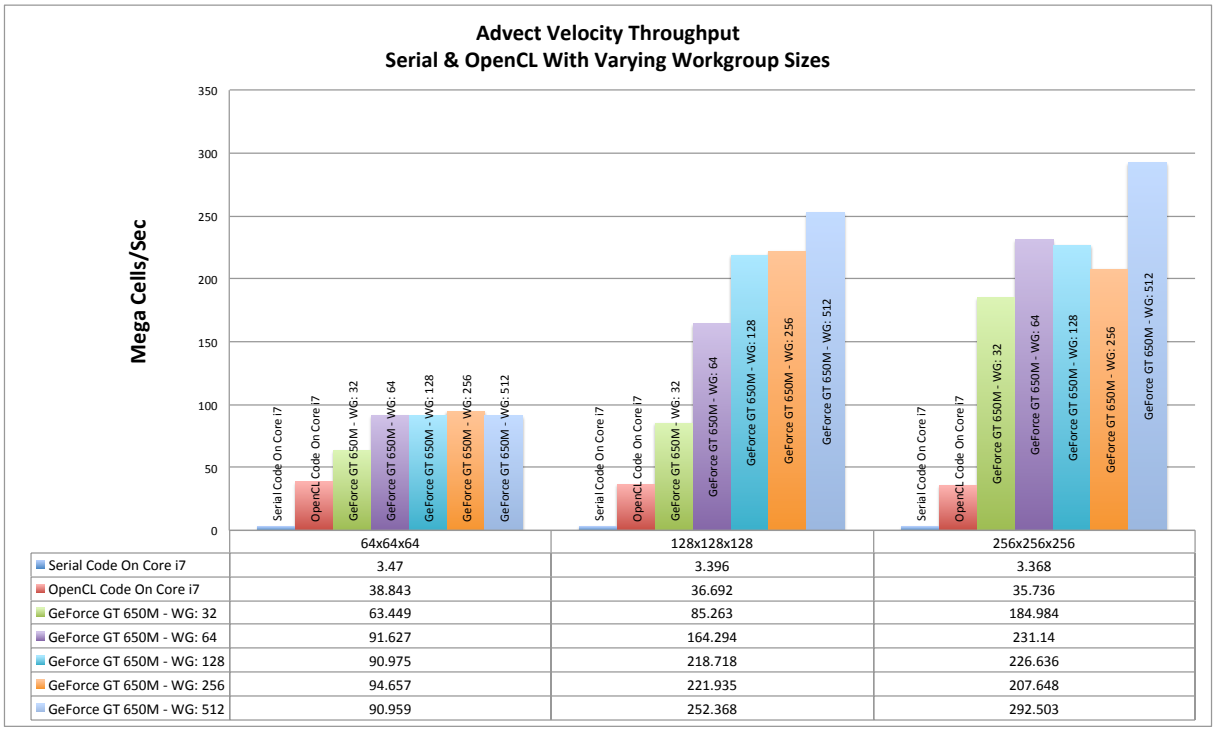
STAM, J. 1999. Stable fluids. *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (Siggraph 2009)* (August), 121–128.

STAM, J. 2003. Real-time fluid dynamics for games. *Proceedings of Game Developer Conference* (March).



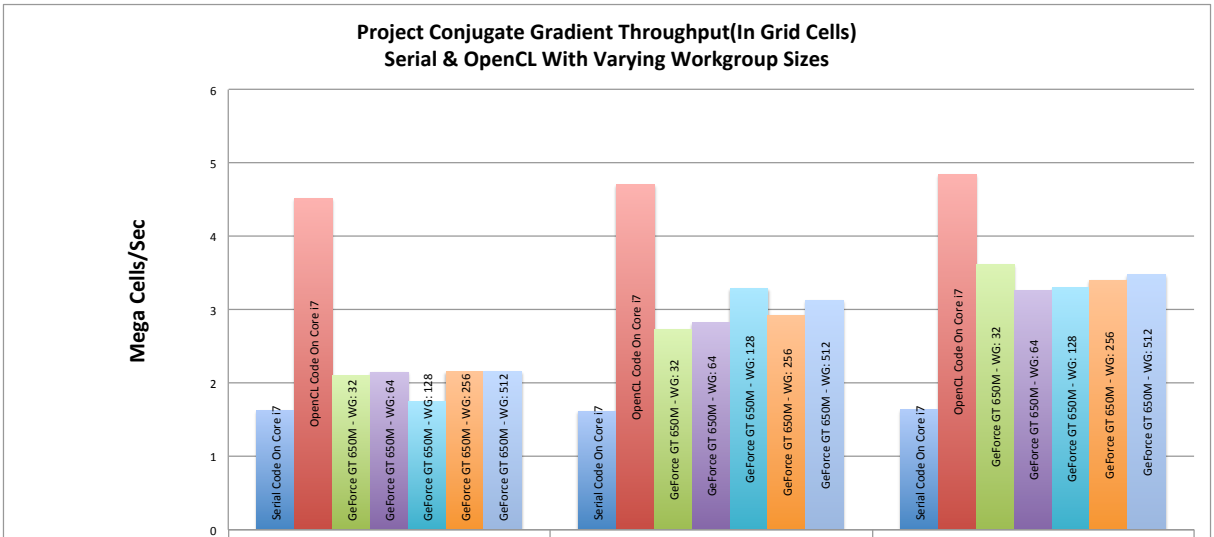
	64x64x64	128x128x128	256x256x256
Serial Code On Core i7	4.726	3.063	3.702
OpenCL Code On Core i7	43.993	42.472	34.537
GeForce GT 650M - WG: 32	131.138	180.069	209.902
GeForce GT 650M - WG: 64	100.028	271.539	362.065
GeForce GT 650M - WG: 128	127.236	332.085	495.767
GeForce GT 650M - WG: 256	113.812	329.513	486.827
GeForce GT 650M - WG: 512	114.608	326.904	491.443

Figure 6: *Throughput of the Advect Density Routine in Serial Code and OpenCL on CPU and GPU*



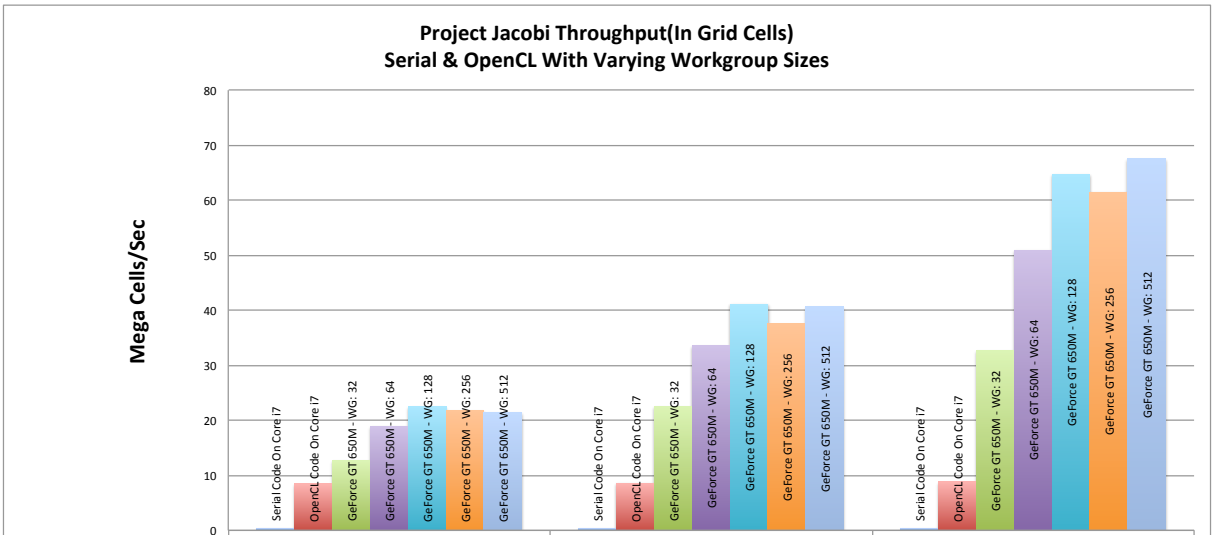
	64x64x64	128x128x128	256x256x256
Serial Code On Core i7	3.47	3.396	3.368
OpenCL Code On Core i7	38.843	36.692	35.736
GeForce GT 650M - WG: 32	63.449	85.263	184.984
GeForce GT 650M - WG: 64	91.627	164.294	231.14
GeForce GT 650M - WG: 128	90.975	218.718	226.636
GeForce GT 650M - WG: 256	94.657	221.935	207.648
GeForce GT 650M - WG: 512	90.959	252.368	292.503

Figure 7: *Throughput of the Advect Velocity Routine in Serial Code and OpenCL on CPU and GPU*



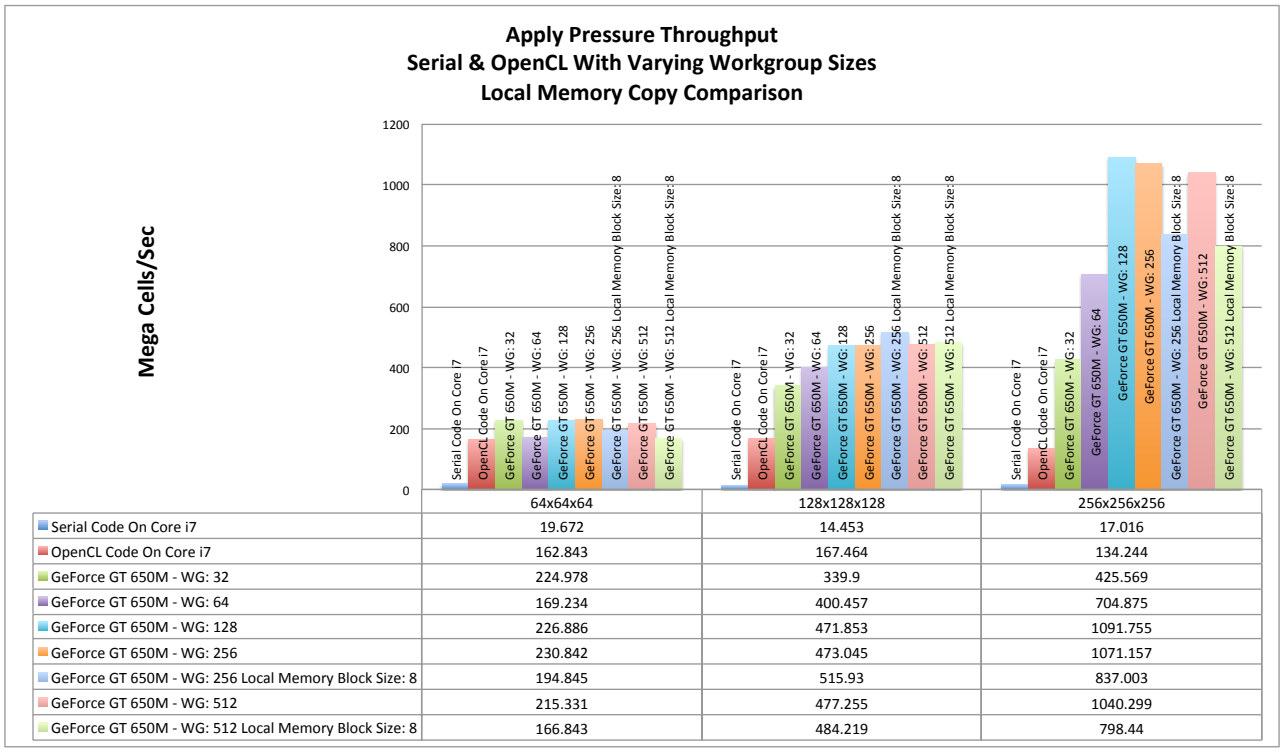
	64x64x64	128x128x128	256x256x256
Serial Code On Core i7	1.619	1.615	1.634
OpenCL Code On Core i7	4.518	4.704	4.844
GeForce GT 650M - WG: 32	2.099	2.724	3.619
GeForce GT 650M - WG: 64	2.15	2.821	3.26
GeForce GT 650M - WG: 128	1.748	3.291	3.301
GeForce GT 650M - WG: 256	2.152	2.924	3.4
GeForce GT 650M - WG: 512	2.155	3.123	3.479

Figure 8: *Throughput of the Projection using Conjugate Gradient Routine in Serial Code and OpenCL on CPU and GPU*



	64x64x64	128x128x128	256x256x256
Serial Code On Core i7	0.429	0.396	0.42
OpenCL Code On Core i7	8.467	8.604	8.843
GeForce GT 650M - WG: 32	12.667	22.55	32.658
GeForce GT 650M - WG: 64	18.9	33.533	50.94
GeForce GT 650M - WG: 128	22.501	41.138	64.751
GeForce GT 650M - WG: 256	21.845	37.572	61.468
GeForce GT 650M - WG: 512	21.383	40.61	67.545

Figure 9: *Throughput of the Projection using Jacobi Iteration Routine in Serial Code and OpenCL on CPU and GPU*



	64x64x64	128x128x128	256x256x256
Serial Code On Core i7	19.672	14.453	17.016
OpenCL Code On Core i7	162.843	167.464	134.244
GeForce GT 650M - WG: 32	224.978	339.9	425.569
GeForce GT 650M - WG: 64	169.234	400.457	704.875
GeForce GT 650M - WG: 128	226.886	471.853	1091.755
GeForce GT 650M - WG: 256	230.842	473.045	1071.157
GeForce GT 650M - WG: 256 Local Memory Block Size: 8	194.845	515.93	837.003
GeForce GT 650M - WG: 512	215.331	477.255	1040.299
GeForce GT 650M - WG: 512 Local Memory Block Size: 8	166.843	484.219	798.44

Figure 10: *Throughput of the Pressure Apply Routine in Serial Code and OpenCL on CPU and GPU. However two scenarios using Local Memory Blocks are shown to compare the performance effect.*