Real-Time CPU Fluid Dynamics

Jesús Martín Berlanga

Scientific Computing
Computer Science Technical College (ETSIINF)
Technical University of Madrid (UPM)

January 19, 2017

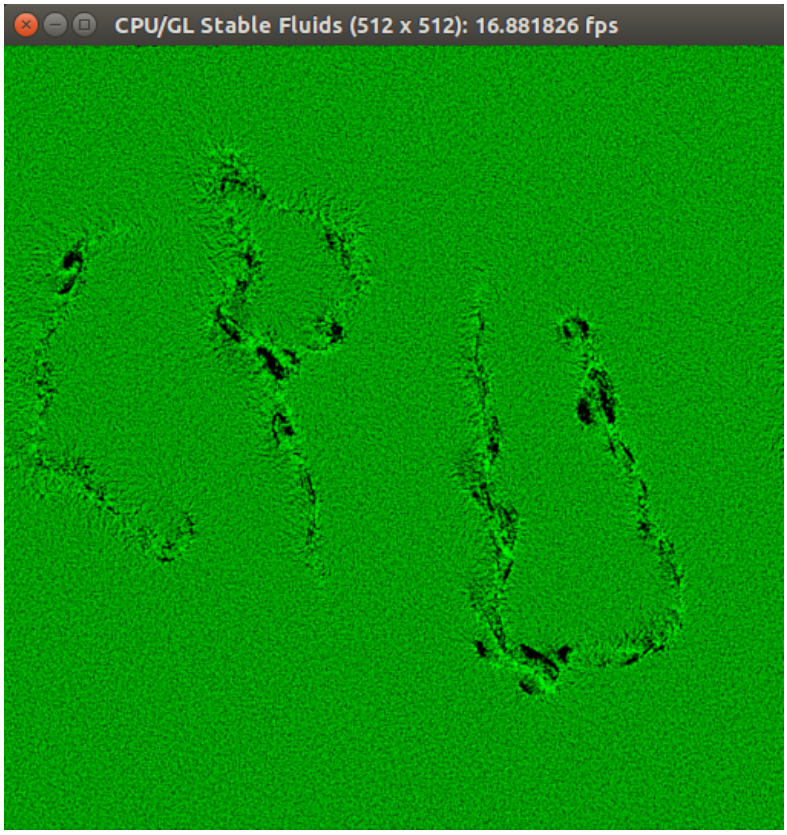

Figure 1: Interactive simulation visualization

1 Preface

Fluids are found in a multitude of different places: In the water of a river, in the cigar's smoke, in the water vapor that creates clouds... There is an increasing need for realistically representing these phenomena in all kinds of graphics applications (e.g. in a video game).

The reader will be introduced to basic fluid concepts and formulae in order to code a *CPU* implementation. This paper is the base for, *Real-Time GPU Fluid Dynamics*, a follow-up where it is further discussed how to parallelize the algorithm here explained for a high performance.

Finally, the resultant *CPU* program has been subjected to some basic optimization procedures. Nonetheless, the emphasis of this paper remains in the understanding of the selected algorithm on account of the application being fully optimized in the *Real-Time GPU Fluid Dynamics* paper that can be found along with the source code and other resources at the `rtfluids.bdevel.org` website.

The development has been made porting the *CUDA/OpenGL Fluid Simulation GPU* demo to a *CPU* version. However, for educational and logical purposes here it is explained as if it would have been developed from scratch.

This document will prove particularly useful to whoever wants to extend his learning from the NVidia documentation on *GPU* fluids since, to date and as far as the author is concerned, *CUDA/OpenGL Fluid Simulation*[2] is quite vague at explaining how the code works; and *GPU Gems: Chapter 39. Fast Fluid Dynamics Simulation on the GPU*[1] is outdated, not taking the advantages of *CUDA* over *Cg* (*C for Graphics* - shading language) for general purpose software.

Contents

2 Introduction

Here is explained how to implement a *2D CPU* fluid simulation and interactively display the results. In order to implement a fluid simulation, first, we need a mathematical foundation of the equations involved in fluids. These equations allow us to elaborate an algorithm for computing, at a given time step, the resultant fluid representation. After we have studied the steps involved in the simulation we will proceed to explain the CPU implementation and how to ease the debugging process. Lastly, we will compile, profile and attempt to optimize the application only to later present the performance results.

The physically-based simulation presented here uses particles that are carried by the fluid velocity field so it can be real-time visualized.

3 Mathematical Background

3.1 Fluid Representation

A fluid can be represented with a Cartesian grid *(fig. 2)* of velocities which indicate how the fluid will move itself as well as other objects inside the fluid.

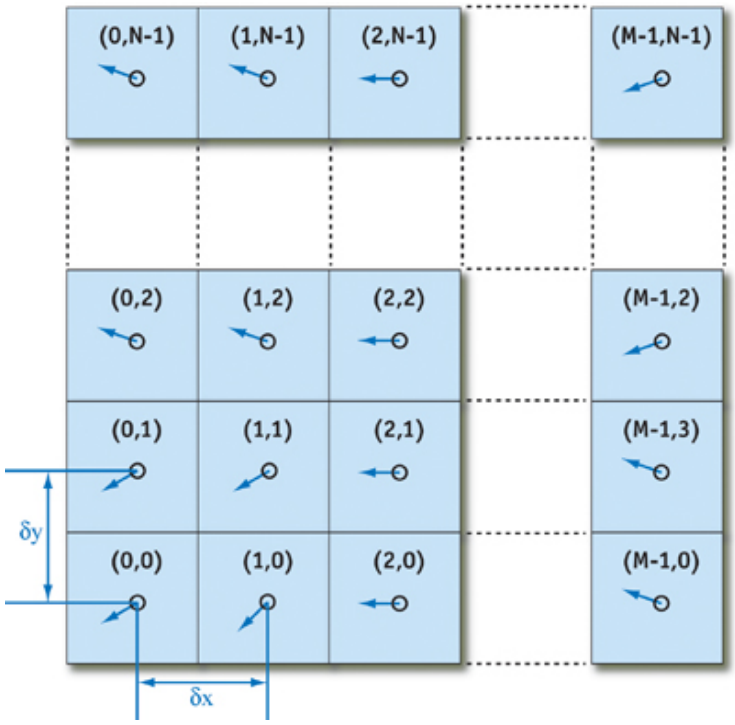

Figure 2: The Fluid Velocity Grid (from *GPU Gems*)

3.2 The Navier-Strokes Equations for Incompressible Flow

In order to simplify the fluid simulation to suit our needs and scope, we assume that the fluid is incompressible and homogeneous. This effectively means that its density is constant in time and space (in any case, this simplification does not decrease the mathematical relevance for, to give an example, water and air simulations).

These simplifications are reflected in the Navier-Strokes equations for incompressible flow:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \bigtriangledown)\vec{u} - \frac{1}{\rho} \bigtriangledown p + \nu \bigtriangledown^2 \vec{u} + \vec{F},$$

$$\bigtriangledown \cdot \vec{u} = 0$$

Where $\vec{u}(\vec{x}, t)$ is the velocity field that represents the fluid in the spatial coordinates of the grid (in this case a two-dimensional grid) $\vec{x} = (x, y)$. $p = (\vec{x}, t)$ is a scalar pressure field, ν is the kinematic viscosity, and $\vec{F}$ represents possible externals forces that apply to the fluid.

The equations can be solved and decomposed in order to obtain a description of the steps involved in the simulation and their formulas. These complex realizations are not described here but are well explained in *GPU Gems: Chapter 39. Fast Fluid Dynamics Simulation on the GPU*[1] and *Stable Fluids*[3].

3.3 Simulation Steps

The simulation orderly consist of:

1. Fluid advection
2. Fluid diffusion
3. Fluid projection
4. Objects (including particles) advection

3.3.1 Advection

The fluid moves itself with his own velocity field (known as self-advection, fluid advection or velocity advection) as well as other objects.

Self-advection is described with the following formula:

$$\vec{u}(\vec{x}, t + 1) = u(-\vec{u}(\vec{x}, t) \cdot dt, t)$$

We trace velocity back (fig. 3) in order to step calculating the next value of velocity. We have to do this for every $\vec{u}$ in our grid.

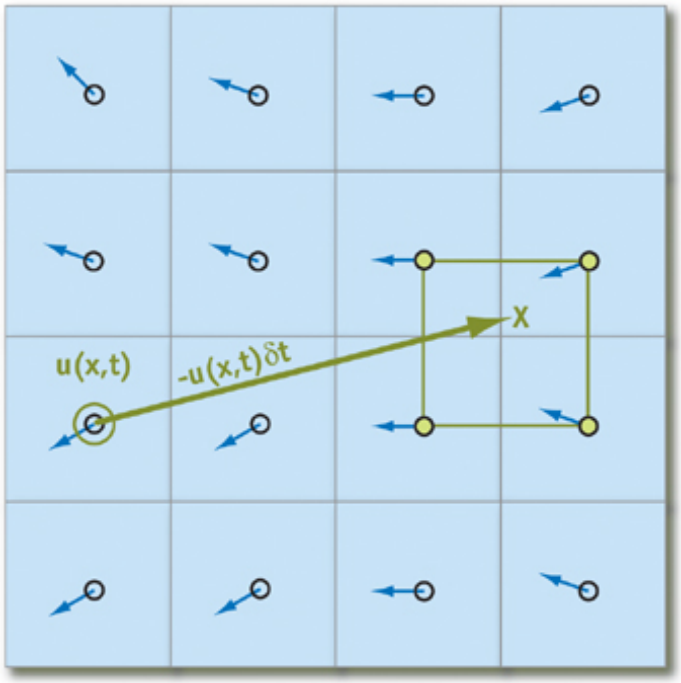

Figure 3: Fluid Advection (from *GPU Gems*)

Object advection can be simply performed by moving the object at position $\vec{p}$, in function of the velocity where the object is located at the current time $\vec{v}(\vec{p}(t))$:

$$\vec{p}(t+1) = \vec{p} + dt * \vec{v}(\vec{p}(t))$$

This equation can be used to move a set of rendered particles that represent the fluid to be visualized by an end-user of the application.

3.3.2 Diffusion

The viscosity ν of a fluid measures how forces diffuse across the velocity field over time. In the diffusion step, we calculate the dissipation of the fluid's velocity applying an operator to our grid. This operator is a simple convolution (this is the pointwise product of two functions in the frequency spatial domain - convolution theorem).

Diffusion takes the following form in the frequency domain:

$$\vec{u}(k, t+1) = \vec{u}(k, t)/(1 + \nu * dt * k^2)$$

Where $\vec{k}$ is the wavenumber.

3.3.3 Projection

We need to make sure our liquid is incompressible in order to our Navier-Strokes Equations for Incompressible Flow remain valid. After applying our diffusion equation our fluid no longer meets the mass conserving requirement for incompressible fluids. For this reason, we have to include an additional projection step to force the velocity field to be non-divergent.

The following projection equation works in the frequency domain:

$$\vec{u}(k, t+1) = \vec{u}(k, t) - \frac{1}{k^2}(k \cdot (\vec{u}(k, t))k$$

3.4 Initial Conditions

A still fluid will be assumed. Thus the simulation starts with zero velocity in the whole grid.

3.5 Boundary Conditions

The boundary states how the fluid will behave in the edges of our domain. We have to set boundary conditions because we have a limited domain, in this case, is the size of the grid.

There are various possibilities in terms of boundary conditions, some of the solutions are rather complex e.g no-slip boundary conditions where the liquid is expected to bounce when colliding with the edges of the domain. A periodic boundary condition (PBC, fig. 4) simulates a spatially infinite system on the different axes - for simplicity, this approach is taken.

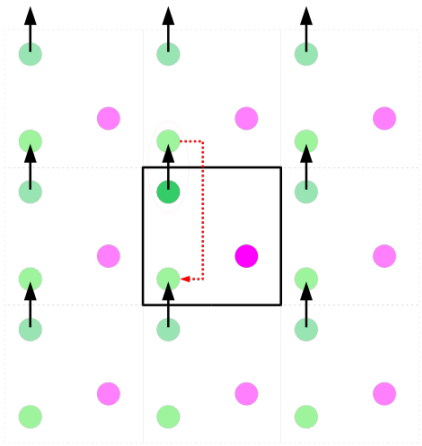

Figure 4: Periodic boundary condition (from *Wikipedia*)

In a PBC, when an object cross one boundary sides it appears on the opposite side with the same velocity. This applies to the fluid particles.

4 Implementation

The implementation is written in *C++*, we have used the *fftw* library to Fourier transform the time domain into the frequency domain and vice versa. For interpolation, needed at the velocity advection step, we have used a source code obtainable at the website *helloacm.com*. For visualization, *GLUT* and *OpenGL* is used.

The solver works by progressively incrementing the time, t, by a fixed value (delta time) in each iteration. In one iteration, all steps (velocity advection, diffusion, projection and particle advection) are executed in order.

4.1 Problem Conditions and Storage of the Fluid's State

The default problems conditions are established in the *defines.c* file, the conditions can be changed at runtime (must be changed before actually starting the simulation). One must call *updateVariables()* after changing these variables to make sure dependable variables are updated with the changes. With this file, we can configure the viscosity constant (VIS) and the delta time (DT) for the iterative solver.

For easiness, a square-shaped domain will be used; the square size is defined in the DIM variable. If we know DIM then the domain size, DS, is $DIM \cdot DIM$.

```
int   DIM   = 512;
...
float DT    = 0.09f;
float VIS   = 0.0025f;
...
void updateVariables() {
     DS    = (DIM*DIM);
     ...
}
```

4.1.1 Real Velocity Vectors

Once we know the total domain size we can allocate memory for our grid (named *hvfield*) of 2D velocity vectors (the *cData* struct represent a vector with two floats: x and y). We make sure we all velocity vectors are set to zero with *memset* to comply with our initial conditions:

```
cData *hvfield = NULL;
...
hvfield = (cData *)malloc(sizeof(cData) * DS);
memset(hvfield, 0, sizeof(cData) * DS);
```

4.1.2 Complex Velocity Vectors

We will also have to allocate memory for complex numbers since we will use the frequency domain in the diffusion and projection steps. We will use the *fftw3* library to transform back and forth between the real numbers stored in *hvfield* and our complex numbers.

Two things must be realized. First, x and y parts of the vector are transformed independently thus they need to be stored separately. Second, complex numbers need twice the memory real numbers require to be designated. Therefore, enough memory will be allocated from the beginning for the complex numbers. Its also to be noted that if we want to transform in-place (without needing to allocate additional memory for the output), we need to allocate extra padding memory (as it's represented and formulated in fig. 5).

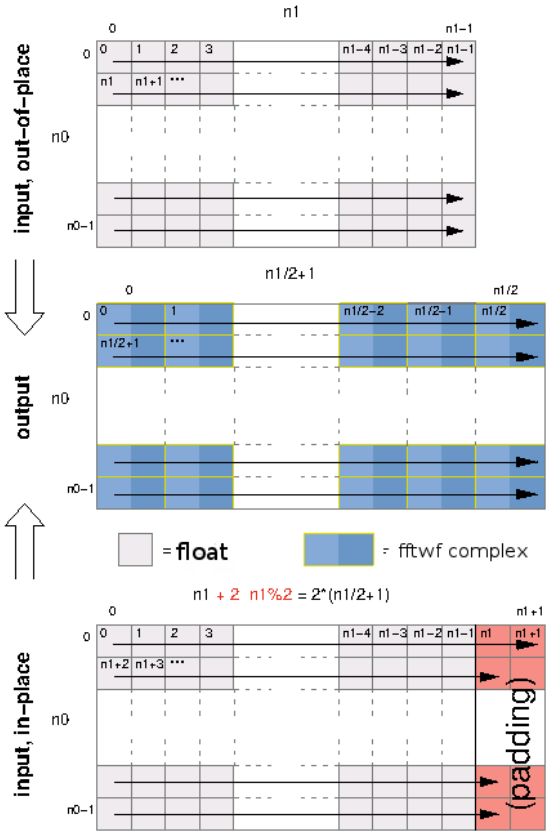

Figure 5: Input and output arrays for Multi-Dimensionals DFTs (discrete Fourier transforms) of Real Data (from *fftw.org*)

Note that a *fftwf complex* is nothing more than float array with enough space, two floats, for the real and imaginary parts of the complex number. As a result, we can use the *fftwf_complex* or the *cData* struct types indistinctly.

The code for the padding domain size (*PDS*) and related variables, as well as the code for assigning dynamic memory for the x (*vxfield*) and y (*vyfield*) complex components of the field can be found below:

```
...
int CPADW = (DIM/2+1);   // Padded width for real->complex in-place FFT
int RPADW = (2*(DIM/2+1)); // Padded width for real->complex in-place FFT
int PDS  = (DIM*CPADW);  // Padded total domain size
... // RealPADW is 2*ComplexPADW since a complex number need two floats
void updateVariables() {
    ...
   CPADW = (DIM/2+1);
   RPADW = (2*(DIM/2+1));
   PDS  = (DIM*CPADW);
    ...
}
...
static cData *vxfield = NULL;
static cData *vyfield = NULL;
...
vxfield = (cData*) malloc(sizeof(cData) * PDS);
vyfield = (cData*) malloc(sizeof(cData) * PDS);
```

4.1.3 Particles

Finally, the particles' array, which will be later rendered, need to be allocated too. We will render as many particles as the domain size, but note that this number could be changed as particles are independent from the velocity field. The particles' position will be randomly generated. The random method output will not vary between platforms, this is particularly useful for debugging and testing.

```
static cData *particles = NULL; // particle positions in host memory
...
particles = (cData *)malloc(sizeof(cData) * DS);
memset(particles, 0, sizeof(cData) * DS);
initParticles(particles, DIM, DIM);
```

4.2 Interaction and Visualization

The application is able to display how the fluid particles move in the canvas. The user can interact with the fluid adding externals forces when clicking and dragging the mouse.

4.2.1 Vertex Buffer

OpenGL is used to display the resultant particles' position after a simulation iteration has taken place. OpenGL can render the points representing the particles after their positions are copied to an OpenGL vertex buffer object (*vbo*). OpenGL can work with several vertex buffer objects so we need to indicate him in which one we want to operate with *glBindBuffer(GLenum target, GLuint buffer_id)*. After this, we can do different kinds of operations with the buffer. Lastly, we need to indicate OpenGL that we are no longer operating with that buffer with a call to *glBindBuffer* passing a 0 in the *buffer_id* parameter.

Here is how to initialize the OpenGL buffer (in this case, an array buffer). A new *buffer_id* is assigned (and stored in the *vbo* variable) to the newly generated buffer. The previously randomly generated particles' positions are copied to the buffer with *glBufferData*.

```
GLuint vbo = 0;
...
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo); // Select buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(cData) * DS,particles,
    GL_DYNAMIC_DRAW_ARB);
glBindBuffer(GL_ARRAY_BUFFER, 0); // Unselect buffer
```

However, it is not enough to copy the initialized particles to the buffer array since the *particles* array will be constantly modified as new simulation iterations take place and the positions, of the particles, are calculated. We need to update the vertex buffer as new changes take place. The whole particles array could just be copied again after particles are updated, but a different approach is taken: The host particles array will be mapped to the vertex buffer so the changes directly made through the mapped pointer will be automatically reflected in the vertex buffer.

```
// read/write mapping
void mapGlBuffer() {
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferMap = (cData*) glMapBufferRange(GL_ARRAY_BUFFER, 0,
        sizeof(cData) * DS, GL_MAP_WRITE_BIT | GL_MAP_READ_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

void unmapGLBuffer() {
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

This way, the mapped pointer can be passed to the particle advection function which will directly change the vertex buffer with the new positions for the

particles. We have to be careful, though: If a buffer is used for feedback at the same time it's read/written for another purpose it will cause an error, this remain true if the buffer is still mapped when calling *glDrawArrays* which will render the particles. Consequently, the *unmapGLBuffer*() has to be included to unmap the buffer when it's not longer needed.

4.2.2 OpenGL initialization

OpenGL is initialized with the *initGL* function where an RGB window is created with dimensions equal to the fluid grid dimensions (although the window dimensions are forced to be at least $512 \cdot 512$). The window can be resized, this functionality is provided by the *reshape* function.

```
static int wWidth = MAX(512, DIM);
static int wHeight = MAX(512, DIM);
...
int initGL(int *argc, char **argv)
{
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(wWidth, wHeight);
    glutCreateWindow("Compute Stable Fluids");
    ...
```

The *GLUT_DOUBLE* option specifies that double buffer rendering will be used. In this strategy, one buffer is being used to display the results on the screen device while the second is, at the same time, manipulated to make rendering operations for the next frame that will be sent to the screen. When the screen has started showing the results of the first buffer (*vsync*) and the next frame processing is finished, the program calls *glutSwapBuffers()* to send the new frame to the display and use the other buffer to render another frame.

Next, event listener functions are registered. These functions will be called when the window need to be redrawn (*display*), the window is resized (*reshape*), a keystroke is pressed (*keyboard*), the mouse is clicked (*click*), or the mouse is dragged (*motion*).

```
    ...
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(click);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);

    return true;
}
```

4.2.3 Main loop

Once *glutMainLoop()* is executed, the *display()* function will be called. In the *display()* function a fluid simulation iteration takes place (*simulateFluids()*, then, after the particles new positions are calculated, *glDrawArrays* is used to render a point (*GL_POINTS*) for each particle in the particles' array. The buffer with the drawn points is then made visible in the screen with *glSwapBuffers()*. Finally, a call to *glutPostRedisplay()* makes sure *display()* is called again, creating a loop. This process is illustrated in fig. 6.

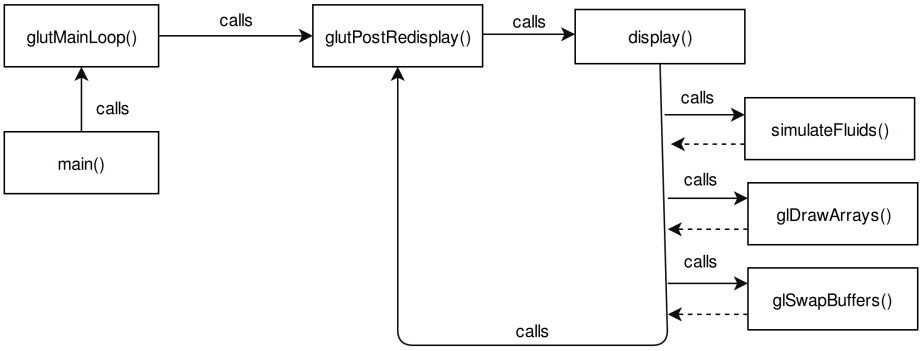

Figure 6: Main loop scheme

4.2.4 Mouse events

Mouse events are used to generate and simulate external forces, giving the user the ability to interact with the fluid.

The *click* listener register in which location the user has clicked the canvas. The listener is called when the mouse is pressed and when the mouse button has been released. We register if the mouse is still pressed with the *clicked* variable. When the mouse is pressed the positions of the click are registered in the *lastx* and *lasty* variables.

```
void click(int button, int updown, int x, int y)
{
    lastx = x;
    lasty = y;
    clicked = !clicked;
}
```

The *motion* listener is called as the mouse moves across the window whether the mouse is pressed or not. For this reason, the *clicked* variable is checked before attempting to add an external force (with *addForces*). If the mouse is clicked, the force to be added is proportional to the distance from the cursor

to it last position (*lastx* and *lasty*). The force is applied in a *FR* radius at a coordinate that reflects the motion of the mouse.

4.2.5 Keyboard events

The keyboard listener checks if the Esc or r key is pressed to provide some useful functionality. When the r key is pressed, the simulation is reset: the particles positions are randomly generated again. This avoids the need to constantly close and run the program. In addition, when the Esc key is pressed, the program exit with success.

4.3 Simulation Steps Implementation

All the simulations steps are inside the *simulateFluids* method, which is repeatedly called by the main loop (note that the external force step was not in our original mathematical algorithm, *addForces* is called from the *motion* listener).

Velocity is measured relative to the domain size. A x velocity component with value of 0.5 in a $10 size\_unit$ horizontal dimension (dx) would have the same absolute velocity as a value of 0.25 in a $20 size\_unit$ horizontal dimension. See fig 7.

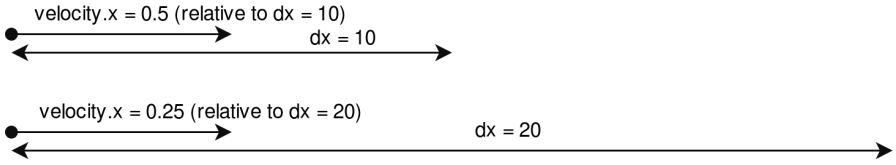

Figure 7: How relative velocities work

This is the process for simulating one simulation iteration:

1. advect real velocity vectors
2. transform real velocity vectors to complex (r2c, real to complex)
3. diffuse complex velocity vectors
4. project complex velocity vectors
5. transform complex numbers back to real ones (c2r, complex to real)
6. normalize velocity vectors
7. map vertex buffer
8. advect vertex buffer particles'
9. unmap vertex buffer

Since both diffuse and project steps (steps 3 and 5) need to operate with complex numbers, both computations are included inside the *diffuseProject* function. Moreover, at the beginning of such function, the real numbers of the input are transformed to complex numbers (step 2). Ultimately, complex numbers are transformed back to the real numbers (step 5) that will be returned.

The normalization is step is required in order the force the velocity values to remain in the domain-size scale (values relative to the domain size). This step is perform by *updateVelocity* which directly saves the normalized vectors in the *hvfield* which will be used by *advectParticles*.

See the simplified simulation workflow that represents the nine steps at fig. 8.

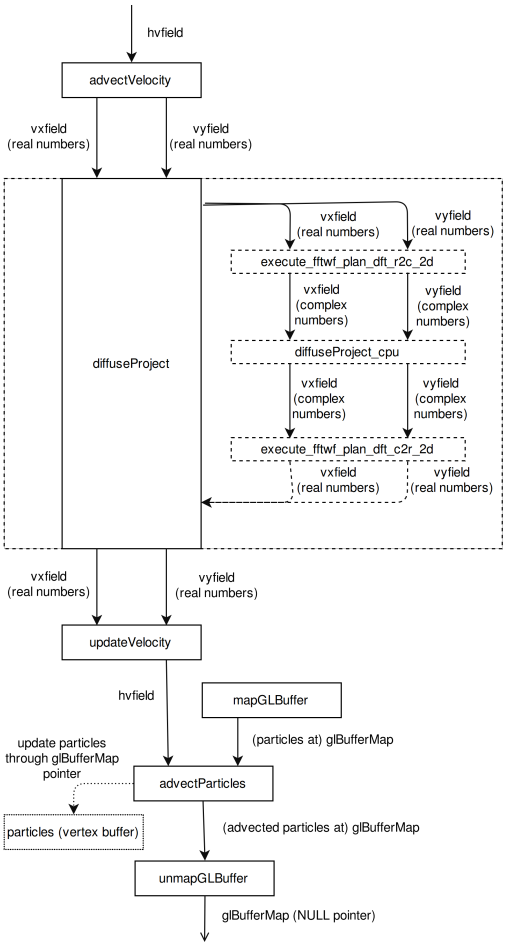

Figure 8: Workflow of the implemented fluid simulation

Note how *advectVelocity* advect the *hvfield* velocity vectors directly storing

the real results in *vxfield* (for x components) and *vyfield* (for y components). By storing the real results directly to *vxfield* and *vyfield* it's not needed to copy the results in an additional step before using *fftw3* and executing *diffuse* and *project*.

4.3.1 Array Indexing

All the steps (except for adding externals forces) need to access every one of the velocity vectors in the grid to perform operations in each one of them (for this reason, a high throughput can be archived using a parallel computing solution like CUDA).

In memory, the elements of same domain's row are stored contiguously. Hence, we iterate our array row by row (in the order the elements are stored in memory). See fig. 9.

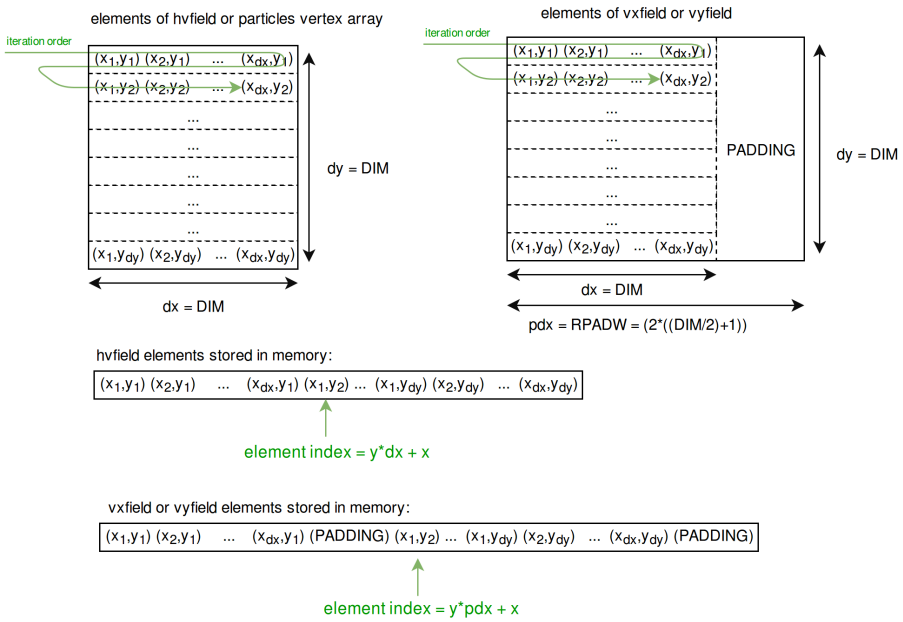

Figure 9: Arrays indexation

Already knowing how arrays will be indexed an iteration template for *advectVelocity*, *diffuseProject*, *updateVelocity* and *advectParticles* can be made:

```
int x, y;
for(x=0; x < dx; x++) {
   for(y=0; y < dy; y++) {
       // hvfield/particles field index: hvfield[f], glBufferMap[f]
      int f = y*dx + x;
```

```
    // vxfield/vyfield index: vxfield[fj], vyfield[fj]
    int fj = y*pdx + x;

    // insert simulation step code here
  }
}
```

Finally, let us note that in some of the simulation steps, a cell in the grid may need to access a neighborhood velocity vector. Because of the periodic boundary conditions, coordinates calculated outside the domain size are transformed in equivalents coordinates that don't exceed the domain dx or dy dimensions nor are in negative values.

4.3.2 advectVelocity

First, the velocity is traced back: $-\vec{u}(\vec{x}, t) \cdot dt$. Relative velocity components are multiplied by the domain horizontal (dx) and vertical (dy) dimensions to obtain the absolute velocity values.

```
cData vterm, ploc;
vterm = vhfield[f];
ploc.x = x - (dt * vterm.x * dx);
ploc.y = y - (dt * vterm.y * dy);
// translate ploc.x and ploc.y into equivalent coordinates.
```

However, the floating position stored in *ploc* can not be used to index and get the needed velocity vector. For this reason, we interpolate the grid velocity values around the *ploc* location as shown in fig. 10.

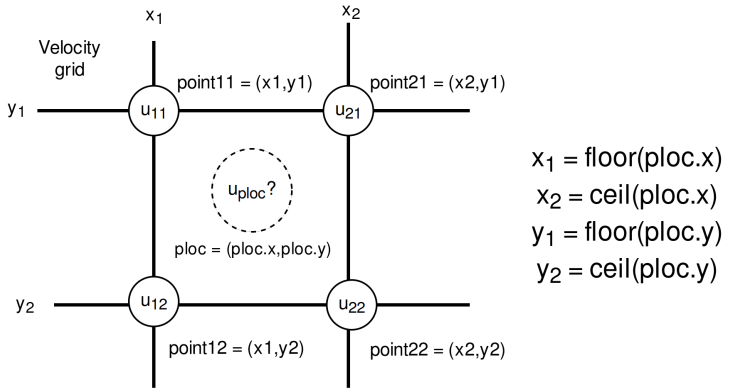

Figure 10: Velocity at *ploc* coordinates does not exist in the grid.

Once the interpolated velocity vector is calculated (each component is interpolated separately) it is stored in *vxfield* and *vyfield*.

```
// Calculate the integer coordinates point11, point12, point22
//  around the floating point ploc coordinate.
// Calculate array indexes and obtain velocity values for
//  each pointXY
vterm.x = BilinearInterpolation(point11, point12, point21, point22,
    u12.x, u12.x, u21.x, u22.x, ploc);
vterm.y = BilinearInterpolation(point11, point12, point21, point22,
    u12.y, u12.y, u21.y, u22.y, ploc);
int fj = y*pdx + x;
vxfield[fj] = vterm.x;
vyfield[fj] = vterm.y;
```

4.3.3 diffuseProject

First of all, the real numbers stored in *vxfield* and *vyfield* will be independently transformed to complex numbers creating a *fftwf* plan and then executing it.

```
  plan = fftwf_plan_dft_r2c_2d(DIM, DIM, (float*) vxfield, (float
      (*)[2]) vxfield, 0);
  fftwf_execute(plan); // fft for the vxfield
```

The remaining *diffuseProject* code pretty much reflects what's explained in the mathematical foundations section for the diffuse and project steps: The squared wavenumber is computed and the diffusion formula is applied to the velocity vector, only to later apply the projection formula.

4.3.4 updateVelocity

This is a very simple step. After applying a convolution in the FFT (fast Fourier transform) at the previous step, the velocity vectors need to be normalized. This is achieved by dividing each vector by the domain dimension ($DS = DIM * DIM = dx * dy$).

4.3.5 advectParticles

First of all, particle positions are relative to the domain size in the same way velocity values are relative to the domain size. Consequently, particle values are in the $[0, 1]$ range, as the OpenGL draw function requires. Since both velocity and position are in the same relative scale we can normally operate and apply the object advection formula.

```
pterm.x += dt * vterm.x;
pterm.y += dt * vterm.y;
```

4.3.6 addForces

In this method, the force (*fx*,*fy*) is added in a *r* radius.

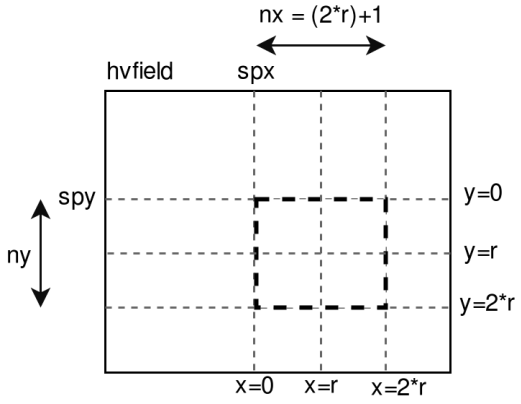

Figure 11: Add forces

The iteration of the velocity array starts at (*spx*,*spy*).

```
for(x=0; x < nx; x++) {
   for(y=0; y < ny; y++)
   { // Here is the code to iterate the array to add forces
      int viy = y + spy;
      int vix = x + spx;
      int f = viy*dx + vix;
```

Before adding the force to each array vector in the iteration, a smooth factor is applied to the force. The smooth permits to apply lower forces to the farthest from center coordinates, making the force addition behave more realistically.

```
int xmr = x-r; // when x = r -> xmr = 0 \
int ymr = y-r; // and y = r -> ymr = 0   |-> s/fx/fy takes maximum value
float s = 1.f / (1.f + xmr*xmr*xmr*xmr + ymr*ymr*ymr*ymr);
vterm.x += s * fx;
vterm.y += s * fy;
```

5 Debugging

Several bugs have occurred during the development (port of the GPU demo to a CPU version) of the application. With the purpose of detecting these errors and verifying that the application works correctly after they have been fixed, the application has a built-in test where the simulation domain is reduced to a $8x8$ square and some predefined forces are added. Because the particle pseudo-

random algorithm is consistent between architectures, each time the test is run, the same results are obtained, making it suitable for debugging.

If the source code is compiled after uncommenting the

```
#define DUMPEABLE
```

preprocessor directive at *fluidsGL.cpp* and *fluidsGL_cpu.cpp*, a series of memory dumps and debug messages will be printed at *dump_test/dump.txt* (relative to the working directory): the content of *hvfield*, *vxfield* and *vyfield* is shown after and before each simulation step-function, and also after and before *addForces*. Additionally, the content of the particles vertex array is shown after mapping and before unmaping. Finally, diverse debug messages permit to follow how each simulation step-function works by showing the content of important variables.

The debug test can be launched with:

```
$ ./fluidsGL test
```

You can find the full trace at the original file from the source code available at `rtfluids.bdevel.org`.

6 Benchmarking

With the purpose of estimating the average frames per second the implementation can achieve, we have codified an automatic test that runs 10,000 simulation iterations before stopping. This measure is notably appropriate to compare the performance with different hardware, compilations, and different implementations (for example, to measure the performance against a CUDA implementation).

When running the test: mouse or keyboard interaction is disabled. Each 10 iterations the benchmark will generate a predefined external force to alter the state of the fluid.

```
void display(void)
{
   ...
   if(stress_test && stress_test_step_index < 10000) {
       if(stress_test_step_index % 10 == 0) {
           // Add Force (simulate click)
           // each 10 steps
         ...
           addForces(hvfield, DIM, DIM, spx, spy, FORCE * DT * fx, FORCE
               * DT * fy, FR);
         }
         stress_test_step_index++;
```

The code for calculating the frames per second is included inside the main loop, inside the *display* function. Every time after a frame is rendered with

glDrawArrays, a frame counter is increased. Approximately every one second the fps is calculated by dividing the frame counter by the time passed since the last fps calculation. The current time is obtained with *glutGet(GLUT_ELAPSED_TIME)*, which reports the time in milliseconds. The measured fps samples, as well as the average fps, are reported via standard output.

```
void display(void)
{
   ...
   fpsCount++;
   time = glutGet(GLUT_ELAPSED_TIME)
   if ((time - base_time) > 1000)
   // atleast 1 second guaranteed between each measure
   // approximately 1 second between each measure
   {
      // multiply by 1000 to convert ms to seconds
      fps=fpsCount*1000.0f/(time - base_time);
      // update last taken fps time (base_time)
      base_time = time;
      // reset fps count
      fpsCount=0;
       ...
```

The benchmark can be launched with:

```
$ ./fluidsGL stress_test
```

7 Basic Compilations

With the premise of the compiler being able to automatically optimize the program, the source has been compiled with all the different available optimization flags (*-O*) in the g++ (GNU C++ compiler) and icpc (Intel C-plus compiler) compilers. Note that the optimization flags are only applied to the *bilinear_interpolation.cpp* and *fluidsGL_cpu.cpp* files where the heavy simulation steps computation are performed: It does not make sense to optimize other files - this hypothesis is confirmed later at the profiling stage.

The basic Makefile rules are as follow:

```
defines.o: defines.c
   $(CXX) $(CXXFLAGS) -c defines.c -o $(BINARY_DIR)/$@
bilinear_interpolation.o: bilinear_interpolation.cpp
   $(CXX) $(CXXFLAGS) $(OPT_FLAGS) -c bilinear_interpolation.cpp -o
       $(BINARY_DIR)/$@
fluidsGL.o: fluidsGL.cpp
   $(CXX) $(CXXFLAGS) -c fluidsGL.cpp -o $(BINARY_DIR)/$@
fluidsGL_cpu.o: fluidsGL_cpu.cpp
```

```
   $(CXX) $(CXXFLAGS) $(OPT_FLAGS) -c fluidsGL_cpu.cpp -o
       $(BINARY_DIR)/$@
fluidsGL: fluidsGL.o fluidsGL_cpu.o bilinear_interpolation.o defines.o
   $(CXX) $(CXXFLAGS) $(BINARY_DIR)/defines.o
       $(BINARY_DIR)/bilinear_interpolation.o
       $(BINARY_DIR)/fluidsGL_cpu.o $(BINARY_DIR)/fluidsGL.o -o
       $(BINARY_DIR)/$@$(BIN_POST_NAME)$(DBG_NAME_APPEND) -lGL -lGLU
       -lGLEW -lglut -lfftw3f
```

This basic options permit to compile all the different binaries by only setting-up the proper flags. The *CXX* flag selects which compiler is used. In the *CXXFLAGS* the *-Wall* option is included to show all warnings when compiling (can be also used with the *-g* option, enabling the gdb, GNU debugger, debugging symbols). The *OPT_FLAGS* present at the *fluidsGL_cpu.o* and *bilinear_interpolation.o* object rules contain the different optimization flags to be used. Finally, for the final binary, an identity string is appended at the end with *BIN_POST_NAME* and *DBG_NAME_APPEND* to differentiate between different types of compilations.

The *build* rule permits to compile all the different binaries. Have a look at the following fragment as a example of how to compile for g++ zero optimization and icpc level 3 (*-O3*) optimization:

```
INTEL_COMPILERS_FOLDER =
    /opt/intel/compilers_and_libraries/linux/bin/intel64
INTEL_ICPC = "$(INTEL_COMPILERS_FOLDER)/icpc"

build:
   make clean-objects; make build-gcc-O0
   make clean-objects; make build-icc-O3
   # ... other builds

build-gcc-O0: CXX = g++
build-gcc-O0: BIN_POST_NAME = -gcc-O0
build-gcc-O0: OPT_FLAGS = -O0
build-gcc-O0: fluidsGL

build-icc-O3: CXX = $(INTEL_ICPC)
build-icc-O3: BIN_POST_NAME = -icc-O3
build-icc-O3: OPT_FLAGS = -O3
build-icc-O3: fluidsGL
```

8 Profiling

8.1 GNU Profiling Tool

The GNU profiling tool (*gprof*) help us to determine which parts of the program are taking most of the execution time.

In the make file there is defined a rule to compile with *gprof*: a g++ build with *-O0*, a g++ build with *-O1* and a icpc build with all optimization flags enabled. The profile comparison between the *-O0* and *-O1* will help us understand what section of the code is being benefited (if there is any performance benefits) by basic optimization. We can as well compare the profile of the build with less optimization operations *-O1* and the one with the more a complex optimization.

Here is how to make the gprof version of the g++ without optimizations: rule (*build-gcc-O0*) is reused, adding an additional *gprof -pg* flag and the gdb debug flag *-g*.

```
build-profile-gcc-O0: DBG_NAME_APPEND = -dbg-gprof
build-profile-gcc-O0: CXXFLAGS += -g -pg
build-profile-gcc-O0: build-gcc-O0
```

Another rule is included to actually run all the profiling test automatically (after they have been compiled), copying the results to the *profiling/*gprof.txt* file.

```
$ make build-profile # build all profile binaries
$ make profile     # run and get profiling results for all gprof binaries
```

If we take a look at the profiling results for the *gcc-O0* build we can observe how the most computationally costly simulation step is velocity advection, followed by particle advection (which need to access to the GPU (vertex buffer) memory). Surprisingly, the the fft-diffusion-projection function is the one that needs less time to execute. The program is the 99.7 percent of the time executing any of these steps.

% time	cumulative seconds	self seconds	calls	name
35.48	198.16	198.16	10004	advectVelocity_cpu
26.70	347.26	149.10	10004	advectParticles_cpu
19.29	454.96	107.70	5244977152	BilinearInterpolation
9.91	510.31	55.35	10004	updateVelocity_cpu
8.41	557.28	46.97	10004	diffuseProject_cpu
0.00	558.91	0.00	1000	addForces_cpu

There is an interesting fact, half the time (198.16/107.70) spent in the velocity convection method is spent interpolating.

index	% time	self	children	called	name
[4]	54.7	198.16	107.70	10004	advectVelocity_cpu [4]
		107.70	0.00	5244977152/5244977152	BilinearInterpolation [7]

The complete profiling results can be found with the source code.

8.2 Advanced Compiler Optimizations

In addition to the four different Intel builds (from *-O0* to *-O3*) an additional build for the Intel compiler is the result of compiling with additional optimization flags. These flags are inspired from the SPEC benchmark where different computers and compilation configurations are listed along with their performance punctuation. We have explored the SPEC floating point 2006 results. The 2006 benchmark features a C test where incompressible fluids are simulated (test identified as *470.lbm*), we have special attention to this part of the results.

The Intel compilers are proprietary software and require license activation. Students versions can be found along the *Parallel Studio XE* software package at the following website: `https://software.intel.com/en-us/qualify-for-free-software/student`.

8.2.1 Profile-Guided Optimization

We have observed that the Intel compiler with Profile-Guided optimization and optimization level 3 is used in almost all configurations. Profile-Guided optimization consists in first building with optimization disabled and the *-prof-gen* flag activated, then when the binary is executed, dynamic information files are generated. These dynamic profile files can be used to build another build with the *-prof-use* flag. With the dynamic profile files, the compiler intelligence to optimize the code is augmented.

8.2.2 Advanced Optimization Flags

The *-no-prec-div* tag is also frequently used, this disable float precision improvement on divides. *-ansi-alias* enables additional aggressive optimization for programs that follow the ISO C Standard. *-qopt-prefetch* enables prefech insertion.

-qopt-malloc-options=3, fairly used too, it let us select which *malloc* algorithm we want to use; a non-zero value tells the compiler to intelligently insert *mallopt()* configuration calls with the purpose of improving execution speed.

Finally, another flag frequently used is *-xSSE*, this enables Streaming SIMD Extensions. With this option the compiler may automatically generate specific SIMD instructions to perform parallel vector operations.

9 Performances

optimization flags	**fps (GNU compiler)**	**fps (Intel compiler)**
-O0	18.041941	17.385107
-O1	26.955856	27.730045
-O2	28.773472	28.940458
-O3	28.413486	31.053715

-xSSSE3 -no-prec-div -qopt-malloc-options=3 -ansi-alias -qopt-prefetch	-O3 -prof-use	33.206783

As it can be appreciated, Intel compilations perform better as the optimization level increases. The Intel performance increase as each optimization level increases, the same can not be said about the GNU builds because the third optimization level performance is nearly the same as the second optimization level performance.

In favor of GNU, we have to say that their average binary size is less than half of the Intel binary size, making it preferable for optimization levels zero, 1 and 2.

The profile-guided optimization and the additional optimization tags successfully provide a performance edge.

The next table shows the reported *gprof* self-seconds for each simulation function for different builds. Optimizations are mainly made in the *advectVelocity* method (and the subsequent bilinear interpolation).

function self-seconds:	**GNU -O0**	**GNU -O1**	**Intel Advanced Opt.**
advectVelocity	305.86	167.44	95.79
diffuseProject	46.97	22.37	16.33
updateVelocity	55.35	37.99	42.43
advectParticles	149.10	86.57	84.11

The machine specifications where the builds have been tested are in the appendix A.

References

[1] Mark J. Harris, *GPU Gems: Chapter 39. Fast Fluid Dynamics Simulation on the GPU*, NVIDIA, University of North Carolina at Chapel Hill. Available online: `http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html`.

[2] Nolan Goodnight, *CUDA/OpenGL Fluid Simulation*, NVIDIA Corporation.

[3] Jos Stam, *Stable Fluids*, In Proceedings of SIGGRAPH 1999. Available online: `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf`.

[4] *Multi-Dimensional DFTs of Real Data*, *fftw.org*. Available online: `http://www.fftw.org/doc/Multi_002dDimensional-DFTs-of-Real-Data.html`.

[5] *Profile-Guided Optimization (PGO) Quick Reference*, Massachusetts Institute of Technology at Computational and Systems Biology Initiative. Available online: `http://www.csbi.mit.edu/technology/intel_fce/doc/main_for/mergedProjects/optaps_for/common/optaps_pgo_bsic.htm`.

[6] *Intel(R) C++ Compiler XE 12.1 User and Reference Guides: Advanced Optimization Options*, Intel Corporation. Available online: `http://technion.ac.il/doc/intel/compiler_c/main_cls/copts/common_content/options_ref_bk_adv_optimiz.htm`.

[7] *CFP2006 Rates Results*, SPEC. Available online: `https://www.spec.org/cpu2006/results/rfp2006.html`.

A Target machine specifications

1. OS: Ubuntu 14.04.4 LTS 64 bits
   kernel: Linux 3.13.0-57-generic (x86_64)
2. CPU: 8x Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz
3. RAM: 18488 MB
4. GPU: GeForce GTX 780
   CUDA Capability 3.5
   NVIDIA Driver Version: 361.93.02
5. Compilers:
   g++ version 4.8.4 (Ubuntu 4.8.4-2ubuntu1 14.04.3)
   icpc 17.0.1