CUDA/OpenGL Fluid Simulation

Nolan Goodnight
ngoodnight@nvidia.com

July 2012

Document Change History

Version	Date	Responsible	Reason for Change
0.1	2/22/07	Nolan Goodnight	Initial draft
1.0	4/02/07	Nolan Goodnight / Mark Harris	Release version
1.0	5/17/07	Nolan Goodnight	Minor changes for Cuda 1.0

Abstract

This document describes an NVIDIA CUDA implementation of a simple fluids solver for the Navier-Stokes equations for incompressible flow. The CUDA algorithms are based on Jos Stam's FFT-based Stable Fluids system [1], and we refer the reader to this paper for mathematical and algorithmic details. We present the basic steps in one time iteration of the solver, and we show how each kernel is implemented using the CUDA C programming language. The results are displayed using OpenGL to render a particle system within the fluid. Figure 1 shows a screenshot of the fluids after several iterations.

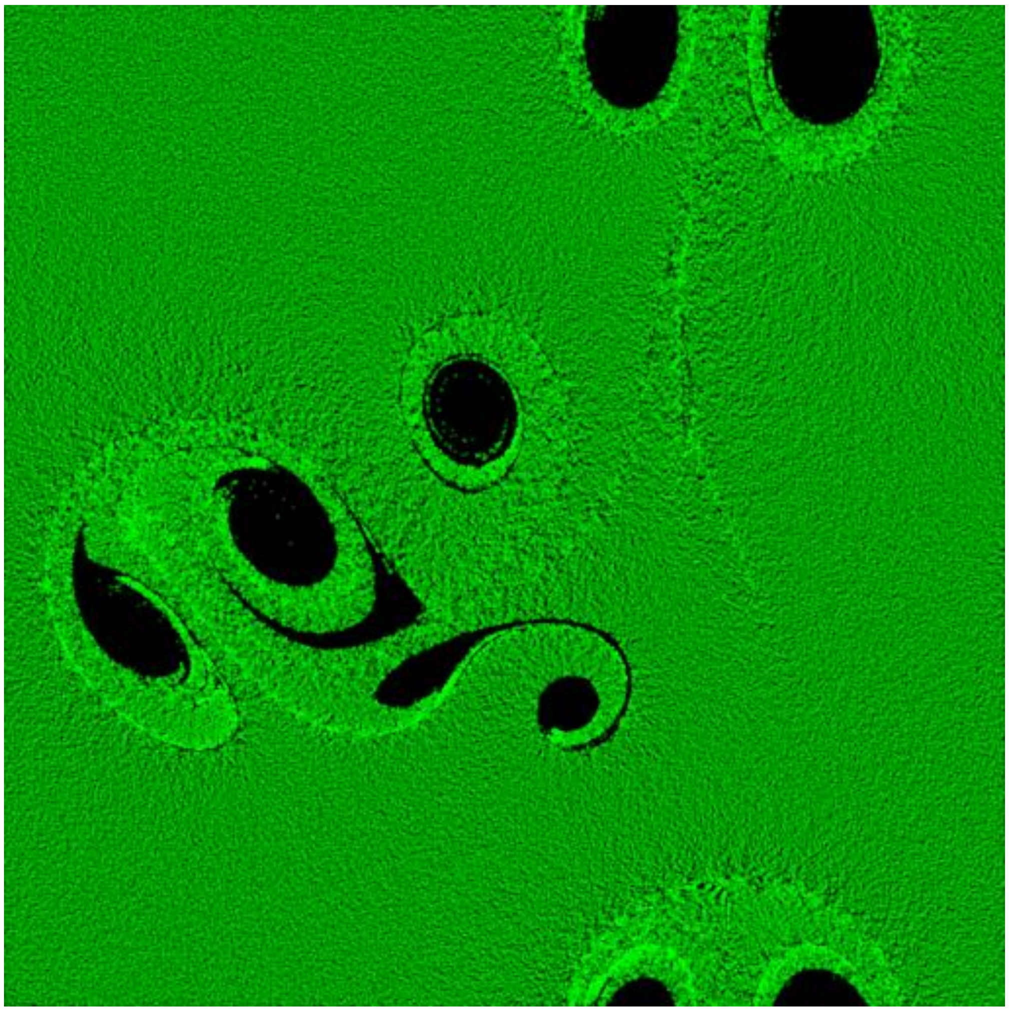

Figure 1. A screenshot of the CUDA fluids solver running on a 512x512 domain. The fluid is displayed using particles that move according to the velocity field.

Motivation

The core component of any fluid simulation is a numeric solver for the Navier-Stokes equations, represented over a uniform grid of spatial locations or grid cells. The solver works iteratively, where at each iteration or time step a set of discrete differential operators are applied to small groups of cells across the domain. These operators update fluid state, such as velocity and pressure, based on the grid values from the previous time step. For this implementation of Stable Fluids [1], one iteration consists of the following steps:

Forces:	Fluid velocity evolves according to force inputs over time. This step updates the fluid velocity field by integrating forces at each grid cell.
Advection:	To update the fluid velocity field, this step transfers velocity values between grid cells. The transfer distance depends on the magnitude of the velocity vectors.
Diffusion:	All fluids have some viscosity, which is a measure of how force inputs diffuse across the velocity field over time. This step simulates viscosity by applying a diffusion operator to the velocity field.
Projection:	For incompressible fluids, the velocity field must remain non-divergent or mass conserving. Otherwise, portions of the fluid will disappear and the simulation will fail to produce correct results. This step forces the updated velocity field to be non-divergent.

The operators (or kernels) used in these steps access data in local neighborhoods around each grid cell. As a result, the algorithms are highly data-parallel and therefore a good match for the CUDA programming model. We parallelize the fluid solver by executing blocks of hundreds of threads that span the computational domain, where each thread applies these operators to a small number of grid cells. We present details in a later section.

In addition to presenting implementation details and kernel code, this sample provides examples of how to use several features of the CUDA runtime API, user libraries, and C language. These features, which are explained in detail in the CUDA Programming Guide, include:

CUDA Texture references: Most of the kernels in this example access GPU memory through texture. In CUDA, this is done using the texture reference type. *(For details, see sections 4.3.4 and 4.5.2.3 of the programming guide.)*

The CUDA Array type: The fluids solver computes results in a 2D grid. In many cases, it is fastest to use 2D Arrays for intermediate results because CUDA Arrays abstract the GPU's memory layout (with optimal 2D texture cache performance), while memory from cudaMalloc() is linear. *(For details, see section 4.5.2.2 of the programming guide.)*

The CUFFT user library: This example implements the FFT-based version of the Stable Fluids algorithm. This approach performs velocity diffusion and mass conservation in the frequency domain, and we use the CUDA FFT library to perform Fourier transforms. *(For details, see the CUFFT documentation.)*

The CUDA/OpenGL interoperability API: To display the fluid, we move particles using the fluid velocity field, and we draw the particles using simple GL calls. The interoperability API allows us to share memory between CUDA and OpenGL, so we can update the particle system using CUDA, and render from the same memory using OpenGL. *(For details, see section 4.5.1.4 of the programming guide.)* Note that the CUDA Developer SDK also includes "fluidsD3D", a version of the fluids simulator that uses Direct3D for rendering and demonstrates the CUDA Direct3D interoperability API.

Implementation

This section describes in detail the CUDA C implementation of each kernel in the Stable fluids algorithm. As is typical for data-parallel algorithms over 2D grids, we execute blocks of threads in which each thread is responsible for all computation within a small region of the fluid domain. For this example, most kernels use the following tile and thread block configuration:

Tile size: The fluid domain is divided into tiles of 64-by-64 cells. We tile the entire domain in this way, using partially full tiles if the domain dimensions don't divide evenly by 64.

Block size: 256 threads divided logically into 64 threads in *x* times 4 threads in *y*. The threads are distributed over the tile such that each thread computes results for a vertical column of 16 grid cells.

This thread configuration gives us a high-performance balance between the number of threads per block, the maximum number of blocks per multiprocessor in the chip, and number of per thread resources (e.g. local registers) available. Figure 1 illustrates this block/thread configuration. Because it's possible that some threads blocks will overlap the fluid domain (Figure 1 b), we use the CUDA C code in Listing 1 to check that each thread is inside the domain boundaries. This code is used in all __global__ functions for the fluid solver, but the constant 16 may be replaced by a variable parameter if the tile size changes.

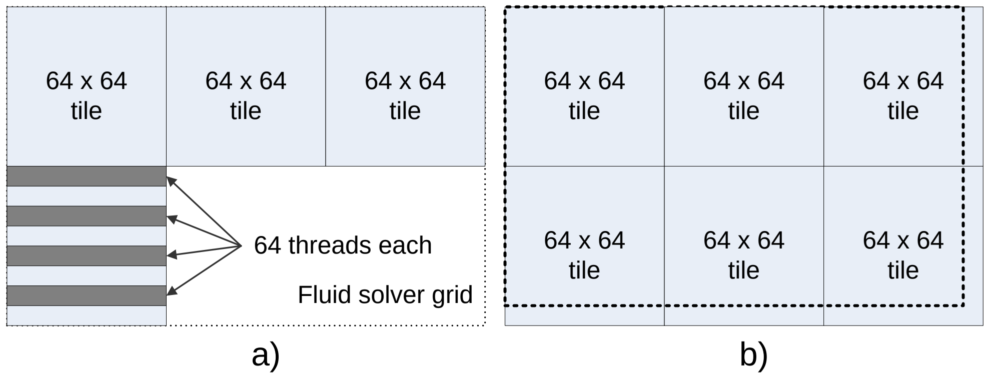

Figure 1: Block diagram of the thread configuration used for all kernels in this sample. A) we use blocks of 64 x 4 threads, where each thread computes results for a vertical column of 16 grid cells. B) If the domain dimensions (dotted line) are not integer multiples of the tiles size, we overlap the domain with extra tiles.

```
int gtidx = blockIdx.x * blockDim.x + threadIdx.x;
int gtidy = blockIdx.y * (16 * blockDim.y) + threadIdx.y * 16;

// gtidx is the domain location in x for this thread
// dx is the domain size in x
if (gtidx < dx) {
    for (int p = 0; p < 16; p++) {
        int fi = gtidy + p;
        // fi is the domain location in y for this thread
        // dy is the domain size in y
        if (fi < dy) {
            // Kernel for each step code goes here
        }
    } // For all cells in a vertical column of 16
} // If this thread is inside the fluid domain
```

Listing 1: CUDA C code for testing whether a thread is inside the fluid domain.

Add Forces

We add forces to the velocity field by translating user mouse motion into 2D force vectors. We launch a single thread block on the GPU, and each thread in the block computes a force value based on the distance from the center cell. Threads at the far edges of the thread block tile add smaller force updates to the velocity than threads near the center grid cell. The following is CUDA C code for adding forces:

```
// tx is the thread location in x
// ty is the thread location in y
// fj is the global thread position in the velocity array
int tx = threadIdx.x;
int ty = threadIdx.y;
cData *fj = (cData*)((char*)v + (ty + spy) * pitch) + tx + spx;

// We offset the local thread position by r, which is the
// radius of the force thread block
// cData is a float2 type
cData vterm = *fj;
tx -= r; ty -= r;
// For smoothness, we compute a 1/(1 + x^4 + y^4) force falloff
float s = 1.f / (1.f + tx*tx*tx*tx + ty*ty*ty*ty);
vterm.x += s * fx;
vterm.y += s * fy;
```

Note that we use a pitch value in the calculation of the global thread position in the velocity field. This is necessary because the velocity memory is allocated using cudaMalloc2D (see section 4.5.2.2 of the *CUDA Programming Guide* for details). All remaining code examples in this document will use a pitch value when calculation global input and output addresses.

Velocity Advection

For velocity advection, we need to compute how the fluid velocity transports across grid cells. We use the implicit advection process proposed by Stam [1], and described in detail by Harris [2]. In this approach, we use the velocity vector at each grid cell to trace back in time to a previous cell, and we replace the starting grid cell velocity with this value from this previous location.

```
cData vterm, ploc;
int fj = fi * pdx + gtidx;
// Fetch the velocity value at the grid cell for the thread
// location (gridx, fi)
vterm = texfetch(texref, (float)gtidx, (float)fi);
// Trace the velocity vector backward to determine a new
// location in the grid
ploc.x = (gtidx + 0.5f) - (dt * vterm.x * dx);
ploc.y = (fi + 0.5f) - (dt * vterm.y * dy);
vterm = texfetch(texref, ploc.x, ploc.y);
```

The second texture fetch in the code above uses bilinear interpolation to get a smooth value. The texture reference object is configured with this type of filtering so that we don't have to manually interpolate velocity values in the CUDA C code and can take advantage of the GPU's built-in texture filtering hardware.

Velocity Diffusion and Projection

This sample implements velocity diffusion and projection in a single kernel. We can do this because both operations are performed on velocity coefficients in the frequency domain. For this step we do the following:

1. Forward FFT of the velocity field
2. Frequency space diffusion to simulate viscosity and projection to make the velocity field non-divergent.
3. Inverse FFT of the velocity coefficients.

For both forward and inversion FFT we use CUFFT, the CUDA FFT library. See the CUFFT documentation for details on how to create and use CUFFT plans. It's important to note that the current version of CUFFT only supports complex-to-complex transforms, while the velocity field is real-valued. Therefore, we allocate extra GPU memory to store complex-valued coefficients for the x and y velocity components.

The differential operator for diffusion in the spatial domain is simply a convolution, which can be implemented as a point-wise multiplication in the frequency domain. Therefore, we can simulate fluid viscosity by scaling the velocity coefficients such that high frequency terms are forced smaller than low-frequency.

```
int fj = fi * dx + gtidx;
xterm = vx[fj];
yterm = vy[fj];

// Compute the index of the wavenumber based on the
// data order produced by a standard NN FFT.
int iix = gtidx;
int iiy = (fi > dy / 2) ? (fi - (dy)) : fi;

// Velocity diffusion
float kk = (float)(iix * iix + iiy * iiy); // k^2
float diff = 1.f / (1.f + visc * dt * kk);
xterm.x *= diff; xterm.y *= diff;
yterm.x *= diff; yterm.y *= diff;

// Velocity projection
if (kk > 0.f) {
      float rkk = 1.f / kk;
      // Real portion of velocity projection
      float rkp = (iix * xterm.x + iiy * yterm.x);
      // Imaginary portion of velocity projection
      float ikp = (iix * xterm.y + iiy * yterm.y);
      xterm.x -= rkk * rkp * iix;
      xterm.y -= rkk * ikp * iix;
      yterm.x -= rkk * rkp * iiy;
      yterm.y -= rkk * ikp * iiy;
}

vx[fj] = xterm;
vy[fj] = yterm;
```

Display

Display of the fluid simulation is performed by rendering points in OpenGL. The positions of the points are initially randomized throughout the domain, and at each time step, the velocity field is used to move the points forward. To do this, `cudaGLMapBufferObject()` is used to map the vertex buffer object of point positions to a CUDA device memory pointer. This allows it to be passed to a CUDA kernel which advects the point positions using the velocity field, as in the following code. The vertex buffer object is then unmapped from the device pointer so that it can be used in normal OpenGL rendering using `glDrawArrays()`.

```
int fj = fi * dx + gtidx;
pterm = part[fj];

int xvi = ((int)(pterm.x * dx));
int yvi = ((int)(pterm.y * dy));
vterm = *((cData*)((char*)v + yvi * pitch) + xvi);

pterm.x += dt * vterm.x;
pterm.x = pterm.x - (int)pterm.x;
pterm.x += 1.f;
pterm.x = pterm.x - (int)pterm.x;
pterm.y += dt * vterm.y;
pterm.y = pterm.y - (int)pterm.y;
pterm.y += 1.f;
pterm.y = pterm.y - (int)pterm.y;

part[fj] = pterm;
```

Bibliography

1. Stam, J. 1999. "Stable Fluids." In Proceedings of SIGGRAPH 1999. http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf

2. Harris, Mark J. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems*, R. Fernando, Ed., ch. 38, pp. 637–665. Addison Wesley, 2004 http://developer.nvidia.com/object/gpu_gems_home.html

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com