Simulation and visualization of fluid flow considering surface smoothness and computation speed

Hu Fangyuan, Murotani Kohei, Shibata Kazuya, and Koshizuka Seiichi*
Koshizuka-Shibata laboratory, the University of Tokyo

A mesh-free method - MPS is implemented for fluid simulation. A Marching Cubes method is applied to obtain the free surface of the fluid which is represented by particles. We used a cubic spline instead of the weight function to map particle data into the scalar field in 3D space. The method shows smoothness in surface representation. Then we give a CUDA implementation of both the simulation part and the rendering part, and for performance consideration, a novel method for neighbor particle searching is given. We finally get a near real-time outcome running at a frame rate of 30fps.

Key Words: Visualization, MPS, Marching Cube; CUDA;

1. Introduction
Computational Fluid Dynamics (CFD) plays an important role in both engineering calculation and virtual reality computer graphics. In both case, construction of the surface of simulated fluid has become the focus of many research. For engineering calculation, the description of an explicit surface means the possibility of obtaining the area of the surface. This is important for chemical simulation and medical simulation. In the case of computer graphics, an explicit surface makes all the surface-based rendering possible.

Research on surface construction and rendering for mesh-based CDF results has been widely developed, like level-set method [1]. This paper deal with surface construction from particle datasets by mesh-free simulation methods. We start from a particle-based simulation by MPS, then after each step we map the particle data onto a 3D volume datasets. Finally we search the volume data for the surface using Marching Cubes method. For high performance, we give an implementation of this process on modern parallel device GPU. This is done by general purpose GPU language CUDA.

1.1 Moving Particle Semi-implicit method
Moving Particle Semi-implicit method (MPS) [2] is a particle-based numerical method for fluid simulation. Each particle has its own position, velocity and other physical values. MPS solves the Lagrangian form of NS equation. For each particle, the governing equation is defined as follows.

$$\frac{Dv}{Dt} = -\frac{1}{\rho} \nabla P + \nu \nabla^2 v + g$$  

(1)

In the standard MPS method, viscosity term and volume force term are calculated first, then by defining a "particle number density": defined in equation (2) for each particle, equation of pressure and particle number density can be built for each particle. This produces a linear equations system. By solving the equations, pressure term can be calculated. Finally we update the velocity. This is why this method is semi-implicit. Details can be found in [2].

$$n_i = \sum_{j\in\Omega, j \neq i} w(r_{ij})$$  

(2)

Here, $j$ is a neighbor particle within a radius and $w$ is a weight function which is inverse proportional to distance between $i$ and $j$.

However, the implicit part of this method will drop performance to the ground. So in this paper, we apply an explicit version of MPS. The pressure term in each step is directly calculated from the particle number density by equation (3), where $\alpha$ is compressibility of the fluid.

$$P_i = \frac{1}{\alpha} \cdot \frac{n_i - n_0}{n_0}$$  

(3)

So the calculation only contains three steps: first calculate the viscosity term and the volume force term. Then calculate the pressure term. Finally update the velocity and position.

This makes calculation about each particle independent, so an implementation by CUDA can be easily done by assigning each thread one particle and do all the calculation. In order to reduce the neighbor particle searching time, we implement a voxel structure, which will be described in Section 2.

1.2 Marching Cubes
Marching Cubes (MC) [3] is a method used most in medical visualization for surface construction of volume datasets. Given a volume dataset, for example, scalar data $\phi$ on a grid structure, MC construct the free surface in the following steps.

First, cubes are built so that they cover the whole volume. Each cube has 8 vertices and 12 edges, as shown in fig.1 (for simplicity in 2D). Then for each vertex, we calculate its $\phi$ by interpolation. If we are looking for a surface, where $\phi = \xi$, vertices can be classified as above the surface ($\phi > \xi$) or below the surface ($\phi < \xi$). So for each cube there are $2^3 = 256$ states. Each state represent a case of how surface intersects the cube. We assume that the surface inside a cube is composed by triangles, so only if we find the vertices of triangles, the surface is defined. Finding the vertices of a triangle is by interpolation on an edge of the cube, as shown in fig.1. Finally, we obtain a set of triangles as the surface. As long as the cubes are small enough, the surface can be represented precisely.

In this paper, in order to exploit the parallelism of a GPU, the
Marching Cubes part is also implemented by CUDA.

2. Implementation

2.1 Novel voxel method for neighbor particle searching

As described in Section 1.1, in order to accelerate the neighbor particle searching, voxels are built at first. The size of voxels are exactly the same as particles’ effective radius. Each voxel records the particles inside.

At the very beginning, particle variables and voxel variables are initialized and sent to the GPU. On the GPU side, first, equation (3) and (1) is calculated. Then, as the position has changed, we need to reconstruct the voxels. This is a time-consuming part in many real-time simulations. We first decide the voxel index for each particle. Then, as the particles in the same voxel may not be neighbors in memory, cache hit miss will affect the performance in later calculation severely, especially in the case of GPU calculation. Furthermore, in order to traverse particles in one voxel, it is usually necessary to implement a link list among particles, which also gives a large overhead.

Conventional methods avoid this by sorting the particles in the memory by their voxel indices, that is, particles with the same voxel index will be put at the same area in the memory. In most cases this sorting is done on the CPU and is very slow. Here we propose as novel method without sorting operation and can exploit more power of GPU.

Our purpose is to rearrange the particles in the memory according to their voxel indices, so we have to give each particle a new ID, which will be their new position in the memory. First, we traverse all the particles and record the voxel index of each particle, at the same time, we record the number of particles in each voxel. We also define a local ID for each particle. The local ID is the current count of particle in its voxel, shown in fig.2. This part can be done in parallel, only with an “atomic” syntax to make sure the number of particles in each voxel is right. Then, we traverse the voxels, for each voxel we calculate the starting number of the new ID, by equation: \( ID_{i} = START_{i-1} + N_{i-1} \), where \( N_{i-1} \) is the number of particles in voxel \( i-1 \). The start ID of each voxel can be used in later iteration. Now we need to find the new ID for each particle. This is very simple because we already have a starting ID in each voxel and a local ID of each particle in that voxel, so the new ID of a particle is calculated by equation: \( ID_{new} = START + ID_{local} \). The last thing is to rearrange particles in the memory in the order of their new ID. This is a mapping operation where no data dependency exists if a temporary array is used, which is shown in fig.2.

Another advantage of this method is that, as only the calculation of number of particles in each voxel is done on CPU, which we believe is faster, the communication from GPU to CPU and vice is reduced. At the beginning of each step, from GPU to CPU, we only need to pass the position information. Then, after the calculation, only \( START \) and \( ID_{local} \) need to be passed from CPU to GPU, because the mapping operation is done by GPU.

2.2 Implementation of MC on GPU

For implementing MC on datasets, we first build the cubes, then calculate particle influence on each vertex. Then, normal value at each vertex is calculated for visualization. The whole process is done on GPU by CUDA.

After building the cubes, the first step is to calculate particle influence on each vertex, that is, to calculate a scalar value for vertices. This is usually done by the metaball method, which consider each particle as a metaball. The influence of each particle is represented by a weight function, like \( W(r_{ij}) \) in equation (2), which is inverse proportional to the distance from the particle. However, in this method, when a particle comes very close to a vertex, the weight value of the vertex becomes infinite, which may cause error when doing interpolation on the edge of cubes. What’s more, this means that only one particle can produce a surface. This may give heavy noise to the final result. Therefore, here we use a cubic spline instead of inverse proportional function to represent the influence of a particle. The cubic spline function is defined by equation (4), where \( q = r/r_{0} \).

\[
W_{spline}(q) = \frac{8}{\pi} \left( \frac{1 - 6q^2 + 6q^3 - 2q^4}{2(1 - q)^3} \right) \quad \text{for} \quad 0 \leq q \leq 1
\]

On the GPU side, we assign each vertex one thread, then we use the voxel in Section 2.1 for neighbor particle searching. Another way is to assign each particle one thread, then calculate its possible neighbor vertices.

In this implementation, we want to render the surface for visualization, so it is necessary to calculate normal vector of the surface. Although it is possible to calculate a normal from the triangle, that will produce artefacts on the final result. So we calculate the normal at each vertex by center difference method, shown in equation (5).

\[
\mathbf{v}_{normal} = -\nabla \phi(p) = \frac{\phi(x_{i+1})-\phi(x_{i-1})}{2\Delta x} \mathbf{i} + \frac{\phi(y_{j+1})-\phi(y_{j-1})}{2\Delta y} \mathbf{j} + \frac{\phi(z_{k+1})-\phi(z_{k-1})}{2\Delta z} \mathbf{k}
\]

The normal at each vertex is calculated after the scalar at each vertex is known. When we do interpolation at edges of the cubes, both vertices position of triangle and the normal vector at those vertices are calculated.

Finally, with the surface described by triangles and normal vectors, we visualize the result by OpenGL.
3. Result and conclusion

The whole process from simulation to visualization is in real-time, shown by fig.3. In this paper we implemented a dam-break simulation test, where the total particle number is about 100K, and the total number of cubes is about 2M. We run the program at a PC with Ubuntu 12.04 LTS system, i7-4820K CPU (3.70GHz) and NVidia GeForce GTX TITAN Black graphic card. Snapshots of the result is shown in fig.4. The program run at 30 step per second. However, in order to maintain stability, the time step in simulation has to be set very small, so even the frame rate is high enough, the program is still far from interactive real-time. Moreover, as the model used in our work does not include surface tension, surface instability still appears to be a problem. These would be our future work.

References: