CS240A Project Report
Multi-GPU Fluid Simulation with Smooth Particle Hydrodynamics

Donghao Ren, Junsheng Guo

March 19, 2014

1 Introduction

The goal of our project is to simulate a block of fluid, like water, with the Smooth Particle Hydrodynamics method in multiple GPUs. We started from a existing SPH code from a previous course (CS280, Fall 2013), which is slow and buggy. In this project, we completed the SPH implementation on the GPU, and also parallelized it for multiple GPUs across nodes.

In this report, we first talk briefly about the SPH formulation of hydrodynamics, then the algorithm and implementation, which consists of single GPU version, multiple GPU version, and rendering. After that, we discuss experimental results. Next we show how to use the code and finally conclude the report.

2 SPH Formulation

Here is a very brief description of the SPH (Smooth Particle Hydrodynamics) algorithm. In fluid dynamics, the law is Navier-Stokes equation:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t}+\mathbf{v}\cdot\nabla\mathbf{v}\right)=-\nabla p+\rho\mathbf{g}+\mu\nabla^2\mathbf{v} \tag{1}$$

This equation can be discretized using a set of particles. Suppose we have a field $A(\mathbf{x})$, we can represent the field with:

$$A(\mathbf{x})=\sum_j m_j\frac{A_j}{\rho_j}W(|\mathbf{x}-\mathbf{x}_j|) \tag{2}$$

The gradient and laplacian operators can be written as:

$$\nabla A(\mathbf{x})=\sum_j m_j\frac{A_j}{\rho_j}\nabla W(|\mathbf{x}-\mathbf{x}_j|) \tag{3}$$

$$\nabla^2 A(\mathbf{x})=\sum_j m_j\frac{A_j}{\rho_j}\nabla^2 W(|\mathbf{x}-\mathbf{x}_j|) \tag{4}$$

Thus we can write the Navier-Stokes equation as the forces for the particles:

$$\mathbf{a}_i = \frac{\partial \mathbf{v}_i}{\partial t} = \frac{\mathbf{f}_i}{\rho_i} \tag{5}$$

$$\mathbf{f}_i^{\text{pressure}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{x}_i - \mathbf{x}_j) \tag{6}$$

$$p = k(\rho - \rho_0) \tag{7}$$

$$\mathbf{f}_i^{\text{viscosity}} = \mu \sum_j m_j \frac{\mathbf{v}_i - \mathbf{v}_j}{\rho_j} \nabla^2 W(\mathbf{x}_i - \mathbf{x}_j) \tag{8}$$

Following the above equations, we can simulate the fluid by solving a set of ordinary differential equations.

3 Algorithm

Here we describe our algorithm. We start from the general procedure for fluid simulation, then go into the single GPU implementation, and finally talk about the multiple GPU version.

The general algorithm for the simulation is outlined as the following.

1. Initialize Particles (initial position, velocity)
2. Simulation Loop
   (a) Compute Forces with the SPH Formulas
   (b) Leapfrog Integration

The initialization and integration steps are pretty straightforward, which only include "embarrassingly parallel" operations which operates on each particle as if they were isolated. We will only talk about the difficult part, the force computation, which involves particle to particle interaction.

3.1 Single GPU Force Computation

We implemented a block-hashing algorithm to compute the neighbor list for particle to particle interaction. Below are the steps of the algorithm.

1. Copy Data to GPU Global Memory
2. Hash Particles
3. Construct Block List
4. Construct Neighbor List for Blocks
5. Compute Density
6. Compute Forces
7. Copy Data Back to Host Memory

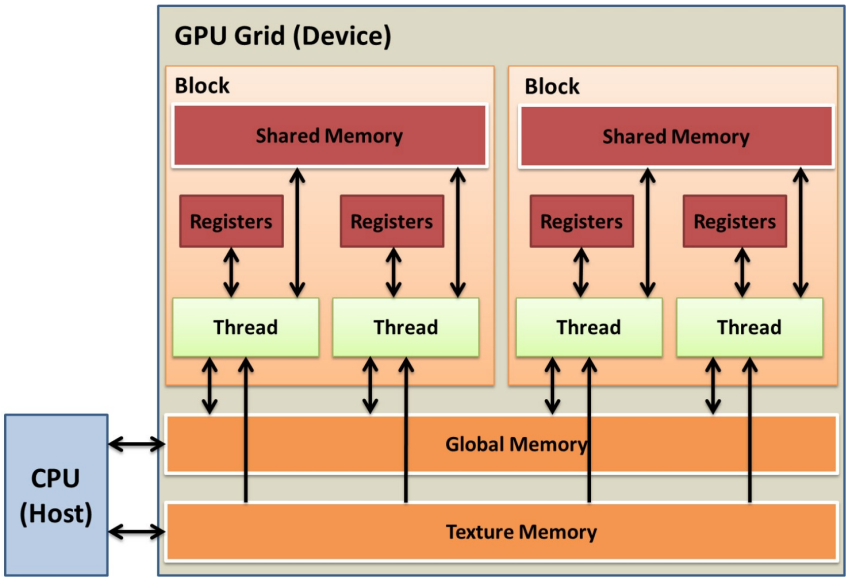

Figure 1: CUDA Architecture

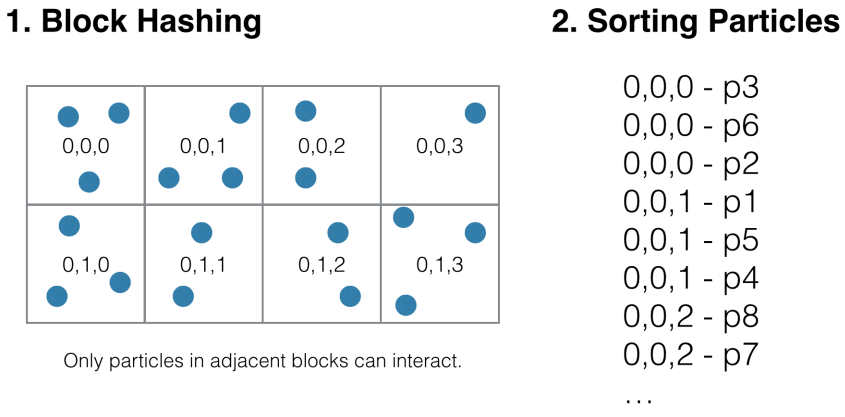

Figure 2: Hashing and Sorting.

As illustrated by Figure 2, the hash function virtually divide the simulation space into blocks, each block has a size proportional to the radius of the kernel function (which is the maximum distance two particles can interact). By dividing particles into blocks, only particles in neighboring blocks can interact with each other. The hash function is quiet simple, just assemble the block index in x, y, z directions into a single 64-bit integer, which can be decomposed to get the location of the block.

Next, we construct the neighbor list. This is done by sorting the particles according to the hash value, so that particles in the same block will stay near to each other, which increases locality. After sorting, we construct a block list (in serial), each block in the list contains a pointer to the first and last particle in the sorted particle list.

Then, we construct the block neighbor list. Each block can have at most 27 neighbors (including itself). We are not storing blocks as an 3D array, thus there is not a predefined simulation region where particles can not fly outside of. In this sense, our algorithm is very flexible, as the fluid may move around, but we do not need to allocate extra space or change the simulation region dynamically to adapt to that.

For the density and force computation, the critical part is to find the neighboring particles for a given particle. In our implementation, each block is run as a CUDA block, and each particle inside a block is run as a CUDA thread. Thus each CUDA thread has to deal with the neighbors for one particle. This is done by the following steps:

1. For Each Neighboring Block of the Block:
   (a) Copy the particles in the neighboring block to shared memory (each thread copy a single particle)
   (b) syncthreads
   (c) Compute particle to particle interaction for the particle and each neighboring particle.
   (d) syncthreads

Here we are using shared memory to reduce the amount of global reads, which has a positive impact on performance, but not that much. One problem for this approach is that the shared memory can contain only a small amount of particles, so we have to divide a block into several blocks if it is too large to fit into the shared memory.

3.2 Multiple GPU Communication

The multiple GPU version works by dividing the simulation region on one axis (x axis in our case). For correct force computation, each process should retrieve a strip of ghost particles from its neighboring processes. In addition, after integration update, particles will move, so the processes should send moved particles to the corresponding process. In summary, the algorithms works like the following (also see Figure 4).

1. Send Ghost Particles to Neighboring Processes
2. Compute Force
3. Leapfrog Integration
4. Send and Receive Moved Particles to Corresponding Processes

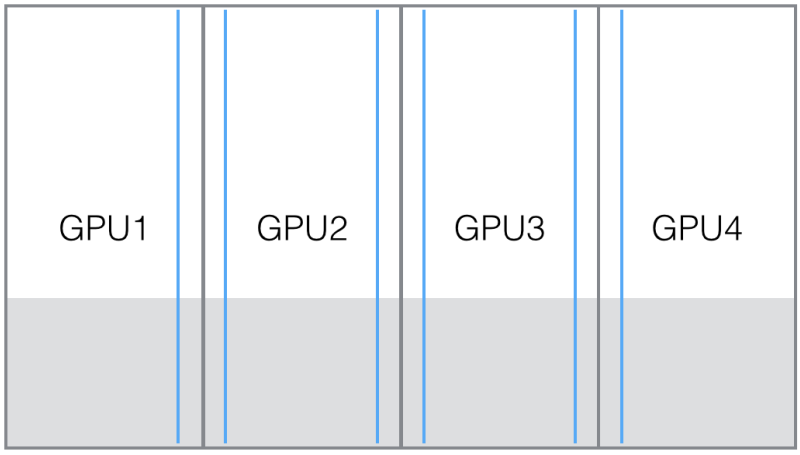

Figure 3: Dividing Particles for Processes.

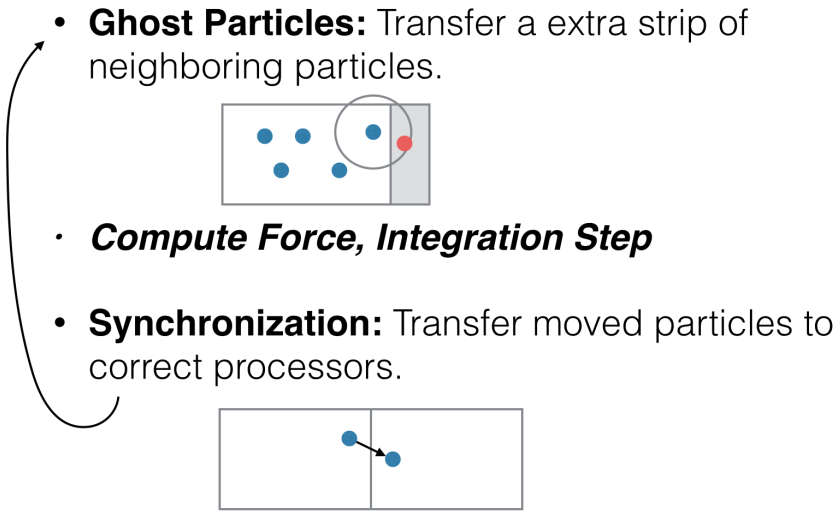

Figure 4: Multiple GPU Algorithm Steps.

The particle transfers are done by a even-to-odd and then odd-to-even approach, in order to avoid deadlocks.

3.3 Rendering

Here we talk about the rendering algorithms used.

3.3.1 Single GPU Ray-tracing in the Volume

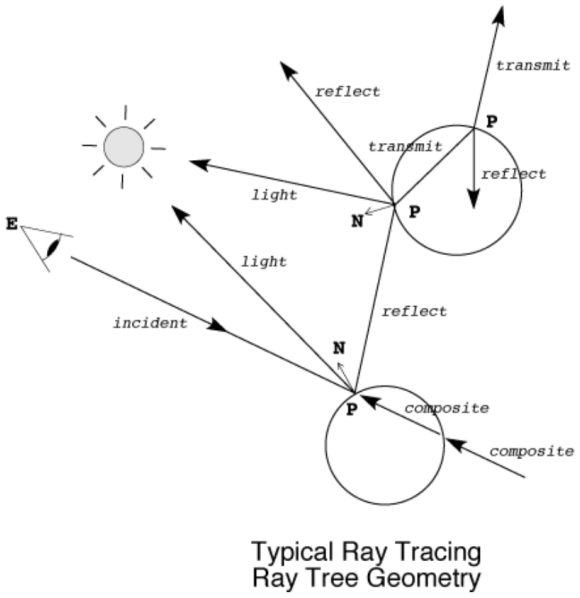

Figure 5: Ray-tracing Illustration.

Initially we implemented a GPU-based ray-tracing algorithm, which recursively trace reflective and refractive rays to a certain depth. To render the fluid, we need to know its surface. This is done by first generating a volume of the density field:

$$\rho(\mathbf{x}) = \sum_j m_j W(|\mathbf{x} - \mathbf{x}_j|) \tag{9}$$

Then defined a threshold, and define the implicit function $A(x) =$ threshold as the surface. When the surface is defined, we can emit a set of rays from the eye to each pixel in the output bitmap, and follow that ray to find the solution of the implicit function. The surface normal is defined as $\nabla A(x)$.

3.3.2 Multiple GPU Diffuse Rendering

The ray-tracing algorithm produces realistic results, however, it is very difficult to parallelize it into multiple GPUs, since ray-tracing for each pixel is done on a CUDA thread, it cannot access other memory, that means when the ray goes into another portion of the volume that lives in another GPU, we have to send the ray along with the tracing information to that

Figure 6: Ray-tracing rendered frame.

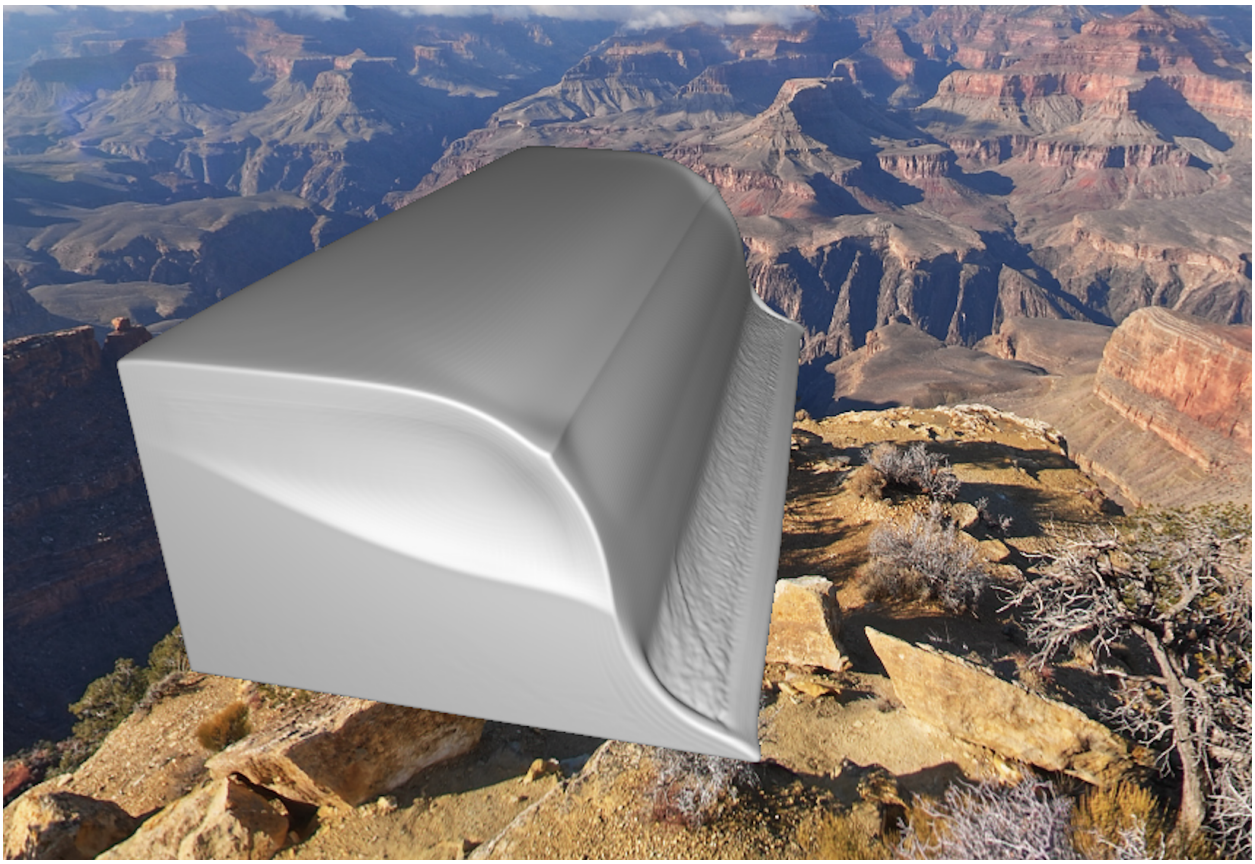

Figure 7: Diffuse color rendered frame.

GPU, which makes the implementation very difficult. Moreover, the algorithm is recursive, that means we have to send the recursion information (the stack) as well. In the future, we would like to try to make the ray-tracing distributed, that could be an interesting scheduling problem. One way to do that is first let to root node emit the rays, each ray maintains a call stack for the recursive rendering algorithm, and then send the rays to appropriate processors, each processor advance the ray inside its region, if the ray get outside, send it to the new processor, and each processor process the recursive rendering using the call stack in the ray, and eventually send the ray back to the root node to indicate that it is finished.

One other experience about this recursive algorithm is that we should try not to use recursive functions in CUDA, if we set ray-tracing depth to 3, then it is better to write 3 versions of the same function (using templates) than just recursively call the same function.

Instead, we implemented a simpler rendering algorithm what actually works on multiple GPUs. Instead of doing ray-tracing, we only render the diffusion color, which makes the fluid looks like milk (and without no reflection)

4 Experiments

In this section we present our experimental results.

4.1 Parameters

Parameters are crucial to the success of the simulation, if any one of ρ_0, k, μ is wrong by a magnitude, the simulation will be incorrect (usually we will see the fluid exploding). For all our tasks, we are using the same set of parameters: $\rho_0 = 1000, k = 2, \mu = 0.35$. The radius of the kernels are automatically set as proportional to the spacing of particles. In the initial lattice, each particle has around 20 neighbors.

Automatic calculation of particle spacing s and kernel radius r from particle mass m and resting density ρ_0.

$$s = 0.9 \left(\frac{m}{\rho_0} \right)^{1/3} \tag{10}$$

$$r = 1.74s \tag{11}$$

4.2 Environment

The computing environment is Triton, which has 3 GPU nodes, each has 4 GTX680s. Each GPU has 1536 CUDA cores, 4GB of global memory and 49152 bytes of shared memory.

4.3 Single GPU Scaling Analysis

First we did an experiment on a single GPU, the results are shown in Table 1, which is pretty good, the running time is very close to linear with respect to the number of particles.

This is because almost everything scales linearly according to the number of particles (except the QuickSort is $O(n \lg n)$, but the QuickSort does not take too much time in comparison to the Force computation).

Table 1: Experiment 1 Numeric Results

Particles	Time(ms)	MP/s
0.1	37.83	2.644
0.2	79.49	2.516
0.3	130.15	2.305
0.4	170.29	2.349
0.5	207.06	2.415
0.6	247.40	2.425
0.7	301.19	2.324
0.8	324.50	2.465
0.9	361.32	2.491
1	413.43	2.419
2	793.22	2.521
3	1,260.19	2.381
4	1,656.91	2.414

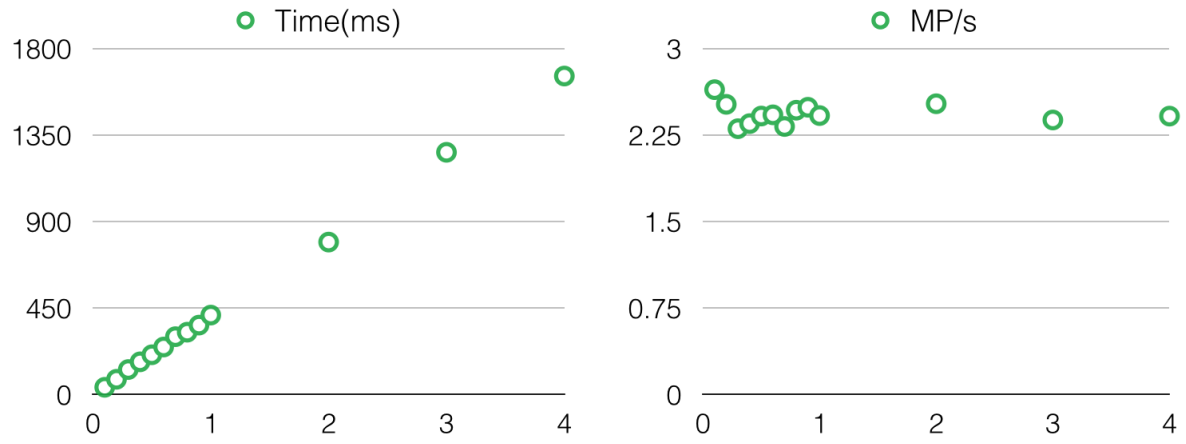

Figure 8: Single GPU Scaling Analysis

4.4 Multiple GPU Scaling Analysis

Important Note: The results are dramatically different from what presented in our presentation. At that time, we were not running the correct processes on the correct GPUs, for example, if we run 8 processes, the last four of them are running on the same GPUs as the first four, that is the reason why the scaling was not good. In the new experiments reported here, we fixed that problem, the results are really different. The reason is that Triton's qsub commond require us to set ppn as 3 times the number of GPUs, and then the nodefile generated by qsub also has that number of processes. For example, when we want to have 8 GPUs on 2 nodes, we write nodes=2,ppn=12, then the node file contains 24 lines, the first 12 in the first node, the second 12 in the second node, and the 8 processes mpirun lanuches will be all in the first node.

Table 2 and Figure 9 shows a run on 4 GPUs, with different number of particles (from 0.1M to 4M). The scaling is still close to linear, and the speed up is about 2.28 for 4 million particles. This is due to communication cost.

Table 2: Multiple GPU Scaling Analysis - Fixed 4 GPUs, Horizontal Axis: Millions of particles.

Particles	Time(ms)	MP/s	Single(ms)	Speedup
0.1	51.74	1.933	37.83	0.73
0.2	97.51	2.051	79.49	0.82
0.3	204.27	1.469	130.15	0.64
0.4	163.65	2.444	170.29	1.04
0.5	139.30	3.589	207.06	1.49
0.6	183.43	3.271	247.40	1.35
0.7	200.49	3.491	301.19	1.50
0.8	359.01	2.228	324.50	0.90
0.9	218.18	4.125	361.32	1.66
1	239.49	4.176	413.43	1.73
2	514.50	3.887	793.22	1.54
3	547.52	5.479	1,260.19	2.30
4	726.77	5.504	1,656.91	2.28

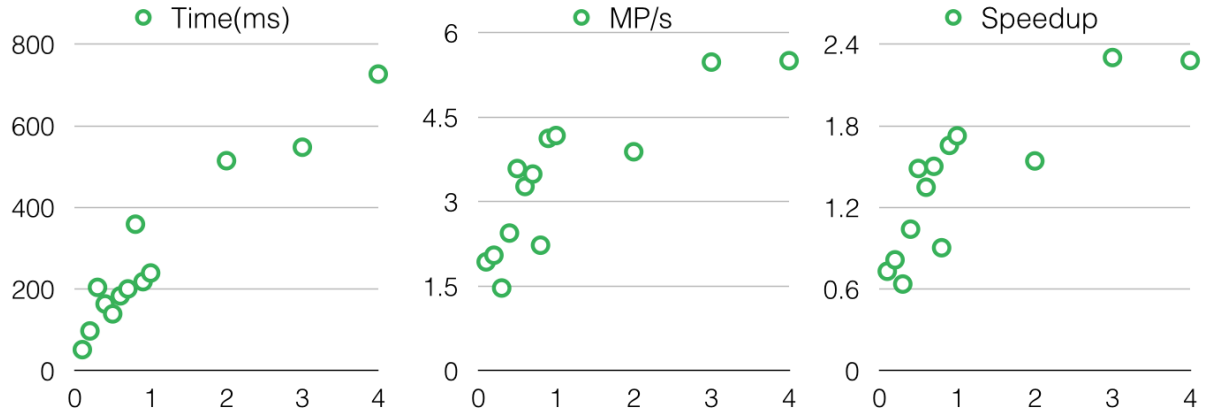

Figure 9: Multiple GPU Scaling Analysis - Fixed 4 GPUs Scaling Analysis

Table 3 and Figure 10 shows the results of a strong scaling analysis on 1 to 8 GPUs. In this experiment, we used the same amount of particles (8 million, $400 \times 200 \times 100$), but vary the number of GPUs. From the figure we can see the parallel efficiency slowly drops over the number of GPUs, but it is still very good. However, in this analysis, as the number of GPUs increase, the communication volume increases, but the communication between adjacent nodes are still the same (given that the water does not move, for our experiments, we only run a few frames, the water stays almost the same as the initial condition, where each processor get the same number of particles), which may cause more overhead depending on the networking infrastructure.

Table 4 and Figure 11 shows the results of a weak scaling analysis on 1 to 8 GPUs. In this analysis, the parallel efficiency is almost the same as the strong scaling analysis, because the GPUs takes constant amount of work now, and the communication cost is also constant. Both the strong scaling analysis and the weak scaling analysis shows roughly the same parallel efficiency.

Another observation is that the parallel efficiency drops from 1 to 2, and 2 to 3, then

Table 3: Multiple GPU Scaling Analysis - Strong Scaling Analysis, 1-8 GPUs

GPUs	Time(ms)	Exchange	Ghost	Parallel Efficiency
1	3313.76	482.96	13.98	1.00
2	2003.98	380.70	40.27	0.83
3	1353.78	270.90	44.95	0.82
4	1084.30	266.74	47.49	0.76
5	855.39	179.36	47.63	0.77
6	740.29	168.89	45.01	0.75
7	642.56	145.41	46.06	0.74
8	559.21	124.17	42.12	0.74

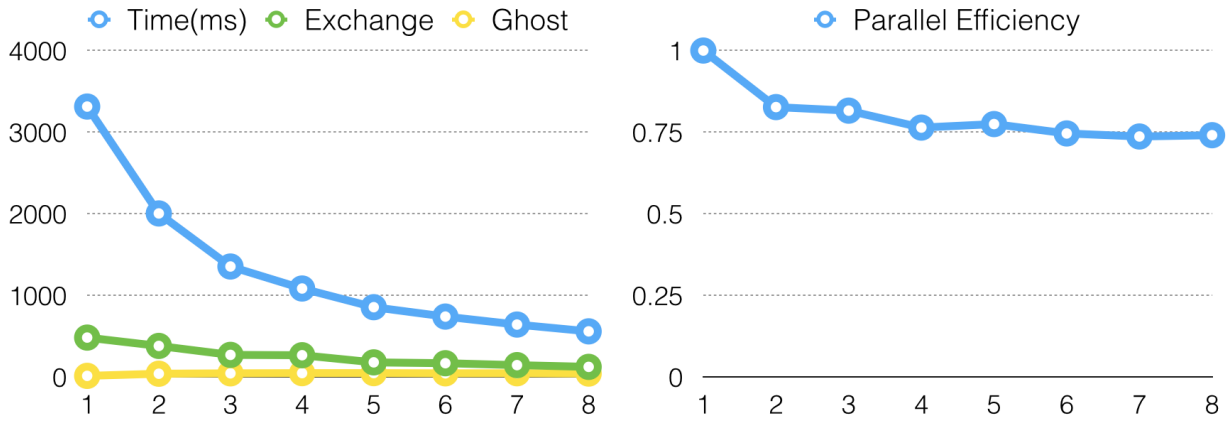

Figure 10: Multiple GPU Scaling Analysis - Strong Scaling Analysis

remain constant. This is because we use a even-to-odd and then odd-to-even communication approach (to avoid deadlock). When there are only 2 nodes, only even-to-odd is needed, so the parallel efficiency is better.

Table 4: Multiple GPU Scaling Analysis - Weak Scaling Analysis

GPUs	Running Time	Exchange	Ghost Particles	Parallel Efficiency
1	399.82	60.53	1.77	1.00
2	459.76	76.71	12.92	0.87
3	514.68	119.67	22.09	0.78
4	511.81	113.92	22.46	0.78
5	518.24	109.35	27.94	0.77
6	513.26	101.86	29.38	0.78
7	505.19	97.10	31.99	0.79
8	526.61	100.92	33.52	0.76

4.5 Large Run

We did a large run on Triton with 24 million particles, on 4 GPUs. The particles are divided evenly, so each GPU gets aroudn 6 million particles. The running time for each simulation frame is 3315 ms.

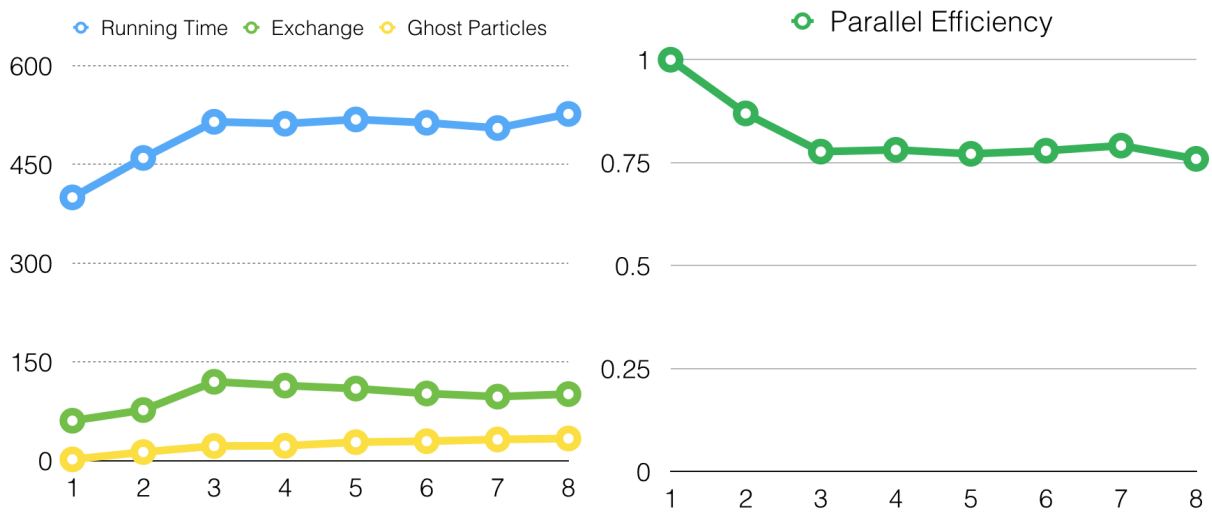

Figure 11: Multiple GPU Scaling Analysis - Weak Scaling Analysis, Horizontal Axis: Number of GPUs and millions of particles

5 The Code

You can find the code in github: `https://github.com/windwish/CUDASPHFluid`.

Here we show how to use the code. There aren't any other dependencies except CUDA, MPI and a C++ compiler. However, for different systems we do need to tweak some compiling options to link CUDA and MPI together. You can find them in our makefile.

To compile on a Mac computer, first install CUDA SDK, and then install MPI (we are using open-mpi provided by homebrew). After that, you should be able to run make in the source directory and get three executables generated: "interface", "mpi_project", "project". The "interface" is the interactive simulation frontend, it uses one GPU. The "project" and "mpi_project" is a commandline application that takes a "profile.txt" and execute the simulation according to the options. "mpi_project" is the multiple GPU executable, while "project" only uses a single GPU (the version before adding MPI support).

The "profile.txt" is used to define simulation parameters. See the one in the repository for an example.

6 Conclusion

In this project, we implemented a SPH algorithm for fluid simulation in multiple GPUs with CUDA. The experiemntal results suggest that the solver scales linearly on a single GPU, and the parallel efficiency for multiple GPUs is constant around 0.75.

In the future, we would like to implement a dynamic boundary adjustment, and different ways (such as 2D or 3D grid) to distribute the particles into multiple nodes. For the simulation part, we would like to incorporate incompressibility constrains (in the video, you can observe that the volume of the water is not constant, that makes the simulation looks a bit unrealistic), and introduce surface tension.