Ripple Simulation Based on OpenGL

Parallel Fluid Mechanics Simulation with OpenMP and CUDA

Yu-Hsun Yuan
Department of Computer Science
National Yang Ming Chiao Tung University, Taiwan
steven112163.cs09@nycu.edu.tw

Han-Chun Chen
Department of Electronics Engineering
National Yang Ming Chiao Tung University, Taiwan
hanz1211.ee09@nycu.edu.tw

Yuan-Hsuan, Wen
Department of Electronics Engineering
National Yang Ming Chiao Tung University, Taiwan
jpm06v0.ee07@nctu.edu.tw

ABSTRACT

The project's goal is to improve ripple simulation on a personal workstation; a model based on thread parallelism is proposed. First, the data structures and rules for data operation are established to meet the needs of the array model. Second, the physical simulation governing ripple motion is transformed discretely for height calculation. Finally, the simulation of ripple is achieved from the heightmap providing normal information used by the calculation of light reflection and refraction in real-time. Experiment results show that the method is robust and efficient to achieve real-time ripple simulation by fully utilizing the excellent implementation of parallelism in hardware using a separate flow of control for each worker.

1 INTRODUCTION

In computer graphics, the simulation of natural scenes is one of the major research topics. As one of the most common natural materials, the realistic representation of water significantly impacts the simulation of natural scenes. It is also crucial in computer games and virtual reality applications to immerse players into virtual worlds, constructed by photorealistic simulations of natural scenes, such as ripple smoke.

A ripple effect occurs when an initial disturbance propagates due to initial vibration to disturb an increasingly larger portion of the system. This phenomenon results from the system trying to spread out the energy to go back to the initial position.

When water is stationary, the pressure on each point of the water surface is balanced, and all the points are in a balanced position. Thus, if a force acts upon one point, it will give the water some energy. Besides, it will produce the acceleration and deviate from the balance position, producing vibration in the position; this disturbance can affect the neighboring points the ripple effect.

2 PROPOSED SOLUTION

In this project, a parallel model to improve the ripple simulation can be defined as a center particle simulation system and isocontour simulation around it.

The droplet simulation can be achieved with a particle system. The droplet is a sphere. The acceleration, velocity, and location will be calculated according to the physics laws in the real world. Once the droplet reaches the water surface, it will trigger the isocontour simulation.

The isocontour can be seen as a circle with a damped cosine amplitude extended outward on a plane of vertices. Each vertex will change its z-coordinate based on its neighbors and a damp coefficient in world coordinates.

Since there are two arrays for storing the current state and next state of the vertices, we can parallelize the computation based on their index in the current state and then aggregate the result as their y-coordinates in the next state.

Once all vertices' world coordinates are decided, we calculate all polygons and normal vectors for the shading process. This process, again, is a repetitive instruction that has no data dependency, thus can be parallelized with CPU (OpenMP) or GPU (CUDA).

2.1 Height Calculation

We use two arrays of vertices to simulate the ripple effect to calculate the new height in the next frame from the old heights in the current frame. One array represents the current state, and the other represents the next state. Each time the ripple function is called, it aggregates the heights of eight neighbors for each vertex in the current state to compute the new height. The new height will be stored in

the next state. By this means, we can simulate the ripple like a damped cosine wave.

2.2 Data Structures

To make shaders in OpenGL operate correctly, all vertices and normals will be arrays of floating-point numbers.

2.1.1 Sphere The proposed sphere can be seen from the below structure. It consists of an array of vertices and an array of normals. The sphere is constructed by subdivision from a tetrahedron. Each triangle in the tetrahedron is divided into four triangles, and the level of subdivision in our model is 6. Once the subdivision reaches the end, it will place the vertices and normals of the triangle into the vectors. Because the tetrahedron is centered at the origin in the object coordinate, the normal vector of each point is the same as its normalized coordinate.

```
Class Surface {
        VECTOR<FLOAT> vertices;
        VECTOR<FLOAT> normals;
};
```

2.1.1 Surface The water surface can be seen from the below structure. It is a plane in 3D containing a set of surface_size * surface_size connected vertices. Two arrays contain the vertices. One contains the vertices in the current state, and the other contains the vertices in the next state. Each time the ripple function is called, it calculates the new height of each vertex from the current state and stores the results into the next state. After updating vertices, it will flip the state to use the updated next state as the current state to simulate the ripple effect. After flipping the state, it uses the vertices in the current state to update the normals.

```
Class Surface {
        FLOAT *vertices[2];
        FLOAT *normals;
        INT surface_size;
        INT vertices_size;
};
```

2.3 Acceleration and Velocity

In the simulation, we need to calculate the acceleration and velocity of the droplet to determine its location. Therefore, there are a variable acceleration and a variable velocity to represent acceleration and velocity.

2.4 System Architecture

Fig. 1 shows the system architecture. The system checks whether the droplet reaches the water surface when the clock ticks. The system will update the vertices on the water surface if it reaches the surface. On the other hand, if the droplet is still in the air, the system will only update the information of the droplet. As aforementioned, the updating process can be parallelized. After updating, OpenGL will render all vertices in our simulation.

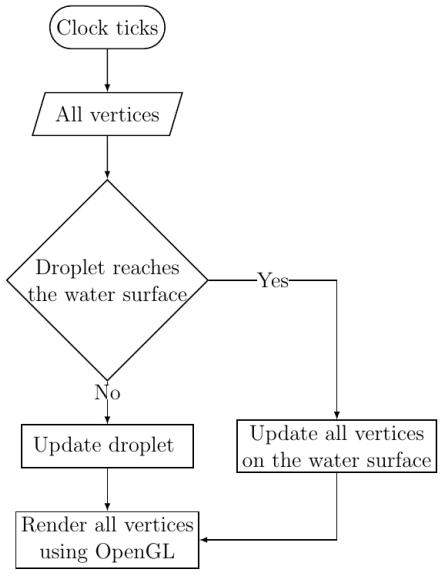

Figure 1: System architecture

2.5 Language Selection

2.5.1 OpenMP. OpenMP is easy to use and consists of only two basic constructs: pragmas and runtime routines. The OpenMP pragmas typically direct the compiler to parallelize sections of code. All OpenMP pragmas begin with #pragma omp. As with any pragma, compilers ignore directives that do not support the feature—in this case, OpenMP.

2.5.2 CUDA. Working with CUDA means that the same code can be efficiently parallelized on the GPU, accelerating the code without data dependency.

3 EXPERIMENTAL METHODOLOGY

3.1 Environment

Based on the proposed method, an application implemented with C++ computer program, with evaluation platform table shown below:

CPU	Intel Core i7-9700(8 cores, 8 threads)
GPU	NVIDIA GeForce GTX 1650 SUPER
RAM	32 GB DDR4 2666MHz
OS	Ubuntu 20.04 LTS
CUDA	v11.5

Table 1: Evaluation platform

3.2 Input Sets

The experiment is measured in 100 iterations (100 frames rendered) unless specified and then records the average/minimum time required to update the mesh data.

The primary input variable is the mesh size (referred to as surface size in the experiment).

For OpenMP experiments, the input settings are 400, 800, 1600. Note the actual data is proportional to the square of this input argument. Also, the thread number is from 1, 2, 4, 6, 8, given the evaluation platform restriction.

For CUDA experiments, the mesh size is extended to measure the program's data capability and throughput, including surface sizes of 400, 800, 1600, 3200, 6400.

4 EXPERIMENTAL RESULTS

4.1 OpenMP Results

Table. 2 and Fig. 2 are the best speedup results, based on the different number of OpenMP threads vs. surface size, measured in 100 iterations and excluding the rendering process.

The result shows this application with multithreading leads to poor scalability w.r.t. data size.

Speedup	400x400	800x800	1600x1600
1 Thread	0.72265	0.62348	0.55210
2 Threads	1.48843	1.21000	1.06258
4 Threads	2.72197	2.28209	2.04213
6 Threads	4.41532	3.17041	2.88578
8 Threads	4.84507	3.94046	3.28829

Table 2: Best speedup

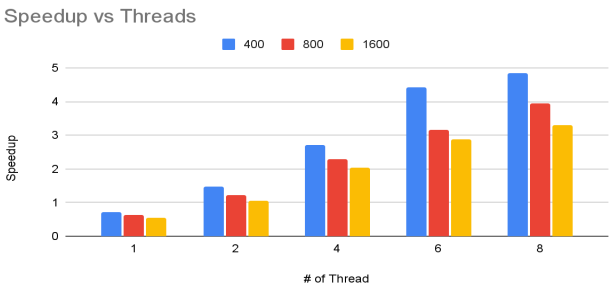

Figure 2: Speedup with different surface sizes and number of threads

4.2 OpenMP Results (Collapse for loop)

The surface is a 2D data structure computed with a nested for loop. We can utilize the OpenMP "collapse" clause during the implementation to flatten the loop, which gives a much better performance than the original nested for loop. The difference becomes larger as the data size increases.

Fig. 3 and 4 are the speedup comparison between collapsed for and native parallel for loop, measured in 100 iterations, average execution time.

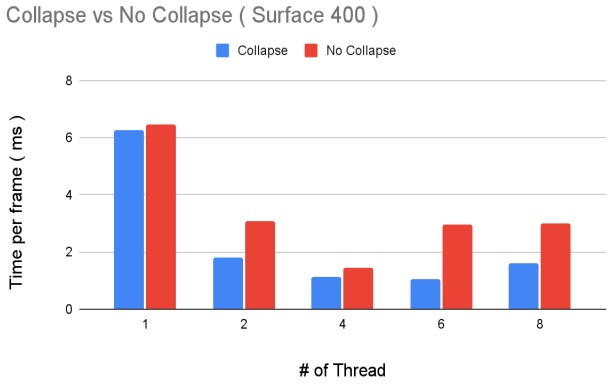

Figure 3: Comparison with surface size 400

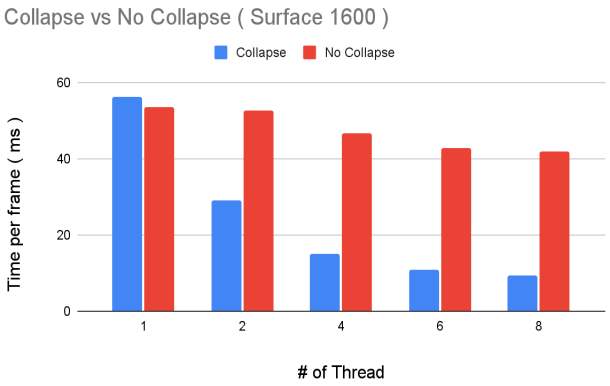

Figure 4: Comparison with surface size 1600

4.3 OpenMP Performance Pitfall

Suppose we include the time required for rendering. In that case, the speedup will be bound at 2~3x as shown in Fig. 5, which is due to the parallelable part of the program being about 43.63% based on profiling results from gprof, the OpenGL mathematics are single-threaded.

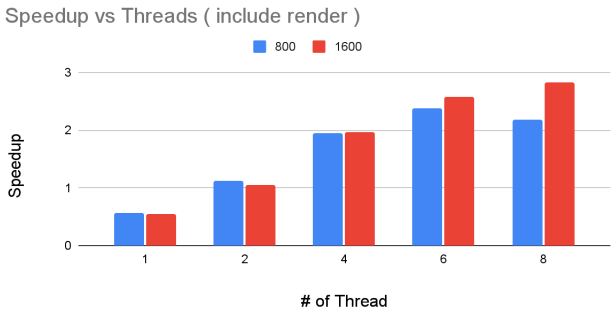

Figure 5: Speedup with render

In Fig. 6, another performance pitfall happened when using a small surface size (400x400). The 8 threads version is slower than the 6 threads version since the OpenGL window and rendering worker requires a dedicated thread, leading to context switching during every iteration.

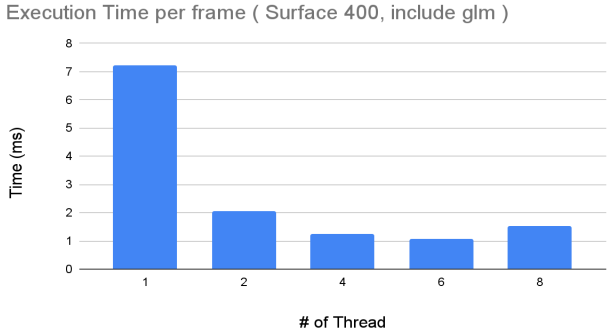

Figure 6: Execution time with different numbers of threads

4.4 CUDA Results

The CUDA implementation can deal with large data throughput. Hence the experiments are executed with surface size from 400x400 to 6400x6400.

Table. 3 and Fig. 7 below show as the surface size increases, the speedup becomes more significant. Measured in 100 iterations, with block dimensions of 25x25 and 8 CUDA streams.

Speedup	Avg	Min	Max
400x400	1.1578	1.1587	0.2689
800x800	1.5039	2.1537	0.3506
1600x1600	2.4585	2.7692	0.5566
3200x3200	3.1782	3.9494	1.2600
6400x6400	5.2789	5.5041	1.9071

Table 3: Speedup with different surface sizes

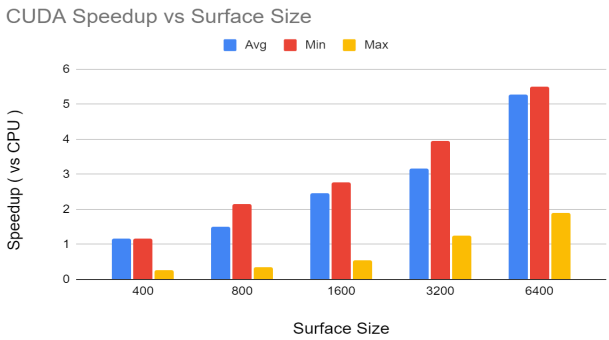

Figure 7: Speedup with different surface sizes

4.5 CUDA Performance Pitfall

During the experiment, while using different numbers of CUDA streams, there is a sudden performance drop when using 2 streams, like Fig. 8 shown below:

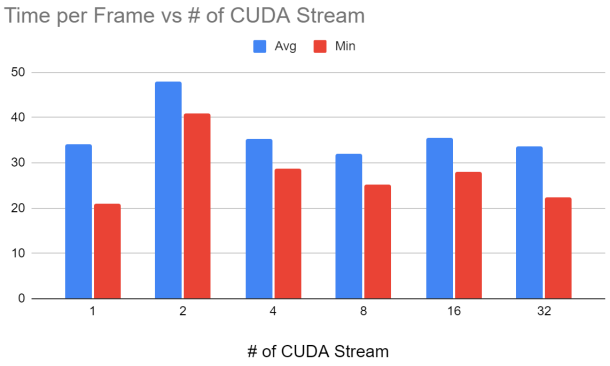

Figure. 8: Time per frame with different numbers of CUDA streams

A possible explanation is because of the workload distribution. Since there are 3 different cases when calculating the surface: inner vertex, edge vertex, corner vertex; each requires a different formula, which leads to branching inside kernel function.

From Fig. 9 shown above, under the 2 streams workload distribution, both kernels will encounter 3 types of branches, which is the same as only 1 stream, but with extra synchronization costs. Thus it results in a performance drop.

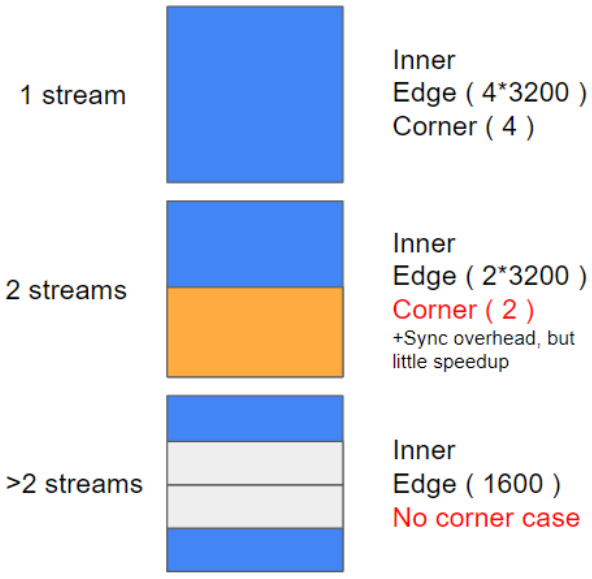

Figure 9: Workload with different numbers of streams

5 RELATED WORK

Water simulation is always challenging as water has dynamic and changeful surfaces, and the lighting effects over water surfaces are complex.

In recent years, with developing computer graphics, many models have attempted to propose different approaches in water simulation. Some methods compute the water animation based on physical models, such as Foster [1-2] developed a 3D Navier-Stokes methodology for the realistic animation of liquids by finite difference approach to obtain the velocity field and pressure field of water. The method produces the height field of the wave, achieving realistic simulation results; Kass [3] used a simplified numerical method to solve the Navier-Stokes equation for water simulation. A pressure-defined height field formulation was used by Chen[4] in fluid simulations with moving obstacles. O'Brien[5] simulated splashing liquids by combining a particle system and height field, while Miller[6] used viscous springs between particles to achieve dynamic flow in 3D. Terzopoulos[7] simulated melting deformable solids using a molecular dynamics approach to simulate the particles in the liquid phase.

6 CONCLUSIONS

In this project, a novel thread parallelism-based model is proposed to improve the water surface simulation. The data structures and rules for data operation are established to meet lockless and the needs of the parallelism requirement. Experimental results show the robustness and efficiency of the proposed method for the real-time simulation of water surface on implementing parallelism using a separate flow of control for each worker.

The code in the project can be made more efficient by moving the rendering of the water wave part of the code to shaders. Doing this, for this algorithm, may necessitate the use of geometry or tessellation shaders. The future work also includes developing other forms of water simulation such as a water fountain, a glass of water, droplets of water on a glass surface, water waves inside a tank. Developing these different forms of water simulation involves deeper analysis of particular mechanisms or working on different concepts of OpenGL, which makes each potential project challenging and interesting to work on.

7 REFERENCES

[1] Foster, Nick, and Dimitri Metaxas. "Realistic animation of liquids." Graphical models and image processing 58.5 (1996): 471-483.

[2] Foster, Nick, and Ronald Fedkiw. "Practical animation of liquids." Proceedings of the 28th annual conference on Computer graphics and interactive techniques. 2001.
[3] Kass, Michael, and Gavin Miller. "Rapid, stable fluid dynamics for computer graphics." Proceedings of the 17th annual conference on Computer graphics and interactive techniques. 1990
[4] Chen, J., Lobo, N.: Toward interactive-rate simulation of fluids with moving obstacles using the Navier-Stokes Equations. Graphical Models and Image Processing 57, 107–116 (1994)
[5] O'Brien, J., Hodgins, J.: Dynamic simulation of splashing fluids. In: Proc. Computer Animation, pp. 198–205 (1995)
[6] Miller, G., Pearce, A.: Globular dynamics: a connected particle system for animating viscous fluids. Computers and Graphics 13, 305–309 (1989)
[7] Terzopoulos, D., Platt, J., Fleischer, K.: Heating and melting deformable models (from goop to glop). In: Proc. Graphics Interface, pp. 219–226 (1989)