Ripple Simulation Based on OpenGL

Parallel Programming Course Project, Team 10

Participants: 袁鈺勛、陳翰群、溫圓萱（已退選）

Outline

- Introduction
- Data Structure
- Experimentation
- Conclusions

Outline

- Introduction
- Data Structure
- Experimentation
- Conclusions

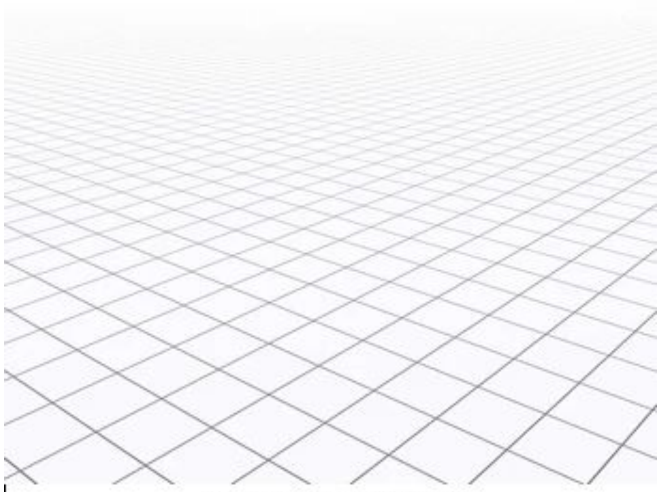
Ripple Simulation

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 so it can go back to the initial position.

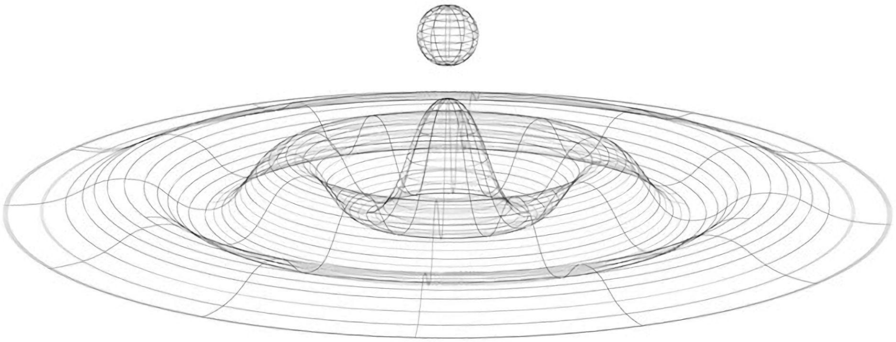
Water Surface Definition

The water surface is a plane in the 3D space, composed of a set of surface_size * surface_size connected vertices, where the vertex has 3 identifiers x, y, and z.



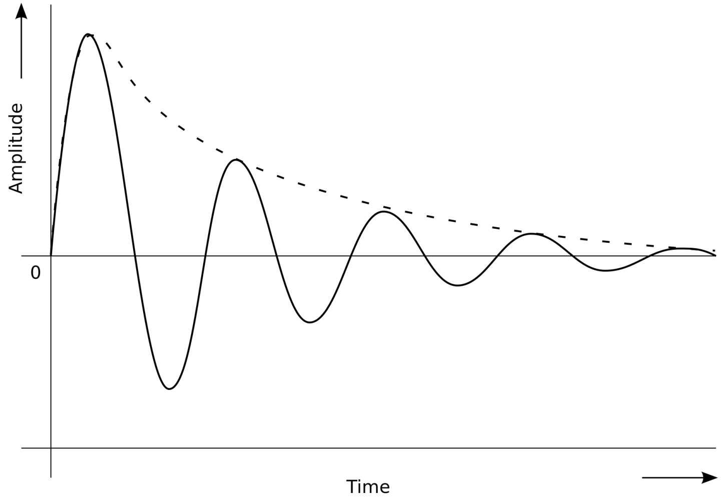
Droplet Simulation

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.

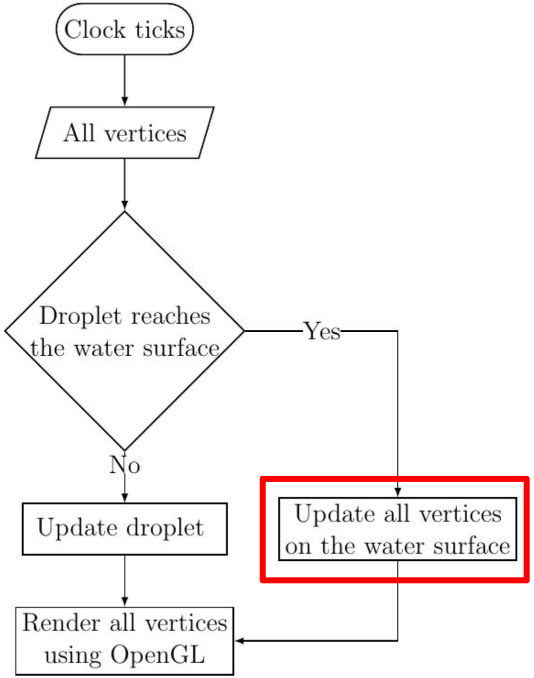


Height Calculation

The isocontour can be seen as a circle with a damped sine amplitude extended outward on a plane of vertices. In world coordinates, each vertex will change its y-coordinate based on its neighbors and a damp coefficient.



System Architecture



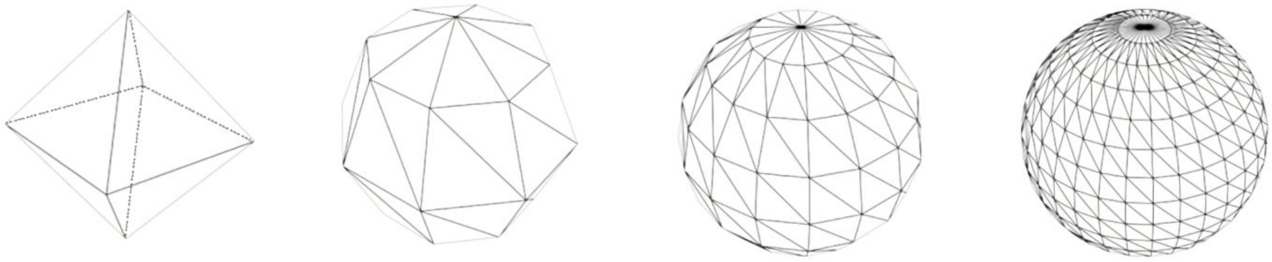
When the clock ticks, the system checks whether the droplet reaches the water surface. The system will update the vertices on the water surface if it reaches the surface.

Outline

- Introduction
- Data Structure
- Experimentation
- Conclusions

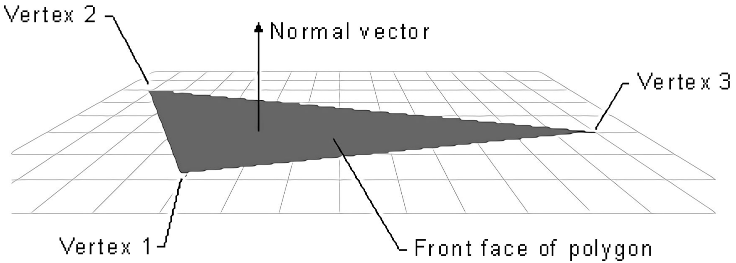
Sphere

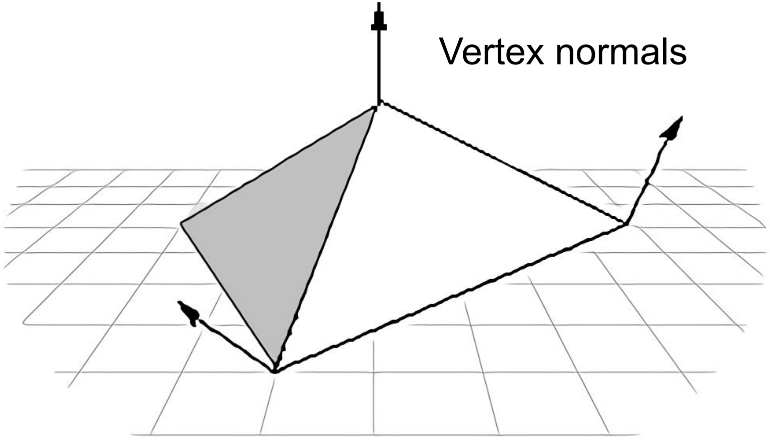
The proposed sphere consists of an array of vertices and an array of normals. The sphere is constructed by subdivision from a tetrahedron. Each triangle in the tetrahedron is divided into four triangles, and the level of subdivision in our model is 6.



Normal Vector of the Sphere

Because the tetrahedron is centered at the origin of the object coordinate, the normal vector of each point is the same as its normalized coordinate.





Water Surface

There are two arrays containing the vertices. One contains the vertices in the current state, and the other contains the vertices in the next state.

Current:	x1	y1	z1	x2	y2	z2	…

Next:	x1	y1	z1	x2	y2	z2	…

Proposed Methods

- OpenMP
- CUDA

Ripple Simulation

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.

0	0	0
0	1	0
0	0	0

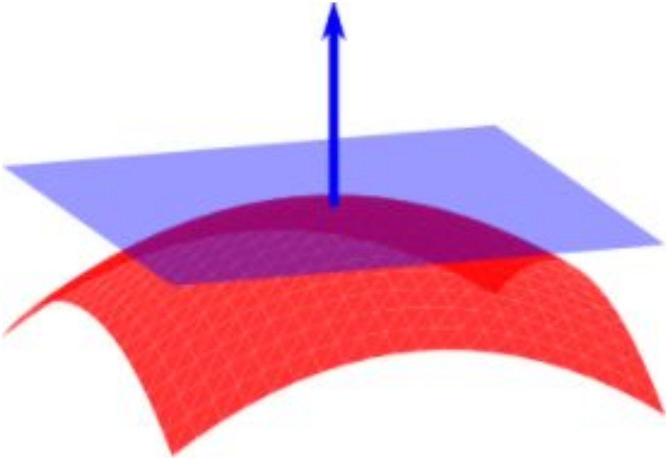
0.4	0.4	0.4
0.4	0.8	0.4
0.4	0.4	0.4

Ripple Simulation (cont.)

```
for (int z = 0; z < surface_size; z++)
    for (int x = 0; x < surface_size; x++) {
        new_y = (sum of heights of neighbors in the current state) / coeff;
        new_y -= height at (z, x) in the next state;
        new_y -= new_y / damp;
        height at (z, x) in the next state = new_y;
    }
```

Ripple Simulation (cont.)

After flipping state, it uses the vertices in the current state to update the normals.



Ripple Simulation (cont.)

```
for (int z = 0; z < surface_size; z++)
    for (int x = 0; x < surface_size; x++) {
        neg_z = negative z direction from (z, x);
        pos_z = positive z direction from (z, x);
        neg_x = negative x direction from (z, x);
        pos_x = positive x direction from (z, x);
        new_normal = normalize(cross(neg_z, neg_x) +
                               cross(neg_x, pos_z) +
                               cross(pos_z, pos_x) +
                               cross(pos_x, neg_z));
    }
```

Outline

- Introduction
- Data Structure
- Experimentation
- Conclusions

Experiment

Environment Settings

CPU: Intel Core i7-9700 (8 cores, 8 threads)

GPU: Nvidia GeForce GTX 1650 SUPER (4 GB GDDR6)

RAM: 32 GB DDR4 2666MHz

OS: Ubuntu 20.04

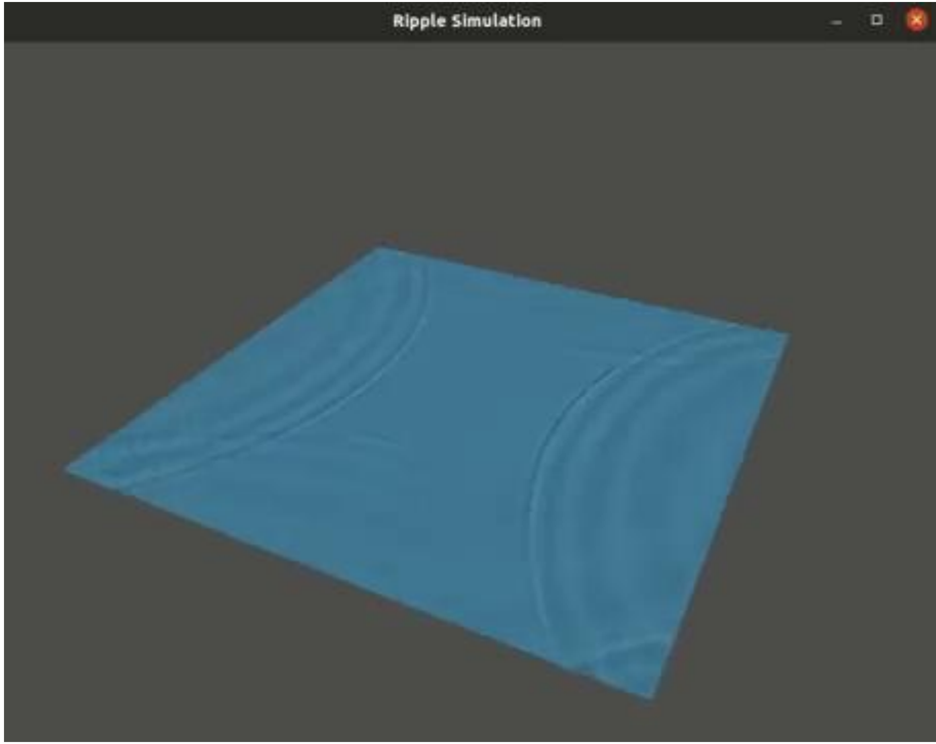
CUDA: 11.5

Experiment

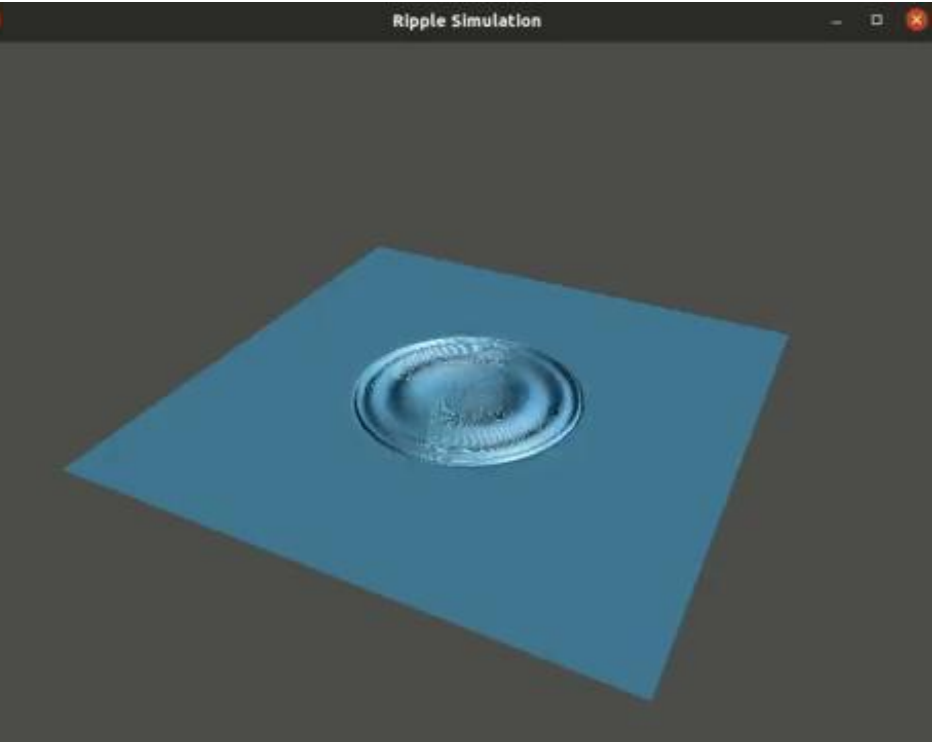
Experiment Variables

- OpenMP
  - # of Threads: 1, 2, 4, 6, 8
  - Surface Size: 400, 800, 1600
  - Iteration: 100, 200
- CUDA
  - Surface Size: 400, 800, 1600, 3200, 6400
  - Block Dim: 25, 50, 100, 200
  - # of cudaStream: 1, 2, 4, 8

Experiment

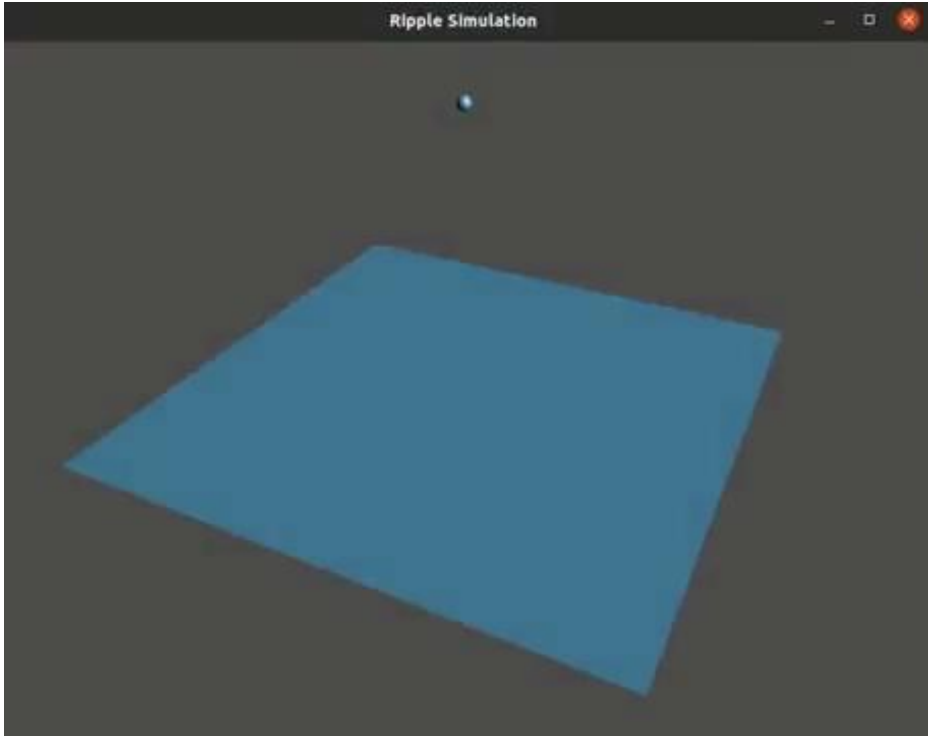


Surface Size 400

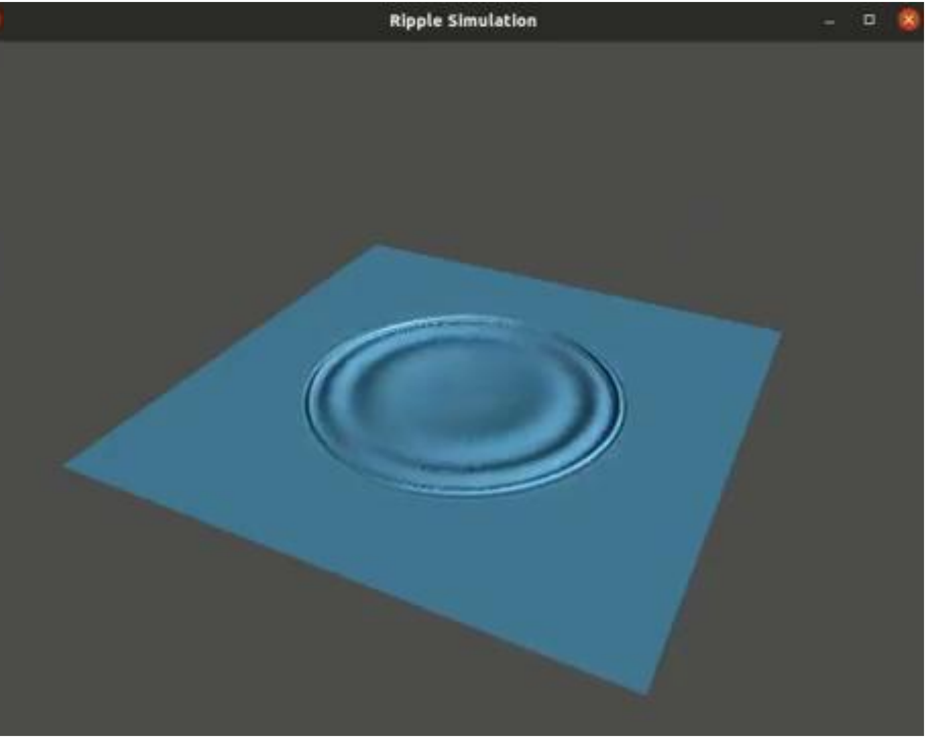


Surface Size 1600

Experiment



1 Thread



4 Threads

Experiment

OpenMP Implementation

Update Surface

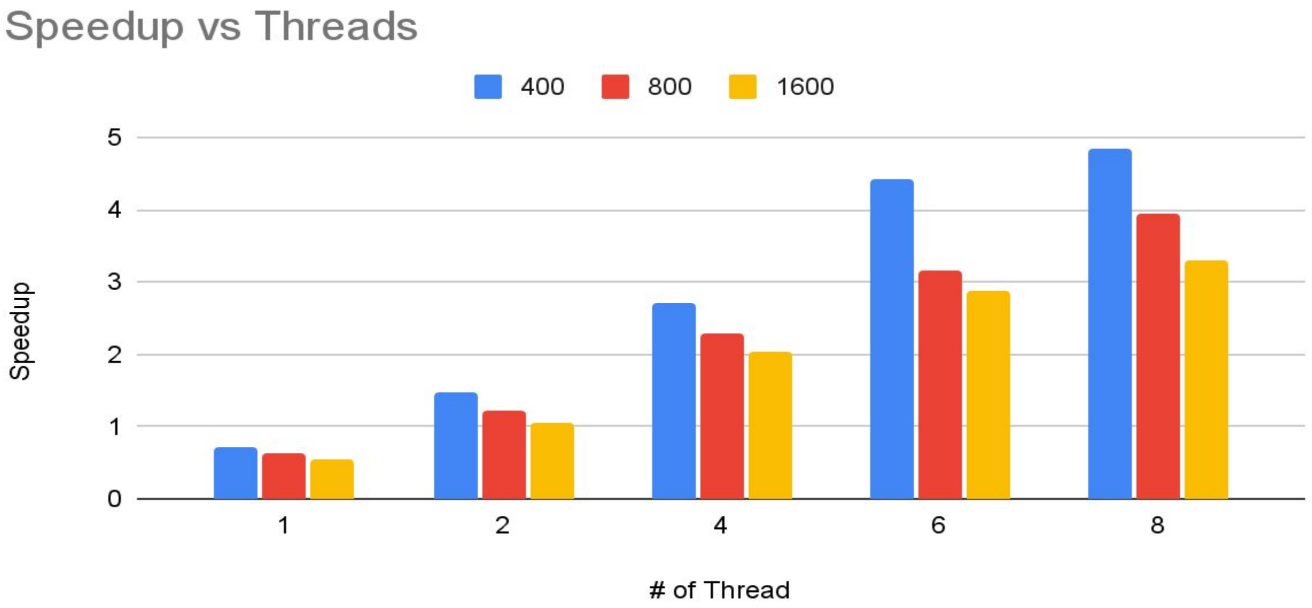
```
#pragma omp parallel
    {
#pragma omp for collapse (2)
        for (x = 1; x < surface_size - 1; x++) {...}

        // Edge vertices
#pragma omp for
        for (x = 1; x < surface_size - 1; x++) {...}

#pragma omp for
        for (z = 1; z < surface_size - 1; z++) {...}

        // Corner vertices
#pragma omp sections
        {
#pragma omp section {...}
#pragma omp section {...}
#pragma omp section {...}
#pragma omp section {...}
        }
    }
```

Synchronization

Update Normals

```
    // Change current state
    state = 1 - state;

#pragma omp parallel
    {
        // Update normals
#pragma omp for collapse(2)
        for (z = 1; z < surface_size - 1; z++) {...}

        // Edge normals
#pragma omp for
        for (x = 1; x < surface_size - 1; x++) {...}

#pragma omp for
        for (z = 1; z < surface_size - 1; z++) {...}

        z = surface_size - 1;
#pragma omp for
        for (x = 1; x < surface_size - 1; x++) {...}

        x = surface_size - 1;
#pragma omp for
        for (z = 1; z < surface_size - 1; z++) {...}

        // 4 Corner normals, 4 sections
#pragma omp sections {...}
    }
}
```

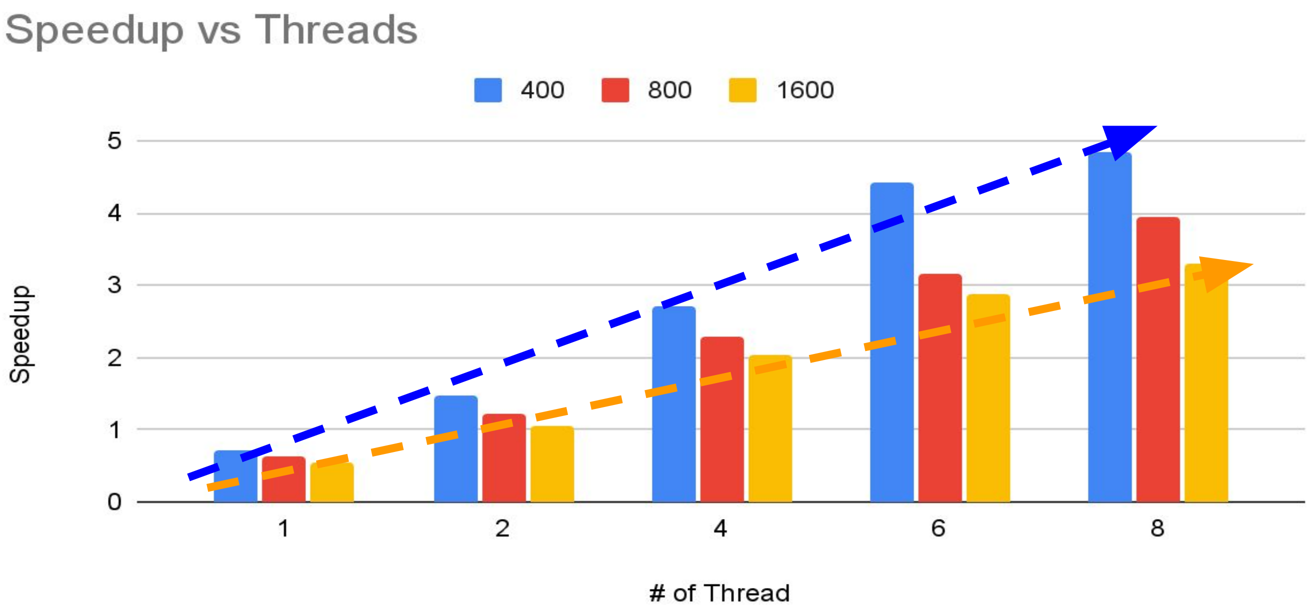
Experiment: OpenMP



Measured in 100 iterations, best speedup per frame

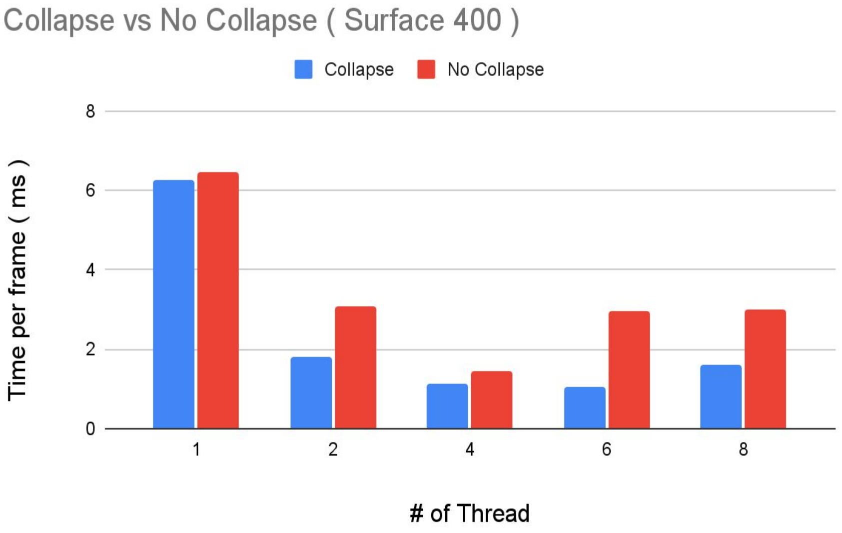
Note that the Time Complixity is O(S^2)

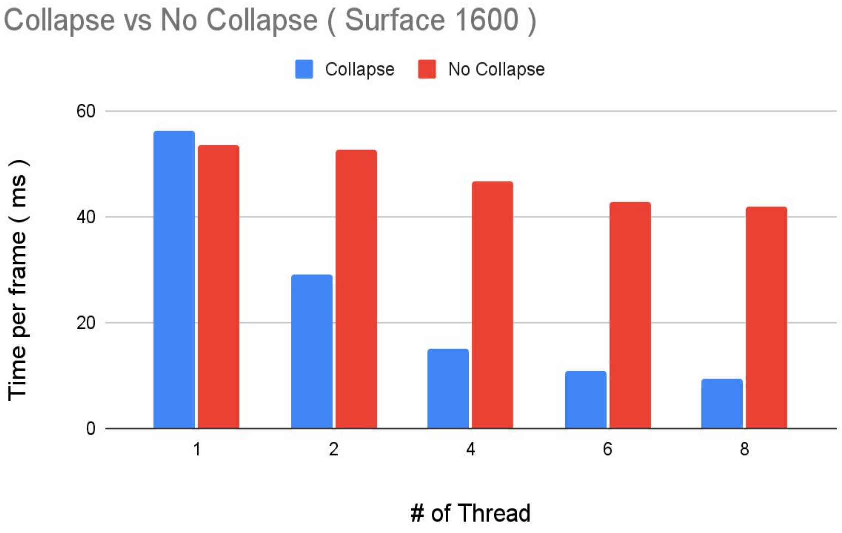
Experiment: OpenMP (cont.)



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

Experiment: OpenMP (for collapse)

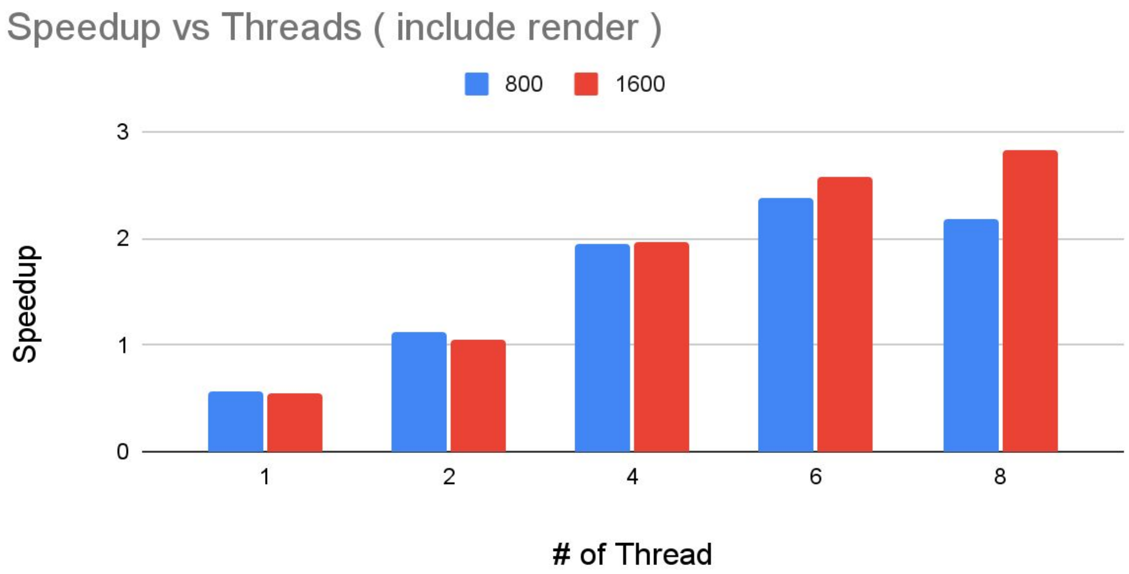




Measured in 100 iterations, average execution time per frame

This shows the benefit of collapse for loop during mesh computation

Experiment: OpenMP (OpenGL performance pitfall)



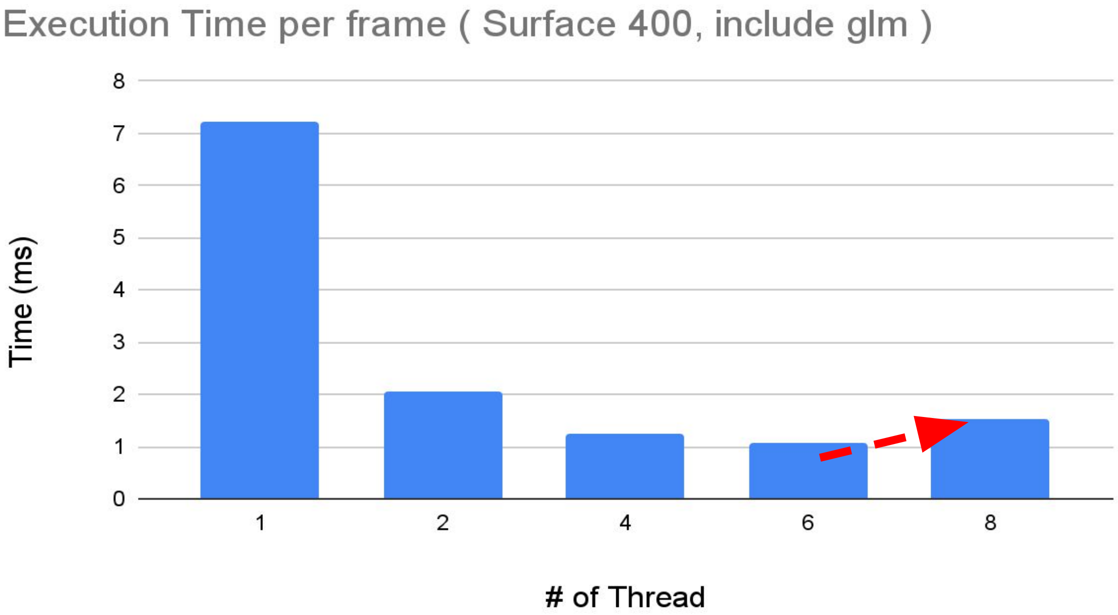
Measured in 200 iterations, average speedup per iteration

Experiment: OpenMP (OpenGL performance pitfall)

gprof shows 56.37% of program is running OpenGL Mathematics

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 43.63      2.43     2.43      100    24.30    54.88  ripple_omp(Surface*, int&)
 12.57      3.13     0.70  8606493     0.00     0.00  glm::detail::compute_cross<float, (glm::qualifier)0, false>
  9.87      3.68     0.55  9277987     0.00     0.00  glm::vec<3, float, (glm::qualifier)0> glm::operator-<float,
  7.18      4.08     0.40 67281668     0.00     0.00  std::vector<float, std::allocator<float> >::operator[](unsi
  7.09      4.48     0.40  6336606     0.00     0.00  glm::vec<3, float, (glm::qualifier)0> glm::operator+<float,
  6.46      4.84     0.36 41015456     0.00     0.00  glm::vec<3, float, (glm::qualifier)0>::vec(float, float, fl
  3.41      5.03     0.19  8379948     0.00     0.00  glm::vec<3, float, (glm::qualifier)0> glm::cross<float, (gl
  3.23      5.21     0.18  2338992     0.00     0.00  glm::vec<3, float, (glm::qualifier)0> glm::operator*<float,
  2.33      5.34     0.13  2367126     0.00     0.00  glm::vec<3, float, (glm::qualifier)0> glm::operator*<float,
  1.44      5.42     0.08  2185603     0.00     0.00  glm::detail::compute_normalize<3, float, (glm::qualifier)0,
  0.90      5.47     0.05  2334716     0.00     0.00  glm::detail::compute_dot<glm::vec<3, float, (glm::qualifier
  0.72      5.51     0.04      204     0.20     0.20  std::vector<float, std::allocator<float> >::data() const
```

Experiment: OpenMP (OpenGL performance pitfall)

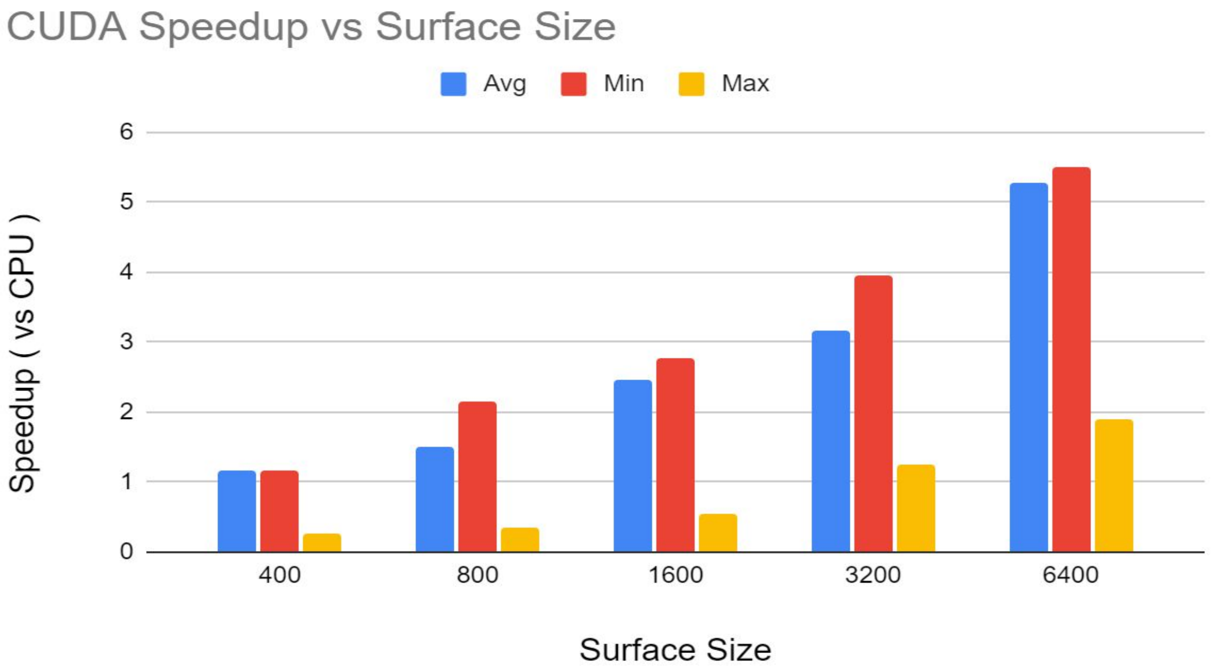


In early implementation, execution time increase if the # of thread == # of CPU cores,

OpenGL Mathematics (glm::cross, etc) requires a dedicated processor

This will result in context switching from frame to frame

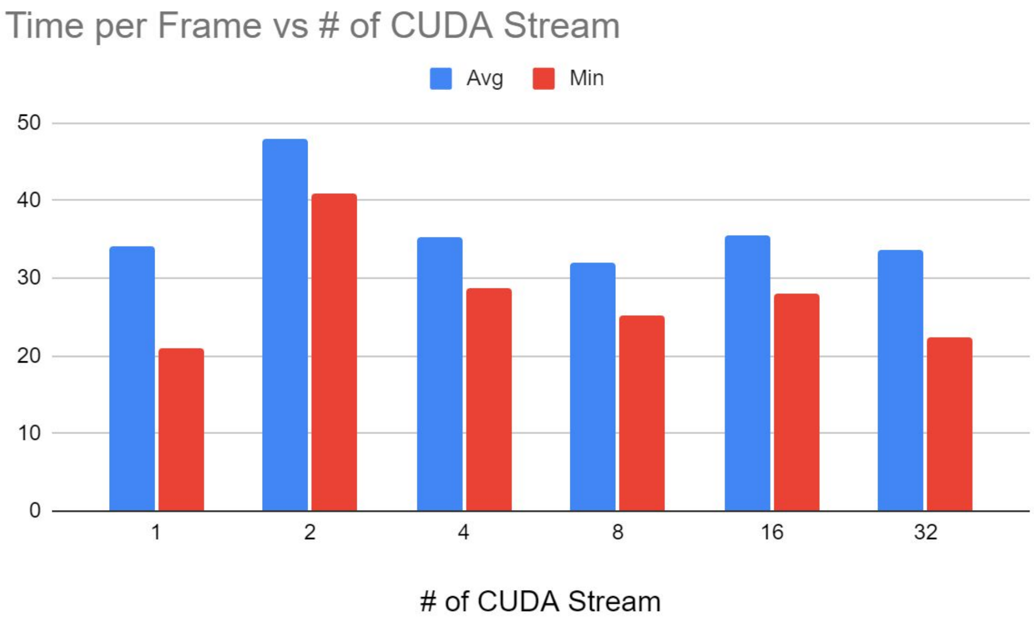
Experiment: CUDA (Surface)



Measured in 100 iterations, only parallel parts

Block Dim 25x25, with 8 CUDA streams

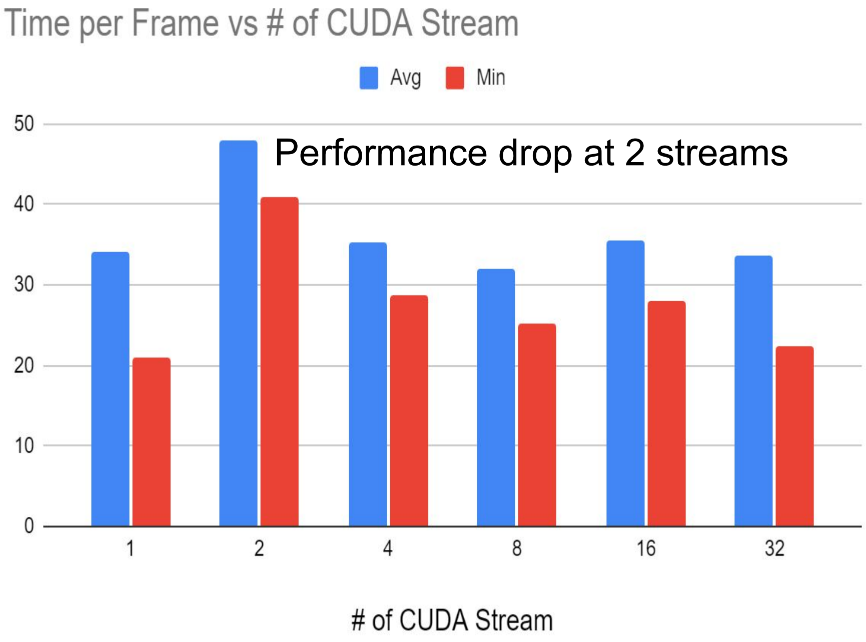
Experiment: CUDA (Stream)

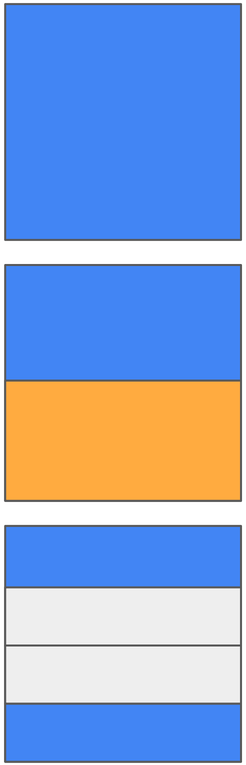


Measured in 100 iterations, only parallel parts

Block Dim 25x25, with 8 CUDA streams

Experiment: CUDA (Branching Pitfall)





1 stream

Inner
Edge (4*3200)
Corner (4)

2 streams

Inner
Edge (2*3200)
Corner (2)
+Sync overhead, but little speedup

>2 streams

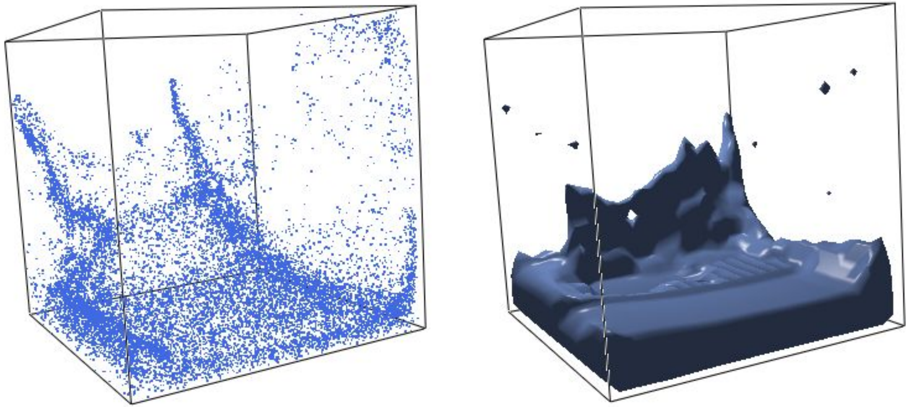
Inner
Edge (1600)
No corner case

Outline

- Introduction
- Data Structure
- Experimentation
- Conclusions

Conclusions

In this project, a novel thread parallelism-based model is proposed to improve the simulation of the water surface. The future work also includes developing other forms of water simulation such as water fountain, a glass of water, droplets of water on a glass surface, water waves inside a tank.



source: 3D Real-Time Fluid Simulation by a1ex90
https://github.com/a1ex90/Fluids3D.git

Q & A