Asychronous Smoothed Particle Hydrodynamics

Andrew Pregent

1 Abstract

A new method for parallelizing asynchronous Smooth-Particle Hydrodynamics is presented, extending the single-threaded and multi-core asychronous SPH algorithms of Reinhardt et al.[18] to the GPU. The method will then be evaluated for performance against single and multi-core implementations, and further directions for improvement will be discussed.

2 Introduction

The field of Computational Fluid Dynamics (CFD) began with the study and simulation of weather on early computers in the 1940s.[20] Thanks to the foundational research done at Los Alamos National Lab in the 50s and 60s, CFD began to see use in optimizing aerodynamics in aerospace and vehicle design.[20] Since then, uses for CFD have continued to grow with ever increasing computational capacity to include physics simulations, visual effects, biological simulations and many other areas.[20]

Computational Fluid Dynamics is predominantly concerned with the numerical integration of the Navier-Stokes equations. These are second order partial differential equations with respect to time (Equation 1) which describe fluid in terms of the *continuum hypothesis*, which models fluids as smooth vector fields[5]. Despite the true nature of fluids being an emergent property of their molecular interactions, the macroscopic behaviours of a fluid can be predicted by this model with sufficient accuracy for most applications[5].

$$d\mathbf{u}/dt = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{g} \tag{1}$$

Velocity of the fluid is represented with vector field $\mathbf{u}$, and density is represented with scalar field ρ (Greek rho, not to be confused with pressure p).

Fluid simulation can broadly be grouped into two categories: *Eulerian* and *Lagrangian*. Eulerian simulations observe physical quantities which move past fixed locations in space. Often these fixed points form a regular grid, however some methods also use meshes. Lagrangian simulations instead track the movement of physical quantities through the flow of the fluid. There also exist hybrid approaches which attempt to make use of both paradigms, such as Particle in a Cell (PIC).[14]

Eulerian methods, especially those using grid structures, lend themselves particularly well to parallelization, since individual steps in a simulation can be designed as filters.[14] Eulerian simulations of free surface flows, however, tend suffer from loss of fluid mass over time.[14] In these simulations the distinct boundary between different fluids is typically modelled by a signed distance function, which to flow along with the fluid is advected with the other physical properties, and then since this distorts the distance values it must be corrected back to a signed distance function again.[14] Over time numerical diffusion of this boundary leads to observable mass loss.

Lagrangian methods track fluid using discrete masses, making them immune to this problem altogether. However, these methods suffer another problem. The forces involved when particles interact with each other or with a boundary can be very large for an infinitesimally small amount of time. This is especially true for incompressible fluids, which resist any and all potential compression with tight spring-like forces. However, if we integrate using a larger timestep, these large forces will cause particles to move much farther than they should. This instability cascades into the simulation diverging, or "exploding". To ensure stability the timestep must always be sufficiently small enough that this will never occur.

Particle-based fluid simulations are also a difficult problem to parallelize efficiently. Similar to the notorious N-Body problem, particles in CFD simulations may potentially interact with any other particle in the system, at any point in time. The random access which this requires is particularly unfriendly to GPUs, which are typically optimized for streamed memory. To make matters worse, to obtain any meaningful results requires a very large number of particles (typically on the order of millions) and the calculations must be done at extremely small time-scales in order to ensure the stability of the system.

3 Literature Review

3.1 Smoothed-Particle Hydrodynamics

Smoothed-particle Hydrodynamics (SPH) is a Lagrangian simulation method first proposed independently by L.B. Lucy and R.A. Gingold and J.J. Monaghan for simulations in astrophysics.[15, 7] J.J. Monaghan later extended the method to free surface flows.[16]

SPH treats discrete masses of fluid as particles which can move freely throughout the simulation domain. These particles are subject to forces due to pressure and viscosity of the fluid. Forces are calculated for each particle based on its current position and velocity and then new position and velocity are computed by integrating these forces over a given time step.

The SPH update step is presented in Algorithm 1. The computation of acceleration has been split as suggested by Ihmsen et al. so that advection forces are considered when computing pressure forces.[12]

There are two main forces which must be calculated each step in SPH: the pressure force $\mathbf{a}_i^{pres}$ and viscosity force $\mathbf{a}_i^{visc}$. External forces such as gravity are modelled with a third force $\mathbf{a}_i^{ext}$.

The first is pressure force which is calculated from density using an Equation of State (EOS). The one presented in Equation 2 follows that of Reinhardt et al.[18]

$$p_i = \begin{cases} \kappa(\rho_i - \rho_0) & \text{if } \rho_i > \rho_0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Equation 2 requires that we determine the density of the fluid at each particle's center point $\mathbf{x}_i$. Using the Parzen-Rosenblatt kernel estimation we can define the density field in terms of the particles as point samples as shown in Equation 3. The volume scaling factor V_i is approximated as m_i/ρ_i using

Algorithm 1 Full SPH step using split forces calculation.

for $i \in \{1, 2, ..., n\}$ **do**
  Compute $\mathbf{a}_i^{visc} = \nu \Delta \mathbf{u}_i / \rho_i$
  Compute $\mathbf{a}_i^{ext}$
  $\mathbf{u}_i \leftarrow \mathbf{u}_i + (\mathbf{a}_i^{visc} + \mathbf{a}_i^{ext}) dt$
end for
for $i \in \{1, 2, ..., n\}$ **do**
  Compute ρ_i by Equation 3
  Compute p_i by Equation 2
end for
for $i \in \{1, 2, ..., n\}$ **do**
  Compute $\mathbf{a}_i^{pres} = -\nabla p_i / \rho_i$
end for
for $i \in \{1, 2, ..., n\}$ **do**
  $\mathbf{u}_i \leftarrow \mathbf{u}_i + \mathbf{a}_i^{pres} dt$
  $\mathbf{x}_i \leftarrow \mathbf{x}_i + \frac{1}{2}(\mathbf{a}_i^{visc} + \mathbf{a}_i^{ext} + \mathbf{a}_i^{pres}) dt^2 + \mathbf{u}_i dt$
end for

the physical quantities of mass m_i and density ρ_i, which allows us to cancel density.[14] We can then use this to estimate the density at each particle, ρ_i, which was previously unknown.

$$\rho(\mathbf{x}) = \sum_i^n \rho_i W(\mathbf{x} - \mathbf{x}_i) V_i = \sum_i^n m_i W(\mathbf{x} - \mathbf{x}_i) \tag{3}$$

To calculate the pressure force $\mathbf{a}_i^{pres}$ we also need a discretization of the gradient operator. We must be careful when discretizing the gradient that the resulting pressure force is symmetric between pairs of particles (respecting Newton's third law)[1], as in Equation 4.

$$\nabla p_i / \rho_i = \sum (pj/\rho_j^2 + p_i/\rho_i^2)(xj - xi) \nabla W(|x_j - x_i|) m_j \tag{4}$$

The second force we must calculate is the viscosity force $\mathbf{a}_i^{visc}$. Viscosity is modelled as the Laplacian of the velocity field. As with the gradient, we also insist in a discretization of the Laplacian which results in symmetric viscosity forces.[1]

$$\nabla \mathbf{u}_i = \sum (\mathbf{u}_j - \mathbf{u}_i) \nabla W(\mathbf{x}_j - \mathbf{x}_i) m_j / \rho_j \tag{5}$$

Until now we have avoided discussing the kernel W. The kernel should obey three properties: first is that the area under its curve must add to one; second, the kernel should vanish after h units (so that $W(h) = 0$), where h is called the support distance; and third, the kernel must be differentiable everywhere on its domain.[14] Many different kernels have been suggested, and we must be careful to use the kernel for the appropriate dimension. Using a kernel designed for $\mathbb{R}^2$ will not have an integral of 1 in $\mathbb{R}^3$ (or vice versa). The kernels presented in Equations 6 and 7 are due to M. Müller.[17]

$$W(\mathbf{r}) = \begin{cases} 315 \frac{(1-|\mathbf{r}|/h)^3}{64\pi h^3} & \text{if } |\mathbf{r}| < h \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$\nabla W(\mathbf{r}) = \begin{cases} -45\mathbf{r} \frac{(1-|\mathbf{r}|/h)^2}{\pi * h^4} & \text{if } |\mathbf{r}| < h \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{7}$$

3.2 Neighborhood Search

Every particle in the simulation enacts a force on and likewise has a force enacted on it by every other particle. This means there are $O(n^2)$ interactions between particles in the system, which is obviously intractable when dealing with millions of particles. Thankfully, the magnitude of these forces vanishes as distance increases, and can be safely ignored after a certain threshold. If we assume an upper limit on the number of possible neighbors a given particle will interact with, we can even reduce the simulation from quadratic to linear.

One possible solution is to use a tree based acceleration structure such as a K-D Tree. However, tree structures are ill suited to the streaming memory models of the GPU. One alternative which is a data structure called a cell list[2]. In a cell list, the simulation domain is split into a regular grid of cubes known as cells. Each cell contains an index to an arbitrary particle contained by it. Each particle has associated with it an index to the next particle in the cell. A unique index value is reserved for denoting that a cell is empty, and that the final particle in a cell list has been reached.[14]

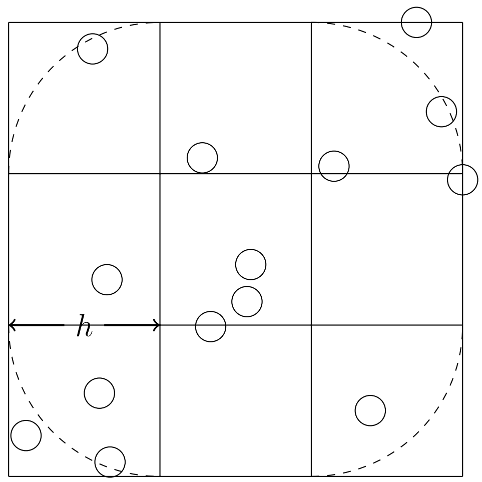

Figure 1: Neighborhood of a cell. Particles inside dashed region are possibly within h of a point in the center cell.

Now when we wish to list neighbors of a given particle we can look up the particle's grid index, and then iterate the particles that share that cell, and those neighboring cells within the given distance threshold. Commonly the cell size is chosen to be the same as the distance threshold, which is also usually the support size of our kernel W, so that we only need to search a $3 \times 3 \times 3$ cube of cells.

3.3 GPU Acceleration

The work of Ihmsen et al. provided much of the groundwork with an early parallel implementation.[12] For this search they used a Z-index sort, a space filling curve which provides a cache-friendly ordering for the particles. Amada et al. present a partial GPU implementation which relies on the CPU for the neighborhood search, providing the information to the GPU as a texture.[3] Harada et al. present an early fully GPU implementation[9]. Later Hérault make use of the programmable pipeline to create a CUDA implementation[11], which they later released as open source[10]. Finally, Rustico et al. extend this to multiple GPUs.[19]

3.4 Time-step

M. Desbrun and M. Gascuel applied the Courant-Friedrichs-Lewy criterion to SPH, providing an upper bound on the time step based on the kernel support size and the maximum particle velocity[6]. They suggest a using an adaptive time-step based on this condition.

Another approach is to avoid a global time-step altogether. P. Goswami and C. Batty propose segmenting the time-step by spatial chunks.[8]. Asynchronous SPH allows every particle to have its own time frame[18][4][6]. This is more efficient when there are only a few fast particles, as is often the case. The method of asynchronous SPH suggested by Reinhardt et al. integrate each particle forward in time individually.[18]

When describing the method it will be useful to speak in terms of a particle's age, which is the total time which it has been integrated during the simulation. All particles in the simulation start at the same age, and ages of particles are tracked individually. A queue was used to sort the particles by their age, so that the youngest particle was always the next one to be stepped forward in time. This ensures that the neighborhood around the particle would all be older, and therefore only requiring that we backtrack particles - keeping the simulation stable.

To make the algorithm multithreaded, Reinhardt et al. suggest dividing the particles into multiple queues, a method due to Kale and Lew.[18, 13] However, now the particle being integrated might not be the youngest particle in its neighborhood, since other threads may be responsible for particles in the particles' neighborhood. In order to solve this, Reinhardt et al. suggest testing for this before integrating. If a younger particle is found, we place the particle in a second list of pending particles. This list is routinely added back to the queue, in the hope that on dealing with a particle the second time another thread will have taken care of the younger particles.

4 Problem Statement

One of the greatest difficulties with SPH over other methods is the incredibly small timesteps required in order for the simulation to remain stable. Since the CFL condition is bounded by the maximum velocity of all the particles in the simulation, only a single fast particle can slow down the entire simulation. By making the simulation asynchronous, we have the opportunity to localize the effect of individual fast particles on the time-step. The problem which will be examined in this report is how to extend the work of Reinhardt et al.[18] and implement asynchronous smoothed-particle hydrodynamics efficiently using the GPU.

5 Method

Originally a per-particle approach was employed in the GPU implementation, much as in the original method. However, instead of a queue, all of the particles were given their own individual thread. The particles who are the youngest of their neighbors are then integrated. However, this was found to be prohibitively expensive to implement on the GPU. While in theory this should be an improvement over the single threaded approach, in tests the simulation ran much slower. The overhead of backtracking the neighborhood of each individual particle costs more than any time saved.

Instead of assigning a single thread to every particle, it was found that instead the problem should be divided by the cells used in the neighborhood search (see Section 3.2). All the particles in a cell are then integrated at once. This allows us to re-use the backtracked neighborhood surrounding the cell for each particle in the cell, greatly reducing the overhead of backtracking while still allowing different areas of the simulation to be integrated at different rates. This is similar to the regional time stepping of P. Goswami and C. Batty[8], while retaining individual timeframes per particle.

A cell is only updated if it contains the youngest particle of all its neighboring cells. This ensures that no region of the simulation is stepped too far ahead of other regions. This also ensures that it is never necessary to extrapolate the location of particles, only interpolation of physical values is required to bring a neighborhood of particles to the same frame of reference in time, increasing the stability of the simulation.

All the particles in a cell are integrated simultaneously by the same time step. The time-step dt is determined by Equation 8, due to Reinhardt et al., which ensures by the Courant-Friedrichs-Lewy condition that the simulation remains stable. The addition of constant term $\epsilon = 0.001$ in the denominators assures the time-step remains defined even for particles which are not moving or accelerating. The scaling factors were determined experimentally.

$$dt = \min_{x_i \in \mathcal{N}} \left\{ 0.02\sqrt{\frac{h}{\epsilon + |a_i|}}, 0.05\frac{h}{\epsilon + |u_i|} \right\} \tag{8}$$

5.1 Implementation

The implementation was written in CUDA and tested on a RTX 3060. The details of this implementation will now be examined. The code for advancing a frame in the simulation is provided below.

First we update the cells for the neighborhood search using `updateCells` (line 1), and copy the result to the GPU (lines 2-9). This is done only once per global update, and so is currently done on the CPU.

```
1 updateCells();
2 cudaMemcpyAsync(particles_dev, particles.data(),
3   numParticles*sizeof(Particle), cudaMemcpyHostToDevice);
4 cudaMemcpyAsync(nextParticles_dev, particles_dev,
5   numParticles*sizeof(Particle), cudaMemcpyDeviceToDevice);
6 cudaMemcpyAsync(cells_dev, cells.data(),
7   200*200*200*sizeof(Cell), cudaMemcpyHostToDevice);
8 cudaMemcpyAsync(nextCells_dev, cells_dev,
9   200*200*200*sizeof(Cell), cudaMemcpyDeviceToDevice);
```

Next we increase the global `time` of the simulation (line 10). Every cell must be at least this old before we finish this cycle in order for us to produce the next frame of animation.

With the updated particle and cell lists on the device, we can begin simulation. This is done by `stepCell`, configured with 8000 workgroups aligned in a $20 \times 20 \times 20$ grid with 160 threads each (line 15).

Memory cannot be written in place, since this would lead to race-time conditions between workgroups. Therefore, two device buffers are used for both the cells and particle lists. One is read from while the other is written to. In order to save on copies, a so called ‘ping-pong buffer’ strategy is employed, where the roles of the buffers are reversed by swapping the pointers (lines 18-19).

```
10 time += 1.0f/60.0f;
11
12 device_ptr<Cell> p = device_pointer_cast(cells_dev);
13 do {
14     for (int k = 0; k < 50; k++) {
15         stepCell<<<dim3(20,20,20),160>>>(cells_dev,
16           nextCells_dev, particles_dev, nextParticles_dev,
17           time);
18         std::swap(cells_dev, nextCells_dev);
19         std::swap(particles_dev, nextParticles_dev);
20     }
21     cudaDeviceSynchronize();
22 } while (find_if(thrust::device, p, p+200*200*200,
23                  isUpdated()) == p+200*200*200);
```

The kernel is called in batches of 50, after which we synchronize the host and device and check if every cell is older than `time`. To do this we use the Thrust library provided by NVIDIA to do perform a fast search through reduction on the cells.

Once all the cells have been sufficiently updated, we then copy the cells and particles back to the host. Before we do this we might optionally sort the particles by cell index, in order to make access to the particle buffer more contiguous and therefore cache friendly. This step was left out of the final implementation as it was measured to be slightly slower. However, for larger numbers of particles or on different GPUs where cache may be more of a bottleneck, this step may serve to help.

```
24 // Optional sort
25 device_ptr<Particle> p = device_pointer_cast(particles_dev);
26 sort(device, p, p+numParticles);
27
28 cudaMemcpyAsync(particles.data(), particles_dev,
29   numParticles*sizeof(Particle), cudaMemcpyDeviceToHost);
```

```
30 cudaMemcpyAsync(cells.data(), cells_dev,
31   200*200*200*sizeof(Cell), cudaMemcpyDeviceToHost);
32 cudaDeviceSynchronize();
```

Now let's examine the kernel `stepCell` itself. Each invocation of `stepCell` is responsible for one main cell, and populates its neighbor list from the neighboring cells. Each workgroup is divided into 160 threads, each sharing the backtracked neighbor list.

Due to the linear nature of the cell list, thread 0 is tasked with populating this list first from the cells. It iterates through all the particles in each cell backtracking them to the same time and then adding the backtracked version to the shared list (lines 58-90).

```
33 __global__
34 void stepCell(Cell* cells, Cell* nextCells, Particle* particles,
35               Particle* nextParticles, float timeLimit) {
36     constexpr static glm::ivec3 deltas[] = {
37         glm::ivec3(-1,-1,-1),
38         // ...
39         glm::ivec3(+1,+1,+1),
40     };
41
42     ivec3 center(90+blockIdx.x, 90+blockIdx.y, 90+blockIdx.z);
43     int centerIndex = center.x+center.y*200+center.z*200*200;
44
45     nextCells[centerIndex] = cells[centerIndex];
46     nextCells[centerIndex].updateCounter = 0;
47
48     Cell& centerCell = cells[centerIndex];
49     if (centerCell.firstParticle == -1
50      || centerCell.time > timeLimit)
51         return;
52
53     __shared__ Particle neighbors[160];
54     __shared__ int numNeighbors;
55     __shared__ float neighborTime;
56     __shared__ bool doStep;
57
58     if (threadIdx.x == 0) {
59         doStep = true;
60         numNeighbors = 0;
61
62         for (int i = centerCell.firstParticle;
```

```
63                  i != -1;
64                  i = particles[i].nextParticle) {
65                 neighbors[numNeighbors++] = particles[i];
66                 assert(numNeighbors != 160);
67             }
68 
69             neighborTime = std::numeric_limits<float>::max();
70             for (ivec3 delta : deltas) {
71                 const ivec3 index = center+delta;
72                 const Cell& cell =
73                     cells[index.z*200*200+index.y*200+index.x];
74                 if (cell.firstParticle == -1)
75                     continue;
76                 else if (cell.time < centerCell.time)
77                     doStep = false;
78 
79                 neighborTime = min(neighborTime, cell.time);
80                 for (int i = cell.firstParticle;
81                      i != -1;
82                      i = particles[i].nextParticle) {
83                     float t = (centerCell.time - cell.time + cell.deltaTime) / cell.deltaTime;
84                     neighbors[numNeighbors++] = backtrace(particles[i], t);
85                     assert(numNeighbors != 160);
86                 }
87             }
88         }
89         __syncthreads();
```

Once this is finished, the n-th thread searches for the n-th particle in the main cell. It then calculates the forces and integrates using `stepParticle` (line 96), storing the result to the next particle buffer (rather than the one being read). As a final step, the first thread updates the main cell's time.

```
90         for (int k = 0, i = centerCell.firstParticle;
91              i != -1;
92              i = particles[i].nextParticle, k++) {
93             if (k == threadIdx.x) {
94                 if (doStep)
95                     nextParticles[i] = stepParticle(neighbors[k],
96                         neighbors, numNeighbors, centerCell.deltaTime);
```

```
97                  else
98                      nextParticles[i] = particles[i];
99                  break;
100             }
101         }
102 
103         if (threadIdx.x == 0 && doStep) {
104             nextCells[centerIndex].updateCounter = 1;
105             nextCells[centerIndex].time =
106                 centerCell.time+centerCell.deltaTime;
107         }
108 }
```

Note that it is important that we copy the particles even if we do not step the cell forward in time (line 101). Otherwise information is not propagated when using the ping-pong buffer strategy.

6 Results

Now the results of the implementation described will be examined. In Figure 6 we see the classic ball drop test. The frames have been displayed by ray-marching a signed distance function produced by spheres centered at each particle and globally smoothed. See accompanying material for the full animation.

The test only involves a small number of particles (approximately a thousand) but shows that the method works and exhibits fluid-like behavior (more obvious in the animations). The OpenMP implementation is roughly four times slower on average, with a peak of 4.5s to calculate the second frame. The CUDA implementation remains nearly constant at 0.25s per frame for the duration of the test.

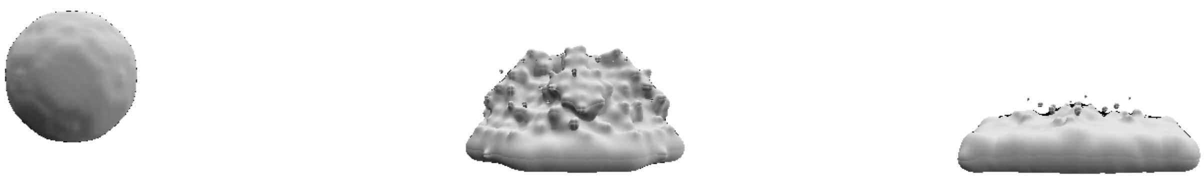

Figure 2: Frames 1,14 and 20 from ball drop test.

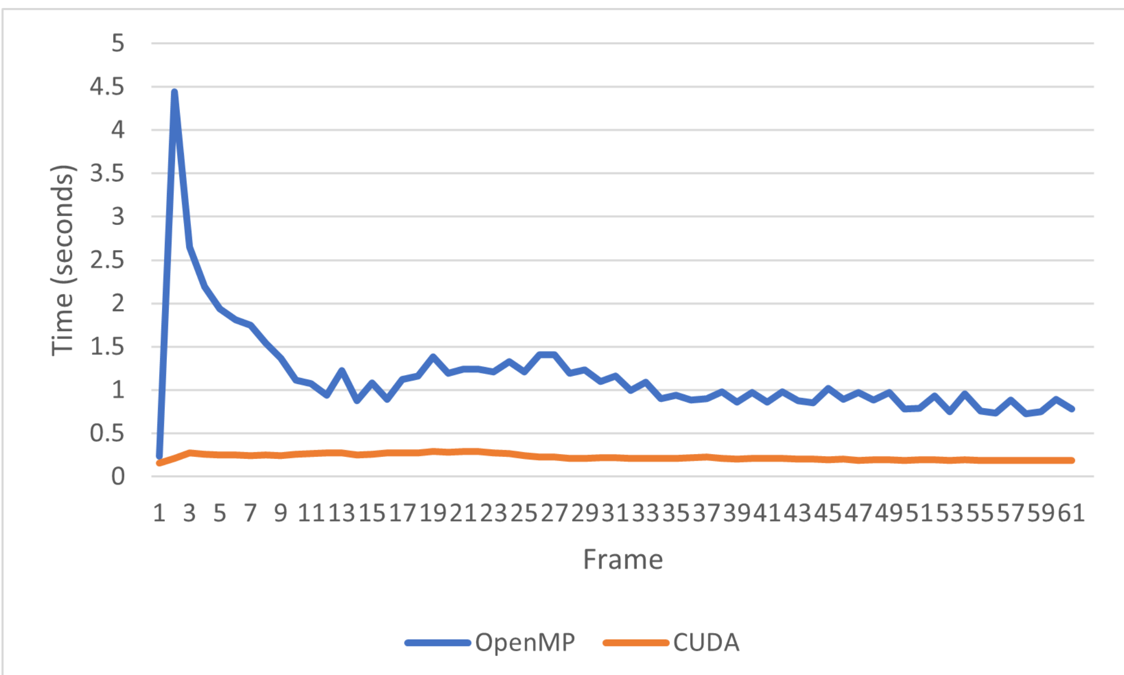

Figure 3: Performance of Ball Drop test.

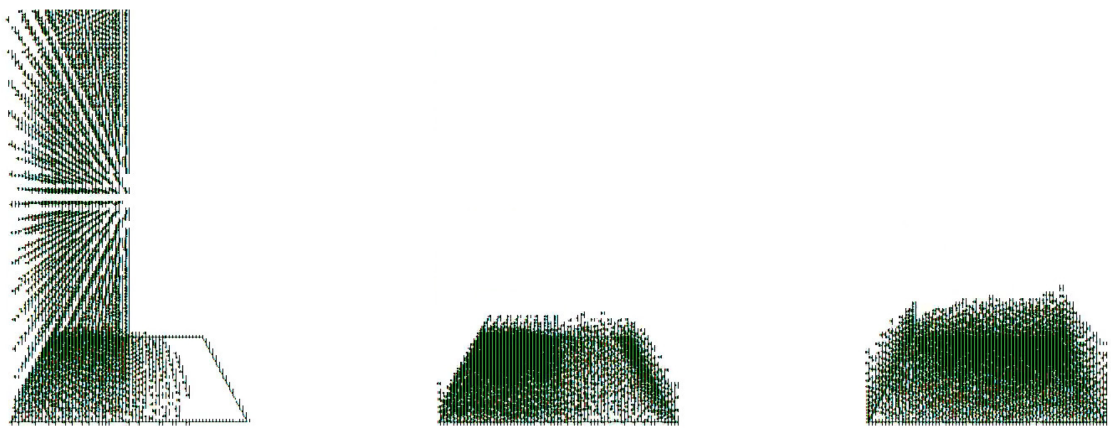

Figure 4: Dam-break test, at approximately 0.5s, 1s and 2s in simulation time. Note the compression which is then resolved over time.

Figure 5 shows the performance results of a larger dam-break test (using just under 7 thousand particles). Full video provided in accompanying animation. Due to the much larger number of particles used this animation was not raymarched. Note the steep spike in computation time for frames 16 to 45. This was likely due to the nature of the test: as the particles become more compressed this increases the number of interactions between particles. Then as the pressure force restores the density of the fluid to the target value the performance begins to increase.

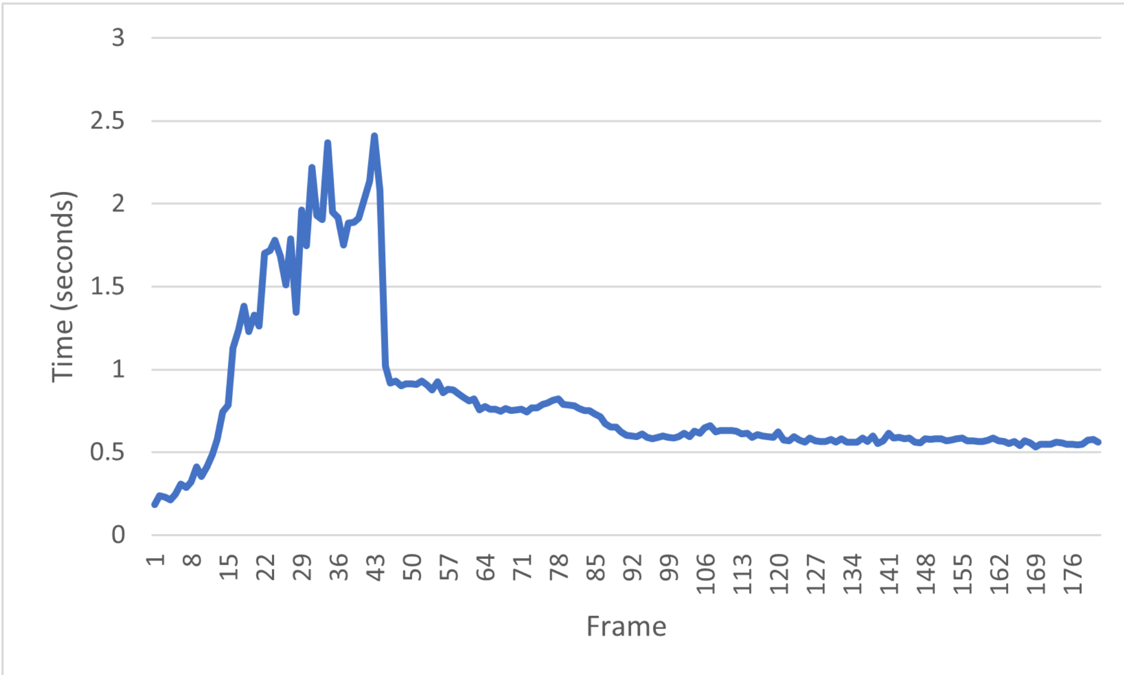

Figure 5: Performance of Dam Break test.

The types of physics simulations which use SPH often require up to tens of millions of particles. Therefore, as a final test, another ball-drop test with approximately one million particles was performed. Unfortunately, the implementation was not able to handle the large amount of neighbors once the ball began to compress after impact with the floor plane. This test was, however, instructive in hilighting some of the areas of future improvement, which will be discussed in the next section.

7 Future Work

A current restriction is the fixed size of the simulation domain. This was done to allow the cells to be parallelized as individual workgroups. A direction of future research would be how to effectively extend the boundary beyond the current limitations in size placed by the number of GPU workgroups. A multigrid approach would be one potential solution: the domain could be broken into chunks of cells which would be small enough to be parallelized. In such an approach one would need to be careful to account for the boundary between chunks to that information properly propagates between them. One possibility here would be to use regions which overlap by a few cells.

Another restriction with the current implementation was found to be the fixed number of neighbors, limited by the size of shared memory between threads in a workgroup. Each potential neighbor also adds a thread, which increases the overall computational cost. In a perfectly incompressible fluid, the upper limit on the possible number of neighbors a particle can have is fixed, since the number of spheres that can be packed in a given volume is fixed. SPH, however, there is a delay between fluids moving too close together and the pressure forces moving them apart again. This delay allows even incompressible fluids such as water to compress momentarily.

This could potentially be addressed by using PCISPH, which enforces the incompressibility of the fluid by relaxing the particles based on the density field.[14] This would ensure that the number of neighbors never rises beyond a constant number.

8 Conclusion

A new method for parallelizing asynchronous SPH for GPGPUs using CUDA has been demonstrated, extending the work of Reinhardt et al.[18]. Though similar to the work of P. Goswami and C. Batty[8], the particles each carry their own time frame, making the simulation fully asynchronous. The method provides visually correct results and shows promising performance on smaller tests. However, future work is needed to scale up to the number of particles required in typical scientific simulations. In particular, the fluid must not be compressible in order to maintain the assumption that the number of neighbors in a cell is fixed.

9 Appendix

The implementation of `stepParticle` is provided below, following the algorithm presented in Section 3.1.

```
__device__
Particle stepParticle(Particle& particle_, Particle* neighbors,
                      int numNeighbors, float dt) {
    Particle particle = particle_;
```

```
    particle.lastPosition = particle.position;
    particle.lastVelocity = particle.velocity;
    particle.lastDensity = particle.density;
    particle.lastPressure = particle.pressure;

    // Compute density
    particle.density = 0.0f;
    for (int i = 0; i < numNeighbors; i++) {
        auto xij = particle.position - neighbors[i].position;
        particle.density += Particle::mass*W(xij);
    }

    // Compute F* (Fvisc + Fext)
    constexpr float nu = 0.015f;
    particle.accel = glm::vec3(0.0f, -9.81f, 0.0f);
    for (int i = 0; i < numNeighbors; i++) {
        auto vij = particle.velocity - neighbors[i].velocity;
        auto xij = particle.position - neighbors[i].position;
        if (length2(xij) > 0.0) {
            particle.accel += 2.0f * nu
                * Particle::mass / neighbors[i].density
                * vij * dot(xij, dW(xij))
                / (dot(xij, xij) + 0.01f * sq(Particle::radius))
                  ;
        }
    }

    // Compute velocity using forces
    particle.velocity += dt*particle.accel;

    // Compute new density
    particle.density = 0.0f;
    for (int i = 0; i < numNeighbors; i++) {
        auto xij = particle.position - neighbors[i].position;
        auto vij = particle.velocity - neighbors[i].velocity;
        particle.density += Particle::mass*W(xij);
        particle.density += dt*dot(dW(xij), vij);
    }

    // Compute pressure and pressure forces
    constexpr float kappa = 0.5f;
    auto accelP = glm::vec3{0.0f,0.0f,0.0f};
    particle.pressure = kappa
        * std::max(particle.density - Particle::rho0, 0.0f);
    for (int i = 0; i < numNeighbors; i++) {
```

```
            auto xij = particle.position - neighbors[i].position;
            if (length2(xij) > 0.0) {
                accelP -= dW(xij) * Particle::mass
                    * (particle.pressure / sq(particle.density)
                    + neighbors[i].pressure / sq(neighbors[i].
                        density));
            }
        }
        particle.accel += accelP;

        // Integrate particle over time using dt
        particle.velocity += dt*accelP;
        particle.position += dt*particle.velocity
            + sq(dt)/2.0f*particle.accel;

        glm::vec3 r = glm::vec3(0.4f,0.3f,0.4f);
        glm::vec3 a = 5.0f-r, b = 5.0f+r;
        for (int d = 0; d < 3; d++) {
            if (particle.position[d] < a[d]) {
                particle.position[d] = a[d]
                    +std::min(b[d]-a[d], a[d]-particle.position[d]);
                particle.velocity[d] *= -0.2f;
            }
            if (particle.position[d] > b[d]) {
                particle.position[d] = b[d]
                    -std::min(b[d]-a[d], particle.position[d]-b[d]);
                particle.velocity[d] *= -0.2f;
            }
        }
        particle.time += dt;
        return particle;
}
```

References

[1] Bart Adams and Martin Wicke. "Meshless Approximation Methods and Applications in Physics Based Modeling and Animation." In: *Eurographics (Tutorials)*. 2009, pp. 213–239.

[2] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Oxford university press, 2017.

[3] Takashi Amada et al. “Particle-based fluid simulation on GPU”. In: *ACM workshop on general-purpose computing on graphics processors*. Vol. 41. Citeseer. 2004, p. 42.

[4] Xiaojuan Ban et al. “Adaptively stepped SPH for fluid animation based on asynchronous time integration”. In: *Neural Computing and Applications* 29.1 (2018), pp. 33–42.

[5] Alexandre Joel Chorin, Jerrold E Marsden, and Jerrold E Marsden. *A mathematical introduction to fluid mechanics*. Vol. 3. Springer, 1990.

[6] Mathieu Desbrun and Marie-Paule Gascuel. “Smoothed particles: A new paradigm for animating highly deformable bodies”. In: *Computer Animation and Simulation’96*. Springer, 1996, pp. 61–76.

[7] Robert A Gingold and Joseph J Monaghan. “Smoothed particle hydrodynamics: theory and application to non-spherical stars”. In: *Monthly notices of the royal astronomical society* 181.3 (1977), pp. 375–389.

[8] Prashant Goswami and Christopher Batty. “Regional time stepping for SPH”. In: *Eurographics 2014*. Eurographics Association. 2014, pp. 45–48.

[9] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. “Smoothed particle hydrodynamics on GPUs”. In: *Computer Graphics International*. Vol. 40. SBC Petropolis. 2007, pp. 63–70.

[10] A. Hérault et al. *GPU-SPH*. http://www.ce.jhu.edu/dalrymple/GPU/GPUSPH/Home.html.

[11] Alexis Hérault, Giuseppe Bilotta, and Robert A Dalrymple. “Sph on gpu with cuda”. In: *Journal of Hydraulic Research* 48.sup1 (2010), pp. 74–79.

[12] Markus Ihmsen et al. “A parallel SPH implementation on multi-core CPUs”. In: *Computer Graphics Forum*. Vol. 30. 1. Wiley Online Library. 2011, pp. 99–112.

[13] Kedar G Kale and Adrian J Lew. “Parallel asynchronous variational integrators”. In: *International Journal for Numerical Methods in Engineering* 70.3 (2007), pp. 291–321.

[14] Doyub Kim. *Fluid engine development*. CRC Press, 2017.

[15] Leon B Lucy. “A numerical approach to the testing of the fission hypothesis”. In: *The astronomical journal* 82 (1977), pp. 1013–1024.

[16] Joe J Monaghan. "Simulating free surface flows with SPH". In: *Journal of computational physics* 110.2 (1994), pp. 399–406.

[17] Matthias Müller, David Charypar, and Markus Gross. "Particle-based fluid simulation for interactive applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2003, pp. 154–159.

[18] Stefan Reinhardt et al. "Fully asynchronous SPH simulation". In: *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 2017, pp. 1–10.

[19] Eugenio Rustico et al. "A journey from single-GPU to optimized multi-GPU SPH with CUDA". In: *7th SPHERIC Workshop*. 2012, p. 56.

[20] Wikipedia contributors. *Computational fluid dynamics — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Computational_fluid_dynamics&oldid=1128577928.