Asychronous Smoothed Particle Hydrodynamics

Andrew Pregent

1 Abstract

In this paper, a new method for parallelizing smooth-particle hydrodynamics using CUDA on the GPU will be presented, building upon asychronous SPH. The method will then be evaluated for performance against single and multi-core implementations, and its accuracy will be compared to a commercial simulation package.

2 Introduction

Computational Fluid Dynamics is predominantly concerned with the integration of the Navier-Stokes equations. These second order partial differential equations with respect to time (see Figure 1) describe fluid in terms of the *continuum hypothesis*, which models fluids as smooth vector fields. Despite the true nature of fluids being an emergent property of their molecular interactions, the macroscopic behaviours of a fluid can be predicted with sufficient accuracy for most applications. The Navier-Stokes equations cannot be solved analytically in general, and require numerical integration. Note that there are many different forms of these equations depending on, for instance, whether the fluid is incompressible or not. The form presented here, due to Jos Stam, splits advection (2) and convection or self-advection (1) of the fluid into two equations.

$$d\mathbf{u}/dt = -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\Delta\mathbf{u} + \mathbf{g} \quad (1)$$
$$d\rho/dt = -(\mathbf{u}\cdot\nabla)\rho + \kappa\Delta\mathbf{u} + S \quad (2)$$

Figure 1: Navier-Stokes equations. 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).

Eulerian methods, especially those using grid structures, lend themselves particularly well to parallelization, since individual steps in a simulation can be designed as filters. Eulerian simulations of free surface flows, however, tend suffer from loss of fluid mass over time. 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. Over time numerical diffusion of this boundary leads to observable mass loss. Different methods have been suggested to try and remedy this flaw, from sharpening the boundary to devising conservative advection methods.

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”. The solution is to ensure that the timestep is always small enough that this will never occur. This is done using the CFL condition, with the constant determined experimentally.

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 Previous Work

In this report we will be focusing on a particle based method called Smoothed-particle Hydrodynamics. Despite being slower, the method is advantageous in scientific computations for its accuracy, and so a large amount of research over the years has attempted to reduce the computational time required, which we will examine now briefly.

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.[13, 5] J.J. Monaghan later extended the method to free surface flows.[14] Smoothed-particle Hydrodynamics models fluid using particles and kernel density estimation to approximate smooth fields. The discretizations of the gradient and Laplacian used here are based on the work of B. Adams and M. Wike.[1] The method will be examined in much greater detail in a later section.

Mathieu Desbrun and Marie-Paule 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[4]. This means that in practice the time step must often be very small in order for the simulation to remain stable. Predictive Corrective Incompressive SPH (PCISPH) attempts to address this problem for incompressible fluids such as water, where the problem is exacerbated by the high stiffness required in the equation of state (EOS)[18].

Another approach is to avoid a global time-step altogether. Prashant

Goswami and Christopher Batty propose segmenting the time-step by spatial chunks.[6]. Asychronous SPH allows every particle to have its own time frame[16][3][4]. This is more efficient when there are only a few fast particles, as is often the case. Reinhardt et al. also suggest using multiple queues in parallel, a method due to Kale and Lew.[16][12]. This is the method which we will extend here to the GPU.

Much research has been done to bring SPH to the GPU. Much of the difficulty lies in efficiently searching a fixed distance neighborhood of each particle, since a brute force search of every particle pair is infeasible. The work of Ihmsen et al. provided much of the groundwork with an early parallel implementation.[11] 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.[2] Harada et al. present an early fully GPU implementation[8]. Later Hérault make use of the programmable pipeline to create a CUDA implementation[10], which they later released as open source[9]. Finally, Rustico et al. extend this to multiple GPUs.[17]

The neighborhood search has also garnered interest on its own. Ohno, Nitta and Nakai look at optimizing the neighborhood search for the GPU. Recently, Groß and Köster look at utilizing modern features of GPUs[7]. In particular, they leverage atomic memory synchronization between work groups in order to optimize the search further.

4 Problem Statement

Following the work of Reinhardt et al., the problem which will be examined is how to implement asynchronous smoothed-particle hydrodynamics efficiently using the GPU. The benefit of such an implementation is clear: one of the great 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 velocity of particles in the simulation, often only a few fast particles will end up greatly reducing the global timestep.

5 Method

Now the original method of SPH will be examined in greater detail, followed by optimizations made to the fixed-radius neighborhood search. We will consider the method of asychronous SPH presented by Reinhardt et al. and suggest a new method for implementing ASPH on the GPU. Finally, the details of the CUDA implementation will be discussed.

5.1 Smoothed-Particle Hydrodynamics

Smoothed-particle Hydrodynamics is a Lagrangian method which 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.

There are two forces at play in SPH which must be calculated. The first is pressure force. Pressure is computed from density using an Equation of State (EOS). There are many different choices for this, the one in Equation 3 is commonly used for water. The adiabatic index γ is typically set to 7 for water, ρ_0 is the reference density of the fluid, and c_s is the speed of sound in the fluid.

$$p_i = \frac{\rho_0 c_s}{\gamma}\left(\left(\frac{\rho_i}{\rho_0}\right)^{\gamma} - 1\right) \tag{3}$$

This requires that we know the density field 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 4. The volume scaling factor V_i is approximated as m_i/ρ_i using the physical quantities of mass m_i and density ρ_i, which allows us to cancel density. We can then use this to estimate the density at each particle, ρ_i, which was previously unknown.

$$\rho(\mathbf{x}) = \sum_i^n \rho_i W(\mathbf{x} - \mathbf{x}_i) V_i = \sum_i^n m_i W(\mathbf{x} - \mathbf{x}_i) \tag{4}$$

One important property which we desire when discretizing the gradient of pressure is that the resulting pressure force is symmetric between pairs of particles, respecting Newton's third law. Because of this, the derivation of the gradient operator is somewhat roundabout, requiring some algebraic manipulation to arrive at a form with the desired property.

$$\begin{aligned}
\nabla(p/\rho) &= (\rho\nabla p - p\nabla\rho)/\rho^2 && \text{Quotient Rule}\\
\rho\nabla p &= \rho^2\nabla(p/\rho) + p\nabla\rho\\
\nabla p_i &= \nabla(p/\rho)[x_i] + (p_i\nabla\rho[x_i])/\rho_i\\
\nabla p_i &= \rho_i(\nabla(p/\rho)[x_i] + (p_i\nabla\rho[x_i])/\rho_i^2)\\
\nabla p_i &= \rho_i\left(\sum(p_j/\rho_j)(xj - xi)\nabla W(|x_j - x_i|)(m_j/\rho_j)\right.\\
&\qquad \left.+(pi/\rho_i^2)\sum(x_j - x_i)\nabla W(|x_j - x_i|)m_j\right)\\
\nabla p_i/\rho_i &= \sum(pj/\rho_j^2 + p_i/\rho_i^2)(xj - xi)\nabla W(|x_j - x_i|)m_j
\end{aligned}$$

The second force we must calculate is the viscosity force. Viscosity is modelled as the Laplacian of the velocity field. As with the gradient, we also insist in a discretization Laplacian which results in symmetric viscosity forces.

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

Finally, the full SPH update step is now presented in Algorithm 1. The computation of acceleration is split as in Ihmsen et al. so that advection forces are considered when computing pressure forces. [11]

Until now we have avoided discussing the kernel W. The kernel should obey three properties. The 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. Third, the kernel must be differentiable everywhere on its domain. Many different kernels have been suggested, and

Algorithm 1 Full SPH step using split forces calculation.

```
for $i \in \{1, 2, ..., n\}$ do
    Compute $\mathbf{a}_i^{visc}$
    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 $\rho_i$
    Compute $p_i$
end for
for $i \in \{1, 2, ..., n\}$ do
    Compute $\mathbf{a}_i^{pres}$
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
```

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).

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

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

The kernels used here are provided in Equations 6 and 7, due to M. Müller.[15]

5.2 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 reduce the simulation from quadratic to linear.

Tree structures are ill suited to the streaming memory models of the GPU and were avoided when optimizing the neighborhood search. Instead, cell lists were used, which are a simple but effective structure commonly chosen for particle simulations.

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.

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. This is the approach also taken in this implementation.

Note that initially sorting by cell index was also performed. In theory this would increase efficiency overall by improving cache coherence, but was instead found to be slightly slower in the tests performed. This indicates that cache was not a bottleneck. However, in larger tests such a sorting step may prove worthwhile.

5.3 Asynchronous SPH

Now we will examine Asynchronous SPH, as proposed by Reinhardt et al. and present modifications to efficiently implement it on the GPU.

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. All the particles in a cell are integrated simultaneously by the same time step.

The time-step dt is determined by Equation 8, 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 timestep remains defined even for particles which are not moving or accelerating. The scaling factors were determined experimentally.

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

The method of asynchronous SPH suggested by Reinhardt et al. integrate each particle forward in time 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. 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.

Originally a per-particle approach was also employed in the GPU implementation. 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 5.2). 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.

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.

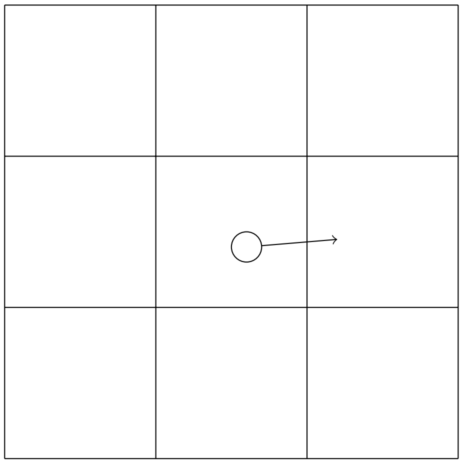

Figure 2: The youngest particle determines the time step of a cell.

5.4 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;
```

```

    __shared__ Particle neighbors[160];
    __shared__ int numNeighbors;
    __shared__ float neighborTime;
    __shared__ bool doStep;

    if (threadIdx.x == 0) {
        doStep = true;
        numNeighbors = 0;

        for (int i = centerCell.firstParticle;
             i != -1;
             i = particles[i].nextParticle) {
            neighbors[numNeighbors++] = particles[i];
            assert(numNeighbors != 160);
        }

        neighborTime = std::numeric_limits<float>::max();
        for (ivec3 delta : deltas) {
            const ivec3 index = center+delta;
            const Cell& cell =
                cells[index.z*200*200+index.y*200+index.x];
            if (cell.firstParticle == -1)
                continue;
            else if (cell.time < centerCell.time)
                doStep = false;

            neighborTime = min(neighborTime, cell.time);
            for (int i = cell.firstParticle;
                 i != -1;
                 i = particles[i].nextParticle) {
                float t = (centerCell.time - cell.time + cell.
                    deltaTime)
                     / cell.deltaTime;
                neighbors[numNeighbors++] = backtrace(particles[
                    i], t);
                assert(numNeighbors != 160);
            }
        }
    }
    __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.

```
 91 |     for (int k = 0, i = centerCell.firstParticle;
 92 |          i != -1;
 93 |          i = particles[i].nextParticle, k++) {
 94 |         if (k == threadIdx.x) {
 95 |             if (doStep)
 96 |                 nextParticles[i] = stepParticle(neighbors[k],
 97 |                     neighbors, numNeighbors, centerCell.
    |                         deltaTime);
 98 |             else
 99 |                 nextParticles[i] = particles[i];
100 |             break;
101 |         }
102 |     }
103 |
104 |     if (threadIdx.x == 0 && doStep) {
105 |         nextCells[centerIndex].updateCounter = 1;
106 |         nextCells[centerIndex].time =
107 |             centerCell.time+centerCell.deltaTime;
108 |     }
109 | }
```

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.

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). We can see in Figure 4 that the performance of the CUDA implementation is much improved over the CPU-based implementation, even when using OpenMP to make use of multiple threads.

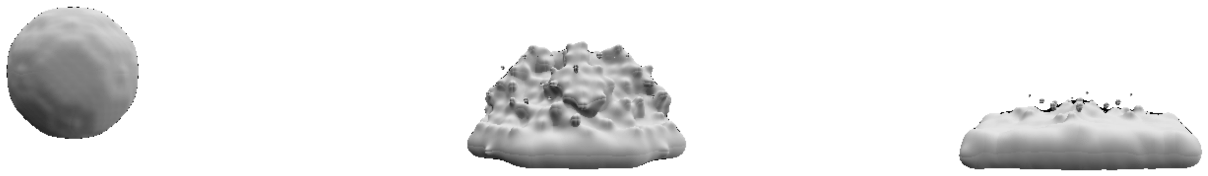

Figure 3: Frames 1,14 and 20 from ball drop test.

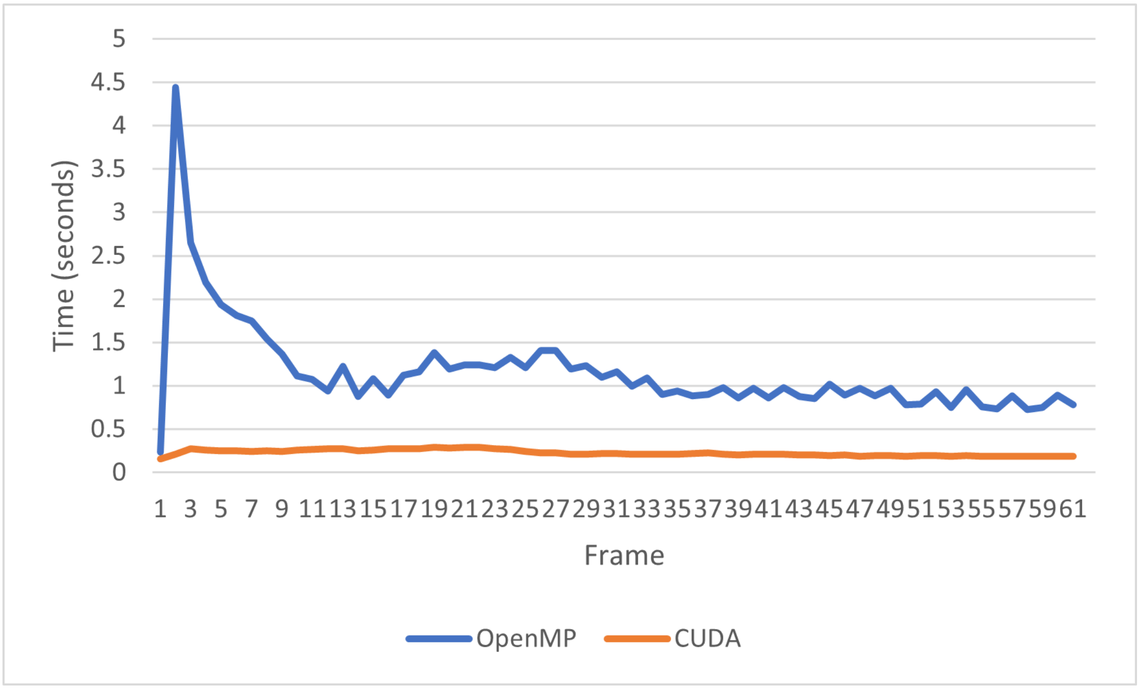

Figure 4: Performance of Ball Drop test. The OpenMP implementation is roughly four times slower on average, with a peak of 4.5s to calculate the second frame. The CUDA implementation remains nearly constant at 0.25s per frame for the duration of the test.

Figure 4 shows the results of a larger test. The frame times spike in the beginning, which is due to the compression of the fluid increasing the number of neighbors for each cell. Once this pressure is resolved the performance improves to just over half of a second per frame.

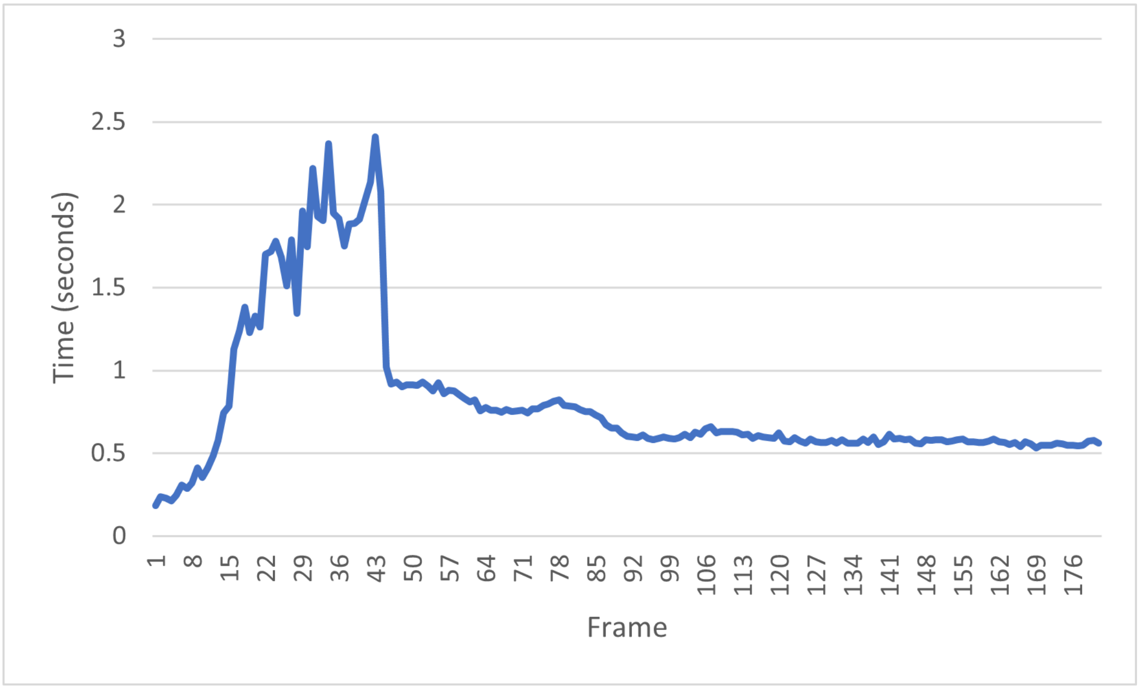

Figure 5: Performance of Dam Break test. Note the steep increase in computation time for frames 16 to 45. This was likely due to the nature of the test: as the particles become more compressed this increases the number of interactions between particles. Then as the pressure force restores the density of the fluid to the target value the performance begins to increase.

Finally another ball-drop test with approximately one million particles was performed. The types of physics simulations which use SPH often require up to tens of millions of particles. 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. In the next section some potential directions for solving this problem will be discussed.

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

8 Conclusion

A method for parallelizing asynchronous SPH for GPGPUs using CUDA has been demonstrated. The method provides 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

```
__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] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] Julian Groß, Marcel Köster, and Antonio Krüger. "Fast and Efficient Nearest Neighbor Search for Particle Simulations." In: *CGVC*. 2019, pp. 55–63.

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] Barbara Solenthaler and Renato Pajarola. "Predictive-corrective incompressible SPH". In: *ACM SIGGRAPH 2009 papers*. 2009, pp. 1–6.