Final Project - Fluid Simulation with SPH

Author: Haris Themistoklis

Date: June 6, 2020
Class: GPU Programming (CS 89)

Acknowledgements: I have used various materials and resources in this projects which weren't developed by me. I cite all those materials when the need for their use arises in this document.

Introduction

The main equation that underlies our simulation is:

$$\boxed{\rho \frac{D\vec{u}}{Dt} = -\nabla p + \mu \nabla^2 \vec{u} + \rho \vec{g}}$$

where:

- ρ is the density of each particle.
- $\vec{u}$ is the velocity of each particle
- p is the pressure for each particle.
- $\vec{g}$ is a sum of external accelerations (mostly gravity)
- the term $\mu \nabla^2 \vec{u}$ is related to the **Cauchy stress vector**

The intricacies of this equation are not very relevant to this project.

We will simulate fluid motion using the **smoothed-particle hydrodynamics (SPH)** method.

From Wikipedia on what this method is:

"This method works by dividing the fluid into a set of discrete moving elements i, j referred to as *particles*. Their Lagrangian nature allows setting their position $\vec{r_i}$ by integration of their vecocity $\vec{v_i}$ as

$$\frac{d\vec{r_i}}{dt} = \vec{v_i}$$

These particles interact through a **kernel function** with characteristic radius knows as the "smoothing length", typically represented in equations by h. This means that the physical quantity of any particle can be obtained by summing the relevant properties of all the particles that lie within the range of the kernel, the latter being used as a weighting function W. First, an arbitrary field A is written as a convolution with W:

$$A(\vec{r}) = \int A(\vec{r'}) W(|\vec{r} - \vec{r'}|, h) dV(\vec{r'})$$

The error in making the above approximation is order h^2. Secondly, the integral is approximated using a Riemann summation over the particles:

$$A(\vec{r}) = \sum_j V_j A_j W(|\vec{r} - \vec{r_j}|, h)$$

where the summation over j includes all particles in the simulation. V_j is the volume of particle j, A_j is the value of the quantity A for particle j and $\vec{r}$ denotes position."

For example, we can calculate the density ρ_i of particles i as

$$\rho_i = \rho(\vec{r_i}) = \sum_j m_j W_{ij}$$

where $m_j = \rho_j V_j$.

Force calculation

Based on the momentum equation above and using our SPH model we can figure out a way to calculate the total force $\vec{f}$ applied on any given particle.

Using Euler-integration, we can recover the velocity and position of each particle for sequential timesteps. That will give us a working fluid simulation!

- Because with SPH we calculate forces in a **volume-weighted fashion,** we divide by **density**, not mass the get the acceleration!

We split the total force into 4 forces:

1. **Body force**: $\vec{f_i^g} = \rho\vec{g} \rightarrow$ mainly considering gravity
2. **Pressure force**
$$\vec{f_i^p} = -\sum_j \frac{(p_i + p_j)}{2}\frac{m_j}{\rho_j}\nabla W_{ij}$$
    - We find pressure as a linear function of density to roughly approximate the incompressibility by $p_i = k(\rho_i - \rho_0)$
3. **Viscocity force**
$$\vec{f_i^v} = \mu\sum_j (\vec{v_j} - \vec{v_i})\frac{m_j}{\rho_j}\nabla^2 W_{ij}$$
4. **Penalty force**, to account for interactions of a particle with some implicit geometry ϕ, like the container:
$$\vec{f_i^b} = k_s(\phi(\vec{c_i}) - r_i)(-\nabla\phi)$$
    - We only add this force if $\phi(\vec{c_i}) < r_i$, where $\vec{c_i}$ is the position of particle i and r_i is its radius.
    - ϕ is the implicit 2D or 3D equation describing our boundary surface. For a sphere, it would be $\phi(x, y, z) = x^2 + y^2 + z^2 - R^2 = 0$
    - $\nabla\phi$ is the surface normal at a specific point (take $\vec{c_i}$).

The total force is the **sum** of those 4 forces.

Kernels

We use 2 types of kernels:

1. The **spiky kernel**
$$W_s(\vec{r}, h) = \frac{15}{\pi h^6}\begin{cases}(h - |\vec{r}|)^3, & 0 \le |\vec{r}| \le h \\ 0, & \text{otherwise}\end{cases}$$
2. The **viscosity kernel**
$$W_v(\vec{r}, h) = \frac{15}{2\pi h^3}\begin{cases}-\frac{|\vec{r}|^3}{2h^3} + \frac{|\vec{r}|^2}{h^2} + \frac{h}{2|\vec{r}|} - 1, & 0 \le |\vec{r}| \le h \\ 0, & \text{otherwise}\end{cases}$$

Why these kernels are used, how they are derived and what their properties are is a story for maybe later.

- An implementation of these is provided in the starter code for the COSC 89.18 assignment. That code was a great piece of inspiration for my project: https://gitlab.com/boolzhu/dartmouth-phys-comp-starter/-/blob/master/proj/a2_particle_physics/ParticleFluid.h

Implementation

- **Spatial Hashing + SPH**
    1. Discretize space into cells of size $\approx h$.
    2. Make a **hash-table** with *keys* cell coordinates (c_x, c_h) and *values* **lists of particle indices** that lie in a specific cell.
    3. At each time-step:
        1. Clear the hash-table
        2. (Re-)Insert all the particles in the hash-table.

```
// for each particle
for (int i=0; i<num_of_particles; i++){
    Cell c = find_cell(particle[i]); // find which cell a particle belongs to
    HashTable[c].append(i); // insert (cell, particle) pair in hash-table
}
```

    4. **h-Neighbor search**
        - IDEA: Only need to check 3^d neighboring cells, due to spatial hashing (d is the dimensionality of the space we are simulating in $\rightarrow d = 2$ for us)
        - One way is to **pre-compute** all of these neighborhoods.
          $\rightarrow$ **Good**: easy, fast access to a list of neighbors when we need them
          $\rightarrow$ **Bad**: takes up too much memory. Especially in GPU, this is a limitation.
        - Another way is to compute a particle's neighborhood on demand. We do not take this route as it requires too many fetches of global memory.
        - In any case, how do we compute a neighborhood?
            - Look at 3^d neighboring cells and only choose the particles within those cells that are within a distance of h from the current point.
    5. **Force calculation**
        - Given a particle i, we now know (or can calculate) its neighboring particles.
        - The force $\vec{f}$ on i is calculated as a sum of 4 sub-forces, as we saw above.
            1. First we have to update the **density** ρ_i of each particle $\rightarrow$ use spiky kernel and mass.
            2. Then, we update the **pressure** p_i of each particle, by using the (current) density ρ_i and the rest-density ρ_0
            3. Then, we accumulate into $\vec{f}$, the forces:
                - $\vec{f_i}^g$ $\rightarrow$ only requires a constant gravitational vector
                - $\vec{f_i}^v$ $\rightarrow$ requires current velocity difference, mass, updated density and viscosity kernel function
                - $\vec{f_i}^p$ $\rightarrow$ requires updated pressure, mass, updated density and spiky kernel
                - $\vec{f_i}^b$ $\rightarrow$ requires particle position and radius and implicit surface normal information.
    6. **Advancing**
        - Now that we know the updated forces on each particle, we have to update the velocities and positions to finish with processing this timestep
        - As timesteps move on, so does our simulation

Parallelization

How do we use the GPU to accelerate the computations for the previous section?

1. **Spatial hashing**
    1. Emptying and then re-filling the hash-table → $O(n)$, where n is the number of particles. This ould be parallelizable. Initial thought: **1 thread per element to be loaded.** However:
        - **PROBLEM: we have competitive writes**
            - Will atomic operations help? `atomicCAS()`, `atomicInc()` I don't think so
        - Parallelizing this may actually be hard!
        - SOLUTION (Suggested by Prof. Bo Zhu in the presentation): utilize `atomicAdd()`. I will try to incorporate this into the actual implementation!
        - **Consider doing minimal updates to the hash-table as we move from timestep to timestep**.
            - This gives a great speed-up to the CPU, but again does not apply to the GPU for the same reason as above: competitive writes.
    2. Neighbor search **can be parallelized** → 1 thread per particle.
       → It is now clear that we will be pre-computing the neighborhoods
2. Force calculation **can be parallelized → 1 thread per particle.**
   For a given particle we have the following attributes:
   → mass, position, velocity, acceleration, radius, pressure, density
    1. Density ρ_i and pressure p_i need to be updated before everything else. We can do this given the neighborhood of each particle.
    2. $\vec{f_i}^g$, $\vec{f_i}^v$, $\vec{f_i}^b$ and $\vec{f_i}^p$ can now all be calculated in one kernel with 1 thread per particle!
    3. Calculate *acceleration* from the total force!
3. **Euler integration → parallelizable**

4 kernels in total!

Data Structures

- **Hash-table** $H \rightarrow N \times N$ matrix of lists. Some sort of guarantee probably exists mathematically about the maximum size of a list, so we manually set an upper bound. It will fluctuate depending on the initial conditions of the problem, so we might run into some errors a few times. The programmer is left to adjust accordingly.
- **Particle Neighborhoods** $P_N \rightarrow n \times N_h$, where N_h is the maximum size of a neighborhood (should be quite small due to incompressibility) and n is the number of particles.
- **Particle array:** holds all relevant information about the particles and is updated in every timestep

```
struct particle{
    double mass;
    double2 position;
    double2 velocity;
    double density;
    double pressure;
    double radius;
}; // 8 doubles (good alignment)

particle Particles[n]; // n: number of particles
```

- It is important to consider how the initializations take place. According to our equations, gravity is the main factor that makes everything move, so we just leave it as that. The use of constants is mainly inspired by the Physical computing class starter code.

Kernels

Invoked in every time-step:

1. **Neighborhood search**

```
__global__ void neighborhood_search(int **Hashtable, particle *particles, int **PN)
{
/*
      TODO: For the current particle, locate its h-neighborhood
            using the Hashtable.
            Write results in array PN[]
*/
}
```

2. **Density and Pressure update**

```
__global__ void density_pressure_update(particle *particles, int n, int **PN)
{
      /*
            TODO: For this particle, update its density and pressure
                        using its neighboring particles.
      */
}
```

3. **Force updates**

```
__global__ void force_updates(particle *particles, int n, int **PN)
{
      /*
            TODO: For this particle, update its acceleration by computing all
                        the forces that act on it.
      */
}
```

4. **Euler integration + a little time-saving trick**

```
__global__ euler_integration(particle *particles, int n)
{
      /*
            TODO: Update velocity and position of given particle.
                  -> Check if cell changed! If it did, move it to a different cell.
      */
}
```

Graphics

We used OpenGL in our implementation of graphics for this project. A lot of time was spent understanding the structure and functionality of the API and many initial ideas were left

unfinished, but I did succeed in implementing the basic fluid simulation.

The particles are triangles in my simulation in the interest of time, as rendering more difficult objects proved to be very challenging for me.

I also only rendered the fluid in 2 dimensions, but I have implemented full functionality of a camera and a model-view-projection pipeline through appropriate shader logic, so an extension to 3 dimensions is very doable.

Lastly, I did not succeed in rendering a container, even though the container is very much there and can be seen in the simulation when the particles collide with it. In 3D, I would hope for a half-sphere container, but that I will have to leave for another day.

Testing

We test our implementation on the CPU with 3 different sets of initialization parameters and then confirm that our GPU parallelized implementation yields very close results.

We use the following sets of parameters:

1. `nx = ny = 10, dx = dy = 0.35, m = 0.1, h = 0.8`
    - We test this configuration with 4 different values for the gravity vector and confirm that they notice that they all yield satisfactory results.
2. `nx = ny = 15, dx = dy = 0.6, m = 0.2, h=0.4`
    - This looks a lot more like a fluid! However, my CPU's capabilities seem to be reaching their limit.
3. `nx = ny = 30, dx = dy = 0.7, m = 0.3, h=0.3`
    - A hard test. Initially a segmentation fault was encountered. Adjusting some parameters and boundary conditions makes it work, albeit slower than before. This test case makes it interesting to observe how the particles interact with one another. One can definitely notice the *incompressibility condition* we imposed and the *boundary conditions* which makes the particles bounce off the wall.

These experiments were run on about 400-900 particles and my CPU still struggled with them. On the contrary, running the code on the cluster with CUDA is a breeze for even 40,000 particles! The improvement is massive and when I get a PC with a fast GPU I will definitely give try running it with more points.

GPU Acceleration Techniques

A number of acceleration techniques were considered to make the CUDA C++ implementation run faster:

1. **Assigning more work to each thread**: classically, each thread is responsible for calculating the neighborhoods and updating the forces and acceleration of only one particle. Assigning more particles (like 4 or 8) to each thread may improve performance because it would reduce the overhead of switching between threads and loading memory.
2. **Memory coalescing**: The spatial hashtable is filled sequentially. For reasons analyzed in the *Parallization* section, it is very difficult to parallelize its construction. While this is a bad thing, we can derive a GPU optimization strategy by being clever in the order of filling the hashtable and then the particle neighborhoods. Specifically, the entries in each cell of the spatial hashtable are particle indices that are either consecutive or separated by a

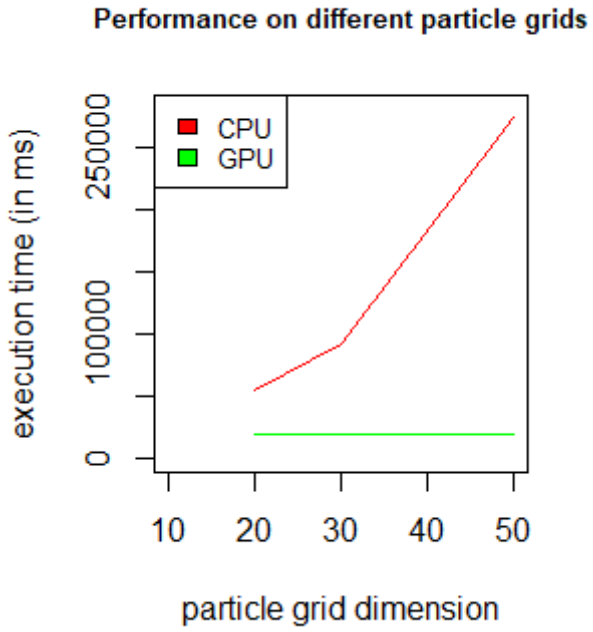
constant stride. Therefore, we avoid random access to global memory locations when we iterate through each particle neighborhood. This probably doesn't provide much of a speed-up, because most time is wasted elsewhere, but it is still a positive step towards an optimal implementation.

3. **Shared memory**: we use shared memory in our implementation because we require many fetches to global memory, mainly through the data structure storing the particle neighborhoods. In our optimization, we first load all particle neighborhoods of a block to shared memory and then access that memory instead of reaching into global memory. This also requires us to *merge our kernels*.
4. **Streams**: we use streams when we want our GPU to be doing work while waiting for the memory to load. There aren't many things we can do while the memory loads, so we cannot use streams in our application.

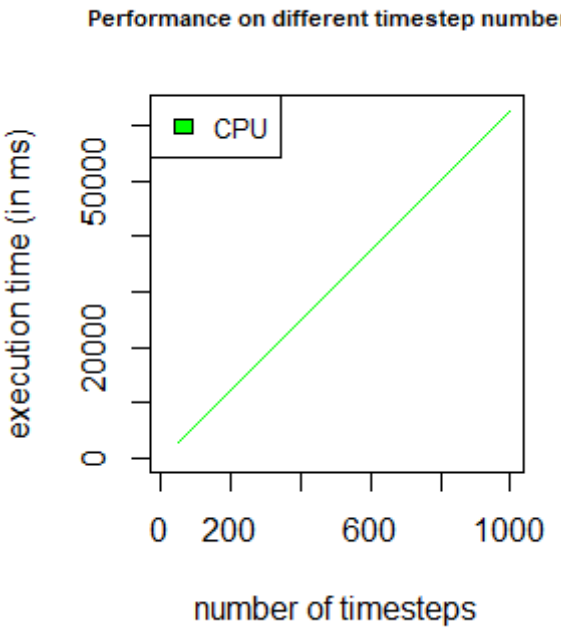
Performance Gain

We can be more quantitative with respect to the question of performance gain of using parallelization vs sequential execution in our implementation.

First we study how different the CPU and GPU implementations are with respect to the number of particles we try to simulate. Realistic fluid simulations require a lot of particles, therefore good performance will make a better simulation. As shown in the figure below, the GPU outperforms the CPU in every grid size we tested. The CPU takes more and more time for more and more particles, but the GPU implementation stays relatively constant. The performance starts to deteriorate for larger grid sizes, at which the CPU has no chance of competing.



Secondly, we study the effect of the number of timesteps on the performance of our implementation. Will having more and more timesteps in our simulation make things worse? As shown in the figure below, such a thing does not happen on GPU: the performance increases linearly, so we can expect our simulation to be evenly smooth thoughout its duration.



Challenges - Future work

During this project, I faced a lot of challenges, learned a lot of things and understood a lot about what I can improve and study in the future. Among those realizations were:

- **OpenGL**: I devoted lots of time and revised my knowledge in computer graphics as I learned beginner OpenGL. I studied OpenGL objects like *Vertex Array Objects and Vertex Buffer Objects*, GLSL *shaders*, *Textures* and *Model loading*. I mainly used the tutorial in this webpage: https://learnopengl.com/. I also used the `Camera` and `Shader` classes in that tutorial, as well as code to initialize OpenGL windows and render triangles. I am still very much a beginner in OpenGL and, as I realized towards the end of the project, my understanding is still lacking in many aspects. I failed to render a basic container for my fluid, which is the first thing I would work on as a future addition to my project.
- **Installation and hardware challenges**: A lot of the software I needed to integrate graphics in my project required lengthy, and many times buggy, installation procedures. Unfortunately, I do not possess a GPU. I spent a lot of time trying to install OpenGL code in the cluster, but even after I succeeded in doing so, I realized that the version installed was much older and I hadn't learned any of the legacy code. I gave up on that attempt fairly quickly. I really wanted to integrate graphics with the simulator, so I resorted to coding up my project in C++ and rendering particles using the CPU, as I demoed in my presentation. This also allowed me to test the correctness of my code. My full project can be found on github at: https://github.com/tkhar/Fluid-Simulation-GPU-OpenGL-
- **Debugging**: I haven't mastered a tool besides `gdb` for debugging CUDA programs. Continuing on with programming GPUs, I will make that my top priority.
- **A research question**: What are upper bounds on the size a particle neighborhood based on the Navier-Stokes equations? What about the size of a cell in the spatial hashtable? There must be some guarantee that can prevent eye-balling in my code...
- **Pertrubations**: It wouldn't be hard to cause pertrubations in the fluid with the project as it is right now. I imagine it would involve some mouse-callback function and possibly an

additional data structure to detect the velocity changes and accumulate them.