Smoothed Particle Hydrodynamics in CUDA

Matteo Princisgh

1 Introduction

Computational fluid simulations can be divided into two main categories:

- **Eulerian methods** where the space is subdivided into a grid and at each step energy transfers between adjacent cells are computed.
- **Lagrangian methods** which are particle-based methods, where at each step we compute the interactions between particles to update the forces that determine each particle's motion.

Both of these methods are used to solve or approximate the Navier-Stokes equations for fluid dynamics, but for this work we will focus only on the Lagrangian methods.

The simplest formulation of the Navier-Stokes equations is the following:

$$\rho \frac{\partial \mathbf{u}}{\partial t} = -\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

where:

- ρ – Fluid density.
- $\mathbf{u}$ – Velocity field.
- p – Pressure.
- ν – Kinematic viscosity.
- $\mathbf{f}$ – External forces (e.g., gravity).

Eq.(1) represents momentum conservation, while Eq.(2) enforces the divergence of the velocity field to be zero (i.e., the volume of the fluid remains constant, which is a necessary condition for an incompressible fluid).

1.1 SPH approximations

Smoothed Particle Hydrodynamics (SPH) methods were introduced independently by astronomers Lucy[1] and Gingold et al.[2] in 1977 for approximating astrophysical fluid dynamics and, due to a lower computational cost compared to Eulerian methods, became widely adopted for real-time particle systems and fluid simulations.
The fundamental principle of SPH methods is that, given a particle i for which we want to estimate a quantity A_i, we can use the following equation to compute such quantity:

$$A_i = \sum_j \frac{m_j}{\rho_j} \cdot A_j \cdot K(\|x_i - x_j\|, h) \tag{3}$$

where:

- A_i – Smoothed value of field quantity A at particle i (e.g., density, velocity, pressure).
- m_j – Mass of particle j.
- ρ_j – Density of particle j.
- A_j – Value of field quantity A at neighboring particle j.
- $K(\|x_i - x_j\|, h)$ – Smoothing kernel, weighting the influence of particle j.

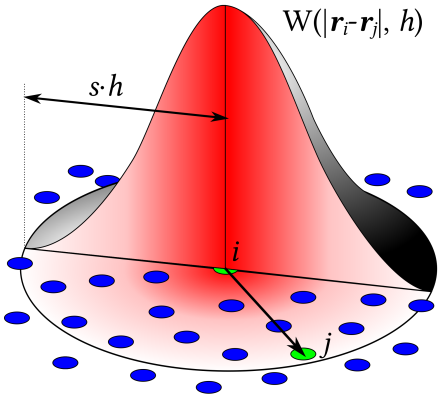


Figure 1: Visualization of the variation of a kernel function influence. Source: Wikipedia

- $\|x_i - x_j\|$ – Displacement of j with respect to i.
- h – Smoothing length, defining the influence radius of the kernel.

In order to obtain an approximation that ensures physical consistency and stability, the kernel function should have at least the following properties:

- **Normalization:** $\int K(\mathbf{r}, h)\, dV = 1$
- **Compact Support:** $K(\mathbf{r}, h) = 0$ for $\mathbf{r} > h$
- **Radial Symmetry:** $K(\mathbf{r}, h) = K(-\mathbf{r}, h)$

In the literature, several kernel functions for density, pressure, and viscosity are proposed. For this work we used those described by Hendra in [3]:

- **Spiky Kernel:**

$$K_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - \|r\|)^3 & 0 \le \|r\| \le h \\ 0 & \text{otherwise} \end{cases}$$

$$\nabla K_{spiky}(r,h) = -\frac{45}{\pi h^6} \begin{cases} \frac{r}{\|r\|}(h - \|r\|)^2 & 0 \le \|r\| \le h \\ 0 & \text{otherwise} \end{cases}$$

- **Viscosity Kernel:**

$$\nabla^2 K_{viscosity}(r,h) = \frac{45}{\pi h^6} \begin{cases} (h - \|r\|) & 0 \le \|r\| \le h \\ 0 & \text{otherwise} \end{cases}$$

1.2 Approximating the Navier-Stokes equations

Volumetric invariance (i.e., the second NS equation) is guaranteed by the fact that the number of particles in the simulation remains constant, whereas computing all the quantities needed for the first equation requires using the SPH approximation of Eq.(3), with some minor adjustments:

- **Density (ρ):**

$$\rho_i = \sum_j \frac{m_j}{\cancel{\rho_j}} \cancel{\rho_j} K_{ij} = \sum_j m_j K_{ij} \tag{4}$$

- **Pressure (p):**

$$p_i = max\{k(\rho_i - \rho_0),\ \varepsilon\}$$

where k is a constant, ρ_0 is the fluid resting density and ε an arbitrarily small constant.

- **Pressure gradient (∇p):**

$$\nabla p_i = \sum_j \frac{m_j}{\rho_j} p_j \nabla K_{ij} \approx \sum_j \frac{m_j}{\rho_j} \frac{p_i + p_j}{2} \nabla K_{ij} \tag{5}$$

- **Velocity Laplacian ($\nabla^2\mathbf{u}$)**:

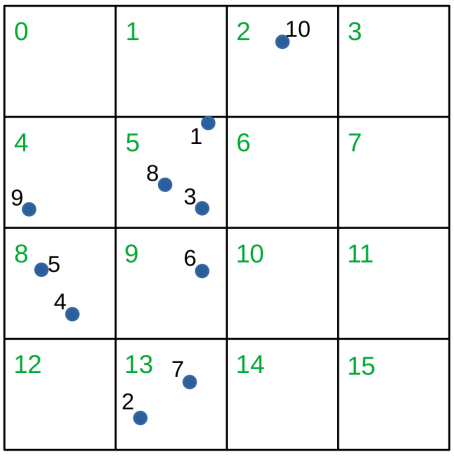
$$\nabla^2\mathbf{u}_i = \sum_j \frac{m_j}{\rho_j}\mathbf{u}_j\nabla^2 K_{ij} \approx \sum_j \frac{m_j}{\rho_j}(\mathbf{u}_i - \mathbf{u}_j)\nabla^2 K_{ij} \tag{6}$$

The last term in Eq.(5) takes the average pressure of two particles, to ensure that their repulsive/attractive forces are symmetric. The same trick is used in Eq.(6), where only the velocity difference between two particles is taken into account for the viscosity calculation.

2 CUDA Implementation

A Lagrangian simulation naturally lends itself to parallelization, where each thread updates a single particle's motion. However, a naive implementation of Eq.(3) would require every thread to iterate over all particles, leading to a $O(n^2)$ complexity per simulation step.

To reduce redundant particle lookups, we adopt the grid-based spatial partitioning method proposed by Green in [4]. The simulation space is divided into a uniform grid with cell sizes that match the kernel smoothing radius. This allows each particle to determine its grid coordinates and efficiently search only the 26 neighboring cells (8 in 2D).



(a)

Index	Unsorted List	Sorted List	Cell Start
0	(5,1)	(2,10)	
1	(13,2)	(4,9)	
2	(5,3)	(5,1)	0
3	(8,4)	(5,3)	
4	(8,5)	(5,8)	1
5	(9,6)	(8,4)	2
6	(13,7)	(8,5)	
7	(5,8)	(9,6)	
8	(4,9)	(13,2)	5
9	(2,10)	(13,7)	7
10			
11			
12			
13			8
14			
15			

(b)

Figure 2: Example of a grid spatial subdivision with the corresponding arrays needed for efficient lookup.

Implementing the grid subdivision requires two additional arrays:

- **Cell-particle array**: which stores, for each particle, a tuple containing the corresponding grid cell ID and the particle ID.
- **Cell start array**: which stores the index (of the cell-particle array) where the n-th cell begins.

In order for the cell-particle array to be useful it must be sorted using the grid cell ID as the sorting key and, given that we are working with integers, we can efficiently sort it using the Radix sort implementation provided by Nvidia's CUB[1] library.
As shown in Figure 2b, once the cell-particle array is sorted, we can iterate over all particles inside a cell, stopping as soon as the next tuple has a different cell ID.

Finally, the simulation loop can be summarized as follows:

[1] https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceRadixSort.html

Algorithm 1: Parallel Fluid simulation

```
1  updateGrid()
2  foreach p_i ∈ Particles parallel do
3      ρ_i ← SPHEstimator(density_func, p_i)
4      f_i^pressure ← SPHEstimator(pressure_func, p_i)
5      f_i^viscosity ← SPHEstimator(viscosity_func, p_i)
6      f_i^gravity ← p_i.velocity.y − 9.81 · ρ_i
7      f_i^total ← f_i^pressure + f_i^viscosity + f_i^gravity
8      a_i ← f_i^total / ρ_i
9      v_i ← p_i.velocity + a_i · Δt
10     p_i.position ← p_i.position + v_i · Δt
11     handleCollisions(p_i)
```

Where the functions `SPHEstimator()` and `updateGrid()` are the following:

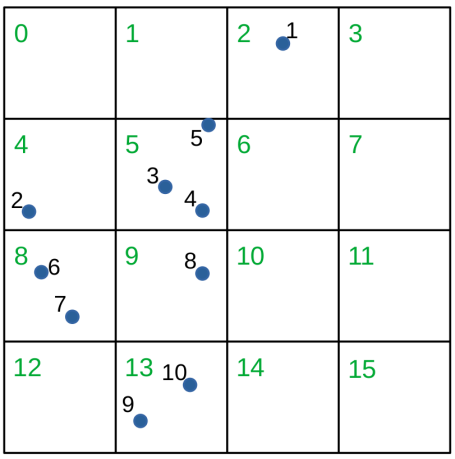
Algorithm 2: SPHEstimator

```
   Input: (func, p_i)
1  cell_ID ← linearGridIndex(p_i)
2  qty ← 0
3  foreach nbrCell ∈ neighbors(cell_ID) do
4      foreach p_j ∈ nbrCell do
5          qty ← qty + func(p_i, p_j)
6  return qty
```

Algorithm 3: updateGrid

```
1  foreach p_i ∈ Particles parallel do
2      cell_ID ← linearGridIndex(p_i)
3      GRID[cell_ID] ← i
```

The implementation described in Algorithm 1 has two key limitations. First, it involves scattered memory accesses to read all particles within a cell, which causes considerable warp divergence. Second, its efficiency decreases significantly as the number of particles per cell increases, which means that the smoothing radius of the kernel has to be carefully tuned to maintain a good performance level.



(a)

Index	Sorted List
0	(2,1)
1	(4,2)
2	(5,3)
3	(5,4)
4	(5,5)
5	(8,6)
6	(8,7)
7	(9,8)
8	(13,9)
9	(13,10)
10	
11	
12	
13	
14	
15	

(b)

Figure 3: Revised example with particle ID sorting

The first issue can be resolved by sorting the particles according to their positions in the grid, as

illustrated in Figure 3. This approach minimizes warp divergence and improves memory access patterns by enabling coalesced reads and writes while taking advantage of warp broadcasting mechanisms. This optimization alone, yields a significant speedup, as summarized in Table 1 and in Fig. 4

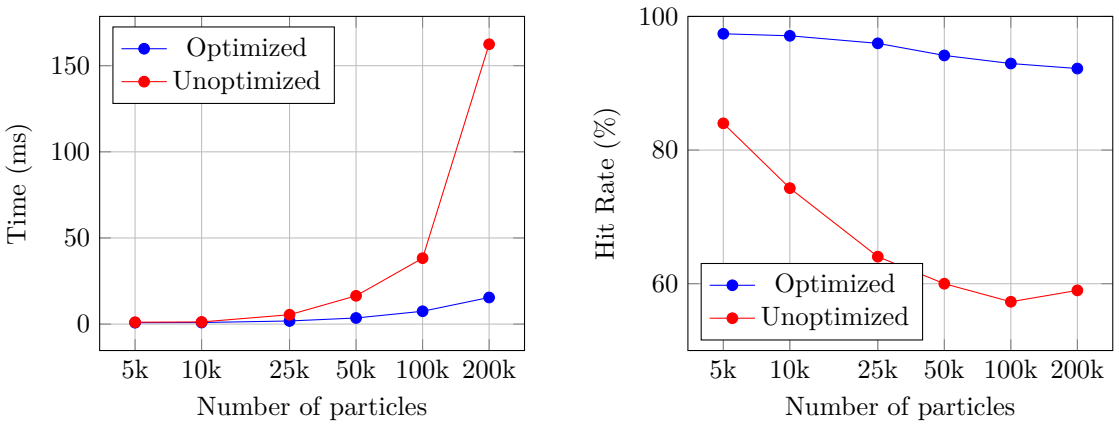


Figure 4: Optimized vs Unoptimized kernel execution times (left) and L1 cache hit rate (right)

`updateParticleKernel`	**5K**	**10K**	**25K**	**50K**	**100K**	**200K**
Execution time (ms)	0.82	0.92	1.81	3.55	7.45	15.45
Compute throughput (%)	14.50	23.05	41.80	56.50	66.41	70.10
L1 Hit rate (%)	97.40	97.10	95.97	94.16	92.95	92.20
L2 Hit rate (%)	44.80	83.20	75.33	92.85	91.90	91.70
Occupancy (%)	11.50	18.80	42.70	72.40	85.42	87.49
Uncoalesced accesses (%)	5.00	4.00	3.00	1.00	1.00	1.00

Table 1: Performance metrics for the optimized kernel with a different number of particles. Measurements taken with a Kernel-function radius of 0.5 units.

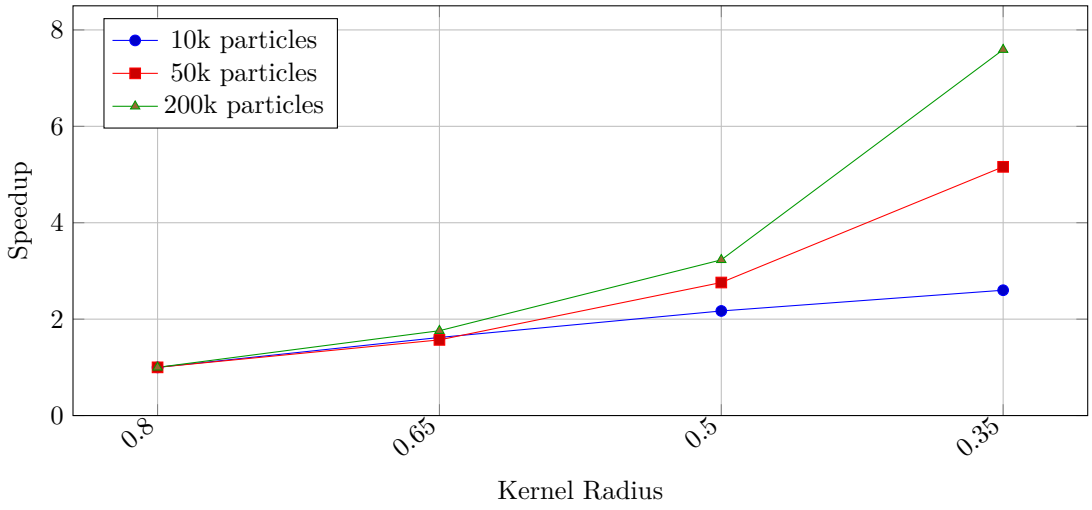


Figure 5: Speedup of kernel execution time for various SPH kernel radii.

Finally, the latter limitation can be addressed by tuning the kernel smoothing radius, in order to reduce the number of particles per grid cell, as shown in Fig.5. Unfortunately, reducing the kernel radius too much increases the instability of the simulation. Such instability is also inversely proportional to the temporal resolution of the simulation, thus, in order to be used in a real-time application with a timestep of 1/60th of a second, the radius should be kept above 0.5 units.

References

[1] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal, vol. 82*, 1977.

[2] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 1977.

[3] Aditya Hendra. Accelerating fluids simulation using sph and implementation on gpu, 2015.

[4] Simon Green. Particle simulation using cuda. *NVIDIA whitepaper*, 2010.