15418 Final Project Report

Kevin You

`https://kskyou.github.io/parallel-fluid-sim/`

1 Summary

I implemented a parallelized version of the Fluid-Implicit-Particle (FLIP) fluid simulator in Taichi, and explored optimizing the conjugate gradient algorithm. We see a significant speedup on the GPU over CPU vectorization.

Note that I have worked with Katerina Nikiforova through the early stages of this project. However, we have decided to take the project to our own directions, and thus we will be submitting separate final reports. I apologize for the troubles this may cause.

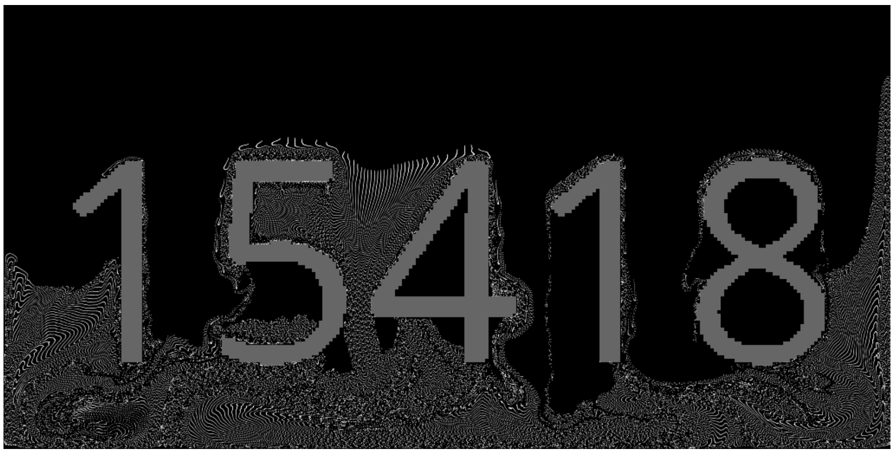

Figure 1: FLIP Simulation

2 Background

Background. Fluid simulation has been intensively studied in the field of computer graphics. Typically methods are classified into Lagrangian or Eulerian methods. Purely Eulerian approaches have been popularized by Stable fluids [3] due to its unconditional stability. However, while these methods excel at force computation, they suffer from numerical dissipation.

On the other hand, Lagrangian methods like the Smooth Particle Hydrodynamics (SPH) are popular for real-time graphics due to their speed and better advection properties, though their force computation are inaccurate. Hybrid Lagrangian/Eulerian combines the advantages of both two methods. The FLIP method [1, 5] is an instance, building on top of older particle-in-cell methods, offers reduced dissipation and more dynamical behavior. It has seen widespread use as a fluid simulator in popular software including Blender and Houdini. However, FLIP is rarely parallelized. One rare instance is that of Wu et al. [4]. Our project explores parallelizing the traditionally serial algorithm.

Overview. The FLIP algorithm's data structure include 1. An array of particles, each having a location and velocity. 2. A staggared Marker-and-Cell (MAC) grid that stores fluid velocities. 3. A conjugate gradient solver. 4. Solid obstacles in the scene represented as voxels with the same sizes as the MAC grid. We perform the following routine per time step dt [2].

1. Transfer the particle velocities to the grid via linear interpolation
2. Advect the particle positions based on the grid velocity with a Runge-Kutta 2 scheme.
3. Apply body forces on the grid velocity
4. Pressure project the grid velocity over all cells containing particles to be divergence free
5. Transfer the grid velocities back to particles via linear interpolation, based on a difference between the current grid velocity and that before body forces, using a FLIP ratio.

See appendix regarding a few small design choices, not relevant to the parallelization, but relevant to get a better simulator.

Parallelization.

- The particle to grid transfer involves grid nodes looking up particle data in parallel. Iterating through all particles will be expensive. This can be made efficient by first scanning the particles into a grid in parallel of the same size as the MAC grid, and then having the grid nodes look into the grid.
- The transfer back is trivially parallelized over particles.
- The pressure projection step involves solving a sparse linear system, which has potential for parallelization. It also tends to be the most expensive step.

3 Approach

Technologies. I elected to use the domain-specific language Taichi for this project. Writing GPU code via CUDA is cumbersome, and various Python frameworks like Taichi utilizes JIT compilation to covert Python code into GPU kernels. Some other frameworks include NVIDIA Wrap and Numba, though I choose Taichi due to its popularity within the computer graphics community, especially among those working in fluid simulation for computer graphics.

This project was nice as it gave me the chance to explore Taichi as a tool (in fact, this is maybe my biggest personal takeaway). The main features of Taichi used in this project are automatic loop parallelization of the outer loop in kernels, and optimizations for reduction operations like sum or max. Taichi also supports other useful features like better memory layouts, but that is not explored in this project.

Mapping. When ran on the CPU, Taichi utilizes both SIMD vectorization and multi-threading. When ran on the GPU, Taichi utilizes wraps. With the convenience of automation, there is not much the user can do to manually tune the exact mapping, though their is an option for block size for GPU.

Iterations. All of the code is coded from scratch, except a small section of the code that performs the parallel scan is adapted. I've tried to do everything at once on my first iteration, which lead to a long debugging time. Taichi's sparse linear solver is also bugged. The LLT solver doesn't work with large matrix sizes, and the conjugate gradient solver appears to have memory leaks. Thus, I implemented my own conjugate gradient solver. I also investigated in speeding up the conjugate gradient solver.

4 Results

Overview. For performance testing I will stay in 2D. The testing machine is equipped with an Intel Core i9-14900KF with 32 GB memory and an NVIDIA GeForce RTX 4080. My primary scene involves a dam break where 2/9 of a 1x1 box is filled water and released under the influence of gravity. The low resolution scene takes 2^{14} particles on a 64x64 grid with a framerate of 60fps and 20 substeps per frame, leading to $dt = 1/1200$. The medium resolution scene takes 2^{14} particles on a 128x128 grid with 40 substesp per frame. Finally, the high resolution is 2^{16} particles on a 256x256 grid with 80 substesp per frame. I take the FLIP ratio to be 0.9. Finally, I run conjugate gradient with a tolerance of 10^{-3} and a maximum iterations of 100.

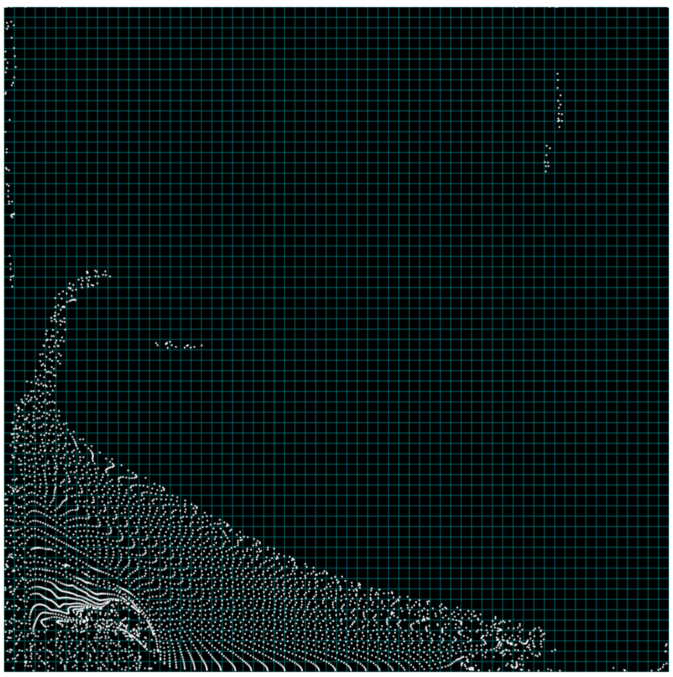

Figure 2: Dam break scene. Low res om frame 100

CG Solver. First I compare by CG solver against the reference solver. With the low resolution scene without gravity I find that simulating 1s of the dam break takes 9.88s with 8.58s spent on CG for the reference solver, and 10.78s with 9.51s spent on CG for my CG solver. Thus, my solver is comparable to the reference solver, and it does not have any memory leaks.

By inspecting the conjugate gradient solver, we realize that much of the computation is spent on cells that have no elements. Therefore, I propose that when initializing the solver, before running all the CG iterations, I first scan through all cells and aggregate an index into all cells that do have particles. Then for the saxpy and matrix vector multiplication procedure, I only loop over cells that do have particles.

To my surprise, this does not improve performance, even though we did a similar thing in assignment 2 that did improve performance. Both the time taken to perform the matrix multiplication and the time to do the saxpy operation increase by around a quarter to a third. But upon more thought, it perhaps makes sense, since doing this does destroy memory coalescing. If fluid cells were very sparse, it might be useful, but now with 2/9 of the cube filled it does not beat the decrease in memory coalescing. There must have been a reason why FLIP methods are not commonly parallelized, and I think it is indeed because of the difficulties of speeding up the conjugate gradient step from a purely programming perspective.

I think this is a very interesting question to consider more generally. In simulation there are some regions that have much more detail than other regions, and we would like to focus

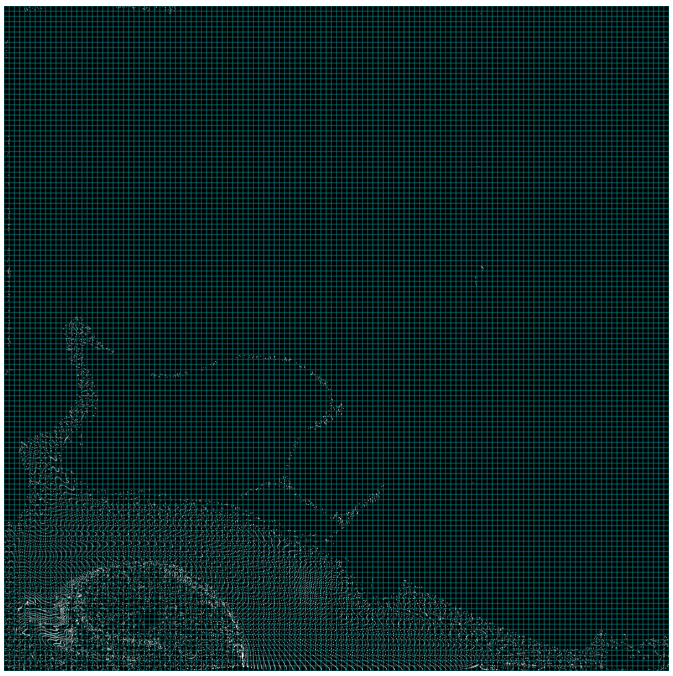

Figure 3: Dam break scene. Medium res om frame 100

our computation resources in these regions. However, because the problem is usually inherently, globally coupled (like the incompressible fluids case), we do not have a good way to assign the work, and we must treat all regions together. Hierarchical structures are maybe the best we can do.

Problem Size. For our hybrid fluid simulation it makes sense to simultaneously scale up the particle size decreased grid sizes to improve accuracy. We tend to bias on the side of more particles since our bottleneck is the pressure projection. On the other hand, we must also simultaneously decrease the time step size as the resolution increases to satisfy the CFL conditions. Naively, one may expect that going from low to medium will requires 4x or 8x computation, 4x due to resolution and 2x due to timestep, but oftentimes what matters more is the number of conjugate gradient steps, which reflects how much the velocity field deviates away from being divergence free. But in our case, we were generally taking up all the conjugate gradient steps. We run the three setups up to 300 frames. However, since the high resolution scene tends to contain more detail as time progresses, we will compare times for the first 100 frames. (These are all various considerations that must be made when analyzing performance)

Speedup. From our table we can clearly see that as the problem size gets larger, our GPU solution gets better than our CPU solution, which is to be expected. Interestingly, changing the code from single threaded to using all threads (verified with system monitor) didn't change much. This means that something weird is going on with work allocation for Taichi's CPU

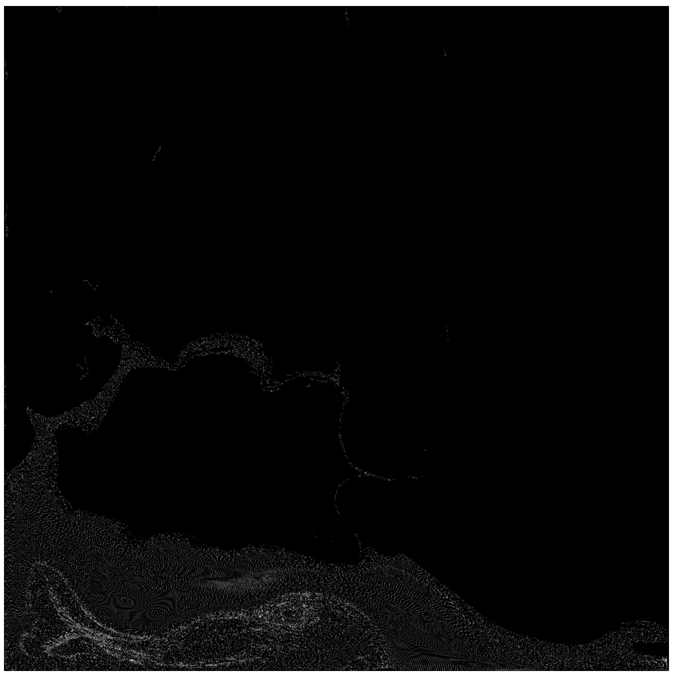

Figure 4: Dam break scene. High res om frame 100

Scene	System	Substeps	Total time (s)	Solver time (s)	Matmul (s)	Total iterations
Low	Singlehread	20	48.63	41.97	6.53	131674
Low	CPU	20	50.72	39.97	7.03	132575
Low	GPU	20	31.96	26.84	2.43	132376
Medium	CPU	40	129.43	121.86	18.6	381373
Medium	GPU	40	81.2	75.9	6.85	374797
High	GPU	80	178.67	172.76	15.57	799742
Bump (high)	GPU	80	169.63	163.92	13.21	796480
15418 (high)	GPU	240	505.72	514.3	45.71	2471779

Table 1: Table of run times for 100 frames, measured by real time elapsed

implementation.

In terms of workload, we see a fairly consistent split the notable components of the algorithm over the different trials. Conjugate gradient has only at most 5 non-zero elements per row, so it also scales the same as other parts of the simulation. Needless to say, the performance is heavily scene dependent, even at the same particle and grid count with different boundary conditions the performace changes dramatically. The scene with the 15418 text is ran at FLIP=0.8 and still eventually exhibits instabilities. See my website for a video of all scenes.

References

[1] Brackbill, J., and Ruppel, H. Flip: A method for adaptively zoned, particle-in-cell

calculations of fluid flows in two dimensions. *Journal of Computational Physics 65* (08 1986), 314–343.

[2] John, C. T. Physics-based simulation animation of fluids.

[3] Stam, J. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., p. 121–128.

[4] Wu, K., Truong, N., Yuksel, C., and Hoetzlein, R. Fast fluid simulations with sparse volumes on the GPU. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018) 37*, 2 (2018), 157–167.

[5] Zhu, Y., and Bridson, R. Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July 2005), 965–972.

A Implementation Details

Boundary. I do not have any ghost rows. The particles can reach anywhere in the grid. The no-penetration boundary condition is enforced throughout the particle to grid transfer, advection, and pressure projection process. On the boundary, particles interpolate from only two instead of four grid points. Again, due to the lack of ghost rows particles tend to be more active near the boundary. If during RK2 a particle leaves the boundary, they are reflected back in.

Solids. Solids are expressed simply as occupation. The no-penetration boundary condition is likewise enforced on all steps. If during RK2 a particle leaves the boundary, their advection is neglected, and they do not move. The hope is that as time progresses, the velocity will settle, leading the particle to eventually be pushed away. If a particle is too close to a boundary (within 0.1 grid length), it will get pushed slightly away from the boundary so the particle can pick up the velocity from further away nodes.