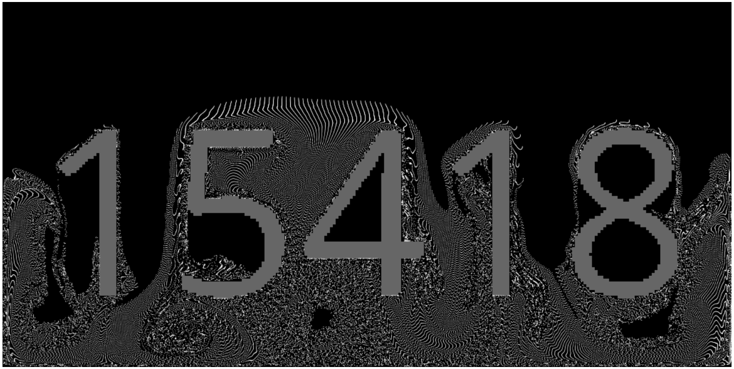
15-418 Parallel FLIP Fluid Simulation

Kevin You

Carnegie Mellon University

Summary

We implement a Fluid-Implicit-Particle (FLIP) fluid simulator in Taichi, with and explored optimizing the pressure projection algorithm. We see 2x to 3x speedup on the GPU over CPU multi-threading and vectorization.



Overview

The FLIP algorithm involves 1. An array of particles, each having a location and velocity. 2. A staggered Marker-and-Cell (MAC) grid that stores fluid velocities. 3. A conjugate gradient solver.

1. Transfer the particle velocities to the grid via linear interpolation
2. Advect the particle positions based on the grid velocity with a second order Runge-Kutta scheme.
3. Apply body forces on the grid velocity
4. Pressure project the grid velocity over fluid cells 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.

Parallelism

- Parallel indexing of particles so grid nodes can look up particles
- The pressure projection step involves solving a sparse linear system via conjugate gradient, which utilizes parallel matrix multiplication, saxpy, and reduction.
- Time breakdown example (GPU): 56.5s total, 52.6s on pressure projection, specifically 28.0s on reduction, 17.7s on saxpy, and 5.6s on sparse amtrix multiplication.
- 3.82s spent on all other operations, including particle-grid and grid-particle transfer

Result

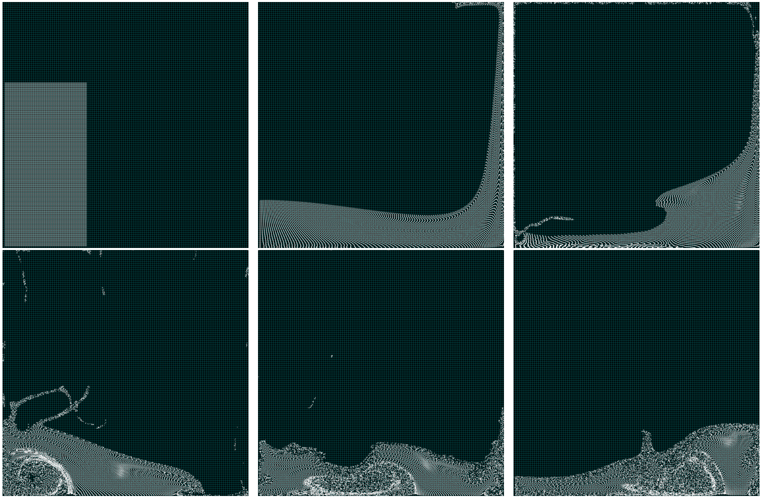


Figure: Dam break scene. Medium resolution, frames 1 to 151.

Result

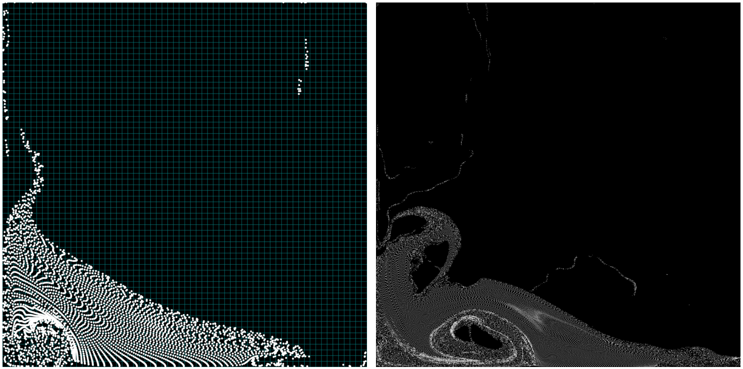


Figure: Dam break scene. Low and high resolution on frame 100

Results

Machine: Intel Core i9-14900KF with 32 GB memory and an NVIDIA GeForce RTX 4080.

GPU	Workload	Total	Total	Solver	Solver	Other	Other
Frame 50	Ratio	Iterations	Times	Time (s)	Ratio	Time (s)	Ratio
Low	1.00	132k	12.36	9.6	1.00	2.76	1.00
Medium	8.00	375k	27.51	24.29	2.53	3.22	1.17
High	64.00	800k	56.46	52.64	5.48	3.82	1.38

- Time increase less than workload increase, suggest our machine scales to larger problems

	CPU				GPU			
Frame 50	Other	Reduce	Saxpy	Matmul	Other	Reduce	Saxpy	Matmul
Low	3.25	7.36	10.27	3.51	2.76	5.1	3.23	1
Medium	4.31	18.33	25.97	8.79	3.22	13.08	8.2	2.57
High	8.11	43.19	60.78	20.89	3.82	28.02	17.74	5.55

- CPU slightly better at reduce, GPU better at all other operations.

Attempts

- Attempted to speed up conjugate gradient algorithm by precomputing indices of all non-empty fluid cells, so that reduce, saxpy, matmul can index into this and avoid empty cells
- Turns out that this leads to slowdowns, significantly increase time on the GPU, and from a slight increase to a slight decrease on the CPU. Generally each of the three operations were affected similarly.
- Explanation? Operations are memory heavy, often streaming. This breaks memory locality and coalescing.
- GPU low: 37% increased solver time. GPU high: 52% increased solver time. GPU low: 12% increased solver time. CPU high: 5% decreased solver time.

Thoughts

- Physics problems are difficult, due to unevenly evolving spatially distributed workload, difficult especially for work balancing.
- Global constrains are difficult
- Solving Poisson problem should be thought not as a linear algebra problem, but a consensus problem. Each worker splats their data to neighborhoods, limited coordination decides their actions, until global consensus is reached.
- Propagating a signal through the network requires as many iterations as grid dimensions
- How can we allocate workers more efficiently than requiring all workers to do the same operations?