Want to see my code? The multi-threaded CPU-based simulator is available here: GitHub repo.

An Interactive Simulator for 2-Dimensional Incompressible Stable Fluids with OpenGL and CUDA

Andrew Blackwell
blackwellaf@appstate.edu
Appalachian State University
(Dated: December 4, 2025)

This report presents the implementation of a two-dimensional incompressible fluid simulator written in CUDA to run in real time on a consumer GPU. After finishing my multi-threaded CPU-based stable-fluids solver over the beginning of winter break, I stumbled upon an NVIDIA RTX 3080 (which tend to be priced reasonably these days) and decided to try a similar hydrophysics project on the GPU and see how far it could be pushed with some creative optimizations. This solver follows Stam's stable fluids method (Stam, 1999): semi-Lagrangian advection on a uniform grid, followed by an iterative pressure projection to enforce incompressibility. OpenGL–CUDA interoperability keeps the velocity and dye fields on the GPU for visualization, while mouse input injects dye and momentum and runtime controls adjust Δt, viscosity, solver iterations, and decay. Results include figures and timing tables, with benchmarks showing large speedups over the earlier CPU version and sustained interactive rates (e.g., 60+ FPS at 1024×1024).

I. INTRODUCTION

Fluid flow phenomena are governed by the Navier–Stokes equations, which, for an incompressible fluid, consist of a momentum conservation equation and a continuity constraint for mass conservation. In continuous form, the equations can be written as:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{F}, \qquad \nabla\cdot\mathbf{u} = 0, \quad (1)$$

where $\mathbf{u}(x,t)$ is the fluid velocity field, $p(x,t)$ is pressure, ρ is fluid density (assumed constant), ν is kinematic viscosity, and $\mathbf{F}(x,t)$ represents external forces (Stam, 1999). The momentum equation captures self-advection, pressure forces, and viscous diffusion; the continuity equation enforces incompressibility by requiring a divergence-free velocity field.

Solving these coupled, nonlinear equations efficiently is difficult in a real-time setting because incompressibility introduces a global dependency: each time step requires a pressure projection, which reduces to solving a Poisson equation over the entire grid. As grid resolution increases, this quickly becomes a memory-bandwidth problem. A 2048×2048 domain contains over four million cells, and a pressure solve that runs for dozens of iterations per frame becomes many full-domain stencil passes.

GPUs are a natural fit for this workload. The simulation step is dominated by dense, regular per-cell operations (advection sampling and stencil updates) with high arithmetic locality and predictable control flow. Compared to a CPU implementation, the GPU can sustain far higher throughput on these kernels by running many threads in parallel and by providing substantially higher memory bandwidth.

This report describes a CUDA implementation of Stam's stable fluids pipeline [1] integrated into an interactive application that allows dye and momentum injection with mouse input and real-time parameter control. The ***Methods*** section will summarize the numerical method (semi-Lagrangian advection and pressure projection). The subsequent ***Implementation*** section will detail the CUDA kernels, data layout, and `OpenGL` interoperability used for real-time visualization. Lastly, the ***Results*** presents performance results across grid sizes and solver settings, followed by a ***Discussion*** on limitations and future work.

FIG. 1. A screenshot of the application running.

II. STABLE FLUIDS ALGORITHM

The simulator uses a grid-based solver for the two-dimensional incompressible Navier–Stokes equations, following the stable fluids approach [1] with operator split-

ting. The domain is discretized into a uniform grid storing velocity and a dye field (a passive scalar used to visualize motion). Time advances in discrete steps, and each step updates these fields through a small set of substeps that isolate the main physical effects: external forcing, advection, diffusion, and pressure projection. This structure keeps the solver stable and makes each stage easy to reason about and implement.

A. External Forces

Each step begins by adding forces into the velocity field. In this application, the dominant forces come from user input: mouse dragging injects momentum (and dye) over a small radius around the cursor. Optional constant forces (e.g., gravity-like terms) fit into the same mechanism. Using explicit Euler integration, the force term updates velocity as

$$\mathbf{u} \leftarrow \mathbf{u} + \Delta t\, \mathbf{F}. \tag{2}$$

B. Advection

Advection transports a quantity along the flow. For velocity, this is the nonlinear self-advection term $(\mathbf{u} \cdot \nabla)\mathbf{u}$; for dye, it simply means the dye rides along with the velocity field. The solver uses semi-Lagrangian advection (Stam, 1999), which traces backward from each cell center to find where the fluid came from over the last time step. For a grid location $\mathbf{x}$,

$$\mathbf{x}_{\text{prev}} = \mathbf{x} - \Delta t\, \mathbf{u}(\mathbf{x}), \tag{3}$$

and the updated field is sampled from the previous field at that departure point,

$$q(\mathbf{x}, t + \Delta t) = q(\mathbf{x}_{\text{prev}}, t). \tag{4}$$

Because $\mathbf{x}_{\text{prev}}$ is generally not on a grid point, sampling uses bilinear interpolation. Semi-Lagrangian advection remains stable for large Δt, but it is numerically diffusive: repeated interpolation gradually smooths fine-scale dye structure and small vortices.

C. Diffusion

Viscosity introduces diffusion through the term $\nu\nabla^2\mathbf{u}$, which damps sharp velocity gradients over time. The diffusion step is treated implicitly to avoid small time-step restrictions. Discretizing diffusion with backward Euler yields a linear system of the form

$$\left(I - \nu\Delta t\, \nabla^2\right) \mathbf{u}_{\text{new}} = \mathbf{u}_{\text{adv}}, \tag{5}$$

where $\mathbf{u}_{\text{adv}}$ is the velocity after advection. The solve is performed with a fixed number of Jacobi iterations, controlled by a runtime parameter. Each iteration replaces a cell's value with a weighted combination of its neighbors and the right-hand side, which gradually relaxes high-frequency components. Increasing viscosity or iteration count produces a smoother, more quickly damped flow; low viscosity preserves sharper vortical motion.

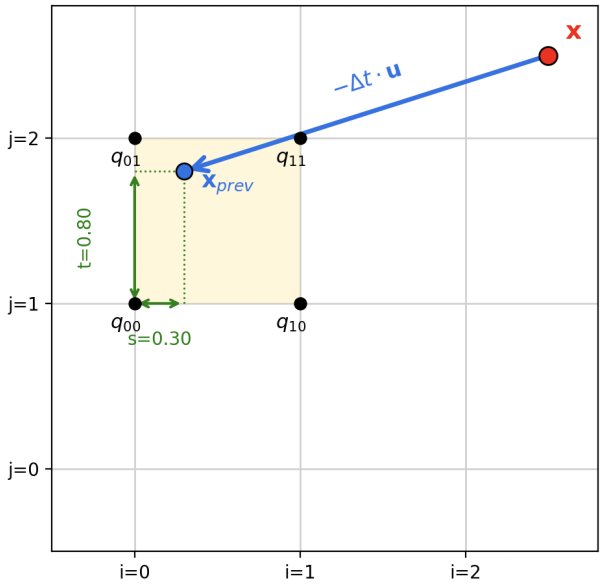

FIG. 2. Semi-Lagrangian backtrace from a cell center $\mathbf{x}$ to a departure point $\mathbf{x}_{\text{prev}}$ and bilinear sampling of the previous field.

D. Pressure Projection

After advection and diffusion, the velocity field is generally not divergence-free. The projection step enforces incompressibility by solving a Poisson equation for a scalar potential whose gradient removes the divergent component of the flow. In continuous form this is often written in terms of physical pressure p as

$$\nabla^2 p = \frac{\rho}{\Delta t}\, \nabla \cdot \mathbf{u}_{\text{new}}, \tag{6}$$

followed by a correction

$$\mathbf{u} \leftarrow \mathbf{u}_{\text{new}} - \frac{\Delta t}{\rho}\, \nabla p. \tag{7}$$

In the implementation, it is more convenient to solve directly for a scaled "pressure-like" field,

$$\tilde{p} \equiv \frac{\Delta t}{\rho}\, p, \tag{8}$$

which yields the equivalent grid-units form

$$\nabla^2 \tilde{p} = \nabla \cdot \mathbf{u}_{\text{new}}, \qquad \mathbf{u} \leftarrow \mathbf{u}_{\text{new}} - \nabla \tilde{p}. \tag{9}$$

With $\Delta x = 1$ in simulation units, the discrete solve uses the standard five-point stencil and Jacobi relaxation. A

moderate iteration count (e.g., 20–50) is sufficient for visually stable incompressible motion, while higher iteration counts reduce residual divergence at a predictable performance cost.

After projection, the velocity field is ready for the next time step. The dye field, after advection, can optionally be decayed each frame to prevent indefinite buildup:

$$d \leftarrow (1 - \lambda \Delta t)\, d, \tag{10}$$

with decay rate λ controlled at runtime.

Boundary conditions close the domain. In this implementation the grid boundary is treated as a solid wall by excluding edge cells from stencil updates and by clamping sampling locations during advection so that back-traced positions remain within the valid interior region. This prevents out-of-bounds reads and keeps dye and momentum contained, but it should be understood as an approximate wall treatment rather than a fully explicit boundary-condition solve. In practice, the boundary handling is implemented implicitly: stencil kernels simply skip the outermost cells, and advection clamps departure points to remain inside the interior. This avoids a dedicated boundary "set" pass and is stable for the interactive use-case here, but it can leave subtle edge artifacts compared to explicitly enforcing boundary values after every stage.

III. IMPLEMENTATION

A. Data Structures and GPU Memory Layout

The simulation state is stored in a small set of 2D grids: velocity in the x and y directions (u and v), pressure p, divergence $\nabla \cdot \mathbf{u}$ (as a temporary for projection), and dye density d. On the GPU these are kept in device global memory as pitched 2D allocations, so each row is aligned for coalesced access. Concretely, each field is allocated once at startup with `cudaMallocPitch`, and the resulting pitch (in bytes) is carried through kernel launches for indexing.

Grid resolution is selected at startup (e.g., 256×256 through 4096×4096). A square domain keeps indexing and visualization simple, but nothing in the layout assumes $N_x = N_y$. Within kernels, a cell (i, j) maps to a linear element index via

$$\texttt{idx} = j \cdot \texttt{pitchElems} + i, \tag{11}$$

where `pitchElems` is the row pitch expressed in elements rather than bytes.

Several steps require reading the previous state while writing a new one (notably advection and the iterative solvers), so the implementation uses ping-pong buffers. Velocity uses either two separate arrays for u and v or a single `float2` array; pressure uses two scalar buffers swapped each Jacobi iteration; dye similarly uses two buffers for advection. This avoids read-after-write hazards and eliminates full-array copies inside tight iteration loops.

Most kernels in the projection and diffusion stages are stencil-based and repeatedly access a cell's immediate neighbors. To reduce redundant global reads, the pressure and diffusion kernels load a tile of the input field into shared memory, including a one-cell halo on each side. Threads then compute their stencil from shared memory and write the updated value back to global memory. This tiling reduces global memory traffic and makes performance less sensitive to cache behavior at larger grid sizes.

B. CUDA Kernels for Simulation Steps

Each stage of the stable fluids pipeline maps onto one or more CUDA kernels launched over a 2D grid of thread blocks. A typical configuration uses 16×16 or 32×32 threads per block, with one thread responsible for one cell update. The CPU side is mostly orchestration: it sequences kernel launches, swaps ping-pong buffers, and runs the iterative solvers for a configured number of sweeps.

Instead of showing a full per-frame step function, Listing 1 shows the core host-side dispatch pattern used throughout the codebase: launch a full-grid kernel, then swap buffers when a stage needs a read-only input and a separate output. The full step is then just these pieces in a fixed order.

```
dim3 block(BLOCK_X, BLOCK_Y);
dim3 grid((width + block.x - 1) / block.x,
          (height + block.y - 1) / block.y);

//advect dye (read prev -> write curr).
std::swap(dye, dye_prev);
advect_dye<<<grid, block>>>(dye, dye_prev, u, v,
     dt, width, height, pitchElems);
```

Listing 1. Host-side dispatch pattern: launch a kernel and swap ping-pong buffers.

Advection is implemented as two kernels (velocity and dye) with the same structure: backtrace from the cell center using the velocity field, clamp the departure point to the valid sampling region, then bilinearly sample the previous field at that location. Listing 2 shows the backtrace and sampling core used by dye advection.

```
int i = (int)(blockIdx.x * blockDim.x +
    threadIdx.x);
int j = (int)(blockIdx.y * blockDim.y +
    threadIdx.y);
if (i >= width || j >= height) return;

int idx = j * pitchElems + i;

float x = (float)i + 0.5f;
float y = (float)j + 0.5f;

float velx = u[idx];
float vely = v[idx];
```

```

float xp = x - dt * velx;
float yp = y - dt * vely;

xp = fminf(fmaxf(xp, 0.5f), (float)width  - 1.5f);
yp = fminf(fmaxf(yp, 0.5f), (float)height - 1.5f);

dyeOut[idx] = bilerp(dyeIn, pitchElems, xp - 0.5f, yp - 0.5f, width, height);
```

Listing 2. Semi-Lagrangian backtrace and bilinear sampling core used for dye advection.

The pressure solve and diffusion solve are both stencil-based Jacobi relaxations. The pressure update follows the standard five-point stencil form,

$$p_{i,j}^{(k+1)} = \frac{1}{4}\left(p_{i-1,j}^{(k)} + p_{i+1,j}^{(k)} + p_{i,j-1}^{(k)} + p_{i,j+1}^{(k)} - \Delta x^2\,(\nabla\cdot\mathbf{u})_{i,j}\right), \tag{12}$$

with $\Delta x = 1$ in simulation units. Each Jacobi sweep is one kernel launch over the grid, followed by a buffer swap. Listing 3 shows a single pressure sweep kernel.

```
int i = (int)(blockIdx.x * blockDim.x + threadIdx.x);
int j = (int)(blockIdx.y * blockDim.y + threadIdx.y);
if (i <= 0 || j <= 0 || i >= width - 1 || j >= height - 1) return;

int idx = j * pitchElems + i;

float pL = pPrev[j * pitchElems + (i - 1)];
float pR = pPrev[j * pitchElems + (i + 1)];
float pB = pPrev[(j - 1) * pitchElems + i];
float pT = pPrev[(j + 1) * pitchElems + i];

pOut[idx] = 0.25f * (pL + pR + pB + pT - div[idx]);
```

Listing 3. One Jacobi sweep for the pressure Poisson solve.

After pressure is computed, projection subtracts the discrete pressure gradient from velocity using centered differences in the interior. Listing 4 shows the per-cell projection update.

```
int i = (int)(blockIdx.x * blockDim.x + threadIdx.x);
int j = (int)(blockIdx.y * blockDim.y + threadIdx.y);
if (i <= 0 || j <= 0 || i >= width - 1 || j >= height - 1) return;

int idx = j * pitchElems + i;

float pL = p[j * pitchElems + (i - 1)];
float pR = p[j * pitchElems + (i + 1)];
float pB = p[(j - 1) * pitchElems + i];
float pT = p[(j + 1) * pitchElems + i];

u[idx] -= 0.5f * (pR - pL);
v[idx] -= 0.5f * (pT - pB);
```

Listing 4. Velocity projection: subtract discrete pressure gradient (interior cells).

C. GPU–OpenGL Interoperability and Visualization

Rendering the simulation in real time is part of the point: without a fast visualization path, the solver turns into an offline batch job. The application uses `OpenGL` for rendering and `SDL2` for windowing and input. CUDA and `OpenGL` are connected through a pixel buffer object (PBO) registered with CUDA via `cudaGraphicsGLRegisterBuffer`. Each frame, the PBO is mapped to a device pointer, a CUDA kernel writes a colorized image from the dye (and optionally velocity) fields directly into that buffer, and the buffer is then unmapped and presented as a fullscreen textured quad. This keeps the entire pipeline GPU-resident and avoids readbacks to CPU memory.

The default visualization maps dye concentration to a simple false-color ramp (low values dark, higher values brighter). Velocity visualization is useful during debugging, so there is an option to render either velocity magnitude or a sparse vector overlay, but dye is the primary display mode for interaction and demos.

Runtime controls are provided through `Dear ImGui`, integrated into the same `SDL2/OpenGL` context. The UI exposes the parameters that matter during experimentation: Δt, viscosity, pressure iteration count, dye decay, and force/dye injection strength. Reset controls clear dye and/or velocity so the simulation can be restarted without restarting the program.

Mouse input drives the interactive part. When the left mouse button is held and dragged across the domain, the code injects momentum in the direction of motion and injects dye at the cursor position. The CPU collects the cursor position and motion each frame, and a small "splat" kernel applies these impulses over a configurable radius in the next external-forces stage. The result feels like painting: drawing quickly produces strong vortices, while slower drags produce smooth stirring.

IV. RESULTS

TABLE I. Time scaling per one frame (forces, advection, diffusion, projection, visualization) and measured speedup.

Grid	Cells	CPU (ms)	GPU (ms)	Speedup
512×512	262,144	55.7	2.49	22.4
1024×1024	1,048,576	317.4	9.87	32.2
2048×2048	4,194,304	1692.8	39.5	42.9
4096×4096	16,777,216	7641.3	157.6	48.5

TABLE II. GPU time breakdown per frame at 1024×1024 (mean over 1000 frames).

Stage	Time (ms)	Share (%)
Advection (velocity + dye)	1.62	16.4
Diffusion (16 iters)	2.78	28.2
Projection (div + pressure + subtract)	4.96	50.3
Visualization (PBO write + draw)	0.51	5.2
Total	9.87	100.0

The CUDA-based simulator produces visually plausible incompressible flow and remains interactive on a consumer GPU. Correctness was checked qualitatively in the ways that matter for an interactive stable-fluids solver: dye does not exhibit obvious volume gain/loss beyond the configured decay, stirring produces coherent vortices that advect dye as expected, and motion damps out under viscosity when input stops. The pressure projection is the difference between "fluid" and "broken": disabling projection causes the velocity field to accumulate divergence and visibly inflate/expand, while enabling projection keeps motion bounded and visually incompressible. Pressure iteration count controls how tight the projection is. Around 40 iterations produces a result that looks effectively divergence-free in motion, while very low counts (e.g., 10) can leave mild compressibility artifacts under strong forcing.

To quantify performance, the GPU implementation was benchmarked against a comparable CPU baseline across multiple grid sizes. Table I reports end-to-end frame time for the full interactive pipeline (forces, advection, diffusion, projection, and visualization) using the same solver settings across grid sizes. Table II breaks down the 1024×1024 GPU entry in Table I into per-stage costs, showing that projection dominates the frame time. Diffusion uses 16 Jacobi iterations and pressure uses 40 iterations.

A breakdown of GPU time by substep was also measured for a 1024×1024 run (averaged over a long interactive capture). With diffusion enabled at 16 Jacobi iterations and pressure at 40 iterations, advection (velocity + dye) took about 1.6 ms, diffusion about 2.8 ms, projection (divergence, pressure solve, gradient subtraction) about 5.0 ms, and visualization about 0.5 ms, for a total of roughly 9.9 ms per frame ($\sim$ 101 FPS). As expected, the pressure solve dominates, as it performs many full-grid stencil passes and tends to be bandwidth-limited. The advection stage is comparatively cheap, and rendering overhead stays small because the visualization path remains GPU-resident.

ACKNOWLEDGMENTS

In writing this software, I was continually inspired by the creations of the Open-Source community, particularly those who share their creations on Github and ShaderToy, as well as the innumerable online resources for learning computer graphics.

[1] J. Stam, Stable fluids, in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)* (ACM Press/Addison-Wesley, 1999).

[2] M. Harris, Fast fluid dynamics simulation on the gpu, in *GPU Gems*, edited by R. Fernando (Addison-Wesley, Boston, MA, 2004) Chap. 38.

[3] R. Fedkiw, J. Stam, and H. W. Jensen, Visual simulation of smoke, Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01) (2001).

[4] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Communications of the ACM **52** (2009).

[5] *CUDA C++ Programming Guide*, NVIDIA Corporation (2024), nVIDIA Developer Documentation.

[6] R. Fernando, ed., *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics* (Addison-Wesley, Boston, MA, 2004).

[7] A. Blackwell, fluid-sim: Multithreaded cpu-based stable fluids simulator (source code), `https://github.com/AndrewBlackwell/fluid-sim/tree/main` (2025), gitHub repository. Accessed 2025-12-04.