Parallelization of Dynamic Fluid Simulation

Michael Dushkoff, Jou-Chi Huang, Emad Taghiye, Eric Zander

March 17, 2025

1 Introduction

1.1 Problem Domain

Fluids are ubiquitous in our physical world, from flowing waterfalls, the air particles you breathe, to smoke and clouds. These phenomena have been of particular interest in the field of computer graphics and physical simulation for decades[4][9][6][3][2][8]. Simulating fluids realistically and in real-time is a computational challenge which fortunately can take advantage of the parallelism offered by GPUs.

Fluid simulations focus on computing the famous Navier-Stokes equations which describe the motion of particles within a domain which interact and influence each other's velocity and pressure[1]. These partial differential equations describe how Newtonian fluids conserve mass, and their relationship of pressure, density, and momentum. When combined with other forces such as those produced by thermodynamical simulations, very complex physical systems can be modeled with decent accuracy [4]. These types of simulations are commonly computed on a fixed grid of points to determine the vector field of the overall fluid velocity at each point, and the density and pressure of fluid flowing in each region of space, a mapping that works well with the SIMD hardware paradigm in GPU devices.

1.2 Motivations for Choosing Area

We all agreed that learning CUDA would be beneficial for expanding our GPU programming skills and enhancing our future career opportunities. While exploring applications on NVIDIA's website, we discovered that fluid simulation is a possible use case for GPU acceleration.

Some of us have a background in electrical engineering and are particularly interested in how PDE and ODE solvers can be efficiently utilized to address fluid simulation problems. Additionally, we aim to introduce a more innovative approach by integrating a probabilistic, real-time, and interactive fluid simulation using the Metropolis-Hastings algorithm, distinguishing our work from traditional deterministic methods.

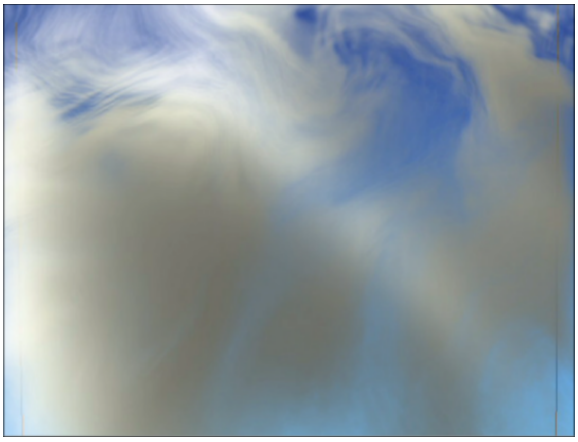

Figure 1: Fluid Simulation Applied to Modeling Clouds[8]

If successful, we believe this project could have practical applications in movie visual effects, weather forecasting, and aerodynamics simulations for vehicle and aircraft design.

2 Project Definition

This project develops a dynamic fluid simulation optimized for GPU parallelization. It begins with sequential initialization on the CPU before shifting to CUDA for parallel execution. Using an Eulerian grid, it models fluid behavior by solving the Navier-Stokes equations, capturing velocity, pressure, advection, diffusion, and external forces at each time step. The focus is on efficient numerical methods to update fluid states while maintaining real-time performance. CUDA acceleration enables large-scale parallelization, optimizing computations across velocity and pressure grids. The goal is to create a high-performance fluid simulation capable of interactive visualization while ensuring computational efficiency and physical accuracy.

3 Methodology

Fluid simulation relies on three key components: advection, diffusion, and pressure projection. These steps are derived from the Navier-Stokes equations, which describe the motion of fluids by expressing the principles of momentum and mass conservation. In their incompressible form, the Navier-Stokes equations are written as:

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

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

The first equation represents the conservation of momentum, describing how the velocity field $\mathbf{u}$ evolves over time under the influence of advection, pressure gradients, viscosity, and external forces. The second equation enforces the incompressibility condition by requiring that the divergence of the velocity field be zero, meaning the fluid maintains a constant density [7, 5].

Each stage of the simulation corresponds to solving or approximating parts of these equations. Advection transports quantities like velocity and color through the fluid, allowing it to move and carry substances along its flow. Diffusion accounts for the spreading and mixing of these quantities, simulating viscosity and energy dissipation over time. Finally, pressure projection ensures the fluid remains incompressible by enforcing a divergence-free velocity field, maintaining realistic and stable fluid behavior. Together, these steps form the foundation of fluid simulations.

3.1 Advection

Advection is responsible for transporting quantities such as velocity and color through the fluid domain over time. In a fluid simulation, this step updates fields by moving them along the current velocity field. It models the natural behavior of fluids carrying their properties—whether mass, momentum, or color—along their flow. Advection is crucial for capturing the motion of the fluid and ensuring that transported quantities evolve consistently within the simulation domain.

3.1.1 Mathematical Foundation

Advection originates from the material derivative in the Navier-Stokes equations, representing the change of a quantity as it moves through space and time with the fluid. To compute advection, we use the semi-Lagrangian method, which traces each point in the grid backwards in time along the velocity field. This backtracking identifies the location where the quantity came from at the previous timestep. In other words, for a given point $\mathbf{x}$, we determine its previous position $\mathbf{x}_{\text{prev}}$ by subtracting the velocity at that point scaled by the timestep:

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

Once we have $\mathbf{x}_{\text{prev}}$, we sample the quantity we are advecting from that location. Since $\mathbf{x}_{\text{prev}}$ typically lies between grid points, we use bilinear interpolation to compute the quantity's value at that position:

$$q(\mathbf{x}) = \text{BilinearInterpolate}(q, \mathbf{x}_{\text{prev}}) \tag{4}$$

This process is repeated for every grid point, effectively transporting the field quantities along the velocity field.

3.1.2 Notes on Stability and Performance

The semi-Lagrangian method used for advection is unconditionally stable, allowing us to use larger timesteps without causing instability. However, this method introduces numerical dissipation, which tends to smooth out fine details in the velocity or color fields over time. In practice, this trade-off is acceptable in real-time fluid simulations, where stability and speed are prioritized over perfect accuracy.

In terms of implementation, efficient memory access and careful handling of boundary conditions are essential for achieving high performance and maintaining accuracy. Wrapping boundaries are used in our simulation to simplify backtracking and interpolation, but clamping or other boundary handling methods can be applied depending on the requirements of the simulation.

3.2 Diffusion

Diffusion simulates the internal friction of a fluid, commonly referred to as viscosity. It represents the tendency of velocity differences within the fluid to smooth out over time as momentum spreads from faster-moving regions to slower ones. In the context of the Navier-Stokes equations, diffusion corresponds to the viscosity term:

$$\nu \nabla^2 \mathbf{u} \tag{5}$$

where ν is the kinematic viscosity coefficient, and $\nabla^2 \mathbf{u}$ is the Laplacian of the velocity field. This term models the transfer of momentum due to molecular interactions in real fluids, causing energy dissipation and smoothing sharp velocity gradients.

In numerical simulations, diffusion is responsible for stabilizing the velocity field by damping out high-frequency fluctuations. Without diffusion, the simulation can produce unrealistic, overly turbulent motion, especially when working with coarse grids or large timesteps. The diffusion step ensures energy spreads smoothly throughout the fluid, preserving physical realism.

3.2.1 Mathematical Foundation

To account for diffusion, one may solve the diffusion equation for velocity. Given constant kinematic viscosity ν:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u} \tag{6}$$

This represents a Poisson equation that is most efficiently solved with iterative methods rather than prohibitively slow direct methods. Here we use Jacobian iteration. Given that $\alpha = \nu \Delta t$ and $\beta = 1 + 4\alpha$:

$$u_{i,j}^{n+1} = \frac{\alpha \left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} \right) + u_{i,j}^{n}}{\beta} \tag{7}$$

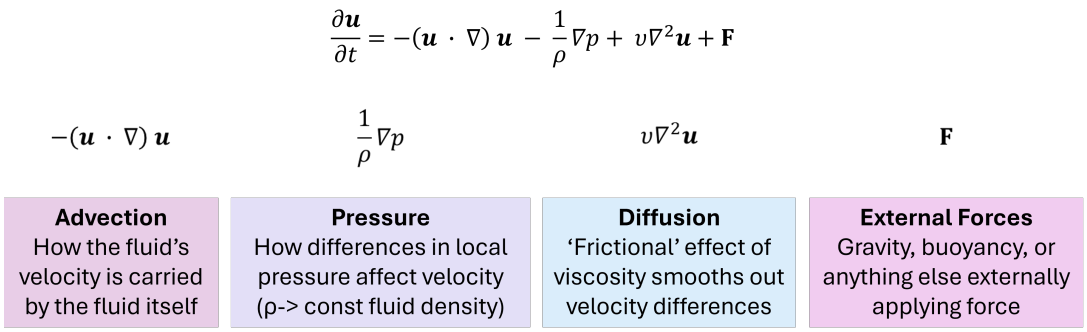

Figure 2: Breakdown of velocity Navier-Stokes equation for incompressible flow.

3.3 Pressure

Enforcing incompressibility is a fundamental requirement in fluid simulation, ensuring that the fluid maintains a constant density throughout its motion. This is typically achieved by computing a pressure field that projects the velocity field onto a divergence-free space. The pressure calculation stage consists of two key mathematical operations: solving the pressure Poisson equation and subtracting the pressure gradient from the velocity field.

3.3.1 Mathematical Foundation

After applying external forces and advection, the intermediate velocity field $\mathbf{u}^*$ may no longer be divergence-free. To restore incompressibility, we solve for a scalar pressure field p that corrects the velocity field:

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \nabla p \tag{8}$$

Taking the divergence of both sides and enforcing $\nabla \cdot \mathbf{u}^{n+1} = 0$, we obtain the Poisson equation for pressure:

$$\nabla^2 p = \nabla \cdot \mathbf{u}^* \tag{9}$$

Once the pressure field is computed, the velocity field is corrected by subtracting the pressure gradient:

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \nabla p \tag{10}$$

In practice, we discretize these equations on a regular grid. The discrete Laplacian in two dimensions is approximated as:

$$\nabla^2 p_{i,j} \approx p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j} \tag{11}$$

Similarly, the divergence and gradient operations are computed using finite differences. Solving the Poisson equation for p typically involves iterative methods such as the Jacobi iteration:

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

3.3.2 Discussion

The pressure calculation step is crucial for enforcing incompressibility in fluid simulation. The CPU and GPU implementations differ in their execution model but follow the same mathematical principles. While the CPU implementation is easier to understand and debug, the GPU version offers a substantial performance advantage, especially for large grids.

More advanced solvers, such as Multigrid methods, can accelerate convergence but require more complex implementation and data structures, particularly on GPUs.

4 Implementation Details

The following concerns details relevant to the specific implementation of the workflow discussed previously.

4.1 Memory Layout

We effectively represented velocity and pressure fields as interlaced images where channels correspond to the velocities horizontal and vertical components and pressure respectively. By keeping elements from each channel close, we take advantage of locality.

4.2 Serial vs. Parallel Implementation

4.2.1 CPU Implementation

The CPU implementation of pressure calculation follows the mathematical formulation closely. The main steps are as follows:

1. **Advection Computation**: Iterate through all the cells in the 2D grid. Use the Semi-Lagrangian Method to trace each cell back along the velocity field to determine where the fluid originated. Find the nearest integer points surrounding the origin, then perform bilinear interpolation to blend the velocity from the four neighboring grid points using the corresponding weights. Additionally, to enable visualization and testing, we implemented a similar function to transport the image's color field based on a velocity field over time, allowing us to observe changes in uploaded images.

2. **Diffusion**: Compute and add the diffusion term to the intermediate velocity field using Jacobi iteration.

3. **Divergence Computation**: The divergence of the intermediate velocity field $\mathbf{u}^*$ is computed at each grid cell. This divergence is the right-hand side of the Poisson equation.

4. **Pressure Solve**: The pressure field is computed iteratively using Jacobi iteration. Each iteration updates the pressure at every grid cell based on its immediate neighbors and the divergence value.

5. **Subtract Pressure Gradient**: After convergence, the velocity field is corrected by subtracting the pressure gradient, ensuring incompressibility.

This process is implemented as a series of loops over the grid. Data is stored in arrays, and buffer swapping is used between iterations to avoid read-write conflicts. The Jacobi solver iterates over the grid multiple times to approximate the solution.

4.2.2 GPU Implementation

The GPU implementation is conceptually identical but optimized for parallel execution. Each thread in a CUDA kernel computes the result for a single grid cell, enabling massive parallelism. The GPU pipeline consists of:

1. **Advection Computation**: Implemented as `advection` and `advection_color` kernels, computing each cell's velocity in parallel while ensuring the results match those of the CPU version.

2. **Divergence Computation**: Implemented as the `computeDivergence` kernel, this step computes the divergence in parallel for each grid cell.

3. **Pressure Solve**: The Jacobi iteration is performed in parallel using the `computePressure` kernel. The host code launches this kernel multiple times to perform the iterative solve, swapping buffers between iterations.

4. **Subtract Pressure Gradient**: The `subtractPressureGradient` kernel corrects the velocity field by subtracting the computed pressure gradient, again in parallel for each grid cell.

The GPU implementation achieves significant speedup over the CPU by distributing computations across thousands of threads. Efficient memory access patterns and synchronization mechanisms are essential to maintaining performance and accuracy. We made use of CUDA pinned memory on the host in order to prioritize device to host transfers and reduce the total latency.

5 Results

The analysis presented here is based on the results obtained from our completed implementation. We categorize the image sizes into a range from 64 to 2048 pixels, referred to as "velocity field" sizes for clarity. This terminology is employed to better conceptualize the results, where the images are translated into velocity values.

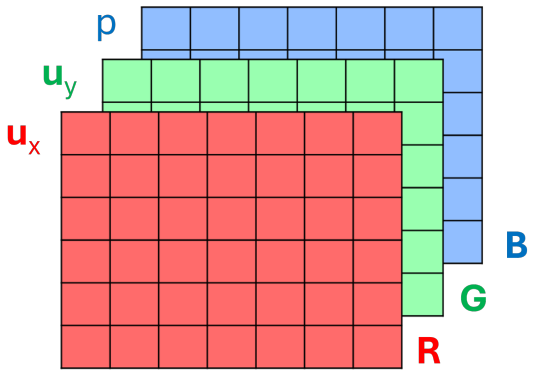

Figure 3: Illustration of memory layout. Texture channels are used for components of velocity and pressure fields. The interlaced representation ensures channel values are close in memory and supports cache coherence.

As evidenced by *Table 1*, which shows CPU execution time, and *Table 2*, which shows GPU execution time, the GPU consistently outperforms the CPU across all image sizes. As the image size increases exponentially with a base of 2, it is observed from *Table 3* and *Figure 4* that the CPU's execution time grows at a faster rate compared to the GPU. Specifically, when the image size reaches 1024 pixels, the GPU's execution time increases by a factor of 5.47 compared to 512 pixels, yet it remains more efficient than the CPU. In contrast, the CPU's execution time increases by a factor of 3.85 for the same image size transition.

Furthermore, *Figure 5* presents a speedup ratio comparison, calculated by dividing the CPU execution time by the GPU execution time. This comparison illustrates that the performance gap between the CPU and GPU is most pronounced when the image size is 512 pixels. However, as the image size increases to 1024 pixels, the gap begins to narrow. Although our experiments did not include image sizes larger than 2048 pixels due to limitations in running such large images on the CPU, it is anticipated that beyond a certain threshold, the performance difference between the CPU and GPU will diminish.

To evaluate the behavior of the fluid simulation under different conditions, we present two sets of visual results.

The first set, shown in Figure 6, demonstrates the simulation applied to three different input images. This visualization highlights how the velocity field transports color data over time, producing natural and continuous motion regardless of the underlying texture. Each example illustrates how the advection step carries the image content through the flow field, demonstrating the stability and consistency of the simulation across varying inputs.

The second set, shown in Figure 7, provides a side-by-side comparison of simulations with different viscosity values. We simulate fluids with viscosities of 0.00005, 0.00002, and

Processor	V_field	Average_Output
CPU	PerlinRG_64.png	8,463,260 us
CPU	PerlinRG_128.png	14,825,840.5 us
CPU	PerlinRG_256.png	40,334,248.5 us
CPU	PerlinRG_512.png	142,020,757.75 us
CPU	PerlinRG_1024.png	547,186,551.5 us
CPU	PerlinRG_2048.png	2,261,300,454 us

Table 1: CPU Performance Data

Processor	V_field	Average_Output
GPU	PerlinRG_64.png	40,172.25 us
GPU	PerlinRG_128.png	45,119.5 us
GPU	PerlinRG_256.png	67,229 us
GPU	PerlinRG_512.png	140,852.5 us
GPU	PerlinRG_1024.png	770,894.5 us
GPU	PerlinRG_2048.png	2,943,143.75 us

Table 2: GPU Performance Data

Image Size Transition	CPU Increase Rate	GPU Increase Rate
64 to 128	1.75x	1.12x
128 to 256	2.72x	1.49x
256 to 512	3.52x	2.10x
512 to 1024	3.85x	5.47x
1024 to 2048	4.13x	3.82x

Table 3: Increase Rate Between Consecutive Image Sizes

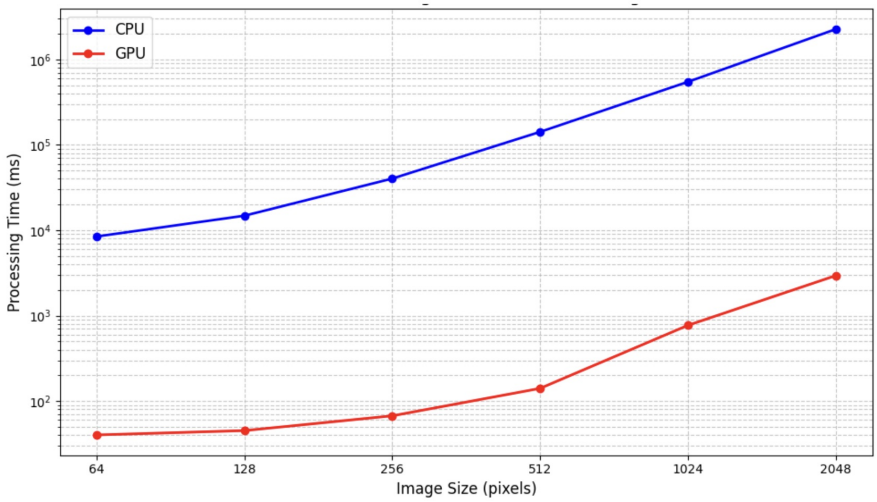

Figure 4: CPU vs GPU Processing Time for Different Image Sizes

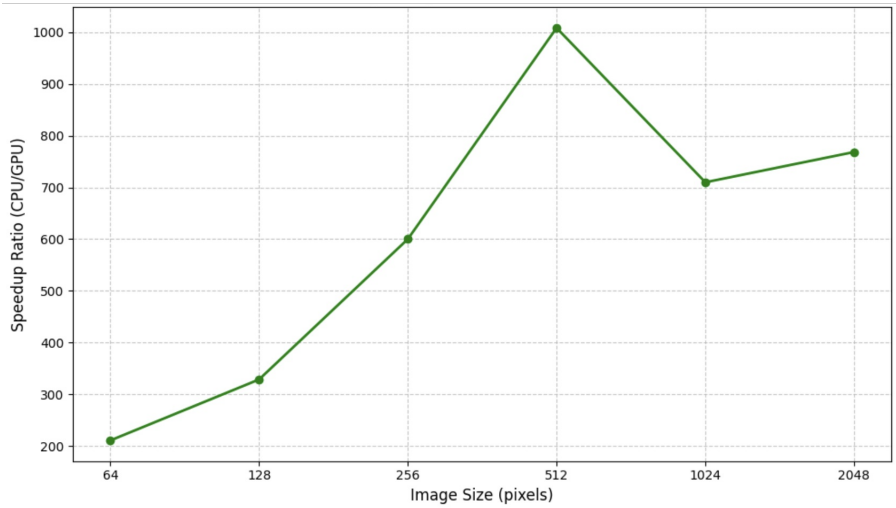

Figure 5: Speedup Ratio of CPU/GPU

Figure 6: Demonstration of fluid simulation applied to three different images.

0, shown from left to right. As expected, higher viscosity produces smoother motion by increasing diffusion and damping out high-frequency variations in the velocity field. In contrast, zero viscosity retains sharper details in the flow, resulting in more turbulent and potentially chaotic behavior.

Together, these visualizations demonstrate the flexibility of the simulation in handling different visual inputs and physical parameters, validating both the accuracy and stability of the system.

Figure 7: Side-by-side comparison of fluid simulations with viscosities of 0.00005, 0.00002, and 0, shown from left to right.

6 Conclusions and Future Directions

We have developed a minimum viable product (MVP) for a GPU-accelerated dynamic fluid simulation system that uses CUDA to solve Partial Differential Equations (PDEs). A CPU-based implementation has also been developed for comparison.

Future research could focus on scaling the project. While the current implementation works with 2D images, it could be expanded to 3D or 4D simulations. Additional effects, such as simulating fog and clouds, could enhance scene diversity and realism. As computational demands grow, GPU acceleration becomes even more essential.

Another area for improvement is enabling the generation of short video clips. Currently, the system produces individual images, but adding video output would increase accessibility and versatility, making it suitable for applications in movie scenes or personal vlogs.

Finally, not only did this project offer valuable experience in programming in CUDA for GPUs, but also in handling differential and Poisson equations in the context of parallel scientific computing.

References

[1] Alexandre Joel Chorin, Jerrold E Marsden, and Jerrold E Marsden. *A mathematical introduction to fluid mechanics*. Vol. 3. Springer, 1990.

[2] Jesús Ojeda Contreras. "Efficient algorithms for the realistic simulation of fluids". In: *Efficient algorithms for the realistic simulation of fluids*. Universitat Polit'ecnica de Catalunya, 2013-02.

[3] Nolan Goodnight et al. "A multigrid solver for boundary value problems using programmable graphics hardware". In: *ACM SIGGRAPH 2005 Courses*. 2005, 193–es.

[4] Mark J Harris et al. "Simulation of cloud dynamics on graphics hardware". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2003, pp. 92–101.

[5] Mark J. Harris. "Fast Fluid Dynamics Simulation on the GPU". In: *GPU Gems*. Ed. by Randima Fernando. Addison-Wesley, 2004. Chap. 38, pp. 637–665.

[6] Matthias Müller, David Charypar, and Markus Gross. "Particle-based fluid simulation for interactive applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Citeseer. 2003, pp. 154–159.

[7] Shahriyar Shahrabi. *A Gentle Introduction to Fluid Simulation for Programmers and Technical Artists*. `https://shahriyarshahrabi.medium.com/gentle-introduction-to-fluid-simulation-for-programmers-and-technical-artists-7c0045c40bac`. 2021.

[8] Jos Stam. "Stable fluids". In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 2023, pp. 779–786.

[9] Enhua Wu, Youquan Liu, and Xuehui Liu. "An improved study of real-time fluid simulation on GPU". In: *Computer Animation and Virtual Worlds* 15.3-4 (2004), pp. 139–146. DOI: `https://doi.org/10.1002/cav.16`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cav.16`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.16`.