CUDA SPH Fluid Pouring Simulation: 418 Final Project

Yashoditya Watal Athan Ferber

April 30, 2026

1 Summary

We built a 3D smoothed particle hydrodynamics (SPH) fluid pouring simulator, with a sequential baseline and 3 CUDA GPU versions: brute force, spatial hashing, and neighbor lists. We used these different methods to study the performance tradeoffs of irregular particle workloads during pouring. Our deliverables are a working cup to cup pouring simulation, performance comparisons across these parallelization strategies, and an analysis of limitations within those strategies. For the poster session we will show a live pouring demo with adjustable parameters, along with timing breakdowns and speedup graphs. All experiments were run on the GHC machines using NVIDIA RTX 2080 GPUs.

2 Background

What is SPH for Liquid pouring

Our project studies a three dimensional smoothed particle hydrodynamics, or SPH, simulation for liquid pouring from one cup into another. In SPH, the fluid is represented as a collection of particles, where each particle stores physical quantities such as position, velocity, force, density, and pressure. The simulation also includes boundary particles that represent the walls of the source and receiver cups. The main inputs to the program are the initial fluid configuration, the geometry and position of the cups, the source cup tilt schedule (when the cup starts and stops tilting), and simulation constants such as timestep, smoothing radius, viscosity, and rest density. The output is a sequence of particle states over time, which we export as CSV frames and visualize in a software called ParaView.

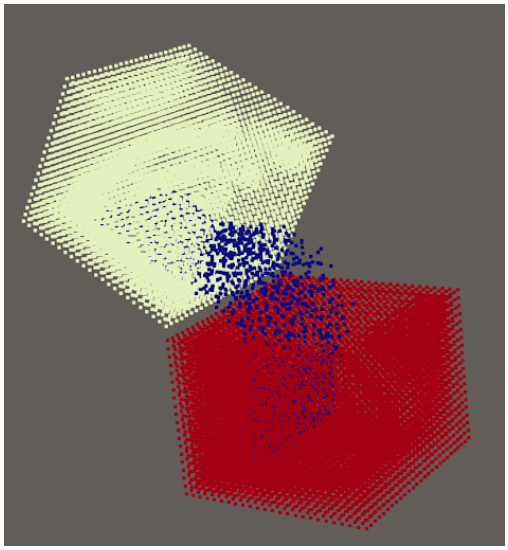

Figure 1: Liquid particle and Cup setup

Each simulation step consists of several stages. For each particle at every substep:

1. The simulator determines which particles are close enough to influence it (based on a tunable radius).
2. It computes density and pressure by summing effects from the nearby fluid and boundary particles.

3. It computes forces, including pressure, viscosity, gravity, and interactions with the container walls.

4. Finally, it updates particle velocities and positions and applies collision handling to keep the fluid inside the cup bounds.

These stages are repeated many times per time step (called a substep, which is another tunable parameter) in order to keep the motion stable and accurate. The main data structures are therefore the fluid particle array, the source and receiver boundary particle arrays, and the respective neighbor search structure, such as a spatial grid or a neighbor list, depending on the implementation, used to decide which particles should interact. (we talk about these below). The operations on these structures are described in each stage above, and include, searching for nearby particles, density, pressure, and force computations, as well as collusion handling.

Why SPH benefits from parallelization

The expensive part of SPH is neighbor interaction. In the simplest version, each fluid particle checks every other fluid particle, and also checks the boundary particles, when computing density and forces. This gives a cost that grows quadratically with the number of particles, which becomes expensive very quickly even for moderate problem sizes. In practice, this makes the density and force stages the dominant cost, since both repeatedly evaluate particle interactions over the entire span of particles, for each particle. Our milestone results also showed that the sequential version slowed down rapidly as particle count increased, which made it clear that a parallel solution was needed.

This workload is a natural fit for CUDA because much of the computation is data parallel. Density can be computed independently for each particle once the positions are known, and force can also be computed independently for each particle once density and pressure have been computed. This means we can assign particles to GPU threads and perform many of these calculations at the same time. The main dependency is between stages, not within a stage. For example, the force pass depends on the density and pressure pass having finished, and the integration pass depends on the force values being available, but within each pass there is substantial parallel work across particles. This gives a clear per particle parallel structure.

At the same time, SPH is not perfectly uniform. Pouring creates thin streams, dense pools, splashing regions, and complicated boundary interactions at the lip of the cup and inside the receiving container. As a result, some particles have many neighbors while others have very few, so the amount of work per thread is irregular. Memory access is also irregular, because nearby particles in space are not always stored near one another in memory. This limits locality and makes the workload less regular than a simple dense array computation. The simulation is still strongly data parallel overall, and the large number of particles makes it well suited to SIMT execution on the GPU. These were exactly the issues we set out to study in the proposal, especially load imbalance, irregular memory behavior, and the effect of neighbor search structure on performance.

Tthe simulation exposes substantial parallelism because each stage performs similar work over all fluid particles, so the amount of available parallel work scales directly with particle count. This makes the computation naturally data parallel which lends to SIMD style execution, although irregular neighbor counts and scattered memory access reduce how efficiently that parallelism can be used in practice.

Parallelization Strategies we explored

Our first GPU version was a basic CUDA baseline in which we moved the density and pressure computation into one CUDA kernel and the force computation into a second CUDA kernel. These were the two most expensive stages, and gave us an immediate speedup over the sequential version. The implementation, however, still used direct brute force neighbor checks, so every particle was still compared against a very large number of fluid and boundary particles. We also considered moving timestep integration to the GPU, but timing results showed that integration was only a small part of the total runtime compared to density and force, so the main benefit came from parallelizing those interaction heavy stages first, while leaving integration on the CPU.

To reduce that cost, we explored spatial hashing as a first optimization to the base CUDA implementation. The basic idea is to divide space into grid cells, place particles into cells based on position, and then only check particles in the local and nearby cells rather than scanning the entire particle set when running a computation for each particle. In principle this should remove much of the wasted work from distant particles that cannot influence one another, but in practice we found that rebuilding the grid every substep introduced a large overhead of its own, so our first spatial hash version actually performed worse than the base CUDA implementation. We then refined the method with several implementation changes, including separating boundary particles used for physics from the denser boundary particles used only for rendering, rebuilding the moving source cup boundary every substep while building the static receiver cup boundary only once, and extending the hashed lookup structure to cover fluid to boundary interactions as well as fluid to fluid interactions (This was the biggest one). After these optimizations, the spatial hash version kept the same visual behavior while outperforming the original brute force CUDA baseline.

We also implemented a neighbor list approach as an alternative CUDA implementation to spatial hashing. In this method, each particle stores a list of nearby fluid and boundary particles, and the density and force passes reuse those saved lists instead of searching again from scratch. The advantage here is that density and force become cheaper once the lists exist. The downside is that building the lists is itself expensive, especially if it is done every substep. For that reason, neighbor lists are most useful when the lists can be reused for several substeps before rebuilding. This gave us a second search strategy to compare against spatial hashing, and it let us study a different performance tradeoff, namely paying more upfront search cost in exchange for cheaper repeated computation later.

Overall, these strategies let us examine the same SPH simulation under several different parallel strategies. The brute force CUDA baseline shows what can be gained from straightforward particle level parallelism. Spatial hashing shows how locality aware neighbor search can reduce unnecessary interaction work. Neighbor lists shows an alternative approach where nearby interactions are cached and reused. Together, these methods form the core of our performance study and help explain where SPH gains speed on the GPU and where irregular particle behavior still creates bottlenecks.

3 Approach

Baseline Simulation and implementation setup

We built the simulator in C++ with CUDA support and ran all experiments on the GHC machines using NVIDIA RTX 2080 GPUs. Before focusing on parallelization, we developed a sequential baseline, since getting the fluid behavior correct was necessary before any performance analysis would be meaningful. The simulator models a cup to cup pouring setup, where fluid particles begin inside a source cup which tilts at a preset time, and the fluid falls into the receiving cup. We

represent the fluid as an array of particles, with each particle storing values like position, velocity, force, density, and pressure, and we represent the cup walls with boundary particles so that the liquid can interact with the container geometry during settling and pouring.

A large part of the early work was devoted to building a stable baseline rather than parallel optimization. The simulator advances in discrete frames, with each frame broken into multiple smaller "substeps" so that density, pressure, force, integration, and boundary handling can be updated at a finer granularity. We implemented the core SPH stages: density and pressure computation, force computation, timestep integration, and boundary collision handling, and we added a CSV export pipeline that writes out the particle state every frame so that the simulation could be inspected in ParaView. This viewing pipeline was important for debugging because many issues were easier to detect visually than numerically, especially problems such as particles starting outside the source cup, unrealistic clumping, unstable settling, and incorrect interaction with the cup walls. A substantial amount of time went into tuning parameters such as mass, rest density, viscosity, timestep, and boundary behavior so that the fluid began in a reasonable initial state and poured in a visually believable way.

We also referenced the physics and logic from an outside reference codebase by vagifaliyev [1] for a basic SPH solver, but our final setup differed substantially because we extended it into a 3D cup pouring system with moving boundaries, a receiver cup, configurable tilt and fill conditions, collision handling, and a CSV based visualization workflow. The reference was a starting point for the basic SPH structure.

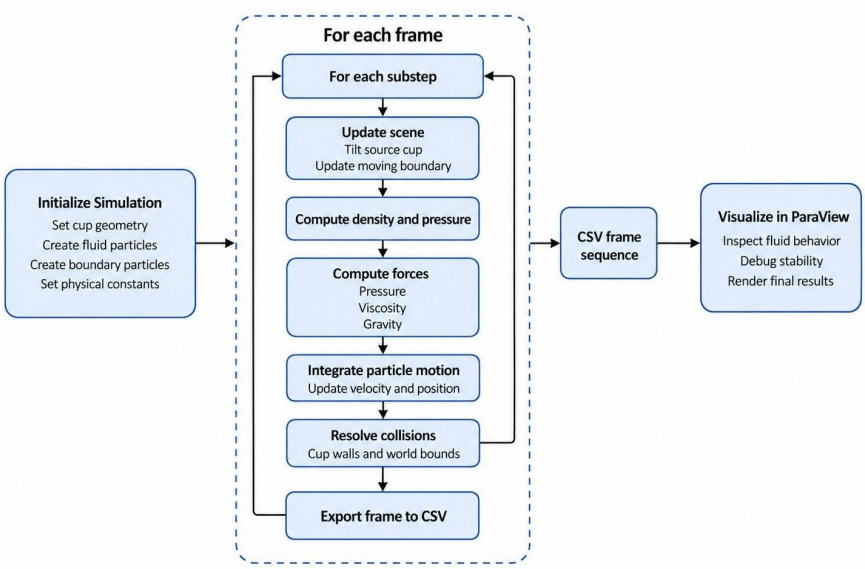

Figure 2: Simulation to Visualization Pipeline

Baseline CUDA

Our first parallel version was a basic CUDA baseline that targeted the two most expensive SPH stages, density and pressure computation, and force computation. We implemented these as two separate CUDA kernels, with one thread assigned to each fluid particle, so that all fluid particles could be processed in parallel within a stage. In the density and pressure kernel, each thread computed the local density and pressure of one particle by checking interactions with all other fluid and boundary particles. In the force kernel, each thread then computed the pressure, viscosity, and gravity terms for one particle using the previously computed densities and pressures. This mapping

was a natural fit for the GPU because the same type of computation had to be repeated over many particles, allowing us to launch many threads and process a large number of particle interactions concurrently.

The simulation loop was controlled by the CPU. The CPU managed the frame loop and the substeps within each frame, launched the density and force kernels once per stage, and synchronized after both of them. In this baseline each CUDA thread was mapped to one fluid particle. A thread reads its own fluid particle state from the fluid array, then scanned across the full fluid array and the two boundary arrays to accumulate density or force contributions from all possible interaction partners. The main parallelization was to assign per particle computations, which were previously sequential, to GPU threads running in parallel. We can do this because, mostly, particle computation is independent.

We did not have to change the underlying serial algorithm. The same structure was preserved, with density and pressure computed before force, and force computed before integration, but the density and force stages were moved into separate CUDA kernels so that one thread could process one fluid particle in parallel with many others, with synchronization after each kernel stage to maintain dependencies between stages. This kept the original structure intact while changing the execution from sequential CPU loops to GPU parallel kernels.

The main limitation of this baseline was that it still used brute force neighbor checks, meaning each thread compared its particle against every fluid and boundary particle. As a result, while CUDA provided an immediate speedup over the sequential version, it did not solve the underlying scaling problem caused by the expensive checks. We also considered moving timestep integration to the GPU, but timing showed that integration contributed only a small fraction of the total runtime compared to density and force, so we focused first on parallelizing the interaction heavy stages while leaving integration on the CPU.

Spatial Hash implementation and optimization path

Our main optimization to the basic CUDA baseline was to replace brute force neighbor checks with a spatial hash based search structure. Instead of each thread scanning the full fluid particle and boundary arrays when computing density and force, we divided the simulation domain into grid cells whose size was tied to the SPH interaction radius. At each substep, particles were assigned to cells based on their position, and the simulator built an auxiliary grid structure that mapped cells to the particles they contained. In our implementation, this meant building cell identifiers for particles, sorting particles by cell, and then computing the start and end ranges for each cell. Once this structure was available, a thread processing one fluid particle no longer had to scan all the particles in the simulation. Instead, it only checked the particles in the local cell and neighboring cells, which is enough because particles outside that region cannot contribute to density or force. This advantage is visualized in Figure 3.

The first version of this spatial hash was not immediately successful. Although it reduced the amount of interaction work done in density and force, it introduced a new cost, rebuilding the grid every substep. Rebuilding the grid was unavoidable for correctness due to the moving nature of the fluid particles, however In practice, we found that this overhead was large enough that the first spatial hash version actually performed worse than the basic brute force CUDA baseline. This was an important result in the project because it showed that simply adding a more advanced search structure was not enough by itself. The cost of maintaining that structure had to be reduced as well, especially in a problem like pouring where particle distributions change quickly.

From there, most of our work went into reducing the overhead around the hashed simulation while preserving the same visual behavior. The first major improvement was to separate boundary

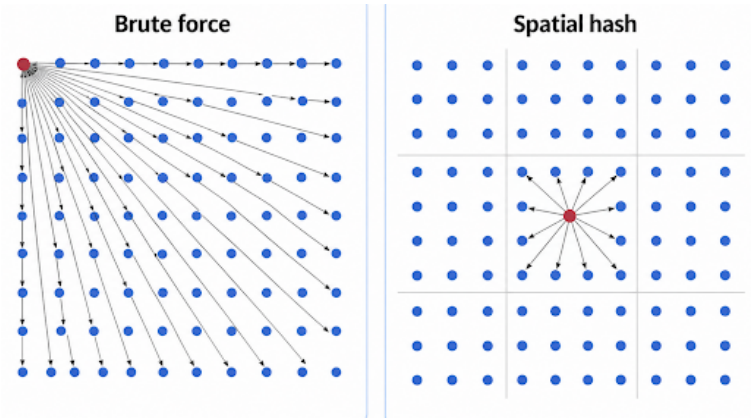

Figure 3: Visualization of Brute force and Spatial Hash particle comparisons
(*Note not all arrows were visualized for brute force, as it became visually crowded)

particles used for physics from boundary particles used only for visualization. Originally each boundary array would carry inner particles used mainly for collision interactions with fluid particles, and outer particles that were far enough from the fluid to not interact with it, but still provide the cup a visual body or padding. Then outer particles were dense, and really bloated the particle count, which in turn bloats computation in the particle comparison phase. We changed this so that the simulation used inner compute boundary arrays, while the denser, outer render boundary was rebuilt only when exporting CSV frames for ParaView. This let us keep smooth cup walls in the visualization without forcing that same dense representation into the inner simulation loop. We also found that the source and receiver cups did not need to be treated identically. The source cup rotates during pouring, so its compute boundary array must be rebuilt every substep, but the receiver cup remains fixed, so its compute boundary can be built once at initialization and reused throughout the run. This removed repeated work that was contributing nothing to correctness. These optimizations were beneficial to both the baseline, and the spatial hash implementations, and did give us minor benefits in speedup for the spatial hash, but did not cause large jumps in scaling, just barely putting the spatial hash at a higher speedup to the baseline.

The third major refinement, and the largest, was to extend the spatial hash beyond fluid to fluid interactions. In the earlier versions, the grid helped reduce fluid to fluid search cost, but fluid to boundary interactions were still comparatively expensive because the wall particles were not fully integrated into the same grid. We therefore gave the fluid particles, the moving source boundary, and the static receiver boundary their own sorted grid based lookup structures, and then used cell neighborhood checks for all of them during density and force. This was the change that produced the largest improvement, which makes sense, because it removed the remaining broad scans over wall particles and made both fluid interactions and boundary interactions local. At that point, the spatial hash version finally behaved the way we had originally hoped, preserving the same pouring behavior while clearly outperforming the brute force CUDA baseline.

This optimization path was one of the most important parts of the project because it was not just a matter of adding a spatial grid and stopping there. The first version was slower, and the final successful version only emerged after several rounds of refinement to the boundary representation, boundary update schedule, and scope of the hashed lookup itself. In that sense, the spatial hash implementation was not one single optimization, but rather a sequence of design changes that gradually reduced the overhead of maintaining the search structure until the benefits of localized neighbor lookup outweighed its cost.

Neighbor list implementation

As an alternative to spatial hashing, we also implemented a neighbor list based CUDA version. In this approach, each fluid particle stores an explicit list of nearby interaction partners, including both nearby fluid particles and nearby boundary particles. The density and force kernels then reuse these saved lists instead of performing a fresh full search during each stage. In terms of mapping to the GPU, the structure was still one thread per fluid particle, but now each thread first used a neighbor list initialization pass to build the set of particles it would later interact with, and then the density and force stages looped only over those stored neighbors. T differs from spatial hashing in that spatial hashing reduces search cost by organizing particles into local cells every substep, while neighbor lists try to pay the search cost up front and then reuse the result across later computations.

Our first neighbor list version rebuilt the lists every substep. This kept the lists accurate, but also made list construction the dominant cost, as building the lists still required scanning over large particle sets. As a result, the density and force stages became cheaper once the lists were available, but the up front cost of rebuilding the lists every substep was large enough to limit the overall speedup. This shows that neighbor lists are only useful when the cost of building them can be spread across multiple later uses. To study that tradeoff, we extended the implementation to support rebuilding the neighbor lists less frequently. Instead of rebuilding at every substep, we experimented with rebuilding every N substeps and then reusing the saved lists in between.

Spatial hashing was designed around rebuilding a shared structure every substep, where particles are first grouped into cells and each thread then reads from that common structure to find nearby partners. Neighbor lists take a different approach, where each particle stores its own list of fluid and boundary neighbors, so after the lists are built, a thread can work from the neighbor data associated with its particle rather than repeatedly accessing a shared global grid. This integrates well with CUDA because the work maps naturally to one thread per particle, and each thread can reuse its own saved neighbor information during density and force computation. The tradeoff, however, is that constructing those per particle lists is still expensive, so neighbor lists only become worthwhile when that upfront cost can be spread across several later substeps.

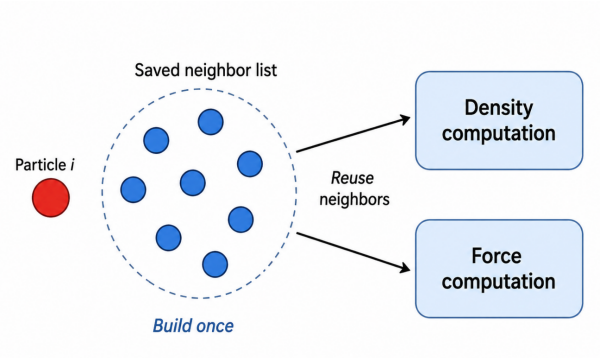

Figure 4: Neighbor list visualization

4 Results and Systems Analysis

Overall Performance

To establish a baseline for our performance analysis, we first measured the execution time of a single physics substep (density, forces, and integration) using our sequential CPU implementation. On the GHC machines' Intel Core i7, a single sequential substep for roughly 56,000 fluid particles took an average of **43,059.4 ms**. Given that our standard simulation requires 1,000 frames at 20 substeps per frame (20,000 total substeps), the sequential approach would require about **239.2**

hours (nearly 10 days) to compute a single pour.

By directly migrating the heavy $O(N^2)$ neighbor interaction math to the GPU via our Brute Force CUDA implementation, we achieved an immediate reduction to a 176.06 ms substep time, which comes out to a 244x speedup. However, the Brute Force method scales poorly as particle counts rise, and leaves a massive amount of GPU performance on the table. By introducing our neighbor-search structure, we drastically improved kernel execution times. The Neighbor List approach greatly reduced density kernel execution time to just 0.3 ms and force kernel execution time to 1.71 ms. However, the `build_neighbor_lists` kernel is still forced to do the $O(N^2)$ search through all particles, resulting in an execution time of 93.56 ms, greatly limiting the speedup of the neighbor list implementation. To mitigate the penalty of rebuilding neighbor lists, we can reduce the frequency of running the rebuild kernel. When rebuilding lists every 10 substeps, we achieved an amortized substep time of 11.34 ms which comes out to a 3,797x speedup over the sequential approach. However, rebuilding the neighbor lists less frequently, introduces the possibility of physics inaccuracies due to particle drift. Ultimately, our fully optimized Spatial Hash implementation brought the substep time down to just **9.54 ms** while still rebuilding hashes every substep, to ensure physical correctness. This represents a **4,513x speedup** over the sequential baseline, reducing the total computation time of the simulation from nearly 10 days down to just **3.18 minutes**.

Kernel Execution Profiles

To understand the performance discrepancies between our three GPU strategies, we profiled the core kernels using NVIDIA Nsight Compute. Figure 5 illustrates the execution time breakdown across the build, density, and force stages.

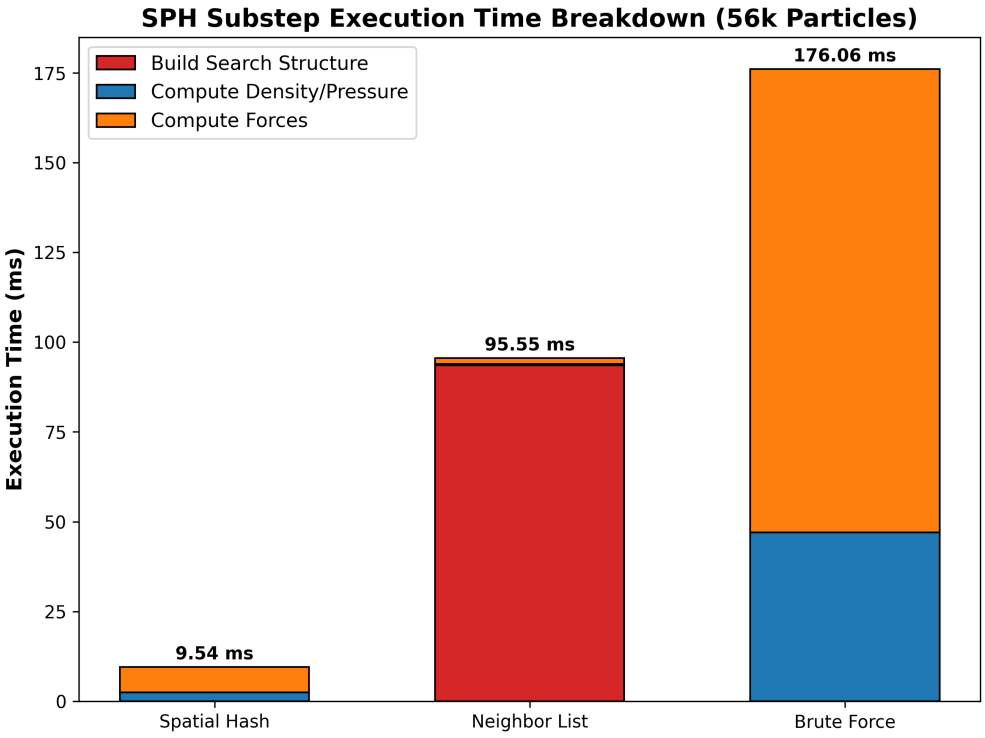

Figure 5: SPH Substep Execution Time Breakdown (56k Fluid Particles).

The **Brute Force** method has zero upfront build time overhead, but severely oversaturates

the GPU's instruction queue due to the sheer volume of redundant math generated by the global $O(N^2)$ distance checks. This results in a 47.03 ms Density kernel and a 129.03 ms Forces kernel, making it computationally bound by wasted operations.

The **Neighbor List** implementation presents a interesting tradeoff between algorithmic minimalism and setup overhead. When analyzing strictly the math kernels, the Neighbor List is the fastest approach by a massive margin, finishing the Density computation in **0.30 ms** and Forces in **1.69 ms**. Because each thread loops exactly over pre-verified neighbors (roughly 30-40 particles), the instruction count is highly optimized. However, generating these explicit lists requires a catastrophic **93.56 ms** build kernel. This is because the neighbor list implementation does not eliminate the $O(N^2)$ complexity search, rather it just shifts the bottleneck from the physics kernels into the setup phase.

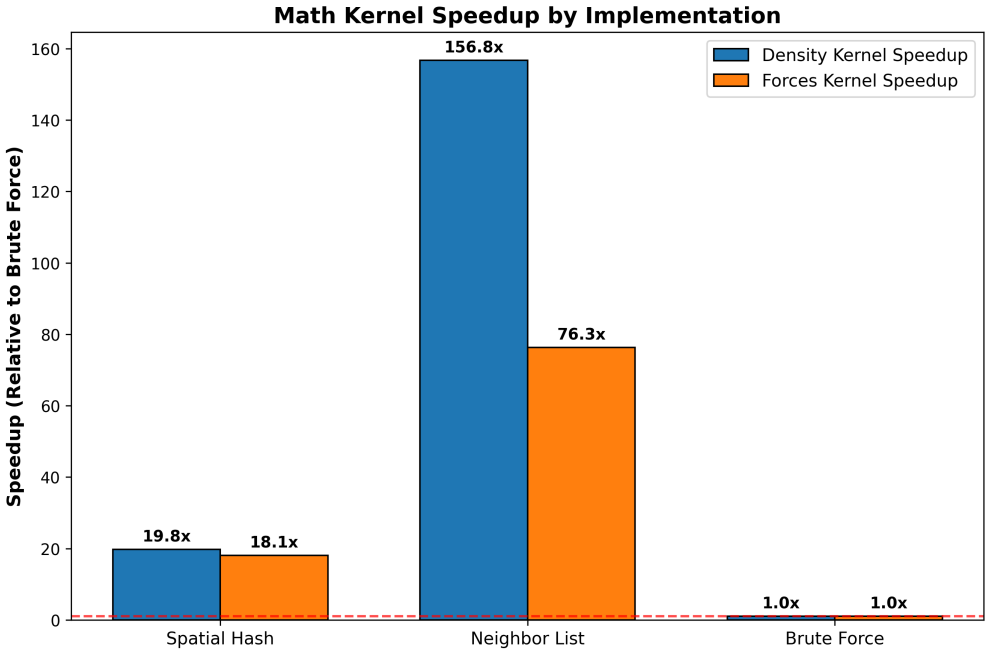

Figure 6: Math Kernel Speedup by Implementation (Excludes Build Time)

The **Spatial Hash** approach proved to be the optimal architectural middle ground. By utilizing `thrust::sort_by_key`, its build phase utilizes highly optimized hardware radix sorting to achieve roughly $O(N)$ complexity, taking only **0.02 ms** to construct the grid. While its Density (2.38 ms) and Forces (7.14 ms) kernels are slower than the explicit Neighbor List, the near-zero build overhead makes it the clear leader in total speedup.

Memory Performance Analysis and Comparison

We originally anticipated severe cache thrashing due to the scattered nature of fluid particles in memory. Nsight Compute profiling of the `compute_density_pressure` kernel revealed some interesting statistics on memory access pattern differences between the Spatial Hash and the Neighbor List that perfectly illustrates the importance of memory coalescing.

During the density kernel, the Spatial Hash algorithm technically performs vastly more memory reads, requesting **42 million sectors** from VRAM compared to the Neighbor List's **16.5 million sectors**. This occurs because the Spatial Hash evaluates all candidates within a 27-cell bounding

box, checking roughly 6 times the volume of the actual SPH smoothing radius. However, because the Spatial Hash physically reorders the `particles` array in VRAM to match the grid layout, neighboring particles sit in the exact same 128-byte cache lines. Profiling showed that the Spatial Hash operated with a highly optimized **1% excessive sector rate**. It fetches more data, but reads it in perfectly contiguous blocks, maximizing memory bus bandwidth.

Conversely, the Neighbor List checks far fewer particles, but looking up `particles[neighbor_list[i]]` relies on random array indices. Even though the Neighbor List requested only 16.5 million total sectors, Nsight Compute revealed that a massive **87% of the data fetched was excessive and discarded**. The memory controller was forced to fetch full 32-byte memory sectors just to retrieve single 4-byte floats, causing severe cache thrashing. The Neighbor List's math kernels remain fast because the GPU has a large enough bandwidth to hide the inefficiencies of the kernel, however, this highly inefficient use of GPU resources presents a glaring scalability issue of this implementation.

Kernel Comparison

Across all three implementations, the compute forces kernel consistently requires 2–3× longer to execute than the density kernel (7.14 ms vs 2.38 ms in the Spatial Hash implementation), making it the dominant cost in the physics loop.

This gap is driven by both increased memory traffic and higher arithmetic intensity per neighbor interaction. The density kernel is relatively lightweight as it evaluates a scalar smoothing function and fetches only position data (x, y, z), totaling 12 bytes per neighbor. In contrast, the forces kernel computes pressure gradients and viscosity Laplacians, requiring more complex operations including square roots and normalization while fetching position, velocity, density, and pressure which is 32 bytes per neighbor, nearly a 3× increase in memory payload.

Nsight Compute reports that the forces kernel uses 58 registers per thread, compared to 46 registers for the density kernel. While this increase is moderate, it still reduces the number of concurrently active warps per SM, slightly lowering achievable occupancy. Combined with the significantly higher memory bandwidth demand and increased instruction count, this limits the GPU's ability to fully hide latency.

As a result, even with efficient neighbor search and memory coalescing, the forces kernel remains the primary bottleneck in the simulation.

Warp Divergence and Load Imbalance

While spatial hashing perfectly aligned our memory accesses, the physical nature of a pouring simulation inherently bottlenecks SIMT execution through load imbalance.

CUDA executes threads in lockstep warps of 32. In our cup-to-cup scenario, the particle distribution is highly non-uniform. The simulation features dense bodies of fluid resting in the receiver cup, alongside thin streams and disconnected splash droplets traveling through the air. Consequently, a single warp will inevitably contain threads mapped to dense pool particles which require evaluation of 32-35 neighbors alongside threads mapped to mid-air splash droplets requiring 0-3 neighbors. Because warps cannot retire and free up SM resources until all 32 threads complete their instructions, the threads computing the droplets finish their loops almost instantly and must sit completely idle waiting for the denser particle threads to finish their extensive neighbor evaluation. We observed that as the fluid poured and transitioned into a more irregular splashing state, frame execution times increased slightly. Profiling the "Average Active Threads per Warp" metric confirmed that branch and loop divergence within the neighbor-search logic deactivated significant portions of the warp during particularly irregular distribution phases of the simulation. We ob-

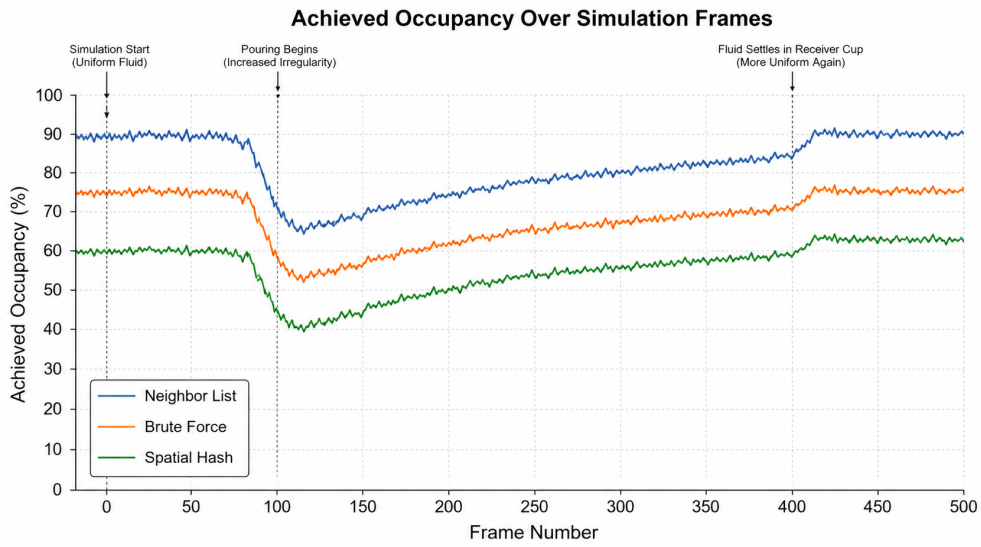

Figure 7: Achieved occupancy over simulation frames

served the achieved occupancy drop 20-30% during the most non-uniform parts of the simulation (illustrated in Figure 7). This demonstrates that while our search structures solved the $O(N^2)$ complexity problem, the irregular, dynamic topology of a fluid simulation remains a fundamental constraint on maximizing GPU hardware utilization.

As shown in Figure 7, all three implementations experience a severe drop in occupancy starting at Frame 100. During the uniform resting state (Frames 0–100), particle neighborhoods are relatively consistent, allowing warps to execute with minimal divergence. However, as the pour begins, the fluid transitions into an irregular topology of thin streams and splashing droplets. This creates massive load imbalance. Threads computing sparse droplets finish rapidly and idle, stalling warp retirement and preventing new warps from scheduling onto the SMs. As the fluid finally settles in the receiver cup (Frame 410+), workload uniformity is restored, and occupancy recovers.

Counterintuitively, Spatial Hash, our fastest algorithm, exhibits the lowest overall Achieved Occupancy, peaking at roughly 63%. This is driven entirely by register pressure. The Neighbor List math kernels execute a simple 1D loop over pre-computed arrays, requiring very few registers per thread and allowing the SMs to reach 90% occupancy. Conversely, the Spatial Hash kernel must calculate 3D grid coordinates, map to 1D hashes, and evaluate nested loops across a 27-cell bounding box. This mathematically intensive logic requires significantly more registers per thread, effectively capping the number of warps that can physically reside on the SM at once.

However, this lower occupancy is heavily outweighed by memory throughput. While the Neighbor List packs the SM with active warps, those warps are constantly stalled waiting on random, uncoalesced memory fetches (87% excessive sector rate mentioned previously). The Spatial Hash sacrifices raw occupancy to maintain localized, perfectly coalesced memory accesses and $O(N)$ algorithmic complexity. This proves that feeding the GPU contiguous data through an algorithmically superior structure is vastly more important than merely maximizing active thread counts, especially in a memory bound program like SPH.

Note

While traditional scalability analysis often involves benchmarking execution times across a range of arbitrary particle counts, we intentionally kept our simulation fixed at roughly 56,000 fluid par-

ticles. In Smoothed Particle Hydrodynamics, altering the particle count within a fixed fluid volume requires meticulously retuning simulation parameters such as smoothing radius, particle mass, rest density, and integration timestep to prevent the fluid from exploding or compressing unrealistically. Because retuning these parameters fundamentally alters the average neighborhood density and workload of each substep, direct execution time vs number of particles comparisons become unreliable or unrepresentative. Instead of plotting basic empirical execution times across different problem sizes, we relied on extensive hardware profiling via NVIDIA Nsight Compute to evaluate the architectural scalability of our implementations. By examining fundamental hardware limits such as the Neighbor List's unsustainable 87% excessive memory sector rate, or the Spatial Hash's register pressure driven occupancy caps we were able to analytically deduce how each algorithm scales under load. In GPU systems, identifying severe cache thrashing, uncoalesced memory fetches, and SIMT warp divergence serves as a much more definitive predictor of an algorithm's asymptotic scalability than simply altering the particle count and measuring the resulting clock time.

5 List of work by each student, and distribution of total credit

- Baseline Sequential Implementation - Athan and Yashoditya
- Sequential Implementation Profiling - Athan
- Porting sequential to CUDA - Yashoditya
- Pofiling of Brute Force CUDA Implementation - Athan
- Spatial Hashing Implementation - Yashoditya
- Spatial Hashing Profiling - Athan
- Neighbor Search Implementation - Yashoditya
- Neighbor Search Profiling - Yashoditya
- Final Report - Yashoditya(60%) Athan(40%)

Total distribution of work: Yashoditya - 50%, Athan - 50%

References

[1] Vagif Aliyev. SPH-Solver: Smoothed Particle Hydrodynamics (SPH) meshless method for solving the Navier-Stokes equation. 2026.

[2] M. Antuono, C. Pilloton, A. Colagrossi, and D. Durante. Clone particles: A simplified technique to enforce solid boundary conditions in sph. *Computer Methods in Applied Mechanics and Engineering*, 409:115973, 2023. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2023.115973. URL `https://www.sciencedirect.com/science/article/pii/S0045782523000968`.

[3] Simon Green. Particle simulation using cuda. 2010.

[4] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Fluid Dynamics*, 2003:154–159, 07 2003.