2D Particle Collision Detection

Charles Njoroge

May 6, 2025

1 Background

To maximize GPU performance in particle simulations, I apply several CUDA optimization strategies. Thread coarsening increases per-thread workload to reduce scheduling overhead and improve arithmetic intensity. Shared memory tiling in collision detection enhances data locality and minimizes global memory access. Parallel scans are used for efficient prefix sums in particle binning, enabling fast cell-based lookups. Warp-level primitives accelerate intra-warp coordination, reducing the need for full synchronization and improving latency. Finally, occupancy tuning balances register usage, block size, and shared memory allocation to achieve high throughput and hardware efficiency across all kernels.

2 Overview of Optimizations

Profiling showed a distinct classification between memory-bound and compute-bound kernels. Memory-bound operations such as exclusive_scan (12.5% occupancy, 0 FLOPs), clear_frame_buffer (63.0% occupancy, 0 FLOPs), and count_particles_per_cell (36.1% occupancy, 0 FLOPs) were dominated by memory access patterns and atomic operations, with negligible arithmetic workload. In contrast, compute-bound kernels like detect_collisions_shared (35.4% occupancy, 27.5 million FLOPs) and update_particle (38.0% occupancy, $\sim$30,151 FLOPs) exhibited substantial arithmetic throughput and are bottlenecked by floating-point operations rather than memory access. The get_particle_id_gpu kernel (28.7% occupancy, 20,000 FLOPs) also trends toward compute-bound due to coordinate transformation calculations. This indicates that performance gains can be achieved through memory access optimization for memory-bound kernels and arithmetic instruction optimization for compute-bound ones.

Thread coarsening is applied in the detect_collisions_shared kernel, where each thread processes multiple neighboring particles in a strided loop, increasing computational workload per thread and reducing overhead from thread management. Shared memory tile reuse is also used in this kernel to store neighboring particles temporarily, reducing redundant global memory accesses and leveraging faster shared memory bandwidth. Parallel scan is implemented in the exclusive_scan kernel to calculate prefix sums over cell counts, which are later used to determine the start and end indices for cell-based particle grouping. Warp shuffle intrinsics are used in my reduce max kernel but showed little performance improvement; instead, traditional synchronization and shared memory methods are used in other similar places. Memory coalescing is used in most kernels, particularly where sequential accesses to global memory are possible, such as in particle updates and rendering. However, some indirect memory accesses, like those involving cell ID indexing, may limit full coalescing potential. Overall, the most complex optimizations are in the collision detection and histogram computation kernels, where performance benefits are important.

3 Observations

Across all three memory modes tested (standard malloc, pinned memory, and zero-copy), kernel occupancy and throughput were measured using nvprof. The clear_frame_buffer kernel achieved the highest average occupancy, reaching 63.15% in mode 2, with perfect warp execution efficiency (100%) across all modes. Similarly, reduce_max consistently exhibited the highest occupancy among computational kernels at approximately 49%, while exclusive_scan showed the lowest occupancy at 12.47%, due to its inherently sequential and synchronization-heavy structure. Other kernels such as update_particle and render_image maintained moderate occupancy levels between 35%–38%, with warp execution efficiencies above 97%. The observed occupancy patterns align with expectations based on register pressure, shared memory usage, and available thread parallelism.

Occupancy did not materially change with increasing iteration counts or particle counts in this configuration, confirming that kernel launch parameters—specifically, block and grid sizes—were the primary influencers. For most kernels, a launch configuration based on CUDA's cudaOccupancyMaxPotentialBlockSize was used, dynamically adjusting block sizes to balance occupancy and resource usage. Notably, detect_collisions_shared, which utilizes dynamic shared memory, sustained lower occupancy (average $\sim$35%) as shared memory per block became the limiting factor. The trade-off between maximizing SM utilization and preventing resource contention was evident across kernels. These findings informed my tuning strategy for maintaining high warp execution efficiency and stable SM throughput across varying memory modes.

Across all kernel configurations, I observed consistent patterns in achieved_occupancy, where clear_frame_buffer and reduce_max consistently yielded high occupancy values, exceeding 0.6 in most modes and input sizes, while exclusive_scan and get_particle_id_gpu consistently underperformed in terms of warp residency and register allocation. As input size increased from $p = 100$ to $p = 100000$, most kernels exhibited improved occupancy, especially update_particle, which scaled from near 0.3 to over 0.85. Meanwhile, detect_collisions_shared showed steady occupancy improvements only for mode `m=2`, reflecting the effectiveness of shared memory for larger workloads. These trends suggest that compute-bound kernels benefit more from scaling, while memory-bound kernels show hardware saturation unless well-optimized.

Looking at eligible_warps_per_cycle and sm_efficiency, I noticed that clear_frame_buffer maintained maximal warp scheduling up to 3.0 across modes for lower p values, before giving way to exclusive_scan at $p = 100000$ which peaked above 4.0 in mode 0. However, mode 2 consistently lagged in this metric, indicating resource bottlenecks or serialization. On the other hand, warp_execution_efficiency was near-ideal (95–100%) across almost all kernels and parameterizations, except for detect_collisions_shared and exclusive_scan at small p values, which improved steadily with workload size. These results suggest my kernels are generally branch-efficient, though occupancy and warp availability can still be improved, especially in data-dependent or irregular computation phases like scan and shared-memory reductions.

As iterations progressed and workload scaled, I noticed time per frame initially improved before plateauing and, in some cases, regressed. For example, performance in get_particle_id_gpu and detect_collisions_shared peaked at mid-range $p = 10000$ then dipped at $p = 100000$ for mode 2, likely due to increased pressure on the PCIe bus or TLB thrashing from mapped memory. In contrast, global memory strategies (`m=0`) exhibited steadier performance across iterations due to explicit control over data layout and prefetch locality. I believe these trends arise from a balance between memory access latency, register allocation, and the kernel's ability to coalesce accesses and saturate SMs. Where all strategies performed similarly—such as in render_image—the kernel's logic is likely memory-bound with minimal divergence, masking backend architectural differences. Overall, performance improved with scale until architectural limits (e.g., shared memory size, memory bandwidth) imposed ceilings, highlighting the importance of tuning memory mode dynamically with workload size.