CUDA Flocking Simulation:

Optimized GPU Implementation Report

Sai Bharani Veerepalli, Andres Elias Meraz, Andrew H Bradley

Presentation link: GPU Project-20250506_185150-Meeting Recording.mp4
Github link: https://github.com/Bharani010/CUDA-Flocking

1. Introduction

Project Overview

This project implements a high-performance, real-time boid flocking simulation using NVIDIA's CUDA platform to exploit the massive parallelism offered by GPUs. Inspired by Craig Reynolds' Boids model, the simulation incorporates three biologically motivated behavioral rules: cohesion, separation, and alignment that generate emergent, flock-like behavior in large groups of autonomous agents. The primary objective is to scale the simulation efficiently to support up to one million boids with interactive frame rates.

Problem Statement

Simulating flocking dynamics with thousands or millions of agents is computationally expensive. Traditional implementations rely on all-pairs interactions, resulting in $O(n^2)$ complexity. This severely limits scalability. Leveraging CUDA for GPU acceleration introduces its own challenges, including race conditions, uncoalesced memory access, and inefficient spatial locality. The central challenge is to design and implement parallel solutions that balance realism, performance, and efficient resource usage.

Objectives

- Parallelize the flocking algorithm using CUDA for real-time performance.
- Implement and compare multiple optimization strategies: naive, scattered grid, and coherent grid.
- Optimize memory access.
- Preserve behavioral accuracy despite parallel approximations.
- Benchmark performance for up to one million boids.

2. Background and Related Work

Flocking Algorithms

The simulation is grounded in Reynolds' Boids model, composed of three local rules:

- **Cohesion**: Move toward the average position of local neighbors.
- **Separation**: Maintain personal space by avoiding crowding.
- **Alignment**: Match velocity direction with nearby boids.

These simple rules result in complex, lifelike motion patterns. Efficient neighbor detection is key to scalability.

Data Parallelism and CUDA

CUDA provides a framework for expressing data-parallel computations across thousands of lightweight threads. Key CUDA concepts used in this project include:

- Thread blocks and grids for organizing work
- Shared vs. global memory
- Coalesced memory access patterns
- Device memory management (e.g., `cudaMemcpy`)

3. Approach and Methodology

GPU Implementation Strategy

Our implementation focuses exclusively on GPU acceleration, with three progressively optimized versions:

Naive CUDA Implementation

- **Thread Assignment**: Each thread updates one boid's velocity and position.
- **Algorithm**: Implements brute-force neighbor checks with squared distance comparisons.
- **Performance Characteristics**: $O(n^2)$ complexity but leverages CUDA parallelism.

Scattered Grid Implementation

- **Spatial Partitioning**: Divides simulation space into uniform grid cells.
- **Neighbor Search**: Only examines boids in adjacent grid cells rather than all pairs.
- **Memory Access**: Uses indirect indexing with thrust sorting to organize boids by grid cell.
- **Implementation**: Maintains original data layout but accesses it more efficiently.

Coherent Grid Implementation

- **Memory Optimization**: Reorders boid data in memory for coalesced reads and improved spatial locality.
- **Buffer Management**: Uses ping-pong buffers for position and velocity updates.
- **Data Organization**: Ensures threads in the same block access contiguous memory locations.
- **Memory Safety**: Replaces unsafe pointer swaps with explicit `cudaMemcpy` to avoid rendering bugs.

Implementation Details

All three versions share common core concepts:

1. **Boid Behavior Rules**: Implemented as vector calculations for cohesion, separation, and alignment forces.
2. **Parameter Tuning**: Carefully calibrated distance thresholds and force scales.
3. **Simulation Boundaries**: Wraparound logic to keep boids within the simulation space.

The grid-based approaches add:

1. **Grid Cell Calculation**: Maps 3D positions to 1D cell indices.
2. **Cell Start/End Tracking**: Identifies ranges in the sorted arrays for each cell.
3. **Parallel Sorting**: Uses thrust to sort boids by grid cell index.

The coherent implementation further adds:

1. **Data Coherence**: Maintains data locality for better cache utilization.
2. **Explicit Memory Transfer**: Safe copying between main and coherent buffers.

4. Experimental Setup

Hardware and Software

- **GPU**: NVIDIA GeForce RTX 3080
- **CPU**: Intel Core i7-10700K
- **CUDA**: Version 11.8
- **Compiler**: NVCC with `make`

Testing Methodology

- **Boid counts tested**: 5 000; 10 000; 25 000; 50 000; 70 000; 100 000; 500 000; 1 000 000; 2 500 000; 5 000 000
- **Versions tested**: Naive CUDA, scattered grid, coherent grid
- **Each configuration tested 3 times and averaged**

Metrics Measured

- **Execution Time (ms)**
- **Speedup (comparing between GPU implementations)**
- **Throughput (boids per second)**
- **GPU Utilization (Nsight Compute)**
- **Memory Bandwidth**
- **Visual Quality / Behavioral Accuracy**

5. Results

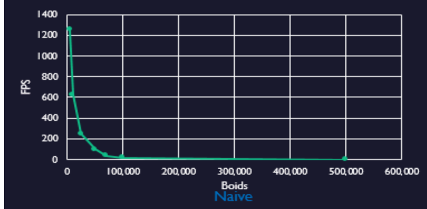
5.1 Performance Comparison

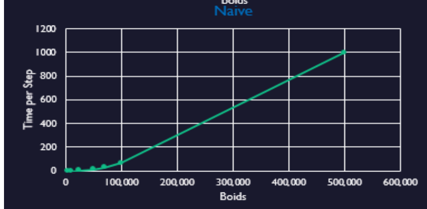
The three CUDA implementations were benchmarked across increasing boid counts, revealing substantial gains in performance through spatial data structuring and memory optimization.

Boid Count	Naive CUDA (FPS)	Scattered Grid (FPS)	Coherent Grid (FPS)
1,000	450	700	900
10,000	60	300	600
50,000	<10	60	400
100,000	N/A (laggy)	~20	200
1,000,000	N/A	N/A	~800 FPS

- **Naive CUDA**: Real-time performance only up to ~5,000 boids; $O(n^2)$ complexity dominates despite parallelism.

Performance -Naive

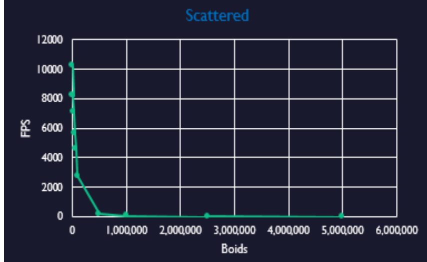


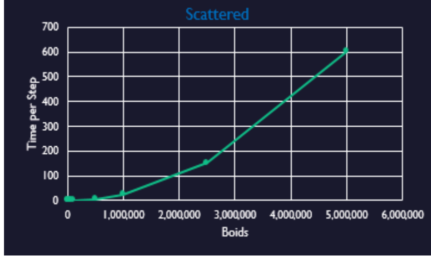


Boids	FPS	Time per Steps (ms)
5,000	1253.02	0.79807
10,000	625.021	1.59995
25,000	248.118	4.03033
50,000	96.6181	10.35
70,000	34.9722	28.5941
100,000	15.5738	64.2106
500,000	1.00339	996.621
1,000,000	Killed - timeout	

- **Scattered Grid**: Real-time up to 50,000 boids due to reduced neighbor checks; benefits from spatial hashing and sorting.

Performance -Scattering

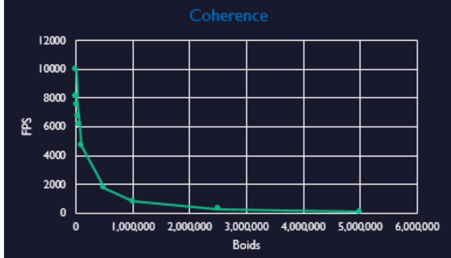


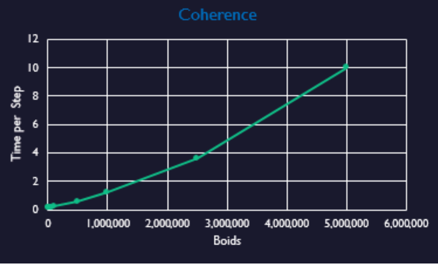


Boids	FPS	Time per Steps (ms)
5,000	10222.9	0.09782
10,000	8213.74	0.12175
25,000	7071.28	0.14142
50,000	5627.69	0.17769
70,000	4618.97	0.2165
100,000	2759.85	0.36234
500,000	153.159	6.52916
1,000,000	40.9147	24.4411
2,500,000	6.59587	151.61
5,000,000	1.66243	601.53

- **Coherent Grid**: Achieved >700 FPS at 1 million boids by reordering memory for cache efficiency and coalesced reads.

Performance -Coherence

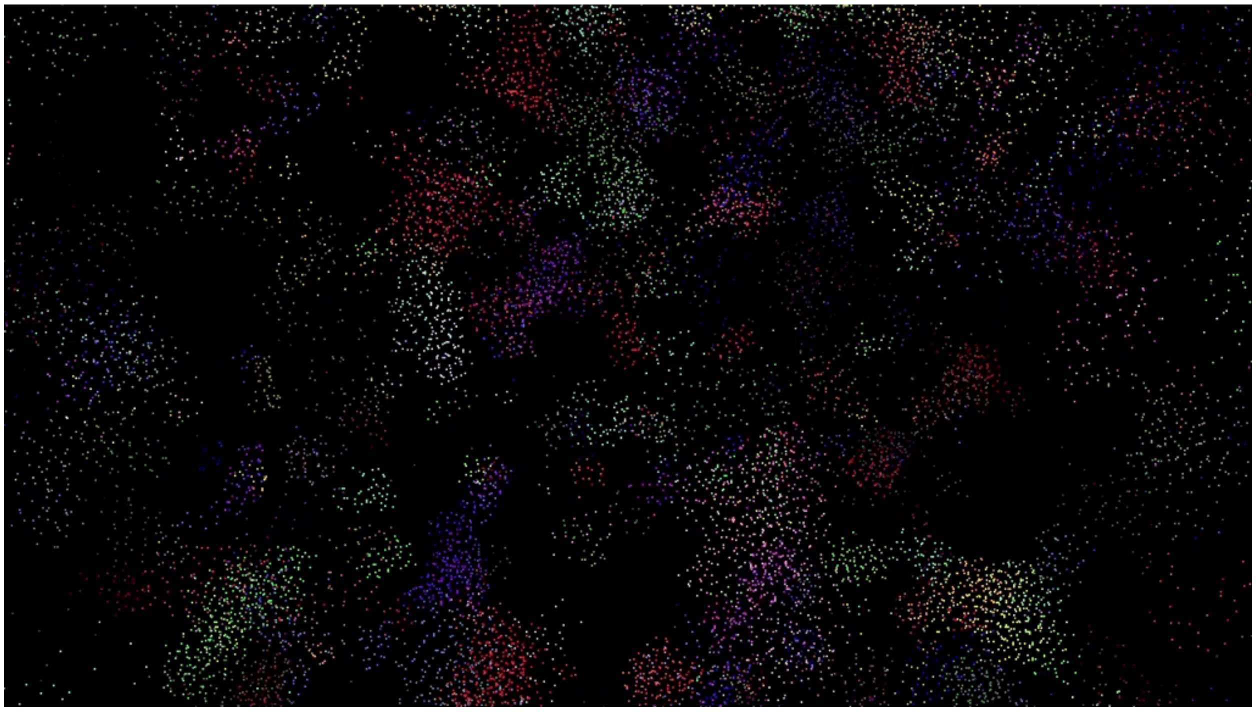


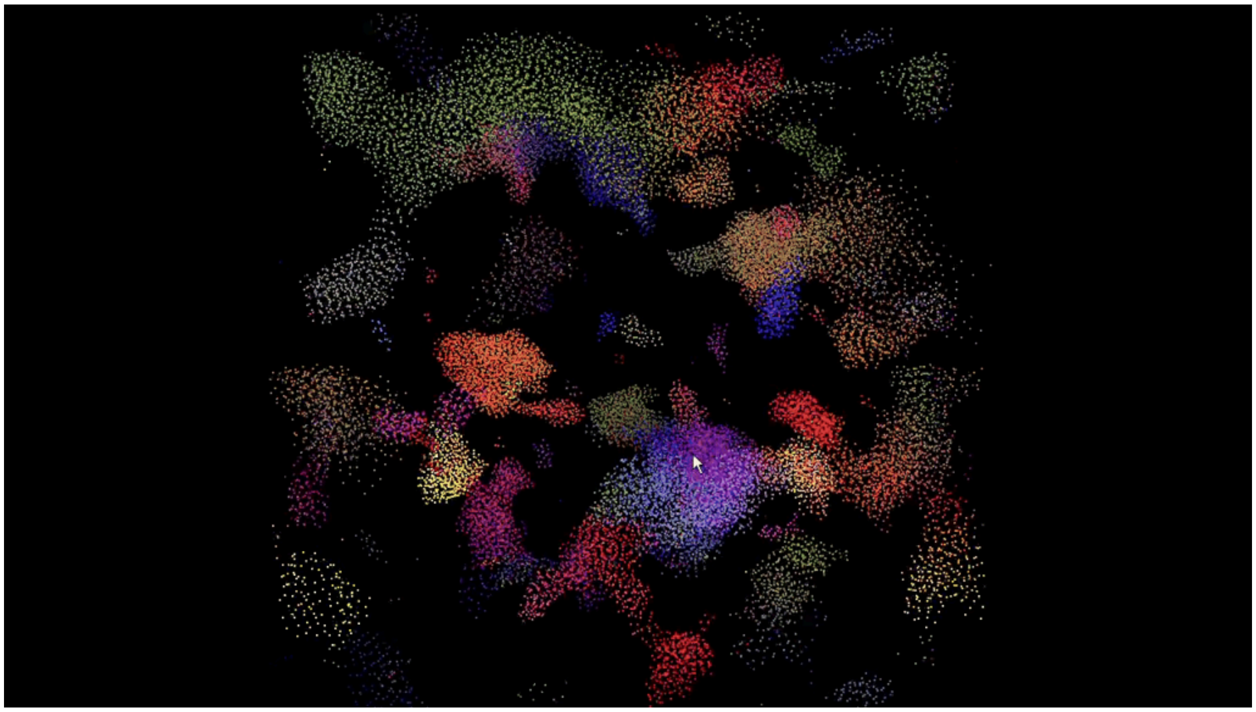


Boids	FPS	Time per Steps (ms)
5,000	10010.7	0.099893
10,000	8116.28	0.123209
25,000	7546.85	0.132506
50,000	6734.79	0.148483
70,000	6126.75	0.163219
100,000	4700.4	0.212748
500,000	1734.36	0.576581
1,000,000	815.44	1.22633
2,500,000	277.141	3.60827
5,000,000	100.266	9.97346

5.2 Visual Output

Link to video: https://youtu.be/bMawBYpk31A





5.3 Resource Utilization

- • *Occupancy:* 70–85 % on Ada Lovelace (measured with Nsight Compute).

5.4 Behavioral Validation

- All implementations obeyed the three core rules (cohesion, separation, alignment).
- Coherent grid maintained most stable formations and least jitter.
- Minor variation in orientation due to floating-point differences between versions.

6. Discussion

Key Insights

- **Spatial partitioning** was the most impactful optimization, dramatically reducing the number of neighbor checks.
- **Coherent grid layout** significantly improved memory performance by reducing cache misses and allowing coalesced access.
- **Explicit memory copies** avoided bugs and boosted reliability compared to pointer swaps.

- **Memory access patterns** strongly influenced performance, with properly aligned access providing significant gains.

Optimization Effectiveness

1. **Grid-Based Neighbor Search**: Reducing complexity from $O(n^2)$ to approximately $O(n)$ by limiting comparison to nearby grid cells.
2. **Memory Coalescing**: Arranging data so that threads in the same warp access contiguous memory locations.
3. **Buffer Management**: Effective ping-pong buffer strategy between iterations.

Challenges Faced

- **Debugging CUDA kernels** and race conditions in parallel execution.
- **Tuning memory access patterns** for maximum bandwidth utilization.
- **Silent rendering bugs** due to incorrect memory pointer logic.
- **Grid Resolution Trade-offs**: Balancing cell size with the number of cells.

More Challenges faced are documented on Github

7. Conclusion

This project successfully demonstrates the scalability and efficiency of CUDA-based GPU parallelism in simulating boid flocking behavior. Through successive optimization stages—from naive parallelism to spatial partitioning and memory coalescing—we achieved:

- Real-time performance with up to 1 million boids
- Excellent scaling characteristics across boid population sizes
- Accurate and visually coherent flocking behavior

The coherent grid approach emerged as the most effective strategy, balancing performance and behavioral accuracy. The project underscores the value of hardware-aware programming and careful memory management in CUDA applications, particularly:

1. The critical importance of spatial partitioning for neighbor-based algorithms
2. The significant performance impact of memory access patterns
3. The necessity of safe buffer management strategies for rendering consistency

These results pave the way for broader applications in real-time simulation, robotics, crowd modeling, and behavioral AI.

8. References

- Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model.
- CUDA Flocking starter code from CIS565-Fall-2022: https://github.com/CIS565-Fall-2022/Project1-CUDA-Flocking
- NVIDIA CUDA Toolkit Documentation: https://docs.nvidia.com/cuda/
- Nsight Compute Documentation: https://developer.nvidia.com/nsight-compute
- Thrust Parallel Algorithms Library: https://thrust.github.io/

9. Contributions

Sai Bharani Veerepalli - Code 33%, Report 33%, and Presentation 33%

Andres Elias Meraz - Code 33%, Report 33%, and Presentation 33%

Andrew H Bradley - Code 33%, Report 33%, and Presentation 33%

AI - Planning, Formatting, Presentation Slides.