jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 1, PP. 31–37
Manuscript received October 10, 2024; revised March 2025. doi: 10.24425/ijet.2025.153541

Efficiency analysis of parallel swarm intelligence using rapid range search in Euclidean space

Łukasz Michalski, Andrzej Sołtysik, and Marek Woda

***Abstract*—Swarm intelligence algorithms are widely recognized for their efficiency in solving complex optimization problems. However, their scalability poses challenges, particularly with large problem instances. This study investigates the time performance of swarm intelligence algorithms by leveraging parallel computing on both central processing units (CPUs) and graphics processing units (GPUs). The focus is on optimizing algorithms designed for range search in Euclidean space to enhance GPU execution. Additionally, the study explores swarm-inspired solutions specifically tailored for GPU implementations, emphasising improving efficiency in video rendering and computer simulations. The findings highlight the potential of GPU-accelerated swarm intelligence solutions to address scalability challenges in large-scale optimization, offering promising advancements in the field.**

***Keywords*—swarm intelligence; parallel computing; range search; CUDA**

I. Introduction

NATURAL ecosystems host a wide array of organisms that exhibit complex, efficient, and fascinating collective movement behaviours. Among these, classic examples include fish swimming in synchronized schools, birds migrating in tightly organized flocks, sheep navigating terrain in cohesive herds, and insects, such as ants or bees, forming intricate swarms. These behaviours reflect an inherent collective intelligence that enables individual members of a group to act in union with minimal direct communication. Ants, in particular, exhibit a highly coordinated foraging strategy, where they follow a specific, organized path to and from a food source. This collective action is not random but follows well-defined principles that ensure the success of the group as a whole. Mimicking such complex, aggregate motion patterns has become increasingly relevant in the fields of artificial life, robotics, and computer animation. These natural phenomena inspire applications ranging from immersive gaming environments to sophisticated cinematography. Moreover, these biological behaviours are frequently leveraged to solve complex computational optimization problems, as evidenced by the development of algorithms like ant colony optimization (ACO) and particle swarm optimization (PSO), both of which draw heavily from nature's strategies for group problem-solving [2].

The formalization of computational models designed to replicate the group dynamics seen in animal motion was first introduced by Craig Reynolds in 1987. His groundbreaking work led to the creation of the *boids* model, a simplified yet powerful simulation for representing collective animal motion in computer-generated environments [3]. The term "boids" is a playful contraction of "bird-oid," referring to the creatures in the simulation that represent birds or other flocking animals. The model's foundation rests on the emergence of flocking behaviour from the interaction of simple rules followed by individual boids. Specifically, the boids model is built upon three essential behavioural rules that govern the movement of each boid: (1) the avoidance of crowding or collision with nearby neighbours, (2) the alignment or synchronization of movements with adjacent boids, and (3) the attraction towards a central point of cohesion, typically the centre of the group. These straightforward rules, when combined, generate lifelike simulations of collective motion without requiring explicit leadership or central control. Over time, the boids model has been expanded and refined to include additional behaviors, such as avoiding obstacles in the environment and steering towards specific goals. These enhancements have broadened its utility across diverse applications, particularly in fields such as robotics, virtual environments, and autonomous vehicle navigation.

Further modifications to the boids model have been developed to simulate more nuanced behaviors. For example, Delgado et al. [4] introduced a sophisticated extension incorporating emotional and fear dynamics into the model. In their version, emotional states are transmitted between individuals through pheromones, much like how insects communicate. These pheromones are treated as particles dispersed in a gas-like medium, simulating free expansion, which can influence the behaviour of neighbouring boids by triggering flight responses or changes in group cohesion. Another significant modification came from Hartman et al. [5], who introduced the concept of "change of leadership," a force that dictates how likely a boid is to assume a leadership role within the group. This additional factor affects how a boid might try to lead the group away from danger or guide it toward a specific target, adding a layer of complexity to group decision-making dynamics.

Since its initial proposal, the boids model has seen widespread application, particularly in computer graphics and animation, where it has been used to create realistic and visually appealing representations of group motion. A prime example of its early adoption can be seen in the gaming

Authors are with Department of Computer Engineering, Wroclaw University of Technology, Poland (e-mail: marek.woda@pwr.edu.pl).

© The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/), which permits use, distribution, and reproduction in any medium, provided that the Article is properly cited.

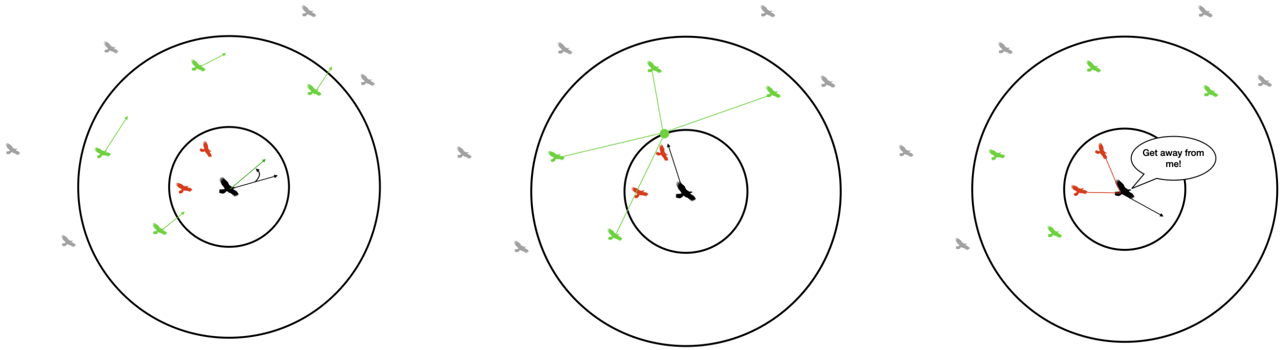

Fig. 1. Three fundamental rules that characterize flocking behaviour: The Alignment Rule (leftmost) - oriented towards the average velocity of nearby boids within the visible range. The Cohesion Rule (centre) - steer towards the centre of mass of boids in the perception range. The Separation Rule (rightmost) - distance oneself from boids within the protective range [1].

industry, where Valve Corporation used the boids model in their 1998 video game *Half-Life* to simulate the flight patterns of bird-like creatures. This represented a major step forward from traditional animation techniques, which often relied on manual, frame-by-frame adjustments. In addition to gaming, the boids model has made significant contributions to the film industry. Its first notable application in cinematic animation occurred in the short film *Stanley and Stella in Breaking the Ice* (1987). Following this, it was prominently featured in the 1992 blockbuster *Batman Returns*, where the model was used to simulate the movement of a large flock of penguins, showcasing its versatility in generating lifelike group behavior. Over the years, the boids model has been integrated into a wide range of gaming, animation, and cinematic projects, highlighting its enduring relevance and adaptability.

This research delves into the implementation of a swarm intelligence algorithm on both multithreaded Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to harness the full potential of modern computational resources. The algorithm leverages CUDA (*Compute Unified Device Architecture*) technology, which is specifically designed to accelerate computational processes on GPU hardware. By utilizing GPUs, which are optimized for parallel processing tasks, the algorithm's runtime is significantly reduced, offering faster and more efficient simulations of swarm behavior. The contribution of this study to the existing body of knowledge is multifaceted. First, it involves a detailed investigation of performance metrics, comparing the execution speed and efficiency of the algorithm on two distinct architectures: the traditional multithreaded CPU and the highly parallelized GPU. This comparative analysis provides insights into how different hardware platforms handle the computational demands of swarm intelligence algorithms, offering valuable guidelines for future implementations.

Another crucial aspect of the study is the visualization of swarm behaviour within constrained three-dimensional Euclidean space, which presents unique challenges. In such restricted environments, individual entities must navigate complex spatial limitations while maintaining the group's collective behaviour. Our work aims to improve the interpretability of these simulations by developing visualization techniques that highlight the emergent properties of the swarm, particularly in scenarios where space is limited. Additionally, the research explores optimization strategies for implementing swarm intelligence algorithms on GPU platforms. This involves adjusting parameters like block sizes to maximize computational efficiency and using profiling tools to fine-tune the algorithm's performance. By focusing on these optimizations, we aim to push the boundaries of what is possible with swarm intelligence in modern computational environments, particularly regarding scalability and runtime efficiency on GPU architectures.

II. Swarm Intelligence

The boid swarm model employs a vector-based approach to represent each behavioural rule, allowing for adaptive responses to the surrounding environment. These vectors, characterized by their magnitude and direction, evolve in response to local conditions and interactions with neighbouring agents. The movement of individual boids, or entities within the swarm, is governed by a linear combination of these behaviour rule vectors. As the model incorporates more rules, the challenge of determining and optimizing the coefficient weights for the movement vector increases. These coefficients are crucial for balancing the relative influence of each rule and ensuring a realistic simulation of flocking behaviour. The collective behaviour of the swarm emerges from the combined movements and interactions of individual boids, each adhering to a set of simple steering behaviours. These behaviours mimic the social reactions observed in real animal flocks, such as responding to the positions and velocities of neighbours.

a) Cohesion: rule ensures that each boid is drawn towards the centre of its local flock. This fundamental principle of flocking aligns with Reynolds' original model and maintains the group unity of the boids. Without cohesion, the boids would disperse, losing their characteristic flocking behaviour. However, excessive cohesion could lead to a single point of convergence, compromising the dynamic nature of the movements. Therefore, cohesion must be balanced with other steering behaviours, such as separation and alignment.

The cohesion vector for a boid ($\overrightarrow{Coh_i}$) is calculated in two steps. First, the algorithm determines the centre of mass (f)

of the flock containing the boid. This is achieved by averaging the positions of all neighbouring boids, as shown in equation 1. The centre of the flock ($\overrightarrow{Fc_i}$) is then calculated by averaging the positions of these neighbouring boids. Finally, the cohesion displacement vector is determined by calculating the direction and magnitude required for the boid to move towards the centre, as represented in equation 2. Here, p_j represents the position of a neighbouring boid j, and N is the total number of boids in the local flock.

$$\overrightarrow{Fc_i} = \sum_{\forall b_j \in f} \frac{\overrightarrow{p_j}}{N} \tag{1}$$

$$\overrightarrow{Coh_i} = \overrightarrow{Fc_i} - \overrightarrow{p_i} \tag{2}$$

b) Alignment: rule ensures that each boid adjusts its heading and speed to match its neighbouring flock mates. This rule fosters synchronized movement, preventing erratic and disorganized motion. By aligning with the average velocity of nearby boids, the boids produce the smooth, flowing patterns observed in natural flocks, such as birds flying in formation or fish swimming in schools.

The alignment vector for a boid ($\overrightarrow{Ali_i}$) is calculated by first determining the average velocity of all nearby boids ($\overrightarrow{Fv_i}$), as shown in equation 3. This average velocity serves as a reference for adjusting the boid's speed and direction. The alignment vector is then derived by subtracting the boid's current velocity from the average flock velocity, as described in equation 4. This difference guides the boid towards alignment with the group's overall movement.

$$\overrightarrow{Fv_i} = \sum_{\forall bj \in f} \frac{\overrightarrow{v_j}}{N} \tag{3}$$

$$\overrightarrow{Ali_i} = \overrightarrow{Fv_i} - \overrightarrow{v_i} \tag{4}$$

c) Separation: rule prevents collisions and overcrowding among flock members. Without this rule, boids would clump together, potentially colliding and losing the realistic spacing seen in natural flocks. The separation rule encourages boids to maintain a comfortable distance from their neighbours, ensuring that the group maintains cohesion and individual spacing.

The separation vector ($\overrightarrow{Sep_i}$) is calculated by summing the vectors that point from the boid in question (b_i) to each of its neighbours (b_j) within its local perception range. The separation steer vector is the negative sum of these individual vectors, as described in equation 5. This negative value forces the boid to move away from any nearby neighbours, preventing crowding and preserving the fluidity of motion.

$$\overrightarrow{Sep_i} = - \sum_{\forall b_j \in f} (\overrightarrow{p_i} - \overrightarrow{p_j}) \tag{5}$$

The overall movement vector for each boid ($\overrightarrow{V_i}$) is calculated by combining the vectors from each of the three primary steering behaviours: cohesion, alignment, and separation. The coefficients (w_1, w_2, and w_3) determine the relative importance of each rule and are adjusted to fine-tune the resulting behaviour, ensuring that the boids exhibit lifelike, realistic motion. The general form of the movement vector is shown in equation 6:

$$\overrightarrow{V_i} = w_1 \cdot \overrightarrow{Coh_i} + w_2 \cdot \overrightarrow{Ali_i} + w_3 \cdot \overrightarrow{Sep_i} \tag{6}$$

The iterative calculation of this vector at each time step governs the continuous movement and interaction of the boids within the simulation, producing emergent flocking behaviour that closely resembles that of real animals.

III. Range Search & Implementation

A. Range Search

A critical component of the boid algorithm is the ability to efficiently identify and interact with nearby boids within a certain perception range. This range search is a computational challenge that directly impacts the performance of the algorithm, especially as the number of boids increases.

1) Linear Search: is a straightforward, brute-force approach to this problem. Where the position and velocity of every other boid in the flock are evaluated based on an exhaustive search. This method, while simple, is computationally expensive, requiring $O(n^2)$ operations, where n is the total number of boids. As the size of the flock grows, the computational cost becomes prohibitively large, making this approach inefficient for large-scale simulations.

2) Fast Range Search: is the key to addressing this issue, with more sophisticated methods like k-d trees and octrees that can be employed to optimize the range search process. K-d trees are particularly useful for organizing spatial data, allowing for faster nearest-neighbour and range searches in multidimensional space. In this structure, space is recursively divided along different dimensions, resulting in more efficient searches for nearby boids. Similarly, octrees partition space into octants and are especially well-suited for 3D simulations. By organizing boids into these hierarchical structures, the algorithm can dramatically reduce the number of comparisons required to find neighbouring boids, significantly improving performance without sacrificing accuracy.

These optimizations are essential for scaling up the boid model to handle larger and more complex simulations, where computational efficiency is a primary concern. Through these advanced data structures, the boid algorithm can simulate the movement of large groups of boids in real time, enabling more realistic and expansive simulations of collective animal behaviour.

B. Implementation

While the proposed solution to improve time complexity is effective, implementing it efficiently on the GPU poses several challenges, particularly when using tree-based data structures. One primary challenge stems from the fact that trees are often considered "pointer machine" data structures, which can lead to fragmentation and gaps in memory, complicating GPU execution. The GPU architecture, specifically CUDA, is optimized for parallelism but requires efficient memory access

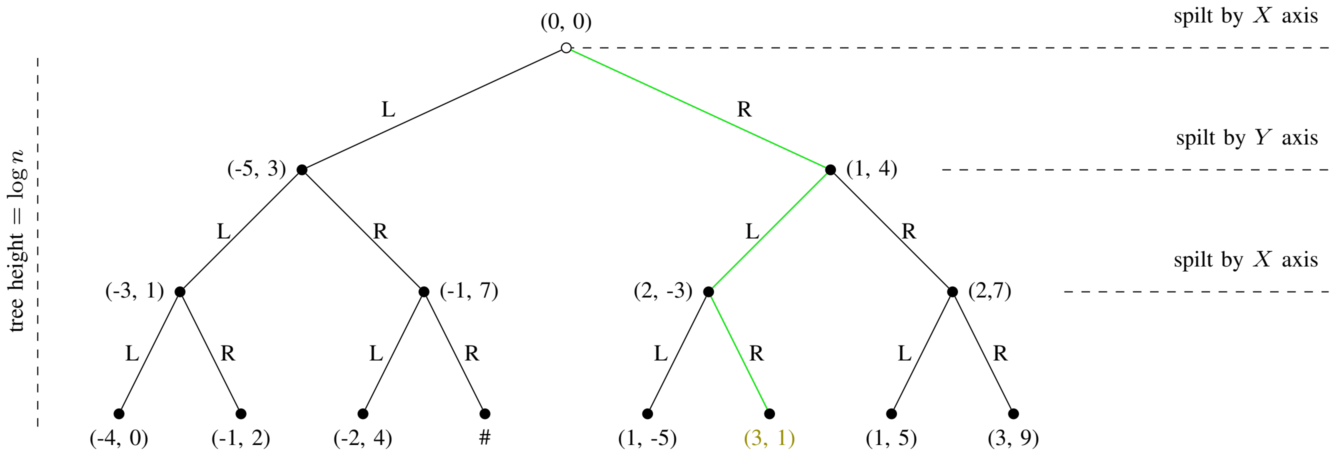

Fig. 2. The k-d tree data structure, characterized by a dimensionality of $k = 2$, is utilized to efficiently locate the nearest neighbour to the point (3, 3). The steps involved in this search are highlighted in green, demonstrating the algorithm's logarithmic time complexity [6].

patterns, which trees do not inherently provide due to their structure.

To mitigate this, one approach is to assign one CUDA thread to each boid in the simulation. Each thread would then loop through the entire position and velocity buffers (excluding its boid) and compute a new velocity based on the local interaction rules. Following this step, another CUDA kernel would apply the updated velocities to modify each boid's position accordingly. While this brute-force method does not take advantage of spatial partitioning, it aligns well with the GPU's architecture and avoids the complexities of managing tree structures in global memory.

a) Proper Data Represenatation: is a key factor in optimizing GPU-based algorithms. Right data representation can greatly minimize memory throughput and increase computational efficiency. One crucial decision is how to store and organize an agent's properties in GPU memory. In a swarm simulation, each agent (boid) has various properties such as position, velocity, and other behavioural parameters.

Storing these properties in a struct-of-array (SoA) format is often the preferred method for improving coalesced memory access, which is critical for maximizing the performance of global memory access on the GPU. In SoA, each property of all agents is stored in a separate array, which ensures

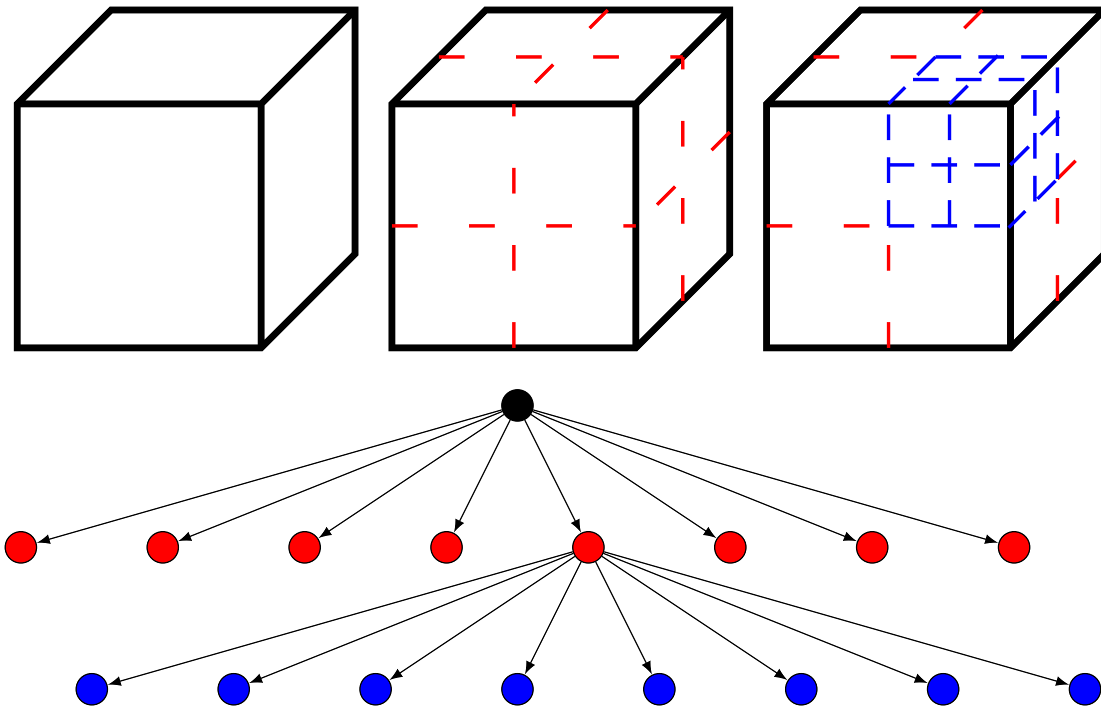

Fig. 3. Octree Data Structure: A recursive subdivision technique that splits a cube into eight distinct regions, called octants. This process creates an octree that visually represents the hierarchical breakdown of Euclidean space into increasingly smaller octants for efficient spatial organization.

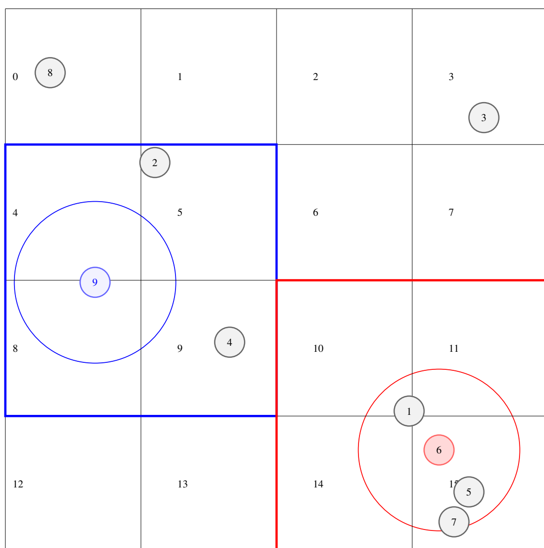

Fig. 4. Range-based search applied to a particle space distributed across a uniform scattered grid structure.

that consecutive threads in a warp access consecutive memory addresses. This minimizes cache misses and enhances throughput. However, the drawback is that it can make the code more difficult to maintain, especially if different agents have different sets of properties.

In contrast, an array-of-structs (AoS) format stores all the properties of each agent together in a contiguous memory. This makes it easier to program and maintain, as each agent's data is self-contained, but it does not facilitate efficient memory access. In the AoS format, consecutive threads may need to access memory locations that are not adjacent, resulting in non-coalesced memory access and poorer performance.

Given the trade-offs, our approach adopts the struct-of-array format, which balances the need for efficient coalesced memory access while also managing the complexity of maintaining the simulation's diverse agent properties. This decision optimizes memory throughput, which is crucial for high-performance GPU simulations.

b) Efficient Range Search on the GPU: Efficiently implementing a spatial data structure on the GPU can greatly enhance the algorithm's performance by reducing the number of boids each thread needs to evaluate. The three primary steering rules—cohesion, alignment, and separation—only apply within a certain neighbourhood radius. Thus, organizing boids into a spatial grid can significantly reduce the computational load.

To achieve this, the simulation space can be divided into a uniform grid where each cell is as wide as the neighbourhood radius. This preprocessing step allows each boid to only check its neighbours within the surrounding cells, minimizing the number of comparisons. In the 3D case, if the grid's cell width is set to twice the neighbourhood radius, each boid only needs to check for interactions within the eight surrounding cells. In 2D, this number reduces to four cells (see Figure 4).

However, constructing a uniform grid on the GPU presents its own set of challenges. In a CPU-based implementation, each boid would be assigned to a grid cell, and pointers to the boids within each cell would be stored in a dynamically resizable array. Unfortunately, this approach is not feasible on the GPU because arrays in CUDA are not resizable, and race conditions can occur if multiple threads attempt to write to the same memory location simultaneously.

To overcome this, a sorting-based approach is used to construct the grid on the GPU. Each boid is first labelled with an index corresponding to its enclosing grid cell. These boids are then sorted by their cell indices, ensuring that all boids belonging to the same cell are contiguous in memory. Once sorted, the array of grid indices is traversed, and boundaries between cells are identified by detecting changes in the indices.

This method allows the GPU to efficiently manage boid-to-cell assignments without the need for resizable arrays or complex synchronization mechanisms. The uniform grid's representation is stored as an array, where each entry corresponds to a specific grid cell, and parallelization is achieved through CUDA's inherent data-parallel execution model. This approach ensures that range searches can be performed efficiently, significantly reducing the number of boids that each thread must process, and making the simulation scalable to larger populations.

By adopting this strategy, the GPU can handle a much larger number of boids in real-time, with computational complexity reduced to manageable levels. The uniform grid method, combined with careful data representation, strikes an effective balance between memory efficiency, computational speed, and scalability.

IV. Results

The comparative analysis conducted in this study focused on evaluating the performance of an algorithm across different computational implementations: CPU single-threaded (ST), multithreaded (MT), and GPU parallel versions linear search (LS) and uniform scattered grid (U-SG). The primary metric used for assessment was frames per second (FPS), with the number of particles N serving as a critical parameter.

In the CPU-based implementations, tested on an AMD Ryzen 5 7600 processor, the simulations encountered performance limitations. Specifically, the single-threaded (ST) setup successfully handled simulations with up to 50,000 particles, while the multithreaded (MT) configuration extended this capacity to 100,000 particles.

Conversely, the GPU implementation, leveraging the NVIDIA GeForce RTX 4070Ti with the CUDA framework, exhibited notable scalability advantages. The naive implementation sustained simulations with 1 million particles, and the fast-range search method extended this capability to 5 million particles. Despite these achievements, the achieved FPS rates did not consistently meet the desired responsiveness levels under these configurations.

Detailed results are summarized in Table II, where metrics are rounded to two significant digits for clarity and comparison. This comprehensive evaluation provides insights into the comparative performance across different computational

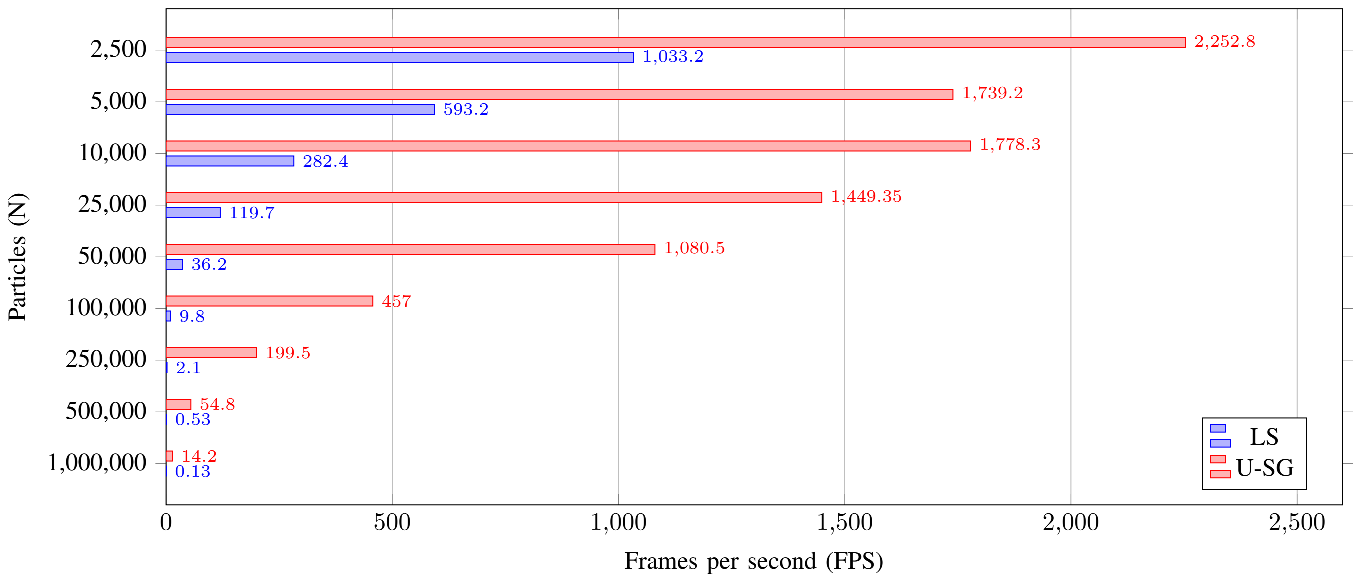

Fig. 5. Performance comparison between GPU-based linear search (LS) and fast-range search with a uniform grid (U-SG) in terms of FPS across varying particle counts (N).

platforms and implementations, highlighting both strengths and limitations in handling varying particle counts.

In addition to the initial comparative analysis, we conducted a thorough assessment of the algorithmic performance on the GPU by systematically varying the threads per block configuration in simulations involving $N = 25,000$ particles. Our objective was to identify the optimal thread block size that would yield the highest performance for the specific workloads and hardware specifications employed in our study.

To achieve this, we implemented a systematic approach that combined experimentation with profiling. This involved analyzing various parameters, including the thread block size, warp occupancy, and other relevant metrics. Through this profiling process, we successfully determined that the most effective configuration for maximizing performance with $N = 25,000$ particles was a block size of 128 threads.

It is crucial to understand the characteristics of warps in this context, particularly when the block size is set to $T < 32$. Under such circumstances, a block would consist of only a single warp. Although some threads within the block may remain unused, each thread is executed individually, which does not yield any algorithmic benefits. This configuration results in inefficient resource utilization, as the benefits of having a larger number of threads per block are lost.

Moreover, utilizing smaller blocks necessitates an increase in the total number of blocks required for the computation. This, in turn, leads to a higher memory allocation demand for each of these numerous blocks—each containing only a single warp. As a result, the performance advantages associated with shared memory within a block are diminished, creating further inefficiencies.

When we analyzed the speedup provided by the GPU for parallel calculations, we observed an impressive average speedup of x50 times compared to a CPU multithreaded solution, particularly when the number of particles N was set to 25,000 or greater. Importantly, the bottleneck related to data transfer between the host and device via the PCIe bus becomes negligible in this scenario. This reduction in data transfer overhead can be largely attributed to the significant advantages gained from highly parallel calculations, especially when working with simulation sizes in the thousands. Overall, our findings underscore the effectiveness of GPU implementations in handling large-scale simulations and highlight the importance of optimizing thread block configurations to achieve maximum performance.

TABLE I
PERFORMANCE RESULTS - FPS TO PARTICLE COUNT N. THE ✗ NOTATION SIGNIFIES CASES WHERE THE SIMULATION COULD NOT BE INITIATED.

N	AMD Ryzen 5 7600 CPU		NVIDIA GeForce RTX 4070Ti GPU	
	ST [FPS]	MT [FPS]	LS [FPS]	U-SG [FPS]
$1.0 \cdot 10^3$	182	450	1700	2300
$2.5 \cdot 10^3$	38	120	1000	2300
$5.0 \cdot 10^3$	9.9	42	590	1700
$1.0 \cdot 10^4$	2.5	13	280	1800
$2.5 \cdot 10^4$	0.43	2.6	120	1500
$5.0 \cdot 10^4$	0.97	0.68	36	1100
$0.1 \cdot 10^6$	✗	0.16	9.8	460
$0.25 \cdot 10^6$	✗	✗	2.1	200
$0.5 \cdot 10^6$	✗	✗	0.53	55
$1.0 \cdot 10^6$	✗	✗	0.13	14
$2.5 \cdot 10^6$	✗	✗	✗	0.58
$5.0 \cdot 10^6$	✗	✗	✗	0.1

V. SUMMARY

The author's parallel implementation of the algorithm demonstrates significant performance gains by utilizing CUDA technology on the GPU compared to traditional CPU approaches. GPU parallelism, particularly through CUDA,

TABLE II
PERFORMANCE RESULTS - FPS TO PARTICLE COUNT N. THE ✗ NOTATION SIGNIFIES CASES WHERE THE SIMULATION COULD NOT BE INITIATED.

N	AMD Ryzen 5 7600 CPU		NVIDIA GeForce RTX 4070Ti GPU	
	ST [FPS]	MT [FPS]	LS [FPS]	U-SG [FPS]
$1.0 \cdot 10^3$	182	450	1700	2300
$2.5 \cdot 10^3$	38	120	1000	2300
$5.0 \cdot 10^3$	9.9	42	590	1700
$1.0 \cdot 10^4$	2.5	13	280	1800
$2.5 \cdot 10^4$	0.43	2.6	120	1500
$5.0 \cdot 10^4$	0.97	0.68	36	1100
$0.1 \cdot 10^6$	✗	0.16	9.8	460
$0.25 \cdot 10^6$	✗	✗	2.1	200
$0.5 \cdot 10^6$	✗	✗	0.53	55
$1.0 \cdot 10^6$	✗	✗	0.13	14
$2.5 \cdot 10^6$	✗	✗	✗	0.58
$5.0 \cdot 10^6$	✗	✗	✗	0.1

greatly enhances the computational efficiency of the multivariate cooperative algorithm by distributing tasks across thousands of threads simultaneously, which CPUs with fewer cores cannot match.

A key factor in this performance boost is the use of CUDA's thread block structure, which enables massive parallelism. However, further improvements can be realized through optimization techniques, especially in shared memory utilization. As discussed in [7], shared memory allows for faster data access within thread blocks, reducing latency compared to global memory. Enhancing the shared memory management could further decrease memory bottlenecks and improve execution speed.

In addition to shared memory optimization, techniques such as dynamic parallelism and warp-level programming could also boost performance. Dynamic parallelism allows for more efficient execution of complex algorithms, while warp-level programming minimizes thread divergence, ensuring optimal thread execution.

In summary, while the CUDA-based GPU implementation significantly improves performance, focusing on optimizing shared memory and exploring advanced CUDA features could lead to even greater gains, further advancing GPU-based parallel computing for large-scale data processing.

REFERENCES

[1] Cornell University ECE 4760 - Obsolete Designing with Microcontrollers., "Boids." [Online]. Available: https://people.ece.cornell.edu/land/courses/ece4760/labs/s2021/Boids/Boids.html

[2] I. Michelakos, N. Mallios, E. Papageorgiou, and M. Vassilakopoulos, *Ant Colony Optimization and Data Mining*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 31–60.

[3] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 25–34.

[4] C. Delgado-Mata, J. Ibáñez-Martínez, S. Bee, R. Ruiz-Rodarte, and R. Aylett, "On the Use of Virtual Animals with Artificial Fear in Virtual Environments," *New Generation Comput.*, vol. 25, pp. 145–169, 02 2007.

[5] C. Hartman and B. Benes, "Autonomous boids." *Journal of Visualization and Computer Animation*, vol. 17, pp. 199–206, 01 2006.

[6] D. R. Karger, "Advanced Algorithms," in *Advanced Algorithms MIT Course No.6.5210/18.415*. MIT OpenCourseWare, 2022. [Online]. Available: https://6.5210.csail.mit.edu/

[7] X. Li, W. Cai, and S. J. Turner, "Efficient Neighbor Searching for Agent-Based Simulation on GPU," ser. DS-RT '14. USA: IEEE Computer Society, 2014, p. 87–96. [Online]. Available: https://doi.org/10.1109/DS-RT.2014.19