Parallel Swarm Intelligence: Efficiency Study with Fast Range Search in Euclidean Space

Łukasz Michalski, Andrzej Sołtysik, and Marek Woda(✉)

Department of Computer Engineering, Wroclaw University of Technology, Janiszewskiego 11-17, 50-372 Wrocław, Poland
marek.woda@pwr.edu.pl

Abstract. Swarm intelligence algorithms are recognised for their effectiveness in solving complex optimisation problems. However, scalability can be a significant challenge, especially for large problem instances. This study focuses on examining the time performance of swarm intelligence algorithms when used with parallel computing on both central processing units (CPUs) and graphics processing units (GPUs). The investigation focuses on algorithms designed for range search in Euclidean space. The research optimizes these algorithms for GPU execution and assesses their effectiveness in handling large-scale instances. The inquiry also explores swarm-inspired solutions tailored for GPU implementation, with an emphasis on enhancing efficiency in video rendering and computer simulations. The research findings show potential for advancing the field by addressing challenges related to large-scale optimization through innovative GPU-accelerated swarm intelligence solutions.

Keywords: swarm intelligence · parallel computing · range search · CUDA

1 Introduction

Numerous organisms in natural settings exhibit collective movement patterns, with examples including fish forming schools, birds flying in flocks, sheep organizing into herds, and insects assembling in swarms. In particular, ants display a coordinated distribution strategy when foraging for food, wherein they collectively follow a designated path. The emulation of such aggregate motion holds significant relevance within the realms of artificial life and computer animation, finding application in diverse domains such as gaming and cinematography. Furthermore, these behavioral dynamics are harnessed to address optimization challenges, exemplified by ant colony optimization and particle swarm optimization methodologies [8]. The conceptualization of a computational model to simulate group animal motion was pioneered by Craig Reynolds in 1987, who introduced the *boids* model [10]. The term "boids" denotes simulated creatures

W. Zamojski et al. (Eds.): DepCoS-RELCOMEX 2024, LNNS 1026, pp. 177–186, 2024.
https://doi.org/10.1007/978-3-031-61857-4_17

within a generic flock. The collective motion of boids emerges from the interplay of uncomplicated behaviors exhibited by individual entities. The boids model encompasses three fundamental rules governing the behavior of each simulated boid: aversion to crowding with neighbors, synchronization and coordination of movements with neighboring entities, and convergence towards group cohesion. Subsequent refinements to the model incorporated additional rules, such as navigating around obstacles and pursuing predefined objectives, within the broader framework of steering behavior. Noteworthy adaptations of the boids model include extensions by Delgado et al. [1], who introduced fear effects by simulating emotional transmission between entities using smell, represented as pheromones modeled as particles in a free expansion gas. Additionally, Hartman et al. [4] introduced a supplementary force termed the "change of leadership," influencing a boid's likelihood to assume a leadership role and attempt an escape. Since its inception, the boids model has been extensively employed in computer graphics to generate lifelike representations of collective motion in groups. The Valve Video Game Company, for instance, utilized the boids model in the 1998 video game Half-Life to simulate bird-like creatures in flight. This marked a significant departure from traditional techniques in computer animation for motion pictures. The inaugural application of the boids model in animation occurred in the short film *Stanley and Stella in Breaking the Ice* (1987), with subsequent integration into the introduction of feature film *Batman Returns* (1992). Over the years, the boids model has found widespread adoption in various gaming and cinematic contexts, underscoring its versatility and enduring significance.

The study implements a swarm intelligence algorithm on both multithreaded Central Processing Units and Graphics Processing Units using CUDA (*Compute Unified Device Architecture*) technology to accelerate the algorithm runtime. The contribution to the existing body of knowledge is clear. The research involves a detailed examination of performance measurements and comparative analyses between two distinct computational architectures. The investigation explores visualizing swarm behavior in constrained three-dimensional Euclidean space problems. This aspect of our work aims to improve the interpretability and comprehensibility of the algorithmic results, especially in situations with limited space. Additionally, our research includes experiments to enhance the implementation of the swarm intelligence algorithm on GPU platforms. This optimization process involves adjusting block sizes strategically and using profiling tools judiciously. The main goal is to improve the computational efficiency of the algorithm on GPU architectures.

2 Swarm Intelligence

The boids swarm model represents each rule with a vector that adapts to the environment through its magnitude and direction. The boid's movement vector is a linear combination of all behavior rule vectors. As the number of behavior rules increases, determining and optimizing the coefficients for the moving vector becomes more challenging. It is crucial to establish realistic moving behavior

by setting these coefficients appropriately. Flock behaviour is the result of the motion and interaction of boids. Each boid follows three simple rules of steering behaviour that describe how it moves based on the positions and velocities of its flock mates (social reaction) (Fig. 1).

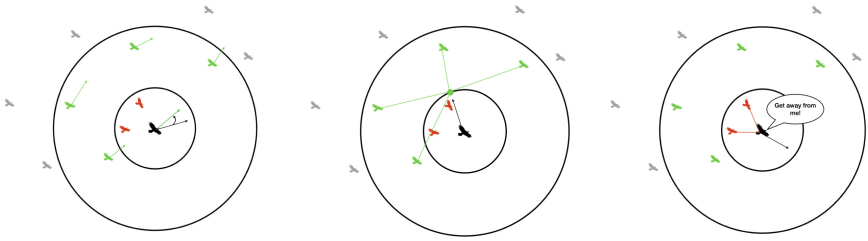

Fig. 1. Three main rules describing flocking behavior. Alignment rule (most left) - align with an average velocity of boids in the visible range. Cohesion rule (center image) - move toward the center of mass of boids in the perception range. Separation rule (most right) - move away from boids in the protected range. Source: *Cornell University ECE 4760 - Obsolete Designing with Microcontrollers.*

Cohesion rule defines the steering force to move toward the average position of local flock mates (as in the original Reynolds' model). Applying the cohesion rule keeps the boids together. This rule acts as the complement of the separation. If only the cohesion rule is applied, all the boids in the flock will merge into one position. Cohesion ($\overrightarrow{Coh_i}$) of the boid (b_i) is calculated in two steps. First, the center $\left(\overrightarrow{Fc_i}\right)$ of the flock (f) that has this boid is calculated as in Eq. 1. Then the tendency of the boid to navigate toward the center of the density of the flock is calculated as the cohesion displacement vector as in Eq. 2. Where p_j is the position of boid j and N is the total number of boids in f.

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

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

Alignment steer to match the heading and the speed of its neighbors. This rule tries to make the boids mimic each other's course and speed. Boids tend to align with the velocity of their flock mates. The alignment (Ali_i) is calculated in two steps. First, the average velocity vector of the flock mates $\left(\overrightarrow{Fv_i}\right)$ is calculated by Eq. 3. Then Ali_i is calculated as the displacement vector in Eq. 4. Where $\overrightarrow{v_i}$ is the velocity vector of boid i. If this rule were not used, the boids would bounce around a lot and not form the beautiful flocking behavior that can be seen in nature.

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

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

Separation rule avoids collisions and overcrowding with other flock mates. There are many ways to implement this rule. An efficient solution to calculate the separation (Sep_i) is by applying Eq. 5. Vectors defined by the position of the boid b_i and each visible boid b_j are summed, then separation steer $\left(\overrightarrow{Sep_i}\right)$ is calculated as the negative sum of these vectors.

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

Overall boid model moving vector V_i for boid b_i is calculated by combining all the steering behavior vectors as in Eq. 6. Where w_i are the coefficients describing the influences of each steering rule and used to balance the rules.

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

The foundational operation of the algorithm involves the iterative refinement of both the motion vector and position grouping for every particle p (??). This meticulous process not only governs the trajectory of individual particles but also facilitates the cohesive grouping of moving objects into well-defined clusters.

Range Search. Within the realm of the algorithm, a crucial computational undertaking commands attention - the judicious allocation of resources to discern and isolate local particles residing within the perceptual purview of a given entity. Conceptually, one can envision each simulated particle as an object on a virtual stage, that identifies and interacts with neighboring objects. The essence of the algorithmic framework lies in the fast filtering of neighboring particles within the immediate perceptual radius of a focal entity, thereby laying the groundwork for subsequent decision-making processes. Selective filtration of particles within this localized sphere of influence serves as a fundamental aspect for optimizing computational efficiency while concurrently affording the algorithm an insightful comprehension of the microcosmic environment surrounding each simulated entity. Not merely a computational optimization strategy, this targeted filtration process becomes the basis for the emergent dynamics defining the collective behavior of the simulated entities. By discerning the presence and relative positioning of proximate particles, the algorithm affords the simulated entities the capacity for responsive actions, encapsulating the principles of cohesion, alignment, and separation emblematic of natural flocking phenomena.

Linear Search is a naive approach for computing the new positions and velocities for the boid b_i. Each particle performs an exhausting search of the entire flock f and filters out boids in their perception range. Then accumulate forces for each boid b_j in the neighborhood and apply the update on boid b_i. This is extremely slow with a runtime of $O(n^2)$. Where n is the number of boids in the flock f.

Faster Range Search can be achieved using tree data structures like k-d tree or octree. In computer science and computational geometry, k-d trees have become a popular data structure used to organize points in k-dimensional space. This is because these structures allow for very efficient searches of points in multidimensional space, including nearest-neighbor searches and range searches (Fig. 2). In a k-d tree, the space is recursively divided into half-spaces along a specific dimension, alternating at each level of the tree - this process is referred to as a spatial partitioning technique [2,3,6]. The selection of the dimension along which to split the space is made based on various criteria, typically maximizing data distribution or minimizing the variance along a particular dimension.

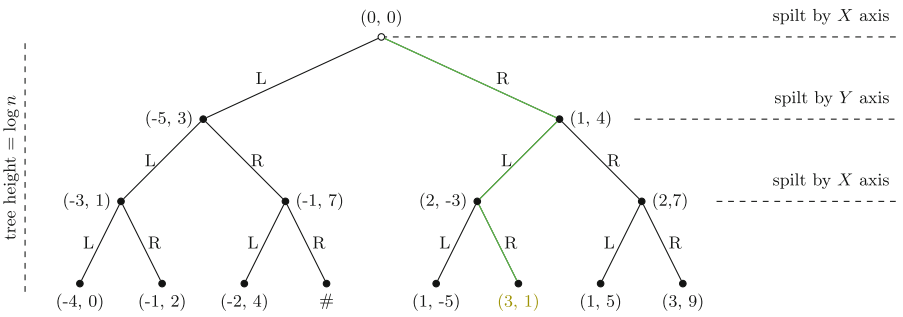

Fig. 2. k-d tree data structure, with a specific value $k = 2$. The steps for searching the nearest neighbor to the point (3, 3) are highlighted in green. This is done in logarithmic time [5].

Octrees similar to k-d trees divide space but into octants (sub-cubes), where each node in the tree corresponds to an octant (Fig. 3). The term *octree* is derived from the fact that each node has eight children, corresponding to the eight octants resulting from the subdivision of a cube. Octrees are particularly well-suited for representing 3D spatial data and volumes. They provide an efficient way to organize and query spatial information hierarchically. One of the strengths of octrees is their ability to adapt to varying point densities in space. They can efficiently represent both dense and sparse regions by adjusting the level of detail in different parts of the space. On the other hand, octree may not be as efficient for uniformly distributed data compared to kd-trees.

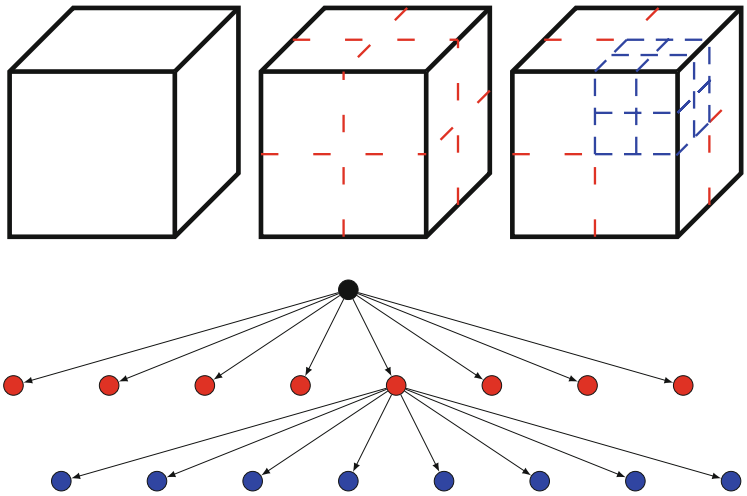

Fig. 3. Octree Data Structure: recursive subdivision of a cube into octants. The resulting octree showcases the hierarchical organization of Euclidean space into octants.

3 Implementation

While the proposed solution reduces time complexity, implementing it on the GPU side using trees is challenging [9]. The tree is a 'pointer machine' data structure with possible gaps in computer memory. To implement this in CUDA, one thread can be assigned to each boid. Each thread would loop over the entire position and velocity buffers (skipping itself) and compute a new velocity for that boid. Another CUDA kernel would then apply the position update using the new velocity data.

Proper Data Representation is a crucial concept in designing parallel algorithms to reduce memory throughput. When organizing an agent's diverse properties in GPU memory, it is important to balance the trade-off between fast access and easy maintenance. Storing data in the struct-of-array format is an effective approach to enhance coalesced access to global memory. In this configuration, values for a common property of all agents are stored in a separate array, reducing the number of cache misses. It is worth noting that agents may have varying properties, and not all properties are necessarily shared by every agent. Therefore, creating a distinct array for each property in the struct-of-array format can introduce programming complexities. Conversely, the array-of-struct format is more favourable in terms of programmability. However, the adoption of this approach does not optimize coalesced access, as the data for a specific agent property becomes interleaved in global memory. Therefore, in our approach, we choose to represent agents using the struct-of-array format to strike a balance between facilitating coalesced access and maintaining programmatic simplicity.

Efficient Range Search on GPU. It is clear that implementing a spatial data structure can significantly improve the performance of the algorithm. By reducing the number of boids that each individual boid must evaluate, we can effectively minimise the computational load per thread. Since the three governing rules apply within a certain radius, organising the boids into a uniformly spaced grid proves to be crucial for performing an efficient neighbour search [7]. A uniform grid consists of cells that are at least as wide as the neighborhood distance and cover the entire simulation domain. In a preprocessing step, we assign the boids to the grid before computing their new velocities. This approach reduces the number of boids that need to be considered. If the cell width is double the neighborhood distance, each boid only needs to be checked against other boids in 8 cells, or 4 in the 2D case (see Fig. 4).

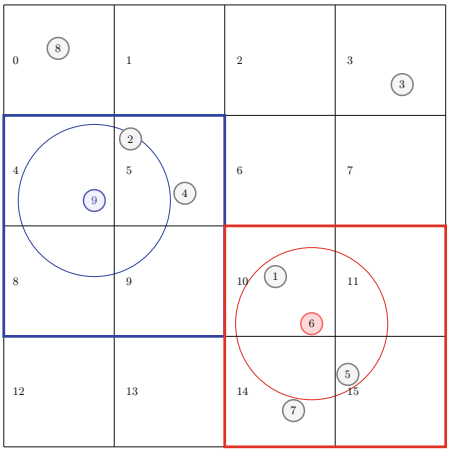

Fig. 4. Range search in uniform scattered grid particles space

To build a uniform grid on the CPU, one must iterate over the boids, determine their enclosing cell, and keep a pointer to the boid in a resizable array that represents the cell. However, this approach is not suitable for the GPU due to unresizable arrays and the potential for race conditions when naively parallelizing the iteration. The construction phase of the uniform grid is achieved through sorting. Each boid is labelled with an index representing its enclosing cell, and the list of boids is sorted by these indices to ensure that pointers to boids in the same cells are contiguous in memory. The array of sorted uniform grid indices is then traversed, and every pair of values is examined. If the values differ, it indicates that we are at the border of the representation of two different cells. Representing the uniform grid can be achieved by storing the locations in a table with an entry for each cell. This table can be an array with space for all cells. The process is data-parallel and can be parallelized.

4 Results

The comparative analysis involved assessing the performance of the algorithm across various implementations, specifically in CPU single-threaded (ST), multithreaded (MT), and GPU parallel versions (LS - linear search & U-SG - uniform scattered grid). The metric utilized for evaluation was frames per second (FPS), and the particle number N served as a key parameter. For the CPU-based implementation on an AMD Ryzen 5 7600 processor, the simulation encountered limitations, allowing for the successful execution of up to 50k and 100k particles for single-threaded (ST) and multithreaded (MT) scenarios, respectively. In contrast, the GPU implementation running on NVIDIA GeForce RTX 4070Ti using the CUDA framework demonstrated greater scalability, running simulations with $N = 1$M particles in naive implementation and 5M using fast range search. However, it is noteworthy that the achieved FPS rate did not meet the desired level of responsiveness in this configuration. Results are presented in Table 1 metric was rounded to two significant digits (Fig. 5).

Table 1. Performance Results - frames per second/particle number N. The ✗ marker indicates simulation launch failure.

N	AMD Ryzen 5 7600		NVIDIA GeForce RTX 4070Ti	
	ST [FPS]	MT [FPS]	GPU LS [FPS]	GPU U-SG [FPS]
1 000	182	450	1700	2300
2 500	38	120	1000	2300
5 000	9.9	42	590	1700
10 000	2.5	13	280	1800
25 000	0.43	2.6	120	1500
50 000	0.97	0.68	36	1100
100 000	✗	0.16	9.8	460
250 000	✗	✗	2.1	200
500 000	✗	✗	0.53	55
1 000 000	✗	✗	0.13	14
2 500 000	✗	✗	✗	0.58
5 000 000	✗	✗	✗	0.1

Furthermore, the assessment of algorithmic performance on the GPU was conducted by varying the threads per block configuration in simulations comprising $N = 25 \cdot 10^3$ particles. In the pursuit of determining the optimal thread block size, a systematic approach involving experimentation and profiling was employed. Through profiling of the thread block size, warp occupancy, and other pertinent parameters, we successfully identified the configuration that maximizes performance for our distinct workloads and hardware specifications. Ultimately, the most effective choice for a specific N was found to be 128 threads

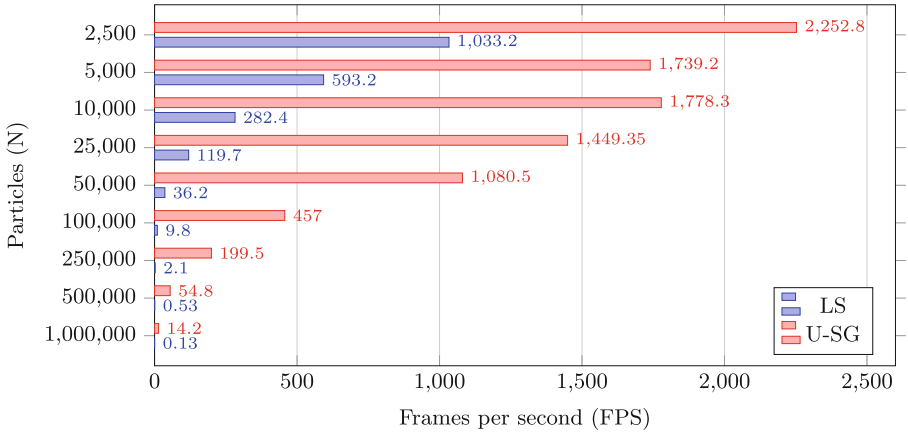

Fig. 5. Comparison of GPU linear search (LS) and fast range search using a uniform grid (U-SG) based on FPS/N - particles

per block. It is noteworthy to highlight the characteristics of warps, particularly in instances where the block size $T < 32$. In such cases, a block consists of a single warp, and despite the presence of unused threads, each thread is executed individually without contributing to any algorithmic advantages. The utilization of smaller blocks necessitates a larger number of blocks. Allocating memory for each of these numerous blocks, each containing a single warp, results in a loss of the performance benefits associated with shared memory within a block. The utilization of the GPU for parallel calculations demonstrates an average speedup of 50 times compared to employing a CPU multithreaded solution when $N \geq 25{\cdot}10^3$. Notably, the bottleneck associated with data transfer between the host and device via the PCIe bus becomes negligible. This is attributed to the substantial advantages derived from highly parallel calculations, particularly when dealing with simulation sizes numbering in the thousands.

5 Summary

The author's parallel implementation of the algorithm demonstrates significant performance gains achieved by implementing the parallel computation model of CUDA technology on GPU compared to traditional CPU approaches. The use of GPU parallelism, particularly in CUDA, leads to significant improvements in the computational efficiency of the multivariate cooperative algorithm analysed. It is noteworthy to mention the potential for further improvements by building on the shared memory concept introduced in the reference paper [7]. This suggests that improvements in shared memory utilisation could contribute to even greater performance improvements in GPU-based parallel computing.

References

1. Delgado-Mata, C., Ibáñez-Martínez, J., Bee, S., Ruiz-Rodarte, R., Aylett, R.: On the use of virtual animals with artificial fear in virtual environments. New Gener. Comput. **25**, 145–169 (2007). https://doi.org/10.1007/s00354-007-0009-5
2. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial partitioning techniques in spatial-hadoop. Proc. VLDB Endow. **8**(12), 1602–1605 (2015).https://doi.org/10.14778/2824032.2824057
3. Gomes, A.J.P., Voiculescu, I., Jorge, J., Wyvill, B., Galbraith, C. (eds.): Spatial Partitioning Methods, pp. 187–225. Springer, London (2009).https://doi.org/10.1007/978-1-84882-406-5_7
4. Hartman, C., Benes, B.: Autonomous boids. J. Vis. Comput. Animat. **17**, 199–206 (2006)
5. Karger, D.R.: Advanced Algorithms. In: Advanced Algorithms MIT Course No.6.5210/18.415. MIT OpenCourseWare (2022). https://6.5210.csail.mit.edu/
6. Li, B.: A Comparative Analysis of Spatial Partitioning Methods for Large-scale, Real-time Crowd Simulation (2014). https://api.semanticscholar.org/CorpusID:54175377
7. Li, X., Cai, W., Turner, S.J.: Efficient Neighbor Searching for Agent-Based Simulation on GPU, pp. 87–96. DS-RT '14, IEEE Computer Society, USA (2014). https://doi.org/10.1109/DS-RT.2014.19
8. Michelakos, I., Mallios, N., Papageorgiou, E., Vassilakopoulos, M.: Ant Colony Optimization and Data Mining, pp. 31–60. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20344-2_2
9. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2023). https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
10. Reynolds, C.W.: Flocks, herds and schools: a distributed behavioral model. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, pp. 25–34 (1987)