Autonomous Boids Extended

CMP202 2023/24 Term 2
Assessment
Diego Andrade (2200905)

Table of Contents

Introduction

This application simulates emergent behaviour that resembles the flocking behaviour of birds or other animals. Originally developed by Craig Reynolds in 1987[1], the original simulation consisted of "boids" who were influenced by three forces: separation, cohesion, and alignment. These forces are determined by other boids in their range of vision. The implementation discussed in this report is based on a more recent version by Hartman and Benes[2] that introduced the concept of leadership. More specifically, boids have a scalar value determined by their position in the flock called "eccentricity". This value is then used in conjunction with a measure of how close to the front of the flock a boid is to attempt a random chance at escape. When a boid escapes, it accelerates quickly to a top speed above other boids' and escapes the flock, causing them to chase the escaping boid briefly, before it stops escaping and returns to a normal speed, and make it the flock's "leader". This is a very interesting addition to the simulation, as it imitates another curious behaviour of birds.

Perhaps unsurprisingly, this simulation's biggest performance bottleneck is checking for visible boids. A sequential program must iterate over all the simulated boids and then iterate over every other boid. This causes the time complexity of just the visibility check to be $O(n^2)$; however, with concurrent programming, we can iterate over the boids in parallel, significantly reducing the computation time per frame.

Application Overview

This implementation of autonomous boids with leadership is heavily object-oriented and the parallelization is built upon the basic sequential classes "Flock" and "Boid". The Flock constructor takes a callable argument that returns a Boid as the flock's shared 'DNA'. All boids in the flock are constructed using the provided callback. The boids contain their own separation, cohesion, and alignment forces and their own eccentricity factor. Each boid has their own update function to calculate all necessary forces and attempt an escape. An important thing to note is that boids do not handle their own visible lists, the flock manages each individual boid's visible list using the look and forget functions. This is by design, having the flock manage the visible lists of the boids makes parallelizing the update functions much simpler. The eccentricity calculation used in this application is taken from Felipe Takaoka's implementation of Hartman and Benes' leadership[3]. The updated eccentricity calculation uses a Gaussian-kernel distribution to make escapes more likely and more consistent.

The application runs 16 threads to concurrently update boids and 1 thread that is idle until the simulation window is closed. The idle thread gets the peak FPS and the past 10 frames' instant FPS and calculates the average FPS (using the past 10 frames) through a custom Channel class and prints out the results for benchmarking. The flock threads need to wait for each other to finish updating their visible lists to continue with the rest of their update functions, as a data race would occur if they tried to update their position while another boid attempts to calculate the distance between them.

CPU Parallelization Strategy and Implementation

The NaiveCPUFlock class adds a boundedUpdate function, that allows threads to update their own section of the boids array in parallel. This function first loops through the boids and updates their visible lists, this is a read-only operation, so no synchronization techniques are used yet. Then the function waits at the "ready" barrier until all threads have finished updating their corresponding visible lists. The rest of the update functions are subsequently called, and the boids' visible lists are cleared. The drawing of the boids is handled in the main thread after all

[1] https://dl.acm.org/doi/10.1145/37402.37406
[2] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=402f6f33fd2db4f17b591002723d793814362b1e
[3] https://github.com/felipe-takaoka/boids-leadership/tree/main

threads have successfully joined back with the main thread. This is because of the way OpenGL (and thus SFML) works. Only one thread can interact with the OpenGL context at the same time, so if each thread handled its own draw calls, a mutex would have to be used with the window and the threads would spend more time waiting on each other than it takes to loop through all the boids once and draw them in the main thread.

The NaiveCPUFlock class also overloads its parent's virtual update function, to allow for runtime polymorphism and simpler main function logic. This function sections the boids evenly and binds a reference to the boundedUpdate function and passes the lower and upper index bounds. The main thread then waits to join all the main threads and handles the boids' draw calls.

On the other hand, the handler thread is spawned before the window is opened with a lambda, which captures the consumer side of a Channel of Stats (a struct that contains peak FPS and a queue of the last 10 frames' FPS). It then attempts to read from the channel and is blocked until either a Close or KeyPressed (space) SFML event occurs. When the simulation window is closed with either event, the producer end of the channel writes the current stats to the Channel and unblocks the handler thread, calculating and printing the results. This is not incredibly useful; however, in a larger scale project with more events, handling events in another thread can significantly reduce input lag and cut down on frame times.

GPU Parallelization Strategy and Implementation

The GPUFlock class overloads the virtual update function declared in the Flock class, which allows it to use the same runtime polymorphic Flock pointer as the sequential and CPU parallelized implementations. The update function first allocates the USM pointers. The visible pointer is for the results array, so enough memory is allocated for n^2 elements, where n is the flock size. This is because it is theoretically possible to have every boid be in visible range of each other. The memory is then populated with necessary data. This includes "flattening" the boids into FlatBoid structs that only hold their position, their visibility radius, and their id. This is done to minimize the amount of memory being shared with the GPU and lower overhead. The toroidal distance between the boid at the first index and the second index is calculated and compared to the boid at the first index's visibility radius. If it is visible, it is added to the visible array. The main thread waits for the kernel to finish executing and then loops through the visible array and pushes the boids defined in the array. This is because unfortunately, SYCL DPC++ kernels don't support STL containers. The rest of the update functions are subsequently called, and the boids visible lists are cleared. Finally, the USM pointers are freed.

CPU Performance Evaluation

This application was run on Windows 11 with an Intel i9-11900H CPU @ 2.5 GHz (16 logical processors) with a flock size ranging from 100 to 500 with 16 flock threads and a flock size of 200 with flock thread count ranging from 2 to 32. All tests are an average of 5.

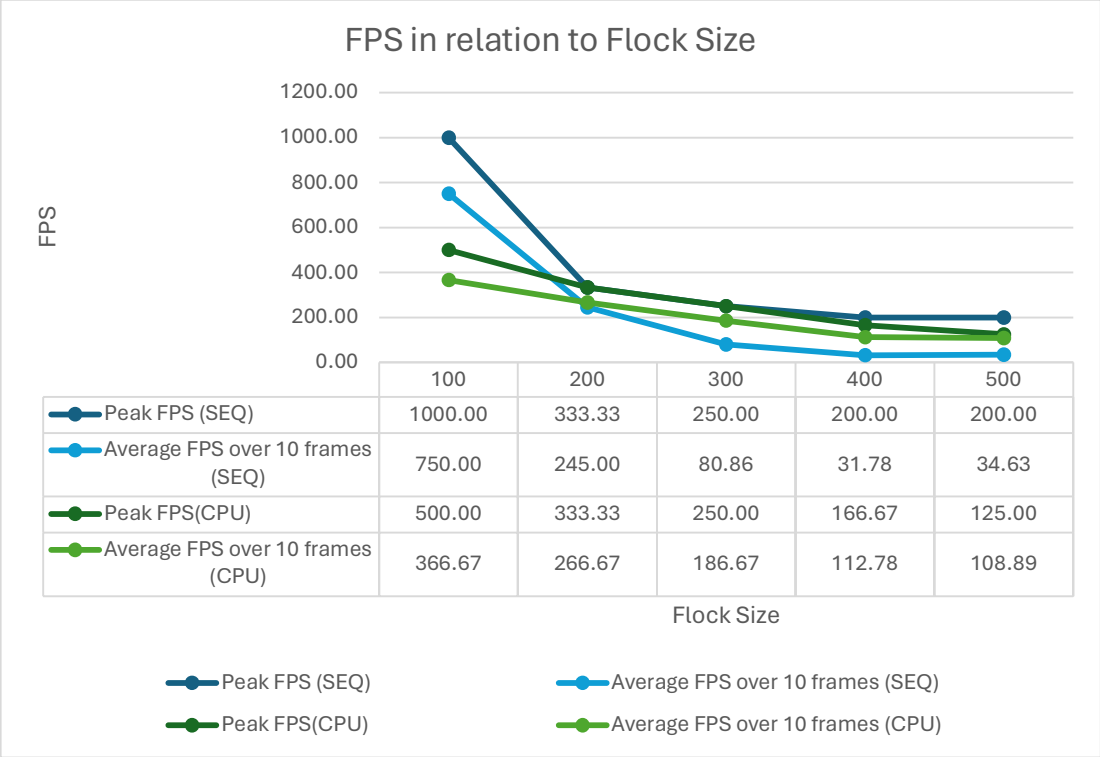


Figure 1. FPS in relation to Flock Size

Here it is clear that although the sequential flock is quicker than the CPU parallelized flock with 100 boids, the CPU flock is consistently better and more stable than its sequential counterpart. The sequential flock's peak FPS is better, however, this is not a good metric, as spikes in FPS are common and throw off the data. We can see that the lowest average FPS produced by the CPU flock is 109 FPS. Most monitors have a refresh rate of 60Hz, which means that any FPS above 60 is imperceptible, as the monitor can only refresh 60 times in a second. The sequential flock on the other hand, dips into 32 FPS, significantly below the 60Hz threshold and visibly lagging.

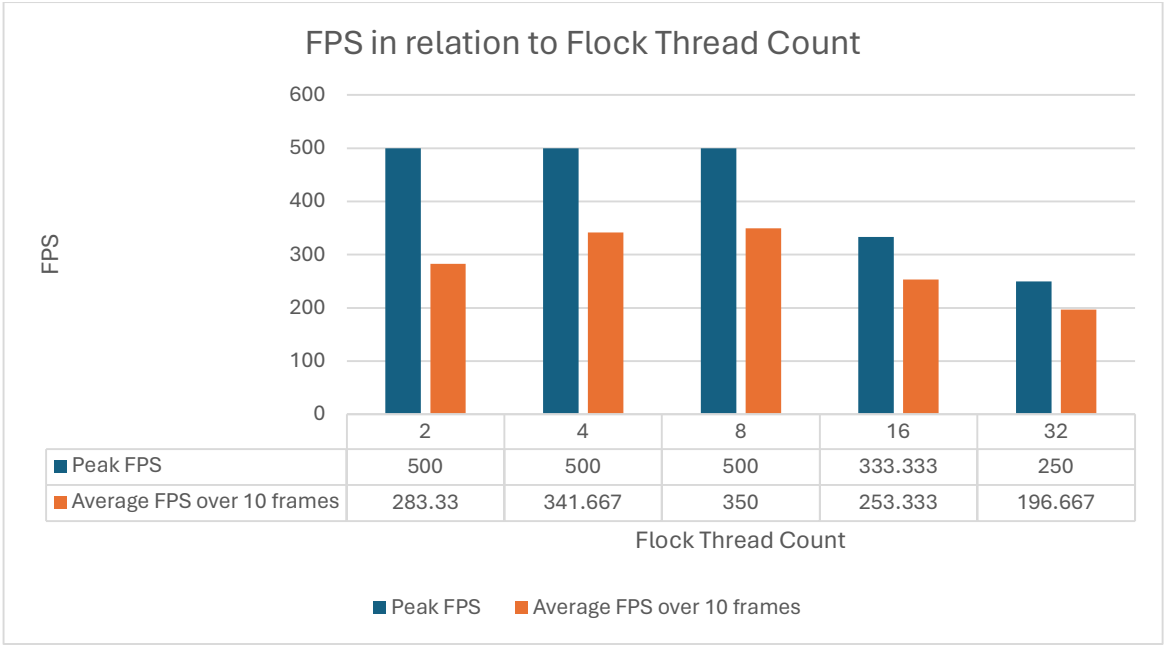


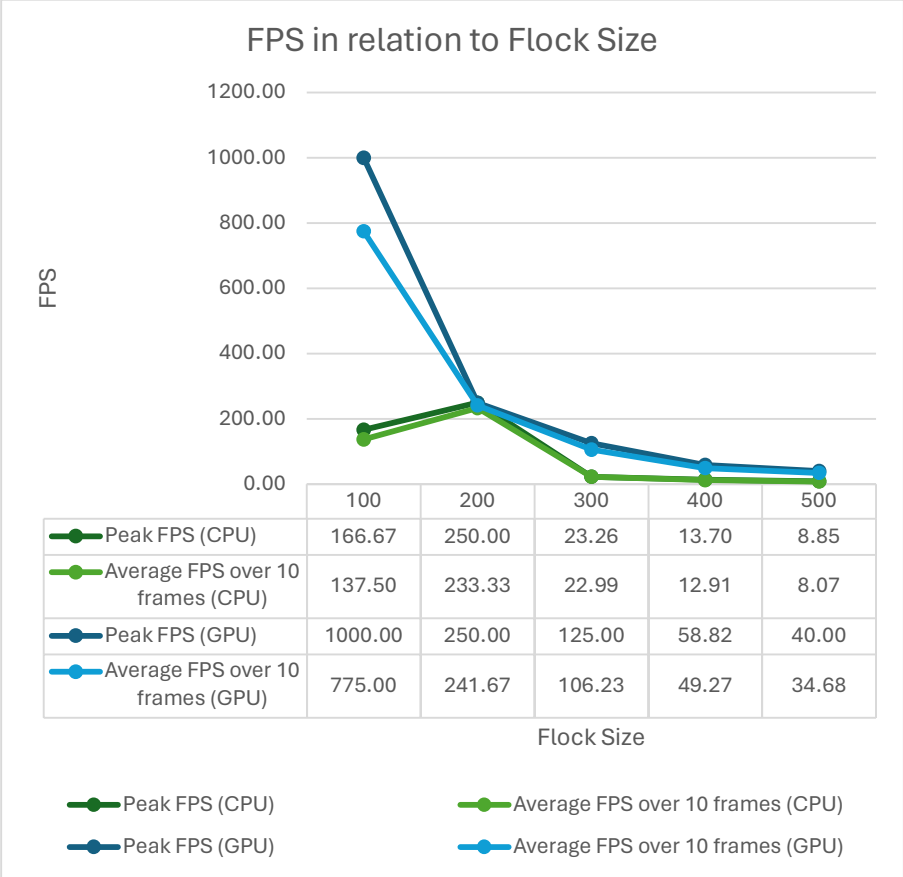
Figure 2. FPS in relation to Flock Thread Count

In Figure 2, the number of threads is increased from 2 threads to double the testing machine's logical processors. There are clear diminishing returns and after a measly increase of ~8 FPS from 4 to 8 threads, the performance starts to deteriorate to worse than 2 threads. This is most likely caused by the threads having to wait for more threads to finish their look operations and using the same amount of time less efficiently.

The parallelized solution is significantly faster than the sequential version; although, this does not mean there aren't improvements to be made. Initially, the design was meant to split the boids into chunks, based on their location on the screen. This was not possible, partly due to bad planning, frustration, and overcomplication. A chunk-based solution would work better, as instead of iterating through the same number of boids in parallel, a chunked solution would iterate through less total boids. This would optimize the time complexity of the algorithm rather than speed up the execution of the algorithm, whilst also running in parallel. The final product is a simplification and refactoring of the initial plan, and there is room to grow.

GPU Performance Evaluation

This application was run on Windows 11 with an Intel i9-11900H CPU @ 2.5 GHz (16 logical processors), unfortunately Nvidia GPUs are not compatible out of the box with Intel's oneAPI (and thus DPC++) so the GPU device selected was the Intel UHD integrated graphics. Tests were run with a flock size ranging from 100 to 500. All tests are an average of 5. This evaluation is compared against the same kernel running on the CPU device.



	100	200	300	400	500
Peak FPS (CPU)	166.67	250.00	23.26	13.70	8.85
Average FPS over 10 frames (CPU)	137.50	233.33	22.99	12.91	8.07
Peak FPS (GPU)	1000.00	250.00	125.00	58.82	40.00
Average FPS over 10 frames (GPU)	775.00	241.67	106.23	49.27	34.68

Figure 1. FPS in relation to Flock Size

From the tests that were run, the GPU was quicker; however, with 200 boids, the CPU matched the GPU's speed. This could be because the CPU device does not have to move memory from the host to the device, thus it saves on overhead. With the GPU device the FPS never dipped below 30. This implementation of was not optimal and has a lot of flaws, namely the number of loops iterated in the host. Since most of the boid update functions cannot occur on the kernel due to a fundamental incompatibility in the way the sequential Boid and Flock classes were implemented.