15418 Project: Optimizing Parallelization of Boid Simulation

Haley Carter and Claire Ko

May 6, 2022

Abstract

We implemented an optimized Boid Movement Calculation on the NVIDIA GPUs in the lab, using CUDA and OpenMP. We used various work distribution strategies and different methods of neighbor computation to create a more efficient simulation of a flock of boids.

We plan to show a visualization/animation representation of the boid simulation for the demo. In this final report, we analyze our CUDA and OpenMP version, specifically what challenges and bottlenecks prevented us from being able to achieve better speedup. We also include speedup graphs to demonstrate the improved performance of our CUDA and OpenMP implementations, focusing on reducing computation time for a fixed problem size.

Contents

1 Background 3

2 Approach 5

3 Results 10

4 References 14

5 Work and Credit Distribution 14

1 Background

The "boids" algorithm seeks to simulate the natural flocking motion of animals, primarily birds. Simulating the behaviors of a large number of moving agents interacting with one another is important in fields such as virtual reality and computer animation (Reynolds, 1987). The boids algorithm also has much in common with other n-body and particle system simulations, and has therefore seen use in domains related to those problems, such as weather simulation and data visualization. Rather than control the motion of the flock globally, the algorithm creates emergent flocking behavior through a few simple behaviors performed by each boid, based on the limited information within its range of vision. When simulating the flock at each time step, each individual boid updates its position and velocity towards some goal based on three forces. The image below describes the three rules for boid movement, in relation to the neighbors within the limited vision of each individual boid:

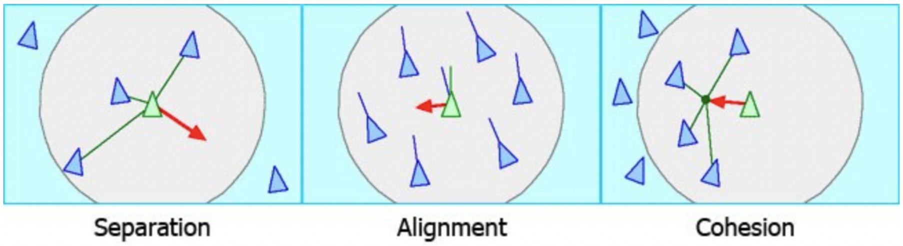


Figure 1: Boid movement rules.

The general structure of the algorithm is as follows: for each time step, for each boid, find the boid's neighbors. For each neighbor, calculate its effect on separation, alignment, and cohesion forces. Calculate the goal-seeking force. Finally, combine those forces to calculate the new velocity and position.

Key data structures include the boid structure, which includes position and velocity in a 2D space, as well as the neighbor data structure. This neighbor structure is a uniform grid across the 2D space, where each grid contains a list of boids in that grid. Key operations include updating the boid's velocity with the movement rules, such as separation and cohesion, and then computing the boid's next position using the updated velocity (Zhou and Zhou, 2004). Finally, we recompute the grid, to get an accurate view of the current position of all boids.

The input to the algorithm is the number of boids and their initial starting positions, as well as the size of the space. We generate 3 different input files of this kind, labeled easy, medium, and hard, with the difference between them being the number of boids. At each time step, we output a file of boid positions, and repeat over any number of time steps. We are able to string these output position files together, to render a continuous representation of the boids using a Python script.

Parallelization:

A major aspect of the boids algorithm that might benefit from parallelism comes from determining the neighbors of each boid. The naive approach has $O(n^2)$ complexity in the number of boids in the flock, which means that the sequential implementation is not very scalable in the number of boids. While this is a challenge for implementation, any potential speedup from a parallel implementation is highly useful. Another aspect of the boids algorithm which could benefit from parallelism is the fact that each boid in the algorithm is performing the same computations in each iteration. Therefore, the algorithm could benefit from SIMD instructions. Reducing the computation time of large scale simulation to real time would allow for significantly more use of the algorithm, such as real-time animation in video games.

As mentioned above, each boid is dependent on its neighbors in the flock, but due to the fact that all boids are constantly updating their positions, the neighbors of a boid will change over time. This necessitates updating the set of neighbors for each boid, which naively has O(n^2) complexity. Therefore, we explore more efficient approaches to finding the neighborhood of each boid.

Additionally, the changing nature of the system combined with the dependence on nearby neighbors provides another challenge for workload distribution, as there could be possibly divergent execution due to varying number of neighbors. However, there is opportunity for optimization as there is high locality between boids near one another in each particular time step, which we could take advantage of with our workload distribution. We will need to consider which partition schemes are most effective for dynamic workload balance, as boids potentially cluster in certain areas of the environment or shift position between spatial divisions rapidly. Updating the partition between processors is costly, so we want to minimize the amount of dynamic load re-balancing we need to perform while still ensuring good load balance.

Our goal was to improve on the current boid algorithms, to build a parallel program with optimized load balancing to simulate a smooth, generally correct movement for each boid.

We compare the CUDA model using GPU with the OpenMP model using CPU. We also determined the tradeoff between better global movement, that accurately follow the movement rules, versus better speedup.

2 Approach

Serial

We initially began by programming a sequential implementation which could provide a correct solution to the problem. We followed a pseudocode from one of our sources (Prudence M Mavhemwa, 2018), to begin the basic boid update algorithm in C. We decided to create a new data structure to more efficiently compute neighbors for a thread, where each grid spot would be a linked list of boids that are in that grid spot. Thus, we only look at the grids just around the boid we are currently computing, to find the neighbors of the boid less than O(n) time (in most cases, because worst case is still O(n)). Our linked list is a doubly linked list, using C macros and an embedded link inside each boid.
As an overview, our approach to parallelizing our algorithm was to begin with a sequential implementation, find the simplest way to parallelize that implementation, then make step-by-step improvements to our parallel implementation. When determining what aspect of the implementation to improve next, we sometimes improved whatever aspect was clearly very easy to fix, and we otherwise looked for the current major limit on performance and what changes could be made to mitigate that.

OpenMP

To create a parallel version, our first decision was whether we wanted parallelism across boids or across space. Based on our sequential code, we realized it was very straightforward to introduce parallelism across possible paths for boids, as there was already a for-loop in the sequential code which computed the updated velocity and position for each boid. As such, we decided to start with parallelism across boids. Performing an initial test on the GHC development machine, we saw limited speedup (around 2x for 4 threads, and then around 1x for 16,64,128 threads, compared to 1 thread). Thus, we looked to analyze which section of the code was taking the most time.

We added timing code around each section, in the code for updating each boid. The first section was the neighborhood computation, the second section was the boid update using the computed neighbors, and the third section was the updating of the grid. The first two sections were parallelized using a pragma omp for loop with dynamic scheduling, while the last section was sequential. For the parallel section, we computed the times for each thread, over each time step iteration, storing the times in an array of size num_threads. From the timing code, we saw that there was a large difference between the neighbor computation time between threads, indicating possible uneven work distribution, as the min and max runtime over all threads had a difference of about 0.0007 to 0.0164 ms, for the medium test case and 16 threads. We saw similar results across the test cases and higher thread counts.

Thus, we reasoned that parallelizing over boids was too much overhead, for both static and dynamic scheduling. This may be because the time it takes a thread to calculate a boid is not fixed but depends on the number of boids in its view, or influence range, and so each threads' runtime is slightly different based on the boids it is computing. Thus, the implicit synchronisation at the end of the parallel section also introduces new waiting times. We reasoned that parallelizing over grids/spatially may be better than parallelizing over boids. Each grid contains a few boids or less, so if each thread is in charge of computing the boids in its grid chunks, then it may be more equal work distribution across threads if we interleave the grid chunks that each thread is in charge of. In this way, large boid chunks would be split over grids, and each grid chunk (along with its boids) would be processed by one thread. We implemented this by parallelizing a loop over the grids, rather than boids, using static scheduling.

Measuring on the GHC development machine, this change increased our speedup in the medium and hard cases to around 4.5x at 64 threads, though the impact wasn't as large in the easy case. This is due to the problem size being too small for evenly sharing across threads, as the easy case has 10 boids and thus is not evenly split over more than 4 threads. The medium and hard cases have 100 and 1000 boids respectively, and so will benefit from parallelism more than the easy case.

Next, we decided to experiment with different parameters to the OpenMP pragma omp for loop. We first tried static scheduling using chunks of varying sizes, due to our hypothesis about the interleaved grid chunks being a more even workload distribution amongst threads. However, this did not improve our speedup, and so we tried dynamic scheduling using chunks of various sizes, because we saw that our time measurements were still showing a considerably large difference between the min and max thread computation times.

After making this change, using dynamic scheduling with a chunk size of 300, we observed an improvement in larger thread count speedup. The speedup for 16 threads greatly increased to around 5x and 6x for the medium and hard cases, becoming the peak speedup across our input files. We also saw that our time measurements were showing a smaller difference between the min and max thread computation times, supporting our conclusion that parallelizing across grids, with the correctly tuned OpenMP parameters, had more even workload distribution across threads. This may be because chunking the grids allow for less overhead, as each thread does not have to dynamically reschedule often. Instead, they work on their individual chunks and reschedule after computing their chunks. By adding a chunk size to the dynamic scheduling, each thread grabs "chunk" iterations off queue until all iterations have been scheduled, resulting in better load-balancing for uneven workloads.

Lastly, we considered parallelizing the sequential update to the grid data structure. We tried implementing more parallelization in how we updated the grids, after every boid's position is updated. For example, we parallelized a for loop over grid chunks, similar to the boid updates, so that each grid's boids were updated by an individual thread. Measured on the GHC machine, we saw a significant loss in speedup over higher thread counts. In our timing code, we saw the times for updating the grid drastically increase from 0.0017 ms to 0.9732 ms, averaging across thread counts in the medium case. We reasoned that there was too much overhead in spawning threads, compared to the amount of work done by each thread (i.e. a few pointer changes, in a doubly linked list, per thread). Also, since we need to remove and add to lists in an atomic fashion, so as not to break the linked lists, there was high synchronization overhead across threads. Thus, we removed this from our final code.

Over all our updates, we did change the original serial algorithm to enable better mapping to a parallel machine, in terms of what we were looping over (originally boids, then changed to grids).

Our measurements for OpenMP were taken on the GHC machines, due to connection issues with the PSC machines.

CUDA

When creating the CUDA implementation of our algorithm, our first step was to make a basic, correct implementation of the boids algorithm, ensuring that data structures were being copied correctly from the CPU to the GPU. As a result, our CUDA implementation didn't recreate the uniform grid data structure initially. It instead used the naive $O(n^2)$ strategy where each boid checks of all other boids when determining its neighborhood to compute updated boid positions and velocities. In terms of memory management, the CUDA implementation loads the initial boid data into CPU memory in the same way as the sequential and OpenMP implementations, then only copies position and velocity data to the GPU's global memory, without recreating the uniform grid data structure. Additionally, because the focus of the initial implementation was on correctness, it creates two arrays of boid data in GPU global memory, representing input data and output data, to ensure that the data each boid reads from its neighbors isn't modified before all boids have performed their updates.

For each iteration of the update function, the CUDA implementation makes a kernel call to update the boids, then makes another kernel call which copies memory from its output to its input data array to reset for the next iteration. As mentioned before, the initial CUDA implementation used the naive strategy where each boid runs a for-loop which checks all other boids to determine if they are close enough to be neighbors. While we considered structuring the code to determine the neighborhood and pass that set to separate functions which would calculate the forces, we realized that the limited local memory for each thread made this strategy unreasonable. Additionally, since the thread loads a boid's position data in order to determine if it's in the neighborhood, we save some memory accesses by re-using that data immediately to get the partial sums used for computing the average position and velocity of the neighborhood.

After ensuring that our implementation was correct, we made a basic improvement to the way we store the boid's data in the GPU's memory. Initially we used the same data structure as the CPU implementations, which stored the position and velocity data of a boid as two structs which each contain two floats (for the x and y dimension). For the CUDA implementation, we instead store an array of float4's in the GPU memory. This allows the compiler to format our memory accesses as vectorized load/store instructions, reducing the total number of memory access instructions and increasing memory bandwidth utilization. This did come with the downside that copying data for output between the CPU and the GPU was more difficult, since the data on the GPU was stored in a different format than on the CPU. Our solution was to create a mirror float4 array on the CPU. While this made the memory footprint on the CPU larger, it ensured that we could continue to use the memcpy instruction to move data from the CPU host to the GPU device, which was the memory transfer with the most latency. Once the float4 data was on the CPU, we then reformat it to match the expected output format. In line with this, another minor change we made was to move the kernel call which copies data from output to input before the kernel which updates the boids. This ensures that we only perform this memory copy for the current iteration as we need the new input data, rather than having one extra copy at the end of execution when there will be no further iterations which needs that input data. After making these changes and testing on the GHC machines, we compared performance on the hard test case, which had the most boids and therefore the most memory accesses due to the naive neighbor determination strategy, and saw a decrease in the average time per iteration from around 1.83 ms to 1.36 ms, about a 30% speedup.

Our next major planned change was to improve our algorithm from the naive O(n^2) neighborhood check to the uniform grid space partitioning scheme. This was in part motivated by the results we saw when using nvprof to check our CUDA implementation. The analysis showed that one of the largest issues with the current strategy was divergent execution. Especially in the medium case, which only has 250 boids and therefore fits all boids inside the same block, a full for loop over all boids means that many threads won't be utilized when the current boid being processed isn't in their neighborhood. Therefore, we wanted to adapt the uniform grid strategy for the CUDA implementation, to reduce the number of boids each thread checks against when determining the neighborhood and to potentially take advantage of data locality, both in the assignment of threads to boids and in memory accesses. Unfortunately, we were unable to complete implementing this feature before the final project due date, as we underestimated the complexity of the algorithms involved and the amount of time required to fully understand them and implement them correctly. However, we'd like to present what our plan was, in part to explain what flaws we saw in the existing design and what strategies we think the CUDA implementation could have used to resolve them. We have also included a copy of the partial implementation in our github repository, called "cudaBoids-partial.cu".

The uniform grid partition used by the CPU implementations required both a 2d array and a linked list for each grid square to hold the dynamically-changing number of boids in that grid. This strategy didn't make sense of the GPU. First, the communication and conflict involved in threads modifying the same linked list in parallel would likely defeat the purpose of speeding up the algorithm. Secondly, a 2d grid would be best to take advantage of data locality. If we tried to avoid a linked list based grid by having each grid square only contain a single boid, the amount of memory required would become O(n^2) and wouldn't scale. After doing research on the existing literature, we saw the existing implementation of the boids algorithm for the GPU solved these issues through two strategies, hashing and space-filling curves. The modified CUDA implementation would first update its grid data structure before updating the boids. First, a kernel call is made where each boid uses its position to calculate the (x,y) value of the grid square it would belong to in a 2d grid, hashes that 2d value to a 1d integer using a Z-order function, and stores that hash in a global memory array. Our implementation of the Morton Code map was based on the pseudo code given on the Wikipedia page for Z-order curves. We can then use the array of hashes to sort the boid data into the input data array in the order which corresponds to our 1 dimensional map of the 2d uniform grid. Using the sorted array of hashes, we can determine the subsequence of indices for the boids in each grid square using a similar strategy to that used for the scan implementation in Assignment 2. Each boid in parallel checks its hash in the sorted list of hashes, then checks if the hash of the previous boid in the list matches its own. If it doesn't, then that boid must be the first boid in that grid square, so it can use its hash (i.e. the value of its grid square) to index into an array which stores the starting index for each grid square and store its index as the starting index.

If we were able to complete the implementation of this change, we would have significantly reduced the divergent execution in the kernel which updates the boids. Additionally, the Z-order curve would allow us to reduce the memory footprint compared to a 2d uniform grid while still allowing us to take advantage of data locality. This could have also allowed us to take better advantage of shared block memory. Using the spatially-ordered list of boids, we could ensure that threads in the same warp or block would have high spatial locality, which would allow us to take advantage of shared memory loads when performing the for loop which checks for neighbors in the surrounding squares in the uniform grid.

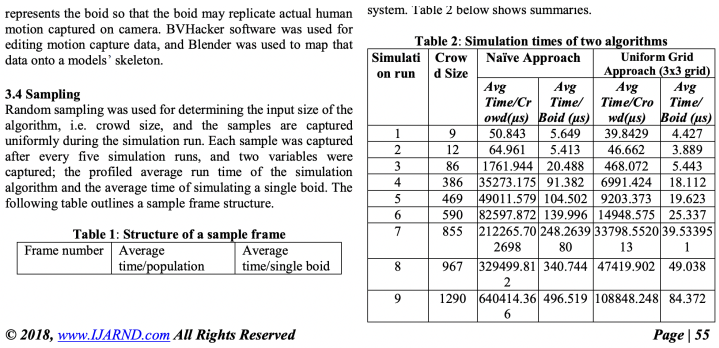
3 Results

OpenMP

Throughout our development, we measured our performance through speedup while varying thread count, computed on the GHC development machines. The time we measured was for each time step, where there is an update boids step and an update grid step. We averaged the measured times across time steps, to get the average updateScene() time in ms.

Our input files were generated by a Python script we wrote, which randomly initialized a given number of boids randomly across a 1000 by 1000 space. The easy test case has 100 boids, the medium has 100 boids, and hard has 1000 boids. We output the boid positions at each time step, to be able to run it through a Python script that visualizes and plots the boids at incrementing time steps. In running the algorithm, the user passes in the thread count, the input file, and the number of time steps to compute and output.

The resulting speedup for each of the input files is as follows. The baseline is an optimized parallel implementation, for 1 thread. A baseline sequential, taken from a 2018 paper, is also included below. Our baseline times correspond to the times in the 2018 paper, and is slightly faster for similar number of boids.



represents the boid so that the boid may replicate actual human motion captured on camera. BVHacker software was used for editing motion capture data, and Blender was used to map that data onto a models' skeleton.

3.4 Sampling

Random sampling was used for determining the input size of the algorithm, i.e. crowd size, and the samples are captured uniformly during the simulation run. Each sample was captured after every five simulation runs, and two variables were captured; the profiled average run time of the simulation algorithm and the average time of simulating a single boid. The following table outlines a sample frame structure.

Table 1: Structure of a sample frame

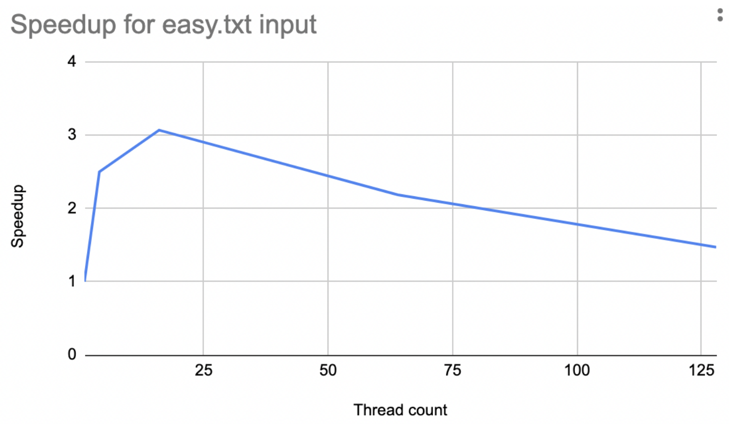
Frame number	Average time/population	Average time/single boid

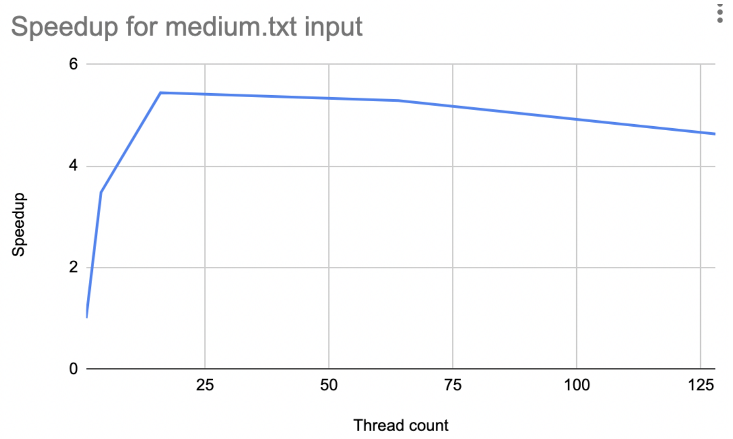
system. Table 2 below shows summaries.

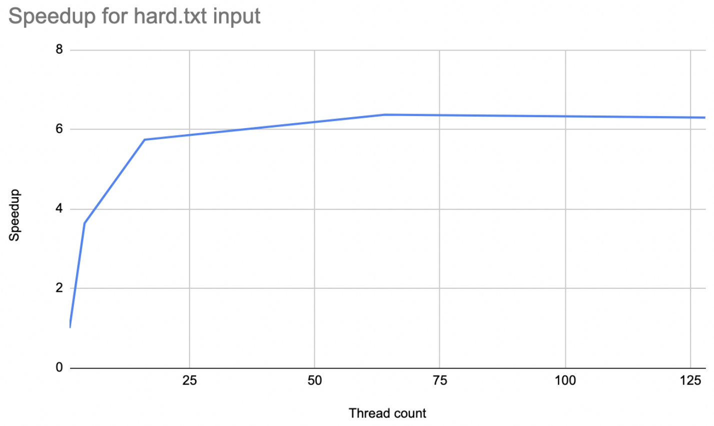
Table 2: Simulation times of two algorithms

Simulation run	Crowd Size	Naïve Approach		Uniform Grid Approach (3x3 grid)	
		Avg Time/Crowd(µs)	*Avg Time/ Boid (µs)*	*Avg Time/Crowd(µs)*	*Avg Time/ Boid (µs)*
1	9	50.843	5.649	39.8429	4.427
2	12	64.961	5.413	46.662	3.889
3	86	1761.944	20.488	468.072	5.443
4	386	35273.175	91.382	6991.424	18.112
5	469	49011.579	104.502	9203.373	19.623
6	590	82597.872	139.996	14948.575	25.337
7	855	212265.702698	248.263980	33798.552013	39.533951
8	967	329499.812	340.744	47419.902	49.038
9	1290	640414.366	496.519	108848.248	84.372

© 2018, www.IJARND.com All Rights Reserved ***Page | 55***







We can see that different problem sizes results in very different raw speedup values, as well as a different trend over increasing thread counts. For the easy.txt input, since there are only 10 boids, there is not much parallelization benefit as with higher thread counts, most threads will be doing no work as the problem is too small. We can see that the peak speedup is at 16 threads, and then decreases greatly from there. For medium and hard, their trends are similar. However, hard has a much higher speedup at 64 and 128 threads. This may be because with more boids, the boids separate into individual "flocks" that cover different grid areas, allowing for more parallellization between flocks. Also, the boids will be more evenly spread across the space, allowing for more even work distribution and better utilization of parallel speedup, as the grid size remains the same. However, visually the hard input did much poorer (visually worse flocking behavior, in terms of the movement rules), possibly because of update synchronization issues. As a boid is being updated by a thread, other threads are reading its value. Thus, even if a boid moves out of the grid, its grid location is not updated, resulting in other threads reading stale/incorrect data about that boid.

There are multiple factors which we think explain why our code is unable to achieve perfect speedup. First, we are likely limited by the amount of sequential work which is done. Each grid update is sequential, as we tried and did not succeed in parallelizing that sequential aspect. Thus, with this fixed fraction of the computation that is inherently sequential, it is impossible to achieve perfect speedup. In our timing code, this sequential update takes 0.00477 ms, out of the total time of 1.03695 ms (measurements taken from medium test case, with 16 threads). Although this is not a large percentage of the total computation time for 1 time step, it is still sequential and prevents perfect speedup. Second, we use shared memory in our current implementation, specifically sharing the boid array as well as the grid array. This may not be ideal for memory accesses, because using perf, we see that we have a cache miss rate of 10.863 percent of all cache references. This is quite high, suggesting that the sharing of global memory and constant, same-time accesses to these global arrays for all threads, may be causing false sharing and/or coherence misses.

One possible optimization we could do, to optimize communication between threads, is to sort the boids by their location, so that we can take advantage of locality, as a thread will access only one area of memory, and will not conflict with a thread that is accessing a different part of the grid. In this way, we can decrease the number of threads contending for a specific area of shared memory.

In breaking our computation into the 3 chunks mentioned above (parallel neighbor computation, parallel boid velocity/position update, and sequential grid update), the time taken for each section is 0.455281, 0.003329, 0.000453 ms respectively (for medium input with 16 threads). We can see that the neighbor computation takes the most time, because we are reading all other boids in our range of vision. The min and max times all thread take across this computation has a difference of 0.011715 to 0.037726 ms (about 300 percent), showing that there is room for improvement, although our optimizations helped. For example, we could try to, instead of holding a constant grid partition across the space, we can do dynamic grid partitioning, every few time steps, to better distribute the workload during the neighbor computation between threads. Currently, grid chunks that contain more boids or have a huge number of neighbours do more work compared to other processes, whose grid chunk might only contain very few boids. Although dynamic allocation of work to threads, using OpenMP's dynamic scheduling, helps with this issue, a consideration would be to dynamically resize the grids, which could help in balancing the workload.

CUDA

Our configuration for measuring the speedup of our CUDA implementation was the same as for the OpenMP implementation. We used the same inputs, tested on the GHC development machines, and measured the average time in ms per iteration to update boids and maintain data structures. For the CUDA implementation, we assigned each thread to an individual boid, so the way that we partitioned work was the same across all input problems.

Input	Baseline Time per Iter	Average Time per Iter	Speedup
easy.txt	1.455722ms	0.0342496ms	42.50
medium.txt	4.472816ms	0.036224ms	123.48
hard.txt	39.226498ms	0.141229ms	277.75

We decided to base our goal speedup on the hardware of the GHC development machines. Its GPU is the GeForce RTX 2080 B GPU, which initially sold for $699 to $799. Its CPU is the Intel(R) Core(TM) i7-9700 GPU @ 3.00GHZ (Model 158), which initially sold for $323. Since a CPU is useful for a larger range of problems than a specialized piece of hardware like a GPU, we decided to use the difference in cost to judge whether the GPU was worth it, so to speak. Since the CPU goal was 8x speedup, for 8 CPU cores, and the GPU costs 2.16 to 2.47 times as much, our goal was to achieve 17.5 to 19.76 times speedup with our CUDA implementation. By that metric, our GPU implementation was a success.

However, as mentioned in our previous section, we feel that we weren't able to fully utilize the GPU in our CUDA implementation. Our main issue was with under-utilization of threads due to divergence in execution in the central for-loop of the update kernel. In conjunction with that, our current implementation is limited because it doesn't take advantage of data locality and shared block memory, which would improve our speedup further.

4 References

References

Prudence M Mavhemwa, Ignatius Nyangani (2018). "Uniform spatial subdivision to improve Boids Algorithm in a gaming environment". In: URL: https://www.ijarnd.com/manuscripts/v3i10/V3I10-1144.pdf.

Reynolds, Craig W. (1987). "Flocks, Herds, and Schools: A Distributed Behavioral Model". In: URL: https://dl.acm.org/doi/pdf/10.1145/37402.37406.

Zhou, B. and S. Zhou (2004). "Parallel simulation of group behaviors". In: URL: https://ieeexplore.ieee.org/abstract/document/1371337.

5 Work and Credit Distribution

Claire Ko (yinghork):

- Implemented OpenMP parallelization and optimization
- Wrote initial sequential baseline
- Wrote Python scripts for generating inputs and visualizing outputs

Haley Carter (hcarter):

- Researched existing work on the boids algorithm
- Implemented CUDA parallelization and optimization