Bachelor of Science in Computer Science
February 2018

Performance Evaluation of Boids on the GPU and CPU

Sebastian Lindqvist

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Sebastian Lindqvist
E-mail: seli13@student.bth.se

University advisor:
M.Sc Diego Navarro
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Agent based models are used to simulate complex systems by using multiple agents that follow a set of rules. One such model is the boid model which is used to simulate movements of synchronized groups of animals. Executing agent based models partially or fully on the GPU has previously shown to increase performance, opening up the possibility for larger simulations. However, few articles have previously compared a full GPU implementation of the boid model with a multi-threaded CPU implementation.
Objectives. The objectives of this thesis are to find how parallel execution of boid model performs when executed on the CPU and GPU respectively, based on the variables frames per second and average boid computation time per frame.
Methods. A performance benchmark experiment will be set up where three implementations of the boid model are implemented and tested.
Results. The collected data is summarized in both tables and graphs, showing the result of the experiment for frames per second and average boid computation time per frame. Additionally, the average results are summarized in two tables.
Conclusions. For the largest flock size the GPGPU implementation performs the best with an average FPS of 42 times over the single-core implementation while the multi-core implementation performs with an average FPS 6 times better than the single-core implementation. For the smallest flock size the single-core implementation is most efficient while the GPGPU implementation has 1.6 times slower average update time and the multi-core implementation has an average update time of 11 times slower compared to the single-core implementation.

Keywords: boid, ABM, agent based model, GPGPU

Contents

Abstract **i**

1 Introduction **1**

1.1 Hypothesis and Research Questions 2
1.2 Outline 2

2 Related Work **3**

2.1 Previous Research 3
2.2 Background 4
2.2.1 The Boid Model 4
2.2.2 GPGPU 5

3 Method **6**

3.1 Development Tools 6
3.2 Implementations 6
3.2.1 Single-core CPU Implementation 7
3.2.2 Multi-core CPU Implementation 8
3.2.3 GPGPU Implementation 9
3.3 Experimentation 9
3.3.1 Experimental Setup 9
3.3.2 Test Cases 10
3.3.3 Validity Threats 10

4 Results **12**

4.1 Single-core CPU 12
4.2 Multi-core CPU 13
4.3 GPGPU 14
4.4 Summary 15

5 Analysis and Discussion **17**

5.1 Multi-core 17
5.2 GPGPU 17

6 Conclusions and Future Work **19**
6.1 Conclusions 19
6.2 Future Work 19

References **20**

A Metrics **22**

B Code **28**
B.1 CPU helper functions 28
B.2 Single-core CPU update function 31
B.3 Multi-core CPU update function 32
B.4 Multi-core CPU thread function 33
B.5 GPGPU update function 34
B.6 Compute shader 35

List of Figures

3.1 *Example initial positions and directions of flock size 64* 7

4.1 *Average FPS of all data points for the single-core implementation.* 12
4.2 *Average update time per frame on all data points for the single-core implementation.* 13
4.3 *Average FPS of all data points for the multi-core implementation.* 13
4.4 *Average FPS of all data points for the multi-core implementation.* 14
4.5 *Average update time per frame on all data points for the GPGPU implementation.* 14
4.6 *Average update time per frame on all data points for the GPGPU implementation.* 15
4.7 *Average FPS of the implementations for each flock size.* 16

List of Tables

3.1 *Properties of the Intel Core i7-6700HQ* 10
3.2 *Properties of the NVIDIA GeForce GTX 950M* 10

4.1 *Average FPS of the multi-core and GPGPU implementations compared to the single-core implementation.* 15
4.2 *Average logic update time per frame of the multi-core and GPGPU implementations compared to the single-core implementation.* 16

A.1 *Average FPS of the single-core, multi-core and GPGPU implementations.* 22
A.2 *Average logic update time per frame of the single-core, multi-core and GPGPU implementations.* 22
A.3 *All data point averages for the single-core implementation* 24
A.4 *All data point averages for the multi-core implementation* 26
A.5 *All data point averages for the GPGPU implementation* 27

Chapter 1

Introduction

As entertainment industry move towards more advanced graphical content, new graphical techniques are utilized to create different visual effects. One technique is the simulation of synchronized groups. These simulations are based on a large amount of individuals, or *agents*, coordinating with each other by individually following a set of rules. This computational model is called *agent-based model* (ABM) [4].

One of the first ABMs for simulating movement in a group of animals was proposed by Reynolds in 1987 and is based on three rules that each agent follows; collision avoidance, velocity matching, and flock centering. Reynolds called the agents in the model *boids*, an abbreviation to *birdoids*, which have been commonly used ever since [7].

In a naive implementation of the boid model where every agent calculates each rule against all other agents in the flock, the algorithm would get a computational complexity of $O(n^2)$ where n is the flock size [7][5]. This means that as the size of the flock grows, the computation needed for the simulation grows quadratically. Multiple implementations have been proposed to achieve simulations with larger flock sizes. Solutions include discarding one of the rules, increasing efficiency of finding neighbors, or by moving parts or the full implementation to the *graphical processing unit* (GPU) [5][3][2][8].

Utilizing *General-Purpose Computing on Graphics Processing Units* (GPGPU) with ABMs can reduce the computation time for iterations due to the GPU being optimized for executing parallel tasks. Previous research has shown that ABM implementations on the GPU can outperform CPU implementations with a speedup of up to 40 times [6]. However, there are few similar measurements for the boid model to this date.

This thesis will evaluate and compare the performance of parallel computing of the boid model on the GPU and the CPU. The experiment will be a benchmark experiment based on three variations of a traditional boid implementation; single-threaded CPU, multi-threaded CPU and GPGPU. The single-threaded CPU implementation will be used as a reference system. A series of tests will be performed where frames per second (FPS) and average computation time for all agents per frame will be compared based on varying flock sizes. Lastly, the implementations will be evaluated and discussed based on their performance in the tests.

1.1 Hypothesis and Research Questions

This thesis aims to answer the following questions:

RQ1: How does a GPGPU implementation of the boid model compare to a multi-threaded CPU implementation looking at most agents simulated in real-time?

RQ2: At what flock size does a GPGPU implementation of the boid model outperform a multi-threaded CPU implementation?

The hypothesis is that the GPGPU implementation may have a better performance compared to the multi-threaded CPU implementation when dealing with large flock sizes. The reason is due to the parallel abilities of the GPU as well as the fact that this has been previously observed on similar ABMs. However, it is suspected that for small flock sizes the multi-threaded CPU implementation will calculate agent logic faster due to less thread overhead.

1.2 Outline

Chapter 2 summarizes previous research related to performance in ABMs and additionally describes the theoretical background of this area. The tools, hardware, implementations and experiment are described in chapter 3. Chapter 4 summarizes the results from the experiment. In chapter 5 the results are discussed and analyzed. Lastly, conclusions and possible future work is discussed in chapter 6.

Chapter 2

Related Work

In this section research related to performance in ABMs is summarized and the theoretical background of this thesis is discussed.

2.1 Previous Research

Reynolds proposed the original boid model which was based on a group of agents interacting through a set of three rules:

- Avoid collision
- Match velocity
- Stay close to the flock

Agents were independently simulated through their observation of the environment which led to a computational complexity of $O(n^2)$. The author stated that "This does not say the algorithm is slow or fast, merely that as the size of the problem (total population of the flock) increases, the complexity increases even faster." To handle bigger flocks it was suggested to use spacial hashing, incremental collision detection or distributed systems [7].

Lee, Cho and Calvo proposed an algorithm for increasing the performance of boid algorithms that use spacial hashing. The method is based on the fact that the k-nearest neighbors (kNN) of boids seldom change. The algorithm can efficiently calculate whether the kNN has changed and only then re-calculate the new kNN. This improvement achieved a performance increase of 57.7% with regards to FPS [5].

Joselli et al. introduced a proximity based data structure which was called "neighborhood grid". Each cell in the grid only contain one agent and can approximate its neighboring cells. The implementation had low parallel complexity which resulted in high performance and scalability while maintaining believability. The technique was implemented in a 3D environment and tested on the GPU with a minimum speedup of 2.94 over two traditional spacial hashing methods [3].

Perumalla and Aaby did a comparison of three ABMs; Mood Diffusion, Schelling Segregation and Game of Life, comparing GPU implementations of the models to optimized traditional CPU implementations as well as equivalent implementations using ABM toolkits. Conclusions drawn were that the GPU implementations executed 100 to 1000 times faster than leading ABM toolkits "at the cost of decrease in modularity, ease of programmability and reusability." GPU implementations also gained a performance increase of up to 40 times over the CPU implementations. Lastly they discussed the challenges faced with parallel ABM execution on the CPU and the GPU [6].

More recently, Hermellin and Michel did an experimental study based on the conclusions by Perumalla and Aaby [6]. They implemented and tested four different computational models using the GPU environmental delegation principle. This principle separates the computation by moving agent behavior to the CPU and environmental dynamics to the GPU, creating a hybrid approach. Additionally the authors stated that "Especially, one major idea underlying this principle is to identify some computations (such as agent-level perceptions) which can be transformed into environmental dynamics." They concluded that while an all-in GPU approach would yield bigger performance gains, their hybrid approach improved reusability, modularity, and since the GPU didn't run the entire simulation "[...] the knowledge required is less important." [1]

In another paper Hermellin and Michel applied the above principle to the boid model which led to a speed up of 25% while also improving reusability [2].

Da Silva, Lages and Chaimowicz proposed a methodology for the boid model where boids, through visibility estimation, only considered other boids which was visible in its field of view and also not blocked by another boid. The methodology was tested in three different GPU implementations: one with GPGPU techniques using the Cg (Central Graphics) shader language, one optimized using Nvidia's CUDA (Compute Unified Device Architecture), and one naive CUDA implementation. The authors concluded that visibility culling could achieve up to three times faster update speed, with the CUDA implementations constructing the grid system quicker and the GPGPU implementation being significantly faster executing the simulation, and thus, also overall [8].

2.2 Background

2.2.1 The Boid Model

The boid model is based on agents, or *boids*, that act based on their individual perception of the environment to simulate a group of moving animals. The model achieves this by using three rules that continuously regulate the direction of all boids. The first rule is that the boid should avoid collision with other boids within a certain radius. The second rule is that boid should match its velocity to other boids. Lastly, the third rule is that the boid should move towards the average position of other boids [7].

Rules can be calculated in any order and return vectors which are added together using vector addition to get the new direction. Each rule vector is multiplied by pre-defined constants that determine the total impact each rule has on the new direction. Altering these constants will affect the behavior of the flock. The number of boids each boid perceive differs between implementations and also affects the behavior of the flock. In this thesis boids perceives all other boids. The boid model can be used to simulate different groups of animals but for the purpose of this thesis, a gathering of agents are referred to as "flock".

2.2.2 GPGPU

General-Purpose Computing on Graphics Processing Units, or GPGPU for short, is the act of utilizing the GPU for non-graphical computation. Modern GPUs are created with thousands of cores and certain parallelizable tasks can benefit from running on a GPU. Multiple smaller cores are especially effective when repeatedly executing the same operation on a data set compared to the fewer bigger cores seen in CPU's which are more suited for general computing.

Compute shaders were made to execute arrays of data in parallel on the GPU. There are several compute shader APIs, the one used for this thesis is DirectCompute.

Chapter 3

Method

In order to answer the research questions in this thesis three different implementations of the boid algorithm were implemented and tested in a performance benchmark experiment. This chapter describes the tools, hardware, implementations as well as the test cases for the experiment.

3.1 Development Tools

Implementations in this thesis are mainly written in C++ using Visual Studio Community 2017 with the exception of parts the GPGPU implementation which is written in HLSL Compute Shader. DirectX 11 is used for the graphical output.

C++ and DirectX 11 were used because the author had previous knowledge and experience with both. DirectX also gave the benefit of GPU programming and rendering support.

3.2 Implementations

The implementations simulate a flock of multiple boids in a 3D environment. Each boid is represented by a model consisting of six triangles in the shape of a pyramid. The flock is contained within a fixed area visualized with the help of a grid. In order to keep the boids interacting with each other, the boids who leave the area are relocated to the opposite side of the grid from the point that they exited. Speed and acceleration is limited in order to get a smooth movement and visible interactions between agents. All boids perceives all other boids when following the three rules described in section 2.2.1. The base loop of the three implementations can be described with the following pseudo-code:

```
WHILE isRunning
  IF shouldUpdateLogic
    SingleCoreUpdate(scene, deltaTime)
    //MultiCoreUpdate(scene, deltaTime)
    //GPGPUUpdate(scene, deltaTime)

  UpdateCamera
  Render(scene)
```

Based on the implementation the dedicated update function is called.

Rendering is identical for all implementations and also separated from the agent logic. Additionally, the new up vector for the models are calculated each frame in order to keep boid direction visible as seen in figure 3.1. The model faces are calculated each frame based on the position and direction vector of the boids.

The code for the three update implementations can be found in appendix B.

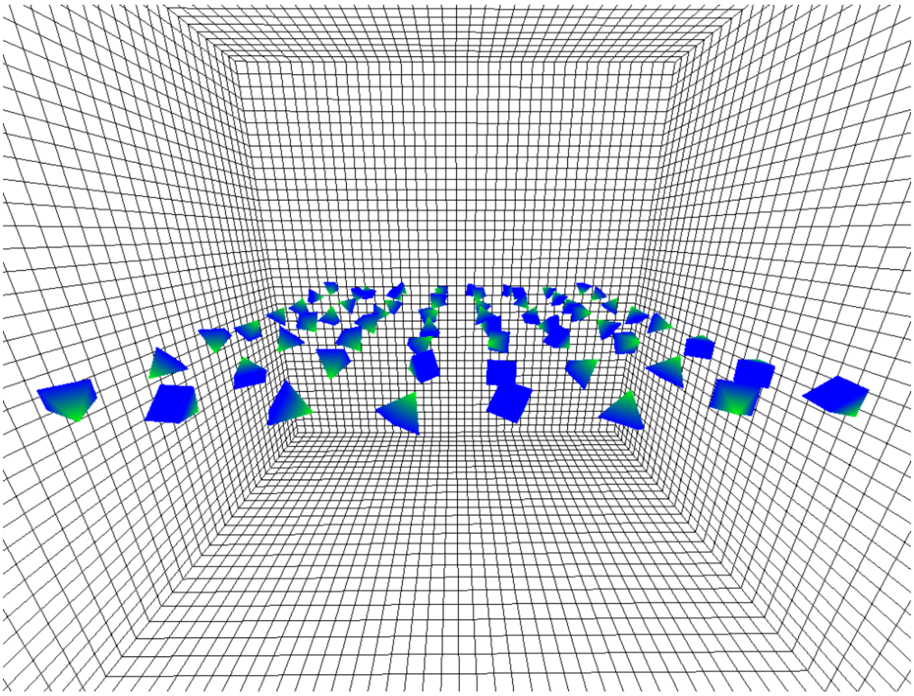

Figure 3.1: *Example initial positions and directions of flock size 64*

3.2.1 Single-core CPU Implementation

The single-core CPU implementation is used as the control point for the benchmark experiment. Agent logic is executed sequentially on the CPU. The boids are stored in two data sets where one set is used for reading the boid data from the previous frame and the other set is used for writing data for the next frame.

The single-core CPU implementation update can be described using the following pseudo code:

```
SingleCoreUpdate(scene, deltaTime)
  scene.SwitchCurrentAndPreviousBoids
  FOR each boid in scene
    calculate and add all rule vectors
    limit new direction vector size
    set new direction vector and model up vector
    calculate new position
    move position if out of bounds
    set boid position

  send boids data to GPU for rendering
```

3.2.2 Multi-core CPU Implementation

This implementation executes boid logic in parallel on the CPU. As in the single-core CPU implementation, boids are stored in two datasets. The multi-core CPU implementation uses the C++ std thread library to create eight threads; one for each hyper-thread in the CPU described in section 3.3.1. Eight threads were chosen to utilize all of the CPU cores fully while also minimizing the number of created threads. Threads compute one eighth of the boids' logic each and the program waits for all of them to finish before continuing.

The multi-core CPU implementation update can be described using the following pseudo code:

```
MiltiCoreUpdate(scene, deltaTime)
  scene.SwitchCurrentAndPreviousBoids
  initThreads(nrOfThreads)
  FOR each thread
    run boid thread function

  FOR each thread
    wait for thread to finish

  send boids data to GPU for rendering
```

The boid thread function is identical to the boid update loop in Section 3.2.1 with the addition of the range of which boid indices to update.

3.2.3 GPGPU Implementation

This implementation executes all agent logic-related functionality on the GPU. Boid positions and velocities are initiated on the CPU and then sent to the GPU memory. The data is then handled solely in the GPU memory for the remainder of the simulation. Data is stored in two buffers; one buffer for writing the boid data being used for the next frame and one buffer for reading the boid data from the previous frame. All helper functions from the CPU implementations which are needed for the boid logic are replicated in the compute shader and are designed to be as similar as possible to achieve a fairer comparison.

The GPGPU implementation update can be described with the following pseudo code.

```
GPGPUUpdate(scene, deltaTime)
  set compute shader
  set deltaTime in buffer
  send deltaTime, constants and boid buffers to GPU memory
  dispatch compute shader
  null resources
  unset compute shader
  switchCurrentAndPreviousBuffers
```

The compute shader runs 64 threads per core and uses the same logic pattern as seen in the loop of the single-core pseudo code.

3.3 Experimentation

The experiment in this thesis test three different implementations of the boid algorithm; single-threaded CPU, multi-threaded CPU and GPGPU. The single-threaded CPU implementation is be used as a reference system.

3.3.1 Experimental Setup

The tests were run on Windows 10 Home x64 with an Intel Core i7-6700HQ @ 2.6GHz processor, 8.0 GB RAM and a GeForce GTX 950M. The CPU main properties are listed in Table 3.1 and the GPU main properties are listed in Table 3.2. Each test run with a resolution of 1024x800 in Visual Studio 64-bit release mode.

The tests measures the number of frames per second and average computation time for all agents per frame. FPS is an occurring unit of measurement in boid model simulations [5][3]. FPS can show us the efficiency of the implementation work flow. Average computation time for all agents per frame can show us efficiency of the agent calculations separate from the rest of the implementation.

This can strengthen that any FPS increases observed are due to a more efficient calculation of the agents new positions.

To get FPS, a frame counter is incremented once for each frame. Each second the total frames for that second is saved to a data set. Computation time for all agents is extracted by starting a timer before the execution of agent logic is initiated and stopping the timer when logic execution for all agents is completed for that frame. That value is then added to a total execution time. Each second the average computation time for that second is calculated and the total execution time is reset. The resulting value is then saved to a data set.

Property	**Value**
Nr of cores	4
Nr of threads	8
Base frequency	2.60 GHz
Cache size	6 MB
Bus speed	8 GT/s DMI3

Table 3.1: *Properties of the Intel Core i7-6700HQ*

Property	**Value**
CUDA cores	640
Processor clock	914 MHz
Memory size	2 GB
Memory bandwidth	32 GB/s
Memory type	128-bit GDDR3

Table 3.2: *Properties of the NVIDIA GeForce GTX 950M*

3.3.2 Test Cases

Each implementation will be run through three different test cases in order to test their individual performance. The implementations will be run through the test cases three times each and will all have identical initial conditions. To observe the performance pattern when flock size grows, test cases will be run with the flock sizes 64, 512 and 4094. These flock sizes are chosen with two things in mind. The first is that all are evenly dividable by 8 as that is the number of logical threads for the CPU used for the tests as seen in Table 3.1. The second reason is that all three sizes are evenly dividable by 64, which is the number of threads used for each GPU core as described in section 3.2.3. This experimental scenario delivered 27 test runs.

All boids will start with a randomly assigned direction and speed within a set range. The camera will be fixed in its initial position. The simulation is then started by the push of a button, also initiating the selected test. The simulation will then run without interference while data is being collected. Each second the measurements are saved as data points in the memory. After 60 seconds the test ends and the collected data is saved to a file.

3.3.3 Validity Threats

The validity of the experiment is reliant on the fact that the implementations are implemented in an equivalent manner to achieve a fair comparison. To achieve this the rendering is separated from the agent logic and identical for all implementations. Additionally, optimizations are only applied when they can improve all implementations in an equal manner. Furthermore, tests will be run with the same background conditions to ensure equal processing power is offered. For each data point to have a fair value the tests are executed three times and then the average is calculated.

The results of the experiment are heavily dependent on the hardware used for the tests. To achieve a fair comparison, the CPU and GPU used for the experiment were released the same year and are both in the mid-tier price class.

Chapter 4

Results

In this chapter the Results from the experiment are summarized and discussed. Results are shown for each individual implementation and a summary is offered in section 4.4. Each data point is the average from three test runs for each flock size as described in section 3.3.2. The graphs show the FPS as well as the average computation time of all agents per frame in milliseconds. For each implementation there are two graphs containing all data points from all tested flock sizes. The compiled data points are available in appendix A.

4.1 Single-core CPU

The single-core implementation has a more predictable decrease in FPS as flock size becomes bigger compared to the other implementations. The average update time for flock size 4096 is higher than the other two flock sizes. For all flock sizes the FPS and update time remains stable throughout the simulation.

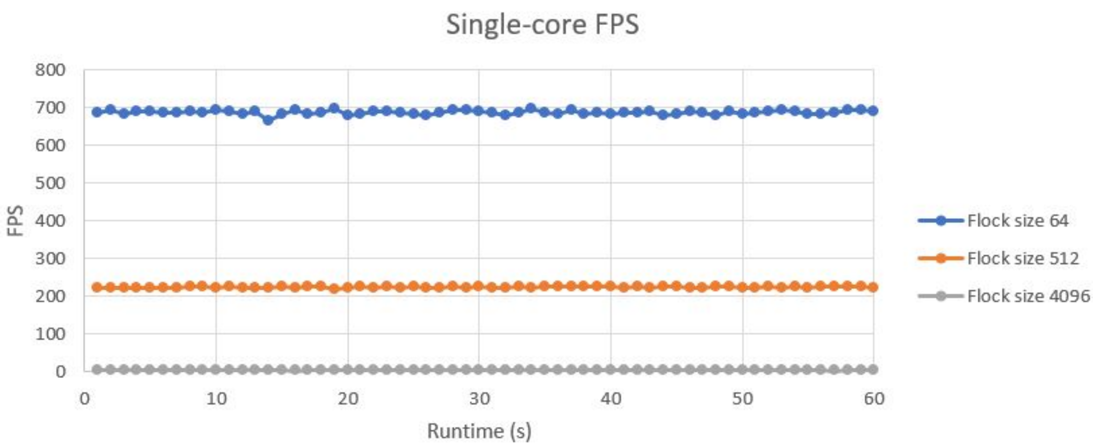

Figure 4.1: *Average FPS of all data points for the single-core implementation.*

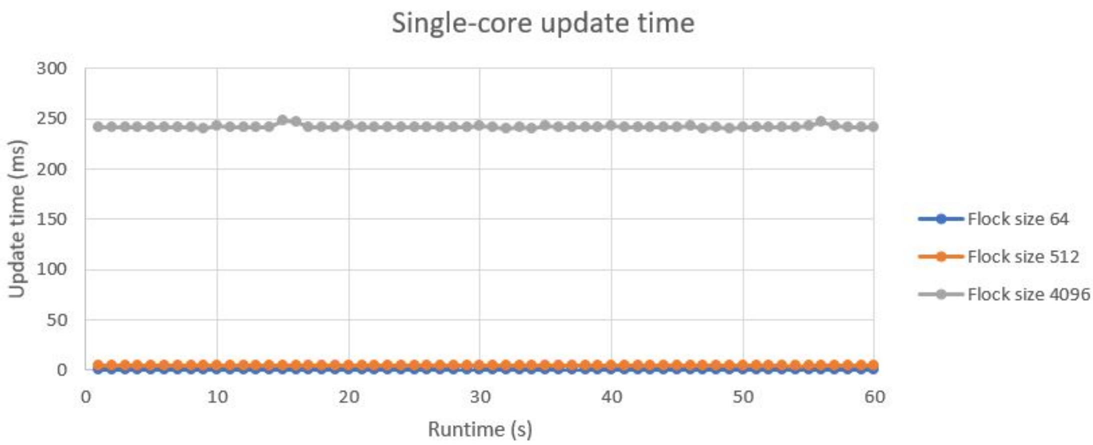

Figure 4.2: *Average update time per frame on all data points for the single-core implementation.*

4.2 Multi-core CPU

FPS in the multi-core implementation is similar for flock size 64 and 512. The same can be seen in the update time. The FPS and update time stays stable throughout the simulation for all flock sizes, though the flock sizes 64 and 512 present a somewhat irregular pattern.

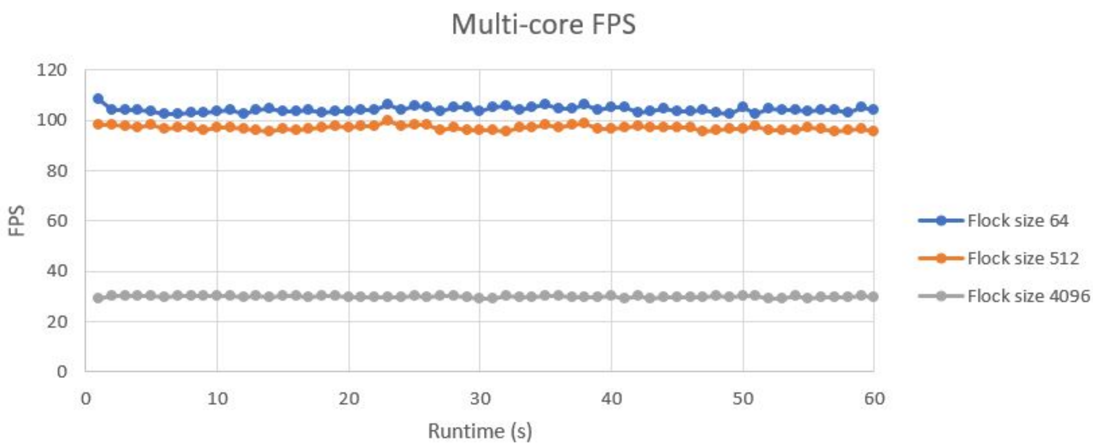

Figure 4.3: *Average FPS of all data points for the multi-core implementation.*

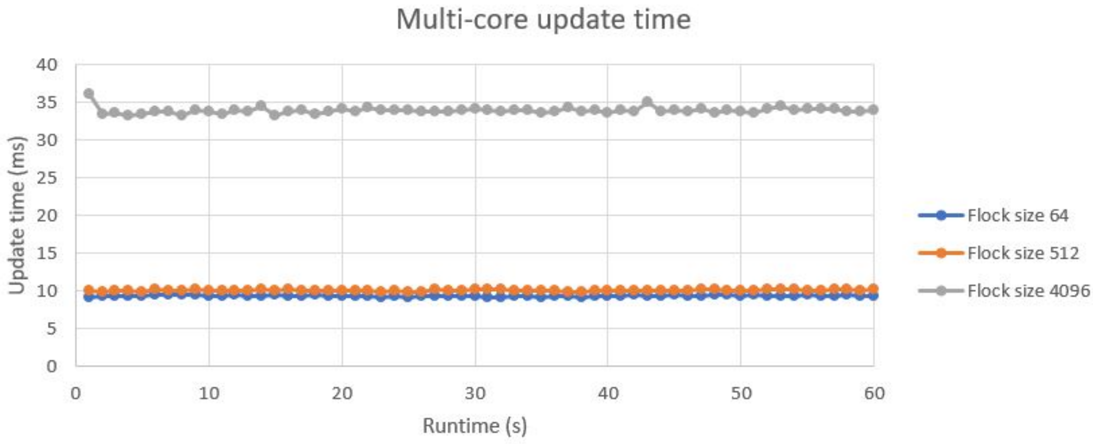

Figure 4.4: *Average FPS of all data points for the multi-core implementation.*

4.3 GPGPU

The FPS and update time for the GPGPU implementation are both similar with 64 and 512 boids if compared to the single-core measurements. In the early stages of the simulation the update time is increased at the same time as the FPS is decreased. The most unstable FPS is for for flock size 64 which can be seen in the early stages of the simulation where the update time is increased at the same time as the FPS is decreased. The other flock sizes are more stable throughout the simulation.

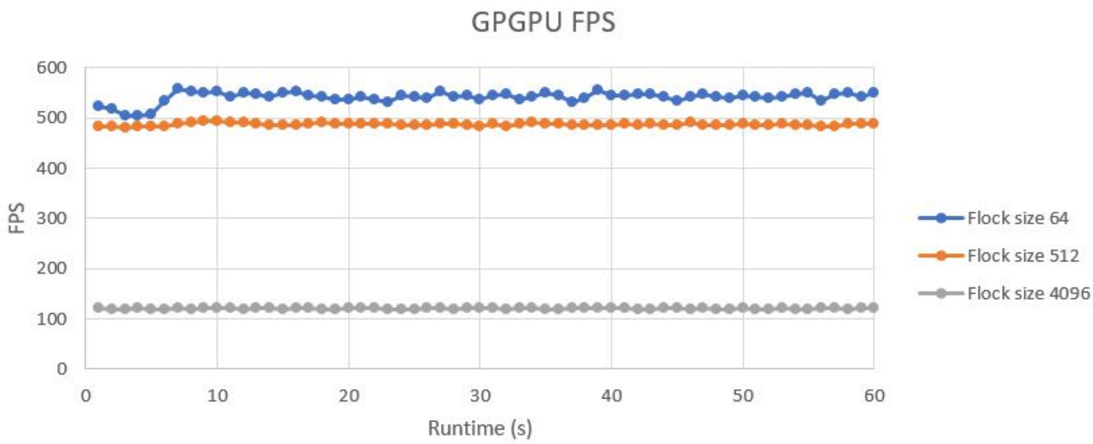

Figure 4.5: *Average update time per frame on all data points for the GPGPU implementation.*

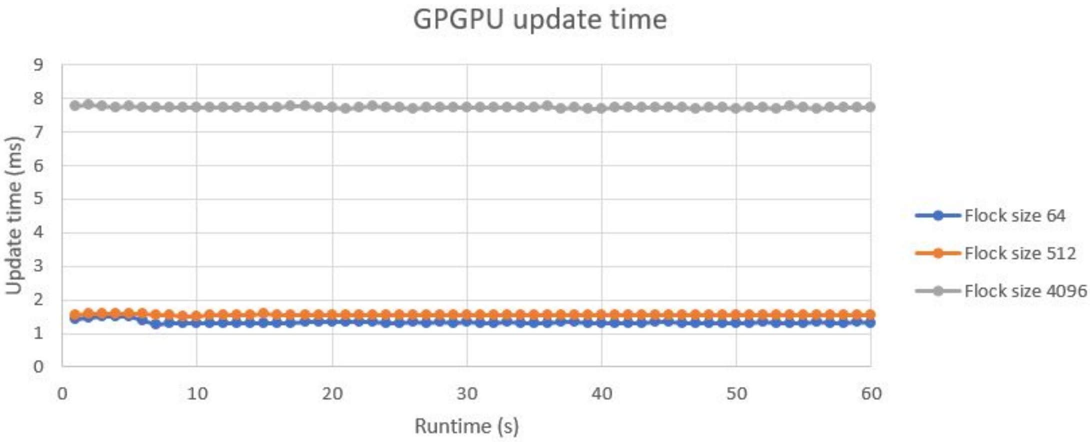

Figure 4.6: *Average update time per frame on all data points for the GPGPU implementation.*

4.4 Summary

The graph below summarizes the average FPS for the full simulation run for each implementation and flock size. Average update time per frame is not summarized in a graph since the difference of the highest and lowest value makes it difficult to illustrate. Average FPS and update time values for each flock size are compiled in appendix A. The tables below assumes the single-core CPU implementation as a benchmark measurement.

In table 4.1 and 4.2 the multi-core implementation outperforms the single-core implementation for flock size 4096. Additionally, the GPGPU implementation outperforms the single-core implementation for flock sizes 512 and 4096. However, for flock size 64 the single-core implementation has the best FPS and average update time.

Flock size	Single-core	Multi-core	GPGPU
64	1.0	0.151	0.787
512	1.0	0.431	2.164
4096	1.0	6.0	42.40

Table 4.1: *Average FPS of the multi-core and GPGPU implementations compared to the single-core implementation.*

Flock size	**Single-core**	**Multi-core**	**GPGPU**
64	1.0	11.161	1.580
512	1.0	2.398	0.368
4096	1.0	0.140	0.032

Table 4.2: *Average logic update time per frame of the multi-core and GPGPU implementations compared to the single-core implementation.*

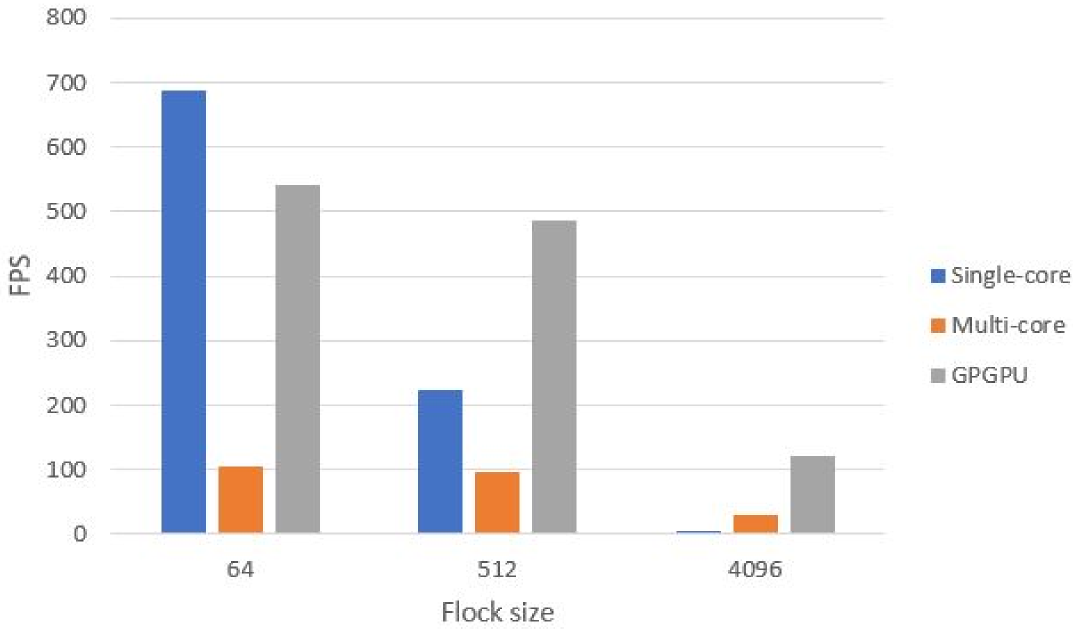

Figure 4.7: *Average FPS of the implementations for each flock size.*

Chapter 5

Analysis and Discussion

In this section the multi-core and GPGPU implementations are discussed and analyzed based on their performance in the tests.

5.1 Multi-core

As mentioned in the hypothesis in section 1.1, the multi-core implementation was expected to have longer boid update time for the lower computation flock sizes due to thread overhead, this has been illustrated in figure 4.7. Additionally, FPS and update time does not vary as much as the single-core implementation when flock size is increased from 64 to 512. The thread overhead is most likely the cause of the bottleneck in contrast to the computation of the boid logic for the single-core implementation. For flock size 4096 figure 4.7 illustrates the multi-core implementation outperforming the single-core implementation.

Tables 4.1 and 4.2 illustrates the rate of which the multi-core implementation increases in performance compared to the single-core implementation as flock size increases. Even though the rate is lower than the GPGPU implementation, the multi-core implementation inherits a speed up of six times over the single-core implementation in terms of FPS at flock size 4096.

5.2 GPGPU

All three implementations yielded a stable performance throughout the tests with the GPGPU being the most volatile for flock size 64 as illustrated in figure 4.5. However, the fluctuations were not by a degree that affects the test in any major aspect.

As initially discussed in the hypothesis, the GPGPU implementation was expected to have the highest total thread overhead and thus would perform worst for the lowest flock size, as illustrated in figure 4.7, this was not the case. The GPGPU implementation outperformed the multi-core implementation at all flock sizes looking at both FPS and update time. At flock size 4096 it performed exceptionally well with a speed up of 42 times in terms of FPS. However, the single-core implementation did achieve the highest FPS and lowest update time per frame for flock size 64 as expected in the hypothesis.

Tables 4.1 and 4.2 illustrates the rate of which the GPGPU implementation excel in performance compared to the single-core implementation as flock size increases. The GPGPU implementation shows a higher rate over the multi-core implementation.

Chapter 6

Conclusions and Future Work

In this chapter the conclusions from the experiment are stated and the future work is discussed.

6.1 Conclusions

It is observed that the GPGPU implementation outperformed the multi-core implementation for all flock sizes in terms of FPS and average boid logic update time per frame. Looking at the performance trend from the flock sizes there is nothing to indicate that the GPGPU implementation's performance advantage would change for lower nor higher flock sizes.

From the results in this thesis it can be concluded that when implementing a parallel basic boid algorithm to simulate large flock sizes it should be implemented on the GPU rather than the CPU when considering performance.

6.2 Future Work

This paper focuses on a basic boid implementation with few additions. There are many different variations and optimizations of the boid algorithm and it would be interesting to see if the same conclusions can be drawn for alternative implementations. Additionally, rendering for the implementations in this thesis does not put much load on the GPU. A possible future work would be to to investigate how the GPGPU implementation performs when the GPU is under heavier graphical computation load.

References

[1] Emmanuel Hermellin and Fabien Michel. GPU delegation: Toward a generic approach for developping MABS using GPU programming. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 1249–1258. International Foundation for Autonomous Agents and Multiagent Systems.

[2] Emmanuel Hermellin and Fabien Michel. GPU environmental delegation of agent perceptions: Application to reynolds's boids. In Benoit Gaudou and Jaime Simão Sichman, editors, *Multi-Agent Based Simulation XVI*, volume 9568, pages 71–86. Springer International Publishing. DOI: 10.1007/978-3-319-31447-1_5.

[3] M. Joselli, E. B. Passos, M. Zamith, E. Clua, A. Montenegro, and B. Feijó. A neighborhood grid data structure for massive 3d crowd simulation on GPU. In *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, pages 121–131.

[4] Yushim Kim and Callie McGraw. Use of agent-based modeling for e-governance research. In *Proceedings of the 6th International Conference on Theory and Practice of Electronic Governance*, ICEGOV '12, pages 531–534. ACM.

[5] Jae Moon Lee, Se Hong Cho, and Rafael A. Calvo. A fast algorithm for simulation of flocking behavior. pages 186–190. IEEE, August 2009.

[6] Kalyan S. Perumalla and Brandon G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 116–123. Society for Computer Simulation International.

[7] Craig W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[8] Alessandro Ribeiro Da Silva, Wallace Santos Lages, and Luiz Chaimowicz. Boids that see: Using self-occlusion for simulating large groups on GPUs. 7(4):51:1–51:20.

Appendix A

Metrics

Time is listed in milliseconds.

Flock size	Single-core	Multi-core	GPGPU
64	688	104	541
512	225	97	487
4096	5	30	121

Table A.1: *Average FPS of the single-core, multi-core and GPGPU implementations.*

Flock size	Single-core	Multi-core	GPGPU
64	0.84141	9.39100	1.32975
512	4.21427	10.10588	1.5501
4096	241.83176	33.93527	7.7305

Table A.2: *Average logic update time per frame of the single-core, multi-core and GPGPU implementations.*

Single-core					
64		**512**		**4096**	
FPS	**Update time**	**FPS**	**Update time**	**FPS**	**Update time**
687	0.842934	224	4.24628	6	241.30367
692	0.841761667	224	4.2269	5	241.48700
682	0.844263667	224	4.23355	5	241.04467
692	0.838079667	224	4.22826	5	241.70367
691	0.838935333	224	4.239266667	5	241.50867
688	0.850306	224	4.237123333	5	240.88567
687	0.861056333	223	4.243783333	5	241.28300
689	0.842557333	225	4.209426667	5	241.29200
686	0.84268	225	4.207333333	5	240.78333
694	0.839623	224	4.225146667	5	242.46333
690	0.841077333	225	4.200526667	5	241.06900
682	0.849865	225	4.204323333	5	241.22533
692	0.853269	224	4.22321	5	240.96467
666	0.865152	225	4.215456667	5	240.90233
683	0.841990333	225	4.205563333	5	248.84733
695	0.844358667	225	4.216606667	5	247.27400
684	0.864163667	225	4.20412	5	241.23267
686	0.842680667	225	4.2079	5	241.14333
697	0.832317667	219	4.335726667	5	241.41867
681	0.840183	222	4.278833333	5	242.98167
684	0.846042	226	4.192766667	5	240.88800
690	0.845846333	224	4.243716667	5	241.88867
692	0.842804667	226	4.19364	5	240.90767
685	0.851129667	225	4.211416667	5	241.22333
684	0.841764	225	4.225133333	5	242.19000
679	0.850309	224	4.22534	5	241.98733
688	0.839654333	225	4.206553333	5	241.18067
694	0.830831667	226	4.197906667	5	241.43167
694	0.824911667	224	4.21775	5	240.98933
689	0.835168667	226	4.19029	5	242.31733
689	0.851253667	224	4.21479	5	242.11433
681	0.842121667	225	4.213293333	5	240.76067
689	0.834648	225	4.19644	5	242.13333
697	0.828838	225	4.209403333	5	240.51100
688	0.832210333	225	4.202176667	5	242.52967
682	0.837025333	225	4.203113333	5	241.80533
696	0.824939667	225	4.210176667	5	241.45433
683	0.827711667	225	4.205633333	5	241.22500
687	0.838273	225	4.200346667	5	241.75700

684	0.855442667	225	4.204943333	5	242.75333
687	0.841007	224	4.2196	5	241.64300
685	0.83641	226	4.199343333	5	241.18367
689	0.847558667	225	4.208233333	5	241.32767
680	0.842190333	226	4.201566667	5	241.97300
685	0.840686	226	4.187566667	5	241.99900
691	0.839681	224	4.220563333	5	242.38367
688	0.854046667	224	4.21329	5	240.80067
681	0.854764	225	4.210953333	5	240.94067
689	0.841733667	225	4.218793333	5	240.65367
683	0.835376333	224	4.218733333	5	241.93900
686	0.826323	224	4.213076667	5	241.88033
692	0.828525667	226	4.179153333	5	241.40600
693	0.838429333	224	4.215866667	5	241.32567
691	0.842371333	227	4.175306667	5	241.15433
685	0.848685667	224	4.206806667	5	243.00933
682	0.833780333	225	4.205713333	5	246.35800
687	0.830802667	226	4.19085	5	242.99767
694	0.830349	226	4.194993333	5	241.04200
695	0.838469	226	4.1929	5	241.00067
691	0.835347667	224	4.228993333	5	242.02567

Table A.3: *All data point averages for the single-core implementation*

Multi-core					
64		**512**		**4096**	
FPS	**Update time**	**FPS**	**Update time**	**FPS**	**Update time**
108	9.1004	98	10.03522333	29	36.15423333
104	9.346226667	98	9.9471	30	33.46286667
104	9.401356667	98	10.05474667	30	33.65126667
104	9.371196667	97	10.10533333	30	33.251
104	9.441483333	98	9.977646667	30	33.5249
103	9.529046667	97	10.18253333	30	33.88403333
103	9.52257	97	10.12916667	30	33.8055
103	9.524723333	97	10.09343333	30	33.3421
103	9.511486667	96	10.22531667	30	33.9541
104	9.428443333	97	10.0965	30	33.888
104	9.409346667	97	10.05856333	30	33.48183333
103	9.507306667	97	10.13376667	30	33.91273333
104	9.40236	96	10.1663	30	33.72726667
105	9.366473333	96	10.2315	30	34.44666667

103	9.478366667	97	10.15693333	30	33.30556667
103	9.426016667	96	10.18296667	30	33.7417
104	9.413823333	97	10.1483	30	33.97983333
103	9.502726667	97	10.07012	30	33.51123333
104	9.433086667	98	10.03469	30	33.78293333
104	9.44074	97	10.07743333	30	34.2239
104	9.433703333	98	10.05966	30	33.74853333
104	9.352096667	98	10.02836333	30	34.28466667
106	9.19756	100	9.835256667	30	33.95933333
104	9.43993	98	10.03188667	30	33.9734
106	9.218763333	98	9.983183333	30	33.95266667
105	9.282523333	98	9.969546667	30	33.8383
104	9.39121	96	10.1836	30	33.85686667
105	9.331683333	97	10.12561	30	33.8023
105	9.278596667	96	10.12047667	30	34.00713333
104	9.400393333	96	10.1862	29	34.15776667
105	9.2668	96	10.19806667	29	34.06713333
106	9.274666667	96	10.27923333	30	33.85473333
104	9.351376667	97	10.0915	30	34.0424
105	9.321796667	97	10.07573333	30	33.96786667
106	9.23631	98	9.998683333	30	33.69323333
105	9.341806667	97	10.05150333	30	33.82513333
105	9.37291	98	9.980763333	30	34.26126667
106	9.210756667	99	9.95962	30	33.7714
104	9.422436667	97	10.11466	30	33.90683333
105	9.332223333	97	10.10963333	30	33.7173
105	9.37791	97	10.05838	29	34.06903333
103	9.476233333	98	10.04162333	30	33.7601
104	9.429143333	97	10.10574333	29	35.01673333
105	9.34337	97	10.08326667	30	33.7492
104	9.46918	97	10.06612	30	34.07156667
104	9.415356667	97	10.05185333	30	33.84933333
104	9.421773333	96	10.24413333	30	34.143
103	9.45799	96	10.2021	30	33.7102
102	9.52271	97	10.1375	30	33.9194
105	9.32122	97	10.11356667	30	33.87826667
103	9.580946667	98	10.04731667	30	33.55573333
105	9.33308	96	10.1989	29	34.08676667
104	9.376686667	96	10.2394	29	34.50936667
104	9.43184	96	10.19856667	30	33.99743333
103	9.476023333	97	10.1054	29	34.08866667
104	9.393996667	97	10.13263333	30	34.14236667
104	9.36877	95	10.26476667	30	34.20323333

103	9.489323333	96	10.17866667	30	33.80816667
105	9.319426667	97	10.11633333	30	33.84596667
104	9.44006	96	10.27555	30	33.99163333

Table A.4: *All data point averages for the multi-core implementation*

GPGPU					
64		**512**		**4096**	
FPS	**Update time**	**FPS**	**Update time**	**FPS**	**Update time**
524	1.42443	483	1.5558	121	7.754283333
517	1.458533333	484	1.582836667	120	7.792076667
506	1.486616667	480	1.580506667	120	7.757483333
503	1.495036667	483	1.587493333	121	7.715336667
508	1.490143333	483	1.58745	120	7.74857
533	1.37188	482	1.57668	120	7.7425
558	1.281296667	489	1.556296667	121	7.716606667
553	1.296526667	491	1.542556667	120	7.71912
550	1.31087	495	1.518456667	121	7.733743333
552	1.292166667	495	1.52331	121	7.709383333
542	1.322786667	490	1.542906667	121	7.719043333
549	1.30417	490	1.53168	120	7.738173333
548	1.290503333	489	1.54362	121	7.72433
541	1.30584	486	1.557386667	121	7.73458
549	1.30523	485	1.566953333	120	7.734926667
552	1.304746667	485	1.561946667	121	7.721436667
543	1.299606667	489	1.549336667	121	7.751053333
542	1.326443333	490	1.541203333	120	7.766193333
538	1.336043333	488	1.546983333	120	7.74221
538	1.333256667	488	1.541966667	121	7.717073333
542	1.335503333	489	1.54801	121	7.70593
536	1.338363333	489	1.549313333	121	7.716736667
532	1.344626667	487	1.545483333	120	7.755203333
544	1.31445	485	1.548566667	120	7.74226
542	1.31736	485	1.557456667	120	7.743453333
538	1.33479	486	1.555346667	121	7.707653333
552	1.299636667	489	1.543273333	121	7.719696667
541	1.324886667	488	1.54936	120	7.732483333
544	1.316596667	487	1.549573333	121	7.743263333
536	1.326313333	484	1.55249	121	7.729466667
543	1.319916667	487	1.545526667	121	7.73451
548	1.307273333	483	1.53179	120	7.73542

538	1.334236667
541	1.31635
551	1.296926667
544	1.304323333
532	1.346326667
539	1.33926
555	1.29484
546	1.308126667
544	1.301043333
547	1.29178
548	1.305246667
542	1.329726667
534	1.35341
541	1.31854
548	1.309766667
542	1.31348
540	1.321676667
545	1.310166667
543	1.322203333
538	1.332463333
541	1.323763333
546	1.30668
549	1.302663333
534	1.341906667
548	1.314206667
551	1.303993333
542	1.331513333
550	1.294576667
488	1.54751
490	1.54133
489	1.54816
487	1.547976667
486	1.55114
485	1.55151
485	1.54441
486	1.548246667
488	1.543873333
486	1.553143333
487	1.54935
486	1.555906667
486	1.555426667
490	1.547346667
486	1.54849
486	1.546966667
485	1.546976667
488	1.551806667
486	1.54711
486	1.54754
489	1.54639
487	1.54821
486	1.552456667
484	1.550596667
484	1.535046667
489	1.545583333
488	1.543143333
488	1.539496667
121	7.726766667
121	7.72923
120	7.747956667
120	7.7661
121	7.705616667
121	7.730343333
121	7.707133333
121	7.702743333
121	7.7327
120	7.74474
120	7.737866667
121	7.71679
121	7.71718
120	7.74183
121	7.705726667
120	7.73846
120	7.732523333
121	7.69445
120	7.72985
120	7.731583333
121	7.7055
120	7.759646667
120	7.742103333
121	7.69872
121	7.710096667
120	7.72606
121	7.714943333
121	7.728496667

Table A.5: *All data point averages for the GPGPU implementation*

Appendix B

Code

Below are the main update functions for the three implementations along with the relevant logic functions.

B.1 CPU helper functions

```
glm::vec3 BoidLogicHandler::CenterRule(Boid* allBoids, int currentBoidIndex) {
    glm::vec3 center = glm::vec3(0.0, 0.0, 0.0);

    for (int i = 0; i < NR_OF_BOIDS; i++) {
        center += allBoids[i].GetPosition();
    }
    center   = allBoids[currentBoidIndex].GetPosition();
    center = center / (float)(NR_OF_BOIDS   1);

    return center * CENTER_FACTOR;
}

glm::vec3 BoidLogicHandler::AvoidRule(Boid* allBoids, int currentBoidIndex) {
    glm::vec3 avoid = glm::vec3(0.0, 0.0, 0.0);
    glm::vec3 currentBoidPos = allBoids[currentBoidIndex].GetPosition();
    glm::vec3 vecToBoid = glm::vec3(0.0, 0.0, 0.0);

    for (int i = 0; i < NR_OF_BOIDS; i++) {
        if (i != currentBoidIndex) {
            vecToBoid = allBoids[i].GetPosition()    currentBoidPos;
            if (glm::length(vecToBoid) < MIN_SEPERATION_DISTANCE) {
                avoid   = vecToBoid;
            }
        }

    }

    return avoid * AVOID_FACTOR;
}
```

```
glm::vec3 BoidLogicHandler::VelocityRule(Boid* allBoids, int currentBoidIndex) {
    glm::vec3 velocity = glm::vec3(0.0, 0.0, 0.0);

    for (int i = 0; i < NR_OF_BOIDS; i++) {
        velocity += allBoids[i].GetVelocity();
    }
    velocity  = allBoids[currentBoidIndex].GetVelocity();
    velocity = velocity / (float)(NR_OF_BOIDS   1);

    return velocity * MATCH_FACTOR;
}

glm::vec3 BoidLogicHandler::LimitSpeed(glm::vec3 oldVelocity, glm::vec3 newVelocity, float deltaTime) {
    glm::vec3 limitedVelocity = newVelocity;
    float newSpeed = glm::length(newVelocity);
    float oldSpeed = glm::length(oldVelocity);

    if (newSpeed > MAX_SPEED || newSpeed < MIN_SPEED) {
        limitedVelocity = oldVelocity;
    }
    else {
        if (newSpeed > oldSpeed) {
            limitedVelocity = glm::normalize(limitedVelocity) * (oldSpeed + (MAX_ACCELERATION * deltaTime));
        }
        else {
            limitedVelocity = glm::normalize(limitedVelocity) * (oldSpeed   (MAX_ACCELERATION * deltaTime));
        }
    }

    return limitedVelocity;
}

glm::vec3 BoidLogicHandler::CalculateNewPos(glm::vec3 oldPosition, glm::vec3 newVelocity, float deltaTime) {
    glm::vec3 newPos = oldPosition + (newVelocity * deltaTime * BOID_SPEED);

    return newPos;
}

glm::vec3 BoidLogicHandler::MoveIfOutOfBounds(glm::vec3 position) {
    glm::vec3 newPosition = position;

    float sideLength = GRID_SIDE_LENGTH;

    float xMax = 0.0f + (sideLength / (float)2);
```

```
    float xMin = 0.0f   (sideLength / (float)2);
    float yMax = 0.0f + (sideLength / (float)2);
    float yMin = 0.0f   (sideLength / (float)2);
    float zMax = 0.0f + (sideLength / (float)2);
    float zMin = 0.0f   (sideLength / (float)2);

    //X
    if (position.x > xMax) {
        newPosition.x = xMin;
    }
    if (position.x < xMin) {
        newPosition.x = xMax;
    }
    //Y
    if (position.y > yMax) {
        newPosition.y = yMin;
    }
    if (position.y < yMin) {
        newPosition.y = yMax;
    }
    //Z
    if (position.z > zMax) {
        newPosition.z = zMin;
    }
    if (position.z < zMin) {
        newPosition.z = zMax;
    }

    return newPosition;
}

void BoidLogicHandler::BoidThread(Scene* scene, int startIndex, int
  endIndex, float deltaTime) {
    Boid* allBoidsPrevious = scene >GetAllBoidsPrevious();
    Boid* allBoids = scene >GetAllBoids();
    glm::vec3 newVelocity = glm::vec3(0.0, 0.0, 0.0);
    glm::vec3 previousVelocity = glm::vec3(0.0, 0.0, 0.0);

    for (int i = startIndex; i < endIndex; i++) {
        previousVelocity = allBoidsPrevious[i].GetVelocity();
        newVelocity = previousVelocity;

        //1. Fly towards center
        glm::vec3 centerRuleVec = CenterRule(allBoidsPrevious, i);

        //2. Avoid boids
        glm::vec3 avoidRuleVec = AvoidRule(allBoidsPrevious, i);

        //3. Match velocity/direction with all boids
        glm::vec3 velocityRuleVec = VelocityRule(allBoidsPrevious, i
  );
```

```

        //Add all rules
        newVelocity += centerRuleVec + avoidRuleVec + velocityRuleVec;

        //Limit speed
        newVelocity = LimitSpeed(previousVelocity, newVelocity, deltaTime);

        //Set new boid velocity and up direction
        allBoids[i].SetVelocityAndUp(newVelocity);

        //Calculate new boid position
        glm::vec3 oldPosition = allBoidsPrevious[i].GetPosition();
        glm::vec3 newPosition = CalculateNewPos(oldPosition, newVelocity, deltaTime);

        //Move if out of bounds
        newPosition = MoveIfOutOfBounds(newPosition);

        //Set boid new position
        allBoids[i].SetPosition(newPosition);
    }
}
```

B.2 Single-core CPU update function

```
void BoidLogicHandler::SingleThreadUpdate(Scene* scene, float deltaTime) {
    scene >SwitchCurrentAndPreviousBoids();
    Boid* allBoidsPrevious = scene >GetAllBoidsPrevious();
    Boid* allBoids = scene >GetAllBoids();
    glm::vec3 newVelocity = glm::vec3(0.0, 0.0, 0.0);
    glm::vec3 previousVelocity = glm::vec3(0.0, 0.0, 0.0);

    for (int i = 0; i < NR_OF_BOIDS; i++) {
        previousVelocity = allBoidsPrevious[i].GetVelocity();
        newVelocity = previousVelocity;

        //1. Fly towards center
        glm::vec3 centerRuleVec = CenterRule(allBoidsPrevious, i);

        //2. Avoid boids
        glm::vec3 avoidRuleVec = AvoidRule(allBoidsPrevious, i);

        //3. Match velocity/direction with all boids
```

```
        glm::vec3 velocityRuleVec = VelocityRule(allBoidsPrevious, i);

        //Add all rules
        newVelocity += centerRuleVec + avoidRuleVec + velocityRuleVec;

        //Limit speed
        newVelocity = LimitSpeed(previousVelocity, newVelocity, deltaTime);

        //Set new boid velocity and up direction
        allBoids[i].SetVelocityAndUp(newVelocity);

        //Calculate new boid position
        glm::vec3 oldPosition = allBoidsPrevious[i].GetPosition();
        glm::vec3 newPosition = CalculateNewPos(oldPosition, newVelocity, deltaTime);

        //Move if out of bounds
        newPosition = MoveIfOutOfBounds(newPosition);

        //Set boid new position
        allBoids[i].SetPosition(newPosition);
    }

    scene >GetBoidBuffer(0) >SetData(scene >GetAllBoids(), sizeof(Boid) * NR_OF_BOIDS);
}
```

B.3 Multi-core CPU update function

```
void BoidLogicHandler::MultiThreadUpdate(Scene* scene, float deltaTime) {
    scene >SwitchCurrentAndPreviousBoids();
    const int THREADS = 8;
    std::thread threadPool[THREADS];

    int startIndex = 0;
    int endIndex = 0;

    for (int i = 0; i < THREADS; i++) {
        startIndex = i * (NR_OF_BOIDS / THREADS);
        endIndex = (i * (NR_OF_BOIDS / THREADS)) + (NR_OF_BOIDS / THREADS);
        threadPool[i] = std::thread(BoidThread, scene, startIndex, endIndex, deltaTime);
```

```
    }

    for (auto& th : threadPool) {
        th.join();
    }


    scene >GetBoidBuffer(0) >SetData(scene >GetAllBoids(), sizeof(Boid) * NR_OF_BOIDS);
}
```

B.4 Multi-core CPU thread function

```
void BoidLogicHandler::BoidThread(Scene* scene, int startIndex, int endIndex, float deltaTime) {
    Boid* allBoidsPrevious = scene >GetAllBoidsPrevious();
    Boid* allBoids = scene >GetAllBoids();
    glm::vec3 newVelocity = glm::vec3(0.0, 0.0, 0.0);
    glm::vec3 previousVelocity = glm::vec3(0.0, 0.0, 0.0);

    for (int i = startIndex; i < endIndex; i++) {
        previousVelocity = allBoidsPrevious[i].GetVelocity();
        newVelocity = previousVelocity;

        //1. Fly towards center
        glm::vec3 centerRuleVec = CenterRule(allBoidsPrevious, i);

        //2. Avoid boids
        glm::vec3 avoidRuleVec = AvoidRule(allBoidsPrevious, i);

        //3. Match velocity/direction with all boids
        glm::vec3 velocityRuleVec = VelocityRule(allBoidsPrevious, i);

        //Add all rules
        newVelocity += centerRuleVec + avoidRuleVec + velocityRuleVec;

        //Limit speed
        newVelocity = LimitSpeed(previousVelocity, newVelocity, deltaTime);

        //Set new boid velocity and up direction
        allBoids[i].SetVelocityAndUp(newVelocity);

        //Calculate new boid position
        glm::vec3 oldPosition = allBoidsPrevious[i].GetPosition();
```

```
        glm::vec3 newPosition = CalculateNewPos(oldPosition, newVelocity, deltaTime);

        //Move if out of bounds
        newPosition = MoveIfOutOfBounds(newPosition);

        //Set boid new position
        allBoids[i].SetPosition(newPosition);
    }
}
```

B.5 GPGPU update function

```
void BoidLogicHandler::GPUUpdate(Scene* scene, float deltaTime) {
    ID3D11DeviceContext* dxContext = this >rendererPtr >GetDxDeviceContext();

    //Set computeshader
    dxContext >CSSetShader(this >computeShader,
        nullptr,
        0);

    //Set delta time
    this >deltaTimeBuffer >SetData(&deltaTime, sizeof(float));

    //Dispatch shader
    ID3D11ShaderResourceView* srvArray[] = { scene >GetBoidBuffer(boidBufferSwitchIndex) >GetShaderResourceView(),
                                            this >deltaTimeBuffer >GetShaderResourceView() ,
                                            this >constantsBuffer >GetShaderResourceView() };
    ID3D11UnorderedAccessView* uavArray[] = { scene >GetBoidBuffer((boidBufferSwitchIndex + 1) % 2) >GetUnorderedAccessView() };
    dxContext >CSSetShaderResources(0, 3, srvArray);
    dxContext >CSSetUnorderedAccessViews(0, 1, uavArray, 0);
    dxContext >Dispatch(NR_OF_BOIDS/64, 1, 1);

    //Null resources
    ID3D11ShaderResourceView* srvNullArray[] = { nullptr };
    ID3D11UnorderedAccessView* uavNullArray[] = { nullptr };
    dxContext >CSSetShaderResources(0, 1, srvNullArray);
    dxContext >CSSetUnorderedAccessViews(0, 1, uavNullArray, 0);

    //Unset computeshader
        dxContext >CSSetShader(nullptr,
        nullptr,
```

```
        0);

    //Switch buffers for next frame
    boidBufferSwitchIndex = 1   boidBufferSwitchIndex;
}
```

B.6 Compute shader

```
struct Boid {
    float3 position: POSITION;
    float3 velocity: VELOCITY;
    float3 up: UP;
};

struct Constants
{
    float MIN_SEPERATION_DISTANCE;
    uint COHESION_THRESHHOLD;
    float BOID_SPEED;
    float MAX_SPEED;
    float MIN_SPEED;
    float MAX_ACCELERATION;

    float CENTER_FACTOR;
    float AVOID_FACTOR;
    float MATCH_FACTOR;

    uint NR_OF_BOIDS;
    float BOID_SEPERATION;

    float GRID_SIDE_LENGTH;
};

StructuredBuffer<Boid> readBufferBoids : register(t0);
RWStructuredBuffer<Boid> writeBufferBoids : register(u0);

StructuredBuffer<float> readBufferDeltaTime : register(t1);

StructuredBuffer<Constants> readBufferConstants : register(t2);


float3 CenterRule(int currentBoidIndex) {
    float3 center = 0.0f;

    for (int i = 0; i < readBufferConstants[0].NR_OF_BOIDS; i++) {
        center += readBufferBoids[i].position;
    }
```

```
    center   = readBufferBoids[currentBoidIndex].position;
    center = center / (float) (readBufferConstants[0].NR_OF_BOIDS
   1);

    return center * readBufferConstants[0].CENTER_FACTOR;
}

float3 AvoidRule(int currentBoidIndex) {
    float3 avoid = 0.0f;
    float3 currentBoidPos = readBufferBoids[currentBoidIndex].
   position;
    float3 vecToBoid = 0.0f;

    for (int i = 0; i < readBufferConstants[0].NR_OF_BOIDS; i++) {
        if (i != currentBoidIndex) {
            vecToBoid = readBufferBoids[i].position
   readBufferBoids[currentBoidIndex].position;
            if (length(vecToBoid) < readBufferConstants[0].
   MIN_SEPERATION_DISTANCE)
            {
                avoid   = vecToBoid;
            }
        }
    }

    return avoid * readBufferConstants[0].AVOID_FACTOR;
}

float3 VelocityRule(int currentBoidIndex) {
    float3 velocity = 0.0f;

    for (int i = 0; i < readBufferConstants[0].NR_OF_BOIDS; i++) {
        velocity += readBufferBoids[i].velocity;
    }
    velocity   = readBufferBoids[currentBoidIndex].velocity;
    velocity = velocity / (float) (readBufferConstants[0].
   NR_OF_BOIDS   1);

    return velocity * readBufferConstants[0].MATCH_FACTOR;
}

float3 LimitSpeed(float3 oldVelocity, float3 newVelocity, float
   deltaTime) {
    float3 limitedVelocity = newVelocity;
    float newSpeed = length(newVelocity);
    float oldSpeed = length(oldVelocity);

    if (newSpeed > readBufferConstants[0].MAX_SPEED || newSpeed <
   readBufferConstants[0].MIN_SPEED)
    {
        limitedVelocity = oldVelocity;
```

```
    }
    else {
        if (newSpeed > oldSpeed) {
            limitedVelocity = normalize(limitedVelocity) * (oldSpeed + (readBufferConstants[0].MAX_ACCELERATION * deltaTime));
        }
        else {
            limitedVelocity = normalize(limitedVelocity) * (oldSpeed + (readBufferConstants[0].MAX_ACCELERATION * deltaTime));
        }
    }

    return limitedVelocity;
}

void SetBoidVelocityAndUp(uint index, float3 newVelocity) {
    float3 forward = normalize(newVelocity);
    float3 newRight = normalize(cross(float3(0.0f, 1.0f, 0.0f), forward));
    float3 newUp = cross(forward, newRight);
    writeBufferBoids[index].up = newUp;

    writeBufferBoids[index].velocity = newVelocity;
}

float3 CalculateNewPos(float3 oldPosition, float3 newVelocity, float deltaTime) {
    float3 newPos = oldPosition + (newVelocity * deltaTime * readBufferConstants[0].BOID_SPEED);

    return newPos;
}

float3 MoveIfOutOfBounds(float3 position) {
    float3 newPosition = position;

    float sideLength = readBufferConstants[0].GRID_SIDE_LENGTH;

    float xMax = 0.0f + (sideLength / (float) 2);
    float xMin = 0.0f   (sideLength / (float) 2);
    float yMax = 0.0f + (sideLength / (float) 2);
    float yMin = 0.0f   (sideLength / (float) 2);
    float zMax = 0.0f + (sideLength / (float) 2);
    float zMin = 0.0f   (sideLength / (float) 2);

    //X
    if (position.x > xMax) {
        newPosition.x = xMin;
    }
    if (position.x < xMin) {
        newPosition.x = xMax;
```

```
    }
    //Y
    if (position.y > yMax) {
        newPosition.y = yMin;
    }
    if (position.y < yMin) {
        newPosition.y = yMax;
    }
    //Z
    if (position.z > zMax) {
        newPosition.z = zMin;
    }
    if (position.z < zMin) {
        newPosition.z = zMax;
    }

    return newPosition;
}

[numthreads(64, 1, 1)]
void main( uint3 DTid : SV_DispatchThreadID ) {
    int i = DTid.x;
    float3 previousVelocty = readBufferBoids[i].velocity;
    float3 newVelocity = previousVelocty;

    //1. Fly towards center
    float3 centerRuleVec = CenterRule(i);

    //2. Avoid boids
    float3 avoidRuleVec = AvoidRule(i);

    //3. Match velocity/direction with all boids
    float3 velocityRuleVec = VelocityRule(i);

    //Add all rules
    newVelocity += centerRuleVec + avoidRuleVec + velocityRuleVec;

    //Limit speed
    float deltaTime = readBufferDeltaTime[0];
    newVelocity = LimitSpeed(previousVelocty, newVelocity, deltaTime
    );

    //Set new boid velocity and up direction
    SetBoidVelocityAndUp(i, newVelocity);

    //Calculate new boid position
    float3 oldPosition = readBufferBoids[i].position;
    float3 newPosition = CalculateNewPos(oldPosition, newVelocity,
    deltaTime);

    //Move if out of bounds
```

```
    newPosition = MoveIfOutOfBounds(newPosition);

    //Set boid new position
    writeBufferBoids[i].position = newPosition;
}
```