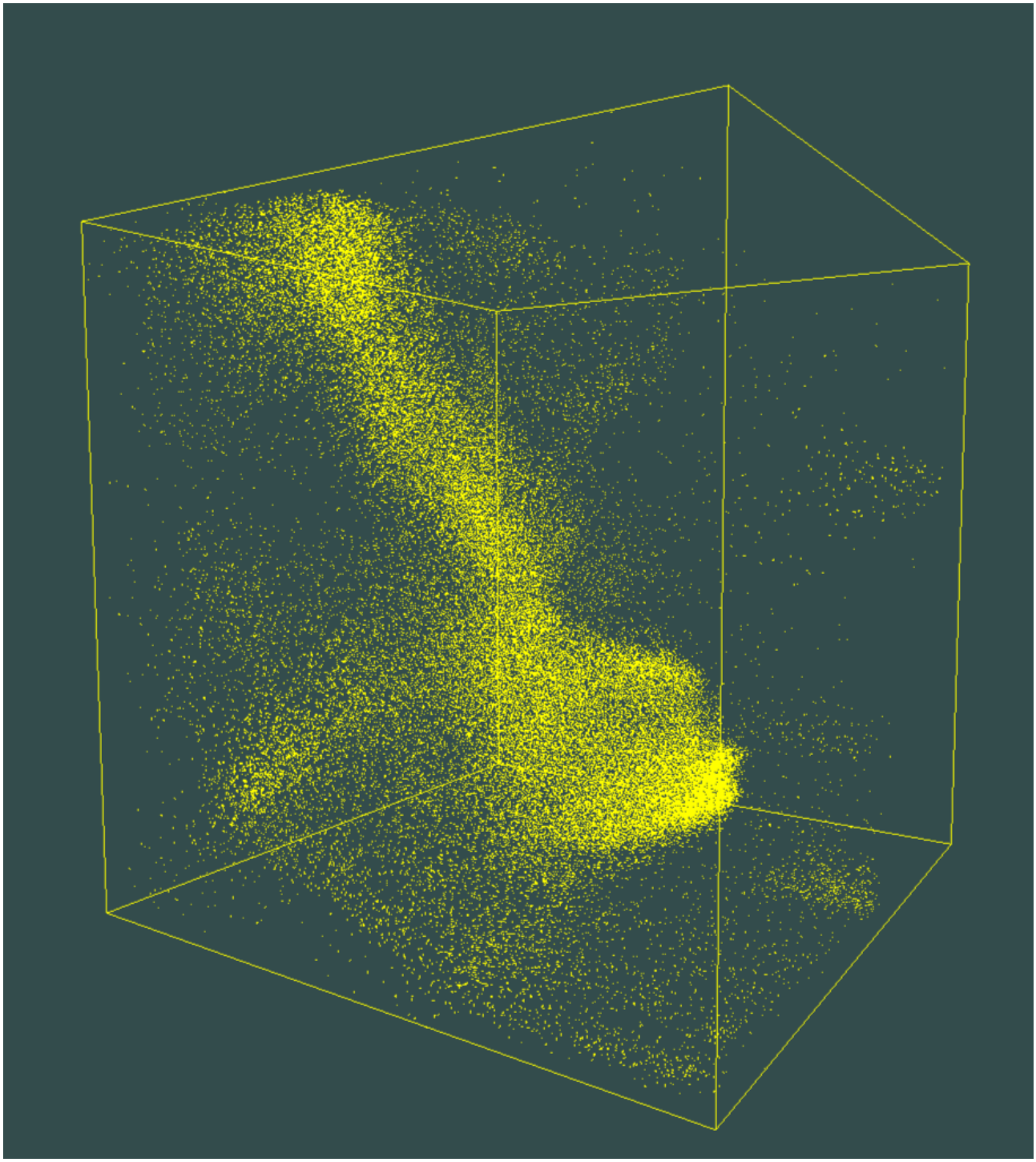
CUDA Boids

A simple boid flocking simulation accelerated by CUDA



Luca Di Gennaro - 961208

Table of Contents

Introduction and goal

The boids and the flocking algorithms, introduced by Craig Reynolds in 1986, are simulation techniques capable of reproducing a group behaviour similar to bird flocks.

Each element of the flock (called “boid”) moves aware of the surrounding flockmates following three rules:

- **Alignment**, steer towards the average heading of local flockmates
- **Cohesion**, steer to move towards the average position (centre of mass) of local flockmates
- **Separation**, steer to avoid crowding local flockmates

These simple rules build up to an emergent behaviour that looks exactly like a flock of birds.

The main **problem** of this algorithm is **computational**: we need to check each boid against every other boid within a certain neighbourhood distance, which in the worst case reaches $O(n^2)$ time complexity.

The goal of this project is to create an interactive 3D boid flocking simulation, where computations are **accelerated by using parallel techniques** based on the CUDA framework and show its effectiveness compared to the sequential counterpart.

Algorithms

We are going to analyze **three different implementations** of the boid flocking algorithm, each implemented both on the CPU (sequentially) and on the GPU (concurrently).

The simplest implementation is the ***naïve*** one, which defaults to the worst-case scenario we mentioned earlier: each boid checks against every other boid, reaching $O(n^2)$ time complexity.

To address this issue, thus reducing the amount of boids checking each other, we can introduce the **uniform grid**, a spatial data structure that partitions the whole simulation volume into a grid of cells: we can use this to localize each boid and check only against those in the neighbouring cells, greatly reducing the computations amount.

Thus, the only information we need to save about each boid is:

- its position
- its velocity
- its grid index

Each of these will be stored in its own separate array, as we’ll see.

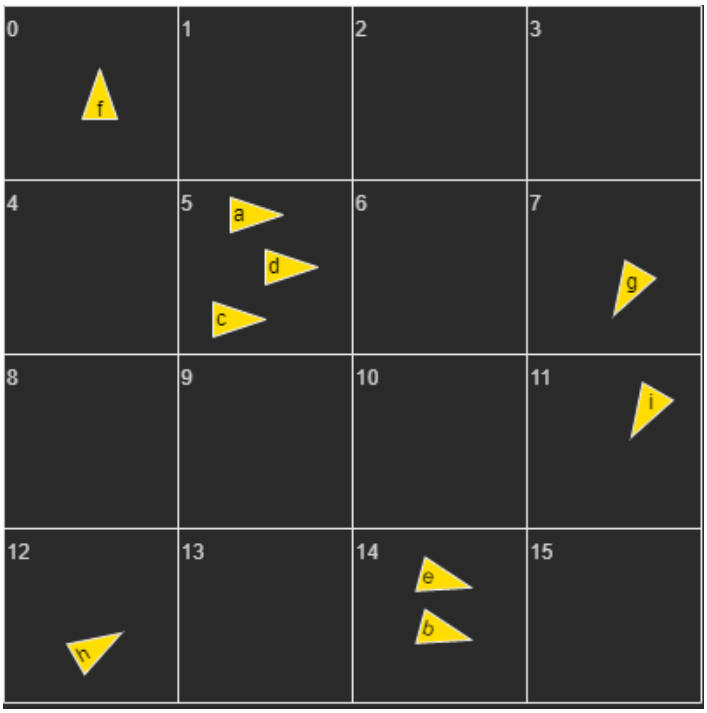


Figure 1: Boids arranged in a uniform grid.

Two implementations stem from using the uniform grid: **scattered** and **coherent**.

Scattered

This is the most intuitive application of the uniform grid:

- Firstly, we calculate the **grid resolution** given the boid field of view (which defines the size of its neighbourhood)
- For each boid, we calculate a **linear grid index** given its position in the simulation space (creating a mapping *boid index* ⇆ *grid index*)
- We **sort** the boid indices and grid indices arrays by ascending grid index, so that neighbouring boids (in the same cell) are locally close inside the array.
- We calculate the start and end indices for each cell in the grid indices array so that we know the **range** we need to iterate into when checking against boids in the same cell; index range is to be considered [inclusive, exclusive).
- We finally compute the three behaviours (alignment, cohesion, separation) for each boid, taking into account only the other boids in the same cell based on the grid indices array built before.

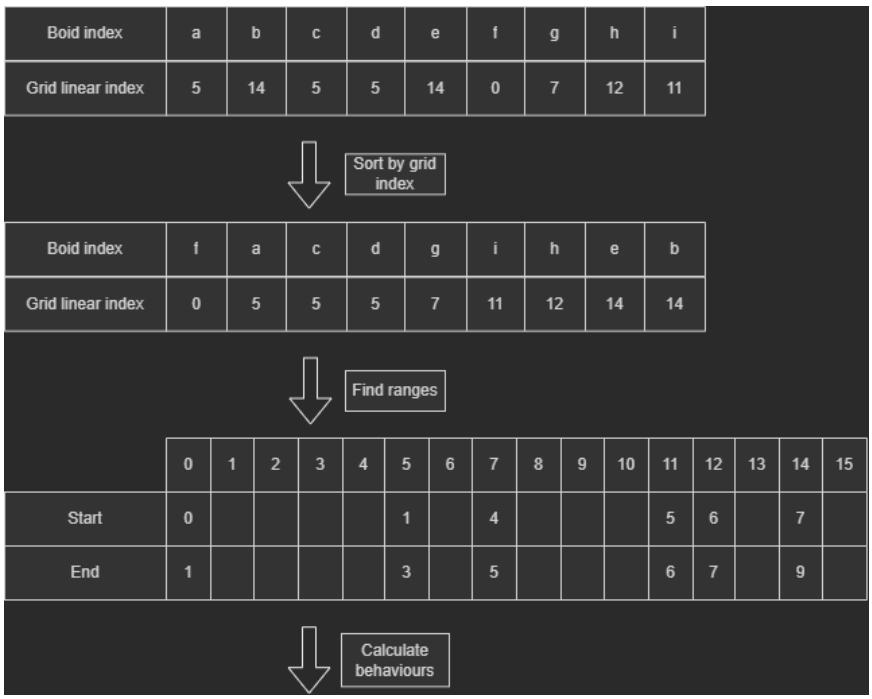


Figure 2: Example of algorithm execution following data from Figure 1

Coherent

Additionally, from the scattered version, we also **sort** the arrays containing the **positions and velocities** of the boids given their grid index before computing the behaviours.

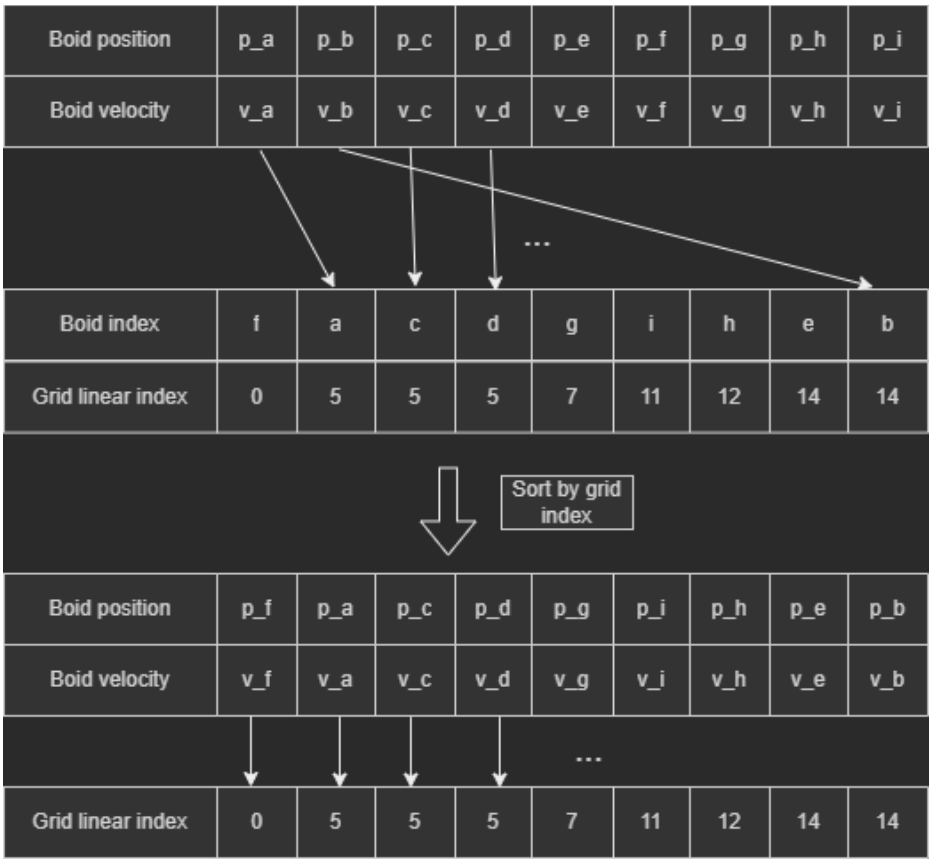


Figure 3 : Data is now contiguous and reads can be better coalesced.

This way all **data is locally close and contiguous**, avoiding us from reading from the middleman indexing array and thus reducing the amount of global memory reads. This will be important later on when discussing about implementations' optimization.

Boid behaviours

As mentioned before, each boid has three basic rules (or behaviours) to follow: alignment, cohesion, and separation.

Each behaviour returns an **acceleration** which, once blended with the others, can be applied to a boid. The blending is **weighted** through coefficients, one for each behaviour, so that their magnitude can be controlled at runtime interactively.

In addition to these three behaviours, a fourth behaviour was added for the simulation's sake: **wall separation**.

Since the simulation space happens inside a fixed size cube, we need to **keep the boids confined** to that area in the most natural way possible: simply clamping the position inside the cube bounds is not sufficient for that purpose.

The cube is nothing but a collection of planes, each with its own normal pointing towards the inside of the cube: the wall separation behaviour job is to simply apply a force along that normal whenever a boid gets too close to a boundary plane.

Environment

Summary

The simulation depends on different libraries:

- OpenGL + GLFW, for visualization and user input purposes
- GLM, for vectorial computation
- Imgui, for the user interface
- CUDA, for the parallel computation framework
- CURAND, for the random generation of initial positions/velocities for the boids
- CUDAGL, for direct interoperability between OpenGL buffer objects and CUDA kernels
- THRUST, for its sorting algorithms

Additional various utils for camera management, compilation, math operations and more are present as well.

Visualization

As we mentioned before, we used OpenGL to visualize the boids in our simulation.

Each boid is a simple triangle, drawn using the **instancing** technique: there is only one mesh which is replicated n-times, then transformed to different positions and orientations given the data from each boid. This makes up for a faster drawing and synergizes with our data setup for the boids.

There were mainly two problems we faced when setting up the visualization step of the simulation: given the elevate number of objects, we need

- a big **enough storage** for the sizable boids data (positions and velocity arrays) in the shader pipeline.
- to **pass the data directly** to the shader pipeline from the kernel, avoiding unnecessary and slow CPU⇆GPU data transfers to fill the buffer objects.

The best compromise was found in the usage of the GL **SSBO** (Shader Storage Buffer Objects), which are a special kind of buffer objects large enough to store our data and capable of interoperability with CUDA.

CUDAGL Interoperability

To let CUDA kernels be able to directly access the area of memory used by the shaders' SSBOs, we created a **simple wrapper** class (namely `cudaGLmanager.h`), which implements the CUDAGL interoperability features.

Through the wrapper, we are able to **register the needed GL resources**, such as our SSBOs, and map them to a **device pointer** usable by CUDA kernels; this is all done in a safe and controlled way, with automatic memory unloading when the manager ceases to exist.

Simulation parameters

The simulation was born with the intent to be as much **interactive** as possible, with the possibility to change the boids behaviour at runtime through the GUI.

For this reason, much of the simulation was parametrized, offering several degrees of liberty through two sets of parameters: dynamic and fixed.

Fixed parameters can only be set at startup via **command line** arguments, which are:

- Boid amount
- Cube size (represents the length of the side of the simulation cube)
- Simulation type (selects the algorithm to use between naïve, uniform or coherent grid)

Dynamic parameters instead can be changed via the **interactive GUI**, which are:

- Boid speed (represents the speed at which boids travel)
- Boid FOV (represents the neighbourhood radius explored by boid behaviours)
- Boid behavioural weights (lists the weights for each of the four behaviours: alignment, cohesion, separation, and wall separation)

Additionally, in the GUI we can find a special **"breath" toggle**, which is an extra function that automatically oscillates the cohesion and alignment weights to simulate the typical expansion and reduction of bird flocks.

Implementation

Runner framework

For the sake of easy comparison between CPU and GPU, we implemented **a "runner" framework**, which is a simple, hierarchical way to generalize a class able to run the simulation. This makes possible and effortless eventual extensions for the simulation.

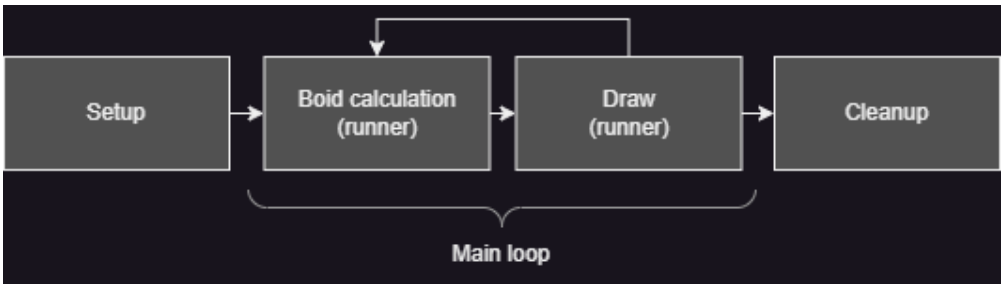
The base abstract class `boid_runner.h` defines the bare **minimum setup for a runner**, including simulation related data and methods which will have to be implemented in the concrete derived classes.

On top of it, we built the abstract class `ssbo_runner.h` which defines the **setup** for a **runner based on GL SSBOs**.

Finally, we developed the runners' concrete implementation classes:

- `cpu_ssbo`, runner designed to execute the simulation **sequentially** on the CPU
- `gpu_ssbo_monolithic`, runner designed to use CUDA kernels to run the simulation **concurrently** on the GPU: the behaviour calculation is "monolithic" and done in a **single kernel**
- `gpu_ssbo_modular`, runner designed to use CUDA kernels to run the simulation **concurrently** on the GPU: the behaviour calculation is done by **different kernels**, one for each behaviour (more on that later)

In the end, the simulation's execution flow resembles something like the following:



The code mentioned is found in the *runners* subfolder of the project.

Algorithms implementation

Naïve

This algorithm is simply implemented by calculating every behaviour for each boid, weigh the resulting accelerations, blend them together and update the boid's velocity and position; no extra steps are required.

The monolithic solution uses a single kernel for it (`flock` function), while the modular one uses a kernel for each behaviour and then blends them together through the `blender` kernel function.

Grid-based (scattered and coherent)

These implementations are very similar, except for the sorting of velocity and position arrays in the coherent case.

Firstly, we create the grid (`assign_grid_indices` function) considering two parameters: the simulation cube size and the boid field of view. Given these, we manage to calculate the **grid's resolution** and **localize each boid** through its world coordinates, assigning each a **linear index** representing the cell the boid is into.

To keep it simple, the STD library (in the sequential case) or the THRUST library (in the concurrent one) were used to **sort** the boids by their cell index.

The ordering by cell index will give us an advantage when iterating over boids in the same cell, since they'll be **adjacent** in that array: we just need to find the **index range** from which each cell starts to which it ends (`find_cell_boid_range` function).

In the coherent case, as mentioned before, we'll also need to **reorder the velocity and position arrays** by the cell index (`reorder_by_bci` function).

Finally, we **apply the behaviours** to each boid and get the final position/velocity for that frame (`flock` function for the monolithic approach, behaviours plus `blender` function for the modular one).

CUDA implementation challenges and optimizations

Problems with modular behaviours

The behaviours were originally thought to be extremely modular and isolated from each other, so that each of them would simply be a function that, given some boid data, would return an acceleration relative to the desired behaviour.

While this was working just fine for the sequential approach, it didn't perform as well in the concurrent implementation of it.

Even by using streams to parallelize as much as possible, the amount of global reads/writes would just tank the performance, as each behaviour would read redundantly the same boid data multiple times, resulting in performance loss.

Furthermore, parallel execution of the behaviours can be hindered when the boids are tightly packed: behaviour kernels struggled to overlap in these cases even if they were on different streams.

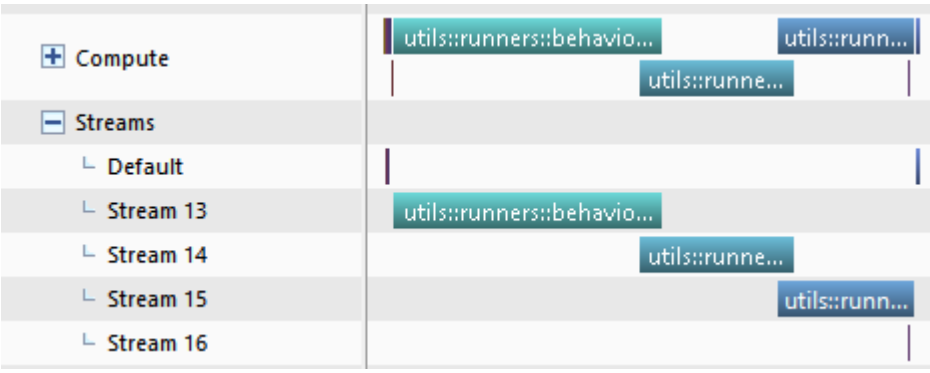


Figure 4 : Poor kernel overlap between alignment(stream 13), cohesion (stream 14), separation (stream 15) in the problematic frames

Consequently, the optimization goal went quickly towards **minimizing global memory data accesses**, while making them the least troublesome possible through **better data alignment and coalescence**.

The following optimizations resulted in the more efficient monolithic kernel implementation (used by the runner `gpu_ssbo_monolithic`) against the modular kernel implementation (used by the runner `gpu_ssbo_modular`).

Optimization through local and constant memory

One of the first things done to reduce global memory accesses was to increase the usage of local memory through **registers**.

By condensing the behaviour calculation into a single kernel (`flock` function) and **reusing the registers data**, we removed the reading redundancy mentioned before.

The amount of registers used does not cause spilling and was balanced to not hinder occupancy too much.

After that, we noticed that the simulation parameters are a great candidate to be stored in **constant memory**, as they are:

- constant throughout the kernels' execution
- highly requested for calculation by behaviours' kernels

Any change in those values can only come from the host in-between a frame and the next one, plus they are rarely subject to continuous changes through normal application usage.

Optimization through better access patterns

Another change that was introduced to increase data coalescence is the **change in design** from AoS (arrays of structs) to SoA (structs of arrays).

For example, instead of using an array of structs to contain the mapping boid id ⇆ cell id (e.g. struct `boid_cell_index` in the code), we used two distinct `int` arrays (e.g. `bci_boid_indices_dptr` and `bci_cell_indices_dptr`).

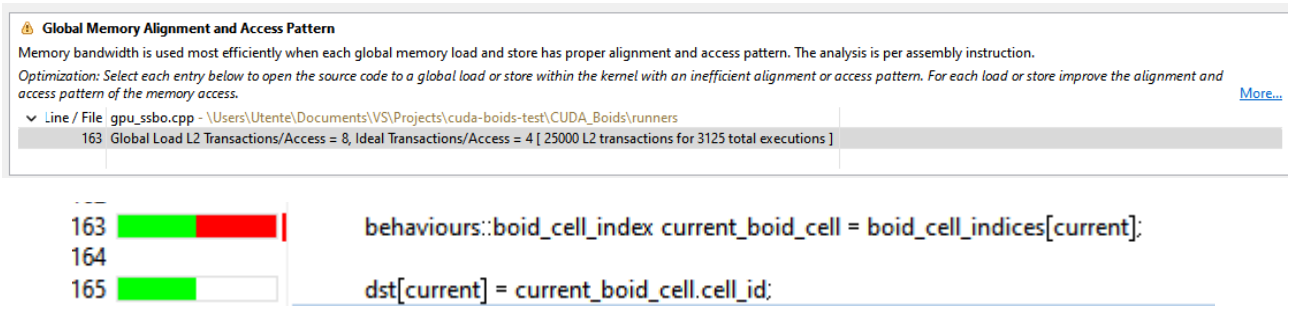


Figure 5 : an example on how the SoA design was wasteful in memory accesses; we needed only one value from the struct (.cell_id) but had to load the whole to access it

The same can be said for the index ranges calculation (e.g. array of `idx_range` structs vs two `int` arrays for start and end indices).

Difficulties for shared memory usage

Another idea was to use shared memory to ease the burden of global memory access, however it revealed to be challenging to properly use.

Each boid appears in its own cell, with its own neighbourhood range, which size is extremely variable and, in crowded simulations, significantly large.

Consequently, these conditions imply **sizable and variable shared memory requirements**, not necessarily consistent between kernels.

As these circumstances felt not ideal for shared memory usage, we opted to prefer a bigger L1 cache to help us reach a bigger cache hit rate.

Reduction of warp divergence

Another attempt to increase occupancy and efficiency was the **avoidance of warp divergence** as much as possible.

This was done extensively inside kernels by using **branchless operations**, where conditions are evaluated and applied as coefficients for values.

For example, behaviours' accelerations are checked against the boid's field of view: if a nearby boid is outside the radius, the condition evaluates to 0, thus zeroing that acceleration and adding nothing to the final result.

Benchmarks

We are going to see how we evaluated the various implementations under a comparative analysis.

Hardware

The simulation testbed is composed by a Nvidia GTX750Ti coupled with a Ryzen 5 3600 and 16GBs of RAM @ 3200 MHz.

Comparative analysis

Methodology

We have two counters, each measuring respectively

- the **amount of frame per second**
- the **duration of the calculation** required by the runner to calculate the new boids' positions and velocities.

These values are calculated as **exponential weighted moving averages**, where new samplings are more impactful than old ones, which decay through an alpha coefficient.

We will consider for our tests, however, only the **calculation time spent by the runner** as we do not want to take into consideration the possible bottlenecks introduced by the effective drawing of the frame by OpenGL: we are only comparing the **runners' efficiency**.

We are going to test the three aforementioned algorithms on CPU (sequential implementation) and on GPU (both modular and monolithic approaches) on increasing boid amounts (10000, 100000, 1000000) for 30 seconds of simulation.

Performance is also affected by the `boid_fov` parameter, which represents the radius of the neighbourhood to check against for each boid: the bigger, the more probable is to have more boids to check against; in these tests, we'll keep it to 5.

Finally, the chosen block size is 128 as suggested by the Nvidia profiler.

Results

With 10’000 boids	CPU	GPU MONOLITHIC	GPU MODULAR
NAIVE	515.5731ms	**13.0743ms**	20.6095ms
SCATTERED GRID	3.3695ms	**0.4845ms**	0.5097ms
COHERENT GRID	2.9662ms	**0.4917ms**	0.6628ms

With 100’000 boids	CPU	GPU MONOLITHIC	GPU MODULAR
NAIVE	Fails to launch	**1412.9448ms**	1986.8654ms
SCATTERED GRID	186.8573ms	**2.6050ms**	3.1146ms
COHERENT GRID	182.8267ms	**2.3131ms**	3.0171ms

With 1’000’000 boids	CPU	GPU MONOLITHIC	GPU MODULAR
NAIVE	Fails to launch	Fails to launch	Fails to launch
SCATTERED GRID	1821.0509ms	**143.1144ms**	205.9928ms
COHERENT GRID	1587.8992ms	**61.2058ms**	201.6042ms

As we can see, in the majority of cases the GPU **monolithic implementation** has the best performance. As expected, the CPU implementation falls behind rapidly as the boid number increases, while the GPU modular is able to stick around but with some performance losses.

We can also notice that the benefit of the coherent grid really starts to show at high amounts of boids, where the speedup can get up to ~2x the scattered version.

Conclusions

All in all, even though there are still possible margins of improvement, we’re satisfied with how the simulation ended up being: the improvement brought by the usage of the parallel architecture is evident as the simulation is smooth and interactable, even with large amounts of boids on screen.