Simulating Flocking Behavior Using OpenMP and OpenACC

In 1987, Craig Reynolds wrote a paper describing how to model the behavior of flocks of animals for computer graphics. He attempted to simulated birds, coining the graphic version he created bird-oids, or "boids" [1]. He pointed out that the behavior he modeled also applied to schools of fish and herds of other animals.

This groundbreaking work by Reynolds ushered in the study of individual agents that can exhibit self-organizing behavior and has been cited in other works well over 14,000 times. One such citation is in a book by Gary Flake called *The computational beauty of nature* [2]. In chapter 16, section 3, Flake describes his implementation of Reynolds' algorithm. We started with Flake's original code when formulating this assignment.

Brief description

The provided file called *"GaryWilliamFlak_1998_163FlocksHerdsAndScho_TheComputationalBeaut.pdf"* contains the section of Chapter 16 from Flake's book that describes the algorithm that models the boid behavior. A given number of boids are placed randomly on a canvas with a given x and y position and initial velocity, then begin moving according to some rules. Flake used three 'rules' from Reynold's work to apply to each boid as it moved and added a fourth rule of his own. The movement takes place by *simulating time* using a for loop for a relatively arbitrary number of times: each time through the loop, each boid applies the rules by observing the location and speed of every other boid. According to Flake, the rules are:

1. **Avoidance.** Move away from boids that are too close, so as to reduce the chance of in air collisions.
2. **Copy.** Fly in the general direction that the flock is moving by averaging the other boids' velocities and directions.
3. **Center.** Minimize exposure to the flock's exterior by moving toward the perceived center of the flock.
4. **View.** Move laterally away from any boid that blocks the view

More details about weighting each rule and combining them are described in the pdf file.

Flake has functions for displaying the boids as arrows moving at each time step in an X window display. A snapshot is shown in Figure 1.

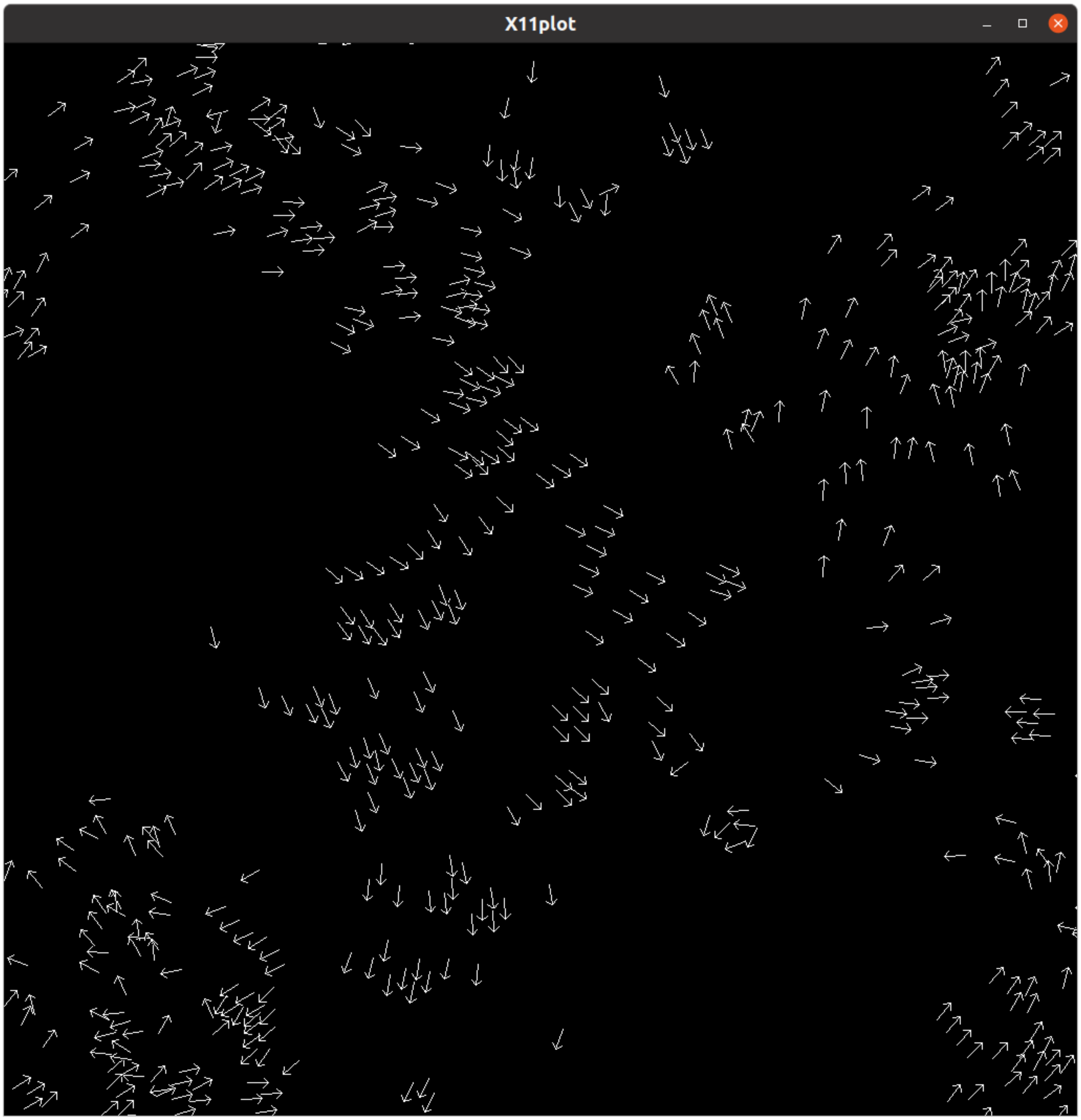

Figure 1. Boids program display

Necessary Background

You will be updating two versions of this program: 1) OpenMP for CPU and 2) OpenACC for GPU. You will need practice using the pragmas for the compilers that provide multithreaded capability using both of these specifications.

You need to have studied some examples from OpenMP to be able to apply them to this situation. One reference is the PDC for Beginners book, chapter 1, where you can see some OpenMP examples in action. Another is the full set of pattern examples in Chapter 2 of the Intermediate PDC book.

To complete an OpenACC version for the GPU, you should study and practice the code examples in Chapters 7 and 8 of the Intermediate PDC book. It is also helpful to consider the CUDA GPU Programming model described in Chapter 4, section 2 of the PDC for Beginners book. The OpenACC compiler creates CUDA code, and the programming model of 1 thread per array data element computation applies here.

To consider the scalability of your parallel solution, you might want to read about how this is determined in PDC programs by reading PDC for Beginners book, chapter 0, section 3.

Starting Point Code

Directory: GPUpsPlot

The code that you will be working with is a revised version of Flake's original code, which contained C code files that enabled an X11 window to display the boids moving on the canvas.

The sequential code file called boids.c, based on Flake's original, is provided. It has been modernized to remove Flake's use of global variables. There is a second file, currently sequential without any pragmas, called boids_acc.c that is almost the same as boids.c.

Here are a few notes about the boids.c and boids_acc.c code and how to run it.

The Makefile is provided to build the *boids* program and one called *boids_acc*. It compiles using proper flags for each version and links in several other files that are primarily for the display. You shouldn't have to change the display files and **the files to concentrate on are boids.c and boids_acc.c**.

compute_new_headings() is the function that applies the rules at each simulated time step.

The main function calls compute_new_headings() for each simulated time step. The number of steps has a large default value of 100000000, so that you can observe the behavior changing over time and see the flocks forming. You can change this and many other parameters- this is described fairly well in the main() function, and some essential parameters are described below. To see what all the parameters are, you can type this in a terminal after you have used 'make' to build the programs:

```
./boids -help
```

Here is one way to run the code with 512 boids and a width and height of 1024 on the display window:

```
./boids -width 1024 -height 1024 -num 512
./boids_acc -width 1024 -height 1024 -num 512
```

Each of the above will take a long time to run, because the number of steps is so large. The ctrl-c combination of characters should stop it. To run it with a certain number of steps, do this:

```
./boids -width 1024 -height 1024 -num 512 -steps 1000
```

And similarly for boids_acc.

After 1000 steps, the result should look like Figure 1, where you can see that the boids, represented as arrows, have formed groups that are moving in a similar direction.

Eventually, you will want to time your code without showing the display. To do this, add '-term none', like this:

```
./boids -width 1024 -height 1024 -num 512 -steps 1000 -term none
./boids_acc -width 1024 -height 1024 -num 512 -steps 1000 -term none
```

Study the code

The decision made for each boid about where it should move next at each time step is independent-- it uses the current information about the boids near it to simply update its own new position and velocity. What this means is that the computations made for each boid during a particular time step can be executed in parallel.

You should try to observe how this is possible. 1-dimensional arrays are used for current x, y positions and velocities of every boid. Here is the gist of the program:

```
for each time step
    compute new headings:
        for each boid
            look at every other boid except myself (another loop)
            use boids near me to determine a *new* heading, kept in separate
arrays

    once all new headings are computed,
    for each boid
        'undraw' the current boid
        copy the new heading values into the arrays holding the curent
headings
        draw the boid in its new position
```

Use this pseudocode to help you study the existing code. This general method is found in a great deal of simulations that use time steps to update positions and other attributes of each element in the world being modeled.

A key element here to help you decide how to parallelize the code is to recognize what calculations in which loop are independent and why.

Your goals

There are two code files so that you can build two parallelized versions of the code: One for OpenMP shared memory parallelism with threads, and one for OpenACC using the GPU. Then you can compare each of them under varying conditions to observe how well they scale as you increase the number of boids and the size of their world (width, height).

Here are your primary tasks:

1. Transform boids.c to use OpenMP pragmas. The boids.c file already has code that sets the number of threads. You do this by using the -t flag, as shown below. Note that the code file has OpenMP calls for timing the code also. Your task is to identify the loops in the code and determine where you can use the parallel for loop pattern to decompose the work onto separate threads. Here is an example test using 4 threads:

   ```
   ./boids -width 1024 -height 1024 -num 512 -steps 1000 -term none -t 4
   ```

   See the section on Debugging below for how you can do some crude tests to see that the code is working correctly when you try multiple threads vs. one thread.

2. Test your OpenMP solution with the same conditions of your choosing while varying the number of threads. It should be possible to see the display go faster and you should see the time to complete go down as you increase the number of threads. Note that you will need to experiment with the number of boids and the size of the world and the number of steps. For deciding on the performance of the program, note that changing the number of steps isn't relevant- just make sure that you use enough so that the flocking behavior is happening and that it runs for several seconds. Other factors are more relevant for determining when the program is scalable.

3. Transform boids_acc.c to use OpenACC pragmas for a GPU version. In this case, consider the GPU programming model carefully; you choose where to use the parallel for loop pattern so that each thread in the GPU is doing one independent update per boid per time step.

   Hints: We have observed a few important aspects of using OpenACC pragmas on code of this complexity. In addition to describing which loop should be split across all the threads available on the GPU, you should describe which loops should not be run in parallel by using **#pragma acc loop seq** above them. It seems to be necessary to use the clause **collapse(1)** on some outer loops (parallel and sequential) so that the compiler gets a hint from you to not collapse the loop. There is one case where using collapse (2) is useful on one inner nested loop. Another important aspect of using the OpenACC compiler on this code is that it is mach better to let it decide when data will need to be moved from the host to the device and back. **In other words, there is no need for pragmas with data directives in this code and using them could cause you problems.**

   See the debugging section below for more tips on ensuring that the code is correct. You will get slightly different results for the GPU version than from the OpenMP CPU version. Can you explain why this is? You could check with your instructor.

4. Test your boids_acc solution on varying sets of initial conditions. You can start with something you can observe using the X display window. Next, for this version on the GPU it is interesting to see how you can

scale the program to a much larger world (width, height) and a much larger number of boids than can easily be displayed, so you will want to use the command line argument *-term none* as you increase width, height, and number of boids. You can compare times to the OpenMP version under the same conditions. Be sure to try out lots of different conditions to see where the GPU version might be faster. Keep the number of steps at 600 and vary -width, -height, and -num. At some point, keep width and height fixed fairly large and increase the number of boids with -num. You should see a point where using a large enough number of boids runs faster than using 8 threads on the OpenMP CPU version.

Write a report

You should report on your solutions and what you have learned about the scalability of them. This should be the kind of report you might write to someone interested in the changes you made and/or how well it performs, including when it is best to use the GPU. You need to provide enough detail that a reader of your report can repeat your work.

Criteria for grading of report

Make a useful, well-written report that describes your code optimizations and the results. Below are some items you should include. You can choose the order in which you present the following ideas in your report, using sections with headers.

- Devise a title for your report.
- Include your name (own your work!)
- Explain briefly what problem you are analyzing and optimizing- orient the reader of your report to the problem.
- Explain the code updates that you made.
- Explain what command line argument(s) you ultimately decided to vary in your experiments. This constitutes your 'problem size' that you change and obtain timing results for, using each version. Explain how you changed it and your observations about performance as it varied.
- Write about the experiments you tried that did not show much scalability and those that did. Write an explanation of why the GPU version performed the way it did.
- Describe your methodology: how many times you ran your experiments and what conditions you used (i.e. cases you tried). Sometimes a table is useful for this.
- You should show regular speedup curves for the OpenMP version for several problem sizes.
- The concept of speedup is different for code on accelerators. Using many smaller, slower cores speeds up code in a different way so that traditional speedup does not apply. Use other methods to show the change in speed, such as bar charts that compare each of the versions for varying problem sizes.
- Ultimately, what did you learn by parallelizing this example with these different compilers and the difference between the fast CPU and slower cores on the GPU?

Debugging

Towards the end of the file and the end of main(), note 3 instances of commented lines where the first line starts with:

```
// for debugging
```

The next line is designed for you to uncomment so that you can see whether the new headings have reasonable values and whether the updates to the current headings seem to be working. **Use these prints with a small number of time steps (10 or so)**; otherwise there will be too much data shown.

We can see if this is repeatable because we have used a very simple seed for the random number generator that is the same every time we run it. You should be able to see that it is correct from the visualization, also.

You can even use these before starting just to get the gist of how the code is working. Note that it is simply following how the zeroth boid is moving. Its final position is also being printed at the end of the program as a crude check of whether it is working.

If there are problems in either version of your code, the positions of the boid will seem wrong, such as really large x or y values, or velocities that get small and/or don't change from time step to time step.

References

[1] Reynolds, Craig W. (1987) Flocks, herds and schools: A distributed behavioral model. Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 25-34.

[2] Flake, Gary William (2000) The computational beauty of nature: Computer explorations of fractals, chaos, complex systems, and adaptation. MIT Press. 270-278