Ben Bennett
CIS 531 Term Project
6/8/17

Project Description

For my project I did a simulation and visualization of flocking behavior. I did the parallelized portions of flocking on a GPU using CUDA as well as a parallel version using OpenMP. I did the visualization in order to verify the flocking behavior was working as expected. The visualization was done using OpenGL. I chose using a GPU because the math behind flocking is relatively simple (averaging values of neighbors) and this would be a great opportunity to get experience using CUDA. I also choose flocking because I enjoy writing programs where you can see what is happening in real time, for me it is very satisfying.

The flocking algorithm is based on three rules. The first is separation, this is the rule that a boid (object that follows a flocking behavior) tries to keep its distance from its neighbors. The second rule is cohesion, this is the rule that a boid tries to stay close to its neighbors. It may seem like these rules will cancel each other out, but the separation value scales relative to how close two boids are to each other, that is, closer boids will have a stronger separation value. The third rule is alignment, this is the rule that a boid travels in the same direction as its neighbors. A weight is also applied to each of these values, adjusting the weights can give system vastly different behaviors.

Implementation

For all implementations I used C++. For the parallel version I used OpenMP. I used CUDA for the GPU parallelization. View the Readme in the project source folder for details on how to build and run the application.

The bulk of the computation comes from updating the positions and directions of each boid. The reason this is so expensive is that each boid has to find all neighbors within a set radius. My implementation does this as a nested for loop resulting in a $O(n^2)$ operation, where n is the number of boids in the system.

My parallelization strategy was to run the update for each boid in parallel (i.e. parallelize the outer for loop). Each thread would still need to perform a loop over all n elements to find nearest neighbors. Assuming infinite threads the runtime will be O(n) per frame.

I chose to do the visualization in 2D. I first tried 3D, but it was extremely difficult to see what was happening. It terms of computation there is very little difference. The 2D version just uses vectors of length 2 instead of 3.

The article I used for the calculation of the flocking rules: link
Citation:

Reynolds, C. W. (1999, March). Steering behaviors for autonomous characters. In *Game developers conference* (Vol. 1999, pp. 763-782).

Testing Environment

For all testing I used the Frankenstein cluster. That cluster has an Intel Xeon E5620 CPU with 16 cores clocked at 2.4 GHz. There are three different GPUs on the cluster. The first is a NVIDIA Tesla K20c. The K20c has 2496 CUDA cores clocked at 706 MHz with 5120 MB of memory. The next is a NVIDIA Tesla C2075. The Tesla C2075 has 448 CUDA cores clocked at 1150 MHz with 6144 MB of memory. The final is a NVIDIA GeForce GTX 480. The GTX 480 has 480 CUDA cores clocked at 1401 MHz with 1536 MB of memory. Each GPU was used during testing.

Results

The metric I used to test performance was average frames per second. For testing I did not draw the graphics, drawing graphics over the network caps the frame rate to around 50-60 frames per second in the best case. I performed the test by running a 15 second simulation, 4 tests per configuration.

Raw data:

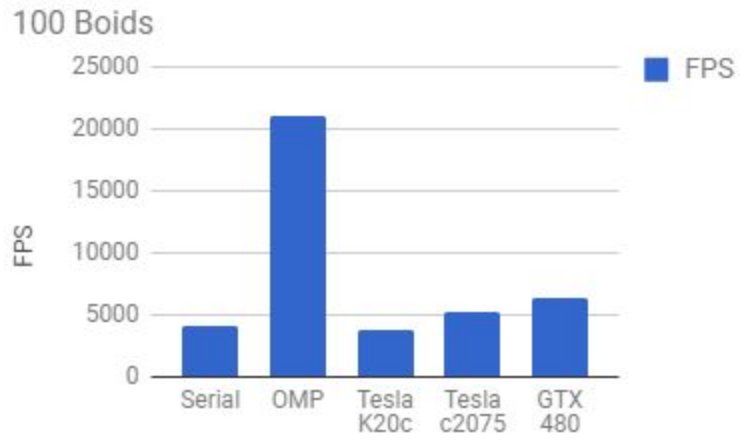
	100	1,000	10,000	50,000	100,000
Serial	4008.93, 4017.66, 4010.61, 4020.68	45.33, 45.46, 55.59, 44.69	0.53, 0.53, 0.53, 0.53	N/A*	N/A*
OMP	22339, 22526.1, 18823.2, 20409.4	393.8, 394.1, 394.5, 379.3	4.19, 3.94, 4.16, 4.17	0.22, 0.22, 0.21, 0.22	N/A*
Tesla K20c	3845.98, 3608.59, 3783.44, 3690.09	563.23, 562.75, 560.0, 568.19	45.05, 45.65, 44.86, 46.33	2.62, 2.62, 2.66, 2.63	0.74, 0.78, 0.75, 0.77
Tesla c2075	5206.52, 5321.36, 5043.35, 5131.66	680.44, 696.92, 694.51, 691.1	31.77, 31.87, 31.12, 31.25	1.98, 1.91, 1.97, 1.97	0.56, 0.57, 0.56, 0.56
GTX 480	6200.45, 6302.08, 6279.59, 6438.37	822.93, 839.24, 830.89, 824.23	36.32, 37.52, 36.56, 36.15	2.40, 2.34, 2.37, 2.32	0.71, 0.70, 0.69, 0.71

* N/A denotes that one frame could not be computed in 15 seconds

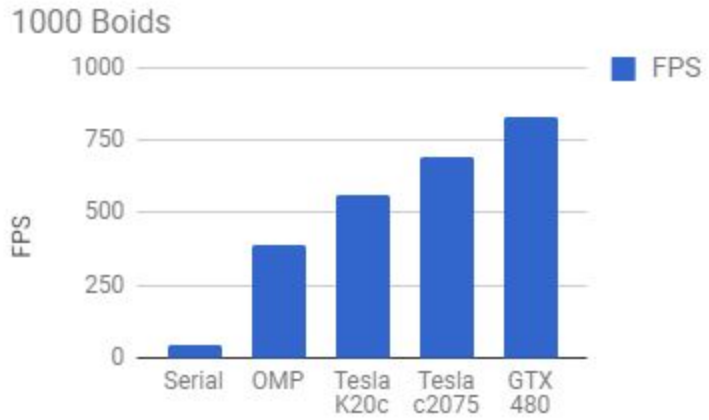
Averages

	100	1,000	10,000	50,000	100,000
Serial	4014.47	47.7675	0.53	N/A	N/A
OMP	21024.425	390.425	4.115	0.2175	N/A
Tesla K20c	3732.025	563.5425	45.4725	2.6325	0.76
Tesla c2075	5175.7225	690.7425	31.5025	1.9575	0.5625
GTX 480	6305.1225	829.3225	36.6375	2.3575	0.7025

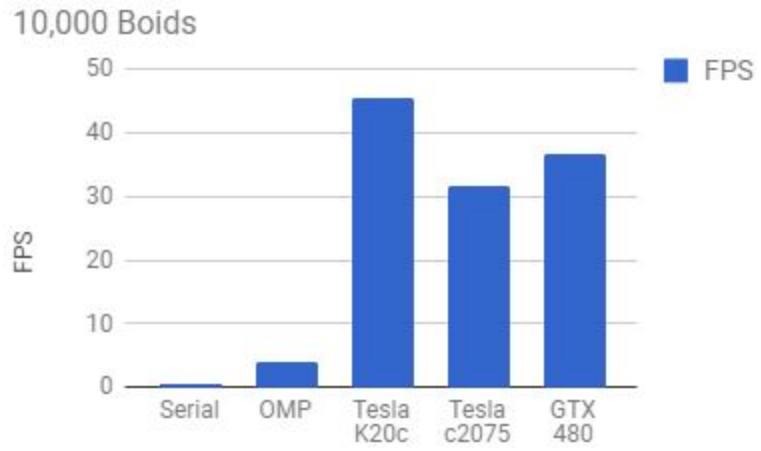
Analysis:



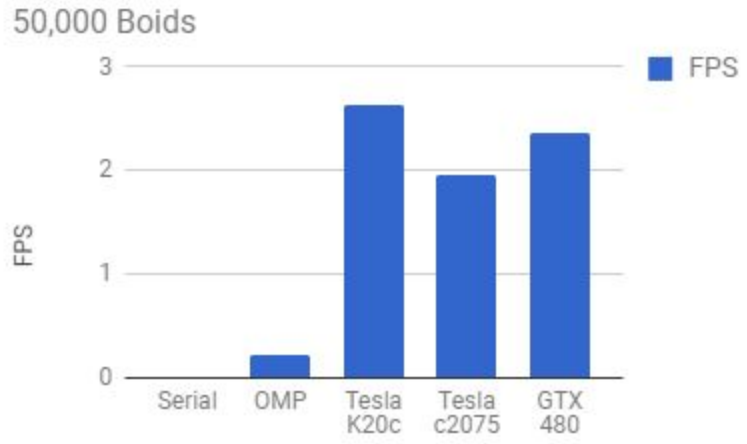
For 100 boids we can see that the OpenMP version far outperformed every other. This is expected when comparing against the serial version, when running in parallel OpenMP uses 16 threads. What is more surprising is that the serial version performs almost as well as the GPUs. But if you look at the clock speeds of the GPU it begins to make sense. Each GPU has enough cores to run an update for each boid on a separate thread in parallel, but when you look at the clock speed of each core it is much lower than the CPU clock speed. This combined with the overhead of copying data to and from the GPU each frame makes it clear why the GPU struggles with a small number of boids to update.



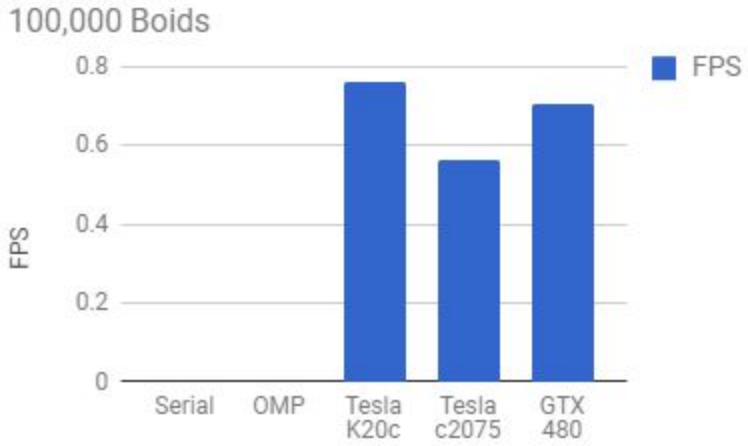
With 1000 boids we are starting to see the benefit of the GPU. The higher clock rate of the Tesla c2075 and the GTX 480 are still outweighing the fact that they have about 5 times fewer cores than the Tesla K20c.



With 10,000 boids the advantage of the GPU is clear. The advantage of the Tesla K20c having 5 times as many cores as the Tesla c2075 and the GTX 480 is having a greater impact.



With 50,000 boids the serial version did not get through one frame in the 15 seconds test time. It is also interesting to note that the ratios of the Tesla K20c vs. Tesla c2075 vs GTX 480 are roughly the same as the results for 10,000 boids even though the actual FPS for all three is much lower.



Both the serial and OpenMP versions could not compute one frame in 15 seconds. Once again the ratio is approximately the same as 10,000 and 50,000. At this point the frame rate is far too slow for any real time visualization.

Lessons Learned and Potential Optimizations

I took the approach of doing the serial version first, followed by the parallel version, and then followed by the CUDA version. If I was going back I would have started with a sample CUDA version. The reason is that I used a lot of different parameters when updating the boids such as; weights for each flocking rule, search radius for close neighbors, etc. Each one of these values needed to get passed to the GPU kernel (in the non GPU versions I just used the values globally). This resulted in a two functions with a massive amount of parameters. My choices were to allocate memory on the GPU for each variable and copy them over or just pass them to the kernel, after doing a little research I chose the later. Knowing this earlier would have made me reevaluate my design early on and I would have come up with a cleaner solution.

I also did very little up front design, it was more along the lines of testing things to see if they work, but then leaving it in. A little up front design would have made for more object oriented code and just cleaner code overall.

A major point of optimization would be with the nearest neighbor search. A more efficient solution would be to divide the world up into a grid and put each boid into a grid cell based on its position. Then in the update you only need to look at the boids in the current grid cell and the surrounding grid cells to find all neighbors. This still could be $O(n^2)$ worst case, but due to the fact that flocking behavior discourages boids getting close together this should never be the case. I would estimate that this method would show massive improvements to run time.

Another optimization that could be had is with drawing the boids to the screen, this is a major point of slowdown with large numbers of boids. Currently I am just doing a standard OpenGL draw where you define vertices and it does the rest, but this has to be done for each boid each frame. What could be done to improve this is using Instanced Drawing. With this you pass an array of positions to a vertex shader and it draws a copy of a single mesh to the screen at each position in the array. This itself is highly parallelized since the vertex shader is run on the GPU, but it is handled by OpenGL.

For fun here is a picture of it running. This is running on the Tesla K20c with 10,000 boids