Rishma Mendhekar and Isaih Porter
COM495/496: Senior Research
Large-Scale Particle Simulation on the GPU

Abstract

This study examines the increased performance of large-scale particle simulation on the Graphics Processing Unit (GPU) against conventional implementation on the CPU. Usage of General Purpose GPU (GPGPU) programming, which utilizes massively parallel computational algorithms, has grown substantially over the last decade and particle simulation is one of such examples. We developed a particle simulation program using a Compute Shader on the GPU to calculate particle motion with a 3 dimensional Perlin noise algorithm. The current implementation shows around 60 frames per second (FPS) in 4K resolution for about 8 million particles of a point primitive type as well as a quad sprite model. The performance gain over the equivalent version on the CPU is about a 200x speedup in frame rate. Both the CPU and GPU versions of the program were created using C# and HLSL with Unity and DirectX. We deployed this program for the art installation of The Posture Portrait Project at Connecticut College to achieve an image-dissolving visual effect where each particle is generated from image pixels. We have also implemented a boids flocking algorithm using the GPU using C# and HLSL. This particle motion requires significantly more computation than Perlin noise as each particle will need to be aware of each other particle's location. We ended up with three versions of boids flocking with differing levels of CPU/GPU involvement-- one version where boids were rendered as GameObjects, as sprites, and as polygons. The GPU used for testing both particle systems was an Nvidia Geforce GTX 1080 and the CPU an Intel Core i7-6700k @ 4GHz.

Introduction:

We are researching methods of creating a large-scale particle simulation using a GPU. One of the problems with dealing with such a large amount of data is that the simulation takes substantial computational time due to the CPU's inability to process and interpret large numbers of pixels in real time. The goal of the research is taking most of the stress away from the CPU and giving the GPU most of the work. There is already a trend of software moving towards general-purpose computing on graphics processing units (GPGPU); such as Adobe Photoshop and Premier, and even MATLAB[6,7]. We analyze and compare the efficiency of the GPU to that of a CPU in order to optimize a real-time particle field using large, dense data. We utilize DirectX in conjunction with Unity, resulting in a Unity plug-in that creates particles based off of millions of pixels from a 4K resolution image. Each of these images contain millions of pixels; the particles will be initiated based on the position and color of these pixels.

We first conducted a literature review on large-scale particle systems. Jesper Hansson Falkenby's bachelor's thesis, Physically-based Fluid-particle System using DirectCompute for Use in Real-time Games, discussed GPU usage for fluid-particle systems in games. Falkenby implemented a real-time fluid-particle system with two different fluid physics models in addition to a gravity-only model. Though all of the fluid models were scalable to around a million particles, the gravity-only model performed much better than the other physics model because it did not require the calculation of grids, sorting, collision response, etc. In Lutz Latta's article

"Building a Million-Particle System", we reviewed stateless and state-preverserving particle systems on the GPU. Latta's paper also discussed particle data storage and a six-step algorithm for simulation and rendering particles. The six steps discussed were -

"1. Process birth and death
2. Update velocities
3. Update positions
4. Sort for alpha blending (optional)
5. Transfer texture data to vertex data
6. Render particles" (Latta, p.1)

Similarly, we first initialize our particles in the CPU-based C# file. The particles are then passed to the compute shader to update position and motion, and then to the vertex and pixel shaders to calculate and apply particle color. We also reviewed John Nickoll's and William J. Dally's 2004 article "The GPU Computing Era" to gain insight on CPU and GPU parallel processing. To implement Boids Flocking, the main text we reviewed was Craig Reynolds' "Flocks, Herds, and Schools: A Distributed Behavioral Model". Using Reynolds' paper, we were able to gain a thorough understanding of the flocking algorithm, specifically the three main steering behaviours-- separation, alignment, and cohesion.

Methods:

During first semester, we created a particle simulation program which uses the GPU to calculate the motion of the particles. The program is made up of four components: a C# script, a compute shader which calculates the motion of each particle, a vertex shader which calculates particle color, and a pixel shader which applies color to each particle. When a user uploads an image, the program generates particles based on the pixels of the image. For example, if the user selects a 4K image (2048 by 4096 pixels) then there will be around 8 million particles. Each particle moves independently and randomly based on a Simplex noise algorithm implemented in the compute shader. We experimented with a variety of motion patterns before implementing the Simplex Noise and plan on implementing numerous motion patterns from which the user can choose.

We then created a second version of the program which uses the CPU to calculate the motion of the particles. More specifically, the motion of each particle is calculated in the C# script before the particle data is passed to the shaders. The CPU version of the program was created so that a comparison between the conventional method (CPU calculated motion) and a GPU-accelerated version of the program could be made. The comparison was achieved by tracking the frame rate of the simulation achieved by each of the two implementations. Finally, although the original particle structure was a point, we converted to a billboard type so that the particles will face Unity's camera regardless of its position. This more advanced type not only looks more aesthetically pleasing, but also requires more computation and rendering time on the GPU.

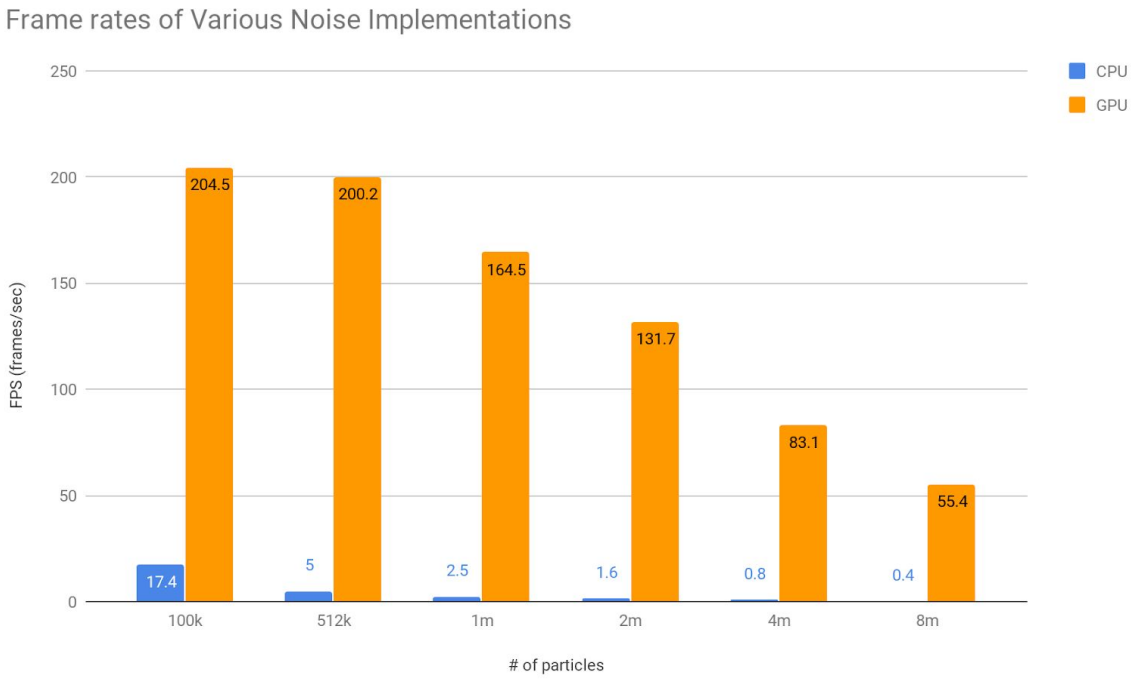
For the tests, an Alienware machine was used: Intel I7-6700k CPU with 32GB RAM and Nvidia Geforce GTX 1080 GPU. By allowing the program to run using varying image sizes from 100 thousand particles to 8 million particles on both the CPU and GPU, we collected and recorded the frames per second (FPS) over 30 seconds and calculated average FPS.

In second semester, we began work on an optimized GPGPU version of boids flocking. Using Daniel Shiffman's version of boids flocking[9], we made changes to Jiadong Chen's

GPGPU implementation of boids flocking[2] found on GitHub. Chen's version of the program calculates boid motion on the compute shader, which is already a marked improvement than calculating the separation, alignment, and cohesion of each boid on the CPU. However, Chen's version of boids flocking renders each boid as a GameObject in Unity using the CPU. Having identified this as the bottleneck point of the graphics rendering process, we now needed a GPU version of rendering the boids. We ended up comparing the CPU GameObject version with a Sprite version of the boids (using 2D sprites) as well as a Polygon object version of the boids (using 3D polygons). The GameObject version instantiated the boids into the scene from the CPU and then calculated their movements on the GPU; however, the overhead of tens of thousands of GameObjects adds up quickly while their data could simply be remembered by the GPU and all drawn at once.. Therefore, we expected the Polygon and Sprite versions to be much more successful in producing a smooth animation.

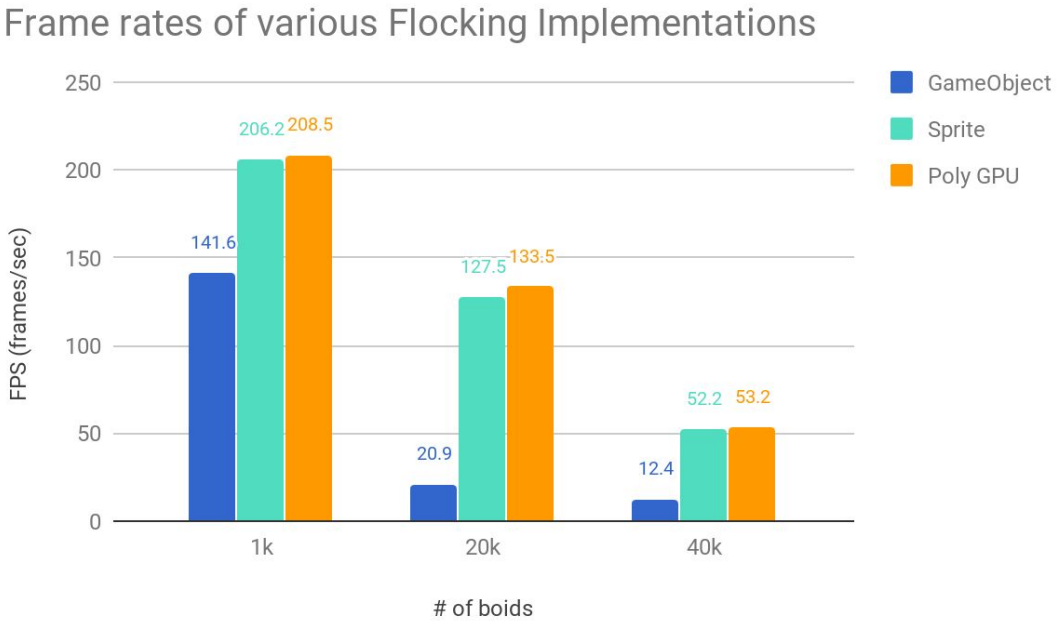
Results:

The graph below illustrates the difference in performance between CPU and GPU implementations of the particle simulation.



While the CPU can handle smaller number of particles at an acceptable frame rate (around 30 FPS), this implementation quickly became unbearable to watch when the number of particles was increased. The version of our program that uses a compute shader to change the particles' properties can maintain a smooth frame rate (>60 FPS) even with 8 million particles.

The graph below shows the difference in average frame rate over 30 seconds for the three versions of boids flocking.



The GPGPU rendering-only version far outperformed the CPU rendering version of boids flocking. Again, the CPU was able to render the boids at an acceptable frame rate for the smaller numbers of boids but was extremely slow as the number of boids increased. The sprite and polygon versions had similar frame rates although they were both much faster than the GameObject version. We speculate that this is due to the large number of calculations needed for all the versions.

Discussion/Conclusion:

From the results found by this study, it is clear that GPU-based computation drastically improves the performance of large-scale particle simulations. With most modern displays running at 60 frames, it is clear that the CPU needs more help with general computation by the GPU in order to create smooth simulations of thousands and even millions of particles. GPGPU programming can be used to accelerate all kinds of software that includes particles simulations, which can take great advantage of parallel computation. In terms of future work, we would like to create Unity plug-ins or packages so that others can utilize large scale particle systems in their own work. Additionally, we would like to work on optimizing the boids flocking further so that the calculations don't weigh down the frame rate as we observed during our tests.

Literature cited:

1. Carr, Nathan. Hegeman, Kyle. Miller, Gavin S.P.. 2006. Particle-Based Fluid Simulation on the GPU. https://link.springer.com/content/pdf/10.1007%2F11758549_35.pdf
2. Chen, Jiadong. October 24, 2017. Unity-Boids-Behavior-On-GPGPU. https://github.com/chenjd/Unity-Boids-Behavior-on-GPGPU
3. Direct to video. 2009. A thoroughly modern particle system. http://directtovideo.wordpress.com/2009/10/06/a-thoroughly-modern-particlesystem/
4. Falkenby, Jesper Hansson. 2014. Physically-based fluid-particle system using DirectCompute for use in real-time games. Belkinge Institute of Technology. http://www.diva-portal.org/smash/get/diva2:832945/FULLTEXT01.pdf
5. Fournier, Antoine. 2015. XParticle. https://github.com/antoinefournier/XParticle
6. Latta, Lutz. 2004. Building a Million Particle System. https://www.gamasutra.com/view/feature/130535/building_a_millionparticle_system.php
7. “MATLAB GPU Computing Support for NVIDIA CUDA-Enabled GPUs.” MATLAB & Simulink , www.mathworks.com/discovery/matlab-gpu.html.
8. “Photoshop Graphics Processor (GPU) Card FAQ.” Adobe , helpx.adobe.com/photoshop/kb/photoshop-cc-gpu-card-faq.html.
9. Reynolds, Craig. “Boids Background and Update.” *Reynolds Engineering and Design*, 29 June 1995, www.red3d.com/cwr/boids/.
10. Shiffman, Daniel. “Flocking.” Processing, processing.org/examples/flocking.html.