Massive Data-Parallel Swarm Simulation and Visualisation using CUDA

Philipp Lucas

Friedrich-Schiller-University Jena
Department of Mathematics and Computer Science
Ernst-Abbe-Platz 1-4, 07743 Jena, Germany
philipp.lucas.1@uni-jena.de

Abstract. We introduce a real-time swarm simulation and 3D visualisation software. It is designed as a playground for virtual experiments on swarm behaviour to analyse and understand its underlying rules. To this end, it provides a highly interactive interface that allows the user to modify many aspects of the simulation at runtime. Since particle simulations are classic problems of parallel computing, the software utilises CUDA to accelerate most parts of its computation. This way the massive power of Graphics Processing Units (GPUs) can be employed. The resulting speed up compared to a native CPU implementation varies depending on simulation settings, but usually is at least 50-fold. This paper gives an overview on capabilities, advantages and limitations of the software.

Key words: swarm simulation, particle simulation, CUDA, OpenGL, 3D graphics

1 Introduction

Membrane Computing emerged as a research area dealing with information processing in and by dynamics of spatial structures [3]. Especially, interactions between particles and delimiting elements are under consideration. Indeed, it has been successfully demonstrated that the resulting framework of membrane systems succeeded in capturing and modelling of numerous phenomena [2, 4]. Particle swarms (and swarm behaviour in general) can be seen as a special branch of membrane computing bridging to the field of amorphous computation. We introduce a widely configurable software able to describe and simulate the complex behaviour of particle swarms in a massively parallelised manner.

Our software was developed as a project during the lecture "Parallel Algorithms and CUDA" at Friedrich-Schiller-University Jena in winter 2009 [5]. CUDA is NVIDIAs parallel computing architecture that implements a more advanced version of SIMD called SIMT and thereby allows parallel general purpose computation on certain NVIDIA graphic cards.

The initial idea for the project was to employ CUDA for simulating natural movement and behaviour of a fish swarm. As a fish swarm consists of many individuals whose behaviour exclusively depend on its local context but does not

require any central organising unit, CUDA can be seen as an ideal candidate. We discretise time by calculating the state of a fish at time t_{i+1} as a function of the state of the whole swarm at time t_i. Since this function is equal for all fishes, the state of the whole swarm can easily be calculated in parallel.

The question arises what kind of rules are typical for fish swarms. In 1986 Craig Reynolds published a paper addressing this issue [6]. He proposed three simple rules: Firstly, *separation*, which lets a fish avoid crowding with its neighbours. Secondly, *alignment*, which lets a fish steer towards the average heading of its neighbours. And thirdly, *cohesion*, which makes a fish steer toward the average position of its neighbours. A neighbour of a fish is a fish that is currently located within its field of view, which can be described by a viewing direction, viewing angle and maximum viewing distance. Each of the rules results in a vector, which describes the desired movement. The weighted sum of these vectors along with the current position gives the new position of a fish.

Actually, swarm simulation is just like simulation of particle systems. In both cases, there are a certain number of different types of objects, each type following its own set of local rules. In the fish swarm scenario, we could for example add a predator. We just need to introduce new rules for the predator and the prey. Necessary modifications on the framework are minor. This feature makes the program highly flexible and useful for further applications.

2 Optimisations and Challenges

The most computationally expensive part of the simulation is to calculate the neighbourhood of a particle. Since there is no structural spatial information, except for positions and directions, from which neighbours could be derived, we need to check all particles against each other. That takes $O(n^2)$ time for n particles. By now, no optimisation other than the obvious parallelisation is implemented, however using k-dimensional trees (kd-trees) or more elaborate data structures like a Bkd-Tree [1] we should be able to induce a significant speed-up. We plan to integrate this approach in further versions of our software.

One of the main challenges to achieve high speed in the simulation was good memory management on both, the GPU device and the host system. This is essential for several reasons:

The CUDA architecture provides various types of GPU memory, each having advantages and disadvantages mainly regarding its size, speed and accessibility. We use the GPU's shared memory as a buffer to minimize global memory access, since one read operation from global memory makes data available to many particles.

Furthermore, in particular the amount of registers needed by a single particle for its calculations has significant influence on the overall performance. This is due to the limited amount of registers per processor on the GPU. If we need too much memory for one particle, there is not enough capacity left to keep all sub-processors of that processor busy. Hence it was most important to optimise register usage, even at the expense of extra computation.

Additionally, CPU and GPU have physically different memories, which turns the necessary communication between both into a bottleneck for many applications. Actually for this simulation we would only need little communication, since data on the GPU, i.e. positions and directions of all particles, are exclusively needed for the next simulation step and the visualisation, which is naturally done on the GPU as well. However, we employ OpenGL and by now there is unfortunately no interface to pass that kind of data from CUDA to OpenGL. Therefore we decided to minimize communication between CPU and GPU, i.e. after each time step only the updated state of particles is sent to the host, but no data transfer from host to device is needed.

3 Features and Limitations

The program combines simulation of many-particle systems with 3D-visualisation and an interface to change most parameters in real-time.

At any time it is possible to pause the simulation, add or remove particles, change the viewers position, change the graphical model of particles, alter parameters and save the current set of simulation parameters to a file or load another set from a file.

From a technical point of view the number of particles is hardly limited. At the moment one particle takes 36 Bytes of global memory on the GPU, but even an older graphic card that supports CUDA provides at least 256 MB memory. That allows more than 7 million particles without *any* changes at the implementation. However, as it was discussed before and as you can see in diagram 1 runtime increases quadratic with number of particles. It is also possible to disable visualisation, which significantly increases simulation speed for smaller numbers of particles.

For the simulation of a fish swarm the rules are rather simple and accordingly we need few registers per particle. However, for more complex rules much more registers might be needed, which will have a significant negative effect on simulation speed as discussed in section 2.

Currently there is no meta-language or GUI to create sets of rules. They must be implemented in the source code, requiring some understanding of C++ and CUDA. However, due to the modular structure which separates GUI, basic framework and simulation, various rules can be specified and selectively activated at runtime.

4 Results

A series of screenshots in figure 2 should demonstrate the various results that can be achieved by altering parameters of a single rule set.

Acknowledgments. I would like to thank Waqar Saleem and Jens Müller for our many discussions, as well as for their useful suggestions and continuous help.

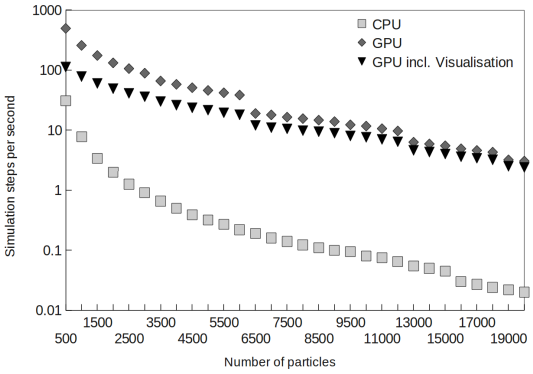

Fig. 1: This figure shows the simulation performance of our software depending on the calculation device. (*CPU*) and (*GPU*) exclude the time needed for rendering all particles with OpenGL. *GPU* also does not include costs for memory copying from the GPU to the host. Refer to (*GPU incl. Visualisation*) for the overall performance.

(a) Explosion (b) Thin torus (c) Large swarm (d) Various streams

Fig. 2: Results from varying rule parameters

Moreover, I very much appreciate the support of Thomas Hinze for making this abstract possible.

References

1. Agarwal, P., Arge L., Procopiuc, O., Vitter J.: Bkd-tree: A Dynamic Scalable kd-tree, In: Proceedings of International Symposium on Spatial and Temporal Databases (2003)
2. Cecilia, J.M., Garcia, J.M., Guerrero, G.D., Martinez-del-Amor, M.A., Perez-Hurtado, I., Perez-Jimenez, M.J.: Simulation of P systems with active membranes on CUDA. In: Briefings in Bioinformatics 64, pp. 1–10 (2009)
3. Ciobanu, G., Perez-Jimenez, M., Paun, G.: Applications of Membrane Computing. Springer, Berlin (2006)
4. Diaz-Pernil, D., Perez-Hurtado, I., Perez-Jimenez, M.J., Riscos-Nunez, A.: A P-Lingua Programming Environment for Membrane Computing. In: LNCS 5391, pp. 187–203 (2009)
5. FSU Jena, Department of Mathematics and Computer Science: Programming with CUDA, `http://theinf2.informatik.uni-jena.de/For_Students-p-9/Lectures/Programming_with_CUDA-p-41/WS_2009_2010.html#projects`
6. Reynolds, C. W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. In: Computer Graphics, 21(4), ACM SIGGRAPH '87 Conference Proc., pp.25–34, Anaheim, California (1987)