Pathfinding Using Swarm AI

1st Yuanhan Zhang
Courant Institute
New York University
New York, U.S.A.
yz4179@nyu.edu

2nd Michael Urciuoli
Courant Institute
New York University
New York, U.S.A.
mlu216@nyu.edu

3rd Tianpei Yang
Courant Institute
New York University
New York, U.S.A.
ty1195@nyu.edu

***Abstract*—We develop a Swarm AI simulator from scratch and implement a Particle Swarm Optimization technique to find paths between two positions in 2D space. After writing a sequential version, which we nickname N^2, we then parallelize that version using CUDA to be run on GPUs. We then further optimize the sequential simulator using Coarse Grid Search, and make an attempt to parallelize that version as well.**

I. Introduction

Swarm intelligence is an exciting field of research that has gained popularity due to its ability to solve nonlinear problems in a computationally tractable way [1]. Many of these algorithms take inspiration from systems found in nature. A main theme within these algorithms is that they involve a collection of agents. Each of these agents perform specific actions, and as a whole a population of agents display a specific collective behavior. This framework can be used to solve many real world tasks. We developed a form of Particle Swarm Optimization that is designed to find a path between two or more objectives, using multi- random tree search.

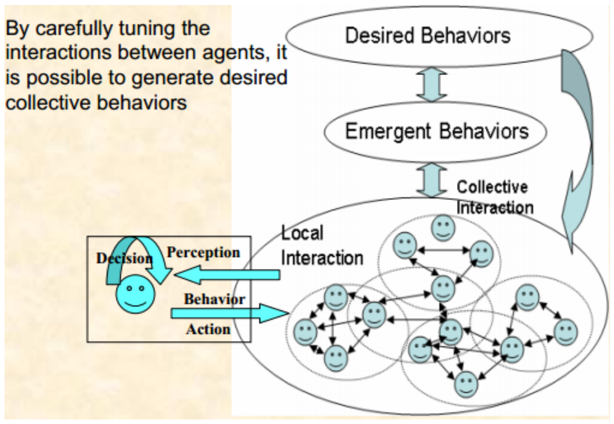
By carefully tuning the interactions between agents, it is possible to generate desired collective behaviors
Desired Behaviors
Emergent Behaviors
Collective Interaction
Local Interaction
Decision
Perception
Behavior
Action

Fig. 1. Inspiration for our Swarm AI [2]

The above figure was used to guide development of our algorithm. As it states, we only modified the Perception-Decision-Behavior/Action loop at the individual agent level in order to achieve pathfinding between objectives.

II. Survey

Nowadays, Swarm Artificial Intelligence or Multi-agent systems has become a popular research field. Swarm AI is made up of multiple artificial agents. The agents can exchange heuristic information and collaborate with other agents to achieve some specific goals. So it is widely used in unmanned aerial vehicles (UAVs) swarm control [3], delivery sorting robots collaboration, ant-based routing algorithm [4], etc.

However, as the amount of data increases and the number of agents increases, the parallelism of the swarm algorithm becomes more and more important. With the help of GPU parallelization, the efficiency of applications in many fields have multiplied, such as financial [5], CNN [6] and sparse direct factorization [7]. In this paper, we mainly focus on how to use GPU to parallel the code of multi-agent systems. Many studies in this field have achieved remarkable results. Guoheng Luo implemented GPU-parallel bees algorithm which is 13 times faster than the CPU version [8]. Alberto Cano used GPU-powered ant programming to handle and classify high dimensional data sets [9]. Wojciech Bura paralleled an ant-based vehicle navigation algorithm which performed well in the experiments with real world data [10].

III. Problem Definition

Inspired by the research above, we determined to develop a swarm robots system to solve real world path finding problems between objects. We will implement a CPU version, parallel it in a GPU version and then do large scale experiments to investigate how GPU improves the performance of swarm algorithm.

IV. Proposed Idea

We developed a simulator, in which a swarm of agents display individual behavior per time step , which is defined by the following few steps:

A. Initialization (only called once after the program begins)

- Put all entities and objectives randomly located inside the given size playground.
- Check whether there are overlaps among all entities and objectives, mark those with collision detected.
- Rearrange those marked entities and objectives.
- Repeat the above 2 steps until every entities are initialized properly (a retry times limit set to 100, saying if there still exists overlaps after 100 retries we simply assume that too much, too big entities or too small playground is given).

B. Sense (Perception)

- Is a non-linked agent near (within a "sensing distance") of an objective?

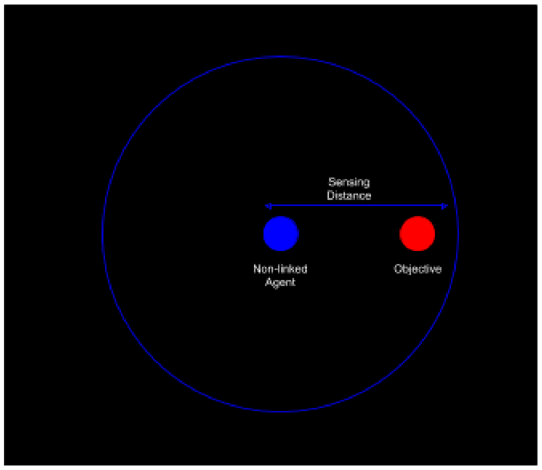

Fig. 2. sensed a non-linked agent

- If not, then is the agent near another agent who is linked to an objective?
  - An agent is linked to an objective either directly, or though a chain of linked agents that ends at an objective.

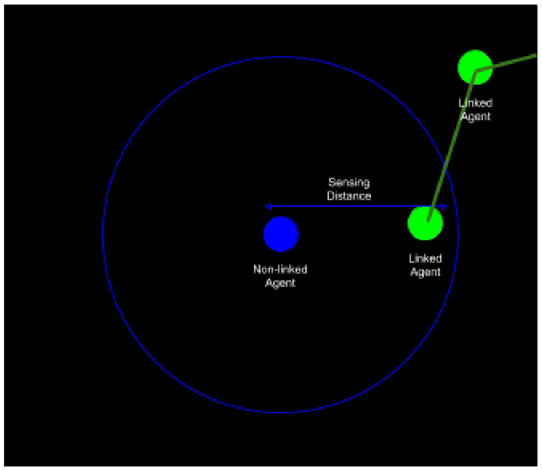

Fig. 3. sensed a linked agent

C. Decide

- If a non-linked agent is near an objective, then it will link to that objective. By linking to the objective, the agent will not move and will allow another agent to link to it.
- If a non-linked agent is near another linked agent, it will link with that agent IFF that agent is only linked to one other agent/objective (it is a leaf within a tree structure).
  - Linking limits do not apply to branch agents, who are a set number of links away from either the objective or another branch agent.

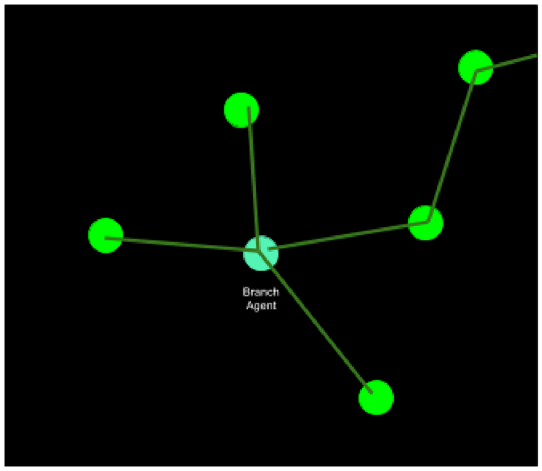

Fig. 4. deciding to link

- If a non-linked agent can link to two different linked agents, and each of those two linked agents are part of different chains (that are rooted at different objectives), then a path is formed.

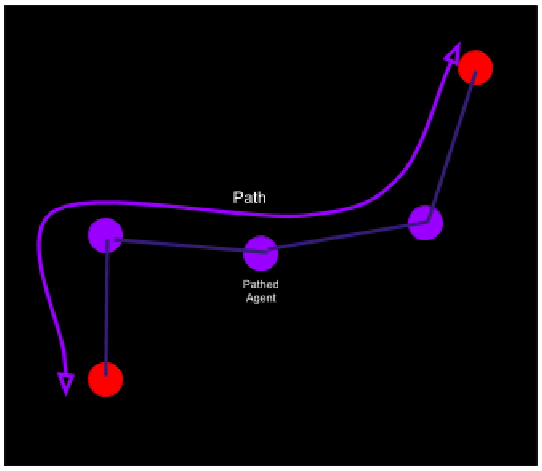

Fig. 5. forming a path

D. Move(Behavior/Action)

- Linked/Pathed agents do not move
- Non-linked agents move at predetermined velocities throughout the simulation (redirecting if they will exit the window). Moving agents will turn around if they detect an oncoming collision.

V. GPU Parallelization

We used CUDA to parallel our Swarm robots path finding algorithm on GPU. CUDA (Computer unified device architecture) is a parallel computing platform and programming model published by Nvidia for computing on GPUs. CUDA enables us to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation. In CUDA, we assign each thread with the computation of each robot. Multiple threads consist of a block which can be executed in streaming multiprocessors. And blocks are organized into grids, the top level hierarchy.

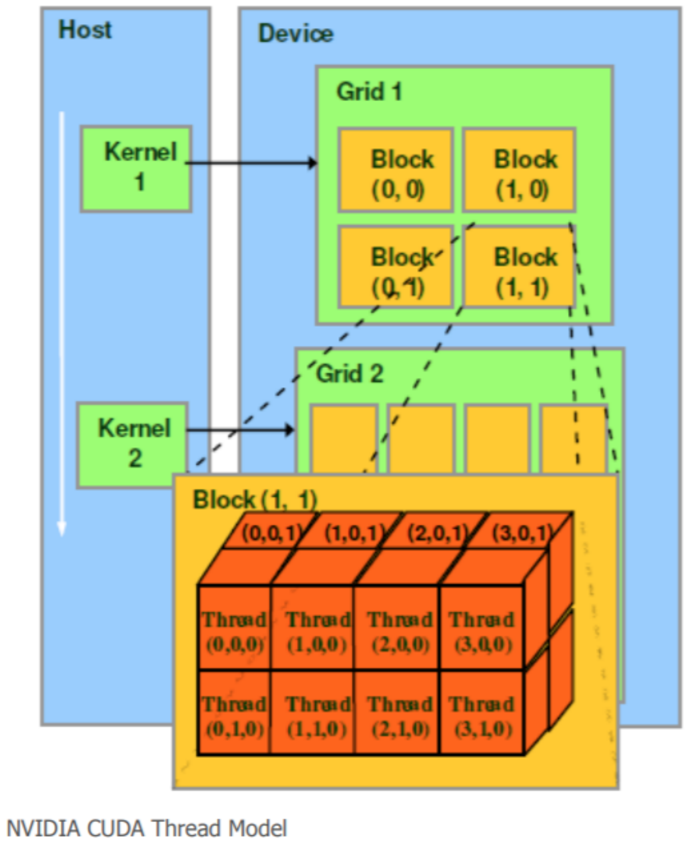

Fig. 6. CUDA Model [13]

After carefully analyzing our algorithm, we find that the "for" loops in Initialization, Sense, Decide, Adjust and Move function are extremely time consuming. Each robot's state judging, movement is executed sequentially. So for each "for" loop, we create a kernel function. There are several key components of our implementation of GPU version:

A. Index

We arrange robots into a 1d vector. So 1D grid of 1D blocks layout is suitable for our application. In each kernel function, we get the index of each robot by index = blockDim.x * blockIdx.x + threadIdx.x.

B. Data Structure

In the sequential version, the robots swarm are implemented by population class. And robots are organized in a vector. Each robot is developed by individual class. However, std::vector is not supported in CUDA. So we need to reconstruct the data structure. In our parallel version, we directly define thrust::device_vector to store position x, position y, velocity x, velocity y, ect. in population class. Our kernel functions use index to directly access and manipulate data on these new data structures.

Also, we redefine the data structure used for linking entities. In the sequential version, it is a tree-like structure which has lots of pointers in each (root, previous and children). However, since we are not allowed to allocate memory on device code which gives a huge inconvenience and pointers to pointers are always error prone, we rewrite this data structure into a struct with only built-in types. Every pointer is switched to an unsigned int identifier instead, lookup is done by indexing from vectors (each field originally storing pointers is changed to a vector) mapping one by one. Though this leads to some memory waste in specific cases, it is much easier for programmers to develop and troubleshoot.

C. Efficiency Concerns

Besides using memory coalescing in one dimensional vectors, we allocated those vectors as fixed size during initialization to avoid memory moving (CPP vector implementation). In other words, we only use the thrust library vector for easier programming, what working behind the scene is the same as pointers living on device memory.

We use CUDA streams to fully utilize all devices parallelizing CUDA kernels. For instance, because entities and objectives have different radius and behaviors, in collision detection (also Sense), we have to check collision (Sense) between entities, between objectives, between entities and objectives. Those mentioned are implemented as three different kernels which are able to work independently (imagine that checking collisions between entities has no relationship with checking collisions between objectives). Under this circumstance, CUDA streams are used to parallelize those kernels.

VI. Step by Step Improvement

As mentioned above, our robots swarm has class methods: Initialization, Sense, Decide, Adjust and Move. They are all developed in our GPU version. In the beginning of these methods, we use thrust::raw_pointer_cast to get the pointer of the data structure that we need. Then determine blocks per grid and threads per block and launch the kernel. In the kernel function, we get the index and do the same corresponding calculation. Eventually, we use cudaDeviceSynchronize to make sure every robot is updated before leaving the method.

After developing an O(N^2) version of our simulator, we tried the following steps in order to parallelize the application:

A. Move

As a beginning, we are thinking about parallelizing some dense calculations, so making Move in parallel first comes to mind. However, instead of seeing a performance improvement, a drastic degrade is observed.

B. Initialization and Collision Detection

In order to find a way to improve our speed, not only we use nvprof to profile our Move Parallelized version, but also we put probes in our program as profilers. By nvprof, we learn that the Move Parallelized is extremely slow because of Memcpy back and forth between host and device (because we have to move device memory back to complete other sequential calculations). By profiling the execution time of all parts in our sequential CPU version, we found Initialization and Adjust are extremely time consuming, both call the collision detection unit. So as a consequence, we rewrite collision detection in CUDA kernels. A faster Initialization worked as expected, while the whole execution remains nearly the same.

C. *Entire Simulation on Device*

Because the problem caused by Memcpy still exists, we are trying to get rid of working on host memory at this point. By parallelizing other core functions including Sense and Decide, we avoid Memcpy from device back to host. This finally gained a speedup over the O(N^2) version.

D. *Coarse Grid*

Upon research of Coarse Grid Search [11], we further optimized the CPU version of our simulation. By splitting the simulation space into the grid and maintaining each agent's location within the grid as they move, we drastically reduce the number of comparisons that need to be made from between every agent to only between agents that are within near proximity to each other. This results in a drastic speed up, comparable to our O(N^2) GPU version. We gain additional speed up by just implementing a 3D kernel for collision computation using this coarse grid (1D for all agents, and then 2D for each of the 9 grid cells surrounding each agent). However, it still has many bugs and is unable to compute the simulation properly. With more time, we believe we could fix and parallelize more of this version, and the speed up gain would be considerate.

VII. Experimental Setup

By default, our simulation logs each rendering action to an outputted text file. This is done so that it can run on the NYU CIMS server. We then use the freeglut [12] open source library to render our simulation environment when running on a stand-along laptop. Otherwise we have developed this application from scratching using C++ and CUDA. The number of objectives, agents, their sizes, and overall simulation space size is all configurable. Objectives and agents start randomly placed, with no initial overlap.

VIII. Challenges

A. *Rendering Graphics on CIMS*

Our first major challenge for this project came in the form of graphics rendering on CIMS cuda servers. We assumed that we would be able to forward graphics using X11 forwarding, however this proved to be unsuccessful. Several attempts were made to work around this problem, and what we ended up doing was logging all renderings to a .txt file to be rendered on a laptop running Linux.

B. *Simulation Correctness*

As we were developing our simulation from scratch, a considerable amount of effort was put into even writing the sequential version. We acknowledge that there are still bugs within our simulation, especially in the GPU Coarse Grid version, and given more time we would continue to improve the stability of the application.

C. *Kernel Parameter Tuning*

Due to time constraints, and with much focus going to correctness, we were unable to properly tune block and grid sizes of our CUDA kernels. This is another area of improvement that we would develop in the future.

D. *Memory Bottleneck*

One significant observation is that when parallelizing massive computations, reducing memory bottleneck is sometimes more important than only taking into consideration parallelizing dense calculations. In our research, only parallelizing the time consuming calculations leads to no performance improving because memory copies (also we can not use constant memory or shared memory as optimizations in our cases) cannot be avoided. Instead, after parallelizing everything including those trivial parts, we avoid memory copy between host and devices, as a result, the speed is increased dramatically. In other words, only if the calculation parts are mostly unrelated to each other, parallelizing parts of the calculations gains speed up, otherwise memory operations would often be bottlenecks.

IX. Experiments and Analysis

As we mentioned before, in total we have 4 versions which are compiled separately (different Makefiles or building scripts). They are the CPU O(N^2), CPU Coarse Grid Algorithm, GPU O(N^2) and GPU Coarse Grid Algorithm. We fix the objective number as 2 in our experiments while varying the number of entities and size of playground.

The parameters pairs (first stands for number of entities and second stands for size of space, notice that it is the side length instead of area for a square playground) are (100, 500), (100, 2000), (100, 5000), (100, 20000), (1000, 2000), (1000, 5000), (1000, 20000), (10000, 5000), (10000, 20000), (100000, 20000).

In order to have an intuitive view of running time among all those versions with the same configuration, we develop a script for auto-testing which runs each parameters pair 3 times, logging (detailed profiling data is shown in the figure below) all the above experiments in parallel, then we compare the update numbers between them, finally we visualize the results from those log files.

To make verify our results, we also test them on different CUDA devices. We find that even if the stats vary, the relationship patterns among the different versions are quite similar and stable.

The entire data set gathered from our experiments can be viewed on our Google Sheets page: https://bit.ly/34ga7sH

Performance of Each Version
(average update time (ms) across simulation space sizes)

# Entities	CPU N^2	GPU N^2	CPU Coarse	GPU Coarse
100	7.64	1.10	0.44	0.77
1000	925.48	2.78	4.45	4.27
10000	294561.59	50.29	66.29	35.94
100000	-	-	3161.07	1043.41

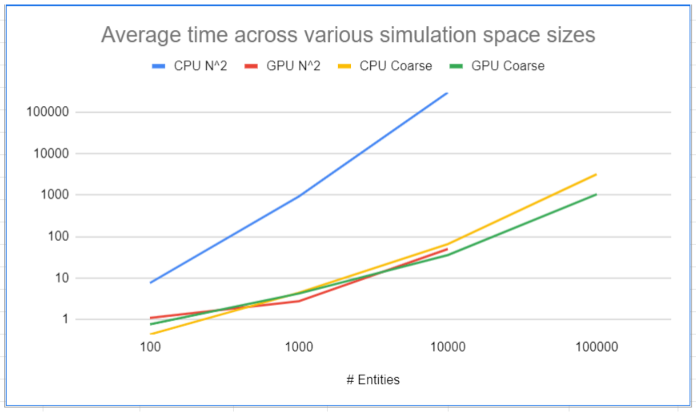

Fig. 7. Average Time

X. Conclusions

In general, the CPU O(N^2) is the slowest overall, the GPU Coarse Grid is slightly faster than GPU O(N^2) while both of them are faster than the CPU Coarse Grid in certain cases. This is saying the Swarm Robotics is a GPU friendly problem and can be highly parallelized to gain acceleration.

References

[1] Yang, Xin-She, and Mehmet Karamanoglu. ”Swarm intelligence and bio-inspired computation: an overview.” Swarm intelligence and bio-inspired computation. Elsevier, 2013. 3-23.
[2] Swarm Intelligence. [Online]. Available: http://www.techferry.com/articles/swarm-intelligence.html. [Accessed: 10-Dec-2020].
[3] Tahir, Anam, et al. ”Swarms of unmanned aerial vehicles—A survey.” Journal of Industrial Information Integration 16 (2019): 100106.
[4] Sharvani, G. S., N. K. Cauvery, and T. M. Rangaswamy. ”Different types of swarm intelligence algorithm for routing.” 2009 International Conference on Advances in Recent Technologies in Communication and Computing. IEEE, 2009.
[5] Grauer-Gray, Scott, et al. ”Accelerating financial applications on the GPU.” Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. 2013.
[6] Rumi, Masuma Akter, et al. ”Accelerating Sparse CNN Inference on GPUs with Performance-Aware Weight Pruning.” Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. 2020.
[7] Rennich, Steven C., Darko Stosic, and Timothy A. Davis. ”Accelerating sparse Cholesky factorization on GPUs.” Parallel Computing 59 (2016): 140-150.
[8] Luo, Guo-Heng, et al. ”A parallel Bees Algorithm implementation on GPU.” Journal of Systems Architecture 60.3 (2014): 271-279.
[9] Cano, Alberto, Juan Luis Olmo, and Sebastián Ventura. ”Parallel multi-objective ant programming for classification using GPUs.” Journal of Parallel and Distributed Computing 73.6 (2013): 713-728.
[10] Bura, Wojciech, and Mariusz Boryczka. ”The parallel ant vehicle navigation system with CUDA technology.” International Conference on Computational Collective Intelligence. Springer, Berlin, Heidelberg, 2011.
[11] C. Ericson, Real-time collision detection. Amsterdam: Elsevier, 2004.
[12] The freeglut Project :: About. [Online]. Available: http://freeglut.sourceforge.net/. [Accessed: 10-Dec-2020].
[13] Introduction to GPUs. [Online]. Available: https://nyu-cds.github.io/python-gpu/02-cuda/ [Accessed: 10-Dec-2020].