GPU Environmental Delegation of Agent Perceptions: Application to Reynolds's Boids

Emmanuel Hermellin, Fabien Michel

LIRMM - CNRS - University of Montpellier, 161 rue Ada, 34095 Montpellier, France
{emmanuel.hermellin, fmichel}@lirmm.fr,

Abstract. Using Multi-Agent Based Simulation (MABS), computing resources requirements often limit the extent to which a model could be experimented with. Regarding this issue, some research works propose to use the General-Purpose Computing on Graphics Processing Units (GPGPU) technology. GPGPU allows to use the massively parallel architectures of graphic cards to perform general-purpose computing with huge speedups. Still, GPGPU requires the underlying program to be compliant with the specific architecture of GPU devices, which is very constraining. Especially, it turns out that doing MABS using GPGPU is very challenging because converting Agent Based Models (ABM) accordingly is a very difficult task. In this context, the *GPU Environmental Delegation of Agent Perceptions* principle has been proposed to ease the use of GPGPU for MABS. This principle consists in making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU. For now, this principle has shown good results, but only on one single case study. In this paper, we further trial this principle by testing its feasibility and genericness on a classic ABM, namely Reynolds's boids. To this end, we first review existing boids implementations and propose our own benchmark model. The paper then shows that applying GPU delegation not only speeds up boids simulations but also produces an ABM which is easy to understand, thanks to a clear separation of concerns.

Keywords: Multi-Agent Based Simulation, Flocking, GPGPU, CUDA

1 Introduction

Because Multi-Agent Based Simulation (MABS) can be composed of many interacting entities, studying their properties using digital simulation usually requires a lot of computing resources. To deal with this issue, the use of General-Purpose computing on Graphics Processing Units (GPGPU) can drastically speed up simulation runs for a cheap cost [8]. GPGPU relies on using the massively parallel architectures of usual graphic cards to perform general-purpose computing. However, this technology implies a very specific programming approach and requires advanced GPU (Graphics Processing Unit) programming skills [11].

Because there are many different MAS (Multi-Agent Systems) models, there is no generic way for implementing MAS using GPGPU. So, it is very difficult to adapt an Agent Based Model (ABM) so that it can be run on the GPU. Considering this issue, hybrid systems represent an attractive solution. Because the execution of the MAS is shared between the Central Processing Unit (CPU) and the GPU, it is thus possible to select only what is going to be translated and executed by the graphics card.

In this paper, we propose to challenge the feasibility and interest of the *GPU Environmental Delegation of Agent Perceptions* (for short *GPU Delegation*) principle which is based on an hybrid approach. This principle consists in delegating to the environment some computations made by the agents. A case study is presented in [9] and shows good results in terms of performances, accessibility and reusability. We propose to trial this principle by using it on a classic ABM, namely Reynolds's boids [15].

In Section 2 we review how Reynolds's Boids is implemented in several MABS platforms and then propose our own flocking model in Section 3. In Section 4, we present the *GPU Environmental Delegation of Agent Perceptions* principle. In Section 5, we describe the implementation of our model and how we applied *GPU Delegation*. In Section 6, we present and discuss the results of our tests. Finally, we conclude and present perspectives in Section 7.

2 Reynolds's Boids

2.1 Original Model Overview

Reynolds wanted to achieve a believable animation of a flock of artificial birds, namely boids [15]. He remarked that it was not possible to use a scripted flock motion to achieve a realistic animation. Reynolds's idea was that boids have to be influenced by the others to flock in a coherent manner: "*Boid behavior is dependant not only on internal state but also on external state.*" [15].

Reynolds proposes that each agent of the model is subjected to forces that make it move by taking into account the interactions with the others. So, each entity has to follow **three behavior rules**:

- **R.1** Collision Avoidance: Avoid collisions with nearby flockmates
- **R.2** Flock Centering: Attempt to stay close to nearby flockmates
- **R.3** Velocity matching: Attempt to match velocity with nearby flockmates

Today, Reynolds's Boids is recognized as one of the most representative agent-based model. So, many agent-based platforms integrate their own boids model.

2.2 Boids in Current MABS Platforms

In this section, we compare several available implementations of boids models. Among the related works found, we only introduce models we were able to download and try with an open source code: NetLogo, StarLogo, Gama, Mason and Flame GPU. For each model, we describe how the three rules are implemented (Collision Avoidance (R.1), Flock Centering (R.2), Velocity matching(R.3)).

NetLogo In NetLogo[1] [19], all the agents (called Turtle in the Logo language) move and try to get closer to their peers. If the distance between them and the nearest neighbor is too small, the agent tries to get away (avoid collision (R.1)), otherwise the agent aligns with its neighbors (R.2). However, there is no speed management (R.3): All the agents have the same velocity during the entire simulation.

StarLogo In StarLogo[2] [14], the agent searches for his closest neighbor. If the distance between him and his peer is too small, it turns and gets away (to avoid collision (R.1)). Otherwise, it moves toward him and use his direction. The search for cohesion (R.2) is not explicitly expressed and the velocity of the agents is fixed throughout the simulation (R.3).

Gama In Gama[3] [3], agents are looking first for a target (similar to a goal) to follow. Once agents have a target, they move according to three functions that implement Reynolds's rules: A separation function to avoid collision (R.1), a cohesion function (R.2) and an alignment function for speed and direction (R.3). The model differs from Reynolds's because the agents need a target to actually make the flocking.

MasOn MasOn[4] [7] uses the computation of several vectors to integrate R.1 and R.2. Each agent computes a motion vector composed of an avoidance vector (this is computed as the sum, over all neighbors, of a vector to get away from the neighbors (R.1)), a cohesion vector (this is computed as the sum, over all live neighbors, of a vector towards the "center of mass" of nearby flockers), a momentum vector (a vector in the direction the flocker went last time), a coherence vector (this is computed as the sum, over all live neighbors, of the direction of other flockers are going (R.2)), and a random vector. The speed is not managed in this model (R.3).

Flame GPU Flame GPU[5] [17] is the only GPGPU implementation that we were able to test. In this model, R.1 R.2 and R.3 are implemented into three independent functions. The special feature of this framework is the necessity to adopt a design formalism using XMML (based on XML) and C to hide the GPGPU from the user. Due to the use of these two programming languages, the implementation is not intuitive.

Summary Table 1 summarizes the implementations of Reynolds's rules, sets out the main features of the models and gives performance informations.

1 https://ccl.northwestern.edu/netlogo/
2 http://education.mit.edu/starlogo/
3 https://code.google.com/p/gama-platform/
4 http://cs.gmu.edu/~eclab/projects/mason/
5 http://www.flamegpu.com/

Performances We evaluate for every model the average computation time in milliseconds for an iteration. The purpose of this evaluation is to give an idea of the possibilities of every implementation. So, we use as common parameter an environment of 512 by 512 containing 4000 agents. Our test machine is composed of an Intel i7-4770 processor (Haswell generation, 3.40 GHz) and an Nvidia K4000 graphics card (768 CUDA cores).

It is necessary to note that for StarLogo, we observed a computation time higher than a second from 400 simulated agents. The performances being very below the other platforms, we did not push the tests farther. Finally, for Flame GPU, it was not possible to modify the number of agents in the simulation which is of 2048.

Table 1. Boids in common MABS platforms

Platform	**Compliance with Reynolds's Model**			**Main characteristics**	**Performances**
	Collision R.1	Cohesion R.2	Velocity R.3		
NetLogo	X	X		R.3 is not implemented: Velocity is fixed throughout the simulation	214 ms (CPU / Logo)
StarLogo	X			A minimalist implementation of behavior rules (only the *collision avoidance* is implemented)	*1000 ms (CPU / Logo)
Gama	X	X	X	Flocking behaviour when agents have a target to follow	375 ms (CPU / GAML)
MasOn	X	X		The rules R.1 and R.2 are reinterpreted into a global vector with addition of random components, no speed management	45 ms (CPU / Java)
Flame GPU	X	X	X	The three rules are explicitly implemented	*82ms (GPU / C,XML)

3 Reynolds's Boids: Our Model and Implementation

Of the previous study, we notice disparities between the various presented models. Indeed, the flocking rules proposed by Reynolds allow a big variety of interpretations. So, we notice that the speed adaptation rule (R.3) is least taken into account in comparison with R.1 and R.2 implemented in every model seen (except StarLogo). However, when R.3 is implemented, the collective behavior becomes much more convincing and the global movement possesses then a dynamics and a more interesting fluidity. Also, in some works, the behavior of alignment and cohesion is merged. The models clarifying the difference between this two behavior offer more interesting movements.

The model that we propose will take into account the interesting points observed previously. We indeed noticed that when the three rules are integrated, the dynamics and the movement of the agents are more interesting. So, our model will integrate R.1, R.2 and R.3 and will follow the KISS (Keep It Simple and Stupid) principle with the aim of creating a minimalist version (with the least parameters possible) focusing on the speed and the orientation of the agent[6].

Each entity has a global behavior which consists in moving while adapting its speed and direction. To this end, the proximity with the other agents is tested and the different Reynolds's rules are activated according to the distance found. More exactly, every agent looks in its vicinity. If no agent is present, it continues to move in the same direction. Otherwise, the agent checks if the neighbors are not too close. Depending on the proximity between entities, agents separate (R.1), align with other entities or create cohesion (R.2). Then agents adapt their speed (R.3), move and restart the process. Figure 1 summarizes the global behavior process.

In our model, we have two types of parameters: 5 constants for the model and 3 attributes specific to each agent. The constants are the following ones:

- *fieldOfView* (agent's field of view);
- *minimalSeparationDistance* (minimum distance between agents);
- *cohesionThreshold* (necessary number of agents to begin cohesion);
- *maximumSpeed* (maximum speed of the agent);
- *maximumRotation* (maximum angle of rotation).

The attributes specific to each agent are the following ones:

- *heading* (agent's heading);
- *velocity* (agent's speed);
- *nearestNeighborsList* (the list containing nearest neighbors).

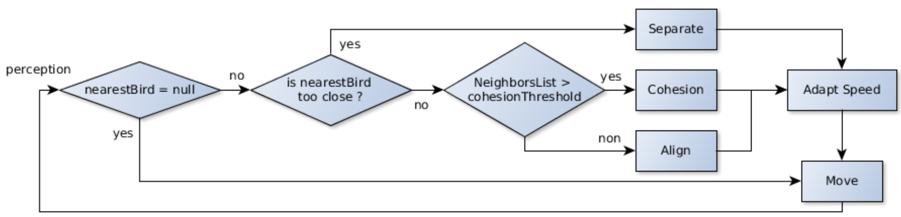

Fig. 1. Flocking: Global behavior process

Separation Behavior R.1 When an agent is too close from an other one, it separates (R.1). This behavior consists in retrieving the heading of both agents. If these two directions lead to a collision, agent rotates to avoid its neighbor (see Algorithm 1).

[6] The orientation is an angle in degree (between 0 and 360) which gives the heading of the agent according to the landmark fixed in the environment

Algorithm 1: Separate behavior

input : *myHeading*, *nearestBird*, *maximumRotation*
output: *myHeading* (the new heading)

```
collisionHeading ← headingToward(nearestBird) ;
if myHeading inTheInterval(collisionHeading, maximumRotation) then
  | changeHeading(myHeading);
end
return myHeading
```

Align Behavior R.2 When an agent comes closer to other entities, it tries to align itself with them, by adjusting his direction according to its nearest neighbor (see Algorithm 2).

Algorithm 2: Alignment behavior

input : *myHeading*, *nearestBird*
output: *myHeading* (the new heading)

```
nearestBirdHeading ← getHeading(nearestBird) ;
if myHeading isClose(nearestBirdHeading) then
  | adaptHeading(myHeading);
end
else
  | adaptHeading(myHeading, maximumRotation);
end
return myHeading
```

Cohesion Behaviors R.2 When multiple agents are quite close to each other without having to separate, they have a cohesion behavior. Each agent retrieves the directions of its neighbors and adjusts its own direction based on the average direction found, thus strengthening the cohesion of the group (see Algorithm 3).

Speed Adaptation R.3 Before moving, the agents adapt their speed (R.3). During all the simulation, every agent modifies its speed according to that of its neighbors. If the agent has just executed the behavior of separation (R.1), it accelerates to get free more quickly. Otherwise, the agent adjusts its speed to make it correspond to that of its neighbors (in the limit authorized by the *maximumSpeed* constant).

Testing Our Model We have put online a set of videos which show our model in action[7]. On this page are also available the source codes of the mentioned models and the necessary resources to test our solution.

Algorithm 3: Cohesion behavior

```
input  : myHeading, nearestNeighborsList
output: myHeading (the new heading)
sumOfHeading, neighborsAverageHeading = 0 ;
foreach bird in nearestNeighborsList do
    sumOfHeading+ = getHeading(bird);
end
neighborsAverageHeading =
sumOfHeading/sizeOf(nearestNeighborsList) ;
if myHeading isClose(neighborsAverageHeading) then
    adaptHeading(myHeading);
end
else
    adaptHeading(myHeading, maximumRotation);
end
return myHeading
```

4 GPU Environmental Delegation of Agent Perceptions

4.1 MABS and GPGPU

About GPGPU To understand the principle of programming associated to GPGPU, it is necessary to have in mind that it is strongly connected to the material architecture of the GPU. One of the main differences between a CPU and a GPU is the number of processing cores which is far more important for the GPU case.

So, GPU are now composed of hundreds, or even thousands of processing core (grouped into Streaming Multiprocessors, SM) forming a highly parallel structure able to perform more varied computing. GPGPU relies on using the SIMD (Single Instruction, Multiple Data) parallel model. Also called *stream processing*, the underlying programming approach consists in performing the same operation on multiple data points simultaneously. In other words, GPGPU relies on the simultaneous execution of a series of computations (*kernels*) on a data set (the flow - *stream*).

The programming models rely on the following philosophy: The CPU is called the *host* and plays the role of scheduler. The *host* manages data and triggers

[7] www.lirmm.fr/∼hermellin/Website/Reynolds_Boids_With_TurtleKit.html

kernels, which are functions specifically designed to be executed by the GPU, which is called the *device*. The GPU part of the code really differs from sequential code and has to fit the underlying hardware architecture. More precisely, the GPU device is programmed to proceed the parallel execution of the same procedure, the *kernel*, by means of numerous *threads*. These *threads* are organized in *blocks* (the parameters *blockDim.x*, *blockDim.y* characterize the size of these blocks), which are themselves structured in a global grid of blocks. Each *thread* has unique 3D coordinates (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*) that specifies its location within a *block*. Similarly, each *block* also has three spatial coordinates (respectively *blockIdx.x*, *blockIdx.y*, *blockIdx.z*) that localize it in the global *grid*. Figure 2 illustrates this organization for 2D case. So each *thread* works with the same *kernel* but uses different data according to its spatial location within the grid[8]. Moreover, each *block* has a limited *thread* capacity according to the hardware in use.

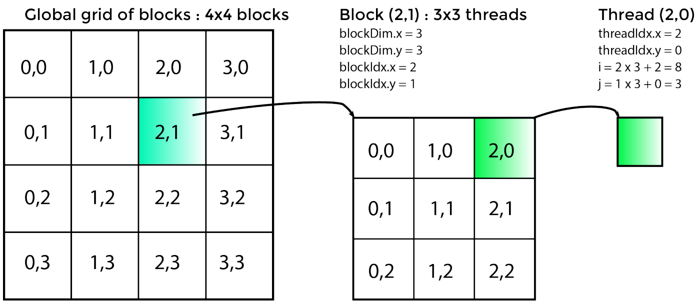

Fig. 2. Thread, blocks, grid organization

So, a multithreaded program is partitioned into blocks of threads that execute independently from each other. The distribution of *blocks* and *threads* on SM may be automatic and is provided by the runtime and drivers. If we take the example of the Nvidia environment (Compute Unified Device Architecture, CUDA), a compiled CUDA program can be executed on any number of multiprocessors as illustrated by Figure 3, and only the runtime system needs to know the physical multiprocessor count.

Implementing MABS Using GPGPU In [4], we realized a state of the art of the use of the GPGPU in the SMA context and identified two approaches allowing to implement a model on GPU: (1) *all-in-GPU*, for which the simulation runs entirely on the graphics card and (2) hybrid, the execution of the simulation is shared between the CPU and the GPU. In the first case (1), it is

[8] *Thread* is similar to the concept of task: A *thread* may be considered as an instance of the *kernel* which is performed on a restricted portion of the data depending on its location in the global grid (its identifier)

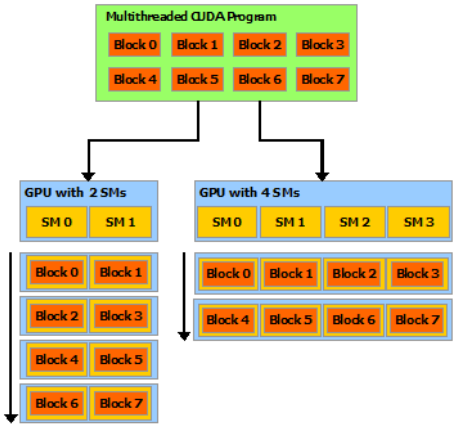

Fig. 3. Automatic Scalability (Source: Nvidia programming guide)

not trivial to take an existing model and translate it to make it work on GPU. GPU are very restrictive in operations and programming and the hardware can only be used in certain ways that requires advanced GPU programming skills. The hybrid approach (2) allows to use jointly the CPU and GPU and thus has two major advantages. Firstly, it brings more flexibility because one can choose what is going to be executed on the GPU, thus providing greater accessibility to the developed tools (as clearly shown in [6] [5], [18], [20]). Secondly, as hybrid systems are modular by design, they make it possible to use agents with complex and heterogeneous architectures. The *GPU Environmental Delegation of Agent Perceptions* principle relies on an hybrid approach.

4.2 Converting Agent Perceptions in Environmental Dynamics

The Principle *GPU Environmental Delegation of Agent Perceptions* principle was proposed in [9]. This principle consists in making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU. The underlying idea is to identify in the behavior of the agents some computations who can be transformed into environmental dynamics. It has been stated as follows: *Any agent perception computation not involving the agents state could be translated to an endogeneous dynamic of the environment, and thus considered as a potential GPU environment module.*

Related Works The GPU delegation can be linked with other works which try to separate and/or move a part of the computations made by the agents in other structures such as the interactions or the environment.

For example, within the MABS context, the EASS (*Environment As Active Support for Simulation*)[1] approach aims at strengthening the role of the environment by delegating him the policy of scheduling and adds a filtering system for the perceptions. IODA (*Interaction Oriented Design of Agent simulations*) [13] is centered on the notion of interaction and considers that agent behaviors can be described in a abstract way as a rule called interaction. Finally, [12] proposes to reduce the complexity of the models by using an centered environment approach: The environment becomes then a shared space dedicated to the execution of dynamics. The purpose is to facilitate the reusability and the integration of the various processes of the agents.

In a more general context, the *artifacts* approach integrates into the environment a set of dynamic entities representing the resources and the tools that the agents are going to be able to use and share [16]. These entities, called *artifacts*, structure and organize the environment by proposing a generic programming model including the features that the agents are going to have access.

GPU Delegation on a Case Study The integration of GPU computations was performed in TurtleKit[9] [10]. TurtleKit is a generic spatial ABM, implemented with Java, wherein agents evolve in a 2D environment discretized in cells. The proposed hybrid approach integrated in TurtleKit focuses on modularity. In this context, this allows to achieve three objectives: (1) maintain accessibility in the agent model while using GPGPU, (2) to scale and work with a large number of agents on large environment sizes and (3) promote re-usability in the particular context of GPU programming.

GPU Delegation has been used only once on a model of multi-level emergence (MLE) [2] of complex structures in TurtleKit. This very simple model relies on a unique behavior which allows to generate complex structures which repeat in a fractale way. The agent behavior is extremely simple and is based on the perception, the spread and the reaction to pheromones. So, in these works, GPU modules dedicated to the perception and the spread of pheromones were proposed.

5 GPU Delegation for Boids

5.1 Application of the GPU Delegation

The *GPU Delegation* stated that we cannot turn all the behaviors into environmental dynamics. Only agent perception computations that do not involve the agents state could be translated. It was clearly visible in the presented case study (MLE): The modules used to compute gradients and diffusion are completely independent from agents' states.

In our flocking model, it is not possible to find a computation independent from agents' attributes. However, we are able to identify some computations independent of agent's behaviors: Cohesion behavior is an ideal candidate and a part of it can be translated into a GPU module.

[9] http://www.turtlekit.org

Cohesion behavior consists in averaging the orientations of neighboring agents according to the selected *FieldOfView*[10]. All agents should therefore perform this computation in their own behavior and use the result to adapt their direction. The sequential implementation of this process is defined below:

---

for $bird\ in\ nearestNeighborsList$ **do**
  $sumOfHeading+ = getHeading(bird)$;
end
$neighborsAverageHeading =$
$sumOfHeading/sizeOf(nearestNeighborsList)$;

This loop is heavy because all the agents perform this computation in their own behavior at every step of simulation.

5.2 GPU Translation of the Average of the Orientations

To succeed the GPU translation according to the principle, we extract informations from agents' attributes (heading) and then delegate the associated computation (the loop) into the environment. To do this, the agents left its heading value, at each simulation step, in a 2D array (*headingArray*, identical in size to the grid of the environment) according to its position. This array is sent to a GPU module that, for each cell according to the selected *FieldOfView*, performs the average of the directions simultaneously. More precisely, each *thread* computes the average for a cell depending on its location in the global GPU grid (its identifiers: i and j in Algorithm 4). The GPU translation thus consists in transforming the sequential computation previously made in the cohesion behavior of the agents by a parallel computation made on the GPU and managed by the environment. Once realized, the average headings are available in all the environment. The agents recover in a 2D array (*flockCentering*, return by the GPU module) the value corresponding to their position and then adapt their movement.

The algorithm 4 present an implementation of the GPU module. Once the coordinates i and j of the *thread* initialized, the algorithm test if the *thread* does not possess coordinates superior to the size of the environment (represented here by the 2D array *headingArray*). We add then in *sumOfheading* all the headings of the neighbors being in the agent field of view then we divide this value by the number of agents taken into account. The module then returns the array *flockCentering* containing all the averages.

---

[10] In the context of TurtleKit, we call "FieldOfView" the number of cells (the radius around the selected cell) which is chosen to take into account for the computation of the average.

Algorithm 4: The Average *Kernel*

input : $width$, $height$, $fieldOfView$, $headingArray$ and $nearestNeighborsList$
output: $flockCentering$ (the average of directions)

$i = blockIdx.x * blockDim.x + threadIdx.x$;
$j = blockIdx.y * blockDim.y + threadIdx.y$;
$sumOfHeading, flockCentering = 0$;
if $i < width$ *and* $j < height$ **then**
  $sumOfHeading =$ getHeading($fieldOfView$, $headingArray[i, j]$);
end
$flockCentering[i, j] = sumOfHeading/sizeOf(nearestNeighborsList)$;

Compared with the sequential version of the algorithm, we see that the loop disappeared. So all the interest of the GPU version holds in the fact that the parallelization of this loop is realized thanks to the material architecture.

We extracted here information of the heading attribute but it would be also possible to make it with other attributes as speed.

5.3 Implementation and Integration of the Average Kernel

The implementation of GPU modules was made with CUDA and JCuda[11]. Figure 4 illustrates the integration of the GPU cohesion module in TurtleKit. The implementation has been easy, thanks to the independence between this module and the agent model.

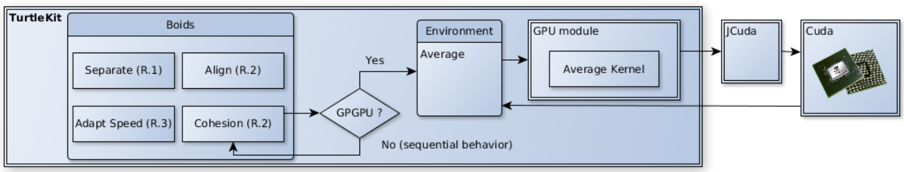

Fig. 4. Integrating GPU modules in TurtleKit

6 Experimentation

6.1 Experimental Protocol

To test our implementation of Reynolds's boids and the application of the GPU delegation principle, we simulate several environment sizes while making the

[11] The JCuda library allows to call GPU kernels, written in C, directly from Java

number of agents change. We execute successively the sequential version of the model (where the average is calculated in the agents behavior) then the GPGPU version (using the *Average* kernel). To estimate the performance and remain coherent with the criteria of analysis used in section 2, we observe the average computation time in milliseconds for an iteration.

6.2 Performance Test

For those tests, we reuse the same configuration as that used in section 2. This computer is composed of an Intel i7-4770 processor (Haswell generation, 3.40 GHz), an Nvidia K4000 graphics card (768 CUDA cores) and 16Go of RAM. Figure 5 presents the results obtained for various populations size in an 256 x 256 environment (top) and in an 512 x 512 environment (bottom).

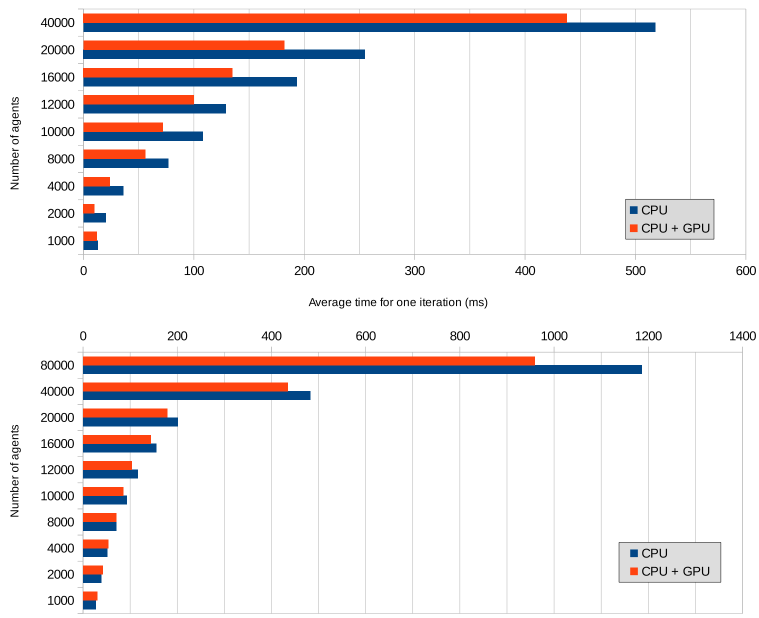

Fig. 5. Comparison of flocking simulations done with and without the GPU. Environment size: 256 (top) and 512 (bottom)

The use of the GPU module increases the performances by 25%. However, we notice that the performances is linked to the density of population. Indeed, when the density of the agents in the environment is lower, agents spend fewer time in cohesion and more to align itself and to separate. The density of the agents affects performance of the model when using the GPU module. The tipping point is clearly visible in the results, when the density of present agents exceeds 5% (respectively 1500 and 8000 entities), the joint use of the CPU and the GPU becomes more effective. So, the more the density of agents in the environment increases, more the observed gains of performances are important.

The performance gains are interesting considering the used hardware: Our Nvidia K4000 embeds 768 CUDA cores while the last Nvidia Tesla K40 card embeds 2880 CUDA cores and the Nvidia Tesla K10 card embeds 3072 cores (two GPU with 1536 cores on the same card). The fast evolution of GPGPU and graphics cards promise very significant gains of performances in the future.

6.3 Discussion

In addition to the observed performance gains, we notice other benefits to applying the *GPU Delegation* principle : The translation of a perception computed in the agent behavior into an environmental dynamics allows to remove a part of the source code and thus simplify the understanding of the behavior. Indeed, the agent makes a direct perception in the environment instead of a sequential computation which can be rather heavy.

Another interesting aspect is the fact that the created modules are independent from models thanks to this approach. They are not thus limited to the contexts for which they were defined. We are going to continue to apply the *GPU Delegation* principle to create new GPU modules and so increase the number of generic modules available. It is going to allow to constitute a usable modules library independant from models. This GPU functions library will improve the accessibility of the approach and the use of the GPGPU dedicated to MABS context with TurtleKit. This improvement in terms of genericness and accessibility is important because work with GPGPU often leads to implementation difficulties due to the specificity of this technology.

The application of the GPU delegation principle is based on a simple criterion independent from the implementation. It allows to convert the model and to create the GPU module in a rather fast way. TurtleKit being still in alpha release, we are going to continue to work on its architecture in order to make the conversion of a model as simple as possible.

Finally, translate a part of the agents behavior into an environmental dynamics allows to simplify its behavior because it takes away some of the source code. It is more readable because the agent does not have to deal with raw data.

7 Conclusion and Perspectives

In this paper, we described how we used the *GPU Environmental Delegation of Agent Perceptions* principle to implement a classic ABM, namely Reynolds's Boids, using GPGPU. Our purpose was to challenge the genericness and the ease-of-use of *GPU Delegation*. However, we made evolved this principle to be able to apply it to the boids model. Indeed, find a computation independent from agents' attributes was impossible, so we have identified in the cohesion behavior some computations independent of agent's behaviors. We thus translated these computations into a GPU module and made some tests to see the advantages brought by the *GPU Delegation*.

Our experiments shows that, using the *GPU Delegation*, it is possible to increase the size of the environments and the number of agents thanks to a speed up which can reach 25% according to the chosen parameters.

From a software engineering perspective, the use of this principle allows to consider important aspects of MABS with respect to the GPGPU context. By promoting a clear separation between the agent behaviors (handled by the CPU) and environmental dynamics (managed by the GPU), *GPU Delegation* represents an interesting design guideline for tackling the genericness issue and promote reusability of created tools. This essential criterion is often neglected in a context GPGPU [4]. Indeed, the application of the principle allowed the creation of a generic GPU module which is independent with respect to the agents.

Both implementations of the delegation principle, realized with MLE in [9] and flocking here, show that if the analysis of the model is made by keeping in mind the characteristics of the approach, the delegation of the computations and the creation of the GPU module could be very easy and fast, which is a valuable aspect of *GPU Delegation*, especially considering the technical difficulties related with the GPGPU context. However, as the GPU delegation still requires specific skills, we plan to apply to other models this principle in order to experiencing and continuing to generalize the approach.

As a perspective, our goal is to propose an explicit design methodology, a development guide consisting in rendering the use of GPU delegation more explicit and accessible to external users. The idea is to allow everyone to take a model and adapt it to make it work in a context GPGPU.

References

1. F. Badeig and F. Balbo. Définition d'un cadre de conception et d'exécution pour la simulation multi-agent. *Revue d'Intelligence Artificielle*, 26(3):255–280, 2012.
2. G. Beurier, O. Simonin, and J. Ferber. Model and Simulation of Multi-Level Emergence. In *ISSPIT'02*, Apr. 2008.
3. A. Grignard, P. Taillandier, B. Gaudou, D. Vo, N. Huynh, and A. Drogoul. GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin, 2013.
4. E. Hermellin, F. Michel, and J. Ferber. Systèmes multi-agents et GPGPU : état des lieux et directions pour l'avenir. In *Principe de Parcimonie - JFSMA 14*, pages 97–106. Cepadues Editions, 2014.
5. G. Laville, K. Mazouzi, C. Lang, N. Marilleau, B. Herrmann, and L. Philippe. MCMAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 544–554. Springer Berlin Heidelberg, 2014.
6. G. Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe. Using GPU for Multi-agent Multi-scale Simulations. In *Distributed Computing and Artificial Intelligence*, volume 151 of *Advances in Intelligent and Soft Computing*, pages 197–204. Springer Berlin Heidelberg, 2012.

7. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A Multiagent Simulation Environment. *Simulation*, 81(7):517–527, 2005.
8. M. Lysenko and R. M. D'Souza. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.
9. F. Michel. Translating Agent Perception Computations into Environmental Processes in Multi-Agent-Based Simulations: A means for Integrating Graphics Processing Unit Programming within Usual Agent-Based Simulation Platforms. *Systems Research and Behavioral Science*, 30(6):703–715, 2013.
10. F. Michel, G. Beurier, and J. Ferber. The TurtleKit Simulation Platform: Application to Complex Systems. In *Workshops Sessions of the Proceedings of the 1st International Conference on Signal-Image Technology and Internet-Based Systems*, pages 122–128. IEEE, november 2005.
11. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
12. D. Payet, R. Courdier, N. Sébastien, and T. Ralambondrainy. Environment as support for simplification, reuse and integration of processes in spatial MAS. In *Proc. of the 2006 IEEE International Conference on Information Reuse and Integration, USA*, pages 127–131. IEEE Systems, Man, and Cybernetics Society, 2006.
13. S. Picault. *From multi-agent simulation to multi-level simulation. Reifying the interactions.* Habilitation à diriger des recherches, Université des Sciences et Technologie de Lille - Lille I, Dec. 2013.
14. M. Resnick. StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 11–12, New York, NY, USA, 1996. ACM.
15. C. W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, volume 21 of *SIGGRAPH Computer Graphics '87*, pages 25–34, New York, NY, USA, 1987. ACM.
16. A. Ricci, M. Piunti, and M. Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.
17. P. Richmond, D. Walker, S. Coakley, and D. M. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3):334–47, 2010.
18. Y. Sano, Y. Kadon, and N. Fukuta. A Performance Optimization Support Framework for GPU-based Traffic Simulations with Negotiating Agents. In *Proceedings of the 2014 Seventh International Workshop on Agent-based Complex Automated Negotiations*, 2014.
19. E. Sklar. NetLogo, a Multi-agent Simulation Environment. *Artificial Life*, 13(3):303–311, 2007.
20. G. Vigueras, J. Orduña, and M. Lozano. A GPU-Based Multi-agent System for Real-Time Simulations. In *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 15–24. Springer Berlin Heidelberg, 2010.