present (despite work such as Flame GPU [29] or [16, 15]). This is especially true considering that the proportion of works dealing with the use of GPGPU in MABS is still very small compared to the number of MABS-based publications and while the need in computing resources is strong and always increasing.

2.3 GPU Environmental Delegation

The purpose of the work presented in this article is to determine whether the conclusions stated by P&A in 2008 are still relevant today despite the evolution of GPGPU and its use in MABS. To this end, the *GPU Environmental Delegation principle* (GPU delegation for short) is used. Based on an hybrid approach, this design principle aims at easing the integration of GPGPU in MABS models rather than making its use transparent, in contrary to the works previously mentioned.

2.3.1 Overview of the Principle

GPU delegation is inspired by an Agent-Oriented Software Engineering (AOSE) trend which consists in using the environment as a first class abstraction in MAS [1]. This idea is today well accepted and has proved to be a relevant approach for modeling and developing MAS [33]. Especially, it could help to enhance the efficiency of agent interactions or simplify the behavioral process of the agents thanks to the environment that produce high level percepts. Therefore, at a high level, such an approach allows to design MAS with a clear separation of concerns.

So, at the implementation level, GPU delegation has to be related to other research works that consider the environment as a core concept of MAS and which reify parts of the agents' computations in external structures. Related examples are EASS (Environment As Active Support for Simulation) [3], IODA (Interaction Oriented Design of Agent simulations) [14], environment-centered approach for MABS [24] and the artifact approach [26]. In the scope of our work, considering the environment as a first class entity enables us to keep the programming accessibility of the agent model in a GPU context and be able to scale up both the number of agents and the size of the environment.

Proposed in [18], the GPU delegation principle is based on the fact that it is very difficult to deport all the behavior of agents on graphics cards (that is why it is especially reactive agents who evolve in simulations which use an all-in-GPU approach [27]). So, this principle consists in making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU. Especially, one major idea underlying this principle is to identify some computations (such as agent-level perceptions) which can be transformed into environmental dynamics.

2.3.2 GPU Delegation in Practice

GPU delegation has been used for the first time on a model of Multi-Level Emergence (MLE) [4] of complex structures [18]. This very simple model relies on a unique behavior which allows to generate complex structures which repeat in a fractal way. The agent behavior is extremely simple and based on the perception, spread and reaction to pheromones. So, in this work, GPU modules dedicated to the perception and the spread of pheromones were proposed.

A second experimentation was proposed in [12] which further trials GPU Delegation by testing its feasibility and genericness on a classic ABM, namely Reynolds's boids[5]. [12] especially 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.

For both of these works, the integration of GPU computations was performed in the TurtleKit platform[6] [19]. 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) maintaining accessibility in the agent model while using GPGPU, (2) being able to scale and work with large number of agents on large environment sizes and (3) promoting reusability in the particular context of GPU programming.

2.3.3 Evolution of GPU Delegation

With respect to the underlying hybrid approach, GPU Delegation is about identifying specific behaviors which can be turned into environmental dynamics. Especially, GPU delegation states that agent perception computations that do not involve the agent's states could be translated into environmental dynamics and thus into a GPU module performing the computation.

However, the flocking model experimentation [12] shows that, according to the model used, GPU delegation could be extended with the aim at finding more computations to be delegated. So, the GPU delegation principle has evolved in order to be applied on a larger number of models and could be now stated as follows: If computations made within the behavior of the agent do not involve or do not modify the agent's states, they could be translated into environmental dynamics.

3. MODELS AND IMPLEMENTATIONS

To provide a comparison between the study conducted by P&A in 2008 and solutions available in 2015, especially hybrid systems, we use GPU Delegation to adapt and implement four models: Two already done by P&A (Conway's Game of Life and Schelling's segregation) and two taken from the Netlogo models library [31] (Fire model and DLA model). In this section, we present the associated technology and describe the implementation's process.

3.1 GPGPU Implementation with CUDA

To program on the graphics card and exploit its GPGPU capabilities, we use CUDA which is the GPGPU programming interface provided by Nvidia. The associated programming model relies on the following philosophy[7]: The CPU is called the *host* and plays the role of scheduler. The *host* manages data and triggers *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

[5] `http://www.lirmm.fr/~hermellin/Website/Reynolds_Boids_With_TurtleKit.html`
[6] *e.g.* http://www.turtlekit.org
[7] *e.g.* http://docs.nvidia.com/cuda/

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 1 illustrates this organization for the 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.

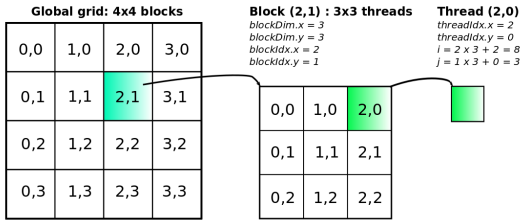

Figure 1: Thread, blocks, grid organization

For using GPGPU in the TurtleKit platform, we use the JCUDA library which allows to use CUDA through Java[9].

3.2 Models and Application of GPU Delegation

In this section, the four selected models and their implementation are introduced. Then, the application of the GPU delegation on each of these models is described.

3.2.1 Game of Life

The first model, which was also in P&A study, is the Conway's Game of Life [11]. This famous model shows that complex patterns can emerge from the implementation of simple rules. While this model does not contain agents (it is a cellular automaton), it defines an environmental dynamics which is representative of those encountered in MAS.

The environment of the Game of Life is a cellular automaton made of a two dimensional grid of square cells, each of which being in one of two possible states: Dead or alive. Each cell interacts with its eight neighbors according to the following rules: (1) Any living cell with less than two or more than three alive neighbors dies, (2) any living cell with two or three living neighbors stays alive and (3) any dead cell with exactly three living neighbors becomes alive.

In our experiments, the grid is initialized randomly[10]. At each time step, all the cells are updated according to the previous rules. Figure 2 illustrates the simulation steps.

Applying GPU Delegation.

The main computational part of the model is located in step (1) which consists in computing a sequential loop: It calculates the new state of each cell for the next step of the simulation. So, the more the environment is large, the more this computation is long. It may therefore be advantageous to transform this computation into a GPU module.

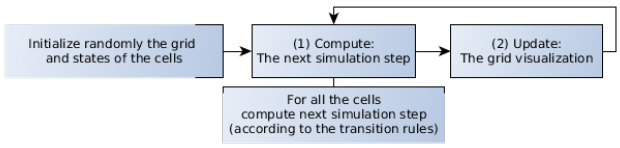

Figure 2: Game of Life simulation

To create the GPU module correctly, it is necessary to focus on the representation of data. Especially, to avoid expensive transfers between CPU and GPU, the data need to be sent only once at each step. To this end, each cell write its state value (1 for alive and 0 for dead) in a 2D array (matching the size of the environment) according to its position. Then, this array is sent to the GPU. This module does the sum of the states (more precisely the sum of the number of alive cells) of all Moore neighborhood cells for each cell of the grid and stores the result in the 2D result array. Thus, each cell of the result array contains a value between 0 (no cell alive) and 8 (all neighbors are alive). The result array is then used to update the grid and compute the next simulation step according to the transition rules. Algorithm 1 presents an implementation of the corresponding GPU kernel.

Algorithm 1: Alive cells *Kernel*

```
   input  : width, height, statesArray
   output: resultArray (the number of alive neighbors)
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  sumOfState = 0 ;
4  if i < width and j < height then
5  |  sumOfState = getNeighborsState(statesArray[i, j]);
6  end
7  resultArray[i, j] = sumOfState ;
```

3.2.2 Schelling's Segregation

The second model is a variant of the Schelling's Segregation model [30]. This project models the behavior of two types of agents in a neighborhood: Red agents and green agents. These agents are scattered across a two dimensional grid. Their purpose is to be happy by staying near like-colored agents (each red agent wants to live near at least some red agents, and the same holds for the green agents). If they are dissatisfied at their position, the agents attempt to move to a new random vacant location for finding a better place according to their objective of happiness.

The simulation shows how these individual preferences ripple through the neighborhood, leading to large-scale patterns. In our experiments, green and red agents are scattered with the same proportion, and distributed randomly over the environment. Figure 3 illustrates the simulation steps.

Applying GPU Delegation.

The most intensive computations are in step (2) and (3) which consist, for each agent, in recovering a neighbor list

[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).

[9] *e.g.* http://www.jcuda.org

[10] The probability for the cell to be dead or alive is the same.

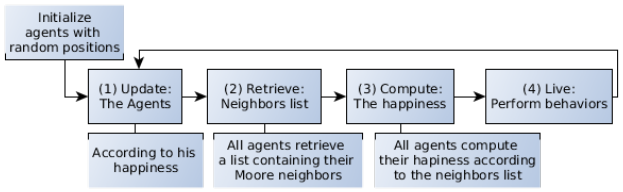

Figure 3: Segregation simulation

and then computing its happiness according to the list. So, many sequential loops have to be done at each time step, and the required computation time clearly increases depending on the number of agents. According to the GPU delegation, the happiness computation can be deported into an environmental dynamics because these computations do not modify agents' states.

In the CPU model, the computation of happiness is done by testing the color of the agents present in the list of neighbors and counting the number of agents in each community according to their state: 1 for green agents and -1 for red agents. However, if we want to deport this computation, the data structure sent to the GPU needs to be adapted. So, at each time step, agents write in a 2D array (matching the size of the environment) their states depending on their position. This table is sent to the GPU that computes the sum of the neighboring states and returns in a result array the values for each cell of the environment. This result array thus contains values between -8 (all agents around are red) and 8 (all agents around are green). The agents then recover the value in the result array with respect to their position and act accordingly. Algorithm 2 presents an implementation of the corresponding GPU kernel.

Algorithm 2: Hapiness *Kernel*

```
   input : width, height, communityArray
   output: resultArray (agent's type around)
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  sumOfCommunityState = 0 ;
4  if i < width and j < height then
5      for agent in getNeighborsCommunity() do
6          agentCommunityState =
           communityArray[i, j];
7          if agentCommunityState == 1 then
8              sumOfCommunityState++;
9          end
10         else if agentCommunityState == −1 then
11             sumOfCommunityState − −;
12         end
13     end
14 end
15 resultArray[i, j] = sumOfCommunityState ;
```

3.2.3 Fire

Inspired by a model from the Netlogo library, this third model is called *Fire* and simulates the spread of a fire through a forest. It shows that the fire's chance of affecting the most possible number of trees in the forest depends critically on the density of trees.

In our model, all the trees are agents placed randomly in the environment (a two dimensional grid). Trees can be alive, burned or dead. When it burns, a tree releases heat which spreads in the environment. This heat can ignite other trees around: A tree ignites when the temperature is above a defined threshold. The threshold of each agent is randomly set within a range of values. A tree dies when its life reaches zero: The life of the tree decreases when it burns. Figure 4 illustrates the simulation steps.

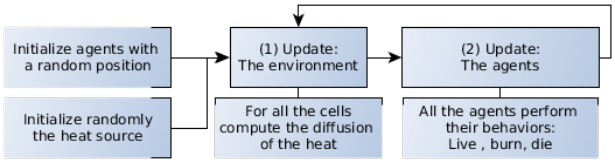

Figure 4: Fire simulation

Applying GPU Delegation.

The most greedy computation loop of this model is in step (1). Indeed, the environment must compute the heat diffusion which requires making a global sequential loop over all the cells. Thus, the computation time increases very quickly according to the size of the environment. Because the heat diffusion is already an environmental dynamics, the delegation of this computation is very easy because it can be directly translated into a GPU module.

So, in the GPU model, the heat diffusion is done as follows: At each time step, agents add in a 2D array (matching the size of the environment) the heat that they release according to their state (alive, burn or dead). Then, this array is sent to the GPU module that compute the sum of heat values from neighboring cells (Moore neighborhood here) for each cell of the environment. More preciselly, the sum consists in adding the heat values already present in the environment (from the previous steps) and the heat generated by the agents, all modulated by a diffusion variable. Once the computation performed, the agents recover the heat value in the array with respect to their position and act accordingly. Algorithm 3 presents an implementation of the corresponding GPU kernel.

Algorithm 3: Heat diffusion *Kernel*

```
   input : width, height, heatArray, radius
   output: resultArray (the quantity of heat)
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  sumOfHeat = 0 ;
4  if i < width and j < height then
5      sumOfHeat =
       getNeighborsHeat(heatArray[i, j], radius);
6  end
7  resultArray[i, j] = sumOfHeat * heatAdjustment ;
```

3.2.4 DLA

The last model is also inspired from a Netlogo model and is called *DLA*. DLA demonstrates diffusion-limited aggregation, in which randomly moving particles stick together to form beautiful treelike branching fractal structures. There are many patterns found in nature that resemble the patterns produced by this model: Crystals, coral, fungi, lightning, and so on.

In this model, particles are agents (initialy red), that move in a random way over the all environment (a two dimensional grid). Randomly, one of the agents stops and stays at the same position and in turn changes its color to green. Then, when a moving red agent encounters a motionless green agent, it stops and changes its color to green while the other agents continue to move in a random way. Figure 5 illustrates the simulation steps.

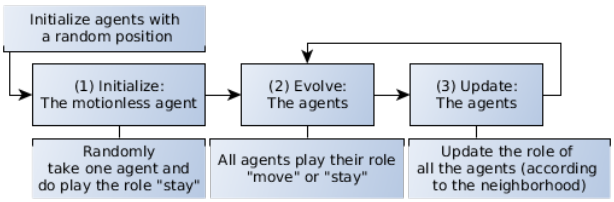

Figure 5: DLA simulation

Applying GPU Delegation.

The most greedy computations are in step (2) and (3) which consist for each agent in recovering a neighbor list and then searching within this list the nearest neighbors. So, the computation time of this model greatly increases according to the number of agents. According to GPU delegation, the agent action which consists in checking if one of the neighbor agents is green (motionless) can be converted into an environmental dynamics and then performed by a GPU kernel because this computation do not modify agents' states.

However, it is necessary to adapt the data structure to enable this transformation. Thus, all agents report, according to their position, their presence in a 2D array (matching the size of the environment): 1 if an agent occupies the cell, 0 otherwise. This array is sent to the GPU that calculates for each cell if there are agents around it. All the cells of the result array thus contain a value representing the number of neighbor agents (0 for an empty cell, 1 or more means there are neighbors around). So, the agents only have to recover the result value in the array and adjust their behavior accordingly. Algorithm 4 presents an implementation of the corresponding GPU kernel.

Algorithm 4: Presence detection *Kernel*

```
   input  : width, height, presenceArray
   output: resultArray (the number of neighbors around)
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  sumOfAgents = 0 ;
4  if i < width and j < height then
5  |  sumOfAgents =
   |  getNeighborsPresence(presenceArray[i, j]);
6  end
7  resultArray[i, j] = sumOfAgents ;
```

4. EXPERIMENTATIONS AND ANALYSIS

In this section, the main idea is not only to evaluate the performance of the experiments but to reflect on the accessibility, modularity and ease of use of the GPU delegation principle (the conceptual part of these implementations).

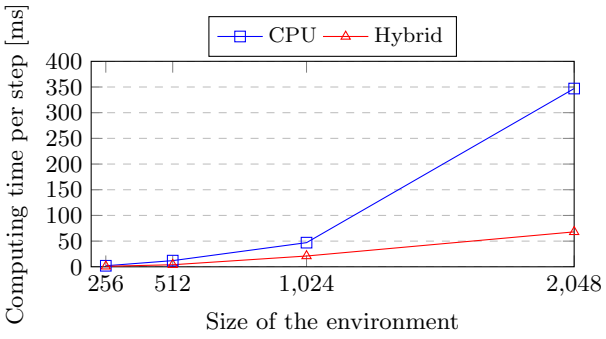

Figure 6: Results for the Game of Life model

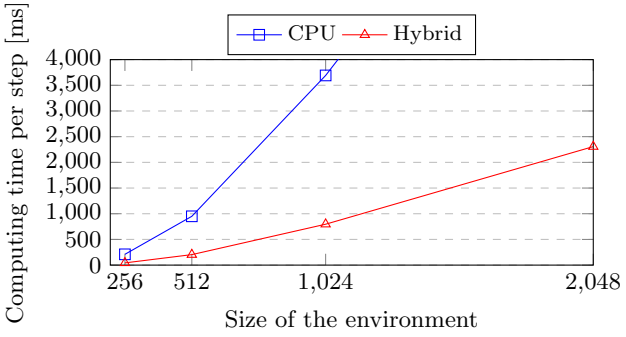

Figure 7: Results for the Segregation model

4.1 Experimentations

The GPU delegation principle produces implementations which are very different from those made by P&A. Moreover, the hardware and associated tools have also greatly evolved since P&A's work. So, we cannot directly compare the performance results of P&A with the experiences that we conduct here.

To test the performance of our implementations, each model is simulated in its CPU and hybrid (CPU + GPU) versions. Moreover, each model is simulated for different environment sizes and various densities of agents. Each simulation is executed several times over a period of 10,000 time steps. We average then the execution time for an iteration (lower execution time is better) allowing to compare the execution performance of each implementation.

For those tests, the configuration is composed of an Intel i7-4770 processor (Haswell generation, 3.40 GHz) and an Nvidia K4000 graphics card (Kepler architecture, 768 CUDA cores). Here is the list of experimentations (with setup) conducted and their results:

- Game Of Life (Figure 6): Environment size 256, 512, 1 024, 2 048; Fixed density of agents: 50%.
- Segregation (Figure 7): Environment size 256, 512, 1 024, 2 048; Fixed density of agents: 90%.
- Fire (Figure 8): Environment size 256, 512, 1 024, 2 048; Density of agents ranging from 10% to 100%.
- DLA (Figure 9): Environment size 256, 512, 1 024, 2 048; Density of agents ranging from 10% to 90%.

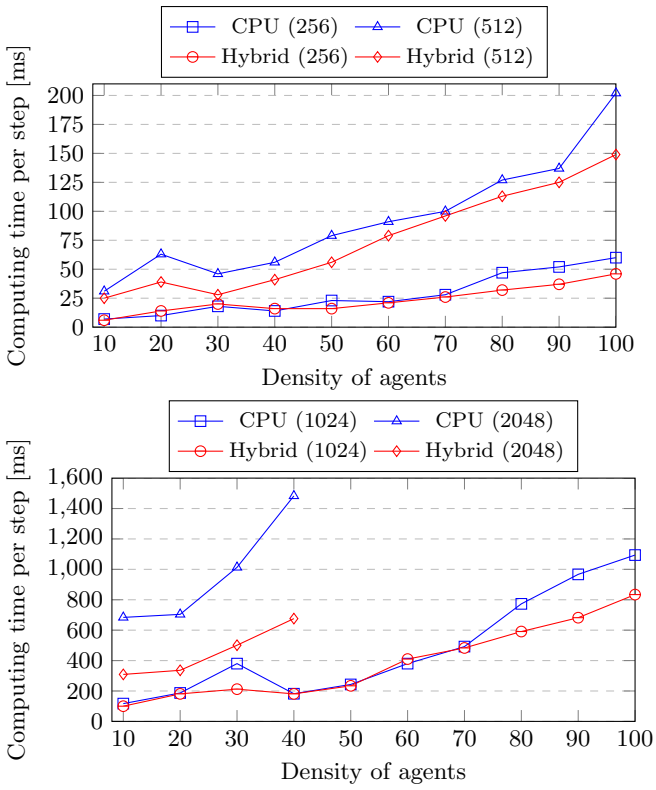

Figure 8: Results for the Fire model

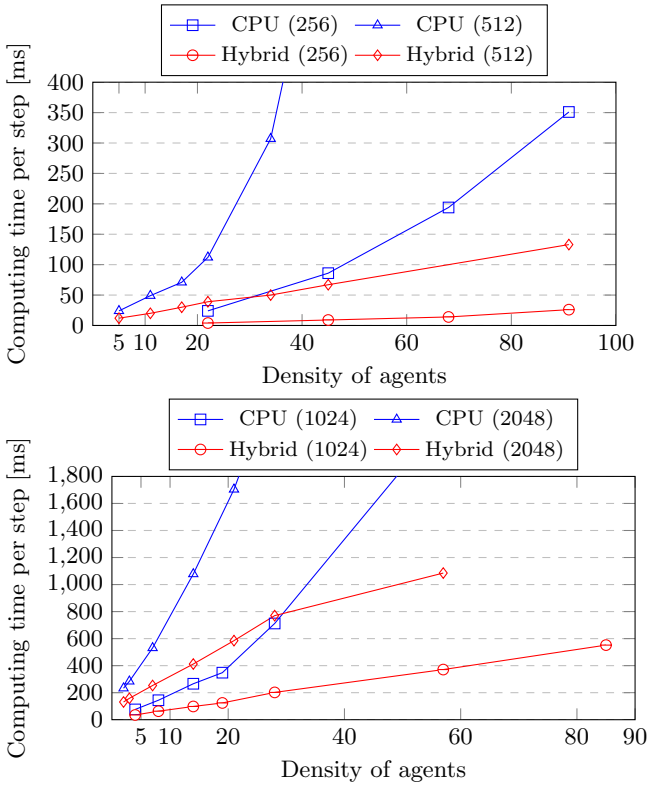

Figure 9: Results for the DLA model

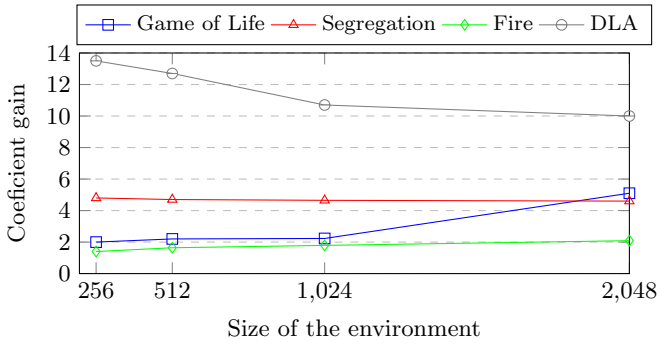

Figure 10: Coefficient of acceleration (between CPU and CPU+GPU versions)

4.2 Analysis

In this section, we analyze the results from both a performance and a conceptual perspective by taking into account accessibility, genericness and ease of use of GPU delegation.

4.2.1 Performances

From Figure 10, we notice that performance gains vary significantly depending on the simulated model and the size of the environment. With GPU delegation, the gain can reach x14 but is more likely between x2 and x5 which is still a good result. Concerning the DLA model, the performance decreases when increasing the size of the environment (which is the opposite of what can be seen with the other models). This decrease is explained by the consumption of resource by the agents that slows the simulation.

P&A's implementations are much more efficient and allow to have greater performance gains that can reach x40 (particularly for the Game of Life model) but generally the gain is around x15 [25]. The differences are huge but must be put in perspective because we have no informations about how the models and CPU implementations were created. Nonetheless, the order of magnitude of performance gains cannot be ignored. This difference is explained by the implementation technique used by P&A. Indeed, they used an all-in-GPU approach that consists in running entirely the model on the GPU with the objective of obtaining maximum performances. GPU delegation offers less impressive performances but brings other significant benefits.

4.2.2 Genericness, Modularity and Reusability

GPU delegation brings more genericness thanks to its hybrid conception. Especially, the created GPU modules can be reused in other simulations. That is the case here: The module created for the Game of Life model was reused in the DLA model. Only the data sent have been adapted. This is a major difference compared to the work and implementations of P&A, and more generally with works using an all-in-GPU approach. Indeed, with such works, the model used is completely redesigned and reprogrammed to run entirely on the GPU and thus cannot be easily reused. Therefore, the solution created is only punctual, which is not satisfying from a software engineering perspective.

On the contrary, more genericness could be obtained with an hybrid approach. This is especially the case with GPU delegation because it allows to reuse the created GPU modules, thus saving a lot of development time.

From an accessibility perspective, GPU delegation increases GPGPU accessibility because it promotes a modular design of the model that produces very simple kernels requiring only very little GPGPU knowledge. Moreover, by deporting only a specific part of the agent computations, it is thus possible to take advantage of the computing power of GPGPU without changing the agent model.

Finally, this modular design allows to choose what is run on the CPU and executed by the GPU. The strategy put forward by GPU delegation is to identify in the model the computations that can be transformed into environmental dynamics and thus translated into GPU modules. Considering the difficulties faced when implementing the behavior of agent on GPU or providing generic tools to do it, this approach greatly eases the use of GPGPU in MABS context.

4.2.3 Toward a Dedicated Methodology

The application of GPU delegation as generic approach for using GPU programming in MABS has shown many advantages whether in [18] or [12]. The experiment conducted in this article confirms it. However, in view of these studies, there are still some open questions such as the difficulty of implementation for a new GPGPU user and the types of models on which GPU delegation could be applied.

Indeed, for now, we have only tested GPU delegation on models that we know compatible. This type of model has the following features: The agents are reactive and operate in a discretized environment. These agents communicate and exchange informations through the environment. Next step will be to consider models with continuous environments or agents having cognitive abilities.

About the implementation difficulties, GPU delegation aims at easing the integration of GPGPU in MABS models by promoting a real delegation strategy that ends with very simple kernels wich are easy to implement and require only very little GPGPU knowledge. However, it will be necessary to improve the support around this approach.

But more generally, our objective is to offer a generic approach which can be applied on a wide variety of models and that eases the use of GPGPU. An approach that is also able to define if the model can benefit from GPGPU power before starting the adaptation process and could determine whether there is an interest in applying the GPGPU delegation on the selected model.

From the previous experiment, a recurring process has appeared during the application of GPU delegation. We have indeed followed an iterative process that allows us to apply our approach on each model. Once formalized, we believe that this process can become a methodology dedicated to GPGPU for MABS. A methodology that would (1) help potential users to decide if they could benefit from GPGPU considering their models and (2) support the translation of MABS model into GPU programming without hiding the underlying technology. And finally such a methodology would help to spread the GPGPU in the MABS community.

5. SUMMARY AND FUTUR WORKS

In this paper, we proposed to study through experiments if the conclusions and issues outlined by P&A in 2008 are still true despite the evolution of GPGPU and MABS. This paper was motivated by the fact that there are actually only very few publications dealing with the use of GPGPU in a MABS context, despite the computing power that these simulations require.

To address some of the challenges that remains when using GPGPU for MABS, we apply GPU delegation on four models and show that it helps to solve some of the problems outlined by P&A. Based on a hybrid approach, GPU delegation proposes to identify in the model what are the computations that can be transformed into environmental dynamics and then translated into GPU modules.

Instead of focusing on accessibility by hiding the use of GPGPU, GPU delegation aims at easing the direct integration of GPU programming in MABS models. So, GPU delegation helps he user during the adaptation process contrary to some solutions which seek to make transparent the use of GPGPU.

This principle was applied on four models and an experimentation which compares the GPU-based runtime speed with the speed of equivalent CPU-based models was conducted. The results of experiments and analysis were then proposed.

From a performance perspective, GPU delegation offers less significant performance gains compared to all-in-GPU solutions. As highlighted by P&A, this performance gap can only be obtained at the expense of the accessibility, modularity and genericity and reflects of a totally different use of the GPU architecture.

From a conceptual point of view, we have seen that GPU delegation improves genericity (reusability of the created GPU modules), provides a more modular design (with the hybrid approach) and enhances accessibility thanks to the fact that we are no longer using an all-in-GPU approach. More precisely, its reusability and design ease the direct use of GPGPU in a MABS context. Given that GPU does not run the entire simulation but only some parts, the adaptation of the model is more simple and the knowledge required is less important. Indeed, the reusability allows to save the time invested and the efforts done to use this technology because it concretely relies on an AOSE perspective thate promotes a clear separation of concerns, and thus modularity and reusability.

Thanks to the evolution of both the GPGPU field and associated tools, GPU delegation shows that it can be easy to implement and efficient while providing significant conceptual advantages. The next step is to develop the user support so that GPU delegation can be simple to use. To this end, it is now necessary to formalize its use and all conception steps.

Especially, during implementation of the four models, we identified some repetitions in the application of the principle. Moreover, some patterns and similarities in the source code can be observed. So, an iterative process begin to appear: A list of the steps necessary to transform the model in order to use GPGPU through GPU delegation.

Based on this work, our next objective is to create a comprehensive design methodology for GPGPU in a MABS context based on the GPU delegation principle. This methodology will allow to take a model and then will help in every step necessary in the translation process in order to let the model takes advantage of GPGPU power. Such a methodology should allow to use GPU programming on a wide variety of models and spread this technology in the MABS community.

REFERENCES

[1] 4th International Workshop, E4MAS 2014 - 10 Years Later, Paris, France, May 6, 2014, Revised Selected and Invited Papers. In D. Weyns and F. Michel, editors, *Agent Environments for Multi-Agent Systems IV*, volume 9068 of *LNCS*. Springer, 2015.

[2] B. G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient Simulation of Agent-based Models on multi-GPU and Multi-core Clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 29:1–29:10, ICST, Brussels, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] F. Badeig, F. Balbo, and S. Pinson. A contextual environment approach for multi-agent-based simulation. In *ICAART'2010, 2nd International Conference on Agents and Artificial Intelligence*, pages pp 212–217, Spain, June 2010. ICAART.

[4] G. Beurier, O. Simonin, and J. Ferber. Model and Simulation of Multi-Level Emergence. In *ISSPIT'02*, Apr. 2008.

[5] A. Bleiweiss. Multi agent navigation on the GPU. *Games Developpement Conference*, 2009.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008. General-Purpose Processing using Graphics Processing Units.

[7] N. Collier and M. North. *Repast HPC: A Platform for Large-Scale Agent-Based Modeling*, pages 81–109. John Wiley and Sons, Inc., 2012.

[8] A. Demeulemeester, C.-F. Hollemeersch, P. Mees, B. Pieters, P. Lambert, and R. Van de Walle. Hybrid Path Planning for Massive Crowd Simulation on the GPU. In J. Allbeck and P. Faloutsos, editors, *Motion in Games*, volume 7060 of *Lecture Notes in Computer Science*, pages 304–315. Springer Berlin Heidelberg, 2011.

[9] R. M. D'Souza, M. Lysenko, and K. Rahmani. SugarScape on steroids: simulating over a million agents at interactive rates. *Proceedings of Agent 2007 conference*, 2007.

[10] U. Erra, B. Frola, V. Scarano, and I. Couzin. An Efficient GPU Implementation for Large Scale Individual-Based Simulation of Collective Behavior. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 51–58, Oct 2009.

[11] M. Gardner. Mathematical games: The fantastic combinations of John Conwayś new solitaire game life. *Scientific American*, 223(4):120–123, 1970.

[12] E. Hermellin and F. Michel. GPU Environmental Delegation of Agent Perceptions: Application to Reynolds's Boids. In *Multi-Agent-Based Simulation XVI - International Workshop, MABS 2015, Istanbul, Turkey, May 5*.

[13] A. V. Husselmann and K. A. Hawick. Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUs. In *International Conference on Parallel and Distributed Computing and Systems*, pages 100–107. IASTED, 2011.

[14] Y. Kubera, P. Mathieu, and S. Picault. IODA: an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3):303–343, 2011.

[15] 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 D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. Scott, and J. Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 544–554. Springer Berlin Heidelberg, 2014.

[16] 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.

[17] 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.

[18] 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.

[19] F. Michel, G. Beurier, and J. Ferber. The TurtleKit Simulation Platform: Application to Complex Systems. In A. Akono, E. Tonyé, A. Dipanda, and K. Yétongnon, editors, *Workshops Sessions of the Proceedings of the 1st International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2005, November 27 - December 1, 2005, Yaoundé, Cameroon*, pages 122–128. IEEE, november 2005.

[20] F. Michel, J. Ferber, and A. Drogoul. Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective. In Adelinde Uhrmacher and Danny Weyns, editors, *Multi-Agent Systems: Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 3–52. CRC Press - Taylor & Francis, June 2009.

[21] 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.

[22] H. Parry and M. Bithell. Large scale agent-based modelling: A review and guidelines for model scaling. In A. J. Heppenstall, A. T. Crooks, L. M. See, and M. Batty, editors, *Agent-Based Models of Geographical Systems*, pages 271–308. Springer Netherlands, 2012.

[23] R. Pavlov and J. Muller. Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors. In L. Camarinha-Matos, S. Tomic, and P. Graca, editors, *Technological*

Innovation for the Internet of Things, volume 394 of *IFIP Advances in Information and Communication Technology*, pages 115–122. Springer Berlin Heidelberg, 2013.

[24] D. Payet, R. Courdier, N. Sébastien, and T. Ralambondrainy. Environment as support for simplification, reuse and integration of processes in spatial MAS. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration, IRI - 2006: Heuristic Systems Engineering, September 16-18, 2006, Waikoloa, Hawaii, USA*, pages 127–131. IEEE Systems, Man, and Cybernetics Society, 2006.

[25] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123, 2008.

[26] 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.

[27] P. Richmond and D. M. Romano. Agent based GPU, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the GPU. In *In Proceedings International Workshop on Super Visualisation (IWSV08)*, 2008.

[28] P. Richmond and D. M. Romano. A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. *European Simulation and Modelling*, 2011.

[29] 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.

[30] T. C. Schelling. *Micromotives and Macrobehavior*. W. W. Norton, revised edition, Sept. 1978.

[31] E. Sklar. NetLogo, a Multi-agent Simulation Environment. *Artificial Life*, 13(3):303–311, 2007.

[32] D. Strippgen and K. Nagel. Multi-agent traffic simulation with CUDA. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 106–114, June 2009.

[33] D. Weyns, H. Dyke Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for Multiagent Systems State-of-the-Art and Research Challenges. In D. Weyns, H. Dyke Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems*, volume 3374 of *Lecture Notes in Computer Science*, pages 1–47. Springer Berlin Heidelberg, 2005.