HAL
open science

GPU Environmental Delegation of Agent Perceptions for MABS

Fabien Michel

To cite this version:

Fabien Michel. GPU Environmental Delegation of Agent Perceptions for MABS. ICCS'12: International Conference on Complex Systems, Nov 2012, Agadir, Morocco. pp.1-6, 10.1109/ICoCS.2012.6458513 . lirmm-00805983

HAL Id: lirmm-00805983

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00805983v1

Submitted on 29 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GPU Environmental Delegation of Agent Perceptions for MABS

Fabien Michel
Laboratoire d'Informatique, de Microélectronique, et de Robotique Montpellier
Université Montpellier II - CNRS
161 rue Ada, Montpellier Cedex 2, France
Email: fmichel@lirmm.fr

***Abstract*—Considering the digital simulation of complex systems, General-Purpose Computing on Graphics Processing Units (GPGPU) is a relevant approach for addressing scalability issues. However, GPU programming is a very specific approach that strongly limits both the accessibility and the re-usability of the frameworks developed using GPGPU. This paper presents our approach for the integration of GPU modules in a Multi-Agent Based Simulation (MABS) platform. Especially, this paper shows how we keep the programming accessibility of the platform while gaining advantages of the GPU power. The paper also presents how this approach could be generalized and proposes a MABS design guideline dedicated to the GPU context.**

***Index Terms*—High Performance Computing, GPGPU, Multi-Agent Based Simulation**

I. Introduction

Because complex systems are composed of many interacting entities, studying their properties using digital simulation usually requires a lot of computing resources. Considering this issue, General-Purpose Computing on Graphics Processing Units (GPGPU) is gaining more attention as it can drastically speed up simulation runs with a cheap cost [1].

However, GPU programming requires a particular programming mindset because it relies on a highly specialized architecture. Indeed, because they are designed for graphics, GPUs are very restrictive in operations and programming and the hardware can only be used in certain ways. Therefore GPU programs have to (1) fit this architectural context and (2) follow the stream processing paradigm[1] [2]. So GPGPU is only useful and effective for problems which can be modeled with respect to these programming and architectural contexts.

This paper focuses on the use of GPGPU for developing Agent-Based Models (ABM). In such models, all the entities of a system are concretely modeled and simulated so that they act and interact on a shared environment [3]. In this scope, the specificities of GPU programming raise problems with respect to the re-usability of previously developed systems and algorithms. Indeed, using stream processing, many advantages of the object oriented programming cannot be used so that reformulating existing agent models accordingly is not a trivial task and requires advanced GPU programming skills [4]. Additionally, both GPU application programming interface (API) and hardware specification rapidly evolve, which limits the portability of the produced code and thus further increases the problem of re-usability. So, existing GPU-based ABMs are thus mostly related to only one particular domain and experiment and do not focus on code genericness.

This paper reports on the work we have done to use GPU programming in TurtleKit, a generic Logo-based Multi-Agent Based Simulation (MABS) platform [5]. Doing so, our goal was to (1) take advantage of the GPU for achieving large scale simulations while (2) preserving the programming accessibility of the platform and especially its object-oriented API. The paper presents this developing experience and proposes a GPU MABS design guideline derived from this experiment.

Section II presents examples of how GPGPU is used for developing ABMs and highlights their limits with respect to our objectives. Section III presents the TurtleKit platform and the simulation model we used as a benchmark. Section IV details how we designed a first GPU module by reformulating two different environmental dynamics and reports on the results obtained. Section V discusses how we designed a second GPU module by delegating agent computations to the environment. Section VI presents a generalization of our work as a MABS design guideline dedicated to the GPU context. Section VII concludes the paper and discusses related perspectives.

II. GPGPU for MAS-based Complex Systems

Thanks to hundreds of cores available on today graphic cards, GPU allows to perform thousands of similar computations at once in parallel on the graphic card, rather than sequentially using the CPU. This characteristic is particularly of interest when considering complex systems which are modeled using the Multi-Agent System (MAS) paradigm. In such systems, similar computations have to be done billions of times. So MAS are a perfect candidate for massive improvement using GPGPU and there are many works reporting on agent-based models developed using GPGPU for different application domains such as large scale crowd simulations (e.g. [6]), biology (e.g. [7]) or flocking simulations (e.g. [8]).

For instance, in [9], the FLAME cellular level agent-based simulation framework shows impressive performance enhancement for GPU simulations, even when compared to a cluster of CPUs. As highlighted by the authors, such improvements are invaluable with respect to the fast development of such

[1]Given a set of data (a stream), a series of operations (kernel functions) is applied to each element in the stream.

978-1-4673-4766-2/12/$31.00 © 2012 IEEE

complex models, especially because this allows for real-time visualization and thus real-time interaction.

To obtain this result, the FLAME agent model has been entirely translated in GPU code, thus raising the problem of programming accessibility. Dealing with this issue, the authors propose to abstract the end-users from knowing GPU by defining a XML-based formalism which is used to specify the behavior of the agents. If this is a good solution with respect to end-users, modifying or extending the proposed agent model still requires GPU knowledge, thus limiting both the scope the framework and its re-usability.

Regarding genericness, [4] proposes an interesting work that considers a whole class of ABMs at once, namely spatial ABMs. Such ABMs consist of a 2D discretized grid containing situated information and a collection of mobile agents. The widely used NetLogo platform [10] is an example of this class of ABMs. [4] clearly explains that the major challenge of this work was to reformulate this generic ABM in terms of stream computation, especially agent mobility, death, replication, execution orders and collision. So the authors propose to map agent states to textures so that these dynamics can be implemented using GPU. With respect to the execution of standard models such as SugarScape, the reported results show impressive enhancements in terms of speed and scalability.

Still, as remarked by the authors, applying such an *all-in-GPU* approach, there is an important trade-off which is that many advantages related with the object oriented programming are lost in the reformulation process. Especially, creating a new simulation model requires to create new GPU kernels to handle its specificity, and therefore GPU programming skills. With respect to end-users, such a lack of programming accessibility is a major limitation for the development of generic ABM frameworks.

More generally, studying the related literature, it is clear that the technical difficulties related with GPU programming naturally tend to both (1) narrow the scope of the developed frameworks and (2) limit their programming accessibility. This because the underlying agent model is often intrinsically related to the GPU code so that one has to directly modify it to make the agent model evolve. All-in-GPU-based agent frameworks are thus mostly restricted to a specific application domain and/or require advanced GPU skills.

III. Integrating GPU within TurtleKit

A. The TurtleKit Platform and the MLE Model

Like NetLogo, TurtleKit [5] is a spatial ABM, implemented with Java, relying on an agent model which is inspired by the Logo programming language. Especially, agents emit and perceive digital pheromones which diffusion and evaporation dynamics are handled by the environment (the 2D grid), thus creating pheromone fields. Handling such dynamics requires a lot of computing resources, which limits both performance and scalability, even when few pheromones are used.

Developing TurtleKit, our primary goal is programming accessibility so that we cannot choose an all-in-GPU approach. That is why we apply an intermediate approach that consists in integrating iteratively GPU parts in the simulation platform while ensuring that the TurtleKit API remains unchanged.

To this end, we choose to prototype and experiment with the model proposed in [11]. In this paper, a model of multi-level emergence (MLE) of complex structures is defined using a unique and very simple recursive agent behavior. More precisely, starting with only one kind of agents (level-0), the agents evolve and build a recursive structure having a circular shape. That is, level-0 agents turn around level-1 agents that turn around level-2 agents and so on.

To achieve this, the agent behavior relies on perceiving, emitting and reacting to three different types of pheromones: (1) *presence*, (2) *repulsion* and (3) *attraction*. Presence is used by an agent to evaluate how many agents are in its vicinity and decide if it has to mutate to the next or preceding level. Roughly a mutation occurs if an area is overcrowded or empty. Repulsion and attraction are both used by the agents to create a circular zone of attraction around them. The behavioral process is decomposed in four stages: Perception, Emission, Mutation and Move. So, the state and the behavior of each agent is completely defined by only one integer which is its current level. The level simply modifies the agent's emission rate and limits perceptions to pheromone of adjacent levels.

B. MLE as a Benchmark for Integrating GPU Modules

Theoretically, the highest level which could be observed with the MLE model is related to only two parameters: (1) the size of the environment, because large structures require room to appear and (2) the number of initial agents, because level-i structures need a certain number of level i-1 agents to appear, and so on.

Considering our objectives, reimplementing the MLE model is thus a perfect benchmark because it rapidly requires to increase both the size of the environment and the number of agents. Moreover, each additional level requires to manage three additional pheromones. So, scaling up MLE simulations, the first issue is related to the computing resources required for applying the diffusion and the evaporation processes: Each pheromone requires to perform computations for each cell of the grid. So, even if simple, the complexity of these computations is quadratic with respect to the grid's side length.

That is why we decided to test the integration of GPU parts in TurtleKit by first translating the diffusion and evaporation dynamics into GPU code. Indeed, because they are completely decoupled from the agent behavioral model, it is possible to create a GPU module that does not modify the agent model API at all.

IV. The GPU Diffusion Module

A. GPU Translation for the Evaporation Process

To explain how these computations have been translated, we now focus on evaporation because it is the simplest one. The evaporation of a pheromone on the grid simply relies on multiplying the quantity which is on each cell by a certain coefficient between 0 and 1 (the evaporation factor). The sequential implementation of this dynamic could be as follows:

Algorithm 1 evaporation($cells, width, height, evapCoef$)

for $i = 0$ to $width$ **do**
  for $j = 0$ to $height$ **do**
    $cells[i][j] \leftarrow cells[i][j] * evapCoef$
  end for
end for

Before presenting the corresponding translation, let us first explain how GPU code is designed so that it could be executed on a GPU device. Roughly, a GPU device is able to proceed the parallel execution of a procedure, namely a *kernel*, by numerous threads. These threads are organized in blocks, which are themselves organized in a grid of blocks. Each thread notably has 3D coordinates, x, y and z localizing it within a block, and each block also has three spatial coordinates that localize it within the grid. Moreover, each block has a limited thread capacity according to the hardware in use.

So, considering only the 2D coordinates of the blocks and threads, it is possible to define a 2D grid of threads that maps a concrete 2D array of data. For instance, if the capacity of a block is 1024 threads, one can work with a grid of 1000×1000 by allocating a grid of blocks which size is 32×32, with each block having a size of 32×32. This produces a global oversized matrix containing 1024×1024 threads. This too large size is not a problem as it will be handled in the GPU code. So, the dimension of the grid and the dimension of the blocks are two fundamental parameters which are used when calling a kernel for execution on the graphic card. In our case, this allows to map each cell of a grid with a unique thread.

So, the evaporation kernel could be programmed as follows:

Algorithm 2 GPU_evap($cells, width, height, evapCoef$)

$i \leftarrow blockIdx.x * blockDim.x + threadIdx.x$;
$j \leftarrow blockIdx.y * blockDim.y + threadIdx.y$;
if $i < width$ **and** $j < height$ **then**
  $cells[i][j] \leftarrow cells[i][j] * evapCoef$
end if

When the execution of this kernel is called on the GPU, all the allocated threads execute the *GPU_evap* procedure. The two first lines of this procedure determine the coordinates of the executing thread. Then a test is done to know if this thread is inside the grid boundaries. If it is the case, the corresponding cell is updated according to its current value.

The diffusion kernel is also easy to derive from its sequential counterpart. So, we produced a GPU module for evaporation and diffusion, called *GPU diffusion module* latter on for simplicity.

B. Results for the GPU Diffusion Module

We compare here the GPU diffusion module with a sequential implementation. Figure 1 shows the results we obtained with tests done outside TurtleKit, thus avoiding noise produced by other treatments. Besides, as we want to keep Java as main language for TurtleKit, we use the JCuda (Java bindings for Cuda[2]) library which allows to call GPU kernels, written in C, directly from Java. The tests have been done using a Intel Xeon CPU @ 2.67GHz, a good CPU and a Nvidia Quadro 4000, an average GPU device.

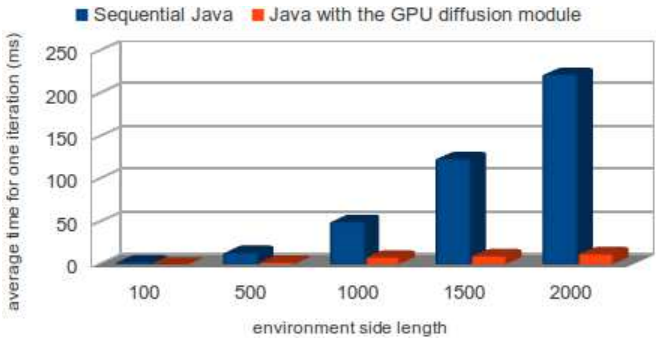

Fig. 1. Diffusion process: Java sequential vs. JCuda

Figure 1 shows the results obtained with different environment sizes for the diffusion and evaporation of one pheromone. Not surprisingly, results show that even for the smallest grid (100×100) the JCuda version performs better. As the environment size increases, the GPU module completely outperforms the sequential version: On an environment of 2000×2000, the GPU module is more than twenty times faster.

With respect to our objectives, in these tests we take into account that the result of one iteration should be made available for use in the Java code, i.e. Java agents should be able to perceive the result at each time step in a real simulation. This requires to call synchronization procedures that synchronize the CPU and the GPU so that they do not modify the data at the same time. If we let the GPU do all the iterations without being interrupted, the GPU then performs more than ten times faster than when using synchronizations.

Other remark, there are a lot of existing Cuda parameters that could be set. Some of them can greatly impact the efficiency of a kernel call. For instance, a general GPU programming rule is to make a kernel call with at least as many blocks as available cores on the graphic card. While not mandatory, not considering this rule can dramatically reduce the efficiency of a kernel call.

Therefore, looking at these results, one has to keep in mind that this is only what is obtained in the context of our particular software and hardware configuration. Depending on the configuration, it is possible to obtain very different results in terms of ratio. Here, the most important information is that the GPU diffusion module does scale very well while the sequential one does not at all.

V. The GPU Field Perception Module

A. Next Bottleneck: The Agents

The integration of the GPU diffusion module within TurtleKit was not a problem thanks its modularity: It only

[2]Compute Unified Device Architecture, Cuda is the programming framework for Nvidia GPU graphic cards.

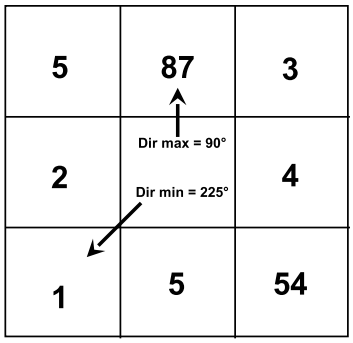

Fig. 2. Example of field min and max directions for a cell

concerns environmental dynamics. Besides, the enhancement provided by the module was easy to observe. So, when we experimented this GPU module on the MLE model, we were able to scale up the size of the environment to values that were out of reach before.

However, as previously explained, obtaining higher level structures for the MLE model requires to simultaneously increase both the size of the environment and the number of initial agents. And as the number of agents was increasing, we observed that most of the execution time was now used for agent-level computations.

Profiling the execution of the MLE model, we easily found out that the agents use most of their (CPU) time computing how they should move with respect to pheromone field gradients. More precisely, each agent has to know the direction of the neighboring cell having the smallest/greatest quantity for a particular pheromone to decide the heading of its next move: Methods such as *getMaxDirection(attractionField)* or *getMinDirection(repulsionField)* are intensively used by the MLE agents. Such computations requires to probe all the cells around the agent one time per pheromone of interest, and then to compute the direction of the minimum and/or maximum values. Figure 2 shows an example of this computation for one cell (the east direction corresponds to 0 degree).

In the present case study, the diffusion and the evaporation processes are the only environmental dynamics used in the MLE model. As we already successfully translated them, it is obvious that the next GPU module to try should take care of the agents. However, sticking to our priority of keeping the programming accessibility of the agent model implementation, a solution that does not imply an all-in-GPU approach has to be found. The next section presents the solution we use and shows how it take advantages of the GPU for the agents while ensuring the stability of the agent API.

B. Delegating Agent Perceptions to the Environment

Considering how MLE agents perceive and analyze pheromone fields, it should be remarked that the related computations do not involve the state of the agent that triggers the perception. So, these computations always give the same result for a particular time step: Pheromone field gradients are the same whatever the state of the agents.

So, thanks to this independence between the agent's state and these computations, the idea is to do these perceptions using a GPU module. However, at first sight, this would mean to finally translate the agent model within a GPU module because these perceptions are triggered by the agents, not by the environment as it is the case for the diffusion process.

To overcome this difficulty, the proposed solution is to compute these high level perceptions directly in the environment everywhere and every time. Doing so literally reifies all these perceptions as one single environmental dynamic, just like the diffusion process is. In other words, we define a new GPU module representing a new environmental process which will be in charge of computing all the perception results which could be asked by the agents.

This solution may seem counterintuitive for anyone who is not used to GPU programming. Especially because many of the computed results will not be used by the agents. But this is where the specificities of GPU programming come into play. Caricaturing and ignoring the details, making a GPU computation for only one cell takes about the same time as doing it for all the cells.

One can reasonably argue that the previous statement is only a rough approximation of reality: Doing unused computations should be avoided as far as possible. To this end, another solution would be to create and maintain another grid keeping track of the agents' presence so that a thread could test if a computation should be done or not. But in our case it turns out that this solution did perform really poorly: The cost induced by the maintenance of this grid is too high and far greater than when this information is ignored and thus not maintained.

Besides performance, the proposed solution completely decouples this new dynamic from the state of the agents. So this new module is more independent and thus reusable, which is desirable from a software engineering point of view.

Following this solution, we have implemented a GPU module which we call the *GPU field perception module*. Implementing this module, we define a new GPU kernel in the same way we have done for the diffusion module. The main difference is that this new kernel works on three grids of data: One for the actual quantity of pheromone on each cell and two others for stocking the minimum and maximum directions of the field for each cell.

C. Results Obtained with the GPU Field Perception Module

This section compares the results which have been obtained on the MLE model with only the GPU diffusion module and with both modules. Considering the hardware configuration, the experimental setup used for obtaining the presented results is the same as in section IV-B. The results reports on MLE simulations where the maximum level of an agent has been set to 5 so that there are 15 pheromone fields to handle for each time step (45 grids of data). Figure 3 compares the simulation speed for various agent population density and environment size. For instance, for a density of 140% in an environment of 2000×2000, there are 5.6 millions of agents interacting on a gird of 4 millions cells.

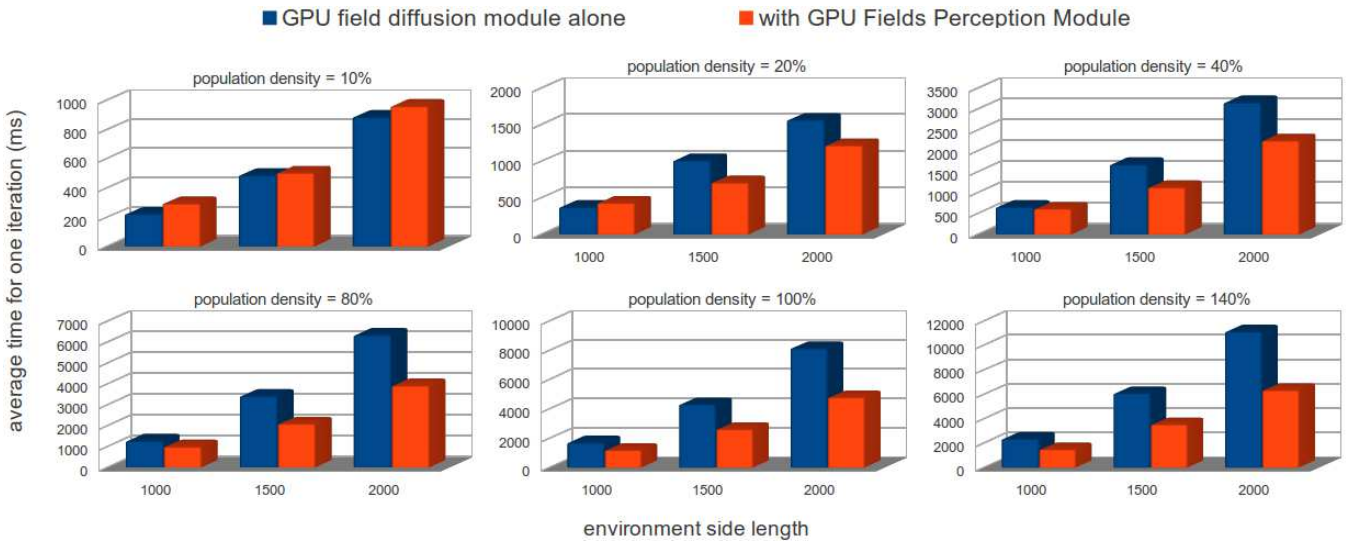

Fig. 3. Comparison of MLE simulations done with and without the GPU field diffusion module

Figure 3 shows that adding the GPU field perception module does not speed up the simulation only for the lowest densities and environment size which are here reported. One partial explanation for this negative result is the overhead induced by the additional synchronizations required between the GPU and the CPU on the different stocked grids. This also shows that, with our particular configuration, there is of course a threshold under which it is not worth to trigger a GPU kernel because of synchronizations, the worst case being only one agent in a huge environment. And indeed, when simulating this worst case the simulation is always slower. However, in our tests this overhead was not very high for densities around 5% and becomes negligible for the 10% density, which is a low value with respect to the MLE benchmark for instance.

So, even with the overhead induced by the additional GPU calls, the field perception module becomes efficient when the agent population density is 20% on the largest environments. Then, the simulation is always faster with the GPU field perception module, especially it is about two times faster for the biggest cases. Considering the fact that we used an average GPU device, these results are really promising and at least show the feasibility and the interest of the proposed approach, especially with respect to scalability.

VI. Generalizing the Delegation Strategy

A. Environment, First Order Abstraction in MAS

From a high level perspective, the proposed solution relies on transforming agent-level perceptions into environmental dynamics. This makes the environment an even more crucial entity in the design of our ABM. So, generalizing our work, the proposed strategy could be related to other research works that consider the environment as a core concept of MAS.

Considering the environment as a first order abstraction in MAS is today well accepted and has proved to be a relevant approach for modeling and developing MAS [12]. Especially, it could help to enhance the efficiency of agent interactions. For instance, in [13], real-world unmanned vehicles (AGVs) use a virtual environment which is in charge of validating their future moves. When detecting a possible future collision, the environment prioritizes the different moves and thus automatically solve spatial conflicts. Doing so, the agents do not have to handle this problem on their own, which allows to (1) decrease the complexity of the agent behavior and (2) make the agents focus on their real task which is to go from point A to B.

More generally, using the environment as an active entity is very interesting for simplifying the behavioral process of the agents. The underlying idea is that agents are in fact usually not interested in low level environmental properties but rather in high level percepts. So, it makes sense to let the environment do the work of producing high level percepts from raw environmental data. Such an approach allows to design MAS with a clear separation of concerns [14].

B. GPU Environmental Delegation of Agent Perceptions

In the scope of our work, considering the environment as a first class entity is the heart of the solution. This enables us to reach our two requirements: (1) keeping the programming accessibility of the agent model in a GPU context and (2) being able to scale up both the number of agents and the size of the environment.

From this developing experience, we derive and propose a design guideline which (1) follows the idea of an active environment and (2) takes into account the context of GPU programming. This guideline, namely *GPU Environmental Delegation of Agent Perceptions*, could be stated as follows:

> Any agent perception computation not involving the agent's state could be translated to an endogeneous dynamic of the environment, and thus considered as a potential GPU environment module.

Such a guideline does not only follow the idea of considering the environment as a first order abstraction, but more importantly also focus on easing re-usability of developed

GPU modules. For instance, in our case we are able to directly use the developed GPU modules with other agent models working on pheromone fields such as ant-based ones.

Therefore we argue that such an approach could help to address the re-usability issue by promoting the development of more generic GPU modules. Indeed, such modules only deal with environmental dynamics and high level information (e.g. perceptions) that do not rely on a particular agent model, but only on a particular model of environment. Additionally, this guideline suggests a more fine grained approach for integrating GPU modules which eases the development and maintenance tasks thanks to a clear separation of concerns.

Considering applicability, for now we explicitly limits the scope of our approach to computations that do not involve agent states. Regarding more complex perceptions, obviously it will not be always possible to find an equivalent environmental model but addressing this limitation should increase the scope of the approach and is in our future research plans.

Finally, we want to emphasize that the interest of GPU environmental delegation is not restricted to our objectives. Indeed it could be also considered when applying an all-in-GPU approach for MAS because its main point is to promote re-usability in the particular context of GPU programming.

VII. Conclusion and Perspectives

Firstly, concerning the integration of GPU modules within TurtleKit, this paper shows how our approach enables us to use the power of GPU devices without changing the agent API, thus fulfilling our primary objectives.

Still, solely based on the experiments we have done so far, it should be remarked that our results are far from being as fast as systems applying an all-in-GPU approach. Obviously, there is still a trade-off to accept for keeping the agent model so that its implementation is safe from the difficulties and specificities of the GPU programming.

But, let us remind again that the presented results are deeply related to the configuration used. Especially, we use a NVidia Quadro 4000 containing 256 cores while the recent NVidia Tesla K10 contains 2 GPUs with 1536 cores each: A total of 3072 cores on only one card. So, the perspectives offered by GPU programming are really promising and encourage us to keep on going with the presented approach for TurtleKit. Especially, we plan to benchmark other models to identify other GPU modules using GPU environmental delegation.

Secondly, we advocated that GPU environmental delegation represents an interesting design guideline for tackling the re-usability issue in the context of GPU programming for MAS. Indeed, on the one hand, it is obvious that GPU programming will be a great help for designing MAS-based complex systems. However, on the other hand GPU programming is so specific that almost all the development efforts done in this modeling scope are simply lost. The complexity of the programs which are obtained is too high. We argue that using such a guideline should help to produce GPU modules that will be more easily reusable, precisely because they will be disconnected from any agent model.

So, one of our long term goals is to develop a library of GPU modules implementing various environment dynamics specificaly designed for spatial ABMs. Today there are several generic GPU libraries which have been produced such as *Nvidia CuBLAS* (Compute Unified Basic Linear Algebra Subprograms), *NPP* (Nvidia Performance Primitives) for image, video, and signal processing, GPU AI path finding, etc. We think that applying such a modular approach in the scope of MAS-based complex systems is the way to go.

References

[1] 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*, vol. 68, no. 10, pp. 1370 – 1380, 2008, general-Purpose Processing using Graphics Processing Units.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] F. Michel, J. Ferber, and A. Drogoul, "Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective," in *Multi-Agent Systems: Simulation and Applications*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, Danny Weyns and Adelinde Uhrmacher, Eds. CRC Press - Taylor & Francis, 05 2009, pp. 3–52.

[4] 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*, vol. 11, no. 4, p. 10, 2008. [Online]. Available: http://jasss.soc.surrey.ac.uk/11/4/10.html

[5] F. Michel, G. Beurier, and J. Ferber, "The TurtleKit simulation platform: Application to complex systems," in *Workshops Sessions, First International Conference on Signal & Image Technology and Internet-Based Systems SITIS' 05*, A. Akono, E. Tonyé, A. Dipanda, and K. Yétongnon, Eds. IEEE, november 2005, pp. 122–128.

[6] 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 *Proceedings of the 4th international conference on Motion in Games*, ser. MIG'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 304–315.

[7] R. M. D'Souza, M. Lysenko, S. Marino, and D. Kirschner, "Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units," in *Proceedings of the 2009 Spring Simulation Multiconference*, ser. SpringSim '09. San Diego, CA, USA: Society for Computer Simulation International, 2009, pp. 21:1–21:12.

[8] A. R. D. Silva, W. S. Lages, and L. Chaimowicz, "Boids that see: Using self-occlusion for simulating large groups on gpus," *Comput. Entertain.*, vol. 7, no. 4, pp. 51:1–51:20, Jan. 2010.

[9] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, 2010.

[10] E. Sklar, "Netlogo, a multi-agent simulation environment," *Artificial Life*, vol. 13, no. 3, pp. 303–311, 2007.

[11] G. Beurier, O. Simonin, and J. Ferber, "Model and simulation of multi-level emergence," in *2nd IEEE International Symposium on Signal Processing and Information Technology, ISSPIT'02*, Marrakesh, Morocco, December 2002, pp. 231–236.

[12] D. Weyns, A. Omicini, and J. Odell, "Environment as a first class abstraction in multiagent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 14, pp. 5–30, 2007, 10.1007/s10458-006-0012-0.

[13] D. Weyns, N. Boucké, and T. Holvoet, "Gradient field-based task assignment in an agv transportation system," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ser. AAMAS '06. New York, NY, USA: ACM, 2006, pp. 842–849.

[14] P. H. Chang, K.-T. Chen, Y.-H. Chien, E. Kao, and V.-W. Soo, "From reality to mind: A cognitive middle layer of environment concepts for believable agents," in *Environments for Multi-Agent Systems, First International Workshop, E4MAS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, ser. LNAI, D. Weyns, H. V. D. Parunak, and F. Michel, Eds., vol. 3374. Springer, 2005, pp. 57–73.