Using GPU for Multi-Agent Multi-Scale Simulations

G. Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe

Abstract Multi-Agent System (MAS) is an interesting way to create models and simulators and is widely used to model complex systems. As the complex system community tends to build up larger models to fully represent real systems, the need for computing power raise significantly. Thus MAS often lead to long computing intensive simulations. Parallelizing such a simulation is complex and it execution requires the access to large computing resources. In this paper, we present the adaptation of a MAS system, Sworm, to a Graphical Processing Unit. We show that such an adaptation can improve the performance of the simulator and advocate for a more wider use of the GPU in Agent Based Models in particular for simple agents.

1 Introduction

Scientists pay more and more attention to models. They aim at simulating real complex systems to understand them, for example to prevent phenomena such as disasters. Models thus (such as Agent Based Models -ABM-) become more and more descriptive: where scientists used to built up conceptual models (KISS model), they now create well described models (KIDS Model) [1]. Tomorrow the complex system community may tend to "world model", a fully descriptive and generic model that scientists can customize according to scientific questions [2]. Making these modeling evolutions possible requires a raise of computing power as the model executions must return a result in a reasonable delay.

If a simulation can be parallelized then increasing the computing power becomes a question of cost: we need to buy more cores to increase the size of the simulation. While standard processors (CPU) are still expensive, modern Graphical Processing Units (GPU) provide good execution performance for a lower cost. These GPUs also give the possibility to execute non-graphic programs using languages such as OpenCL or CUDA.

Typical simulators are however based on a sequential design and only use one core of the main processor. So taking advantage of the computer architecture implies to develop distributed simulators. In this context, Multi-Agent System (MAS) is an

Nicolas Marilleau
Institut de Recherche pour le Développement (IRD), France, e-mail: nicolas.marilleau@ird.fr

Christophe Lang · Guillaume Laville · Kamel Mazouzi · Laurent Philippe
Institut FEMTO, CNRS / Université de Franche-Comté, France, e-mail: name.surname@univ-fcomte.fr

interesting way to create this kind of models and simulators. ABM are indeed often used to simulate natural or collective phenomenons whose actors are too numerous or various to provide a unified algorithm describing the system evolution.

We propose in this paper to delegate part of the ABM execution to the graphical unit of the computer, based on our experience in parallelizing and implementing part of an ABM on a GPU. We then run it on both a standard CPU processor and on a GPU. We get very good performance results and advocate for a more wider use of the GPU in ABM in particular for simple agents. In Section 2 of the paper we present the work related to agent and parallelization. We give an overview of the sworm simulator in Section 3 and we detail its GPU implementation in 4. We present the experiments and results in Section 5 then we conclude on the possible generalization of our work.

2 Related works

To develop ABM, many frameworks are now available (e.g. Repast [3] or NetLogo [4]). Only a few of them introduces distribution in agent simulation (Madkit [5] or MASON [6]). In this context, parallel implementations are often based on threads using shared memory or on the integration of cluster libraries such as MPI. Even as multi-core or multi-socket setups become more and more readily available, these solutions stay limited to a relatively small number of parallel tasks, if not using a fully-fledged computing cluster.

Parallelizing a simulation is however complex as space and time constraints must be enforced. Time constraints are linked to the simulation execution. It is usually based on a synchronous execution of time steps by the agents and the environment. Distributing the simulation or delegating part of this execution to others processors as GPUs [7] must thus be carefully done to enforce the synchronism [8]. Space constraints are linked to the environment distribution. Using an environment on a set of computers leads to classical parallelism issues: data coherency if the environment is shared amongst all the computers or data exchanges if the environment is distributed. In the particular case of multi-scale simulations such as the Sworm simulation [9] the environment may be used at different levels. This characteristic, especially in a fractal model, could be the key of the distribution. For instance, each branch of a fractal environment could be identified as an independent area and parallelized. In addition Fractal is a famous approach to describe multi-scale environment (such as soil) and its organization [10].

3 Sworm

Sworm model aims at simulating soil functioning especially the role of macro-fauna on microbial activity and on carbon production. A first version presented in [9] focuses on soil bioturbations caused by earthworms. Soil zones, which have been

Algorithm 1 Sequential evolution algorithm

```
for all mm ∈ mmList do
  breathNeed ← world.respirationRate * mm.carbon
  growthNeed ← world.growthRate * mm.carbon
  if totalAccessibleCarbon(mm) > breathNeed then
    mm.active ← true
    consumCarbon(mm, breathNeed)
    world.CO2 ← world.CO2 + breathNeed
    if totalAccessibleCarbon(mm) > 0 then
      growthConsum ← max(totalAccessibleCarbon(mm), growthNeed)
      consumCarbon(mm, growthConsum)
      mm.carbon ← mm.carbon + growthConsum
    end if
  else
    mm.active ← false
  end if
end for
```

modified by these earthworms, are considered by biologist as hot spots of a microbial activity (an important process that transforms organic matters into carbon and so gives the fertility to the soil). The MIOR -MIcro-ORganisms- model tackles reproducing the microbial activity of these hot spots. MIOR focuses on a nearby 0.002 mm sized cube of soil while Sworm models 20 cm sized volume of soil.

Since these different models do not act at the same level of detail, this simulation is split into multiple, recursive levels corresponding to different scales. It uses a fractal pattern which allows lazy environment allocation: the different kinds of agents do not work at the same scale, depending of their respective sizes. As an example, worms tend to interact on a macroscopic scale (few millimeters), whereas microbial colonies work at a microscopic scale (10^{-4} millimeters). Figure 1 illustrates the fractal data representation of Sworm. White cells (in the left part of the cell) represent soil cavities, black cells represent mineral matter and gray ones represent composite cells that must be further decomposed..

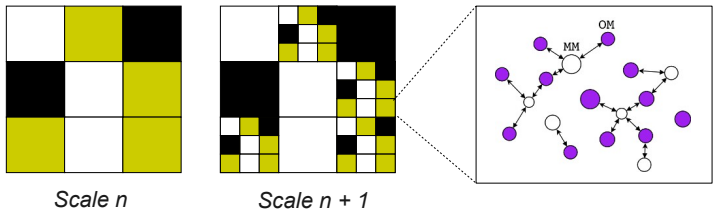

Fig. 1 Sworm environment representation

A MIOR simulation is based on two types of agents. MMs -Meta-MIOR- represent colonies of carbon-consuming microbes naturally found in soil. OMs -Organic Matters- are deposits that represent the base resources randomly distributed in the modeled environment. Two main processes account for the model evolution, repre-

senting the MM metabolism: breathing (conversion of mineral carbon to CO_2) and growth (fixation of carbon into microbial colonies). This second process only occurs if the colony needs are covered by the environment, as described in Algorithm 1.

Even with the sparse fractal representation, simulating the microscopic-scale evolution quickly becomes time-consuming on a traditional CPU unit, where much execution resources are already required for the macroscopic simulation (nearly one week per simulation on a personal computer).

4 GPU Implementation

CUDA and OpenCL are the main programming languages for GPU scientific programming. They are based on a C-like syntax with additions to support GPU-specific features. Various open-source bindings, such as JOCL [11] and JCUDA [12] exploit GPUs with the Java language. Since our implementation must be integrated in a Java-based implementation and must be portable JOCL is a natural choice.

Sworm requires two species of agents, cognitive ones (representing worms) and reactive ones (MIOR microbial colonies). The two kinds are not as good candidates for a GPU implementation. Reactive agents tend to rely on data parallelism with few, if any, kind of branching in the code to be executed. On the opposite, cognitive agents have a variety of possible behaviors at each iteration.

Our work therefore mainly focuses on the adaptation of the MIOR part of the Sworm simulation to a GPU architecture. Since the environment modeling and evolution if often a costly aspect in many ABM, theses changes can provide a tangible benefit on the simulator executions. By using simultaneously the GPU for the environment execution and the CPU for the cognitive agents we also improve the quality of modeling (replacement of the precomputed static carbon evolution graph by dynamically computed ABMs in our case).

4.1 OpenCL representation

Java can only be used to control the model initialization and the OpenCL executions. So a mapping of the Java Object-Oriented concepts to the C-like data structures of OpenCL is required to access and manipulate them in the GPU code. Here are some guidelines that can be used in this mapping process. The data aspect (type, attributes) can be represented as OpenCL structures. Methods can be implemented as device-local functions, and each phase of the simulation can have its own high-level function calling them. Consecutive function calls with the same dimensions and subjects may be gathered into the same kernel (set of threads) to minimize the number of context switches between host and GPU.

By applying these mapping rules, four main data structures appears in the MIOR simulator: one array of MM structures for microbial colonies, one array of OM structures for organic deposits, one neighboring matrix used to store accessibility

between MM and OM and one single environment used to store global parameters such as agent numbers and output data such as the quantity of CO_2.

The design of the computing topology is of importance for the parallelism of the execution. Due to their simple behavior the reactive agents usually have limited interactions and thus allow a high level of parallelism. In our case, the action radius of each MM agent is determined by a fixed parameter of the environment, RA. All organic matter deposits within this distance should be accessible for the metabolism.

Since there are no dependencies between each distance calculation, we can use a two dimension kernel in OpenCL where each thread represents an (OM, MM) couple. This approach can be generalized with n dimension kernels in OpenCL depending on the modelized environment and on the GPU characteristics (usually one to three dimensions).

4.2 Data Representation and Data Dependencies

The limited interactions between reactive agents also impact the data representation as the density of interaction matrices may be low. In a typical MIOR simulation each agent is linked to a small (less than 10) number of OM on a total of some hundredth. The topology computation thus produces a low-density matrix. During the following evolution process, going through these matrices may induce numerous costly global memory accesses without actual use of the information. This issue can be addressed by creating dense continuous representations of the matrices. So on the topology stage we create two dense continuous representations of the neighboring matrix. The additional memory cost of these two data structures (one for the OM, one for the MM) can thereafter be balanced thanks to the model living steps.

Data dependency is a much difficult issue as it usually depends on the ABM characteristics. Some guidelines may be given but good results can only be obtained thanks to experience. Most of the time, as in our case, an algorithm adaptation will be needed. We present two cases of algorithm adaptation in this section.

In the MIOR simulation the data dependencies comes from the need to ensure fair access to carbon deposits for each microbial colony and thus to synchronize these accesses across the simulation. Since the colonies are randomly placed at the model initialization, no obvious geographic static locality can be extracted to split the carbon deposits in local memories for the sets of microbial colonies (MM). Therefore much data has to remain into the global slow GPU memory. A first implementation based on mutual execution and global memory synchronization thus resulted in GPU performances two order of scale slower than a sequential Java simulation. This clearly indicated the need for an algorithm adaptation. Our solution is to reduce the number of synchronization required for the execution of a given step of simulation. For that we split the living cycle process into three steps:

1. Scattering: carbon deposit resource is evenly scattered in parts across all microbial colonies accessing to it,
2. Living: each microbial colony consumes carbon for its breathing/growing process and produces CO_2,

3. Gathering: each carbon deposit content is recomputed from the remaining carbon in each part.

There are also synchronization issues in the breathing and growth part of the initial algorithm. Since multiple MM can share a OM carbon deposit, two synchronization issues must be addressed to avoid introducing a bias in the simulation. First, a synchronization on the resources is needed as multiple MM agents should not modify the OM's properties at the same time. Second, the fair access to resources from the MM must be enforced and no OM should be over-exploited.

4.3 Data transfer between Host and GPU

Since the reference algorithm is sequentially implemented, a synchronization of the simulation steps must be implemented to prevent some agents to evolve faster than their peers. A first naive implementation may attempt to copy all modified data from and to Java objects between each simulation step. Data copy between the CPU system and the GPU-dedicated memory however uses a connexion limited to a fraction of the main memory bandwidth and with a much higher latency. So an obvious course of action to improve this is to allow the simulation to run several steps at once, or up to the termination avoiding these costly transfers. The only exception must be for values needed to detect simulation termination or convergence. The better improvements are achieved in the case where only the final state of the system is needed.

4.4 Proposed implementations

We propose three consecutive GPU implementations of the MIOR living process with an increased adaptation level. The first implementation, further referenced as GPU v1.0, is based on a straightforward adaptation of the CPU algorithm where the relations between agents are stored as a simple two-dimensions sparse matrix located in global memory. Each simulation step iterates over the whole matrix (including empty cells). The second GPU implementation, called GPU v2.0, replaces this matrix by a compressed one using the same representation format as described in [13]: neighbors are stored in a contiguous way in the matrix along with a precomputed count of neighbor agents. This representation reduces the required number of iterations and of memory accesses. To minimize global memory access overhead, the third GPU implementation, referenced as GPU v3.0, uses the GPU private memory to store often-used data such as carbon parts and MM neighboring information.

5 Experiments

To assess the performance of our work we compare the sequential version with our MIOR implementations on two platforms, representing what a researcher could

except as dedicated computing hardware or personal computer graphic card. The first platform is a GPU node with two Intel Xeon X5550 CPU at 2.67GHz and a Tesla C1060 GPU card running at 1.3GHz (240 cores organized in 30 streaming multiprocessors). The second platform is a personal computer with an Intel Q9300 CPU at 2.5GHz and a mainstream GPU card: a GeForce 8800GT at 1.5GHz (112 cores organized in 14 streaming multiprocessors).

Figures 2 and 3 give the average execution time of 50 simulations. The problem size is given by the scaling factor. A scaling factor of 6 means that the number of agents is multiplied by 6. At scale 1, the model contains 38 MM and 310 OM.

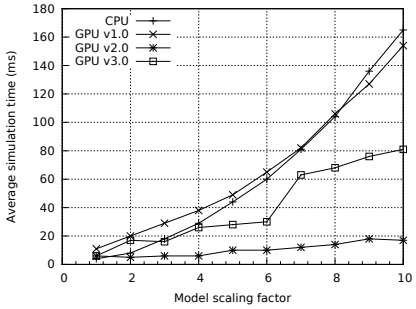

Fig. 2 CPU and GPU executions on a Tesla C1060

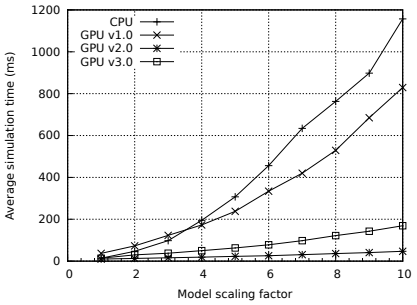

Fig. 3 CPU and GPU executions on a GeForce 8800GT

We can note that small sized problems result in similar execution times across all the implementations. The GPU simulations do not have enough agents to benefit from the algorithm's parallelism. From scale 5 GPU versions v2.0 and v3.0 are clearly faster than the CPU or the simple GPU implementations. At scale 10, a ratio of 10 can be observed between the v3.0 GPU implementation and the CPU one.

Several important remarks must be reported. First, the optimized GPU versions give much better performance than the non-optimized GPU one. So it really matter to work on the algorithm parallelization. Second, even a mainstream desktop graphic card provides the same order of performance gain that a much more costly solution, the Tesla card. This is rather interesting as a much larger public could benefit from GPU adaptations. Third, the same code is run on the GeForce card as on the Tesla card without any modification. So the performance does not depend on the GPU characteristics and the same benefit could be expected on other graphic cards.

6 Conclusion

In this paper we have described an adaptation of an existing ABM simulation using GPU hardware. The first result of this work is that adapting the algorithm to a GPU architecture is possible for ABMs. This adaptation may provide a significant perfor-

mance improvement without so much effort, or at least not much than for a standard multi-threaded parallelization. We have shown that this approach especially suits the case of multi-scale ABMs. We also have generalized this work to show the main issues to be addressed when parallelizing an ABM and how to take benefit from the GPU architecture. Last, we show that using a mainstream card, as the GPU card of a standard computer, can even lead to a significant performance improvement and avoid the use of a costly parallel cluster.

We are currently working on defining more general guidelines for adapting SMA simulations to the GPU architecture. The aim is to use both CPU and GPU at the same time to support multi-scale ABMs. With cognitive agents being run on the CPU and reactive agents or environment being run on the GPU, we could run larger simulations thanks to the performance improvement provided by this specific architecture. In this perspective, we are working on the design of a generic library to support reactive agent simulation on graphic cards.

Acknowledgements Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

References

1. B. Edmonds and S. Moss. "from kiss to kids: An" anti-simplistic" modeling approach". In *MABS 2004*, pages 130–144, 2004.
2. E. Amouroux. *"KIMONO: using the modelling process as an aid for research orientation"*. PhD thesis, UPMC, Paris, France, 2011.
3. M.J. North, T.R. Howe, N.T. Collier, and J.R Vos. A declarative model assembly infrastructure for verification and validation. In Springer, editor, *Advancing Social Simulation: The First World Congress*, Heidelberg, FRG, 2007.
4. E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311, 2011.
5. O. Gutknecht and J. Ferber. Madkit: a generic multi-agent platform. In *Proceedings of the fourth international conference on Autonomous agents*, AGENTS '00, pages 78–79, New York, NY, USA, 2000. ACM.
6. L. Sean, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multi-agent simulation environment. *Simulation: Transactions of the society for Modeling and Simulation International*, 82(7):517–527, 2005.
7. Avi Bleiweiss. Multi agent navigation on the gpu. *GDC09 Game Developers Conference 2009*, 2008.
8. N. Marilleau, C. Lang, P. Chatonnay, and L. Philippe. An agent based framework for urban mobility simulation. In *PDP*, pages 355–361, France, 2006.
9. E. Blanchart, N. Marilleau, A. Drogoul, E. Perrier, JL. Chotte, and C. Cambier. Sworm: an agent-based model to simulate the effect of earthworms on soil structure. *EJSS. European Journal of Soil Science*, 60:13–21, 2009.
10. N Bird and E. Perrier. The psf model and soil density scaling. *European Journal of Soil Science*, 54(3):467–476, 2003.
11. JOCL: Java bindings for OpenCL. `http://www.jocl.org/`. [11-oct-2011].
12. JCUDA: Java bindings for CUDA. `http://www.jcuda.org/`. [11-oct-2011].
13. J. Gómez-Luna, J.-M. González-Linares, J.-I. Benavides, and N. Guil. Parallelization of a video segmentation algorithm on cuda—enabled graphics processing units. In *15th Euro-Par Conference*, pages 924–935, Berlin, Heidelberg, 2009. Springer-Verlag.