Defining a Methodology Based on GPU Delegation for Developing MABS using GPGPU

Emmanuel Hermellin[1] and Fabien Michel[1]

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

Abstract. Multi-Agent Based Simulation (MABS) is used to study complex systems in many research domains. As the number of modeled agents is constantly growing, using General-Purpose Computing on Graphics Units (GPGPU) appears to be very promising as it allows to use the massively parallel architecture of the GPU (Graphics Processing Unit) to do High Performance Computing (HPC). However, this technology relies on a highly specialized architecture, implying a very specific programming approach. So, to benefit from GPU power, a MABS model need to be adapted to the GPU programming paradigm.
Contrary to some recent research works that propose to hide GPU programming to ease the use of GPGPU, we present in this paper a methodology for modeling and implementing MABS using GPU programming. The idea is to be able to consider any kind of MABS rather than addressing a limited number of cases. This methodology defines the iterative process to be followed to transform and adapt a model so that it takes advantage of the GPU power without hiding the underlying technology. We experiment this methodology on two MABS models to test its feasibility and highlight the advantages and limits of this approach.

Keywords: MABS, GPGPU, Methodology, GPU delegation

1 Introduction

Using Multi-Agent Based Simulation (MABS), computing resources requirements often limit the extent to which a model could be experimented [16]. Considering this issue, General-Purpose computing on Graphics Processing Units (GPGPU) is a relevant way of speeding up MABS. Indeed, Graphics Processing Unit (GPU) is an excellent computational platform which is able to perform general-purpose computations [15]. GPGPU relies on using the massively parallel architecture of usual PC graphics cards for accelerating very significantly the performance of programs[1] [4].

Still, implementing MABS using GPGPU is very challenging because GPU programming relies on a highly specialized hardware architecture [18, 1]. Based on the SIMD (*Single Instruction, Multiple Data*) parallel computing model, also

[1] *e.g.* https://developer.nvidia.com/about-cuda

called Stream Processing, which consists in executing simultaneously a series of operations on a dataset, an efficient GPU implementation requires that the MABS is modeled by means of distributed and independent data structures. Moreover, usual object oriented features, which are very common in Agent-Based Model (ABM), are no longer available using GPGPU [3].

Among research works that aim at enabling the use of GPGPU in a MABS context, most of them release dedicated tools and frameworks which integrate GPGPU through a transparent use of this technology (*e.g.* [21]). However, doing so, such approaches have to abstract many parts of the MABS models and thus handle only specific cases, while there exists a wide variety of MABS models.

In [8], we have studied the relevance of directly using GPGPU (transform or adapt a model) and promoted the idea that a dedicated methodology would be a valuable contribution to the field. Especially, the purpose of such methodology would be twofold: (1) helping potential users to decide if they could benefit from GPGPU considering their models and (2) describing the modeling and implementation process of MABS models without hiding GPU programming.

From a Software Engineering (SE) perspective, this paper details the methodology extracted from the experiment presented in [8] and the development aspects related to this solution. Then, we test this methodology on two models to highlight the advantages and limits of such an approach. Section 2 presents the evolution of the use of GPGPU in MABS. Section 3 describes the methodology which is proposed in this paper. Section 4 experiments the methodology on two models. Section 5 concludes this paper by listing the advantages and limits of the proposed methodology and outlines planed improvements.

2 Related Works and Motivations

Initially designed for graphics rendering, GPU are now able to perform general-purpose computations. The associated programming paradigm consists in executing simultaneously a series of operations on a dataset. When the data structure is suitable (and only if), the massively parallel architecture of the GPU can provide very high performance gains (up to thousands of times faster) [4]. Empirical results from various experiments in a MABS context show that high simulation speeds can be achieved especially with very large agents populations [6]. However, this excellent speedup comes at the expense of modularity, ease of programmability and reusability [18].

The release of CUDA[2] and OpenCL[3] have simplified GPGPU and greatly contributed to increase the number of MABS using this technology. Flame GPU [21] is a flagship example of the possibilities offered by the rise of specialized GPU programming tools for MABS: It is a ready-to-use solution for creating and simulating MABS using GPGPU.

[2] Compute Unified Device Architecture, *e.g.* `https://developer.nvidia.com/what-cuda`

[3] Open Computing Language, *e.g.* `http://www.khronos.org/opencl`

Nonetheless, the existing frameworks are still difficult to reuse and target only a limited number of MABS use cases. Therefore, most of the new research works still start from scratch and put all their attention on acquiring the best computational gains without considering the accessibility, reusability and modularity aspects.

Moreover, as pointed out in [1], implementing a model using GPGPU does not necessarily imply an increase of performance, notably in the field of MABS where many different and heterogeneous architectures can be conceived. Indeed, achieving an efficient implementation requires to take into account the specific programming model that comes with GPU. Therefore, most of MABS using GPGPU are realized in an ad hoc way and only represent one-off solutions.

Until 2011, the most used approach to implement MABS with GPGPU consisted in executing completely the model on the GPU. Called here all-in-GPU, this approach is useful when the main objective is only to accelerate the simulation. But from a software engineering point of view, it is not adapted because all development efforts are lost. Indeed, all-in-GPU implementations are very specific and therefore cannot be reused in other contexts. This is especially true in the scope of works that address the study of flocking [7], crowd [20], traffic simulations [23] or autonomous navigation [2].

Considering these issues, hybrid approaches have been proposed and represent a very attractive alternative because they consist in sharing the execution of the MABS between the CPU and the GPU. Despite the fact that an all-in-GPU implementation is more efficient than an hybrid one, the latter has two main advantages. Firstly, hybrid approaches enable a step further toward more complex MAS models because one can choose what is executed on the GPU according to the nature of the computations (*e.g.* [12, 11]). Secondly, by removing the programming constraints related to all-in-GPU systems, hybrid approaches are by definition more flexible and open to other technologies [12, 13], which in turn brings greater modularity and reusability (*e.g.* the explicit implementation distinction between the agents and the environment in [13, 17]).

So, from this overview, works dealing with GPGPU in MABS can be divided into two categories: (1) works that are only interested in performance gains, and which are hardly reusable and (2) works that take into account aspects related to modularity, genericness, reusability and accessibility. However, works from the later category mostly rely on hiding the use of GPGPU through predefined programming languages or interfaces which are based on specific agent and environment models (*e.g.* [19]). Even though they represent concrete solutions for easing the use of GPGPU for MABS, such approaches cannot take into account the wide variety of MABS which can be conceived because they rely on predefined software structures and conceptual models [8].

Consequently, instead of hiding GPGPU, we here argue on the idea that it would be interesting to provide the MABS community with a methodology that would concretely help to adapt and implement a MABS model using directly GPU programming. This would allow to take into account a larger number of models because such an approach would not rely on a predefined agent model and

implementation. This paper presents the methodology on which we are working according to this objective.

3 Defining a GPU Methodology Dedicated to MABS

3.1 The GPU Delegation Principle

The GPU delegation principle [13] is based on the fact that it is very difficult to deport the entire MABS model on graphics cards. Inspired by an Agent-Oriented Software Engineering (AOSE) trend which consists in using the environment as a first class abstraction in MAS [24, 25], GPU delegation uses an hybrid approach which divides the execution of the MAS model between the CPU and the GPU. Especially, this principle consists in making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU.

To this end, the design guideline underlying this principle is to identify agent computations which can be transformed into environmental dynamics and thus implemented into GPU modules (called *kernel*, these modules contain the computations executed on the GPU). The GPU delegation principle can be stated as follows: *Any agent perceptions and computations that do not modify the agent's states could be translated to an endogenous dynamic of the environment, and thus considered as a potential GPU environment module.*

3.2 Objectives of the GPU Delegation Methodology

As previously mentionned, using GPGPU in the context of MABS remains difficult mainly because of accessibility and reusability issues. In this context, [10] has proposed an overview of several case studies on using the GPU delegation principle for adapting MABS models to GPU programming. Moreover, the various practical results obtained with this approach are detailed and discussed. Especially, all these experiments [13, 9, 8] showed that this approach is an original and relevant solution which can be generalized in a methodology.

Furthermore, this methodology is different from other developed solutions because it does not hide the used technology and it puts forward a modular iterative modeling process focusing on the reusability of created tools. In this context, this methodology intends to reach four main objectives:

1. Simplify the use of GPGPU in the context of multi-agent based simulations by describing the modeling and implementation process to follow;
2. Define a generic approach which can be applied on a wide variety of models;
3. Promote the reusability of created tools;
4. Help potential users to decide whether they can benefit from GPGPU according to their models.

3.3 Definition of the GPU Delegation Methodology

All the experiments carried out within the scope of GPU delegation [10] allow to extract a design methodology based on the GPU delegation principle and divided into 5 distinct phases (illustrated in figure 1). The first step consists in decomposing all the computations which are presents in the model. The second step consists in identifying, among the above listed computations, those which are compliant with the criteria of the GPU delegation principle. The third step consists in checking if the computations identified as compatible with the GPU delegation principle have already been converted into environmental dynamics and therefore if there is a dedicated GPU module that can be reused. The fourth step verifies the compatibility of selected computations with the GPU architecture. The idea is to choose and apply the GPU delegation principle only on computations that will give the best performance gains once translated into GPU modules. The fifth step consists in concretely implementing the GPU delegation principle on computations that respect all previous constraints. So, the workflow of the methodology can be summarized as follows:

1. Decomposing all the computations;
2. Selecting eligible computations according to the GPU delegation criterion;
3. Reusing GPU modules;
4. Evaluating if computations are compatible with GPU architecture;
5. Implementing the GPU delegation.

Step 1: Decomposing Model's Computations This phase consists in decomposing all the computations used in the model. Carry out such a decomposition is interesting because a number of computations present in the model are not explicit. Highlighting all the computations that are used by the agents to perform their behaviors, by decomposing them into the most possible primitive, will help to implement GPU delegation and thus increase its efficiency on the considered model. With this approach, we do not work with one large *kernel* containing all the GPU computations but with many small and simple *kernels* which allows to capitalize on the modular and hybrid aspect of the GPU delegation principle.

So, the more the model is decomposed in simple computations, the more GPU delegation could be then successfully applied. This decompositing of actions was also identified as important in [5], where a new division of the actions of agents limits the concurrent access to data what increases the overall performance of the model using GPGPU.

Step 2: Identifying Compatible Computations The selection of computations is an essential step because it relies on deciding which one respect the criterion of the GPU delegation principle and could benefit from GPGPU. If no part of the model is compliant with the GPU delegation criterion, it is therefore useless to go further because, in such a case, the gains brought by GPGPU

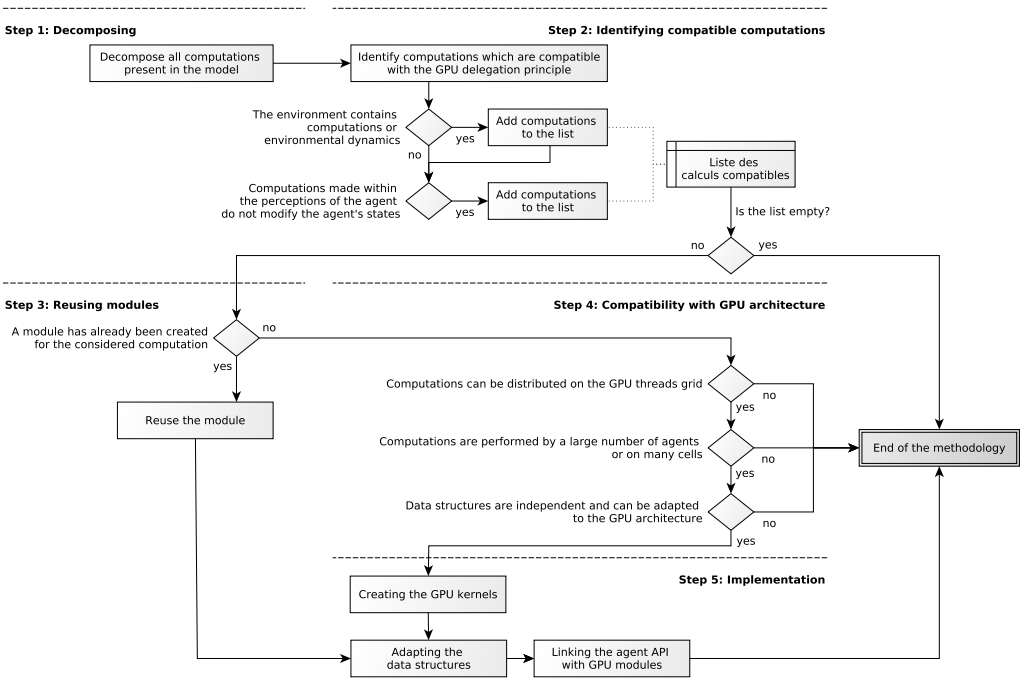

Fig. 1. Diagram of the proposed methodology

could be insignificant or even negative [12]. Moreover, this identification process is different depending on whether the computation is in the environment or in the agent behaviors.

For the environment If the environment is not static and if it contains dynamics, these dynamics must be applied on the entire environment and have a global impact. Indeed, the impact of the dynamics is an important parameter. Take the example of an environmental dynamics which reveals a random amount of food in the environment (at a given position), at each time step of the simulation. This dynamic is well apply to the whole environment but will only have a very localized impact. In this case, translate this dynamic into a GPU module is not justified because the expected gains will be insignificant. Otherwise, if the dynamic has a global impact and respects all specified requirements, the compatibility with the GPU delegation criterion is established and its translation into GPU module is then possible and relevant.

For the agents If computations made within the perceptions of the agent do not modify the agent's states, they could be translated into environmental dynamics and then performed by a dedicated GPU module. The idea is to transform a computation realized locally into an environmental dynamic applied in the whole environment.

Step 3: Reusing GPU Modules One objective of the methodology is to promote the reusability of the created GPU modules. So, given that compatible computations have been identified, it is worth checking if one of the modules created previously could be reused. If this is the case, it is possible to skip to Step 5 in order to adapt the data structures of the computation to correspond with those of the reused module.

Step 4: Computations and GPU Architecture Before applying GPU delegation on the selected computations, it is necessary to evaluate if computations could fit the massively parallel architecture of the GPU. Indeed, the compatibility of a computation with the criterion of the GPU delegation does not necessarily imply an improvement of performances once this principle applied. Under these conditions, an estimate of the expected gains must be carried out to evaluate if the identified computations will bring performance gains in order to not waste time in useless developments. This assessment phase can be achieved by answering three questions:

- *Do identified computations could be distributed on the GPU ?*
  These computations must be independent and simple and do not contain too many conditional tests which can cause problems or slowing down the execution in GPGPU context (*e.g.* divergence of *threads*, [22]). Computations containing iterative loops are better suited to parallel architectures.
- *Do identified computations are performed in a global way ?*
  Because of the very high data transfer costs between GPU and CPU, if computations are rarely used, triggering a GPU computation could be not efficient even if their are compatible with the principle. So, it is necessary to verify that computations are performed by a large number of agents or applied on a lot of cells (for the environment).
- *Do the data structures associated to the identified computations could fit the GPU architecture ?*
  The data structures used by these computations must be independent from each other and must fit the GPU architecture. Indeed, if the data are not stored by taking into account the constraints of the memory architecture on the GPU, this will impact the overall performance of the model (see [3] for more information on this aspect).

To give an example, based on our different case studies, we recommend in the case of discretized environments the use of arrays or data structures that fit the environment size. So, data will be more suited to the structure of the GPU because, in such case, each cell of the environment will be computed by a *thread*[4]. Figure 2 illustrates the use of arrays with GPU delegation and section 4.1 described in details the architecture of a GPU and the associated programming philosophy. With this data structure, agents will only drop off and perceive information (see the example of heatbugs model in section 4.2).

[4] *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 of the GPU (its identifier).

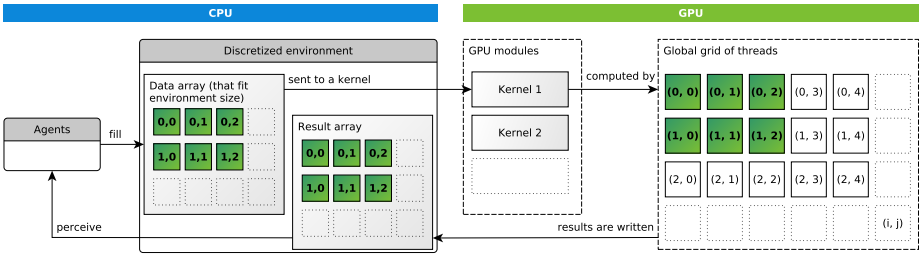

Fig. 2. Structuring data with GPU delegation and a discretized environment

Step 5: Implementation of the GPU Delegation Implementing GPU delegation can be divided into three parts for each selected computation:

1. Creating the GPU *kernels*;
2. Adapting the data structures;
3. Linking the agent API with the GPU modules.

Applying GPU delegation starts with the creation of the GPU *kernel*, that is the GPU programming version of the selected computation. Thanks to the decomposition which have been done in the identifying step, little GPGPU knowledge is required and the produced *kernels* are easy to implement through a few lines of code (*e.g.* [8]). Then, the data structures need to be adapted to the new GPU module. This adaptation is based on the nature of both the computations and the environment model (arrays fitting the discretization of the environment are mostly used, as recommended previously). Finally, these new elements must be integrated and linked with the CPU part of the model. So, new functions must be created to allow the agents and the environment to collect and use the data computed by the GPU module[5].

4 Experimenting the GPU Delegation Methodology

In this section, we experiment the proposed methodology on two MABS models: heatbugs and prey/predator. Specifically, the application of the method on these two models was conducted so as to make explicit the 5 steps of the process in order to define what are the advantages and limitations of such an approach. But first, we present some basics about GPU programming.

4.1 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

[5] The TurtleKit platform (`http://www.turtlekit.org`, [14]) has been used for the development of the GPU delegation principle and methods for the integration of GPGPGU were defined only once at the beginning and then reuse for all the next experiments.

associated programming model relies on the following philosophy[6]: 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 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*. So each *thread* works with the same *kernel* but uses different data according to its spatial location within the grid. Moreover, each *block* has a limited *thread* capacity according to the hardware in use. In the remainder of this document, the identifiers of the *threads* in the global grid of the GPU will be denoted by i and j. Figure 3 illustrates this organization for the 2D case. More informations about GPU programming are available in [15] and [22].

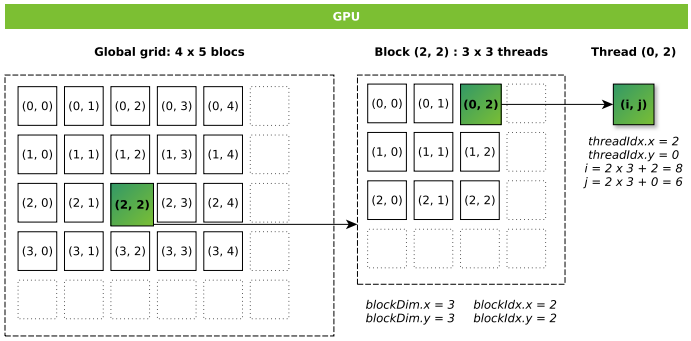

Fig. 3. Thread, blocks, grid organization

For both of these implementations (heatbugs and prey/predator), the integration of GPU computations was performed in the TurtleKit platform by using the JCUDA library which allows to use CUDA through Java[7].

4.2 The Heatbugs Model

Heatbugs is a model of biologically-inspired agents that attempt to maintain an optimum temperature around themselves. In this model, the bugs (the agents) move around on a 2D environment discretized in cells. A bug may not move to a cell that already has another bug on it. Each bug radiates a small amount

[6] *e.g.* `http://docs.nvidia.com/cuda/`
[7] *e.g.* `http://www.jcuda.org`

of heat which gradually diffuses through the world. Moreover, each bug has an "ideal" temperature it wants to be. The bigger the difference between the cell's temperature and the bug's ideal temperature is high, the more "unhappy" the bug is. When a bug is unhappy (the cell is too cold or too hot), it moves randomly to find a place that better suits those expectations.

Applying the Methodology The first step consists in enumerating and decomposing all the computations presents in the model (Figure 4 illustrates this decomposition):

- Environment: Diffusion of the heat emitted by agents (C1).
- Agent: Bugs move (C2), radiate (C3), compute the temperature difference between that of the cell and their ideal temperature (C4) and adjust their happiness (C5).

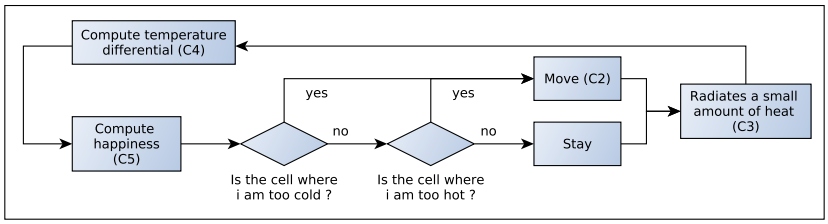

Fig. 4. Summary of behavioral processes of agents in the Heatbugs model

Secondly, we identify eligible computations. The heat diffusion (C1) is an environmental dynamic. So, it is eligible and can be transformed into a GPU *kernel*. C5 consists in perceiving a temperature information and computing the difference between the ideal temperature of the bug and the present temperature according to the value perceived. Because it does not modify the agents' states, it is thus eligible and can be transformed into an environmental dynamics. However, C2, C3 and C4 modify the agents' states, so we do not consider them for the next steps.

Thirdly, we check whether a GPU module exists for the identified computations. C4 has never been implemented in a GPU module in contrary to C1 which consists in computing a diffusion in the environment and was performed several times before [10]. Therefore, we reuse the corresponding GPU module for C1.

Fourthly, we evaluate if these computations can fit the GPU architecture. The heat diffusion is performed for all the cells and data structures used for this computation (a 2D array) are particularly well adapted to the GPU architecture so that GPU delegation could be applied. Considering C4, it can benefit from the GPU power because it consists in computing the difference between two values and can be easily distributed on the whole GPU grid. Moreover, this computation is performed by all the agents at each time step. Finally, we can use 2D arrays for storing the data from this computation.

Fifthly, we implement GPU delegation on the two selected computations. For C1, we use a 2D array (matching the size of the environment) containing the heat value for each cell. It is sent to the GPU that computes simultaneously the heat's diffusion for all the environment. More precisely, for each cell, a sum of heat values from neighboring cells is performed and modulated by a diffusion variable. Algorithm 1 presents the implementation of the corresponding GPU *kernel*[8].

After the execution of this *kernel*, the heat of each cell is used to compute the *delta* value (C4): The difference between the temperature of the cell where the agent is and the agent's ideal temperature. To this end, agents have previously filled their ideal temperature in a 2D array (fitting the environment size) according to their position. Then, once this computation is done, the agents recover the resulting value (the delta value) in the array and adjust their behavior accordingly. Algorithm 2 presents an implementation of this GPU *kernel*. So, instead of a computation performed in their behavior, the agents now drop information in the environment and then realize a perception which is precomputed by a GPU *kernel*.

Algorithm 1: Heat diffusion *Kernel*

input : $width$, $height$, $heatArray$, $radius$
output: $resultArray$ (the quantity of heat)

$i = blockIdx.x * blockDim.x + threadIdx.x$;
$j = blockIdx.y * blockDim.y + threadIdx.y$;
$sumOfHeat = 0$;
if $i < width$ *and* $j < height$ **then**
  $sumOfHeat =$ getNeighborsHeat$(heatArray[i, j], radius)$;
end
$resultArray[i, j] = sumOfHeat * heatAdjustment$;

Algorithm 2: Delta computation *kernel*

input : $width$, $height$, $heatArray$, $idealTemperatureArray$
output: $resultArray$ (the delta value)

$i = blockIdx.x * blockDim.x + threadIdx.x$;
$j = blockIdx.y * blockDim.y + threadIdx.y$;
$happiness = 0$;
if $i < width$ *and* $j < height$ **then**
  $happiness = heatArray[i, j] - idealTemperatureArray[i, j]$;
end
$resultArray[i, j] = happiness$;

[8] i and j are the coordinates of a *thread* which is considered as an instance of the *kernel*. Each *thread* is performed on a restricted portion of the data depending on its location (these coordinates) in the global GPU architecture grid.

To evaluate model's performance after the application of the methodology, we compare the CPU and hybrid versions[9]. The model is simulated for different environment sizes and a fixed density of agents (40%). Figure 5 presents the acceleration coefficients obtained between the two versions of the model. From this results, we notice that the acceleration coefficient obtained for the environment is more important when the environment is big (*e.g.* the gain reaches x7.5 for the biggest environment). However, the gain for the agents' behavior is low (about 5%). We can explain these results as follows: Environmental dynamics is applied to all the cells and performed by a GPU *kernel* while only a small part of computation made within the agent behavior (the computation of the delta value) has been delegated to a GPU module. Moreover, for the latter, the gain highly depends on the density of agents: If it is too low the gain may be negative.

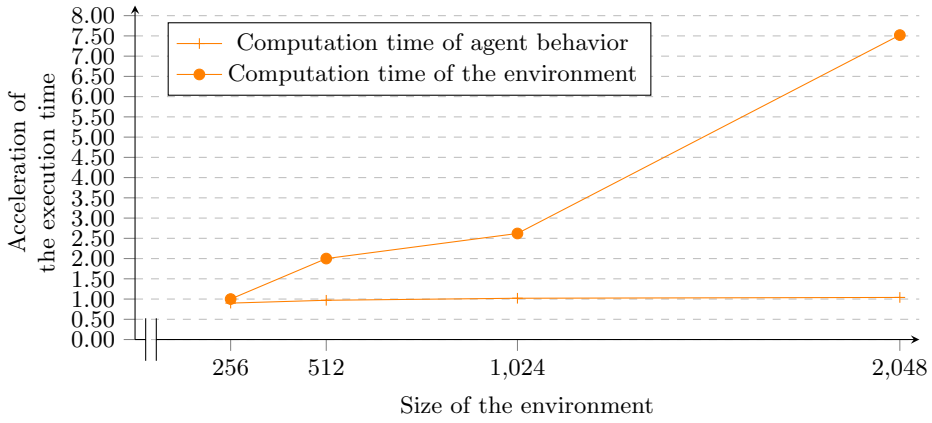

Fig. 5. Performance gains between CPU and Hybrid versions of the Heatbugs model

4.3 The Prey/Predator Model

The Prey/Predator model describes the dynamics of biological systems in which two species interact, one as a predator and the other as prey. In our model, the agents evolve in a 2D environment discretized in cells. Predators and prey are placed randomly in the environment. All predators have a *Field Of Vision* (FOV) that reaches 10 cells around them. Predators search for a prey in their FOV. If no prey can be targeted, they move randomly. In the other case, they head to the targeted prey. Prey have a smaller FOV. They randomly move in the environment and when a predator is in their field of vision, they run away

[9] 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).

in the opposite direction. A prey dies when it is targeted and when one predator is on the same cell.

Applying the Methodology The first step consists in enumerating and decomposing all the computations present in the model (Figure 6 illustrates this decomposition):

- The environment is static and does not have any endogenous dynamics.
- Agents: Predators (C1) compute the intercept heading toward the targeted prey and (C2) move, prey (C3) compute the escape heading that allows them to flee from the nearest predator and (C4) move.

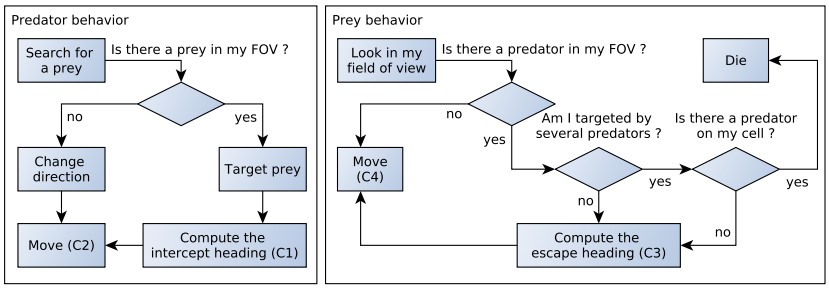

Fig. 6. Summary of behavioral processes of agents in the Prey/Predators model

Secondly, we identify eligible computations. Among these four computations, C2 and C4 modifying the agent's states (the agent's position) while C1 and C3 consist in computing displacement directions that do not modifying the agent's states. So, C1 and C3 can be transformed into environmental dynamics. These dynamics will compute for each cell of the environment the direction toward the closest agent (prey and predator). The agents will only perceive, according to their type, the direction that interest them and act accordingly.

Thirdly, we check whether a GPU module exists for the identified computations. For C1 and C3, we can reuse the *GPU field perception* module previously created in [13] which computes a pheromone field gradients. Indeed, this module computes for each cell of the environment the direction of neighboring cells with the greatest / smallest amount of a given data. Here, the data is the presence or absence of agents in the neighborhood.

Fourthly, we evaluate if these computations can fit the GPU architecture. Given that we reuse an existing module, and no new computation has been identified as compatible, we can directly go to step 5 because we know that C1 and C can fit the GPU architecture.

Fifthly, we implement GPU delegation on the two selected computations. For C1 and C3, we reuse one *kernel* already created in previous works. It will just be necessary to adapt the data that will be sent to this *kernel*. C1 and C3

Algorithm 3: The presence gradient *kernel*

```
   input  : width, height, preyMark[]
   output: preyMaxDirection[]
 1  i = blockIdx.x * blockDim.x + threadIdx.x ;
 2  j = blockIdx.y * blockDim.y + threadIdx.y ;
 3  float max = 0 ;
 4  int maxIndex = 0 ;
 5  if i < width and j < height then
 6  |  for int u = 1 ; u < 8; u++ do
 7  |  |  float current = getNeighborsValues(u, preyMark[i, j]);
 8  |  |  if max < current then
 9  |  |  |  max = current;
10  |  |  |  maxIndex = u;
11  |  |  end
12  |  end
13  |  preyMaxDirection[i, j] = maxIndex * 45 ;
14 end
```

being similar computations, we only take as an example the implementation of C3. So, each prey files a presence mark in a two-dimensional array (`preyMark`) according to its location. The presence mark is all the greater as there are prey in the neighborhood. Then, this array is sent to the GPU module which tests the vicinity of each cell of the environment and determines the direction leading to the strongest presence mark. The directions are written in a second array (`preyMaxDirection`). Predators only have to perceive in this array the heading value leading to the nearest prey. Algorithm 3 presents an implementation of this GPU *kernel*.

It is the same process for C1: Each predators files a presence mark in a two-dimensional array (`predatorsMark`) which is sent to the GPU module. Prey only perceive in the result array (`predatorsMaxDirection`) the heading value leading to the nearest predators and flee according to this value.

To evaluate model's performance after the application of the methodology, we compare the CPU and hybrid versions[10]. The model is simulated for different environment sizes and a fixed density of agents (40%). The distribution between prey and predators is the following: 90 % of prey and 10 % of predators. Figure 7 presents the computation time for one time step obtained for the two versions of the model.

From this results, we notice that the performance difference between the two versions of the model increase with the size of the environment. This observation has already been made in our previous work [10].

[10] For those tests, we reuse the same configuration as previously detailed.

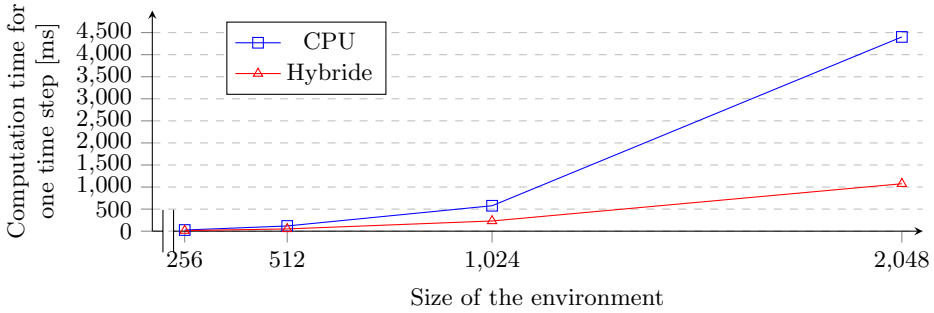

Fig. 7. Performance between CPU and Hybrid versions of the Prey/Predator model.

5 Conclusion and Future Work

This paper presented a methodology for modeling and implementing MABS using GPU programming, namely GPU delegation. It is based on [13, 9] and extracted from the experiment conducted in [8, 10]. The long term goal of the GPU delegation methodology is to provide a complete workflow for actually considering GPU programming in the context of MABS, that is (1) without hiding this technology to the user and (2) by promoting an iterative modeling process that put forward software engineering aspects such as modularity and reusability.

Compared to existing works which are related to the use of GPU programming in MABS, one main advantage of the proposed methodology is accessibility. Indeed, considering the two experiments presented in this paper, we have seen that applying the GPU delegation methodology workflow is easy and helps to identify which parts of a MABS model could be considered for GPU programming. Especially, we have seen that it was possible to find eligible computations on the two selected models. Moreover, we have seen that this workflow promotes modularity and thus reusability, which is an advantage of this approach compared to other existing works. For instance, considering the heatbugs model, we have been able to directly reuse a *kernel* (for the heat diffusion) which has been achieved in another context (for [8]).

Another advantage of GPU delegation relies on its versatility in the sense that it does not make any assumption on the kind of MABS which could be envisaged. Especially, considering all the adapted models and experiments which have led to the definition of this methodology (*e.g.* [13, 9, 8]), one can see that a wide variety of use cases have been implemented: Reynolds boids, game of life, Schelling's segregation, fire spreading, heatbugs, prey/predator, etc.

As a first limitation, this last point has to be moderated by the fact that most of our use cases embed discretized environments for which GPU delegation is relatively easy to achieve in terms of implementation. So, one future work will be to test GPU delegation of more heterogeneous models and use cases (*e.g.* with continuous environments), strengthening its scope of applicability.

Another limit is about its ability to be used for deciding if a particular model is worth porting on the GPU or not. Indeed, as we have seen in this paper, even if a model validates the second step (containing eligible computations), in some cases, the performance gains could be low. This is particularly true when the model does not contain environmental dynamics. In such a case, obtaining performance gains only depends on the number of agents which is simulated. If this number is small, the gain could be insignificant or even negative (*e.g.* as for the heatbugs model). In fact, we are here facing one limit of the proposed methodology in the sense that we can not predict in advance the benefits of the application of the methodology. In such a case, the application of the methodology is very dependent on the parameters of the model and on the hardware configuration. So, determining the threshold above which GPU delegation could be useful still requires an empirical evaluation.

For addressing this last issue, one research perspective is to develop a software solution (benchmark), that one could run on his particular hardware configuration to have an idea of the threshold above which a GPU implementation could be worth doing. More specifically, the idea is to develop a set of common agent computation patterns (GPU *kernels*) which would be used to test the relevance of applying GPU delegation considering both the hardware platform and the MABS model.

References

1. 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).
2. A. Bleiweiss. Multi agent navigation on the GPU. *Games Developpement Conference*, 2009.
3. M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4):583–600, 2014.
4. 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.
5. S. Coakley, P. Richmond, M. Gheorghe, S. Chin, D. Worth, M. Holcombe, and C. Greenough. *Intelligent Agents in Data-intensive Computing*, chapter Large-Scale Simulations with FLAME, pages 123–142. Springer International Publishing, Cham, 2016.
6. 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.
7. 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.

8. E. Hermellin and F. Michel. GPU Delegation: Toward a Generic Approach for Developping MABS using GPU Programming. In *(To be published) the proceedings of the international conference on Autonomous Agents and Multiagent Systems, AAMAS, Singapor*, pages –, 2016.
9. E. Hermellin and F. Michel. *Multi-Agent Based Simulation XVI: International Workshop, MABS 2015, Istanbul, Turkey, May 5, 2015, Revised Selected Papers*, volume 9568, chapter GPU Environmental Delegation of Agent Perceptions: Application to Reynolds's Boids, pages 71–86. Springer International Publishing, 2016.
10. E. Hermellin and F. Michel. Overview of Case Studies on Adapting MABS Models to GPU Programming. In J. Bajo, M. J. Escalona, S. Giroux, P. Hofa-Dabrowska, V. Julian, P. Novais, N. Sanchez-Pi, and R. A.-S. Rainer Unland, editors, *Highlights Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection.*, pages –. Springer International Publishing, 2016. To be published.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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*, 26(1):80–113, 2007.
16. 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.
17. R. Pavlov and J. Mller. Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors. In L. Camarinha-Matos, S. Tomic, and P. Graa, 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.
18. 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.

19. P. Richmond, S. Coakley, and D. M. Romano. A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, volume 2 of *AAMAS '09*, pages 1125–1126, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.
20. P. Richmond and D. M. Romano. A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. *European Simulation and Modelling*, 2011.
21. 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.
22. J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson, 2011.
23. 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.
24. 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.
25. D. Weyns and F. Michel. *Agent Environments for Multi-Agent Systems IV, 4th International Workshop, E4MAS 2014 - 10 Years Later, Paris, France, May 6, 2014, Revised Selected and Invited Papers*, volume 9068 of *LNCS*. Springer, 2015.