Délégation GPU des perceptions agents : intégration itérative et modulaire du GPGPU dans les simulations multi-agents

Application sur la plate-forme TurtleKit 3

Fabien Michel

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
161 rue Ada, 34392 Montpellier Cedex 5, France
fmichel@lirmm.fr

RÉSUMÉ. *La simulation multi-agent de systèmes complexes peut nécessiter de considérer un grand nombre d'entités, ce qui pose des problèmes de performance et de passage à l'échelle. Dans ce cadre, la programmation sur carte graphique (General-Purpose Computing on Graphics Processing Units GPGPU) est une solution attrayante : elle permet des gains de performances très conséquents sur des ordinateurs personnels. Le GPGPU nécessite cependant une programmation extrêmement spécifique et cette spécificité limite à la fois son accessibilité et la possibilité de réutiliser les développements qui sont réalisés par différents acteurs. Nous présentons ici l'approche que nous avons utilisée pour intégrer du calcul sur GPU dans la plate-forme TurtleKit. L'objectif de cette approche est de conserver l'accessibilité de la plate-forme, en termes de simplicité de programmation, tout en tirant parti des avantages offerts par le GPGPU. Nous montrons ensuite que cette approche peut être généralisée sous la forme d'un principe de conception spécifiquement dédié à la simulation de SMA dans un contexte GPGPU.*

ABSTRACT. *Simulating complex systems may require to handle a huge number of entities, raising scalability issues. In this respect, GPGPU is a relevant approach. However, GPU programming is a very specific approach that limits both accessibility and re-usability of developed frameworks. We here present our approach for integrating GPU in TurtleKit, a multi-agent based simulation platform. Especially, we show how we keep the programming accessibility 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.*

MOTS-CLÉS : *calcul haute performance, GPGPU, simulation multi-agent.*

KEYWORDS: *high performance computing, GPGPU, multi-agent based simulation.*

DOI:10.3166/RIA.28.485-511

1. Introduction

Parce qu'ils sont parfois composés d'un très grand nombre d'entités en interaction, étudier les propriétés des systèmes complexes à l'aide de simulations multi-agents (Michel *et al*., 2009) peut nécessiter beaucoup de puissance de calcul. La masse d'agents n'est d'ailleurs pas toujours l'unique goulot d'étranglement. La taille de l'environnement et surtout la complexité de ses dynamiques endogènes peuvent aussi constituer des postes de consommation très importants. De fait, les performances d'exécution représentent souvent un obstacle qui limite fortement le cadre dans lequel un modèle peut être étudié.

Parallèlement, dans le cadre du calcul haute performance, la programmation GPU, c'est-à-dire l'utilisation de cartes graphiques pour la réalisation de calculs généralistes (*General-Purpose Computing on Graphics Processing Units GPGPU*), possède une place à part car elle permet d'obtenir des gains de performances considérables pour un coût financier très faible (Che *et al*., 2008 ; Bourgoin, 2013). La majorité des cartes graphiques 3D actuelles sont en effet équipées de capacités GPGPU.

Cependant la programmation sur GPU n'est pas chose aisée car elle repose sur une architecture matérielle spécifique qui définit un contexte de développement très particulier. Ce contexte architectural nécessite notamment de suivre les principes de la programmation par traitement de flot de données (*stream processing paradigm* [1]) (Owens *et al*., 2007). En particulier, il n'est pas possible d'adopter une démarche orientée objet classique. De fait, les modèles de simulation multi-agent couramment utilisés, parce qu'ils reposent sur des implémentations orientées objet, ne peuvent être utilisés sur une architecture GPU sans un effort de traduction conséquent et non trivial. Un modèle multi-agent classique doit en effet être repensé intégralement pour pouvoir être exécuté sur GPU, ce qui nécessite des connaissances particulièrement pointues (Lysenko, D'Souza, 2008).

Ainsi, malgré l'existence de travaux démontrant les possibilités de gains offertes par la programmation sur GPU (jusqu'à des centaines de fois plus rapide qu'avec des plates-formes classiques (Lysenko, D'Souza, 2008)), les spécificités de cette dernière posent de nombreux problèmes qui limitent fortement l'accessibilité et la réutilisabilité des algorithmes et des implémentations développés dans différents contextes. Il est donc compréhensible que peu de travaux soient enclins à investir du temps dans l'utilisation de cette technologie car la pérennité du code produit est difficile à obtenir. La programmation sur GPU n'est ainsi pas encore très répandue dans la communauté et pour l'instant les travaux mêlant GPU et simulation multi-agent sont majoritairement liés à des expérimentations très spécifiques effectuées dans des contextes ponctuels, ce qui ne permet pas leur réutilisation. Ainsi aucune plate-forme générique, notamment NetLogo (Sklar, 2007) et RePast (North *et al*., 2007), n'intègre aujourd'hui de GPGPU.

1. Étant donné un flot de données, une série d'opérations (des fonctions *kernel*) est appliquée à chaque élément du flot.

Dans cet article nous revenons en détail sur la manière dont nous avons intégré l'utilisation du calcul sur GPU dans TurtleKit, une plate-forme de simulation multi-agent générique (Michel *et al.*, 2005). Dans ce travail, nous avons deux objectifs. Premièrement, il s'agit d'obtenir des gains de performances grâce au GPU, ceci afin d'être capable de réaliser des simulations larges échelle sur TurtleKit, c'est-à-dire avec un grand nombre d'agents et des environnements de grande taille. Deuxièmement, pour éviter les écueils que nous avons cités, il s'agit de conserver l'accessibilité et la facilité de réutilisation de la plate-forme, et plus particulièrement son interface de programmation orientée objet.

La section 2 présente des travaux qui utilisent le GPGPU pour développer des simulations multi-agents et identifient leurs limites par rapport à nos objectifs. La section 3 présente TurtleKit et le modèle multi-agent que nous avons utilisé pour tester l'intégration de modules GPU dans la version 3 de cette plate-forme. La section 4 décrit comment nous avons conçu un premier module GPU en traduisant deux dynamiques environnementales : la diffusion et l'évaporation de phéromones digitales. La section 5 détaille l'implémentation de ce module dans TurtleKit 3 et les choix technologiques associés. La section 6 analyse les différents résultats que nous avons obtenus avec ce premier module. La section 7 présente tout d'abord un second module GPU, réalisé en transformant certains calculs réalisés par les agents en dynamiques environnementales calculées par le GPU, avant de discuter les performances obtenues. La section 8 présente une généralisation de notre approche et propose le principe de *délégation GPU des perceptions agents*. La section 9 conclut l'article et discute des perspectives liées à notre approche dans le contexte du GPGPU.

2. GPGPU et simulation multi-agent

Grâce aux centaines de cœurs aujourd'hui disponibles sur les cartes graphiques, la programmation GPU permet de réaliser un très grand nombre de calculs similaires en parallèle [2]. Cette caractéristique est particulièrement intéressante lorsqu'on considère les systèmes complexes modélisés à l'aide du paradigme multi-agent. Dans de tels systèmes, des calculs similaires sont souvent réalisés des millions, voire des milliards de fois au cours d'une même simulation. Les modèles multi-agents ont donc un fort potentiel en termes de gains de performances. Il existe ainsi de nombreux travaux qui décrivent des simulations multi-agents utilisant le GPU avec succès dans différents domaines d'application tels que la simulation de foule large échelle (e.g. (Demeulemeester *et al.*, 2011)), la biologie (e.g. (Laville *et al.*, 2012 ; D'Souza *et al.*, 2009)) ou encore la simulation de mouvements collectifs complexes comme le flocking (e.g. (Silva *et al.*, 2010)).

Dans (Richmond *et al.*, 2010), des gains de performances impressionnants sont obtenus sur le logiciel FLAME (cellular level agent-based simulation framework)

2. Le lecteur trouvera une description récente et très complète du domaine du GPGPU dans la thèse de Bourgoin (Bourgoin, 2013).

grâce au GPGPU. En particulier, ceux-ci s'avèrent même plus importants qu'avec l'utilisation d'un cluster de CPU. Comme le soulignent les auteurs, de tels gains de performances sont extrêmement appréciables car ils facilitent le développement rapide de modèles complexes. Notamment, cela permet une visualisation en temps réel des dynamiques d'un modèle, ce qui facilite l'interaction que l'utilisateur peut avoir avec ce dernier, permettant ainsi une compréhension accrue de son fonctionnement.

Pour obtenir ces résultats, le modèle multi-agent FLAME a été entièrement traduit en code GPU, soulevant ainsi le problème de l'accessibilité du logiciel. Pour atténuer ce problème et ne pas obliger l'utilisateur à avoir des connaissances en programmation GPU, les auteurs proposent un formalisme basé sur XML qui permet de spécifier le comportement des agents. Bien que cela soit une bonne solution au regard des utilisateurs finaux, modifier ou étendre le modèle multi-agent lui-même requiert tout de même des connaissances en programmation GPU et la réutilisation du logiciel dans un autre contexte reste problématique.

En ce qui concerne la généricité, (Lysenko, D'Souza, 2008) propose de considérer la reprogrammation GPU de toute une classe de modèles multi-agents en s'attaquant aux simulations utilisant une grille en deux dimensions pour environnement. Dans ce type de modèles, parfois qualifiés de spatialisés, un environnement 2D est discrétisé en cellules sur lesquelles se déplacent des agents. La très utilisée plate-forme NetLogo (Sklar, 2007) possède un tel modèle. (Lysenko, D'Souza, 2008) explique clairement que la difficulté majeure de ce travail a été de reformuler ce modèle multi-agent générique en fonction des contraintes imposées par la programmation GPU, et que par conséquent tout doit être repensé. En résumé, les auteurs proposent une solution qui consiste à faire correspondre l'état des agents avec une texture gérée par la carte graphique, de telle sorte que l'ensemble des dynamiques est calculé grâce au GPU. Les résultats obtenus sur des modèles standards comme *SugarScape* sont impressionnants et démontrent l'intérêt du GPU pour la simulation multi-agent, à la fois en termes de vitesse d'exécution et de scalabilité des modèles.

Néanmoins, comme le remarquent eux-mêmes les auteurs de ce travail, appliquer une telle approche, que nous qualifions de *tout-sur-GPU*, ne se fait pas sans perdre les avantages de la programmation orientée objet. Par conséquent, créer un nouveau modèle requiert de manipuler du code GPU, et donc d'avoir des connaissances dans ce domaine. Ce qui pose à nouveau le problème de l'accessibilité de l'interface de programmation sous-jacente. Un tel manque d'accessibilité est sans nul doute le principal frein au développement du GPU dans les plates-formes multi-agents génériques.

Enfin, il nous faut à nouveau souligner que, même dans le cadre de travaux où l'optimisation prime sur la généricité, il est parfois difficile, voire impossible, de convertir un modèle multi-agent complet en code GPU du fait des contraintes mentionnées en introduction. Dans un tel cas, il peut être néanmoins très intéressant de réaliser une simulation hybride où seule une partie du modèle est convertie en code GPU. Cette solution est par exemple utilisée dans (Laville *et al.*, 2012) pour optimiser l'exécution des agents réactifs grâce au GPU tandis que les agents plus cognitifs restent exécutés par le CPU.

De plus, (Laville *et al*., 2012) illustre aussi très bien les difficultés qu'il y a à écrire du code GPU. En effet, même pour la partie du modèle qui se prête le mieux à une conversion en code GPU, les auteurs proposent différentes versions de son implémentation sur GPU et montrent qu'elles sont loin d'être toutes équivalentes. En particulier, la première adaptation proposée dans cet article s'avère même être moins efficace que l'algorithme original codé pour CPU. Ainsi, le simple fait de convertir un modèle en code GPU n'est pas en soi la garantie d'une efficacité accrue. Les auteurs montrent par la suite tout l'intérêt et la nécessité de bien penser les structures de données utilisées et proposent deux autres adaptations beaucoup plus efficaces.

D'une manière générale, la littérature montre clairement que les difficultés techniques liées à la programmation GPU ont deux conséquences majeures : elles limitent fortement (1) le champ d'application des logiciels développés de par une réutilisabilité faible et (2) l'accessibilité des programmes réalisés du fait des connaissances requises pour pouvoir faire évoluer le modèle multi-agent, celui-ci étant fortement influencé par son implémentation sur GPU. Notamment, les approches *tout-sur-GPU* sont restreintes à un domaine d'application spécifique et/ou nécessitent des connaissances avancées en programmation GPU.

3. Première intégration du calcul sur GPU dans TurtleKit 3

3.1. TurtleKit et le modèle MLE

À l'instar de NetLogo, TurtleKit [3] est une plate-forme qui utilise un modèle multi-agent spatialisé où l'environnement est discrétisé sous la forme d'une grille de cellules (Michel *et al*., 2005). Implémentée en Java à l'aide de la librairie de développement multi-agent MaDKit [4] (Gutknecht *et al*., 2001), TurtleKit repose sur des modèles d'agents et d'environnements inspirés par le langage de programmation Logo. En particulier, les agents peuvent émettre et percevoir des phéromones digitales possédant des dynamiques de diffusion et d'évaporation. Ces dynamiques permettent de créer des champs de gradients qui sont utilisés par les agents pour modéliser divers comportements, comme le suivi de traces par une fourmi. Calculer de telles dynamiques demande énormément de ressources de calcul, ce qui limite à la fois les performances et la scalabilité des modèles qui les utilisent, même lorsque peu de phéromones sont mises en jeu.

L'un des objectifs principaux de TurtleKit est de fournir aux utilisateurs finaux une interface de programmation (API) facilement accessible et extensible. En particulier, l'API de TurtleKit est orientée objet et son utilisation repose sur l'héritage de classes prédéfinies. Nous ne pouvons donc pas adopter une stratégie de conception *Tout-sur-GPU* qui va à l'encontre de nos objectifs. C'est pourquoi nous avons opté pour une approche intermédiaire, que l'on peut qualifiée d'hybride, qui consiste à intégrer de

3. http://www.turtlekit.org
4. http://www.madkit.org

manière itérative des modules utilisant de la programmation GPU tout en conservant inchangée l'API de TurtleKit.

Pour atteindre cet objectif, nous avons choisi de réaliser un premier prototype que nous avons testé en réalisant une nouvelle implémentation du modèle proposé dans (Beurier *et al*., 2002). Dans cette référence, un modèle multi-agent est proposé pour étudier un phénomène d'émergence multi-niveaux (MLE). Ce modèle très simple repose sur un unique comportement qui permet de générer des structures complexes qui se répètent de manière fractale. Plus précisément, à partir d'un unique ensemble d'agents non structuré de niveau 0, les agents évoluent pour former des structures de niveau 1 (des cercles) qui servent ensuite à former des structures de niveau 2 et ainsi de suite. Autrement dit, les agents de niveau 0 forment des cercles autour des agents de niveau 1 qui forment eux-mêmes des cercles autour des agents de niveau 2, etc.

Le comportement agent utilisé est extrêmement simple et repose uniquement sur la perception, l'émission et la réaction à trois types de phéromones différents : (1) *présence*, (2) *répulsion* et (3) *attraction*.

La phéromone de présence est utilisée par un agent pour évaluer combien d'agents d'un certain niveau se trouvent à proximité. Cette phéromone sert ainsi à faire muter un agent vers un niveau supérieur ou inférieur. Dit de manière simplifiée, une mutation peut se produire lorsqu'une zone est surpeuplée ou au contraire vide.

Les phéromones de répulsion et d'attraction sont utilisées par les agents pour créer une zone d'attraction circulaire autour d'eux, grâce à des taux d'évaporation et de diffusion différents. La phéromone d'attraction est émise en faible quantité mais s'évapore doucement, au contraire de la phéromone de répulsion, ce qui permet de créer un zone d'attraction circulaire autour d'un agent.

Le comportement d'un agent est décomposé en quatre étapes : *Perception*, *Émission*, *Mutation* et *Mouvement*. Le comportement des agents est identique pour tous les niveaux. L'état d'un agent est ainsi entièrement défini par un seul entier qui spécifie son niveau. Pour un niveau déterminé, un agent considère uniquement les phéromones de niveaux adjacents.

La figure 1 montre un exemple d'évolution de l'exécution du modèle MLE. Tous les agents sont d'abord de niveau 0 (rouge). Apparaissent ensuite des agents de niveau 1 (écran 2) puis de niveau 2 (3), jusqu'au niveau 3 dans l'étape 4 de cet exemple.

3.2. Scalabilité du modèle MLE

Sur le papier, le plus haut niveau d'émergence qui peut être atteint grâce au modèle MLE est uniquement lié à deux paramètres : (1) la taille de l'environnement, car un niveau ne peut apparaître que si suffisamment de place est disponible et (2) le nombre initial d'agents : il faut un minimum d'agents de niveau $i-1$ pour voir apparaître des structures de niveau i.

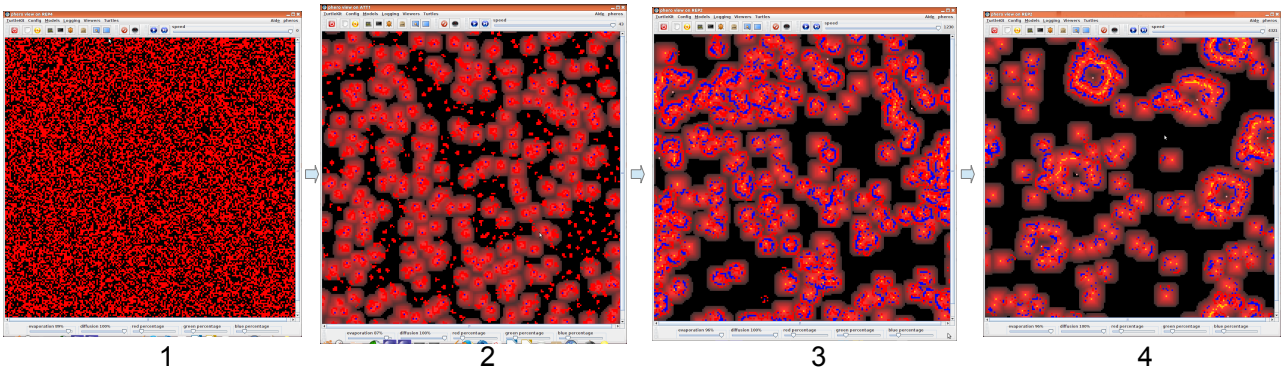

Figure 1. Exemple d'évolution du modèle MLE

Par rapport aux objectifs que nous poursuivons, créer une nouvelle implémentation du modèle MLE constitue un test parfait car, pour voir apparaître des niveaux d'émergence plus élevés, il est rapidement nécessaire d'accroître à la fois la taille de l'environnement et le nombre des agents. De plus, chaque niveau supplémentaire nécessite de gérer trois nouvelles phéromones, ce qui augmente significativement le besoin en ressources de calcul du modèle et rend particulièrement difficile le passage à l'échelle.

En particulier, pour chaque phéromone, réaliser les calculs qui correspondent aux dynamiques de diffusion et d'évaporation nécessite d'effectuer des opérations mathématiques pour toutes les cellules de la grille à chaque pas de temps. Ainsi, bien que ces calculs soient simples, la complexité qui leur est associée évolue de manière quadratique avec la taille de l'environnement.

En outre, il s'avère que la dynamique globale engendrée est de plus en plus coûteuse au fur et à mesure que la simulation avance dans le temps. En fait, les calculs correspondants aux phéromones sont tout d'abord simples car les différentes grilles n'en contiennent qu'une quantité relativement faible, voire nulle pour les niveaux supérieurs. Ce qui a aussi notamment pour effet que les agents réagissent peu et appliquent un comportement d'exploration très peu coûteux. Par la suite, l'accumulation de phéromones de différents niveaux aboutit à une augmentation très significative des temps de calculs associés. Ainsi, les performances de la simulation s'écroulent littéralement après quelques centaines d'itérations, rendant son exploitation en temps réel quasi impossible.

Dans un premier temps, notre attention s'est donc focalisée sur le calcul des dynamiques liées aux phéromones. Par rapport à nos objectifs, ce choix était d'autant plus naturel que ces dynamiques sont complètement découplées du modèle comportemental des entités simulées. Il est donc possible de créer un module GPU, dédié au calcul de ces dynamiques, sans avoir à modifier quoi que ce soit de l'API du modèle agent. De plus, comme nous allons maintenant le voir, elles sont relativement faciles à

traduire car les calculs correspondants sont naturellement spatialement distribués, ce qui convient très bien à une adaptation en code GPU.

4. Le module de diffusion GPU

4.1. Traduction GPU de la dynamique d'évaporation

Pour expliquer simplement la traduction des dynamiques d'évaporation et de diffusion en code GPU, nous présentons tout d'abord le mécanisme d'évaporation car il est extrêmement simple. L'évaporation d'une phéromone sur la grille consiste simplement à multiplier la quantité présente dans une cellule par un coefficient compris entre 0 et 1 (le taux d'évaporation, *evapCoef*). L'implémentation séquentielle de cette dynamique peut être définie par l'algorithme 1 :

Algorithme 1 : évaporation en séquentiel

Entrées : Une *grille* de cellules représentant une phéromone, sa *hauteur*, sa *largeur* et le coefficient d'évaporation à lui appliquer, *evapCoef*.

```
début
    pour i=1 à largeur faire
        pour j=1 à hauteur faire
            grille[i][j] ← grille[i][j] × evapCoef;
        fin
    fin
fin
```

Voyons maintenant quelques principes liés à la programmation GPU. Une carte graphique équipée de capacités GPGPU est capable de réaliser l'exécution d'une même procédure, appelée *kernel*, sur de très nombreux *threads* s'exécutant de manière concurrente. Ces threads sont organisées en blocs (*block*), eux-mêmes organisés dans une grille englobant tous les blocs. Chaque thread possède par ailleurs des coordonnées en trois dimensions, x, y et z qui le localisent dans son bloc, chaque bloc étant lui-même localisé de la même manière dans la grille [5].

Ainsi, il est possible de considérer uniquement les coordonnées en deux dimensions des blocs et des threads pour définir une grille de threads correspondant à une grille de données. Par exemple, si la capacité d'un bloc est de 1024 threads (celle-ci dépend du matériel), il est possible de travailler avec une grille de 1000×1000 en allouant une grille de blocs d'une taille de 32×32, avec chaque bloc ayant lui-même une taille de 32×32. Ce qui produit une matrice surdimensionnée de 1024×1024 threads.

5. Le terme thread s'apparente ici à la notion de tâche : un thread peut être considéré comme une instance du kernel qui s'effectue sur une partie restreinte des données en fonction de son identifiant, c'est-à-dire suivant sa localisation dans la grille globale.

Cette taille trop large par rapport aux données considérées est habituelle et ne pose pas de problème car elle est gérée au niveau du code GPU. Ainsi, la taille d'une grille de blocs et la dimension des blocs sont des paramètres fondamentaux qui sont définis pour chaque exécution d'un kernel sur le GPU.

Dans notre cas, cela va nous permettre de faire correspondre chaque cellule de la grille de l'environnement des agents avec un unique thread. Le kernel correspondant au processus d'évaporation peut ainsi être défini en GPU de la manière suivante :

Algorithme 2 : évaporation en GPU

Entrées : Une *grille* de cellules représentant une phéromone, sa *hauteur*, sa *largeur* et le coefficient d'évaporation à lui appliquer, *evapCoef*.

début
  i ← blockIdx.x × blockDim.x + threadIdx.x;
  j ← blockIdx.y × blockDim.y + threadIdx.y;
  si *i* < *largeur et j* < *hauteur* **alors**
    grille[i][j] ← grille[i][j] × *evapCoef*;
  fin
fin

Lorsque l'exécution de ce kernel est appelée sur le GPU, tous les threads alloués exécutent l'algorithme 2 simultanément.

Les deux premières lignes de cette procédure déterminent les coordonnées du thread qui est en train de s'exécuter. Notamment, les variables *blockIdx*, *threadIdx* et *blockDim* sont prédéfinies pour tout kernel. *blockIdx* contient l'abscisse, *blockIdx.x*, et l'ordonnée, *blockIdx.y*, du bloc courant dans la grille globale, *blockIdx.z* étant fixé à 1 dans le cas d'une grille 2D. De façon similaire, *threadIdx* correspond aux coordonnées du thread courant à l'intérieur de son bloc. Enfin, *blockDim* permet d'accéder à la taille d'un bloc, en termes de thread, pour chaque dimension considérée : *blockDim.x* pour la largeur et *blockDim.y* pour la hauteur. La figure 2 illustre ces différents points en prenant pour exemple une grille 2D carrée de taille 4 contenant des blocs 2D carrés de dimension 3.

Un test est ensuite effectué pour savoir si ce thread n'est pas en dehors des limites de la grille de données. Si c'est le cas, la valeur de la cellule correspondante est mise à jour. Il est important de noter que le fait d'opérer sur une matrice légèrement surdimensionée n'engendre aucune perte de performance car le temps d'un cycle GPU est forcément calé sur les threads qui nécessitent le plus long temps de calcul.

Par rapport à la version séquentielle de l'algorithme d'évaporation, on voit que la double boucle a disparu. Ainsi tout l'intérêt de la version GPU tient dans le fait que la parallélisation de cette double boucle est intrinsèquement réalisée grâce à l'architecture matérielle et non à l'aide de code additionnel : le code réside dans la structure.

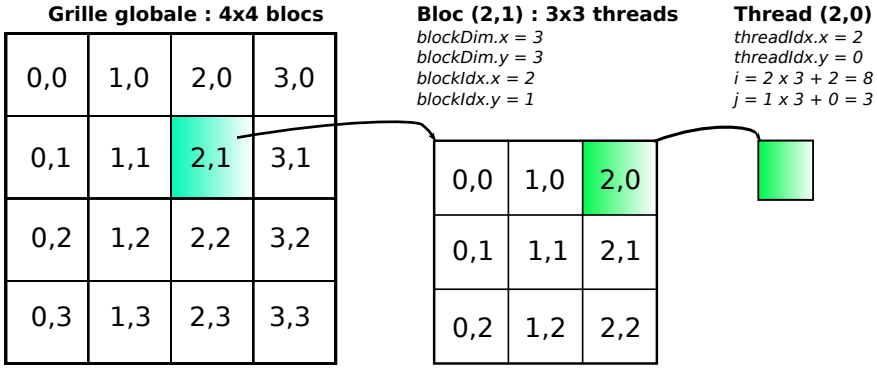

Figure 2. Calcul des coordonées d'un thread dans une grille globale en 2D

4.2. Algorithme GPU de la dynamique de diffusion

La diffusion d'une phéromone dans l'environnement consiste à faire en sorte que toutes les cellules de la grille transmettent une partie de leur contenu vers leurs voisins en fonction d'un coefficient de diffusion, *diffCoef*, compris entre 0 et 1. Ainsi, la traduction de cette dynamique en code GPU repose sur une adaptation similaire à ce que nous avons vu pour l'évaporation.

Cette adaptation est cependant légèrement plus complexe car le calcul correspondant doit nécessairement se faire en deux temps. Il faut tout d'abord (1) calculer la quantité de phéromone cédée par chaque cellule à ses voisins et stocker ce résultat dans une grille de données tampon puis (2) mettre à jour toutes les valeurs de la grille à partir du résultat précédent.

Dans les faits, cela nécessite de définir deux kernels différents qui seront appelés successivement pour assurer la cohérence des données. En effet, l'étape 1 doit être entièrement terminée avant que l'étape 2 ne commence et, bien qu'il existe des primitives permettant de synchroniser les threads d'un même bloc, il est par contre impossible d'effectuer une synchronisation sur la globalité du calcul depuis l'intérieur d'un thread. On obtient ainsi les algorithmes 3 et 4.

À l'aide de ces différents algorithmes, nous avons donc développé un module GPU pour l'évaporation et la diffusion que nous appellerons par la suite le *module de diffusion GPU*. La section suivante détaille la manière dont nous avons réalisé l'implémentation de ce module au sein de la plate-forme TurtleKit 3.

Algorithme 3 : diffusion vers la grille tampon

Entrées : La *grille* de phéromone, sa *hauteur*, sa *largeur*, son coefficient de diffusion, *diffCoef* et une grille de données tampon, *grilleTampon*.

début

$i \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$;
$j \leftarrow blockIdx.y \times blockDim.y + threadIdx.y$;
si $i < largeur$ *et* $j < hauteur$ **alors**
$grilleTampon[i][j] \leftarrow grille[i][j] \times$ *diffCoef*;
fin

fin

Algorithme 4 : mise à jour de la diffusion

Entrées : La *grille* de phéromone, sa *hauteur*, sa *largeur* et *grilleTampon*.

début

$i \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$;
$j \leftarrow blockIdx.y \times blockDim.y + threadIdx.y$;
si $i < largeur$ *et* $j < hauteur$ **alors**
pour $voisin \in$ *voisins(*$grilleTampon[i][j]$*)* **faire**
$grille[i][j] \leftarrow grille[i][j] + voisin$;
fin
fin

fin

5. Implémentation du module de diffusion GPU dans TurtleKit 3

5.1. Choix technologique

Implémenter un programme orienté GPGPU nécessite de définir à la fois des kernels, qui s'exécuteront sur le GPU, mais aussi des procédures, destinées à être exécutées par le CPU pour organiser l'exécution des kernels et récupérer les données ainsi produites. Il existe actuellement deux solutions logicielles majeures permettant de mettre en œuvre de tels programmes : Cuda [6] et OpenCL [7]. Ces technologies reposent toutes les deux sur des API basées sur des extensions du langage C. Cuda (*Compute Unified Device Architecture*), développé par la société Nvidia spécifiquement pour les cartes graphiques de la même marque, fut le premier environnement de développement dédié au GPGPU (première version sortie début 2007). OpenCL

6. http://www.nvidia.com/object/cuda_home_new.html
7. http://www.khronos.org/opencl

(*Open Computing Language*), développé par le Khronos Group [8] et officiellement distribué pour la première fois fin 2008, repose sur une technologie qui n'est pas restreinte à une marque particulière et se veut être un standard ouvert pour l'utilisation des systèmes parallèles hétérogènes (CPU et/ou GPU). En cela, l'objectif d'OpenCL ne repose pas uniquement sur l'optimisation mais concerne principalement la portabilité des programmes qui sont écrits selon ce standard. En particulier, OpenCL permet le *fallback* des kernels qui sont écrits grâce à ce standard, c'est-à-dire que ces derniers peuvent être exécutés en utilisant les différents cœurs du CPU si aucune carte graphique possédant des capacités GPGPU n'est présente, fonctionnalité qui n'existe pas avec Cuda.

Étant donnés nos objectifs en termes d'accessibilité, il était par ailleurs fondamental de conserver Java comme langage principal pour TurtleKit. C'est pourquoi nous avons tout d'abord cherché une solution logicielle permettant de faire du GPGPU depuis Java. Parmi les librairies existantes, on peut notamment citer *Aparapi*, *JCuda* et *JOCL*.

JCuda [9] (*Java bindings for Cuda*) et JOCL [10] (*Java bindings for OpenCL*) sont des portages respectifs de Cuda et d'OpenCL en Java qui suivent strictement l'API originale correspondante. Ces bibliothèques permettent ainsi de commander l'exécution de kernels écrits en langage Cuda ou OpenCL (donc dans une syntaxe dérivée du C) directement depuis des fichiers sources Java.

Projet démarré par la compagnie AMD, Aparapi est depuis fin 2011 un projet open source qui propose une API Java permettant de programmer, directement en Java, des fragments de code qui sont automatiquement analysés syntaxiquement pour être traduits en code OpenCL au moment de l'exécution. Tous les détails du lancement d'un kernel sont ainsi cachés à l'utilisateur : les paramètres de lancement d'un kernel sont automatiquement calculés en fonction de la taille de la donnée.

Programmer avec Aparapi a l'avantage d'être très simple car il n'est pas nécessaire d'utiliser un autre langage que Java, ni même de connaître les détails de l'architecture d'un GPU. Cependant, en abstrayant l'utilisateur de la majorité des détails liés à la mise en place d'un programme GPGPU, des aspects importants de l'optimisation sur GPU restent cachés et cette API ne permet pour l'instant pas de régler finement la manière dont le GPU sera finalement utilisé, ce qui peut limiter fortement son efficacité. Au contraire, modulo le changement de syntaxe lié au langage Java, utiliser JCuda ou JOCL revient à utiliser respectivement Cuda ou OpenCL.

Dans ce cadre, l'antériorité et la qualité reconnue de Cuda lui permettent de posséder la plus grande des deux communautés d'utilisateurs. De plus, bien que la portabilité offerte par la technologie OpenCL soit un un atout majeur de cette dernière,

8. Consortium industriel travaillant sur la mise en place d'APIs standardisées publiques et gratuites comme par exemple OpenGL.

9. http://www.jcuda.org

10. http://www.jocl.org

elle a cependant pour conséquence de limiter les optimisations qui sont possibles lorsqu'on rentre dans le détail de la programmation d'un kernel. Notamment en ce qui concerne la gestion des différentes parties de la mémoire alors que Cuda propose depuis toujours des primitives qui permettent de gérer finement les différents types de mémoire qui peuvent être utilisés (privée au niveau thread, locale à un bloc, globale, etc.). Ainsi, même si les différentes spécifications évoluent très vite [11] et qu'OpenCL représente sans aucun doute un standard d'avenir vers lequel nous nous tournerons, nous avons initialement opté pour la technologie Cuda, et par conséquent pour JCuda, car elle n'abstrait aucun détail de l'architecture matérielle et nous a permis d'avoir une idée précise des différentes possibilités d'optimisation offertes par le GPGPU.

5.2. Intégration du module de diffusion GPU dans TurtleKit 3

La figure 3 illustre l'intégration du module de diffusion GPU dans le prototype de TurtleKit 3 à l'aide de JCuda. Celle-ci n'a pas posé de problème, notamment grâce à la modularité offerte par l'indépendance de ce module vis-à-vis du modèle agent. En particulier, on peut voir qu'un agent (une turtle) manipule indifféremment, suivant le contexte matériel disponible, des phéromones utilisant une implémentation classique de la diffusion ou le module GPU correspondant. Le modèle agent n'a donc pas besoin d'être modifié.

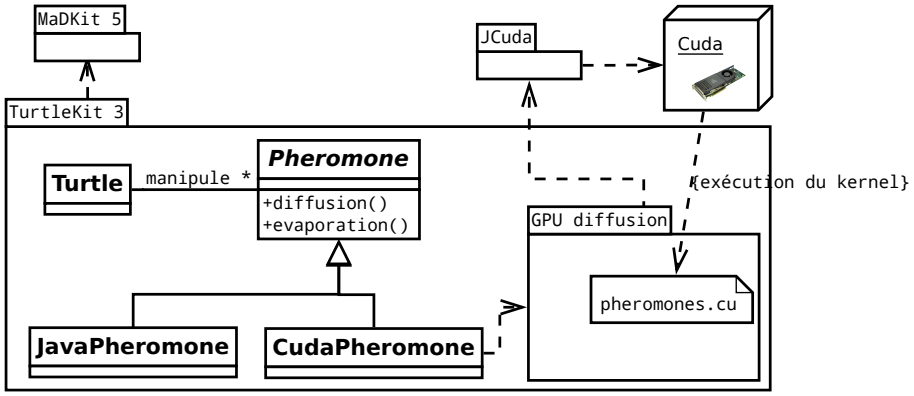

Figure 3. Intégration de la diffusion GPU dans TurtleKit 3

Si le matériel le permet, les phéromones créées lors de l'exécution sont donc des instances de la classe *CudaPheromone*. Pour calculer la diffusion et l'évaporation, celles-ci interagissent avec le module de diffusion GPU qui accède aux fonctions Cuda grâce à JCuda. Les kernels eux-mêmes sont codés en langage Cuda dans le fichier *pheromones.cu*.

11. Les spécifications de la nouvelle version majeure d'OpenCL, la 2.0, viennent d'être finalisées fin 2013.

5.3. Résultats obtenus par le module de diffusion GPU

La figure 4 compare les résultats obtenus avec une implémentation séquentielle du processus de diffusion, puis avec le module de diffusion GPU, pour différentes tailles d'environnement et une unique phéromone. Ces tests ont été réalisés sur des simulations ne contenant pas d'agents afin d'éviter les bruits provenant d'autres traitements. Ces résultats ont été obtenus, avec Cuda 4 et JCuda-0.4.1 sous Linux, à l'aide d'un CPU Xeon @ 3.2GHz (6 cœurs) et d'une carte graphique Nvidia Quadro 4000 dotée de 256 cœurs, ce qui constitue une valeur moyenne à l'heure actuelle.

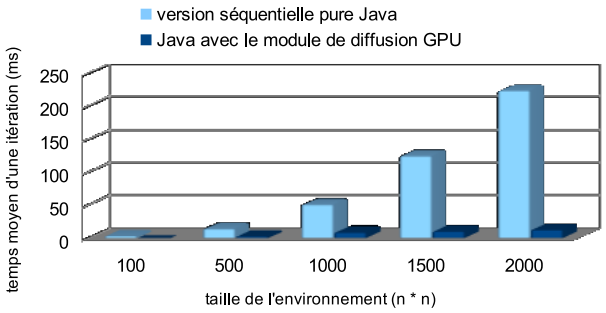

Figure 4. Temps d'exécution de la diffusion : séquentiel et GPU avec JCuda

Comme attendu, les résultats montrent que, même pour la plus petite grille testée (100×100), la version utilisant JCuda s'exécute avec de meilleures performances. Au fur et à mesure que les dimensions de l'environnement sont augmentées, le module GPU surpasse la version séquentielle jusqu'à obtenir des performances plus de 20 fois supérieures pour une taille de 2000×2000.

Pour que ce test soit significatif par rapport à nos objectifs, il nous faut préciser que nous avons pris en compte le fait que le résultat d'une itération doit être rendu disponible pour une utilisation sur le CPU [12]. En effet, dans une simulation classique, les agents devront avoir accès aux résultats à chaque pas de temps pour pouvoir calculer leur comportement. Cette contrainte requiert d'utiliser, pour chaque itération, des primitives qui permettent de forcer la synchronisation entre les exécutions du CPU et du GPU. Ces primitives sont très coûteuses : par exemple, lorsqu'on supprime la contrainte de synchronisation et qu'on laisse le GPU exécuter la diffusion sans être interrompu, on obtient des résultats bien supérieurs, au moins 200 fois plus rapides que

12. Cette technique est régulièrement utilisée dans les bancs d'essai réalisés par Nvidia car de nombreuses applications nécessitent d'alterner et de synchroniser les calculs entre CPU et GPU pour pouvoir utiliser les résultats produits par ce dernier.

le séquentiel. Ainsi, il est important de bien organiser les différentes tâches exécutées par le GPU afin de minimiser les synchronisations nécessaires [13].

Par ailleurs, il faut savoir qu'il est possible de modifier de nombreux paramètres d'exécution pour un même kernel avec Cuda. Certains d'entre eux peuvent influencer les performances dans des proportions très significatives. Par exemple, une astuce peu documentée consiste à systématiquement définir au moins autant de blocs qu'il existe de cœurs sur la carte graphique. Loin d'être une règle clairement établie, nous avons pu constater un impact très important sur les résultats. De plus, comme nous l'avons mentionné, la gestion de la mémoire influe fortement sur l'ampleur des résultats. Dans notre cas, une optimisation très rentable a consisté à faire en sorte que la grille tampon soit uniquement instanciée dans la mémoire du GPU car ses données n'ont pas besoin d'être accédées en dehors du calcul effectué par le GPU pour la diffusion.

Dans l'analyse de ces résultats, il est donc important de garder à l'esprit qu'ils ont été obtenus dans le contexte particulier formé par le logiciel et le matériel utilisés pour cette expérience. Il est possible d'obtenir des résultats très différents en termes de ratio en fonction des configurations. Par contre, ces résultats montrent clairement que le module de diffusion GPU permet le passage à l'échelle alors que ce n'est pas du tout le cas avec la version séquentielle.

6. Simulation du modèle MLE avec le module de diffusion GPU

Sur le modèle MLE, le gain de performance associé à l'utilisation du module de diffusion GPU a été immédiatement significatif. La figure 5 présente quelques-uns des résultats que nous avons obtenus sur le modèle MLE avec différentes tailles d'environnement et de densités de population. Par exemple, pour une densité de 80 % dans un environnement de 2000×2000, il y a 3.2 millions d'agents interagissant sur une grille de 4 millions de cellules.

On voit encore une fois clairement que seule la version GPU permet le passage à l'échelle, ce qui était l'un de nos objectifs. Nous avons pu ainsi augmenter la taille de l'environnement à des valeurs que nous n'avions jamais pu tester auparavant dans des conditions d'exécution assez bonnes pour travailler avec le modèle. En particulier, contrairement à la version Java originale, la vitesse de simulation ne s'écroule pas au fur et à mesure des itérations et varie très peu, ce qui est très appréciable lorsqu'on prototype le modèle. Ainsi, dans cette expérience, la simulation GPU du modèle MLE est environ sept fois plus rapide que la version originale.

On voit cependant aussi que, bien que la différence soit très marquée pour les petites densités de population, le gain tend à diminuer lorsque le nombre d'entités devient important. Ceci s'explique par le fait que les calculs consacrés aux agents,

13. Dans le cas du modèle MLE, cela se traduit par l'utilisation d'une seule synchronisation par pas de temps, demandée une fois que les calculs de toutes les phéromones sont lancés sur le GPU.

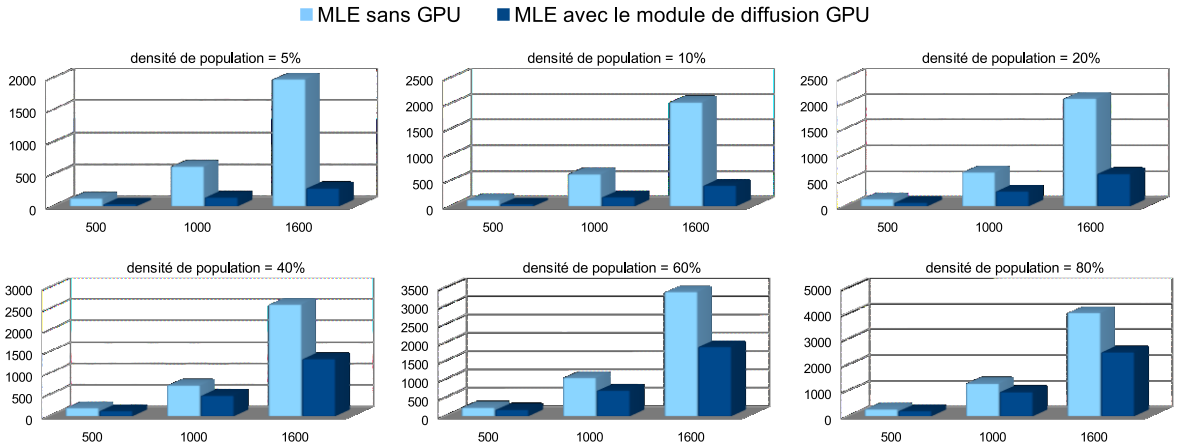

Figure 5. Comparaison des temps d'exécution obtenus sur le modèle MLE, sans et avec le module de diffusion GPU

réalisés par le CPU, prennent de plus en plus de temps si bien que la partie GPU pèse de moins en moins dans le total du temps de simulation.

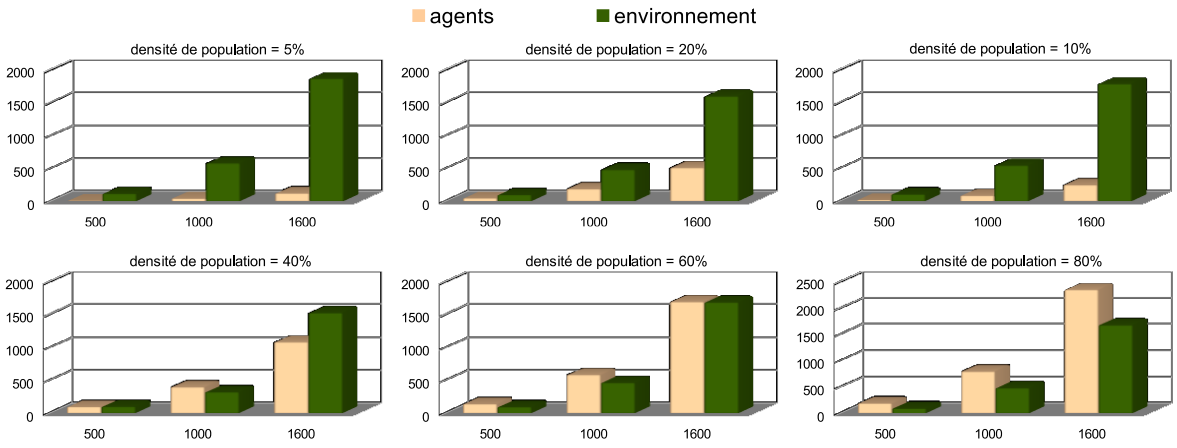

Figure 6. Comparaison des temps d'exécution des agents et de l'environnement sur le modèle MLE sans le module de diffusion GPU

Les figures 6 et 7 permettent de mieux comprendre cet aspect en détaillant le temps pris par chaque partie du modèle, c'est-à-dire (1) les agents et (2) l'environnement, respectivement pour les deux modes de simulations, pure Java et avec le module de diffusion GPU.

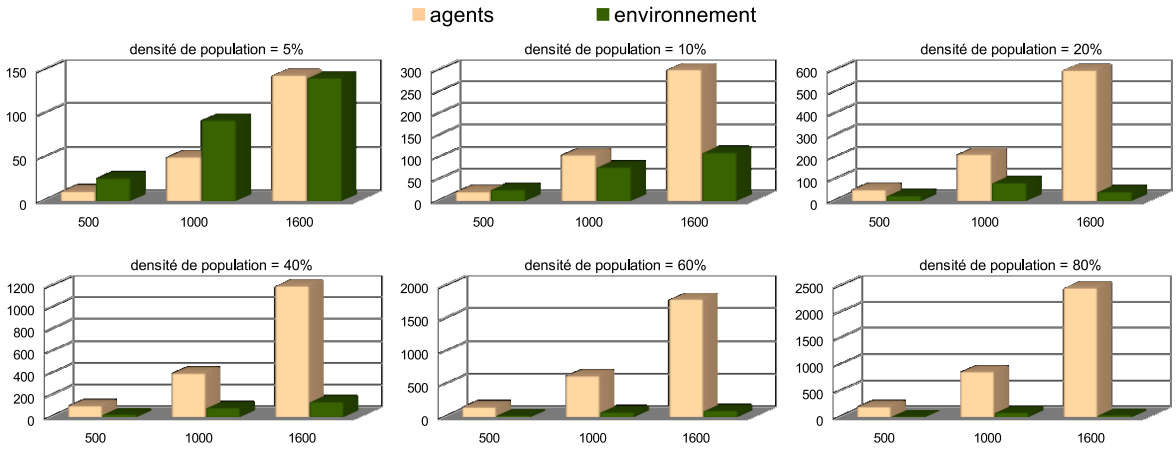

Figure 7. Comparaison des temps d'exécution des agents et de l'environnement sur le modèle MLE avec le module de diffusion GPU

Pour la version sans GPU, on voit bien que le problème majeur réside dans le coût des dynamiques environnementales pour le passage à l'échelle. Mais, bien que la version GPU règle ce problème, on voit aussi qu'au fur et à mesure que le nombre d'agents grossit, les calculs associés augmentent au point de devenir un nouveau goulot d'étranglement lorsqu'on travaille avec de grandes populations d'agents. Et, comme nous l'avons dit, atteindre des niveaux d'émergence supérieurs avec le modèle MLE nécessite d'augmenter non seulement la taille de l'environnement mais aussi le nombre total d'agents initial.

7. Le module de perception de gradients GPU

7.1. Goulot d'étranglement suivant : les agents

Lorsqu'on analyse dans le détail le temps d'exécution des agents (i.e. du temps CPU), on voit qu'une partie importante de leur comportement est consacrée à analyser les différents gradients de phéromone pour calculer leur prochain mouvement. En effet, chaque agent réalise un calcul pour connaître la cellule voisine qui possède le plus, ou le moins, de phéromone d'un certain type pour décider l'orientation de son mouvement futur. Des méthodes telles que *getMaxDirection(attractionField)* ou *getMinDirection(repulsionField)* sont intensivement utilisées par les agents du modèle. De tels calculs nécessitent de sonder l'ensemble des cellules qui se trouvent autour de l'agent pour chaque phéromone d'intérêt, puis de calculer la direction à prendre en fonction des valeurs minimales ou maximales trouvées. La figure 8 illustre ce calcul pour une cellule.

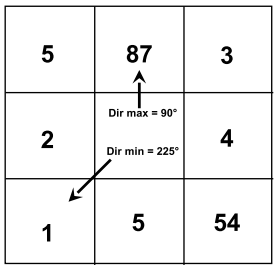

Figure 8. Exemple du calcul des directions min et max pour un gradient de phéromone

Dans le modèle MLE, les processus de diffusion et d'évaporation sont les seules dynamiques environnementales existantes. De fait, pour améliorer plus avant les performances, il était clair que nous n'avions d'autre choix que de réaliser un module GPU chargé d'optimiser l'exécution des agents. Cependant, afin de ne pas renier nos objectifs, il était crucial de ne pas adopter une approche qui aurait pu nous conduire vers du *tout-sur-GPU*. La section suivante présente la solution que nous utilisons et montre comment nous tirons parti du GPU pour les agents sans pour autant modifier leur API.

7.2. Délégation GPU de la perception des agents

En ce qui concerne la manière dont les agents du modèle MLE perçoivent et analysent les gradients de phéromones, il faut remarquer que les calculs correspondants n'impliquent pas l'état de l'agent qui les réalise. D'une manière générale, calculer la direction d'un gradient n'a rien à voir avec l'état dans lequel se trouvent les agents. L'idée est donc d'utiliser à nouveau cette indépendance pour pouvoir effectuer ces calculs à l'aide d'un module GPU sans modifier le modèle agent.

À première vue, il reste cependant un problème de dépendance car ce sont les agents qui déclenchent ces perceptions, et non l'environnement comme c'était le cas pour le processus de diffusion. Ce qui pourrait amener à penser qu'il est nécessaire de passer une partie du comportement des agents dans le module GPU.

Pour surmonter cette difficulté, la solution que nous avons employée consiste à calculer ces perceptions directement dans l'environnement partout et tout le temps. Dit autrement, il s'agit de réifier l'ensemble de ces perceptions dans une unique dynamique environnementale calculée par le GPU.

Cette solution peut sembler contre intuitive dans la mesure où un grand nombre de calculs seront effectués pour rien car non utilisés par les agents. Mais nous bénéficions en fait ici des spécificités de la programmation GPU. En effet, effectuer un calcul sur le GPU pour une cellule prend, en ordre de grandeur, à peu près le même temps que si on le réalise sur toutes les cellules à la fois.

Nous avons donc défini un second module GPU qui implémente cette dynamique environnementale et calcule l'ensemble des perceptions possibles pour les agents. Nous l'appellerons ici le *module perception de gradient GPU*. La particularité du kernel associé repose sur l'utilisation de deux tableaux de données supplémentaires qui sont utilisés pour stocker les directions minimales et maximales pour un gradient dans chaque cellule.

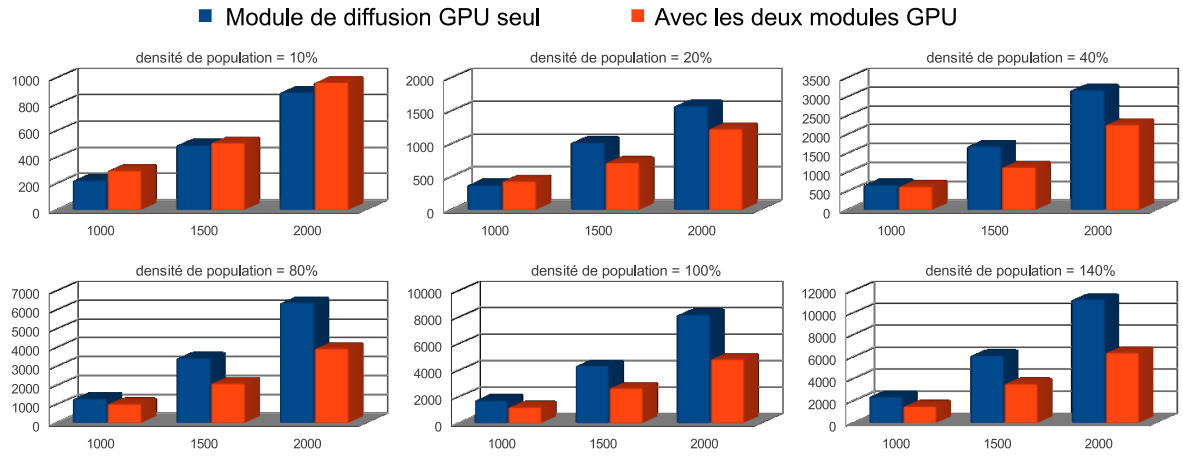

Figure 9. Comparaison des temps d'exécution obtenus sur le modèle MLE avec et sans le module de perception de gradient GPU

7.3. Résultats obtenus grâce au module de perception de gradients GPU

Cette analyse repose sur des tests réalisés sur le modèle MLE avec uniquement le module de diffusion GPU, puis avec les deux modules. Par ailleurs, pour ces simulations, le niveau maximum d'un agent a été fixé à 5, de telle sorte qu'il existe au plus 15 phéromones à gérer.

Sur la figure 9, on voit que l'ajout du module de perception de gradient GPU n'améliore pas la vitesse pour la plus faible densité d'agents testée. Ce résultat négatif montre que, dans le cadre du modèle MLE et de la configuration que nous avons utilisée, il existe un seuil de population en deçà duquel il n'est pas rentable de déclencher un calcul sur GPU pour la perception des gradients. Le pire cas étant celui où l'on met un seul agent dans l'environnement : celui-ci est toujours plus lent lorsqu'on utilise les deux modules. Les tests que nous avons effectués montrent cependant que le surcoût engendré par le deuxième module n'est pas très élevé pour des densités avoisinant les 5 % et qu'il devient quasiment négligeable pour une densité de 10 %. Le module devient efficace dès qu'on considère 20 % de population et des environnements larges.

7.4. Évaluation du seuil de rentabilité du module de perception de gradients GPU

Afin de mieux mesurer l'impact du module de perception de gradients GPU et d'avoir une idée plus précise du seuil à partir duquel il apporte un gain, nous avons réalisé un nouveau banc d'essai avec des agents aux comportements minimalistes définis en 3 étapes : (1) dépôt d'une quantité de phéromone, (2) consultation du gradient et (3) mouvement vers le minimum, seul gradient calculé dans cette expérience. Les figures 10 et 11 présentent quelques-uns des résultats obtenus pour, respectivement, 1 puis 10 phéromones.

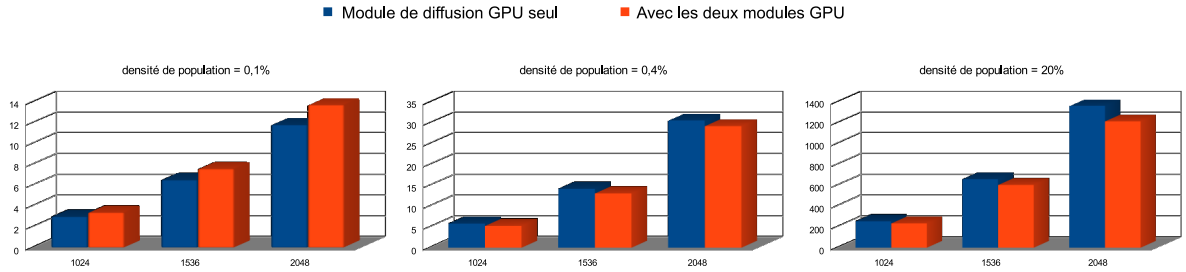

Figure 10. Évaluation du seuil de rentabilité pour une seule phéromone

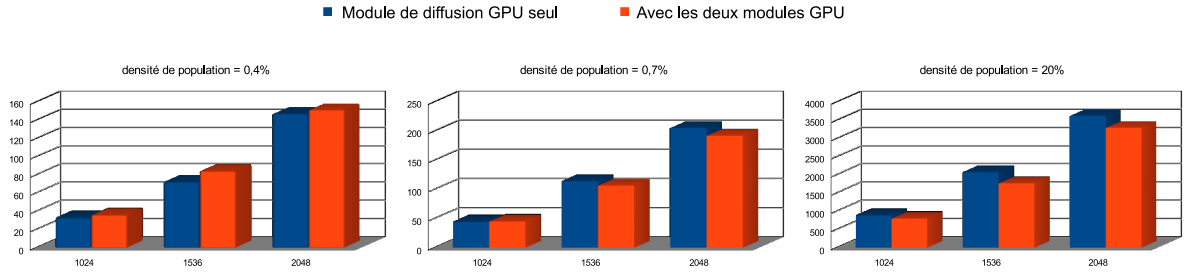

Figure 11. Évaluation du seuil de rentabilité pour 10 phéromones

On voit qu'il faut descendre très bas en termes de densité de population pour trouver le seuil de rentabilité. Pour une seule phéromone, celui-ci se trouve en dessous de 0,4 %. Il est légèrement plus élevé pour 10 phéromones mais reste inférieur à 0,7 %. Une analyse détaillée des différents temps d'exécution (agent et environnement) permet de mieux comprendre. En fait, avec notre configuration, le surcoût lié au calcul du gradient par le GPU se situe environ entre 10 % et 20 % suivant les différents paramètres d'une instance. Dans le même temps, le gain obtenu du côté des agents est du même ordre de grandeur si bien que le seuil de rentabilité se situe environ au moment où leur temps de calcul approche celui pris par l'environnement.

Ainsi, bien que ce seuil soit très difficile à évaluer a priori, il serait relativement facile de le détecter pendant l'exécution d'une simulation grâce à une simple comparaison entre les temps de calcul respectifs des agents et de l'environnement. Il sera

donc intéressant d'envisager un mode de fonctionnement adaptatif capable d'activer ou non l'utilisation du module de perception de gradient GPU, automatiquement, en fonction des ressources matérielles disponibles. C'est un point relativement important compte tenu du fait que TurtleKit est destinée à être une plate-forme portable : les gains de performances qui pourront être obtenus seront fortement dépendants du contexte matériel sous-jacent.

7.5. Autres optimisations possibles sur le modèle MLE

Nos dernières expérimentations avec le modèle MLE nous ont par ailleurs ouvert une nouvelle piste de recherche. En effet, parmi les optimisations que nous avons testées, la désynchronisation entre le CPU et le GPU est particulièrement intéressante. Comme nous l'avons expliqué, par défaut la simulation évolue en alternant les calculs agents (CPU) avec les dynamiques environnementales (GPU) de manière synchrone. Lorsqu'on supprime cette synchronisation, on parallélise de fait le CPU et le GPU si bien que la simulation devient encore beaucoup plus rapide. En fait, l'état d'avancement d'une simulation du modèle MLE étant directement lié au rythme des agents, le temps pris par l'environnement devient littéralement nul, car toujours plus faible que le temps CPU.

Bien sûr ceci ne va pas sans renoncer à un ordre causal dans l'accès aux données. Par conséquent, d'une part toute réplication à l'identique de la simulation devient alors impossible et d'autre part l'évolution du système peut devenir incohérente. Mais dans notre cas il se trouve que le modèle MLE s'avère tout de même robuste et qu'il est possible de retrouver son comportement nominal en jouant sur le nombre d'agents et sur les quantités de phéromones qu'ils déposent.

Ainsi, il nous semble intéressant de travailler sur une évolution du modèle MLE où les agents seraient capables de s'adapter automatiquement aux conditions environnementales, notamment en modifiant eux-mêmes les taux d'évaporation et de diffusion des différentes phéromones.

7.6. TurtleKit 3 et le modèle MLE en ligne

TurtleKit 3 est disponible au téléchargement en version alpha sur turtlekit.org et fonctionne aujourd'hui avec Cuda 5.5 pour la partie GPU. Le modèle MLE peut quant à lui être testé directement en ligne grâce à Java Web Start [14]. Après diverses optimisations, les simulations que nous utilisons habituellement pour travailler sur le modèle MLE lui-même (512×512 avec 10 % de population) sont aujourd'hui environ 10 fois plus rapides que leur équivalent sans GPU, cela même lorsque les calculs liés aux phéromones sont parallélisés sur les différents cœurs du CPU. On monte jusqu'à un ratio de 20 lorsque le CPU et le GPU sont désynchronisés.

14. http://www.lirmm.fr/ fmichel/mle. En cas de dysfonctionnement de Java Web Start, un fichier jar exécutable est aussi disponible au téléchargement. Des vidéos sont aussi disponibles sur cette même page.

8. Généralisation de l'approche

8.1. L'environnement comme abstraction de premier ordre dans les SMA

L'approche que nous avons utilisée consiste donc à transformer des perceptions agents en dynamique environnementale. L'environnement prend donc un peu plus d'importance dans l'architecture de la plate-forme TurtleKit. En fait, pour généraliser notre démarche, il convient de la rapprocher des travaux de recherche qui considèrent l'environnement comme un concept fondamental des SMA.

Considérer l'environnement comme une abstraction de premier ordre est une idée aujourd'hui bien acceptée et son intérêt pour la modélisation et le développement de SMA n'est plus à prouver (Weyns *et al*., 2007). En particulier, elle permet d'augmenter l'efficacité des interactions entre agents. Par exemple, dans (Weyns *et al*., 2006), de véritables véhicules automatisés (*Automated Guided Vehicles AGV*) utilisent un environnement virtuel chargé de calculer la validité de leurs mouvements futurs. Lorsque l'environnement détecte une possible collision future, il établit un ordre de priorité entre les mouvements afin de résoudre automatiquement les conflits spatiaux, sans que les agents aient besoin d'intervenir. Ainsi, les agents n'ont pas besoin de traiter ce problème. Ce qui permet de (1) diminuer la complexité du comportement des agents et ainsi (2) faire en sorte que les agents se focalisent sur leur tâche principale qui est d'aller d'un point A à un point B.

Plus généralement, utiliser l'environnement comme une entité active est une approche très intéressante pour la simplification du processus décisionnel des agents. L'idée sous-jacente est que les agents ont finalement besoin de manipuler des percepts de haut niveau pour calculer leur comportement : ils ne sont pas intéressés par les données environnementales de bas niveau qui nécessitent un traitement pour être intelligibles. Il est donc intéressant de soulager les agents de ces traitements et de déléguer à l'environnement le soin de produire des perceptions de haut niveau à partir des données environnementales brutes (Chang *et al*., 2005). Une telle approche permet de concevoir des SMA où il existe une séparation claire entre ce qui relève véritablement des agents et ce qui peut être calculé par ailleurs dans l'environnement.

8.2. Délégation GPU des perceptions agents

Dans le cadre de notre expérience, considérer l'environnement comme une partie fondamentale du système est au cœur de notre solution. Cela nous a permis d'atteindre nos deux objectifs : (1) conserver l'accessibilité de notre modèle agent dans un contexte GPU et (2) passer à l'échelle et travailler avec un grand nombre d'agents sur de grandes tailles d'environnement.

Pour généraliser notre expérience à d'autres simulations de SMA, nous proposons un principe de conception qui repose sur deux piliers : (1) l'environnement modélisé comme une entité active et (2) la prise en compte du contexte particulier que représente

la programmation GPU. Ce principe de conception, que nous appelons *délégation GPU des perceptions agents*, peut-être énoncé de la manière suivante :

> *Tout calcul de perception agent qui n'implique pas l'état de l'agent peut être transformé dans une dynamique endogène de l'environnement, et ainsi considéré pour une implémentation dans un module GPU indépendant.*

Le principal intérêt de ce principe est de promouvoir une réutilisation simplifiée des modules GPU qui seront développés selon celui-ci, car ils n'auront aucun lien avec le modèle d'agent qui les utilisera. En dehors du contexte GPU, on retrouve cette idée de délégation d'une partie des processus agents dans l'environnement dans (Payet *et al*., 2006) avec pour objectif de (1) réduire la complexité du modèle et surtout (2) faciliter la réutilisabilité et l'intégration des différents processus qui sont définis. Dans notre cas, cela nous a effectivement permis d'intégrer le calcul sur GPU sans avoir à modifier l'API du modèle agent. Par exemple nous avons pu réutiliser dans TurtleKit les modules GPU présentés avec d'autres modèles agents qui utilisent des phéromones (e.g. les modèles à base de fourmis).

C'est pourquoi nous pensons que l'intérêt du principe de délégation GPU des perceptions agents ne se limite pas aux objectifs que nous avons poursuivis. Celui-ci peut être appliqué non seulement sur d'autres plates-formes similaires (e.g. NetLogo), mais aussi en dehors du contexte d'une simulation spatialisée, car un espace topologique n'est qu'un cas particulier d'environnement. En effet, si on adopte un point de vue où l'environnement est défini par toutes les données qui ne font pas partie des agents alors, pour peu qu'il soit constitué de données distribuées que les agents doivent analyser après perception, on peut lui déléguer le calcul de cette analyse si l'état des agents en est indépendant. Par exemple, dans une simulation du système de la bourse, on peut imaginer que les agents calculent tous de la même manière certains indicateurs sur différents sous groupes de données. Il peut alors être opportun de déporter les calculs correspondants dans l'environnement afin que les agents accèdent directement aux résultats souhaités. Ainsi, bien qu'envisager le principe de délégation soit bien sûr plus intuitif dans le cas d'environnements spatialisés, le point important se situe dans la structure des données manipulées : celles-ci doivent être distribuées et peu dépendantes les unes des autres afin qu'une implémentation GPU efficace puisse être envisagée.

Par ailleurs, le principe de délégation GPU peut aussi être utilisé dans le cadre d'une approche tout-sur-GPU. Tout son intérêt est de promouvoir la réutilisabilité des modules GPU développés dans différents contextes, ceci grâce à une séparation claire entre le modèle agent et le modèle environnemental.

Enfin, sur un plan conceptuel, il est important de noter que bien que la modélisation de l'environnement soit ici prépondérante, cela n'entame en rien l'autonomie décisionnelle des agents. En effet, même si le fait de déporter une partie du comportement des agents dans l'environnement peut laisser penser que leur autonomie est atteinte, il est important de remarquer que l'énoncé du principe de délégation stipule

que son application ne vaut que si l'état de l'agent n'est pas impliqué dans les calculs considérés. De fait, ces calculs ne font pas intervenir de décision de la part de l'agent dans le sens où le résultat est entièrement indépendant de celui-ci : ils ne modélisent donc pas un comportement au sens propre mais représentent simplement une perception de haut niveau, utilisée par l'agent pour faire sa décision en toute autonomie.

9. Conclusion et perspectives

Il est évident que le GPGPU est une technologie d'avenir pour l'étude des systèmes complexes modélisés à l'aide du paradigme multi-agent. Mais la programmation GPU est si spécifique que de nombreux efforts de développement mêlant SMA et GPU sont tout simplement perdus : les programmes obtenus sont trop complexes.

Dans ce contexte, l'un des objectifs que nous poursuivons est de profiter de la puissance du GPU dans TurtleKit tout en conservant inchangée l'API de son modèle agent. Dans cet article, nous avons tout d'abord présenté la manière dont nous avons réalisé le module de diffusion GPU en appliquant une approche centrée environnement qui consiste à traduire des dynamiques environnementales en code GPU.

Les gains de performances que nous avons obtenus sont sans doute encore bien inférieurs à ceux que nous aurions pu obtenir en appliquant une approche tout-sur-GPU. Pour l'instant, les contraintes que nous nous sommes imposées en termes d'accessibilité sont bien sûr un facteur limitant mais, même avec cette limite, les différents résultats que nous avons obtenus sont déjà très significatifs et encourageants.

De plus, il faut rappeler ici que ces résultats sont fortement liés à notre configuration. En particulier, nous avons utilisé une carte graphique NVidia Quadro 4000 contenant 256 cœurs et ayant une puissance théorique de 486.4 gigaflops en simple précision. La récente NVidia Tesla K10 contient 2 GPU équipés chacun de 1536 cœurs : un total de 3072 cœurs sur une seule carte pour une puissance théorique de 4.58 teraflops. Les perspectives offertes par la programmation GPU sont donc extrêmement prometteuses et nous encouragent à continuer d'intégrer progressivement de nouveaux modules GPU dans TurtleKit.

Dans le cadre de cette démarche, nous avons proposé le principe de délégation GPU des perceptions agents et montré comment celui-ci nous a permis d'aller plus loin et de créer un deuxième module GPU. D'une manière générale, nous pensons que ce principe de conception constitue un moyen intéressant d'adresser le problème de la réutilisabilité dans le cadre de la programmation GPU de simulations multi-agents. En effet, les modules ainsi développés ne reposent pas sur un modèle d'agent particulier, ce qui permet d'envisager l'intégration de code GPU dans une plate-forme classique de manière simplifiée et itérative.

Nous comptons notamment travailler avec d'autres modèles afin d'identifier de nouveaux modules en appliquant le principe de délégation GPU. À long terme notre objectif est donc de développer une librairie de modules GPU dédiée à la simulation de systèmes multi-agents. Aujourd'hui, plusieurs librairies GPU génériques ont déjà été

implémentés dans d'autres domaines : *Nvidia CuBLAS* (*Compute Unified Basic Linear Algebra Subprograms*), *NPP* (*Nvidia Performance Primitives*) pour le traitement du signal, l'image et la vidéo, GPU AI *path finding*, etc.

Sur des aspects plus prospectifs, nous pensons que le GPGPU offre un cadre idéal pour la mise en place de simulations basées sur le modèle influence/réaction (Michel, 2007). En effet, dans ce modèle, le goulot d'étranglement majeur réside dans le calcul de la réaction de l'environnement aux influences des agents. Ainsi, lorsqu'on met en place un tel principe de simulation, on est très rapidement limité par la puissance du CPU et peu de combinaisons d'influences peuvent être considérées. Grâce au GPGPU, il devrait être possible d'implémenter ce modèle de manière beaucoup plus efficace et même d'envisager des versions plus élaborées capables de prendre en compte des aspects multiniveaux (e.g. (Morvan *et al.*, 2011)).

Par ailleurs, alors que nous avons dans cet article concentré nos efforts sur les relations entre agent et environnement, un point intéressant de cette perspective est qu'elle ouvre la voie aux interactions entre agents. En effet, dans le modèle influence/réaction, la réaction calcule le résultat des interactions entre agents en combinant les influences qu'ils ont mises dans l'environnement.

Toujours dans le cadre des interactions entre agents, il serait aussi intéressant de regarder dans quelle mesure une approche orientée interaction pour la conception de simulation telle que IODA (Kubera *et al.*, 2011) pourrait bénéficier du GPGPU, notamment car dans cette perspective les interactions sont modélisées comme des éléments génériques, indépendants des agents, et que les calculs associés sont effectués à l'aide de structures matricielles appelées *matrices d'interaction*.

À plus long terme il s'agit pour nous de produire des outils de simulation qui nous permettront d'étudier des modèles multi-agents originaux reposant sur des dynamiques environnementales toujours plus complexes. Nous tenons ainsi à conclure en soulignant que les possibilités offertes par le GPGPU ne se réduisent pas à permettre une optimisation des modèles existants. Elles vont bien au-delà dans le sens où elles permettront de mettre en place de nouvelles dynamiques environnementales originales qui n'auront pas d'équivalent séquentiel : le GPGPU est un paradigme de programmation en soit. À ce titre, il est certain que le GPGPU va ouvrir de nouvelles perspectives pour les SMA en général.

Remerciements

Je tiens à remercier les différentes personnes qui ont effectué le travail d'évaluation et de relecture pour cet article, de sa version pour les JFSMA 2013 à cette dernière.

Bibliographie

Beurier G., Simonin O., Ferber J. (2002, December). Model and simulation of multi-level emergence. In *2^{nd} ieee international symposium on signal processing and information technology, isspit'02*, p. 231–236. Marrakesh, Morocco.

Bourgoin M. (2013). *Abstractions performantes pour cartes graphiques*. Thèse de doctorat non publiée, Université Pierre et Marie Curie.

Chang P. H., Chen K.-T., Chien Y.-H., Kao E., Soo V.-W. (2005). From reality to mind: A cognitive middle layer of environment concepts for believable agents. In D. Weyns, H. V. D. Parunak, F. Michel (Eds.), *Environments for multi-agent systems, first international workshop, e4mas 2004, new york, ny, usa, july 19, 2004, revised selected papers*, vol. 3374, p. 57–73. Springer.

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

Demeulemeester A., Hollemeersch C.-F., Mees P., Pieters B., Lambert P., Walle R. Van de. (2011). Hybrid path planning for massive crowd simulation on the gpu. In *Proceedings of the 4th international conference on motion in games*, p. 304–315. Berlin, Heidelberg, Springer-Verlag.

D'Souza R. M., Lysenko M., Marino S., Kirschner D. (2009). Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 spring simulation multiconference*, p. 21:1–21:12. San Diego, CA, USA, Society for Computer Simulation International.

Gutknecht O., Ferber J., Michel F. (2001). Integrating tools and infrastructures for generic multi-agent systems. In *Proceedings of the fifth international conference on autonomous agents, aa 2001*, p. 441–448. ACM Press.

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

Laville G., Mazouzi K., Lang C., Marilleau N., Philippe L. (2012). Using GPU for Multi-agent Multi-scale Simulations. In S. Omatu, J. F. De Paz Santana, S. R. González, J. M. Molina, A. M. Bernardos, J. M. C. Rodríguez (Eds.), *Distributed computing and artificial intelligence*, vol. 151, p. 197-204. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-642-28765-7_23

Lysenko M., D'Souza R. M. (2008). A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, vol. 11, n° 4, p. 10. http://jasss.soc.surrey.ac.uk/11/4/10.html

Michel F. (2007). Le modèle IRM4S, de l'utilisation des notions d'influence et de réaction pour la simulation de systèmes multi-agents. *Revue d'Intelligence Artificielle*, vol. 27, n° 5-6, p. 757–779.

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

Michel F., Ferber J., Drogoul A. (2009, 22 05). Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective. In Danny Weyns, Adelinde Uhrmacher (Eds.), *Multi-Agent Systems: Simulation and Applications*, p. 3–52. CRC Press - Taylor & Francis.

Morvan G., Veremme A., Dupont D. (2011). IRM4MLS: The Influence Reaction Model for Multi-Level Simulation. In T. Bosse, A. Geller, C. M. Jonker (Eds.), *Multi-Agent-Based Simulation XI - International Workshop, MABS 2010, Toronto, Canada, May 11, 2010, Revised Selected Papers*, vol. 6532, p. 16-27. Springer Berlin Heidelberg.

North M., Tatara E., Collier N., Ozik J. (2007, November). Visual agent-based model development with Repast Simphony. In *Agent 2007 conference on complex interaction and social emergence*, p. 173–192. Argonne, IL, USA, Argonne National Laboratory.

Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E. *et al*. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, vol. 26, n° 1, p. 80–113.

Payet D., Courdier R., Sébastien N., Ralambondrainy T. (2006). Environment as support for simplification, reuse and integration of processes in spatial mas. In *Proceedings of the 2006 IEEE international conference on information reuse and integration, IRI - 2006: Heuristic systems engineering, september 16-18, 2006, waikoloa, hawaii, usa*, p. 127-131. IEEE Systems, Man, and Cybernetics Society.

Richmond P., Walker D., Coakley S., Romano D. (2010). High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, vol. 11, n° 3, p. 334-347.

Silva A. R. D., Lages W. S., Chaimowicz L. (2010, janvier). Boids that see: Using self-occlusion for simulating large groups on gpus. *Computers in Entertainment (CIE) - Special issue: Games*, vol. 7, n° 4, p. 51:1–51:20.

Sklar E. (2007). Netlogo, a multi-agent simulation environment. *Artificial Life*, vol. 13, n° 3, p. 303-311.

Weyns D., Boucké N., Holvoet T. (2006). Gradient field-based task assignment in an agv transportation system. In *Proceedings of the fifth international joint conference on autonomous agents and multiagent systems*, p. 842–849. New York, NY, USA, ACM.

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