Intégration du calcul sur GPU dans la plate-forme de simulation multi-agent générique TurtleKit 3

Fabien Michel

Intégration du calcul sur GPU dans la plate-forme de simulation multi-agent générique TurtleKit 3

Fabien Michel
fmichel@lirmm.fr

Laboratoire d'Informatique, de Microélectronique, et de Robotique Montpellier
Université Montpellier II - CNRS
161 rue Ada, Montpellier Cedex 2, France

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 de SMA spécifiquement dédié au contexte GPGPU.

Mots-clés : *Calcul haute performance, GPGPU, simulation multi-agent*

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.

Keywords: *HPC, GPGPU, MABS*

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 [8] peut nécessiter beaucoup de puissance de calcul. De fait, les performances d'exécution représentent souvent un obstacle qui limite fortement le cadre dans lequel un modèle peut être étudié, notamment ce qui concerne le nombre d'entités et la taille de leur environnement.

Parallèlement, dans le cadre du calcul haute performance, la programmation sur carte graphique (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 [3]. La majorité des cartes graphiques 3D actuelles sont 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 programmation très particulier. Ce contexte architectural nécessite notamment de suivre les principes de la programmation par traitement de flot de données (stream processing paradigm [1]) [10]. 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 [6].

Ainsi, malgré l'existence de travaux démontrant les possibilités de gains offertes par la

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

programmation sur GPU (jusqu'à des centaines de fois plus rapide qu'avec des plates-formes classiques [6]), 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 [13] et RePast [9], n'intègre aujourd'hui de GPGPU.

Nous décrivons dans cet article comment nous avons réalisé l'intégration de calcul GPU dans TurtleKit, une plate-forme de simulation multi-agent générique [7]. 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 la programmation GPU pour développer des simulations multi-agents et identifient leurs limites par rapport à nos objectifs. La section 3 présente TurtleKit et le modèle de simulation multi-agent que nous avons utilisé pour tester l'intégration de modules GPU dans la future 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 présente un second module GPU obtenu en transformant certains calculs réalisés par les agents en dynamiques environnementales calculées par le GPU. La section 6 présente une généralisation de notre approche et propose le principe de *délégation GPU des perceptions agents*. La section 7 conclut l'article et discute des perspectives qui lui sont associées.

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. 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. [4]), la biologie (e.g. [5]) ou encore la simulation de mouvements collectifs complexes comme le flocking (e.g. [12]).

Dans [11], des gains de performances impressionnants sont obtenus sur le logiciel FLAME (cellular level agent-based simulation framework) grâce au GPU. 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, [6] propose de considérer la reprogrammation GPU de toute une classe de modèles multi-agents en s'attaquant aux simulations multi-agents utilisant une grille en deux dimensions pour environnement. Dans ce type de modèles, parfois quali-

fiés de spatialisés, un environnement 2D est discrétisé en cellules sur lesquelles se déplacent des agents. La très utilisée plate-forme NetLogo [13] possède un tel modèle. [6] 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 de simulation 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.

D'une manière générale, la littérature montre clairement que les difficultés techniques liées à l'utilisation de 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. Ainsi, les approches *tout-sur-GPU* sont restreintes à un domaine d'application spécifique et/ou nécessitent des connaissances avancées en programmation GPU.

3 Intégration du calcul sur GPU dans TurtleKit 3

3.1 TurtleKit et le modèle MLE

À l'instar de NetLogo, TurtleKit [2] [7] est une plate-forme qui utilise un modèle multi-agent spatialisé. TurtleKit est implémentée en Java et repose sur un modèle agent inspiré 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 adopté une approche intermédiaire qui consiste à intégrer de 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 [1]. 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 correspondant 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

2. http ://www.turtlekit.org

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.

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 nombre minimum d'agents de niveau i-1 pour voir apparaître des structures de niveau i.

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.

Dans un premier temps, notre attention s'est donc focalisée sur le calcul des dynamiques de diffusion et d'évaporation. En effet, ces dynamiques nécessitent d'effectuer des calculs pour chaque cellule de la grille à chaque pas de temps. Ainsi, bien que ces calculs soient simples, la complexité associée évolue de manière quadratique avec la taille de l'environnement. Nous avons donc décidé de traduire ces dynamiques en code GPU.

Par rapport à nos objectifs, ce choix était aussi naturel car ces dynamiques sont complètement découplées du modèle comportemental d'un agent. Il est donc possible de créer un module GPU correspondant sans avoir à modifier quoi que ce soit de l'API du modèle agent. De plus, comme nous allons le voir, elles sont aussi relativement faciles à traduire car les calculs correspondants sont naturellement spatialement distribués, ce qui convient bien au calcul sur GPU.

4 Le module de diffusion GPU

4.1 Traduction GPU de l'évaporation

Pour expliquer simplement la traduction des dynamiques d'évaporation et de diffusion en code GPU, nous présentons ici 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 de la manière suivante :

```
Pour i de 0 à largeur faire
    Pour j de 0 à hauteur faire
        grille[i][j] ←
                grille[i][j] × evapCoef ;
    Fin Pour
Fin Pour
```

Algorithme 1: évaporation en séquentiel

Voyons maintenant quelques principes de 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 en grille de 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.

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 définir 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 contenant 1024×1024 threads. 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 utilisés pour l'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 programmé en GPU de la manière suivante :

```
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 Si
```

Algorithme 2: évaporation en GPU

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. Ensuite, un test est 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.

Le kernel correspondant à la dynamique de diffusion est un peu plus complexe car il nécessite l'utilisation d'une grille de données temporaire. Sa traduction a été cependant relativement simple et repose sur un principe similaire. Nous avons ainsi développé un module GPU pour l'évaporation et la diffusion que nous appellerons par la suite le *module de diffusion GPU* par simplicité.

Étant donné nos objectifs en termes d'accessibilité, il était fondamental de conserver le langage Java. C'est pourquoi nous avons utilisé la librairie JCuda (Java bindings for Cuda [3]) qui permet de commander l'exécution de kernels GPU, écrits en langage Cuda, directement depuis Java.

3. Compute Unified Device Architecture, Cuda est une plate-forme de programmation développée par Nvidia pour les cartes graphiques de cette marque. Elle possède actuellement le plus grand nombre de fonctionnalités par rapport aux autres solutions comme OpenCL.

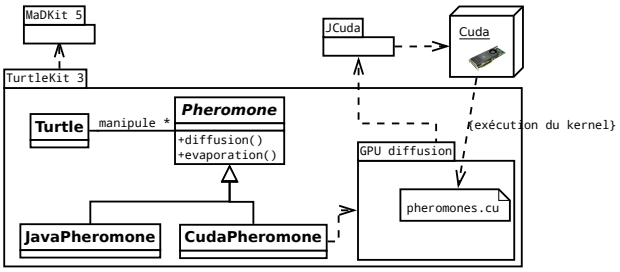

FIGURE 1 – Intégration de la diffusion GPU dans TurtleKit 3

La figure 1 illustre l'intégration du module de diffusion GPU dans le prototype de TurtleKit 3. En particulier, on peut voir qu'un agent (une turtle) manipule indifféremment, suivant le contexte matériel, des phéromones utilisant une implémentation classique de la diffusion ou le module de diffusion GPU. Le modèle agent n'a donc pas besoin d'être modifié. 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 dans le fichier Cuda *pheromones.cu*.

4.2 Résultats obtenus par le module de diffusion GPU

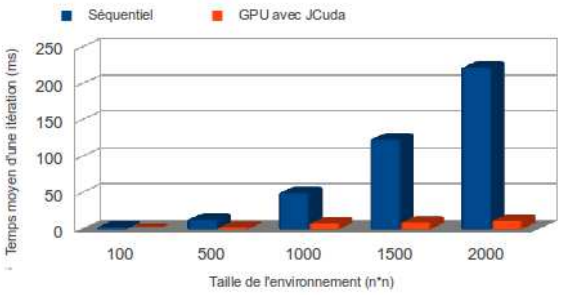

FIGURE 2 – Temps d'exécution de la diffusion : séquentiel et GPU avec JCuda

La figure 2 compare les résultats obtenus avec le module de diffusion GPU et avec une implémentation séquentielle du processus de diffusion. Ces tests ont été réalisés en dehors de toute autre simulation afin d'éviter les bruits provenant d'autres traitements. Ces tests ont été réalisés à l'aide d'un Xeon @ 2.67GHz et d'une carte graphique Nvidia Quadro 4000 dotée de 256 cœurs, ce qui constitue une valeur moyenne

à l'heure actuelle.

La figure 2 montre les résultats obtenus avec différentes tailles d'environnement pour une unique phéromone. Comme attendu, les résultats montrent que même pour la plus petite grille (100×100) , la version utilisant JCuda s'exécute avec de meilleures performances. Au fur et à mesure que la taille de l'environnement est augmentée, le module GPU surpasse la version séquentielle jusqu'à obtenir des performances plus de 20 fois supérieures pour un environnement de 2000×2000.

Pour que ce test soit significatif par rapport à nos objectifs, il faut noter que nous avons pris en compte le fait que le résultat d'une itération doit être rendu disponible pour une utilisation sur le CPU. En effet, dans une simulation classique, les agents devront avoir accès aux résultats à chaque pas de simulation pour pouvoir calculer leur comportement. Cette contrainte requiert d'appeler à chaque pas de temps des primitives qui permettent la synchronisation entre les exécutions du CPU et du GPU. Ces primitives sont coûteuses : si on laisse le GPU exécuter tous ses traitements sans être interrompu, on obtient des résultats encore 20 fois supérieurs, environ 200 fois plus rapide que le séquentiel.

Par ailleurs, il faut savoir qu'il existe de nombreux paramètres qui peuvent être pris en compte pour l'exécution d'un kernel avec Cuda. Certains d'entre eux peuvent influencer les performances dans des proportions 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 important sur les résultats.

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 le cas avec la version séquentielle.

5 Le module GPU perception de gradients

5.1 Prochaine étape : les agents

L'intégration du module de diffusion GPU dans TurtleKit n'a pas posé de problème grâce à sa modularité. Le gain de performance associé a été immédiatement significatif. 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.

Cependant, 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. En augmentant progressivement le nombre d'agents, il nous est alors apparu évident que la majorité du temps d'exécution de la simulation était maintenant utilisée par les agents pour calculer leur comportement. En fait, les agents passaient la plupart de leur temps (i.e. du temps CPU) à analyser les différents gradients de phéromone pour calculer leur 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 3 illustre ce calcul pour une cellule.

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 sur 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.

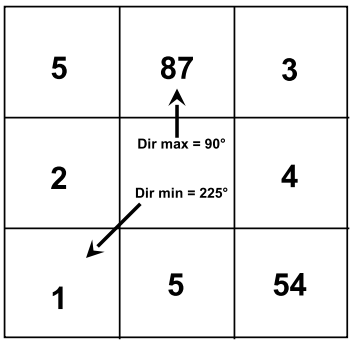

FIGURE 3 – Exemple du calcul des directions min et max pour un gradient de phéromone

5.2 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. Calculer la direction d'un gradient n'a rien à voir avec l'état dans lequel se trouve l'agent. 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 c'est 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 ils ne seront pas 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.

5.3 Analyse des résultats obtenus grâce au module de perception de gradient GPU

Cette analyse repose sur des tests réalisés sur le modèle MLE avec uniquement le module de diffusion GPU, puis avec les deux modules. Le matériel utilisé pour obtenir ces résultats est le même que dans la section 4.2. Très peu documentées, les sources de ce projet sont cependant disponibles sur www.turtlekit.org. Elles ont été testées sous Linux avec Cuda 4 et JCuda-0.4.1. Par ailleurs, pour ces simulations, le niveau maximum d'un agent a été fixé à 5, de telle sorte qu'il existe 15 phéromones à gérer.

La figure 4 compare la vitesse d'exécution pour différentes tailles d'environnement et populations d'agents. Par exemple, pour une densité de 140% dans un environnement de 2000×2000, il y a 5.6 millions d'agents interagissant sur une grille de 4 millions de cellules.

La figure 4 montre 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 peut être expliqué par le coût supplémentaire induit par les synchronisations additionnelles, entre CPU et GPU, requises par les tableaux de données supplémentaires. Cela montre aussi que, dans le cadre 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 un seul agent se trouve dans un large environnement. Et en effet, ce pire cas est toujours plus lent lorsqu'on utilise les deux modules. Cependant, les tests que nous avons effectués montrent que le surcoût engendré par l'utilisation du deuxième module n'est pas très élevé pour des densités avoisinant les 5%, et qu'il devient quasiment négligeable pour une densité de 10%, valeur basse pour le modèle MLE.

Ainsi, même avec le surcoût induit par les appels GPU supplémentaires, le module de perception de gradient GPU devient efficace dès qu'on considère une densité de population de 20% et des environnements larges. Pour les valeurs supérieures, la simulation est toujours plus

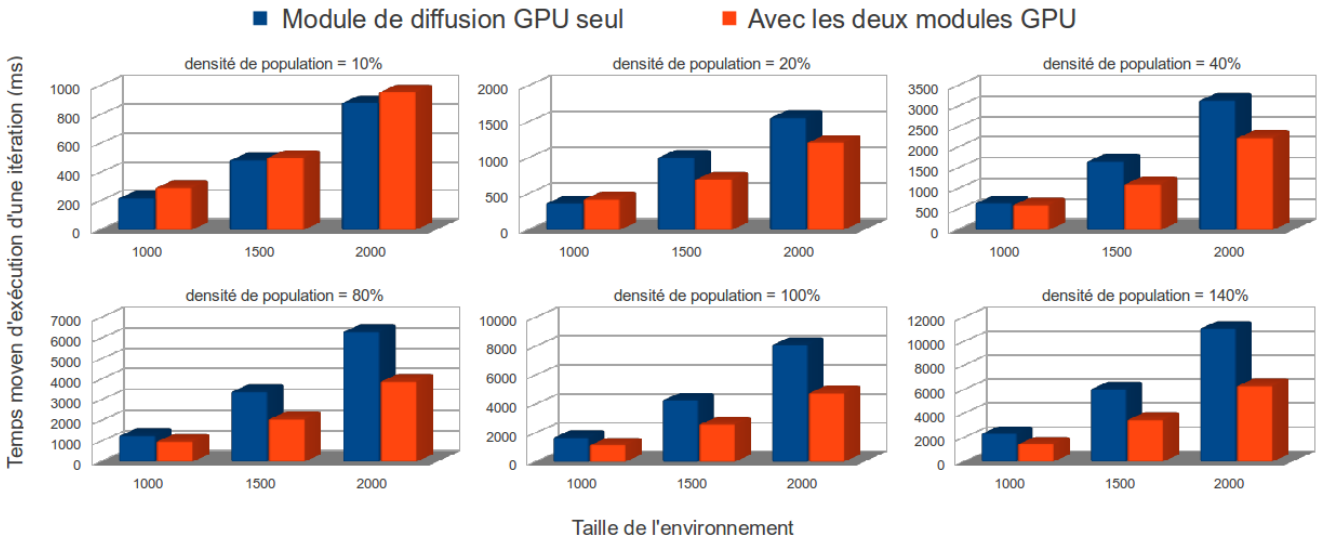

FIGURE 4 – Comparaison des temps d'exécution obtenus sur le modèle MLE avec et sans le module de perception de gradient GPU

rapide avec les deux modules, jusqu'à presque deux fois pour le plus gros cas que nous avons testé. Si on prend en compte que nous avons utilisé une carte graphique moyenne, ces résultats sont très encourageants et montrent la faisabilité et l'intérêt de l'approche que nous proposons, surtout en ce qui concerne le passage à l'échelle.

6 Généralisation de l'approche

6.1 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 aujourd'hui une idée bien acceptée et son intérêt pour la modélisation et le développement de SMA n'est plus à prouver [15]. En particulier, elle permet d'augmenter l'efficacité des interactions entre agents. Par exemple, dans [14], de véritables véhicules automatisés (Automated Guided Vehicles AGV) utilise 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 comportemental 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 données de perception haut niveau à partir de données environnementales brutes [2].

6.2 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, nous proposons un principe de conception qui repose sur deux piliers : (1) l'utilisation de l'environnement 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. 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).

Plus généralement, nous pensons que son intérêt ne se limite pas aux objectifs que nous avons ici 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. Il peut même 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.

7 Conclusion et perspectives

il est évident que le GPU 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, notre travail a un objectif : 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 comment 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 imposés en terme d'accessibilité sont bien sûr un facteur limitant mais, même avec cette limite, les résultats obtenus par ce premier module sont déjà très significatifs.

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 alors que la récente Tesla K10 contient 2 GPU équipés chacun de 1536 cœurs : un total de 3072 cœurs sur une seule carte. 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.

Nous avons ensuite proposé le principe de délégation GPU des perceptions agents et montré comment celui-ci nous a permis d'aller plus loin dans notre objectif. 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 pour les SMA. 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. Nous pensons qu'il est aujourd'hui intéressant d'adopter une démarche similaire dans le cadre des SMA.

Références

[1] Grégory Beurier, Olivier Simonin, and Jacques Ferber. Model and simulation of multi-level emergence. In *2^{nd} IEEE International Symposium on Signal Processing and Information Technology, ISSPIT'02*, pages 231–236, Marrakesh, Morocco, December 2002.

[2] Paul H. Chang, Kuang-Tai Chen, Yu-Hung Chien, Edward Kao, and Von-Wun Soo. From reality to mind : A cognitive middle layer of environment concepts for believable agents. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems, First International Workshop, E4MAS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3374 of *LNAI*, pages 57–73. Springer, 2005.

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10) :1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units.

[4] Aljosha Demeulemeester, Charles-Frederik Hollemeersch, Pieter Mees, Bart Pieters, Peter Lambert, and Rik Van de Walle. Hybrid path planning for massive crowd simulation on the gpu. In *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 304–315, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Roshan M. D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 21 :1–21 :12, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[6] Mikola Lysenko and Roshan M. D'Souza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4) :10, 2008.

[7] Fabien Michel, Grégory Beurier, and Jacques Ferber. The TurtleKit simulation platform : Application to complex systems. In Alain Akono, Emmanuel Tonyé, Albert Dipanda, and Kokou Yétongnon, editors, *Workshops Sessions, First International Conference on Signal & Image Technology and Internet-Based Systems SITIS' 05*, pages 122–128. IEEE, november 2005.

[8] Fabien Michel, Jacques Ferber, and Alexis Drogoul. Multi-Agent Systems and Simulation : a Survey From the Agents Community's Perspective. In Danny Weyns and Adelinde Uhrmacher, editors, *Multi-Agent Systems : Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 3–52. CRC Press - Taylor & Francis, 05 2009.

[9] M.J. North, E. Tatara, N. Collier, and J. Ozik. Visual agent-based model development with Repast Simphony. In *Agent 2007 Conference on Complex Interaction and Social Emergence*, pages 173–192, Argonne, IL, USA, November 2007. Argonne National Laboratory.

[10] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[11] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, 11(3) :334–347, 2010.

[12] Alessandro Ribeiro Da Silva, Wallace Santos Lages, and Luiz Chaimowicz. Boids that see : Using self-occlusion for simulating large groups on gpus. *Comput. Entertain.*, 7(4) :51 :1–51 :20, January 2010.

[13] Elizabeth Sklar. Netlogo, a multi-agent simulation environment. *Artificial Life*, 13(3) :303–311, 2007.

[14] Danny Weyns, Nelis Boucké, and Tom Holvoet. Gradient field-based task assignment in an agv transportation system. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 842–849, New York, NY, USA, 2006. ACM.

[15] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14 :5–30, 2007. 10.1007/s10458-006-0012-0.