Méthodologie pour la modélisation et l'implémentation de simulations multi-agents utilisant le GPGPU

E. Hermellin[a] hermellin@lirmm.fr
F. Michel[a] fmichel@lirmm.fr

[a]Laboratoire Informatique Robotique et Microéléctronique de Montpellier (LIRMM), Université de Montpellier, France

Résumé

L'utilisation du GPGPU (General-Purpose computing on Graphics Processing Units) dans le cadre de la simulation multi-agent permet de lever, en partie, les contraintes liées au passage à l'échelle. Cependant, à cause de l'architecture massivement parallèle des GPU (Graphics Processing Units) sur laquelle repose le GPGPU, les modèles voulant bénéficier des avantages de cette technologie doivent être adaptés au préalable. Prenant le parti de ne pas suivre le courant actuel qui vise à rendre transparent l'utilisation du GPGPU dans le but de simplifier son usage, le principe de délégation GPU propose plutôt de transformer un modèle afin qu'il bénéficie des avantages du GPGPU mais sans cacher la technologie utilisée. Dans cet article, nous présentons une méthodologie basée sur la délégation GPU pour le développement de simulations multi-agents. L'idée est de proposer une généralisation du processus d'application du principe de délégation GPU que nous expérimenterons ensuite sur deux cas d'études afin de définir quels sont les avantages et limites d'une telle approche.

Mots-clés : *Simulation multi-agent, GPGPU, Méthodologie, Délégation GPU*

Abstract

Using General-Purpose computing on Graphics Processing Units (GPGPU) in Multi-Agent Based Simulation (MABS) appears to be very promising as it allows to use the massively parallel architecture of the GPU (Graphics Processing Units) to improve the scalability of MABS models. However, this technology relies on a highly specialized architecture, implying a very specific programming approach. So, it turns out that MAS models need to be adapted if they want to benefit from GPU power. Contrary to some recent research works that propose to hide GPU programming to ease the use of GPGPU, we present in this paper a methodology for modeling and implementing MABS using GPU programming. The idea is to be able to consider any kind of MABS rather than addressing a limited number of cases. This methodology defines the iterative process to be followed to transform and adapt a model so that it takes advantage of the GPU power without hiding the underlying technology. We experiment this methodology on two MABS models to test its feasibility and highlight the advantages and limits of this approach.

Keywords: *MABS, GPGPU, Methodology, GPU delegation*

1 Introduction

Dans le cadre de la simulation multi-agent, le passage à l'échelle et les performances d'exécution représentent souvent des obstacles qui limitent fortement le cadre dans lequel un modèle peut être étudié [12].

Par ailleurs, le GPGPU (*General-Purpose computing on Graphics Processing Units*) est aujourd'hui reconnu comme un moyen efficace d'accélérer les logiciels qui nécessitent beaucoup de puissance de calcul [3]. Cependant, dans le cadre des simulations multi-agents, l'utilisation de cette technologie apparait comme assez difficile car elle repose sur l'architecture massivement parallèle des GPU (*Graphics Processing Units*) qui s'accompagne d'un contexte de développement très spécifique. Ainsi, pour être efficiente, une implémentation utilisant le GPGPU exige que le modèle considéré soit au préalable adapté voire transformé. En effet, alors que la simulation multi-agent (et plus généralement les systèmes multi-agents) utilisent fréquemment des fonctionnalités de la programmation orientée objet, ces dernières ne sont plus disponibles dans un contexte de programmation GPU [2, 11]. De nombreux travaux se sont donc intéressés à la simplification du GPGPU dans le contexte des simulations multi-agents.

Parmi ces travaux, la plupart d'entre eux ont

choisi de le faire au travers d'une utilisation transparente de cette technologie. Cependant, cette approche nécessite d'abstraire un certain nombre de parties du modèle agent, ce qui limite le champ d'application des solutions proposées car elles reposent sur des éléments de modélisation prédéfinis (*e.g.* Flame GPU contient des primitives agents pré-programmés [15]).

Afin d'offrir une approche plus générique, nous avons étudié le principe de délégation GPU comme solution d'implémentation pour des simulations multi-agents voulant bénéficier du GPGPU [6]. Ce principe repose sur une utilisation directe du GPGPU qui se fait au travers d'une adaptation et transformation du modèle considéré. Prenant le parti de ne pas cacher la technologie sous-jacente, les différents cas d'études menés sur ce principe [10, 5, 6] ont confirmé les avantages conceptuels d'une telle approche ainsi que les bénéfices en terme de performance.

L'objectif de cet article est de proposer une généralisation du principe de délégation GPU sous la forme d'une méthodologie. Pour ce faire, nous allons dans un premier temps résumer tous les cas d'études basés sur le principe de délégation GPU pour ensuite s'intéresser à la définition de la méthodologie. Ensuite, nous proposons de l'expérimenter sur deux nouveaux modèles afin de souligner ses avantages et ses limites. La section 2 propose un retour d'expérience et une analyse du principe de délégation GPU. La section 3 introduit la méthodologie basée sur ce principe. La section 4 expérimente cette méthodologie sur deux nouveaux modèles. La section 5 conclut l'article et énonce les perspectives de recherche associées.

2 Le principe de délégation GPU

2.1 Énoncé du principe

Inspiré d'autres travaux liés au génie logiciel orienté agent, le principe de délégation GPU utilise l'environnement comme une abstraction de premier ordre [17] et arbore un point de vue environnement-centré. Ainsi, à l'instar d'autres approches telles que EASS (*Environment As Active Support for Simulation*) [1], IODA (*Interaction Oriented Design of Agent simulations*) [8] et l'approche des *artefacts* [14], la délégation GPU consiste à réifier une partie des calculs effectués dans le comportement des agents dans de nouvelles structures (*e.g.* dans l'environnement) dans le but de répartir la complexité du code et de modulariser son implémentation.

Proposé dans [10], le principe de délégation GPU est basé sur une approche hybride (CPU + GPU) et vise à répondre, en partie, aux difficultés occasionnées par le GPGPU [7]. En effet, les spécificités de la programmation GPU ainsi que l'architecture matérielle sur laquelle elle repose font qu'il est très difficile d'implémenter une simulation multi-agents sur GPU. Surtout si on considère comme essentiel des aspects tels que l'accessibilité, la généricité et la reutilisabilité. D'ailleurs, [13] conclut dans son étude qu'une recherche de performance pure ne peut se faire qu'au détriment de ces aspects dans un contexte GPGPU.

Avec pour objectif d'obtenir des gains de performances grâce au GPU mais aussi pour conserver l'accessibilité et la facilité de réutilisation d'une interface de programmation orientée objet, le principe de délégation GPU consiste en la réalisation d'une séparation explicite entre le comportement des agents, géré par le CPU, et les dynamiques environnementales traitées par le GPU. En effet, la spécificité du GPGPU fait que les comportements des agents sont difficiles à traduire en code GPU car ils comportent généralement de nombreuses structures conditionnelles qui ne sont pas adaptées à la programmation sur GPU. Au contraire, les calculs correspondant à l'application des dynamiques environnementales se prêtent beaucoup mieux à une parallélisation sur architecture parallèle car ils consistent généralement à appliquer de manière uniforme un calcul dans tout l'environnement.

Ainsi, l'idée sous-jacente est d'identifier dans le modèle agent des calculs qui peuvent être transformés en dynamiques environnementales afin qu'ils puissent bénéficier de la puissance du GPU. Le principe de délégation GPU peut être énoncé de la manière suivante : "*Tout calcul de perception agent qui n'implique pas l'état de l'agent peut être transformé en une dynamique endogène de l'environnement, et ainsi considéré pour une implémentation dans un module GPU indépendant*".

2.2 Retour d'expérience

Le principe de délégation GPU a été appliqué pour la première fois sur un modèle d'émergence multi-niveaux (MLE) [1][10]. Cette expérimentation a rempli deux objectifs : (1) garder l'accessibilité du modèle agent dans un contexte

GPU (le modèle originel n'a pas été modifié) et (2) augmenter la taille de l'environnement ainsi que le nombre d'agents. Cette étude a aussi montré que la délégation GPU permet d'aborder la question de la généricité en favorisant la réutilisation des modules GPU créés, car ils peuvent être réutilisés dans d'autres contextes.

Suite à ces premiers résultats, la délégation GPU a été appliquée sur le modèle des *boids* de Reynolds [1] [5] dans le but de mettre à l'épreuve la faisabilité et la généricité de l'approche. Ce cas d'étude a montré que la délégation GPU permet de prendre en compte des modèles très différents et a occasionné une évolution du critère de sélection du principe. En effet, lors de cette expérimentation, nous avons vu qu'il est aussi possible d'identifier des calculs de perceptions ne *modifiant* pas les états des agents. Ces derniers peuvent être transformés en dynamiques environnementales et donc considérés pour une implémentation en module GPU.

Pour continuer d'expérimenter ce principe, il a ensuite été appliqué sur quatre autres modèles (Game of Life, Schelling's segregation, Fire et DLA) [1] [6] . Cette expérimentation a souligné une nouvelle fois la capacité de réutilisation des modules GPU produits par le principe de délégation GPU : il a été possible de réutiliser des modules GPU issus des précédents cas d'études. De plus, grâce à la modularité apportée par l'approche hybride utilisée, les modules GPU créés reposent sur des kernels [2] très simples (comportant seulement quelques lignes de code dans la plupart des cas) améliorant de ce fait l'accessibilité du GPGPU.

2.3 Bilan actuel

Toutes les applications du principe de délégation GPU ont mené à des diminutions du temps total d'exécution et cela quelque-soit le modèle considéré. De plus, au-delà des gains de performance obtenus, ces expérimentations ont confirmé les avantages de l'approche par rapport à des aspects souvent négligés dans le cadre de l'utilisation du GPGPU pour les simulations multi-agents.

Ainsi, au vu de ces différentes preuves de concept, le principe de délégation a rempli ses objectifs initiaux tant d'un point de vue des performances que d'un point de vue conceptuel en améliorant l'accessibilité, la généricité et la réutilisabilité des outils créés. Pourtant, elles font également apparaître quelques autres questions, dont nous rediscuterons, telles que la difficulté de mise en œuvre pour un utilisateur novice en GPGPU ou bien sur quels types de modèles la délégation GPU pourrait être appliquée.

2.4 Vers une généralisation de l'approche

Sur ces différents cas d'études, le principe de délégation GPU a été appliqué de manière *ad hoc* : l'objectif était de réaliser des preuves de concepts. Cependant, durant ces différentes expérimentations, nous avons observé une récurrence dans le processus d'implémentation. L'objectif de cet article est donc de généraliser ce processus d'adaptation afin de proposer une méthodologie basée sur la délégation GPU pour le développement de simulations multi-agents.

En plus d'améliorer le champ d'application du principe, cette méthodologie a pour objectifs de : (1) simplifier l'utilisation du GPGPU dans le contexte des simulations multi-agents en décrivant le processus de modélisation et d'implémentation à suivre, (2) définir une approche générique pouvant être appliquée sur une grande variété de modèles, (3) promouvoir la réutilisabilité des outils créés ou des modèles transformés (par exemple en réutilisant les modules GPU déjà développés) et (4) aider les utilisateurs potentiels à décider s'ils peuvent bénéficier du GPGPU compte tenu de leurs modèles.

3 Définition de la méthodologie

De la généralisation du processus d'application et d'implémentation du principe de délégation GPU, nous avons extrait une méthodologie divisée en 5 phases distinctes (voir figure 1) :

1. La première étape consiste en la décomposition de tous les calculs qui sont utilisés dans le modèle.

2. La deuxième étape consiste à identifier, parmi les calculs précédemment listés, ceux répondant aux critères du principe de délégation GPU.

3. La troisième étape consiste à vérifier si les calculs identifiés comme compatibles avec le principe de délégation GPU ont déjà été transformés en dynamiques environnementales et donc si il existe un module GPU dédié pouvant être réutilisé.

1. `http://www.lirmm.fr/~hermellin/Website/simu.html`
2. Un *kernel* est un noyau de calcul exécuté par le GPU.

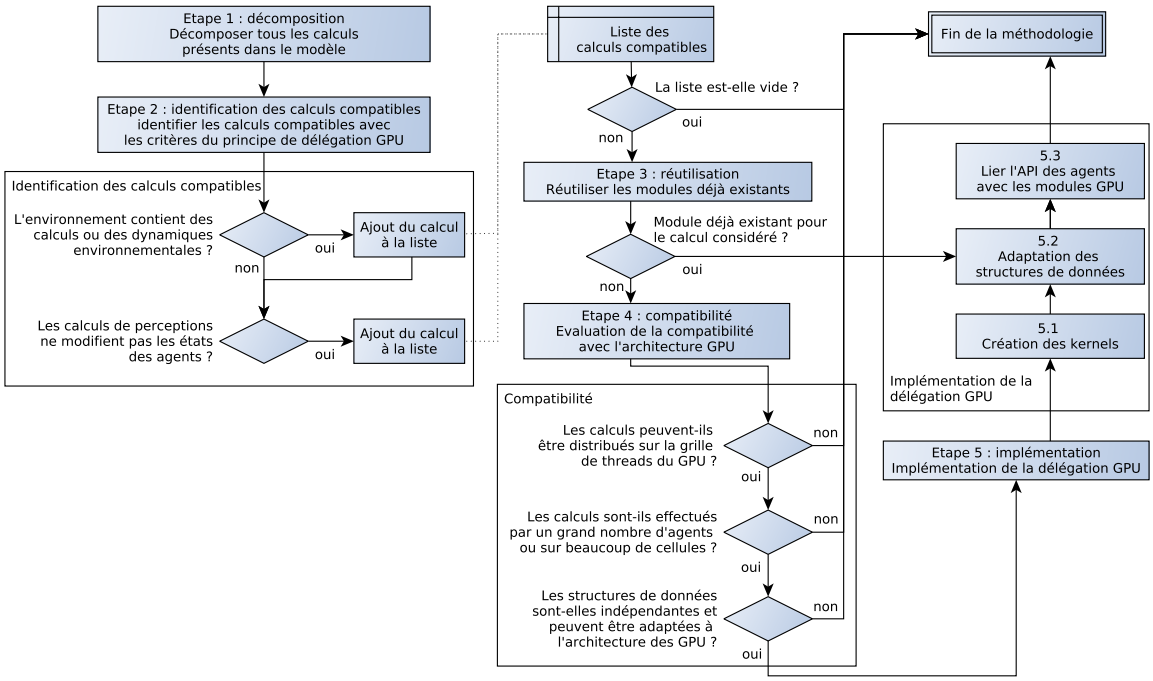

FIGURE 1 – Diagramme de la méthodologie

4. La quatrième étape consiste à établir la compatibilité des calculs sélectionnés avec l'architecture d'un GPU. L'idée étant de choisir et d'appliquer le principe de délégation uniquement sur les calculs qui vont apporter le plus de gain une fois traduits en modules GPU.

5. La cinquième étape consiste à implémenter concrètement le principe de délégation GPU sur les calculs respectant l'ensemble des contraintes précédentes.

3.1 Étape 1 : décomposition des calculs

Cette phase consiste à décomposer tous les calculs qui sont utilisés par le modèle. L'intérêt d'une telle décomposition réside dans le fait qu'un certain nombre de calculs présents dans le modèle ne sont pas explicites. Expliciter les calculs nécessaires à la réalisation des différents comportements des agents, en les décomposant en le plus de primitives possibles, va permettre d'augmenter l'efficacité de l'application du principe de délégation GPU sur le modèle considéré. En effet, l'intérêt d'une telle décomposition permet de ne pas avoir un seul "gros" *kernel* contenant tous les calculs GPU, mais de capitaliser sur l'aspect modulaire et hybride du principe de délégation GPU avec plusieurs *kernels* très simples mais qui tirent partis de la décomposition des calculs : chaque calcul (compatible) peut alors prétendre à avoir son propre *kernel*. Ainsi, la délégation GPU sera d'autant plus efficace qu'il va être possible de décomposer le modèle en calculs "élémentaires". D'ailleurs, dans le contexte de l'utilisation du GPGPU pour les simulations multi-agents, cette décomposition des actions a également été identifiée comme cruciale dans [4]. En introduisant une nouvelle division des actions des agents qui limite les accès concurrents aux données, [4] montre qu'il a été possible d'augmenter significativement les performances globales des modèles qui utilisent le GPGPU.

3.2 Étape 2 : identification des calculs compatibles

L'identification des calculs compatibles est une étape essentielle car elle consiste à vérifier quels sont les calculs qui répondent aux critères du principe de délégation GPU et qui pourront ainsi bénéficier du GPGPU. Les calculs sont identifiés comme compatibles si :

- *Pour l'environnement* : si l'environnement n'est pas statique et qu'il contient des dynamiques, ces dernières doivent (1) ne pas contenir de processus décisionnels, (2) s'appliquer sur tout l'environnement et (3) avoir un impact global [3]. Dans ce cas, leur compatibilité est éta-

3. L'impact des dynamiques est un paramètre important. Prenons l'exemple d'une dynamique environnementale qui consiste à faire appa-

blie et leur traduction en modules GPU est possible et pertinente.

- *Pour les agents* : si des calculs de perceptions ne modifient pas les états des agents, ils peuvent être considérés comme compatibles et transformés en dynamiques environnementales pour être implémentés dans des modules GPU indépendants. L'idée est de transformer un calcul de perception réalisé de manière locale en une dynamique environnementale s'appliquant globalement dans l'environnement.

Si aucune partie du modèle n'est conforme aux critères d'application du principe de délégation GPU, il est inutile de continuer à suivre cette méthodologie car, dans un tel cas, les gains apportés par le GPGPU pourraient être insignifiants voire même négatifs (voir [9]).

3.3 Étape 3 : réutilisation des modules GPU

Un des objectifs de la méthodologie est de promouvoir la réutilisabilité des modules GPU créés. En effet, comme vu précédemment, le principe de délégation GPU sur lequel se base la méthodologie, se distingue par la création de modules GPU génériques pouvant être réutilisés dans des contextes différents de ceux pour lesquels ils ont été créés. Ainsi, il convient de vérifier que les calculs sélectionnés n'aient pas déjà un module GPU dédié. Si c'est le cas, il est possible de passer directement à l'étape 5 afin d'adapter les structures de données au module existant.

3.4 Étape 4 : évaluation de l'adaptation des calculs sur l'architecture des GPU

Avant d'appliquer concrètement la délégation GPU sur les calculs identifiés comme compatibles, il est nécessaire d'évaluer si ces calculs vont pouvoir s'adapter à l'architecture massivement parallèle des GPU. Ainsi, la compatibilité d'un calcul avec les critères du principe de délégation GPU n'implique pas forcement une amélioration des performances une fois ce principe appliqué. Il est donc nécessaire d'évaluer empiriquement la pertinence d'appliquer la délégation GPU sur les calculs identifiés.

Cette phase de vérification peut être réalisée en répondant à trois questions :

raitre de manière stochastique dans l'environnement, à chaque x pas de temps, une source de nourriture à une position donnée. Cette dynamique va bien s'appliquer sur tout l'environnement mais ne va avoir qu'un impact très localisé. Dans ce cas, traduire cette dynamique dans un module GPU n'est pas justifié car les gains espérés seront nuls (voir négatifs).

Est-ce que les calculs identifiés vont pouvoir être distribués sur le GPU ?
En effet, les calculs doivent être indépendants et simples et ne pas contenir de trop nombreuses structures conditionnelles, ces dernières pouvant causer des problèmes de divergence des *threads* ou des ralentissements lors de l'exécution. Des calculs contenant des boucles itératives s'adaptent mieux aux architectures parallèles [16].

Est-ce que les calculs identifiés sont réalisés de manière globale (par un grand nombre d'agents ou appliqués sur un grand nombre de cellules de l'environnement) ?
En raison des coûts très élevés (en temps) qu'occasionnent les transferts de données entre GPU et CPU, des calculs rarement utilisés vont accroitre le temps de calcul général du modèle.

Est-ce que les structures de données associées aux calculs identifiés vont pouvoir s'adapter aux contraintes d'une utilisation sur GPU ?
En l'occurrence, il s'agit principalement de s'assurer que les données peuvent être distribuées sur le GPU (en tenant compte des spécificités des différents types de mémoire) et traitées avec un degré élevé d'indépendance afin d'avoir une implémentation GPU efficace ([2] donne plus d'informations sur cet aspect).

Pour donner un exemple, en se basant sur nos différents cas d'étude, il est possible d'utiliser, dans le cas d'environnements discrétisés, des structures de données (des tableaux par exemple) de la taille de l'environnement ce qui les rend très adaptées à l'architecture du GPU. En effet, chaque cellule de l'environnement aura ainsi un *thread* qui lui sera entièrement dédié. La figure 2 illustre comment ces structures de données peuvent être utilisées lors d'une application du principe de délégation GPU sur un modèle possédant un environnement discrétisé.

3.5 Étape 5 : implémentation

Les calculs étant identifiés, sélectionnés et évalués, il convient d'appliquer sur chacun d'entre eux la délégation GPU. Cette application est divisée en trois étapes :

1. Création des modules GPU ;
2. Adaptation des structures de données ;
3. Création des interfaces entre la partie CPU du modèle et les modules GPU.

L'application de la délégation GPU commence par la création des modules GPU contenant les

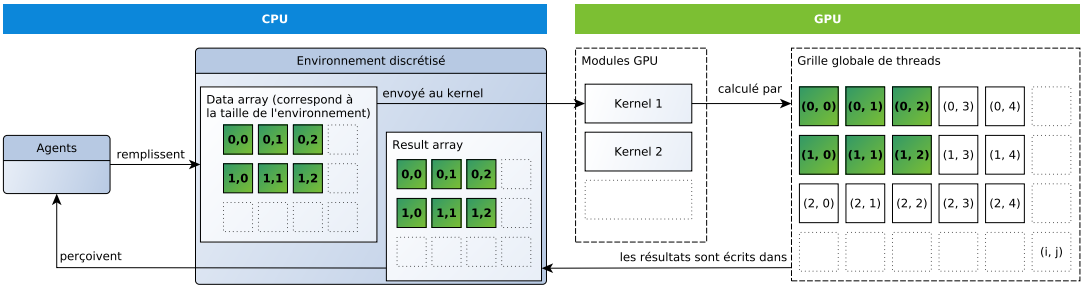

FIGURE 2 – Structuration et utilisation des données pour un environnement discrétisé

kernels de calculs (la traduction des calculs séquentiels en version parallèle). Grâce à la décomposition effectuée précédemment, ces *kernels* ne nécessitent que peu de connaissances en GPGPU et tiennent en seulement quelques lignes de code. En général, ces modules sont divisés en quatre parties : (1) initialisation du *thread* qui va servir au calcul, (2) test conditionnel sur la position du *thread* dans la grille globale du GPU pour l'accès aux données correspondantes, (3) réalisation du calcul et (4) écriture du résultat dans le tableau correspondant (*e.g.* voir [6]).

Ensuite, les structures de données doivent être adaptées aux modules GPU créés. Cette adaptation est basée sur la nature des calculs effectués et sur le type d'environnement utilisé (des tableaux correspondant à la discrétisation de l'environnement sont le plus souvent utilisés).

Enfin, ces nouveaux éléments doivent être intégrés et reliés à la partie CPU du modèle. Pour ce faire, de nouvelles versions des fonctions de perceptions qui encapsulent l'utilisation du GPU doivent être créées pour permettre aux agents et à l'environnement de recueillir et d'utiliser les données calculées par les modules GPU. La plupart du temps, ces fonctions sont de simples primitives d'accès aux données.

4 Expérimentation

Dans cette section, nous expérimentons la méthodologie proposée sur les modèles Heatbugs et Proie/Prédateur. L'application de la méthodologie sur le modèle Heatbugs est détaillée étape par étape avec, à la fin, des résultats de performance. Quant à l'application sur le modèle Proie/Prédateur, elle se concentre principalement sur certains aspects de la méthodologie qui nécessitent des améliorations. Nous commençons par présenter quelques notions de base relatives à la programmation GPU.

4.1 Notions relatives au GPGPU

Pour comprendre le principe de programmation associé au GPGPU, il faut avoir à l'esprit qu'il est fortement lié à l'architecture matérielle massivement multicœurs des GPU. Pour utiliser les capacités GPGPU de notre carte graphique, nous utilisons CUDA [4] (*Compute Unified Device Architecture*, l'interface de programmation dédiée fournie par Nvidia) et JCUDA [5], une librairie permettant d'utiliser CUDA au travers de Java. La philosophie de fonctionnement de cette interface de développement est la suivante : le CPU (*host*) gère la répartition des données et exécute les *kernels* : des fonctions spécialement créées pour s'exécuter sur le GPU (*device*). Le GPU est capable d'exécuter un *kernel* de manière parallèle grâce aux *threads*. Ces *threads* sont regroupés par *blocs* (les paramètres *blockDim.x*, *blockDim.y* définissent la taille de ces blocs), qui sont eux-mêmes rassemblés dans une *grille globale*. Chaque *thread* au sein de cette structure est identifié par des coordonnées uniques 3D (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*) lui permettant d'être localisé. De la même façon, un *bloc* peut être identifié par ses coordonnées dans la *grille* (respectivement *blockIdx.x*, *blockIdx.y*, *blockIdx.z*). Les *threads* exécutent le même *kernel* mais traitent des données différentes selon leur localisation spatiale. Dans la suite de ce document, les identifiants des *threads* dans la *grille globale* du GPU seront notés *i* et *j* (environnements 2D).

4.2 Le modèle Heatbugs

Dans le modèle Heatbugs, les agents tentent de maintenir une température optimale autour d'eux. Plus précisément, les agents se déplacent dans un environnement 2D discrétisé en cellules. Un agent ne peut pas occuper une cellule

4. *e.g.* https ://developer.nvidia.com/what-cuda
5. *e.g.* http ://www.jcuda.org

déjà occupée par un autre agent. Un agent émet de la chaleur qui se diffuse dans l'environnement. Enfin, chaque agent possède une température idéale qu'il cherche à atteindre. Ainsi, plus la température de la case sur laquelle il se trouve va être différente de sa température idéale et plus l'agent va être mécontent. Lorsque un agent n'est pas satisfait (sa case est trop froide ou trop chaude), il se déplace aléatoirement pour trouver une meilleur place. La figure 3 résume ces comportements.

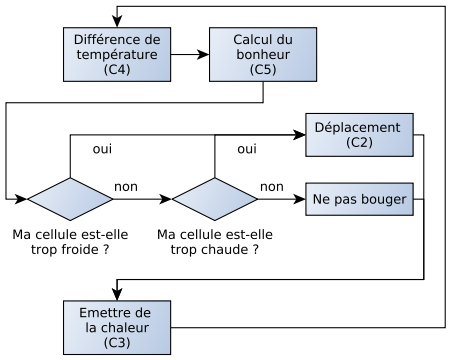

FIGURE 3 – Comportement des agents dans le modèle Heatbugs

Application de la méthodologie.
Étape 1 : décomposition.
- Dans l'environnement : diffusion de la chaleur (C1).
- Pour les agents : déplacement (C2), émission de chaleur (C3), calcul des différences de température (C4) et calcul du bonheur (C5).

Étape 2 : identification.
C1 est une dynamique globale de l'environnement, elle est donc compatible et peut être implémentée dans un module GPU. C4 consiste à percevoir la température de la case sur laquelle se trouve l'agent et calculer la différence entre cette température et la température idéale de l'agent. Ce calcul ne modifie pas les états de l'agent, il est donc compatible et peut être transformé en une dynamique environnementale. C2, C3 et C5 modifient les états des agents, nous ne les considérerons donc pas pour la suite.

Étape 3 : réutilisation.
C1 est un calcul de diffusion et a déjà été effectué dans le modèle *Fire* [6], nous réutilisons donc le module GPU correspondant.

Étape 4 : compatibilité.
Au contraire de C1, C4 est réalisé pour la première fois, nous devons donc effectuer cette phase de vérification. Ainsi, C4 consiste en la réalisation d'une différence, il peut être facilement distribué sur la grille de *threads* du GPU. De plus, tous les agents calculent cette différence à chaque pas de simulation. Transformer ce calcul est donc pertinent car réalisé de très nombreuses fois. Enfin, les données utilisées pour ce calcul peuvent être stockées dans des tableaux à deux dimensions s'adaptant parfaitement aux contraintes des architectures GPU.

Étape 5 : implémentation.
Pour C1, nous utilisons le tableau 2D déjà défini contenant la chaleur pour chaque cellule. Il est ensuite envoyé au GPU qui calcule simultanément la diffusion de la chaleur sur tout l'environnement. Plus précisément, pour chaque cellule, un *thread* calcule la somme de la chaleur des cellules voisines qui est ensuite modulée par une variable de diffusion. L'algorithme 1 présente l'implémentation du *kernel* de calcul correspondant.

Algorithme 1 : *Kernel* : diffusion de la chaleur

entrées : $width, height, heatArray, radius$
sortie : $resultArray$ (la quantité de chaleur)
1 $i = blockIdx.x * blockDim.x + threadIdx.x$;
2 $j = blockIdx.y * blockDim.y + threadIdx.y$;
3 $sumOfHeat = 0$;
4 **si** $i < width$ ***et*** $j < height$ **alors**
5 | $sumOfHeat =$ getNeighborsHeat($heatArray[i, j], radius$);
6 **fin**
7 $resultArray[i, j] = sumOfHeat * heatAdjustment$;

Après l'exécution de ce *kernel*, la chaleur de chaque cellule est utilisée pour calculer la différence entre cette valeur et la température idéale de l'agent qui se trouve sur la cellule. Pour ce faire, chaque agent dépose, en fonction de sa position, sa température idéale dans un tableau 2D (correspondant à la taille de l'environnement). Les deux tableaux sont envoyés au GPU qui effectue le calcul de la différence. L'algorithme 2 présente l'implémentation du calcul dans un *kernel*. Les agents utilisent ces résultats pour calculer leur valeur de bonheur, donc à partir d'une perception qui est précalculée par un *kernel* GPU.

Algorithme 2 : *Kernel* : calcul de différence

entrées : $width, height, heatArray, idealTemperatureArray$
sortie : $resultArray$ (la différence de température)
1 $i = blockIdx.x * blockDim.x + threadIdx.x$;
2 $j = blockIdx.y * blockDim.y + threadIdx.y$;
3 $diff = 0$;
4 **si** $i < width$ ***et*** $j < height$ **alors**
5 | $diff = heatArray[i, j] - idealTemperatureArray[i, j]$;
6 **fin**
7 $resultArray[i, j] = diff$;

Performance du modèle.
Pour évaluer les performances, nous comparons les exécutions CPU puis hybride du modèle [6]. Pour chacune des versions, nous simulons plusieurs tailles d'environnement et avons fixé la densité des agents à 40%. La figure 4 présente les résultats obtenus.

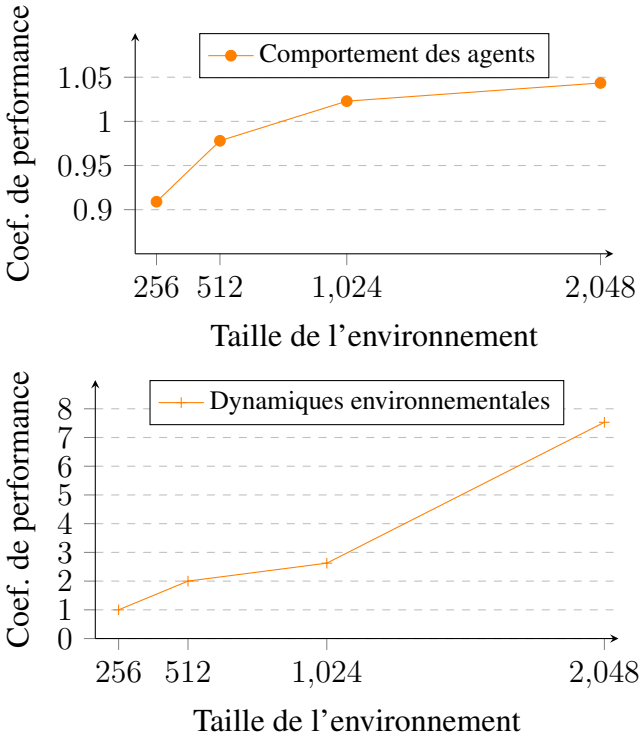

FIGURE 4 – Coefficient d'accélération entre une exécution CPU et hybride du modèle Heatbugs

De ces graphiques, nous notons que le gain obtenu pour le calcul de l'environnement est d'autant plus important que l'environnement est grand. Cependant, le gain correspondant au calcul des agents reste faible (environ 5%). Nous expliquons ces résultats de la façon suivante : la dynamique de diffusion (C1) est appliquée à toutes les cellules alors que seule une petite partie des calculs effectués par les agents a été transformée (C4).

4.3 Le modèle Proie/Prédateur

Dans notre modèle Proie/Prédateur, les agents évoluent dans un environnement 2D discrétisé en cellules. Les prédateurs, tout comme les proies, sont placés aléatoirement. Tous les prédateurs possèdent un champ de vision (*Field Of View*, FOV) de dix cellules autour d'eux. Si une proie se trouve dans ce FOV, ils la ciblent et se dirigent vers elle, sinon ils se déplacent aléatoirement. Les proies se déplacent aussi de manière aléatoire. Lorsqu'un prédateur se trouve dans le FOV d'une proie (qui est de 6), cette dernière tente de s'échapper en s'enfuyant dans la direction opposée. Une proie meurt quand elle est ciblée par un prédateur et quand ce dernier se trouve sur la même case qu'elle. La figure 5 résume ces comportements.

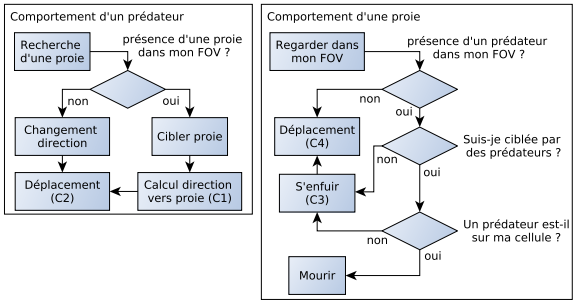

FIGURE 5 – Comportement des agents dans le modèle Proie/Prédateur

Application de la méthodologie.
Étape 1 : décomposition.
- L'environnement est statique et ne possède aucune dynamique environnementale ;
- Agents : les prédateurs (C1) déterminent la direction vers la proie la plus proche et (C2) se déplacent. Les proies (C3) calculent leur direction de fuite et (C4) se déplacent.

Étape 2 : identification.
C1 et C3 consistent à calculer la direction de fuite des proies face aux prédateurs (C1) et la direction vers la proie ciblée pour les prédateurs (C3). Ces deux calculs peuvent être pensés comme une perception pré-calculée par une dynamique environnementale. Dans ce cas, cette dynamique calculera pour chaque cellule de l'environnement les directions vers les proies et les prédateurs les plus proches mais aussi les directions opposées. La transformation de ces calculs en dynamiques environnementales traitées par un module GPU est possible car ces modules ne modifient pas les états des agents.

Étape 3 : réutilisation.
Concernant C1 et C3, il est possible de réutiliser le module *GPU field perception* créé pour le modèle MLE [10]. En effet, ce module calcule pour chaque cellule la direction vers des cellules voisines possédant la plus grande / faible quantité d'une variable donnée. Ce calcul est

6. Notre configuration de test est composée d'un processeur Intel Core i7 (génération Haswell, 3.40GHz) et d'une carte graphique Nvidia Quadro K4000 (768 cœurs).

similaire à C1 et C3 qui calculent les directions maximum (direction d'interception) et minimum (direction de fuite) vers des agents : la variable donnée est donc ici l'orientation des agents.

Étape 4 : compatibilité.
Vu que nous réutilisons un module déjà existant et qu'aucun nouveau calcul n'a été identifié comme compatible, nous pouvons directement passer à l'étape 5.

Étape 5 : implémentation.
Pour implémenter le module dédié à C3, deux tableaux 2D sont utilisés : chacun dédié a un type d'agent et contenant une marque de présence (plus il y a d'agents et plus la marque est forte. Ils sont envoyés au GPU qui calcule respectivement la direction vers le prédateur le plus proche et la direction opposée pour chaque cellule de l'environnement. Les agents utilisent ces résultats pour déterminer leur orientation et se déplacer dans la bonne direction.

Performance du modèle.
Avec ce modèle, et parce qu'il n'y a pas de dynamique environnementale, il est difficile de dire à l'avance quels vont être les gains de performance envisageables. En effet, les performances du modèle vont fortement dépendre de la configuration matérielle utilisée ainsi que des caractéristiques du modèle considéré, comme le nombre d'agents par exemple. Si ce nombre est faible, le gain peut être négatif. En fait, nous sommes ici confrontés au problème de l'évaluation des performances : lorsqu'un modèle ne présente aucune dynamique environnementale et que seuls des calculs de perceptions peuvent être transformés, nous ne pouvons prévoir à l'avance si le modèle va bénéficier du GPGPU en terme de performances. Nous rediscutons de cet aspect dans la section suivante.

Cependant, la figure 6 montre que pour ce modèle les résultats obtenus sont très bons avec une exécution jusqu'à 4 fois plus rapide comparée à la version séquentielle du modèle.

5 Conclusion et perspectives

Depuis son apparition, le GPGPU est considéré comme une technologie innovante profitant à de nombreux domaines où les temps de calcul peuvent être critiques (*e.g.* finance, physique, biologie, etc.). Cependant, dans le domaine des simulations multi-agents (et plus généralement des systèmes multi-agents), nous remarquons

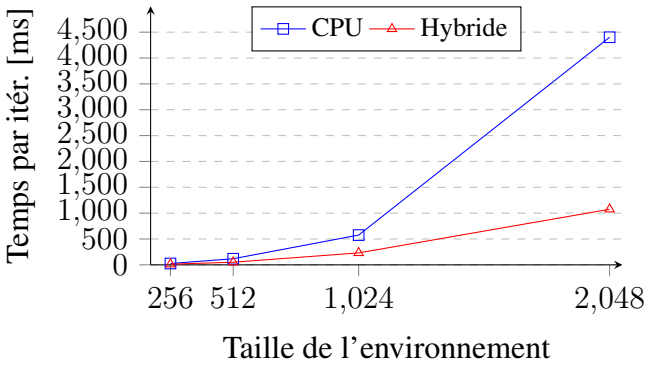

FIGURE 6 – Modèle Proie-prédateur, résultats de performance (densité des agents : 20 %).

que cette technologie peine à s'imposer et cela malgré les besoins en puissance de calcul toujours plus importants. Des études ont montré que le manque d'accessibilité, de réutilisabilité et de généricité pouvaient expliquer ce manque d'intérêt de la part de notre communauté [13].

Ainsi, depuis plusieurs années, nous présentons des travaux pour répondre à ces différents problèmes. Présenté au départ sous la forme d'un principe de conception (le principe de délégation GPU [10, 5]), nous avons proposé dans cet article une méthodologie utilisant ce principe pour le développement de simulations multi-agents. Extraite de diverses expérimentations [6], l'objectif de cette méthodologie est de promouvoir un cadre à l'utilisation de la programmation GPU dans le contexte des simulations multi-agents tout en (1) ne cachant pas la technologie utilisée à l'utilisateur et (2) mettant en avant un processus de modélisation itératif modulaire prônant la réutilisabilité des outils créés.

Sur la base des deux expérimentations présentées dans cet article, nous avons vu qu'un des principaux avantages de notre méthodologie est l'accessibilité. En effet, il a été très facile, en suivant la méthodologie, d'identifier des parties des différents modèles qui ont pu être transformées pour bénéficier du GPGPU. De plus, cette méthodologie produit des modules génériques réutilisables (cf. section 4.3).

Un autre avantage de la méthodologie est sa polyvalence dans le sens où elle peut être appliquée sur une grande variété de modèles, comme le montre les différentes expérimentations menées jusqu'ici : MLE [10], les *boids* de Reynolds [5], Game Of Life et Schelling's Segregation [6], Heatbugs, Proie/Prédateur, etc.

Cependant, il nous faut modérer ce dernier avan-

tage. S'il est vrai que nous avons appliqué ce principe sur une grande variété de comportements, tous nos cas d'études ont porté sur des modèles possédant un environnement discrétisé pour lequel le principe de délégation GPU est facilement applicable (en terme d'implémentation). Ainsi, un de nos objectifs futurs sera de considérer d'autres types de modèle sur lesquels appliquer notre méthodologie (*e.g.* possédant des environnements continus) afin de renforcer son champ d'application.

Autre limite actuelle, il est pour l'instant difficile d'évaluer précisément les gains de performance avant l'application de la méthodologie. En effet, même si un modèle valide la deuxième étape (éligibilité des calculs), les gains de performance peuvent ne pas être intéressants car ces derniers dépendent fortement de la configuration matérielle utilisée ainsi que des caractéristiques du modèle considéré.

Une solution pourrait être de proposer un *benchmark* qui permettrait de définir l'intérêt d'appliquer notre méthodologie sur le modèle considéré en fonction du nombre d'agents et de la configuration matérielle utilisée. Ainsi, une des perspectives de recherche serait de développer cette solution logicielle qui, une fois exécutée, donnerait ainsi une idée du seuil au-dessus duquel une implémentation GPU pourrait être efficiente. L'idée étant de proposer un outil implémentant un ensemble de fonctions agents récurrentes réutilisant les modules GPU déjà créés grâce à notre méthodologie.

Références

[1] Fabien Badeig and Flavien Balbo. Définition d'un cadre de conception et d'exécution pour la sim. multi-agent. *Revue d'Intelligence Artificielle*, 26(3) :255–280, 2012.

[2] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4) :583–600, 2014.

[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.

[4] Simon Coakley, Paul Richmond, Marian Gheorghe, Shawn Chin, David Worth, Mike Holcombe, and Chris Greenough. *Intelligent Agents in Data-intensive Computing*, chapter Large-Scale Simulations with FLAME, pages 123–142. Springer International Publishing, Cham, 2016.

[5] Emmanuel Hermellin and Fabien Michel. Délégation GPU des perceptions agents : Application aux boids de reynolds. In Laurent Vercouter and Gauthier Picard, editors, *Environnements sociotechniques - JFSMA 15 - Vingt-troisièmes Journées Francophones sur les Systèmes Multi-Agents, Rennes, France*, pages 185–194. Cépaduès Éditions, 2015.

[6] Emmanuel Hermellin and Fabien Michel. GPU Delegation : Toward a Generic Approach for Developping MABS Using GPU Programming. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 1249–1258, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[7] Emmanuel Hermellin, Fabien Michel, and Jacques Ferber. État de l'art sur les sim. multi-agents et le GPGPU. *Revue d'Intelligence Artificielle*, 29(3-4) :425–451, 2015.

[8] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Ioda : an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3) :303–343, 2011.

[9] Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, and Laurent Philippe. Using GPU for Multi-agent Multi-scale Simulations. In *Distrib. Computing and Artificial Intelligence*, volume 151 of *Advances in Intelligent and Soft Computing*, pages 197–204. Springer Berlin Heidelberg, 2012.

[10] Fabien Michel. Délégation GPU des perceptions agents : intégration itérative et modulaire du GPGPU dans les simulations multi-agents. Application sur la plate-forme TurtleKit 3. *Revue d'Intelligence Artificielle*, 28(4) :485–510, 2014.

[11] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens KrÃ$\frac{1}{4}$ger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[12] Hazel R. Parry and Mike Bithell. Large scale abm : A review and guidelines for model scaling. In Alison J. Heppenstall, Andrew T. Crooks, Linda M. See, and Michael Batty, editors, *Agent-Based Models of Geographical Systems*, pages 271–308. Springer Netherlands, 2012.

[13] Kalyan S. Perumalla and Brandon G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123, 2008.

[14] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in MAS : an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2) :158–192, 2011.

[15] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela M. Romano. High performance cellular level agent-based simu. with FLAME for the GPU. *Briefings in bioinformatics*, 11(3) :334–47, 2010.

[16] Jason Sanders and Edward Kandrot. *CUDA par l'exemple*. Pearson, 2011.

[17] Danny Weyns and Fabien Michel. *Agent Environments for Multi-Agent Systems IV, 4th International Workshop, E4MAS 2014 - 10 Years Later, Paris, France, May 6, 2014, Revised Selected and Invited Papers*, volume 9068 of *LNCS*. Springer, 2015.