Délégation GPU des perceptions agents : application aux boids de Reynolds

Emmanuel Hermellin, Fabien Michel

Délégation GPU des perceptions agents : application aux boids de Reynolds

Emmanuel Hermellin[a]
emmanuel.hermellin@lirmm.fr

Fabien Michel[a]
fmichel@lirmm.fr

[a]LIRMM - Université de Montpellier - CNRS
161 rue Ada, 34090 Montpellier, France

Résumé

L'utilisation du GPGPU (General-Purpose Computing on Graphics Processing Units) pour la simulation multiagent permet d'améliorer les performances des modèles et lève une partie des contraintes liées au passage à l'échelle. Cependant, adapter un modèle pour qu'il utilise le GPU est une tâche complexe car le GPGPU repose sur une programmation extrêmement spécifique et contraignante. C'est dans ce contexte que la délégation GPU des perceptions agents a été proposée. Ce principe consiste à réaliser une séparation claire entre les comportements de l'agent (gérés par le CPU) et les dynamiques environnementales (manipulées par le GPU) dans le but de faciliter l'implémentation de SMA sur GPU. Il a été appliqué sur un cas d'étude et a montré de bons résultats en termes de performances et de conception. Dans cet article, nous proposons de tester la généricité de cette approche en appliquant le principe de délégation GPU sur un modèle agent très classique : les boids de Reynolds. Nous montrons que le principe de délégation offre des résultats intéressants au niveau des performances mais aussi d'un point de vue conceptuel.

Mots-clés : *SMA, GPGPU, Flocking, CUDA*

Abstract

General-Purpose Computing on Graphics Processing Units (GPGPU) allows to extend the scalability and performances of Multi-Agent Based Simulations (MABS). However, GPGPU requires the underlying program to be compliant with the specific architecture of GPU devices, which is very constraining. In this context, the GPU Environmental Delegation of Agent Perceptions principle has been proposed to ease the use of GPGPU for MABS. This principle consists in making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU. For now, this principle has shown good results, but only on one single case study. In this paper, we further trial this principle by testing its feasibility and genericness on a classic ABM, namely Reynolds's boids. To this end, we first review existing boids implementations and propose our own benchmark model. The paper then shows that applying GPU delegation not only speeds up boids simulations but also produces an ABM which is easy to understand, thanks to a clear separation of concerns.

Keywords: *GPGPU, MABS, Flocking, CUDA*

1 Introduction

La *délégation GPU des perceptions agents* (que nous nommerons *délégation GPU*) fait partie des approches visant à faciliter l'utilisation du GPGPU (*General-Purpose Computing on Graphics Processing Units*) dans le domaine des simulations multiagents. Réelle révolution technologique, le GPGPU utilise l'architecture massivement parallèle des cartes graphiques pour effectuer du calcul généraliste. Il est ainsi possible d'utiliser des ordinateurs classiques pour accélérer les simulations multiagents [6]. Cependant, de par l'architecture matérielle très spécifique des GPU (*Graphics Processing Unit*), ce type de programmation requiert des connaissances avancées qui limitent son utilisation [9].

La *délégation GPU* est basée sur une approche hybride qui repose sur une utilisation conjointe du CPU (*Central Processing Unit*) et du GPU. En offrant la possibilité de sélectionner ce qui va être adapté puis exécuté par le GPU, les approches hybrides offrent une solution attractive pour contourner les difficultés occasionnées par le GPGPU [4]. La *délégation GPU* consiste à déplacer dans l'environnement certains calculs effectués par les agents au sein de leur propre comportement. Un cas d'étude a été proposé dans [7] et montre de bons résultats en terme de performances, d'accessibilité et de réutilisabilité.

Dans le but de tester la généricité et les avan-

tages que peut apporter ce principe, nous appliquons, dans cet article, la *délégation GPU* sur un modèle classique de SMA : les *boids* de Reynolds [13].

La section 2 présente le modèle de Reynolds et son implémentation dans différentes plates-formes SMA. Nous introduisons notre propre modèle en section 3. La section 4 se focalise sur le principe de *délégation GPU des perceptions agents*. La section 5 décrit l'application de la *délégation GPU* sur notre modèle de *flocking* ainsi que son implémentation. Les résultats sont présentés et discutés en section 6. Enfin, la section 7 conclut l'article et présente des perspectives de recherche qui lui sont associées.

2 Les boids de Reynolds

2.1 Présentation du modèle original

Reynolds a remarqué qu'il n'était pas possible d'utiliser des scripts pour réaliser des animations réalistes de nuées d'oiseaux artificiels (*boids*) [13]. Son idée a donc été la suivante : les *boids* doivent être influencés par les autres pour se déplacer d'une manière cohérente et crédible. La citation issue de [13] résume cette approche : "*Boid behavior is dependant not only on internal state but also on external state*".

Chaque agent va ainsi suivre trois règles comportementales :

- **R.1** *Collision Avoidance* : éviter les collisions entre entités ;
- **R.2** *Flock Centering* : rester le plus proche possible des autres entités ;
- **R.3** *Velocity matching* : adapter sa vitesse à celles des autres entités.

Le modèle de *flocking* de Reynolds est l'une des simulations multiagents la plus représentative et connue. Ainsi, de nombreuses plates-formes spécialisées dans le développement de SMA l'intègrent en proposant leur propre implémentation.

2.2 Les boids dans les plates-formes SMA

Dans cette section, nous comparons plusieurs implémentations du modèle de Reynolds. Parmi tous les travaux trouvés, nous sélectionnons uniquement les modèles pouvant être testés et qui mettent à disposition leur code source : NetLogo, StarLogo, Gama, Mason et FlameGPU. Pour chaque modèle, nous décrirons comment l'implémentation des trois lois de Reynolds a été réalisée.

NetLogo Dans NetLogo [1] [16], tous les agents (*turtle*) se déplacent et essayent de se rapprocher les uns des autres. Si la distance entre eux est trop faible, ils tentent de se dégager pour éviter de rentrer en collision (R.1), sinon ils s'alignent (R.2). Cependant, R.3 n'est pas implémenté ; il n'y a aucune gestion de la vitesse qui reste ainsi constante tout au long de la simulation.

StarLogo Dans StarLogo [2] [12], chaque agent recherche son plus proche voisin. Si la distance entre eux est trop faible, (R.1) il tourne et s'éloigne pour éviter de rentrer en collision. Sinon, il s'approche de lui et s'aligne sur sa direction de déplacement. La recherche de cohésion n'est pas explicitement exprimée (R.2) et comme pour le modèle précédent, la gestion de la vitesse n'est pas présente (R.3).

Gama Dans Gama [3] [3], les agents commencent par rechercher une cible (assimilable à un but) à suivre. Une fois la cible acquise, les agents se déplacent grâce à trois fonctions indépendantes qui implémentent les règles de Reynolds : une fonction pour éviter les collisions (R.1), une fonction de cohésion (R.2) et une fonction permettant d'adapter la vitesse des agents (R.3). Ce modèle diffère de celui présenté par Reynolds car les agents ont besoin d'une cible pour avoir un comportement de *flocking*.

MasOn MasOn [4] [5] utilise un ensemble de vecteurs pour implémenter R.1 et R.2. Ainsi, le mouvement de chaque agent est calculé à partir d'un vecteur global, ce dernier étant composé d'un vecteur d'évitement, d'un vecteur de cohésion (un vecteur dirigé vers le "centre de masse" du groupe d'entités (R.2)), d'un vecteur moment (un vecteur du déplacement précédent), d'un vecteur de cohérence (un vecteur du mouvement global) et d'un vecteur aléatoire. La vitesse est ici aussi constante pendant toute la simulation, R.3 n'étant pas implémentée.

FlameGPU FlameGPU [5] [15] est la seule implémentation GPGPU que nous avons pu tester. Dans ce modèle, R.1, R.2 et R.3 sont implémentées dans trois fonctions indépendantes. La particularité de ce framework est la nécessité d'adopter un formalisme de conception de

1. https ://ccl.northwestern.edu/netlogo/
2. http ://education.mit.edu/starlogo/
3. https ://code.google.com/p/gama-platform/
4. http ://cs.gmu.edu/˜eclab/projects/mason/
5. http ://www.flamegpu.com/

Plate-forme	Implémentation règles de Reynolds			Caractéristiques principales	Performances
	Collision R.1	Cohésion R.2	Vitesse R.3		
NetLogo	X	X		R.3 n'est pas implémentée : la vitesse des agents est fixée pendant toute la simulation	214 ms (CPU / Logo)
StarLogo	X			Implémentation minimaliste (seul l'évitement d'obstacle est implémenté)	*1000 ms (CPU / Logo)
Gama	X	X	X	Comportement de *flocking* seulement lorsque les agents acquièrent une cible	375 ms (CPU / GAML)
MasOn	X	X		Les règles R.1 et R.2 sont réinterprétées en un calcul de vecteur global qui intègre de l'aléatoire, aucune gestion de la vitesse	45 ms (CPU / Java)
Flame GPU	X	X	X	Les trois règles sont respectées et implémentées telles que définies par Reynolds	*82 ms (GPU / C, XML)

TABLE 1 – Le flocking dans les plate-formes SMA

SMA, basé sur les langages XML et C, qui n'est pas intuitif. Le but est de cacher à l'utilisateur toute la partie GPGPU.

Résumé Dans le but d'avoir un aperçu global des différentes implémentations des *boids* de Reynolds au sein des plates-formes SMA, le tableau 1 résume pour chaque modèle les règles de Reynolds implémentées, énonce les caractéristiques principales des modèles et donne des informations de performances.

Performances Nous évaluons pour chaque modèle le temps de calcul moyen en millisecondes pour une itération. Le but de cette évaluation est de donner une idée des possibilités de chaque implémentation. Ainsi, nous utiliserons comme paramètre commun un environnement de 512 par 512 contenant 4000 agents. 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).

Il faut noter que pour StarLogo, nous avons observé un temps de calcul d'une seconde dès 400 agents simulés. Les performances étant très en dessous des autres plates-formes, nous n'avons pas poussé les tests plus loin. Enfin, pour FlameGPU, il n'a pas été possible de modifier le nombre d'agents dans la simulation qui est de 2048.

3 Proposition d'un modèle de boids

De l'étude précédente, nous remarquons des disparités entre les différents modèles présentés. En effet, les règles de *flocking* proposées par Reynolds autorisent une grande variété d'interprétations. Ainsi, nous remarquons que la règle pour l'adaptation de la vitesse (R.3) est la moins prise en compte (en comparaison de R.1 et R.2 implémentées dans chaque modèle vu à l'exception de StarLogo). Cependant, lorsque R.3 est implémentée, les comportements collectifs deviennent beaucoup plus convaincants et le mouvement global possède alors une dynamique et une fluidité plus intéressante. De même, dans certains travaux, les comportements d'alignement et de cohésion sont confondus ou fusionnés. Les modèles explicitant la différence entre ces deux comportements offrent cependant des déplacements plus intéressants.

Le modèle que nous proposons prendra en compte les points intéressants observés précédemment. Nous avons en effet remarqué que lorsque les trois règles sont intégrées, la dynamique et le mouvement des agents sont plus intéressants. Cela est d'autant plus vrai quand R.3 est prise en compte. Ainsi, notre modèle intègrera R.1, R.2 et R.3 et suivra le principe de parcimonie dans le but de créer une version minimaliste (avec le moins de paramètres possible) se focalisant sur la vitesse et l'orientation de l'agent [6].

Chaque entité a un comportement global qui consiste à se déplacer dans l'environnement tout en adaptant sa vitesse et sa direction en fonction de ses voisins. Ainsi, la proximité avec les autres agents est testée et selon la distance trouvée, les différentes règles de Reynolds sont activées. Plus précisément, chaque agent vérifie si il n'a pas de voisin. Si aucun agent n'est présent dans son champ de vision, il continue à se déplacer dans la même direction. Sinon, l'agent vérifie sa proximité avec ses voisins. Selon la distance trouvée, l'agent va soit se séparer (R.1) s'ils sont trop proches, s'aligner si le nombre de voisins est inférieur à un seuil de cohésion ou

6. L'orientation est un angle en degré (entre 0 et 360) qui donne la direction de l'agent en fonction du repère fixé dans l'environnement

former un groupe dans le cas ou le nombre de voisins est supérieur au seuil défini (R.2). Ensuite, l'agent adapte sa vitesse (R.3), se déplace et recommence le processus. La figure 1 illustre ce comportement global.

Dans notre modèle, il existe deux types différents de paramètres : 5 constantes pour le modèle et 3 attributs spécifiques aux agents. Les constantes sont les suivantes :

- *fieldOfView* (le champ de vision de l'agent) ;
- *minimalSeparationDistance* (la distance minimum entre deux agents) ;
- *cohesionThreshold* (le nombre minimal de voisins pour déclencher un comportement de cohésion) ;
- *maximumSpeed* (la vitesse maximum) ;
- *maximumRotation* (l'angle maximum de rotation).

Les attributs spécifiques à chaque agent sont les suivants :

- *heading* (son orientation) ;
- *velocity* (sa vitesse) ;
- *nearestNeighborsList* (la liste des voisins présents dans son champ de vision).

Comportement de séparation R.1 Ce comportement consiste en la récupération des deux directions (celle de l'agent et de son plus proche voisin). Si ces deux directions mènent à une collision, les deux agents tournent pour s'éviter (voir l'algorithme 1).

Comportement d'alignement R.2 L'alignement se produit lorsque deux agents se rapprochent l'un de l'autre. Ils vont dans ce cas adapter leur orientation de mouvement pour s'aligner et ainsi se diriger vers la même direction (voir l'algorithme 2).

Comportement de cohésion R.2 Quand plusieurs agents sont proches sans avoir besoin de se séparer, ils ont un comportement de cohésion. Ce dernier consiste à calculer la direction moyenne de tous les agents présents dans le champ de vision. Chaque agent va ensuite adapter son orientation en fonction de la valeur trouvée (voir l'algorithme 3).

Adaptation de la vitesse R.3 Avant de se déplacer, les agents doivent adapter leur vitesse (R.3). Durant toute la simulation, chaque agent modifie sa vitesse en fonction de celle de ses voisins. Si l'agent vient d'exécuter le comportement de séparation (R.1), il accélère pour se dégager plus rapidement. Sinon, l'agent ajuste sa vitesse pour la faire correspondre à celle de ses voisins (dans la limite autorisée par la constante *maximumSpeed*).

Tester notre modèle Nous avons mis en ligne un ensemble de vidéos qui montrent notre modèle en action [7]. Sur cette page sont aussi disponibles les codes sources des différents modèles mentionnés et les ressources nécessaires pour tester la solution.

4 Délégation GPU des perceptions agents

4.1 Simulations multiagents et GPGPU

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 multicœur des GPU. 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* (les fils d'instructions). 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* (la figure 2 donne un exemple de l'utilisation d'une grille 2D). 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* [8] exécutent le même *kernel* mais traitent des données différentes selon leur localisation spatiale (identifiant).

Techniques d'implémentation Dans [4], nous avons réalisé un état de l'art de l'utilisation du GPGPU dans les SMA et identifié deux approches permettant d'implémenter un modèle multiagent sur GPU : (1) *tout-sur-GPU* qui consiste à exécuter entièrement la simulation sur la carte graphique et (2) *hybride* pour laquelle la simulation est partagée entre le CPU et le GPU. [4] montre que l'approche hybride

7. www.lirmm.fr/∼hermellin/Website/Reynolds_Boids_With_TurtleKit.html

8. 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.

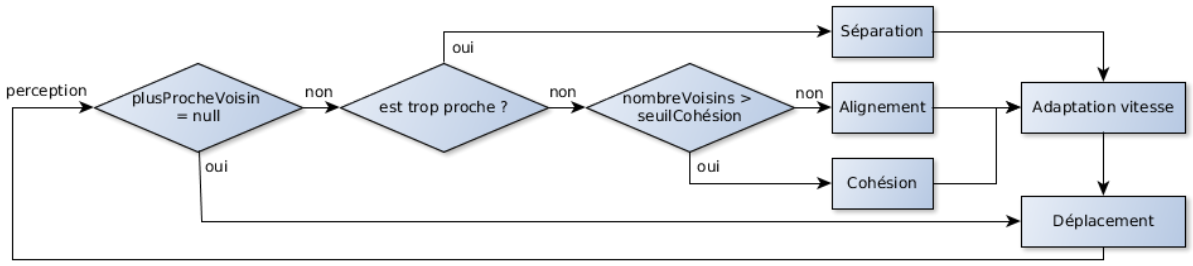

FIGURE 1 – Flocking : comportement global

Algorithme 1 : Séparation

entrées : $myHeading$, $nearestBird$, $maximumRotation$
sortie : $myHeading$ (la nouvelle orientation de l'agent)

$collisionHeading \leftarrow$ headingToward($nearestBird$) ;
si $myHeading$ *seSitueDansInterval*($collisionHeading, maximumRotation$) **alors**
  changeHeading($myHeading$);
fin
return $myHeading$

Algorithme 2 : Alignement

entrées : $myHeading$, $nearestBird$
sortie : $myHeading$ (la nouvelle orientation de l'agent)

$nearestBirdHeading \leftarrow$ getHeading($nearestBird$) ;
si $myHeading$ *estProche*($nearestBirdHeading$) **alors**
  adaptHeading($myHeading$);
fin
sinon
  adaptHeading($myHeading, maximumRotation$);
fin
return $myHeading$

Algorithme 3 : Cohésion

entrées : $myHeading$, $nearestNeighborsList$
sortie : $myHeading$ (la nouvelle orientation de l'agent)

$sumOfHeading, neighborsAverageHeading = 0$;
pour $bird$ *in* $nearestNeighborsList$ **faire**
  $sumOfHeading += getHeading(bird)$;
fin
$neighborsAverageHeading = sumOfHeading/sizeOf(nearestNeighborsList)$;
si $myHeading$ *estProche*($neighborsAverageHeading$) **alors**
  adaptHeading($myHeading$);
fin
sinon
  adaptHeading($myHeading, maximumRotation$);
fin
return $myHeading$

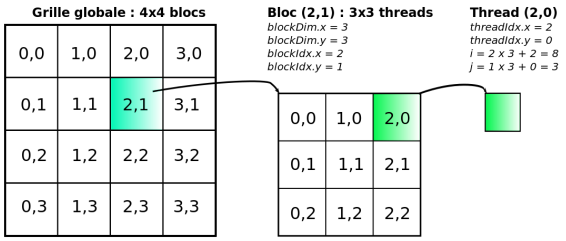

FIGURE 2 – Thread, blocks et grille

apparaît comme la plus prometteuse car elle autorise une plus grande flexibilité (il est possible de choisir ce qui va être exécuté sur le GPU) et augmente l'accessibilité des outils développés.

4.2 Délégation GPU des perceptions agents

Le principe La *délégation GPU des perceptions agents* est basée sur une approche hybride et a été proposée dans [7]. Ce principe 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. L'idée sous-jacente est d'identifier dans les comportements des agents des calculs qui peuvent être transformés en dynamiques environnementales. Ce principe de conception a été 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*".

Travaux connexes Cette approche de *délégation GPU* peut être rapprochée d'autres travaux visant à séparer et/ou déplacer une partie des calculs effectués par les agents dans d'autres structures tels que les interactions ou l'environnement dans le but de répartir la complexité du code et de modulariser son implémentation.

Par exemple, dans le cadre des simulations multiagents, l'approche EASS (*Environment As Active Support for Simulation*)[1] vise à renforcer le rôle de l'environnement en lui déléguant la politique d'ordonnancement ainsi qu'un système de filtrage des perceptions. IODA (*Interaction Oriented Design of Agent simulations*) [11] est quant à elle centrée sur la notion d'interaction et considère que tout comportement réalisable par des agents est décrit de façon abstraite (c'est-à-dire en exprimant ce qu'il a de général) sous la forme d'une règle appelée interaction. Enfin, [10] propose une réduction de la complexité des modèles en se basant sur une approche environnement-centrée : l'environnement devient alors un espace d'échange dédié à l'exécution de dynamiques visant à faciliter la réutilisabilité et l'intégration des différents processus des agents.

Dans un contexte plus générale, l'approche des *artefacts* intègre dans l'environnement un ensemble d'entités dynamiques représentant les ressources et les outils que les agents vont pouvoir utiliser et partager [14, 17]. Ces entités, appelées artefacts, structurent et organisent l'environnement en proposant un modèle de programmation générique englobant les fonctionnalités auxquelles les agents vont avoir accès.

La délégation GPU sur un cas d'étude L'intégration du calcul sur GPU à été réalisée dans TurtleKit [9] [8] qui est une plate-forme de simulation multiagent générique, implémentée en Java, qui utilise un modèle multiagent spatialisé où l'environnement est discrétisé sous la forme d'une grille de cellules. L'approche hybride présente dans TurteKit se focalise sur la modularité et permet d'atteindre plusieurs objectifs : (1) conserver l'accessibilité du modèle agent dans un contexte GPU, (2) passer à l'échelle et travailler avec un grand nombre d'agents sur de grandes tailles d'environnement et (3) promouvoir la réutilisabilité des travaux effectués.

La *délégation GPU* n'a été appliquée pour l'instant que sur un seul modèle dans TurtleKit : un modèle d'émergence multi-niveaux (MLE) [2]. Ce modèle très simple repose sur un unique comportement qui permet de générer des structures complexes qui se répètent de manière fractale. Le comportement agent correspondant est extrêmement simple et repose uniquement sur la perception, l'émission et la réaction à des phéromones. Ainsi, dans ces travaux, des modules GPU pour la perception et la diffusion des phéromones ont été proposés.

5 Délégation GPU et flocking

5.1 Application de la délégation GPU

La *délégation GPU* permet de convertir des comportements ne faisant pas intervenir l'état de l'agent en dynamiques environnementales. Dans notre modèle de *flocking*, le comportement de cohésion se prête à l'application du principe de délégation. En effet, ce comportement

9. http ://www.turtlekit.org

Algorithme 4 : Calcul parallèle : *Kernel* Average

entrées : $width$, $height$, $fieldOfView$, $headingArray$ and $nearestNeighborsList$
sortie : $flockCentering$ (la moyenne des directions)

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 sumOfHeading, flockCentering = 0 ;
4 si i < width et j < height alors
5 |  sumOfHeading = getHeading(fieldOfView, headingArray[i, j]);
6 fin
7 flockCentering[i, j] = sumOfHeading/sizeOf(nearestNeighborsList) ;
```

consiste à réaliser la moyenne des orientations des agents en fonction du champ de vision [10]. Tous les agents doivent réaliser ce calcul qui consiste en une récupération d'une liste de voisins (*nearestNeighborsList*) et d'un parcours séquentiel de la liste pour calculer la moyenne des orientations :

```
pour bird in nearestNeighborsList faire
|  sumOfHeading+ = getHeading(bird);
fin
neighborsAverageHeading =
sumOfHeading/sizeOf(nearestNeighborsList) ;
```

Ce parcours de boucle est lourd car effectué par tous les agents dans leur propre comportement et à chaque pas de simulation.

5.2 Traduction GPU du calcul de la moyenne des orientations

Le calcul de la moyenne est indépendant de l'état des agents et peut être déporté dans l'environnement. Pour ce faire, un tableau 2D (*headingArray*, correspondant à la grille de l'environnement), stocke l'orientation de tous les agents en fonction de leur position. Ce tableau est ensuite envoyé au module GPU *Average Kernel* qui se charge du calcul des orientations moyennes. La traduction GPU consiste donc à transformer le calcul de boucle séquentiel précédemment effectué dans le comportement de cohésion des agents par un calcul parallèle effectué sur le GPU et géré par l'environnement. En fonction du champ de vision de l'agent (*fieldOfView*), le module calcule de manière simultanée la moyenne pour tout l'environnement. Plus précisément, chaque *thread* du GPU calcule la moyenne des orientations d'une cellule selon sa propre position dans la grille GPU (ses identifiants *i* et *j*, algorithme 4). Une fois réalisées, les orientations moyennes sont disponibles dans tout l'environnement. Les agents n'ont donc plus qu'à récupérer dans un tableau 2D (*flockCentering*, retourné par le module GPU) la valeur correspondant à leur position et à adapter leur mouvement.

L'algorithme 4 présente une implémentation du module GPU. Après avoir initialisé les coordonnées *i* et *j* du *thread* utilisé et les variables temporaires (*sumOfheading* et *flockCentering*), on test si le *thread* ne possède pas des coordonnées supérieures à la taille de l'environnement (représenté ici par le tableau 2D *headingArray*). On ajoute ensuite dans *sumOfheading* l'ensemble des orientations des voisins se trouvant dans le champs de vision puis on divise cette valeur par le nombre de voisins pris en compte. Le module retourne ensuite le tableau *flockCentering* contenant toutes les moyennes.

Par rapport à la version séquentielle de l'algorithme, on voit que la boucle a disparu. Ainsi tout l'intérêt de la version GPU tient dans le fait que la parallélisation de cette boucle est réalisée grâce à l'architecture matérielle.

5.3 Implémentation et intégration du module Average

L'implémentation du module GPU dans TurtleKit a été réalisée en CUDA [11] et JCuda [12]. La figure 3 illustre l'application du principe sur notre modèle au sein de TurtleKit.

10. Dans TurtleKit, on appelle champ de vision le nombre de cellules (le rayon autour de la cellule sélectionnée) à prendre en compte pour le calcul de la moyenne.

11. CUDA (Compute Unified Device Architecture), version utilisée : 6.5. http ://www.nvidia.fr/object/cuda-parallel-computing-fr.html

12. La librairie JCuda autorise l'appel de *kernels* GPU, écrits en CUDA, directement depuis Java. Version utilisée : 0.6.5

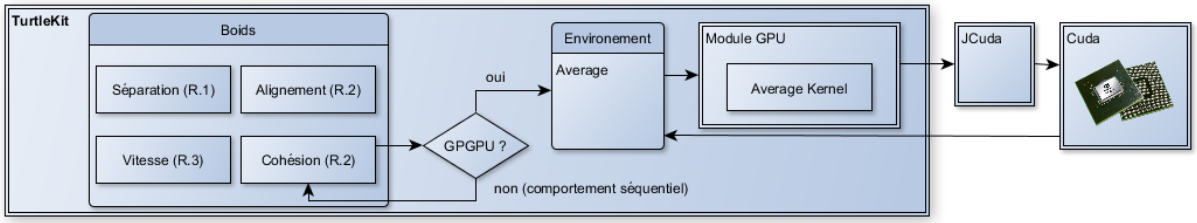

FIGURE 3 – Délégation de la moyenne dans TurtleKit

6 Expérimentation

6.1 Protocole expérimental

Afin de tester notre implémentation des boids de Reynolds et l'application du principe de délégation GPU associée, nous simulons plusieurs tailles d'environnement tout en faisant varier le nombre d'agents et exécutons successivement la version séquentielle du modèle (où la moyenne est calculée dans le comportement des agents) puis la version GPGPU (utilisant le module GPU *Average*). Pour évaluer la performance et rester cohérent avec les critères d'analyse des modèles de la section 2, nous relevons le temps de calcul moyen en millisecondes pour une itération.

6.2 Tests de performances

Pour ces tests, nous réutilisons la même configuration que celle utilisée en section 2 et composée d'un processeur Intel Core i7 (génération Haswell, 3.40GHz), d'une carte graphique Nvidia Quadro K4000 (768 cœurs CUDA) et de 16Go de RAM. La Figure 4 présente les résultats obtenus pour diverses populations d'agents dans des environnements de 256 x 256 et 512 x 512.

Dans l'environnement de 256 x 256, on observe un gain de performance atteignant jusqu'à 25%. Pour l'environnement de 512 x 512, le gain observé est de 15% au maximum. Cette différence de performances s'explique car la densité des agents dans l'environnement est plus faible. Les agents passent donc moins de temps en cohésion et plus à s'aligner et se séparer. Ainsi, selon la densité des agents présents, l'utilisation du module GPU affectera les performances du modèle. Le basculement est clairement visible dans les résultats, lorsque la densité d'agents présents dépasse 5% (respectivement 1500 et 8000 entités), l'utilisation conjointe du CPU et du GPU devient plus efficace. Ainsi, plus la densité d'agents dans l'environnement augmente et plus les gains de performances observés sont importants.

Il faut aussi prendre en compte que les performances obtenues sont intéressantes si l'on considère le matériel utilisé : notre carte graphique Nvidia Quadro K4000 n'est composée que de 768 cœurs CUDA alors que la Nvidia Tesla K40, en contient 2880 et que la Nvidia Tesla K10 en a 3072 (deux GPU de 1536 cœurs sur la même carte).

6.3 Discussion

En plus des résultats de performance observés, on remarque d'autres avantages à l'application du principe de *délégation GPU* : la traduction d'une perception calculée dans le comportement de l'agent en une dynamique de l'environnement permet d'enlever une partie du code source du comportement de l'agent, ce qui simplifie la compréhension de la règle R.2. En effet, l'agent effectue alors une perception directe dans l'environnement à la place d'un calcul séquentiel pouvant être assez lourd.

Un autre aspect intéressant vient du fait que les modules créés, grâce à cette approche, sont indépendants des modèles. Ils ne sont donc pas limités aux contextes pour lesquels ils ont été définis. Nous allons donc continuer d'appliquer le principe de *délégation GPU* pour créer de nouveaux modules GPU et ainsi augmenter le nombre de modules GPU génériques indépendants disponibles ce qui permettra de constituer à terme une librairie de modules utilisable quelque soit le modèle simulé. Cette bibliothèque de fonctions GPU améliorera l'accessibilité de l'approche et l'utilisation du GPGPU dans le cadre des simulations multiagents avec TurtleKit. Cette avancée possible en terme de généricité et d'accessibilité est importante car travailler dans un contexte GPGPU amène souvent des difficultés d'implémentations de par la spécificité de cette technologie.

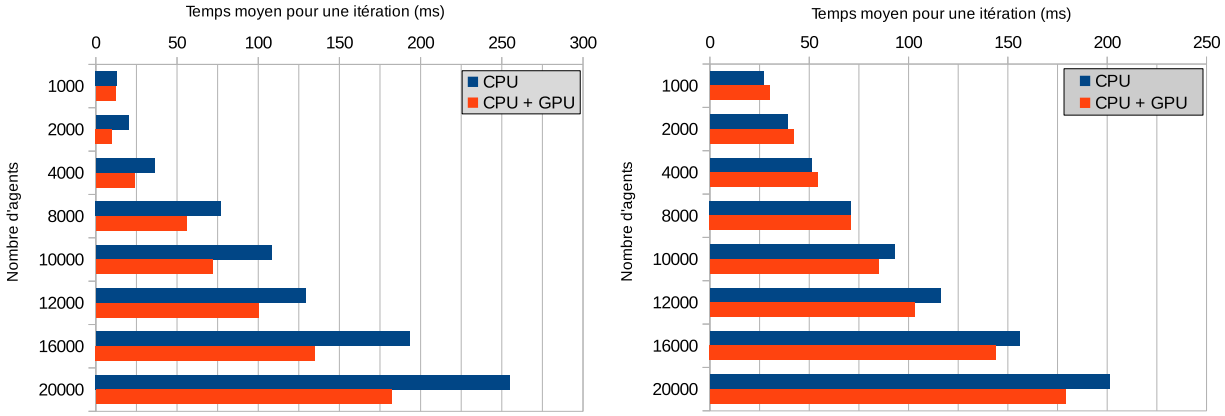

FIGURE 4 – Performance du modèle pour des environnements de taille 256 (gauche) et 512 (droite)

L'application du principe de *délégation GPU* se base sur un critère simple indépendant de l'implémentation. Cela permet de convertir le modèle et de créer le(s) module(s) GPU de manière assez rapide. TurtleKit étant encore en version alpha, nous allons continuer de travailler sur son architecture pour que la conversion d'un modèle soit la plus simple possible.

7 Conclusion et perspectives

Dans cet article, nous avons décrit comment la *délégation GPU des perceptions agents* pouvait être utilisée pour implémenter les boids de Reynolds, en utilisant le GPGPU. Notre objectif était de tester la généricité et les avantages que peut apporter cette approche. Après avoir effectué une analyse des différentes implémentations des modèles de *flocking* au sein des plates-formes multiagents, nous avons proposé notre propre version du modèle. Il a été possible d'appliquer le principe de *délégation GPU* sur le comportement de cohésion. Nous avons ensuite traduit ce comportement en une dynamique environnementale et réalisé des tests de performances.

Nos expérimentations ont montré que la *délégation GPU* a permis d'augmenter le nombre d'agents ainsi que la taille de l'environnement grâce à une accélération de la simulation pouvant atteindre 25% selon les paramètres choisis.

Le principe de délégation représente un modèle de développement qui permet de promouvoir la réutilisabilité des outils créés. Ce critère essentiel est souvent délaissé dans un contexte GPGPU [4]. D'un point de vue génie logiciel, l'utilisation de la *délégation GPU* permet une séparation explicite entre le modèle agent (les comportements de l'agent) et les dynamiques environnementales. L'application du principe autorise ainsi la création de modules GPU génériques indépendants du modèle agent.

Les deux implémentations du principe de délégation, réalisées avec MLE dans [7] et le modèle de *flocking* ici, ont montré que si l'analyse du modèle est faite en gardant à l'esprit les caractéristiques de l'approche, l'identification des comportements, la délégation des calculs et la création des modules GPU peuvent être faciles et rapides. La *délégation GPU des perceptions* nécessitant encore des compétences spécifiques, nous comptons appliquer sur d'autres modèles ce principe dans le but d'éprouver et de continuer à généraliser l'approche.

À long terme notre objectif est donc de proposer une méthodologie explicite de conception, un guide de développement consistant à rendre l'utilisation de la *délégation GPU* plus explicite et plus accessible à des utilisateurs externes. L'idéal étant au final de permettre à chacun de prendre un modèle et l'adapter pour le faire fonctionner dans un contexte GPGPU.

Références

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

[2] Gregory Beurier, Olivier Simonin, and Jacques Ferber. Un modele de systeme multi-agents pour l'emergence multi-niveaux. In *11eme journees Francophones sur les Systemes Multi-Agents*, Revue des

Sciences et Technologies de l'Information, pages 235–247. Hermes, 2003.

[3] Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, DucAn Vo, NghiQuang Huynh, and Alexis Drogoul. GAMA 1.6 : Advancing the Art of Complex Agent-Based Modeling and Simulation. In Guido Boella, Edith Elkind, BastinTonyRoy Savarimuthu, Frank Dignum, and MartinK. Purvis, editors, *PRIMA 2013 : Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg, 2013.

[4] Emmanuel Hermellin, Fabien Michel, and Jacques Ferber. Systèmes multi-agents et GPGPU : état des lieux et directions pour l'avenir. In *Principe de Parcimonie - JFSMA 14 - Vingt-deuxièmes Journées Francophones sur les Systèmes Multi-Agents*, pages 97–106. Cepadues Editions, 2014.

[5] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON : A Multiagent Simulation Environment. *Simulation*, 81(7) :517–527, 2005.

[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. 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.

[8] 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 of the Proceedings of the 1st International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2005, November 27 - December 1, 2005, Yaoundé, Cameroon*, pages 122–128. IEEE, november 2005.

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[10] Denis Payet, Rémy Courdier, Nicolas Sébastien, and Tiana Ralambondrainy. 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*, pages 127–131. IEEE Systems, Man, and Cybernetics Society, 2006.

[11] Sébastien Picault. *From multi-agent simulation to multi-level simulation. Reifying the interactions.* Habilitation à diriger des recherches, Université des Sciences et Technologie de Lille - Lille I, December 2013.

[12] Mitchel Resnick. StarLogo : An Environment for Decentralized Modeling and Decentralized Thinking. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 11–12, New York, NY, USA, 1996. ACM.

[13] Craig W. Reynolds. Flocks, Herds and Schools : A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, volume 21 of *SIGGRAPH Computer Graphics '87*, pages 25–34, New York, NY, USA, 1987. ACM.

[14] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems : 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 simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3) :334–47, 2010.

[16] Elizabeth Sklar. NetLogo, a Multi-agent Simulation Environment. *Artificial Life*, 13(3) :303–311, 2007.

[17] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. Engineering MAS environment with artifacts. In Danny Weyns, H. Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems*, volume 3830 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin Heidelberg, 2006.