Expérimentation du principe de délégation GPU pour la simulation multiagent

Emmanuel Hermellin, Fabien Michel

Expérimentation du principe de délégation GPU pour la simulation multiagent

Les boids de Reynolds comme cas d'étude

Emmanuel Hermellin , Fabien Michel

LIRMM - Laboratoire Informatique Robotique et Microélectronique de Montpellier
Université de Montpellier - CNRS - 161 Rue Ada, 34090 Montpellier, France
{hermellin,fmichel}@lirmm.fr

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 ainsi 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. L'idée derrière ce principe est d'identifier dans le modèle des calculs qui peuvent être transformés en dynamiques environnementales afin d'être calculés par des modules 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 mettre à l'épreuve la faisabilité et la généricité de cette approche en appliquant le principe de délégation GPU sur un modèle agent 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 (généricité, accessibilité, etc.).*

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. The idea is to identify in the model some computations which can be transformed into environmental dynamics and then translated into GPU modules. In this paper, we further trial this principle by testing its feasibility and genericness on a classic ABM, namely Reynolds's boids. 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.*

MOTS-CLÉS : *Simulation multiagent, GPGPU, Boids de Reynolds, CUDA.*

KEYWORDS: *GPGPU, MABS, Flocking, CUDA.*

DOI:10.3166/RIA.30.109-132

1. Introduction

Les simulations multiagents sont utilisées pour étudier, comprendre, prédire et analyser des systèmes complexes dans de nombreux domaines (robotique, biologie, économie, gestion urbaine, etc.) (Michel *et al.*, 2009) pourvu que ces derniers puissent être représentés par un ensemble d'entités autonomes en interaction appelées agents. Cependant, ce genre de simulations peut nécessiter une puissance de calcul très importante surtout si l'on augmente la taille de l'environnement, le nombre d'agents, leur capacité d'interactions, etc. De fait, les performances fournies par nos CPU (*Central Processing Unit*) représentent souvent un verrou majeur qui limite fortement le cadre dans lequel un modèle peut être conçu et expérimenté (Parry, Bithell, 2012).

Dans le but de fournir les ressources en calcul nécessaire, il est possible d'utiliser le HPC (*High Performance Computing*) et en particulier le GPGPU (*General-Purpose computing on Graphics Processing Units*). Cette technologie permet d'effectuer du calcul générique sur carte graphique en utilisant l'architecture massivement parallèle de ces cartes. Ainsi, suivant le contexte, il est possible d'accélérer considérablement les performances des programmes qui l'utilisent (l'application peut être 100 fois plus rapide) [1] (Che *et al.*, 2008). Réelle révolution technologique, il est ainsi possible d'utiliser des ordinateurs classiques pour accélérer les simulations multiagents (Lysenko, D'Souza, 2008).

Cependant, de par l'architecture matérielle très spécifique des GPU (*Graphics Processing Unit*), ce type de programmation requiert des connaissances avancées ce qui limite son utilisation (Owens *et al.*, 2007). En effet, une implémentation sur GPU est bien plus complexe qu'un simple changement de langage de programmation (Bourgoin, 2013). En particulier, le problème doit pouvoir être représenté par des structures de données distribuées et indépendantes. De nombreux travaux ont donc été publiés depuis 2007 (date de début de l'utilisation du GPGPU dans le domaine des systèmes multiagents) dans le but de présenter, faciliter, aider et proposer des solutions permettant d'utiliser le GPGPU (Hermellin *et al.*, 2015). La *délégation GPU des perceptions agents* (que nous nommerons *délégation GPU*) fait partie des approches visant à faciliter l'utilisation du GPGPU.

La délégation GPU est basée sur une approche hybride qui repose sur une utilisation conjointe du CPU 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 (Hermellin *et al.*, 2015). 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 sur un modèle d'émergence multiniveau (MLE) (Beurier *et al.*, 2003) a été proposé dans (Michel, 2014) avec comme objectif de conserver l'accessibilité de la plate-forme TurtleKit [2] (Michel *et al.*, 2005), quant à la simplicité de programmation, tout en tirant

1. *e.g.* https://developer.nvidia.com/about-cuda
2. *e.g.* http://www.turtlekit.org

parti des avantages offerts par le GPGPU. Ces travaux ont montré de bons résultats en ce qui concerne les performances, l'accessibilité et la réutilisabilité.

Le but de cet article est d'évaluer la généricité et les avantages que peut apporter ce principe de délégation sur un type de modèle différent de celui utilisé dans (Michel, 2014). Ainsi, la délégation GPU va être appliquée sur un modèle classique de SMA : les *boids* de Reynolds (Reynolds, 1987). 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. La section 6 présente l'expérimentation et évalue les résultats de cette dernière. La section 7 discute des avantages et limites du principe de délégation GPU. Enfin, la section 8 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 globaux pour réaliser des animations réalistes de nuées d'oiseaux artificiels (*boids*) (Reynolds, 1987). Son idée a donc été d'utiliser une approche individu-centrée : les *boids* doivent être influencés par les autres pour se déplacer d'une manière cohérente et crédible. Ainsi, Reynolds propose que chaque agent soit sensible à des forces régissant les interactions avec les autres agents qui l'entoure. Ainsi, chaque entité va devoir 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.

2.2. Les boids dans les plates-formes SMA

Le modèle de *flocking* de Reynolds fait partie des simulations multiagents les plus connues. Ainsi, de nombreuses plates-formes spécialisées dans le développement de SMA l'intègrent en proposant leur propre implémentation. Parmi tous les travaux identifiés, nous sélectionnons uniquement les modèles pouvant être testés et qui mettent à disposition leur code source : NetLogo, StarLogo, Gama, Mason, Repast et Flame GPU.

Dans le but d'avoir un aperçu de comment ce modèle a pu être interprété, nous comparons dans cette section plusieurs implémentations du modèle. Pour ce faire, nous utilisons deux critères de comparaisons :

- Est-ce que toutes les règles comportementales ont été implémentées ?
- Ces règles sont-elles cohérentes avec celles énoncées par Reynolds ?

2.2.1. L'implémentation des différents modèles

Dans NetLogo [3] (Sklar, 2007), tous les agents 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 d'entrer en collision (R.1), sinon ils s'alignent (R.2). Cependant, R.3 n'est pas implémentée : il n'y a aucune gestion de la vitesse des agents qui reste ainsi constante tout au long de la simulation.

Dans StarLogo [4] (Resnick, 1996), chaque agent recherche son plus proche voisin. Si la distance entre eux est trop faible, il tourne et s'éloigne pour éviter d'entrer en collision (R.1). Sinon, il s'approche de lui. La recherche de cohésion n'est pas explicitement exprimée (R.2) et comme pour le modèle précédent, la variation de la vitesse n'est pas présente (R.3).

Dans Gama [5] (Grignard *et al.*, 2013), 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 aux agents d'adapter leur vitesse à celle des voisins (R.3). Ce modèle diffère de celui présenté par Reynolds car les agents ont besoin d'une cible pour avoir un comportement de *flocking*.

Dans MasOn [6] (Luke *et al.*, 2005), le modèle 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.

Dans Repast [7] (North *et al.*, 2007), les règles R.1 et R.2 sont explicitement implémentées. Cependant, R.1 et R.2 sont exécutées par les agents à la suite dans un comportement unique. De plus, dans le comportement de cohésion, la distance entre les agents est forcée (et ne résulte donc pas des interactions entre ces derniers). Enfin, nous notons que R.3 n'est pas implémentée.

Flame GPU [8] (Richmond *et al.*, 2010) est la seule implémentation GPGPU répondant à nos critères que nous avons pu tester [9]. Dans ce modèle, R.1, R.2 et R.3 sont explicitement implémentées dans trois fonctions indépendantes.

---

3. *e.g.* https://ccl.northwestern.edu/netlogo/
4. *e.g.* http://education.mit.edu/starlogo/
5. *e.g.* https://code.google.com/p/gama-platform/
6. *e.g.* http://cs.gmu.edu/~eclab/projects/mason/
7. *e.g.* http://repast.sourceforge.net/, *e.g.* https://code.google.com/p/repast-demos/wiki/BoidsJava
8. *e.g.* http://www.flamegpu.com/
9. La particularité de ce framework est la nécessité d'adopter un formalisme de conception de 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.

	Implémentation règles de Reynolds			Caractéristiques	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 par itér. (CPU / Logo)
StarLogo	X			Implémentation minimaliste (seul l'évitement d'obstacle est implémenté)	*1000 ms par itér. (CPU / Logo)
Gama	X	X	X	Comportement de *flocking* seulement lorsque les agents acquièrent une cible	375 ms par itér. (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 par itér. (CPU / Java)
Repast	X	X		R.3 non implémentée, R.1 et R.2 exécutées à la suite par les agents dans un comportement unique	(CPU / Java)
Flame GPU	X	X	X	Les trois règles sont respectées et implémentées telles que définies par Reynolds	*82 ms par itér. (GPU / C, XML)

Tableau 1. Le flocking dans les plates-formes SMA

2.2.2. Performances

Nous évaluons pour chaque modèle le temps de calcul moyen en millisecondes pour une itération (*ms par itér.*), les temps les plus faibles étant les meilleurs. 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 cellules contenant 4 000 agents. Notre configuration de test est composée d'un processeur Intel Core i7 (génération Haswell, 3.40 GHz) et d'une carte graphique Nvidia Quadro K4000 (768 cœurs). Les données de performances sont visibles dans le tableau 1.

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 Flame GPU, il faut préciser deux points importants : (1) la simulation est exécutée dans un environnement 3D et (2) il n'a pas été possible de modifier le nombre d'agents dans la simulation qui est fixé à 2 048. Les resultats de performance de ce dernier ne sont donc pas vraiment comparables avec ceux des autres plates-formes.

Le tableau 1 résume pour chaque modèle les règles de Reynolds implémentées, énonce les caractéristiques principales des modèles et donne des informations de performances [10].

10. Le sigle * indique que les paramètres de test sont légèrement différents de ceux énoncés dans cette section, à cause de restrictions propres à la plate-forme utilisée ou à l'implémentation.

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 sont plus intéressants. En effet, bien qu'il s'agisse là d'une appréciation subjective du rendu visuel, on voit que la prise en compte de cette règle influence concrètement le mouvement global des agents et fait apparaitre des dynamiques plus intéressantes car les agents n'ont pas tous la même vitesse.

De même, certains travaux divisent la règle R.2 en deux comportements distincts : un comportement d'alignement et un de cohésion. Les modèles explicitant cette différence offrent des dynamiques plus complexes dans le mouvement global des agents.

Ainsi, pour notre modèle, nous prendrons en compte les points intéressants observés précédemment. Notre modèle intègrera donc R.1, R.2 (en distinguant l'alignement et la cohésion) et R.3. Il suivra aussi 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.

3.1. Définition du modèle

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 s'il n'a pas de voisin dans son entourage. 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) dans le cas ou ses voisins sont trop proches, s'aligner si le nombre de voisins est inférieur à un seuil de cohésion ou rentrer en cohésion dans le cas où 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.

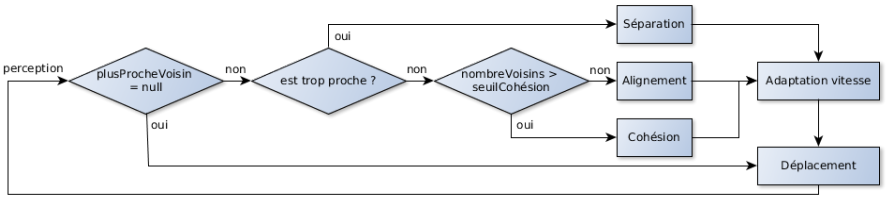

Figure 1. Flocking : comportement global des agents

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 de voisins pour rentrer en cohésion ;
– *maximumSpeed* : la vitesse maximum ;
– *maximumRotation* : l'angle maximum de rotation.

Les attributs spécifiques à chaque agent sont les suivants :

– *heading* : son orientation [11] ;
– *velocity* : sa vitesse ;
– *nearestNeighborsList* : la liste des voisins présents dans son champ de vision.

3.2. Définition des comportements

3.2.1. Comportement de séparation R.1

La séparation se produit lorsque deux agents sont trop proches l'un de l'autre. Ce comportement consiste en la récupération des deux directions (celles 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).

Algorithme 1 : Comportement de séparation (R.1)

entrées : $myHeading$, $nearestBird$, $maximumRotation$
sortie : $myHeading$ (la nouvelle orientation de l'agent)
1 $collisionHeading \leftarrow$ headingToward($nearestBird$) ;
2 **si** $myHeading$ *isInTheInterval(*$collisionHeading, maximumRotation$*)* **alors**
3 adaptHeading($myHeading, maximumRotation$);
4 **fin**
5 **return** $myHeading$

3.2.2. 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).

3.2.3. 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

11. 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

Algorithme 2 : Comportement d'alignement (R.2)

entrées : $myHeading, nearestBird$
sortie : $myHeading$ (la nouvelle orientation de l'agent)
$nearestBirdHeading \leftarrow$ getHeading($nearestBird$) ;
si $myHeading$ *isNear(*$nearestBirdHeading$*)* **alors**
  adaptHeading($myHeading, rotationAngle$);
fin
sinon
  adaptHeading($myHeading, maximumRotation$);
fin
return $myHeading$

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).

Algorithme 3 : Comportement de cohésion (R.2)

entrées : $myHeading, nearestNeighborsList$
sortie : $myHeading$ (la nouvelle orientation de l'agent)
$sumOfHeading, neighborsAverageHeading = 0$;
pour $bird$ ***dans*** $nearestNeighborsList$ **faire**
  $sumOfHeading += getHeading(bird)$;
fin
$neighborsAverageHeading = sumOfHeading/sizeOf(nearestNeighborsList)$;
si $myHeading$ *isNear(*$neighborsAverageHeading$*)* **alors**
  adaptHeading($myHeading, rotationAngle$);
fin
sinon
  adaptHeading($myHeading, maximumRotation$);
fin
return $myHeading$

3.2.4. 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*).

3.3. Implémentation de notre modèle de flocking

L'implémentation de notre modèle de *flocking* a été réalisée dans TurtleKit 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. Toutes les ressources nécessaires (code source et documentation)

pour exécuter ce modèle sont disponibles en ligne [12]. De plus, un ensemble de vidéos montrant notre modèle en action ainsi que les codes sources des différents modèles mentionnés sont disponibles sur cette page.

4. Délégation GPU des perceptions agents

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 multicœur des GPU. Une des grandes différences entre l'architecture CPU et l'architecture GPU vient donc du nombre de processeurs (d'ALU, *Arithmetic Logic Units*) qui composent un GPU, bien plus important que pour un CPU, comme le montre la figure 2.

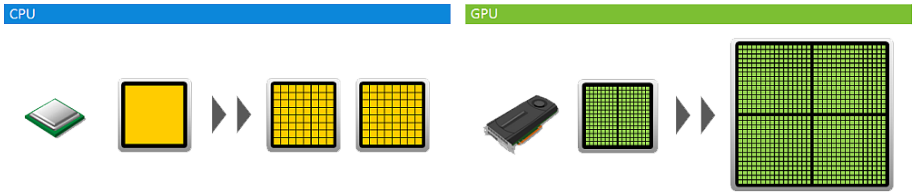

Figure 2. Evolution des CPU et GPU vers des architectures "many-core"

Ainsi, les GPU ont évolué et ne sont plus destinés au seul affichage graphique. Ils sont maintenant composés de centaines d'ALU formant une structure hautement parallèle capable de réaliser des tâches plus variées. Ces ALU ne sont pas très rapides (beaucoup moins qu'un CPU [13]) mais permettent d'effectuer des milliers de calculs similaires de manière simultanée. Le paradigme de programmation derrière le GPGPU est basé sur le modèle de calcul parallèle SIMD (*Single Instruction Multiple Data*) que l'on appelle aussi *Stream Processing*. Il consiste en l'exécution simultanée d'une série d'opérations (un noyau de calcul – *kernel*) sur un jeu de données (le flux – *stream*).

Pour utiliser les capacités GPGPU de notre carte graphique, nous utilisons CUDA [14] (*Compute Unified Device Architecture*) qui est l'interface de programmation dédiée fournie par Nvidia. 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* [15] (les

12. *e.g.* http://www.lirmm.fr/~hermellin/Website/Reynolds_Boids_With_TurtleKit.html

13. En effet, les ALU les plus rapides tournent à une fréquence de 1.2 GHz alors que le cœur d'un CPU peut monter à 5.0 GHz (en fonction du processeur).

14. *e.g.* https://developer.nvidia.com/what-cuda

15. 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.

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*. 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. En général (et dans la suite de ce document), les identifiants des *threads* dans la *grille globale* du GPU seront notés *i* et *j* (nous travaillons sur des environnements 2D). La figure 3 donne un exemple de l'utilisation d'une grille 2D.

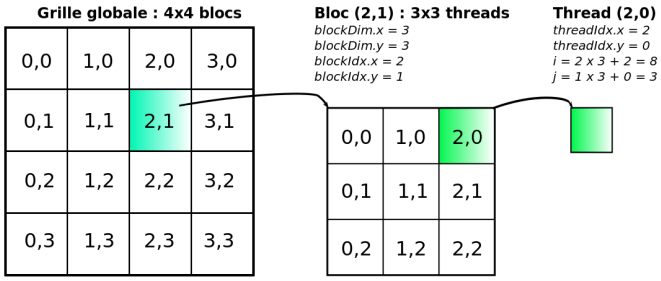

Figure 3. Thread, blocks et grille

CUDA fournit un environnement logiciel de haut niveau permettant aux développeurs de contrôler les GPU par le biais d'une extension du langage C. Cependant, il existe des librairies permettant d'utiliser CUDA au travers d'autres langages comme Java avec JCUDA [16]. Nous utilisons cette dernière librairie pour intégrer le GPGPU dans la plate-forme TurtleKit.

4.2. Simulations multiagents et GPGPU

De l'état de l'art de l'utilisation du GPGPU dans les SMA (Hermellin *et al.*, 2015), il est ressorti que l'utilisation des cartes graphiques permettaient d'accélérer grandement les simulations multiagents surtout lorsque l'on s'intéresse à des populations d'agents ou des tailles d'environnement importantes. Cependant, l'architecture particulière des GPU nécessite des méthodes de programmation spécifiques ainsi que la résolution de problèmes d'implémentation, d'exécution et de structuration des données propres à ce matériel hautement parallèle. Ainsi, une utilisation performante du GPU n'est souvent possible qu'au dépend de la modularité, la programmabilité, l'accessibilité et la réutilisabilité des modèles.

En effet, (Hermellin *et al.*, 2015) identifie deux approches pour implémenter un modèle en utilisant le GPGPU : (1) *tout-sur-GPU* qui consiste à exécuter entièrement

16. *e.g.* http://www.jcuda.org

la simulation sur la carte graphique et (2) *hybride* pour laquelle la simulation est partagée entre le CPU et le GPU. Dans le premier cas, il n'est pas trivial de prendre un modèle et de le transformer pour le faire fonctionner sur la carte graphique. En effet, les GPU peuvent être très restrictifs quant au type de données utilisables ou fonctions exécutables. L'approche hybride (2) est certe moins performante qu'une approche tout-sur-GPU (1) cependant le fait qu'elle autorise une utilisation conjointe du CPU et du GPU apporte plusieurs avantages majeurs. En permettant de choisir ce qui va être exécuté sur le GPU et par le CPU, l'approche hybride se veut plus flexible et ouverte aux autres technologies augmentant ainsi l'accessibilité des outils développés (ce qui est clairement visible dans (Laville *et al*., 2012 ; Sano *et al*., 2014 ; Vigueras *et al*., 2010)). De plus, cette approche est plus modulaire par sa conception, rendant ainsi possible l'utilisation d'architectures agents plus complexes et hétérogènes. Enfin, même si la généricité n'est pas l'objectif premier, ce type d'approche permet la réutilisabilité des outils créés, la librairie MCMAS (Laville *et al*., 2014) en est un bon exemple.

4.3. Délégation GPU des perceptions agents

L'approche hybride est, à l'heure actuelle, la solution la plus adaptée et la plus prometteuse pour permettre une utilisation performante et efficace du GPGPU dans un contexte de simulations multiagents (Hermellin *et al*., 2015). Elle permet, en effet, de traiter des problèmes d'accessibilité, réutilisabilité, modularité, etc. récurrents avec une approche tout-sur-GPU.

Le principe de *délégation GPU des perceptions agents* se base donc sur cette approche et a été proposée dans (Michel, 2014). Ce principe repose sur 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. Le but étant de déplacer les calculs les plus coûteux et les calculs qui n'interfèrent pas avec les comportements, sur le GPU. 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*".

La *délégation GPU* n'a été appliquée pour l'instant que sur un seul modèle dans TurtleKit (Michel, 2014) : un modèle d'émergence multi-niveaux (MLE) (Beurier *et al*., 2003). 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.

L'intégration du calcul sur GPU a été réalisée dans TurtleKit et se focalise sur la modularité (grâce à l'approche hybride). Cela a permit 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.

4.4. Relation avec d'autres travaux de recherche

Plus généralement, la délégation GPU s'inspire d'une approche AOSE (*Agent-Oriented Software Engineering*) qui consiste à utiliser l'environnement comme une abstraction du premier ordre dans les SMA (Weyns *et al*., 2005 ; Weyns, Michel, 2015). Cette idée est bien acceptée de nos jours et a prouvé sa pertinence dans le cadre du développement de SMA (Weyns, Holvoet, 2005). Arborant un point de vue environnement-centrée, la délégation GPU doit être rattacher à d'autres travaux tels que l'approche EASS (*Environment As Active Support for Simulation*) (Badeig, Balbo, 2012) qui 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*) (Kubera *et al*., 2011) 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, (Payet *et al*., 2006) 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 général, l'approche des *artefacts* intègre dans l'environnement un ensemble d'entités dynamiques structurant les ressources et les outils que les agents vont pouvoir utiliser et partager (Ricci *et al*., 2011 ; Viroli *et al*., 2006).

Ainsi, tous ces travaux de recherche visent à réifier une partie des calculs effectués dans le comportement des agents dans de nouvelles structures telles que les interactions ou l'environnement dans le but de répartir la complexité du code et de modulariser son implémentation.

5. Délégation GPU et flocking

5.1. Adaptation de la délégation GPU

La délégation GPU consiste donc à identifier des calculs au sein des comportements qui peuvent être déportés dans des dynamiques environnementales pour ensuite être calculés par des modules GPU. Cette délégation précise que ce sont les perceptions agent qui n'impliquent pas les états des agents qui peuvent être transformées. Ainsi, dans le cas d'étude présentant une adaptation du modèle MLE, ce sont les calculs relatifs à la diffusion, évaporation et au suivi de gradient des phéromones dans l'environnement qui ont été déportés dans des dynamiques environnementales gérées par des modules GPU. Ces calculs sont indépendants des états des agents car ces derniers n'utilisent les valeurs calculées qu'en tant que perceptions pour leurs comportements de déplacement ou d'évolution.

Cependant, dans notre modèle de *flocking*, il n'est pas possible de trouver des calculs indépendants des états des agents. Il est donc nécessaire de faire évoluer le principe de délégation GPU afin qu'il puisse être appliqué sur un plus grand nombre de modèles. Cette évolution va porter sur le type de calculs qu'il est possible de déporter. S'il n'existe pas de calculs indépendants des états des agents, il est cependant possible d'identifier des calculs ne *modifiant* pas les états des agents. Ces derniers peuvent aussi être transformés en dynamiques environnementales et donc convertis en module GPU.

5.2. Application de la délégation GPU

Dans notre modèle de *flocking*, le comportement de cohésion se prête bien à l'application de cette nouvelle version du principe de délégation. En effet, ce comportement consiste à réaliser la moyenne des orientations des agents en fonction du champ de vision [17]. 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. Ce processus est résumé dans la figure 4 et l'algorithme 4 présente le calcul effectué.

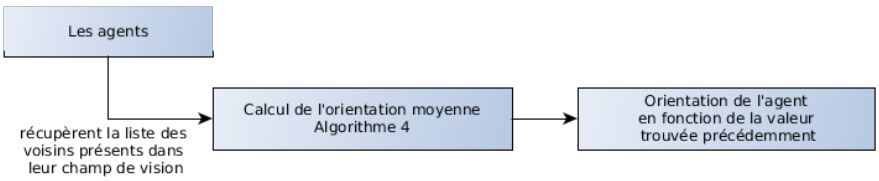

Figure 4. Déroulement du comportement de cohésion avant délégation

Algorithme 4 : Calcul séquentiel de la moyenne des orientations

pour $bird$ ***dans*** $nearestNeighborsList$ **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 à chaque pas de simulation. Ainsi, le temps de calcul et les ressources nécessaires sont directement impactés par le nombre d'agents présents dans la simulation et par le champ de vision des agents (plus le champ est grand et plus la liste d'agents récupérée peut être longue). Il est donc intéressant de considérer la transformation de ce calcul en dynamique environnementale calculée par un module GPU.

17. 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.

5.3. Traduction GPU du calcul de la moyenne des orientations

Pour pouvoir appliquer ce principe de délégation GPU sur notre modèle, nous allons extraire de l'information des attributs des agents (*heading*) et déléguer le calcul associé (la boucle séquentielle) dans une dynamique environnementale et donc à la carte graphique. 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 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 5). 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. Le processus est résumé par la figure 5.

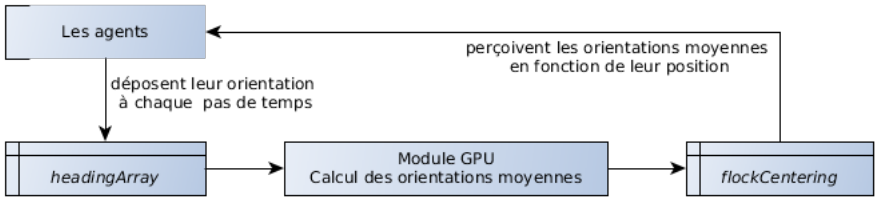

Figure 5. Déroulement du comportement de cohésion après délégation

L'algorithme 5 présente une implémentation du module GPU. Après avoir initialisé les coordonnées *i* et *j* du *thread* utilisé et les variables temporaires (*sumOfheading* et *flockCentering*), nous testons 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.

Algorithme 5 : Calcul parallèle : *Kernel* Average

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

$i = blockIdx.x * blockDim.x + threadIdx.x$;
$j = blockIdx.y * blockDim.y + threadIdx.y$;
$sumOfHeading, flockCentering = 0$;
si $i < width$ ***et*** $j < height$ **alors**
  $sumOfHeading =$ getHeading($fieldOfView, headingArray[i, j]$);
fin
$flockCentering[i, j] = sumOfHeading / sizeOf(nearestNeighborsList)$;

5.4. Intégration du module GPU

Tout comme pour le cas d'étude présenté dans (Michel, 2014) (le modèle d'émergence multiniveau), l'intégration du module GPU *Average* a été réalisée dans la plateforme TurtleKit en version 3.0.0.4 [18]. L'environnement de développement CUDA a été utilisé en version 6.5 [19] et la version 0.6.5 de la librairie JCUDA [20] a servi pour faire l'interface entre CUDA et TurtleKit. La figure 6 illustre l'application du principe sur notre modèle et l'intégration du module au sein de TurtleKit.

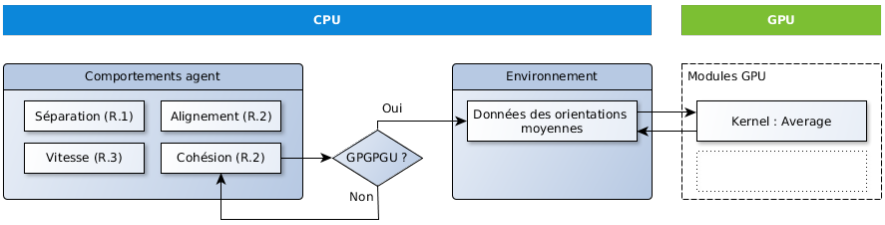

Figure 6. Délégation de la moyenne dans TurtleKit

6. Expérimentation

6.1. Protocole expérimental

Dans la section 2, nous avons présenté pour chacun des modèles de *flocking* sélectionné, des données relatives aux performances. Cependant, au vu de la très grande disparité entre les différentes implémentations ainsi que de par l'hétérogénéité des

18. http://www.turtlekit.org
19. http://www.nvidia.fr/object/cuda-parallel-computing-fr.html
20. La librairie JCuda autorise l'appel de *kernels* GPU, écrits en CUDA, directement depuis Java. http://www.jcuda.org

plates-formes (et matériels utilisés), nous ne prendrons pas en compte ces valeurs dans nos résultats de performances concernant notre modèle et l'application de la *délégation GPU* sur ce dernier.

Le protocole expérimental permettant de tester les performances de nos implémentations est le suivant : 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 toutes les simulations lancées, le champ de vision des agents est fixé à 10 (un rayon de 10 cases autour de l'agent). Pour 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 (les temps les plus faibles sont les meilleurs).

Pour ces tests, nous réutilisons la même configuration que celle utilisée en section 2 qui est composée d'un processeur Intel Core i7 (génération Haswell, 3.40 GHz), d'une carte graphique Nvidia Quadro K4000 (768 cœurs CUDA) et de 16 Go de RAM. La figure 7 présente les résultats obtenus pour un environnement de taille 256 et 512 avec une densité d'agents variant entre 1 % et 60 %.

6.2. Évaluation des performances

Des résultats de performances obtenus, il est possible d'éditer deux graphiques contenant les coefficients d'accélération obtenus entre une exécution séquentielle du modèle et une exécution parallèle utilisant le GPGPU. La figure 8 présente ces graphiques et donne les coefficients d'accélération pour des environnements de taille 256 et 512.

De ces graphiques, nous pouvons tirer les observations suivantes : dans l'environnement de taille 256, on observe que le gain augmente assez rapidement puis stagne ensuite rapidement autour d'un gain de performances de 40 à 50 %. Pour l'environnement de taille 512, le profil d'accélération est assez différent. Il est assez lent au départ puis augmente rapidement jusqu'à un gain de 40 %.

D'une manière générale, les gains de performances et le profil d'accélération sont fortement reliés à la densité des agents présents dans l'environnement. En effet, lorsque cette dernière est faible (inférieure à 10 %), une grande partie des agents ne se trouvent pas dans un comportement de cohésion : les agents passent moins de temps en cohésion et plus à s'aligner et se séparer. La simulation profite donc moins de l'accélération offerte par l'utilisation du module GPU. Cependant, passé un certain seuil (une densité d'agents d'environ 15 %), le basculement devient clairement visible dans les résultats et l'utilisation conjointe du CPU et du GPU devient plus efficace. Ainsi, plus la densité d'agents dans l'environnement est importante et plus les gains de performances observés augmentent.

L'utilisation du GPGPU permet donc de travailler avec des environnement plus grands et/ou des populations d'agents plus importantes. C'est d'ailleurs dans ces si-

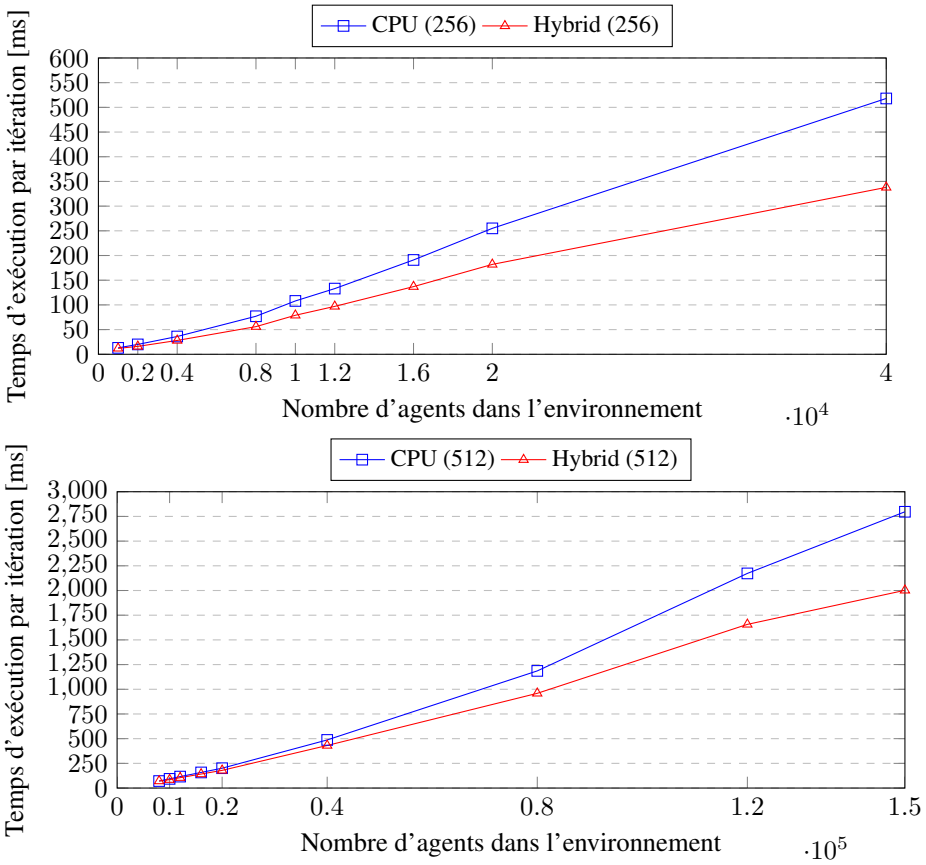

Figure 7. Performance du modèle pour un environnement de taille 256 et 512 avec une densité d'agents variant entre 1 % et 60 %

tuations que cette technologie exprime tout son potentiel. Cependant, il y a des limites. En effet, on peut observer, dans l'environnement de taille 256, une stagnation des performances au dessus d'une certaine densité d'agents (25 %). Cela s'explique car, dans notre modèle, seul le comportement de cohésion profite du GPGPU. Les ressources consommées par les autres comportements, par l'ordonnancement des agents, etc. limitent les performances générales (ces derniers étant toujours exécutés par le CPU). Étant basé sur une approche hybride, le principe de délégation GPU ne délègue que les calculs identifiés comme coûteux et seul une partie du modèle profite de la parallélisation sur GPU. Il y a donc un compromis à faire entre performances et avantages conceptuels : généricité, accessibilité, modularité, etc. Ces points sont d'ailleurs abordés en section 7.

Enfin, il faut aussi prendre en compte que les performances obtenues sont significatives si l'on considère le matériel utilisé : notre carte graphique Nvidia Quadro

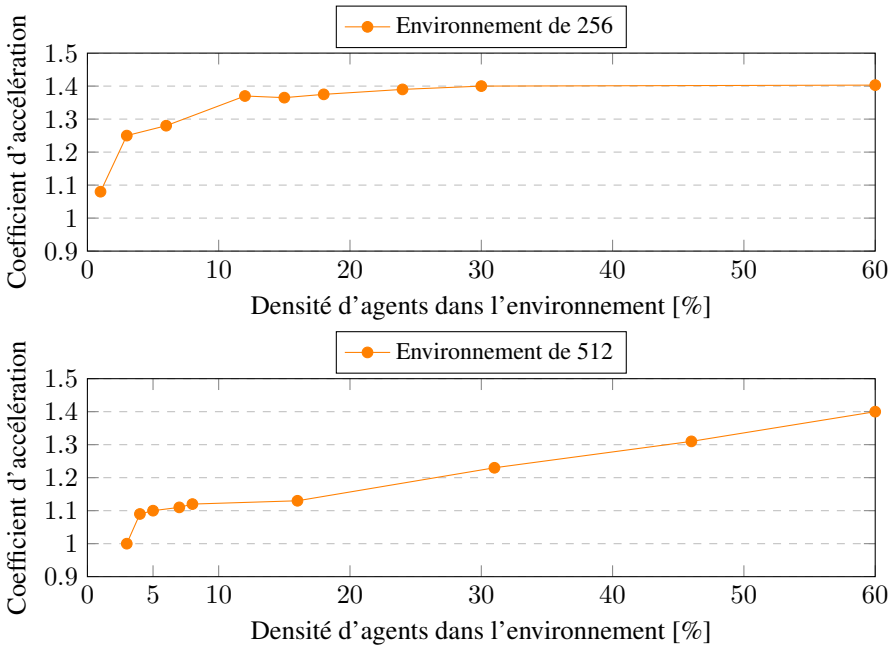

Figure 8. Coefficient d'accélération obtenu entre une exécution CPU et Hybride pour des environnements de taille 256 et 512

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). De plus, l'optimisation des outils utilisés (TurtleKit), de l'utilisation du GPGPU (nombre de *threads*, *blocks*, etc.) ainsi que de la structuration des données rentrent en jeu et peuvent impacter les performances.

7. Discussion autour de la délégation GPU

L'utilisation du principe de délégation GPU a permis d'obtenir une exécution du modèle jusqu'à 40 % plus rapide en comparaison d'une implémentation sur CPU. Mais, plus que l'aspect performance, dans cette section, nous discutons des avantages qu'apporte l'application de ce principe et nous montrons qu'il existe encore des limites dans son utilisation.

7.1. Généricité, accessibilité et modularité

Le principe de délégation GPU utilisé dans cet article se base sur une approche hybride car cette dernière permet, de par l'utilisation conjointe et efficiente du CPU et du GPU, de répondre à des problèmes qu'une approche tout-sur-GPU ne peut résoudre (Hermellin *et al.*, 2015). L'approche hybride se veut plus flexible (augmentant ainsi

l'accessibilité des outils développés), plus générique (offrant la possibilité de réutiliser les outils créés) et plus modulaire de par sa conception.

Ces bénéfices ont pu être observés lors de l'application du principe sur notre modèle et lors des expérimentations. Un des premiers avantages est que le module GPU créé à la suite de la délégation du calcul des orientations moyennes est au final indépendant du modèle pour lequel il a été conçu. En effet, les agents ne font que déposer de l'information dans des structures de données adaptées et gérées par les dynamiques environnementales qui sont ensuite envoyées au module GPU.

Ainsi, les modules réalisés grâce au principe de délégation GPU ne font que recevoir et traiter un flux de données. Ils ne sont donc pas limités aux contextes pour lesquels ils ont été définis. Ce résultat est important car une grande majorité des travaux traitant de l'utilisation du GPGPU dans un contexte de simulations multiagents sont souvent à "usage unique" et non réutilisable (Hermellin *et al.*, 2015). On a donc une augmentation de la réutilisatilité des outils créés. Cela avait déjà été souligné dans le cas d'étude sur le modèle d'émergence (MLE) (Michel, 2014) dans lequel des modules GPU pour la perception et la diffusion des phéromones ont été proposés.

Ce point est important car, au fur et à mesure que ce principe de délégation GPU va être appliqué, il va être possible de récupérer les modules GPU et ainsi créer une librairie de modules GPU génériques indépendants utilisables quels que soient le contexte et le modèle simulé.

La définition de cette bibliothèque de fonctions GPU est un bon moyen d'améliorer l'accessibilité du GPGPU dans le cadre des simulations multiagents (à l'instar de ce qui est fait avec MCMAS dans (Laville *et al.*, 2014)). 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.

De plus, 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. Cependant, 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.

Enfin, nous avons vu en section 4.4 que le principe de délégation GPU pouvait être lié à d'autres travaux réifiant une partie des calculs effectués dans le comportement des agents dans de nouvelles structures (EASS, IODA, etc.). On retrouve des propriétés et une certaine similarité entre ces travaux et notre principe de délégation GPU. En effet, 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. Ceci a pour effet de simplifier le codage du comportement, sa lisibilité et donc de faciliter la compréhension de ce dernier.

7.2. Dynamique globale du système

Lorsque nous exécutons les versions CPU puis CPU + GPU (hybride), nous pouvons remarquer que, visuellement parlant, la dynamique globale du système est complètement différente entre les deux versions. En effet, nos simulations de *flocking* donnent des comportements collectifs très différents suivant qu'on les exécute avec ou sans GPU.

Dans la version séquentielle (CPU), les *boids* perçoivent et agissent directement l'un après l'autre. Ainsi, un pas de simulation va faire converger l'orientation moyenne des agents du système vers une valeur commune. En effet, le premier *boid* s'aligne et modifie sa direction, le deuxième en fait autant mais perçoit la nouvelle direction du premier et la prend en compte, et ainsi de suite. Au final, le dernier *boid* qui agit perçoit un système dont la configuration est très homogène : tous les agents vont dans la même direction [21], modulo une petite variation aléatoire sur les directions individuelles. Il en résulte un *flocking* statique (mouvement coordoné dans une unique direction modulo de petites variations) similaire à ce que l'on peut observer dans les plates-formes que nous avons présentées.

Dans la version utilisant le module GPU, tous les agents perçoivent le même état de l'environnement pour un même instant t. Les données utilisées par les agents pour effectuer leur perception sont calculées au préalable par l'environnement (par le biais des dynamiques environnementales et des modules GPU). Ainsi, on obtient localement des comportements de *flocking* mais le système ne converge pas vers une valeur unique pour l'orientation au fil de la simulation, comme c'est le cas pour la version séquentielle. Il en résulte des comportements collectifs plus riches en terme de dynamique, avec des changements de directions collectifs soudains et importants.

L'application du principe de délégation sur un modèle peut donc ne pas être neutre de par la séparation entre les calculs liés aux agents (CPU) et les calculs liés à l'environnement (GPU). C'est pour cela que l'on obtient des dynamiques globales très différentes.

Il serait donc intéressant d'étudier l'impact de cette transformation sur d'autres modèles. En particulier, il semble possible d'obtenir des dynamiques qui ne sont pas naturelles à implémenter en séquentiel, comme celles basées sur le modèle IRM4S (influence / réaction) (Michel, 2015). Ainsi, le GPGPU devrait permettre d'explorer de nouvelles dynamiques et à ouvrir d'autres champs de recherche intéressants dans le contexte des simulations multiagents.

21. Nous notons qu'une activation différente des agents n'apporte aucun changement : le système converge toujours vers une valeur moyenne.

8. 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. Nous avons montré qu'il était nécessaire de faire évoluer le principe de *délégation GPU* pour rendre possible son application sur le comportement de cohésion de notre modèle (et de manière générale sur une plus grande variété de modèles). Cette évolution se base sur le type de calculs qu'il est possible de déporter. Auparavant, les calculs déportés devaient être indépendants des comportements. Maintenant, ils doivent seulement ne pas modifier les états des agents. Suite à cette évolution, le principe a été appliqué sur le modèle de flocking et la partie calculatoire du comportement de cohésion a été traduite en une dynamique environnementale. Des tests de performances ont ensuite été proposés.

D'un point de vue performance, nous avons vu que l'utilisation du GPGPU au travers du principe de délégation GPU a permis d'obtenir des gains de performances allant jusqu'à 40 % lors de nos expérimentations. Ce résultat est bien sûr fortement lié à la densité des agents ainsi qu'à la taille de l'environnement mais aussi à l'optimisation des outils et au matériel utilisé.

De plus, 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 (Hermellin *et al.*, 2015). 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. Enfin, nous avons vu que l'utilisation de modules GPU produit des dynamiques collectives plus complexes.

Les résultats obtenus lors de ces premières expérimentations (MLE et *flocking*) sont encourageants mais ne sont que préliminaires. Ils représentent une preuve de concept mais un certain nombre d'étapes restent à franchir afin de constituer un ensemble cohérent de modèles, d'outils et de principes méthodologiques associés au contexte du GPGPU. Dans l'immédiat, notre objectif est de finaliser l'architecture de la plate-forme utilisée afin d'améliorer l'accessibilité et la réutilisabilité des outils de TurtleKit qui sont liés au GPGPU.

Pour atteindre cet objectif, il nous faut tout d'abord appliquer le principe de délégation sur plus de modèles multiagents. À l'heure actuelle, nous n'avons pas encore assez de recul sur son utilisation et donc sur la meilleure façon de l'implémenter. Ceci nous permettra par ailleurs d'évaluer expérimentalement la capacité de notre approche à produire des éléments facilement réutilisables dans d'autres simulations et par là ses avantages et ses limites.

Enfin, nous pensons à élaborer une bibliothèque de dynamiques environnementales que nous construirons de manière itérative et modulaire. Par exemple, les modules GPU (diffusion, gradients, average) ne sont pas liés à un modèle multiagent spécifique. Ces modules sont donc suffisamment génériques pour pouvoir être utilisés dans d'autres modèles. Ici, notre objectif sera de rendre le GPGPU accessible au plus grand nombre en facilitant l'implémentation de simulations grâce aux dynamiques environnementales que nous aurons prédéfinies.

À moyen terme, il est important de se concentrer sur le support et l'accompagnement autour du principe de délégation. Ainsi, notre prochain objectif est de définir une méthodologie basée sur le principe de délégation GPU et adaptée à l'utilisation du GPGPU dans le contexte des simulations multiagents. Une méthodologie explicite de conception tel un guide de développement qui consistera à rendre l'utilisation de la *délégation GPU* plus explicite et plus accessible à des utilisateurs externes. Cette méthodologie permettra de faciliter l'adaptation d'un modèle afin qu'il puisse profiter de la puissance du GPGPU.

L'accessibilité représente aujourd'hui un verrou majeur pour l'adoption du GPGPU dans les simulations multiagents et nous pensons qu'une telle contribution constituerait un pas important vers cet objectif et vers une plus grande diffusion de cette technologie dans la communauté multiagent.

Bibliographie

Badeig F., Balbo F. (2012). Définition d'un cadre de conception et d'exécution pour la simulation multi-agent. *Revue d'Intelligence Artificielle*, vol. 26, n° 3, p. 255-280. Consulté sur http://dx.doi.org/10.3166/ria.26.255-280

Beurier G., Simonin O., Ferber J. (2003). Un modèle de système multi-agent pour l'émergence multi-niveau. *Technique et Science Informatiques*, vol. 22, n° 4, p. 235–247. Consulté sur http://tsi.revuesonline.com/article.jsp?articleId=4795

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

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

Grignard A., Taillandier P., Gaudou B., Vo D., Huynh N., Drogoul A. (2013). GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In G. Boella, E. Elkind, B. Savarimuthu, F. Dignum, M. Purvis (Eds.), *Prima 2013: Principles and practice of multi-agent systems*, vol. 8291, p. 117-131. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-44927-7_9

Hermellin E., Michel F., Ferber J. (2015). État de l'art sur les simulations multi-agents et le GPGPU. *Revue d'Intelligence Artificielle*, vol. 29, n° 3-4, p. 425–451. Consulté sur http://dx.doi.org/10.3166/ria.29.425-451

Kubera Y., Mathieu P., Picault S. (2011). Ioda: an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, vol. 23, n° 3, p. 303-343. Consulté sur http://dx.doi.org/10.1007/s10458-010-9164-z

Laville G., Mazouzi K., Lang C., Marilleau N., Herrmann B., Philippe L. (2014). MCMAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation. In D. an Mey *et al*. (Eds.), *Euro-par 2013: Parallel processing workshops*, vol. 8374, p. 544-554. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-54420-0_53

Laville G., Mazouzi K., Lang C., Marilleau N., Philippe L. (2012). Using GPU for Multi-agent Multi-scale Simulations. In *Distributed computing and artificial intelligence*, vol. 151, p. 197-204. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-28765-7_23

Luke S., Cioffi-Revilla C., Panait L., Sullivan K., Balan G. (2005). MASON: A Multiagent Simulation Environment. *Simulation*, vol. 81, n° 7, p. 517–527. Consulté sur http://dx.doi.org/10.1177/0037549705058073

Lysenko M., D'Souza R. M. (2008). A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation*, vol. 11, n° 4, p. 10. Consulté sur http://jasss.soc.surrey.ac.uk/11/4/10.html

Michel F. (2014). 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*, vol. 28, n° 4, p. 485–510. Consulté sur http://dx.doi.org/10.3166/ria.28.485-510

Michel F. (2015). *Approches environnement-centrées pour la simulation de systèmes multi-agents. pour un déplacement de la complexité des agents vers l'environnement*. Habilitation à diriger des recherches, Université de Montpellier.

Michel F., Beurier G., Ferber J. (2005, november). The TurtleKit Simulation Platform: Application to Complex Systems. In A. Akono, E. Tonyé, A. Dipanda, K. Yétongnon (Eds.), *Workshops Sessions 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*, p. 122-128. IEEE.

Michel F., Ferber J., Drogoul A. (2009, 3 June). Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective. In Adelinde Uhrmacher, Danny Weyns (Eds.), *Multi-Agent Systems: Simulation and Applications*, p. 3–52. CRC Press - Taylor & Francis. Consulté sur http://www.crcpress.com/product/isbn/9781420070231

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

Owens J. D., Luebke D., Govindaraju N., Harris M., Kruger J., Lefohn A. E. *et al*. (2007). A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, vol. 26, n° 1, p. 80-113. Consulté sur http://www.ingentaconnect.com/content/bpl/cgf/2007/00000026/00000001/art00009

Parry H., Bithell M. (2012). Large scale agent-based modelling: A review and guidelines for model scaling. In A. J. Heppenstall, A. T. Crooks, L. M. See, M. Batty (Eds.), *Agent-based models of geographical systems*, p. 271-308. Springer Netherlands. Consulté sur http://dx.doi.org/10.1007/978-90-481-8927-4_14

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

Resnick M. (1996). StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking. In *Conference companion on human factors in computing systems*, p. 11-12. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/257089.257095

Reynolds C. W. (1987). Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th annual conference on computer graphics and interactive techniques*, vol. 21, p. 25-34. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/37401.37406

Ricci A., Piunti M., Viroli M. (2011). Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, vol. 23, n° 2, p. 158-192. Consulté sur http://dx.doi.org/10.1007/s10458-010-9140-7

Richmond P., Walker D., Coakley S., Romano D. M. (2010). High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, vol. 11, n° 3, p. 334-47. Consulté sur http://bib.oxfordjournals.org/content/11/3/334.short

Sano Y., Kadon Y., Fukuta N. (2014). A Performance Optimization Support Framework for GPU-based Traffic Simulations with Negotiating Agents. In *Proceedings of the 2014 seventh international workshop on agent-based complex automated negotiations.*

Sklar E. (2007). NetLogo, a Multi-agent Simulation Environment. *Artificial Life*, vol. 13, n° 3, p. 303-311.

Vigueras G., Orduña J., Lozano M. (2010). A GPU-Based Multi-agent System for Real-Time Simulations. In Y. Demazeau, F. Dignum, J. Corchado, J. Pérez (Eds.), *Advances in practical applications of agents and multiagent systems*, vol. 70, p. 15-24. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-12384-9_3

Viroli M., Omicini A., Ricci A. (2006). Engineering MAS environment with artifacts. In D. Weyns, H. Dyke Parunak, F. Michel (Eds.), *Environments for multi-agent systems*, vol. 3830, p. 62-77. Springer Berlin Heidelberg.

Weyns D., Dyke Parunak H., Michel F., Holvoet T., Ferber J. (2005). Environments for Multiagent Systems State-of-the-Art and Research Challenges. In D. Weyns, H. Dyke Parunak, F. Michel (Eds.), *Environments for multi-agent systems*, vol. 3374, p. 1-47. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-540-32259-7_1

Weyns D., Holvoet T. (2005). On the Role of Environments in Multiagent Systems. In *First international workshop, e4mas 2004*, p. 127-141. Springer.

Weyns D., Michel F. (2015). *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* (vol. 9068). Springer.