Compas'2016 : Parallélisme / Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Gestion efficace des ressources mémoire et de calcul pour l'exécution de systèmes multi-agents sur architectures parallèles avec OpenCL

Anne Jeannin-Girardon, Vincent Rodin

Université de Bretagne Occidentale,
Lab-STICC UMR 6285 - Dpt Informatique - 20 avenue le Gorgeu - 29200 Brest - France
contact@jeannin-girardon.net vincent.rodin@univ-brest.fr

Résumé
Des plate-formes de programmation comme OpenCL permettent de réaliser des implémentations parallèles de modèles pouvant s'exécuter sur différentes architectures matérielles. Cependant, certaines contraintes doivent être gérées lorsque l'on utilise un tel framework. En particulier, OpenCL ne permet pas d'allouer dynamiquement de la mémoire à l'exécution. Dans le cas de la simulation de systèmes multi-agents, présentant une dynamicité importante en matière de taille de population avec des entités apparaissant et disparaissant, il est nécessaire de gérer efficacement la quantité fixe de ressources disponibles durant la simulation. Nous proposons une méthode de gestion de ces ressources basée sur un mécanisme de double tampon. Ces tampons sont des tableaux contenant les identifiants des ressources disponibles pour créer une entité durant l'exécution d'un système multi-agents. Cette méthode est simple en matière d'implémentation et efficace en matière d'exécution, comme nous le montrons à travers une étude de performances réalisée à l'aide d'un système proies/prédateurs nécessitant en permanance la création et la destruction d'entités.

Mots-clés : systèmes multi-agents, simulation, parallélisme, OpenCL, gestion de ressources.

1. Introduction

La possibilité de programmer des architectures aussi variées que des micro processeurs multicœurs, des processeurs graphiques ou encore des FPGAs a contribué à l'essor de plate-formes de programmation de systèmes hétérogènes comme OpenCL [7]. Les CPU (Central Processing Units) multi-cœurs et les GPU (Graphics Processing Units) sont des matériels largement accessibles. En outre, les GPU ne sont plus uniquement dédiés au rendu graphique mais permettent également de faire du calcul généraliste, en particulier scientifique : le GPGPU (General Purpose computing on Graphics Processing Units). D'autres plate-formes de programmation permettent de programmer les GPU spécifiquement, par exemple CUDA [12]. Nous nous sommes tournés vers OpenCL car il s'agit d'un standard, ouvert et maintenu par un consortium important d'acteurs académiques et des nouvelles technologies, qui permet en outre de programmer une grande diversité de matériels.

Dans le domaine de la simulation, les systèmes multi-agents (SMA) [3] font partie des principaux formalismes de modélisation et sont utilisés dans de nombreux champs d'application (biologie moléculaire [1], biologie du développement [6], trafic routier [2], mouvements de foule [16], etc.) et les SMA exhibent un certain degré de parallélisme de manière inhérente

(exécution parallèle des agents par exemple). Cependant, la complexité des SMA rend difficile la simulation de phénomènes modélisés par ce formalisme : un nombre important d'entités peut composer le système (possiblement des millions) et les interactions de ces entités entre elles et avec l'environnement dans lequel elles évoluent ne fait qu'ajouter à la complexité globale du système. Ces deux éléments ont de fortes répercussions sur la gestion des ressources mémoire et de calcul lors de l'implémentation de tels systèmes sur des matériels présentant des contraintes comme les GPU : en particulier, il n'est actuellement pas possible, avec OpenCL, de réaliser de l'allocation dynamique de mémoire afin de prendre en compte la dynamicité inhérente des populations d'agents en matière de création et de destruction de ces agents. Partant d'une quantité fixe de mémoire et de ressources de calcul, il est nécessaire de gérer efficacement ces ressources en cours de simulation afin de permettre la création de nouveaux agents et le « recyclage » des ressources qui étaient utilisées par un agent détruit.

Dans cet article, nous proposons une méthode pour la gestion des ressources lors de l'exécution de systèmes multi-agents implémentés avec OpenCL. Cette méthode se veut simple en matière de conception et d'implémentation tout en offrant de bonnes performances à l'exécution du système. La seconde partie précise le contexte dans lequel nous nous situons à travers une présentation des méthodes existantes pour l'implémentation de SMA et la gestion des ressources associées lors de l'utilisation de matériels et plate-formes d'implémentation contraignants. Dans la troisième partie, nous décrivons tout d'abord la méthode que nous avons conçue puis nous y présentons une étude de performance de notre méthode utilisée dans l'implémentation multi-agents d'un système proies/prédateurs car on observe, dans ce type de système, des créations et destructions permanentes d'entités. Finalement, nous concluons par un bilan de notre méthode.

2. Implémentation de SMA avec OpenCL et gestion des ressources

La gestion des ressources mémoire et de calcul est un point peu abordé dans la littérature traitant de l'implémentation de systèmes multi-agents. Les travaux traitant de l'implémentation parallèle de SMA sont davantage des marches à suivre généralistes [13], des propositions de plate-formes de simulations génériques [14], des propositions visant à adapter des plate-formes de simulation existantes (diffusion ou perception d'agents implémentées en parallèle) [11] ou encore des propositions de briques logicielles constituant des bibliothèques génériques pour la mise au point de simulations multi-agents en parallèle [9].

Précisons que notre problématique n'est pas l'allocation mémoire au sens strict telle qu'on la réalise, par exemple, en C au moyen de fonctions comme malloc ou realloc : dans le cas de la programmation GPU, la plate-forme CUDA offre aujourd'hui la possibilité de réaliser ce type d'allocation. Cependant cette plate-forme est dédiée à la programmation de matériel NVidia exclusivement et l'impossibilité d'utiliser ces programmes sur différents types de matériel est restrictive. On peut néanmoins raisonnablement penser que l'évolution de OpenCL va rendre possible, dans le futur, le fait de réaliser de l'allocation dynamique. D'ici là, nous nous situons dans un contexte où nous souhaitons pouvoir gérer une quantité fixe de ressources : pouvoir utiliser les ressources disponibles est indispensable dans le cas, par exemple, de la modélisation de tissus biologiques à cause de la dynamicité des populations de cellules dans ces systèmes. Pour autant, il n'est pas nécessaire de disposer d'une quantité illimitée de ressources : toujours dans le cas de la modélisation de développement de tissus biologiques, la croissance d'un tissus est régulée et le nombre de morts cellulaires et de divisions est équilibré (homéostasie). L'apparition et la disparition de cellules doit cependant être possible si des ressources sont disponibles. Sans entrer dans le détail de l'utilisation de la plate-forme OpenCL (dont on peut trouver plus d'information dans un ouvrage tel que celui de Gaster et coll. [4]), précisons

FIGURE 1 – Fragmentation mémoire lors d'une simulation. Les cases noires représentent des entités actives, les cases blanches des entités non-actives. (a) En début de simulation, les entités actives sont stockées de manière contiguës en mémoire. (b) Au fur et à mesure que les pas de simulation s'enchaînent, des entités meurent ou se répliquent et les cases blanches et noires se « mélangent » dans la mémoire : on parle alors de fragmentation.

qu'un programme OpenCL se décompose en deux parties principales : le code hôte et le code cible. Le code hôte permet en particulier la gestion du matériel, l'allocation de buffers mémoire sur la plate-forme d'exécution cible et l'ordonnancement des tâches à exécuter. Le code cible constitue l'implémentation parallèle du modèle lui-même.

Un certain nombre de work-items (threads) vont exécuter en parallèle le code cible : nous associons un agent à un work-item (un work-item étant équivalent à une ressource de calcul) et des données sont associées à ce work-item permettant de stocker des informations relatives aux agents, comme leur position dans l'environnement ou encore leur état (actif / non-actif). Les ressources associées aux agents non-actifs sont donc disponibles et peuvent être allouées pour un agent nouvellement créé. Une simulation peut contenir au plus N agents, à déterminer selon le type de modèle à implémenter, et ce sont donc N work-items qui sont instanciés. Une structure de tableaux de tailles N permet de stocker les informations relatives aux agents. La coalescence des accès mémoire est assurée par le fait que le work-item i accède aux éléments i des tableaux contenus dans la structure. Plus de détails concernant les structures de données de notre modèle sont accessibles dans la référence [5]. Le problème se posant lors de la simulation d'un SMA ayant une population dynamique est que de la fragmentation mémoire va apparaître, tel qu'illustré sur la figure 1. Le problème est de trouver un work-item inactif lors de la création d'une nouvelle entité. La méthode intuitive de rechercher une ressource disponible dans un tableau de ressources fragmenté est trop coûteuse et il convient de mettre au point une méthode plus adaptée au problème.

A notre connaissance, trois manières de gérer cette quantité fixe de ressources ont été proposées. Une première solution est de trier les tableaux de ressources, comme suggéré dans [15] : les tableaux de ressources sont défragmentés à une certaine fréquence. Cette opération n'est pas realisée en parallèle par plusieurs work-items mais par un work-item unique. Nous avons proposé dans [6] une amélioration de cette méthode en utilisant un algorithme de tri parallèle [8] de complexité $O(n \log n)$ afin de limiter une dégradation des performances liée à une défragmentation séquentielle des tableaux de ressources. La prochaine ressource disponible est accessible par un pointeur sur cette ressource qui est incrémenté lorsque la ressource est allouée. Cette opération nécessite l'utilisation d'une opération atomique car un seul agent doit pouvoir accéder à une ressource. Les tableaux de ressources sont défragmentés lorsque le pointeur atteint la fin des tableaux. Une autre méthode consiste à utiliser un allocateur randomisé [10]. Cet allocateur ne dépend pas de l'état de la mémoire : celle-ci ne nécessite pas d'être défragmentée. Il n'est pas non plus requis de faire appel à des mécanismes de synchronisation. Il s'agit de mettre en correspondance des entités souhaitant se répliquer avec des ressources disponibles dans les tableaux de ressources. Une fonction réciproque est alors utilisée : $f : A \to A$, où $A = \{a_0, a_1, \dots, a_{N-1}\}$ est l'ensemble de toutes les entités : $f(a_i) = a_{i+r}$ et $f(a_i)^{-1} = a_{i-r}$ où r est un entier aléatoire compris entre 1 et N global à toutes les entités : ce faisant, chaque correspondance entre une entité et une ressource est unique. Toutefois, une mise en correspondance a d'autant moins de chance de se produire que les tableaux de ressources se remplissent. Réaliser plusieurs itérations de l'algorithme permet de pallier dans une certaine mesure cet inconvénient sans pour autant garantir qu'une entité pourra se répliquer alors même que des

ressources sont disponibles. En outre, le fait de réaliser plusieurs itérations nécessite des vas et viens entre l'hôte et la cible : r étant global à tous les agents, les itérations sont réalisées dans le code hôte (r est généré puis, en parallèle, les agents tentent d'allouer une ressource). Souhaitant pouvoir bénéficier de l'efficacité de cet allocateur randomisé tout en assurant que les agents de la simulation peuvent se répliquer si des ressources sont disponibles, nous avons proposé dans [5] d'utiliser conjointement la défragmentation des tableaux de ressources et un allocateur randomisé. Nous passons d'une méthode à l'autre en fonction du remplissage des tableaux de ressources : lorsque moins de 70% des ressources sont disponibles, l'allocateur randomisé est utilisé (au maximum 10 itérations sont réalisables par les agents pour tenter d'allouer une ressource). Passé 90%, les tableaux de ressources sont défragmentés et un pointeur sur la prochaine ressource disponible est utilisé (utiliser un intervalle pour passer d'une méthode à l'autre permet de limiter le nombre d'échanges d'une méthode à l'autre). Il est à noter que les bornes de cet intervalle ont été déterminées empiriquement [6].
Désirant améliorer les mécanismes de réplication des agents de notre modèle, nous proposons dans cet article une méthode simple (en matière de conception et d'implémentation) et efficace (en matière de temps d'exécution) reposant sur l'utilisation d'un double tampon pour stocker les identifiants de work-items non-actifs. Nous décrivons cette méthode dans la partie suivante.

3. Proposition : gestion simple et efficace des ressources de calculs pour SMA avec OpenCL

En premier lieu, nous décrivons ici la méthode que nous proposons pour l'allocation de ressources par des entités souhaitant se répliquer. Puis, nous présenterons une comparaison de cette méthode par rapport à l'allocateur hybride que nous avions précédemment proposé [5].

3.1. Description de la méthode

Au maximum N entités peuvent être allouées lors d'une simulation. Initialement, n entités sont effectivement allouées, c'est-à-dire dans un état actif. Les données relatives aux entités sont toujours stockées dans une structure de tableaux de tailles N, chaque entité identifiée par i (qui, pour rappel, est également l'identifiant du work-item exécutant le comportement de l'entité) et accède à ses données à la position i des tableaux. Deux tableaux supplémentaires de taille N sont utilisés pour stocker les identifiants des entités non-actives ; autrement dit les identifiants accessibles pour la création de nouvelles entités.
Les tableaux peuvent contenir des identifiants valides $0 \leq i < N$ ou invalides $i = -1$. Le tableau `Alloc` est utilisé par les entités cherchant à allouer des ressources. Le tableau `Lib` est utilisé par les entités devenant inactives : elles y inscrivent leur identifiant qui redevient donc accessible pour une nouvelle entité. Trois pointeurs permettent de naviguer dans ces tableaux : `ptrAlloc` permet d'accéder au prochain identifiant disponible ; `ptrLib` permet à une entité de remettre son identifiant avant de devenir inactive ; `ptrLibPrec` est utilisé pour gérer les tableaux (quel tableau est utilisé pour récupérer un identifiant et quel tableau est utilisé pour remettre un identifiant ; il s'agit là du point central de notre méthodologie). Précisons celle-ci par l'exemple illustré sur la figure 2. Imaginons un cas de figure dans lequel trois entités identifiées par 0, 1 et 2, sont actives dans le système. (a) Deux de ces entités souhaitent se répliquer et ont récupéré les identifiants 3 et 4 qui ont été remplacés par `-1` dans le tableau `Alloc`. Le pointeur `ptrAlloc` a été déplacé de deux cases. Dans le même temps, l'entité 3 est devenue inactive et a donc remis son identifiant dans le tableau `Lib`, à la case pointée par `ptrLib`. Ce pointeur a été déplacé sur la case suivante. (b) la simulation se poursuit et tous les identifiants disponibles dans le tableau `Alloc` ont été alloués. Cependant, un identifiant est disponible dans le tableau `Lib` : on va alors échanger les deux tableaux. Le tableaux `Lib` va devenir le tableau `Alloc` accédé par les entités voulant se répliquer. Dans la suite, nous détaillons les opérations à réaliser pour utiliser ce mécanisme d'allocation.

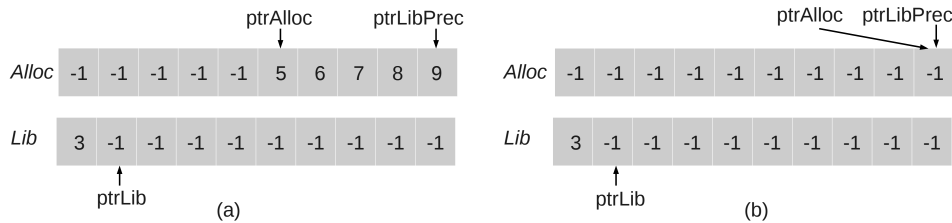

FIGURE 2 – Exemple d'utilisation des tableaux `Alloc` et `Lib`. (a) Les entités voulant se répliquer peuvent accéder à des identifiants disponibles grâce au pointeur `ptrAlloc`. Les entités devenant non-actives remettent leur identifiants dans `Lib` grâce au pointeur `ptrLib`. (b) Lorsqu'il n'y a plus d'identifiant disponibles dans `Alloc`, les deux tableaux sont échangés, `Lib` devient `Alloc` et les entités peuvent accéder aux éventuels identifiants précédemment remis. `ptrLibPrec` est utilisé pour la gestion des tableaux.

Notons que l'utilisation d'un buffer circulaire pour la gestion des ressources n'est pas possible à cause de l'impossibilité d'utiliser des sections critiques de type mutex sur GPU avec OpenCL. En outre les opérations atomiques que nous utilisons vont toujours modifier les pointeurs et le succès et l'échec de l'allocation sont vérifiées avec la valeur de retour de ces opérations atomiques. Dit autrement, à un instant donné dans le programme, il est impossible de savoir quelles sont les valeurs des pointeurs. Utiliser un buffer circulaire nécessiterait de replacer le pointeur sur sa valeur d'origine en cas d'échec. Quant à l'utilisation d'une liste pour gérer la distribution de ressources, s'il est vrai que la gestion de cette liste peut être réalisée sur l'hôte, la méthode proposée dans cet article vise à minimiser la gestion des ressources sur l'hôte.

Allocation et remise d'identifiants L'accès aux pointeurs `ptrAlloc` et `ptrLib` doit être réalisé en section critique pour assurer l'unicité de l'identification de chaque entité. OpenCL fournit un ensemble de fonctions pour réaliser des opérations atomiques sur des variables partagées. L'opération atomique nous intéressant incrémente une variable partagée et retourne la valeur de cette variable avant incrémentation : `res = atomic_int(var)`. L'allocation ou la remise d'un identifiant sont réalisées avec les algorithmes suivants :

```
/* *** Allocation *** */
ptr = atomic_inc(ptrAlloc)
si ptr < N et Alloc[ptr] != -1 alors
  identifiant = Alloc[ptr]
  Alloc[ptr] = -1
  retourner identifiant
fsi
```

```
/* *** Remise *** */
ptr = atomic_inc(ptrLib)
Lib[ptr] = identifiant
```

Le fait que, d'une part, les pointeurs ne se déplacent que dans un seul sens et que, d'autre part, l'échange de tableaux – et donc la remise à zéro des pointeurs – soit réalisé sur l'hôte garantit que seule une entité va pouvoir écrire dans `Alloc[ptrAlloc-1]` ou `Lib[ptrLib-1]`.

Conditions d'échange des tableaux `Alloc` et `Lib` Trois conditions permettent de déterminer s'il est nécessaire d'échanger les deux tableaux : si `ptrAlloc` est supérieur au nombre maximal d'entités qu'il est possible d'allouer N ; si une entité n'a pu récupérer un identifiant valide et a initialisée un booléen pour le signifier. Ce cas peut se produire lorsqu'il n'y a plus d'identifiant disponible mais pendant que cette entité a tenté de récupérer un identifiant, d'autres ont pu devenir inactives et remettre des identifiants dans `Lib`. Enfin, si la valeur pointée par `ptrLibPrec` vaut `-1`. Ce pointeur est utilisé par l'hôte pour déterminer s'il est nécessaire d'échanger les tableaux : si `Alloc[ptrLibPrec-1]` vaut `-1` alors il n'y a plus d'identifiant disponible (`ptrLibPrec` permet également de garder le compte des entités

actives dans le système en calculant le nombre d'identifiants disponibles : `ptrLibPrec - ptrAlloc + ptrLib`; `ptrLib` est assigné à `ptrLibPrec` à chaque pas de simulation et lorsque les tableaux sont échangés). Dans la section suivante, nous proposons une étude des performances de notre mécanisme d'allocation de ressources, comparée à la méthode hybride que nous avions précédemment proposée dans [5].

3.2. Performances du système

Protocole expérimental Notre modèle est composé de deux types d'entités, des proies et des prédateurs, dont les comportements sont modélisés de la façon suivante : les deux types se déplacent de manière aléatoire dans l'environnement discret via un voisinage de von Neumann. Si un prédateur a pour voisin immédiat une proie, celle-ci est consommée et le prédateur voit son énergie augmenter. Cette énergie lui permet de se reproduire (la reproduction n'est pas possible si l'énergie est insuffisante ; le prédateur meurt si son énergie baisse jusqu'à un seuil critique). Nous considérons que les proies ont accès à des ressources de nourritures non limitées et peuvent donc se reproduire indépendamment d'une énergie. Les deux types d'entités meurent passé un certain âge.

Notre système multi-agents est suffisament robuste (tailles et de fluctuations des populations) pour permettre d'observer des dynamiques similaires dans l'évolution des populations indépendamment de la méthode de gestion de ressources utilisée. Pour chaque méthode de gestion de ressources, nous présentons deux jeux de tests (figure 3) réalisés en variant le nombre maximal de ressources disponibles et la taille de l'environnement (l'environnement est discret, découpé en cases et, dans le cas présent, sans métrique) : d'une part, le nombre d'entités maximal est fixé à 65 536 dans un environnement de 512*512 (figures 3 a et 3 b) et, d'autre part, le nombre d'entités maximal est fixé à 131 072 dans un environnement de 1024*1024 (figures 3 c et 3 d).

Les quantités de ressources maximales ont été choisies de manière à être des puissances de deux. Les mesures présentées dans la suite sont extraites d'une simulation de 200 000 pas.

Cette simulation a été réalisée sur un processeur Intel Core i7 4790. Si le choix de l'architecture matériel à une influence sur l'exécution du modèle, l'utilisation du framework OpenCL sous-entend une volonté de ne pas prendre en compte l'architecture sous-jacente et de proposer une méthode généraliste.

Résultats Lors de l'utilisation de la méthode hybride, sur les courbes (a) et (c) on observe clairement que le goulot d'étranglement en matière de performance se situe lorsque la mémoire doit être défragmentée (autrement dit lorsque l'occupation mémoire est supérieure à 90%). C'est encore plus flagrant dans la courbe (c) où la population sature : en permanence à la limite des 131 072 agents possibles, la mémoire est défragmentée de façon quasiment systématique et le temps d'exécution total du système se situe entre 35 et 40 ms par pas de simulation. Dans le cas de l'utilisation du double tampon, les temps d'exécution sont réguliers indépendamment de la dynamique de la population (on note que les temps d'exécution des deux méthodes sont similaires lorsque la taille des populations ne dépasse pas 90% du nombre maximal d'entités possible). L'utilisation du double tampon est donc plus avantageuse car cela permet de ne pas ralentir l'exécution du modèle à cause d'une saturation de la population. Il est à noter que l'overhead mémoire est limité à l'utilisation d'un tableau de N*4 octets supplémentaire pour le stockage des identifiants allouables par les entités.

4. Conclusion

Nous avons présenté, dans cet article, une méthode simple et efficace permettant la gestion des ressources mémoire et de calcul lors de l'exécution de systèmes multi-agents avec OpenCL. OpenCL ne permettant actuellement pas de réaliser une allocation dynamique de ressources, il

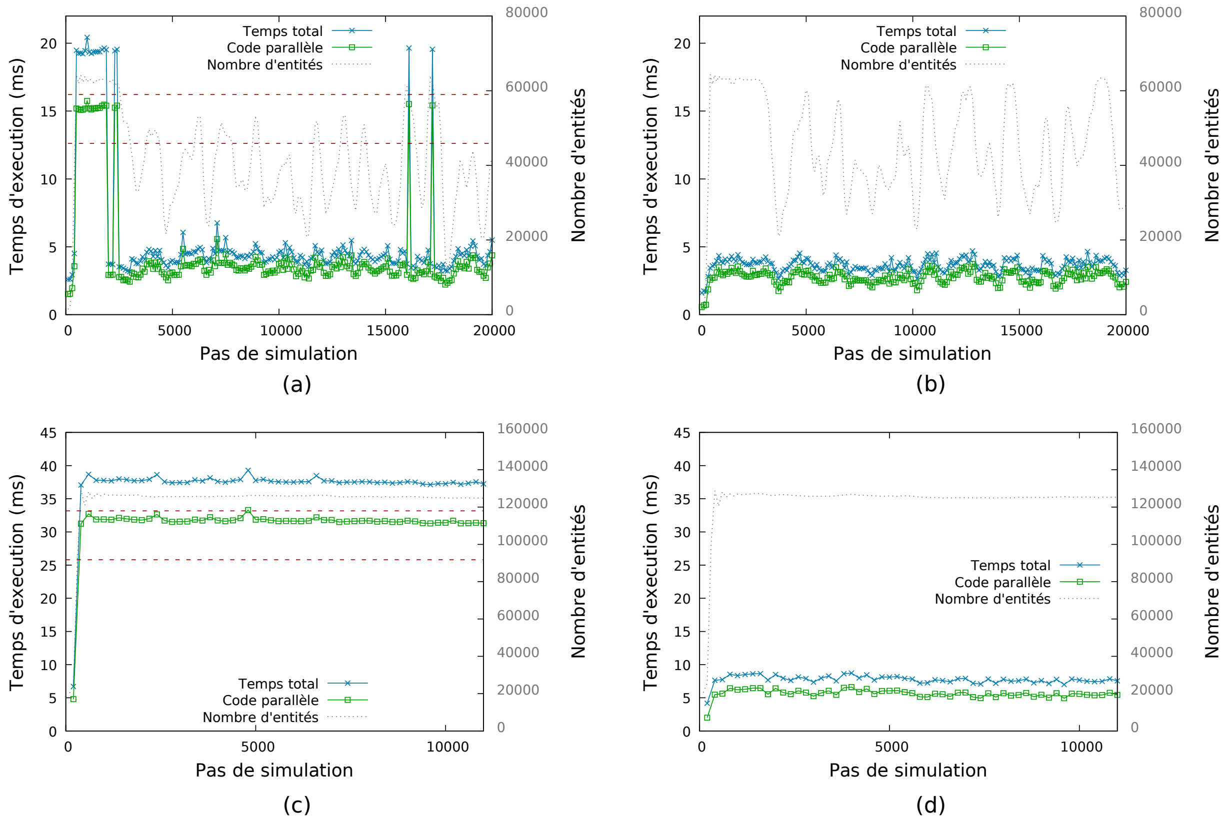

FIGURE 3 – Comparaison des performances de notre méthode hybride (figures (a) et (c)) et de notre méthode par double tableaux (figures (b) et (d)). Figures (a) et (b) : le nombre d'entités maximal est fixé à 65 536 ; l'environnement a pour taille 512*512. Figures (c) et (d) : le nombre d'entités maximal est fixé à 131 072 ; l'environnement a pour taille 1024*1024. Le temps total (courbe bleue) représente les temps d'exécution cumulés du code hôte et cible. Le temps d'exécution du code parallèle est représenté par les courbes vertes. Les courbes grises en pointillé sont les tailles cumulées des populations de proies et de prédateurs dont l'échelle est représentée en gris sur la droite des graphiques. Dans les courbes (a) et (c), les deux lignes rouges en pointillés indiquent la plage d'occupation mémoire dans laquelle les méthodes sont échangées.

est nécessaire de pouvoir utiliser efficacement la quantité fixe de ressources à disposition lors de la simulation d'un modèle représentant des populations aux tailles dynamiques (croissance de tissus biologiques par exemple). La méthode que nous proposons repose sur l'utilisation d'un double tampon permettant de stocker les identifiants des ressources disponibles. Un des tampons est utilisé par les entités pour récupérer des identifiants tandis que le second tampon permet à des entités de remettre l'identifiant associé à leurs ressources avant de devenir inactives. Si aucun identifiant n'est disponible dans le premier tampon, les entités vont consulter le second tampon qui contient potentiellement des identifiants disponibles. Nous avons comparé notre méthode à une méthode hybride que nous avons précédemment proposée, reposant sur l'utilisation d'un allocateur randomisé et d'une défragmentation de la mémoire. Cette méthode hybride devient peu efficace lorsque le nombre d'entités actives se rapproche du nombre maximal d'entités alors que la méthode par double tampon permet de conserver des temps d'exécution bas et réguliers indépendamment de l'état de la mémoire du système.

Le revers de l'utilisation de deux tampons pour la gestion de ressources est que deux tableaux d'entiers sont nécessaires en plus des données du systèmes. Cependant, les quantités de mémoires disponibles sont aujourd'hui très importantes alors qu'il devient rapidement difficile de

gagner en performances en matière de temps d'exécution, notamment avec des systèmes complexes parfois difficiles à paralléliser. Il nous semble donc que l'utilisation de mémoire dédiée au stockage des tampons est un compromis acceptable.

Bibliographie

1. Avellaneda (F.), Bustacara (C.), Garzon (J. P.) et Gonzalez (E.). – Implementation of a molecular simulator based on a multiagent system. – In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT'06*, pp. 117–120, Washington, DC, USA, 2006. IEEE Computer Society.
2. Bazzan (A. L. C.) et Klügl (F.). – A review on agent-based technology for traffic and transportation. *The Knowledge Engineering Review*, vol. 29, 6 2014, pp. 375–403.
3. Ferber (J.). – *Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence*. – Addison Wesley, London, 1999.
4. Gaster (B.), Howes (L.), Kaeli (D.), Mistry (P.) et Schaa (D.). – *Heterogeneous computing with OpenCL*. – Morgan Kaufmann, 2011.
5. Jeannin-Girardon (A.), Ballet (P.) et Rodin (V.). – A software architecture for multi-cellular system simulations on graphics processing units. *Acta Biotheoretica*, vol. 61, n3, 2013, pp. 317–327.
6. Jeannin-Girardon (A.), Ballet (P.) et Rodin (V.). – Large scale tissue morphogenesis simulation on heterogenous systems based on a flexible biomechanical cell model. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 12, n5, Sept 2015, pp. 1021–1033.
7. Khronos group. – The open standard for parallel programming of heterogeneous systems https ://www.khronos.org/opencl/, Accédé en février 2016.
8. Knuth (D. E.). – *The art of computer programming, volume 3 : sorting and searching*. – Redwood City, CA, USA, Addison Wesley Longman Publishing Co., Inc., 1973.
9. Laville (G.), Lang (C.), Herrmann (B.), Philippe (L.), Mazouzi (K.) et Marilleau (N.). – Mcmas : A toolkit for developing agent-based simulations on many-core architectures. *Multiagent and Grid Systems*, vol. 11, n1, 2015, pp. 15–31.
10. Lysenko (M.) et D'Souza (R. M.). – A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, vol. 11, n4, 2008, p. 10.
11. Michel (F.). – Translating agent perception computations into environmental processes in multi-agent-based simulations : A means for integrating graphics processing unit programming within usual agent-based simulation platforms. *Systems Research and Behavioral Science*, vol. 30, n6, 2013, pp. 703–715.
12. NVidia. – Parallel programming and computing platform http ://www.nvidia.com/object/cuda_home_new.html, Accédé en février 2016.
13. Parry (H.) et Bithell (M.). – Large scale agent-based modelling : A review and guidelines for model scaling. *In : Agent-Based Models of Geographical Systems*, éd. par Heppenstall (A. J.), Crooks (A. T.), See (L. M.) et Batty (M.), pp. 271–308. – Springer Netherlands, 2012.
14. Richmond (P.), Coakley (S.) et Daniela (R.). – A high performance agent based modelling framework on graphics card hardware with cuda. – In *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems, AAMAS 2009*, pp. 334–47, 2009.
15. Tripodi (S.), Ballet (P.) et Rodin (V.). – GPU implementation and performance analysis of reactive agents having division and mobility capacities. *Multiagent and Grid Systems, IOS Press*, vol. 8, n4, 2012, pp. 289–309.
16. van Toll (W. G.), Jaklin (N. S.) et Geraerts (R.). – Towards believable crowds : A generic multi-level framework for agent navigation. – In *ICT.OPEN 2015*, 2015.