Systèmes Multi-Agents et GPGPU : état des lieux et directions pour l'avenir

Emmanuel Hermellin[a] emmanuel.hermellin@lirmm.fr
Fabien Michel[a] fmichel@lirmm.fr
Jacques Ferber[a] ferber@lirmm.fr

[a]LIRMM - Laboratoire Informatique, Robotique et Micro électronique de Montpellier,
Université Montpellier 2 - CNRS, 161 Rue Ada, 34090 Montpellier, France

Résumé

Dans le domaine des systèmes multi-agents (SMA), la puissance des CPU représente parfois un frein qui limite fortement le cadre dans lequel un modèle peut être conçu et expérimenté. Le Calcul Haute Performance (HPC) et notamment le GPGPU (General-Purpose computing on Graphics Processing Units) est une solution avantageuse en termes de performance et de coût. Cette technologie vise à utiliser les GPU pour réaliser du calcul généraliste mais souffre cependant d'une très grande complexité d'implémentation et d'accessibilité qui ne favorise pas son adoption. Cet article propose un état de l'art des contributions traitant de l'utilisation du GPGPU dans le contexte des SMA et identifie les directions les plus prometteuses en vue d'une généralisation de cette technologie dans la communauté multi-agents.

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

Abstract

Using Multi-Agent Systems (MAS) may require to handle a very high computing power but the computational resources which are needed cannot be fulfilled by the CPU of single Personal Computers (PC) any more. Considering this issue, General-Purpose computing on Graphics Processing Units (GPGPU) appears to be very promising as it enables huge speed up on regular PC. However, this technology relies on a highly specialized architecture, implying a very specific programming approach and mindset. This paper reviews the literature reporting on contributions which are at the intersection between MAS and GPGPU and identifies the most promising directions for a generalization of this technology in our community.

Keywords: *MAS, GPGPU, CUDA, OpenCL*

1 Introduction

Les systèmes multi-agents (SMA) sont utilisés dans de nombreux domaines pour modéliser des systèmes très variés, pourvus que ces derniers puissent être représentés par un ensemble d'entités autonomes en interaction appelées agents. Ce type de simulation peut nécessiter une puissance de calcul très importante. De fait, les performances des processeurs classiques représente souvent un verrou majeur qui limite fortement le cadre dans lequel un modèle peut être conçu et expérimenté. Cette limitation devient parfois si importante qu'il est nécessaire de se tourner vers le HPC (*High Performance Computing*).

Dans le cadre du HPC, le GPGPU (*General-Purpose computing on Graphics Processing Units*), tient une place à part. Réelle révolution technologique, le GPGPU permet d'utiliser l'architecture massivement parallèle des cartes graphiques pour effectuer du calcul généraliste, et ainsi accélérer très significativement les performances en utilisant uniquement un ordinateur grand public. Bourgoin [4] offre, dans sa thèse, une très bonne présentation du GPGPU et donne une vision générale des possibilités, de son évolution et de son utilisation. Che [6] propose une approche plus pratique en s'intéressant à des implémentations au travers d'exemples et de comparaisons entre modèles CPU et GPU en exploration de données et en logique combinatoire. Ces références discutent aussi des limites du GPGPU en termes d'accessibilité : de par l'architecture matérielle très spécifique du GPU, ce type de programmation requiert des connaissances pointues et une façon de penser radicalement nouvelle. En particulier, le problème doit pouvoir être représenté par des structures de données distribuées et indépendantes.

Cet article propose un état de l'art des contributions traitant de l'utilisation du GPGPU pour la modélisation et l'implémentation de SMA. Comme nous le verrons, malgré les gains de performances impressionnants obtenus par certaines contributions, la jeunesse du GPGPU et son manque d'accessibilité limitent pour l'instant son utilisation aux domaines où le temps de calcul est critique. Ainsi, la grande majorité des

contributions recensées traitent de simulations multi-agents. De plus, nous verrons que la plupart de ces travaux ont été réalisés de manière ad hoc. Nous avons donc fait le choix de regrouper les contributions par catégorie technologique : utilisation des fonctions graphiques, interfaces spécialisées, approche hybride).

La section 2 introduit la notion de GPGPU et les différents outils disponibles. Les trois sections suivantes présentent des travaux à l'intersection des SMA et du GPGPU : la section 3 revient sur les premiers SMA implémentés sur GPU, le section 4 passe en revue des recherches réalisées à l'aide des interfaces de programmation spécialisées et la section 5 se focalise sur les systèmes hybrides utilisant à la fois le CPU et le GPU. La section 6 analyse ensuite les contributions recensées selon deux critères : (1) la généricité du modèle agent et (2) l'accessibilité du framework proposé. Enfin, la section 7 conclut cet article et donne des pistes de recherche pour l'avenir.

2 Introduction au GPGPU

De par son design et son architecture, le CPU permet d'effectuer du calcul généraliste. Au contraire, les cartes graphiques ont été développées dans le but de soulager les processeurs de la charge que demandait la gestion des pixels et de l'affichage sur l'écran. Avec l'engouement pour les graphismes, les GPU ont évolué et sont maintenant composés de centaines d'ALU (*Arithmetic Logic Units*) formant une structure hautement parallèle capable de réaliser des tâches plus variées. Ces ALU ne sont pas très rapides (beaucoup moins qu'un CPU) mais permettent d'effectuer des milliers de calculs similaires de manière simultanée. Une des grandes différences entre l'architecture CPU et l'architecture GPU vient donc du nombre de processeurs (d'ALU) qui composent un GPU, bien plus important que pour un CPU (voir figure 1). Le paradigme de programmation derrière le GPGPU est basé sur le modèle de calcul parallèle SIMD (*Single Instruction Multiple Data*), il consiste en l'exécution simultanée d'une même instruction sur plusieurs données. Lorsque la structure de données s'y prête, l'architecture massivement parallèle du GPU permet d'obtenir des gains de performances très élevés (des milliers de fois plus rapide).

C'est en 1999 que le GPGPU commence à être utilisé. Il n'existait alors aucune interface de programmation spécialisée et les rares personnes intéressées détournaient donc "à la

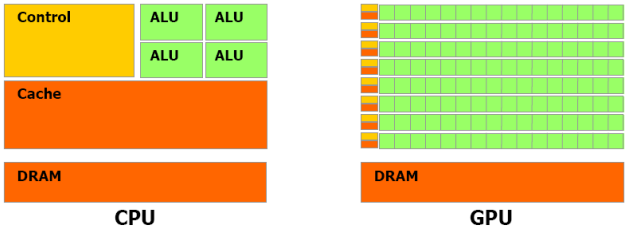

FIGURE 1 – Architecture CPU et GPU

main" les fonctionnalités graphiques de la carte pour réaliser des calculs relevant d'un autre contexte. Par exemple, il était nécessaire d'utiliser les textures graphiques comme structures de données : chaque *texel* [1] de la texture permettait de stocker 4 variables (de 8 bits) à la place des valeurs usuelles rouge, bleu, vert et alpha. Ces données étaient ensuite traitées et manipulées par des scripts, les *shaders*, exécutés par le GPU. Du fait de la complexité associée à cette utilisation non conventionnelle du GPU, celle-ci est longtemps restée réservée à une petite communauté de programmeurs.

Afin d'offrir un meilleur support et une plus grande accessibilité, c'est en 2007 que Nvidia mit à disposition la première interface de programmation dédiée au GPGPU : CUDA [2] (*Compute Unified Device Architecture*). Ainsi, CUDA fournit un environnement logiciel de haut niveau permettant aux développeurs de définir et contrôler les GPU de la marque Nvidia par le biais d'une extension du langage C. Cette solution a été suivie en 2008 par OpenCL [3] (*Open Computing Language*), un framework de développement permettant l'exécution de programmes sur des plateformes matérielles hétérogènes (un ensemble de CPU et/ou GPU sans restriction de marques). OpenCL est donc un standard ouvert, fourni par le consortium industriel Khronos Group, qui s'utilise aussi comme une extension du langage C.

La philosophie de fonctionnement de ces deux solutions reste la même (pour l'illustrer, nous avons mis en ligne sur Github des exemples simples [4]). Le CPU, appelé *host*, joue le rôle du chef d'orchestre. Il va gérer la répartition des données et exécuter les *kernels* : les fonctions spécialement créées pour s'exécuter sur le GPU, qui est lui qualifié de *device*. Le code GPU diffère donc complètement d'une approche sé-

1. Un texel est l'élément le plus petit composant une texture.
2. https://developer.nvidia.com/what-cuda
3. http://www.khronos.org/opencl
4. https://github.com/ehermellin/IntroHPC

quentielle et doit s'adapter à l'architecture matérielle de ces cartes. En effet, le GPU est capable d'exécuter un *kernel* des milliers de fois en parallèle grâce aux *threads* (les fils d'instructions). Ces *threads* sont regroupés par *blocs*, qui sont eux-mêmes rassemblés dans une *grille globale*. Chaque *thread* au sein de cette structure est identifié par des coordonnées 3D (x, y et z) lui permettant d'être localisé. De la même façon, un *bloc* possède aussi des coordonnées 3D qui lui permettent d'être identifié dans la *grille*. Les *threads* vont ainsi exécuter le même *kernel* mais vont traiter des données différentes selon leur localisation spatiale (identifiant).

CUDA et OpenCL sont actuellement les deux solutions les plus populaires pour faire du GPGPU. Il existe aussi des librairies permettant d'utiliser CUDA et OpenCL au travers d'autres langages comme Java [5] ou Python [6].

3 Premiers SMA sur GPU

SugarScape est le premier SMA qui a profité d'un portage sur carte graphique grâce au GPGPU [12]. C'est un modèle multi-agents très simple dans lequel des agents réactifs évoluent dans un environnement discrétisé en cellules contenant une ressource, le sucre, et suivent des règles comportementales basiques (déplacement, reproduction, etc.). Avec cette implémentation, la simulation *SugarScape* était exécutée entièrement sur le GPU. Il était ainsi possible d'avoir un rendu en temps réel de la simulation avec plusieurs millions d'individus, ce qui reste encore aujourd'hui quasi impossible avec les CPU actuels. La performance était d'autant plus encourageante qu'elle a été réalisée à l'aide d'un ordinateur grand public équipé d'une carte graphique comportant seulement 128 cœurs. D'un point de vue performance, cette adaptation GPU surpassait ainsi toutes les versions séquentielles de *SugarScape* tout en permettant d'avoir une population d'agents encore jamais vue dans des environnements de plus en plus larges.

Motivés par ces très bons résultats, Lysenko et D'Souza se sont ensuite attaqués au problème de l'accessibilité en généralisant leur approche et en proposant un framework destiné à faciliter le portage sur GPU de modèles similaires à SugarScape [21]. Celui-ci était ainsi composé de fonctions de bases telles que la gestion de données environnementales, la gestion des interactions entre agents, naissance, mort, etc.

5. JCUDA `www.jcuda.org` et JOCL `www.jocl.org`
6. PyCUDA et PyOpenCL `http://mathema.tician.de/`

Suivant cette tendance, le framework *ABGPU* se proposait d'être une interface de programmation un peu plus générique et surtout plus accessible grâce à une utilisation transparente du GPU reposant sur une extension du langage C/C++ basés sur un système de mots clés [28]. À cela s'ajoutaient des fonctions et des classes pré-programmées, facilitant d'avantage l'implémentation de comportements et fonctions agents complexes. *ABGPU* était capable de simuler et d'afficher un banc de poissons composé de 65536 agents en temps réel et en 3D en utilisant une carte graphique composée de 112 cœurs.

Même dans le cas d'architectures agents très simples, ces travaux pionniers mettent en évidence la difficulté de mettre en place une méthode de conversion générique pour le portage des SMA sur GPU, du fait de l'hétérogénéité des modèles. À ce propos, [25] a étudié les difficultés associées à de telles conversions en réalisant l'implémentation de différents modèles (Mood Diffusion, Game of Life et Schelling Segregation) sur CPU (avec *NetLogo* et *Repast*), puis sur GPU. Ces cas d'études montrent bien que les spécificités de la programmation sur GPU rendent difficile, voire impossible, le processus de conversion qui est bien plus complexe qu'un simple changement de langage de programmation. En effet, avec le GPGPU, de nombreux concepts présents en programmation séquentielle ne sont plus disponibles. En particulier, des caractéristiques importantes du monde orienté objet ne peuvent être utilisées (héritage, composition, etc.). Du fait de ces difficultés, le besoin en outils et interfaces spécialisés était très grand et leur apparition va permettre une explosion du GPGPU en général.

4 CUDA et OpenCL : l'essor du GPGPU

Avec l'arrivée de CUDA et OpenCL, l'utilisation du GPGPU a été grandement simplifiée et celui-ci est devenu une solution incontournable dans de nombreux domaines où le temps de calcul est critique. Ainsi, le nombre de SMA utilisant le GPGPU ne fait qu'augmenter et de nouveaux outils et frameworks ont vu le jour.

4.1 Flame GPU

Flame GPU [7] est un exemple emblématique de ce qu'a permis l'émergence des outils spécia-

7. `http://www.flamegpu.com/`

lisés. En effet, il est une solution clef en main pour la création de simulations multi-agents sur GPU qui possède plusieurs avantages. Tout d'abord, *Flame GPU* se focalise sur l'accessibilité en s'appuyant sur une utilisation transparente du GPGPU. Pour cela, il utilise les *X-machines* [7] : un formalisme de représentation d'agents s'appuyant sur une extension du langage XML : le XMML. Ainsi, les états des agents sont implémentés dans des fichiers XMML pendant que leurs comportements sont programmés en C. Le code GPU est quant à lui généré grâce aux fichiers XMML combinés dans des templates GPUXMML puis vérifiés par des processeurs XSLT.

De plus, *Flame GPU* fournit un framework open source contenant des modèles agents pré-programmés facilement réutilisables. Ainsi, *Flame GPU* peut être utilisé pour simuler un large éventail de modèles dans des contextes différents : en biologie, e.g. simulation de cellules de peau [30, 27], en intelligence artificielle, e.g. simulation proie - prédateur [26], ou pour des simulations de foules e.g. [18].

Bien que les avancées apportées par *Flame GPU* en termes d'accessibilité et de généricité soient remarquables, la solution proposée par ce framework nécessite d'adhérer à une modélisation peu intuitive basée sur XML. De plus les abstractions utilisées pour cacher la complexité du GPGPU réduisent naturellement les performances. C'est pourquoi, comme nous allons maintenant le voir, beaucoup de travaux existants repartent de zéro et focalisent toute leur attention sur les gains de performances.

4.2 Les simulations de flocking

Motivée par la possibilité de simuler un grand nombre d'agents, la simulation de modèles de type flocking est un domaine qui s'est rapidement tourné vers le GPGPU et a proposé plusieurs contributions significatives. On peut notamment citer [23], la première simulation de flocking sur GPU. Celle-ci fût suivie par le travail de Li qui ajoute la notion d'évitement d'obstacles dans son modèle [20].

L'article d'Erra [13] est très intéressant car il propose une description complète et détaillée des étapes suivies pour implémenter un modèle sur GPU. Ce travail offre une vision globale de la faisabilité et des performances que l'on peut obtenir en utilisant le GPGPU pour des modèles de flocking. Dans ces simulations, des millions d'individus sont rendus en temps réel et en 3D par une carte graphique comportant 128 cœurs.

Dans [8], pour accélérer les calculs et le rendu des simulations de flocking, Silva propose un nouveau modèle intégrant une technique de *self-occlusion*. L'idée proposée est que chaque agent est plus ou moins visible en fonction de sa distance avec l'agent sur lequel on se focalise. De plus, Silva présente aussi une comparaison entre deux implémentations de son travail : l'une utilisant directement les fonctions graphiques de la carte et l'autre basée sur CUDA. Les résultats obtenus montrent que, même dans le cas de CUDA, abstraire la couche matérielle ne peut se faire qu'au détriment des performances : le modèle utilisant directement les fonctions graphiques du GPU reste plus rapide.

Finalement, [17] propose un modèle de flocking sur GPU plus complexe capable de simuler un environnement comportant plusieurs espèces d'entités différentes. Ainsi, ces travaux font partie des premiers à intégrer la notion d'hétérogénéité. En effet, cette contribution ajoute au modèle de flocking de Reynolds une nouvelle règle (le but, chaque agent va avoir une position à atteindre) et la possibilité de donner à chaque espèce différente une "personnalité" (caractérisée par des paramètres propres aux agents). L'étude de ce système va être rendue possible grâce à la puissance de calcul offerte par le GPGPU et va ainsi permettre de voir apparaître de nouveaux comportements émergents notamment le *flock separation behaviour*. L'implémentation de ce modèle a été testée sur 5 cartes graphiques différentes (de la carte d'entrée de gamme comportant 192 cœurs jusqu'à la carte professionnelle contenant 512 cœurs) et les résultats montrent un temps de calcul et de rendu 3D par image entre 0.08 secondes et 0.14 secondes pour environ 37000 entités simulées.

4.3 Les simulations de foules et de trafics

Les simulations de foules et de trafics font aussi partie des domaines pour lesquels il est pertinent d'étudier des environnements et des populations d'agents toujours plus grands. Ainsi, les gains de performances offerts par le GPGPU sont clairement visibles dans des travaux comme [33] et [32] sur les simulations de trafics.

Évolution du framework *ABGPU* (cf. section 3), le *Pedestrian framework* [29], basé sur CUDA, ne se focalise pas seulement sur la performance. En effet, il propose de modéliser et de simuler

des agents cognitifs pouvant utiliser les forces sociales d'Helbing décrites dans [15]. Pour cela, [29] est la première contribution à distinguer explicitement agent et environnement, ce dernier étant chargé de représenter des forces environnementales virtuelles qui attirent les agents vers des points d'intérêt (vitrines de magasins, événements spéciaux, etc.). En utilisant ce framework, il est possible de simuler 65536 agents en 3D et en temps réel avec une carte graphique contenant seulement 96 cœurs.

Avec le nombre important de travaux exploitant le GPGPU dans le domaine des simulations de foules ou de trafics, la question de la réutilisation et de la généralisation s'est posée. Dans ce cadre, la navigation autonome et la planification de chemin ont été rapidement identifiées comme des fonctions couramment utilisées. De plus, de par leur aspect distribué, ces algorithmes s'adaptent très bien aux architectures massivement parallèles et de très gros gains de performances peuvent être obtenus. Bleiweiss [2] est le premier à proposer une implémentation des algorithmes *A** et *Dijkstra* sur GPU. Ceux-ci seront ensuite modifiés par Caggianese et Erra afin de s'adapter en temps réel tout au long de la simulation [5]. Santos [10] apporte aussi sa contribution avec un cas d'étude proposant une implémentation de l'algorithme *A** dans une nouvelle architecture agent respectant le standard FIPA (*Foundation for Intelligent Physical Agents*).

Cependant, ces algorithmes fonctionnent en ayant une représentation globale de l'environnement et sont connus pour donner des comportements peu réalistes car issus d'agents omniscients. En considérant ce problème, de nombreuses recherches ont proposé des algorithmes centrés sur l'agent, qui prennent en compte le principe de localité. On peut citer par exemple l'algorithme *BVP Planner* [14] qui utilise une carte globale couplée à des cartes locales gérées par les agents. Ces cartes locales contiennent des buts intermédiaires, générés en fonction des perceptions de l'agent, lui permettant ainsi de réagir de manière plus réaliste dans un environnement dynamique. Autre exemple, l'algorithme RVO (*Reactive Velocity Obstacles*) [3] se focalise sur l'évitement dynamique d'obstacles en intégrant le comportement réactif des autres agents et permet de produire des mouvements visuellement très réalistes. Enfin, [9] propose un algorithme qui donne la possibilité aux agents de définir des ROI (*Region Of Interest*) qui évoluent en même temps que les objectifs de ces agents. Ainsi, tous ces travaux ont fait le choix de proposer des outils et algorithmes génériques et réutilisables, notamment grâce à une nette séparation entre agent et environnement.

Dans l'ensemble, lorsque le GPGPU est utilisé dans des simulations multi-agents, on observe une nette accélération des simulations (au minimum 2 fois plus rapide qu'une implémentation sur CPU), en particulier compte tenu du fait qu'ils ont été obtenus en utilisant des cartes graphiques standards comportant quelques centaines de cœurs. Cependant, comme le souligne [1], implémenter un modèle sur GPU n'implique pas obligatoirement un gain de performance, surtout dans le domaine des SMA où l'on peut trouver de nombreuses architectures différentes. En effet, [1] montre clairement l'influence des choix d'implémentation sur les résultats et performances. Ainsi, en dépit d'outils de qualité comme CUDA et OpenCL, effectuer une implémentation GPGPU efficace requiert toujours de prendre en compte les spécificités liées au GPGPU. Par rapport à ce problème, nous allons voir maintenant que les systèmes hybrides représentent une alternative très intéressante pour la diffusion de la technologie GPGPU dans la communauté multi-agents.

5 Les systèmes hybrides

Jusqu'à maintenant, nous avons recensé des contributions pour lesquelles le système est exécuté dans son ensemble par le GPU (*tout-sur-GPU*). Cette section va présenter des *systèmes hybrides*, pour lesquels l'exécution du SMA est répartie entre le CPU et le GPU.

Par exemple, [31] propose un framework visant à aider l'utilisateur dans la conception et le déploiement de simulations dans le domaine du trafic routier. Ce framework est voulu très modulaire et peut faire appel, grâce à une approche hybride, à la librairie *MAT-Sim* (*Multi-Agent Transport Simulation*) et à des algorithmes parallélisés permettant d'adapter et exécuter automatiquement certains comportements agents les plus gourmands en ressources sur le GPU (comme les algorithmes de navigations ou de négociations). Ainsi, un avantage important de l'approche hybride tient au fait qu'elle autorise une plus grande flexibilité et de nouvelles opportunités pour les modèles agents car elle permet une ouverture sur d'autres technologies.

Dans [19], la conversion du modèle *Sworm* vers une version utilisant le GPGPU passe aussi par

une approche hybride. Celle-ci est motivée par le fait que *Sworm* est une simulation multi-niveaux intégrant deux types d'agents bien différents : (1) des agents réactifs (niveau micro) simulés par le GPU et (2) des agents cognitifs (niveau macro) gérés par le CPU. En effet, les agents cognitifs invoquent des processus complexes qui peuvent reposer sur de nombreuses données et beaucoup de structures conditionnelles. De fait, ils ne peuvent généralement pas être portés efficacement sur GPU. Ainsi, en éliminant les contraintes du *tout-sur-GPU*, l'approche hybride autorise une intégration plus facile d'agents ayant des architectures hétérogènes. De plus, cette contribution illustre encore une fois le fait que le portage d'un SMA sur GPU ne garantit pas un gain de performance et montre l'importance des choix de structures de données. En comparant plusieurs implémentations différentes du modèle *Sworm*, les résultats montrent qu'une version GPU peut être moins performante qu'une version CPU si les structures de données ne s'y prêtent pas. En l'occurrence, il s'agit principalement de s'assurer que les données peuvent être traitées avec un degré élevé d'indépendance.

Un autre exemple de l'intérêt des systèmes hybrides est donné dans [24]. Ces travaux présentent trois approches différentes pour l'implémentation d'un gestionnaire de tâche et d'ordonnancement des actions dans un SMA : (1) approche *tout-sur-CPU*, (2) approche *tout-sur-GPU*, et (3) *approche hybride*. Les avantages et inconvénients de chacune des solutions montrent que l'approche hybride est la plus prometteuse pour ce contexte applicatif, car la contrainte d'exécuter des tâches simples et indépendantes n'existe plus. Il faut aussi noter que ces travaux sont les premiers à considérer l'utilisation du GPGPU pour des SMA en dehors d'un contexte de simulation.

Dans [22], une approche hybride a été choisie pour intégrer des modules GPU indépendants dans *TurtleKit*, une plateforme de simulation multi-agents générique. L'objectif de ce travail est d'utiliser le GPGPU tout en conservant l'interface de programmation orientée objet des agents afin de préserver la réutilisabilité et l'accessibilité. Pour cela, l'approche hybride qui est proposée repose sur une distinction explicite entre le comportement des agents (géré par le CPU) et les dynamiques de l'environnement (gérées par le GPU). L'environnement est ainsi considéré comme une entité active du système qui va intégrer le concept de *délégation GPU des perceptions agents* qui consiste en une transformation des processus de perceptions agents en dynamiques endogènes de l'environnement, calculées par le GPU. Grâce à cette architecture hybride, le modèle de programmation de la plate-forme *TurtleKit* reste inchangé.

Enfin, certaines recherches proposent une architecture logicielle de haut niveau qui se focalise sur le déploiement de SMA sur des systèmes matériels hétérogènes et distribués. Par exemple, dans le contexte des simulations de foules, [34] définit une architecture logicielle divisée en deux : le *Action Server* (AS) et le *Client Process* (CP). L'AS doit prendre en charge le calcul de la simulation pendant que le CP s'occupe de la gestion du comportement des agents et de leurs états. On voit ici que la flexibilité d'une approche hybride permet de considérer des systèmes beaucoup plus évolués en termes de fonctionnalités et d'architectures logicielles.

Pour conclure, la figure 2 donne une chronologie (1) des événements les plus marquants relatifs au GPGPU et présente aussi les différentes contributions classées et colorées en accord avec leurs caractéristiques d'implémentations : (2) utilisation directe des fonctions graphiques du GPU, (3) utilisation de CUDA ou OpenCL, (4) utilisation d'une approche hybride. L'évolution des architectures des cartes graphiques Nvidia est aussi là pour donner une idée de l'augmentation du nombre de cœurs au sein des GPU.

6 Généricité et accessibilité des approches existantes

Des sections précédentes ressortent deux perspectives essentielles en vue d'une adoption plus grande du GPGPU par la communauté multi-agents : la généricité et l'accessibilité. Dans cette section, nous proposons donc une étude des contributions recensées en fonction de (1) la nature et la généricité du modèle de SMA et (2) le degré d'accessibilité de la solution.

6.1 Nature et généricité des modèles

La nature des modèles de SMA utilisant le GPGPU est fortement liée à l'évolution de cette technologie et des outils associés. En 2008, le faible nombre de contributions pouvait s'expliquer par : (1) la complexité à modéliser des SMA en utilisant directement les fonctions graphiques et (2) les limitations du matériel alors disponible (taille mémoire, bande pas-

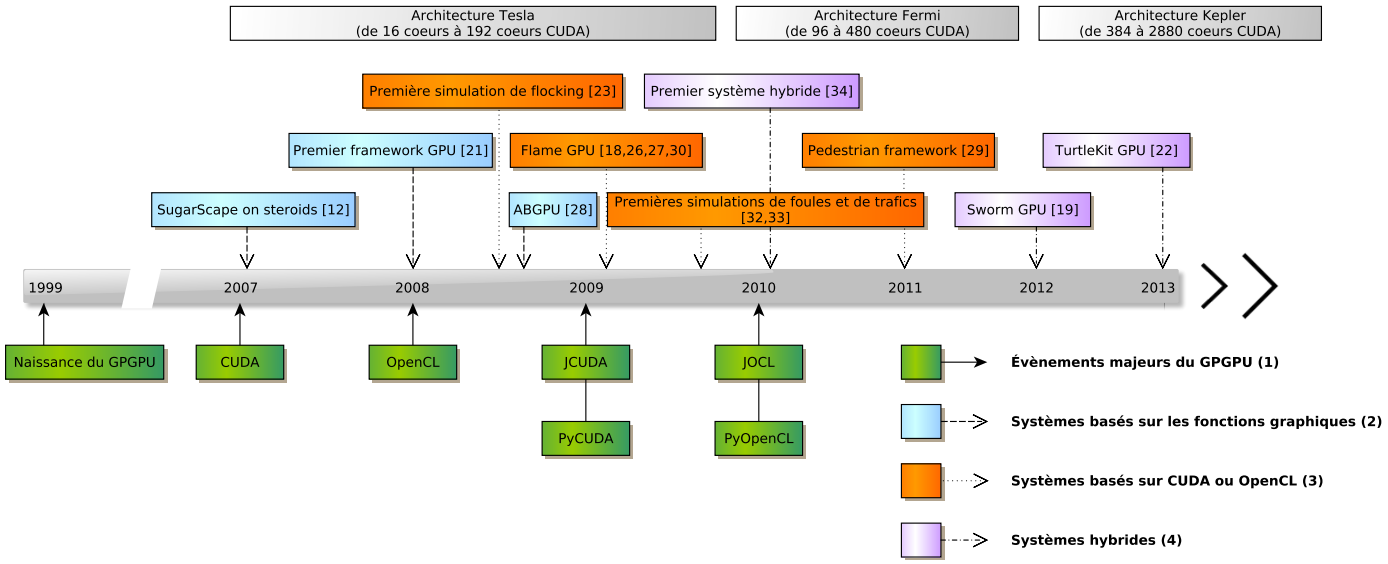

FIGURE 2 – Chronologie de l'interaction entre SMA et GPGPU

sante, etc.). Sans surprise, au vue des difficultés énoncées, la très grande majorité des modèles implémentés sur GPU mettait en scène des agents purement réactifs évoluant dans des environnements minimalistes.

Avec la sortie de CUDA et d'OpenCL, le nombre de contributions a considérablement augmenté. Mais, malgré la simplification importante apportée par ces outils, peu de travaux ont porté leur attention sur l'amélioration de la généricité. De fait, l'augmentation des performances reste la motivation première et la plupart des implémentations se font de zéro, limitant donc le modèle agent produit au domaine pour lequel il a été créé.

On peut tout de même remarquer que la complexité des modèles agents proposés augmente et que certains travaux reposent sur des architectures cognitives (e.g. [29]) ou hétérogènes (e.g. [17, 11]). En particulier, le framework *Flame GPU* fournit des modèles agents prédéfinis capables de s'adapter à plusieurs domaines applicatifs différents (biologie : e.g. [30, 27], science comportementale : e.g. [18], etc.).

Par ailleurs, grâce au haut niveau de gestion des données mis en place dans CUDA et OpenCL, il est devenu possible de séparer plus facilement le modèle agent de celui de l'environnement afin de lui attribuer un véritable rôle, notamment en le rendant actif dans le processus de simulation comme c'est le cas dans les simulations de foules avec la gestion (1) de forces sociales [29] et (2) d'algorithmes de mouvement [3, 14, 9]. Grâce à cette séparation, ces derniers travaux représentent d'importantes contributions du point de vue de la généricité car ils se concentrent sur la généralisation d'algorithmes pouvant être appliqués dans de nombreux domaines.

Plus flexibles, les approches hybrides ne sont pas restreintes et permettent d'implémenter toutes sortes de SMA. En effet, seules les parties du modèle qui s'y prêtent utiliseront le GPGPU, de telle sorte qu'une approche hybride facilite la création de SMA dans lesquels des agents réactifs et cognitifs cohabitent (e.g. [19]) et permet aussi une plus grande indépendance entre agent et environnement (e.g. [22, 24]). Il devrait être ainsi possible d'envisager l'implémentation d'agents possédant des architectures plus complexes, par exemple BDI (*Beliefs-Desires-Intentions*), en déléguant une partie de leur comportement sur GPU. Une telle perspective apporte également de meilleures possibilités de modularité dans la conception du SMA en autorisant l'utilisation de systèmes matériels hétérogènes et distribués (e.g. [34]).

Mise à part l'approche hybride qui facilite l'intégration de modèles différents, au final la généricité n'est que très peu mise en avant. Seules quelques contributions comme [21], *ABGPU* [28], *Flame GPU* [30] et [22] abordent explicitement la question en fournissant des cadres reposant sur un modèle agent définissant des primitives de haut niveau pouvant être transposées dans des contextes différents.

6.2 L'accessibilité des modèles

La nécessité de simplifier l'utilisation et la programmation sur GPU est très vite devenue une évidence et cela dès l'émergence de cette technologie [25]. *Lysenko* [21] et *ABGPU* [28] sont les premières contributions à considérer l'accessibilité comme un critère essentiel. En effet, ces travaux ne nécessitaient que peu de connaissances en GPGPU car ils fournissaient des fonctions GPU prédéfinies de haut niveau, directement utilisables depuis le langage C/C++.

Par la suite, bien que CUDA et OpenCL aient grandement simplifié l'utilisation du GPGPU, l'accessibilité des solutions créées est restée une problématique secondaire et la majorité des travaux requièrent toujours des connaissances importantes en GPGPU. Seul *Flame GPU* et [29, 22, 19] rendent son utilisation transparente.

Même si leur but n'est pas de rendre l'utilisation du GPGPU totalement transparente, il faut noter les orientations prises par certains travaux liés à la simulation de foules et de trafics. En travaillant sur la réutilisation des outils créés (e.g. algorithmes de *PathPlanning* [14, 3, 9]), ces travaux font un pas certain vers une accessibilité renforcée en insistant sur la capitalisation des efforts passés, et donc sur la réutilisation. Ainsi, cette démarche nous apparaît comme cruciale pour l'avenir : la constitution de bibliothèques d'algorithmes spécifiquement dédiées au monde SMA ne peut qu'augmenter significativement l'accessibilité du GPGPU pour la communauté multi-agents. C'est d'ailleurs sous cette forme, de bibliothèques prêtes à l'emploi, que de nombreux domaines utilisent cette technologie, comme c'est le cas dans le domaine du traitement de l'image [8].

Par ailleurs, l'approche hybride constitue encore une fois une piste très intéressante en ce qui concerne l'accessibilité. Tout d'abord, elle s'accorde naturellement bien avec une vision modulaire du modèle et de son implémentation, et donc avec l'idée de librairie réutilisable. De plus, une approche hybride est par nature ouverte aux autres technologies car elle lève les contraintes du *tout-sur-GPU*. Par exemple, [19] et [22] utilisent la programmation orientée objet en parallèle du GPGPU.

8. Par exemple, les librairies NVIDIA Performance Primitives (NPP) et CUDA Fast Fourier Transform (cuFFT) sont très abouties et leur utilisation est largement répandue dans la communauté image.

7 Conclusion et perspectives

De l'état de l'art que nous avons réalisé dans le domaine des SMA, il est clair que le GPGPU représente une technologie très intéressante pour tous les domaines où le temps d'exécution et/ou les aspects temps réels sont cruciaux. L'utilisation du GPGPU permet en effet d'augmenter la taille des environnements, le nombre d'agents, de diminuer le temps de rendu, de permettre un affichage temps réel, etc. Il faut noter que la plupart des travaux présentés utilisent des cartes graphiques grand public comportant "seulement" quelques centaines de cœurs, alors que l'une des dernières cartes professionnelles, la Nvidia Tesla K10, en contient 3072. Ainsi, il est normal que la quasi totalité des travaux recensés se rapporte à la simulation de SMA, domaine où les gains obtenus peuvent être considérables, uniquement avec des ordinateurs personnels classiques.

Les travaux sur l'utilisation du GPGPU pour l'allocation de tâches réalisés dans [24] sont une exception notable qui préfigure cependant du fait que le GPGPU peut apporter beaucoup au domaine des SMA en général. En effet, le GPGPU est un paradigme où la décentralisation des données et des traitements est intrinsèque. Ainsi, il y a fort à parier que le GPGPU va se répandre dans la communauté multi-agents dès lors que l'accessibilité et la généricité des outils couplant GPGPU et SMA seront matures.

À l'image des travaux de thèse de Bourgoin [4] qui référencent l'accessibilité et la généricité comme critères essentiels pour le domaine du GPGPU, nous retrouvons les mêmes perspectives pour l'utilisation du GPGPU dans le domaine des SMA. Ainsi, nous avons considéré ces deux aspects comme fondamentaux en vue d'une adoption plus rapide du GPGPU dans la communauté multi-agents. Ceci nous a amené à identifier plusieurs pistes de recherche qui nous semblent pertinentes.

Tout d'abord d'un point de vue technique, on peut affirmer que les solutions hybrides représentent une solution d'avenir, notamment du fait de deux avantages majeurs : (1) elles permettent d'utiliser des APIs de programmation classique, apportant ainsi une plus grande accessibilité aux outils créés et (2) les systèmes hybrides offrent une grande modularité de modélisation et d'implémentation, rendant possible l'utilisation d'agents ayant des architectures plus complexes, hétérogènes et/ou cognitives.

Deuxièmement, sur un plan plus conceptuel, nous avons vu qu'une séparation explicite entre le modèle des agents et celui de l'environnement est sans aucun doute une piste avec un fort potentiel. Ainsi, pour aller plus loin, on pourrait imaginer de concevoir des modèles d'environnements génériques propulsés par le GPU et dans lesquels il serait possible d'intégrer différents types d'architectures agents (gérées par le CPU ou le GPU) sans effort.

Ensuite, une meilleure généricité et accessibilité passent probablement par une abstraction performante de l'utilisation du GPGPU. C'est l'objectif de deux langages de programmation de plus haut niveau : *Spoc* [4] et *Harlan* [16]. Ils permettent, en effet, de s'abstraire de manipuler explicitement de nombreux paramètres matériels comme la mémoire ou le placement des calculs sur les différentes unités et offrent une plus grande portabilité des outils développés.

Pour finir, nous pensons que les architectures massivement parallèles sont une opportunité de repenser notre façon de modéliser et comprendre les SMA. En effet, le GPGPU repose fondamentalement sur la réalisation de calculs locaux et décentralisés. Ainsi, à plus long terme, il paraît pertinent de définir de nouveaux modèles multi-agents mieux adaptés, c'est-à-dire des modèles contenant des patterns permettant une adéquation plus naturelle entre ces architectures massivement parallèles et les SMA.

Références

[1] Brandon G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient simulation of agent-based models on multi-GPU and multi-core clusters. *Proc. of the 3rd Int. ICST Conference on Simulation Tools and Techniques*, 2010.

[2] Avi Bleiweiss. GPU accelerated pathfinding. *Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, 2008.

[3] Avi Bleiweiss. Multi agent navigation on the gpu. *Games Developpement Conference*, 2009.

[4] Mathias Bourgoin. *Abstractions performantes pour cartes graphiques*. PhD thesis, Université Pierre et Marie Curie, 2013.

[5] Giuseppe Caggianese and U. Erra. GPU Accelerated Multi-agent Path Planning Based on Grid Space Decomposition. In *Proc. of the Int. Conference on Computational Science*, volume 9, pages 1847–1856. Elsevier, 2012.

[6] Shuai Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10) :1370 – 1380, 2008.

[7] Simon Coakley, R. Smallwood, and M. Holcombe. Using x-machines as a formal basis for describing agents in agent-based modelling. *Proc. of 2006 Spring Simulation Multiconference*, pages 33–40, 2006.

[8] Alessandro R. Da Silva, W. S. Lages, and L. Chaimowicz. Boids that see : Using self-occlusion for simulating large groups on GPUs. *Computers in Entertainment*, 7(4), 2009.

[9] Aljosha Demeulemeester and C.-F. Hollemeersch. Hybrid path planning for massive crowd simulation on the GPU. *Motion in Games*, pages 304–315, 2011.

[10] Luiz Guilherme Oliveira dos Santos, E. Gonzales Clua, and F. Bernardini. A parallel fipa architecture based on gpu for games and real time simulations. In *Entertainment Computing - ICEC 2012*, pages 306–317. 2012.

[11] Roshan M. D'Souza, M. Lysenko, S. Marino, and D. Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on gpu. In *Proc. of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 1–12, 2009.

[12] Roshan M. D'Souza, M. Lysenko, and K. Rahmani. SugarScape on steroids : simulating over a million agents. *Proc. of Agent 2007 conference*, 2007.

[13] Ugo Erra, B. Frola, V. Scarano, and I. Couzin. An Efficient GPU Implementation for Large Scale Individual-Based Simulation of Collective Behavior. *2009 Int. Workshop on High Performance Computational Systems Biology*, 0 :51–58, 2009.

[14] Leonardo G. Fischer, R. Silveira, and L. Nedel. Gpu accelerated path-planning for multi-agents in virtual environments. In *Proc. of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, pages 101–110, 2009.

[15] Dirk Helbing, I. J. Farkas, P. Molnar, and T. Vicsek. Simulation of pedestrian crowds

in normal and evacuation situations. *Pedestrian and Evacuation Dynamics*, pages 21–58, 2002.

[16] Eric Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for gpus. In *PARCO*, pages 297–304, 2011.

[17] Alwyn V. Husselmann and K. A. Hawick. Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUs. In *Int. Conference on Parallel and Distributed Computing and Systems*, pages 100–107, 2011.

[18] Twin Karmakharm, P. Richmond, and D. M. Romano. Agent-based Large Scale Simulation of Pedestrians With Adaptive Realistic Navigation Vector Fields. In *Theory and Practice of Computer Graphics*, pages 67–74, 2010.

[19] Guillaume Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe. Using gpu for multi-agent multi-scale simulations. In *Distributed Computing and Artificial Intelligence*, volume 151 of *Advances in Intelligent and Soft Computing*, pages 197–204, 2012.

[20] Hong Li and A. Kolpas. Parallel simulation for a fish schooling model on a general purpose graphics processing unit. *Concurrency and Computation : Practice and Experience*, 21 :725–737, 2009.

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

[22] Fabien Michel. Intégration du calcul sur gpu dans la plate-forme de simulation multi-agent générique turtlekit 3. In *Dynamiques, couplages et visions intégratives - JFSMA13*, pages 189–198, 2013.

[23] Erick Passos, M. Joselli, and M. Zamith. Supermassive crowd simulation on gpu based on emergent behavior. In *In Proc. of the Seventh Brazilian Symposium on Computer Games and Digital Entertainment*, pages 81–86, 2008.

[24] Roman Pavlov and Jörg Müller. Multi-Agent Systems meet GPU. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, volume 394, pages 115–122, 2013.

[25] Kalyan S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. *Proc. of the 2008 Spring simulation multiconference*, pages 116–123, 2008.

[26] Paul Richmond, S. Coakley, and D. M. Romano. A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proc. of The 8th Int. Conference on Autonomous Agents and Multiagent Systems*, pages 1125–1126, 2009.

[27] Paul Richmond, S. Coakley, and D. M. Romano. Cellular Level Agent Based Modelling on the Graphics Processing Unit. In *2009 Int. Workshop on High Performance Computational Systems Biology*, pages 43–50, 2009.

[28] Paul Richmond and D. M. Romano. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *In Proc. Int. Workshop on Super Visualisation (IWSV08)*, 2008.

[29] Paul Richmond and D. M. Romano. A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. *European Simulation and Modelling*, 2011.

[30] Paul Richmond, D. Walker, S. Coakley, and D. M. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, pages 334–347, 2010.

[31] Yoshihito Sano, Y. Kadon, and N. Fukuta. A Performance Optimization Support Framework for GPU-based Traffic Simulations with Negotiating Agents. In *Proc. of the 2014 7th Int. Workshop on Agent-based Complex Automated Negotiations*, 2014.

[32] Zhen Shen, K. Wang, and F. Zhu. Agent-based traffic simulation and traffic signal timing optimization with GPU. *Intelligent Transportation Systems*, pages 145–150, 2011.

[33] David Strippgen and K. Nagel. Multi-agent traffic simulation with CUDA. In *Int. Conference on High Performance Computing& Simulation*, pages 106–114, 2009.

[34] Guillermo Vigueras, J. M. Orduña, and M. Lozano. A GPU-Based Multi-agent System for Real-Time Simulations. In *Advances in Practical Applications of Agents and Multiagent Systems, 8th Int. Conference on Practical Applications of Agents and Multiagent Systems*, volume 70, pages 15–24, 2010.