HAL
open science

État de l'art sur les simulations multi-agents et le GPGPU Évolution et perspectives de recherches

Emmanuel Hermellin, Fabien Michel, Jacques Ferber

To cite this version:

Emmanuel Hermellin, Fabien Michel, Jacques Ferber. État de l'art sur les simulations multi-agents et le GPGPU Évolution et perspectives de recherches. Revue des Sciences et Technologies de l'Information - Série RIA : Revue d'Intelligence Artificielle, 2015, 29 (3-4), pp.425-451. 10.3166/ria.29.425-451 . lirmm-01236844

HAL Id: lirmm-01236844

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01236844v1

Submitted on 2 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

État de l'art sur les simulations multi-agents et le GPGPU
Évolution et perspectives de recherches

Emmanuel Hermellin , Fabien Michel , Jacques Ferber

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

RÉSUMÉ. *Dans le domaine des systèmes multi-agents, la hausse constante du nombre d'entités implique un besoin en ressource de calcul de plus en plus important. Cependant, les nombreux outils et plates-formes permettant le développement de simulations multi-agents sur CPU ne supportent plus cette demande en perpétuelle augmentation. Une solution est de se tourner vers le calcul haute performance et notamment vers le calcul sur carte graphique : le GPGPU. Apportant un rapport performance/prix imbattable, cette technique souffre cependant d'une programmation très spécifique qui limite son adoption par la communauté. Dans cet article, nous faisons un état des lieux des simulations multi-agents sur GPU et identifions les solutions les plus prometteuses en vue d'une généralisation de l'utilisation de cette technologie dans la communauté multi-agent.*

ABSTRACT. *In some application domains, using a Multi-Agent Systems (MAS) modeling approach may require to handle a large number of agents (crowds, traffic, ecosystems, etc.). In such cases, the computational resources which are needed often raise scalability problems. Considering this kind of issues, General-Purpose computing on Graphics Processing Units (GPGPU) appears to be an appealing solution as it enables huge speed up on a regular PC. However, this technology relies on a highly specialized architecture, implying a very specific programming approach. That is the reason why GPGPU is not widespread technology in the MAS community. This paper reviews the literature which is at the intersection between MAS and GPGPU. The different approaches used are presented and the most promising solutions for a generalization of GPGPU technology in our community will be highlighted.*

MOTS-CLÉS : *SMA, GPGPU, CUDA, OpenCL, architecture hybride.*

KEYWORDS: *MAS, GPGPU, CUDA, OpenCL, hybrid architecture.*

DOI:10.3166/RIA.29.425-451 © 2015 Lavoisier

1. Introduction

En réalisant un état de l'art des contributions traitant de l'utilisation du GPGPU dans des systèmes multi-agents, nous nous retrouvons à l'intersection de deux domaines : (1) les systèmes multi-agents (SMA), avec en particulier les simulations multi-agents, et (2) le calcul haute performance (*High Performance Computing* ou HPC).

La simulation multi-agent est utilisée pour l'ingénierie et l'étude des systèmes complexes dans de nombreux domaines (robotique collective, biologie, économie, gestion urbaine, etc.) 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 et/ou le nombre d'agents. De plus, si l'on veut ajouter une visualisation en temps réel [1] et une possibilité d'interaction avec le modèle, on accentue encore grandement la puissance nécessaire. 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. Cette limitation devient parfois si importante qu'il est nécessaire de se tourner vers le HPC.

Parallèlement, l'utilisation du HPC se généralise avec l'utilisation de grilles de calculs, *cluster*, GPU (*Graphical Processing Unit*), etc. Parmi ces technologies, le calcul sur GPU, aussi appelé GPGPU (*General-Purpose computing on Graphics Processing Units*), possède des avantages non négligeables : (1) son rapport prix/performances est excellent et (2) il est disponible sur de nombreuses machines. En effet, de nos jours, quasiment tous les ordinateurs ont une carte graphique dédiée ou intégrée au CPU possédant des capacités GPGPU. Réelle révolution technologique, le GPGPU permet donc 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 *et al*. (2014) offrent une introduction détaillée du GPGPU et donnent une vision générale des possibilités, de son évolution et de son utilisation. Che *et al*. (2008) proposent 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.

Ainsi, les nombreuses spécificités de la programmation sur GPU font que les travaux mêlant simulation multi-agent et GPU restent ponctuels, difficiles à généraliser et quasiment impossibles à réutiliser dans d'autres contextes. Une implémentation sur GPU est bien plus complexe qu'un simple changement de langage de programmation.

1. On qualifie la visualisation de temps réel lorsqu'elle est assez rapide pour rendre la simulation exploitable.

En particulier, le problème doit pouvoir être représenté par des structures de données distribuées et indépendantes. La jeunesse du GPGPU et son manque d'accessibilité limitent pour l'instant son utilisation aux domaines où le temps de calcul est critique.

C'est dans ce contexte que nous réalisons un état de l'art des contributions traitant de l'utilisation du GPGPU pour la modélisation et l'implémentation de simulations multi-agents. La section 2 présente en détail le calcul haute performance et en particulier le GPGPU. La section 3 introduit les systèmes multi-agents. La section 4 présente des contributions utilisant le GPGPU pour l'implémentation de simulations multi-agents tout en respectant l'évolution de cette technologie et des outils associés. La section 5 propose une analyse des travaux recensés selon deux critères que nous considérons comme essentiels : (1) la généricité du modèle agent et (2) l'accessibilité du *framework* proposé. Enfin, la section 6 conclut cet article et donne des pistes de recherche pour l'avenir. Les solutions les plus prometteuses pour une généralisation de la technologie GPGPU dans notre communauté seront ainsi mises en évidence.

2. Le calcul haute performance

A l'origine, le *Calcul Haute Performance* désignait la science relative à la conception et au développement des grands centres de calcul. Le HPC fait maintenant aussi référence aux développements d'architectures matérielles et logicielles permettant d'accéder à une plus grande puissance de calcul. Il est donc utilisé pour des tâches telles que les prévisions météorologiques, l'étude du climat, la modélisation moléculaire (calcul des structures et propriétés de composés chimiques, etc.), les simulations physiques (simulations aérodynamiques, calculs de résistance des matériaux, simulation nucléaire, étude thermodynamique, etc.), les simulations en finance et assurance (calcul stochastique), etc.

La puissance de calcul des super-ordinateurs a augmenté parallèlement aux progrès de l'électronique et suit la théorie énoncée par Gordon Moore. Pour mesurer cette puissance, on utilise une unité : le *flops* (opérations à virgule flottante par seconde, *Floating point Operations Per Second*). La figure 1 présente une évolution de la puissance des 500 meilleurs super-calculateurs de 1999 à 2014.

Cependant, les programmes exécutés sur ces machines ne sont pas écrits de la même manière que les logiciels que l'on retrouve sur nos ordinateurs personnels. Ils nécessitent une programmation particulière leur permettant de profiter pleinement de l'architecture matérielle de ces super ordinateurs.

2.1. Le parallélisme

Le *parallélisme* consiste à utiliser des architectures matérielles permettant de traiter des informations de manière simultanée dans le but de réaliser le plus grand nombre d'opérations par seconde. Les architectures parallèles sont devenues le paradigme dominant pour tous les ordinateurs depuis les années 2000. En effet, multiplier le nombre de cœurs est devenu l'alternative à la course à la fréquence des processeurs *mono-*

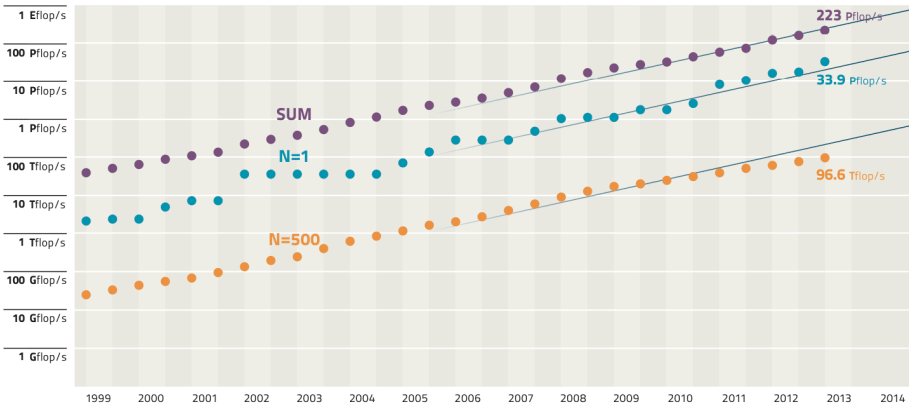

Figure 1. Évolution de la puissance des super-calculateurs (Sum représente la puissance combinée des 500 plus gros super-calculateurs, N = 1 la puissance du super-calculateur le plus rapide et N = 500 la puissance du super-calculateur se trouvant à la place 500, top500.org)

cœur : une augmentation de fréquence apporte des problèmes de refroidissement, de stabilité, de consommation. La création de processeurs *multi-cœurs* résout en partie ces problèmes.

La *taxonomie* établie par Michael J. Flynn (1972) distingue quatre types principaux de parallélisme et classe les ordinateurs selon le type d'organisations des flux de données et d'instructions :

– **Architecture SISD** : systèmes séquentiels qui traitent une donnée à la fois (*Single instruction Single Data*).

– **Architecture SIMD** : systèmes parallèles traitant de grandes quantités de données d'une manière uniforme (*Single instruction Multiple Data*).

– **Architecture MIMD** : systèmes parallèles traitant de grandes quantités de données de manières hétérogènes (*Multiple instruction Multiple Data*).

– **Architecture MISD** : systèmes parallèles traitant une seule donnée de manière hétérogène (*Multiple instruction Single Data*).

La figure 2 présente cette classification.

Cette approche montre clairement deux types de parallélismes différents : le parallélisme par flot d'instructions également nommé *parallélisme de traitement* ou de contrôle, où plusieurs instructions différentes sont exécutées simultanément, qui correspond au *MIMD* et le *parallélisme de données*, où les mêmes opérations sont répétées sur des données différentes, le *SIMD*. C'est à cette catégorie de parallélisme qu'appartient le GPGPU.

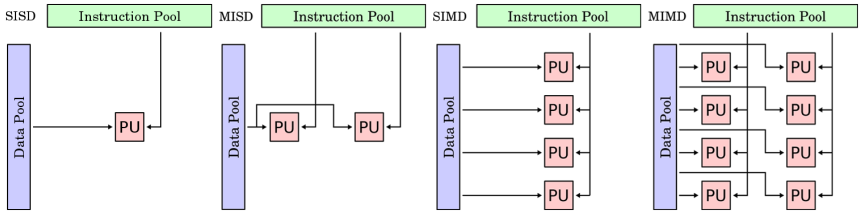

Figure 2. Classification des quatre types principaux de parallélisme

2.2. General-purpose computing on graphics processing units

La parallélisation d'un programme ou d'un algorithme doit être pensée selon le matériel utilisé. Elle peut tirer partie des processeurs *multi-cœurs* et *multi-threads*, utiliser la programmation sous GPU ou utiliser les deux. Ce choix est important car les techniques de développement associées, les philosophies de programmation ainsi que les langages utilisés sont très différents voir même incompatibles. Ce choix impactera aussi les coûts de développement ainsi que les performances, très variables selon le matériel et l'architecture. Le GPGPU est intéressant car il ne nécessite pas de gros investissements et offre de très bonnes performances.

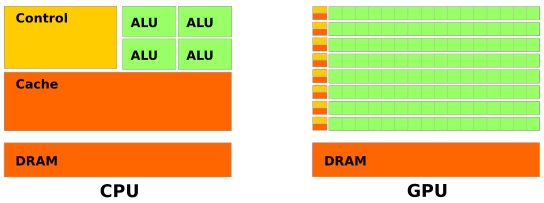

Figure 3. Différences entre architecture CPU et GPU

A l'origine, les cartes graphiques ont été développées dans le but de soulager les processeurs (CPU) de la charge que demandait l'affichage graphique et la gestion des pixels. 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, comme le montre la figure 3. 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*). 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). La figure 4 illustre la différence de performance théorique entre GPU et CPU.

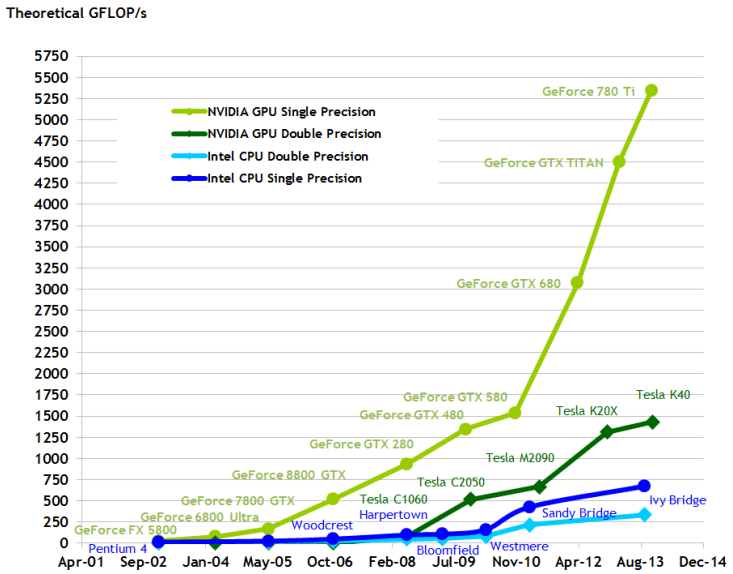

Figure 4. Comparaison des performances théoriques entre CPU et GPU

C'est au début des années 2000 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 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* [2] 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.

Nvidia mit à disposition, en 2007, la première interface de programmation dédiée au GPGPU : CUDA [3] (*Compute Unified Device Architecture*). L'objectif était d'offrir un meilleur support et une plus grande accessibilité pour cette technologie. 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 [4] (*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 restric-

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

tion de marques). OpenCL est 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. 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 séquentielle 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* (les paramètres *blockDim.x*, *blockDim.y* définissent la taille de ces blocs), qui sont eux-mêmes rassemblés dans une *grille globale* (la figure 5 donne un exemple de l'utilisation d'une grille 2D). Chaque *thread* au sein de cette structure est identifié par des coordonnées uniques 3D (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*) lui permettant d'être localisé. De la même façon, un *bloc* possède aussi des coordonnées 3D qui lui permettent d'être identifié dans la *grille* (respectivement *blockIdx.x*, *blockIdx.y*, *blockIdx.z*). Les *threads* [5] 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 [6] ou Python [7].

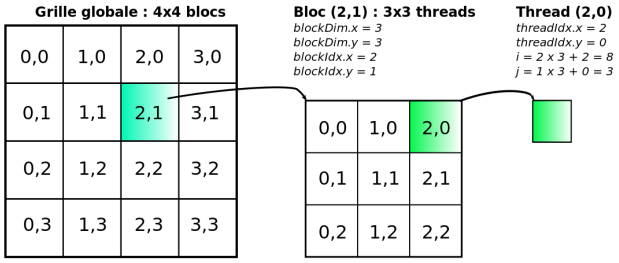

Figure 5. Illustration du concept de grille, bloc, thread

Pour illustrer plus en détail le GPGPU et sa philosophie, nous avons mis en ligne sur *GitHub* des exemples simples de programmes (addition d'un vecteur, calcul de pi, etc.) ainsi que leur parallélisation sur CPU et GPU [8]. Les travaux d'Owens *et al*. (2007) permettent aussi d'avoir une meilleure compréhension du domaine car ils présentent en détails les différents aspects techniques autour de la programmation sur GPU.

5. 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.
6. JCUDA www.jcuda.org et JOCL www.jocl.org
7. PyCUDA et PyOpenCL http://mathema.tician.de/
8. https://github.com/ehermellin/IntroHPC

3. Système multi-agent

Les systèmes multi-agents sont des systèmes composés d'un ensemble d'entités, appelés *agents*, potentiellement organisés partageant un environnement commun dans lequel ils peuvent interagir (Ferber, 1999). Les systèmes multi-agents sont donc basés sur un certain nombre de principes et concepts avec notamment *les agents*, *l'environnement*, *les interactions*.

Un *agent* est une entité logicielle ou matérielle située dans un *environnement* réel ou virtuel. L'agent est capable d'agir dans son environnement, d'être dirigé par ses tendances (objectifs individuel, buts, satisfaction, survie), posséder ses propres ressources, communiquer (directement ou indirectement), avoir une représentation partielle du monde dans lequel il évolue. Son comportement et son autonomie sont la conséquence de ses perceptions, représentations et interactions dans le monde avec les autres agents. Cet agent possède une structure interne que l'on appelle *architecture* qui peut être soit *réactive*, on ne considère que les perceptions-actions (ou réponse à des stimulis), soit *cognitive*, l'agent fait alors preuve de réflexion.

Le concept d'agent seul n'est pas suffisant lorsque l'on parle de systèmes multi-agents (Odell *et al.*, 2003) : il est indissociable de celui d'environnement. En effet, l'ensemble des perceptions et des actions qu'un agent est susceptible de réaliser est entièrement défini par rapport à l'environnement où celui-ci va opérer. L'environnement définit ainsi les conditions d'existence de ce genre d'entités. Il n'est donc pas possible de parler d'agent sans parler d'environnement car ce dernier représente la modélisation du monde dans lequel évolue les agents (Weyns *et al.*, 2005).

L'approche multi-agent se fonde sur une démarche individus-centrée (Michel *et al.*, 2009). En effet, elle considère qu'il est possible de modéliser, non seulement les individus et leurs comportements, mais aussi les interactions qui se déroulent entre ces individus. Elle considère ainsi que la dynamique globale d'un système, au niveau macroscopique, résulte directement des interactions entre les individus qui composent ce système au niveau microscopique. Ainsi, alors que les modèles classiques modélisent les relations qui existent entre les différentes entités identifiées d'un système à l'aide d'équations mathématiques, l'approche multi-agent modélise directement les interactions engendrées par des comportements individuels.

Les systèmes multi-agents sont donc un moyen de comprendre, modéliser, et implémenter un très grand nombre de systèmes distribués. Il peuvent être aussi utilisé comme paradigme de développement pour des logiciels, particulièrement dans des contextes de calcul où le contrôle global n'est pas possible ou pour la représentation de phénomènes réels ou virtuels, en décomposant ces systèmes en entités individuelles pouvant interagir entre elles.

4. État de l'art des simulations multi-agents sur GPU

Nous proposons un état de l'art des contributions traitant de l'utilisation du GPGPU dans un contexte 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 *adhoc*.

L'implémentation d'un modèle sur GPU peut se faire de deux façons différentes, chacune possédant des avantages et inconvénients : (1) *tout-sur-GPU*, la simulation va être exécuté entièrement par la carte graphique et (2) *hybride*, l'exécution de la simulation va être partagée entre le CPU et le GPU. Nous avons donc fait le choix de suivre ce découpage pour classer les contributions : (1) dans un premier temps nous présenterons les recherches s'exécutant entièrement sur GPU, puis (2) nous introduirons les approches hybrides.

4.1. Implémentation tout-sur-GPU

Il existe deux façon d'implémenter un modèle entièrement sur GPU : utiliser (1) les fonctions graphiques des GPU ou (2) les environnements de développement CUDA et OpenCL. Les détails de ces techniques d'implémentation ont été énoncés en section 2.2 et sont explicités dans les travaux d'Owens *et al*. (2007).

4.1.1. Utilisation des fonctions graphiques des GPU

SugarScape est le premier SMA ayant profité d'un portage sur GPU grâce au GPGPU (D'Souza *et al*., 2007). 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* permettait de voir évoluer en temps réel plus de 2 millions d'agents dans un environnement d'une résolution de 2 560 par 1 024. 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 l'implémentation de modèle sur GPU (Lysenko, D'Souza, 2008). Celui-ci était composé de fonctions de bases telles que la gestion de données environnementales, la gestion des interactions entre agents, naissance, mort, etc. Cependant, il ne permettait que des implémentations de modèles simplistes similaires à *SugarScape* avec des agents réactifs aux comportements peu évolués.

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 un ensemble de classes C/C++ et un système de

mots clés (Richmond, Romano, 2008). À cela s'ajoutaient des fonctions et des classes pré-programmées, facilitant d'avantage l'implémentation de comportements et fonctions agents un peu plus évoluées (fonctions de synchronisations, communications, etc). *ABGPU* se voulait optimisé pour des simulations dans lesquelles évoluaient des agents réactifs aux comportements simples qui arrivaient à créer des déplacements complexes et des actions coordonnées. *ABGPU* était ainsi capable de simuler et d'afficher 65 536 agents en mouvements et en interactions 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 de SMA sur GPU, du fait de la grande diversité des modèles. À ce propos, Perumalla et Aaby (2008) ont étudié les contraintes associées à de telles conversions en réalisant l'implémentation de différents modèles (*Mood Diffusion* [9], *Game of Life* [10] et *Schelling Segregation* [11]) sur CPU (avec *NetLogo* (Sklar, 2007) et *Repast* (North *et al*., 2007)), deux environnements de développement de simulations multi-agents très connus), 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, comme nous allons le voir maintenant.

4.1.2. Utilisation des environnements de développement

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 n'a fait qu'augmenter et de nouveaux outils et *frameworks* ont vu le jour.

Le plus représentatif est *Flame GPU* [12] (Richmond *et al*., 2010). En effet, *Flame GPU* est une solution clef en main pour la création de simulations multi-agents sur GPU et 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* (Coakley *et al*., 2006) : un formalisme de représentation d'agents s'appuyant sur une extension du langage XML : le XMML. Ainsi, les données initiales de la simulation ainsi que les états des agents sont implémentés dans des fichiers

9. L'humeur de l'agent est initialisée avec une valeur aléatoire et varie en fonction d'une tendance homéostatique propre à l'agent et aux influences des humeurs de ces voisins.

10. Le modèle Game of Life de Conway.

11. Un des premiers exemples d'émergence fondé sur des interactions sociales.

12. http://www.flamegpu.com/

XMML pendant que les comportements de ces derniers sont programmés en C. Les fichiers XMML vont être combinés dans des templates GPUXMML [13] et ensuite vérifié par un processeur XSLT [14] dans le but d'être compilé dans le langage GPU avec les fichiers de comportements en C. La simulation est ensuite générée puis exécutée par le GPU. La figure 6 (Richmond *et al.*, 2010) illustre le processus de génération d'une simulation avec Flame GPU.

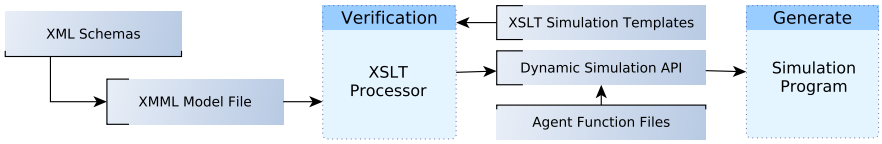

Figure 6. Illustration du processus de génération d'une simulation avec Flame GPU

De plus, *Flame GPU* fournit un *framework* open source contenant des modèles agents pré-programmés facilement réutilisables. Ainsi, il devient le premier *framework* utilisé pour simuler un large éventail de modèles dans des contextes et domaines différents : en biologie, *e.g.* simulation de cellules de peau (Richmond *et al.*, 2009a), en intelligence artificielle, *e.g.* simulation proie - prédateur (Richmond *et al.*, 2009b), ou pour des simulations de foules *e.g.* (Karmakharm *et al.*, 2010 ; Karmakharm, Richmond, 2012).

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 le voir maintenant, beaucoup de travaux existants sont repartis de zéro et se sont focalisés sur les gains de performances.

4.1.2.1. Les simulations de *flocking*

Reynolds (1987) est le premier à avoir proposé un modèle de *flocking*. Chaque individu était alors implémenté de manière indépendante et pouvait naviguer grâce à ces propres perceptions dans un environnement dynamique. Chaque entité devait obéir, selon ces perceptions, à trois lois de comportement :

1. Éviter les collisions avec ses voisins.
2. Avoir la même vitesse que ses voisins.
3. Rester le plus prêt possible de ses voisins et du centre du groupe.

Dans ce genre de modèles, les entités sont considérées comme des agents uniques possédant des règles de comportements et d'interactions. Appelés IBM (*Individual*

13. Les templates sont des modèles qui contiennent du code CUDA afin de générer le code source de la simulation automatiquement à partir des fichiers XMML fournis.
14. Le processeur XSLT va vérifier que le code XMML est correct et correspond au template utilisé.

Based Model (Michel *et al*., 2009)), ces agents peuvent ainsi reproduire de manière expérimentale des comportements de groupes et de modèles biologiques réalistes.

Ce genre de simulation peut faire intervenir un très grand nombre d'entités (jusqu'à plusieurs millions) et nécessite donc une puissance de calcul très importante. Se tourner vers le GPGPU a donc été une évidence et plusieurs contributions significatives ont vu le jour. On peut notamment citer la contribution de Passos *et al*. (2008) introduisant la première simulation de *flocking* sur GPU. Celle-ci fût suivie par le travail de Li *et al*. (2009) qui ajoute la notion d'évitement d'obstacles dans son modèle.

L'article d'Erra *et al*. (2009) est très intéressant car il propose une description complète et détaillée des étapes suivies pour implémenter un modèle sur GPU en utilisant l'API de Nvidia. Reprenant les bases énoncées dans les travaux de Reynolds, les agents de ce modèle ne vont avoir qu'une représentation locale de leur environnement et se coordonnent avec leurs plus proches voisins. 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 à l'écran en temps réel et en 3D par une carte graphique comportant 128 cœurs.

Silva *et al*. (2010) propose un nouveau modèle intégrant une technique de *self-occlusion* dans le but d'accélérer les calculs et le rendu des simulations de *flocking*. 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 même modèle utilisant directement les fonctions graphiques du GPU reste plus rapide.

Finalement, Husselmann et Hawick (2011) proposent 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. La possibilité de donner à chaque espèce différente une "personnalité" caractérisée par des paramètres propres aux agents va aussi être ajoutée. L'étude de ce système est rendue possible grâce à la puissance de calcul offerte par le GPGPU et va ainsi permettre de voir apparaître des comportements émergents entre agents hétérogènes (*flock separation behaviour*) et donc d'avoir des modèles de *flocking* plus complexes. 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 37 000 entités simulées.

4.1.2.2. Les simulations de foules et de trafics

La simulation de foules est une des plus gourmande en ressource de calcul et nécessite une approche agent afin de reproduire le plus fidèlement possible le comportement d'individus dans le but de recréer des mouvements de foules humaines réalistes. Les simulations de foules font ainsi partie des domaines pour lesquels il est pertinent d'étudier des environnements et des populations d'agents toujours plus grands : le GPU devient alors indispensable.

Évolution du *framework ABGPU*, le *Pedestrian framework* (Richmond, Romano, 2011), basé sur CUDA, ne se focalise pas seulement sur la performance. En effet, il propose de pouvoir modéliser pour la première fois des agents ayant un comportement cognitif tels que ceux décrit dans Romano *et al*. (2005) : qualifiés de sociaux, ils s'adaptent à leur environnement et s'expriment au travers de leurs actions et leurs gestes. Ce *framework* permet aussi d'intégrer dans la simulation des forces sociales et en particulier celles d'Helbing *et al*. (2002). Pour cela, la proposition de Richmond et Romano (2011) est la première 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 65 536 agents en 3D et en temps réel avec une carte graphique contenant seulement 96 cœurs.

Les travaux de Varga et Mintal (2014) présentent un modèle de simulation de mouvement d'agents pédestres évoluant dans un environnement discrétisé en cellules. Les agents ne vont avoir qu'une perception locale de leur environnement mais vont posséder un espace local autour d'eux. Cet espace local est lui aussi divisé en cellules, chacune caractérisée par un état et une valeur. L'état de la cellule peut être "libre" ou "bloqué", la valeur correspond à l'opportunité de se déplacer dans cette cellule. Cette valeur est calculée en fonction des champs de gradient diffusés par les objectifs à atteindre, les objets de l'environnement, les autres agents, les obstacles, etc. Ce calcul peut être effectué de manière indépendante sur chaque cellule ce qui rend la parallélisation et l'utilisation du GPGPU très effectif. L'agent va ainsi ce déplacer de cellule en cellule vers son but en s'adaptant à ce qui se passe dans l'environnement.

Assez similaire aux simulations de foules, il existe des recherches portant sur des modèles simulant des réseaux routiers dans des environnements plus ou moins dynamiques. Les motivations qui poussent à la réalisation de ces outils sont l'amélioration de la sécurité et l'évitement des congestions des réseaux de circulation. En effet, ces simulations vont permettre de prendre des décisions et améliorer les évacuations en cas de situation d'urgence. Nécessitant aussi une grande puissance de calcul, les gains de performances offerts par le GPGPU sont clairement visibles dans les travaux de Strippgen et Nagel (2009) et Shen *et al*. (2011).

4.1.2.3. Les algorithmes de navigation

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 le même temps, les systèmes multi-agents ont gagné en popularité auprès des développeurs de jeux vidéo et surtout pour le développement d'intelligences artificielles. Dans ce cadre, la navigation autonome et la planification d'itinéraires ont été rapidement identifiées comme des fonctions couramment utilisées. Ainsi, les travaux que nous allons voir maintenant proposent des implémentations d'algorithmes permettant de résoudre le problème de *Pathfinding / Pathplanning* [15] dans un contexte agent sur GPU. 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 (2008) est le premier à proposer une implémentation des algorithmes *Dijkstra* [16] et $A*$ [17] sur GPU. Ceux-ci seront ensuite modifiés par Caggianese et Erra (2012) afin de s'adapter en temps réel tout au long de la simulation. Santos *et al.* (2012) apportent aussi leur contribution en proposant une nouvelle standardisation pour l'utilisation des GPU dans un contexte agent en respectant le standard FIPA (*Foundation for Intelligent Physical Agents*) afin de rendre possible la modélisation de comportements plus complexes. Pour tester cette approche, un cas d'étude a été implémenté : celui-ci consiste en une simulation de foule dans laquelle les agents utilisent l'algorithme $A*$ pour trouver leur chemin.

Cependant, ces algorithmes fonctionnent en ayant une représentation globale de l'environnement et sont connus pour donner des comportements peu réalistes. Pour considérer ce problème, des algorithmes centrés sur l'agent et basés sur des perceptions locales ont été proposés dans le cadre du GPGPU. On peut citer par exemple l'algorithme *BVP Planner* (Fischer *et al.*, 2009) 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*) (Bleiweiss, 2009) 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, Demeulemeester *et al.* (2011) proposent un algorithme qui donne la possibilité aux agents de définir des ROI (*Region Of Interest*) qui évoluent en même temps que les objectifs de ses agents. Présents dans de nombreux modèles multi-agents, les algorithmes de navigation sont un très bon exemple de la réutilisabilité qu'il est possible d'obtenir dès lors que l'on augmente la modularité des solutions trouvées. En effet, en dissociant les algorithmes des comportements des agents, il est alors possible de créer des algorithmes génériques pouvant s'adapter à de nombreux modèles et contextes différents ce qui améliore donc grandement leur réutilisation.

---

15. On appelle *PathFinding* ou *PathPlanning*, les techniques et algorithmes permettant de résoudre les problèmes de navigation des agents.

16. L'algorithme de *Dijkstra* permet de déterminer le plus court chemin dans un graphe en utilisant le poids lié aux arêtes.

17. L'algorithme $A*$ est une extension de *Dijkstra* et permet des recherches de chemin dans un graphe entre un nœud initial et final.

4.2. Bilan des implémentations tout-sur-GPU

On ne peut contredire que l'arrivée des environnements de développement spécialisés dans le GPGPU aient simplifié l'utilisation de cette technologie. De plus, lorsque le GPGPU est utilisé dans des simulations multi-agents, on observe, dans l'ensemble, de nettes accélérations des simulations (au minimum 2 fois plus rapides qu'une implémentation sur CPU), en particulier compte tenu du fait qu'elles ont été obtenues en utilisant des cartes graphiques standards comportant quelques centaines de cœurs.

Cependant 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, Aaby *et al*. (2010) montrent 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.

L'approche utilisée jusqu'à présent qui consiste à exécuter entièrement le modèle sur le GPU, approche *tout-sur-GPU*, a donc permis des avancées. Mais par contre, d'un point de vue génie logiciel, cette approche n'est pas adaptée car le but est de fournir des outils et *framework* réutilisables, modulaires, avec de bonnes performances et une accessibilité importante. Il est donc nécessaire, pour résoudre ces problèmes, de trouver une nouvelle approche. Nous allons voir maintenant que les systèmes hybrides représentent une alternative très intéressante pour la diffusion de la technologie GPGPU dans la communauté multi-agent.

4.3. Les approches hybrides comme solutions d'implémentation

Jusqu'à maintenant, nous avons recensé des contributions pour lesquelles le système est exécuté dans son ensemble par le GPU. Cette conception des simulations limite fortement les possibilités car il est alors nécessaire de repenser entièrement ces structure de données, la simultanéité des actions, les fonctions à implémenter (une fonction sur GPU se doit d'être indépendante et simple), etc. Pour palier ces désagréments, il est intéressant de considérer l'approche *hybride* qui consiste en une exécution répartie entre le CPU et le GPU d'un système multi-agents selon la nature des calculs et instructions (la figure 7 illustre cette approche). Moins performante qu'une approche *tout-sur-GPU*, l'approche hybride possède cependant de nombreux avantages.

Figure 7. Illustration d'une approche hybride

Par exemple, Sano et Fukuta (2013) proposent 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-Agents 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 navigation). 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 déjà existantes et éprouvées.

Pour Laville *et al*. (2012), 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 très 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 par Pavlov et Müller (2013). Leurs 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.

Michel (2013) présente un autre aspect de l'approche hybride. En considérant l'environnement comme une entité active, il est possible de simplifier le processus comportemental des agents. L'idée sous-jacente est que les agents ont finalement besoin de manipuler des percepts de haut niveau pour calculer leur comportement : ils ne sont pas intéressés par les données environnementales de bas niveau qui nécessitent un traitement pour être intelligibles. Il est donc intéressant de soulager les agents de ces traitements et de déléguer à l'environnement le soin de produire des perceptions de haut niveau à partir des données environnementales brutes (Chang *et al*., 2005 ; Payet

et al., 2006). Cette séparation permet de (1) réduire la complexité du modèle et surtout (2) faciliter la réutilisabilité et l'intégration des différents processus qui sont définis. Le principe de délégation GPU des perceptions agents (Michel, 2013) se place dans la continuité d'une approche de programmation dans l'environnement avec un délégation d'une partie des processus de l'agent. Cependant, elle essaye de répondre aussi aux problèmes de performance induits par cette déportation de la partie calculatoire. Ce principe considère toujours l'environnement comme une partie fondamentale du système ce qui permet d'atteindre deux objectifs : (1) conserver l'accessibilité du modèle agent dans un contexte GPU et (2) travailler avec un grand nombre d'agents avec des environnements plus importants. Le principal intérêt est de promouvoir une réutilisation simplifiée des modules GPU, car ils n'ont aucun lien avec le modèle d'agent qui les utilisera. Cette approche a permis d'intégrer le calcul sur GPU dans la plate forme TurtleKit (Michel *et al.*, 2005) sans avoir à modifier l'API du modèle agent grâce à une séparation claire entre le modèle agent et le modèle environnemental.

Laville *et al.* (2014) proposent un ensemble d'outils appelé MCMAS (*Many Core MAS*) qui doit permettre de faciliter l'implémentation de simulations multi-agents sur des architectures parallèles et ainsi mieux exploiter la puissance de ces dernières. MCMAS est donc une boite à outils composée de fonctions usuelles et de structures de données pouvant être utilisée comme une librairie par les plates-formes SMA existantes afin de simplifier l'adaptation des modèles existants et abstraire l'utilisation du GPU aux utilisateurs. Il est aussi très modulaire car il se veut facilement extensible par l'ajout de *plugins* afin de lui ajouter des fonctionnalités. Implémenté en Java et OpenCL, MCMAS est un exemple frappant de l'intérêt des approches hybrides. Il permet en effet de faire tourner des modèles (1) entièrement sur GPU, (2) entièrement sur CPU ou (3) en utilisant conjointement le CPU et le GPU. Les expériences réalisées dans ces travaux montrent que l'utilisation optimisée du GPU avec le CPU permet un gain de performance très important alors qu'une exécution uniquement sur GPU peut stagner au niveau d'une exécution sur CPU.

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, Vigueras *et al.* (2010) définissent 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.

4.4. Chronologie, synthèse des travaux mêlant GPGPU et simulations multi-agents

Afin d'avoir un aperçu global de l'intersection entre le GPGPU et les systèmes multi-agents, la figure 8 propose 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 là pour donner une idée de l'augmentation du nombre de cœurs au sein des GPU.

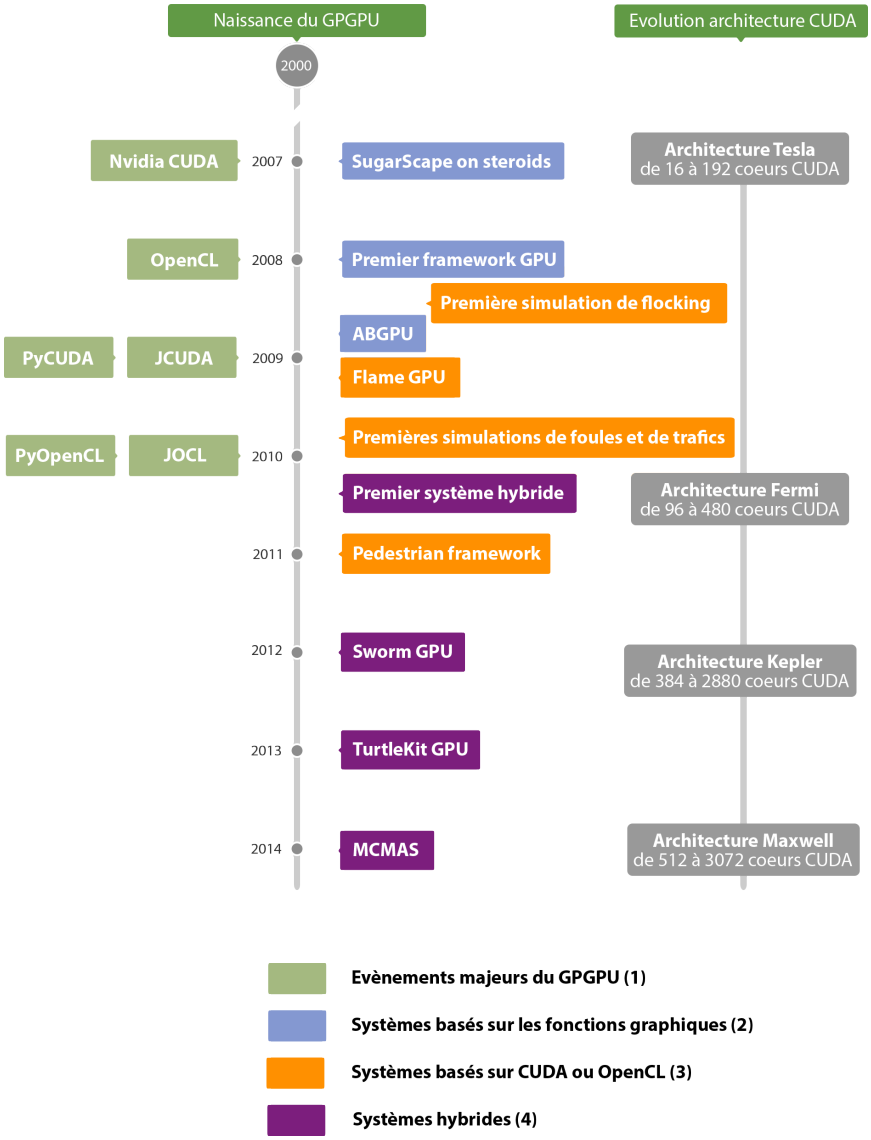

Figure 8. Chronologie de l'interaction entre SMA et GPGPU

Le tableau 1 est une synthèse regroupant les informations les plus importantes des différents *frameworks* et contributions vus au cours de cet état de l'art.

Tableau 1. Synthèse de l'état de l'art

Référence	Approche	Implémentation	Types d'agents	Comportements	GPU	Agents
(D'Souza *et al.*, 2007)	*Tout-sur-GPU*	Fonctions graphiques	Réactif	Très simples	128 cœurs	10^6
(Lysenko, D'Souza, 2008)	*Tout-sur-GPU*	Fonctions graphiques	Réactif	Très simples	128 cœurs	10^6
(Richmond, Romano, 2008)	*Tout-sur-GPU*	Fonctions graphiques	Réactif	Simples	112 cœurs	10^4
(Perumalla, Aaby, 2008)	*Tout-sur-GPU*	Fonctions graphiques	Réactif	Très simples	112 cœurs	10^6
(Richmond *et al.*, 2010)	*Tout-sur-GPU*	CUDA	Réactif et cognitif	Variables	256 cœurs	10^5
(Erra *et al.*, 2009)	*Tout-sur-GPU*	CUDA	Réactif	Très simples	128 cœurs	10^6
(Husselmann, Hawick, 2011)	*Tout-sur-GPU*	CUDA	Réactif et hétérogène	Très simples	512 cœurs	10^4
(Richmond, Romano, 2011)	*Tout-sur-GPU*	CUDA	Réactif et cognitif	Évolués	96 cœurs	10^4
(Bleiweiss, 2009)	*Tout-sur-GPU*	CUDA	-	-	240 cœurs	10^4
(Fischer *et al.*, 2009)	*Tout-sur-GPU*	CUDA	-	-	256 cœurs	10^3
(Laville *et al.*, 2012)	*Hybride*	OpenCL	Réactif et cognitif	Évolués	240 cœurs	10^3
(Pavlov, Müller, 2013)	*Hybride*	CUDA	Réactif et cognitif	Évolués	-	-
(Michel, 2013)	*Hybride*	CUDA (JCUDA)	Réactif et cognitif	Évolués	256 cœurs	10^3
(Laville *et al.*, 2014)	*Hybride*	OpenCL	Réactif et cognitif	Évolués	240 cœurs	10^3

5. Généricité et accessibilité des travaux existants

De l'état de l'art que nous venons de présenter, il est clair que le GPGPU est une technologie d'avenir pour les simulations multi-agents mais aussi pour les SMA. Pourtant, il est également évident que l'utilisation de cette technologie dans le cadre d'une programmation agent reste une tâche difficile. Nous discernons deux raisons principales qui expliquent ces difficultés : (1) le faible degré de généricité des modèles considérés et (2) le manque d'accessibilité des *frameworks* existants. Ces deux problématiques ont d'ailleurs été identifiées par Holk *et al.* (2011) et Bourgoin *et al.* (2014) comme cruciales pour le développement du GPGPU en général. Dans cette section, nous proposons une étude des différentes contributions à l'égard de la façon dont ils ont tenu compte de ces deux aspects.

5.1. 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 passante, 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. Dans ce contexte, *ABGPU* a été le premier *framework* à mettre en avant la généricité en généralisant des comportements agents communs (Richmond, Romano, 2008).

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.

Flame GPU est une exception remarquable car il fournit des modèles d'agents prédéfinies qui peuvent être adaptés à différents domaines d'application tels que la biologie (*e.g.* Richmond *et al.* (2009a) et Richmond *et al.* (2010)) ou les sciences comportementales (*e.g.* Karmakharm *et al.* (2010)). Cependant, 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.* Richmond et Romano (2011)) ou hétérogènes (*e.g.* D'Souza *et al.* (2009) et Husselmann et Hawick (2011)).

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 (Richmond, Romano, 2011) et (2) d'algorithmes de mouvement (Bleiweiss, 2009 ; Fischer *et al.*, 2009 ; Demeulemeester *et al.*, 2011). 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 plusieurs domaines.

L'approche hybride permet de faire un pas en avant vers la réalisation de modèles agents plus complexes. Tout d'abord, cette approche permet de créer plus facilement des modèles dans lesquels les agents vont avoir des architectures différentes (par exemple cognitive et réactive (Laville *et al.*, 2012). Deuxièmement, une approche hybride possède une grande modularité ce qui facilite, entre autres, la séparation explicite entre agents et environnements (Michel, 2013 ; Pavlov, Müller, 2013).

Ainsi, même si le caractère générique n'est pas nécessairement un objectif explicite des systèmes hybrides, il est clair que cette approche présente une architecture logicielle mettant en avant la modularité et la réutilisabilité. La librairie MCMAS

(Laville *et al*., 2014) en est un exemple marquant : l'objectif de cette librairie est de proposer une structure de programmation générique pour les simulation multi-agents sur GPU. Il devrait être ainsi possible d'envisager, dans un futur proche, l'implémentation d'agents possédant des architectures plus complexes, par exemple BDI (*Beliefs-Desires-Intentions*), en déléguant une partie de leur comportement sur GPU grâce à ce genre de librairies.

5.2. 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 comme l'expliquent Perumalla et Aaby (2008). Lysenko et D'Souza (2008) et *ABGPU* (Richmond, Romano, 2008) 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* (Richmond *et al*., 2010) et les travaux de Michel (2013) et Laville *et al*. (2012) rendent son utilisation transparente.

Ainsi, même si l'objectif n'est pas d'abstraire entièrement l'utilisation du GPU, 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 *Path-Planning* (Fischer *et al*., 2009 ; Bleiweiss, 2009 ; Demeulemeester *et al*., 2011)), 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. On peut citer pour exemple les librairies Nvidia *CuBLAS* (*Compute Unified Basic Linear Algebra Subprograms*) et *NPP* (*Nvidia Performance Primitives*) spécialisées dans le traitement de signaux, d'images et de vidéos ou encore *cuFFT* (*CUDA Fast Fourier Transform*) pour le calcul des transformées de Fourier. Elles sont très abouties et leur utilisation est largement répandue dans leur communauté respective.

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, comme c'est le cas avec MCMAS (Laville *et al*., 2014). De plus, une approche hybride est par nature ouverte aux autres technologies car elle lève les

contraintes du *tout-sur-GPU*. Par exemple, (Laville *et al.*, 2012) et (Michel, 2013) utilisent la programmation orientée objet en parallèle du GPGPU.

6. Conclusion et perspectives

De l'état de l'art que nous avons réalisé dans le domaine des SMA, le GPGPU représente une technologie très intéressante pour toutes les applications 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 la Nvidia Tesla K40 en contient 2 880 et la Nvidia Tesla K10 en contient 3 072 (deux GPU de 1 536 cœurs sont présents sur cette dernière). 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.

Les travaux sur l'utilisation du GPGPU pour l'allocation de tâches, réalisés dans (Pavlov, Müller, 2013), 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-agent dès lors que l'accessibilité et la généricité des outils couplant GPGPU et SMA seront matures.

À l'image des travaux de Bourgoin *et al.* (2014) 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-agent. 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 API de programmation classiques, 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) de manière simple. Adapter le concept des artefacts (Ricci *et al.*, 2011) au contexte GPGPU en créant des artefacts GPU intégrés dans l'environnement pourrait

aussi permettre de mettre en place un modèle de programmation générique englobant les fonctionnalités GPU auxquelles les agents vont avoir accès.

Il existe donc de nombreuses pistes de recherche pour améliorer l'utilisation du GPGPU dans un contexte SMA. Pour aller plus loin, on peut cependant remarquer que la grande majorité des travaux présentés reposent sur l'adaptation au GPGPU de modèles multi-agents déjà existants, dont l'architecture est donc par essence pensée pour le séquentiel. Il est donc finalement assez naturel que la communauté rencontre des difficultés à les rendre compatibles avec cette nouvelle technologie. Il faut garder à l'esprit que le GPGPU et la programmation sur CPU sont radicalement différents. Mais, bien que ceci constitue aujourd'hui une difficulté majeure, nous pensons qu'il est aussi pertinent de le voir comme une formidable opportunité de repenser notre manière de modéliser et de comprendre les SMA. Ce tournant technologique doit en effet être l'occasion de définir de nouveaux modèles multi-agents qui auront la particularité d'être pensés, avant tout, pour être exécutés dans un contexte matériel composé d'architectures massivement parallèles. Cette dernière perspective de recherche nous paraît avoir un potentiel d'innovation particulièrement prometteur pour les années qui viennent.

Bibliographie

Aaby B. G., Perumalla K. S., Seal S. K. (2010). Efficient Simulation of Agent-based Models on multi-GPU and Multi-core Clusters. In *Proceedings of the 3rd international icst conference on simulation tools and techniques*, p. 29:1-29:10. ICST, Brussels, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). Consulté sur http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8822

Bleiweiss A. (2008). GPU accelerated pathfinding. In *Proceedings of the 23rd acm siggraph/eurographics symposium on graphics hardware*, p. 65-74. Aire-la-Ville, Switzerland, Eurographics Association. Consulté sur http://dl.acm.org/citation.cfm?id=1413957.1413968

Bleiweiss A. (2009). Multi agent navigation on the GPU. *Games Developpement Conference*. Consulté sur http://www.cs.uu.nl/docs/vakken/mcrs/papers/28.pdf

Bourgoin M., Chailloux E., Lamotte J.-L. (2014). Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, vol. 42, n° 4, p. 583-600. Consulté sur http://dx.doi.org/10.1007/s10766-013-0261-x

Caggianese G., Erra U. (2012). GPU Accelerated Multi-agent Path Planning Based on Grid Space Decomposition. In *Proceedings of the international conference on computational science*, vol. 9, p. 1847–1856. Elsevier. Consulté sur http://linkinghub.elsevier.com/retrieve/pii/S1877050912003249

Chang P. H., Chen K.-T., Chien Y.-H., Kao E., Soo V.-W. (2005). From reality to mind: A cognitive middle layer of environment concepts for believable agents. In D. Weyns, H. V. D. Parunak, F. Michel (Eds.), *Environments for multi-agent systems, first international workshop, e4mas 2004, new york, ny, usa, july 19, 2004, revised selected papers*, vol. 3374, p. 57-73. Springer.

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. (General-Purpose Processing using Graphics Processing Units)

Coakley S., Smallwood R., Holcombe M. (2006). Using x-machines as a formal basis for describing agents in agent-based modelling. *Proceedings of 2006 Spring Simulation Multiconference*, p. 33-40. Consulté sur http://staffwww.dcs.shef.ac.uk/people/S.Coakley/media/pdf/ADS005_Coakley.pdf

Demeulemeester A., Hollemeersch C.-F., Mees P., Pieters B., Lambert P., Walle R. Van de. (2011). Hybrid Path Planning for Massive Crowd Simulation on the GPU. In J. Allbeck, P. Faloutsos (Eds.), *Motion in games*, vol. 7060, p. 304-315. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-25090-3_26

D'Souza R. M., Lysenko M., Marino S., Kirschner D. (2009). Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 spring simulation multiconference*, p. 21:1-21:12. San Diego, CA, USA, Society for Computer Simulation International.

D'Souza R. M., Lysenko M., Rahmani K. (2007). SugarScape on steroids: simulating over a million agents at interactive rates. *Proceedings of Agent 2007 conference*.

Erra U., Frola B., Scarano V., Couzin I. (2009, Oct). An Efficient GPU Implementation for Large Scale Individual-Based Simulation of Collective Behavior. In *High performance computational systems biology, 2009. hibi '09. international workshop on*, p. 51-58. Consulté sur http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5298705

Ferber J. (1999). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence* (1st éd.). Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Fischer L. G., Silveira R., Nedel L. (2009). GPU accelerated path-planning for multi-agents in virtual environments. In *Proceedings of the 2009 viii brazilian symposium on games and digital entertainment*, p. 101–110. IEEE Computer Society. Consulté sur http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5479104

Flynn M. (1972, Sept). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, vol. C-21, n° 9, p. 948-960.

Helbing D., Farkas I. J., Molnar P., Vicsek T. (2002). Simulation of pedestrian crowds in normal and evacuation situations. *Pedestrian and Evacuation Dynamics (Berlin, Germany)*, p. 21-58.

Holk E., Byrd W. E., Mahajan N., Willcock J., Chauhan A., Lumsdaine A. (2011). Declarative Parallel Programming for GPUs. In *Applications, tools and techniques on the road to exascale computing, proceedings of the conference parco 2011, 31 august - 3 september 2011, ghent, belgium*, vol. 22, p. 297-304. IOS Press. Consulté sur http://dx.doi.org/10.3233/978-1-61499-041-3-297

Husselmann A. V., Hawick K. A. (2011). Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUs. In *International conference on parallel and distributed computing and systems*, p. 100-107. IASTED. Consulté sur http://www.actapress.com/PaperInfo.aspx?paperId=453044

Karmakharm T., Richmond P. (2012). Large Scale Pedestrian Multi-Simulation for a Decision Support Tool. In *Theory and practice of computer graphics, rutherford, united kingdom,*

2012. proceedings, p. 41-44. Eurographics Association. Consulté sur http://dx.doi.org/10.2312/LocalChapterEvents/TPCG/TPCG12/041-044

Karmakharm T., Richmond P., Romano D. M. (2010). Agent-based Large Scale Simulation of Pedestrians With Adaptive Realistic Navigation Vector Fields. In *Theory and practice of computer graphics, sheffield, united kingdom, 2010. proceedings*, p. 67-74. Eurographics Association. Consulté sur http://dx.doi.org/10.2312/LocalChapterEvents/TPCG/TPCG10/067-074

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

Li H., Kolpas A., Petzold L., Moehlis J. (2009, avril). Parallel Simulation for a Fish Schooling Model on a General-purpose Graphics Processing Unit. *Concurrency and Computation: Practice and Experience*, vol. 21, n° 6, p. 725-737. Consulté sur http://dx.doi.org/10.1002/cpe.v21:6

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. (2013). 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, n° 6, p. 703-715. Consulté sur http://dx.doi.org/10.1002/sres.2239

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.

Odell J., Van Dyke Parunak H., Fleischer M., Brueckner S. (2003). Modeling Agents and Their Environment. In F. Giunchiglia, J. Odell, G. Weiß (Eds.), *Agent-oriented software engineering iii*, vol. 2585, p. 16-31. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/3-540-36540-0_2

Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger 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

Passos E., Joselli M., Zamith M. (2008). Supermassive crowd simulation on GPU based on emergent behavior. In *In proceedings of the seventh brazilian symposium on computer games and digital entertainment*, p. 81-86. Consulté sur http://www.sbgames.org/papers/sbgames08/cp/paper/p10.pdf

Pavlov R., Müller J. (2013). Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors. In L. Camarinha-Matos, S. Tomic, P. Graça (Eds.), *Technological innovation for the internet of things*, vol. 394, p. 115-122. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-37291-9_13

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.

Perumalla K. S., Aaby B. G. (2008). Data parallel execution challenges and runtime performance of agent simulations on GPUs. *Proceedings of the 2008 Spring simulation multiconference*, p. 116-123. Consulté sur http://dl.acm.org/citation.cfm?id=1400564

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., Coakley S., Romano D. M. (2009a). Cellular Level Agent Based Modelling on the Graphics Processing Unit. In *2009 international workshop on high performance computational systems biology*, p. 43-50. IEEE. Consulté sur http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5298704

Richmond P., Coakley S., Romano D. M. (2009b). A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA. In *Proceedings of the 8th international conference on autonomous agents and multiagent systems - volume 2*, vol. 2, p. 1125-1126. Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems. Consulté sur http://dl.acm.org/citation.cfm?id=1558109.1558172

Richmond P., Romano D. M. (2008). Agent based GPU, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the GPU. In *In proceedings international workshop on super visualisation (iwsv08).* Consulté sur http://www.flame.ac.uk/pubs/pdf/10.1.1.144.734.pdf

Richmond P., Romano D. M. (2011). A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. *European Simulation and Modelling*. Consulté sur http://www.paulrichmond.staff.shef.ac.uk/publications/pedestrians.pdf

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

Romano D. M., Sheppard G., Hall J., Miller A., Ma Z. (2005). BASIC: A Believable Adaptable Socially Intelligent Character for Social Presence. In *Presence 2005, the 8th annual international workshop on presence*, p. 21-22.

Sano Y., Fukuta N. (2013). A GPU-based Framework for Large-scale Multi-Agent Traffic Simulations. In *Proceedings of the 2013 second iiai international conference on advanced applied informatics*, p. 262–267.

Santos L. dos, Gonzales Clua E., Bernardini F. (2012). A Parallel Fipa Architecture Based on GPU for Games and Real Time Simulations. In M. Herrlich, R. Malaka, M. Masuch (Eds.), *Entertainment computing - icec 2012*, vol. 7522, p. 306-317. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-33542-6_26

Shen Z., Wang K., Zhu F. (2011, Oct). Agent-based traffic simulation and traffic signal timing optimization with GPU. In *Intelligent transportation systems (itsc), 2011 14th international ieee conference on*, p. 145-150. Ieee. Consulté sur http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6083080

Silva A. R. D., Lages W. S., Chaimowicz L. (2010, jan). Boids That See: Using Self-occlusion for Simulating Large Groups on GPUs. *Comput. Entertain.*, vol. 7, n° 4, p. 51:1-51:20. Consulté sur http://doi.acm.org/10.1145/1658866.1658870

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

Strippgen D., Nagel K. (2009, June). Multi-agent traffic simulation with CUDA. In *High performance computing simulation, 2009. hpcs '09. international conference on*, p. 106-114.

Varga M., Mintal M. (2014, Jan). Microscopic pedestrian movement model utilizing parallel computations. In *Applied machine intelligence and informatics (sami), 2014 ieee 12th international symposium on*, p. 221-226.

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

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