Modélisation et implémentation de simulations multi-agents sur architectures massivement parallèles

Emmanuel Hermellin

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM**

Spécialité: **Informatique**

Présentée par **Emmanuel HERMELLIN**

Modélisation et implémentation de simulations multi-agents sur architectures massivement parallèles

Soutenue le 18 novembre 2016 devant le jury composé de

M. Olivier SIMONIN	Professeur	INSA, Université de Lyon	Président du jury
M. Fabien MICHEL	Maître de conférences HDR	Université de Montpellier	Directeur
M. Jacques FERBER	Professeur	Université de Montpellier	Co-directeur
M. Vincent CHEVRIER	Professeur	Université de Lorraine	Rapporteur
M. Alessandro RICCI	Professeur	Université de Bologne	Rapporteur
M. Jean-Christophe SOULIÉ	Cadre de recherche HDR	Cirad, Montpellier	Examinateur

Université Montpellier

LIRMM

Université de Montpellier

Spécialité doctorale “Informatique”

Thèse présentée et soutenue publiquement par

Emmanuel HERMELLIN

le 18 novembre 2016

Modélisation et implémentation de simulations multi-agents sur architectures massivement parallèles

sous la direction de
Fabien MICHEL
Jacques FERBER

Rapporteurs
Vincent CHEVRIER
Alessandro RICCI

Examinateurs
Olivier SIMONIN
Jean-Christophe SOULIÉ

« *Pour tirer le meilleur parti des connaissances acquises, pour en extraire toute la richesse, il importe de ne pas s'y habituer trop vite, de se laisser le temps de la surprise et de l'étonnement.* »

Hubert Reeves

Quelques mots de remerciements...

Merci Fabien de t'être battu pour me permettre d'obtenir cette bourse et ainsi avoir la chance de vivre cette expérience unique qu'est une thèse. Merci pour tes conseils, ton investissement et ta disponibilité. C'est ton encadrement qui a permis d'arriver aux résultats que l'on a obtenu.

Une pensée pour toute l'équipe SMILE, une équipe unique et atypique composée de personnes attachantes et humaines.

Merci Nicolas pour la bienveillance dont tu fais preuve auprès de chacun de tes petits doctorants. Toutes nos réussites sont un peu grâce à toi.

Merci à tous les doctorants du LIRMM pour tous ces moments passés ensemble : les bons comme les mauvais. Et tout particulièrement, une grosse pensée pour Adel, Guillerme, Julien et Vincent[1] *avec qui les débats et discussions animés chaque midi au RA (mais pas que) ont rendu ces trois années inoubliables !*

Mais surtout, merci à ma famille et à toutes les personnes qui comptent...

1. Trouvez l'intrus

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

PRÉFACE

Depuis la naissance des systèmes multi-agents (SMA), on a reconnu la possibilité de distribuer le fardeau de leur calcul sur plusieurs processeurs (CPU). Et, en fait, dans les domaines robotiques, cette distribution est automatique parce que les robots sont distincts les uns des autres. Mais de plus en plus, les SMA servent pour la simulation ou même l'optimisation, applications qui n'imposent aucune distribution intégrale sur plusieurs CPU, mais dans lesquelles la vitesse de calcul est encore plus importante que dans la robotique. Bien que les méthodes soient bien développées pour la simulation numérique distribuée, les SMA n'ont pas profité de ces avancées. Malgré quelques efforts provisoires, la plupart des chercheurs continuent de travailler avec des agents qui sont limités par un seul processeur de type von Neumann. *Quand bien même, des processeurs avec plusieurs cœurs d'exécutions sont de plus en plus accessibles par exemple grâce au GPGPU.*

L'œuvre du Dr. Hermellin est la première vue d'ensemble de ce problème. Il a soigneusement sondé les œuvres précédentes, deviné les verrous qui gênent la diffusion du GPGPU dans les SMA, proposé une résolution qui est à la fois pratique et théoriquement élégante, et évalué cette solution en détail du point de vue de sa performance dans plusieurs problèmes bien connus. Sa thèse est destinée à devenir un guide principal pour les chercheurs et ingénieurs dans le domaine des SMA.

Cette œuvre s'intéresse également à l'environnement comme une abstraction de premier ordre dans les SMA, qui était exposée dans l'équipe du Prof. Ferber, et en particulier par le Dr. Michel, les directeurs de cette thèse. On mesure la valeur d'une théorie scientifique par ses prédictions et par la clarté qu'elle apporte aux analyses subséquentes. Les résultats de Dr. Hermellin nous donnent de l'évidence formidable pour des conceptions centrées sur l'environnement. Pour nous, qui avons souligné depuis longtemps l'importance de l'environnement, voici une confirmation forte de notre intuition.

Désormais, cette thèse deviendra une référence essentielle pour la conception des SMA. J'offre mes félicitations au Dr. Hermellin, et aussi à ses encadrants le Dr. Michel et le Prof. Ferber, pour cette contribution éclatante.

H. Van Dyke Parunak, Ph.D.
Président et Chief Scientist
ABC Research, LLC
Ann Arbor, MI, USA
le 14 novembre 2016

INTRODUCTION

À l'intersection entre système multi-agent (SMA) et simulation numérique, la simulation multi-agent constitue une solution pertinente pour l'ingénierie et l'étude des systèmes complexes dans de nombreux domaines (vie artificielle, biologie, économie, etc.) grâce à sa représentation individu-centrée des systèmes considérés. En effet, la simulation multi-agent met en œuvre des modèles où les individus (des agents autonomes), leur environnement et leurs interactions sont directement représentés. Ainsi, la description de la dynamique du système ne se fait plus en déclarant des équations portant sur des propriétés macroscopiques de la simulation, mais sur le comportement individuel de chaque entité.

La simulation multi-agent est de plus en plus répandue car elle est devenue, au même titre que la simulation numérique, un outil indispensable de compréhension des phénomènes rivalisant avec l'expérimentation et le prototypage mais pour un coût bien moindre. Elle est ainsi utilisée par une grande partie de la communauté scientifique mais aussi par de nombreux acteurs industriels et repose sur des savoir-faire pointus en modélisation et en programmation.

Cependant, la simulation numérique requiert parfois énormément de ressources de calcul, ce qui représente un verrou technologique majeur qui restreint, dans le cas de la simulation multi-agent, les possibilités d'étude des modèles envisagés (passage à l'échelle, expressivité des modèles proposés, interaction temps réel, etc.). Pour pallier ces problèmes de performance, il a donc été nécessaire de se tourner vers des architectures matérielles et des solutions logicielles capables de traiter et/ou calculer une quantité de données toujours plus grande.

C'est donc guidé par le besoin des utilisateurs de simuler des problèmes toujours plus complexes, que le domaine de la simulation numérique s'est fortement lié à celui du calcul haute performance [2] (aussi appelé calcul intensif ou HPC, de l'anglais *High Performance Computing*). Cette association a ainsi permis de remplacer, en partie, les expériences qui ne peuvent être menées en laboratoire quand elles sont dangereuses (accidents), de longue durée (climatologie), inaccessibles (astrophysique) ou interdites (essais nucléaires) tout en améliorant également la productivité de ces dernières grâce à des gains de temps d'exécution important.

La météorologie est un exemple très représentatif des possibilités offertes par le calcul haute performance car elle est une des sciences les plus gourmandes en termes de besoins en ressources de calcul de ces dernières années. Ainsi, l'utilisation de super-ordinateurs a permis d'augmenter la finesse des modèles et d'avoir une résolution de l'ordre du kilomètre pour les prévisions météorologiques les plus performantes. Cette évolution est d'autant plus impressionnante lorsque l'on repense à la précision de ces mêmes prévisions et modèles juste 30 ans plus tôt. On comprend alors pourquoi Météo-France possède quelques-uns des super-ordinateurs les plus puissants de France [3].

Cependant, l'architecture matérielle des super-ordinateurs est bien différente de celle de nos ordinateurs personnels (PC, de l'anglais *Personnal Computer*). Massivement parallèles, ces architectures s'inscrivent dans le mouvement *many-core* initié depuis la fin du XX^{e} siècle qui consiste à multiplier le nombre de cœurs et/ou de processeurs au sein d'une machine. De ce fait, le développement de ces super-ordinateurs ainsi que leur coût de maintenance représentent des investisse-

2. Le calcul haute performance désigne le domaine dédié au développement des architectures matérielles et ordinateurs ainsi que des méthodes et techniques de programmation massivement parallèles qui leurs sont associées, capables de faire du calcul intensif sur de gros volumes de données.

3. Plus de détails sur `http://www.cnrm-game-meteo.fr/spip.php?rubrique68&lang=fr`.

ments pouvant atteindre plusieurs millions d'euros. Ce coût très important d'exploitation restreint fortement l'accessibilité de ces super-ordinateurs limitant ainsi leur utilisation aux grands groupes et entreprises possédant les ressources nécessaires à leur fonctionnement [4].

C'est dans ce contexte que d'autres architectures matérielles dédiées au calcul intensif ont vu le jour. Avec comme objectif d'offrir le meilleur rapport en termes de performance, prix et consommation, les cartes graphiques (GPU, de l'anglais *Graphics Processing Units*) ont commencé à percer dès les années 2000 car ces dernières proposent des solutions de calculs peu coûteuses profitant de l'augmentation quasi exponentielle du nombre de cœurs (plusieurs milliers pour la dernière architecture *Pascal* de Nvidia) et de la puissance de leurs puces. Initialement utilisées dans le jeu vidéo, ces cartes sont en train de devenir un choix apprécié pour ceux qui cherchent à concevoir des machines à haute performance tout en contenant le facteur énergétique. De plus, ces cartes étant disponibles aussi bien pour les professionnels que pour le grand public, elles ouvrent de nouvelles perspectives de recherches en permettant aux plus grands nombres d'accéder à des solutions de calculs intensifs.

Il existe de nombreux exemples concrets soulignant les différents avantages apportés par l'utilisation du GPGPU (de l'anglais *General-Purpose computing on Graphics Processing Units*, qui désigne la technologie permettant d'utiliser les cartes graphiques comme accélérateur de calcul). Pour le grand public, la plupart des applications majeures dédiées à la photographie et à la vidéo utilisent le GPGPU, via des solutions *open-source* (dans les logiciels de la société Adobe et DxO par exemple) ou via des solutions propriétaires (dans les produits d'Elemental Technologies ou MotionDSP), permettant ainsi d'accélérer grandement les temps de traitements et de rendus. En matière de finance, Numerix et CompatibL ont intégré cette technologie dans leur application de recherche de risques de contrepartie, accélérant jusqu'à 18× les procédures de calcul existantes. La plateforme Numerix est utilisée par plus de 400 institutions financières. Du côté de la recherche scientifique, le GPGPU a permis d'améliorer, entre autre, les performances d'un programme de simulation de dynamique moléculaire (AMBER) utilisé par plus de 60 000 chercheurs du public et du privé afin d'accélérer la découverte de nouveaux médicaments pour l'industrie pharmaceutique. Le GPGPU a aussi permis l'émergence de domaines de recherche jusqu'alors peu considéré par manque de solutions dédiées au calcul intensif. C'est le cas actuellement pour le domaine du *Big Data*, de la *Réalité Virtuelle* ou encore du *Deep Learning*. D'ailleurs, un exemple symbolique de l'utilisation du GPGPU pour effectuer du *Deep Learning* est celui du logiciel *Srez* [5] (en cours de développement) qui reconstitue des photos de visages pixelisés par apprentissage automatique. Ce logiciel montre qu'il est possible d'utiliser une carte graphique grand public et des outils relativement faciles à maîtriser pour profiter d'une puissance qui était réservée à des super-ordinateurs il y a seulement quelques années.

Néanmoins, alors que l'adoption du GPGPU croît fortement dans de nombreux domaines, cette technologie peine à s'imposer dans celui des systèmes multi-agents (et par extension aux simulations multi-agents). En effet, la communauté multi-agent fait preuve d'un relatif immobilisme face au GPGPU et cela malgré les problèmes de performances que l'on peut rencontrer lorsque l'on souhaite simuler des modèles multi-agents. En fait, il s'avère que le GPGPU s'accompagne d'un contexte de développement très spécifique qui nécessite une transformation profonde et non triviale des modèles multi-agents. Ainsi, malgré l'existence de travaux pionniers qui démontrent l'intérêt du GPGPU, cette difficulté explique, en partie, le faible engouement de la communauté multi-agent pour le GPGPU.

Dans cette thèse, nous montrons que, parmi les travaux qui visent à faciliter l'usage du GPGPU dans un contexte agent, la plupart le font au travers d'une utilisation transparente de cette technologie. Cependant, cette approche nécessite d'abstraire un certain nombre de parties du modèle, ce qui limite fortement le champ d'application des solutions proposées. Pour pallier ce problème,

4. La gestion de la forte consommation d'électricité de ces super-ordinateurs est aussi devenue une priorité car ces machines peuvent consommer plus de 10 mégawatts (ce qui équivaut à l'électricité nécessaire pour approvisionner 30 000 foyers en électricité) augmentant de ce fait les coûts d'exploitation (*source* : `http://www.top500.org`).

5. `https://github.com/david-gpu/srez`

et au contraire des solutions existantes réalisées de manière *ad hoc* et non réutilisables, nous proposons d'utiliser cette technologie grâce à une approche basée sur du génie logiciel orienté agent (environnement-centré précisément) et hybride (l'exécution de la simulation est partagée entre le processeur et la carte graphique). L'objectif étant de mettre l'accent sur l'accessibilité et la réutilisabilité grâce à une modélisation qui permet une utilisation directe et facilitée des paradigmes de programmation massivement parallèles, le GPGPU n'étant qu'une instance de ces derniers.

Plan de la thèse

Ainsi, dans cette thèse, nous souhaitons proposer une approche de modélisation qui facilite l'utilisation des architectures massivement parallèles, quelles qu'elles soient, sur des modèles multi-agents. Pour ce faire, le document est structuré en quatre grandes parties et huit chapitres (le plan est illustré par la figure 1). La première introduit les différents domaines sur lesquels reposent ce travail de thèse. La deuxième présente nos travaux réalisés dans le cadre de l'utilisation du GPGPU pour les simulations multi-agents. La troisième conclut ce travail en explorant les conséquences liées à l'utilisation des paradigmes de programmation massivement parallèles dans les simulations multi-agents et discute des perspectives associées. Enfin, la quatrième contient des précisions et travaux annexes aux contributions de cette thèse.

Partie I : contexte

Le chapitre 1 présente le domaine des systèmes mutli-agents en se focalisant sur les simulations multi-agents et sur les problèmes de performances inhérents à ce domaine. Le chapitre 2 introduit le calcul haute performance et en particulier le GPGPU. Ce chapitre discute également des verrous d'utilisation liés à cette technologie. Enfin, le chapitre 3 propose un état de l'art de l'utilisation du GPGPU pour la modélisation et le développement de simulations multi-agents et pose les problématiques relatives à ce travail de thèse.

Partie II : contributions

Le chapitre 4 décrit, par l'expérience, le principe de délégation GPU : une approche proposée pour répondre aux problématiques soulevées par l'utilisation du GPGPU dans le cadre des simulations multi-agents. Le chapitre 5 expérimente le principe de délégation GPU sur plusieurs cas d'étude et évalue son intérêt vis à vis des problématiques et des objectifs fixés. Le chapitre 6 présente une généralisation de nos travaux sous la forme d'une méthode de conception de simulations multi-agents basée sur le principe de délégation GPU.

Partie III : conclusion et perspectives

Pour finir, le chapitre 7 dresse les conclusions de notre travail de thèse et le chapitre 8 introduit quelques-unes des perspectives de recherche, à court et moyen termes, associées à ce travail.

Partie IV : annexes

L'annexe A présente l'architecture de la plate-forme de simulation multi-agent TurtleKit avant et après intégration du GPGPU. L'annexe B détaille certaines notions autour de la programmation avec CUDA. L'annexe C propose une solution d'optimisation basée sur OpenGL pour le rendu graphique dans la plate-forme TurtleKit.

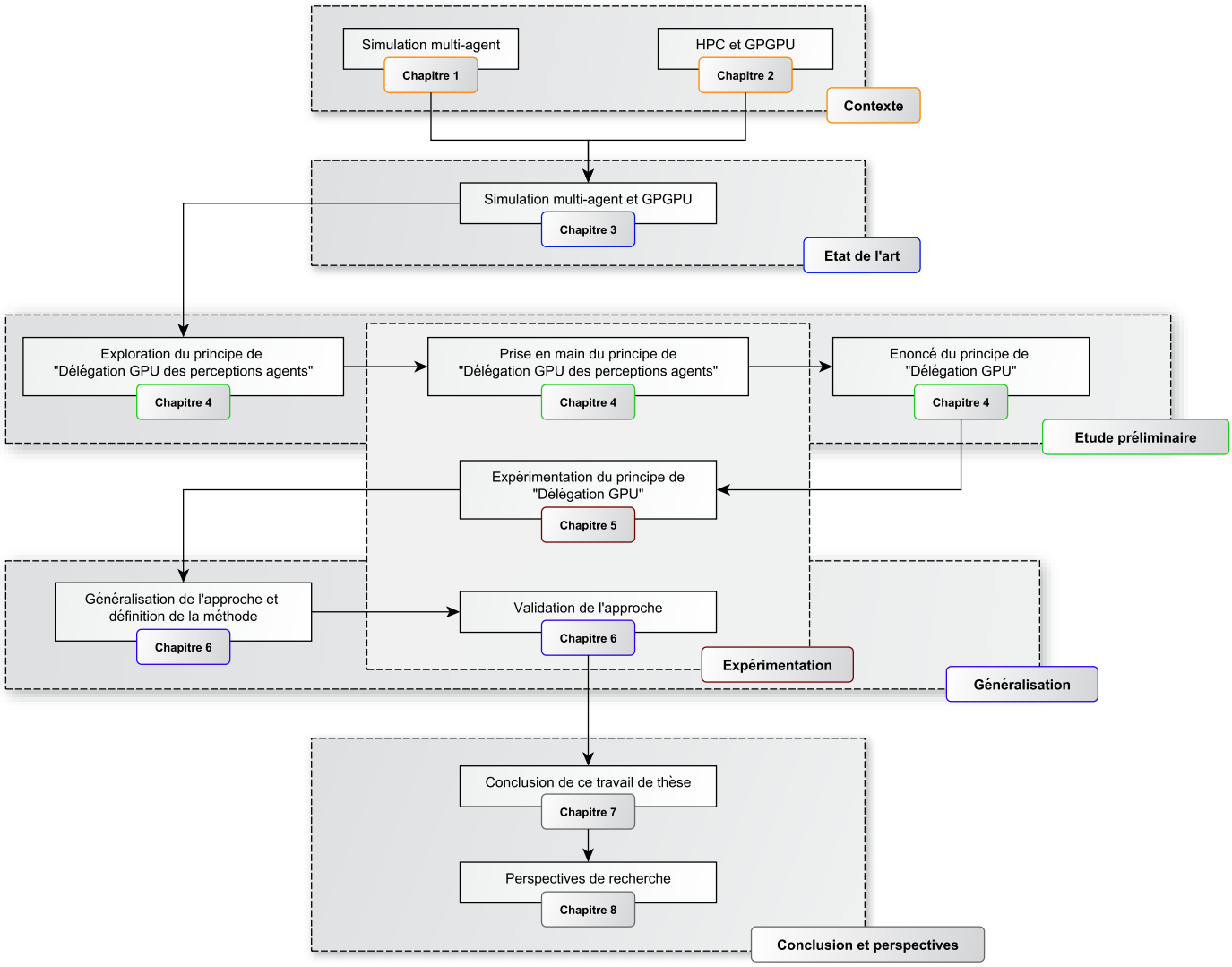


FIGURE 1 – Plan du manuscrit de thèse.

PREMIÈRE PARTIE

CONTEXTE DE LA THÈSE

CHAPITRE 1

SIMULATION MULTI-AGENT

Sommaire

Le domaine des Systèmes Multi-Agents (SMA) est né du besoin grandissant en communication et en distribution pour la résolution de problèmes en Intelligence Artificielle (IA) (*e.g.* architecture de type tableau noir [Hayes-Roth, 1985]) et de l'évolution des langages de programmation [Hewitt, 1977] (problèmes de programmation orientée objet et de programmation distribuée). Les travaux de recherche tels que les *acteurs* de HEWITT [Hewitt et al., 1973], le système DVMT (*Distributed Monitoring Vehicle Testbed*) de LESSER [Lesser and Corkill, 1983], le *Contract Net* de SMITH [Smith, 1980] sont autant d'exemples qui ont renforcé l'idée qu'il était pertinent de décentraliser les traitements et les données du fait que ces problèmes considérés étaient eux-mêmes de nature distribuée.

Ainsi, les systèmes multi-agents s'inscrivent dans le cadre de l'Intelligence Artificielle Distribuée (IAD) [Erceau and Ferber, 1991, Sycara, 1998] et trouvent des applications dans de très nombreux domaines : objets communicants, systèmes informatiques distribués, segmentation d'images, simulations de catastrophes, d'interactions sociales, de réseau routier, de propagation d'épidémie, de réaction en chaîne, etc. En fait, toute situation dans laquelle on peut identifier des entités précises bien définies et en interaction.

Dans le cadre de notre travail de thèse, nous utilisons un champ d'application particulier des systèmes multi-agents : *la simulation* qui a pour but la modélisation et l'étude du comportement de systèmes complexes. Notre objectif est de répondre aux problématiques de performance inhérentes à ce domaine en proposant des solutions de modélisations et d'implémentations dédiées aux simulations multi-agents. Nous discuterons également de la possibilité d'étendre les propositions faites au domaine des systèmes multi-agents en général.

1.1 Les systèmes multi-agents

On peut définir un système multi-agent comme étant un système composé d'un ensemble d'entités autonomes, que l'on appelle agents, situés dans un certain environnement et interagissant selon certaines relations. D'une manière plus formelle, [Ferber, 1995] caractérise un système multi-agent par :

— Un environnement (E) ;

— Un ensemble d'objets (O) situés dans E. Ces objets peuvent être perçus, créés, détruits et modifiés par les agents ;

— Un ensemble d'agents (A), qui sont des objets particuliers ($A \subseteq O$), lesquels représentent les entités actives du système ;

— Un ensemble de relations (R) qui unissent des objets (et donc des agents) entre eux.

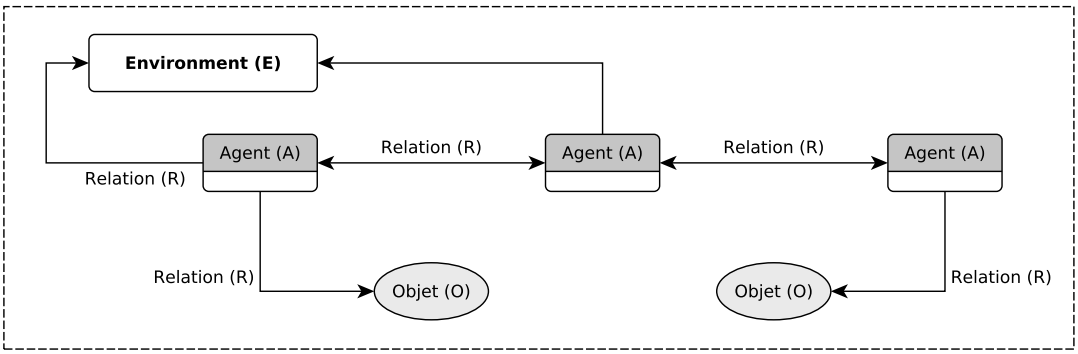


FIGURE 1.1 – Représentation d'un système multi-agent d'après [Ferber, 1995].

D'après [Ferber, 1995], les systèmes multi-agents sont donc basés sur un certain nombre de principes et concepts avec notamment les agents, l'environnement, les interactions. L'approche *Voyelles* (AEIO) [Demazeau, 1995] ajoute à ces trois concepts celui de l'organisation qui fait référence aux moyens utilisés pour structurer l'ensemble des entités. Dans le contexte de ce travail, nous allons surtout nous concentrer sur les notions d'agents et d'environnement.

1.1.1 Le concept d'agent

En se basant sur les définitions de [Ferber, 1995] et [Wooldridge, 2000, Wooldridge, 2002], il est possible de définir un agent comme une entité logicielle autonome (capable de fonctionner sans interventions externes), située (qui évolue dans un environnement qu'il perçoit et qu'il est capable de modifier), sociable (qui interagit et communique avec d'autres agents), active (capable de prendre l'initiative en fonction de ses objectifs individuels).

Ainsi, un agent peut être considéré comme une entité dynamique définit par un ensemble de perceptions (entrées) traitées par sa structure interne et produisant, grâce à des mécanismes qui caractérisent sa dynamique, un ensemble d'actions (sorties) (voir figure 1.2).

La notion de perception est essentielle car elle permet l'interaction agent/agent et agent/environnement [1]. Ces perceptions peuvent être réalisées de manière très différentes (transmission de message, dépôt/collecte d'information dans l'environnement, etc.) afin de refléter les mécanismes réels qu'on souhaite représenter. Ainsi, quand il doit prendre une décision, un agent fait appel à ses connaissances et sa structure interne tout en évaluant ce que sa perception lui permet de sentir (autres agents, obstacles, signaux divers).

Généralement, la structure interne d'un agent peut prendre deux formes différentes :

1. Dans le cadre de notre travail, la notion de perception est primordiale car c'est sur elle que repose une partie de l'approche présentée dans ce document, comme nous le verrons à partir du chapitre 4.

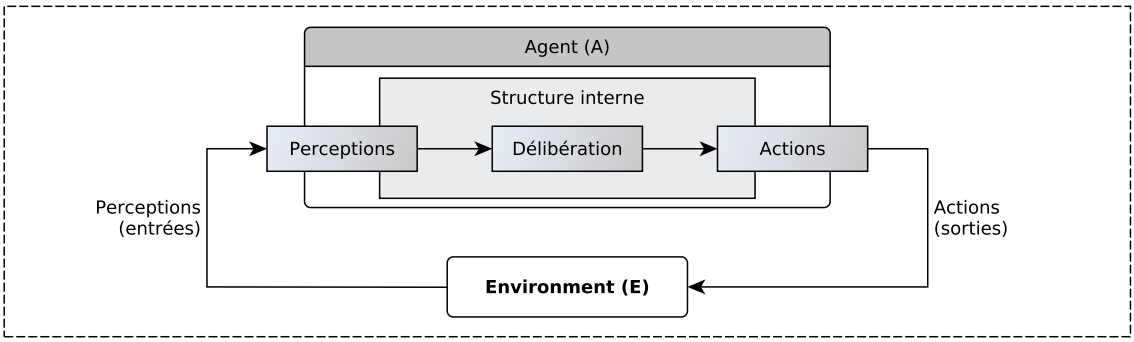


FIGURE 1.2 – Représentation d'un agent en tant qu'entité dynamique.

— soit réactive (on parle d'agents réactifs), on ne considère alors que les perceptions-actions (ou réponse à des stimulus) ;

— soit cognitive (on parle d'agents cognitifs), dans ce cas l'agent peut faire preuve de réflexion et peut avoir une représentation explicite de son environnement et des autres agents.

Les agents réactifs font leur apparition dans les années 80, et se basent sur l'idée qu'un agent n'a pas besoin d'avoir une représentation du monde dans lequel il évolue et qu'il peut agir et développer un comportement évolué sans processus cognitif (*e.g.* [Brooks, 1991]). L'agent utilise alors ses perceptions (entrées) pour percevoir son environnement afin d'agir en conséquence et d'effectuer les actions qu'il a à mener. Par exemple, ce type d'agent permet la représentation de systèmes complexes d'animaux, comme les colonies de fourmis (*e.g.* [Drogoul, 1993, Parunak, 1997]) ou de créatures artificielles (*e.g.* [Maes, 1990]).

Contrairement aux agents réactifs, les agents qualifiés de cognitifs possèdent une structure interne plus complexe qui les dote d'une capacité de raisonnement et de décision ainsi que d'une mémoire. Le comportement de ce type d'agent est déterminé par ses intentions correspondant aux objectifs qu'il veut atteindre. Ainsi, son processus comportemental comporte une étape de délibération (*Perception* / *Délibération* / *Action*) qui, grâce à sa mémoire et la représentation du monde dans lequel il évolue, va orienter ces choix entre plusieurs actions possibles. Ce type d'agent est utilisé pour représenter des entités douées d'une intelligence propre. L'exemple le plus connu d'architecture cognitive est celui des agents BDI [Rao and Georgeff, 1995] (de l'anglais *Belief Desire Intension*). Cette conception des agents consiste à élaborer un processus décisionnel qui utilise les notions de croyance, désir et intention.

Il existe une troisième approche, qualifiée d'hybride, qui consiste à allier le coté réactif et cognitif. Ainsi, avec comme idée de profiter des avantages des deux structures, les agents combinent, en leur sein, un sous-système cognitif qui leur permet de se représenter des connaissances et de planifier des actions et un sous-système réactif leurs offrant une plus grande flexibilité (*e.g.* [Malcolm and Smithers, 1990]).

Dans le cadre de nos travaux, nous considérerons en grande majorité des modèles contenant des agents réactifs car ces derniers possèdent une structure interne moins complexe qui s'accorde mieux aux spécificités d'une programmation sur architectures parallèles. Comme nous le verrons au chapitre 2, le calcul intensif, et plus particulièrement la programmation sur GPU, peut être très difficile à mettre en œuvre. Ainsi, implémenter des architectures cognitives avec ces technologies représentent toujours un challenge que très peu de travaux ont relevé.

1.1.2 Le concept d'environnement

Dans le domaine des systèmes multi-agents, il n'est pas possible de parler du concept d'agent sans parler de celui d'environnement. [Ferber, 1995] définit ce dernier comme un espace E doté d'une métrique, dans lequel sont situées les entités (agents ou objets) du système. Les actions des

agents sont sujettes aux lois de l'univers qui déterminent les effets des actions dans l'environnement. L'environnement est doté de processus qui déterminent sa réaction aux actions des agents.

De plus, d'après [Russell and Norvig, 1995], l'environnement est responsable de la structuration physique et sociale des agents. Il supporte également les moyens de communications (directs ou indirects) et gère l'accès aux ressources et aux différents services fournis aux agents. Enfin, l'environnement possède ses propres caractéristiques : accessible ou inaccessible, déterministe ou non déterministe, statique ou dynamique, discret ou continu.

Dans le cadre du travail présenté dans cette thèse, nous considérerons comme définition d'un environnement celle énoncée dans [Weyns et al., 2007] s'inscrivant dans la perspective E4MAS (de l'anglais, *Environments for Multi-agent Systems*) [Weyns and Michel, 2015] :

> The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.

De cette définition, il ressort que l'environnement doit être considéré comme une abstraction de premier ordre, conçu indépendamment des agents, de sorte que ses responsabilités soient clairement identifiées. De plus, il fournit les conditions d'existence des agents (actions et perceptions). 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. Enfin, parce qu'il est le lieu de l'interaction, l'environnement peut contenir des dynamiques additionnelles qui permettront la coopération et la collaboration entre agents. Nous reviendrons plus en détails sur ces aspects dans le chapitre 4 de ce document.

1.2 La simulation multi-agent

La simulation numérique[2] de systèmes complexes constitue un enjeu majeur dans de nombreux domaines comme ceux des sciences sociales [Conte et al., 1998], des écosystèmes [Simonin, 2001], ou des transports [Mandiau et al., 2008]. Elle permet de tester des hypothèses sur l'existant et d'en formuler de nouvelles a posteriori. Ces propriétés font de la simulation un outil d'étude et d'investigation pertinent quel que soit le domaine considéré.

Afin de modéliser ces systèmes complexes, l'approche qui vise à utiliser les systèmes multi-agents en tant qu'outil de conception de simulations est de plus en plus populaire [Michel et al., 2009]. Appelée simulation multi-agent (MABS, de l'anglais *Multi-Agent Based Simulation*), elle permet d'appréhender des phénomènes très différents grâce à sa représentation individu-centrée de ces phénomènes. Ayant réellement émergé à partir des années 80 (grâce notamment à l'augmentation des capacités de calculs des ordinateurs), les simulations multi-agents sont devenues de véritables laboratoires (virtuels) ayant entre autre comme objectifs la prédiction de l'évolution d'un système, l'explication de ses mécanismes, la découverte de phénomènes inattendus, etc. [Axelrod, 1997]. Les simulations multi-agents ont depuis été utilisées pour simuler de nombreux phénomènes complexes tels des phénomènes biologiques [Parunak, 1997] ou bio-inspirés [Bourjot et al., 2002] , du trafic routier [Bazzan et al., 1999], un modèle proie-prédateur [Haynes and Sen, 1996], une nuée d'oiseaux [Reynolds, 1987], un écosystème [Bousquet and Page, 2004], des sciences sociales [Epstein and Axtell, 1996], des réseaux sociaux [Zhao et al., 2014].

1.2.1 Modèles et simulation multi-agent

Avant d'aller plus loin, il convient de définir plus en détails le terme *modèle* qui reviendra de manière récurrente dans ce travail de thèse. Très simplement, un *modèle* propose une représentation schématique de la réalité (une représentation abstraite particulière d'un phénomène) en vue de l'étudier, de la comprendre. D'une manière plus formelle, et d'après la théorie de la modélisa-

2. Le concept de simulation numérique est détaillé dans [Fishwick, 1994], [Shannon, 1998] et [Zeigler et al., 2000].

tion et de la simulation (M&S) énoncée par ZEIGLER [3] [Zeigler et al., 2000], le *modèle* désigne l'ensemble des spécifications d'un *système source* qui permet de l'exécuter via un *simulateur* (ZEIGLER le qualifie aussi de *modèle de simulation*). Un *modèle* dans le processus de simulation est donc une spécification de l'ensemble des instructions qui permet de générer le comportement du système. C'est une description haut niveau d'un programme qui doit être implémentable.

La simulation multi-agent est donc basée sur la simulation de modèles multi-agents (ABM, de l'anglais *Agent-Based Model*) [Michel et al., 2009]. Les ABM sont des modèles individus-centrés dans lesquels on modélise directement le comportement d'agents autonomes évoluant dans un environnement ainsi que les interactions engendrées par ces comportements individuels. Les ABM adoptent une approche ascendante (modélisation *bottom-up* [Sabatier, 1986]) où le comportement du modèle est défini de manière locale par un ensemble d'algorithmes représentant les comportements locaux des phénomènes présents dans le système. Ainsi, la description de la dynamique du système ne se fait plus en déclarant des équations portant sur des propriétés macroscopiques de la simulation, comme c'est le cas avec les modèles à base d'équations mathématiques [4] (EBM, de l'anglais *Equation-Based Model*), mais sur le comportement individuel de chaque entité. Les ABM permettent de prendre en compte la complexité des interactions au niveau microscopique, ce que ne permettaient pas les paramètres des équations au sein des EBM [Parunak et al., 1998].

1.2.2 Plates-formes de simulations multi-agents

Un modèle est donc une spécification de l'ensemble des instructions qui permet de générer le comportement du système. Ainsi, définir un modèle de simulation multi-agent consiste à définir une fonction *Évolution* [Michel, 2004] qui calcule, pour un état $\sigma(t)$ du modèle, l'état suivant $\sigma(t+\delta(t))$ de manière déterministe :

$$\sigma(t+\delta(t)) = \mathit{Évolution}(\sigma(t))$$

Cependant, du fait de la très grande variété de modèles qu'il est possible d'élaborer, il existe un nombre incalculable de façons de définir une telle fonction. Cela explique, en partie, le nombre très important de plates-formes dédiées au développement de systèmes et simulations multi-agents. Dans [Kravari and Bassiliades, 2015], 24 de ces plates-formes ont été comparées et évaluées (il en existe bien d'autres) selon de nombreux critères. Il en ressort que chacune d'entre elle possède un champ d'application bien spécifique. Ainsi, certaines vont être spécialisées dans la simulation à large échelle (*e.g.* GAMA), d'autres en simulation économique (*e.g.* JADE) ou en biologie et études sociales (*e.g.* Jason ou Repast), etc.

Si l'on s'intéresse à des aspects tels que la simplicité d'utilisation, la facilité de prise en main, la scalabilité du modèle simulé ou encore les performances de ces plates-formes (des thèmes proches de ceux que nous rencontrerons tout au long de ce document), il est possible d'extraire de [Kravari and Bassiliades, 2015] le tableau [5] 1.1. Plus précisément, nous considérons les critères suivants :

— Le critère *Simplicité* caractérise la facilité d'utilisation de la plate-forme (la facilité de développer sur cette dernière) ;

— Le critère *Apprentissage* caractérise la rapidité de prise en main de la plate-forme et si ces mécanismes sont compréhensibles ;

— Le critère *Support* quantifie les documents et ressources à disposition de l'utilisateur ;

— Le critère *Scalabilité* donne des informations sur la capacité de la plate-forme à gérer des problèmes de tailles différentes ;

3. La théorie M&S de ZEIGLER [Zeigler et al., 2000] est décrite et expliquée plus en détails dans [Michel et al., 2009].

4. Un exemple historique est le modèle Proie/Prédateur qui représente l'évolution d'une population dynamique de deux espèces animales [Volterra, 1926].

5. Le choix des plates-formes s'est fait en fonction de leur notoriété et/ou de leur capacité à implémenter des simulations multi-agents. Ce tableau n'est en rien exhaustif, il se veut juste informel.

— Le critère *Performance* caractérise la rapidité d'exécution de la plate-forme.

Plate-forme	Simplicité	Apprentissage	Support	Scalabilité
GAMA [Grignard et al., 2013]	Simple, fonctions dédiées à la simulation, interface utilisateur	Rapide	Important	Bonne
Jason [Bordini et al., 2007]	Simple, interface utilisateur intuitive	Rapide	Important	Bonne
MadKit [Gutknecht and Ferber, 2001]	Moyenne, beaucoup de fonctions, interface utilisateur complexe	Moyen	Bon	Bonne
MasOn [Luke et al., 2005]	Moyenne, interface utilisateur complexe	Moyen	Convenable	Moyenne
NetLogo [Sklar, 2007]	Simple, nombreuses fonctions dédiées à la simulation, interface utilisateur	Rapide	Bon	Bonne
RePast [North et al., 2007]	Simple, interface utilisateur limitée	Rapide	Convenable	Bonne
Swarm [6]	Moyenne, interface utilisateur complexe	Moyen	Bon	Moyenne

TABLEAU 1.1 – Évaluation de plates-formes dédiées au développement de systèmes et simulations multi-agents (extrait de [Kravari and Bassiliades, 2015]).

Cependant, nous considérons le critère *Performance* comme trop subjectif pour l'intégrer. En effet, dans [Kravari and Bassiliades, 2015], il n'est fait à aucun moment mention du protocole expérimental utilisé permettant de définir les termes associés à ce critère d'évaluation et qui se limitent à des mots génériques tels que "excellente", "bonne" et "moyenne". De ce fait, nous proposons à la place de considérer le critère *Solution HPC* que nous trouvons plus pertinent et que nous définissons de la manière suivante :

— Le critère *Solution HPC* spécifie si la plate-forme est compatible avec le calcul intensif.

Le tableau 1.2 intègre ce critère et présente les différences entre les méthodes d'implémentation proposées par les plates-formes sélectionnées précédemment.

Plate-forme	Maturité technologique	Langages de programmation	Solution HPC
GAMA	Stable, développement actif	GAML	Non
Jason	Stable, développement actif	Java, AgentSpeak	Non
MadKit	Stable, développement actif	Java, C/C++, Python	Non
MasOn	Stable, développement actif	Java	Oui, [Ho et al., 2015]
NetLogo	Stable, développement actif	Logo	Non
RePast	Stable, développement actif	Java, C#, C++, Lisp, Prolog, Python	Oui, [Collier and North, 2013]
Swarm	Stable, développement actif	Java	Oui, [Pallickara and Pierce, 2008]

TABLEAU 1.2 – Différences entre les méthodes d'implémentation proposées par les plates-formes de développement (extrait de [Kravari and Bassiliades, 2015]).

1.2.3 Performance des simulations multi-agents

Malgré le fait que chaque plate-forme semble offrir une scalabilité pour les modèles simulés assez élevée (d'après le tableau 1.1), le manque de performance reste un problème inhérent à la simulation multi-agent. En effet, les simulations multi-agents ont souvent des besoins en ressources de calcul très importants. Cette contrainte pourrait être négligeable si elle était seulement liée à un unique problème de temps. Hors, les modèles multi-agents se construisent, la plupart du temps, de manière itérative et par retour visuel comme expliqué dans [Michel, 2015].

Ainsi, le temps d'exécution d'une simulation devient une contrainte importante qui préfigure de notre capacité à explorer un modèle. De plus, à l'heure actuelle, il est de plus en plus question de

6. Pour Swarm, seul un site internet est actuellement disponible mais en cours d'élaboration `http://www.swarm.org` (dernière visite : septembre 2016).

simulations large échelle [Picault and Mathieu, 2011] et/ou multi-niveaux [Camus et al., 2012,Morvan, 2012] faisant intervenir jusqu'à plusieurs millions d'agents dans des environnements de plus en plus grands. Autant de facteurs qui préfigurent d'une augmentation exponentielle des besoins en puissance de calcul à court et moyen termes.

Cependant, malgré cette croissance des besoins, très peu de plates-formes de simulations ont fait le choix de se tourner vers des solutions de calculs intensifs afin de pallier ce manque de ressources de calcul. Seul RePast HPC [Collier and North, 2013,Collier et al., 2015] et MasOn [Ho et al., 2015] (voir tableau 1.2) semblent se démarquer en proposant des solutions pour développer des modèles multi-agents sur des architectures parallèles (utilisation d'un ensemble de processeurs pour l'un, utilisation des cartes graphiques pour l'autre).

1.2.4 Implémentation de simulations multi-agents

Du tableau 1.1, nous observons également que la plupart des solutions dédiées au développement de simulations multi-agents proposent un grand nombre de ressources et une facilité d'utilisation permettant une bonne prise en main de ces différents outils. NetLogo en est un bon exemple car ce dernier est très utilisé dans le milieu académique pour sa simplicité et ses atouts pédagogiques.

Cependant, nous énoncions précédemment la très grande variété de modèles qu'il est possible d'élaborer et le caractère spécifique de chaque plate-forme dédiés au développement de systèmes et simulations multi-agents. Il en résulte que chacune de ces plates-formes va proposer sa propre méthode d'implémentation de modèles multi-agents (illustré par le tableau 1.2).

De plus, d'après [Michel, 2004], la conception d'un modèle de simulation multi-agent se fait autour de quatre aspects fondamentaux (voir figure 1.3) qui sont les comportements des agents, l'environnement, l'ordonnancement (la gestion du temps) et la gestion des interactions. Une fois encore, il n'apparait aucun consensus entre les plates-formes vis à vis de ces quatre aspects. Chacune d'entre elles proposant des solutions qui leur sont propres pour implémenter (ou encapsuler) certaines de ces fonctions-clés (ordonnancement des agents, modélisation de l'interaction, etc.). Nous pouvons citer l'exemple de IODA (*Interaction-Oriented Design of Agent simulations*) [Kubera et al., 2011] et de son moteur de simulation JEDI [Kubera et al., 2009] qui propose une méthode de modélisation de simulations multi-agents très différentes des approches usuelles car centrée sur les interactions.

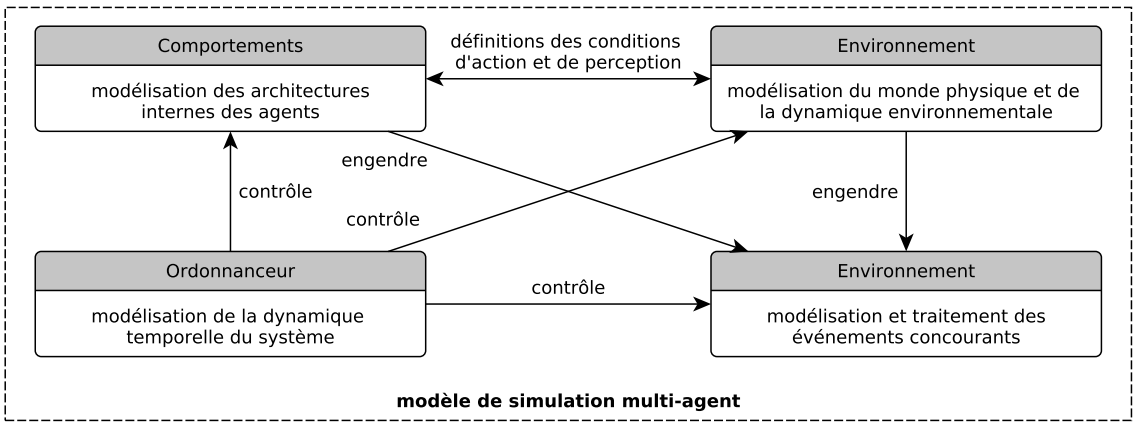


FIGURE 1.3 – Les quatres aspects d'un modèle de simulation multi-agent d'après [Michel, 2004].

Toutes ces spécificités et caractéristiques propres aux plates-formes sont donc à prendre en considération car l'utilisation de chacune d'entre elles va nécessiter un temps d'adaptation (malgré les ressources disponibles) et d'apprentissage pouvant limiter leur accessibilité. De plus, la réutilisabilité d'un modèle entre ces plates-formes est fortement contraint (voir impossible) du

fait de leurs particularités techniques et de leur champ d'application, ce qui réduit fortement la généricité des modèles créés.

Ainsi, nous identifions trois critères permettant d'évaluer les outils et solutions dans le cadre du développement de simulations multi-agents : la *généricité*, l'*accessibilité* et la *réutilisabilité*. Voici leur définition dans le contexte de ce document :

Définition 1. Généricité
*La **généricité** caractérise la capacité de la solution considérée à prendre en compte (à pouvoir être appliquée sur) une grande variété de modèles multi-agents.*

Définition 2. Accessibilité
*L'**accessibilité** fait référence à la simplicité de programmation ainsi qu'à la facilité de prise en main de la solution considérée.*

Définition 3. Réutilisabilité
*La **réutilisabilité** souligne la capacité de la solution considérée à promouvoir des outils ou des éléments facilement réutilisables pour d'autres modèles que ceux pour lesquels ils ont été créés.*

Nous verrons, dans les chapitres suivants, que le problème d'accessibilité (mais aussi de réutilisabilité) déjà présent pour l'implémentation de simulations multi-agents avec des plates-formes dédiées (sur des architectures séquentielles) sera bien plus important lorsque nous considérerons l'utilisation d'architectures matérielles différentes (telles des architectures parallèles) pour résoudre les problèmes de performances énoncés ci-dessus. Ainsi, la *généricité*, l'*accessibilité* et la *réutilisabilité* deviendront des critères essentiels dans la suite de ce document aussi bien pour comparer les travaux présentés (dans l'état de l'art au chapitre 3) que pour évaluer nos propres développements (dans la partie II de ce document).

1.3 Résumé du chapitre et objectifs premiers de la thèse

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, 1995]. Ils sont donc basés sur un certain nombre de principes et concepts avec notamment les agents et l'environnement.

La simulation multi-agent est un sous-domaine des systèmes multi-agents basée sur une approche individu-centrée [Michel et al., 2009] qui modélise 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 (EBM), l'approche multi-agent modélise directement les interactions engendrées par des comportements individuels.

Nous avons vu dans ce chapitre que les simulations multi-agents nécessitent des ressources de calcul importantes. Malgré cette demande en constante augmentation, la proportion d'outils proposant des solutions visant à améliorer les performances de ces simulations restent extrêmement faible. Il apparait donc clairement, à l'heure actuelle, que la question des performances reste un des verrous majeurs pour la simulation multi-agent.

Dans ce contexte, nos objectifs pour ce travail de thèse sont de répondre à ces problèmes de performance et de trouver des solutions pour permettre des simulations multi-agents à plus large échelle. Pour ce faire, nous explorerons l'idée d'utiliser le calcul haute performance (HPC, de l'anglais *High Performance Computing*) que nous présentons au chapitre 2. Cependant, il nous faudra

aussi prendre en compte que l'utilisation du HPC s'accompagne d'architectures matérielles particulières et de paradigmes de programmation complexes. Ainsi, alors que le développement de simulations multi-agents souffre déjà d'un manque d'accessibilité et de réutilisabilité, il y a fort à parier que tous ces problèmes vont être exacerbés dès lors que nous considérerons d'implémenter des simulations multi-agents sur des architectures matérielles différentes. Comme nous le verrons au chapitre 3, toute la difficulté va être de trouver des solutions d'implémentation de simulations multi-agents dans un contexte HPC respectant nos trois critères d'évaluation (*généricité, accessibilité* et *réutilisabilité*) définis dans ce chapitre.

CHAPITRE 2

CALCUL HAUTE PERFORMANCE ET GPGPU

Sommaire

Le calcul haute performance (aussi appelé calcul intensif) fait référence au domaine dédié au développement des architectures matérielles capables d'exécuter plusieurs milliards d'opérations à la seconde ainsi qu'aux méthodes et techniques de programmation massivement parallèles qui leurs sont associées. Parmi les architectures dédiées au HPC, les super-ordinateurs peuvent être vus comme un grand nombre de machines (*e.g.* des serveurs de calcul) composées de plusieurs milliers de processeurs reliés entres eux par des réseaux à très haut débit et capables de pouvoir traiter plus rapidement :

— Une simulation donnée en partageant sur un ensemble de serveurs le travail à réaliser (en parallélisant le calcul) ;

— Un grand nombre de simulations exécutées sur l'ensemble des serveurs de calcul en faisant varier par exemple des paramètres d'entrées comme la température ou la pression d'un fluide, la déformation d'un solide, la volatilité d'un produit dérivé financier, etc.

Ainsi, un calcul qui s'exécute en 24 heures sur un PC (de l'anglais *Personnal Computer*) classique ne prendrait que quelques minutes sur un super-ordinateur. Ces gains en temps permettent de réduire les coûts à chaque étape de la vie d'un produit ou d'un process (conception, optimisation, validation) et concernent potentiellement de nombreux domaines de recherche et applicatifs industriels tels que l'environnement, l'automobile, l'aéronautique et le spatial, la chimie, la médecine et la biologie, les matériaux, l'énergie, la finance, le traitement massif de données multimédia, etc.

2.1 Une réponse à des besoins de calculs

Les super-ordinateurs sont devenus des outils omniprésents dans le monde de la recherche et du développement car ils offrent la possibilité d'évaluer des situations, d'expérimenter mais aussi de réaliser des expériences qui ne peuvent être menées en laboratoire à cause de coûts trop élevés, de danger, ou d'impossibilité physique de mise en œuvre.

Ainsi, la puissance disponible au sein des super-ordinateurs a augmenté parallèlement aux progrès de l'électronique et suit la théorie énoncée par Gordon MOORE. Cette dernière énonce un doublement de la puissance disponible tous les 18 mois. Pour mesurer cette puissance, il est usuel d'utiliser une unité dédiée : le Flops (de l'anglais *Floating point Operations Per Second*). La figure 2.1 présente une évolution de l'évolution de cette puissance [1] pour les 500 meilleurs super-calculateurs de 1993 à 2014.

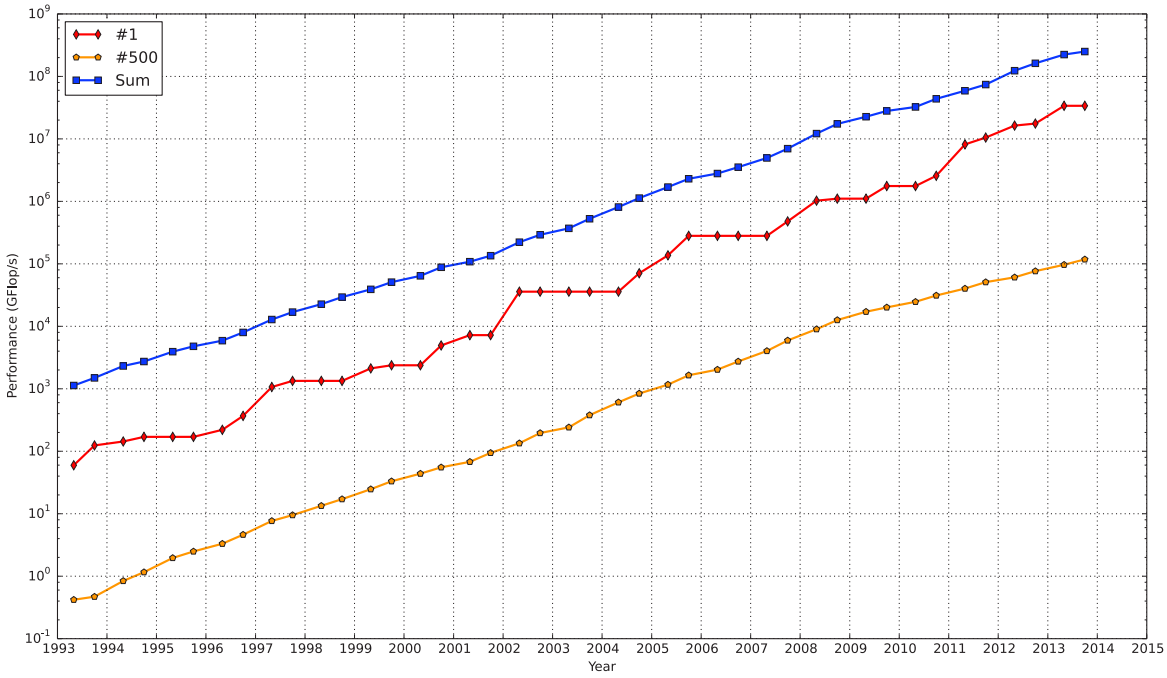


FIGURE 2.1 – Évolution de la puissance des super-calculateurs (*source : www.top500.org*).

Pour atteindre de telles capacités de calcul, les super-ordinateurs sont obligés d'utiliser un grand nombre de processeurs (plusieurs milliers), aussi appelés CPU (de l'anglais *Central Processing Unit*), qui vont fonctionner de manière conjointe. Cette utilisation parallèle des CPU est une spécificité introduite par ces super-ordinateurs.

En effet, avant l'émergence de ce mouvement de parallélisation, les ordinateurs ne contenaient qu'un seul CPU dont la fréquence de fonctionnement était la seule caractéristique permettant d'évaluer la performance de la machine. La règle était alors relativement simple : plus la fréquence était élevée, plus le CPU était rapide. Ainsi, il était alors possible de permettre à un programme limité par la vitesse du CPU de s'exécuter plus rapidement sans la moindre adaptation (un CPU plus rapide permet d'effectuer plus d'opérations par seconde).

Cependant, l'augmentation de la fréquence au sein d'un CPU est limitée par l'apparition de multiple obstacles physiques, notamment en termes de miniaturisation (finesse de gravure du CPU) et de dissipation thermique. Le parallélisme est donc apparu comme la solution pour contourner ces difficultés afin d'augmenter les performances des ordinateurs. L'accroissement de la puis-

1. 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.

sance de calcul est alors réalisée grâce à la multiplication du nombre de CPU et du nombre de cœurs d'exécution (processeur multi-cœurs) et/ou par une interconnexion entre de nombreuses machines (serveurs de calcul).

2.2 Le parallélisme

Les architectures parallèles sont devenues omniprésentes et chaque ordinateur possède maintenant un processeur reposant sur ce type d'architecture matérielle. Le parallélisme consiste à utiliser ces architectures parallèles pour traiter des informations et des données de manière simultanée dans le but de réaliser le plus grand nombre d'opérations par seconde.

2.2.1 La taxonomie de Flynn

La *taxonomie* établie par Michael J. FLYNN [Flynn, 1972] est l'un des premiers systèmes de classification qui classe les architectures des ordinateurs selon le type d'organisation du flux de données et du flux d'instructions qu'ils utilisent. Il référence ainsi quatre types différents d'architecture qui vont du plus simple traitant uniquement une donnée à la fois, ils sont dits *séquentiels*, aux plus complexes traitant de nombreuses données et instructions en simultané, ils sont qualifiés de *parallèles*. La figure 2.2 présente cette classification :

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

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

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

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

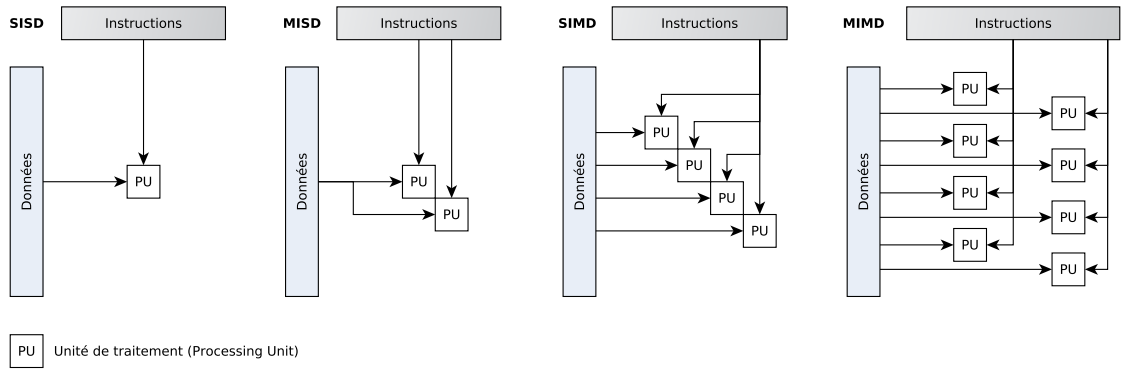


FIGURE 2.2 – Représentation de la taxonomie de FLYNN.

Cette classification montre clairement deux types de parallélismes différents : le *MIMD* et le *SIMD*. Le *MIMD* correspond au parallélisme par flot d'instructions, également nommé *parallélisme de traitement* ou de contrôle, dans lequel plusieurs instructions différentes sont exécutées simultanément. Le *SIMD* fait référence au *parallélisme de données*, où les mêmes opérations sont répétées sur des données différentes.

2.2.2 Efficacité du parallélisme

Cependant, il serait naïf de penser que la parallélisation d'un programme ou d'un algorithme apporte nécessairement un gain de performance important. De façon idéale, l'accélération in-

duite par la parallélisation devrait être linéaire. En doublant le nombre d'unités de calcul, on devrait réduire de moitié le temps d'exécution, et ainsi de suite. Malheureusement très peu de programmes peuvent prétendre à de tels résultats. En effet, l'accélération que peut apporter la parallélisation est limitée par le nombre d'exécutions parallèles possibles au sein de l'algorithme ou du programme.

Dans les années 1960, Gene AMDAHL formula une loi empirique restée célèbre [Amdahl, 1967]. La loi d'AMDAHL (voir figure 2.3) affirme qu'il existe, dans tout programme, une partie du code source qui ne peut être parallélisée et qui limite la vitesse d'exécution globale. Ainsi, cette loi établie une relation entre le ratio de code parallélisé et la vitesse globale d'exécution du programme. Dans la pratique, si un programme peut être parallélisé à 90 %, l'accélération maximale théorique sera de ×10, quel que soit le nombre de processeurs utilisés (voir figure 2.3).

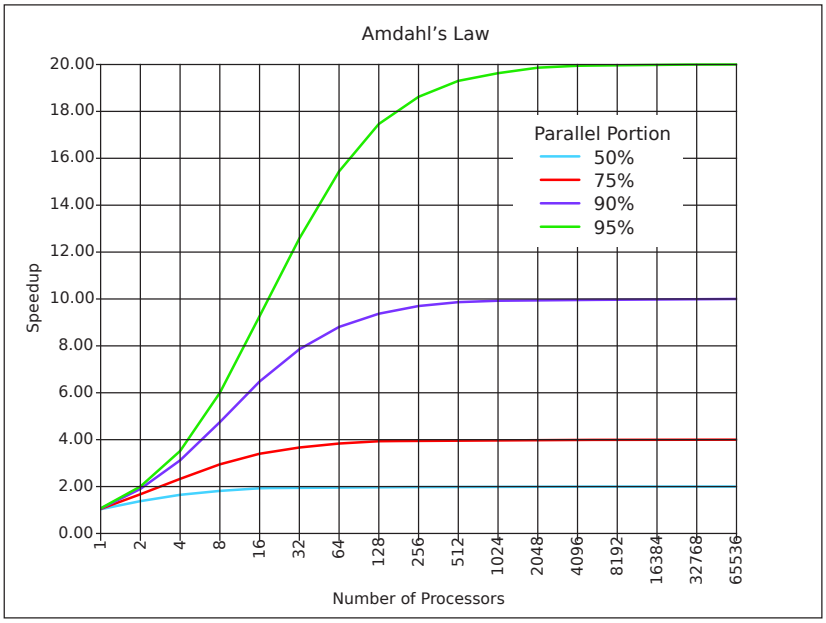


FIGURE 2.3 – Représentation de la loi d'AMDAHL.

D'autres lois ont suivi, telle que la loi de GUSTAFSON [Gustafson, 1988] qui prédit que le gain de vitesse obtenu est proportionnel à la fois au taux que représente la partie non-parallélisable et au nombre de processeurs. La métrique de KARP-FLATT [Karp and Flatt, 1990], proposée en 1990, est plus complexe et efficace que les deux autres lois. Elle intègre le coût lié au temps d'exécutions des instructions qui mettent en œuvre le parallélisme.

2.2.3 Solutions de parallélisation

Avant de paralléliser un programme ou algorithme, il est important de choisir le matériel sur lequel va être implémenté le programme. Il existe plusieurs types d'architectures matérielles dédiées au parallélisme, chacune d'entre elles possédant des caractéristiques qui lui sont propres (types et taille de la mémoire, vitesse, opérations réalisables, etc.) et appartenant donc à un type de parallélisme particulier (SIMD, MIMD, MISD). Parmi les architectures matérielles les plus connues, on retrouve donc :

— Processeur multi-cœurs : processeur composé de plusieurs unités de calcul "autonomes" permettant d'effectuer plusieurs opérations en parallèle, et donc d'accroître les performances globales du système. Cependant, les programmeurs doivent modifier leurs programmes pour bénéficier des avantages apportés par ces puces.

— Coprocesseur : processeur composé d'unités de calcul spécialisées, c'est à dire dédiées à des traitements particuliers tels que les opérations mathématiques sur les flottants, le calcul vectoriel, etc. Les efforts nécessaires à la transformation du code d'un programme pour

prendre en compte cette architecture matérielle sont réalisés en partie par les compilateurs et les bibliothèques dédiées.

— Carte graphique (GPU, de l'anglais *Graphics Processing Unit*) : processeur composé de centaines (voir de milliers) de cœurs dédiés, à la base, au calcul et rendu graphique. Ils sont cependant de plus en plus utilisés pour faire du calcul généraliste. Cette technologie nécessite une transformation complète du code du programme pour bénéficier des performances offertes par ces cartes.

Cependant, de par la spécificité de ces architectures matérielles, les programmes doivent être entièrement pensés dans l'optique d'être utilisés sur ce matériel particulier : l'implémentation d'un programme est directement impactée par le choix du matériel sur lequel il va être exécuté. De plus, un programme réalisé pour une architecture spécifique n'est pas compatible avec une autre et ne peut que très rarement être réutilisé. En effet, paralléliser un programme sur plusieurs unités de traitements (pour un processeur multi-cœurs par exemple) est très différent d'une parallélisation sur une carte graphique possédant des milliers d'unités de traitement. Enfin, les performances offertes ainsi que l'efficacité énergétique de ces différentes architectures varient énormément sans forcément se valoir. Faire le bon choix d'architecture est donc essentiel.

Une fois le matériel choisi, il est nécessaire d'utiliser un langage de programmation adapté à la programmation parallèle [2]. Parmi les plus connus, nous pouvons en présenter quatre :

— OpenMP [3] (*Open Multi-Processing*) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. OpenMP est portable et permet de développer rapidement des applications parallèles en restant proche du code séquentiel.

— MPI [4] (*The Message Passing Interface*) est une interface de programmation dédiée au calcul parallèle aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. MPI est grandement portable (car MPI a été implémenté sur presque toutes les architectures de mémoire) et rapide (car chaque implémentation a été optimisée pour le matériel sur lequel il s'exécute).

— OpenCL [5] (*Open Computing Language*) est la combinaison d'une API (de l'anglais *Application Program Interface*) et d'un langage de programmation dérivé du C, proposé comme un standard ouvert par le Khronos Group. OpenCL est conçu pour programmer des systèmes parallèles hétérogènes comprenant par exemple à la fois un CPU multi-cœurs et un GPU.

— CUDA [6] (*Compute Unified Device Architecture*) est aussi une combinaison entre une API et un langage de programmation dérivé du C. Il est dédié à la programmation sur GPU, c'est-à-dire qu'il permet d'utiliser un processeur graphique (GPU) pour exécuter des calculs généraux habituellement exécutés par le processeur central (CPU).

2.2.4 Les cartes graphiques : une alternative intéressante pour le calcul intensif

Les processeurs multi-cœurs et les coprocesseurs font partis des architectures matérielles privilégiées pour faire du calcul intensif. Cependant, bien qu'offrant de très bonnes performances [Hamidouche et al., 2009], ces solutions n'offrent pas toujours d'outils d'assez haut niveau pour être exploitables massivement par les programmeurs [Bourgoin, 2013]. De plus, l'investissement nécessaire pour développer une application sur ce type d'architecture (achat du matériel et temps de développement) ainsi que le coût d'exploitation (consommation énergétique) peut vite devenir très important.

2. De nombreux langages de programmation, au départ séquentiel (Java, C++, etc.), permettent de faire ce que l'on appelle du multi-tâches et donc de profiter, en partie, de l'architecture parallèle des CPU. Ici, nous considérons uniquement les langages de programmation dédiés au calcul intensif.

3. `https://www.openmp.org`

4. `https://www.open-mpi.org`

5. `https://www.khronos.org/opencl/`

6. `https://developer.nvidia.com/cuda-zone`

Ces contraintes ont poussé à étudier d'autres solutions, comme la possibilité d'utiliser les cartes graphiques pour faire du calcul intensif. En effet, ces cartes sont déjà bien connues dans le monde du graphisme et offrent de très bonnes performances [Owens et al., 2007, Che et al., 2008]. De plus, de nos jours, tout ordinateur personnel est équipé d'une carte graphique. Enfin, ces cartes possèdent un autre avantage non négligeable : leur prix très accessible.

Ainsi, c'est leur rapport prix/performance et leur disponibilité qui nous ont poussés à choisir cette plateforme matérielle pour nos développements et implémentations. Cependant, la technologie associée, appelée GPGPU (de l'anglais *General-Purpose computing on Graphics Processing Units*, qui permet de faire du calcul intensif sur ces cartes), s'appuie sur un parallélisme de type SIMD. Très différent d'une approche de programmation séquentielle classique, ce type de parallélisme qui nécessite notamment de suivre les principes de la programmation par traitement de flot de données est très spécifique et complexe. Nos travaux de thèse s'appuyant en grande partie sur l'utilisation de cette technologie, nous proposons maintenant d'introduire en détails le GPGPU.

2.3 General-Purpose computing on Graphics Processing Units

Une carte graphique repose sur une architecture matérielle *many-core* proposant un très grand nombre de cœurs d'exécution. Très rapidement, l'intérêt d'utiliser ces cartes pour faire du HPC est venu du fait qu'elles sont faites pour être performantes en calcul numérique (*e.g.* calcul géométrique dans les jeux vidéo, calcul matricielle pour le graphisme) et offrent un rapport performance/coût supérieur aux autres solutions existantes (voir tableau 2.1). Cependant, effectuer du calcul généraliste n'a pas toujours été un des rôles des GPU. Comme nous allons le voir, ces architectures ont énormément évoluées avec le temps pour devenir, aujourd'hui, de véritables plateformes dédiées au calcul intensif.

	CPU		**GPU Desktop**	**GPU HPC**
	Core i7-4790	Core i7-6700K	Geforce GTX 980	Tesla K80
# Unités de calcul	4 (8 Threads)	**4 (8 Threads)**	2 048	**4 992**
Fréquence	3.6 GHz	**4.0 GHz**	1.2 GHz	**875 MHz**
Mémoire Max	32 GB	64 GB	4 GB	24 GB
GFlops (simple précision)	27.15	**43.83**	4 612	**8 736**
GFlops (double précision)	15.34	22.76	144	2 912
Prix	325 €	425 €	520 €	5 000 €

TABLEAU 2.1 – Comparaison entre les caractéristiques des CPU et GPU.

2.3.1 Historique et évolution des capacités des GPU

Dans les années 1980, les cartes graphiques ont été développées avec, comme objectif premier, de soulager les processeurs (CPU) de la charge que demandait l'affichage du texte et, plus tard, la gestion graphique de chaque pixel de l'écran. Ces cartes sont des processeurs (GPU) secondaires, dotés de leur propre espace mémoire, connectés et contrôlés par le CPU.

Avec l'engouement des années 90 pour les graphismes dans le jeu vidéo (couleur, animations et nombre de pixels à afficher de plus en plus important), ces cartes ont rapidement été amenées à traiter de plus en plus d'opérations (déléguées par le CPU) et se sont donc perfectionnées jusqu'à devenir des GPU programmables capables d'exécuter des programmes spécialement écrits pour eux. Ces capacités de programmation sont très limitées et ne sont accessibles que par le biais de bibliothèques de rendu graphique comme OpenGL[7] (de l'anglais *Open Graphics Library*) et DirectX[8].

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

7. https://www.opengl.org
8. https://en.wikipedia.org/wiki/DirectX

"à la main" les fonctionnalités graphiques de la carte pour réaliser des calculs relevant d'un autre contexte [Owens et al., 2007]. Par exemple, il était nécessaire d'utiliser les textures graphiques comme structures de données : chaque *texel*[9] 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*[10], 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.

Il aura fallu attendre 2007 pour que sorte la première interface de programmation dédiée au GPGPU. Mise à disposition par Nvidia, CUDA a pour objectif de fournir un meilleur support et une plus grande accessibilité à 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, 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 un standard ouvert, fourni par le consortium industriel *Khronos Group*, qui s'utilise aussi comme une extension du langage C. CUDA et OpenCL sont actuellement les deux solutions les plus populaires pour faire du GPGPU.

De nos jours, l'utilisation des GPU comme architecture dédiée au calcul intensif devient si populaire que la marque Nvidia communique de plus en plus sur les capacités en terme de puissance de calcul de ses cartes alors que cette communication était auparavant axée sur les capacités graphiques dans les jeux vidéo.

2.3.2 Une nouvelle architecture d'exécution

Afin de développer les capacités GPGPU des cartes graphiques, les GPU ont évolué et sont maintenant composés de centaines d'ALU (de l'anglais *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 cœurs d'exécution (d'ALU) qui composent un GPU, bien plus important que pour un CPU, comme le montre la figure 2.4.

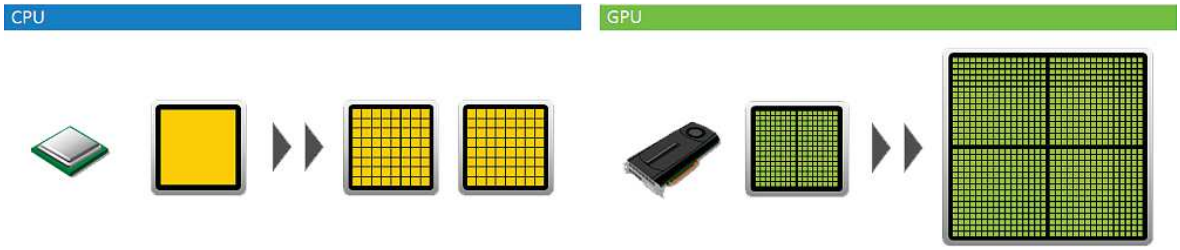


FIGURE 2.4 – Évolution des CPU et GPU vers des architectures *many-core*.

Ainsi, les GPU sont des architectures hautement parallèles qui exigent des modèles de programmation spécifiques [Bourgoin, 2013]. Le paradigme de programmation derrière le GPGPU est basé sur le modèle de calcul parallèle SIMD que l'on appelle aussi *Stream Processing* (*cf.* traitement de flux). Il consiste en l'exécution simultanée d'une série d'opérations (un noyau de calcul – *kernel*[11]) sur un jeu de données (le flux – *stream*). Lorsque la structure de données s'y prête,

9. Un texel est l'élément le plus petit composant une texture.

10. Un shader est un programme dédié au traitement graphique, qui sera exécuté par le GPU. Il permet de manipuler les données graphiques (pixel, vertex) directement depuis la carte graphique et de profiter des multiples unités de calcul qu'elle possède. D'abord dédiés à la génération d'effets visuels pré-calculés, les shaders ont fini par devenir des scripts de calculs généralistes.

11. On appelle *kernel* le noyau de calcul qui va s'exécuter sur le GPU.

l'architecture massivement parallèle du GPU permet d'obtenir des gains de performance très élevés [12].

L'architecture GPGPU des cartes se base sur l'utilisation de plusieurs éléments :

— Des *threads* [13] ;

— Des *blocs* contenant un certain nombre de *threads* ;

— Une *grille* contenant un certain nombre de *blocs*.

La figure 2.5 illustre, de manière simplifiée, cette architecture GPGPU en présentant une grille ainsi que l'architecture mémoire qui lui est associée. Celle-ci est divisée en plusieurs catégories possédant chacune des caractéristiques propres (vitesse de lecture et écriture, taille, droits d'accès, latence, etc.) :

— Une mémoire *globale* accessible par tous les *threads* de la grille ;

— Une mémoire *partagée* entre tous les *threads* au sein d'un même bloc ;

— Une mémoire *locale* à chaque *thread* ;

— Des mémoires *spécifiques* dépendant du matériel utilisé.

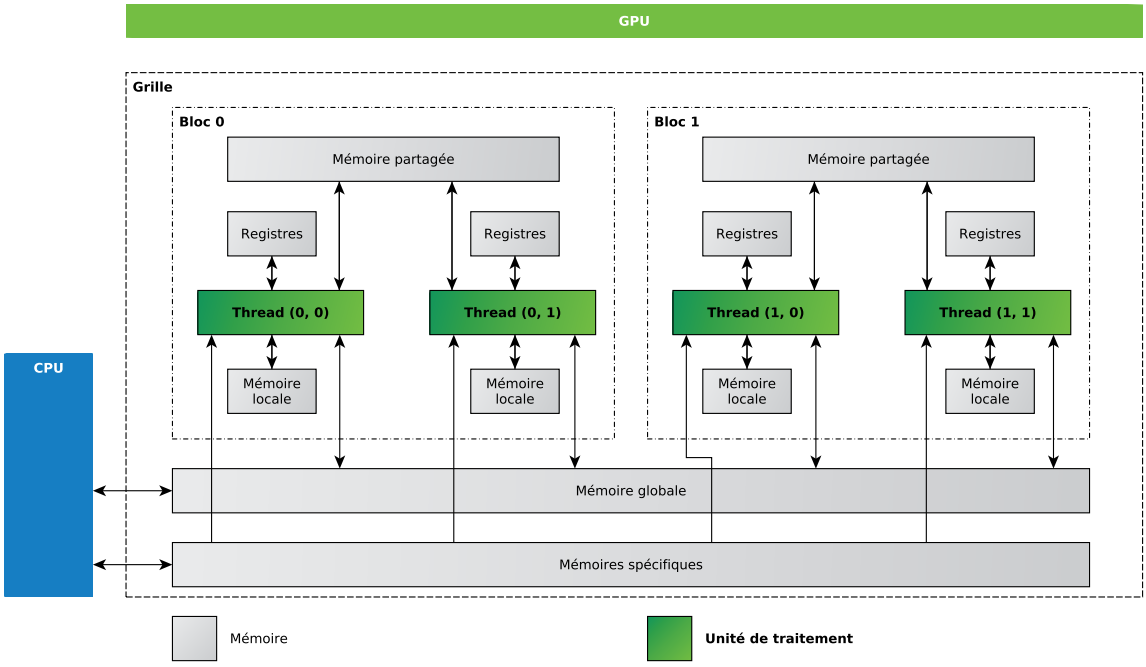


FIGURE 2.5 – Représentation simplifiée de l'architecture d'un GPU.

2.3.3 Modèle de programmation

La figure 2.5 a présenté une schématisation très simplifiée de l'architecture GPGPU d'une carte graphique avec uniquement deux blocs de deux *threads*. En pratique, le modèle de programmation GPGPU se caractérise par l'utilisation la plus importante possible du découpage en *threads* et en *blocs* : il n'est pas rare d'avoir des *kernels* s'exécutant sur des milliers de *threads* en simultané.

Que l'on utilise CUDA ou OpenCL, l'utilisation de ces différents éléments (*grille*, *blocs* et *threads*) reste la même et peut être définie de la façon suivante : le CPU, appelé *host*, joue le rôle du chef

12. Des exemples d'utilisation de cette technologie ainsi que les résultats de performance associés sont disponibles aux adresses suivantes : `https://www.nvidia.com/object/gpu-applications.html` et `https://www.nvidia.com/object/tesla-case-studies.html`.

13. Un *thread* est une unité logique de traitement qui va exécuté une instance du *kernel*.

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*. Ce dernier est capable d'exécuter un *kernel* des milliers de fois en parallèle grâce aux *threads*. Ces *threads* sont regroupés par *blocs* (les paramètres *blockDim.x, blockDim.y* définissent la taille de ces blocs), qui sont eux-mêmes rassemblés dans une *grille globale*. Chaque *thread* au sein de cette structure est identifié par des coordonnées uniques 3D (*threadIdx.x, threadIdx.y, threadIdx.z*) lui permettant d'être localisé. De la même façon, un *bloc* possède aussi des coordonnées 3D qui lui permettent d'être identifié dans la *grille* (respectivement *blockIdx.x, blockIdx.y, blockIdx.z*).

Une fois la grille définie (nombre de *blocs* et de *threads*), elle est projetée sur les multiprocesseurs matériels du dispositif utilisé. En général, et dans la suite de ce document, les identifiants des *threads* dans la *grille* globale du GPU seront notés i et j (nous travaillons sur des environnements 2D) : ces identifiants correspondent aux coordonnées spatiales 2D du *thread*. Ainsi, les *threads* vont exécuter le même *kernel* mais vont traiter des données différentes selon leur localisation spatiale dans la *grille* globale (identifiant) :

— Coordonée x d'un *thread* : $i = blockIdx.x * blockDim.x + threadIdx.x$.

— Coordonée y d'un *thread* : $j = blockIdx.y * blockDim.y + threadIdx.y$.

La figure 2.6 illustre le calcul de coordonnées d'un *thread* pour une grille en deux dimensions.

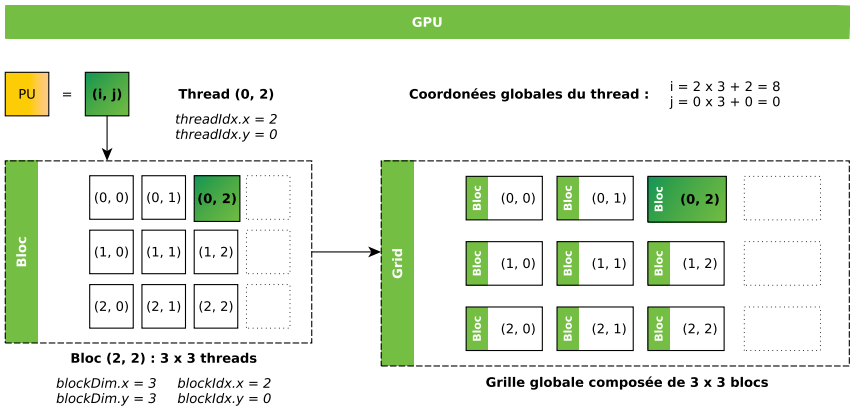


FIGURE 2.6 – Calcul des coordonnées d'un *thread* dans une grille globale en 2D.

Cependant, la carte graphique ne fonctionne pas de manière indépendante : son utilisation est contrôlée par le CPU. Ainsi, le modèle de programmation GPGPU impose une préparation de l'exécution et des données, puis une récupération des résultats et une libération des ressources après l'exécution des *kernels* sur le GPU. Toutes ces étapes sont initiées et régulées par le CPU. Le processus de calcul sur GPU est donc découpée en six phases (voir figure 2.7) :

— Phase (1) : allocations des ressources sur le GPU (par le CPU).

— Phase (2) : copie des données dans la mémoire globale du GPU (par le CPU).

— Phase (3) : initialisation de l'exécution d'un ou plusieurs *kernels* sur le GPU (par le CPU).

— Phase (4) : exécution des différents *kernels* de calcul sur le GPU.

— Phase (5) : copie des résultats (par le CPU).

— Phase (6) : libération des ressources allouées sur le GPU (par le CPU).

Une des particularités étant que toutes les phases de ce cycle sont indépendantes les unes des autres et peuvent être exécutées de manière concurrente : il est alors possible d'initier un calcul sur GPU (avec un kernel) aussi bien que plusieurs *kernels* en simultané [14].

14. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#asynchronous-concurrent-execution`

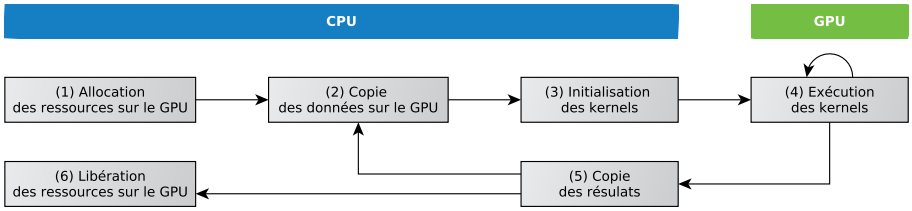


FIGURE 2.7 – Représentation du processus d'exécution d'un (ou plusieurs) *kernel* sur GPU.

Finalement, un programme utilisant le GPGPU est divisé en deux parties distinctes composées d'un côté des *kernels* qui s'exécutent sur le GPU et de l'autre du code *host* CPU qui contrôle l'exécution des kernels. L'algorithme 2.1 donne un exemple générique de la structuration d'un *kernel* et du code *host* associé [15].

Algorithme 2.1 : Exemple de structuration d'un *kernel* et de son code *host* (en C / CUDA C)

```
#include "main.h"
#define sizeGrid 1024

__global__ type fonction() { // Definition du kernel de calcul
        // Initialisation du thread
        i = blockIdx.x * blockDim.x + threadIdx.x;
        j = blockIdx.y * blockDim.y + threadIdx.y;
        [...]
        // Test conditionnel sur la taille de la grille de threads
        if(i < sizeGrid && j < sizeGrid){
            [...] // Realisation des calculs
        }
        [...] // Ecriture des resultats
}

int main( void ) { // Definition du programme host
        // Definition et initialisation des variables
        [...]
        // Allocation memoire sur le GPU (phase 1)
        cudaMalloc( ... );
        // Copie des donnees sur le GPU (phase 2)
        cudaMemcpy( ..., cudaMemcpyHostToDevice );
        // Initialisation de l'execution du kernel (phase 3)
        int threads = 32;
        int blocs = sizeGrid / threads;
        // Execution du kernel (phase 4)
        fonction<<<blocs,threads>>>( ... );
        // Copie des resultats (phase 5)
        cudaMemcpy( ..., cudaMemcpyDeviceToHost );
        // Liberation des ressources allouees (phase 6)
        cudaFree( ... );
}
```

2.3.4 Exemple d'implémentation

Comme le montre la section précédente, le modèle de programmation associé au GPGPU est très particulier. Ainsi, nous proposons d'illustrer son utilisation avec un exemple simple et concret : le calcul de π. Afin de souligner les particularités du GPGPU, nous le comparons à une implémentation en C et en MPI C.

15. Une description de la syntaxe utilisée est consultable en annexe B.

L'implémentation du calcul de π en C (algorithme 2.2) est la plus simple à comprendre car le programme ne présente aucune difficulté. Nous retrouvons en effet une programmation séquentielle classique comprenant une seule boucle itérative.

Algorithme 2.2 : Calcul de π en C (séquentiel, CPU)

entrées : $precision, borneInf, borneSup$
sortie : π

```
1  x = 0 ;
2  pour i = 1 à precision faire
3  |   x = (i * 1.0/precision);
4  |   π = π + ((1.0/precision) * 1/(1 + (x * x))) ;
5  fin
6  return π = π * 4
```

Au contraire, l'implémentation du calcul de π en MPI C (algorithme 2.3) est bien plus complexe. En effet, le langage MPI permet de paralléliser le calcul sur CPU ce qui nécessite d'intégrer dans le programme des variables relatives aux caractéristiques du processeur utilisé (nombre de cœurs, identifiant du cœur, etc.) ainsi que des fonctions permettant la communication entre les différents cœurs du processeur [16].

Algorithme 2.3 : Calcul de π en MPI C (parallèle, CPU)

entrées : $precision, borneInf, borneSup, nbCPU, idCPU$
sortie : π

```
1  x = 0 ;
2  tmp = 0 ;
3  intervalParCPU = (bornsup − borninf)/nbCPU ;
4  borneIdCPU = intervalParCPU * idCPU ;
5  pour i = 1 à precision faire
6  |   x = borneIdCPU + (i * intervalParCPU/precision);
7  |   tmp = tmp + ((intervalParCPU/precision) * 1/(1 + (x * x))) ;
8  fin
9  Reduce(tmp, nbCPU, idCPU);
10 si idCPU == 0 alors
11 |   π = π * 4;
12 fin
13 return π
```

Enfin, l'implémentation parallèle du calcul de π en CUDA C (algorithme 2.4) est relativement simple à comprendre (en plus d'être courte). Elle ne contient que l'initialisation du *thread*, le test conditionnel sur la taille de la grille [17] et le calcul de π sur les données correspondant à l'identifiant du *thread* initialisé : la boucle itérative a disparu. Ainsi, tout l'intérêt de la version GPU tient dans le fait que la parallélisation de cette boucle est réalisée grâce à l'architecture matérielle et non à l'aide de code additionnel : le code réside maintenant dans la structure.

Algorithme 2.4 : Calcul de π en CUDA C (parallèle, GPU)

entrées : $precision, interval[]$
sortie : π[]

```
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  si i < precision alors
3  |   π[i] = ((1.0/precision) * 1/(1 + (interv[i] * interv[i])));
4  fin
5  return π[i]
```

16. Une introduction au langage MPI est disponible à l'adresse suivante : `http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml`.

17. Plus de détails sur ce test conditionnel peuvent être trouvés dans [Sanders and Kandrot, 2011] en page 42.

Cependant, comme indiqué précédemment, le calcul sur GPU s'accompagne de plusieurs phases de préparation (copie des données, allocation des ressources, initiation des calculs) et de finalisation (copie des résultats, libération des ressources allouées) qui complexifient énormément son utilisation (voir algorithme 2.5). Ces phases sont essentielles car elles vont conditionner l'obtention de bonnes performances.

Algorithme 2.5 : Code *host* contrôlant le calcul de π en CUDA C

```
   entrées : precision, interval[], π[]
   sortie  : π
1  pour i = 1 à precision faire
2  |   interval[i] = i * (1.0/precision);
3  |   π[i] = 0 ;
4  fin
   /* Allocation mémoire sur le GPU                                  */
5  cudaMalloc(interval[], π[]) ;
   /* Copie des données sur le GPU                                   */
6  cudaMemcpy(interval[], π[]) ;
   /* Éxécution du kernel                                            */
7  calculPiGPU(precision,interval[], π[]) ;
   /* Copie des résultats                                            */
8  cudaMemcpy(π[]) ;
   /* Libération des ressources allouées                             */
9  cudaFree(interval[], π[]) ;
10 pour i = 1 à precision faire
11 |   π = π + π[i] ;
12 fin
13 return π
```

Nous avons mis à disposition d'autres exemples (avec leur parallélisation sur CPU et GPU) en ligne sur *GitHub*[18]. Les travaux d'OWENS [Owens et al., 2007] permettent également d'avoir une meilleure compréhension du domaine car ils présentent en détails les différents aspects techniques autour de la programmation sur GPU. Enfin, les sites internet de Nvidia[19] et OpenCL[20] regorgent de cours et d'informations permettant une meilleure prise en main de cette technologie qui peut sembler au premier abord très complexe.

2.3.5 Aspects importants autour de la programmation GPU

L'architecture GPU est très spécifique de par son design mais aussi de par la programmation particulière qu'elle nécessite pour être utilisée de manière efficiente. Ainsi, il existe un ensemble de recommandations sur la bonne manière de programmer sur GPU dans le but d'obtenir un programme efficace [Nvidia, 2015].

La minimisation des coûts de transferts

Comme nous l'avons vu précédemment, une exécution est précédée d'une phase de préparation des données (formatage, transferts, etc.). Le coût des transferts étant important, il est nécessaire d'en réduire le nombre ou de les regrouper. Cependant, réduire le nombre de transferts n'est pas toujours possible car chaque donnée utilisée sur le GPU doit être explicitement copiée pour qu'elle puisse être utilisée par les *kernels* de calcul. Dans ce cas, une solution peut être de réduire la fréquence de synchronisation de la valeur de cette donnée entre CPU et GPU, ou en augmentant le temps passé sur le périphérique entre chaque retour sur le CPU. Le regroupement est un autre moyen de minimiser le coût total des transferts, en utilisant la bande passante importante

18. https://github.com/ehermellin/IntroHPC
19. https://developer.nvidia.com/what-cuda
20. http://www.khronos.org/opencl

disponible entre CPU et GPU. Ce regroupement est aussi facilité par les différents mécanismes de copie mis à disposition par les différentes interfaces de programmation GPU.

Dans le cadre des simulations multi-agents, les agents doivent avoir accès aux résultats à chaque pas de temps pour pouvoir calculer leur comportement. Cette contrainte requiert d'utiliser, pour chaque itération, des primitives qui permettent de forcer la synchronisation entre les exécutions du CPU et du GPU. Ces primitives sont très coûteuses car elles obligent des transferts de données entre CPU et GPU. Il est donc important de bien organiser les différentes tâches exécutées par le GPU afin de minimiser les transferts nécessaires. Ainsi, nous effectuerons qu'une seule synchronisation par pas de temps et nous utiliserons cette optimisation pour les différentes expérimentations menées dans ce manuscrit.

Maximiser l'occupation

Un des points essentiels, pour l'obtention de bonnes performances sur GPU, est de *maximiser l'occupation* c'est-à-dire utiliser le plus efficacement possible les nombreux cœurs d'exécution offerts par l'architecture GPU. Il faudra donc veiller sur les ressources consommées par chaque *thread* (une utilisation trop importante des registres par les *threads* est susceptible d'empêcher l'utilisation de tous les cœurs d'exécution disponibles), sur le nombre de conditions présentes dans l'algorithme (une utilisation trop importante des structures conditionnelles est susceptible de causer une réduction de l'occupation des cœurs d'exécution) et sur le nombre de *threads* total lancés (une utilisation efficace du GPU repose sur une utilisation adéquate de l'ensemble des *threads* disponibles).

Par exemple, si la capacité d'un bloc est de 1 024 *threads* (celle-ci dépend du matériel), il est possible de travailler avec une grille de 1 000 × 1 000 en allouant une grille de blocs d'une taille de 32 × 32, avec chaque bloc ayant lui-même une taille de 32 × 32. Ce qui produit une matrice surdimensionnée de 1 024 × 1 024 *threads*. Cette taille, trop large par rapport aux données considérées, ne pose pas de problème car elle est gérée au niveau du code GPU et permet d'utiliser un maximum de *threads*.

Accès mémoires

Un autre point sur lequel joué est l'*optimisation des accès mémoires*. En effet, il existe plusieurs types de mémoire au sein d'un GPU (locales, globales, partagées), chacune possédant des caractéristiques propres et des latences d'accès aux données différentes. Ainsi, il est possible d'optimiser l'exécution des *kernels* et améliorer les performances des programmes en minimisant les latences d'accès aux données en utilisant de manière réfléchie chaque type de mémoire en tirant partie de leurs différents avantages.

Une optimisation difficile

Cette liste de conseils n'est pas exhaustive, il existe encore de nombreux points sur lesquels jouer pour améliorer l'utilisation du GPGPU au sein des programmes. En effet, il y a beaucoup d'aspects à prendre en compte qui peuvent directement impacter les performances d'exécution. Il est très facile de créer un *kernel* de calcul sur GPU (quelques minutes suffises), l'obtention de gains de performances importants nécessitera cependant une longue phase d'optimisation.

Par ailleurs, un des avantages de cette technologie vient de sa communauté très dynamique qui gravite autour d'elle permettant ainsi d'avoir un nombre très important de ressources et de cours.

2.4 Résumé du chapitre et début de problématique

Les performances constituent un verrou majeur dans de nombreux domaines et les orientations qui sont prises en direction du calcul haute performance par différents groupes de re-

cherche et industriels le montrent clairement. En effet, le calcul intensif, de par l'utilisation de super-ordinateurs, permet de subvenir aux besoins toujours croissant en puissance de calcul. Néanmoins, ces super-ordinateurs nécessitent des investissements très lourds et des coûts de développement difficilement supportables pour un grand nombre d'acteurs de la recherche et de l'industrie.

Ainsi, de nouvelles solutions dédiées au HPC ont vu le jour, comme le GPGPU qui propose d'utiliser l'architecture massivement parallèle des cartes graphiques pour faire du calcul généraliste. Réelle révolution technologique, le GPGPU possède de nombreux avantages comme son rapport prix/performance parmi les meilleurs et sa disponibilité. En effet, de nos jours, quasiment tous les ordinateurs ont une carte graphique dédiée ou intégrée au CPU possédant des capacités GPGPU capables, suivant le contexte, d'accélérer considérablement les performances des programmes qui l'utilisent [Che et al., 2008]. Dans le cadre des simulations multi-agents, utiliser le GPGPU peut être une solution pour pallier les problèmes de performances et le manque en ressources de calcul que l'on peut rencontrer lors de la simulation de modèles multi-agents (*cf.* chapitre 1).

Cependant, la programmation sur GPU n'est pas chose aisée car elle repose sur une architecture matérielle spécifique qui définit un contexte de développement très particulier. Ce contexte architectural nécessite notamment de suivre les principes de la programmation par traitement de flot de données [Owens et al., 2007] et requiert des connaissances avancées qui limitent son utilisation. Ainsi, une implémentation sur GPU est bien plus complexe qu'un simple changement de langage de programmation [Bourgoin, 2013]. En particulier, le problème doit pouvoir être représenté par des structures de données distribuées et indépendantes. De plus, il n'est pas possible d'adopter une démarche orientée objet classique. De fait, les modèles de simulation multi-agent couramment utilisés, parce qu'ils reposent souvent sur des implémentations orientées objet, ne peuvent être utilisés sur une architecture GPU sans un effort de traduction conséquent et non trivial. De plus, la majorité des problèmes liée à l'implémentation de simulations multi-agents (soulevée au chapitre 1) va être exacerbée par l'utilisation de cette technologie.

C'est dans ce contexte que nous proposons, dans le chapitre 3, un état de l'art des contributions traitant de l'utilisation du GPGPU pour la modélisation et l'implémentation de simulations multi-agents. Dans ce chapitre, nous verrons que le faible engouement pour le GPGPU de la communauté multi-agent vient, en partie, de l'architecture spécifique des GPU qui impose des contraintes très fortes sur à la fois, ce qui peut être parallélisé, et la manière de le faire. Non seulement le GPGPU est difficile d'accès, mais il est également si contraint qu'il oblige à repenser la modélisation multi-agents. Ainsi, il est donc compréhensible que peu de travaux soient enclins à investir du temps dans l'utilisation du GPGPU car la pérennité du code produit est difficile à obtenir. Finalement, cet état de l'art permettra d'identifier les deux principaux verrous liés à la technologie GPGPU qui limitent son utilisation : l'accessibilité et la réutilisabilité, deux critères déjà rencontrés au chapitre précédent et que nous considérerons comme essentiels pour une meilleure adoption du GPGPU dans les simulations multi-agents.

CHAPITRE 3

SIMULATIONS MULTI-AGENTS ET GPGPU

Sommaire

Comme nous l'avons évoqué au chapitre 1, le temps d'exécution d'une simulation multi-agent est une contrainte importante qui préfigure de notre capacité à explorer et étudier le modèle simulé. Ainsi, plutôt que de devoir le simplifier et/ou faire des compromis (jouer sur la scalabilité du modèle et/ou sur sa visualisation), utiliser le GPGPU, afin d'améliorer les performances des simulations et ainsi lever une partie des contraintes liées au passage à l'échelle, peut être une piste de recherche intéressante [Che et al., 2008].

Cependant, comme énoncé au chapitre 2, le GPGPU repose sur l'utilisation d'une architecture matérielle spécifique, hautement parallèle, nécessitant une programmation particulière [Bourgoin et al., 2014] qui va forcément avoir une incidence sur la mise en œuvre des modèles multi-agents. Nous proposons donc, dans ce contexte, de réaliser un état de l'art sur l'utilisation du GPGPU dans le cadre de la simulation multi-agent [Hermellin et al., 2014, Hermellin et al., 2015] afin de souligner dans quelle mesure les spécificités de cette technologie vont impacter la modélisation et l'implémentation des simulations multi-agents sur GPU.

Cette question est d'autant plus pertinente que très peu des plates-formes de développement spécialisées dans la simulation multi-agent n'intègrent, en leur sein, de fonctionnalités dédiées au GPGPU ou plus généralement au calcul intensif (*cf.* chapitre 1). NetLogo [Sklar, 2007] conserve une implémentation séquentielle classique alors que RePast [North et al., 2007] intègre le calcul intensif via une utilisation des clusters de CPU [Collier and North, 2012]. Seul MasOn [Luke et al.,

2005] propose une intégration du GPGPU (avec même une possibilité d'utiliser plusieurs GPU [Ho et al., 2015]).

Ainsi, nous constaterons que les travaux mêlant GPGPU et simulations multi-agents sont majoritairement liés à des expérimentations très particulières limitant leur réutilisation. D'ailleurs, ces questions autour de la facilité de programmation et de l'accessibilité à cette technologie sont courantes dans le domaine du HPC et ne sont pas seulement propre au GPGPU. En effet, même une parallélisation d'une simulation sur CPU pose des problèmes comme le montre [McCabe et al., 2016].

Pour figurer dans l'état de l'art, que nous proposons dans ce chapitre, chacune des contributions sélectionnées devra répondre aux critères suivants :

— Les contributions doivent traiter de l'utilisation du GPGPU dans un contexte multi-agent ;

— Les travaux sélectionnés doivent soit introduire de nouveaux outils, frameworks, bibliothèques dédiées, soit présenter des études, méthodologies ou *benchmarks* relatifs à l'utilisation du GPGPU dans le contexte multi-agent.

Dans le cas ou plusieurs contributions discutent du même aspect, ou présentent un même outil, nous sélectionnons celui qui est le plus complet afin d'éviter toute redondance. Nous tenons à noter que cet état de l'art ne se veut pas exhaustif, son objectif premier est de donner un aperçu de ce qu'il se fait et est possible de faire avec le GPGPU dans la communauté multi-agent.

La présentation des différentes contributions sélectionnées est faite en fonction des choix d'implémentations et des techniques utilisées. En effet, implémenter une simulation multi-agent sur GPU peut se faire de deux manières distinctes, chacune possédant des avantages et inconvénients. La première consiste à exécuter entièrement la simulation sur le GPU, nous la nommons *tout-sur-GPU*. La seconde partage l'exécution de la simulation entre le CPU et le GPU et est classiquement qualifiée d'*hybride*. Nous choisissons de suivre cette distinction afin de classer les contributions sélectionnées.

3.1 Implémentation tout-sur-GPU

Historiquement, l'implémentation d'une simulation multi-agent entièrement sur GPU (voir figure 3.1) se faisait via l'utilisation et le détournement des fonctions graphiques du GPU, du fait du manque d'API dédiées. Par la suite, des API spécialisées dans le GPGPU, telles que CUDA et OpenCL, ont été créées et ont eu comme objectif de rendre accessible l'utilisation de cette technologie. Les détails techniques d'implémentation associés à ces deux méthodes d'utilisation du GPGPU ont été énoncés dans le chapitre 2 et sont explicités dans [Owens et al., 2007].

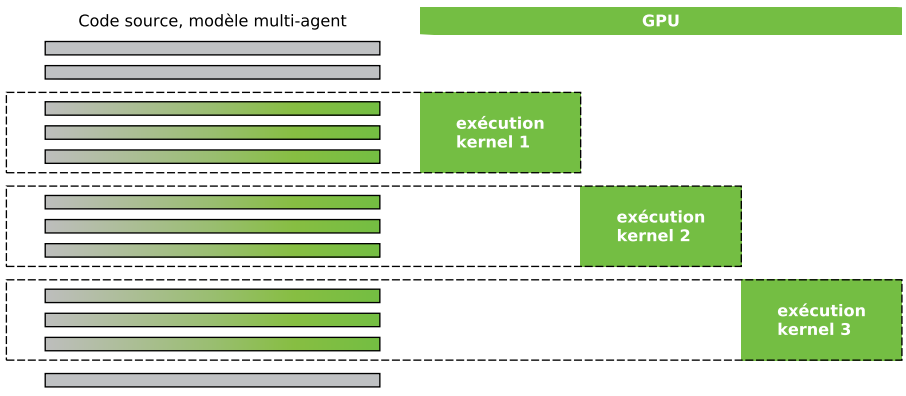


FIGURE 3.1 – Illustration d'une approche tout-sur-GPU.

3.1.1 Utilisation des fonctions graphiques des GPU

SugarScape est le premier SMA ayant profité d'un portage sur GPU [D'Souza et al., 2007]. C'est un modèle multi-agent 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 plus larges.

LYSENKO et D'SOUZA, motivés par ces très bons résultats, se sont ensuite attaqués au problème de l'accessibilité (*cf.* section 1.2.4) en généralisant leur approche et en proposant un framework destiné à faciliter l'implémentation de modèle sur GPU [Lysenko and D'Souza, 2008]. Celui-ci était composé de fonctions de bases telles que la gestion de données environnementales, la gestion des interactions entre agents, des fonctions pour la naissance et mort des agents, etc. Cependant, il ne permettait que des implémentations de modèles similaires à *SugarScape* comportant uniquement des agents réactifs aux comportements peu évolués du fait de la difficulté d'implémenter des architectures agents complexes sur GPU.

Suivant cette tendance, et dans le but de simplifier encore plus l'utilisation du GPGPU, le framework ABGPU se proposait d'être une interface de programmation un peu plus générique (similaire à celle que l'on pouvait trouver sur CPU) 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 and 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 (*e.g.* simulations de *flocking*). 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.

Cependant, même dans le cas d'architectures agents très simples, ces travaux pionniers soulignent la difficulté de mettre en place une méthode de conversion générique pour le portage de simulations multi-agents sur GPU, notamment du fait de la grande diversité des modèles. À ce propos, [Perumalla and Aaby, 2008] a étudié les contraintes associées à de telles conversions en réalisant l'implémentation de différents modèles (*Mood Diffusion*[1], *Game of Life*[2] et *Schelling Segregation*[3]) sur CPU (avec *NetLogo* [Sklar, 2007] et *Repast* [North et al., 2007]), 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. Du fait de ces difficultés, le besoin en outils et interfaces spécialisés était très grand et leur apparition va permettre une expansion du GPGPU.

3.1.2 Utilisation des API dédiées au 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 (comme illustré par [Zhang et al., 2011] avec un modèle multi-level permettant la visualisation de la progression d'une tumeur cérébrale). Ainsi, le nombre de travaux utilisant le GPGPU a pu augmenter et de nouveaux outils et frameworks ont vu le jour.

1. 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.
2. Le modèle Game of Life de Conway.
3. Un des premiers exemples d'émergence fondé sur des interactions sociales.

Le plus représentatif est Flame GPU [4] [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 XMML pendant que les comportements de ces derniers sont programmés en C. Les fichiers XMML vont être combinés dans des templates GPUXMML [5] et ensuite vérifiés par un processeur XSLT [6] dans le but d'être compilés dans le langage GPU avec les fichiers de comportements en C. La simulation est ensuite générée puis exécutée par le GPU. La figure 3.2 illustre le processus de génération d'une simulation avec Flame GPU.

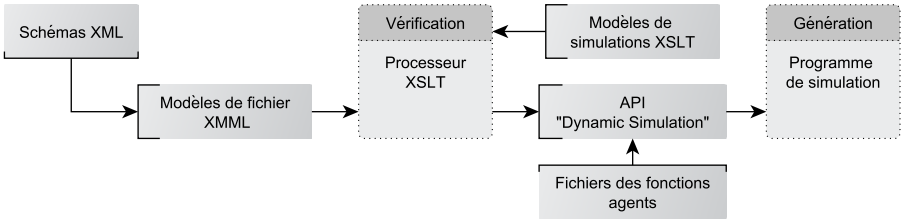


FIGURE 3.2 – Flame GPU, génération d'une simulation (*source : [Richmond et al., 2010]*).

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

Dernièrement, Flame GPU a évolué et se base maintenant sur l'architecture Flame II [Coakley et al., 2016]. Cette nouvelle version met encore plus l'accent sur la généricité et intègre (pour la version GPU) un nouveau découpage des actions des agents (les comportements des agents sont décomposés) permettant une limitation des accès concurrents aux données et une meilleure gestion de la mémoire, ce qui augmente la performance générale du modèle.

Cependant, 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 partent de zéro et se focalisent uniquement sur les gains de performances. Cela est particulièrement vrai dans le cadre de travaux portant sur l'étude de modèles de *flocking*, de foules ou de simulations de trafic.

Les simulations de *flocking*

Le modèle de *flocking* de REYNOLDS [Reynolds, 1987] fait partie des simulations multi-agents les plus connues. C'est un modèle multi-agent qui permet de réaliser des animations réalistes de nuées d'oiseaux artificiels (*boids*) grâce à une approche individu-centrée. En effet, dans ce modèle, chaque entité est considérée comme un agent unique possédant des règles de comportements et

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

5. 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.

6. Le processeur XSLT va vérifier que le code XMML est correct et correspond au template utilisé.

d'interactions. Ces agents peuvent ainsi reproduire de manière expérimentale des comportements de groupes et de modèles biologiques réalistes (*cf.* IBM, de l'anglais *Individual Based Model* [Michel et al., 2009]) .

Motivé par la possibilité de simuler un très grand nombre d'agents (jusqu'à plusieurs millions), des simulations de *flocking* ont été portées sur GPU dans le but de profiter des performances de ces cartes. Ainsi, plusieurs contributions significatives ont vu le jour. On peut notamment citer [Passos et al., 2008] qui introduit la première simulation de *flocking* sur GPU ou encore [Li et al., 2009] qui ajoute la notion d'évitement d'obstacles dans son modèle.

Le travail présenté dans [Erra et al., 2009] est aussi 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.

Dans le but d'accélérer le calcul des simulations de *flocking* et pour en améliorer le rendu graphique, [Silva et al., 2010] 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 et al., 2010] 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 and Hawick, 2011] propose un modèle de *flocking* sur GPU plus complexe capable de simuler un environnement comportant plusieurs espèces d'entités différentes (hétérogénéité des agents). De plus, il est possible de donner à chaque type d'agents une personnalité différente caractérisée par des paramètres spécifiques à l'agent. 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 (comportant de 192 cœurs jusqu'à 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.

Les simulations de foule et de trafics

Les simulations de foules [7] font aussi partie des domaines pour lesquels il est pertinent d'étudier des environnements et des populations d'agents toujours plus grands. Dans ce cas, et comme pour le *flocking*, l'utilisation du GPGPU devient pertinente.

Évolution du framework ABGPU, le *Pedestrian framework* [Richmond and Romano, 2011], basé sur CUDA, ne se focalise pas seulement sur la performance. En effet, il propose de modéliser pour la première fois des agents ayant un comportement cognitif tels que ceux décrits dans [Romano et al., 2005]. Qualifiés de sociaux, ces agents s'adaptent à leur environnement et s'expriment au travers de leurs actions et gestes. Ce framework permet aussi d'intégrer dans la simulation des forces sociales et en particulier celles énoncées par HELBING [Helbing et al., 2002]. Pour cela, [Richmond and Romano, 2011] propose de 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

7. La simulation de foule peut bénéficier d'une approche agent afin de reproduire le plus fidèlement possible le comportement d'individus pour recréer des mouvements de foules humaines réalistes.

ce framework, il est possible de simuler 65 536 agents en 3D et en temps réel avec une carte graphique contenant seulement 96 cœurs.

[Varga and Mintal, 2014] présente 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 "bloquée", la valeur correspond à l'opportunité de se déplacer dans cette cellule. Cette valeur est calculée en fonction des champs de gradients 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 se déplacer de cellule en cellule vers son but en s'adaptant à ce qui se passe dans l'environnement.

Dans [Chen et al., 2015], un framework de simulation dédié aux scénarios d'évacuation a été développé. Chaque individu est modélisé comme un agent dirigé par un mécanisme de prise de décision (chacune des décisions étant en plus pondérée par un poids) offrant, au final, la possibilité de mieux comprendre les interactions entre individus, entre groupes ou encore entre individus et environnement. Ce travail propose aussi un schéma innovant permettant de réduire la surcharge provoquée par un accès trop important aux différents paramètres globaux du système par tous les agents. Pour toutes les expérimentations menées, une carte Nvidia GTX 580 (possédant 512 cœurs CUDA) a été utilisée, ce qui a permis de simuler l'évacuation du Stade National de BEIJING contenant pas moins de 90 000 agents. Ces tests représentent un bon exemple de ce que peut apporter le GPGPU, l'implémentation utilisant CUDA ayant été 38 fois plus rapide que l'implémentation séquentielle correspondante.

Assez similaire aux simulations de foules et d'évacuation, 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 [Strippgen and Nagel, 2009] et [Shen et al., 2011].

Cependant, la simulation de trafic à grande échelle est une tâche difficile car elle nécessite de prendre en compte différents niveaux au sein du même modèle. En effet, il peut être intéressant de faire cohabiter ensemble des modèles microscopiques, pour les sections urbaines, et des modèles macroscopiques, pour les zones d'autoroutes permettant de ce fait d'améliorer le réalisme des modèles simulés. C'est ce que propose de faire JAM-FREE [Abouaissa et al., 2015], un framework multi-agent dédié à la simulation de trafic possédant plusieurs niveaux de représentation. Basé sur SIMILAR (*SImulations with MultI-Level Agents and Reactions*) [8], JAM-FREE permet de simuler des réseaux de trafic de grande taille efficacement en adaptant dynamiquement le niveau de détails tout en testant de nouveaux algorithmes de régulation, d'observation, de routage, etc. Son moteur de simulation étant considéré comme une abstraction de haut niveau, il met à la disposition des utilisateurs les outils et fonctions nécessaires à l'utilisation d'architectures parallèles (tels que les GPU) pour simuler les modèles.

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 implé-

8. `http://www.lgi2a.univ-artois.fr/~morvan/similar/docs/README.html`

mentations d'algorithmes permettant de résoudre le problème de *Pathfinding* et *Pathplanning*[9] 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 de l'algorithmes de DIJKSTRA [10] et A* [11] sur GPU. Ceux-ci seront ensuite modifiés dans [Caggianese and Erra, 2012] afin de s'adapter en temps réel tout au long de la simulation. [dos Santos et al., 2012] apporte aussi une contribution en proposant une nouvelle standardisation pour l'utilisation des GPU dans un contexte agent en respectant le standard FIPA (de l'anglais *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] 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 ses agents.

Cette motivation à créer des algorithmes plus génériques se retrouvent aussi auprès des algorithmes spécialisés dans la recherche de voisins. Ce type de calcul étant extrêmement coûteux en temps et en mémoire. [Li et al., 2014] propose une stratégie visant à les accélérer grâce au GPGPU et les tests menés ont montré que la solution développée a accéléré de manière significative (89 fois) l'algorithme de recherche comparé à son implémentation sur CPU grâce à une carte Nvidia Tesla K20 GPU et ces 2 496 cœurs CUDA. Ces travaux ont ensuite été améliorés en 2016 [Li et al., 2016] avec une meilleure prise en compte des allocations mémoires dynamiques que l'utilisation du GPGPU implique de par son architecture.

Présents dans de nombreux modèles multi-agents, les algorithmes de navigation ou de recherche de voisins 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. Nvidia fournit d'ailleurs une librairie (nvGRAPH [12]) spécialement optimisée pour fonctionner sur les GPU de la marque et dédiée à ce type d'algorithme.

3.1.3 Bilan des implémentations tout-sur-GPU

Jusqu'à présent, les contributions analysées adoptaient toutes une implémentation *tout-sur-GPU* qui consiste à exécuter entièrement le modèle sur le GPU. Grâce à cette approche, on a observé, dans l'ensemble, de nettes accélérations des simulations (au minimum deux fois plus

9. La recherche de chemin, couramment appelée *Pathfinding* ou *Pathplanning*, est un problème de l'intelligence artificielle qui se rattache plus généralement au domaine de la planification et de la recherche de solution. Il consiste à trouver comment se déplacer dans un environnement entre un point de départ et un point d'arrivée en prenant en compte différentes contraintes.

10. L'algorithme de DIJKSTRA détermine le plus court chemin dans un graphe en utilisant le poids lié aux arêtes.

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

12. `https://developer.nvidia.com/nvgraph`

rapides qu'une implémentation sur CPU). Ces résultats sont encourageants compte tenu du fait qu'ils ont été obtenus en utilisant seulement des cartes graphiques standards comportant quelques centaines de cœurs.

Une implémentation tout-sur-GPU est donc intéressante lorsque le but principale est la recherche de performance. Cependant, du fait qu'elle nécessite que le modèle soit entièrement transformé pour pouvoir être exécuté sur le GPU, il est presque impossible de le réutiliser ou bien même de le comprendre car son code devient en grande partie incompréhensible pour toutes personnes n'ayant pas de connaissances solides en GPGPU. Cette approche est donc limitée, d'un point de vue génie logiciel, car elle néglige des aspects tels que la généricité, l'accessibilité et la réutilisabilité que nous avons pourtant défini comme essentielles dés le chapitre 1.

De plus, implémenter un modèle sur GPU n'implique pas obligatoirement un gain de performance, surtout dans le domaine des simulations multi-agents où la diversité des modèles est très grande. La qualité et les choix d'implémentation impactent directement les résultats et les performances qu'il est possible d'obtenir [Aaby et al., 2010]. 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.

Ces difficultés maintiennent le fiable engouement vis à vis du GPGPU ce qui explique en partie le faible nombre de contributions qui traite de l'utilisation de cette technologie dans un contexte agent (*cf.* section 3.5 et figure 3.5). C'est pourquoi il peut être pertinent de trouver des méthodes d'implémentation différentes.

En effet, considérer une nouvelle approche d'implémentation capable de fournir des outils et framework plus réutilisables, plus modulaires, offrant de bonnes performances et une meilleure accessibilité permettra au GPGPU d'être utilisé par un public plus large. Nous allons voir maintenant que l'approche *hybride* représente une solution attractive qui permet de répondre aux différents problèmes soulevés jusqu'ici par l'implémentation tout-sur-GPU.

3.2 Implémentation hybride

Contrairement à une approche tout-sur-GPU, l'approche de conception hybride partage l'exécution d'un système multi-agent entre le CPU et le GPU (la figure 3.3 illustre cette approche). Ainsi, il est possible de choisir ce qui va être exécuté par le GPU en fonction de la nature des calculs et instructions. Moins performante qu'une approche tout-sur-GPU, l'approche hybride possède cependant de nombreux avantages.

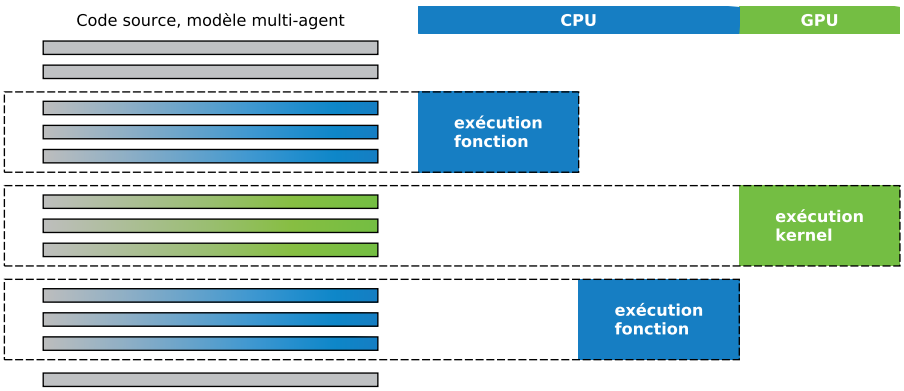


FIGURE 3.3 – Illustration d'une approche hybride.

Par exemple, [Sano and Fukuta, 2013] 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 l'approche hybride utilisée, 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.

Dans [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 la contrainte du tout-sur-GPU de devoir transformer entièrement le modèle, l'approche hybride autorise une intégration plus facile d'agents ayant des architectures hétérogènes.

Un autre exemple de l'intérêt des systèmes hybrides est donné dans [Pavlov and Müller, 2013]. Ces travaux présentent trois approches différentes pour l'implémentation d'un gestionnaire de tâches 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, 2013b] 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, via des modules de calcul GPU, le soin de produire des perceptions de haut niveau à partir des données environnementales brutes [Chang et al., 2005, Payet et al., 2006]. Cette contribution représente précisément le point de départ de nos travaux présentés dans ce manuscrit de thèse et est explicité plus en détails dans le chapitre 4.

[Laville et al., 2014] propose un ensemble d'outils appelé MCMAS (*Many Core MAS*) dont l'objectif est 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 travaux présentés dans [Ho et al., 2015] sont les premiers à examiner la possibilité d'utiliser plusieurs GPU dans le but d'augmenter la scalabilité des modèles agents. Motivés par le besoin évident en terme de visualisation et d'interfaces graphiques dédiées à l'analyse en temps réel des simulations, ces travaux explorent l'implémentation d'un système générique de développement multi-agent utilisant les GPU comme accélérateurs de calcul, le tout porté par un framework écrit en Java et en CUDA basé sur la plateforme MASON [Luke et al., 2005] (*Multi-Agent Simulator Of Neighborhoods*). Ainsi, un framework adapté à l'utilisation de plusieurs GPU ayant la capacité de traiter des données sur plusieurs appareils et offrant également une plus grande accessibilité de programmation a été proposé. Une étude de performance autour de cet outil a montré le potentiel d'accélération de ce dernier en simulant des modèles comportant des millions d'agents. Le gain

de performance d'un ordre de grandeur de deux minimum (par rapport à une implémentation CPU) a été obtenu grâce à des cartes Nvidia Tesla K20 (2 496 cœurs CUDA) et Nvidia GeForce GTX 690 (3 072 cœurs CUDA).

[Shekh et al., 2015] propose une simulation à base d'agents dédiée à l'analyse des risques pandémiques autour de la grippe. Cette simulation définit des prévisions de diffusion de la maladie dans le but d'aider les différents acteurs de la santé publique à prendre les bonnes décisions en cas d'urgence. Basée sur une approche hybride capable de déléguer les calculs les plus lourds au GPU, [Shekh et al., 2015] discute aussi des stratégies envisageables pour porter des MABS sur un ensemble de GPU et pour améliorer le traitement de données en temps réel. Cette simulation est capable de simuler des populations d'agents très importantes d'environ 100 millions d'individus grâce à plusieurs Nvidia Tesla K20 (2 496 cœurs CUDA) [13].

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] définit une architecture logicielle divisée en deux parties : l'AS (*Action Server*) et le CP (*Client Process*). 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.

3.3 Chronologie et synthèse

Dans le but d'avoir un aperçu global des différentes contributions analysées mais aussi pour voir l'évolution de l'utilisation du GPGPU dans un contexte agent, le tableau 3.1 présente une synthèse regroupant les informations les plus importantes des différents frameworks et contributions vus au cours de cet état de l'art.

Référence	Approche	Implémentation	Type d'agent	GPU	Agents
[D'Souza et al., 2007]	*Tout-sur-GPU*	Fonctions graphiques	Réactif	128 cœurs	$\approx 10^6$
[Lysenko and D'Souza, 2008]	*Tout-sur-GPU*	Fonctions graphiques	Réactif	128 cœurs	$\approx 10^6$
[Richmond and Romano, 2008]	*Tout-sur-GPU*	Fonctions graphiques	Réactif	112 cœurs	$\approx 10^4$
[Perumalla and Aaby, 2008]	*Tout-sur-GPU*	Fonctions graphiques	Réactif	112 cœurs	$\approx 10^6$
[Erra et al., 2009]	*Tout-sur-GPU*	CUDA	Réactif	128 cœurs	$\approx 10^6$
[Bleiweiss, 2009]	*Tout-sur-GPU*	CUDA	-	240 cœurs	$\approx 10^4$
[Fischer et al., 2009]	*Tout-sur-GPU*	CUDA	-	256 cœurs	$\approx 10^3$
[Richmond et al., 2010]	*Tout-sur-GPU*	CUDA	Réactif et cognitif	256 cœurs	$\approx 10^5$
[Husselmann and Hawick, 2011]	*Tout-sur-GPU*	CUDA	Reactif et hétérogène	512 cœurs	$\approx 10^4$
[Richmond and Romano, 2011]	*Tout-sur-GPU*	CUDA	Réactif et cognitif	96 cœurs	$\approx 10^4$
[Laville et al., 2012]	*Hybride*	C + OpenCL	Réactif et cognitif	240 cœurs	$\approx 10^3$
[Pavlov and Müller, 2013]	*Hybride*	C + CUDA	Réactif et cognitif	-	-
[Michel, 2013b]	*Hybride*	Java + CUDA	Réactif et cognitif	256 cœurs	$\approx 10^3$
[Laville et al., 2014]	*Hybride*	Java + OpenCL	Réactif et cognitif	240 cœurs	$\approx 10^3$
[Ho et al., 2015]	*Hybride*	CUDA	Réactif et cognitif	2 496 cœurs	$\approx 10^6$
[Chen et al., 2015]	*Tout-sur-GPU*	C++ + CUDA	Réactif et cognitif	512 cœurs	$\approx 10^4$
[Shekh et al., 2015]	*Hybride*	C + CUDA et OpenCL	Réactif et cognitif	2 496 cœurs	$\approx 10^8$

TABLEAU 3.1 – État de l'art des travaux mélant GPGPU et MABS.

La figure 3.4 présente une chronologie des différentes contributions traitant de l'utilisation du GPGPU pour les MABS les plus marquantes tout en intégrant les événements les plus importants relatifs au GPGPU. Ainsi, la frise 1 classe les contributions en accord avec leurs caractéristiques d'implémentations : (1) utilisation directe des fonctions graphiques du GPU (*en italique sur la*

13. Plus précisément, ces performances ont été obtenues grâce à l'utilisation de la fonction *portable pinned memory* de CUDA qui est une mémoire spécifique capable d'être mappée sur l'ensemble des GPU tout en gardant des adresses d'accès uniques permettant ainsi à chaque GPU utilisé d'avoir un accès direct à cette même mémoire aussi bien en écriture qu'en lecture. Cet exemple montre en tout cas l'importance des connaissances nécessaires à avoir dans le matériel (à bas niveau) si l'on veut obtenir des résultats importants.

frise), (2) utilisation des interfaces de développement CUDA ou OpenCL (**en gras sur la frise**) ou (3) utilisation d'une approche hybride. Pendant que la frise 2 présente une évolution des architectures des cartes Nvidia, dans le but de donner au lecteur une idée de la croissance du nombre de cœurs présents à l'intérieur des GPU.

Frise 1 : *Chronologie des travaux mêlant GPGPU et MABS.*

2007 – *SugarScape on steroids* [D'Souza et al., 2007]

2008 – *Premier framework GPU* [Lysenko and D'Souza, 2008]

– *ABGPU* [Richmond and Romano, 2008]

– **Première simulation de flocking** [Passos et al., 2008]

2009 – **Flame GPU** [Richmond et al., 2010]

– **Première simulation de foule et de trafic** [Strippgen and Nagel, 2009]

2010 – Premier système Hybride [Vigueras et al., 2010]

2011 – **Pedestrian framework** [Richmond and Romano, 2011]

2012 – Sworm GPU [Laville et al., 2012]

– TurtleKit GPU [Michel, 2013b]

2013 – [Pavlov and Müller, 2013]

2014 – MCMAS [Laville et al., 2014]

2015 – SIMILAR [Abouaissa et al., 2015]

– MASON with multiple GPU [Ho et al., 2015]

2016 – **Flame GPU with Flame II** [Coakley et al., 2016]

Frise 2 : *Évolution du GPGPU et des architectures Nvidia*

2000 – Début du GPGPU

2006 – Présentation de CUDA

2007 – Sortie de CUDA

– Architecture Tesla : de 16 à 192 cœurs CUDA

2008 – Sortie d'OpenCL

2009 – Architecture Fermi : de 96 à 480 cœurs CUDA

– Sortie de PyCUDA et JCUDA

2010 – Sortie de PyOpenCL et JOCL

2012 – Architecture Kepler : de 384 à 2880 cœurs CUDA

2014 – Architecture Maxwell : de 512 à 3072 cœurs CUDA

2016 – Architecture Pascal : jusqu'à 3 840 cœurs CUDA

FIGURE 3.4 – Chronologie des travaux mélant GPGPU et MABS.

3.4 Synthèse de l'utilisation du GPGPU dans les MABS

De cet état de l'art, il est clair que le GPGPU représente 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 orientée agent reste une tâche difficile. Nous discernons deux raisons principales qui expliquent ces difficultés : (1) le faible degré de généricité des modèles considérés (et la faible réutilisabilité qui en découle) et (2) le manque d'accessibilité des frameworks existants. Ces deux problématiques ont d'ailleurs été identifiées dans [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 différents travaux présentés précédemment à l'égard de la façon dont ils ont tenu compte de ces deux aspects.

3.4.1 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 and 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 prédéfinis qui peuvent être adaptés à différents domaines d'application tels que la biologie (*e.g.* [Richmond et al., 2009b] 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 and Romano, 2011]) ou hétérogènes (*e.g.* [D'Souza et al., 2009] et [Husselmann and 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 and 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 (comme nous le verrons au chapitre 4).

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 la plus à même de fournir le cadre nécessaire à une meilleure intégration du GPGPU dans les simulations multi-agents grâce à la modularité et à la réutilisabilité qu'elle permet. La librairie MCMAS [Laville et al.,

2014] en est d'ailleurs un exemple marquant [14].

3.4.2 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 expliqué dans [Perumalla and Aaby, 2008]. [Lysenko and D'Souza, 2008] et ABGPU [Richmond and Romano, 2008] sont les premières contributions à considérer l'accessibilité comme un critère essentiel. En effet, ces travaux avaient pour ambition de ne requérir que peu de connaissances en GPGPU car ils fournissaient des fonctions GPU prédéfinies de haut niveau, directement utilisables depuis le langage C/C++. Cependant, malgré les efforts d'abstraction réalisés, l'objectif n'a pas été atteint.

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. Il faut cependant 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 [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 via la constitution de bibliothèques d'algorithmes s'exécutant sur le GPU et spécifiquement dédiées au monde SMA.

C'est d'ailleurs sous cette forme, de bibliothèques prêtes à l'emploi, que de nombreux domaines utilisent le GPGPU. 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 leurs communautés respectives.

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]. Mais aussi, de par son ouverture aux autres technologies, elle lève une partie des contraintes du tout-sur-GPU (*e.g.* [Laville et al., 2012] et [Michel, 2013b], utilisation de la programmation orientée objet en parallèle du GPGPU).

3.5 Problématiques abordées dans la thèse

Dans le chapitre 2, nous présentions le GPGPU comme une technologie très intéressante pour toutes les applications où le temps d'exécution et/ou les aspects temps réels sont cruciaux. De l'état de l'art mené dans ce chapitre sur les contributions associant les simulations multi-agents avec le calcul sur carte graphique, nous observons que l'utilisation du GPGPU permet de prendre en considération les problèmes de passage à l'échelle telles que les difficultés de gérer des nombres d'agents de plus en plus importants et/ou des environnements de plus en plus grands dans des systèmes aux ressources souvent limités [15]. De plus, les travaux sur l'utilisation du GPGPU pour l'allocation de tâches, réalisés dans [Pavlov and Müller, 2013], sont une exception notable qui préfigure du fait que le GPGPU peut aussi apporter beaucoup au domaine des SMA en général.

Cependant, nous observons aussi que, malgré les possibilités offertes par le GPGPU, le nombre de publications mêlant GPGPU et simulations multi-agents (ou plus généralement systèmes multi-agents) est toujours très faible comparé à d'autres domaines de recherches. Comme illustré par la

14. L'objectif de cette librairie est de proposer une structure de programmation générique pour les simulations multi-agents sur GPU.

15. 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 Titan X (Pascal) en contient 3 584.

figure 3.5, alors que les contributions traitant du GPGPU et du GPGPU dans la simulation possède un profil de publications (nombre de publications par année) assez similaire (forte accélération à partir de 2007), les travaux mêlant GPGPU et agents peinent à décoller. Il existe donc un réel problème d'appropriation de cette technologie par la communauté agent.

À l'image des travaux de BOURGOIN [Bourgoin et al., 2014] qui référencent l'accessibilité et la généricité comme critères essentiels pour un bon usage du GPGPU, nous retrouvons les mêmes perspectives pour l'utilisation du GPGPU dans le domaine des SMA (*cf.* chapitre 1). Ces perspectives nous ont permis d'identifier deux principaux facteurs limitant l'essor de l'utilisation du GPGPU dans notre communauté : (1) sa difficulté d'implémentation amplifiée par son manque d'accessibilité et (2) le fait que la grande majorité des travaux de recherche traitant de l'utilisation de la programmation GPGPU pour les MABS se focalisent uniquement sur la recherche de performance et sont ainsi très difficilement réutilisables en l'état.

Ces difficultés ont d'ailleurs déjà été évoquées en 2008 par PERUMALLA et AABY qui avaient conclu à l'époque que l'augmentation des performances grâce au GPGPU ne pouvait se faire qu'au détriment de la modularité, réutilisabilité et de l'accessibilité des solutions développées [Perumalla and Aaby, 2008]. Depuis cette étude et malgré les évolutions notables des cartes graphiques et des interfaces de programmation dédiées, le constat reste le même et les limites énoncées perdurent.

Ainsi, les différents travaux de recherche s'intéressant à l'utilisation du GPGPU pour la simulation multi-agent sont toujours divisés en deux catégories distinctes : (1) celle des travaux ne se focalisant que sur la recherche de performance pure et (2) celle des travaux visant à prendre en considération les problèmes d'accessibilité, de réutilisabilité et de programmabilité. Les contributions de cette deuxième catégorie sont, pour la plupart, basées sur une approche hybride qui représente, comme nous l'avons vu précédemment, l'approche la plus prometteuse dans la résolution des différentes limites du GPGPU dans un contexte agent. Cependant, la grande majorité de ces travaux proposent des solutions et outils qui cachent l'utilisation de cette technologie. En effet, le recours au GPGPU se fait de manière transparente pour l'utilisateur.

Au vu de ces différentes conclusions, nous pouvons constater que :

— La proportion de travaux traitant de l'utilisation du GPGPU dans les MABS (et le nombre de publications associées, voir figure 3.5) reste toujours très faible alors que le besoin en ressource de calcul ne cesse d'augmenter.

— Une très grande majorité des travaux essayant d'améliorer l'accessibilité, la généricité et la réutilisabilité du GPGPU considèrent uniquement une utilisation transparente de cette technologie limitant de ce fait le champ d'application des solutions proposées [16].

Ainsi, nous choisissons, dans la suite de ce document, d'explorer une autre piste de recherche. Plutôt que de se focaliser sur l'accessibilité, la réutilisabilité et la généricité des solutions via une utilisation transparente du GPGPU, nous proposons d'utiliser une approche de conception permettant d'adapter et transformer un modèle agent afin qu'il tire partie des architectures massivement parallèles (et donc des GPU) mais sans cacher la technologie sous-jacente.

16. Á cause de la très grande variété des modèles, rendre l'utilisation du GPGPU transparente pour les MABS n'est pas assez générique et ne permet pas de prendre en compte tous les cas et besoins qu'une implémentation de modèles multi-agents avec le GPGPU peut nécessiter.

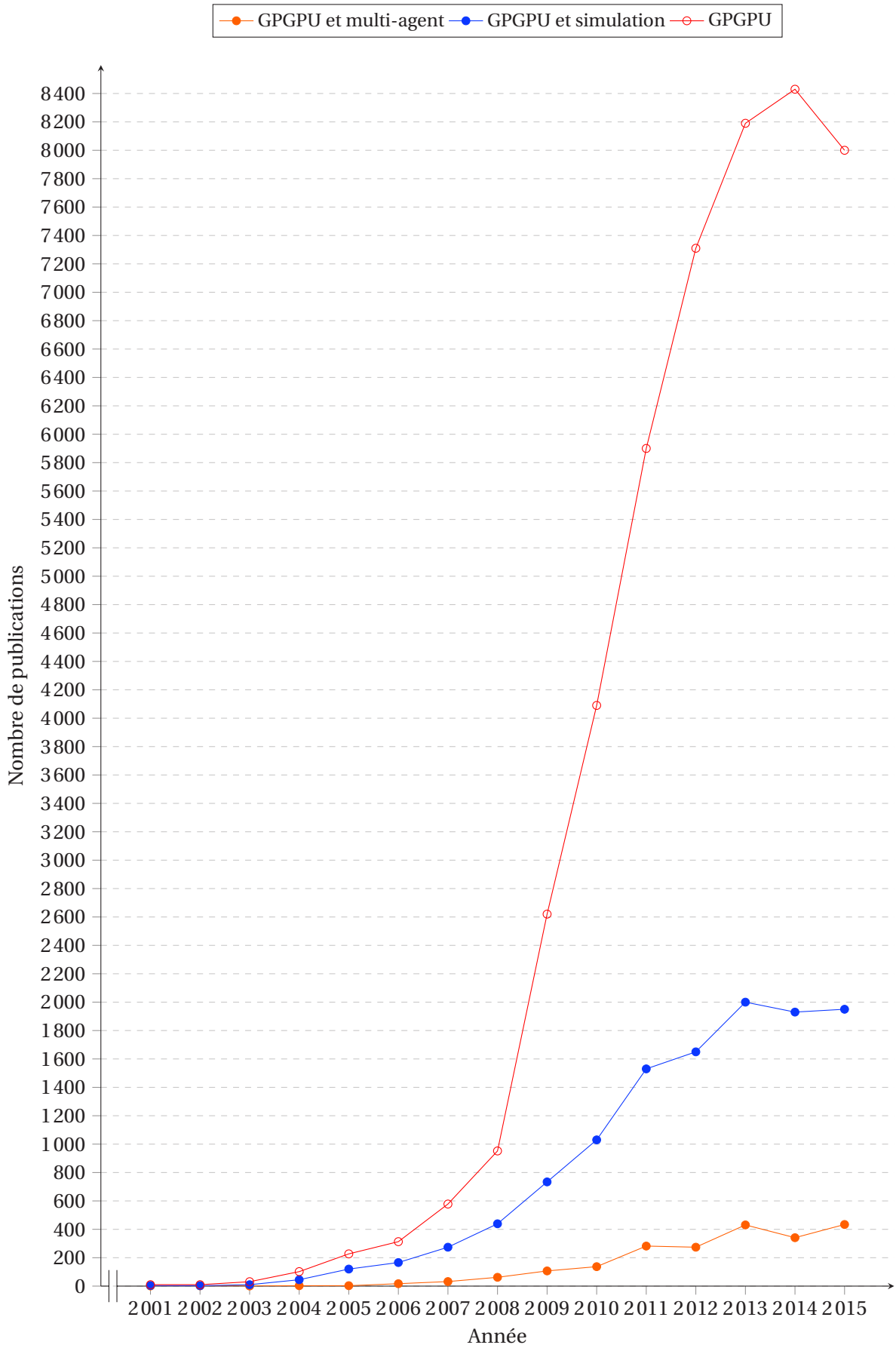


FIGURE 3.5 – Nombre de publications par mots-clés (*source : Google Scholar*).

DEUXIÈME PARTIE

CONTRIBUTIONS

CHAPITRE 4

LE PRINCIPE DE DÉLÉGATION GPU DES PERCEPTIONS AGENTS : ORIGINE ET ÉVOLUTION

Sommaire

De l'état de l'art présenté au chapitre 3, nous avons identifié que les travaux utilisant le GPGPU dans un contexte agent pouvaient être divisés en deux catégories bien distinctes : celle des travaux misant tout sur la performance et celle des travaux tenant compte des difficultés d'implémentation relatives à l'utilisation du GPGPU. Alors que les travaux de la première catégorie adaptent les modèles multi-agents afin qu'ils s'exécutent entièrement sur le GPU, et ainsi profitent de la puissance de ces derniers, ceux de la deuxième catégorie adoptent une approche différente. Qualifiée d'hybride (utilisation conjointe du CPU et GPU), elle permet d'améliorer l'accessibilité, la réutilisabilité et la généricité des solutions développées. La prise en compte de ces trois critères est d'ailleurs considérée comme essentielle pour une meilleure intégration du GPGPU [Bourgoin et al., 2014].

Cependant, rendre l'utilisation du GPGPU transparente limite la généricité des solutions proposées. En effet, du fait de la grande diversité des modèles, les abstractions mises en place pour cacher cette technologie ne permettent pas de prendre en compte tous les cas et besoins qu'une implémentation de modèles multi-agents avec le GPGPU peut nécessiter. Ainsi, notre idée a été de trouver une approche de conception qui simplifie l'utilisation du GPGPU mais sans cacher la technologie sous-jacente.

Parmi tous les travaux vus et référencés, un seul proposait une telle approche : le principe de *délégation GPU des perceptions agents*. Présenté pour la première fois dans [Michel, 2013a] et [Mi-

chel, 2014], la *délégation GPU des perceptions agents* est un principe de conception environnement-centré appartenant au courant E4MAS [Weyns and Michel, 2015]. Ce travail avait pour objectifs de bénéficier des performances du GPGPU, afin d'être capable de réaliser des simulations à large-échelle, tout en conservant l'accessibilité et la facilité de réutilisation d'une interface de programmation orientée objet. De plus, au contraire des travaux cités précédemment, ce principe prônait une utilisation directe du GPGPU.

Poursuivant donc des objectifs similaires à ceux énoncés dans ce travail de thèse, nous avons fait le choix d'utiliser ce principe comme base pour nos développements. Cependant, ce travail n'avait été appliqué que sur un seul cas d'étude bien particulier. Il nous a donc fallu l'étudier en détails (voir section 4.1) avant de pouvoir le prendre en main. Par la suite, notre première expérimentation, basée sur la *délégation GPU des perceptions agents*, a été réalisée sur un modèle de *flocking* [Hermellin and Michel, 2015, Hermellin and Michel, 2016b, Hermellin and Michel, 2016d] et a mené à l'élaboration d'une version plus générique du principe que nous avons nommé principe de délégation GPU (voir section 4.2).

Dans ce chapitre, nous présentons le principe original et sa première application. Nous détaillons également notre première expérimentation de cette approche qui nous a permis de juger de sa faisabilité ainsi que de ses avantages et limites face aux problématiques soulevées jusqu'ici.

4.1 Exploration du principe de délégation GPU des perceptions agents

Le principe de délégation GPU des perceptions agents est une approche qu'il convient de rapprocher des travaux de recherche qui considèrent l'environnement comme un concept fondamental des systèmes multi-agents [Weyns et al., 2007, Ricci et al., 2011].

4.1.1 L'environnement comme abstraction de premier ordre dans les SMA

Considérer l'environnement comme une abstraction de premier ordre est une idée aujourd'hui bien acceptée et son intérêt pour la modélisation et le développement de SMA n'est plus à prouver [Weyns et al., 2007]. En particulier, cela permet de déplacer la complexité du comportement des agents vers l'environnement et donc de bien mieux la gérer [Ricci et al., 2011, Weyns and Michel, 2015].

Un exemple emblématique (validé dans un contexte industriel) est donné dans [Weyns and Holvoet, 2008] où des véhicules automatisés (AGV, *Automated Guided Vehicles*) utilisent un environnement virtuel chargé de calculer la validité de leurs mouvements futurs. Lorsque l'environnement détecte une possible collision, il établit un ordre de priorité entre les mouvements afin de résoudre automatiquement les conflits spatiaux, sans que les agents aient besoin d'intervenir. N'ayant pas à traiter ces problèmes, les agents peuvent alors se focaliser sur leur tâche principale qui est d'aller d'un point A à un point B. Ainsi, la complexité de leur comportement diminue en conséquence. Plus récemment, d'autres travaux ont suivi cette perspective comme notamment l'approche multi-environnements de [Galland et al., 2014].

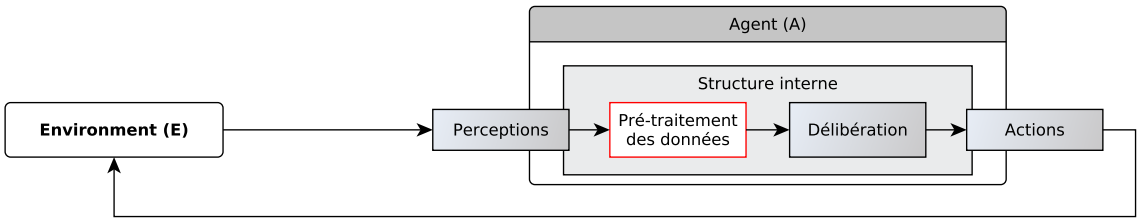
Dans le cadre de la simulation multi-agent, [Payet et al., 2006] propose par exemple de considérer l'environnement comme un point central de la modélisation et montre que cela permet (1) de réduire la complexité du modèle et de (2) grandement faciliter la réutilisabilité et l'intégration des différents processus de simulation qui sont définis. L'approche EASS (*Environment As Active Support for Simulation*) [Badeig and Balbo, 2012] propose de renforcer le rôle de l'environnement en lui déléguant la politique d'ordonnancement ainsi qu'un système de filtrage des perceptions. IODA (*Interaction Oriented Design of Agent simulations*) [Picault and Mathieu, 2011] est quant à elle centrée sur la notion d'interaction et considère que tout comportement réalisable par des agents peut être décrit de façon abstraite (c'est-à-dire en exprimant ce qu'il a de général) sous la forme d'une règle appelée interaction. Dans un contexte plus général, l'approche des *artefacts* intègre dans l'environnement un ensemble d'entités dynamiques structurant les ressources et les outils que les agents vont pouvoir utiliser et partager [Viroli et al., 2006, Ricci et al., 2011].

La considération de l'environnement comme une entité active est donc une approche très intéressante pour la simplification du processus décisionnel 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. Ainsi, tous les travaux de recherche présentés ici visent à réifier une partie des calculs effectués dans le comportement des agents dans de nouvelles structures telles que les interactions ou l'environnement dans le but de répartir la complexité du code et de modulariser son implémentation. Une telle approche permet de concevoir des SMA où il existe une séparation claire entre ce qui relève véritablement des agents et ce qui peut être calculé par ailleurs dans l'environnement. C'est dans cette continuité que se place le principe de *délégation GPU des perceptions agents*.

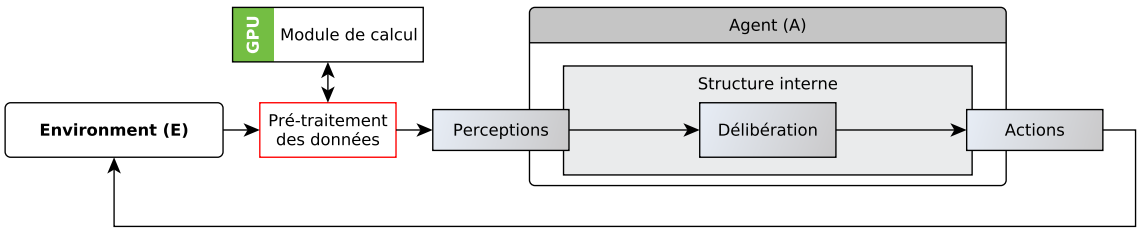
4.1.2 Énoncé du principe

Le principe de *délégation GPU des perceptions agents* arbore un point de vue environnement-centré et s'inspire d'une approche liée au génie logiciel orienté objet tout en se basant sur une utilisation hybride du matériel.

En fait, il a été remarqué qu'une partie des perceptions effectuées par les agents nécessitent un pré-traitement pour rendre les données perçues intelligibles et exploitables [Payet et al., 2006, Michel, 2014] (voir figure 4.1a). Ces calculs, parfois coûteux, n'impliquent nullement l'état interne des agents et peuvent alors être réalisés sans connaître leur état. Il est donc intéressant de soulager les agents de ces traitements et de les déléguer à l'environnement. Le principe de délégation GPU des perceptions agents propose donc de transformer ces calculs en dynamiques appliquées à l'ensemble de l'environnement et traitées par des modules de calcul GPU (voir figure 4.1b). Autrement dit, c'est maintenant l'environnement qui pré-calcule, de manière globale, le résultat des perceptions dont les agents auront besoin.



(a) Avant délégation GPU



(b) Après délégation GPU

FIGURE 4.1 – Application du principe de délégation GPU des perceptions agents.

Ces dynamiques, que nous pouvons qualifier d'environnementales car globales, font parties des notions centrales de cette approche et donc du travail développé dans cette thèse. Il convient donc d'en proposer une définition. Ainsi, d'après [Weyns et al., 2007, Michel et al., 2009, Ricci et al., 2011], une dynamique environnementale peut être définie de la manière suivante :

Définition 4. Dynamiques environnementales
Les dynamiques environnementales sont l'ensemble des dynamiques endogènes de l'environnement qui, contrairement aux agent, ne présentent aucun processus décisionnel. Ces dynamiques s'appliquent à tout l'environnement (elles sont globales) et ne changent pas dans le temps.

Le choix de déporter uniquement les dynamiques environnementales sur le GPU vient des spécificités du GPGPU qui font que les comportements des agents sont difficiles à traduire en code GPU car ils comportent généralement de nombreuses structures conditionnelles qui ne sont pas adaptées à la programmation sur GPU. Au contraire, les calculs correspondant aux dynamiques environnementales se prêtent beaucoup mieux à une parallélisation car ils consistent généralement en un traitement global d'informations dans l'environnement [1].

Finalement, ce principe repose sur une séparation explicite entre le comportement des agents, géré par le CPU, et les dynamiques environnementales traitées par le GPU. L'idée sous-jacente est donc d'identifier, dans le modèle, les calculs les plus coûteux pouvant être transformés en dynamiques environnementales traitées par des modules GPU. Ce principe de conception peut ainsi être énoncé de la manière suivante :

Définition 5. Délégation GPU des perceptions agents [Michel, 2014]
Tout calcul de perception agent qui n'implique pas l'état de l'agent peut être transformé dans une dynamique endogène de l'environnement, et ainsi considéré pour une implémentation dans un module GPU indépendant.

4.1.3 Premier cas d'étude du principe

Le premier cas d'étude du principe de *délégation GPU des perceptions agents* [Michel, 2014] a été réalisé en implémentant un modèle d'émergence multi-niveaux (MLE) [Beurier et al., 2003] dans la plate-forme TurtleKit.

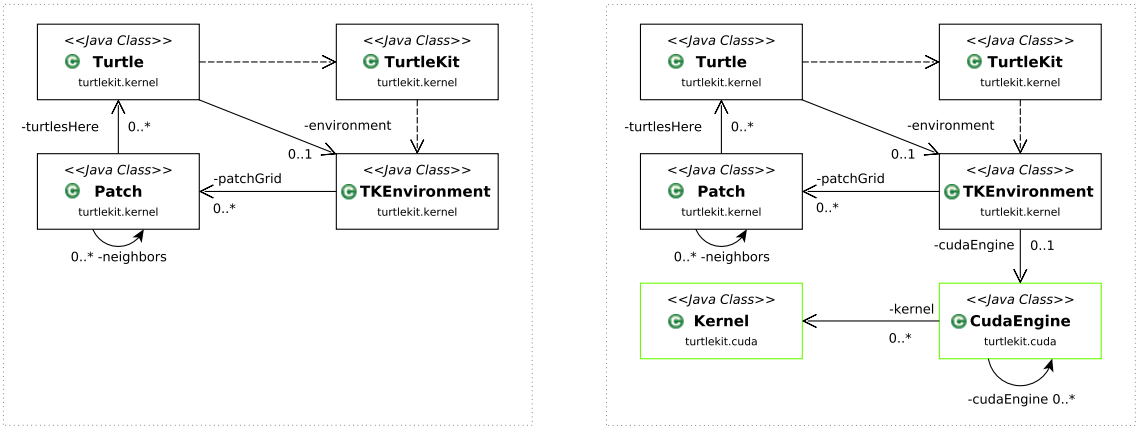
TurtleKit

TurtleKit [2] est une plate-forme qui utilise un modèle multi-agent spatialisé où l'environnement est discrétisé sous la forme d'une grille de cellules [Michel et al., 2005]. Implémentée en Java à l'aide de la librairie de développement multi-agent MaDKit [3] [Gutknecht and Ferber, 2001], TurtleKit repose sur des modèles d'agents et d'environnements inspirés par le langage de programmation Logo. L'un des objectifs principaux de TurtleKit est de fournir aux utilisateurs finaux une API facilement accessible et extensible. En particulier, l'API de TurtleKit est orientée objet et son utilisation repose sur l'héritage de classes prédéfinies.

De par sa conception et l'objectif poursuivi, l'intégration du GPGPU dans TurtleKit a suivi une approche hybride qui consiste à intégrer, de manière itérative, des modules utilisant de la programmation GPU tout en conservant inchangée l'API de TurtleKit. Cependant, implémenter un programme orienté GPGPU nécessite de définir à la fois des *kernels*, qui s'exécuteront sur le GPU, mais aussi des procédures, destinées à être exécutées par le CPU pour organiser l'exécution des *kernels* et récupérer les données ainsi produites. La partie programmation GPU utilise CUDA alors que la partie CPU conserve Java comme langage principal pour TurtleKit. La liaison entre ces deux parties (Java et CUDA) est réalisée par la librairie JCUDA [4] (*Java bindings for Cuda*) qui permet

1. Le parallèle peut être fait entre les dynamiques environnementales et les lois de l'univers du modèle IRM4S (influence/réaction) [Michel, 2007].
2. `http://www.turtlekit.org`
3. `http://www.madkit.org`
4. `http://www.jcuda.org`

d'utiliser directement CUDA dans Java (modulo le changement de syntaxe lié au langage Java). La figure 4.2 illustre l'intégration du GPGPU dans la plate-forme TurtleKit [5]



(a) Avant intégration du GPGPU

(b) Après intégration du GPGPU

FIGURE 4.2 – Intégration du GPGPU dans la plate-forme TurtleKit.

Le choix d'utiliser la technologie CUDA pour intégrer le GPGPU dans TurlteKit est motivé par son antériorité et sa qualité reconnue qui lui permet de posséder la plus grande des communautés d'utilisateurs (en comparaison à OpenCL). De plus, bien que la portabilité offerte par la technologie OpenCL soit un atout majeur de cette dernière, elle a cependant pour conséquence de limiter les optimisations possibles lorsqu'on rentre dans le détail de la programmation d'un *kernel*. Notamment en ce qui concerne la gestion des différentes parties de la mémoire, CUDA propose depuis toujours des primitives qui permettent de gérer finement les différents types de mémoire qui peuvent être utilisés (privée au niveau des *threads*, locale à un bloc, globale, etc.). Enfin, CUDA n'abstrait aucun détail de l'architecture matérielle et permet d'avoir une idée précise des différentes possibilités d'optimisation offertes par le GPGPU [6].

Présentation du modèle MLE

Le modèle multi-agent MLE [Beurier et al., 2003] est très simple et repose sur un unique comportement qui permet de générer des structures complexes qui se répètent de manière fractale. Plus précisément, à partir d'un unique ensemble d'agents non structuré de niveau 0, les agents évoluent pour former des structures (des cercles) de niveau 1 qui servent ensuite à former des structures de niveau 2 et ainsi de suite. Autrement dit, les agents de niveau 0 forment des cercles autour des agents de niveau 1 qui forment eux-mêmes des cercles autour des agents de niveau 2, etc. Le comportement agent utilisé repose uniquement sur la perception, l'émission et la réaction à trois types de phéromones digitales différentes : (1) *présence*, (2) *répulsion* et (3) *attraction*.

— La phéromone de présence (1) est utilisée par un agent pour évaluer combien d'agents d'un certain niveau se trouvent à proximité. Cette phéromone sert ainsi à faire muter un agent vers un niveau supérieur ou inférieur. Dit de manière simplifiée, une mutation peut se produire lorsqu'une zone est surpeuplée ou au contraire vide.

— Les phéromones de répulsion (2) et d'attraction (3) sont utilisées par les agents pour créer une zone d'attraction circulaire autour d'eux, grâce à des taux d'évaporation et de diffusion différents. Au contraire de la phéromone de répulsion, la phéromone d'attraction est émise en faible quantité mais s'évapore doucement, ce qui permet de créer une zone d'attraction circulaire autour de l'agent.

5. L'annexe A présente plus en détails l'architecture de la plate-forme TurtleKit et l'intégration du GPGPU.
6. L'annexe B détaille certaines notions autour de la programmation avec CUDA

Basé sur l'utilisation de ces trois phéromones digitales, le comportement de l'agent peut être décomposé en quatre étapes (illustré par la figure 4.3) : *Perception*, *Émission*, *Mutation* et *Mouvement*. Ainsi, l'état d'un agent est défini uniquement par un entier faisant référence à son niveau. Pour un niveau déterminé, un agent considère uniquement les phéromones de niveaux adjacents.

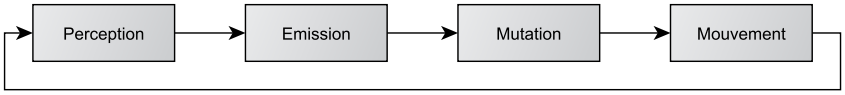


FIGURE 4.3 – Modèle *MLE*, comportement global des agents.

Ainsi, le modèle MLE repose sur l'émission et la perception de phéromones digitales possédant des dynamiques de diffusion et d'évaporation. Ces dynamiques permettent de créer des champs de gradients qui sont utilisés par les agents et vont influencer leurs divers comportements. Cependant, calculer de telles dynamiques demande énormément de ressources de calcul, ce qui limite à la fois les performances et la scalabilité des modèles qui les utilisent, même lorsque peu de phéromones sont mises en jeu. Dans ce contexte, il est pertinent de se tourner vers le GPGPU pour accélérer le calcul de ces dynamiques afin d'améliorer les performances du modèle.

Application de la délégation GPU des perceptions agents sur le modèle MLE

L'application du principe de délégation GPU des perceptions agents sur le modèle MLE [Michel, 2014] a permis de traduire les dynamiques d'évaporation et de diffusion liées aux phéromones car ces calculs sont complètement découplés du modèle comportemental des entités simulées en plus d'être naturellement spatialement distribués (ce qui convient très bien à une adaptation en code GPU). En plus de ces deux dynamiques, l'ensemble des perceptions relatives aux gradients a été réifié dans une unique dynamique environnementale calculée par le GPU. Ainsi, pour ce modèle, deux modules de calculs GPU ont été créés.

Traduction GPU de la dynamique d'évaporation et de diffusion

L'évaporation d'une phéromone consiste simplement à multiplier la quantité présente dans une cellule de l'environnement par un taux d'évaporation, *evapCoef*, compris entre 0 et 1. La traduction en code GPU de cette dynamique a consisté à faire correspondre à chaque cellule de la grille de l'environnement un unique *thread*, identifié par ses coordonnées *i* et *j*. Ainsi, lorsque l'exécution du *kernel* correspondant au processus d'évaporation est appelée sur le GPU, tous les *threads* alloués exécutent l'algorithme 4.1 simultanément.

Algorithme 4.1 : Modèle *MLE*, évaporation en parallèle (GPU)

entrées : Une *grille* de cellules représentant une phéromone, sa *hauteur*, sa *largeur* et le coefficient d'évaporation à lui appliquer, *evapCoef*.

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 si i < largeur et j < hauteur alors
4 |   grille[i][j] = grille[i][j] * evapCoef;
5 fin
```

Quant à elle, la diffusion d'une phéromone dans l'environnement consiste à faire en sorte que toutes les cellules de la grille transmettent une partie de leur contenu vers leurs voisins en fonction d'un coefficient de diffusion, *diffCoef*, compris entre 0 et 1. Ainsi, la traduction de cette dynamique en code GPU repose sur une adaptation similaire à celle de l'évaporation. Cette adaptation est cependant plus complexe car le calcul correspondant doit nécessairement se faire en deux temps pour assurer la cohérence des données. Il faut tout d'abord (1) calculer la quantité de phéromone cédée par chaque cellule à ses voisins et stocker ce résultat dans une grille de données tampon

(*grilleTampon*) puis (2) mettre à jour toutes les valeurs de la grille à partir du résultat précédent. De ce fait, deux *kernels* différents ont été définis et sont appelés successivement (algorithmes 4.2 et 4.3).

Algorithme 4.2 : Modèle *MLE*, diffusion vers la grille tampon en GPU

entrées : La *grille* de phéromone, sa *hauteur*, sa *largeur*, son coefficient de diffusion, *diffCoef* et une grille de données tampon, *grilleTampon*.

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 si i < largeur et j < hauteur alors
4 |   grilleTampon[i][j] = grille[i][j] * diffCoef;
5 fin
```

Algorithme 4.3 : Modèle *MLE*, mise à jour de la diffusion en GPU

entrées : La *grille* de phéromone, sa *hauteur*, sa *largeur* et la *grilleTampon*.

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 si i < largeur et j < hauteur alors
4 |   pour cellule dans voisinageMoore(grilleTampon[i][j]) faire
5 |   |   grille[i][j] = grille[i][j]+quantitéPhero(cellule);
6 |   fin
7 fin
```

Traduction GPU de la perception de gradients

Dans le modèle MLE, chaque agent réalise, à chaque pas de la simulation, un calcul lui permettant de connaître quelle cellule voisine possède le plus, ou le moins, de phéromone d'un certain type afin de décider de son mouvement futur. De tels calculs nécessitent, pour chaque phéromone d'intérêt, de sonder l'ensemble des cellules qui se trouvent autour de l'agent puis de calculer la direction à prendre en fonction des valeurs minimales ou maximales trouvées (voir figure 4.4).

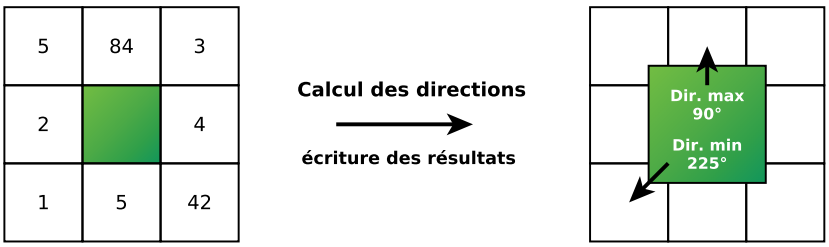


FIGURE 4.4 – Modèle *MLE*, exemple du calcul des directions pour un gradient de phéromone.

Du fait que le calcul de la direction d'un gradient soit indépendant de l'état des agents, il a été possible de réifier l'ensemble des perceptions relatifs aux gradients dans une unique dynamique environnementale calculée par le GPU. L'algorithme 4.4 présente ce *kernel* de calcul dédié aux orientations des gradients. La particularité de ce *kernel* est qu'il repose sur l'utilisation de deux tableaux de données stockant les directions minimales et maximales pour un gradient dans chaque cellule.

Intégration des modules GPU dans TurtleKit

Les traductions des dynamiques d'évaporation, de diffusion et de perception de gradients ont mené à la création de deux modules GPU [7] : un module GPU pour l'évaporation et la diffusion

7. Le code source de ces modules GPU est disponible en annexe B.

Algorithme 4.4 : Modèle *MLE*, perception des gradients en GPU

entrées : La *grille* de phéromone, sa *hauteur*, sa *largeur*
sortie : Deux tableaux contenant les directions minimales *tabMin* et maximales *tabMax*

```
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  si i < largeur et j < hauteur alors
4      pour cellule dans voisinageMoore(grille[i][j]) faire
5          si getDirectionMin(cellule)< directionMin) alors
6              tabMin[i][j] = directionMin;
7          fin
8          si getDirectionMax(cellule)> directionMax) alors
9              tabMax[i][j] = directionMax;
10         fin
11     fin
12 fin
```

(le module *Diffusion*, que nous avons réutilisé au chapitre 5) et un second module GPU pour la perception des gradients de phéromones pour les agents (le module *Perception de gradients*, que nous avons réutilisé au chapitre 6).

L'intégration de ces deux modules GPU dans TurtleKit, illustrée par la figure 4.5, n'a pas posé de problème notamment grâce à la modularité offerte par l'indépendance de ces modules vis-à-vis du modèle agent. En particulier, on peut voir qu'un agent manipule indifféremment, suivant le contexte matériel disponible, des phéromones utilisant une implémentation classique (séquentielle) ou le module GPU correspondant. Le modèle agent n'a donc pas besoin d'être modifié.

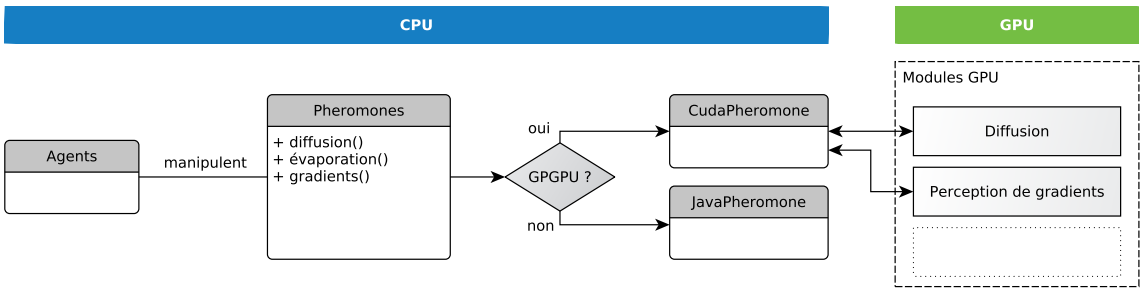


FIGURE 4.5 – Modèle *MLE*, intégration des modules GPU.

Test de performance

Des tests ont été menés dans [Michel, 2014] pour juger des performances de l'approche proposée. Plusieurs simulations ont donc été exécutées utilisant consécutivement la version CPU (séquentielle) puis hybride (utilisant les deux modules GPU) du modèle pour des tailles d'environnement différentes. Dans ces simulations, le niveau maximum d'un agent a été fixé à 5, de telle sorte qu'il existe au plus 15 phéromones à gérer. Ces tests ont été réalisés avec CUDA 4 et JCUDA-0.4.1 sous Linux, à l'aide d'un CPU Xeon @ 3.2 GHz (6 cœurs) et d'une carte graphique Nvidia Quadro 4000 dotée de 256 cœurs (architecture Fermi). La figure 4.6 regroupe les résultats ayant été obtenus lors de ces tests et montre que la version hybride du modèle permet de considérer des environnements beaucoup plus grands et avec des densités plus fortes d'agents. Les gains de performance associés à l'utilisation des modules GPU sont significatifs : la simulation GPU du modèle MLE est jusqu'à sept fois plus rapide que la version originale.

Cependant, bien que la différence soit très marquée pour les petites densités de population, le gain tend à diminuer lorsque le nombre d'entités devient important. Les calculs consacrés aux agents, réalisés par le CPU, prennent de plus en plus de temps si bien que la partie GPU pèse de moins en moins dans le total du temps de simulation.

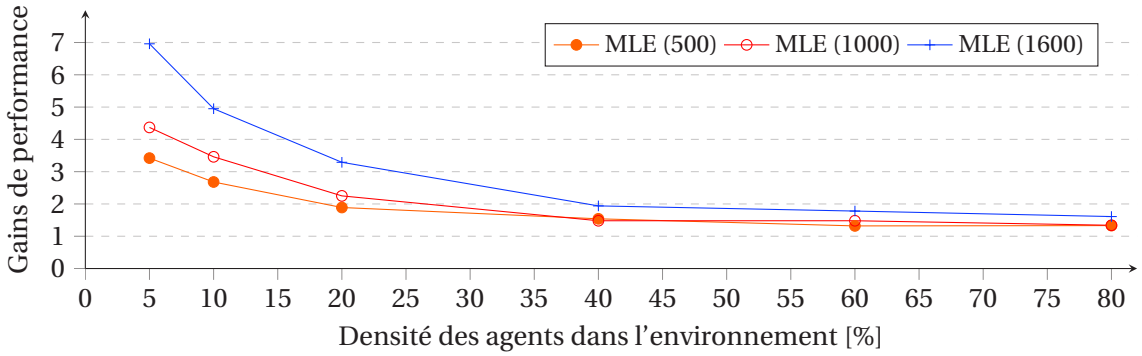


FIGURE 4.6 – Modèle *MLE*, gains de performance entre la version CPU et hybride.

4.1.4 Bilan de l'expérimentation du principe sur le modèle MLE

Grâce à l'étude de cette expérimentation, nous avons conclu que la délégation GPU des perceptions agents, qui est basée sur une approche hybride, se présentait comme une solution attractive capable de répondre aux difficultés occasionnées par le GPGPU. Cette approche prouvait notamment qu'un modèle multi-agent pouvait bénéficier du GPGPU en utilisant cette technologie de manière directe plutôt que de façon transparente. Ainsi, ces travaux offraient de bons résultats en termes de performances mais aussi d'accessibilité et de réutilisabilité.

En effet, cette expérimentation a montré que l'application de ce principe de conception sur le modèle MLE et son intégration dans TurtleKit, en plus de se focaliser sur la modularité (grâce à l'approche hybride), permettait de (1) conserver l'accessibilité du modèle agent dans un contexte GPU (le modèle originel n'a pas été modifié), de (2) passer à l'échelle et travailler avec un grand nombre d'agents sur de grandes tailles d'environnement, (3) promouvoir la réutilisabilité des travaux effectués et enfin (4) déplacer la complexité des agents vers l'environnement. En l'occurrence, il s'agit à la fois de la complexité du code comportemental et de la complexité algorithmique des calculs effectués.

Cependant, malgré le fait que nous avons trouvé cette approche très prometteuse car elle répondait en partie aux différentes problématiques soulevées par l'usage du GPGPU dans un contexte agent, son expérimentation sur le modèle MLE ne représentait qu'un coup d'essai réalisé de manière *ad hoc*. Cette unique expérimentation ne nous permettait donc pas de présager ni des avantages réels de l'approche, ni de sa capacité à pouvoir être appliquée sur d'autres types de modèles. Dans ce contexte, nous avons donc choisi de réaliser un nouveau cas d'étude de ce principe.

4.2 Les *boids* de Reynolds comme cas d'étude

L'objectif étant d'appliquer le principe de délégation GPU des perceptions agents dans un cadre différent de celui proposé par le modèle MLE, nous avons choisi de tester ce principe de conception sur un modèle classique de SMA : les *boids* de REYNOLDS [Reynolds, 1987]. Ainsi, nous pensions être plus à même de juger de la faisabilité et la généricité de cette approche à la suite de cette nouvelle étude [Hermellin and Michel, 2015, Hermellin and Michel, 2016b, Hermellin and Michel, 2016d].

4.2.1 Les *boids* de Reynolds

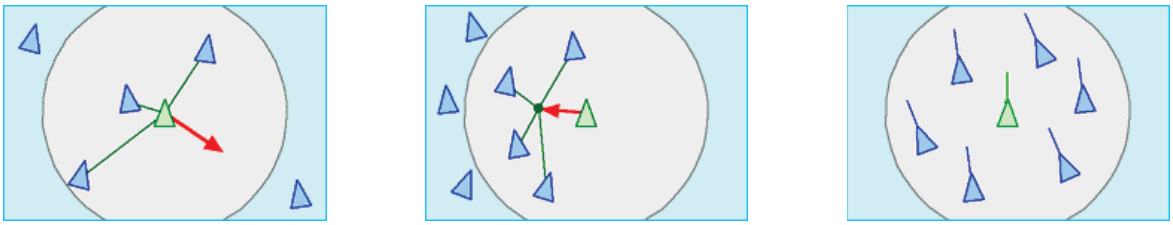
En 1987, lorsque REYNOLDS a voulu créer une animation réaliste d'une nuée d'oiseaux virtuels (que l'on nomme *boids*), il s'est rendu compte qu'il n'était pas possible d'utiliser un script global pour réaliser ce genre d'animation. Son idée a alors été la suivante : les *boids* doivent s'influencer entre eux pour pouvoir se déplacer de manière cohérente et crédible [Reynolds, 1987]. Il propose donc que chaque entité du modèle soit soumise à des forces leur permettant de se déplacer tout

en prenant en compte les mouvements des autres individus présents. Ainsi, chaque entité doit suivre trois règles comportementales :

— **R.1** *Collision Avoidance* : éviter les collisions entre entités (figure 4.7a) ;

— **R.2** *Flock Centering* : rester le plus proche possible des autres entités (figure 4.7b) ;

— **R.3** *Velocity matching* : adapter sa vitesse à celles des autres entités (figure 4.7c).



(a) **R.1** *Collision Avoidance* (b) **R.2** *Flock Centering* (c) **R.3** *Velocity matching*

FIGURE 4.7 – Modèle de *flocking*, illustration des règles comportementales.

Le modèle de *flocking* de REYNOLDS fait partie des simulations multi-agents les plus connues [8]. Ainsi, de nombreuses plates-formes spécialisées dans le développement de SMA l'intègrent en proposant leur propre implémentation. Parmi tous les travaux identifiés, seuls les modèles pouvant être testés et qui mettent à disposition leur code source ont été sélectionnés : NetLogo, StarLogo, GAMA, MasOn, Repast et Flame GPU.

Dans le but d'avoir un aperçu sur la manière dont ce modèle a pu être interprété, nous avons comparé plusieurs implémentations de ce dernier. Pour ce faire, nous avons utilisé deux critères de comparaisons :

— Est-ce que toutes les règles comportementales ont été implémentées ?

— Ces règles sont-elles cohérentes avec celles énoncées par REYNOLDS ?

Implémentation du modèle sur différentes plates-formes

Dans NetLogo [9] [Sklar, 2007], tous les agents se déplacent et essayent de se rapprocher les uns des autres. Si la distance les séparant est trop faible, ils tentent de se dégager pour éviter d'entrer en collision (R.1), sinon ils s'alignent (R.2). Cependant, R.3 n'est pas implémentée : il n'y a aucune gestion de la vitesse des agents qui reste ainsi constante tout au long de la simulation.

Dans StarLogo [10] [Resnick, 1996], chaque agent recherche son plus proche voisin. Si la distance entre eux est trop faible, il tourne et s'éloigne pour éviter d'entrer en collision (R.1). Sinon, il s'approche de lui. La recherche de cohésion n'est pas explicitement exprimée (R.2) et comme pour le modèle précédent, la variation de la vitesse n'est pas présente (R.3).

Dans GAMA [11] [Grignard et al., 2013], les agents commencent par chercher une cible (assimilable à un but) à suivre. Une fois la cible acquise, les agents se déplacent grâce à trois fonctions indépendantes qui implémentent les règles de REYNOLDS : une fonction pour éviter les collisions (R.1), une fonction de cohésion (R.2) et une fonction permettant aux agents d'adapter leur vitesse à celle des voisins (R.3). Ce modèle diffère de celui présenté par REYNOLDS car les agents ont besoin d'une cible pour avoir un comportement de *flocking*.

Dans MasOn [12] [Luke et al., 2005], le modèle utilise un ensemble de vecteurs pour implémenter R.1 et R.2. Ainsi, le mouvement de chaque agent est calculé à partir d'un vecteur global, ce dernier étant composé d'un vecteur d'évitement, d'un vecteur de cohésion (un vecteur dirigé vers le "centre de masse" du groupe d'entités (R.2)), d'un vecteur moment (un vecteur du déplacement

8. Nous avons déjà rencontré ce modèle au chapitre 3.
9. `https://ccl.northwestern.edu/netlogo/`
10. `http://education.mit.edu/starlogo/`
11. `https://code.google.com/p/gama-platform/`
12. `http://cs.gmu.edu/~{}eclab/projects/mason/`

précédent), d'un vecteur de cohérence (un vecteur du mouvement global) et d'un vecteur aléatoire. La vitesse est ici aussi constante pendant toute la simulation, R.3 n'étant pas implémentée.

Dans Repast [13] [North et al., 2007], les règles R.1 et R.2 sont explicitement implémentées. Cependant, R.1 et R.2 sont exécutées par les agents à la suite dans un comportement unique. De plus, dans le comportement de cohésion, la distance entre les agents est forcée (et ne résulte donc pas des interactions entre ces derniers). Enfin, nous notons que R.3 n'est pas implémentée.

Flame GPU [14] [Richmond et al., 2010] est la seule implémentation GPGPU qui répond à nos critères et que nous avons pu tester. Dans ce modèle, R.1, R.2 et R.3 sont explicitement implémentées dans trois fonctions comportementales indépendantes.

Performances des implémentations et résumé de l'étude

Nous avons également évalué, pour chaque modèle, le temps de calcul moyen en millisecondes pour une itération (*ms par itér.*, voir tableau 4.1), les temps les plus faibles étant les meilleurs. Le but de cette évaluation était de donner une idée des possibilités de chaque implémentation. Ainsi, nous avons utilisé comme paramètre commun un environnement de 512 × 512 cellules contenant 4 000 agents. Notre configuration de test était alors composée d'un processeur Intel Core i7 (génération Haswell, 3.40GHz) et d'une carte graphique Nvidia Quadro K4000 (768 cœurs). Il faut noter que pour StarLogo, les résultats observés étaient d'une seconde dès 400 agents simulés. Les performances étant très en dessous des autres plates-formes, nous n'avons pas poussé les tests plus loin. Pour Repast, l'environnement n'est pas discrétisé mais continu. Enfin, pour Flame GPU, il faut préciser deux points importants : (1) la simulation est exécutée dans un environnement 3D et (2) il n'a pas été possible de modifier le nombre d'agents dans la simulation qui est fixé à 2 048. Les résultats de performance de ce dernier n'ont donc pas été pris en compte.

Le tableau 4.1 résume pour chaque modèle les règles de REYNOLDS implémentées, énonce les caractéristiques principales des modèles et donne des informations de performance [15].

	Implémentation règles de Reynolds			**Caractéristiques principales**	**Performances**
	Collision R.1	Cohésion R.2	Vitesse R.3		
NetLogo	X	X		R.3 n'est pas implémentée : la vitesse des agents est fixée pendant toute la simulation	214 ms par itér. (CPU / Logo)
StarLogo	X			Implémentation minimaliste (seul l'évitement d'obstacle est implémenté)	*1000 ms par itér. (CPU / Logo)
GAMA	X	X	X	Comportement de *flocking* seulement lorsque les agents acquièrent une cible	375 ms par itér. (CPU / GAML)
MasOn	X	X		Les règles R.1 et R.2 sont réinterprétées en un calcul de vecteur global qui intègre de l'aléatoire, aucune gestion de la vitesse	45 ms par itér. (CPU / Java)
Repast	X	X		R.3 non implémentée, R.1 et R.2 exécutées à la suite par les agents dans un comportement unique	*37 ms par itér. (CPU / Java)
Flame GPU	X	X	X	Les trois règles sont respectées et implémentées telles que définies par REYNOLDS	*82 ms par itér. (GPU / C, XML)

TABLEAU 4.1 – Le *flocking* dans les plates-formes SMA.

4.2.2 Proposition d'un modèle

De l'aperçu des différentes implémentations des *boids* de REYNOLDS, il apparaît que les règles de *flocking* autorisent une grande variété d'interprétations. Ainsi, la règle pour l'adaptation de la vitesse (R.3) est la moins prise en compte comparée à R.1 et R.2 qui sont implémentées dans

13. `http://repast.sourceforge.net/` et `https://code.google.com/p/repast-demos/wiki/BoidsJava`
14. `http://www.flamegpu.com/`
15. Le sigle * indique que les paramètres de test sont légèrement différents de ceux énoncés dans cette section, à cause de restrictions propres à la plate-forme utilisée ou à l'implémentation.

chaque modèle rencontré (à l'exception de StarLogo). Cependant, lorsque R.3 est implémentée, les comportements collectifs deviennent plus intéressants. En effet, bien qu'il s'agisse là d'une appréciation subjective du rendu visuel, on note que la prise en compte de cette règle influence concrètement le mouvement global des agents et fait apparaitre des dynamiques plus intéressantes car les agents n'ont pas tous la même vitesse. De même, certains travaux divisent la règle R.2 en deux comportements distincts : un comportement d'alignement et un de cohésion. Les modèles explicitant cette différence offrent des dynamiques plus complexes dans le mouvement global des agents.

Ainsi, afin de prendre en compte les points intéressants observés précédemment, nous avons choisi de définir notre propre modèle de *flocking* intégrant explicitement R.1, R.2 (en distinguant l'alignement et la cohésion) et R.3. Nous avons également fait le choix de suivre le principe de parcimonie pour créer un modèle minimaliste (avec le moins de paramètres possible) se focalisant sur la vitesse et l'orientation de l'agent.

Définition du modèle

Dans notre modèle, chaque entité a un comportement global qui consiste à se déplacer dans l'environnement tout en adaptant sa vitesse et sa direction en fonction de ses voisins. Ainsi, la proximité avec les autres agents est testée et selon la distance trouvée, les différentes règles de REYNOLDS sont activées. Plus précisément, chaque agent vérifie s'il n'a pas de voisin dans son entourage. Si aucun agent n'est présent dans son champ de vision, il continue à se déplacer dans la même direction sinon l'agent vérifie sa proximité avec ses voisins. Selon la distance trouvée, l'agent va soit se séparer (R.1) dans le cas où ses voisins sont trop proches, soit s'aligner si le nombre de voisins est inférieur à un seuil de cohésion ou rentrer en cohésion dans le cas où le nombre de voisins est supérieur au seuil défini (R.2). Ensuite, l'agent adapte sa vitesse (R.3), se déplace et recommence le processus. La figure 4.8 illustre ce comportement global.

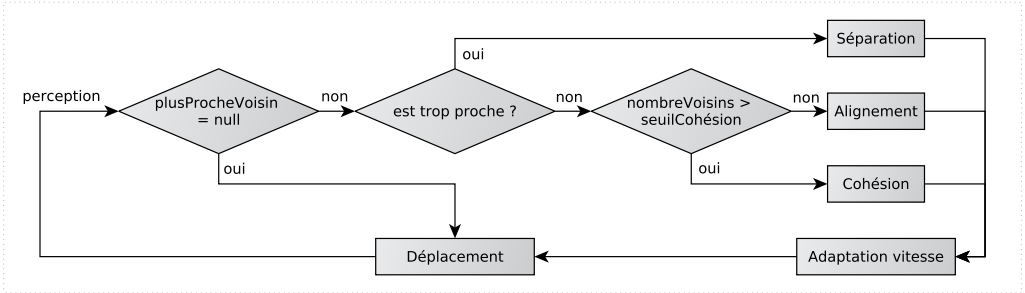


FIGURE 4.8 – Modèle de *flocking*, comportement global des agents.

Ce modèle comporte deux types différents de paramètres : 5 constantes pour le modèle et 3 attributs spécifiques aux agents. Les constantes sont les suivantes :

— *fieldOfView* : le champ de vision de l'agent ;
— *minimalSeparationDistance* : la distance minimale entre deux agents ;
— *cohesionThreshold* : le nombre de voisins pour rentrer en cohésion ;
— *maximumSpeed* : la vitesse maximale ;
— *maximumRotation* : l'angle maximum de rotation.

Les attributs spécifiques à chaque agent sont les suivants :

— *heading* : son orientation [16] ;
— *velocity* : sa vitesse ;
— *nearestNeighborsList* : la liste des voisins présents dans son champ de vision.

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

Définition des comportements

Comportement de séparation R.1 : la séparation se produit lorsque deux agents sont trop proches l'un de l'autre. Ce comportement consiste en la récupération des deux directions : celles de l'agent et de son plus proche voisin. Si ces deux directions mènent à une collision, les deux agents tournent pour s'éviter (voir l'algorithme 4.5).

Algorithme 4.5 : Modèle de *flocking*, comportement de séparation (R.1)

```
   entrées : heading, nearestBird, maximumRotation
   sortie  : heading (la nouvelle orientation de l'agent)
1  collisionHeading ← headingToward(nearestBird) ;
2  si heading isInTheInterval(collisionHeading, maximumRotation) alors
3  |  adaptHeading(heading, maximumRotation);
4  fin
5  return heading
```

Comportement d'alignement R.2 : l'alignement se produit lorsque deux agents se rapprochent l'un de l'autre. Ils vont, dans ce cas, adapter leur orientation de mouvement pour s'aligner et ainsi se diriger vers la même direction (voir l'algorithme 4.6).

Algorithme 4.6 : Modèle de *flocking*, comportement d'alignement (R.2)

```
   entrées : heading, nearestBird
   sortie  : heading (la nouvelle orientation de l'agent)
1  nearestBirdHeading ← getHeading(nearestBird) ;
2  si heading isNear(nearestBirdHeading) alors
3  |  adaptHeading(heading, rotationAngle);
4  fin
5  sinon
6  |  adaptHeading(heading, maximumRotation);
7  fin
8  return heading
```

Comportement de cohésion R.2 : quand plusieurs agents sont proches sans avoir besoin de se séparer, ils ont un comportement de cohésion. Ce dernier consiste à calculer la direction moyenne de tous les agents présents dans le champ de vision. Chaque agent va ensuite adapter son orientation en fonction de la valeur trouvée (voir l'algorithme 4.7).

Algorithme 4.7 : Modèle de *flocking*, comportement de cohésion (R.2)

```
   entrées : heading, nearestNeighborsList
   sortie  : heading (la nouvelle orientation de l'agent)
1  sumOfHeading, neighborsAverageHeading = 0 ;
2  pour bird dans nearestNeighborsList faire
3  |  sumOfHeading+ = getHeading(bird);
4  fin
5  neighborsAverageHeading = sumOfHeading/sizeOf(nearestNeighborsList) ;
6  si heading isNear(neighborsAverageHeading) alors
7  |  adaptHeading(heading, rotationAngle);
8  fin
9  sinon
10 |  adaptHeading(heading, maximumRotation);
11 fin
12 return heading
```

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

voisins (dans la limite autorisée par la constante *maximumSpeed*, voir l'algorithme 4.8).

Algorithme 4.8 : Modèle de *flocking*, adaptation de la vitesse (R.3)

entrées : $velocity$, $nearestBird$, $maximumSpeed$
sortie : $velocity$ (la nouvelle vitesse de l'agent)
1 $velocity \leftarrow$ adaptSpeed(speedOf($nearestBird$), $maximumSpeed$);
2 **return** $velocity$

Implémentation de notre modèle de *flocking* : l'implémentation de notre modèle de *flocking* a été réalisée dans TurtleKit (comme pour MLE). Toutes les ressources nécessaires (code source et documentation) pour exécuter ce modèle sont disponibles en ligne [17]. De plus, un ensemble de vidéos montrant notre modèle en action ainsi que les codes sources des différents modèles mentionnés sont également disponibles sur cette page.

4.2.3 Application du principe de délégation GPU des perceptions agents

Évolution du principe

Nous avons vu que la délégation GPU des perceptions agents consiste à identifier des calculs de perceptions pouvant être transformés en dynamiques environnementales pour ensuite être calculés par des modules GPU. Cette transformation ne peut être effectuée que si les calculs de perceptions n'impliquent pas les états des agents. Ainsi, pour le modèle MLE, ce sont les calculs relatifs à la diffusion, à l'évaporation et au suivi de gradient des phéromones dans l'environnement qui ont été transformés en dynamiques environnementales gérées par des modules GPU. Ces calculs sont indépendants des états des agents car ils n'utilisent les valeurs calculées qu'en tant que perceptions pour leurs comportements de déplacement ou d'évolution.

Cependant, en l'état, le principe de délégation GPU des perceptions agents ne pouvait pas être appliqué sur le modèle de *flocking* que nous avons proposé. En effet, le critère de ce principe ne permettait pas d'identifier des calculs de perception indépendants des états des agents. Dans notre modèle, chacun des calculs fait intervenir un état de l'agent (par exemple son orientation, sa vitesse, etc. voir section précédente). De ce fait, nous avons cherché à faire évoluer le principe afin qu'il puisse prendre en compte notre modèle.

Nous avons ainsi remarqué que même si chacun des calculs faisaient intervenir un état de l'agent, certains ne les modifiaient pas. Basée sur cette observation, nous avons proposé l'évolution suivante : s'il n'existe pas de calculs de perception indépendants des états des agents, il est possible d'identifier des calculs de perception ne **modifiant** pas les états des agents. Ce changement ne remet pas en cause le principe original mais étend son critère de sélection à des calculs précédemment non considérés alors que ces derniers peuvent aussi être transformés en dynamiques environnementales traitées par un module GPU (car ils ne touchent pas à l'état des agents). Nous avons donc proposé une nouvelle version du principe que nous avons nommé principe de *délégation GPU* :

Définition 6. Principe de délégation GPU
Tout calcul de perception ne modifiant pas l'état de l'agent peut être transformé dans une dynamique endogène de l'environnement, et ainsi considéré pour une implémentation dans un module GPU indépendant.

Traduction GPU du calcul des orientations moyennes

Ainsi, dans le modèle de *flocking* que nous avons proposé, le comportement de cohésion contient un calcul de perception qui se prête bien à l'application de cette nouvelle version du principe de délégation GPU. En effet, ce comportement comporte en son sein un calcul qui consiste à

17. `http://www.lirmm.fr/~hermellin/Website/Reynolds_Boids_With_TurtleKit.html`

réaliser la moyenne des orientations des agents en fonction du champ de vision [18]. Pour ce faire, les agents récupèrent une liste de leurs plus proches voisins (*nearestNeighborsList*) et la parcours de manière séquentielle pour calculer la moyenne des orientations de chaque agent présent dans cette liste. Ce processus est résumé dans la figure 4.9 et l'algorithme 4.9 présente le calcul effectué.

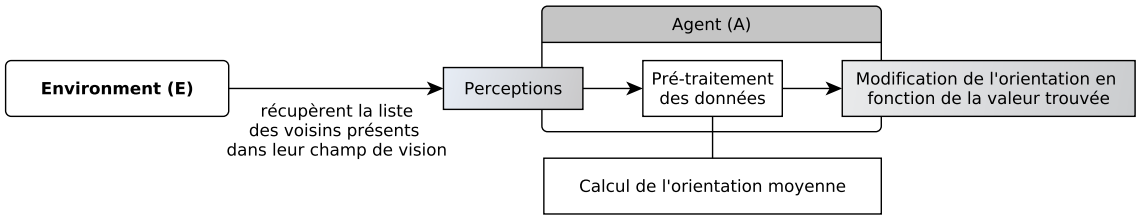


FIGURE 4.9 – Modèle de *flocking*, comportement de cohésion avant délégation GPU.

Algorithme 4.9 : Modèle de *flocking*, calcul de la moyenne des orientations (CPU)

entrées : $nearestNeighborsList$
1 **pour** $bird$ ***dans*** $nearestNeighborsList$ **faire**
2 $\quad sumOfHeading += getHeading(bird)$;
3 **fin**
4 $neighborsAverageHeading = sumOfHeading / sizeOf(nearestNeighborsList)$;
5 **return** $neighborsAverageHeading$

Ce parcours de boucle est lourd car effectué par tous les agents dans leur propre comportement à chaque pas de simulation. Ainsi, le temps de calcul et les ressources nécessaires sont directement impactés par le nombre d'agents présents dans la simulation et par le champ de vision des agents (plus le champ est grand et plus la liste d'agents récupérée peut être longue). Il est donc intéressant de considérer la transformation de ce calcul en dynamique environnementale calculée par un module GPU.

Pour bien comprendre le code des *kernels* de calcul, une précision s'impose. Dans chacun des *kernels*, le lecteur notera que les tableaux de données en entrée et en sortie sont de dimension une alors que nous avons pris le temps d'expliquer comment définir et utiliser une grille de *threads* en deux dimensions sur le GPU. En fait, les spécificités de la programmation GPU font qu'il est plus performant d'utiliser des tableaux à une dimension car ils bénéficient de temps d'accès aux données plus rapide. Par ailleurs, afin de maximiser l'occupation (*cf.* chapitre 2), il est aussi plus efficace d'utiliser une grille de *threads* à deux dimensions plutôt qu'une grille de *threads* à une dimension. Ainsi, nous avons défini une fonction permettant de convertir les coordonnées 2D des *threads* en une coordonnée 1D. Cela nous autorise à utiliser des tableaux de données à une dimension tout en conservant une structuration à deux dimensions pour la grille de *threads*. L'algorithme 4.10 présente cette fonction (nommée `convert1D()`). Cette optimisation est utilisée pour toutes les implémentations réalisées dans ce manuscrit.

Algorithme 4.10 : Conversion d'une coordonée 2D en une coordonée 1D (`convert1D()`)

entrées : x, y, $sizeArray$
sortie : une coordonée 1D
1 **return** $y * sizeArray + x$;

Pour pouvoir appliquer le principe de délégation GPU, il a fallu extraire de l'information des attributs *heading* des agents et transformer le calcul associé (la boucle séquentielle) en une dyna-

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

mique environnementale. Ainsi, un tableau à une dimension [19] (*headingArray*, correspondant à la grille de l'environnement) stocke l'orientation de tous les agents en fonction de leur position. Ce tableau est ensuite envoyé au *kernel Average* qui se charge du calcul des orientations moyennes. En fonction du champ de vision de l'agent (*fieldOfView*), le *kernel* calcule de manière simultanée la moyenne pour tout l'environnement. Plus précisément, chaque *thread* du GPU calcule la moyenne des orientations des cellules voisines d'une cellule en fonction de sa propre position dans la grille GPU, c'est à dire ses identifiants *i* et *j*. Une fois réalisé, les orientations moyennes sont disponibles dans tout l'environnement. Les agents n'ont donc plus qu'à récupérer (à percevoir) dans un tableau à une dimensions (*flockCentering*) la valeur correspondant à leur position (pré-calculée par le *kernel*) et à adapter leur mouvement. Le processus est résumé par la figure 4.10 et illustré par la figure 4.11.

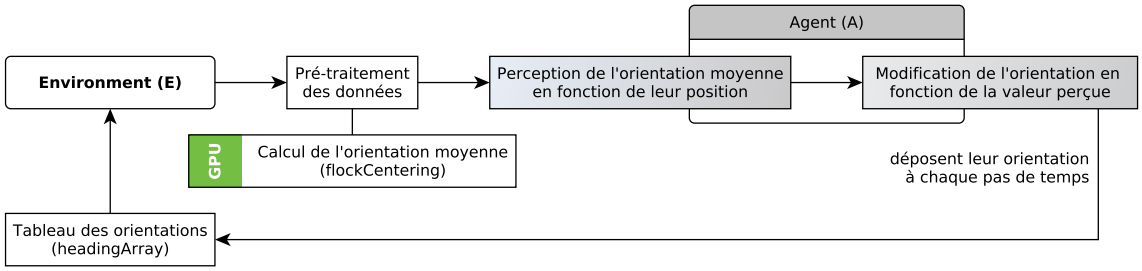


FIGURE 4.10 – Modèle de *flocking*, comportement de cohésion après délégation GPU.

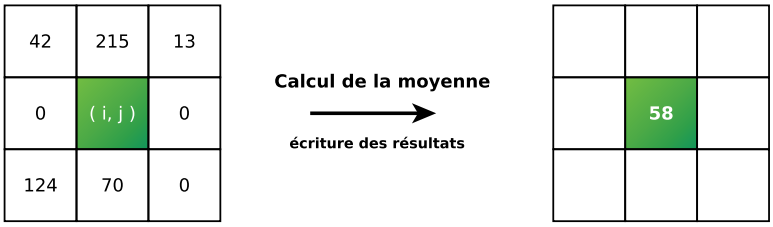


FIGURE 4.11 – Modèle de *flocking*, exemple d'un calcul de moyenne sur le GPU.

L'algorithme 4.11 présente une implémentation du *kernel* de calcul GPU. Après avoir initialisé les coordonnées *i* et *j* du *thread* utilisé et les variables temporaires (*sumOfheading* et *flockCentering*), un test conditionnel vérifie si le *thread* en question ne possède pas des coordonnées supérieures à la taille de l'environnement (représenté ici par le tableau à une dimension *headingArray*). L'ensemble des orientations des voisins se trouvant dans le champ de vision est ensuite additionné puis stocké dans la variable *sumOfheading*. Cette variable est enfin divisée par le nombre de voisins pris en compte. Pour finir, le module retourne le tableau *flockCentering* contenant toutes les moyennes.

Par rapport à la version séquentielle de l'algorithme, on voit que la boucle a disparu. Ainsi, nous profitons du fait que la parallélisation de cette boucle est réalisée grâce à l'architecture matérielle du GPU.

Implémentation et intégration dans TurtleKit

La traduction du calcul des orientations moyennes par le *kernel Average* a permis la création d'un module GPU [20] : le module GPU *Average*. Tout comme pour le cas d'étude portant sur le

19. Les tableaux à une dimension offrent de meilleures performances que ceux en deux dimensions. TurtleKit fournit les outils nécessaires permettant de faire le lien entre les environnements en deux dimensions et des tableaux à une dimension (fonction de conversion de coordonées, etc.).

20. Le code source de ce module GPU est disponible en annexe B.

Algorithme 4.11 : Modèle de *flocking*, calcul de la moyenne des orientations (GPU)

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

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 sumOfHeading, flockCentering = 0 ;
4 si i < width et j < height alors
5 |   sumOfHeading = getHeading(fieldOfView, headingArray[convert1D(i, j)]);
6 fin
7 flockCentering[convert1D(i, j)] = sumOfHeading/sizeOf(nearestNeighborsList) ;
```

modèle MLE, l'intégration du module GPU *Average* a été réalisée dans la plate-forme TurtleKit en version 3.0.0.4. L'environnement de développement CUDA a été utilisé en version 6.5 et la version 0.6.5 de la librairie JCUDA a servi pour faire l'interface entre CUDA et TurtleKit. La figure 4.12 illustre l'application du principe sur notre modèle et l'intégration du module au sein de TurtleKit. La figure 7.1 présente une capture d'écran du modèle de *flocking*, avec module GPU, en action.

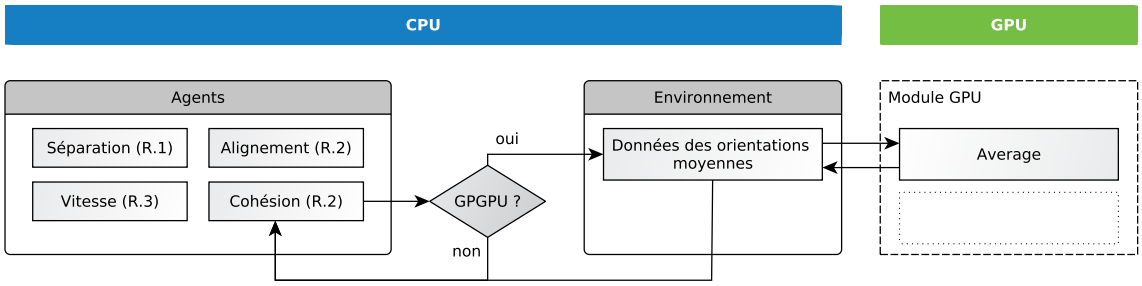


FIGURE 4.12 – Modèle de *flocking*, intégration du module GPU.

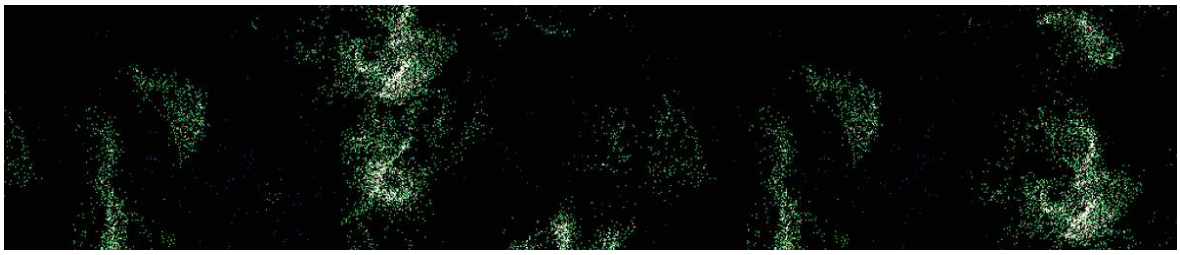


FIGURE 4.13 – Modèle de *flocking*, capture d'écran de la simulation.

4.2.4 Résultats

Évaluation des performances

Dans la section 4.2.1, nous avons présenté pour chacun des modèles de *flocking* sélectionnés, des données relatives aux performances. Cependant, au vu de la très grande disparité entre les différentes implémentations et à cause de l'hétérogénéité des plates-formes (et matériels utilisés), nous n'avons pas pu prendre en compte ces valeurs dans nos résultats de performances concernant notre modèle et l'application de la *délégation GPU* sur ce dernier.

Le protocole expérimental permettant de tester les performances de nos implémentations a été le suivant : nous avons simulé plusieurs tailles d'environnement tout en faisant varier le nombre d'agents et exécuté successivement la version séquentielle du modèle puis la version GPGPU (utilisant le module GPU *Average*). Pour toutes les simulations lancées, le champ de vision des agents était fixé à 10 (un rayon de 10 cases autour de l'agent). Pour rester cohérent avec

les critères d'analyse des modèles de la section 4.2.1, nous avons relevé le temps de calcul d'une itération en millisecondes [21] (les temps les plus faibles étant les meilleurs).

Pour ces tests, et pour tous ceux réalisés dans la suite de ce document, nous avons utilisé une configuration composée d'un processeur Intel Core i7 (génération Haswell, 3.40GHz), d'une carte graphique Nvidia Quadro K4000 (768 cœurs CUDA) et de 16Go de RAM. La figure 4.14 présente les résultats obtenus pour des environnements de taille 256 et 512 avec une densité d'agents variant entre 1 % et 60 %.

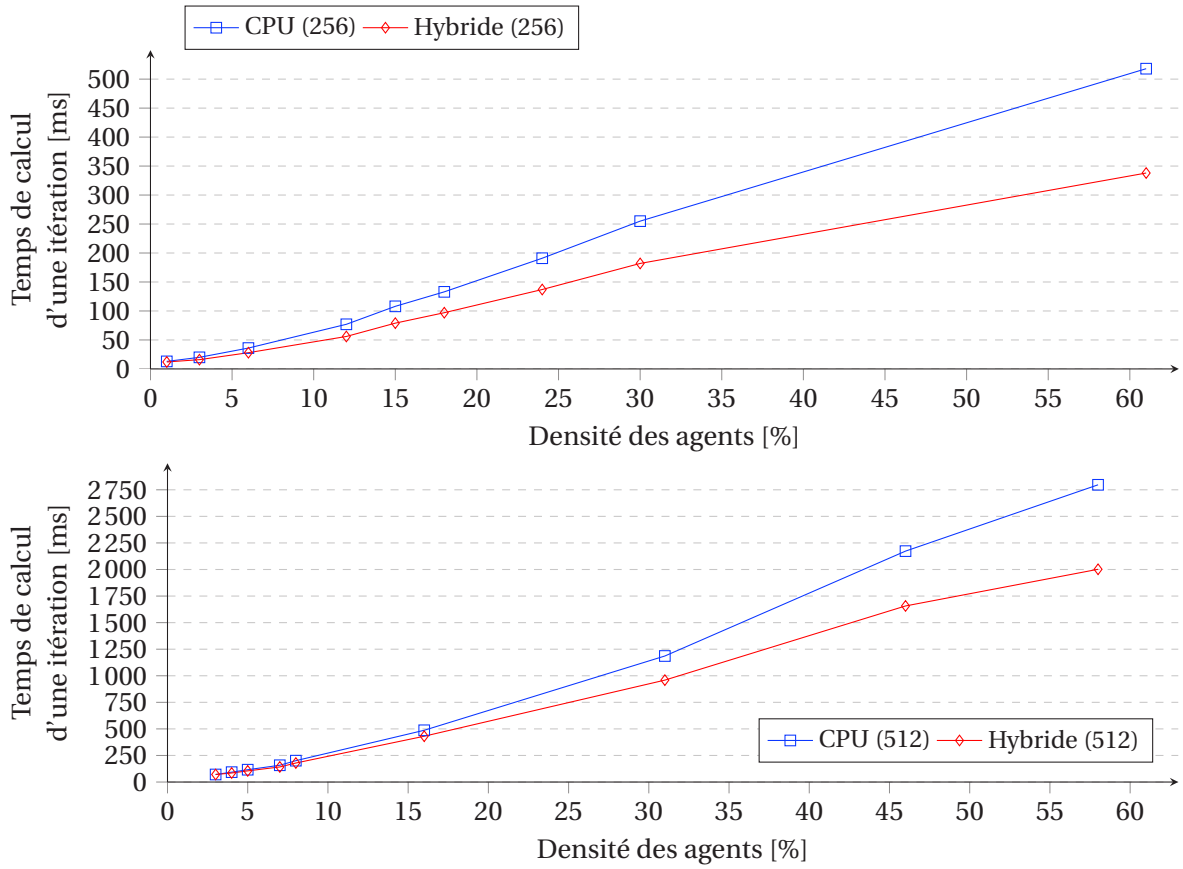


FIGURE 4.14 – Modèle de *flocking*, résultats de performance.

Des graphiques issus de la figure 4.14, nous observons que dans l'environnement de taille 256, le gain augmente assez rapidement puis stagne ensuite autour d'un gain de performances de 40 à 50 % alors que dans l'environnement de taille 512, le profil d'accélération est assez différent. Il est assez lent au départ puis croît rapidement jusqu'à 40 % de gain. Ces profils de gains sont illustrés par la figure 4.15.

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

L'utilisation du GPGPU permet donc de travailler avec des environnements plus grands et/ou des populations d'agents plus importantes. C'est d'ailleurs dans ces situations que cette technologie exprime tout son potentiel. Cependant, il y a des limites. En effet, on peut observer, dans

21. Plus précisément, nous avons effectué la moyenne des temps de calcul d'une itération sur 10 000 pas de temps et pour plusieurs exécutions d'une même simulation.

l'environnement de taille 256, une stagnation des performances au dessus d'une certaine densité d'agents (25 %). Cela s'explique par le fait que, dans notre modèle, seul un calcul profite du GPGPU. Ainsi, les ressources consommées par les comportements et par l'ordonnancement des agents limitent les performances générales (ces derniers étant toujours exécutés par le CPU). Étant basé sur une approche hybride, le principe de délégation GPU ne transforme que les calculs identifiés comme coûteux et répondant aux critères de ce dernier. Ainsi, seul une partie du modèle profite de la parallélisation sur GPU.

Enfin, il faut aussi prendre en compte que les performances obtenues sont significatives si l'on considère le matériel utilisé : notre carte graphique Nvidia Quadro K4000 n'est composée que de 768 cœurs CUDA alors que la nouvelle Nvidia Titan X (Pascal) en contient 3 584.

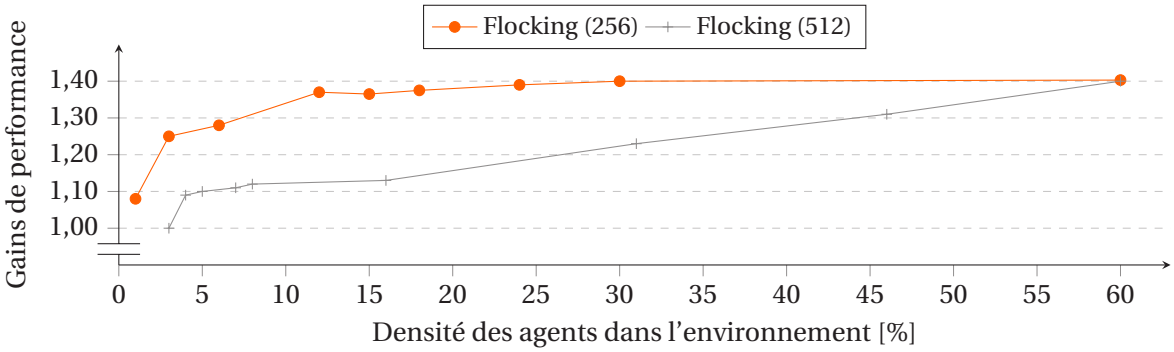


FIGURE 4.15 – Modèle de *flocking*, gains de performance entre la version CPU et hybride.

Avantages conceptuels de l'approche

Un des premiers avantages que nous pouvons souligner est que le module GPU créé à la suite de la délégation du calcul des orientations moyennes est au final indépendant du modèle pour lequel il a été conçu. En effet, les agents ne font que déposer de l'information dans des structures de données adaptées et gérées par les dynamiques environnementales qui sont ensuite envoyées au module GPU.

Ainsi, le module réalisé grâce au principe de délégation GPU ne fait que recevoir et traiter un flux de données. Il n'est donc pas limité au contexte pour lequel il a été conçu. Ce résultat est important car une majorité des travaux traitant de l'utilisation du GPGPU dans un contexte de simulations multi-agents est souvent à "usage unique" et non réutilisable (*cf.* chapitre 3). On a donc une augmentation de la réutilisatilité des outils créés.

De plus, l'application du principe de délégation GPU se base sur un critère simple indépendant de l'implémentation. Cela permet de convertir le modèle et de créer le(s) module(s) GPU de manière assez rapide.

Enfin, la traduction d'une perception calculée dans le comportement de l'agent en une dynamique de l'environnement permet de déplacer une partie de la complexité des agents vers l'environnement. Ceci a pour effet de simplifier le codage du comportement, sa lisibilité et donc de faciliter la compréhension de ce dernier.

4.3 Résumé du chapitre et orientation de recherche

Dans ce chapitre, nous avons introduit la *délégation GPU des perceptions agents* [Michel, 2014] : un principe de conception permettant de répondre, en partie, aux problèmes de réutilisabilité, généricité et accessibilité souvent occasionnés par l'utilisation du GPGPU dans le contexte des simulations multi-agents. Ce principe se veut différent des autres solutions déjà existantes car il propose de transformer un modèle pour qu'il puisse tirer partie de la puissance du GPU plutôt que d'imposer une utilisation transparente de cette technologie impactant la généricité de la solution.

Cependant, la délégation GPU des perceptions agents n'avait été expérimentée que sur un seul modèle multi-agent et implémentée de manière *ad hoc*. La possibilité de réutiliser ce principe dans un autre contexte n'était donc pas avérée. Afin de juger des capacités réelles de cette approche, nous avons proposé de réaliser un nouveau cas d'étude autour de ce principe.

À cause du caractère très restrictif du principe de délégation GPU des perceptions agents (il n'est possible de déléguer que les calculs de perception indépendants des états des agents), nous avons dû proposer une évolution au critère de ce principe car, en l'état, il ne pouvait être appliqué que sur le modèle multi-agent avec lequel il a été expérimenté. Ainsi, étendre le critère du principe aux calculs de perception ne modifiant pas les états des agents, nous a permis d'appliquer le principe de délégation GPU sur notre modèle de *flocking*.

Grâce aux tests effectués, nous avons pu constater que, du point de vue des performances, l'utilisation du GPGPU au travers du principe de délégation GPU a permis d'obtenir des gains de performance intéressant (l'exécution du modèle de *flocking* est jusqu'à 40 % plus rapide). Ces résultats sont bien sûr fortement liés à la densité des agents, à la taille de l'environnement ainsi qu'à l'optimisation des outils et au matériel utilisé mais préfigure d'un très bon potentiel. D'autre part, le principe de délégation GPU représente un modèle de développement qui permet de promouvoir la réutilisabilité des outils créés alors que ce critère essentiel est souvent délaissé dans un contexte GPGPU. En effet, l'utilisation de la délégation GPU permet une séparation explicite entre le modèle agent (les comportements de l'agent) et les dynamiques environnementales autorisant ainsi la création de modules GPU génériques indépendants du modèle agent et donc réutilisables dans d'autres contextes (pour le modèle de *flocking*, un module *Average* a été créé et est maintenant réutilisable).

Ainsi, les résultats observés lors de ces premières expérimentations (MLE et *flocking*) ont été encourageants que ce soit vis-à-vis des performances ou d'un point de vue conceptuel. Ces deux études ont représenté une preuve de concept soulignant que l'utilisation du GPGPU pour le développement multi-agents est possible, mais elles n'étaient que préliminaires et un certain nombre d'étapes restait alors à franchir. En effet, malgré que l'on ait amélioré le champ d'application du principe de délégation (le rendant plus générique), il était encore trop tôt pour présager des réelles avancées qu'allait permettre ce principe de conception. Il nous fallait tout d'abord l'appliquer sur plus de modèles multi-agents.

CHAPITRE 5

EXPÉRIMENTATION DU PRINCIPE DE DÉLÉGATION GPU

Sommaire

Les résultats obtenus lors des premières expérimentations du principe de délégation GPU ont confirmé le potentiel de cette approche au regard des objectifs poursuivis. Cependant, les deux cas d'études précédemment présentés ne nous offraient pas un recul suffisant sur son utilisation et donc sur la meilleure façon de l'implémenter. Notamment, il nous était nécessaire de tester de manière plus poussée la nouvelle version du principe de délégation GPU [Hermellin and Michel, 2016c] proposée lors de l'expérimentation sur le modèle de *flocking*. Ceci afin de nous permettre d'évaluer, par la pratique, le champ d'application réel de l'approche ainsi que sa capacité à produire des éléments facilement réutilisables dans d'autres simulations et par là ses avantages et ses limites [Hermellin and Michel, 2016f]. Nous présentons donc, dans ce chapitre, une expérimentation du principe de délégation GPU sur quatre nouveaux modèles multi-agents (*Game of Life*, *Segregation*, *Fire* et *DLA*) ainsi qu'un bilan autour de l'utilisation de ce principe.

5.1 Expérimentations

Les modèles utilisés pour ces nouveaux tests n'ont pas été choisis au hasard. En 2008, AABY et PERUMALLA ont été les premiers à proposer une étude sur l'intérêt d'utiliser le GPGPU dans le contexte des simulations multi-agents [Perumalla and Aaby, 2008]. Pour ce faire, ils ont sélectionné plusieurs modèles multi-agents différents et les ont adaptés via une approche tout-sur-GPU. Cette variété dans le choix des modèles a permis de proposer un aperçu plus ou moins global de ce qu'il était possible de faire avec le GPGPU dans un contexte agent.

La diversité étant une des caractéristiques clé du domaine agent (comme l'illustre le grand nombre de plates-formes de développement de simulations multi-agents, *cf.* chapitre 1 et les nombreuses approches permettant d'utiliser le GPGPU dans ces simulations, *cf.* chapitre 3), nous avons considéré que prendre également en compte des modèles très différents allait apporter une certaine crédibilité à l'approche proposée en plus de tester sa généricité. Nous avons donc choisi de reprendre deux des modèles utilisés dans l'étude de PERUMALLA et AABY (*Game of Life* et *Segregation*) [Perumalla and Aaby, 2008] et, pour introduire plus de variété, deux modèles parmi ceux disponibles dans la librairie de NetLogo (*Fire* et *DLA*).

Pour chacun de ces quatre modèles, nous avons suivi le schéma suivant :

— Nous présentions le modèle et sa dynamique globale ;

— Nous identifiions les calculs les plus gourmands qui pourraient bénéficier du GPGPU ;

— Nous implémentions le principe de délégation GPU ;

— Nous réalisions des tests de performance.

Tous les tests de performance menés ont suivi le même protocole expérimental qui a consisté à simuler consécutivement les versions CPU et hybride (CPU + GPU) de chaque modèle en faisant varier la taille de l'environnement ainsi que la densité des agents. Chaque simulation est alors exécutée plusieurs fois sur une période de 10 000 pas de temps pour permettre de calculer le temps moyen d'exécution d'une itération (le temps le plus faible étant le meilleur), nous donnant ainsi une valeur permettant de comparer les deux versions du modèle. Pour ces tests, nous avons réutilisé la même configuration que précédemment [1]. Niveau logiciel, la version 3.0.0.4 de la plateforme TurtleKit a été utilisée ainsi que CUDA en version 6.5 et JCUDA en version 0.6.5. Les codes sources et les vidéos de ces différentes expérimentations sont disponibles en ligne [2].

5.1.1 Le modèle *Game of Life*

Présentation du modèle

Le jeu de la vie défini par CONWAY [Gardner, 1970] n'est pas vraiment un jeu au sens ludique du terme. C'est à l'origine un automate cellulaire qui démontre que des motifs complexes peuvent émerger de la mise en application de règles simples. Le jeu se déroule sur une grille à deux dimensions dont les cases peuvent prendre deux états distincts : vivantes ou mortes. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines à partir des règles pré-établies suivantes :

— Une cellule vivante possédant moins de deux ou plus de trois voisines vivantes meurt ;

— Une cellule vivante possédant deux ou trois voisines vivantes le reste ;

— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).

Ainsi, bien qu'il ne contienne pas d'agents, il définit une dynamique de l'environnement qui est représentative de celles rencontrées dans les systèmes multi-agents (ce qui justifie son intégration dans l'étude menée).

Pour notre expérimentation, nous avons défini un modèle contenant seulement un environnement représenté sous la forme d'une grille carrée à deux dimensions torique et discrétisée dans laquelle les cellules sont initialisées de manière aléatoire [3]. À chaque pas de simulation, les états des cellules sont mises à jour en fonction des règles précédemment définies. La figure 5.1 illustre les différentes étapes de la simulation.

1. Pour rappel, cette configuration est composée d'un processeur Intel i7-4770 (génération Haswell, 8 cœurs cadencés à 3.40 GHz) et d'une carte graphique Nvidia K4000 (architecture Kepler, 768 cœurs CUDA).

2. `http://www.lirmm.fr/~hermellin/` section "Doctorat".

3. La probabilité pour qu'une cellule soit morte ou vivante est la même.

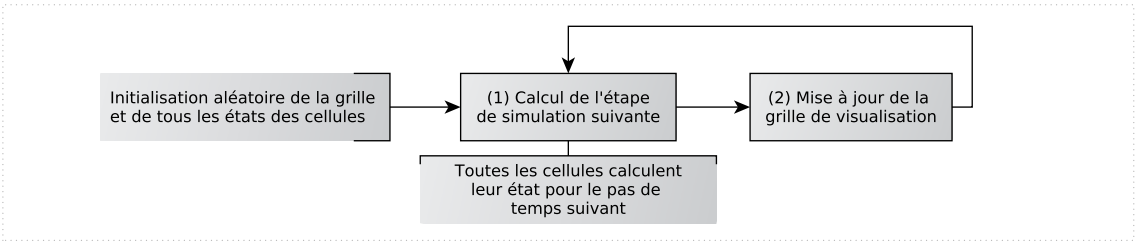


FIGURE 5.1 – Modèle *Game of Life*, dynamique globale du modèle.

Application du principe de délégation GPU

Le calcul qui nécessite le plus de ressource dans ce modèle se situe à l'étape (1) et consiste en la réalisation d'un calcul itératif (une boucle séquentielle) sur toutes les cellules de l'environnement : il met à jour les états des cellules pour le prochain pas de simulation. Ainsi, plus l'environnement est grand et plus le temps nécessaire à la réalisation de ce calcul est important. Vu que ce modèle est un automate cellulaire, l'évolution de l'état des cellules (et leur mise à jour) est déjà une dynamique de l'environnement compatible avec le critère du principe de délégation GPU. Il est donc possible de traduire le calcul de cette dynamique en module GPU.

Afin de créer un module GPU efficient, il est nécessaire de se focaliser sur les structures de données qui vont lui être envoyées. Plus précisément, pour éviter d'avoir des temps de transfert trop importants entre le CPU et le GPU, les données doivent être envoyées seulement une fois par pas de simulation. Ainsi, les états de chaque cellule (1 pour une cellule vivante, 0 pour une cellule morte) sont écrits dans un tableau à une dimension (`statesArray`, correspondant à la taille de l'environnement) en fonction de sa position. Ce tableau est ensuite envoyé au GPU qui réalise la somme des états des cellules (ce qui revient à faire la somme des cellules vivantes) présentes dans le voisinage de MOORE de chaque cellule de l'environnement. La figure 5.2 illustre ce calcul.

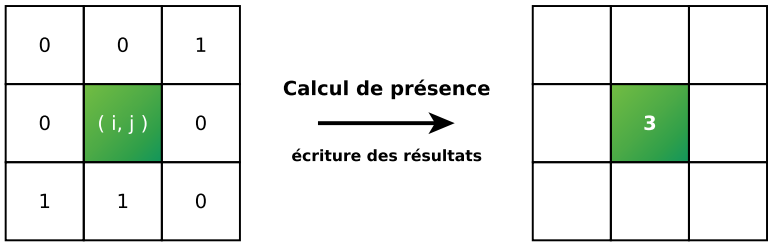


FIGURE 5.2 – Modèle *Game of Life*, exemple d'un calcul de présence sur le GPU.

Le résultat de cette somme est ensuite sauvegardé dans un nouveau tableau à une dimension (`resultArray`). Ainsi, chaque cellule du tableau résultat contient une valeur comprise entre 0 (aucune cellule vivante dans mon voisinage) et 8 (toutes mes voisines sont vivantes). Ce tableau résultat est utilisé pour mettre à jour la grille et calculer le prochain pas de simulation en accord avec les règles de transitions du modèle. L'algorithme 5.1 présente une implémentation du *kernel* GPU correspondant.

La traduction du calcul du nombre de cellules voisines vivantes a mené à la création d'un nouveau module GPU : le module *Voisinage*. En effet, alors que ce module de calcul peut sembler très simpliste (il réalise une simple somme sur les cellules voisines), il peut être utilisé d'une manière plus générique en tant que fonction d'analyse de voisinage (type d'agents, nombre d'agents, etc.). De ce fait, il a été réutilisé dans les implémentations des modèles *Segregation* et *DLA*. Le seul prérequis est de bien formater les données qui lui sont envoyées.

La figure 5.3 illustre l'intégration du module GPU *Voisinage* dans le modèle *Game of Life*.

Algorithme 5.1 : Modèle *Game of Life*, détection du nombre de voisins vivants (GPU)

entrées : $width$, $height$, $statesArray$[]
sortie : $resultArray$[] (le nombre de voisins vivants)

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 sumOfState = 0 ;
4 si i < width et j < height alors
5     pour cellule dans mooreNeighborhood(statesArray[convert1D(i, j)]) faire
6         sumOfState+ = getNeighborsValue(cellule);
7     fin
8 fin
9 resultArray[convert1D(i, j)] = sumOfState ;
```

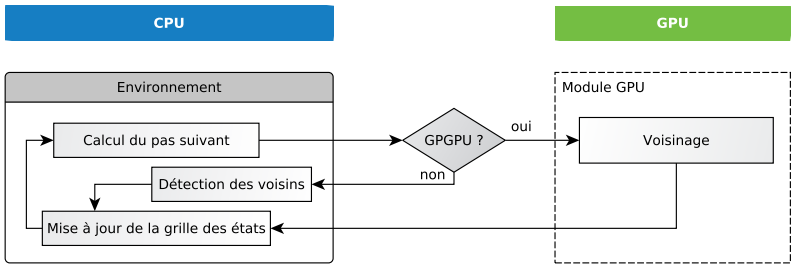


FIGURE 5.3 – Modèle *Game of Life*, intégration du module GPU.

Performance du modèle

Pour tester les performances du modèle *Game of Life*, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fixé la densité des agents à 50 %. La figure 5.4 présente les résultats obtenus.

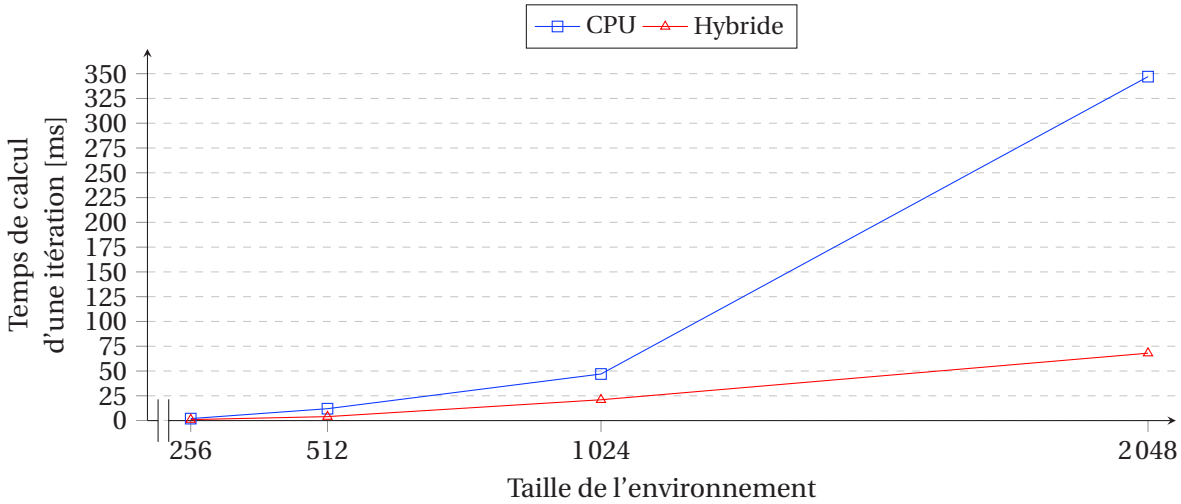


FIGURE 5.4 – Modèle *Game of Life*, résultats de performance.

Le graphique issu de la figure 5.4 montre clairement que l'intégration du module *Voisinage* augmente les performances du modèle. Sans surprise, ce gain est d'autant plus important que l'environnement est grand : jusqu'à 5 fois plus rapide qu'une exécution séquentielle du modèle.

5.1.2 Le modèle *Segregation*

Présentation du modèle

Le modèle social *Segregation* développé par SCHELLING [Schelling, 1978] traite de la dynamique du partage de l'espace entre différents groupes. Dans le contexte d'un quartier où plusieurs types d'entités sont mélangés, il démontre à quelles conditions ce même quartier peut devenir ségrégé. Et cela même si ce n'est pas ce que souhaitaient ses habitants : si chacun admet, voire souhaite, un voisinage différent de lui (mais "pas trop" sinon il quitte le quartier) le résultat final dépendra de la proportion de départ et de ce dernier seuil. SCHELLING montre, qu'à raison de cette tolérance limitée, le quartier peut se retrouver dans deux situations stables possibles : une de ségrégation pure ou une où les entités restent mélangées.

Pour notre expérimentation, nous avons défini un modèle dans lequel deux types d'agents sont distribués aléatoirement (dans les mêmes proportions) dans un environnement carré et discrétisé à deux dimensions : les agents rouges et les agents verts. Le but de ces agents est d'être heureux en restant à proximité d'agents de même couleur (chaque agent rouge veut vivre près d'au moins un certain nombre d'agents rouges, il en est de même pour les agents verts). Si ils ne sont pas satisfaits à leur position, les agents se déplacent aléatoirement vers un nouvel emplacement afin de trouver un meilleur endroit qui correspondra mieux à leur objectif de bonheur. La figure 5.5 illustre les différentes étapes de la simulation.

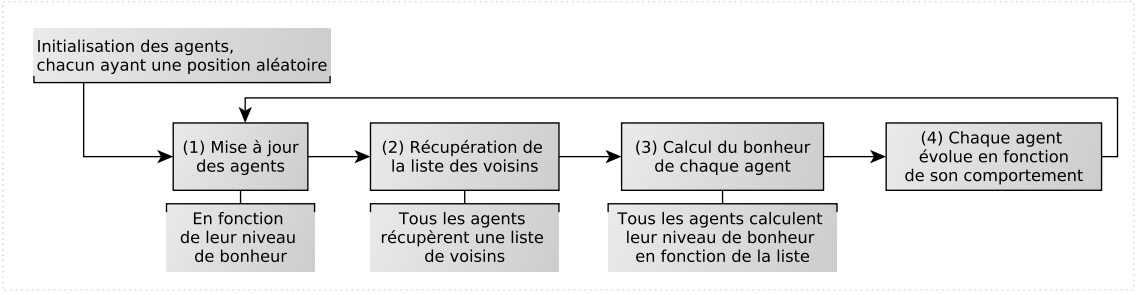


FIGURE 5.5 – Modèle *Segregation*, dynamique globale du modèle.

Application du principe de délégation GPU

Les calculs les plus gourmands dans ce modèle sont situés aux étapes (2) et (3) qui consistent à éditer, pour chaque agent, une liste des voisins les plus proches puis à parcourir cette liste afin de calculer leur bonheur (calculer en fonction du nombre et du type d'agents présents autour d'eux). Vu que ces deux étapes reposent sur la réalisation et le parcours de nombreuses boucles séquentielles (et cela à chaque pas de simulation), le temps de calcul requis va considérablement s'accroitre lorsque le nombre d'agents va augmenter. En accord avec les critères du principe de délégation GPU, le calcul du bonheur ne peut être transformé en dynamique environnementale car il modifie l'état de l'agent. Cependant, il est possible de transformer la perception préalable à ce calcul de bonheur qui consiste à établir le voisinage de chaque agent (le type et nombre d'agents).

Le bonheur des agents est défini en testant la couleur des voisins et en additionnant le nombre d'agents de chaque communauté. Cela revient à compter le nombre d'agents de chaque communauté présent dans le voisinage pour chaque cellule de l'environnement. Il est donc possible de réutiliser le module *Voisinage* précédemment créé. Pour cela, il nous faut adapter la structure de données afin d'utiliser correctement ce module GPU. Ainsi, chaque agent va spécifier, à chaque pas de simulation, à quelle communauté il appartient en écrivant dans un tableau à une dimension (`communityArray`, qui correspond à la taille de l'environnement) un nombre propre à sa communauté : 1 pour la communauté verte et -1 pour la rouge. Ce tableau est ensuite envoyé au GPU qui réalise la somme de ces nombres (le voisinage de MOORE est une nouvelle fois utilisé)

pour tout l'environnement et écrit le résultat dans un nouveau tableau (`resultArray`). La figure 5.6 illustre ce calcul. Chaque case du tableau résultat contient donc une variable pouvant prendre une valeur comprise entre -8 (tous les agents du voisinage sont rouges) et 8 (tous les agents sont verts). Les valeurs présentes dans ce tableau sont ensuite utilisées par les agents, en fonction de leur position, pour calculer leur bonheur.

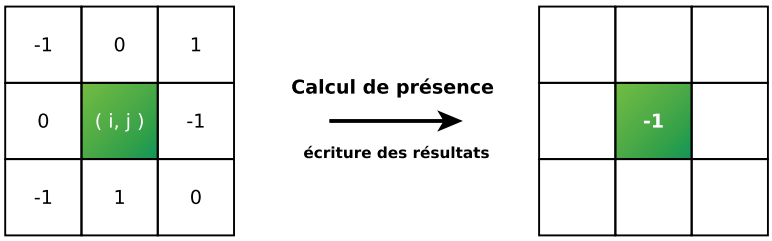


FIGURE 5.6 – Modèle *Segregation*, exemple d'un calcul de présence par type d'agents sur le GPU.

Ainsi, dans le cadre de l'application du principe de délégation GPU sur le modèle *Segregation*, il a été possible de réutiliser le module *Voisinage*, initialement créé lors de l'implémentation du modèle *Game of Life*, en adaptant seulement les données envoyées à ce module. La figure 5.7 et l'algorithme 5.4 illustrent l'intégration du module *Voisinage* dans le modèle *Segregation*.

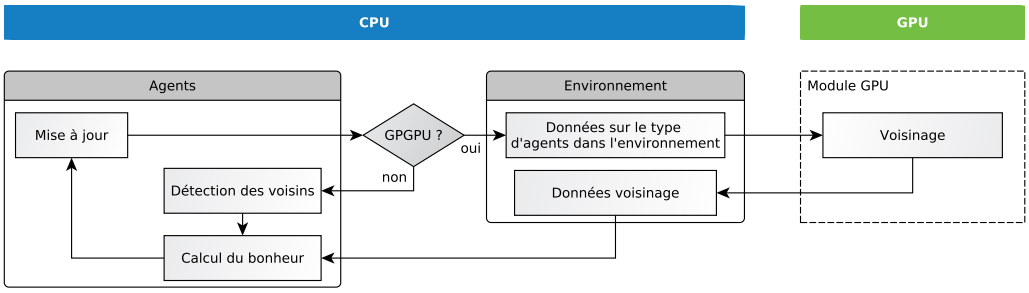


FIGURE 5.7 – Modèle *Segregation*, intégration du module GPU.

Algorithme 5.2 : Modèle *Segregation*, ordonnancement de la simulation

```
tant que simulationRunning == true faire
    /* Activation des agents                                    */
1   pour agent dans listOfAgents faire
2       computeHapiness(communityArray[]);
3       updateAgent();
4       getNeighbors();
5       fillCommunityArray(communityArray[]);
6   fin
    /* Activation de l'environnement                            */
7   executeGPUKernel( Voisinage(communityArray[]) );
fin
```

Performance du modèle

Pour tester les performances du modèle *Segregation*, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fixé la densité des agents à 90 %. La figure 5.8 présente les résultats obtenus.

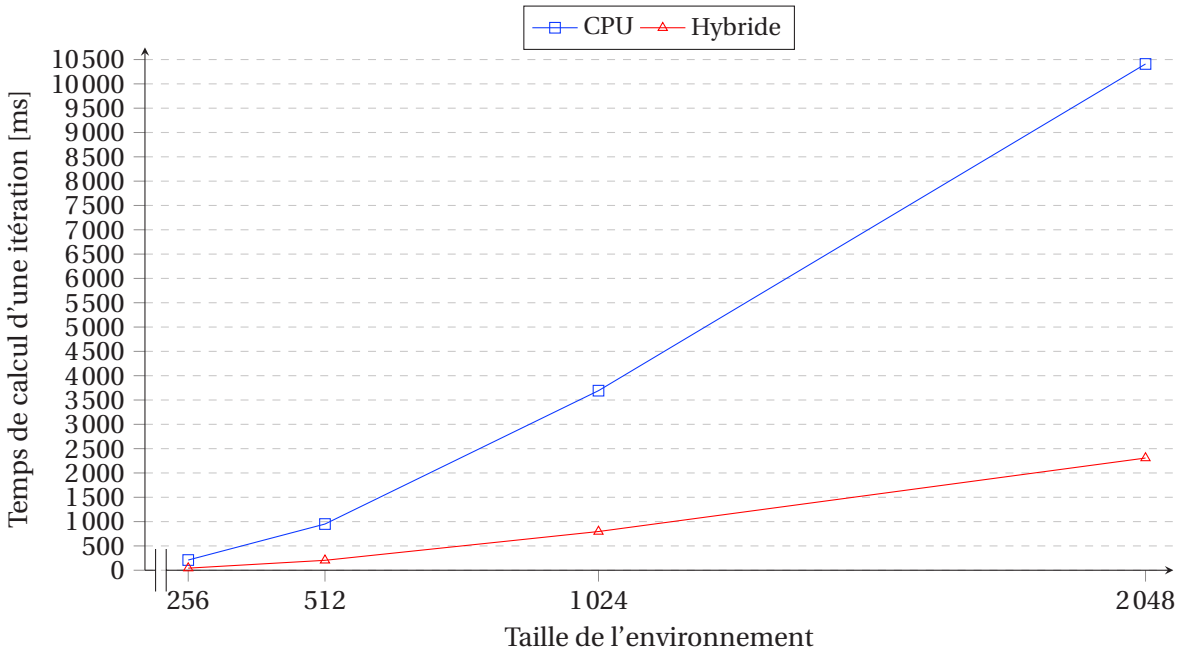


FIGURE 5.8 – Modèle *Segregation*, résultats de performance.

Le graphique issu de la figure 5.8 montre, comme pour le modèle *Game of Life*, que l'intégration du module *Voisinage* augmente les performances du modèle. Cependant, pour le modèle *Segregation*, le gain reste constant quelque soit la taille de l'environnement : environ 5 fois plus rapide que l'exécution séquentielle du modèle.

5.1.3 Le modèle *Fire*

Présentation du modèle

Le modèle *Fire* est inspiré d'une simulation présente dans la librairie de NetLogo qui simule la propagation d'un incendie dans une forêt. Il montre que les chances pour que l'incendie affecte le plus grand nombre d'arbres possibles dépend essentiellement de la densité des arbres dans l'environnement.

Pour notre expérimentation, nous avons défini un modèle dans lequel tous les arbres sont considérés comme des agents placés aléatoirement dans un environnement à deux dimensions carré et discrétisé. Ces arbres peuvent être dans trois états différents : (1) vivant, (2) en feu et (3) mort. Lorsqu'il est en feu, un arbre dégage de la chaleur qui se diffuse dans l'environnement. Cette chaleur, quand elle atteint un certain seuil, peut enflammer d'autres arbres aux alentours. Ce seuil est défini aléatoire pour chaque agent (dans une plage de valeurs). Un arbre voit sa vie (un paramètre propre à chaque arbre) décroitre lorsqu'il brûle. Il meurt lorsque sa vie atteint 0. La figure 5.9 illustre les différentes étapes de la simulation.

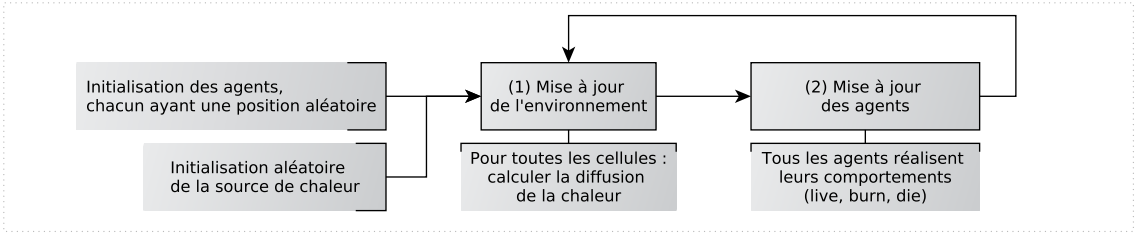


FIGURE 5.9 – Modèle *Fire*, dynamique globale du modèle.

Application du principe de délégation GPU

Le calcul séquentiel nécessitant le plus de ressources est présent dans l'étape (1) et consiste à calculer la diffusion de la chaleur émise par les arbres dans l'environnement. Pour réaliser cette diffusion, il est nécessaire d'effectuer une boucle séquentielle sur toutes les cases de l'environnement. Ainsi, plus l'environnement est grand et plus le temps de calcul de la diffusion est important. Ce calcul étant déjà une dynamique de l'environnement, il est compatible avec le critère du principe de délégation GPU et peut être traduit en module GPU.

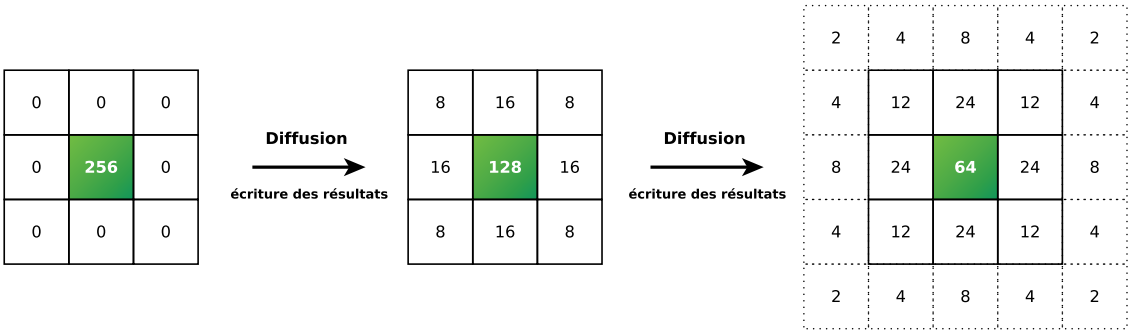


FIGURE 5.10 – Modèle *Fire*, exemple d'un calcul de diffusion sur le GPU.

Dans le cas d'étude sur le modèle MLE (*cf.* chapitre 4), un module GPU dédié au calcul de la diffusion a été créé. Il est possible de le réutiliser pour le modèle *Fire* en adaptant uniquement les structures de données qui vont lui être envoyées. Chaque agent va ainsi déposer une certaine quantité de chaleur (cette valeur dépend de l'état de l'agent) en fonction de sa position dans un tableau à une dimension (`agentHeatArray`, qui correspond à la taille de l'environnement). Un deuxième tableau (`envHeatArray`) contenant les valeurs de chaleur déjà présentes dans l'environnement est également utilisé. Ces deux tableaux sont ensuite envoyés au GPU qui calcule la diffusion sur tout l'environnement. Plus précisément, le *kernel* GPU réalise la somme de la chaleur dans le voisinage de MOORE de chaque cellule, puis ajoute la chaleur déjà présente à cette position et module enfin le résultat global par une variable de diffusion avant d'écrire dans un tableau (`resultArray`) le résultat. La figure 5.10 illustre ce calcul et l'algorithme 5.3 présente une implémentation du *kernel* GPU correspondant. Une fois le calcul réalisé, les agents n'ont plus qu'à percevoir dans l'environnement la valeur de chaleur correspondant à leur position et agir en conséquence.

Algorithme 5.3 : Modèle *Fire*, diffusion de la chaleur dans l'environnement (GPU)

entrées : $width$, $height$, $agentHeatArray[]$, $envHeatArray[]$, $radius$
sortie : $resultArray[]$ (la quantité de chaleur)

```
1 i = blockIdx.x * blockDim.x + threadIdx.x ;
2 j = blockIdx.y * blockDim.y + threadIdx.y ;
3 sumOfHeat = 0 ;
4 si i < width et j < height alors
5 |   pour cellule dans voisinage(agentHeatArray[convert1D(i, j)], radius) faire
6 |   |   sumOfHeat = envHeatArray[convert1D(i, j)] + getNeighborsValue(cellule);
7 |   fin
8 fin
9 resultArray[convert1D(i, j)] = sumOfHeat * heatAdjustment ;
```

Bien que le module GPU *Diffusion* ait été réutilisé ici, il a fallu le modifier légèrement afin de le rendre plus générique. Ayant été créé spécialement pour la diffusion de phéromones, il ne pouvait en l'état accepter d'autres types de données. Nous l'avons donc généralisé en ce sens. De plus, il est désormais capable de prendre en compte des données venant des agents et des données déjà présentes dans l'environnement, cela afin d'augmenter le champ d'application de ce module.

La figure 5.11 et l'algorithme 5.3 illustrent l'intégration du module *Diffusion* dans le modèle *Fire*.

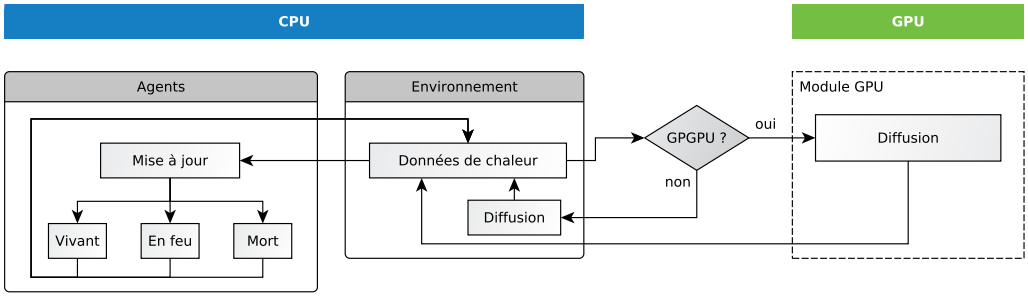


FIGURE 5.11 – Modèle *Fire*, intégration du module GPU.

Algorithme 5.4 : Modèle *Fire*, ordonnancement de la simulation

```
tant que simulationRunning == true faire
    /* Activation des agents                                          */
1   pour agent dans listOfAgents faire
2       updateAgent();
3       live();
4       fillHeatArray(agentHeatArray[]);
5   fin
    /* Activation de l'environnement                                  */
6   updateEnvironment();
7   executeGPUKernel( Diffusion(agentHeatArray[], envHeatArray[]) );
fin
```

Performance du modèle

Pour tester les performances du modèle *Fire*, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fait varier la densité des agents entre 10 % et 100 %. Les figures 5.12 et 5.13 présentent les résultats obtenus.

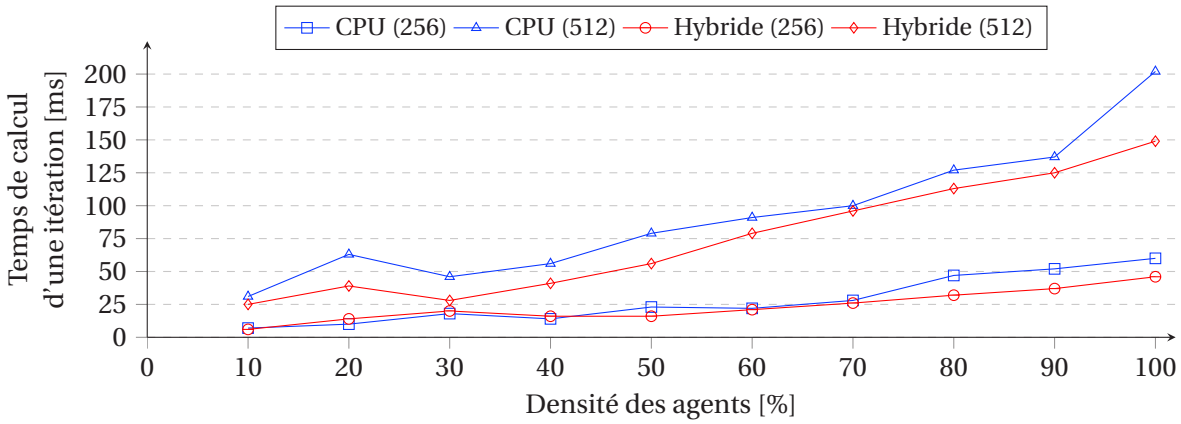


FIGURE 5.12 – Modèle *Fire*, résultats de performance (environnement 256 et 512).

Les graphiques issus des figure 5.12 et 5.13 montrent que, contrairement aux deux précédents modèles, les gains de performance sont moins significatifs. En effet, l'accélération n'est au maximum que de × 2 pour des environnements de grande taille (1 024 et 2 048) et stagne autour de

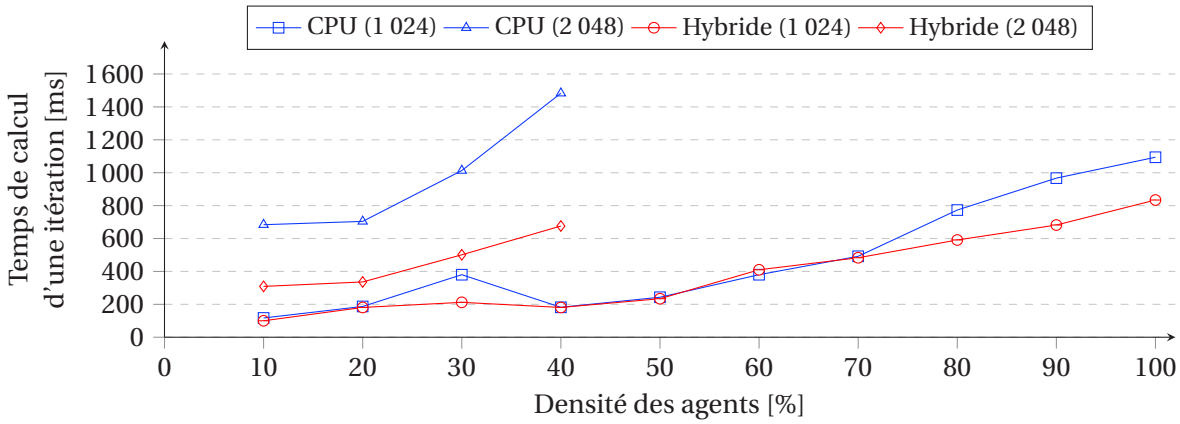


FIGURE 5.13 – Modèle *Fire*, résultats de performance (environnement 1 024 et 2 048).

×1.5 pour des environnements plus petits. La densité des agents n'a pas non plus un réel impact sur les résultats de ces tests.

5.1.4 Le modèle *DLA*

Présentation du modèle

Le modèle *DLA* est également tiré de la librairie NetLogo. Il permet de simuler des phénomènes d'agrégation dans lesquels des particules se déplaçant aléatoirement se rassemblent pour former des structures fractales ressemblant à ce que l'on peut trouver dans la nature : cristaux, champignons, foudre, etc.

Pour notre expérimentation, nous avons défini un modèle dans lequel chaque particule est considérée comme un agent qui se déplace aléatoirement dans un environnement à deux dimensions carré et discrétisé. De manière aléatoire, un des agents en mouvement (coloré en rouge) devient statique et change de couleur (il devient vert). Par la suite, lorsqu'un agent rouge rencontre un agent vert, il s'arrête et change également de couleur. Les autres continuent de se déplacer. La figure 5.14 illustre les différentes étapes de la simulation.

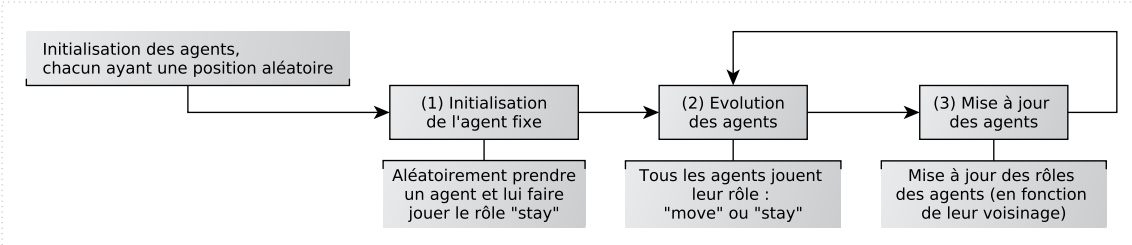


FIGURE 5.14 – Modèle *DLA*, dynamique globale du modèle.

Application du principe de délégation GPU

Les calculs qui requièrent le plus de puissance sont localisés dans les étapes (2) et (3) et consistent en l'établissement d'une liste de voisins présents dans un certain rayon et au parcours de cette liste pour identifier le plus proche. Le temps de calcul nécessaire est donc fortement dépendant du nombre d'agents. Vu que l'action de vérifier si un agent vert se trouve dans le voisinage proche ne modifie pas les états des agents, il est possible de la transformer en une dynamique environnementale réalisée pour tout l'environnement à chaque pas de simulation et calculée par un module GPU.

Cette recherche de voisins a déjà été réalisée pour le modèle *Game of Life*, nous allons donc réutiliser une nouvelle fois le module créé et seulement adapter les données envoyées. Ainsi, chaque agent va déposer une marque de présence en fonction de sa position et du rôle joué dans un tableau à une dimension (`presenceArray`, correspondant à la taille de l'environnement) : 1 pour indiquer sa présence si l'agent est vert et 0 dans les autres cas. Ce tableau est ensuite envoyé au GPU qui réalise la somme des états des cellules présentes dans le voisinage de MOORE pour chaque cellule de l'environnement et écrit le résultat dans un nouveau tableau (`resultArray`). Chaque cellule du tableau contient donc une valeur étant soit égale à 0 (aucun voisin dans le voisinage), soit supérieur à 0 (il y a des voisins dans mon proche voisinage). Une fois ce calcul réalisé, les agents n'ont plus qu'à percevoir dans l'environnement la valeur de présence correspondant à leur position et agir en conséquence.

Tout comme pour le modèle *Segregation*, le module GPU *Voisinage* a été réutilisé tel quel sans modification. La figure 5.15 et l'algorithme 5.5 illustrent l'intégration de ce module dans le modèle *DLA*.

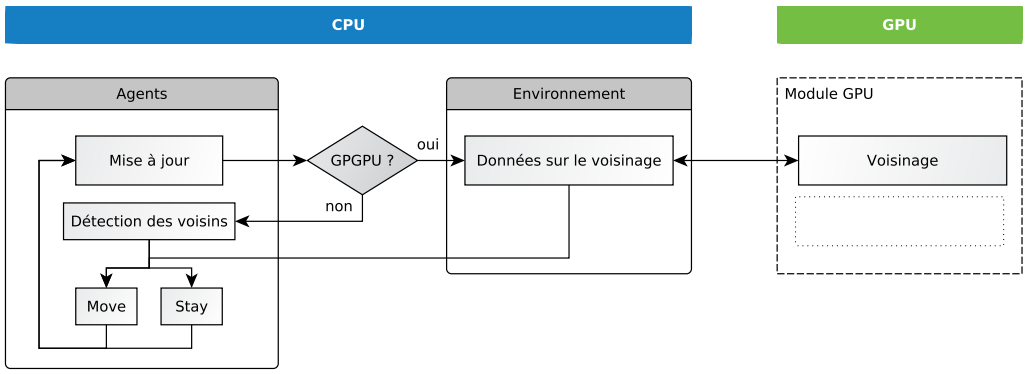


FIGURE 5.15 – Modèle *DLA*, intégration du module.

Algorithme 5.5 : Modèle *DLA*, ordonnancement de la simulation

```
    tant que simulationRunning == true faire
        /* Activation des agents                                    */
1       pour agent dans listOfAgents faire
2           updateAgent();
3           live();
4           fillPresenceArray(presenceArray[]);
5       fin
        /* Activation de l'environnement                            */
6       executeGPUKernel( Voisinage(presenceArray[]) );
    fin
```

Performance du modèle

Pour tester les performances du modèle *DLA*, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fait varier la densité des agents entre 10 % et 90 %. Les figures 5.16 et 5.17 présentent les résultats obtenus.

Les graphiques issus des figure 5.16 et 5.17 montrent que l'intégration du module *Voisinage* augmente les performances du modèle. De plus, ce gain s'accentue d'autant plus que la densité des agents dans l'environnement est importante (jusqu'à 14 fois plus rapide qu'une exécution séquentielle du modèle). Cependant, les performances diminuent lorsqu'on augmente la taille de l'environnement ce qui va à l'encontre de tout ce que l'on a pu observer jusqu'ici.

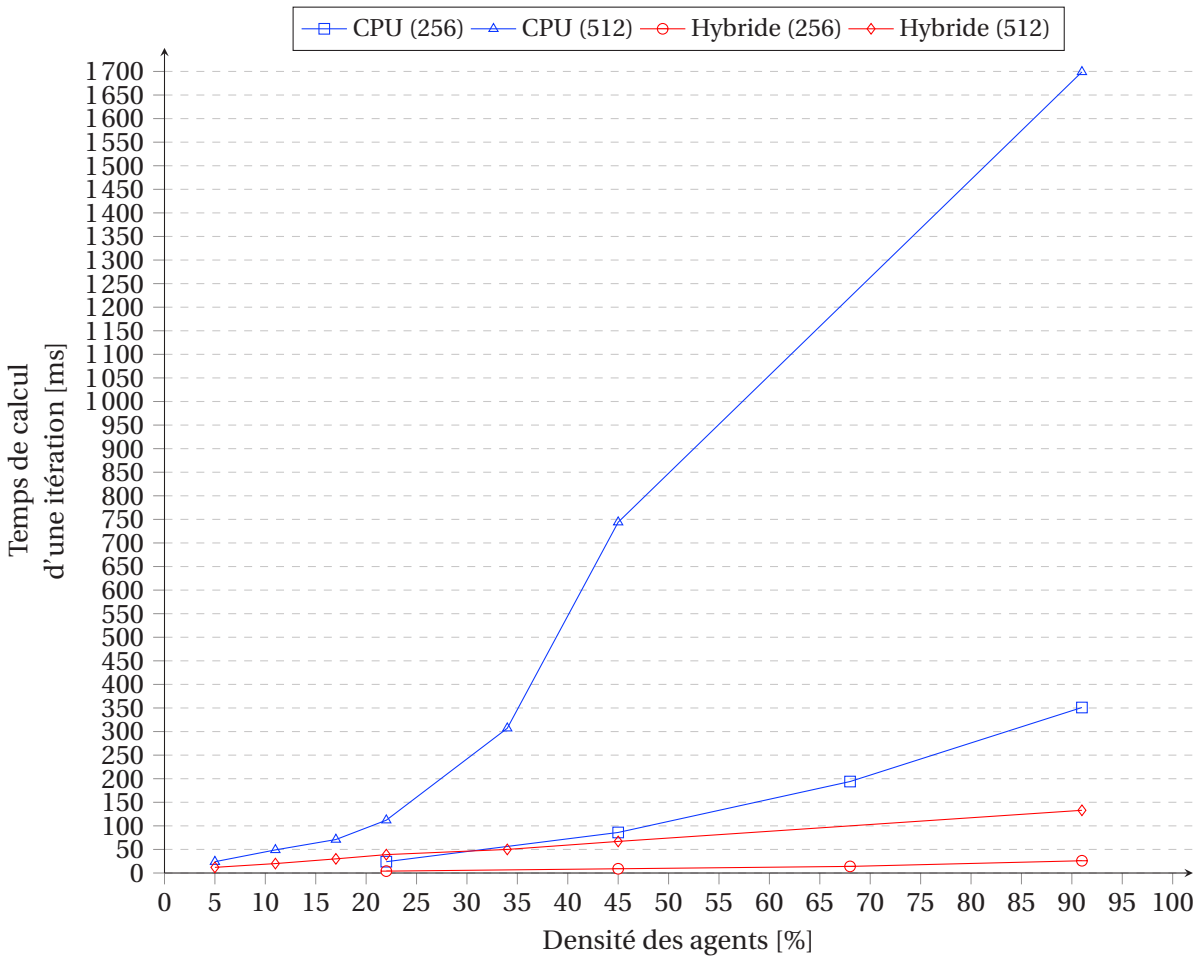


FIGURE 5.16 – Modèle *DLA*, résultats de performance (environnement 256 et 512).

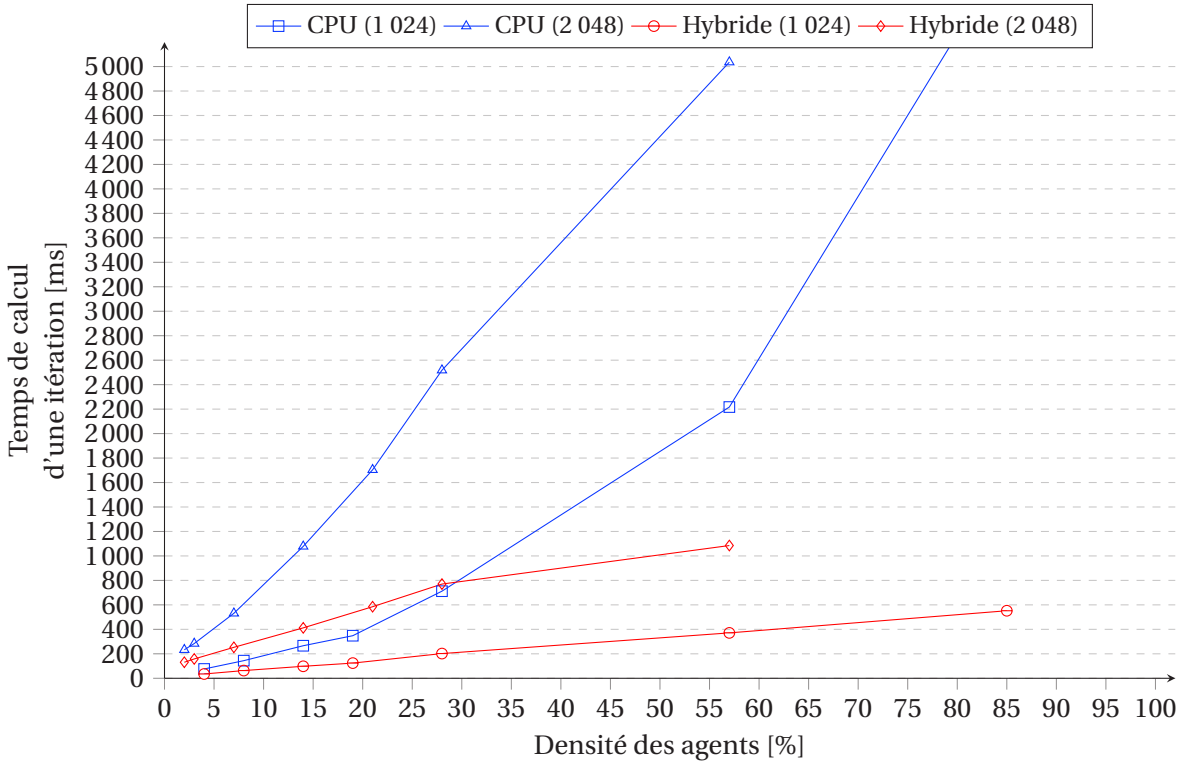


FIGURE 5.17 – Modèle *DLA*, résultats de performance (environnement 1 024 et 2 048).

5.2 Résultats de l'expérimentation du principe de délégation GPU

5.2.1 Du point de vue des performances

À partir des différents résultats de performance établis pour chacun des quatre modèles précédents (résumés par la figure 5.18), on observe que les gains de performance, obtenus grâce à l'application du principe de délégation GPU, varient principalement à cause de deux facteurs :

— la taille de l'environnement : le gain est d'autant plus grand que l'environnement est large ;

— la densité des agents : le gain augmente d'autant plus que la densité est importante.

Cependant, des exceptions ont été observées. Concernant le modèle *Fire*, le gain de performance entre les deux versions du modèle est très limité et reflète probablement une mauvaise utilisation (ou optimisation) du module GPU. Pour le modèle *DLA*, le gain de performance a évolué à l'encontre de ce qui a pu être observé avec les autres modèles en diminuant d'autant plus que l'environnement grandissait. Du fait que seule une partie du modèle bénéficie du GPGPU, nous expliquons cette baisse de performance par une hausse de la consommation des ressources par les agents ce qui a pu ralentir la simulation.

D'une manière générale, on note que chaque application du principe de délégation GPU est unique en termes de performance. En effet, les gains obtenus varient significativement en fonction du modèle considéré : ils peuvent atteindre ×14 mais se situent la plupart du temps autour de ×2 et ×5 ce qui est un gain non négligeable. Cependant, une limite à cette approche est le fait de ne pouvoir prédire à l'avance quels vont être les gains de performance. L'utilisateur peut juger inintéressant d'investir du temps dans la modification de son modèle si cela ne lui apporte pas l'accélération escomptée. L'idée est donc de proposer une solution à ce problème, nous en discutons plus en détails dans les chapitres 6 et 8.

Enfin, si nous comparons les résultats obtenus (en termes de performance) entre le principe de délégation GPU et l'étude menée dans [Perumalla and Aaby, 2008], il apparaît clairement que les implémentations réalisées dans cette dernière sont bien plus efficientes. En effet, elles autorisent des gains de performance bien plus élevés (qui peuvent atteindre jusqu'à ×40) car ces dernières reposent sur une approche tout-sur-GPU. Cette différence par rapport à ce que nous proposons doit cependant être mise en perspective. Comme indiqué dans l'étude, l'objectif de ces implémentations était la recherche de performance pure et elle n'a pu se faire qu'en impactant directement certains aspects conceptuels pourtant identifiés comme essentiels. L'approche de délégation GPU est certes moins performante mais permet de considérer des perspectives liées au génie logiciel telles que l'accessibilité, la réutilisabilité ou la généricité. Cela a déjà été souligné par les expériences précédentes et est une nouvelle fois vérifié ici.

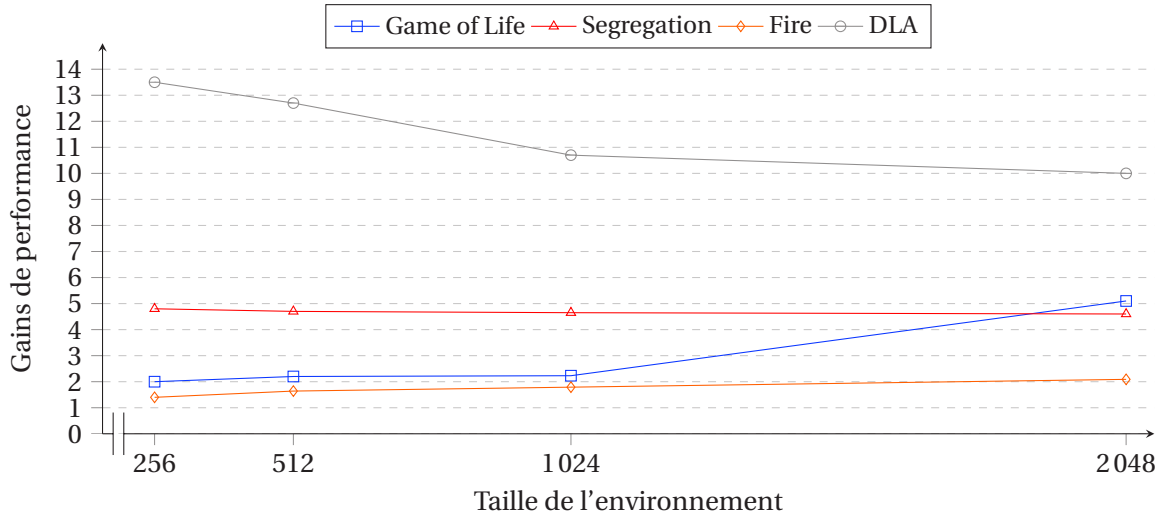


FIGURE 5.18 – Gains de performance entre les versions CPU et hybride des différents modèles.

5.2.2 D'un point de vue conceptuel

L'étude menée par PERUMALLA et AABY [Perumalla and Aaby, 2008] concluait qu'utiliser le GPGPU dans les simulations multi-agent ne pouvait se faire qu'au détriment de la généricité, de l'accessibilité et surtout de la réutilisabilité.

> "Execution is two to three orders of magnitude faster with a GPU [...] but at the cost of decrease in modularity, ease of programmability and reusability. [...] Effective use of data parallel execution, in general, seems to require resolution of modeling and execution challenges."

Le principe de délégation GPU montre que, même si les performances offertes sont moins impressionnantes (voir section précédente), son application remet en cause les conclusions énoncées dans [Perumalla and Aaby, 2008]. En effet, la conception hybride, sur laquelle repose le principe de délégation GPU, lui permet d'être plus générique en lui offrant un champ d'application plus large (prise en compte d'une plus grande variété de modèles). De plus, les modules GPU produits grâce à cette approche sont génériques, indépendants et plus facilement réutilisables. Nous en avons fait l'expérience dans l'expérimentation menée dans ce chapitre avec la réutilisation de deux modules créés pour des modèles complètements différents : réutilisation du module *Diffusion* créé pour le modèle MLE dans le modèle *Fire*, réutilisation du module *Voisinage* créé pour le modèle *Game of Life* dans le modèle DLA. Ce point est important et représente une différence majeure avec les travaux conduits dans [Perumalla and Aaby, 2008].

Si l'on considère l'accessibilité du GPGPU, la délégation GPU permet de considérablement l'améliorer. En effet, grâce à son aspect modulaire, elle permet de produire des *kernels* très simples ne nécessitant que très peu de connaissances dans cette technologie. De plus, en ne transformant seulement qu'une partie spécifique des calculs de l'agent, il est possible de tirer parti de la puissance de calcul du GPU sans changer le modèle de l'agent.

Finalement, en permettant de choisir ce qui va être exécuté sur le GPU et sur le CPU, la stratégie proposée par le principe de délégation GPU permet de surmonter les difficultés rencontrées lors de l'implémentation des comportements des agents sur GPU, notamment dans une approche tout-sur-GPU. Cela a également comme conséquence de faciliter l'utilisation du GPGPU dans le contexte des simulations multi-agents.

5.3 Vers une généralisation de l'approche

Dans un premier temps, nous avons décidé, dans ce chapitre, d'appliquer le principe de délégation GPU sur quatre nouveaux modèles multi-agents que nous avons voulu très différents : *Game of Life*, *Segregation*, *Fire* et *DLA*. Ceci pour juger d'une part du champ d'application de l'approche et d'autre part de ses avantages et limites.

Cette expérimentation a souligné une nouvelle fois des bonnes performances de l'approche ainsi que de sa capacité à prendre en compte les trois critères que nous avons considérés comme essentiels dés le début de ce manuscrit (généricité, accessibilité, réutilisabilité). On peut également noter que le principe n'a pas eu besoin d'être à nouveau modifié pour pouvoir être appliqué sur ces nouveaux modèles.

Ainsi, nous avons considéré comme étape finale à ce travail, de formaliser la méthode implicitement utilisée tout au long de ces différentes expérimentations. En effet, nous avons remarqué de nombreuses redondances dans l'application du principe de délégation GPU. Nous avons notamment suivi à chaque fois le même processus itératif d'application consistant à (1) identifier les calculs les plus gourmands, (2) adapter les structures de données des calculs considérés et enfin (3) implémenter la délégation GPU (créer les modules GPU et les intégrer aux modèles).

La généralisation de l'approche sous la forme d'une méthode de conception est proposée au chapitre suivant. Nous pensons qu'une telle méthode devrait faciliter l'utilisation de notre approche et promouvoir l'utilisation du GPGPU dans les simulations multi-agents.

CHAPITRE 6

DÉFINITION D'UNE MÉTHODE DE CONCEPTION DE MABS BASÉE SUR LA DÉLÉGATION GPU

Sommaire

Dans ce chapitre, nous présentons la formalisation de l'approche developpée tout au long de ce manuscrit sous la forme d'une méthode de conception de simulation multi-agents basée sur le principe de délégation GPU [Hermellin and Michel, 2016a]. Cette méthode, en plus d'instaurer un cadre à l'utilisation de la programmation GPU dans le contexte des simulations multi-agents, se veut différente des autres solutions développées de par le fait qu'elle ne cache pas à l'utilisateur la technologie utilisée (utilisation directe du GPGPU) et qu'elle met en avant un processus de modélisation itératif modulaire prônant la réutilisabilité des outils créés. De plus, nous pensons que définir une telle méthode de conception permettra, en plus de diversifier le champ d'application du principe de délégation GPU, d'accroître la diffusion de l'approche dans la communauté multi-agent et de faciliter son utilisation.

Enfin, en adéquation avec les enjeux énoncés tout au long de ce manuscrit, la méthode de conception proposée poursuit quatre objectifs majeurs :

— simplifier l'utilisation du GPGPU dans le contexte des simulations multi-agents en décrivant le processus de modélisation et d'implémentation à suivre ;

— définir une approche générique pouvant être appliquée sur une grande variété de modèles ;

— promouvoir la réutilisabilité des outils créés ou des modèles transformés (par exemple en réutilisant les modules GPU déjà développés) ;

— aider les utilisateurs potentiels à décider s'ils peuvent bénéficier du GPGPU compte tenu de leurs modèles.

6.1 Énoncé de la méthode

Les différentes expérimentations menées jusqu'ici permettent d'extraire une méthode de conception basée sur le principe de délégation GPU et divisée en 5 phases distinctes (illustrée par la figure 6.1) [Hermellin and Michel, 2016e, Hermellin and Michel, 2016a] :

1. La première étape décompose tous les calculs qui sont utilisés dans le modèle.
2. La deuxième étape identifie, parmi les calculs précédemment listés, ceux répondant aux critères du principe de délégation GPU.
3. La troisième étape consiste à vérifier si les calculs identifiés comme compatibles avec le principe de délégation GPU ont déjà été transformés en dynamiques environnementales et donc s'il existe un module GPU dédié pouvant être réutilisé.
4. La quatrième étape établit la compatibilité des calculs sélectionnés avec l'architecture d'un GPU. L'idée étant de choisir et d'appliquer le principe de délégation uniquement sur les calculs qui vont apporter le plus de gain une fois traduits en modules GPU.
5. Enfin, la cinquième étape consiste à implémenter concrètement le principe de délégation GPU sur les calculs respectant l'ensemble des contraintes précédentes.

6.1.1 Étape 1 : décomposition des calculs du modèle

Cette phase consiste à décomposer tous les calculs qui sont utilisés par le modèle. L'intéret d'une telle décomposition réside dans le fait qu'un certain nombre de calculs présents dans le modèle ne sont pas explicites. Expliciter les calculs nécessaires à la réalisation des différents comportements des agents, en les décomposant en le plus de primitives possibles, va permettre d'augmenter l'efficacité de l'application du principe de délégation GPU sur le modèle considéré. En effet, l'intérêt d'une telle décomposition permet de ne pas avoir un seul "gros" *kernel* contenant tous les calculs GPU [1], mais de capitaliser sur l'aspect modulaire et hybride du principe de délégation GPU avec plusieurs *kernels* très simples mais qui tirent partie de la décomposition des calculs. Chaque calcul (compatible) peut alors prétendre à avoir son propre *kernel*. Ainsi, la délégation GPU sera d'autant plus efficace qu'il va être possible de décomposer le modèle en calculs "élémentaires". D'ailleurs, dans le contexte de l'utilisation du GPGPU pour les simulations multi-agents, cette décomposition des actions a également été identifiée comme cruciale dans [Coakley et al., 2016]. En introduisant une nouvelle division des actions des agents qui limite les accès concurrents aux données, [Coakley et al., 2016] montre qu'il a été possible d'augmenter significativement les performances globales des modèles qui utilisent le GPGPU.

6.1.2 Étape 2 : identification des calculs compatibles

L'identification des calculs compatibles est une étape essentielle car elle consiste à vérifier quels sont les calculs qui répondent aux critères du principe de délégation GPU et qui pourront ainsi bénéficier du GPGPU. Dans le cas où aucune partie du modèle n'est conforme aux critères d'application du principe de délégation GPU, il est inutile de continuer à suivre cette méthode car, dans un tel cas, les gains apportés par le GPGPU pourraient être insignifiants voire même négatifs (*cf.* [Laville et al., 2012]). Les conditions permettant d'identifier un calcul comme compatible

1. Avoir un seul "gros" *kernel* limiterait les avantages de notre approche car son implémentation ressemblerait à celle d'une approche tout-sur-GPU avec les contraintes qu'on lui connaît (*cf.* chapitres 2 et 3).

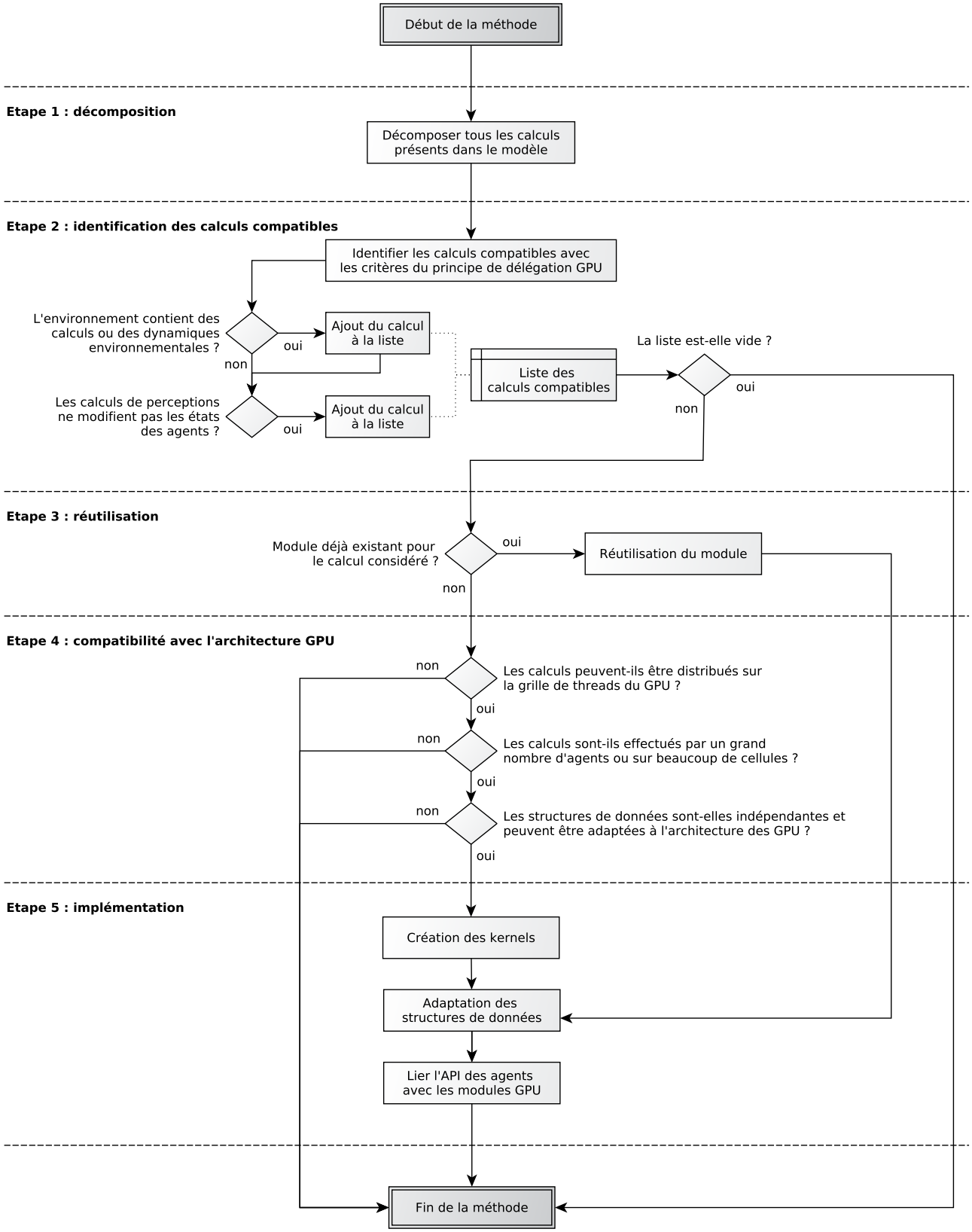


FIGURE 6.1 – Schéma récapitulatif de la méthode proposée.

sont différentes selon si le calcul est présent dans l'environnement ou dans le comportement des agents.

Pour l'environnement

Si l'environnement [2] n'est pas statique et qu'il contient des dynamiques, ces dernières doivent (1) s'appliquer sur tout l'environnement et (2) avoir un impact global. En effet, l'impact des dynamiques est un paramètre important. Prenons l'exemple d'une dynamique environnementale qui consiste à faire apparaitre de manière stochastique dans l'environnement, à chaque x pas de temps, une source de nourriture à une position donnée. Cette dynamique va bien s'appliquer sur tout l'environnement mais ne va avoir qu'un impact très localisé. Dans ce cas, traduire cette dynamique dans un module GPU n'est pas justifié car les gains espérés seront nuls (voir négatifs). Dans le cas contraire, si la dynamique possède un impact global et répond à toutes les contraintes énoncées, sa compatibilité est établie et sa traduction en module GPU est alors possible et pertinente.

Pour les agents

Si des calculs de perceptions ne modifient pas les états des agents, ils peuvent être considérés comme compatibles et transformés en dynamiques environnementales pour être implémentés dans des modules GPU indépendants. L'idée est de transformer un calcul de perception réalisé de manière locale en une dynamique environnementale s'appliquant globalement dans l'environnement.

6.1.3 Étape 3 : réutilisation des modules GPU existants

Un des objectifs de la méthode est de promouvoir la réutilisabilité des modules GPU créés. En effet, comme vu précédemment, le principe de délégation GPU sur lequel se base la méthode, se distingue par la création de modules GPU génériques pouvant être réutilisés dans des contextes différents de ceux pour lesquels ils ont été créés. Ainsi, il convient de vérifier que les calculs identifiés précédemment n'aient pas déjà un module GPU dédié. Si c'est le cas, il est possible de le réutiliser et de passer directement à l'étape 5 afin d'adapter les structures de données au module existant.

6.1.4 Étape 4 : évaluation de l'adaptation des calculs sur l'architecture des GPU

Avant d'appliquer concrètement la délégation GPU sur les calculs identifiés comme compatibles, il est nécessaire d'évaluer si ces calculs vont pouvoir s'adapter à l'architecture massivement parallèle des GPU. En effet, la compatibilité d'un calcul avec les critères du principe de délégation GPU n'implique pas forcement une amélioration des performances une fois ce principe appliqué. Ainsi, comme nous avons pu le constater précédemment, nos différentes expérimentations ont souligné de grosses variations autour des gains de performance obtenus avec même, par moment, des résultats que l'on peut qualifier de médiocres (*cf.* chapitre 5, expérimentation du modèle *Fire*). Dans ces conditions, il est nécessaire d'avoir une estimation même empirique des gains escomptés afin d'évaluer la pertinence d'appliquer la délégation GPU sur les calculs identifiés.

Cette phase de vérification peut être réalisée en répondant à trois questions :

— *Est-ce que les calculs identifiés vont pouvoir être distribués sur le GPU ?*
Comme vu dans le chapitre 2, ces calculs doivent être indépendants et simples et ne pas contenir de trop nombreuses structures conditionnelles, ces dernières pouvant causer des problèmes de divergence des *threads* (*cf.* [Sanders and Kandrot, 2011] en page 78) ou des

2. Il est important de bien distinguer qu'un environnement peut être implémenté par un agent (*e.g.* comme c'est le cas dans MaDKit et TurtleKit), mais cela n'a rien à voir avec les dynamiques qu'il gère : il y a une différence entre la modélisation du modèle multi-agent et son implémentation dans un framework.

ralentissements lors de l'exécution. Des calculs contenant des boucles itératives s'adaptent mieux aux architectures parallèles.

— *Est-ce que les calculs identifiés sont réalisés de manière globale (par un grand nombre d'agents ou appliqués sur un grand nombre de cellules de l'environnement) ?*
En raison des coûts très élevés (en temps) qu'occasionnent les transferts de données entre GPU et CPU, des calculs rarement utilisés vont accroitre le temps de calcul général du modèle.

— *Est-ce que les structures de données associées aux calculs identifiés vont pouvoir s'adapter aux contraintes d'une utilisation sur GPU ?*
En l'occurrence, il s'agit principalement de s'assurer que les données peuvent être distribuées sur le GPU (en tenant compte des spécificités des différents types de mémoire) et traitées avec un degré élevé d'indépendance afin d'avoir une implémentation GPU efficace. Pour plus d'informations sur cet aspect, [Bourgoin et al., 2014] peut être consulté.

Pour donner un exemple, en se basant sur nos différents cas d'étude, il est possible d'utiliser, dans le cas d'environnements discrétisés, des structures de données (des tableaux par exemple) de la taille de l'environnement. Ce qui les rend très adaptées à l'architecture du GPU. En effet, chaque cellule de l'environnement aura ainsi un *thread* qui lui sera entièrement dédié. La figure 6.2 illustre comment ces structures de données peuvent être utilisées lors d'une application du principe de délégation GPU sur un modèle possédant un environnement discrétisé.

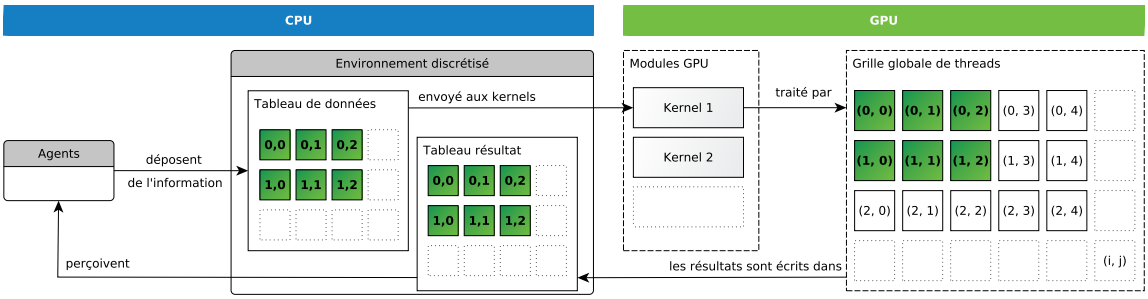


FIGURE 6.2 – Exemple d'application du principe de délégation GPU.

6.1.5 Étape 5 : implémentation du principe de délégation GPU

Les calculs étant identifiés et évalués, il convient d'appliquer sur chacun d'entre eux la délégation GPU. Cette application est divisée en trois étapes :

1. Création des modules GPU ;
2. Adaptation des structures de données ;
3. Création des interfaces entre la partie CPU du modèle et les modules GPU.

L'application de la délégation GPU commence par la création des modules GPU contenant les *kernels* de calculs (la traduction des calculs séquentiels en version parallèle). Grâce à la décomposition effectuée précédemment, ces *kernels* ne nécessitent que peu de connaissances en GPGPU et tiennent en seulement quelques lignes de code. En général, ces modules sont divisés en quatre parties (voir l'algorithme 6.1 associé) :

— (1) Initialisation du *thread* qui va effectuer le calcul ;

— (2) Test conditionnel sur la position du *thread* dans la grille globale du GPU permettant l'accès aux données correspondantes ;

— (3) Réalisation du calcul ;

— (4) Écriture des résultats.

Algorithme 6.1 : Exemple de structuration d'un *kernel* de calcul GPU

```
   /* (1) Initialisation du thread                                          */
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  ... ;
   /* (2) Test conditionnel sur la taille de la grille de threads[3]        */
4  si i < width et j < height alors
   |  /* (3) Réalisation des calculs                                        */
5  |  ... ;
6  fin
   /* (4) Écriture des résultats                                            */
7  return ... ;
```

Ensuite, les structures de données doivent être adaptées aux modules GPU créés. Cette adaptation est basée sur la nature des calculs effectués et sur le type d'environnement utilisé (des tableaux correspondant à la discrétisation de l'environnement sont le plus souvent utilisés).

Enfin, ces nouveaux éléments doivent être intégrés et reliés à la partie CPU du modèle. Pour ce faire, de nouvelles versions des fonctions de perceptions qui encapsulent l'utilisation du GPU doivent être créées afin de permettre aux agents et à l'environnement de recueillir et d'utiliser les données calculées par les modules GPU. La plupart du temps, ces fonctions sont de simples primitives d'accès aux données.

6.2 Validation de l'approche

L'édition de cette méthode a représenté une étape importante dans la formalisation du travail mené. Il était cependant nécessaire de l'expérimenter. Nous avons donc choisi de l'appliquer sur deux nouveaux modèles multi-agents : le modèle *Heatbugs* et Proie-prédateur. Plus précisément, l'application de la méthode sur ces deux modèles a été réalisée de manière à faire apparaître explicitement les 5 étapes du processus afin de pouvoir définir quels sont les avantages et limites d'une telle approche.

À la suite de chacune des expérimentations, nous avons conduit un test de performance entre les deux versions du modèle (CPU et hybride) en faisant varier la taille de l'environnement ainsi que la densité des agents. Chaque simulation est exécutée plusieurs fois sur une période de 10 000 pas de temps. On calcule ensuite le temps moyen d'exécution pour une itération, nous donnant ainsi une valeur permettant de comparer les deux versions du modèle. Pour ces tests, nous avons réutilisé la même configuration matérielle et logicielle que précédemment. Les codes sources, ressources et vidéos de ces modèles sont aussi disponibles en ligne [4].

6.2.1 Le modèle *Heatbugs*

Présentation du modèle

Le modèle *Heatbugs*[5] est un modèle multi-agent bio-inspiré dans lequel des agents tentent de maintenir une température optimale autour d'eux. Plus précisément, ces agents se déplacent dans un environnement à deux dimensions discrétisé en cellules (une cellule ne peut être occupée que par un seul agent) et émettent de la chaleur qui se diffuse dans l'environnement. Chaque agent possède une température idéale qu'il cherche à atteindre. Ainsi, plus la température de la case sur laquelle il se trouve va être différente de sa température idéale et plus l'agent va être mécontent. Lorsque un agent n'est pas satisfait (sa case est trop froide ou trop chaude), il se déplace aléatoirement pour trouver une place qui correspondra mieux à ces attentes.

3. Plus de détails sur ce test conditionnel peuvent être trouvés dans [Sanders and Kandrot, 2011] en page 42.
4. `http://www.lirmm.fr/~hermellin/` section "Doctorat".
5. `http://ccl.northwestern.edu/netlogo/models/Heatbugs`

Application de la méthode

Étape 1.

On décompose les calculs présents dans le modèle (illustrée par la figure 6.3).

— Dans l'environnement : diffusion de la chaleur (C1).

— Pour les agents : déplacement (C2), émission de chaleur (C3), calcul des différences de température (C4) et calcul du bonheur (C5).

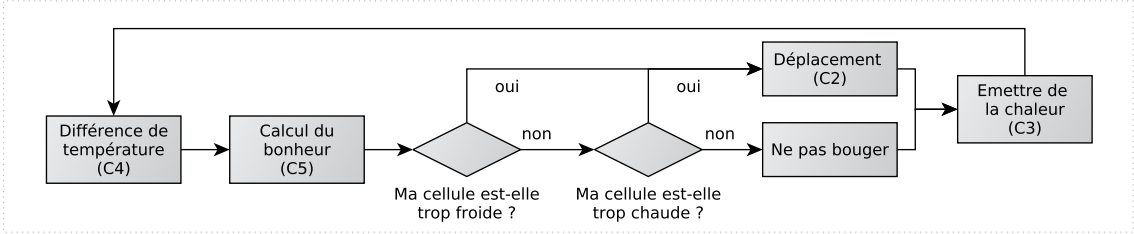


FIGURE 6.3 – Modèle *Heatbugs*, décomposition des calculs.

Étape 2.

Parmi les 5 calculs présents dans le modèle, on identifie maintenant ceux qui sont compatibles avec les critères du principe de délégation GPU. C1 est une dynamique globale de l'environnement (une diffusion d'information, ici la chaleur) qui ne contient aucun processus décisionnel. Cette dynamique s'applique sur tout l'environnement et possède un impact global : elle est donc compatible avec les critères du principe. C4 consiste à percevoir la température de la case sur laquelle se trouve l'agent et calculer la différence entre cette température et la température idéale de l'agent. Ce calcul ne modifie pas les états de l'agent, il est donc compatible avec le principe et peut être transformé en une dynamique environnementale. Cependant, les calculs C2, C3 et C5 modifient les états des agents, on ne les considère donc pas pour la suite.

Étape 3.

Seuls C1 et C4 sont identifiés comme compatibles avec le principe de délégation GPU. Le calcul C4 n'a pour l'instant jamais été implémenté dans un module GPU à la différence de C1 qui consiste en une diffusion (de chaleur) et qui a déjà été effectuée plusieurs fois (dans le cas d'étude sur le modèle MLE, *cf.* chapitre 4, et le modèle *Fire*, *cf.* chapitre 5). On réutilise donc le module GPU correspondant à ce calcul.

Étape 4.

C1 étant un calcul compatible déjà utilisé dans plusieurs modèles, il n'est pas nécessaire d'évaluer son adaptation sur l'architecture des GPU. Cependant, C4 étant réalisé pour la première fois, il est nécessaire d'effectuer cette phase de vérification. Ainsi, C4 consiste en la réalisation d'une différence, il peut être facilement distribué sur la grille de *threads* du GPU (chaque *thread* va réaliser cette opération sur les données correspondant à sa position dans la grille). De plus, tous les agents calculent cette différence à chaque pas de simulation. Transformer ce calcul est donc pertinent car réalisé de très nombreuses fois. Enfin, les données utilisées pour ce calcul peuvent être stockées dans des tableaux s'adaptant parfaitement aux contraintes des architectures GPU.

Étape 5.

Suite aux 4 étapes précédentes, on peut appliquer le principe de délégation GPU sur C1 et C4 qui remplissent toutes les exigences requises.

Pour C1, le module GPU *Diffusion* est réutilisé, seules les données envoyées nécessitent d'être adaptées. Dans ce modèle, les données relatives à la chaleur sont stockées dans un tableau à une

dimension (`heatArray`, correspondant à la taille de l'environnement) dans lequel les agents déposent, en fonction de leur position, la quantité de chaleur qu'ils émettent à chaque pas de simulation. Ce tableau est ensuite envoyé au GPU qui calcule simultanément la diffusion de la chaleur pour toutes les cellules de l'environnement. Plus précisément, pour chaque cellule, un *thread* calcule la somme de la chaleur des cellules voisines qui est ensuite modulée par une variable de diffusion.

Après l'exécution du module GPU *Diffusion*, la chaleur de chaque cellule est utilisée pour calculer la différence entre cette valeur et la température idéale de l'agent qui se trouve sur la cellule. Cette différence calculée au départ dans le comportement des agents est maintenant réalisée dans une dynamique environnementale pour tout l'environnement. Plus précisément, chaque agent dépose, en fonction de sa position, sa température idéale dans un tableau à une dimension (`idealTemperatureArray`, correspondant à la taille de l'environnement). Les deux tableaux sont envoyés au GPU qui effectue le calcul de la différence. L'algorithme 6.2 présente l'implémentation du calcul dans un *kernel*. Les agents utilisent ensuite ces résultats pour calculer leur valeur de bonheur, à partir d'une perception pré-calculée par un *kernel* GPU.

Algorithme 6.2 : Modèle *Heatbugs*, calcul de la différentielle des températures (GPU)

entrées : $width$, $height$, $heatArray[]$, $idealTemperatureArray[]$
sortie : $resultArray[]$ (la différence de température)

1 $i = blockIdx.x * blockDim.x + threadIdx.x$;
2 $j = blockIdx.y * blockDim.y + threadIdx.y$;
3 $diff = 0$;
4 **si** $i < width$ ***et*** $j < height$ **alors**
5 $\quad diff = heatArray[\text{convert1D}(i,j)] - idealTemperatureArray[\text{convert1D}(i,j)]$;
6 **fin**
7 $resultDiffArray[\text{convert1D}(i,j)] = diff$;

Le *kernel*, présenté dans l'algorithme 6.2, a mené à la création d'un nouveau module GPU nommé *Différence*. Il faut noter qu'enchaîner l'exécution de ces deux modules permet d'éviter des transferts de données coûteux et inutiles qui occasionneraient des baisses importantes de performance lors de l'exécution de la simulation. Dans notre cas, le module *Différence* réutilise les données tout juste calculées par le module *Diffusion* et qui sont toujours présentes dans la mémoire du GPU. L'algorithme 6.3 illustre l'intégration de ces deux modules dans le modèle *Heatbugs*.

Algorithme 6.3 : Modèle *Heatbugs*, ordonnancement de la simulation

tant que $simulationRunning == true$ **faire**
$\quad$ `/* Activation des agents */`
1 $\quad$ **pour** $agent$ ***dans*** $listOfAgents$ **faire**
2 $\quad\quad$ computeHapiness($resultDiffArray[]$);
3 $\quad\quad$ live();
4 $\quad\quad$ fillHeatArray($heatArray[]$);
5 $\quad$ **fin**
$\quad$ `/* Activation de l'environnement */`
6 $\quad$ executeGPUKernel(Diffusion($heatArray[]$));
7 $\quad$ executeGPUKernel(Différence($heatArray[]$, $idealTemperatureArray[]$));
fin

Performance du modèle

Pour tester les performances du modèle *Heatbugs*, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fixé la densité des agents à 40 %. La figure 6.4 présente les résultats obtenus.

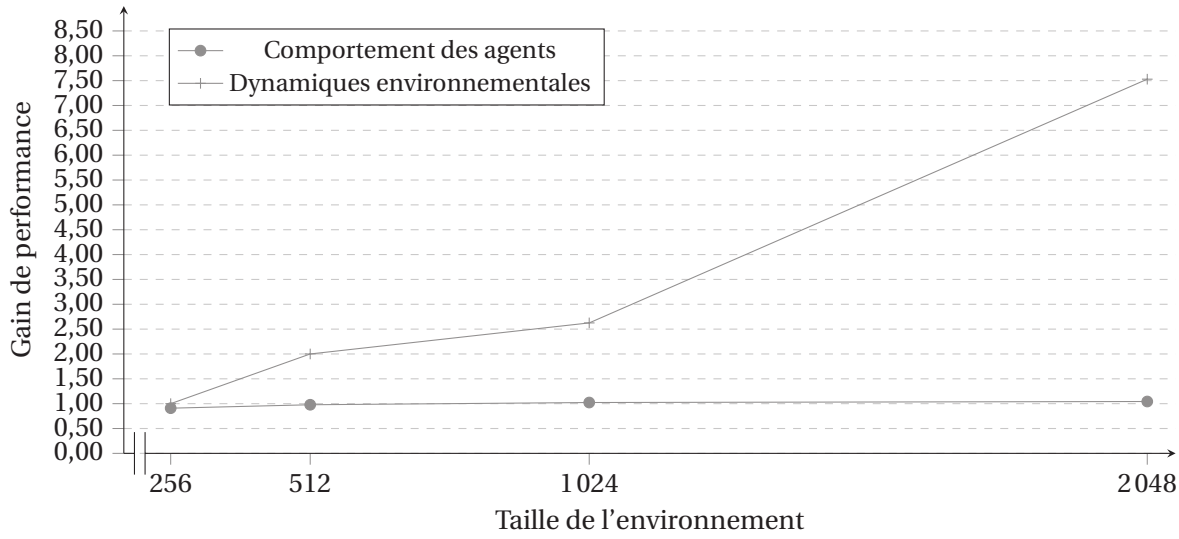


FIGURE 6.4 – Modèle *Heatbugs*, gains d'exécution entre la version CPU et hybride.

Des graphiques issus de la figure 6.4, on observe que le gain de performance obtenu pour le temps d'exécution de l'environnement est d'autant plus important que l'environnement est grand. Cependant, les temps d'exécution correspondant au calcul des agents reste faible (environ 5 %). Ces résultats s'expliquent de la façon suivante : la dynamique de diffusion (C1) est appliquée à toutes les cellules et profite donc grandement de la puissance du GPU alors que seule une petite partie des calculs effectués par les agents a été transformée (C4). Ainsi, la partie séquentielle des comportements des agents brident les performances globales du modèle.

6.2.2 Le modèle Proie-prédateur

Présentation du modèle

Proposé indépendamment par Alfred James LOTKA en 1925 [Lotka, 1925] et Vito VOLTERRA en 1926 [Volterra, 1926], le modèle Proie-prédateur est aussi connu sous le nom d'équations de Lotka-Volterra. Il décrit, sous la forme d'équations, la dynamique de systèmes biologiques où deux entités interagissent : un prédateur et sa proie. Le modèle multi-agent qui s'en inspire modélise les entités de ce système sous la forme d'agents qui évoluent dans un environnement. Dans notre modèle, cet environnement est discrétisé en une grille de cellules en deux dimensions. Les prédateurs, tout comme les proies, y sont placés aléatoirement. Tous les prédateurs possèdent un champ de vision (*Field Of View*, FOV) de dix cellules autour d'eux. Si une proie se trouve dans leur FOV, ils la ciblent et se dirigent vers elle, sinon ils se déplacent aléatoirement. Les proies se déplacent aussi de manière aléatoire. Lorsqu'un prédateur se trouve dans le FOV d'une proie (qui est de six cellules autour d'elle), cette dernière tente de s'échapper en s'enfuyant dans la direction opposée. Une proie meurt quand elle est ciblée par un prédateur et quand ce dernier se trouve sur la même case.

Application de la méthode

Étape 1.

Comme précédemment, on commence par décomposer les différents calculs présents. Cette décomposition est décrite par la figure 6.5.

— L'environnement est statique et ne possède aucune dynamique environnementale.

— Pour les agents : Les prédateurs (C1) déterminent la direction vers la proie la plus proche et (C2) se déplacent. Les proies (C3) calculent leur direction de fuite face aux prédateurs et (C4) se déplacent.

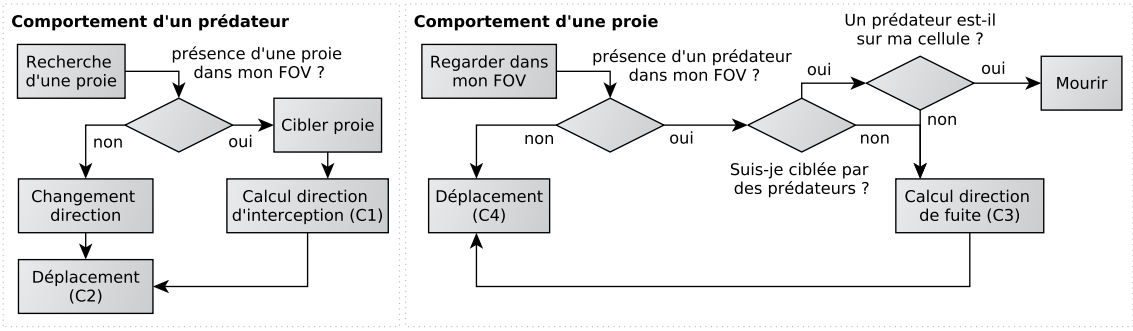


FIGURE 6.5 – Modèle Proie-prédateur, décomposition des calculs.

Étape 2.

Parmi ces 4 calculs présents dans le modèle, on identifie maintenant ceux qui sont compatibles avec les critères du principe de délégation GPU. C2 et C4 consistent en des déplacements modifiant de ce fait les états des agents (leur position). Ainsi, on ne les considère pas pour la suite car ces derniers ne respectent pas les critères d'éligibilité du principe de délégation GPU. C1 et C3, quant à eux, consistent à calculer la direction de fuite des proies face aux prédateurs (C1) et la direction vers la proie ciblée pour les prédateurs (C3). Ces deux calculs, à l'origine réalisés dans les comportements des agents (création d'une liste des voisins, parcours de la liste, calcul des directions), peuvent être pensés comme une perception pré-calculée par une dynamique environnementale. Dans ce cas, cette dynamique calculera pour chaque cellule de l'environnement les directions vers les proies et les prédateurs les plus proches mais aussi les directions opposées. Les agents n'auront plus qu'à percevoir en fonction de leur type la direction qui les intéresse et agir en conséquence. La transformation de ces calculs en dynamiques environnementales traitées par un module GPU est possible car ces modules ne modifient pas les états des agents.

Étape 3.

C1 et C3 sont identifiés comme compatibles avec le principe de délégation GPU. Pour ces calculs, il est possible de réutiliser le module *GPU field perception* créé pour le modèle MLE (*cf.* chapitre 4). En effet, ce module calcule pour chaque cellule la direction vers des cellules voisines possédant la plus grande/faible quantité d'une variable donnée. Ici, cette variable correspond à la présence ou non d'agents dans le voisinage et peut donc être considérée comme un gradient de présence du voisinage. Ce gradient de présence calcule ainsi les directions maximales et minimales vers les agents les plus proches. Il est cependant essentiel d'adapter correctement les structures de données envoyées au module GPU.

Étape 4.

Du fait que l'on réutilise un module déjà existant et qu'aucun nouveau calcul n'a été identifié comme compatible, il est possible de passer directement à l'étape 5.

Étape 5.

Suite aux 4 étapes précédentes, on peut appliquer le principe de délégation GPU sur C1 et C3 qui remplissent toutes les exigences requises.

Pour implémenter le module dédié à C3, un tableau à une dimension est utilisé (`preyMark`) contenant une marque de présence déposée par chaque proie à chaque pas de temps de la simulation. Plus le nombre de proies va être important à un endroit précis de l'environnement et plus la marque de présence sera importante. Ce tableau est ensuite envoyé au module *GPU field perception* qui va tester le voisinage proche de chaque cellule de l'environnement et déterminer la direction menant à la marque de présence des proies la plus forte. Plus précisément, cette vérification va consister à prendre une cellule voisine de la cellule considérée, récupérer sa valeur de

présence, tester si cette valeur récupérée est plus grande que la valeur en mémoire. Ensuite, l'index de la case ayant la plus forte marque de présence est utilisé pour calculer l'angle vers cette case [6]. Le résultat est ensuite écrit dans un nouveau tableau (`preyMaxDirection`). L'algorithme 6.4 présente l'implémentation du *kernel* de calcul correspondant. Ainsi, les prédateurs n'ont plus qu'à percevoir, en fonction de leur position, dans le tableau `preyMaxDirection` la valeur d'orientation correspondant à la direction vers la proie la plus proche.

Algorithme 6.4 : Modèle Proie-prédateur, perception des gradients de direction en GPU

```
   entrées : width, height, preyMark[]
   sortie  : preyMaxDirection[] (le tableau des directions vers les proies)
1  i = blockIdx.x * blockDim.x + threadIdx.x ;
2  j = blockIdx.y * blockDim.y + threadIdx.y ;
3  float max = 0 ;
4  int maxIndex = 0 ;
5  si i < width et j < height alors
6      pour int u = 1 ; u < 8 ; u++ faire
7          float current = getNeighborsValues(u, preyMark[convert1D(i, j)]);
8          si max < current alors
9              max = current;
10             maxIndex = u;
11         fin
12     fin
13     preyMaxDirection[convert1D(i, j)] = maxIndex * 45 ;
14 fin
```

Pour C1, le calcul est similaire, ce qui permet d'utiliser une nouvelle instance de ce module. Un tableau contenant les marques de présence des prédateurs est donc envoyé au module GPU et un tableau résultat contenant les directions vers les prédateurs est retourné. Cependant, les proies ne cherchent pas la direction vers les prédateurs mais la direction opposée. Ainsi, et pour ne pas modifier le module utilisée précédemment, chaque direction pointant vers les prédateurs se voit ajouter 180 degrés. De ce fait, ces directions représentent maintenant des directions de fuite face aux prédateurs. À la suite de ce calcul, les proies n'ont plus qu'à percevoir, en fonction de leur position, dans le tableau résultat la valeur d'orientation correspondant à la direction de fuite face aux prédateurs les plus proches. L'algorithme 6.5 illustre l'intégration des deux instances du module *GPU field perception* dans le modèle Proie-prédateur.

Algorithme 6.5 : Modèle Proie-prédateur, ordonnancement de la simulation

```
   tant que simulationRunning == true faire
       /* Activation des agents                                          */
1      pour agent dans listOfAgents faire
2          live();
3          si agent == prey alors
4              fillMarkArray(preyMark[]);
5          fin
6          si agent == predator alors
7              fillMarkArray(predatorMark[]);
8          fin
9      fin
       /* Activation de l'environnement                                  */
10     executeGPUKernel( GPUFieldPerception(preyMark[]) );
11     executeGPUKernel( GPUFieldPerception(predatorMark[]) );
   fin
```

6. Nous rappelons que, dans TurtleKit, l'orientation d'un agent est un angle en degré dont la valeur est comprise entre 0 et 360 et se fait en fonction d'un repère fixé dans l'environnement.

Performance du modèle

Pour tester les performances du modèle Proie-prédateur, nous avons simulé successivement les versions CPU et hybride pour des environnements de taille 256, 512, 1 024 et 2 048. Pour chacune de ces tailles d'environnement, nous avons fixé la densité des agents à 20 % puis à 40 %. La répartition entre proies et prédateurs est la suivante : 90 % de proies et 10 % de prédateurs. Les figures 6.6 et 6.7 présentent les résultats obtenus.

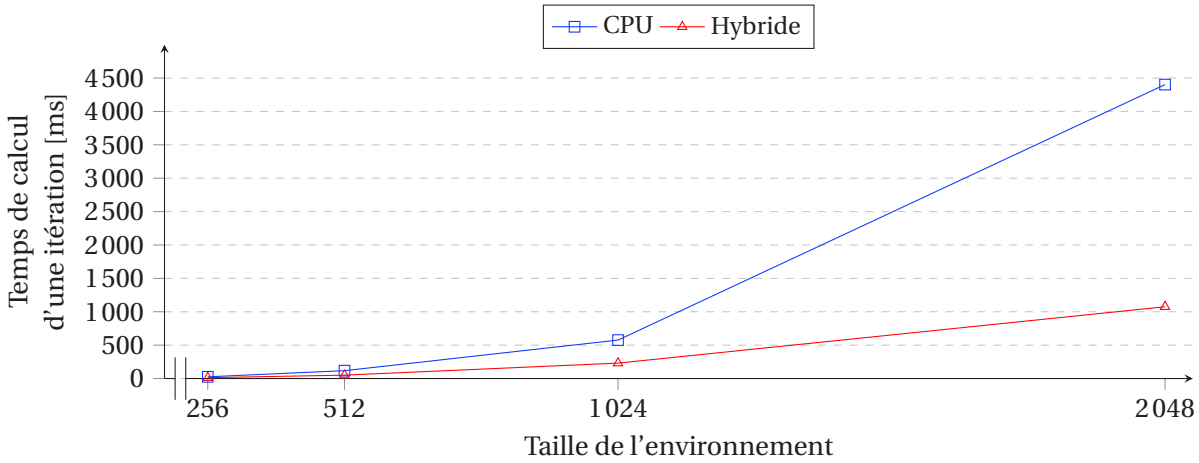


FIGURE 6.6 – Modèle Proie-prédateur, résultats de performance (densité des agents : 20 %).

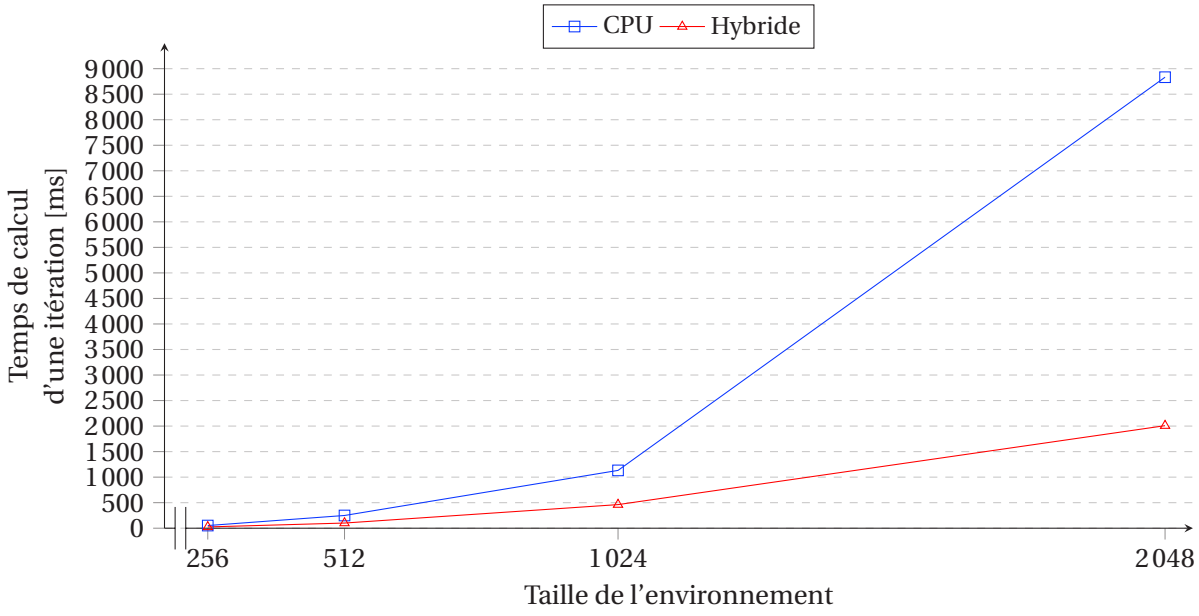


FIGURE 6.7 – Modèle Proie-prédateur, résultats de performance (densité des agents : 40 %).

Des graphiques issus des figures 6.6 et 6.7, on observe une nouvelle fois que la différence de performance entre les deux versions du modèle est d'autant plus importante que l'environnement est grand. De plus, on note que l'augmentation de la densité des agents accélère cette différence.

6.3 Avantages et limites de la méthode proposée

6.3.1 Du point de vue des performances

Tout comme pour les expérimentions précédentes, l'application de la méthode sur les modèles *Heatbugs* et Proie-prédateur permet d'obtenir une amélioration des temps d'exécution comparé à

une exécution séquentielle des modèles, comme l'illustre la figure 6.8. La recherche de performance n'étant pas l'unique objectif de cette méthode, il faut noter que l'accélération obtenue (jusqu'à 4 fois plus rapide) est tout de même significative et ouvre des possibilités importantes vis-à-vis de la scalabilité des modèles simulés.

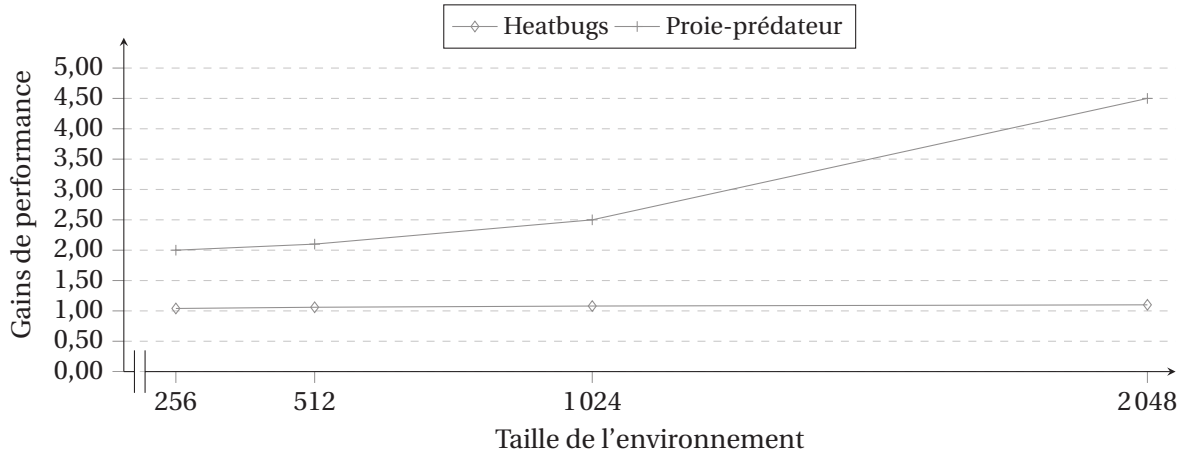


FIGURE 6.8 – Gains de performance entre les versions CPU et hybride des différents modèles.

Cependant, en observant la figure 6.8, on retrouve une des limites déjà rencontrée au chapitre précédent : les résultats de performance varient énormément en fonction du modèle considéré. En effet, alors que le gain de performance pour le modèle Proie-prédateur atteint 4.5, celui du modèle *Heatbugs* n'excède pas 1,1. Ainsi, même si le modèle considéré est compatible avec les critères du principe de délégation GPU et rempli les exigeances des 5 étapes de la méthode, il est possible d'obtenir de faibles gains de performance. Dans le cas du modèle *Heatbugs*, ces résultats s'expliquent par la part du modèle qui utilise le GPGPU. En effet, seul un faible pourcentage des calculs bénéficient du calcul sur carte graphique. Ainsi, les calculs qui continuent d'utiliser le CPU brident les performances globales du modèle.

Plus généralement, ce bridage des performances peut aussi venir du fait que le modèle considéré ne contient aucunes dynamiques environnementales. Ces dernières étant les plus facilement parallélisables, elles offrent généralement de très gros gains de performance une fois traitées par un module GPU. De plus, lorsque dans le modèle seuls des calculs de perceptions peuvent être transformés, les performances du modèle deviennent alors très dépendantes du nombre d'agents présents : si ce nombre est faible, le gain peut être négatif. Les performances vont donc dépendre des caractéristiques du modèle considéré.

De plus, les gains de performance vont aussi varier en fonction du type (professionnel ou grand public) et du nombre de GPU. Le fait d'avoir utilisé, pour nos expérimentations, qu'une seule carte graphique a certainement impacté la qualité des résultats obtenus. En effet, vu que cette dernière doit aussi gérer l'affichage du système d'exploitation et de la simulation, ses ressources dédiées aux calculs GPGPU en sont d'autant diminuées [7]. Ainsi, les performances du modèle vont aussi fortement dépendre de la configuration matérielle utilisée.

Nous pensons donc qu'une évaluation des performances escomptées doit être réalisée au préalable de toute application afin de ne pas perdre de temps dans des développements qui n'apporteraient aucun bénéfice en termes de performance. Actuellement, cela représente une limite pour notre méthode qui à ce stade ne définit pas le niveau d'accélération qu'il va être possible d'obtenir. Considérant cette limite, nous proposons au chapitre 8 une première piste de recherche pour y répondre.

7. L'annexe C présente une solution pour améliorer les performances de rendu de TurtleKit qui utilise le moteur OpenGL lorsque l'ordinateur utilisé ne contient qu'une seule carte graphique.

6.3.2 Du point de vue conceptuel

Tous les cas d'étude et expérimentations menés jusqu'ici soulignent la polyvalence dont fait preuve notre approche dans le sens où elle peut être appliquée sur une grande variété de modèles : MLE, les *boids* de REYNOLDS, *Game Of Life*, *Schelling's Segregation*, *DLA*, *Fire*, *Heatbugs* et enfin Proie-prédateur.

Un autre avantage de notre méthode est son accessibilité. En effet, il a été très facile, en suivant la méthode, d'identifier des parties des différents modèles qui ont pu être transformées pour bénéficier du GPGPU. Les 5 étapes amènent l'utilisateur à identifier les parties compatibles et à créer le ou les modules GPU correspondants aux calculs éligibles. De plus, les modules GPU possèdent toujours la même structure (*cf.* algorithme 6.1) se basant sur des *kernels* très simples à écrire comportant seulement quelques lignes de codes (très similaire au langage C).

D'ailleurs, cette méthode produit des modules génériques et donc réutilisables dans d'autres contextes que ceux pour lesquels ils ont été créés. Cette réutilisabilité a été illustrée de nombreuses fois pendant nos expérimentations et dernièrement par le modèle Proie-prédateur avec la réutilisation du module *GPU field perception* initialement créé pour le modèle MLE.

Enfin, dans le chapitre 4, nous avons vu que le principe de délégation GPU suit la perspective qui vise à déplacer une partie de la complexité des agents vers l'environnement afin de mieux la gérer. Cette perspective est clairement visible dans le dernier modèle sur lequel la méthode a été appliquée : pour Proie-prédateur, l'application de la méthode a transformé tous les calculs annexes aux comportements des agents permettant de ce fait d'avoir un modèle (et donc un code source) très lisible et facilement compréhensible. Nous rediscutons d'ailleurs de cet aspect au chapitre suivant.

6.4 Résumé du chapitre

Comme point final à ce travail, nous avons considéré de formaliser dans ce chapitre le principe de délégation GPU expérimenté tout au long de ce manuscrit. Ainsi, une méthode de conception divisée en 5 étapes a été proposée. Avec comme objectif de définir un cadre pour l'implémentation de simulations multi-agents sur le GPU, cette méthode se devait également de prendre en considération des aspects tels que la *généricité*, l'*accessibilité* et la *réutilisabilité* (nos trois critères d'évaluation). L'expérimentation réalisée par la suite sur deux nouveaux modèles multi-agents a montré que cette méthode prenait en compte ces critères tout en offrant des gains de performance significatifs.

À la vue de tous ces résultats, il convient de conclure ce travail et de discuter de l'impact que peut avoir une telle méthode sur le développement de simulations multi-agents. C'est ce que nous proposons maintenant de faire au chapitre suivant.

TROISIÈME PARTIE

CONCLUSION ET PESPECTIVES

CHAPITRE 7

CONCLUSION

Sommaire

La grande diversité des objectifs que la simulation multi-agents peut adresser (étude de systèmes complexes, de phénomènes biologiques, conception de systèmes multi-agents pour la robotique, etc.) montre toute la richesse du paradigme multi-agent. L'augmentation des capacités de calcul de nos ordinateurs actuels ainsi que la qualité (et la variété) des outils existants, dédiés à ce paradigme de conception, ont permis une envolée spectaculaire du nombre de simulations multi-agents et témoigne de sa popularité grandissante.

Cependant, nous avons vu au chapitre 1 que les performances constituent un verrou majeur dans ce domaine. Ainsi, il n'est pas rare de devoir simplifier le modèle ou de faire des compromis sur les caractéristiques de ce dernier pour compenser les limites introduites par ce manque de performance. D'autant plus, à l'heure où il est de plus en plus question de simulations à large échelle et/ou multi-niveaux [1], on peut facilement prédire que le besoin en ressources de calcul va augmenter de manière quasi exponentielle à court et moyen termes.

Ainsi, la motivation de notre travail de thèse a été d'apporter des solutions aux problèmes de performance que l'on peut rencontrer quand on simule des modèles multi-agents. Suivant les orientations prises en direction du calcul haute performance par différents groupes de recherche et industriels, nous nous sommes tournés en particulier vers l'utilisation de la puissance computationelle des cartes graphiques via le GPGPU. Comme énoncé au chapitre 2, cette technologie offre en effet un des meilleurs rapport en termes de performance, prix et consommation énergétique tout en permettant à tout un chacun d'utiliser une solution de calcul intensif grâce à la disponibilité de cette dernière dans de nombreux ordinateurs.

Néanmoins, malgré les avantages que peut avoir le GPGPU, nous avons souligné au chapitre 2 que cette technologie est difficile à mettre en œuvre. Le GPGPU s'accompagne en effet d'un contexte de programmation particulier du fait qu'il s'appuie sur un parallélisme de type SIMD qui nécessite notamment de suivre les principes de la programmation par traitement de flux de données (*stream processing paradigm*). Utiliser le GPGPU de manière efficiente peut donc être très complexe selon le cadre et les objectifs poursuivis.

1. Un exemple flagrant illustrant ce courant est celui de *IBM Research* qui parle de simulations de trafic routier comptant des milliards d'agents [Suzumura and Kanezashi, 2012].

7.1 Problématiques abordées

Dans le cadre des simulations multi-agents, les spécificités d'implémentation qui accompagnent la programmation sur GPU obligent à repenser la modélisation multi-agents, notamment car il n'est pas possible d'adopter une conception orientée objet classique. De ce fait, nous avons vu au chapitre 2 que les modèles multi-agents usuels ne peuvent être simulés sur GPU sans un effort de traduction conséquent et non trivial.

De plus, considérer l'utilisation des GPU, reposant sur des architectures massivement parallèles, pour accélérer l'exécution des simulations exacerbe les problèmes d'implémentation déjà présents en séquentiel, que nous avons mentionné au chapitre 1. En effet, les différents outils dédiés à la simulation multi-agent proposent une programmation haut niveau limitant la réutilisabilité des modèles sur d'autres plates-formes. Cela oblige également d'adopter une philosophie et une modélisation propre à chacun des outils ce qui restreint fortement l'accessibilité de ces derniers.

Dans ce contexte, nous avons réalisé au chapitre 3 un état de l'art des travaux mêlant GPGPU et simulations multi-agents afin d'examiner les directions de recherche prises par la communauté. Cet état de l'art [Hermellin et al., 2014, Hermellin et al., 2015] nous a permis d'identifier deux approches d'implémentation différentes.

— La première consiste à implémenter la simulation entièrement sur le GPU (tout-sur-GPU). Cependant, de par la difficulté de proposer des solutions de modélisation multi-agent réutilisables et accessibles intégrant le GPGPU, la plupart des travaux basés sur cette approche reste focalisée sur une recherche de performances brutes dans un contexte applicatif. Il existe bien certains travaux qui tentent tout de même de pallier les contraintes du GPGPU en encapsulant son usage dans un langage de plus haut niveau, mais ces solutions restent compliquées du point de vue de l'accessibilité.

— La deuxième est dite hybride et consiste à partager l'exécution d'une simulation entre le CPU et le GPU. Ainsi, au contraire du tout-sur-GPU, cette approche se veut plus modulaire de par le fait qu'il est possible de choisir ce qui va être exécuté ou non sur le GPU. Cela permet d'envisager des simulations multi-agents plus complexes avec, par exemple, des modèles d'agents qui mélangent architectures réactives et cognitives. Grâce à sa conception, l'approche hybride offre également la possibilité de prendre en considération des aspects tels que l'accessibilité, la réutilisabilité et la généricité des solutions proposées.

Grâce à cet état de l'art, nous avons également remarqué que la majorité des travaux basés sur une approche hybride cache l'utilisation du GPGPU aux utilisateurs : le recours au GPGPU se fait alors de manière totalement transparente. Ce choix, à cause de la très grande variété des modèles, n'est cependant pas assez générique et ne permet pas de prendre en compte tous les cas et besoins que l'implémentation d'un modèle multi-agent exécuté sur le GPGPU peut nécessiter.

Ainsi, et comme conclusion à cette étude, nous avons argumenté qu'une meilleure intégration du GPGPU ne pouvait se faire qu'avec une amélioration de l'accessibilité, de la réutilisabilité et de la généricité des solutions proposées [2] et qu'elle passait nécessairement par l'utilisation d'une approche hybride. La difficulté était donc de trouver une solution répondant à ces critères.

7.2 Résumé des contributions

Dans le chapitre 4, nous avons présenté le principe de *délégation GPU des perceptions agents*. Celui-ci propose d'utiliser de manière directe le GPGPU (et donc de profiter de la puissance des GPU) au travers d'une approche hybride tout en conservant l'accessibilité et la facilité de réutilisation d'une interface de programmation orientée objet. Proche des objectifs poursuivis dans

2. Nous avons d'ailleurs défini ces aspects comme essentiels dès le chapitre 1.

cette thèse et des critères que nous avons identifiés comme nécessaires à une meilleure intégration du GPGPU, nous avons ainsi choisi ce principe comme base de départ à nos recherches et développements.

Ce principe de conception, qui appartient au courant E4MAS [Weyns and Michel, 2015], s'inspire de travaux tels que [Weyns et al., 2007, Ricci et al., 2011]. Ainsi, de la même manière que pour les *artefacts* [Ricci et al., 2011], ce principe propose de réifier une partie des calculs du modèle (comme les calculs de perception) dans l'environnement sous la forme de dynamiques environnementales traitées par des modules de calcul GPU. Cependant, ce principe n'ayant été appliqué que sur un cas d'étude prometteur [Michel, 2013a], il était difficile de présager de ses capacités réelles.

Nous avons donc décidé de mettre à l'épreuve la faisabilité et la réutilisabilité de ce principe en l'appliquant dans un premier temps sur le modèle des *boids* de REYNOLDS [Hermellin and Michel, 2015, Hermellin and Michel, 2016d, Hermellin and Michel, 2016b] (*cf.* section 4.2). Grâce à cette première expérimentation, nous avons pu nous approprier ce principe de conception et surtout le faire évoluer. En effet, nous avons étendu le critère du principe de délégation GPU afin de le rendre plus générique.

Pour tester sa généricité, nous avons appliqué cette nouvelle version du principe de délégation GPU sur quatre autres modèles multi-agents : *Game of life*, *Schelling's Segregation*, *Fire* et *DLA* [Hermellin and Michel, 2016c]. Cette expérimentation, détaillée au chapitre 4, a souligné la capacité de l'approche à produire des modules GPU indépendants et réutilisables. De plus, grâce à la modularité apportée par l'approche hybride utilisée, les modules GPU créés reposent sur des *kernels* très simples (seulement quelques lignes de code) améliorant de ce fait l'accessibilité du GPGPU.

D'ailleurs, il est apparu durant ces applications successives du principe de délégation GPU que nous suivions toujours le même processus d'implémentation [Hermellin and Michel, 2016f] (le tableau 7.1 et la figure 7.2 résument les expérimentations menées, la figure 7.1 présentent des captures d'écran des simulations en cours d'exécution). Ainsi, dans le chapitre 6, nous avons généralisé l'application du principe de délégation GPU sous la forme d'une méthode de conception basée sur ce principe. Ici, notre objectif a été de permettre à un développeur n'ayant aucune expérience dans le domaine de la programmation GPU de l'aborder d'une manière simple, et surtout itérative.

Modèle	**Gain de performance** Taille de l'environnement 256	512	1 024	2 048	**Densité d'agents**	**Calcul traduit**	**Module GPU**
Flocking	1.40	1.40			entre 1 % et 61 %	calcul des orientations moyennes	*Average*
Game of Life	2.00	2.20	2.23	5.10	fixée à 50 %	mise à jour des états des cellules	*Voisinage*
Segregation	4.80	4.70	4.65	4.60	fixée à 90 %	détection des voisins	*Voisinage*
Fire	1.40	1.64	1.79	2.09	entre 10 % et 100 %	diffusion de la chaleur	*Diffusion*
DLA	13.5	12.7	10.7	10.0	entre 10 % et 90 %	détection de présence	*Voisinage*
Heatbugs	1.04	1.06	1.08	1.10	fixée à 40 %	diffusion de la chaleur, delta de température	*Diffusion* et *Différence*
Proie-prédateur	2.0	2.1	2.5	4.5	fixée à 20 % puis à 40 %	gradient de présence	*GPU field perception*

TABLEAU 7.1 – Résumé des expérimentations menées.

Nous avons donc défini une méthode générique de conception qui se focalise sur l'accessibilité et permet de capitaliser sur les efforts de modélisation et d'implémentation. Au vu des résultats obtenus suite aux différentes expérimentations réalisées (*cf.* sections 4.2.4, 5.2 et 6.3), la définition de cette méthode nous a permis d'atteindre les objectifs définis au début de ce manuscrit en pro-

posant une solution qui :

— améliore l'**accessibilité** du GPGPU dans le contexte des simulations multi-agents ;

— définit une approche **générique** permettant de considérer des modèles multi-agents plus hétérogènes ;

— promeut une **réutilisabilité** des outils créés et du code développé.

Ainsi, alors que dans les chapitres 1 et 3 nous avons montré que le GPGPU peinait à être adopté dans le domaine des simulations multi-agents, le cadre conceptuel qu'offre la méthode développée dans ce manuscrit devrait permettre une meilleure intégration de cette technologie dans ce domaine. En cela, elle devrait également accroître la diffusion de cette technologie dans la communauté multi-agent.

De plus, alors qu'au départ le principe de délégation GPU permettait de transformer en dynamiques environnementales uniquement des calculs de perception n'impliquant pas les états des agents, la définition de la méthode au chapitre 6 nous montre que maintenant tous les calculs présents dans le modèle multi-agent peuvent être pris en compte. En effet, lors de la première étape de cette méthode, nous considérons tous les calculs du modèle afin de vérifier leur compatibilité avec les critères du principe de délégation. Ainsi, cette méthode propose d'identifier et de déléguer à l'environnement des calculs de pré-traitements et cela sans distinction de contexte (perception, délibération, etc.). Cela ouvre des perspectives intéressantes d'un point de vue génie logiciel orienté agent surtout dans le contexte E4MAS dans lequel on se trouve (complexité du modèle agent, conception et modelisation des modèles agents, etc.). Le chapitre détaille cette perspective.

En ce qui concerne les limites de notre approche, on voit en particulier que les résultats de performance obtenus sont très différents en fonction des modèles considérés. De très bons pour le modèle *DLA*) à peu significatifs (pour le modèle *Heatbugs*), comme l'illustre le tableau 7.1 et la figure 7.2, ces résultats variables soulignent la difficulté actuelle de notre méthode à prédire le niveau de gain qu'il va être possible d'obtenir. De ce fait, il va donc être nécessaire de travailler sur une meilleure intégration de cette méthode dans les outils qui lui sont associés et de proposer des solutions claires autour de ces variations de performance. Cela représente une partie des perspectives de recherche autour de ce travail et nous allons les détailler dans le prochain chapitre.

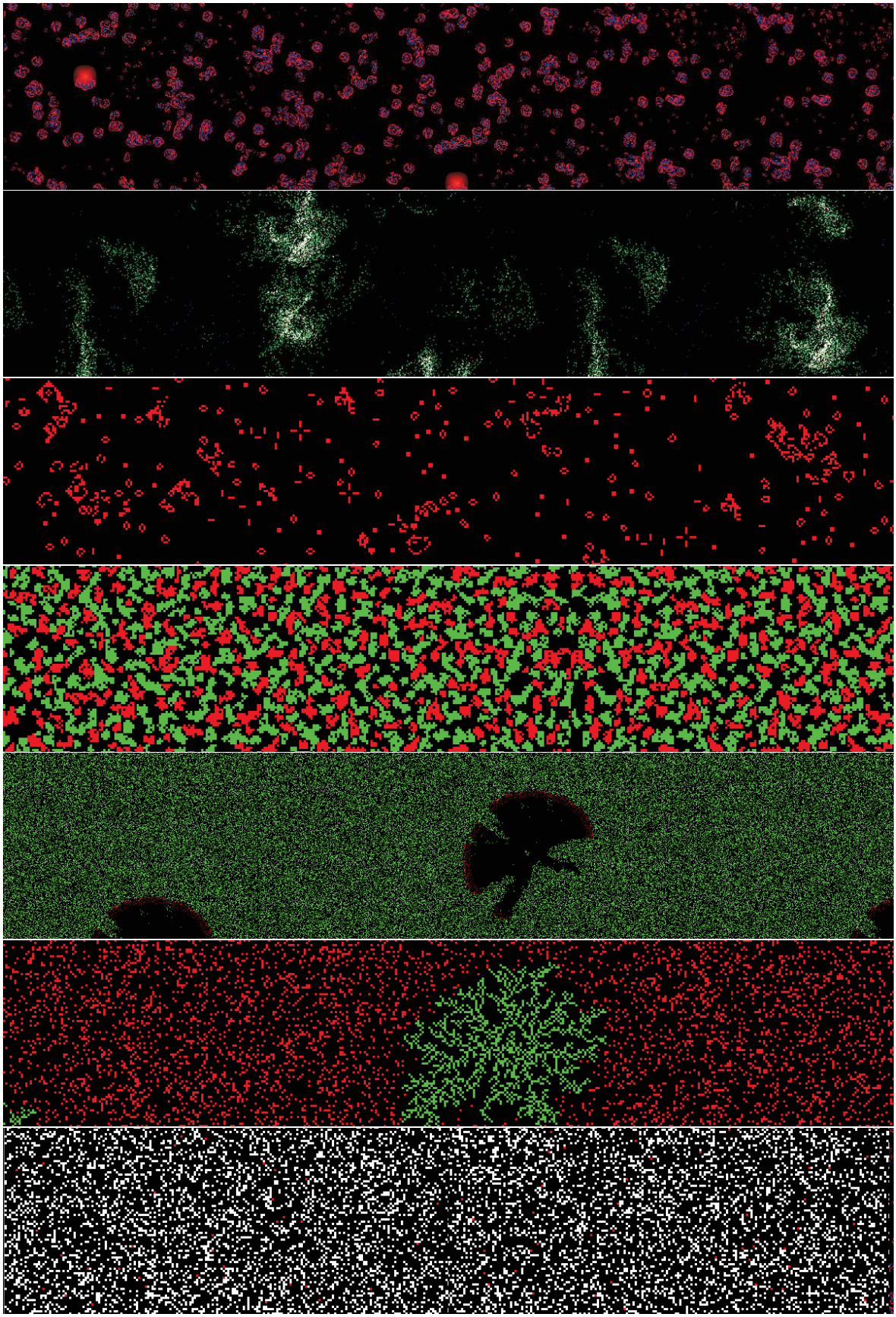


FIGURE 7.1 – Captures d'écran des différentes simulations avec de haut en bas : *MLE, Flocking, Game of Life, Segregation, Fire, DLA* et enfin *Proie-prédateur*.

FIGURE 7.2 – Résumé des gains de performance obtenus lors des différentes expérimentations menées.

CHAPITRE 8

PERSPECTIVES DE RECHERCHE

Sommaire

Si la méthode de conception que nous avons proposé est fonctionnelle et offre un cadre opérationnel permettant d'utiliser le GPGPU dans les simulations multi-agents, elle représente une solution appelée à évoluer, d'une part pour combler ses limites, d'autre part pour répondre à un domaine très dynamique. Ainsi, nous présentons maintenant quelques-unes des perspectives de recherche que nous considérons comme importantes à explorer. Celles-ci peuvent être divisées en deux parties :

— À court terme, il s'agit d'abord de travailler sur les outils logiciels associés à notre méthode et sur son intégration en leur sein. Ceci afin de rendre ces outils encore plus accessibles et génériques mais aussi pour étendre le champ d'application de notre méthode tout en répondant aux différentes limites soulevées tout au long de ce manuscrit.

— À moyen et long termes, il nous semble important de réfléchir au positionnement de notre méthode dans le domaine des simulations multi-agents et, d'une manière plus générale, à l'apport que peut avoir une telle méthode dans le cadre du génie logiciel orienté agent.

8.1 Perspectives à court terme

8.1.1 Améliorer les outils et l'intégration de la méthode dans ces derniers

Pour tous nos tests d'intégration du GPGPU dans des simulations multi-agents, nous avons utilisé la plate-forme TurtleKit. Cependant, l'architecture de cette dernière n'a jamais été pensée dans l'optique d'intégrer cette technologie et encore moins des modules de calcul GPU. Ainsi,

maintenant que nous avons défini une méthode fonctionnelle autour du GPGPU et malgré le fait que TutleKit permet d'intégrer les modules créés avec cette méthode, il convient maintenant de repenser son architecture.

Dans un premier temps, l'architecture de TurleKit se doit d'être plus modulaire ceci afin de simplifier le développement et l'intégration des modules GPU. Ensuite, cette architecture devra également permettre la réutilisation directe des modules entre les simulations et cela sans avoir à modifier au préalable le noyau de la plate-forme (ce qui est le cas actuellement). Enfin, nous pouvons imaginer la création d'une solution logicielle accompagnant l'utilisateur dans l'application des différentes étapes de la méthode. Celle-ci pourrait prendre par exemple la forme d'un assistant de configuration tel qu'on en trouve dans bon nombre de logiciels.

D'une manière générale, cette piste de recherche à court terme consiste globalement à proposer une meilleure intégration de notre méthode dans la plate-forme TurtleKit afin d'améliorer toujours plus son accessibilité et favoriser la réutilisation des modules.

8.1.2 Variation des performances et estimation des gains

Nous avons observé dans les chapitres 5 et 6, que les gains de performance (obtenus grâce à notre méthode) variaient énormément en fonction du modèle considéré. En effet, même si un modèle valide la deuxième étape de notre méthode (compatibilité des calculs), les gains de performance peuvent ne pas être intéressants car ces derniers dépendent fortement de la configuration matérielle utilisée ainsi que des caractéristiques du modèle considéré. Cette disparité des résultats représente clairement une limite de notre méthode car il devient alors difficile d'estimer avec précision les gains escomptés. Mais surtout il est très complexe, dans ce contexte, d'identifier si un modèle pourra bénéficier ou non du GPGPU.

Pour remédier à ce problème, une des solutions que nous envisageons repose sur la création d'un outil de benchmark qui permettrait de définir l'intérêt d'appliquer notre méthode sur le modèle considéré en fonction du nombre d'agents et de la configuration matérielle utilisée. Autrement dit, l'idée est de proposer un outil qui implémente plusieurs modèles jouets (des exemples de simulations multi-agents très représentatives des dynamiques que l'on peut rencontrer, *toys models*) comportant un ensemble de fonctions agents et de dynamiques environnementales récurrentes, ces dernières réutilisant les modules GPU créés avec notre méthode. Ainsi, l'utilisateur qui exécutera ces simulations pourra avoir une idée du seuil au-dessus duquel une implémentation GPU pourrait être efficiente et se faire un avis sur les gains de performances qu'il sera possible d'obtenir en fonction des caractéristiques de son propre modèle et de la configuration utilisée.

8.1.3 Élargir le champ d'application de la méthode

Au chapitre 6, nous avons souligné la polyvalence de notre méthode dans le sens où elle peut être appliquée sur une grande variété de modèles. Cependant, il nous faut modérer cet avantage car tous nos cas d'études ont porté sur des modèles possédant des caractéristiques communes (comme un environnement discrétisé par exemple). En effet, un des cadres de notre méthode a été d'appliquer des dynamiques environnementales sur un environnement discrétisé pour lequel le principe de délégation GPU est facilement implémentable.

Cependant, notre méthode n'est pas limitée à ce contexte. Il n'est jamais fait mention dans ses critères ou au cours des différentes étapes d'une obligation pour les modèles de posséder un environnement discrétisé. Ainsi, un de nos objectifs sera de considérer d'autres types de modèle sur lesquels appliquer notre méthode de conception comme par exemple des modèles possédant des environnements continus.

La question qui se posera alors, est de savoir s'il faudra faire évoluer notre méthode pour prendre en compte concrètement les environnements continus ou s'il faudra proposer une conversion des modèles continus vers une version discrète. Cette question est légitime car l'application de cette méthode est rendue plus facile par la discrétisation de l'environnement. Cette dernière simplifie en effet la parallélisation (adéquation entre l'environnement et l'architecture du GPU)

mais aussi l'application des dynamiques environnementales. Un bon exemple est celui de notre modèle de *flocking* qui est de base un modèle continu mais qui, de par son implémentation dans TurtleKit (qui plonge les modèles dans un environnement discret), a permis une application rapide du principe de délégation GPU.

Ouvrir notre méthode à de nouveaux types de modèle devrait, dans tous les cas, renforcer son champ d'application et permettre de définir de nouveaux modules GPU réutilisables.

8.2 Perspectives à moyen et long termes

8.2.1 Impact sur le génie logiciel orienté agent

Nous avons à plusieurs reprises souligné la complexité liée à la conception de simulations multi-agents et à l'utilisation des plates-formes dédiées. Dans le chapitre 1, nous avons montré que la grande diversité des plates-formes de développement et leurs spécificités n'offraient pas des conditions optimales de développement dans le sens où leur accessibilité ainsi que la réutilisabilité des modèles créés avec ces outils pouvaient être limitées. Aux chapitres 2 et 3, nous avons établi que les architectures parallèles allaient exacerber les problèmes déjà présents en séquentiel. Ainsi, et d'une manière générale, on a besoin de solutions qui facilitent la modélisation et l'implémentation de simulations multi-agents et cela quelque soit le contexte technologique.

Le courant E4MAS, que nous avons évoqué plusieurs fois dans ce manuscrit, s'inscrit dans cette perspective de simplification au travers d'une décomplexification des comportements agents rendus possible grâce à la place donnée aux environnements (l'environnement est considéré comme essentiel). Bien que l'on ait mis de côté cet aspect [1], la question de la complexité prend tout son sens avec la méthode que nous proposons car le fait de se baser sur le principe de délégation GPU (qui explicite la séparation entre le modèle agent et l'environnement) et le fait de choisir ce qui va être exécuté par le GPU (grâce à l'approche hybride), permet de fournir à l'agent des percepts et des moyens d'action de haut niveau, afin de simplifier son comportement et d'en garder l'essentiel. En ce sens, nos contributions poursuivent bien les perspectives du courant E4MAS et s'inscrivent dans la lignée de travaux tels que [Viroli et al., 2006, Weyns et al., 2007, Ricci et al., 2011, Michel, 2015, Weyns and Michel, 2015].

Ainsi, toujours dans une démarche de génie logiciel orienté agent, il semblerait pertinent d'étudier l'impact que peut avoir notre méthode sur la programmation multi-agents. Plus précisément, il conviendrait d'évaluer la capacité de notre méthode à simplifier l'implémentation et la modélisation multi-agent et cela en dehors d'un contexte GPGPU. Un exemple qui nous conforte dans l'intérêt de cette perspective de recherche est celui de l'expérimentation du modèle Proie-prédateur réalisée au chapitre 6. L'application de notre méthode sur ce modèle a mené à la disparition de tous les calculs annexes aux comportements des agents permettant de ce fait d'avoir un modèle comportemental (et donc son code source) très lisible et facilement compréhensible [2].

8.2.2 Ordonnancement et dynamiques

Un autre aspect n'a pas pu être abordé dans cette thèse : celui de l'apparition de nouvelles dynamiques suite aux modifications engendrées par l'application de notre méthode sur les modèles considérés. Pourtant, nos simulations de *flocking* ont montré que, selon la version du modèle exécutée (séquentielle ou hybride), les comportements collectifs observés pouvaient être très différents.

En effet, dans la version séquentielle, l'ordonnancement du modèle (les agents perçoivent et agissent directement les uns après les autres) va faire converger l'orientation moyenne des agents

1. La diminution de la complexité n'était pas le but premier de nos travaux de thèse, ce qui explique que l'on ait mis cette question en suspens jusqu'ici.

2. L'annexe B présente le code source des comportements des agents avant et après l'application de notre méthode.

du système vers une valeur commune [3] modulo une petite variation aléatoire sur les directions individuelles. Visuellement parlant, il en résulte un *flocking* statique (mouvement coordonné dans une unique direction modulo de petites variations) similaire à ce que l'on peut observer dans les plates-formes que nous avons présentées au chapitre 4 (*cf.* section 4.2). Dans la version hybride utilisant le module GPU *Average*, tous les agents perçoivent le même état de l'environnement pour un même instant t [4]. Ainsi, le système ne converge pas vers une valeur unique, comme c'est le cas pour la version séquentielle. Il en résulte des comportements collectifs de *flocking* plus riches en terme de dynamique, avec des changements de directions collectifs soudains et importants.

L'application de notre méthode peut donc ne pas être neutre mais elle souligne également l'importance que l'on doit donner à l'ordonnancement (l'ordre dans lequel les agents sont activés et produisent leurs perceptions et actions). Il existe d'ailleurs de nombreux de travaux qui traitent de l'ordonnancement dans les systèmes multi-agents et de l'impact qu'il peut avoir sur l'exécution et la dynamique des systèmes. Parmi ces travaux, on peut citer par exemple [Fatès and Chevrier, 2010] qui montre que même avec seulement deux agents aux comportements minimalistes, plongés dans un environnement simple, il est possible d'obtenir des dynamiques très différentes suivant la manière dont le reste du modèle est élaboré (perception, action, temps, interaction, etc.) ou encore IRM4S qui traite de l'utilisation des notions d'influence et de réaction pour la simulation de systèmes multi-agents [Michel, 2007, Chevrier and Fatès, 2008].

De plus, notre méthode, de par sa conception hybride, impose de séparer les calculs liés aux agents (CPU), des calculs liés à l'environnement (GPU) ce qui introduit un mécanisme en deux phases sur les parties du modèle concernées. Ce mécanisme peut être, à juste titre, relié à une approche de type IRM4S (comme souligné dans [Michel, 2015]). Ainsi, il pourrait être très intéressant dans un premier temps de voir dans quelle mesure il est possible d'intégrer le modèle IRM4S dans nos travaux.

D'une manière générale, une de nos perspectives de recherche est donc de mener une réflexion autour de l'ordonnancement des agents et sur son impact sur la dynamique des systèmes dans un contexte d'utilisation hybride du GPGPU ou d'architectures matérielles massivement parallèles.

3. Le premier agent s'aligne et modifie sa direction, le deuxième en fait autant mais perçoit la nouvelle direction du premier et la prend en compte, et ainsi de suite. Le dernier agent qui agit perçoit donc un système dont la configuration est très homogène.

4. Les données utilisées par les agents pour effectuer leur perception sont calculées au préalable par le biais d'une dynamique environnementale traitée par le module GPU.

Quatrième partie

Annexes

ANNEXE A

ARCHITECTURE DE TURTLEKIT

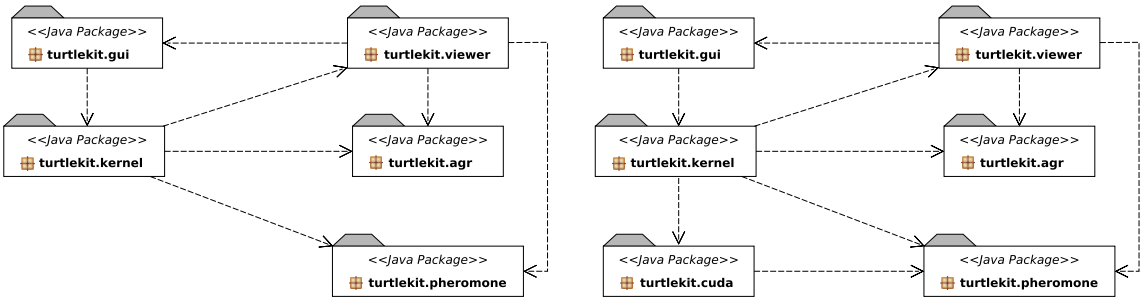
TurtleKit [1] est une plate-forme de simulation qui utilise un modèle multi-agent spatialisé où l'environnement est discrétisé sous la forme d'une grille de cellules à deux dimensions [Michel et al., 2005]. Implémentée en Java à l'aide de la librairie de développement multi-agent MaDKit [2] [Gutknecht and Ferber, 2001], TurtleKit repose sur des modèles d'agents et d'environnements inspirés par le langage de programmation Logo. L'un des objectifs principaux de TurtleKit est de fournir aux utilisateurs finaux une plateforme open source, généraliste facilement accessible et extensible. En particulier, l'API de TurtleKit est orientée objet et son utilisation repose sur l'héritage de classes prédéfinies.

Dans cette annexe, nous présentons l'architecture de la plate-forme TurtleKit grâce à différents diagrammes :

— Les figures A.1a et A.1b représentent l'architecture de TurtleKit sous forme de diagrammes de package.

— Les figures A.2 et A.3 représentent l'architecture de TurtleKit sous forme de diagrammes de classes.

Pour chacun de ces cas, nous présentons l'architecture de TurleKit avant l'intégration du GPGPU (figures A.1a et A.2) et après l'intégration du calcul sur GPU (figures A.1b et A.3), l'exemple des phéromones (modèle MLE), présenté au chapitre 4, a été utilisé pour illustrer l'intégration de cette technologie dans l'architecture de la plate-forme.



(a) TurtleKit sans GPGPU

(b) TurtleKit avec GPGPU

FIGURE A.1 – Architecture de TurtleKit, diagramme de packages.

1. http ://www.turtlekit.org
2. http ://www.madkit.org

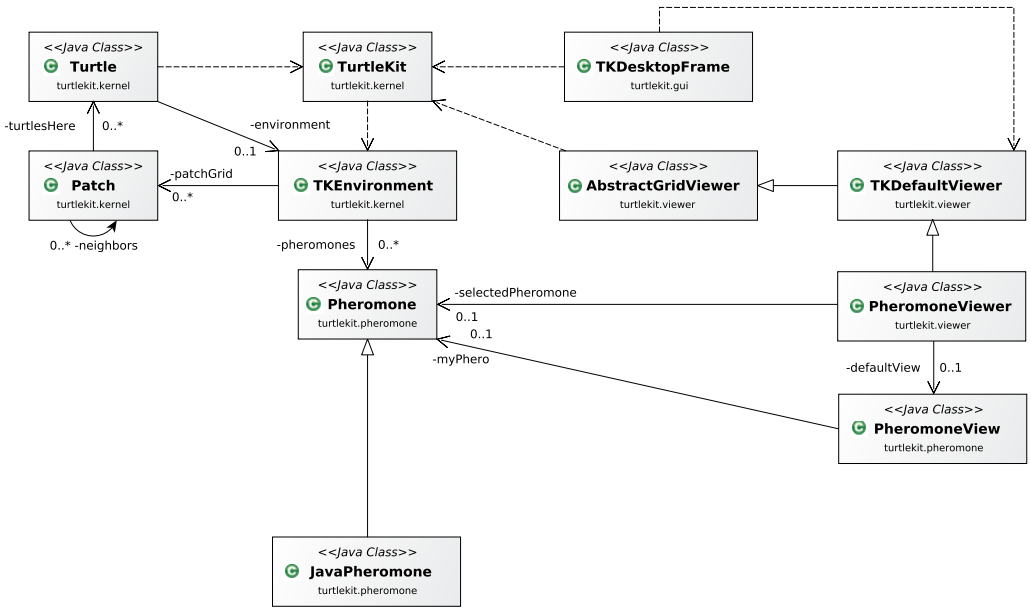


FIGURE A.2 – Architecture de TurtleKit, diagramme de classes simplifié sans GPGPU.

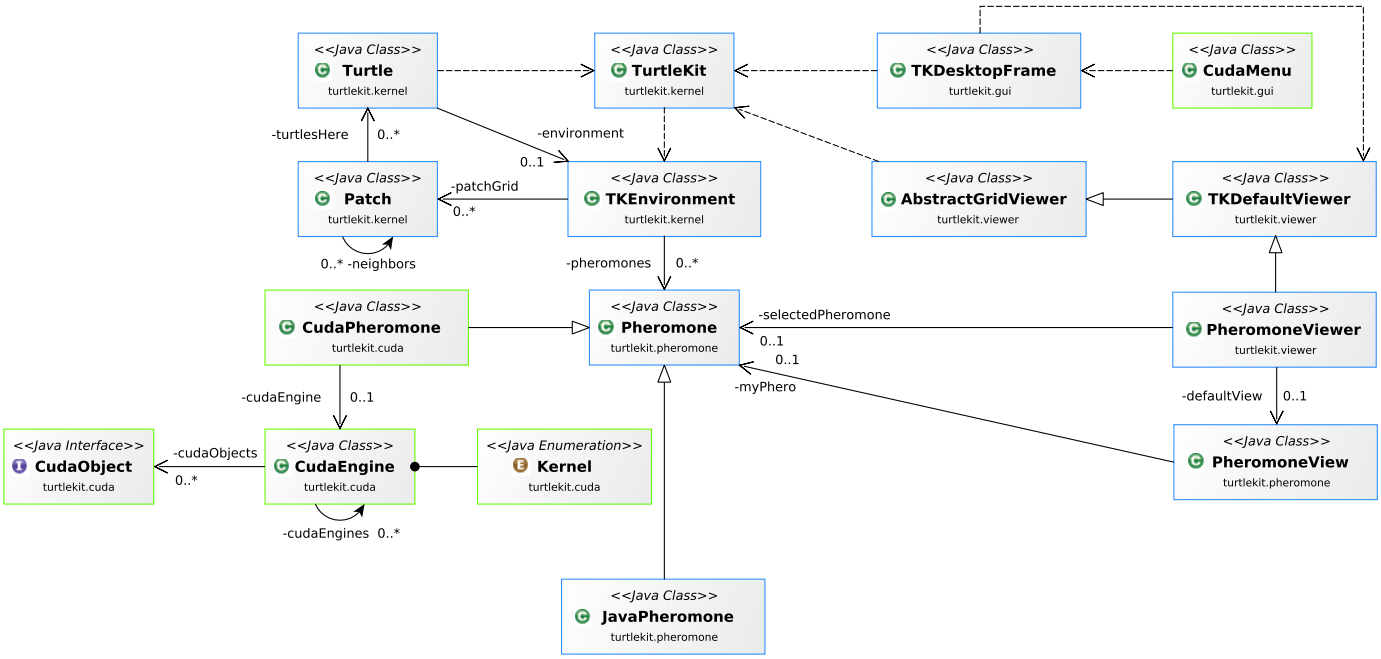


FIGURE A.3 – Architecture de TurtleKit, diagramme de classes simplifié avec GPGPU.

ANNEXE B

PROGRAMMER EN CUDA

Dans cette annexe, nous proposons un aperçu des techniques et méthodes de programmation autour de CUDA. Pour le lecteur qui voudrait en apprendre d'avantage, nous conseillons le site de Nvidia [1] et deux ouvrages de références : [Sanders and Kandrot, 2011] et [Cook, 2013].

CUDA est une architecture de traitement parallèle développée par Nvidia permettant de décupler les performances de calcul du système en exploitant la puissance des processeurs graphiques. L'environnement de développement CUDA inclut des extensions C et C++ qui permettent l'expression de données denses et complexes dans un contexte de parallélisme. Les programmeurs peuvent choisir d'exprimer le parallélisme avec des langages à hautes performances comme C, C++, Fortran.

Une fois ce programme écrit, il doit être traité par le compilateur NVCC fournit par l'environnement CUDA. Ce dernier va séparer les fichiers en C/C++, qui vont être compilés par le compilateur GCC puis traités par le CPU, des fichiers PTX, générés par NVCC et contenant les instructions exécutées par le GPU (la figure B.1 illustre ce processus).

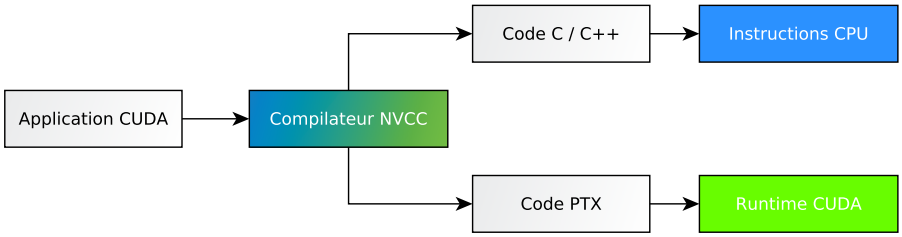


FIGURE B.1 – Processus de compilation d'un programme avec CUDA.

Le programme généré peut interagir avec CUDA de trois manières distinctes (voir figure B.2) :

— via les librairies : CUDA est livré avec des implémentations d'algorithmes optimisées pour cette architecture matérielle ;

— via le *Runtime* : c'est l'interface entre le GPU et l'application ;

— via le *Driver* : son rôle est de transmettre les calculs de l'application au GPU.

Quelques définitions

Host : c'est l'ordinateur qui sert d'interface avec l'utilisateur et qui contrôle le *device* utilisé pour exécuter les parties de calculs intensifs basés sur un parallélisme de données. Avec CUDA, l'*host* peut lire et écrire dans la *Global memory*, *Constant memory* et seulement écrire dans la *Texture memory*. C'est l'*host* qui est responsable de l'exécution des parties séquentielles de l'application.
Device : c'est le GPU connecté à l'*host* et qui va exécuter les parties de calcul intensif basé sur un parallélisme de données. Le *device* est responsable de l'exécution de la partie parallèle de l'application.

1. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`

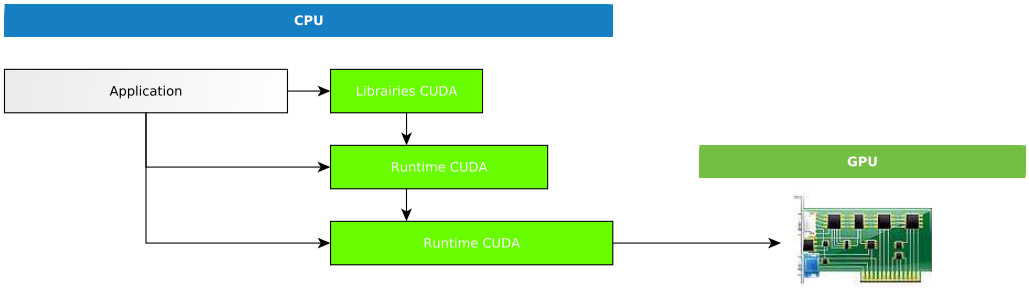


FIGURE B.2 – Intéraction entre un programme et CUDA.

Kernel : c'est une fonction qui peut être appelée depuis l'*host* et qui est exécutée sur le *device* simultanément par des milliers de *threads*.

Notions de *threads*, grille et *blocs*

CUDA adopte la hiérarchie suivante : un *kernel* est une programme exécuté par un ensemble de *threads* indépendants s'exécutant en parallèle sur le GPU. Les *threads* (fils d'instructions) sont groupés en *blocs* qui coopèrent et traitent le même code écrit en CUDA sur des données différentes liées au numéro de chaque *thread*. Les *blocs* sont groupés en grille et s'exécutent dans n'importe quel ordre, il n'y a pas de synchronisation entre les *blocs*. Les *threads* de deux *blocs* différents ne peuvent donc pas communiquer.

Identificateurs des *threads* et des *blocs*

Nombre de *blocs* dans la grille : `dim3 gridDim (gridDim.x, grimDim.y, grimDim.z)`
Nombre de *threads* par *blocs* : `dim3 blockDim (blockDim.x, blockDim.y, blockDim.z)`
Indice du *bloc* dans la grille : `dim3 blockIdx (blockIdx.x, blockIdx.y, blockIdx.z)`
Indice du *thread* dans le *bloc* : `dim3 threadIdx (threadIdx.x, threadIdx.y, threadIdx.z)`

Exemple d'utilisation :
bloc 1D : `threadID = threadIdx.x = x`
bloc 2D : `threadID = threadIdx.x + blockIdx.y * blockDim.x`
bloc 3D : `threadID = threadIdx.x + threadIdx.y * Dx + threadIdx.z * Dx * Dy` avec `Dx = blockDim.x, Dy = blockDim.y, Dz = blockDim.z`

Bien dimensionner les *blocs* et la grille

Pour obtenir de bonnes performances avec CUDA, il est important de bien dimensionner les *blocs* et la grille. Un *bloc* s'exécute sur un unique multiprocesseur. Donc, si une carte graphique possède n multiprocesseurs, il faut alors que la grille contienne au moins n blocks pour que chaque multiprocesseur soit actif. En effet, la carte sera plus efficiente si tous ces multiprocesseurs sont en activités. De plus, les *threads* d'un *bloc* sont exécutés par *warp*, c'est à dire par paquet de 32. Il est donc idéal d'avoir un nombre de *threads* par *bloc* qui soit un multiple de 32 afin d'éviter de se retrouver avec des *warps* contenant des *threads* inactifs. Enfin, il est important d'allouer un nombre important de *threads* par *bloc*, de manière à masquer les temps de latence des opérations mémoires et des calculs. En effet, lorsqu'une opération est effectuée par un *thread*, son résultat n'est pas disponible immédiatement. Il se passe quelques dizaines à quelques centaines de cycles d'horloge avant qu'il devienne disponible. Pendant ce temps-là, si la suite de l'exécution du *thread* dépend de ce résultat, le *thread* est bloqué. C'est pourquoi, il est nécessaire que le multiprocesseur ait d'autres *threads* de disponibles afin de continuer d'exécuter les instructions suivantes pendant ce temps de latence : c'est ce qu'on appelle masquer la latence.

Pour autant, il ne suffit pas d'allouer un nombre important de *threads* à un *bloc* pour avoir de

bonnes performances, et ce pour deux raisons :

— En fonction de sa capacité de calcul [2] une carte pourrait être limitée. Par exemple, une carte graphique avec une capacité de calcul de 1.3 ne peut avoir que 1024 *threads* d'allouer par *bloc*.

— Plus un *bloc* contient de *threads* et plus il utilise des registres (une mémoire spécifique aux *threads*), or cette quantité est très limitée.

Déclaration de fonctions et variables sous CUDA

Il existe trois types de fonctions principales lorsque l'on utilise l'environnement de développement CUDA :
`__host__` est une fonction exécutée sur le CPU (par défaut).
`__global__` est une fonction exécutée sur le GPU mais appelée par le CPU.
`__device__` est une fonction exécutée sur le GPU et appelée depuis une fonction GPU.
Chacune de ces fonctions s'utilise dans un contexte bien particulier car elles possèdent des restrictions d'utilisation :
`__global__` est une fonction appelée par l'*host*, exécutée par le GPU mais qui ne peut pas contenir de récursivité, de variable statique, de listes variables de paramètres et ne peut rien retourner.
`__device__` est une fonction appelée par le GPU, exécutée sur le GPU pouvant faire intervenir de la récursivité.
Ces fonctions sont accompagnées par trois types de variables différentes :
`__device__ int variable` est une variable stockée dans la mémoire globale du GPU (lente, valide pendant la durée du programme), accessible en interne par les *threads*, et allouée par le CPU.
`__shared__ int variable` est une variable stockée dans la mémoire partagée du *bloc* (rapide, taille réduite, durée de vie correspondante à celle du *bloc*), commune aux *threads* d'un même *bloc*.
`__device__ int variable` est une variable stockée dans la mémoire constante du GPU, visible par tous les *threads*.
Dans le cas ou rien n'est spécifié, la variable déclarée sera enregistrée dans le registre qui est une mémoire rapide associée aux *threads*.

Fonctions principales de l'API CUDA

`cudaThreadSynchronize()` est une fonction qui bloque l'exécution du *kernel* jusqu'à ce que tous les *threads* alloués pour ce calcul soient tous arrivés au même point de synchronisation.
`cudaChooseDevice()` est une fonction qui retourne une liste de *devices* disponibles.
`cudaGetDevice()` est une fonction qui retourne le *device* utilisé pour exécuter les *kernels* de calcul.
`cudaGetDeviceCount()` est une fonction qui retourne le nombre de *device* capable de faire du GPGPU.
`cudaGetDeviceProperties()` est une fonction qui retourne les informations et propriétés relatives au *device* sélectionné.
`cudaMalloc()` est une fonction d'allocation en mémoire globale (son fonctionnement est assez similaire de `malloc()` en C).
`cudaFree()` est une fonction qui libère la mémoire allouée avec `cudaMalloc()`.
`cudaMemcpy()` est une fonction qui permet la copie de données entre l'*host* et le *device*.

Exemple du code source d'un *kernel* de calcul

L'algorithme B.1 introduit le code source d'un *kernel* de calcul et le programme en C permettant de l'initialiser. Par cohérence, nous reprenons l'exemple du calcul de π énoncé au chapitre 2.

2. La capacité de calcul, appelée *compute capability* ou *SM version* en anglais, est représentée par un nombre *x.x* identifiant les fonctionnalités supportées par le GPU. Ce nombre est utilisé par les applications lors de leur exécution pour déterminer si le GPU sélectionné supporte les instructions et fonctions nécessaires à l'exécution des calculs.

Algorithme B.1 : Code source, calcul de π en CUDA C

```
#include "main.h"

#define Precision 10000

__device__ float calculIntervalle(int tid){
    return tid * (1.0 / Precision);
}

__global__ void calc(float *result) {
    int tid = blockIdx.x;
    float x = calculIntervalle(tid);
    if (tid < Precision)
        result[tid] = ((1.0/Precision) * 1 / ( 1 + (x*x) ));
}

int calculPiGPU(void) {
    float pi = 0.0;
    float resultat[Precision];
    float *dev_result;

    // allocate the memory on the GPU
    cudaMalloc((void**)&dev_result, Precision * sizeof(float));

    calc<<<Precision,1>>>(dev_result);

    // copy the array 'resultat' back from the GPU to the CPU
    cudaMemcpy(resultat, dev_result, Precision * sizeof(float),
                            cudaMemcpyDeviceToHost);

    // display the results
    for (int i=0; i<Precision; i++) {
        pi = pi + resultat[i];
    }

    pi = pi * 4;
    printf( "pi = %f\n", pi);

    // free the memory allocated on the GPU
    cudaFree(dev_result);

    system("pause");
    return 0;
}
```

Code source des modules GPU présentés dans le manuscrit

Nous présentons maintenant le code source des modules GPU créés pour le modèle MLE (algorithme B.2), pour le modèle de *flocking* (algorithme B.3), pour les modèles *Game of Life*, *Segregation* et *DLA* (algorithme B.4) et enfin pour le modèle Proie-prédateur (algorithme B.5).

Algorithme B.2 : Code source, calcul de la diffusion et de l'évaporation en CUDA C

```
1
2  //Convert 2D coordinates into one 1D coordinate
3  __device__ int convert1D(int x, int y,int width){
4    return y * width +x;
5  }
6
7  //Normalize coordinates for infinite world
8  __device__ int normeValue(int x, int width){
9      if(x < 0) //-1
10         return width - 1;
11     if(x == width)
12         return 0;
13     return x;
14 }
15
16 __device__ int* neighborsIndexes(int i, int j, int width, int height){
17     int dir[8];
18     dir[0] = convert1D(normeValue(i+1,width), j, width);
19     dir[1] = convert1D(normeValue(i+1,width), normeValue(j+1,height),width);
20     dir[2] = convert1D(i, normeValue(j+1,height),width);
21     dir[3] = convert1D(normeValue(i-1,width), normeValue(j+1,height),width);
22     dir[4] = convert1D(normeValue(i-1,width), j, width);
23     dir[5] = convert1D(normeValue(i-1,width), normeValue(j-1,height),width);
24     dir[6] = convert1D(i, normeValue(j-1,height),width);
25     dir[7] = convert1D(normeValue(i+1,width), normeValue(j-1,height),width);
26     return dir;
27 }
28
29 __device__ float getTotalUpdateFromNeighbors(float* tmp, int i, int j, int
       width, int height){
30     int iPlusOne = i + 1; int jPlusOne = j + 1;
31     int iMinusOne = i - 1; int jMinusOne = j - 1;
32        return
33         tmp[convert1D(normeValue(iPlusOne,width), j, width)] +
34         tmp[convert1D(normeValue(iPlusOne,width), normeValue(jPlusOne,height)
               ,width)] +
35         tmp[convert1D(i, normeValue(jPlusOne,height),width)] +
36         tmp[convert1D(normeValue(iMinusOne,width), normeValue(jPlusOne,height
               ),width)] +
37         tmp[convert1D(normeValue(iMinusOne,width), j, width)] +
38         tmp[convert1D(normeValue(iMinusOne,width), normeValue(jMinusOne,
               height),width)] +
39         tmp[convert1D(i, normeValue(jMinusOne,height),width)] +
40         tmp[convert1D(normeValue(iPlusOne,width), normeValue(jMinusOne,height
               ),width)];
41 }
42
43 //Diffusion and Evaporation Kernel
44 extern "C"
45 __global__ void DIFFUSION( int width, int height, float *values, float* tmp,
       float evapCoef)
46 {
47        int i = blockIdx.x * blockDim.x + threadIdx.x;
48        int j = blockIdx.y * blockDim.y + threadIdx.y;
49
50        if (i < width && j < height ){
51         int k = convert1D(i,j,width);
52         float total = values[k] + getTotalUpdateFromNeighbors(tmp, i, j,
               width, height);
53         values[k] = total - total * evapCoef;
54         }
55 }
```

Algorithme B.3 : Code source, calcul de la moyenne des orientations en CUDA C

```
1
2  //Convert 2D coordinates into one 1D coordinate
3  __device__ int convert1D(int x, int y,int width){
4    return y * width + x;
5  }
6
7  //Normalize coordinates for infinite world
8  __device__ int normeValue(int x, int width){
9      if(x < 0)
10         return x + width;
11     if(x > width - 1)
12         return x - width;
13     return x;
14 }
15
16 //Average Kernel
17 extern "C"
18 __global__ void AVERAGE(int envSizeX, int envSizeY, float* envData, float*
       result, int depth){
19     int tidX = blockIdx.x * blockDim.x + threadIdx.x;
20     int tidY = blockIdx.y * blockDim.y + threadIdx.y;
21
22     float moyenne = 0;
23     float nbNombre = 0;
24
25     if(tidX < envSizeX && tidY < envSizeY){
26         int borneInfX = tidX - depth;
27         int borneSupX = tidX + depth;
28         int borneInfY = tidY - depth;
29         int borneSupY = tidY + depth;
30         for(int i = borneInfX; i <= borneSupX; i++){
31             for(int j = borneInfY; j <= borneSupY; j++){
32                 float valeur = envData[convert1D(normeValue(i,envSizeX),
                       normeValue(j,envSizeY),envSizeY)];
33                 if(valeur != -1){
34                     moyenne += valeur;
35                     nbNombre++;
36                 }
37             }
38         }
39         if(nbNombre != 0){
40             result[envSizeY * tidX + tidY] = moyenne / nbNombre;
41         }
42     }
43 }
```

Algorithme B.4 : Code source, calcul du type de voisinage en CUDA C

```
1
2  //Convert 2D coordinates into one 1D coordinate
3  __device__ int convert1D(int x, int y,int width){
4    return y * width + x;
5  }
6
7  //Normalize coordinates for infinite world
8  __device__ int normeValue(int x, int width){
9      if(x < 0)
10         return x + width;
11     if(x > width - 1)
12         return x - width;
13     return x;
14 }
15
16 //State Computation Kernel
17 extern "C"
18 __global__ void VOISINAGE(int envSizeX, int envSizeY, int* envData, int*
       result, int depth){
19     int tidX = blockIdx.x * blockDim.x + threadIdx.x;
20     int tidY = blockIdx.y * blockDim.y + threadIdx.y;
21
22     int temp = 0;
23
24     if(tidX < envSizeX && tidY < envSizeY){
25         int borneInfX = tidX - depth;
26         int borneSupX = tidX + depth;
27         int borneInfY = tidY - depth;
28         int borneSupY = tidY + depth;
29         for(int i = borneInfX; i <= borneSupX; i++){
30             for(int j = borneInfY; j <= borneSupY; j++){
31                 //if(!(i == tidX && j == tidY)){
32                     if(envData[convert1D(normeValue(i,envSizeX),normeValue(j,
                           envSizeY),envSizeY)] == -1){
33                         temp--;
34                     }
35                     if(envData[convert1D(normeValue(i,envSizeX),normeValue(j,
                           envSizeY),envSizeY)] == 1){
36                         temp++;
37                     }
38                 //}
39             }
40         }
41
42         result[envSizeY * tidY + tidX] = temp;
43     }
44 }
```

Algorithme B.5 : Code source, calcul des directions de fuite et d'interception en CUDA C

```
1
2  //Convert 2D coordinates into one 1D coordinate
3  __device__ int convert1D(int x, int y,int width){
4    return y * width +x;
5  }
6
7  //Normalize coordinates for infinite world
8  __device__ int normeValue(int x, int width){
9      if(x < 0) //-1
10         return width - 1;
11     if(x == width)
12         return 0;
13     return x;
14 }
15
16 __device__ int* neighborsIndexes(int i, int j, int width, int height){
17     int dir[8];
18     dir[0] = convert1D(normeValue(i+1,width), j, width);
19     dir[1] = convert1D(normeValue(i+1,width), normeValue(j+1,height),width);
20     dir[2] = convert1D(i, normeValue(j+1,height),width);
21     dir[3] = convert1D(normeValue(i-1,width), normeValue(j+1,height),width);
22     dir[4] = convert1D(normeValue(i-1,width), j, width);
23     dir[5] = convert1D(normeValue(i-1,width), normeValue(j-1,height),width);
24     dir[6] = convert1D(i, normeValue(j-1,height),width);
25     dir[7] = convert1D(normeValue(i+1,width), normeValue(j-1,height),width);
26     return dir;
27 }
28
29 //Computing heading
30 extern "C"
31 __global__ void FIELD_DIR(int width, int height, float *values, float*
       patchMax)
32 {
33         int i = blockIdx.x * blockDim.x + threadIdx.x;
34         int j = blockIdx.y * blockDim.y + threadIdx.y;
35
36        if (i < width && j < height ){
37             int k = convert1D(i,j,width);
38             int maxIndex = 0;
39             int* neighbors = neighborsIndexes(i,j,width,height);
40             float max = values[neighbors[0]];
41             for(int u=1 ; u < 8 ; u++){
42                 float current = values[neighbors[u]];
43                 if(max < current){
44                     max = current;
45                     maxIndex = u;
46                 }
47             }
48             patchMax[k] = maxIndex * 45;
49         }
50 }
```

Illustration de la simplification des comportements d'un modèle

Enfin, nous présentons le code source du comportement des entités proies dans le modèle Proie-prédateur avant (algorithme B.6) et après (algorithme B.7) l'application de notre méthode de conception. Ceci afin d'illustrer les capacités de simplification offertes par notre méthode dans certains cas et que nous avons évoqué au chapitre 8.

Algorithme B.6 : Code source, comportement avant application de la méthode

```
1  public class Prey extends Turtle {
2
3      private double life;
4      private int visionRadius = 1;
5
6      protected void activate() {
7          super.activate();
8          playRole("prey");
9          randomHeading();
10         randomLocation();
11         setColor(Color.white);
12         setNextAction("live");
13     }
14
15     public String live() {
16         final List<Predator> predatorsHere = getPatch().getTurtles(Predator.
               class);
17         if(predatorsHere.size() > 2){
18             int targetedBy = 0;
19             for (Predator predator : predatorsHere) {
20                 if(predator.getTarget() == this && ++targetedBy == 4){
21                     return null;
22                 }
23             }
24         }
25         Predator turtle = getNearestTurtle(visionRadius,Predator.class);
26         if(turtle != null){
27             setHeading(towards(turtle)+180);
28         }
29
30         wiggle(20);
31         return "live";
32     }
33 }
```

Algorithme B.7 : Code source, comportement après application de la méthode

```
public class Prey extends Turtle {

    private Environment myEnv;

    protected void activate() {
        super.activate();
        playRole("prey");
        randomHeading();
        randomLocation();
        setColor(Color.white);
        setNextAction("live");
        myEnv = (Environment) getEnvironment();
    }

    public String live() {
        final List<Predator> predatorsHere = getPatch().getTurtles(Predator.
            class);
        if(predatorsHere.size() > 2){
            return null;
        }

        // perception et changement de direction
        wiggle((int) myEnv.getPredatorHeading(this.get1DIndex())+180);

        // depot de la marque de presence dans l'environnement
        myEnv.setPreyMark(this.get1DIndex(), myEnv.getPreyMarkValue(this.
            get1DIndex()) + 1.0f);

        return "live";
    }
}
```

ANNEXE C

IMPLÉMENTATION D'UNE SOLUTION DE RENDU GRAPHIQUE DANS TURTLEKIT

Le GPGPU, comme nous l'avons vu au chapitre 2 et dans tout ce manuscrit, permet la réalisation de calculs génériques sur carte graphique. Cependant, le rôle premier de ces cartes est de gérer l'affichage et d'effectuer les calculs liés aux graphismes. Ainsi, en plus d'utiliser notre GPU pour faire du GPGPU, ne serait-il pas intéressant de l'utiliser pour réaliser aussi l'affichage du déroulement des simulations dans TurtleKit ? Ceci permettrait d'atteindre deux objectifs : (1) nous limiterions, en partie, les nombreux transferts de données entre CPU et GPU (que nous avons identifiés comme extrêmement coûteux durant nos travaux) et (2) nous utiliserions un matériel dédié à l'affichage plus rapide que les différentes solutions utilisant le CPU.

C.1 OpenGL

OpenGL (de l'anglais *Open Graphics Library*), lancé par Silicon Graphics en 1992, se présente sous la forme d'une librairie offrant un ensemble normalisé de fonctions de calcul d'images 2D ou 3D. Regroupant environ 250 fonctions différentes, OpenGL permet l'affichage de scènes tridimensionnelles complexes à partir de simples primitives géométriques. En effet, OpenGL permet à un programme de déclarer la géométrie d'objets sous forme de points, de vecteurs, de polygones, de bitmaps et de textures. OpenGL effectue ensuite des calculs de projection en vue de déterminer l'image à l'écran, en tenant compte de la distance, de l'orientation, des ombres, de la transparence et du cadrage.

Cette interface de programmation est disponible sur de nombreuses plates-formes et est utilisée par une grande majorité d'applications scientifiques, industrielles, artistiques 3D et certaines applications 2D vectorielles. Cette bibliothèque est également utilisée dans l'industrie du jeu vidéo où elle est souvent en rivalité avec la bibliothèque de Microsoft : *Direct3D*. Une version nommée OpenGL ES a été conçue spécifiquement pour les applications embarquées (téléphones portables, agenda de poche, consoles de jeux, etc.).

En fait, si OpenGL est 100 % portable avec tous les systèmes d'exploitation, c'est tout simplement parce que ce n'est pas lui qui se charge de l'affichage. En fait, OpenGL est obligé de s'appuyer sur un système d'exploitation ayant une interface graphique : il sert d'intermédiaire entre le programme et le système d'exploitation, en adaptant et traduisant les informations données par le programme et en les envoyant au système d'exploitation.

Pour comprendre plus en détails comment OpenGL fonctionne, décrivons les étapes nécessaire à l'initialisation d'OpenGL : un environnement graphique dans le système d'exploitation (c'est-à-dire une fenêtre, dans le cas de Windows) doit être initialisé. Cette fenêtre sera définie de manière unique par un *Device Context*, une sorte d'interface qui permet au programme d'accéder à cette fenêtre et de lui faire subir toutes les opérations d'affichage possibles (affichage de pixels, etc.). Ensuite, le travail sera d'associer à ce *Device Context* un *Rendering Context*, qui va permettre à OpenGL de s'adresser à la fenêtre pour lui envoyer ses ordres de dessin (la figure C.1 illustre ce processus).

FIGURE C.1 – Processus d'utilisation d'OpenGL.

OpenGL se contente d'adapter les informations et de les transférer au système d'exploitation qui s'occupera par la suite de les afficher. La seule contrainte pour le système d'exploitation est de réussir à communiquer avec OpenGL. Ce dialogue est rendu possible par la mise à disposition de *drivers* disponibles dès l'installation et qui s'occupent de la traduction des données OpengGL en données compréhensibles pour la carte graphique. Et c'est là qu'intervient l'accélération matérielle car OpenGL est désormais accéléré matériellement par toutes les cartes graphiques actuelles (grâce aux drivers fournis par les constructeurs de cartes).

C.2 L'affichage dans TurtleKit

TurtleKit met à disposition des utilisateurs des *Viewers* qui permettent de visualiser l'évolution de la simulation (voir figure C.2). Au nombre de deux, ces *viewers* utilisent la bibliothèque *Swing* qui est une bibliothèque graphique pour le langage de programmation Java. Elle a la possibilité de créer des interfaces graphiques identiques quel que soit le système d'exploitation sous-jacent. Pour ce faire, Swing hérite d'AWT (de l'anglais *Abstract Window Toolkit*) qui est une bibliothèque graphique pour Java employant les composants natifs du système d'exploitation, alors que Swing utilise des composants en pur Java. Cette abstraction offerte par Swing augmente sa portabilité mais limite ses performances.

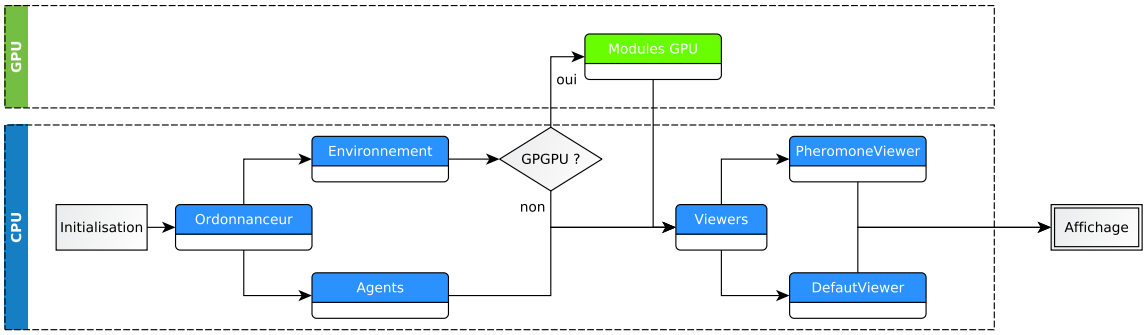


FIGURE C.2 – Affichage des simulations dans TurtleKit.

TurtleKit étant maintenant capable d'effectuer du calcul intensif grâce au GPGPU, il peut être intéressant d'optimiser l'affichage des simulations afin de ne pas perdre le temps gagné grâce au GPGPU à cause de l'affichage. Nous proposons donc d'utiliser OpenGL pour réaliser l'affichage dans TurtleKit via la librairie Java *JoGL* [1].

C.2.1 Intégration d'OpenGL dans TurtleKit

Le choix a été d'intégrer OpenGL via l'utilisation d'un moteur de jeu spécialement conçu pour le développement 3D : *jMonkeyEngine*. Ce moteur permet d'obtenir une gestion de la visualisation plus poussée avec, entre autre, la possibilité de se déplacer dans l'espace de simulation avec le clavier et la souris. De plus, il est capable de gérer un espace en 3 dimensions, limitant donc

1. JoGL est une librairie qui permet d'utiliser les primitives OpenGL directement dans Java (`http://jogamp.org/jogl/www/`).

les modifications à effectuer au cas où TurtleKit décide de prendre en compte ce genre de simulations à l'avenir. Enfin, il possède un code source ouvert nous autorisant à le modifier selon notre convenance.

L'affichage fonctionne, pour les simulations en 2 dimensions, grâce à un ensemble de grilles 2D qui vont être activées en fonction de ce que l'on décide d'afficher à l'écran. En effet, chacune de ces grilles se sert de primitives OpenGL fournies par la bibliothèque JoGL pour afficher une partie bien précise de la simulation : une grille pour les agents, une grille pour les phéromones, une grille pour l'environnement, etc. Grâce à cette construction sous forme de couches (voir figure C.3), il est très facile d'étendre ou de modifier le *viewer 3D* mis en place.

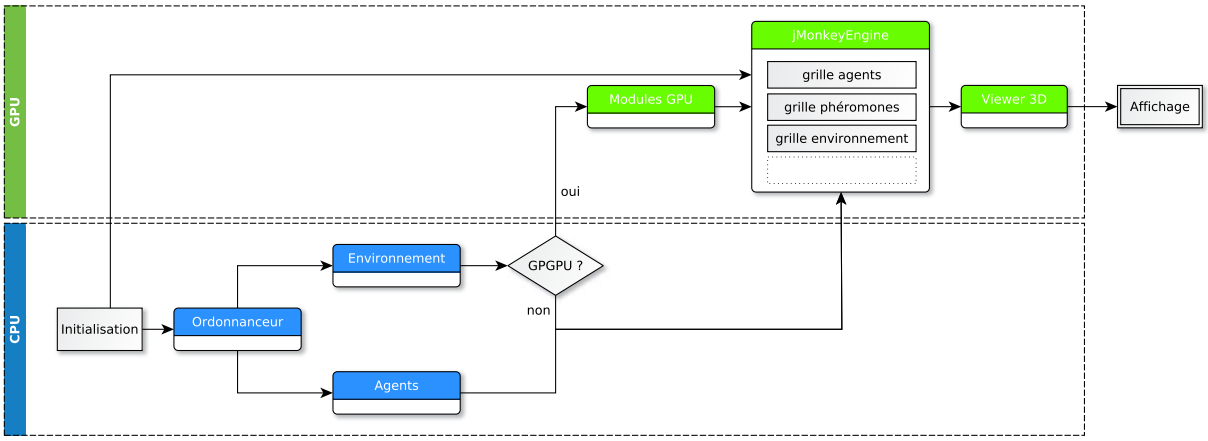


FIGURE C.3 – Affichage des simulations dans TurtleKit avec OpenGL.

En plus de ces grilles, un ensemble de caméras et de dispositifs d'interaction (par exemple avec le clavier et souris) sont intégrés grâce au moteur *jMonkeyEngine*. Des fonctions supplémentaires pourront donc être facilement implémentées comme la gestion de la lumière, de la transparence ou encore l'utilisation de *shaders* pour la réalisation d'affichages plus complexes.

Maintenant que l'affichage se fait par OpenGL et donc par le GPU, il peut être intéressant d'optimiser encore ce processus d'affichage en utilisant l'interopérabilité qui existe entre CUDA (les modules GPU) et OpenGL afin de limiter toujours plus les différents transferts entre CPU et GPU.

C.2.2 Interopérabilité entre CUDA et OpenGL

Des mécanismes d'interopérabilité entre CUDA et OpenGL existent. Ces mécanismes permettent d'effectuer en OpenGL le rendu et l'affichage de données calculées sur GPU avec CUDA. Ces données sont présentes dans la mémoire du GPU et leur affichage ne nécessite donc pas de transferts entre le CPU et le GPU. Il existe de nombreuses techniques permettant d'obtenir cette interopérabilité, nous présentons ici celle basée sur l'utilisation des PBO (de l'anglais *Pixel Buffer Object*).

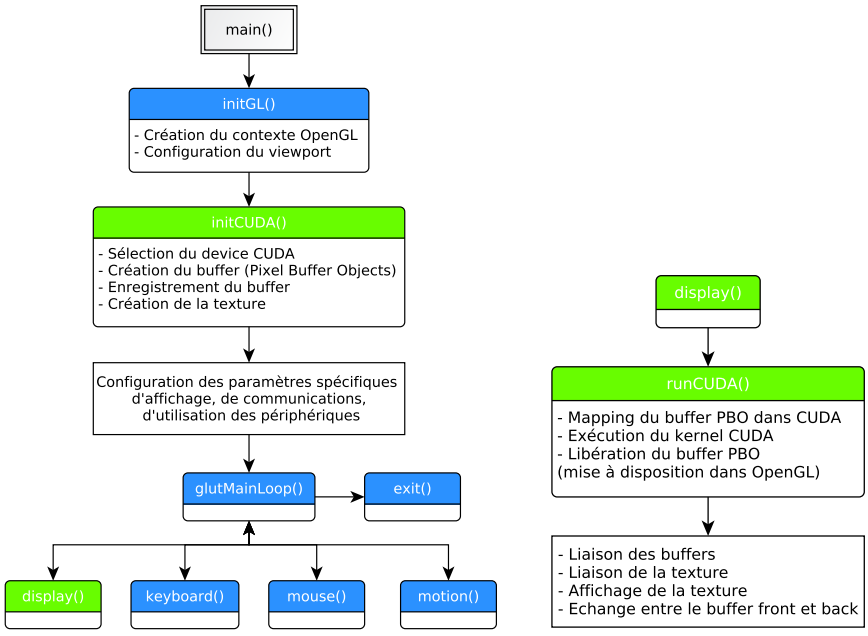
Utiliser OpenGL, via les PBO, pour réaliser un affichage de données calculées par CUDA est un processus qui peut se diviser en deux phases distinctes, elles mêmes divisées en plusieurs étapes. La première consiste à créer, initialiser et configurer tous les objets nécessaires à cette interopérabilité :

1. Création et initialisation d'un contexte OpenGL ;
2. Création et initialisation d'un contexte CUDA ;
3. Configuration du *GLviewport* et de son système de coordonnées ;
4. Génération d'un ou plusieurs *GL buffers* ;
5. Partage des *buffers* avec CUDA.

Une fois tous ces éléments configurés, nous pouvons les utiliser pour réaliser un affichage via OpenGL :

1. Allocation des *buffers* OpenGL correspondant à la taille de l'image à afficher ;
2. Allocation des *textures* OpenGL (ayant la même taille que les *buffers*).
3. Mapping des *buffers* OpenGL dans la mémoire CUDA ;
4. Écriture des données générées par CUDA dans le *buffers* OpenGL ;
5. Libération du *buffers* OpenGL ;
6. Liaison des textures aux *buffers* OpenGL
7. Création d'un carré spécifiant les coordonnées de la texture correspondant aux 4 coins de l'image à afficher.
8. Affichage (et échange entre les *buffers* front et back).

L'image C.4 illustre ce processus d'affichage.



(a) Initialisation (b) Affichage

FIGURE C.4 – Intéropérabilité entre CUDA et OpenGL.

PUBLICATIONS

Revue d'Intelligence Artificielle

[Hermellin and Michel, 2016b] Hermellin, E. et Michel, F. (2016). Expérimentation du principe de délégation GPU pour la simulation multiagent. les boids de Reynolds comme cas d'étude. *Revue d'Intelligence Artificielle*, 30(1-2) :109–132.

[Hermellin et al., 2015] Hermellin, E., Michel, F., et Ferber, J. (2015). État de l'art sur les simulations multi-agents et le GPGPU. *Revue d'Intelligence Artificielle*, 29(3-4) :425–451.

Conference on Autonomous Agents and Multiagent Systems (AAMAS)

[Hermellin and Michel, 2016c] Hermellin, E. et Michel, F. (2016). GPU delegation : Toward a generic approach for developping MABS using GPU programming. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, AAMAS '16*, pages 1249–1258, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Multi-Agent-Based Simulation (AAMAS - MABS)

[Hermellin and Michel, 2016a] Hermellin, E. et Michel, F. (2016). Defining a methodology based on GPU delegation for developing MABS using GPGPU (currently undergoing a second review process before publication in the forthcoming post-proceedings : MABS XVII). In Antunes, L. and Nardin, L. G., editors, 17th Multi-Agent-Based Simulation (MABS) workshop, pages 68–83. @AAMAS 2016.

[Hermellin and Michel, 2016d] Hermellin, E. et Michel, F. (2016). GPU environmental delegation of agent perceptions : Application to Reynolds's boids. In Gaudou, B. and Sichman, S. J., editors, *Multi-Agent Based Simulation XVI*, volume 9568 of Lecture Notes in Computer Science, pages 71–86. Springer International Publishing.

Journées Francophones sur les Systèmes Multi-Agents (JFSMA)

[Hermellin and Michel, 2016e] Hermellin, E. et Michel, F. (2016). Méthodologie pour la modélisation et l'implémentation de simulations multi-agents utilisant le GPGPU. In Michel, F. and Saunier, J., editors, *Systèmes Multi-Agents et simulation - Vingt-quatrièmes journées francophones sur les systèmes multi-agents, JFSMA 16, Saint-Martin-du-Vivier (Rouen), France, Octobre 5-7, 2016.*, pages 107–116. Cépaduès Éditions.

[Hermellin and Michel, 2015] Hermellin, E. et Michel, F. (2015). Délégation GPU des perceptions agents : Application aux boids de Reynolds. In Vercouter, L. and Picard, G., editors, *Environnements socio-techniques - JFSMA 15 - Vingt-troisièmes Journées Francophones sur les Systèmes Multi-Agents, Rennes, France, June 30th-July 1st, 2015*, pages 185–194. Cépaduès Éditions.

[Hermellin et al., 2014] Hermellin, E., Michel, F., et Ferber, J. (2014). Systèmes multi-agents et GPGPU : état des lieux et directions pour l'avenir. In Courdier, R. and Jamont, J., editors, *Principe de Parcimonie - JFSMA 14 - Vingt-deuxièmes Journées Francophones sur les Systèmes Multi-Agents, Loriol-sur-Drôme, France, Octobre 8-10, 2014*, pages 97–106. Cépaduès Éditions.

Complex Networks and Social Computation (PAAMS - CNSC)

[Hermellin and Michel, 2016f] Hermellin, E. et Michel, F. (2016). Overview of case studies on adapting MABS models to GPU programming. In Bajo, J., Escalona, M. J., Giroux, S., Hoffa-Dabrowska, P., Julián, V., Novais, P., Pi, N. S., Unland, R., and Silveira, R. A., editors, *Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection : International Workshops of PAAMS 2016, Sevilla, Spain, June 1-3, 2016*. Proceedings, volume 616 of Communications in Computer and Information Science, pages 125–136. Springer International Publishing.

BIBLIOGRAPHIE

[Aaby et al., 2010] Aaby, B. G., Perumalla, K. S., and 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*, SIMUTools '10, pages 29 :1–29 :10, ICST, Brussels, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
(cf. page 40)

[Abouaissa et al., 2015] Abouaissa, H., Yoann, K., and Morvan, G. (2015). Modélisation hybride dynamique de flux de trafic. Research report, LGI2A, Université d'Artois. version française étendue de http ://arxiv.org/abs/1401.6773.
(cf. page 38, 43)

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM.
(cf. page 22)

[Axelrod, 1997] Axelrod, R. (1997). *Simulating Social Phenomena*, chapter Advancing the Art of Simulation in the Social Sciences, pages 21–40. Springer Berlin Heidelberg, Berlin, Heidelberg.
(cf. page 12)

[Badeig and Balbo, 2012] Badeig, F. and Balbo, F. (2012). Définition d'un cadre de conception et d'exécution pour la simulation multi-agent. *Revue d'Intelligence Artificielle*, 26(3) :255–280.
(cf. page 52)

[Bazzan et al., 1999] Bazzan, A. L. C., Wahle, J., and Klügl, F. (1999). Agents in traffic modelling — from reactive to social behaviour. In Burgard, W., Cremers, A., and Cristaller, T., editors, *KI-99 : Advances in Artificial Intelligence*, volume 1701 of *Lecture Notes in Computer Science*, pages 303–306. Springer Berlin Heidelberg.
(cf. page 12)

[Beurier et al., 2003] Beurier, G., Simonin, O., and Ferber, J. (2003). Un modèle de système multi-agent pour l'émergence multi-niveau. *Technique et Science Informatiques*, 22(4) :235–247.
(cf. page 54, 55)

[Bleiweiss, 2008] Bleiweiss, A. (2008). GPU accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 65–74, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.
(cf. page 39)

[Bleiweiss, 2009] Bleiweiss, A. (2009). Multi agent navigation on the GPU. *Games Developpement Conference*.
(cf. page 39, 42, 44, 45)

[Bordini et al., 2007] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons.
(cf. page 14)

[Bourgoin, 2013] Bourgoin, M. (2013). *Abstractions performantes pour cartes graphiques*. PhD thesis, Université Pierre et Marie Curie.
(cf. page 23, 25, 32)

[Bourgoin et al., 2014] Bourgoin, M., Chailloux, E., and Lamotte, J.-L. (2014). Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4) :583–600.
(cf. page 33, 44, 46, 51, 89)

[Bourjot et al., 2002] Bourjot, C., Chevrier, V., and Thomas, V. (2002). How social spiders inspired an approach to region detection. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 426–433. ACM.
(cf. page 12)

[Bousquet and Page, 2004] Bousquet, F. and Page, C. L. (2004). Multi-agent simulations and ecosystem management : a review. *Ecological Modelling*, 176(3–4) :313 – 332.
(cf. page 12)

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47(1–3) :139 – 159.
(cf. page 11)

[Caggianese and Erra, 2012] Caggianese, G. and Erra, U. (2012). GPU Accelerated Multi-agent Path Planning Based on Grid Space Decomposition. In *Proceedings of the International Conference on Computational Science*, volume 9, pages 1847–1856. Elsevier.
(cf. page 39)

[Camus et al., 2012] Camus, B., Siebert, J., Bourjot, C., and Chevrier, V. (2012). Modélisation multi-niveaux dans AA4MM. *CoRR*, abs/1210.5936.
(cf. page 15)

[Chang et al., 2005] Chang, P. H., Chen, K.-T., Chien, Y.-H., Kao, E., and Soo, V.-W. (2005). From reality to mind : A cognitive middle layer of environment concepts for believable agents. In Weyns, D., Parunak, H. V. D., and Michel, F., editors, *Environments for Multi-Agent Systems, First International Workshop, E4MAS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3374 of *LNAI*, pages 57–73. Springer.
(cf. page 41)

[Che et al., 2008] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10) :1370–1380.
(cf. page 24, 32, 33)

[Chen et al., 2015] Chen, D., Wang, L., Zomaya, A., Dou, M., Chen, J., Deng, Z., and Hariri, S. (2015). Parallel simulation of complex evacuation scenarios with adaptive agent models. *Parallel and Distributed Systems, IEEE Transactions on*, 26(3) :847–857.
(cf. page 38, 42)

[Chevrier and Fatès, 2008] Chevrier, V. and Fatès, N. (2008). Multi-agent Systems as Discrete Dynamical Systems : Influences and Reactions as a Modelling Principle. Research report.
(cf. page 110)

[Coakley et al., 2016] Coakley, S., Richmond, P., Gheorghe, M., Chin, S., Worth, D., Holcombe, M., and Greenough, C. (2016). *Intelligent Agents in Data-intensive Computing*, chapter Large-Scale Simulations with FLAME, pages 123–142. Springer International Publishing, Cham.
(cf. page 36, 43, 86)

[Coakley et al., 2006] Coakley, S., Smallwood, R., and Holcombe, M. (2006). Using x-machines as a formal basis for describing agents in agent-based modelling. *Proceedings of 2006 Spring Simulation Multiconference*, pages 33–40.
(cf. page 36)

[Collier and North, 2012] Collier, N. and North, M. (2012). *Repast HPC : A Platform for Large-Scale Agent-Based Modeling*, pages 81–109. John Wiley and Sons, Inc.
(cf. page 33)

[Collier and North, 2013] Collier, N. T. and North, M. J. (2013). Parallel agent-based simulation with repast for high performance computing. *Simulation*, 89(10) :1215–1235.
(cf. page 14, 15)

[Collier et al., 2015] Collier, N. T., Ozik, J., and Macal, C. M. (2015). Large-scale agent-based modeling with repast HPC : A case study in parallelizing an agent-based model. In Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Requena, M. E. G., Scarano, V., Varbanescu, A. L., Scott, S. L., Lankes, S., Weidendorfer, J., and Alexander, M., editors, *Euro-Par 2015 : Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, volume 9523 of *Lecture Notes in Computer Science*, pages 454–465. Springer.
(cf. page 15)

[Conte et al., 1998] Conte, R., Gilbert, N., and Sichman, J. S. (1998). *MAS and Social Simulation : A Suitable Commitment*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg.
(cf. page 12)

[Cook, 2013] Cook, S. (2013). *CUDA Programming : A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.
(cf. page V)

[Demazeau, 1995] Demazeau, Y. (1995). From interactions to collective behaviour in agent-based systems. In *Proceedings of the 1st. European Conference on Cognitive Science. Saint-Malo*, pages 117–132.
(cf. page 10)

[Demeulemeester et al., 2011] Demeulemeester, A., Hollemeersch, C.-F., Mees, P., Pieters, B., Lambert, P., and Van de Walle, R. (2011). Hybrid Path Planning for Massive Crowd Simulation on the GPU. In Allbeck, J. and Faloutsos, P., editors, *Motion in Games*, volume 7060 of *Lecture Notes in Computer Science*, pages 304–315. Springer Berlin Heidelberg.
(cf. page 39, 44, 45)

[dos Santos et al., 2012] dos Santos, L., Gonzales Clua, E., and Bernardini, F. (2012). A Parallel Fipa Architecture Based on GPU for Games and Real Time Simulations. In Herrlich, M., Malaka, R., and Masuch, M., editors, *Entertainment Computing - ICEC 2012*, volume 7522 of *Lecture Notes in Computer Science*, pages 306–317. Springer Berlin Heidelberg.
(cf. page 39)

[Drogoul, 1993] Drogoul, A. (1993). *De la simulation multi-agent à la résolution collective de problèmes. Une étude De l'émergence de structures d'organisation dans les systèmes multi-agents*. PhD thesis, Université Paris 6.
(cf. page 11)

[D'Souza et al., 2009] D'Souza, R. M., Lysenko, M., Marino, S., and 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*, SpringSim '09, pages 21 :1–21 :12, San Diego, CA, USA. Society for Computer Simulation International.
(cf. page 44)

[D'Souza et al., 2007] D'Souza, R. M., Lysenko, M., and Rahmani, K. (2007). SugarScape on steroids : simulating over a million agents at interactive rates. *Proceedings of Agent 2007 conference*.
(cf. page 35, 42, 43)

[Epstein and Axtell, 1996] Epstein, J. M. and Axtell, R. (1996). *Growing Artificial Societies : Social Science from the Bottom Up*. The Brookings Institution, Washington, DC, USA.
(cf. page 12)

[Erceau and Ferber, 1991] Erceau, J. and Ferber, J. (1991). L'intelligence artificielle distribuée. *La recherche*, (233) :750–758.
(cf. page 9)

[Erra et al., 2009] Erra, U., Frola, B., Scarano, V., and Couzin, I. (2009). 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*, pages 51–58.
(cf. page 37, 42)

[Fatès and Chevrier, 2010] Fatès, N. and Chevrier, V. (2010). How important are updating schemes in multi-agent systems ? an illustration on a multi-turmite model. In van der Hoek, W., Kaminka, G. A., Lespérance, Y., Luck, M., and Sen, S., editors, *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3*, pages 533–540. IFAAMAS.
(cf. page 110)

[Ferber, 1995] Ferber, J. (1995). *Les Systèmes multi-agents : vers une intelligence collective*. I.I.A. Informatique intelligence artificielle. InterEditions.
(cf. page vii, 10, 11, 16)

[Fischer et al., 2009] Fischer, L. G., Silveira, R., and 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*, pages 101–110. IEEE Computer Society.
(cf. page 39, 42, 44, 45)

[Fishwick, 1994] Fishwick, P. A. (1994). Simulation model design. In *Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 173–175, San Diego, CA, USA. Society for Computer Simulation International.
(cf. page 12)

[Flynn, 1972] Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960.
(cf. page 21)

[Galland et al., 2014] Galland, S., Gaud, N., Rodriguez, S., Balbo, F., Picard, G., and Boissier, O. (2014). Contextualiser l'interaction entre agents en combinant dimensions sociale et physique au sein de l'environnement. In *JFSMA 2014. Systèmes Multi-Agents. Principe de Parcimonie*, pages pp 65–74, Loriol-sur-Drôme, France. Cepadues.
(cf. page 52)

[Gardner, 1970] Gardner, M. (1970). Mathematical games : The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4) :120–123.
(cf. page 72)

[Grignard et al., 2013] Grignard, A., Taillandier, P., Gaudou, B., Vo, D., Huynh, N., and Drogoul, A. (2013). GAMA 1.6 : Advancing the Art of Complex Agent-Based Modeling and Simulation. In Boella, G., Elkind, E., Savarimuthu, B., Dignum, F., and Purvis, M., editors, *PRIMA 2013 : Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg.
(cf. page 14, 60)

[Gustafson, 1988] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5) :532–533.
(cf. page 22)

[Gutknecht and Ferber, 2001] Gutknecht, O. and Ferber, J. (2001). MadKit : a generic multi-agent platform. In *Agents*, pages 78–79.
(cf. page 14, 54, III)

[Hamidouche et al., 2009] Hamidouche, K., Cappello, F., and Etiemble, D. (2009). Comparaison de MPI, OpenMP et MPI+ OpenMP sur un nœud multiprocesseur multicœurs AMD à mémoire partagée. *Rencontres francophones du Parallélisme (RenPar'19), Toulouse, France*, page 8.
(cf. page 23)

[Hayes-Roth, 1985] Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26(3) :251 – 321.
(cf. page 9)

[Haynes and Sen, 1996] Haynes, T. and Sen, S. (1996). Evolving behavioral strategies in predators and prey. In Weiß, G. and Sen, S., editors, *Adaption and Learning in Multi-Agent Systems*, volume 1042 of *Lecture Notes in Computer Science*, pages 113–126. Springer Berlin Heidelberg.
(cf. page 12)

[Helbing et al., 2002] Helbing, D., Farkas, I. J., Molnar, P., and Vicsek, T. (2002). Simulation of pedestrian crowds in normal and evacuation situations. *Pedestrian and Evacuation Dynamics (Berlin, Germany)*, pages 21–58.
(cf. page 37)

[Hermellin and Michel, 2015] Hermellin, E. and Michel, F. (2015). Délégation GPU des perceptions agents : Application aux boids de reynolds. In Vercouter, L. and Picard, G., editors, *Environnements socio-techniques - JFSMA 15 - Vingt-troisièmes Journées Francophones sur les Systèmes Multi-Agents, Rennes, France, June 30th-July 1st, 2015*, pages 185–194. Cépaduès Éditions.
(cf. page 52, 59, 103)

[Hermellin and Michel, 2016a] Hermellin, E. and Michel, F. (2016a). Defining a methodology based on gpu delegation for developing mabs using gpgpu (currently undergoing a second review process before publication in the forthcoming post-proceedings : Mabs xvii). In Antunes, L. and Nardin, L. G., editors, *17th Multi-Agent-Based Simulation (MABS) workshop*, pages 68–83. @AAMAS 2016.
(cf. page 85, 86)

[Hermellin and Michel, 2016b] Hermellin, E. and Michel, F. (2016b). Expérimentation du principe de délégation GPU pour la simulation multiagent. les boids de reynolds comme cas d'étude. *Revue d'Intelligence Artificielle*, 30(1-2) :109–132.
(cf. page 52, 59, 103)

[Hermellin and Michel, 2016c] Hermellin, E. and Michel, F. (2016c). Gpu delegation : Toward a generic approach for developping mabs using gpu programming. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 1249–1258, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.
(cf. page 71, 103)

[Hermellin and Michel, 2016d] Hermellin, E. and Michel, F. (2016d). Gpu environmental delegation of agent perceptions : Application to reynolds's boids. In Gaudou, B. and Sichman, S. J., editors, *Multi-Agent Based Simulation XVI*, volume 9568 of *Lecture Notes in Computer Science*, pages 71–86. Springer International Publishing.
(cf. page 52, 59, 103)

[Hermellin and Michel, 2016e] Hermellin, E. and Michel, F. (2016e). Méthodologie pour la modélisation et l'implémentation de simulations multi-agents utilisant le GPGPU. In Michel, F. and Saunier, J., editors, *Systèmes Multi-Agents et simulation - Vingt-quatrièmes journées francophones sur les systèmes multi-agents, JFSMA 16, Saint-Martin-du-Vivier (Rouen), France, Octobre 5-7, 2016.*, pages 107–116. Cépaduès Éditions.
(cf. page 86)

[Hermellin and Michel, 2016f] Hermellin, E. and Michel, F. (2016f). Overview of case studies on adapting mabs models to gpu programming. In Bajo, J., Escalona, M. J., Giroux, S., Hoffa-Dabrowska, P., Julián, V., Novais, P., Pi, N. S., Unland, R., and Silveira, R. A., editors, *Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection : International Workshops of PAAMS 2016, Sevilla, Spain, June 1-3, 2016. Proceedings*, volume 616 of *Communications in Computer and Information Science*, pages 125–136. Springer International Publishing.
(cf. page 71, 103)

[Hermellin et al., 2014] Hermellin, E., Michel, F., and Ferber, J. (2014). Systèmes multi-agents et GPGPU : état des lieux et directions pour l'avenir. In Courdier, R. and Jamont, J., editors, *Principe de Parcimonie - JFSMA 14 - Vingt-deuxièmes Journées Francophones sur les Systèmes Multi-Agents, Loriol-sur-Drôme, France, Octobre 8-10, 2014*, pages 97–106. Cépaduès Éditions.
(cf. page 33, 102)

[Hermellin et al., 2015] Hermellin, E., Michel, F., and Ferber, J. (2015). État de l'art sur les simulations multi-agents et le GPGPU. *Revue d'Intelligence Artificielle*, 29(3-4) :425–451.
(cf. page 33, 102)

[Hewitt, 1977] Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3) :323 – 364.
(cf. page 9)

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
(cf. page 9)

[Ho et al., 2015] Ho, N., Thoai, N., and Wong, W. (2015). Multi-agent simulation on multiple GPUs. *Simulation Modelling Practice and Theory*, 57 :118 – 132.
(cf. page 14, 15, 34, 41, 42, 43)

[Holk et al., 2011] Holk, E., Byrd, W. E., Mahajan, N., Willcock, J., Chauhan, A., and 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*, volume 22 of *Advances in Parallel Computing*, pages 297–304. IOS Press.
(cf. page 44)

[Husselmann and Hawick, 2011] Husselmann, A. V. and 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*, pages 100–107. IASTED.
(cf. page 37, 42, 44)

[Karmakharm and Richmond, 2012] Karmakharm, T. and 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*, pages 41–44. Eurographics Association.
(cf. page 36)

[Karmakharm et al., 2010] Karmakharm, T., Richmond, P., and 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*, pages 67–74. Eurographics Association.
(cf. page 36, 44)

[Karp and Flatt, 1990] Karp, A. H. and Flatt, H. P. (1990). Measuring parallel processor performance. *Commun. ACM*, 33(5) :539–543.
(cf. page 22)

[Kravari and Bassiliades, 2015] Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1) :11.
(cf. page v, 13, 14)

[Kubera et al., 2009] Kubera, Y., Mathieu, P., and Picault, S. (2009). How to avoid biases in reactive simulations. In Demazeau, Y., Pavón, J., Corchado, J. M., and Bajo, J., editors, *7th International Conference on Practical Applications of Agents and Multi-Agent Systems, PAAMS 2009, Salamanca, Spain, 25-27 March 2009*, volume 55 of *Advances in Intelligent and Soft Computing*, pages 100–109. Springer.
(cf. page 15)

[Kubera et al., 2011] Kubera, Y., Mathieu, P., and Picault, S. (2011). IODA : an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3) :303–343.
(cf. page 15)

[Laville et al., 2014] Laville, G., Mazouzi, K., Lang, C., Marilleau, N., Herrmann, B., and Philippe, L. (2014). MCMAS : A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation. In an Mey, D., Alexander, M., Bientinesi, P., Cannataro, M., Clauss, C., Costan, A., Kecskemeti, G., Morin, C., Ricci, L., Sahuquillo, J., Schulz, M., Scarano, V., Scott, S., and Weidendorfer, J., editors, *Euro-Par 2013 : Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 544–554. Springer Berlin Heidelberg.
(cf. page 41, 42, 43, 44, 45)

[Laville et al., 2012] Laville, G., Mazouzi, K., Lang, C., Marilleau, N., and Philippe, L. (2012). 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. Springer Berlin Heidelberg.
(cf. page 41, 42, 43, 44, 45, 86)

[Lesser and Corkill, 1983] Lesser, V. R. and Corkill, D. D. (1983). The distributed vehicle monitoring testbed : A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3) :15–33.
(cf. page 9)

[Li et al., 2009] Li, H., Kolpas, A., Petzold, L., and Moehlis, J. (2009). Parallel Simulation for a Fish Schooling Model on a General-purpose Graphics Processing Unit. *Concurrency and Computation : Practice and Experience*, 21(6) :725–737.
(cf. page 37)

[Li et al., 2014] Li, X., Cai, W., and Turner, S. (2014). Efficient neighbor searching for agent-based simulation on gpu. In *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on*, pages 87–96.
(cf. page 39)

[Li et al., 2016] Li, X., Cai, W., and Turner, S. J. (2016). Supporting efficient execution of continuous space agent-based simulation on gpu. *Concurrency and Computation : Practice and Experience*, pages 1–20.
(cf. page 39)

[Lotka, 1925] Lotka, A. J. (1925). *Elements of Physical Biology*. Williams and Wilkins Company.
(cf. page 93)

[Luke et al., 2005] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). MASON : A Multiagent Simulation Environment. *Simulation*, 81(7) :517–527.
(cf. page 14, 33, 41, 60)

[Lysenko and D'Souza, 2008] Lysenko, M. and D'Souza, R. M. (2008). A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation*, 11(4) :10.
(cf. page 35, 42, 43, 45)

[Maes, 1990] Maes, P. (1990). Situated agents can have goals. *Robotics and Autonomous Systems*, 6(1–2) :49 – 70. Designing Autonomous Agents.
(cf. page 11)

[Malcolm and Smithers, 1990] Malcolm, C. and Smithers, T. (1990). Symbol grounding via a hybrid architecture in an autonomous assembly system. *Robotics and Autonomous Systems*, 6(1–2) :123 – 144. Designing Autonomous Agents.
(cf. page 11)

[Mandiau et al., 2008] Mandiau, R., Champion, A., Auberlet, J.-M., Espié, S., and Kolski, C. (2008). Behaviour based on decision matrices for a coordination between agents in a urban traffic simulation. *Applied Intelligence*, 28(2) :121–138.
(cf. page 12)

[McCabe et al., 2016] McCabe, S., Brearcliffe, D., Froncek, P., Hansen, M., Kane, V., Taghawi-Nejad, D., , and Axtell, R. L. (2016). A comparison of languages and frameworks for the parallelization of a simple agent model. In Antunes, L. and Nardin, L. G., editors, *Multi-Agent Based Simulation XVII, International Workshop, MABS 2016, Singapore, May 10, 2016*, pages –. Workshop Pre-Proceedings, to be published.
(cf. page 34)

[Michel, 2004] Michel, F. (2004). *Formalisme, outils et éléments méthodologiques pour la modélisation et la simulation multi-agents*. PhD thesis, Université Montpellier II.
(cf. page vii, 13, 15)

[Michel, 2007] Michel, F. (2007). Le modèle irm4s, de l'utilisation des notions d'influence et de réaction pour la simulation de systèmes multi-agents. *Revue d'Intelligence Artificielle*, 21(5-6) :757–779.
(cf. page 54, 110)

[Michel, 2013a] Michel, F. (2013a). Intégration du calcul sur GPU dans la plate-forme de simulation multi-agent générique TurtleKit 3. In Hassas, S. and Morge, M., editors, *Dynamiques, couplages et visions intégratives - JFSMA 13 - Vingt-et-unièmes journées francophones sur les systèmes multi-agents, Lille, France, Juillet 3-5, 2013*, pages 189–198. Cepadues Editions.
(cf. page 51, 103)

[Michel, 2013b] Michel, F. (2013b). 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*, 30(6) :703–715.
(cf. page 41, 42, 43, 45)

[Michel, 2014] Michel, F. (2014). Délégation GPU des perceptions agents : intégration itérative et modulaire du GPGPU dans les simulations multi-agents. Application sur la plate-forme TurtleKit 3. *Revue d'Intelligence Artificielle*, 28(4) :485–510.
(cf. page 51, 53, 54, 56, 58, 69)

[Michel, 2015] Michel, F. (2015). *Approches environnement-centrées pour la simulation de systèmes multi-agents. Pour un déplacement de la complexité des agents vers l'environnement*. PhD thesis, Université de Montpellier.
(cf. page 14, 109, 110)

[Michel et al., 2005] Michel, F., Beurier, G., and Ferber, J. (2005). The TurtleKit Simulation Platform : Application to Complex Systems. In Akono, A., Tonyé, E., Dipanda, A., and Yétongnon, K., editors, *Workshops Sessions of the Proceedings of the 1st International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2005, November 27 - December 1, 2005, Yaoundé, Cameroon*, pages 122–128. IEEE.
(cf. page 54, III)

[Michel et al., 2009] Michel, F., Ferber, J., and Drogoul, A. (2009). Multi-Agent Systems and Simulation : a Survey From the Agents Community's Perspective. In Adelinde Uhrmacher and Danny Weyns, editors, *Multi-Agent Systems : Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 3–52. CRC Press - Taylor & Francis.
(cf. page 12, 13, 16, 37, 53)

[Morvan, 2012] Morvan, G. (2012). Multi-level agent-based modeling bibliography. *arXiv preprint arXiv :1205.0561*.
(cf. page 15)

[North et al., 2007] North, M., Tatara, E., Collier, N., and Ozik, J. (2007). Visual agent-based model development with Repast Simphony. In *Agent 2007 Conference on Complex Interaction and Social Emergence*, pages 173–192, Argonne, IL, USA. Argonne National Laboratory.
(cf. page 14, 33, 35, 61)

[Nvidia, 2015] Nvidia (2015). Cuda c programming guide. Technical report, Nvidia Corporation.
(cf. page 30)

[Owens et al., 2007] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1) :80–113.
(cf. page 24, 25, 30, 32, 34)

[Pallickara and Pierce, 2008] Pallickara, S. L. and Pierce, M. (2008). Swarm : Scheduling large-scale jobs over the loosely-coupled hpc clusters. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 285–292.
(cf. page 14)

[Parunak, 1997] Parunak, H. V. D. (1997). "Go to the ant" : Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75(0) :69–101.
(cf. page 11, 12)

[Parunak et al., 1998] Parunak, H. V. D., Savit, R., and Riolo, R. L. (1998). *Multi-Agent Systems and Agent-Based Simulation : First International Workshop, MABS '98, Paris, France, July 4-6, 1998. Proceedings*, chapter Agent-Based Modeling vs. Equation-Based Modeling : A Case Study and Users' Guide, pages 10–25. Springer Berlin Heidelberg, Berlin, Heidelberg.
(cf. page 13)

[Passos et al., 2008] Passos, E., Joselli, M., and 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*, pages 81–86.
(cf. page 37, 43)

[Pavlov and Müller, 2013] Pavlov, R. and Müller, J. (2013). Multi-Agent Systems Meet GPU : Deploying Agent-Based Architectures on Graphics Processors. In Camarinha-Matos, L., Tomic, S., and Graça, P., editors, *Technological Innovation for the Internet of Things*, volume 394 of *IFIP Advances in Information and Communication Technology*, pages 115–122. Springer Berlin Heidelberg.
(cf. page 41, 42, 43, 45)

[Payet et al., 2006] Payet, D., Courdier, R., Sébastien, N., and 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*, pages 127–131. IEEE Systems, Man, and Cybernetics Society.
(cf. page 41, 52, 53)

[Perumalla and Aaby, 2008] Perumalla, K. S. and Aaby, B. G. (2008). Data parallel execution challenges and runtime performance of agent simulations on GPUs. *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123.
(cf. page 35, 42, 45, 46, 71, 72, 83, 84)

[Picault and Mathieu, 2011] Picault, S. and Mathieu, P. (2011). An interaction-oriented model for multi-scale simulation. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pages 332–337. AAAI Press.
(cf. page 15, 52)

[Rao and Georgeff, 1995] Rao, A. S. and Georgeff, M. P. (1995). BDI-agents : from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, pages 312–319, San Francisco.
(cf. page 11)

[Resnick, 1996] Resnick, M. (1996). StarLogo : An Environment for Decentralized Modeling and Decentralized Thinking. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 11–12, New York, NY, USA. ACM.
(cf. page 60)

[Reynolds, 1987] 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*, volume 21 of *SIGGRAPH Computer Graphics '87*, pages 25–34, New York, NY, USA. ACM.
(cf. page 12, 36, 59)

[Ricci et al., 2011] Ricci, A., Piunti, M., and Viroli, M. (2011). Environment programming in multi-agent systems : an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2) :158–192.
(cf. page 52, 53, 103, 109)

[Richmond et al., 2009a] Richmond, P., Coakley, S., and Romano, D. M. (2009a). 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*, volume 2 of *AAMAS '09*, pages 1125–1126, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.
(cf. page 36)

[Richmond et al., 2009b] Richmond, P., Coakley, S., and Romano, D. M. (2009b). Cellular Level Agent Based Modelling on the Graphics Processing Unit. In *2009 International Workshop on High Performance Computational Systems Biology*, pages 43–50. IEEE.
(cf. page 36, 44)

[Richmond and Romano, 2008] Richmond, P. and 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)*.
(cf. page 35, 42, 43, 44, 45)

[Richmond and Romano, 2011] Richmond, P. and Romano, D. M. (2011). A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. *European Simulation and Modelling*.
(cf. page 37, 42, 43, 44)

[Richmond et al., 2010] Richmond, P., Walker, D., Coakley, S., and Romano, D. M. (2010). High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3) :334–47.
(cf. page vii, 36, 42, 43, 44, 61)

[Romano et al., 2005] Romano, D. M., Sheppard, G., Hall, J., Miller, A., and Ma, Z. (2005). BASIC : A Believable Adaptable Socially Intelligent Character for Social Presence. In *PRESENCE 2005, The 8th Annual International Workshop on Presence*, pages 21–22.
(cf. page 37)

[Russell and Norvig, 1995] Russell, S. J. and Norvig, P. (1995). *Artificial intelligence - a modern approach : the intelligent agent book*. Prentice Hall series in artificial intelligence. Prentice Hall.
(cf. page 12)

[Sabatier, 1986] Sabatier, P. A. (1986). Top-down and bottom-up approaches to implementation research : a critical analysis and suggested synthesis. *Journal of Public Policy*, 6 :21–48.
(cf. page 13)

[Sanders and Kandrot, 2011] Sanders, J. and Kandrot, E. (2011). *CUDA par l'exemple*. Pearson.
(cf. page 29, 88, 90, V)

[Sano and Fukuta, 2013] Sano, Y. and 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*, pages 262–267.
(cf. page 40)

[Schelling, 1978] Schelling, T. C. (1978). *Micromotives and Macrobehavior*. W. W. Norton, revised edition.
(cf. page 75)

[Shannon, 1998] Shannon, R. E. (1998). Introduction to the art and science of simulation. In *Proceedings of the 30th Conference on Winter Simulation*, WSC '98, pages 7–14, Los Alamitos, CA, USA. IEEE Computer Society Press.
(cf. page 12)

[Shekh et al., 2015] Shekh, B., de Doncker, E., and Prieto, D. (2015). Hybrid multi-threaded simulation of agent-based pandemic modeling using multiple GPUs. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 1478–1485.
(cf. page 42)

[Shen et al., 2011] Shen, Z., Wang, K., and Zhu, F. (2011). Agent-based traffic simulation and traffic signal timing optimization with GPU. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 145–150. Ieee.
(cf. page 38)

[Silva et al., 2010] Silva, A. R. D., Lages, W. S., and Chaimowicz, L. (2010). Boids That See : Using Self-occlusion for Simulating Large Groups on GPUs. *Comput. Entertain.*, 7(4) :51 :1–51 :20.
(cf. page 37)

[Simonin, 2001] Simonin, O. (2001). *Le modèle satisfaction-altruisme : coopération et résolution de conflits entre agents situés réactifs, application à la robotique*. PhD thesis, Université Montpellier II.
(cf. page 12)

[Sklar, 2007] Sklar, E. (2007). NetLogo, a Multi-agent Simulation Environment. *Artificial Life*, 13(3) :303–311.
(cf. page 14, 33, 35, 60)

[Smith, 1980] Smith, R. (1980). The contract net protocol : High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12) :1104–1113.
(cf. page 9)

[Strippgen and Nagel, 2009] Strippgen, D. and Nagel, K. (2009). Multi-agent traffic simulation with CUDA. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 106–114.
(cf. page 38, 43)

[Suzumura and Kanezashi, 2012] Suzumura, T. and Kanezashi, H. (2012). Highly scalable x10-based agent simulation platform and its application to large-scale traffic simulation. In *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '12, pages 243–250, Washington, DC, USA. IEEE Computer Society.
(cf. page 101)

[Sycara, 1998] Sycara, K. P. (1998). Multiagent systems. *AI Magazine*, 19(2) :79–92.
(cf. page 9)

[Varga and Mintal, 2014] Varga, M. and Mintal, M. (2014). Microscopic pedestrian movement model utilizing parallel computations. In *Applied Machine Intelligence and Informatics (SAMI), 2014 IEEE 12th International Symposium on*, pages 221–226.
(cf. page 38)

[Vigueras et al., 2010] Vigueras, G., Orduña, J., and Lozano, M. (2010). A GPU-Based Multi-agent System for Real-Time Simulations. In Demazeau, Y., Dignum, F., Corchado, J., and Pérez, J., editors, *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 15–24. Springer Berlin Heidelberg.
(cf. page 42, 43)

[Viroli et al., 2006] Viroli, M., Omicini, A., and Ricci, A. (2006). Engineering MAS environment with artifacts. In Weyns, D., Parunak, H. V. D., and Michel, F., editors, *Environments for Multi-Agent Systems*, volume 3830 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin Heidelberg.
(cf. page 52, 109)

[Volterra, 1926] Volterra, V. (1926). Fluctuations in the abundance of a species considered mathematically. *Nature*, 118 :558–560.
(cf. page 13, 93)

[Weyns and Holvoet, 2008] Weyns, D. and Holvoet, T. (2008). Architectural design of a situated multiagent system for controlling automatic guided vehicles. *Int. J. Agent-Oriented Softw. Eng.*, 2(1) :90–128.
(cf. page 52)

[Weyns and Michel, 2015] Weyns, D. and Michel, F. (2015). *Agent Environments for Multi-Agent Systems IV, 4th International Workshop, E4MAS 2014 - 10 Years Later, Paris, France, May 6, 2014, Revised Selected and Invited Papers*, volume 9068 of *LNCS*. Springer.
(cf. page 12, 52, 103, 109)

[Weyns et al., 2007] Weyns, D., Omicini, A., and Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1) :5–30.
(cf. page 12, 52, 53, 103, 109)

[Wooldridge, 2000] Wooldridge, M. (2000). Intelligent agents. In Weiss, G., editor, *Multiagent Systems : A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–77. MIT Press, Cambridge, MA, USA, 1st edition.
(cf. page 10)

[Wooldridge, 2002] Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley & Sons, 1st edition.
(cf. page 10)

[Zeigler et al., 2000] Zeigler, B. P., Kim, T. G., and Praehofer, H. (2000). *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition.
(cf. page 12, 13)

[Zhang et al., 2011] Zhang, L., Jiang, B., Wu, Y., Strouthos, C., Sun, P., Su, J., and Zhou, X. (2011). Developing a multiscale, multi-resolution agent-based brain tumor model by graphics processing units. *Theoretical Biology and Medical Modelling*, 8(1) :46.
(cf. page 35)

[Zhao et al., 2014] Zhao, C., Kaulakis, R., Morgan, J. H., Hiam, J. W., Ritter, F. E., Sanford, J., and Morgan, G. P. (2014). Building social networks out of cognitive blocks : factors of interest in agent-based socio-cognitive simulations. *Computational and Mathematical Organization Theory*, 21(2) :115–149.
(cf. page 12)

« *Look up at the stars and not down at your feet. Try to make sense of what you see, and wonder about what makes the universe exist. Be curious.* »

Stephen Hawking

Modélisation et implémentation de simulations multi-agents sur architectures massivement parallèles

Résumé : La simulation multi-agent représente une solution pertinente pour l'ingénierie et l'étude des systèmes complexes dans de nombreux domaines (vie artificielle, biologie, économie, etc.). Cependant, elle requiert parfois énormément de ressources de calcul, ce qui représente un verrou technologique majeur qui restreint les possibilités d'étude des modèles envisagés (passage à l'échelle, expressivité des modèles proposés, interaction temps réel, etc.).
Parmi les technologies disponibles pour faire du calcul intensif (*High Performance Computing*, HPC), le GPGPU (*General-Purpose computing on Graphics Processing Units*) consiste à utiliser les architectures massivement parallèles des cartes graphiques (GPU) comme accélérateur de calcul. Cependant, alors que de nombreux domaines bénéficient des performances du GPGPU (météorologie, calculs d'aérodynamique, modélisation moléculaire, finance, etc.), celui-ci est peu utilisé dans le cadre de la simulation multi-agent. En fait, le GPGPU s'accompagne d'un contexte de développement très spécifique qui nécessite une transformation profonde et non triviale des modèles multi-agents. Ainsi, malgré l'existence de travaux pionniers qui démontrent l'intérêt du GPGPU, cette difficulté explique le faible engouement de la communauté multi-agent pour le GPGPU.
Dans cette thèse, nous montrons que, parmi les travaux qui visent à faciliter l'usage du GPGPU dans un contexte agent, la plupart le font au travers d'une utilisation transparente de cette technologie. Cependant, cette approche nécessite d'abstraire un certain nombre de parties du modèle, ce qui limite fortement le champ d'application des solutions proposées. Pour pallier ce problème, et au contraire des solutions existantes, nous proposons d'utiliser une approche hybride (l'exécution de la simulation est partagée entre le processeur et la carte graphique) qui met l'accent sur l'accessibilité et la réutilisabilité grâce à une modélisation qui permet une utilisation directe et facilitée de la programmation GPU. Plus précisément, cette approche se base sur un principe de conception, appelé délégation GPU des perceptions agents, qui consiste à réifier une partie des calculs effectués dans le comportement des agents dans de nouvelles structures (*e.g.* dans l'environnement). Ceci afin de répartir la complexité du code et de modulariser son implémentation. L'étude de ce principe ainsi que les différentes expérimentations réalisées montre l'intérêt de cette approche tant du point de vue conceptuel que du point de vue des performances. C'est pourquoi nous proposons de généraliser cette approche sous la forme d'une méthode de modélisation et d'implémentation de simulations multi-agents spécifiquement adaptée à l'utilisation des architectures massivement parallèles.

Mots clés : Calcul Haute Performance (HPC), GPGPGU, Système Multi-Agents (SMA), Simulation Multi-Agent (MABS).

Modeling and implementing multi-agents based simulations on massively parallel architectures

Abstract: Multi-Agent Based Simulations (MABS) represents a relevant solution for the engineering and the study of complex systems in numerous domains (artificial life, biology, economy, etc.). However, MABS sometimes require a lot of computational resources, which is a major constraint that restricts the possibilities of study for the considered models (scalability, real-time interaction, etc.).
Among the available technologies for HPC (*High Performance Computing*), the GPGPU (*General-Purpose computing on Graphics Processing Units*) proposes to use the massively parallel architectures of graphics cards as computing accelerator. However, while many areas benefit from GPGPU performances (meteorology, molecular dynamics, finance, etc.). Multi-Agent Systems (MAS) and especially MABS hardly enjoy the benefits of this technology: GPGPU is very little used and only few works are interested in it. In fact, the GPGPU comes along with a very specific development context which requires a deep and not trivial transformation process for multi-agents models. So, despite the existence of works that demonstrate the interest of GPGPU, this difficulty explains the low popularity of GPGPU in the MAS community.
In this thesis, we show that among the works which aim to ease the use of GPGPU in an agent context, most of them do it through a transparent use of this technology. However, this approach requires to abstract some parts of the models, what greatly limits the scope of the proposed solutions. To handle this issue, and in contrast to existing solutions, we propose to use an hybrid approach (the execution of the simulation is shared between both the processor and graphics card) that focuses on accessibility and reusability through a modeling process that allows to use directly GPU programming while simplifying its use. More specifically, this approach is based on a design principle, called GPU delegation of agent perceptions, consists in making a clear separation between the agent behaviors, managed by the processor, and environmental dynamics, handled by the graphics card. So, one major idea underlying this principle is to identify agent computations which can be transformed in new structures (*e.g.* in the environment) in order to distribute the complexity of the code and modulate its implementation. The study of this principle and the different experiments conducted show the advantages of this approach from both a conceptual and performances point of view. Therefore, we propose to generalize this approach and define a comprehensive method relying on GPU delegation specifically adapted to the use of massively parallel architectures for MABS.

Keywords: High Performance Computing (HPC), GPGPU, Multi-Agent System (MAS), Multi-Agent Based Simulation (MABS).