SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Exécution efficace de systèmes multi-agents sur GPU

Guillaume LAVILLE

SPIM

Thèse de Doctorat

UFC

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X

THÈSE présentée par

Guillaume LAVILLE

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Exécution efficace de systèmes multi-agents sur GPU

Soutenue le 27 juin 2014 devant le Jury :

Christophe CAMBIER	Rapporteur	Chargé de recherche (HDR) à l'Université Pierre et Marie Curie, Paris
Michaël KRAJECKI	Rapporteur	Professeur à l'Université de Reims Champagne-Ardenne
Stéphane GENAUD	Examinateur	Professeur à l'Université de Strasbourg
Fabrice BOUQUET	Examinateur	Professeur à l'Université de Franche-Comté
Laurent PHILIPPE	Directeur de thèse	Professeur à l'Université de Franche-Comté
Kamel MAZOUZI	Encadrant	Ingénieur de Recherche au Mésocentre de calculs de Franche-Comté
Christophe LANG	Encadrant	Maître de Conférences à l'Université de Franche-Comté

SOMMAIRE

REMERCIEMENTS

Je tiens à remercier en premier lieu le Professeur Laurent Philippe, mon directeur de thèse, pour sa sympathie, sa disponibilité, ses idées, ses conseils et ses encouragements durant mes quatre années de thèse. Je voudrais également le remercier pour sa relecture et sa patience à corriger cette thèse.

Je remercie M. Kamel Mazouzi, Ingénieur de Recherche au Mésocentre de Calculs, pour son soutien indéfectible et ses conseils précieux tout au long de cette thèse. Son aide et ses remarques sur MCMAS ont en particulier eu une influence déterminante sur l'architecture et l'interface de la bibliothèque obtenue.

Je remercie M. Christophe Lang pour son aide au cours et en dehors de nombreuses réunion, et en particulier pour ses contributions et son expertise en systèmes multi-agents, qui m'ont été d'une assistance précieuse dans mes travaux et mon mémoire. J'exprime également ma plus profonde gratitude à M. Nicolas Marilleau pour avoir été un acteur clé du choix de ce sujet et de sa réalisation, en tant que personne également confrontée aux problématiques d'implémentations de systèmes multi-agents au quotidien.

M. Christophe Cambier et M. le Professeur Michaël Krajecki ont accepté d'être les rapporteurs de cette thèse, et je les en remercie, de même que pour leur participation au Jury. Ils ont également contribué par leurs nombreuses remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissant.

MM. les professeurs Fabrice Bouquet et Stéphane Genaud m'ont fait l'honneur de participer au Jury de soutenance ; je les en remercie profondément.

Tous mes remerciements vont également au Mésocentre de Calcul de Franche-Comté, qui m'a recruté en 2009 et sans lequel je n'aurais eu la chance unique d'être dans un cadre me permettant d'entreprendre et de réaliser cette thèse. Je remercie en particulier Cédric pour l'ambiance quotidienne dans le bureau et son expertise technique qui a été très utile à plusieurs reprises.

J'adresse également mes remerciements à tous les membres du DISC pour leur accueil au sein du Département d'Informatique des Systèmes Complexes de l'institut FEMTO-ST. Cela a été pour moi un honneur de devenir leur collègue après les avoir rencontrés en tant qu'enseignants au cours de ma formation.

Je tiens à remercier le personnel de l'école doctorale SPIM pour son aide précieuse dans les démarches administratives.

Je tiens enfin à remercier ma famille pour son soutien indéfectible au cours de la rédaction de cette thèse, tant d'un point de vue humain que rédactionnel, pour quelqu'un comme moi qui apprécie un peu trop les longues phrases...

INTRODUCTION

Ces dernières années ont consacré l'émergence du parallélisme dans de nombreuses branches de l'informatique, tant au niveau matériel que logiciel. Elle s'est manifestée au niveau matériel, tout d'abord, du fait de la stagnation de l'augmentation des fréquences de fonctionnement des unités de calcul, avec l'apparition d'architectures dotées de très grands nombres de coeurs. Elle s'est ensuite manifestée au niveau logiciel avec la démocratisation de plates-formes d'exécution parallèle telles que MPI ou OpenMP, ou l'apparition de nouvelles solutions comme OpenCL et CUDA, pour exploiter ce parallélisme matériel croissant.

Cette démarche de parallélisation de l'exécution peut être rapprochée du parallélisme conceptuel mis en œuvre dans les modèles multi-agents pour faciliter la description de systèmes complexes. Dans ce type de modèle, l'approche choisie est de décomposer un problème difficile ou impossible à appréhender de manière globale en sous-problèmes dont la résolution est plus simple, de manière à obtenir une solution globale. Ces sous-problèmes sont associés à des entités, ou agents, accomplissant chacun leurs tâches de manière simultanée et faisant évoluer le système dans son ensemble. Si l'adéquation entre un parallélisme d'exécution logiciel et conceptuel semble naturelle, la parallélisation reste une démarche difficile, du fait du déroulement séquentiel des opérations et des dépendances présents dans de très nombreux modèles agents. Les plates-formes d'exécution évoquées dans le paragraphe précédent sont généralistes, et ne sont pas spécifiquement adaptées aux problématiques multi-agents. Cette absence de support spécialisé impose au concepteur de nombreux développements de structures de données ou de traitements propres à son modèle, ou l'utilisation d'une plate-forme multi-agents parallélisée fournissant déjà ces outils.

L'objectif de cette thèse est de proposer une solution commune pour faciliter l'implémentation de tels modèles sur une plate-forme d'exécution massivement parallèle telle que le GPU, dont le nombre important de coeurs permet d'envisager l'exécution simultanée de grands nombres d'agents. Notre mémoire est pour cela découpé en deux parties : la présentation de notre contexte, puis celle de nos contributions.

Pour cerner notre contexte, nous présentons dans un premier temps les concepts de simulation et de modèle. Nous décrivons leur rôle en complément ou en remplacement de l'expérience pour permettre une meilleure compréhension du monde qui nous entoure. Cette présentation est également l'occasion d'introduire l'amélioration constante en précision et en taille des simulations nécessaire à l'avancée des connaissances et l'augmentation correspondante des ressources requises. Ce besoin motive à l'heure actuelle la recherche de nouvelles solutions d'exécution pour des simulations même modestes, exploitant efficacement plusieurs ressources matérielles.

Nous décrivons ensuite un type de système particulier, au coeur de notre sujet de thèse : les systèmes multi-agents. Ces systèmes permettent, en décomposant le modèle à simuler en entités indépendantes, les agents, d'appréhender des modèles sans loi globale de comportement. La dynamique de ces systèmes ne dépend plus alors uniquement de règles générales, mais de l'interaction entre un ou plusieurs algorithmes s'exécutant en parallèle. La simulation de ces systèmes rencontre, comme la simulation de manière générale, un problème de disponibilité de ressources en calcul et en mémoire dans le cas de grands espaces ou nombres d'individus, que nous illustrons sur quelques exemples connus.

L'identification de ce besoin en ressources nous amène à considérer les différentes approches de parallélisation permettant d'y répondre, avec leurs avantages et leurs contraintes en termes d'exécution et de programmation. Cette présentation est l'occasion d'introduire les GPU, ou cartes graphiques, qui offrent une capacité de calcul normalement inaccessible sur le CPU d'une seule machine. Ces matériels permettent au programme de partager aisément des données dans une même mémoire globale tout en offrant l'accès à plusieurs centaines de coeurs. Leur utilisation est cependant associée à de nombreuses contraintes, tant en termes de découpage de l'exécution qu'en termes d'utilisation et d'accès aux données, pour permettre une exécution efficace.

À la suite de cette présentation des solutions de parallélisation, nous évoquons leur application dans les simulations multi-agents. Pour cela, nous commençons par présenter les différentes approches de découpage de l'exécution et des données généralement utilisées dans le cas de modèles multi-agents. Nous présentons ensuite des plates-formes multi-agents supportant l'exécution parallèle du modèle comme MadKit, Repast HPC, JADE ou encore FLAME. Nous décrivons ensuite l'état de l'art des solutions permettant actuellement d'utiliser le GPU pour exécuter tout ou partie d'un modèle multi-agents. Ces solutions peuvent être classées en deux catégories principales, l'utilisation directe de modèles de programmation génériques comme CUDA ou OpenCL ou l'utilisation d'une bibliothèque d'abstraction telle que FLAME-GPU.

Ces deux catégories laissent cependant une ouverture pour une approche intermédiaire qui faciliterait la réalisation de simulations ou de traitements multi-agents sur GPU sans imposer l'utilisation d'une plate-forme de développement multi-agents particulière, contrairement à FLAME-GPU avec le formalisme FLAME. La définition de cette problématique nous sert de transition pour la présentation de nos contributions, en seconde partie, et en particulier de MCMAS [1], une bibliothèque d'exécution multi-agents sur GPU développée pour répondre à ces besoins.

Notre première contribution est la présentation de l'adaptation d'un modèle multi-agents connu, le système proie-prédateur, sur GPU, pour mettre en évidence sur un cas concret les changements en termes de structures de données et de découpage de l'exécution nécessaires au portage de ce type de simulation.

Cet exemple concret nous sert ensuite de fil rouge pour définir trois grandes approches de parallélisation du modèle sur GPU : une adaptation complète de la simulation, une délégation manuelle de certains traitements, ou la réutilisation de fonctions de haut niveau existantes. Ces approches nous permettent de définir les interfaces attendues par ces scénarios. L'adaptation complète ou partielle du modèle nécessite en effet une connaissance ainsi qu'un contrôle fin d'un modèle d'exécution tel que OpenCL ou CUDA. Au contraire, la parallélisation de certains traitements uniquement encourage une interface de programmation la plus simple possible pour le concepteur, de manière à faciliter son intégration et son utilisation dans de nombreux modèles existants sans connaissance particulière des détails d'implémentation.

Notre bibliothèque MCMAS vient répondre à ces types d'utilisations au moyen de deux interfaces de programmation, une couche de bas niveau MCM [2] et un ensemble de plugins utilisables sans connaissances GPU. Nous présentons tout d'abord l'architecture qui résulte de ces deux perspectives d'utilisation, ainsi que la manière dont certaines de ces fonctions sont assurées, avant de décrire l'utilisation de l'interface haut niveau de notre bibliothèque et l'ajout de fonctionnalités au moyen de nouveaux plugins.

Nous étudions ensuite l'utilisation de cette bibliothèque sur trois systèmes multi-agents distincts : le modèle proie-prédateur, notre fil rouge, le modèle MIOR et le modèle Collemboles. Ces

1. Many-Core Multi-Agent Systems
2. Many-Core Manager

applications sont l'occasion d'effectuer une étude des performances obtenues sur plusieurs types et générations de cartes graphiques par chaque modèle et des facteurs contribuant à une exécution efficace sur GPU.

Nous présentons également une synthèse de l'expérience acquise en proposant quelques conseils pour implémenter un modèle sur cette architecture. Ces observations, tant en termes de ressources ou de stockage de données qu'en termes de précision des traitements, visent à faciliter une utilisation efficace du grand nombre de supports d'exécution gérés par MCMAS.

Nous dressons enfin un bilan du travail et des réflexions présentées dans notre mémoire, avant d'évoquer quelques pistes possibles d'amélioration de notre solution. L'objectif de ces pistes est de favoriser l'extension et l'utilisation de notre bibliothèque, en proposant des couches d'adaptations dans des plates-formes existantes, la gestion de nouvelles structures de données, ou encore le support transparent d'une plus grande variété de configurations d'exécution.

I

Contexte

Dans cette première partie, nous présentons tout d'abord le contexte de nos travaux, de manière à définir la portée de notre sujet et à introduire la problématique à laquelle nous avons souhaité répondre : la parallélisation efficace de systèmes multi-agents sur architecture à grand nombre de cœurs.

Nous commençons par introduire le domaine de la simulation et des systèmes multi-agents et ce qu'ils représentent. Nous abordons ensuite la problématique du besoin en ressources rencontrées par ces modèles, lorsque nous cherchons à améliorer la précision et/ou la taille du modèle, et en quoi la parallélisation est une solution à ce besoin. Nous évoquons alors en quoi les moyens matériels associés à cette parallélisation peuvent être coûteux, et présentons les GPU, une architecture matérielle permettant de disposer de plusieurs centaines de coeurs d'exécution sur une machine locale. Après avoir présenté cette architecture, nous dressons un état de l'art des développements et portages de systèmes multi-agents déjà réalisés sur GPU, ainsi que la présentation d'une plate-forme multi-agents générique d'exécution sur GPU, FLAME-GPU.

1

Les systèmes multi-agents

Avant de présenter les systèmes multi-agents et l'utilisation que nous en feront, il est nécessaire de présenter le rôle d'une simulation, mais également de définir les concepts de modèle et de modélisation qui seront utilisés très largement dans la suite de notre propos.

1.1 Science et simulation

Dans cette section, nous commençons par présenter le contexte d'apparition de la simulation numérique, puis son principe. Nous définissons ensuite les termes de modèle et simulation avant d'étudier plusieurs classifications possibles des approches de modélisation permettant de passer d'un modèle à une simulation.

1.1.1 Principe de la simulation

La résolution de problèmes est l'un des moteurs de la recherche et de l'innovation technique. Si cette résolution a longtemps été effectuée manuellement, elle est de plus en plus confiée aux ordinateurs à même de réaliser d'importants volumes d'opérations. Avant de pouvoir résoudre un problème, il est cependant essentiel de disposer d'outils permettant de le décrire puis de le mesurer. C'est le rôle de l'expérience et de la simulation.

Une simulation est par nature la reproduction d'un phénomène en dehors du contexte dans lequel il se déroule habituellement. Cette simulation peut être de nature physique, sous la forme d'une expérience, ou dématérialisée sur un support informatique, auquel cas on parlera de simulation numérique. L'objectif est généralement de pouvoir étudier le phénomène en le reproduisant et en l'observant.

Un premier moyen d'observer et de décrire un phénomène est la mise en place d'un protocole expérimental. Ce protocole décrit un ensemble de conditions fixées ou variables où sera observé l'évolution de certaines métriques. Son objectif est de permettre un contrôle des résultats en assurant que l'observation soit ciblée et reproductible.

La réalisation ou la reproduction d'un phénomène dans sa globalité n'est cependant pas toujours financièrement ou pratiquement réalisable. Il est alors nécessaire de recourir à une représentation alternative généralement simplifiée du réel, le modèle. Dans le cas d'études topographiques sur l'érosion, il n'est ainsi pas possible de mettre sous serre une vaste étendue de territoire de manière à assurer des conditions contrôlées et reproductibles. De la même manière, le fait de demander à plusieurs milliers d'individus de reproduire à loisir un comportement pré-établi implique une coordination stricte faussant les résultats attendus.

L'objectif d'un **modèle** est de proposer une représentation de la réalité, de manière à en faciliter la compréhension. Sa conception se base sur des lois déduites d'un corpus d'observations et d'expériences.

Ce modèle peut ensuite être associé à des scénarios d'exécution reproduisant le phénomène observé correspondants à des conditions particulières pour en faire une **simulation** informatique.

Après cette courte introduction, nous allons maintenant définir formellement ces concepts.

1.1.2 Définitions : modèle, simulation

Le modèle est une représentation d'un phénomène ou d'un système permettant de le rendre plus aisément manipulable, comme souligné par cette définition proposée par Peter Haggett en 1973 [Hag73] :

Définition (modèle) : *les modèles sont des représentations schématiques de la réalité, élaborés en vue de la comprendre et de la faire comprendre.*

Cette simplification implique une approximation du système simulé : un modèle est donc une vision simplifiée de la réalité.

Wilson [Wil74] propose de son côté une définition de la simulation indépendante de toute notion de modèle :

Définition (simulation) : *par nature, une simulation est quelque chose pouvant être lancé, modifié, et produisant des résultats (exemple du crash-test). Peut être de nature physique (expérience dans un environnement contrôlé) ou dématérialisée (informatique).*

Cette définition met en avant l'indépendance entre les concepts de modèle et de simulation : une simulation est avant tout un moyen de produire des résultats, que ce soit à de manière physique ou informatique.

Le passage d'un système concret à un modèle de simulation correspond à un processus nommé **modélisation**.

1.1.3 Un continuum d'approches de modélisation

Notre contexte de travail est celui des systèmes multi-agents. Afin de situer ce contexte, nous rappelons ici les caractéristiques des principales approches de modélisation, qu'elles reposent sur l'utilisation de lois mathématiques de type équations différentielles ou statistiques ou sur la conception d'algorithmes représentant le comportement d'entités individuelles.

Nous proposons ici deux axes possibles de caractérisation de ces approches de modélisation. Ces caractérisations ne doivent pas être considérées comme hermétiques, car certaines démarches reprennent des éléments de chacune de ces approches de modélisation pour décrire des aspects différents d'un même modèle.

Modélisation ascendante ou modélisation descendante

Un premier axe de classification des approches de modélisation est la direction, descendante ou ascendante, dans laquelle cette démarche est appliquée au système simulé [CKQ+07, Jac98].

Dans une *approche descendante* (ou *top-down*), un comportement global est appliqué à tous les éléments particuliers du modèle. Il est par exemple possible, en observant l'évolution de la quantité d'oxygène présente dans l'environnement de bactéries aérobies, d'en déduire une loi décrivant la dynamique générale du système. Cette loi peut alors être utilisée pour reproduire cette évolution de manière informatique sous forme de modèle, sans avoir à réaliser à nouveau l'expérience concrète.

En appliquant les mêmes lois à tout le système, cette modélisation rend cependant difficile la description de comportements émergents des entités simulées entraînant une évolution non linéaire du modèle du fait de conditions particulières.

Dans le cas de ces comportements émergents en effet, seul le comportement de chaque élément est connu : il devient nécessaire d'adopter une *approche ascendante* (ou *bottom-up*). Le comportement du modèle dans son ensemble n'est alors plus globalement décrit par des lois globales mais par une combinaison d'algorithmes représentant les comportements locaux présents dans le système. Dans de nombreux systèmes biologiques, l'équilibre du métabolisme est basé sur l'interaction de processus antagonistes tels que la constitution de réserves de nutriments et la reproduction. L'évolution du système est alors directement déterminée par les conditions environnementales et l'impact résultant de chacun de ces mécanismes, ce qui rend une prévision *a priori* de l'évolution globale du système moins accessible. L'approche ascendance permet alors une modélisation plus adaptée, basée sur la description des comportements de chaque sous-élément du système.

Le choix de l'une ou l'autre de ces approches de modélisation est fonction du niveau de connaissance initial du système et du type de résultats souhaités, locaux ou portant sur l'évolution globale du modèle.

Simulation continue et simulation à événements discrets

Une autre distinction est effectuée dans la littérature [BPL+06, Fuj03] entre les simulations continues et les simulations à événements discrets (DES).

Une simulation continue permet de représenter des phénomènes par nature ininterrompus dits continus. Dans le cas d'une diffusion thermique dans un solide, il est ainsi possible de définir l'état du système à n'importe quel instant au moyen de fonctions mathématiques, généralement des équations différentielles. Dans ce cas, le choix de l'échelle de temps retenue est arbitraire et dépend uniquement de la durée et de la fréquence de l'observation demandée.

Une simulation à événements discrets permet au contraire de décrire des systèmes dont l'évolution dépend d'événements particuliers : en l'absence de ces éléments déclencheurs, la simulation demeure statique. Un exemple de système à événements discret est une chaîne de production, inactive en l'absence de tâches à traiter. Ce type de simulation peut être décrit sous la forme de réseaux de Petri conçus pour la description de systèmes basés sur des variables discrètes, ou encore sous forme de systèmes multi-agents.

Certaines simulations peuvent présenter à la fois des comportements discrets et continus. L'évolution de la position d'une balle en chute libre obéit ainsi à une loi continue, mais le sens du mouvement de cette balle est modifié de manière discrète par tout contact avec un autre objet, qu'il s'agisse du sol ou d'un autre obstacle. Il est dans ce cas possible de recourir à des simulations continues à événements discrets, ou simulations hybrides, associant ces deux fonctionnements.

1.2 Les systèmes multi-agents

Après avoir présenté la simulation de manière générale, nous nous intéressons plus spécifiquement aux systèmes et aux simulations multi-agents. La simulation à base d'agents centre le modèle sur des entités indépendantes nommées agents. Des comportements et des données sont associés à chacun de ces agents, de manière à obtenir des informations sur le modèle global : la modélisation à base d'agents est donc une modélisation de type ascendante permettant de simuler un environnement à partir de ses composants élémentaires.

Les modèles basés sur ce paradigme de conception sont dits *modèles multi-agents*. Les simulations réalisées à partir de ce type de modèles sont alors appelées *simulations multi-agents*.

Ce type particulier de simulations peut être décomposé en deux sous-classes [Fuj03] de simulations à événement discrets :

— Les simulations discrètes par pas de temps (time-driven). Dans ce cas l'évolution du système est guidée par le temps découpé en pas réguliers parcourus par la simulation.
— Les simulations discrètes par événement (event-driven). Dans ce cas l'évolution du système est guidée par une chaîne chronologique d'événements.

1.2.1 Définition et concepts

Il est important de définir le concept d'agent pour comprendre celui de système multi-agents. Pour cela, nous nous référons à la définition proposée par Jacques Ferber dans [Fer95] :

On appelle agent une entité physique ou virtuelle

— *qui est capable d'agir dans un environnement,*
— *qui peut communiquer directement avec d'autres agents,*
— *qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),*
— *qui possède des ressources propres, et qui est capable de percevoir (mais de manière limitée) son environnement,*
— *qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),*
— *qui possède des compétences et offre des services,*
— *qui peut éventuellement se reproduire,*
— *dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.*

Cette définition met en avant les capacités d'action sur l'environnement et de communication associées à ces agents. Elle souligne également la vision partielle de l'environnement associée à chaque agent, dont l'évolution est déterminée par cette perception partielle plutôt que par une connaissance globale du modèle.

Cette notion d'agent n'a de sens que comme partie d'un système plus large, le système multi-agents, sans lequel ces possibilités de communication sont inutiles. Ferber propose également, dans le même ouvrage, une définition de ces systèmes :

On appelle système multi-agents (ou SMA) un système composé des éléments suivants :

— *Un environnement E, c'est-à-dire un espace disposant généralement d'une métrique.*
— *Un ensemble d'objets O. Ces objets sont situés, c'est-à-dire que, pour tout objet, il est*

possible, à un moment donné, d'associer une position dans E. Ces objets sont passifs, c'est-à-dire qu'ils peuvent être perçus, créés, détruits et modifiés par les agents.

- *Un ensemble A d'agents, qui sont des objets particuliers ($A \in O$), lesquels représentent les entités actives du système.*
- *Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux.*
- *Un ensemble d'opérations Op permettant aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O.*
- *Des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification, que l'on appellera les lois de l'univers.*

L'implémentation d'un système multi-agents débute par la conception ou le choix d'un modèle multi-agents basé sur des agents, un environnement et les interactions entre ces entités [DSJD02]. Ces interactions et cette organisation définissent la fonction, le type et les scénarios de communication possibles dans le système simulé [JOF03]. En fonction du modèle, ces communications peuvent être directes ou indirectes, par le biais des mises à jour de l'environnement perçues ensuite par d'autres individus.

Les systèmes multi-agents représentent un continuum de simulation très large, s'étendant d'exemples très simples à des problèmes proches de l'intelligence artificielle. Cette diversité des problématiques est reflétée par le vaste vocabulaire employé par cette communauté scientifique, mêlant des concepts tels que celui d'agent à des notions moins directes d'objectif, de croyance ou de perception.

Un aspect présent dans de nombreux systèmes multi-agents est celui d'**environnement**. L'environnement décrit l'espace dans lequel évoluent les agents, sa structure (composition, agencement) et sa dynamique. Il peut être considéré comme un agent spécifique ou comme un simple ensemble de structures de données partagées. Il est typiquement chargé du stockage des propriétés globales au modèle, mais peut être associé à d'autres fonctions :

- Il peut remplir des fonctions à l'échelle du modèle telles que la gestion du temps ou la mise à jour des paramètres globaux à chaque itération.
- Il peut servir d'espace de stockage de tout ou partie des informations des agents.
- Il peut également remplir le rôle de médium de communication.

Des normes telles que FIPA [fip] ont été proposées pour standardiser l'implémentation de ce type de simulations. Cette norme, publiée en 1997, établit de nombreuses règles liées aux modes de communications et d'interactions entre agents par le biais d'échanges de messages. Elle est basée autour de trois rôles particuliers :

- Le système de gestion d'agents (Agent Management System, ou AMS), responsable de la supervision de l'accès et de l'usage de la plate-forme. Il assure en particulier l'authentification des agents présents et le contrôle des nouveaux enregistrements.
- Le canal de communications entre agents (Agent Communication Channel, ou ACC) fournit l'infrastructure de communication entre agents. Cette interface doit être compatible avec le protocole IIOP, pour garantir l'interopérabilité entre plates-formes multi-agents.
- L'assistant d'annuaire (Directory Facilitator, ou DF) propose un service de recherche aux agents de la plate-forme pour découvrir facilement les autres agents présents dans le modèle.

1.2.2 Agents réactifs, agents cognitifs

Il est courant d'effectuer dans les systèmes multi-agents une distinction entre agent cognitif et réactif [WD92, CDJM01] en fonction de leurs capacités d'action et de raisonnement.

Un agent cognitif dispose d'une mémoire de son passé et de son environnement lui permettant d'effectuer des déductions sur celui-ci et d'en prédire de futures évolutions. Le comportement de l'agent est déterminé par des intentions, correspondant à des objectifs à atteindre, et orientant les choix effectués entre plusieurs actions possibles. Ce type d'agent est utilisé pour représenter des individus dotés d'une intelligence propre. Celle-ci est alors souvent décrite sous la forme d'un moteur d'inférence intégré dans l'agent. Un exemple d'agent cognitif est ainsi le modèle proposé par J. Doran [DP93] pour décrire les évolutions sociales des sociétés du Paléolithique dans le sud-ouest de la France en fonction de la répartition des ressources. Cet article met en évidence l'importance des décisions prises par des individus particuliers sur la base d'une vision à moyen et long terme, plutôt qu'en simple réaction à une situation immédiate, pour expliquer les évolutions de peuplement observées en archéologie.

Un agent réactif ne peut au contraire que réagir à l'état instantané du système. Son comportement peut être caractérisé en se basant sur la psychologie comportementale comme purement S-R (Stimulus-Reaction), où S représente un état particulier de l'environnement et R une série d'actions élémentaires entreprises par l'agent en réaction à cet état. De tels comportements sont rencontrés aussi bien pour des animaux [McF87] que pour des créatures artificielles [Mae90].

La séparation entre agents réactifs et cognitifs n'est pas très nette, et certains agents mélangent donc des comportements réactifs et cognitifs. Cette vision à plus ou moins long terme de l'environnement a un impact important sur la complexité de chaque agent et donc sur celle de son implémentation.

1.2.3 Implémentation de modèles agents

Les systèmes multi-agents les plus simples peuvent être implémentés sous forme d'automates cellulaires. Ces automates sont également souvent utilisés pour représenter l'environnement de systèmes multi-agents plus complexes [SFS10].

Comme toute simulation informatique, l'exécution de simulations multi-agents requiert des ressources tant en termes de mémoire, pour stocker les données du système, qu'en temps de calcul pour le faire évoluer. Dans le cas des simulations multi-agents, ces besoins dépendent de deux facteurs principaux :

— Des ressources requises par l'environnement. S'il s'agit d'un environnement stockant des données pour chaque unité de l'espace de simulation, les ressources mémoires requises pour représenter cet espace seront alors proportionnelles à sa taille. Si un traitement est associé sur chacune de ces unités, le temps d'exécution associé aux processus de l'environnement va également en augmentant.
— Des ressources requises par les agents. Une augmentation du nombre d'agents implique une augmentation du nombre d'attributs à représenter, ainsi que du nombre d'individus à faire évoluer.

D'autres parties de la simulation, comme la récupération ou le stockage de résultats, contribuent également à ces besoins en ressource. Leur impact n'est cependant pas nécessairement lié à la taille du système simulé. Dans la suite de cette section, nous allons illustrer dans le cadre de quelque simulations multi-agents connues l'impact des agents ou de l'environnement sur la consommation

en ressources en fonction de la taille du système simulé.

1.2.4 Exemples de modèles

Jeu de la vie

L'exemple le plus connu de modèle multi-agents simple implémenté sous la forme d'automate cellulaire est probablement le Jeu de la Vie (Game of Life), imaginé en 1970 par John Horton Conway [Gar70]. Ce modèle est constitué d'une simple grille dont chaque cellule est soit "vivante" soit "morte". L'évolution de l'état de chaque cellule à la prochaine itération est déterminé par le nombre de ses voisines vivantes à l'itération précédente :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante.
— Une cellule vivante possédant deux ou trois voisines vivantes le reste.
— Une cellule vivante meurt dans le reste des cas.

Ce système simple est souvent employé comme exemple d'introduction à l'utilisation de plates-formes agent [Mic02, net], pour en présenter les concepts et la syntaxe fondamentale dans le cadre d'un modèle connu.

La seule structure de données du système dans ce cas est l'environnement. Comme cet environnement décrit toutes les cellules possibles, la mémoire requise est directement fonction de sa taille : si celle-ci double, la consommation en mémoire sera alors multipliée par quatre (espace en deux dimensions).

Le temps d'exécution est également directement lié à la taille de cet environnement grille, le même traitement devant systématiquement être appliqué à chaque cellule. Comme chaque traitement ne s'applique qu'à la cellule locale, la quantité totale de traitements à exécuter à chaque itération est proportionnelle à la taille de l'environnement. Ce modèle très simple peut donc devenir coûteux à grande échelle et, de ce fait, nécessiter des ressources de calculs parallèle pour explorer de grandes tailles de modèles [MCM12].

Abeilles

Un autre modèle largement représenté dans les différentes plates-formes multi-agents et la littérature est le mouvement d'essaims d'abeilles. Dans cet essaim, chaque agent est associé à une position dans un espace de simulation en deux ou trois dimensions, l'environnement. La position de chaque individu est ensuite mise à jour à chaque itération de manière à pouvoir observer le comportement global de l'essaim.

Ce modèle met en jeu deux types d'agents :

— La reine : cet agent particulier se déplace aléatoirement dans l'espace.
— L'abeille ouvrière : cet agent tend à se rapprocher de la reine de l'essaim en ajustant sa direction de déplacement. Si plusieurs reines sont en présence, l'individu sélectionne l'une de ces reines, ce qui peut induire des changements d'essaim.

Ce modèle est une excellente illustration de l'apparition d'un comportement émergent complexe, la création, la fusion et l'évolution de la forme d'un ou plusieurs essaims, à partir d'algorithmes simples. Le comportement observé varie en fonction des paramètres de la simulation et en particulier de la vitesse de déplacement ou du champ de vision de chaque individu.

Dans ce modèle, contrairement au jeu de la vie, l'environnement n'est plus une véritable struc-

ture de données séparées, mais un espace dans lequel une position est associée à chaque agent. Il n'est pas non plus associé à ses propres traitements.

Les besoins en mémoire et en calcul sont donc cette fois directement dépendants du nombre d'agents présents dans le système. Les besoins en mémoire évoluent de manière linéaire avec le nombre total d'agents présents dans le système, et donc la quantité d'attributs à stocker. L'évolution des besoins en calcul est cette fois encore linéaire, mais proportionnelle au nombre d'abeilles ouvrières présentes dans le modèle, plutôt qu'à la taille de l'environnement.

Fourmis

Un autre modèle multi-agents connu est celui de la colonie de fourmis, représentant le déplacement des individus d'une fourmilière à la recherche de nourriture. Il est possible d'identifier trois types d'agents dans ce système :

— La fourmilière. Cet agent fixe représente le point de départ et de retour des fourmis. Il est souvent responsable du stockage de la nourriture de manière à permettre la présence de plusieurs colonies dans une même simulation.
— Le dépôt de nourriture. Il est représenté soit sous la forme d'un agent fixe dans le cas d'un espace de simulation continu, soit sous la forme d'une donnée associée à chaque unité de l'environnement.
— La fourmi, seul agent mobile capable de se déplacer dans l'environnement. Sa fonction est de localiser et de ramener de la nourriture à sa fourmilière.

L'évolution globale de la simulation est déterminée par le mouvement des fourmis et la répartition géographique des fourmilières et des ressources dans l'environnement. La vitesse de collecte de nourriture peut alors être utilisée comme métrique d'évaluation de différentes stratégies de déplacement appliquées aux fourmis. Dans les cas les plus simples, ces déplacements sont effectués de manière aléatoire, mais un comportement plus réaliste est le dépôt et la prise en compte de phéromones dans l'environnement. Ces marqueurs chimiques encouragent l'individu à privilégier certaines directions de déplacement, et permettent l'émergence puis l'optimisation de chemins particuliers pour la récolte des ressources sans intelligence centrale directrice. Dans ce cas, l'environnement joue à la fois le rôle de mémoire et de médium d'interaction indirect entre individus.

Ce troisième exemple représente un cas où l'environnement et les agents correspondent chacun à des structures de données et des traitements distincts, et contribuent donc tous deux aux besoins en termes de mémoire et de calcul. L'évolution des ressources en fonction de la taille du modèle et du nombre d'agents reprend à la fois des aspects du jeu de la vie et des abeilles :

— La consommation en mémoire est proportionnelle à la taille de l'environnement et du nombre d'agents : si la taille du modèle est multipliée par deux, le nombre de cellules devant être stockées est multiplié par quatre, si une grille discrète de phéronomes est utilisée. De même, si le nombre d'agents fourmis est multiplié par deux, la mémoire est également multipliée par deux, pour stocker les données de ces individus supplémentaires.
— La consommation en temps de calcul est proportionnelle de la même manière à la taille de l'environnement, du fait de la nécessité de calculer la diffusion des phéromones dans la grille à chaque itération. Elle est également proportionnelle au nombre d'individus dont le déplacement doit être géré.

Il est important de noter que ces constatations ne sont valables que si l'environnement utilisé est une grille. Dans le cas où les phéromones seraient considérées comme des agents fixes, l'évolution des besoins en ressources se rapproche à nouveau de celle du modèle des abeilles.

1.2.5 Représentation de l'espace de simulation

Si les modèles évoqués jusqu'à présent stockent les informations de positionnement soit sous forme de coordonnées à l'intérieur de chaque agent, soit sous forme de structures de grille en deux ou trois dimensions, de nombreuses autres solutions de représentation sont possibles pour l'environnement du système et l'emplacement des agents.

Galland et all [GGDK09] proposent ainsi pour le positionnement en milieu urbain deux approches complémentaires :

— L'utilisation d'une carte de hauteur (heighmap), où chaque pixel indique l'altitude du point de l'espace simulé correspondant. L'information est alors encore une fois représentée sous forme de grille, ici une image.
— L'utilisation d'un modèle de positionnement des objets.

L'objectif de cette seconde représentation est de permettre un accès rapide à la position et à l'orientation des objets présents dans le modèle. L'environnement est découpé en zones décrites par un graphe, pour un environnement en une dimension, ou par un arbre spatial. Chacun des objets du système est alors associé au noeud correspondant aux zones où il est situé, de manière à rapidement pouvoir déterminer les objets présents ou non dans un espace donné. Il est possible à un objet d'appartenir à plusieurs zones, s'il se trouve sur une frontière : dans ce cas, l'objet est copié et stocké à plusieurs endroits de la structure.

Si dans ce cas la représentation sous forme de graphe est utilisée en complément d'une grille, de nombreux environnements multi-agents basés sur des axes de circulation discrets peuvent être entièrement représentés sous forme de graphe. Ces structures se retrouvent au sein de nombreux modèles de recherche de chemin dans la littérature agent, en particulier dans le cas de simulations de trafic routier [SN09].

1.2.6 Synthèse

L'étude des modèles du jeu de la vie, des abeilles ou des fourmis permet de mettre en évidence que l'exécution d'une simulation multi-agents peut rapidement devenir coûteuse, particulièrement dans le cas où l'environnement est représenté sous la forme d'une structure de données de type grille ou si le temps d'exécution de chaque agent est proportionnel à la quantité d'individus présents dans le modèle. Plusieurs scénarios sont à même d'imposer des simulations de taille importante, en espace de simulation ou en nombre d'agents.

Un premier scénario est la volonté de simuler des systèmes mettant eux-même en jeu des espaces géographiques ou des populations importantes. C'est par exemple le cas de la simulation d'une ville : une simulation doit alors idéalement être capable de traiter tout son espace et ses habitants dans une même exécution, pour garantir une bonne représentation du système. Cette problématique est au coeur de projets comme MIRO [BBMC+10], qui vise à étudier la mobilité urbaine.

Un deuxième scénario est celui des systèmes multi-échelles, où des simulations de portées très différentes doivent être couplées. Un exemple de tel système est Sworm [BMD+09], dédié à la modélisation de l'évolution des sols. Dans ce modèle, les principaux intervenants sont les vers de terre, à même de consommer et diffuser de la matière organique dans le sol. Cette matière organique fait également l'objet d'une évolution d'origine microbienne. Dans ce cas, la simulation même d'un petit volume de sol implique la réalisation de très grands nombres de simulations microscopiques MIOR.

Un dernier scénario enfin est de vouloir garantir l'apparition des comportements observés en pratique au sein de la simulation. Une simulation de trafic proposée par Strippgen [SN09] met en avant, même dans le cas où le système peut être décomposé à loisir, l'importance de taille ou de populations minimales pour voir émerger certains comportements. Dans ce cas, utiliser une simulation de trop petite taille, même quand c'est possible, est susceptible de fausser les résultats observés par rapport à une situation de taille plus importante.

Cette utilisation d'environnements de grande taille ou de populations agents importantes est susceptible d'amener deux types de problèmes :

— Des besoins en mémoire ne pouvant plus être assurés par une seule machine.
— Des temps d'exécution très longs. Ces temps sont ainsi de l'ordre de la semaine dans le cas du modèle Sworm.

Le recours à la parallélisation de l'exécution du système est une solution possible à ces deux limitations, comme nous l'abordons dans la suite.

2

Modèles d'exécution et de programmation parallèles

La parallélisation est une solution pour accélérer l'exécution d'un programme ou pour permettre le traitement de données de taille plus importante en les répartissant sur plusieurs machines. Il est possible de distinguer deux grands modèles d'exécution en parallèle : la parallélisation en mémoire partagée et la parallélisation en mémoire distribuée. Dans ce chapitre, nous présentons ces deux modèles de parallélisation ainsi que des exemples d'outils en facilitant l'exploitation. Nous illustrons également leur impact sur le découpage des données et de l'exécution d'un programme.

Dans les sections suivantes, nous présentons tout d'abord en quoi cette parallélisation vient en réponse aux besoins en ressources de calcul. Nous nous focalisons ensuite sur la parallélisation en mémoire partagée, et son exploitation par le biais des interfaces de programmation OpenMP et OpenACC. Nous présentons ensuite la parallélisation en mémoire distribuée et l'interface de programmation MPI, avant d'évoquer la parallélisation hybride combinant ces deux approches. Nous nous intéressons enfin à une nouvelle architecture d'exécution, le GPU, et voyons son modèle de programmation et ses apports par rapports aux architectures d'exécution traditionnelles en mémoire partagée ou en mémoire distribuée. Ces présentations nous permettent de définir les concepts utilisés pour la parallélisation des systèmes multi-agents.

2.1 Une réponse à des besoins en calcul

Une constante universelle de la recherche scientifique est la nécessité permanente d'avancer plus loin dans la connaissance. Ce progrès peut être obtenu en ouvrant de nouvelles voies de recherches ou en améliorant les connaissances existantes, au moyen d'expérimentations plus précises ou de taille plus importante.

A l'origine, ce processus a été purement mécanique, motivant l'invention de capteurs ou de méthodes de mesure du temps toujous plus fiables. Son application aux simulations informatiques se traduit désormais en besoins croissants en ressources mémoires et d'exécution.

La progression en puissance de calcul du matériel a longtemps été assurée par l'accroissement des fréquences de fonctionnement des processeurs. Une augmentation en fréquence a en effet pour avantage de permettre à un programme limité par la vitesse du processeur de s'exécuter plus rapidement sans la moindre adaptation, à performance constante par cycle d'horloge.

Cette augmentation de la fréquence a toutefois été freinée par l'apparition de multiple obstacles physiques, notamment en termes de miniaturisation et de densité thermique. L'accroissement de la puissance de calcul implique à présent une multiplication du nombre de coeurs d'exécution

soit de manière locale (processeur multi-coeurs), soit de manière distante en interconnectant de nombreuses machines (clusters de calculs). La parallélisation des programmes est alors un moyen de tirer parti de cette nouvelle répartition des ressources d'exécution.

La démarche de parallélisation est également motivée par l'accroissement des besoins en mémoire des programmes. Si le passage de nombreuses architectures au 64 bits permet maintenant l'adressage de très grands espaces de travail, les quantités de mémoire physiquement utilisables sur une même machine restent limitées. Dans ce cas la parallélisation permet d'additionner les capacités mémoires fournies par plusieurs machines.

Ce besoin croissant en puissance de calcul est illustré par l'augmentation d'année en année des capacités offertes par les plus puissants clusters mondiaux du TOP500 [1]. Les premières places de ce classement étaient ainsi occupées par des solutions de l'ordre de la centaine de TeraFlops (1000 milliards, ou 10^{12} opérations flottantes par seconde) en juin 2005, puis du PetaFlops (10^{15} opérations) en juin 2009. En novembre 2013, les premières machines du classement proposent maintenant des puissances de plusieurs dizaines de PetaFlops.

Si cette parallélisation est un moyen d'accéder à davantage de ressources d'exécution, les gains en termes de temps obtenus dépendent directement de la fraction de temps d'exécution du programme à même d'être parallélisée par rapport à celle devant demeurer séquentielle. La loi d'Amdahl [Amd67], énoncée en 1967, rappelle que le gain de performance pouvant être attendu de la parallélisation d'une partie d'un programme est directement proportionnel à la fraction du temps d'exécution correspondant.

$$T_a = (1 - s)T + \frac{sT}{Ac}$$

$$S = \frac{T}{T_a} = \frac{1}{(1 - s) + \frac{s}{Ac}}$$

Où sont représentés :
— T le temps d'exécution du programme avant parallélisation.
— T_a le temps d'exécution du programme après parallélisation.
— s la fraction du temps T concernée par l'amélioration.
— Ac le facteur d'accélération obtenue sur la portion concernée.
— S le facteur d'accélération globale.

En pratique, l'application de cette loi se manifeste par une stagnation des performances au-delà d'un certain nombre de coeurs. Celle-ci survient d'autant plus rapidement que la fraction de temps parallélisée diminue, tel qu'illustré par la Figure 2.1.

2.2 Parallélisation en mémoire partagée

2.2.1 Modèle d'exécution

La parallélisation en mémoire partagée est un modèle d'exécution permettant de tirer parti de ressources de calcul parallèles sur une même machine en découpant l'exécution du programme en plusieurs fils d'exécutions disposant d'un accès à un espace mémoire commun (Figure 2.2).

1. http ://www.top500.org/

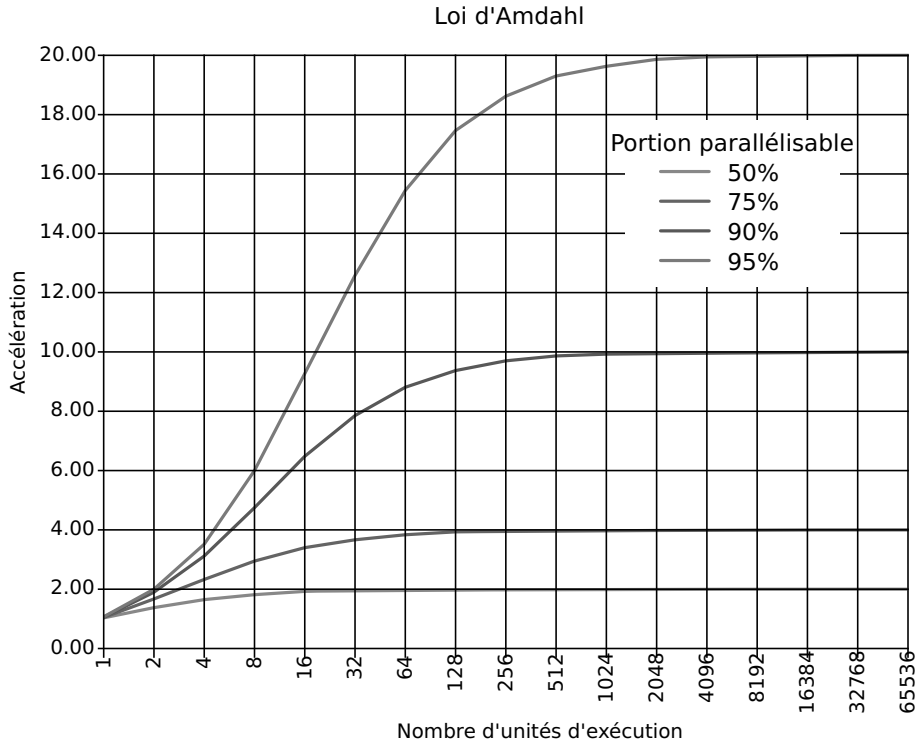

FIGURE 2.1 – Exemples d'applications de la loi d'Amdhal

Une première manière de paralléliser l'exécution du programme est de faire appel à des processus légers, ou threads. Ces processus sont dit légers car ils partagent l'ensemble de leur espace mémoire, ce qui réduit les coûts de création ou de destruction associés à ces threads par rapport à des processus distincts, dits lourds.

Une autre manière de paralléliser une exécution est de partager des portions de mémoires entre processus indépendants, soit en recopiant les données d'un processus parent au moment de la création d'une processus fils (fork), soit en utilisant les primitives de mémoire partagée proposées par le système. Cette technique est souvent employée pour des services où un unique processus parent demeure en attente de traitements à confier à un ou plusieurs processus fils. Ce fonctionnement est qualifié de maître-esclave.

La parallélisation en mémoire partagée n'implique par nécessairement la création de plusieurs processus et peut également être réalisée au niveau de l'instruction. Dans ce cas, une même opération est appliquée à plusieurs données (SIMD) indiquées sous forme de vecteurs. Ces instructions sont pour cette raison dites vectorielles. Ce mode d'exécution est à la base de l'exécution sur GPU.

La parallélisation en mémoire partagée est le modèle de parallélisation le plus aisé à exploiter car il permet de conserver un seul espace mémoire pour toutes les tâches. Ce partage facilite l'adaptation d'un algorithme séquentiel avec un minimum de modifications, sans répartition particulière des données. Comme pour toute ressource partagée, il devient cependant nécessaire de gérer la cohérence des données mémoires puisque plusieurs tâches peuvent les modifier de manière simultanée. Suivant le langage de programmation utilisé, cette synchronisation de l'accès aux données peut être intégrée au niveau des structures de données fournies (structure de données "thread-safe") ou être de la responsabilité du développeur.

L'obligation de conserver toutes les ressources sur une même machine rend difficile l'utilisation de ce type de parallélisation au-delà de quelques dizaines de coeurs et d'une centaine de gigaoc-

tets de mémoire vive avec des processeurs traditionnels. Ces limites correspondent de plus à des machines dédiées à ce type de parallélisation et sont donc en pratique beaucoup plus basses pour des machines de bureau ou des ordinateurs portables.

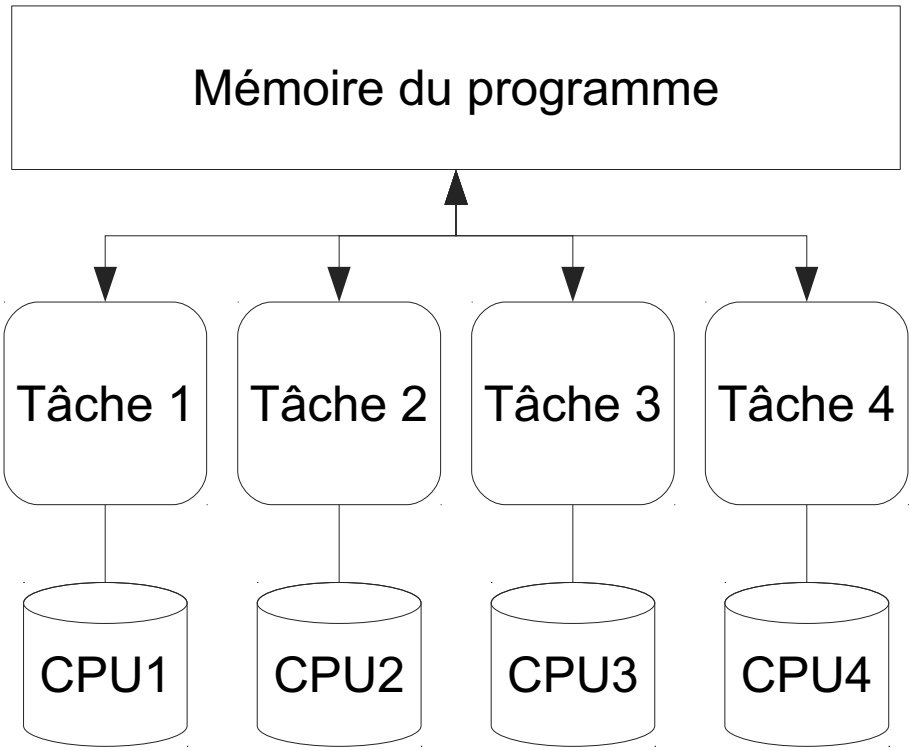

FIGURE 2.2 – Découpage d'une exécution en mémoire partagée

L'utilisation de la parallélisation en mémoire partagée est grandement facilitée par des bibliothèques génériques telles que OpenMP, utilisé par de nombreuses simulations agents.

2.2.2 OpenMP, un modèle de programmation

OpenMP [CDK+01] est un modèle de programmation pour les langages C, C++ et Fortran permettant le calcul parallèle en mémoire partagée. Il a été pour la première fois proposé en 1997.

La parallélisation offerte par ce modèle repose sur la création et l'exécution implicite de sections parallèles d'un programme par un ensemble de threads, ou workers, alloués et gérés automatiquement par OpenMP. Ces sections parallèles sont indiquées par le biais de directives de préprocesseur classées en instructions de contrôle d'exécution (boucles parallèles), en directives de partage des données (privées, partagées), en outils de synchronisation permettant de coordonner la progression de l'exécution des threads et en fonctions de gestion de l'environnement.

La possibilité d'annoter un code source existant facilite grandement l'utilisation d'OpenMP pour paralléliser de manière incrémentale un programme séquentiel. OpenMP permet également de configurer le nombre de coeurs d'exécution locaux à utiliser, pour mesurer aisément l'impact de la parallélisation sur les performances obtenues.

2.3 Parallélisation en mémoire distribuée

Les ressources fournies par une seule machine ne suffisent pas toujours à atteindre les objectifs en termes de temps d'exécution ou de mémoire requis par le programme. Dans ce cas, il devient intéressant de pouvoir exploiter simultanément plusieurs machines. Cette parallélisation est souvent utilisée sur des grilles (machines hétérogènes reliées par un réseau informatique) ou des clusters (ensemble de noeuds de calculs homogènes, souvent reliés par un réseau informatique haute performance tel que Infiniband).

2.3.1 Modèle d'exécution

La parallélisation en mémoire distribuée requiert une distribution explicite des données entre tâches d'exécution (Figure 2.3) de manière à permettre la répartition des tâches sur plusieurs machines dotées de mémoire indépendantes. Cette distribution des données permet également de s'affranchir des limitations en termes de taille mémoire imposées par une seule machine, en ne stockant pour chaque unité de traitement que les données nécessaires à son exécution. Cette répartition implique cependant la copie de certaines données communes dont la modification doit être ensuite répercutée dans les autres processus.

La parallélisation en mémoire partagée impose également des modifications en profondeur de l'algorithme pour prendre en compte ce découpage mémoire, ce qui rend son utilisation pour un programme existant moins aisée que la parallélisation en mémoire partagée.

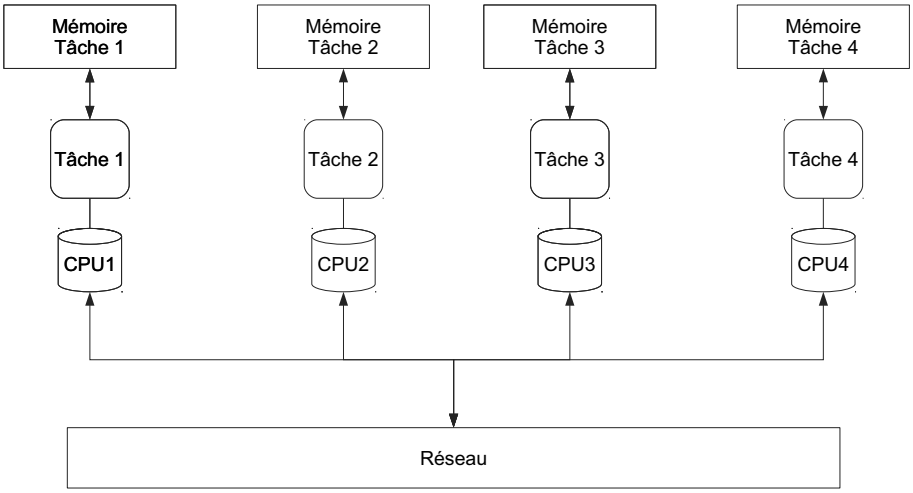

FIGURE 2.3 – Découpage d'une exécution en mémoire distribuée

La parallélisation en mémoire partagée requiert une prise en compte explicite du découpage des données par le concepteur du programme, mais peut elle aussi être facilitée au moyen de modèles de programmation comme MPI, capables d'abstraire la localisation et les communications entres portions du programme.

2.3.2 Un modèle de programmation standard : MPI

MPI [Mes09] est une norme définissant un ensemble de fonctions de communication entre processus locaux ou distants. Des implémentations pour les langages C, C++ et Fortran en sont

disponibles sur de nombreuses plates-formes, ce qui en fait un standard pour la réalisation de parallélisations distribuées.

L'objectif de ces fonctions est de permettre de bonnes performances d'exécution aussi bien sur une même machine qu'entre des machines distantes. MPI repose pour cela sur un ensemble de primitives de communication de haut niveau susceptibles d'exploiter les mécanismes optimisés d'échange de données offerts par le système d'exploitation et le matériel.

Une exécution MPI est constituée d'un ensemble de processus associés à des numéros de rang indépendants de leur localisation physique (Figure 2.4). Ces numéros de rang permettent à chaque processus d'adapter ses traitements en fonction de son rôle dans le groupe, en se comportant par exemple en maître distribuant des tâches ou en esclave traitant ces calculs.

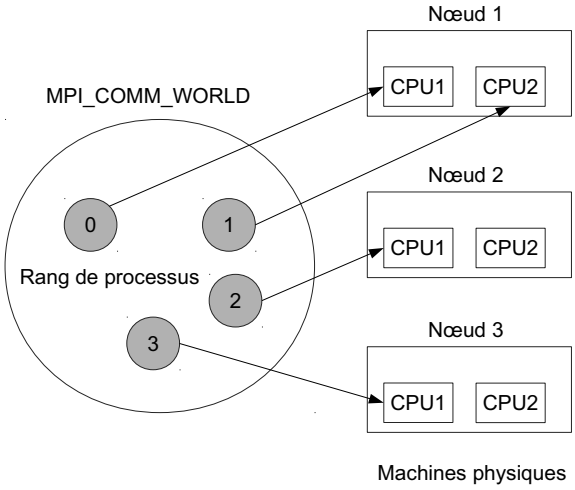

FIGURE 2.4 – Exemple d'association entre processus MPI et matériel physique

MPI propose deux catégories d'opérations de communications :

— **Les opérations point-à-point** : ces communications mettent en jeu un unique émetteur et destinataire dans le groupe de processus.
— **Les opérations de groupe**, également qualifiées de collectives ou de multicast, impliquent la participation d'une partie ou de tous les processus MPI pour réaliser un même traitement. Un exemple de tel traitement commun est la diffusion d'une donnée en début de calcul, ou la mise en commun de résultats partiels à la fin de l'exécution MPI.

La plupart de ces opérations de communication possèdent des variantes synchrones et asynchrones, de manière à faciliter la gestion du déroulement de l'exécution ou la poursuite de traitements en tâche de fond dans l'attente de communications.

La seconde version de MPI sortie en 1997 apporte la possibilité d'intégrer ou de créer dynamiquement des processus en cours d'exécution MPI. Elle permet également la gestion en parallèle de flux d'entrée/sortie vers des fichiers à l'aide des fonctions MPI-IO.

Les implémentations libres les plus connues du standard MPI sont MPICH [2] et OpenMPI [3]. A côté de ces implémentations généralistes, de nombreux constructeurs proposent des alternatives optimisées pour leurs solutions logicielles et matérielles telles que IntelMPI pour les processeurs Intel. Si le standard MPI officiel est dédié aux langages C et C++, des solutions équivalentes pour Java telles que MPJ Express [SMH+10] ou JACE [BDM04] existent également.

2. http ://www.mcs.anl.gov/project/mpich-high-performance-portable-implementation-mpi
3. http ://www.open-mpi.org/

2.4 Parallélisation hybride

2.4.1 Modèle d'exécution

La récente popularisation des processeurs multi-coeurs et l'apparition de nouvelles solutions d'exécution comme les cartes graphiques ont favorisé l'apparition de parallélisations dites hybrides, mettant à contribution dans un même programme plusieurs modèles de programmation distincts comme OpenMP, MPI ou le GPGPU [4].

L'utilisation judicieuse de ces différentes solutions permet d'exploiter l'ensemble des ressources présentes sur une même machine mais impose toutefois un certain nombre de précautions pour éviter tout conflit entre les modèles d'exécution. L'utilisation simultanée de MPI et d'OpenMP (Figure 2.5) requiert en particulier une certaine vigilance pour éviter tout problème de cohérence de l'état des processus ou des données.

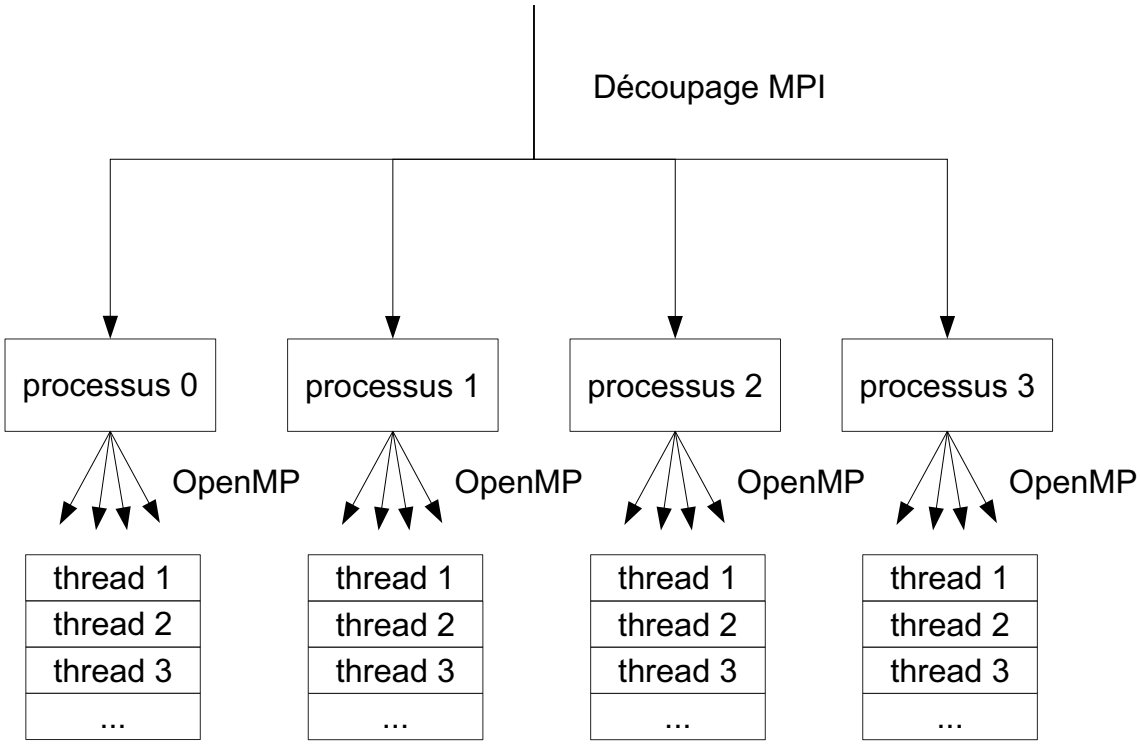

FIGURE 2.5 – Parallélisation associant OpenMP et MPI

Certains modèles, comme OpenACC, visent encore une fois à faciliter l'utilisation de ce type de distribution en regroupant les deux approches de parallélisation au sein d'un même formalisme, inspiré de OpenMP.

2.4.2 OpenACC, un modèle de programmation

OpenACC [WSTaM12] est un modèle de programmation soutenu par les sociétés Gray, CAPS, PGI et NVIDIA permettant de tirer parti à la fois d'architectures processeurs traditionnelles et de

4. General-Purpose Computing on Graphics Processing Units.

cartes graphiques (Figure 2.6). Il se différencie d'OpenMP par le support des cartes graphiques comme architecture d'exécution. Son utilisation, à base de directives de pré-processeur, est par ailleurs très similaire, de manière à faciliter son adoption en remplacement de la solution précédente pour des architectures CPU ou hybrides.

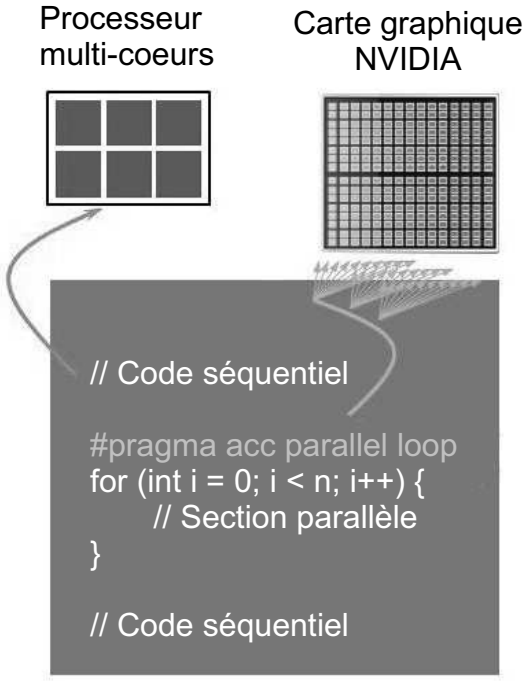

FIGURE 2.6 – Exemple d'utilisation du processeur et de la carte graphique avec OpenACC

Le support d'OpenACC est pour l'instant limité à quelques compilateurs commerciaux fournis par les société participantes au standard, dont le compilateur PGI. Il est néanmoins disponible sous forme de branche expérimentale depuis septembre 2013 dans le compilateur libre GCC. Un implémentation de recherche nommée accULL [5] est également en phase de développement.

2.5 Une nouvelle architecture d'exécution : le GPU

Après ce panorama des différents types de parallélisation connus, nous allons maintenant présenter une architecture d'exécution spécifique, le GPU. Cette architecture reprend à la fois des concepts de parallélisation en mémoire partagée et en mémoire distribuée à l'intérieur d'un même modèle de programmation. Elle nécessite toutefois une bonne connaissance des contraintes matérielles de la carte pour tirer parti efficacement des ressources fournies, particulièrement sur les architectures les plus anciennes.

Pour cela, nous commençons par présenter l'origine et l'architecture matérielle des GPU. Nous étudions ensuite le modèle de programmation associé à cette architecture, avant d'évoquer les contraintes posées par ce modèle d'exécution. Nous présentons ensuite des bibliothèques permettant d'utiliser cette architecture dans des programmes existants sans connaissance directe de ce modèle de programmation. Enfin, nous replaçons l'exécution GPU dans le contexte d'une évolution plus large de la parallélisation vers des architectures matérielles dites many-cores, proposant de nombreux coeurs d'exécution.

5. http ://accull.wordpress.com/

2.5.1 Genèse des GPU

Les cartes graphiques ou GPU [6] ont été à l'origine conçues pour décharger le CPU des calculs coûteux liés à l'introduction de rendus graphiques dans les programmes. Ces calculs, qu'il s'agisse de compositions de texture ou de calculs dans l'espace, se caractérisent en effet par l'application vectorielle d'une même opération à d'importants volumes de données. En proposant plusieurs dizaines ou centaines de coeurs d'exécution, le GPU permet d'appliquer ces traitements en parallèle et ainsi de réduire le temps total nécessaire pour effectuer ces opérations. Cette spécialisation permet également de réduire la complexité de chaque coeur et ainsi d'en augmenter la densité sur une surface donnée.

Au départ, les premières cartes graphiques ne déchargeaient le processeur que de certains traitements graphiques. L'augmentation de la résolution d'une part, et des attentes en qualité de rendu graphique d'autre part, ont rapidement amené la délégation de plus en plus d'opérations à ces cartes, jusqu'à l'apparition des premiers GPU programmables, c'est à dire capables d'exécuter des portions de programme. Ces possibilités de programmation, initialement très limitées, n'ont été accessibles dans un premier temps que par le biais de bibliothèques de rendu graphique comme OpenGL et DirectX. Elles ont été pour la première fois pleinement accessibles au développeur en 2008, avec la GeForce 8, par le biais du modèle de programmation CUDA. Le modèle OpenCL [Khr08] apparaît également la même année et permet l'utilisation des matériels graphiques d'autres fabriquants, notamment Intel et AMD, ainsi que l'exécution sur processeur traditionnel par le biais de OpenMP. Un autre modèle de programmation GPU, DirectCompute [Joh12], a depuis été proposé par Microsoft dans sa bibliothèque DirectX pour les systèmes d'exploitation Windows.

2.5.2 Architecture matérielle

Une carte graphique est constituée d'un très grand nombre de coeurs graphiques organisés en multi-processeurs. Chacun de ces coeurs dispose d'un accès à une vaste hiérarchie mémoire. Une partie de cette mémoire lui est propre, et une autre partie est partagée avec les autres coeurs du multi-processeur ou de la carte. L'apparition de la programmation GPGPU [7] a eu un impact sur l'architecture matérielle des cartes graphiques, où il est possible de déceler deux générations principales (Figure 2.8) :

— Les cartes graphiques antérieures à l'architecture Fermi, basées sur une hiérarchie mémoire complexe sans caches implicites. Dans ce cas, la gestion des latences d'accès aux données pour éviter un ralentissement de l'exécution est du ressort du programme.
— Les cartes graphiques plus récentes qui introduisent un mécanisme de caches mémoire L1 et L2 analogues à ceux présents sur CPU. Ces caches permettent le stockage dans une mémoire rapide, de manière transparente, des données les plus fréquemment utilisées par les unités de calcul.

Il est possible d'identifier trois niveaux de mémoires principaux dans cette hiérarchie :

— **Les registres** : chaque multi-processeur dispose de plusieurs centaines de registres. Ces registres sont partagés de manière statique entre les coeurs graphiques en début de programme. Ils sont d'accès très rapide et permettent le stockage des données intermédiaires entre les instructions consécutives du même programme.

6. (Graphical Processing Unit)
7. General-Purpose Processing on Graphics Processing Units

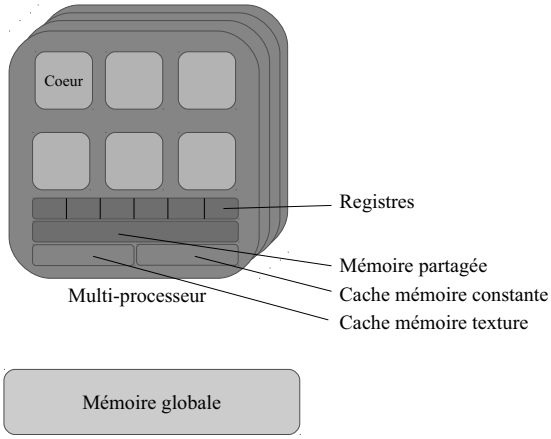

FIGURE 2.7 – Architecture matérielle pre-Fermi

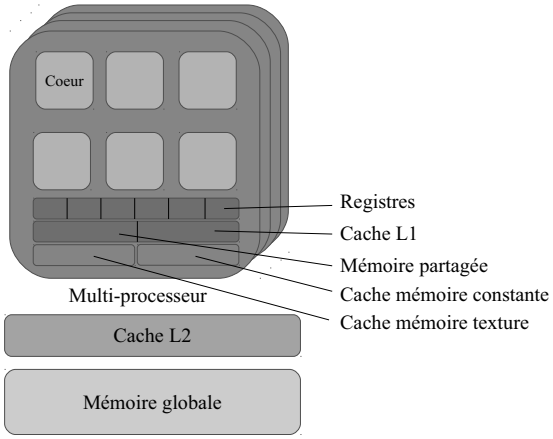

FIGURE 2.8 – Architecture matérielle Fermi, Kepler

— **La mémoire partagée** : cette mémoire est partagée entre tous les coeurs situés dans un même multi-processeur. Sa latence d'accès reste faible et permet de partager des données temporaires ou souvent réutilisées.
— **La mémoire globale** : cette mémoire est accessible à l'ensemble des coeurs graphiques de la carte. Elle permet le stockage de la plupart des données d'entrée ou de sortie du programme, avec ses quelques gigaoctets de capacité. Elle se caractérise toutefois par des temps de latence d'accès beaucoup plus importants, de plusieurs centaines de cycles d'horloge. Cette mémoire est la seule accessible depuis le CPU pour l'échange de données.

Avant l'apparition du cache L1 et L2 sur les architectures GPU les plus récentes, différentes techniques ont été imaginées pour dissimuler les latences d'accès à cette mémoire globale. L'une d'elles [RRB+08] est l'utilisation de zones de mémoire globale d'utilisation spécifiques, pour faciliter l'optimisation de ces scénarios par le matériel.

— **Mémoire locale** : cette zone, accessible en lecture/écriture, permet de stocker d'éventuelles informations ne tenant pas en registre.
— **Mémoire constante** : cette zone permet de stocker des données accessibles en lecture seule par l'ensemble des coeurs graphiques. La plupart des matériels utilisent alors une mémoire cache spécialisée pour réduire la latence d'accès aux données constantes les plus utilisées.
— **Mémoire texture** : cette zone permet le stockage de textures graphiques. Comme pour la mémoire constante, elle est associée sur de nombreux matériels à une mémoire cache spécialisée au niveau de chaque multi-processeur. Chacune de ces textures n'est accessible qu'en lecture ou en écriture seule au niveau d'un même programme.

Ce système peut toujours être utilisé sur les cartes récentes en complément d'un cache L1 propre à chaque multi-processeur et d'un cache L2 global à la carte. La gestion du partage des ressources mémoires entre ces deux mécanismes est également possible sur les cartes NVIDIA.

Les copies de données entre CPU et GPU sont réalisées par le biais de l'interface PCI-Express de la carte graphique. Les restrictions d'accès en lecture ou écriture à ces différentes mémoires ne s'appliquent qu'aux programmes en exécution sur le GPU : le CPU dispose toujours d'un accès complet à l'ensemble de la mémoire globale.

2.5.3 Modèle de programmation

Le modèle de programmation GPU se caractérise par l'utilisation la plus large possible du découpage en threads en remplacement des boucles présentes dans l'algorithme. Cette démarche de parallélisation fine se justifie par les coûts d'exécution différents rencontrés sur CPU et sur GPU.

Un processeur traditionnel est conçu pour traiter un nombre limité de processus s'exécutant sur une longue durée à l'échelle du matériel : secondes, minutes, heures. La création et la destruction de processus est ainsi un traitement coûteux, car elle requiert l'allocation ou la libération d'un environnement mémoire et système complet. Les threads, ou processus légers, permettent de réaliser des traitements ponctuels en évitant cette allocation d'environnement, comme évoqué dans notre section sur la parallélisation en mémoire partagée.

Au contraire, un GPU est conçu pour permettre l'application d'un petit nombre d'opérations sur de grands volumes de données. Ce type d'exécution se caractérise par des tâches brèves et remplacées très fréquemment de manière à assurer le remplissage des centaines de coeurs proposés par l'architecture. Les latences mémoires, importantes en regard du temps de traitement de chaque tâche, encouragent également la ré-allocation des ressources matérielles de calculs bloqués en attente d'opérations mémoire à d'autres traitements. Dans ces circonstances, l'utilisation de nombres très importants de threads permet à l'ordonnanceur GPU de disposer d'un grand nombre de candidats pour optimiser le remplissage des ressources d'exécution fournies par la carte. Ces candidats sont regroupés en *warp*, ou paquet d'exécution, au moment de leur attribution à un multi-processeur matériel particulier.

Pour permettre ce découpage de l'exécution, les modèles de programmation CUDA et OpenCL sont tous deux basés sur trois concepts fondamentaux illustrés par la figure 2.9 :

- le *kernel* représente la suite d'instructions à exécuter sur le GPU. Il se présente sous la forme d'une fonction admettant un ensemble de paramètres en entrée et en sortie. Il est possible au kernel d'utiliser les primitives fournies par la plate-forme de programmation elle-même, mais il ne peut faire appel à aucune bibliothèque ou fonctionnalité offerte par le CPU.
- le *work-item (OpenCL) / thread (CUDA)* (ou tâche) représente le support d'exécution d'une instance de kernel. Chaque thread a accès à son propre espace mémoire, comme évoqué dans la présentation de l'architecture matérielle GPU, mais également aux données partagées de la carte.
- le *work-group (OpenCL) / bloc (CUDA)* (ou groupe de tâches) représente une grille de une à trois dimensions de tâches d'exécution GPU. Ce bloc permet de gérer le partage de ressources entre les traitements manipulant des données proches en mémoire et le découpage des données du traitement. Dans le cas d'une matrice, par exemple, il est possible d'associer un bloc de tâches à chaque ligne de la matrice, de manière à permettre l'échange de données et des synchronisations locales entre ces tâches.

La nature déportée de la carte graphique impose une préparation de l'exécution et des données, puis une récupération des résultats et des ressources après l'exécution sur GPU. La réalisation d'un traitement est ainsi découpée en cinq phases (Figure 2.10) :

- **Chargement du programme**. Les traitements à exécuter sont envoyés sur la carte graphique sous forme de binaires pré-compilés (CUDA) ou de code source (OpenCL, CUDA) devant auparavant passer par une compilation gérée par la plate-forme de programmation.
- **Allocations mémoires des paramètres et copie des données d'entrée**. Les paramètres du

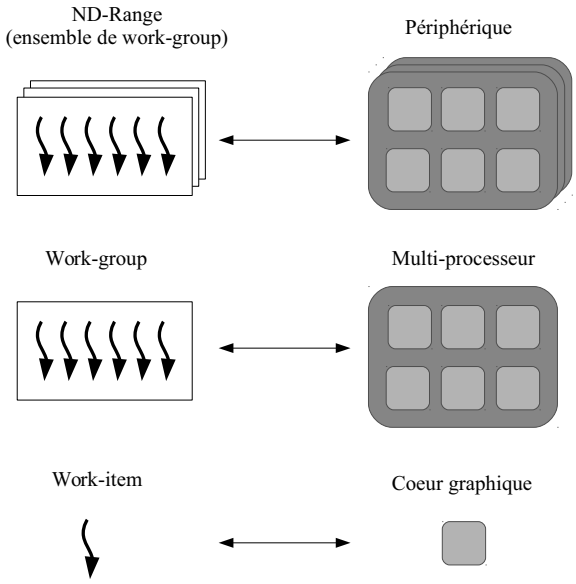

FIGURE 2.9 – Correspondances entre les structures de découpage OpenCL et l'architecture matérielle GPU

programme sont alloués par le processeur central dans la mémoire globale de la carte, et les données d'entrée recopiées ou rendues accessibles sous forme de partage mémoire au GPU.

— **Lancement de l'exécution d'un ou de plusieurs kernels**. Une fois les paramètres préparés sur la carte, le processeur soumet un ou plusieurs kernels d'exécution à l'ordonnanceur GPU. Ces soumissions peuvent être effectuées de manière synchrone, auquel cas le programme CPU demeurera bloqué jusqu'à la fin de l'exécution, ou asynchrone. Des dépendances peuvent être définies entre kernels d'exécution, de manière à garantir leur ordre de passage sur GPU.
— **Exécution non interruptible sur la carte graphique** Les traitements sont lancés par la plate-forme GPU dès que des ressources sont disponibles, à la discrétion de l'ordonnanceur. Il n'est pas possible, une fois un traitement lancé, de l'interrompre depuis le CPU, ce qui peut causer un blocage du programme de durée importante en cas de lancement synchrone. Plusieurs kernels sont susceptibles d'être lancés simultanément par l'ordonnanceur.
— **Récupération des résultats et libération des ressources**. Une fois l'exécution terminée, les données résultats stockées dans la mémoire globale de la carte peuvent être récupérées par le programme CPU, en vue de traitements supplémentaires, d'affichage, ou de stockage des résultats. La libération des ressources n'est pas automatique, et doit également être effectuée explicitement pour ne pas bloquer ou pénaliser de futures exécutions.

CUDA et OpenCL permettent l'utilisation de plusieurs cartes graphiques par un même programme. Dans ce cas, l'utilisation de soumissions asynchrones permet la gestion simultanée de plusieurs files d'exécution. L'utilisation efficace des ressources matérielles GPU requiert toutefois une connaissance de ce mécanisme de soumission et des optimisations mémoires effectuées à l'exécution.

Si les interfaces de programmation CUDA et OpenCL ne sont directement accessibles que depuis des programmes C ou C++, l'utilisation du calcul sur GPU n'est pas limitée à ces deux langages de programmation, grâce aux couches de liaisons avec des bibliothèques natives proposées

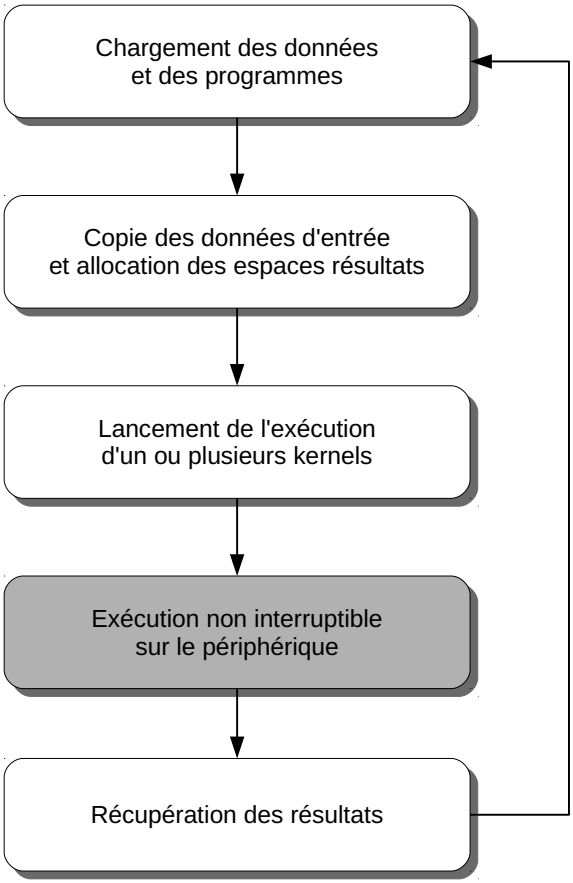

FIGURE 2.10 – Démarche d'exécution GPGPU

par de nombreux langages de plus haut niveau. Il est ainsi possible d'exploiter CUDA ou OpenCL en Java à l'aide des bibliothèques JCUDA [8] ou JOCL [9] pour ne citer que quelques solutions disponibles.

2.5.4 Synchronisation des opérations

Après avoir évoqué les deux modèles de programmation, nous allons maintenant présenter quelques aspects plus spécifiques d'OpenCL en termes de synchronisation des opérations.

Une première caractéristique d'OpenCL est le mode de lancement des opérations à réaliser de manière asynchrone. Après l'obtention d'un contexte d'exécution, une des premières opérations d'un programme OpenCL est de créer une ou plusieurs files d'attente dans lesquelles soumettre les différentes tâches à exécuter. La soumission de chaque tâche permet d'obtenir en retour un objet événement (cl_event), qui peut être utilisé pour construire un arbre de dépendances (DAG) entre les tâches à exécuter. Ces dépendances peuvent être utilisées pour s'assurer que la copie des données, les traitements et la copie des résultats auront lieu en séquence, ou encore pour chaîner plusieurs opérations, sans intervention intermédiaire du programme, comme illustré par la Figure 2.11.

OpenCL fournit également des opérations de synchronisation permettant d'attendre de manière bloquante la fin du traitement de la file d'attente ou d'une tâche particulière, pour synchroniser le

8. http ://www.jcuda.org
9. http ://www.jocl.org

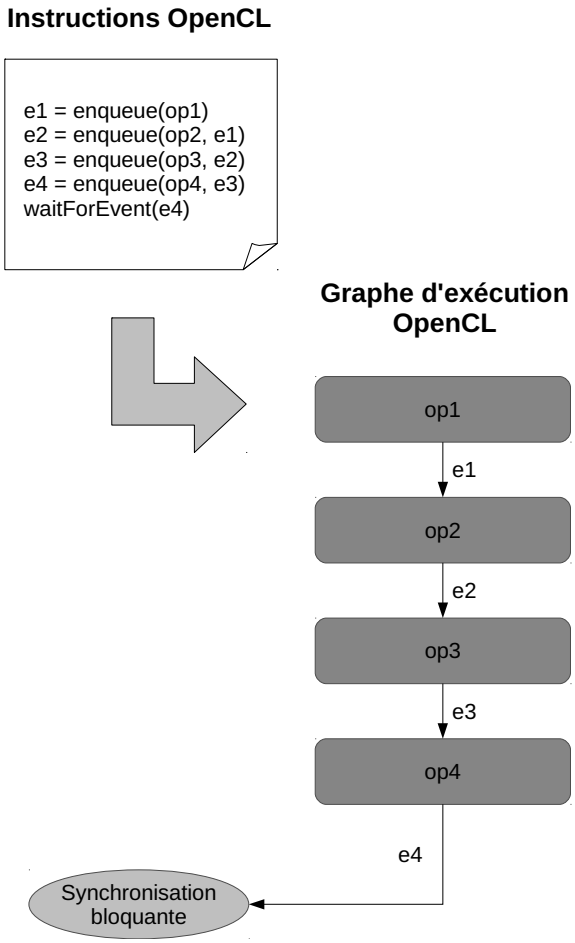

FIGURE 2.11 – Utilisation des dépendances pour gérer la synchronisation en OpenCL

flux du reste du programme. Si les options de suivi des performances sont activées, ces événements stockent également les informations de passage associées à chaque tâche : temps d'attente, temps d'exécution sur le périphérique.

Ce premier type de dépendance est extérieur au kernel OpenCL, et ne permet donc pas de contrôler l'accès aux données partagées par chaque thread d'exécution. Pour cela, un second type de synchronisation est utilisé, à base de barrières d'exécution. Celles-ci permettent au développeur de s'assurer que tous les threads concernés atteindront un point de l'algorithme au même moment. Elles sont indispensables dans de nombreuses parallélisations de traitements comme la multiplication de matrices, où chaque thread sera responsable du traitement d'une ligne avant de récupérer les informations de ses voisins pour la suite de l'opération. Dans ce cas, une barrière d'exécution permet de s'assurer que la première opération est bien terminée, de manière à éviter de fausser le résultat.

Une barrière d'exécution OpenCL peut être appliquée à l'ensemble (barrière globale) ou un groupe particulier de threads (barrière locale). Dans ce dernier cas, elle peut par exemple être utilisée pour protéger la création et l'utilisation d'un cache de données locales.

Un intérêt de ces files d'attente OpenCL est de permettre une gestion explicite des ressources pour chaque périphérique d'exécution : en créant plusieurs contextes d'exécution et files d'attente, il est ainsi possible de gérer directement le flux d'exécution de plusieurs matériels. En contrepartie, cette gestion est exclusivement du ressort du développeur : OpenCL ne fournit à l'heure actuelle

pas de mécanisme qui permette de répartir la charge de calcul de manière transparente sur plusieurs matériels distincts.

2.5.5 Regroupement des accès mémoire

La mémoire embarquée sur carte graphique se caractérise, comme nous l'avons vu dans la présentation de l'architecture matérielle, par une bande passante et des latences importantes. Si les mécanismes de recouvrement d'exécution évoqués dans la présentation du modèle de programmation permettent d'amortir l'impact de ces latences, ils n'améliorent pas l'utilisation de cette bande passante.

Pour cela, un autre mécanisme intervient sur GPU au niveau de chaque multi-processeur, le regroupement des accès mémoire (memory collapsing). L'objectif de ce mécanisme est de grouper les lectures de données proches en mémoire en requêtes de lecture de taille plus importante, tel qu'illustré par la Figure 2.12. Ces requêtes consolidées mettent à meilleure contribution la bande passante offerte par la mémoire et permettent également de regrouper les latences de chaque accès individuel.

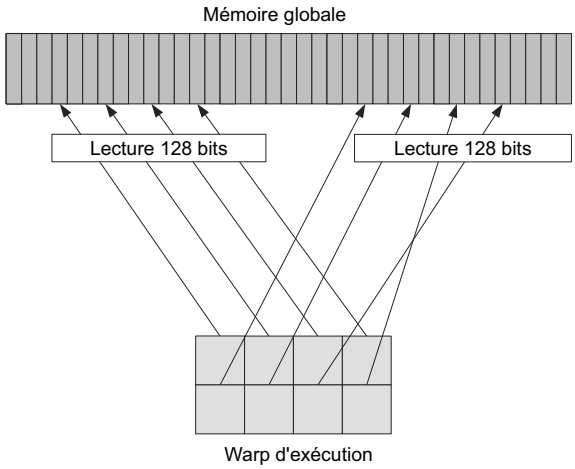

FIGURE 2.12 – Mécanisme de regroupement des accès mémoire sur GPU

La détection de ces accès contigus est dépendante, sur les architectures matérielles les plus anciennes, d'accès mémoires très réguliers (tâche n accédant à l'adresse $n + 1$ en mémoire). Les matériels plus récents permettent le regroupement d'accès moins ordonnés, ainsi que leur consolidation en requêtes de taille plus importante, pour maximiser l'utilisation de ce mécanisme sur des programmes plus irréguliers.

Il est important de prendre en compte ce mécanisme lors de la conception d'un programme sur GPU, particulièrement sur des architectures dénuées de cache L1 et L2. Une mauvaise exploitation de ces regroupements peut en effet multiplier le nombre de lectures mémoires nécessaires pour traiter les mêmes instructions et brider l'exploitation des coeurs d'exécution offerts par la carte graphique, indépendamment de tout gain lié à la parallélisation.

2.5.6 Bibliothèques d'exécution sur GPU

L'utilisation des ressources GPU ne requiert pas nécessairement une connaissance des concepts ou des modèles de programmation GPU : de nombreuses bibliothèques de traitement vectoriel ou

matriciel traditionnellement utilisées sur CPU sont maintenant disponibles pour cette architecture. Ces bibliothèques fournissent souvent une interface de programmation similaire à leur équivalent CPU, de manière à faciliter leur utilisation dans un programme parallélisé existant.

Voici quelques exemples de bibliothèques de ce type basées sur CUDA :

— **cuBLAS** est une implémentation du standard d'algèbre linéaire BLAS. Certaines opérations deviennent ainsi 6x à 17x plus rapides que leur équivalent CPU. Cette bibliothèque fait partie des bibliothèques optimisées GPU fournies par la société NVIDIA [10].
— **cuFFT**, pour CUDA Fast Fourier Transform library, permet le calcul de transformées rapides de Fourrier sur GPU. Cette bibliothèque est également fournie par la société NVIDIA.
— **CUSP** (C++ Templated Sparse Matrix Library) est une bibliothèque d'algèbre linéaire à faible densité. Elle permet également la manipulation et le traitement de graphes. Son utilisation repose sur le mécanisme des templates C++ pour permettre la génération de code GPU parallélisé en fonction des traitements demandés par l'utilisateur.
— **cuSparse** est une bibliothèque de traitements matriciels fournie par NVIDIA. Les formats de représentations de matrice creuses les plus courants (COO, CSR, CSC, ELL/HYB) et leur manipulation sont gérés de manière native en CUDA.

Si de nombreuses bibliothèques utilisent le modèle de programmation CUDA, des alternatives basées sur OpenCL existent également :

— **clMath (anciennement AMD APPML)** [amd] recouvre à la fois les opérations proposées par BLAS et le traitement des transformées de Fourrier. L'utilisation de cette bibliothèque est souvent combinée à cette de clMAGMA [CDD+13], qui fournit de nombreux solveurs linéaires et solutions de factorisation, réduction ou transformation de matrices.
— **clpp** [clp] est un autre projet fournissant des primitives de traitement en parallèle de structures de données. Ces traitements incluent notamment la recherche par préfixe (“scan”), le tri, ou la réduction de valeurs, de manière à faciliter la parallélisation de traitements plus complexes sur des structures telles que des graphes ou des arbres.
— **VexCL** [Dem] facilite également le traitement de matrices et de vecteurs en OpenCL. Cette bibliothèque est plus particulièrement orientée vers la réduction de la quantité de code nécessaire à la préparation et à la gestion des traitements sur GPU, au moyen de l'architecture objet C++.

Certaines bibliothèques supportent plusieurs plates-formes d'exécution comme OpenCL, CUDA ou OpenMP avec une même interface de programmation. C'est notamment le cas d'OpenCV [Bra00], pour la manipulation d'images en temps réel, ou ViennaCL [RWR10], pour la résolution de problèmes d'algèbre linéaire. Des comparaisons de l'utilisation et des performances de VexCL et ViennaCL sont disponibles dans la littérature [DARG12].

Une bibliothèque logicielle particulièrement intéressante est SnuCL [KSL+12]. Cette solution propose des opérations parallèles similaires à MPI pouvant être utilisées de manière transparente sur des clusters de CPU ou de GPU. Ces opérations permettent au concepteur de totalement se détacher de la plate-forme d'exécution, mais reste toutefois réservée pour l'instant aux langages C et C++.

10. https://developer.nvidia.com/gpu-accelerated-libraries

2.5.7 Optimisation de la soumission de tâches sur GPU

La possibilité de soumettre des tâches de manière asynchrone et de définir des dépendances entre ces tâches rapproche l'utilisation efficace du GPU des problématiques d'ordonnancement présentes dans la littérature.

Dans [TPO10], les auteurs analysent ainsi l'impact de différentes stratégies d'ordonnancement sur le temps nécessaire pour traiter un lot de tâches irrégulières. Pour cela, les traitements à réaliser sont groupées en kernels de manière dynamique, plutôt que directement soumises sur la plate-forme, pour optimiser l'occupation des ressources.

Plutôt que d'intervenir sur le regroupement des traitements, d'autres études reposent sur l'utilisation de l'historique d'exécution pour optimiser le passage des tâches sur GPU. Les auteurs de [ATN09] proposent ainsi un ordonnancement basé sur la mémorisation du temps d'exécution des tâches sur plusieurs architectures distinctes. De cette manière, les prochaines tâches peuvent alors être soumises sur la plate-forme permettant la terminaison la plus rapide. Les résultats obtenus sont très intéressants, mais très dépendants de cette prévisibilité pour assurer un bon remplissage des ressources. L'utilisation du processeur en parallèle du GPU permet de gagner 30% en performance par rapport à l'utilisation du seul GPU dans un autre article [GBHS11]. L'utilisation de l'historique dans cet article permet également d'assurer un remplissage à 80% des ressources, malgré la forte disparité en performance entre matériel CPU et GPU.

Les auteurs de [MGR+11] étudient également l'impact de la décomposition des traitements en un ou plusieurs kernels sur l'ordonnancement OpenCL à l'aide de la plate-forme SURF [BETVG08] adaptée à instrumentalisation de traitements d'images. Les mesures effectuées illustrent l'impact du nombre, de la durée et de la dimension de chaque kernel sur les performances obtenues. L'interface de soumissions asynchrone proposée par OpenCL est utilisée pour gérer finement les dépendances entre chaque kernel et obtenir les informations de temps précises de début et fin des traitements. La mesure du décalage entre temps de soumission et temps de lancement du kernel permet de déterminer le moment le plus pertinent pour lancer les prochaines requêtes, de manière à ne pas pénaliser l'exécution.

Un défi de l'optimisation du passage de tâches sur GPU est l'absence de contrôle sur l'ordonnanceur lui-même. Si certains articles [NSL+11] suggèrent des améliorations possibles en termes d'exécution des warps pour une meilleure occupation des ressources des coeurs d'exécution, il est difficile de savoir si ces améliorations sont ou seront reprises dans les implémentations CUDA ou OpenCL existantes. La mesure des performances est donc un outil indispensable pour guider l'optimisation de l'exécution sur GPU, même en présence d'outils proposés par des sociétés comme NVIDIA permettant de déterminer à priori les ressources utilisées par un programme GPU donné.

2.5.8 Bonnes pratiques de programmation sur GPU

Au vu de ces éléments sur l'architecture d'exécution GPU, de nombreux ensembles de recommandations existent sur la bonne manière de programmer sur GPU pour obtenir un programme efficace [cud09, Cor12, AG13].

Ces recommandations s'articulent autour de quatre objectifs principaux :

Minimisation des coûts de transferts

Cette minimisation peut être effectuée à deux niveaux, soit en réduisant le nombre de transferts effectués, soit en regroupant ces transferts.

La réduction du nombre de transferts n'est pas toujours possible, chaque donnée utilisée sur le périphérique devant être explicitement copiée avant son utilisation. Elle peut toutefois être obtenue en réduisant 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 temps total des transferts, en utilisant la bande passante importante fournie par l'interface PCI-Express pour mettre en commun plusieurs copies de données. Cette mise en commun est facilitée par les mécanismes de copies asynchrone proposés par les modèles de programmation GPU.

Optimisation des accès mémoires

L'optimisation des accès mémoires correspond à deux problématiques distinctes sur GPU : la minimisation des latences d'accès et la maximisation de l'utilisation de la bande passante mémoire.

La minimisation des latences d'accès est possible au moyen des mémoires spécifiques (constantes, globales, locales) proposées par l'architecture matérielle. L'utilisation de la mémoire partagée permet également d'éviter de récupérer à plusieurs reprises des données fréquemment utilisées par chaque traitement.

La maximisation de l'utilisation de la bande passante mémoire est dépendante du mécanisme de regroupement des accès mémoires de l'ordonnanceur et du l'ordre et de la proximité des données accédées en mémoire. Pour faciliter ce regroupement, il est recommandé d'utiliser les structures de données les plus régulières possibles sur GPU.

Maximisation de l'occupation

Un dernier point essentiel pour l'obtention de bonnes performances sur GPU est d'utiliser le plus efficacement possible les nombreux coeurs d'exécution offerts par l'architecture. Cette occupation dépend de trois paramètres :

Les ressources consommées par chaque thread. Chaque multi-processeur ne dispose que d'un nombre limité de registres, partagés de manière statique au lancement du programme. L'utilisation d'un trop grand nombre de registres par threads est susceptible d'empêcher l'utilisation de tous les coeurs d'exécution disponibles.

Le nombre de conditions présentes dans l'algorithme. Du fait des limitations en termes de branchements de l'architecture, l'utilisation de conditions impose l'évaluation des deux branches par le matériel, pour ne conserver ensuite que les résultats de la branche effectivement retenue. L'utilisation de nombreuses branches est alors susceptible de cause une réduction important de l'occupation des coeurs d'exécution.

Le nombre de threads total lancé. Comme évoqué précédemment, l'ordonnanceur d'exécution GPU est capable de dissimuler des latences d'exécution en attribuant automatiquement plusieurs

threads à un même coeur d'exécution. Ce mécanisme de recouvrement dépend de la présence de nombreux threads à exécuter pour être pleinement efficace.

2.6 Vers une convergence many-core

Tandis que l'exécution sur GPU devient toujours plus générique, avec l'apparition de mécanismes de cache ou l'implémentation de toujours plus d'opérations au niveau de la plate-forme, d'autres architectures dotées de grands nombres de coeurs (dites architectures many-core) se démocratisent également.

Les circuits programmables ou FPGA sont ainsi de plus en plus étudiés comme support d'exécutions parallèles économes en énergie et peu coûteux [WLL+, BRT11]. La puissance offerte par les circuits les plus récents permet notamment d'envisager l'utilisation de compilateurs et de modèles de programmation existants plutôt que d'une expertise individuelle des instructions pour réduire le temps et les coûts de développement. La société Altera propose ainsi depuis 2011 une implémentation d'OpenCL sur ses matériels FPGA [11].

Intel propose également depuis 2013 la première architecture many-core basée sur des CPU traditionnels, le Xeon Phi. Cette nouvelle plate-forme peut être utilisée en tant qu'accélérateur séparé, par le biais des modèles de programmation OpenCL ou OpenACC, ou directement comme un processeur multi-coeurs traditionnel à l'aide du modèle de programmation OpenMP.

La possibilité d'utiliser OpenCL sur ces trois plates-formes illustre la tendance actuelle à la convergence entre ces solutions many-core, de manière à permettre à un même programme de s'exécuter sur une grande variété de plates-formes matérielles. Cette convergence est également illustrée par celle des supports physiques, toutes ces nouvelles plates-formes étant basées sur l'utilisation de cartes connectées en PCI-Express à un ordinateur existant (Figure 2.6).

FIGURE 2.13 – Carte graphique NVIDIA Tesla

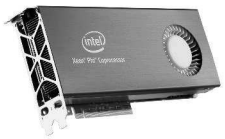

FIGURE 2.14 – Carte Intel Xeon Phi

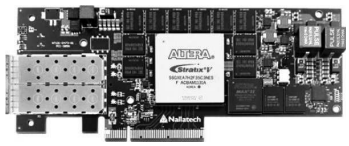

FIGURE 2.15 – Accélérateur FPGA Altera

2.7 Synthèse

Les cartes graphiques sont un type d'architecture matérielle permettant une exécution en mémoire partagée. Leur utilisation s'intègre dans le cadre d'un mouvement récent des problématiques de parallélisation vers les architectures many-core, proposant un très grand nombre de coeurs d'exécution sur un même matériel. Ce parallélisme matériel peut être exploité indirectement, par

11. http ://www.altera.com/products/software/opencl/opencl-index.html

le biais de bibliothèques de parallélisation proposant des opérations de haut niveau, ou via des plates-formes telles que OpenCL ou CUDA. Dans ce dernier cas se posent toutefois de nombreuses considérations d'implémentations pour obtenir un programme performant, exploitant les ressources offertes par le matériel de manière efficace.

L'utilisation de cette architecture d'exécution est déjà possible dans le cadre de nombreuses bibliothèques d'algèbre linéaire. Si ce type de problème n'est pas forcément directement utilisé dans les systèmes multi-agents, cette possibilité de proposer des traitements parallélisés en fait une piste intéressante pour l'accélération de programmes existants. Les cartes graphiques se distinguent également par leur disponibilité sur de nombreuses machines personnelles, par opposition à des solutions matérielles spécialisées comme les grilles ou les clusters, ce qui en fait une piste intéressante pour la parallélisation de systèmes multi-agents.

3

Parallélisation de systèmes multi-agents

Comme évoqué dans notre présentation des systèmes multi-agents, l'utilisation de simulations de tailles importantes peut rapidement engendrer des besoins en temps d'exécution et en mémoire importants. L'objectif du recours à la parallélisation est de résoudre ces problèmes en permettant l'accès à davantage de ressources mémoires ou d'exécution.

Comme nous venons de le voir, la démarche de parallélisation d'un programme implique une répartition de son exécution et parfois de ses données. Ce découpage est facilité dans le cas des modèles multi-agents par la décomposition du système en agents indépendants dotés de comportements et de données propres. La parallélisation d'une simulation multi-agents est un processus complexe en temps et en ressources, du fait de l'exécution naturellement synchrone de nombreux modèles sur la base de pas de temps ou d'événements. Ce synchronisme impose en effet de nombreux échanges de données en cours de simulation. La délégation d'une partie de la simulation à chaque hôte ou l'exécution en mémoire partagée du système sont donc généralement les parallélisations les plus aisées à réaliser [Ble09] car elles minimisent le nombre d'échanges nécessaires à la synchronisation. L'environnement représente alors la seule structure globale devant être partagée entre l'ensemble de l'exécution.

Dans les sections suivantes, nous présentons ces différentes approches de parallélisation de systèmes multi-agents pour souligner les problématiques qui devront être abordées dans notre proposition. Pour chacune, nous évoquons son principe ainsi que ses indications ou contre-indications. Nous voyons ensuite leur utilisation dans le cadre de plates-formes multi-agents parallèles qui déchargent le concepteur d'une partie importante de la gestion de cette parallélisation. Enfin, nous présentons des applications de ces techniques de parallélisation au GPU dans le cadre de divers types de systèmes multi-agents, avant de présenter FLAME-GPU, une première plate-forme multi-agents générique pour l'exécution sur GPU.

3.1 Stratégies de parallélisation

Il est possible de décomposer l'exécution d'un système multi-agents en trois grandes dynamiques [MFD09], illustrées sur la Figure 3.1 :

- Celle de l'environnement, qui définit l'évolution de l'espace simulé.
- Celle des comportements des différents agents présents dans le système.
- Celle de l'ordonnanceur, qui contrôle l'exécution et la synchronisation des deux dynamiques précédentes. C'est cet ordonnanceur qui détermine si l'exécution de la simulation est guidée par des pas de temps (time-driven) ou par des événements (event-driven).

La parallélisation du modèle multi-agents implique une intervention sur une ou plusieurs de ces dynamiques en conservant la cohérence du modèle découlant de leur interaction.

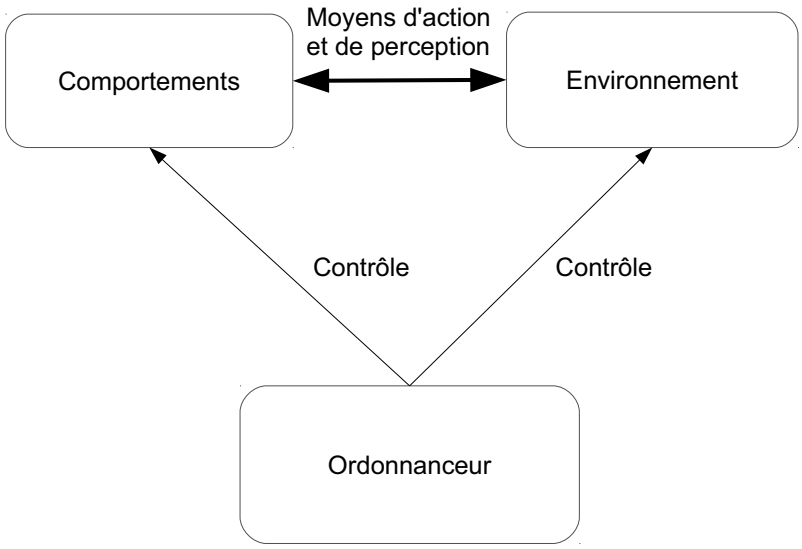

FIGURE 3.1 – Rôle de l'ordonnanceur multi-agents

3.1.1 Parallélisation de l'ordonnanceur

Une première manière de paralléliser le déroulement d'une simulation multi-agents est d'en lancer plusieurs instances séquentielles simultanément [BCC+11, CDFD10]. Cette parallélisation de lancement très simple, dite par lots, permet d'exploiter aisément plusieurs processeurs ou machines sans apporter aucune modification à la simulation. Elle est particulièrement intéressante pour tester de vastes ensembles de scénarios, en permettant d'obtenir une quantité plus importante de résultats dans la même période de temps.

Il est cependant important de noter que cette parallélisation par lots ne réduit ni le temps, ni la quantité de mémoire nécessaires à l'exécution de chaque simulation. Elle ne représente donc pas une solution dans le cas où les ressources mémoires locales empêchent le lancement d'une simulation. Elle ne permet pas non plus d'obtenir les premiers résultats plus rapidement qu'une implémentation séquentielle, et en augmente simplement la quantité.

Une approche de parallélisation de l'ordonnancement moins naïve consiste à répartir l'exécution de l'environnement et des agents entre plusieurs ordonnanceurs. Dans ce cas, un ordonnanceur est lancé pour chaque ressource d'exécution et collabore avec les autres ordonnanceurs pour traiter la simulation. Cette approche requiert toutefois la possibilité de pouvoir découper le système multi-agents en ensembles d'exécution distincts, de manière à minimiser les échanges de données et les synchronisations entre ordonnanceurs. Elle est proposée par plusieurs plates-formes multi-agents parallèles, comme nous le verrons plus loin.

Certaines simulations multi-agents [BMD+09] sont dites multi-échelles : dans ce cas l'évolution du modèle est gouvernée par plusieurs modèles agents représentant des échelles de simulation ou des aspects distincts d'un même système. Si chacun de ces modèles évolue de manière semi-indépendante, une approche de parallélisation intuitive consiste à confier l'exécution des différents aspects de la simulation à plusieurs acteurs, de manière à permettre l'exécution en parallèle de la simulation sans modifier fondamentalement chaque modèle.

3.1.2 Parallélisation des comportements

Une seconde approche de parallélisation consiste à paralléliser l'exécution des comportements des agents du modèle. Cette parallélisation peut être effectuée au niveau du système dans son ensemble, au niveau de l'agent, ou au niveau de chaque comportement.

Dans le premier cas, l'ensemble des traitements de même type dans le système sera exécuté simultanément. Pour des raisons d'équité, l'exécution des agents demeure synchronisée, ce qui permet d'assurer que tous les agents se déplacent, respirent, ou consomment de l'énergie en même temps. Ce type de parallélisation, consistant à lancer le même traitement pour un grand nombre d'agents, est bien adapté à une exécution sur GPU.

Dans le second cas, la parallélisation permet le traitement simultané de plusieurs types de comportements, de manière à gérer l'évolution de manière asynchrone à l'intérieur d'un pas d'évolution. Il est dans ce cas impossible de garantir que tous les agents progressent à la même vitesse dans leurs traitements : un agent est ainsi susceptible de se déplacer avant qu'un autre ne respire ou inversement. Cette seconde approche est sans doute l'une des plus réaliste pour des modèles où l'équité est introduite par la modélisation, mais elle est en pratique une des plus difficiles à mettre en place et à contrôler.

Dans le dernier cas, enfin, l'algorithme d'évolution des agents n'est pas modifié et seul le traitement du comportement lui-même est parallélisé. Cette dernière approche est particulièrement intéressante dans le cas d'agents effectuant des actions coûteuses, à même d'être parallélisées, dans le cadre de leur évolution. Des exemples d'actions candidates à la parallélisation sont la collecte de données dans un périmètre étendu, ou encore le calcul de déplacements complexes mettant en jeu de nombreux paramètres.

3.1.3 Parallélisation de l'environnement

Un dernier axe de parallélisation concerne la dynamique de l'environnement. En fonction du système multi-agents décrit, cette dynamique peut être inexistante, si l'environnement est utilisé comme un simple repère spatial comme dans le cas des modèles d'essaims, ou au contraire très complexe comme dans le cas de modèles comme les fourmis. Si le temps de traitement de la mise à jour de l'environnement représente une portion significative du temps d'exécution du modèle, il devient dans ce cas intéressant de paralléliser cette mise à jour.

La parallélisation de l'environnement est également souvent requise parce qu'il s'agit de la structure de données dont taille est la plus importante dans le modèle. Dans ce cas, la parallélisation ne vise plus uniquement l'obtention de meilleures performances, mais également la possibilité de simuler des environnements de taille plus importante. Le découpage de l'environnement implique généralement la répartition des agents présents dans le modèle, de manière à pouvoir conserver les portions d'environnements et leurs agents associés sur les mêmes unités d'exécution. Ce type de partitionnement est également proposé par de nombreuses plates-formes agents parallèles.

3.2 Plates-formes multi-agents

La parallélisation de simulations multi-agents rencontre de nombreuses difficultés liées à l'adaptation d'un programme en mémoire partagée ou distribuée. Il est ainsi possible de citer la nécessité d'identifier les sections parallèles de l'algorithme et de synchroniser l'accès aux données partagées en mémoire partagée, ou la décomposition explicite des données et de l'algorithme ainsi

que la prise en compte des échanges et des communications requises par l'exécution en mémoire distribuée.

Heureusement, le formalisme multi-agents propose une décomposition du système en agents ou en environnement dont la gestion et les échanges peuvent être traités par des plates-formes multi-agents spécialisées. Cette prise en charge d'une partie de l'exécution permet également à de telles plates-formes de faciliter cette démarche de parallélisation.

3.2.1 Madkit

MadKit (Multi-Agent Development Kit) [1] est une plate-forme générique de développement et d'exécution de systèmes multi-agents [GF00a] réalisée en Java. Elle est développée au sein du LIRRM [2].

Le modèle AGR (Agent, Groupe, Rôle) [Gut01] est à la base des modèles et de l'architecture de la plate-forme, dont les différents services sont implémentés par des agents pour un maximum de flexibilité. Le noyau de Madkit se caractérise par sa légèreté et n'assure que les services nécessaires à la mise en place de ces agents : la gestion des groupes et des rôles, un ordonnancement synchrone, et une infrastructure d'échange de messages entre agents locaux.

Par défaut MadKit associe un thread à chaque agent autonome présent dans le système. Pour éviter l'utilisation de milliers de threads dans le cas de nombres importants d'agents, il est possible de créer des agents découplés de l'ordonnanceur, gérés et mis à jour par un ou plusieurs agents observateurs associés à des threads d'exécution. Ce modèle multi-thread permet une parallélisation aisée des agents en mémoire partagée.

La parallélisation du modèle en mémoire distribuée est rendue possible par la possibilité de lancer plusieurs noyaux MadKit et de surcharger le service d'échange de messages pour permettre à toutes ces simulations de communiquer [GF00b]. Cette surcharge d'un service système est permise par la présence de points d'accroche (hook) permettant de surveiller les messages échangés ou de remplacer un service particulier. Un agent permettant ce fonctionnement nommé *Communicator* est fourni par défaut avec MadKit. Chaque noyau exécute alors une instance de cette agent *Communicator* pour gérer l'échange des messages. Un nouveau mécanisme de distribution permettant le dialogue par le biais d'un agent réseau sans connaître l'emplacement des instances distantes, *NetComm* [RHK06], a depuis également été proposé. Ces mécanismes de communication permettent soit une distribution maître-esclave, où un noyau possède le modèle de référence mis à jour par les agents distants, soit sur une duplication du modèle sur chaque instance. Ces deux approches de distribution sont étudiées plus en détail dans [MBF02].

3.2.2 JADE

La plate-forme JADE [BCG07] est une plate-forme multi-agents développée en Java par le groupe de recherche CSELT (partie de Gruppo Telecom). Elle permet la réalisation de systèmes multi-agents conformes à la norme FIPA [fip].

Les services FIPA sont fournis directement par la plate-forme JADE, ce qui rend le support de la norme transparent pour le concepteur de modèle.

L'intégration d'un nouvel agent dans un modèle JADE en cours d'exécution est décomposée

1. http://www.madkit.org
2. Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

en plusieurs étapes :

— L'enregistrement de l'agent auprès de la plate-forme agent.
— L'attribution d'un nom et d'une adresse unique à l'agent.
— L'utilisation des services de recherche et de communication pour s'interfacer avec les autres agents.

Ce découplage des agents de leur emplacement physique rend possible la migration d'agents entre machines en cours d'exécution, au moyen d'un ensemble d'outils prenant en charge le déploiement et le suivi du modèle. La distribution est assurée par l'utilisation des threads et une interface de communication proposée par JADE. Cette interface repose sur le protocole RMI proposé par Java pour communiquer entre instances distantes [VQC02].

JADE ne prend en charge que la création, l'évolution et la communication entre agents, et ne propose aucune structure de données pour la représentation de l'environnement ou d'autres données partagées. La représentation de ces éléments est néanmoins possible sous forme d'objets indépendant ou d'agents spécifiques dans la simulation. Il est ensuite possible d'utiliser des messages pour envoyer ou recevoir des informations sur ces structures partagées.

En prenant en compte toutes les communications, JADE facilite l'utilisation d'une architecture en mémoire partagée ou distribuée pour la simulation multi-agents. Cette plate-forme se limite toutefois à ce rôle, et ne propose pas de mécanisme de synchronisation ou de partage automatique des données entre instances d'exécution : cette gestion reste de la responsabilité du concepteur du modèle, en utilisant les structures de données fournies par le langage Java.

3.2.3 FLAME

FLAME [3] [HCS06] est un générateur de simulations multi-agents parallélisées. Il se base pour cela sur la description des modèles sous la forme de machines à états (X-Machine) en XMML, version étendue du XML. Cette description abstraite permet de découpler l'exécution du système multi-agents de toute plate-forme d'exécution spécifique.

La description d'un modèle FLAME repose sur la spécification d'un état initial, d'un ensemble d'états intermédiaires et d'un ou plusieurs état finaux. Le passage entre ces états est décrit sous forme de fonctions de transition, exécutées pour chaque agent à chaque pas de temps de la simulation. Une itération, ou pas de temps, est définie comme la fenêtre de temps nécessaire à chaque agent pour progresser de son état initial à l'un des états finaux du graphe de transitions. Ce processus est reproduit à chaque itération.

En parallèle de ces états représentant les stades d'exécution de chaque agent, FLAME associe à chaque agent une mémoire pouvant contenir des variables lues et modifiées par les différentes fonctions de transition.

La communication entre agents est assurée par la possibilité d'envoyer et de recevoir des messages au niveau de ces mêmes fonctions de transition. Leur transmission est réalisée de manière synchrone, pour garantir la réception simultanée de chaque message par tous ses destinataires : cette synchronisation est particulièrement importante lors de l'exécution d'un modèle sur une architecture distribuée HPC pour permettre qu'aucun agent ne soit favorisé ou défavorisé par son ordre de passage. Les messages sont distribués à l'ensemble des agents du modèle par la bibliothèque *Libmboard* basée sur MPI pour les échanges de messages [KRH+10]. Il est ensuite possible à chaque agent de filtrer les seuls messages le concernant.

3. FLexible Large-scale Agent-based Modeling Environment

La modélisation d'un modèle en FLAME est décomposée en quatre étapes :

— La description de chaque agent et de sa fonction.
— La description des états correspondant à leur évolution à chaque itération
— L'identification des variables utilisées pour le déclenchement et dans le traitement de chaque fonction de transition définie dans le modèle.
— L'identification des messages émis ou reçus par ces fonctions de transition.

Ce processus peut être représenté sous la forme d'un diagramme de transition. La Figure 3.2 illustre une représentation possible d'un modèle d'essaim (Swarm) avec FLAME.

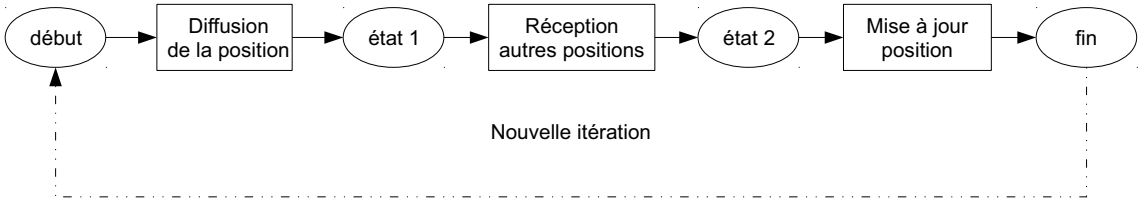

FIGURE 3.2 – Représentation conceptuelle d'une itération de modèle Swarm

L'implémentation du modèle économique européen EURACE, mettant en jeu des agents intervenants sur plusieurs marchés économiques, prévoit d'utiliser ces mécanismes [DvdHD08].

La plate-forme FLAME permet de s'abstraire totalement de la démarche de parallélisation en mémoire partagée ou en mémoire distribuée, en prenant en charge la totalité de la génération de la simulation. Cette abstraction dépend toutefois d'une description très fine du modèle et de ses interactions par le concepteur, qui contraint fortement la définition des modèles.

3.2.4 Repast HPC

Repast HPC [CN11] est une bibliothèque dédiée au calcul sur architectures hautes performances. Elle propose une implémentation des concepts fondamentaux de RepastSimphony sur des architectures mémoire distribuées et plus particulièrement sur les clusters de calculs. Le développement de modèles agents avec Repast HPC peut être effectué directement à l'aide des composants de la bibliothèque ou en manipulant des concepts d'emplacements et de tortues inspirés de Logo.

L'implémentation des agents est réalisée sous forme d'instances de classes C++ encapsulées dans un *Contexte* représentant leur environnement. Leur organisation dans le modèle est assurée par la définition de *Projections*. Une projection grille place ainsi les agents dans une structure grille où chaque agent correspond à une cellule, tandis qu'une projection réseau permet la mise en place de relations entre agents. Une simulation Repast HPC est ainsi composée d'agents, d'au moins un contexte, et de zéro ou plusieurs projections.

La distribution des agents en Repast HPC est basée sur un parallélisme à mémoire distribuée. Les agents du modèle sont répartis entre plusieurs processus responsables du traitement de leurs agents locaux. L'interaction avec un agent distant requiert sa copie en mémoire locale, la modification de cette copie, puis sa synchronisation avec l'agent original, pendant que l'exécution de l'agent distant est suspendue [rep]. Pour faciliter la gestion de ces copies, chaque agent est identifié de manière unique par trois informations : un identifiant attribué par l'utilisateur, l'index de son processus de lancement et son type. Chaque agent stocke également l'index du processus

l'exécutant actuellement.

La synchronisation et l'échange des agents entre processus sont assurés via le protocole de communication MPI [rep] par le biais de son implémentation BoostMPI [4].

Ce mécanisme de découpage permet à Repast HPC de prendre en charge de nombreuses problématiques de la parallélisation en mémoire distribuée et en particulier la copie et l'exécution des agents présents sur les différents noeuds d'exécution.

3.2.5 D-MASON

La plate-forme D-MASON [CCC+12] est une version parallèle de la bibliothèque MASON, ajoutant une couche supplémentaire permettant la distribution de la simulation en mémoire distribuée sur des machines hétérogènes.

La distribution de la simulation en D-MASON est basée sur trois blocs fonctionnels, un gestionnaire, des travailleurs (**workers**) correspondant à des threads Java et des communications. Le rôle de gestionnaire est assuré par une application maîtresse qui prépare la simulation et gère ensuite son déroulement en pas de temps synchrones en coordonnant les différents processus travailleurs.

Cette répartition des tâches repose sur le partitionnement de l'espace à simuler en régions pouvant être assignées à un worker particulier. Chaque worker est ensuite responsable de l'exécution des agents présents dans sa région, ainsi que de la synchronisation des traitements ou de la migration des agents entre régions. Les échanges requis pour ces opérations sont gérés par le biais de JMS [CCM+11], une interface de programmation permettant d'envoyer et de recevoir des messages asynchrones entre composants Java.

Cette répartition automatique de l'environnement par la plate-forme, associée à celle de l'ordonnancement et des traitements, permet la gestion des trois approches de parallélisation de systèmes multi-agents. D-MASON se caractérise de manière générale par la volonté d'introduire la distribution à tous les niveaux du système, plutôt que de se focaliser uniquement sur les performances, pour résoudre les limitations en ressources, en particulier mémoires, de manière transparente pour le concepteur.

3.2.6 Pandora

Une dernière plate-forme permettant la distribution d'un système multi-agents sur plusieurs noeuds de cluster est Pandora [pan]. Cette plate-forme permet le prototypage rapide de modèles à l'aide du langage de programmation Python, ou la réalisation de modèles plus complexes à l'aide de C++. Ces deux langages d'implémentation partagent la même interface de programmation et les mêmes concepts, de manière à faciliter l'adaptation de modèles entre les deux syntaxes. Il est plus particulièrement conçu pour la simulation de milliers d'agents dans un espace géographique.

La distribution en mémoire partagée des systèmes Pandora repose sur la distribution de portions de l'environnement et des agents sur chaque noeud du système à l'aide de OpenMP et de MPI [ASÁ01]. La parallélisation locale de la simulation est basée sur l'observation par ses concepteurs d'une décomposition standard du cycle agent dans de nombreux modèles multi-agents :

— Évaluation de l'environnement et des stimulis.
— Prise de décision quant à l'action à effectuer.
— Réalisation de l'action.

4. http://www.boost.org/

— Mise à jour des variables internes.

Pandora permet la parallélisation de l'évaluation de l'environnement et de la prise de décision de manière automatique avec OpenMP. La suite des traitements est séquentialisée pour garantir la cohérence des mises à jour du modèle. L'originalité de cette plate-forme réside dans la gestion automatique de la distribution et de la copie des informations situées à la frontière de deux portions voisines de l'environnement à l'aide des champs d'action [WRC12].

3.3 État de la simulation multi-agents sur GPU

La simulation de systèmes multi-agents met en jeu la parallélisation, qu'elle soit en mémoire partagée ou distribuée. Cette démarche de parallélisation peut être effectuée manuellement ou à l'aide de plates-formes multi-agents parallélisées. Dans les deux cas, elle implique la répartition des traitements du modèle sur plusieurs coeurs d'exécution, que ce soit au niveau de l'ordonnancement, de l'environnement, ou des agents.

L'utilisation de nombres importants de coeurs CPU requiert toutefois le recours à des environnements HPC spécialisés, grilles ou clusters de calculs. Ces clusters de calculs ne sont pas forcément à la portée de tout chercheur, soit pour des raisons techniques, soit pour des raisons financières. Dans ces circonstances, les GPU sont une alternative intéressante pour exécuter des nombres importants d'agents, avec leur coût réduit et leurs centaines de coeurs d'exécution. Ils présentent également l'intérêt d'être présents dans la totalité des machines personnelles actuelles.

Dans ce chapitre, nous présentons tout d'abord les différents domaines de simulation multi-agents ayant déjà fait l'objet d'adaptations sur GPU. Nous évoquons ensuite FLAME-GPU, une première plate-forme d'exécution générique sur GPU, et en montrons les caractéristiques et les limites permettant d'envisager d'autres approches d'exécution de systèmes multi-agents sur GPU.

3.3.1 Parallélisations indépendantes

De nombreux systèmes multi-agents ont déjà été parallélisés de manière indépendante sur architecture GPU, dans le cadre d'optimisations de modèles existants ou d'étude de cette plate-forme. Ces adaptations peuvent être rassemblées en grandes thématiques, souvent associées à la modélisation agent dans la littérature. La diversité de ces thématiques se retrouve dans les approches d'adaptations retenues, mais permet toutefois de dégager quelques grandes tendances de modélisation sur GPU.

Automates cellulaires

Les automates cellulaires sont un premier type d'implémentation utilisée pour les systèmes multi-agents se prêtant naturellement à une parallélisation du fait de leur découpage en grille et de l'application du même algorithme d'évolution à chacune des cellules de cette grille. Une telle exécution est dite SPMD (Single Program, Multiple Data ou "Un seul programme, de nombreuses données") et favorise le découpage du traitement sur de multiples unités de calculs. L'utilisation de grilles pour le stockage du système facilite également le partage des données en un ou plusieurs lancements, dans le cas où les besoins en mémoire du modèle seraient trop importants pour un seul GPU.

Le Jeu de la Vie est un exemple d'automate cellulaire souvent utilisé pour illustrer ce type de

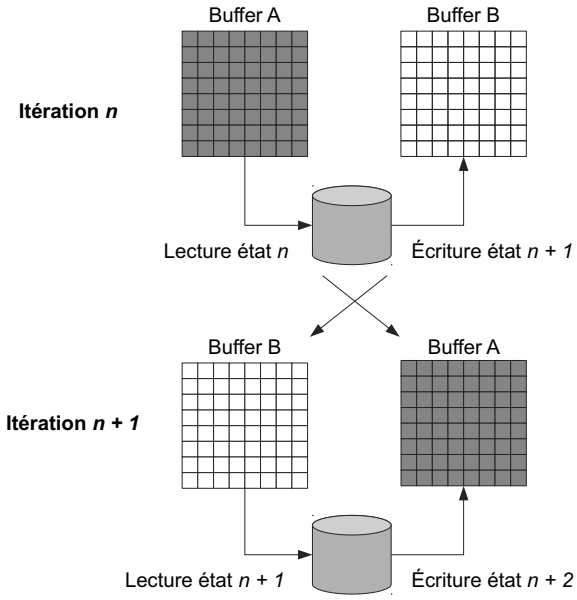

FIGURE 3.3 – Concepts de double bufferisation des données entre itérations

parallélisation, du fait de ses règles simples d'évolution et de l'indépendance de l'évolution de chaque cellule. La seule synchronisation de l'exécution du modèle a lieu entre chaque itération, de manière à assurer la progression de tous les agents au même rythme et la prise en compte des modifications effectuées à l'itération précédente. Cette synchronisation peut être implémentée au moyen d'une barrière d'exécution ou en découpant le traitement en plusieurs lancements. Un découpage en plusieurs lancements permet alors d'utiliser les mécanismes de dépendances fournis par OpenCL et CUDA pour inclure d'autres opérations dans le déroulement de la simulation, telles que l'échange d'une grille d'entrée et d'une grille de sortie (Figure 3.3) entre chaque itération.

Un autre exemple d'automate cellulaire parallélisé est SugarScape, un modèle décrivant l'évolution de populations d'individus en fonction de la répartition de glucose dans un environnement en grille. Une parallélisation en CUDA proposée pour ce modèle [DLR07] repose sur l'utilisation de textures graphiques pour représenter cette grille, en tirant parti de la possibilité d'utiliser les canaux de couleurs pour stocker les propriétés de chaque cellule dans un seul pixel, et ainsi de représenter l'environnement dans un seul objet texture. La mise à jour du modèle est effectuée en associant une tâche GPU à chaque cellule de la grille. L'utilisation, comme dans le cas du Jeu de la Vie, d'une texture d'entrée et d'une texture de sortie permet de gérer plus facilement l'affichage de la simulation pendant les traitements, et de contourner la limitation des accès en lecture seule ou écriture seule à ces textures au sein d'un même kernel. Cette impossibilité de lire et d'écrire dans la même texture au cours du même lancement empêche, contrairement au Jeu de la Vie, l'utilisation d'une simple barrière d'exécution entre deux itérations.

Cette parallélisation de SugarScape met en avant les problèmes rencontrés par le stockage des agents dans une structure en liste indépendante. Le décès et l'apparition de nouveaux agents nécessitent des mécanismes efficaces de mise à jour de cette liste sur GPU, et en particulier de recherche d'emplacements libres pour stocker les nouveaux individus. La solution retenue dans cette adaptation est de rechercher un emplacement libre décalé de n cases par rapport à l'emplacement de chaque agent en attente de reproduction, et de faire varier ce décalage jusqu'à ce qu'une case libre se dégage pour toutes les reproductions en attente. Cette approche stochastique repose sur l'observation qu'une solution est généralement trouvée en quelques itérations et permet d'éviter de synchroniser chaque reproduction, au prix de la perte d'un peu de temps de calcul.

Ces deux exemples illustrent l'importance des traitements et des échanges de grilles pour paralléliser ce type d'implémentation.

Agents indépendants

Un autre modèle souvent parallélisé sur GPU est celui des fourmis [CGU+11, UIN12], dont la mise à jour peut être décomposée en deux processus distincts, comme nous l'avons vu dans la description du modèle :

— La mise à jour de l'environnement, et en particulier la simulation de la diffusion et de l'évaporation au cours du temps des phéromones déposées par les fourmis.
— La gestion du déplacement des fourmis (agents) elles-mêmes, au vu de ces données environnementales et de leur état actuel, à la recherche de nourriture ou en train de rapporter des ressources. Ce second état conditionne le dépôt ou non de phéromones par l'individu.

La parallélisation de ce modèle sur GPU pose le problème de synchroniser l'exécution de ces deux processus à chaque itération. L'utilisation de textures impose, comme dans le cas du modèle SugarScape, le recours à plusieurs kernels d'exécution distincts pour pouvoir accéder à ces textures alternativement en écriture, pour la mise à jour des quantités de phéromones, puis en lecture au moment de la détermination du comportement de chaque individu. Cette parallélisation met l'accent sur l'importance de la synchronisation, et plus généralement de la possibilité de mettre en place un graphe de dépendance, pour l'exécution de la simulation en parallèle sur GPU.

Un autre exemple de modèle multi-agents avec des agents indépendants est le mouvement de volées de différentes espèces d'oiseaux dans un espace en deux dimensions [Wei13]. Ce mouvement, inspiré du modèle Boids décrit par Craid Reynolds en 1987 [Rey87], illustre l'apparition de comportements émergents au niveau de volées d'oiseaux à partir de règles simples assignées à chaque individu, aboutissant à un compromis de déplacement à chaque itération. Les règles utilisées sont les suivantes :

— **Séparation** : choix de la direction de manière à éviter une trop grande agglomération avec les oiseaux voisins.
— **Alignement** : choix de la direction de manière à adopter la même direction de déplacement que les oiseaux voisins.
— **Regroupement** : choix de la direction de manière à se rapprocher du centre de masse des oiseaux à proximité.

Les oiseaux décrits sont positionnés dans un espace continu et il n'existe donc plus cette fois aucune grille sur laquelle baser un découpage pour GPU. Le positionnement de chaque agent dans l'environnement est à la place indiqué par un jeu de coordonnées associé à chaque individu. Le mouvement de chaque individu est également stocké dans ses propriétés sous la forme d'un vecteur. Dans ce cas, le découpage d'exécution retenu n'est plus basé sur une sous-division de l'environnement, mais sur l'association d'une tâche de traitement à chaque individu. La synchronisation entre les individus est effectuée à l'aide de barrières d'exécution dans un même lancement de kernel d'exécution. Pour minimiser le nombre et donc les coûts d'accès à la mémoire globale, les oiseaux sont regroupés en ensembles (*clusters*) dont les positions et les vitesses sont recopiées en mémoire partagée.

Un dernier modèle d'agent concerne la simulation de la propagation de la tuberculose dans les cellules du système immunitaire [DLMK09]. Cet exemple réintroduit un découpage de l'environnement en espace discret, sous la forme d'une grille d'agents représentant alternativement des macrophages ou des lymphocytes-T. Chacun de ces agents est associé à un automate à états finis

déterminant les états accessibles à chaque agent en fonction de son état actuel. La parallélisation du modèle est effectuée par le biais de plusieurs kernels lancés à des fréquences différentes. Un premier kernel, exécuté cent fois plus fréquemment que les autres traitements, gère ainsi la diffusion des marqueurs chimiques dans l'environnement. Un second kernel gère la reproduction des bactéries dans l'environnement de simulation, tandis que quatre autres kernels gèrent la mise à jour de l'état de chaque macrophage et lymphocyte T, puis leur déplacement et le recrutement de nouveaux individus. Cet exemple met en avant le nombre importants d'opérations différentes pouvant être présentes dans un même modèle agent, et l'importance de pouvoir exécuter ces traitements à la suite sur GPU, après initialisation et copie des données depuis le CPU.

Recherche de chemins (pathfinding)

Un autre domaine d'application multi-agents souvent rencontré dans les exemples de parallélisation sur GPU est la recherche de chemins en deux ou trois dimensions, dans des environnements continus ou discrets [FSN09, GCK+09]. La popularité de ce type de traitement s'explique par l'existence d'une forte demande au niveau de ce type d'algorithme dans l'industrie vidéo-ludique (déplacement d'unités) et les simulations urbaines ou de transport.

Ces modèles agents considèrent généralement un individu associé à une position dans l'espace, et éventuellement une taille, se déplaçant dans un environnement constitué de points de passage et d'obstacles. Dans le cadre d'un découpage de l'environnement en grille, des algorithmes de recherche de chemin dans des grilles et des graphes comme l'algorithme A* [HNR68] peuvent alors être utilisés pour déterminer le plus court chemin vers un point destination. Ce type de modèle implique encore une fois de disposer de facilités de traitement de grilles sur GPU, ainsi que la possibilité de mettre à jour ces grilles de manière concurrente pour gérer la cohérence des déplacements de chaque agent. Cette grille peut être intrinsèque au repère de positionnement, ou être appliquée à chaque itération pour discrétiser des positions réelles, de manière à permettre l'application des algorithmes de recherche de plus court chemin.

Un premier exemple de modèle parallélisé concerne le déplacement des unités dans des jeux de stratégie sur la base de champs de potentiels [SFF+10]. L'utilisation de champs de potentiels, et donc de vecteurs de déplacement, permet dans ce cas non seulement d'obtenir des chemins efficaces (distance) mais également plus naturels en adoucissant notamment les courbes ou en limitant les comportements peu réalistes (virage à 90°) déclenchés par la détection soudaine d'un obstacle. La difficulté de parallélisation de ce modèle est la nécessité de prendre en compte l'ensemble de l'espace de simulation pour déterminer le mouvement calculé par chaque tâche GPU. Pour limiter le nombre d'accès mémoire à effectuer, une carte locale des obstacles et de la direction projetée de son objectif est préalablement créée pour chaque agent et un vecteur associé à chacun de ces éléments. Le déplacement de l'agent est ensuite calculé à partir de la combinaison de ces vecteurs, puis projeté sur l'environnement de simulation. L'utilisation du GPU permet ici de considérer ces cartes locales et ces opérations vectorielles en parallèle pour chaque agent, plutôt que d'effectuer un parcours linéaire de chaque individu intégrant un autre parcours des cellules voisines à cet individu.

Une simulation de déplacement de piétons [RR08] se base sur le découpage des données de chaque agent en canaux de textures. Cet exemple met également en valeur, dans le cas de la recherche de chemins, l'utilité du découpage des données en structures d'entrée et structures de sortie déjà rencontré pour les automates cellulaires.

Le portage de ce type de simulation sur GPU, notamment par Bleiweiss [Ble09], a également amené des réflexions sur d'autres architectures vectorielles [GCK+09] telles que SSE, mettant en

avant le potentiel de ce type de parallélisation sur architecture many-core massivement parallèle de manière générale.

Il est également possible de citer, pour une réflexion approfondie sur la comparaison de différentes heuristiques permettant de résoudre ce type de problèmes sur GPU, le mémoire de thèse de A. Delévacq [Del13]. L'auteur commence pour cela par établir une présentation des métaheuristiques parallèles existantes sur GPU, avant d'en proposer une taxonomie orientée autour de deux axes, en fonction du niveau d'utilisation du GPU dans la métaheuristique d'une part (population, solution ou élément), et en fonction de l'utilisation faite de chaque type de mémoire proposé par le modèle de programmation d'autre part (globale, texture, constante, partagée, registres). Une fois cette taxonomie posée, l'auteur compare alors deux approches différentes de la résolution du problème du voyageur de commerce, tout d'abord à base de colonies de fourmis, puis à base de recherche locale (RL) de solutions. L'approche à base de colonies de fourmis repose sur l'utilisation d'individus, les fourmis, parcourant le graphe selon une heuristique de type min-max, de manière à construire progressivement des listes solutions. L'approche RL repose, quant à elle, sur l'amélioration d'une solution existante en évaluant l'impact de modifications locales à certaines positions de la liste. Elle est ainsi susceptible de compléter d'autres métaheuristiques pour la recherche de solutions efficaces. Les résultats obtenus montrent des gains de performance significatifs avec une implémentation GPU, avec un impact sur la qualité des solutions demeurant entre -1.33% et 1.64%. Au contraire, des dégradations significatives sont observées sur la parallélisation de l'approche RL, liées aux découpages en termes de synchronisation rendues nécessaires par l'adaptation sur GPU. Ce mémoire met en évidence l'importance des adaptations dans ce modèle de programmation et celle de la validation par des mesures de leurs conditions d'utilisation.

Réflexions sur des exécutions hybrides GPU et CPU

La parallélisation de systèmes multi-agents a également donné lieu à des réflexions plus génériques, orientées sur la distribution de la charge en général sur des plates-formes parallèles [CCDCS11], ou sur le cas plus particulier de clusters de GPU et de machines multi-coeurs.

En particulier dans [APS10], l'auteur propose la comparaison d'une implémentation à base de threads et de CUDA du même modèle agent. Pour minimiser le nombre d'échanges nécessaires dans le cas de systèmes multi-agents, où des portions de l'environnement sont réparties dans de multiples copies distantes, l'article propose la mise en place d'un système de zone-frontière au niveau du découpage des données. Ces zones frontières situées autour des données directement utilisées par les agents exécutés en local permettent de prendre en compte la diffusion des erreurs de mise à jour liées à l'exécution indépendante de chaque portion du modèle. La largeur de ces frontières permet de déterminer le nombre de cycles pouvant être exécutés avant que les divergences n'impactent d'autres portions de l'environnement simulé, et donc le temps pendant lequel une synchronisation peut être reportée sans impact sur le résultat de la simulation générale. En réduisant le nombre d'échanges de données nécessaires, ce système d'isolation permet de maximiser le temps passé en calcul pour chaque tâche et donc d'améliorer les performances obtenues. L'implémentation GPU proposée est basée sur le modèle de programmation CUDA et des blocs mémoires pour chaque agent ou groupe d'agent. L'utilisation d'OpenCL est toutefois prévue, pour pouvoir tirer parti d'une plus grandes variété d'architectures. Cet article met en évidence l'impact important que peut avoir la fréquence de synchronisation des données sur les performances obtenues, en variant la largeur des frontières autour de la copie de l'environnement de chaque agent.

3.3.2 Une plate-forme multi-agents orientée GPU : FLAME-GPU

Si l'utilisation du GPU est à l'étude pour des plates-formes comme JADE [ZG12] ou Turtle-Kit [Mic13], seule FLAME-GPU [Ric11], une extension pour la plate-forme FLAME [CGH+12], permet à l'heure actuelle l'utilisation du GPU pour l'ensemble du modèle.

FLAME-GPU permet la génération d'implémentations GPU de modèles FLAME avec un minimum d'adaptations, de manière à afficher ou exploiter plus facilement de grandes quantités d'agents. Cette utilisation transparente du GPU est rendue possible du fait de la décomposition très fine de la simulation en états et en traitements. La syntaxe XXML, proposée par FLAME, est également étendue pour permettre d'indiquer le nombre maximum d'agents présents dans le modèle, de manière à pouvoir allouer à l'avance les structures de données sur GPU. Ces extensions incluent également des directives permettant de choisir les algorithmes de traitement des fonctions de transition à utiliser en CUDA.

L'application d'une fonction de transition sur GPU aux agents d'un modèle FLAME est décomposée en deux étapes :

— Un premier kernel vérifie les pré-conditions d'application de la fonction de transition pour chaque agent du modèle. Si les conditions d'application sont remplies pour cet agent, il est ajouté à la liste des agents devant être traités.
— Une fois ce filtrage effectué, la fonction de transition est appliquée à tous les agents en attente de traitement.

Cette décomposition facilite le lancement de traitements réguliers sur le GPU dont l'exécution peut être ensuite parallélisée aisément par l'ordonnanceur CUDA.

FLAME-GPU a déjà établi les gains en performance pouvant être obtenus dans plusieurs types de systèmes multi-agents [RWCR10, KRR10] mais impose, tout comme FLAME, un cadre très strict de modélisation. Ce cadre est basé sur la décomposition du modèle en chacun de ces comportements, des données et de toutes leurs interactions, pour permettre à la plate-forme de générer le programme capable de les exécuter. Son utilisation n'est donc pas envisageable dans le cadre d'une plate-forme multi-agents ou d'une implémentation existante, et requiert une réécriture complète du modèle. Cette absence de portage incrémental impose une barrière d'entrée à l'utilisation du GPU et rend également plus difficile la comparaison du modèle obtenu avec l'original, la gestion du programme même n'étant plus directement du ressort du concepteur.

Si le formalisme et les plates-formes supportées par FLAME sont extensibles, comme l'illustre l'existence de FLAME-GPU, cette extension se limite également à l'utilisation de CUDA, ce qui limite son utilisation aux seuls matériels NVIDIA, et ne permet pas de tirer parti des cartes graphiques d'autres fabriquants présents sur des machines personnelles.

3.4 Synthèse

Dans ce chapitre, nous avons montré les différentes approches de parallélisation possibles pour des systèmes multi-agents, au niveau de l'ordonnanceur, des comportements ou de l'environnement. Nous avons ensuite présenté la manière dont ces approches sont gérées et rendues plus aisément accessibles pour le concepteur par quelques plates-formes multi-agents parallélisées. La plupart de ces plates-formes permettent une exécution aussi bien en mémoire distribuée ou partagée, en distribuant le passage des traitements à l'aide d'une collaboration de plusieurs ordonnanceurs. Certaines de ces plates-formes gèrent aussi automatiquement la répartition de l'environne-

ment entre chaque tâche, pour résoudre tout problème de taille mémoire. De nombreuses solutions, comme JADE, laissent toutefois cette responsabilité au concepteur, plus à même de décider si ces structures doivent être recopiées ou accédées de manière distante dans sa simulation.

Si la parallélisation sur CPU est aujourd'hui supportée par de nombreuses plates-formes, la parallélisation sur carte graphique est moins répandue. Cette architecture montre pourtant son intérêt, dans le cadre de milliers d'agents à exécuter, par sa possibilité de disposer de plusieurs centaines de coeurs sur une machine personnelle. Des modèles comme SugarScape, ou des implémentations sous forme d'automates cellulaires ou de recherche de chemin mettent ainsi en avant l'intérêt en termes de performance du GPU pour l'exécution de traitements homogènes, de type SIMD, souvent rencontrés dans les systèmes multi-agents réactifs.

Le support du GPU n'est cependant pas totalement absent des plates-formes agents : des solutions permettant d'utiliser CUDA existent déjà dans des modèles comme JADE. Des couches d'abstraction, soit des perceptions dans le cas de TurtleKit, soit de tout le modèle dans le cas de FLAME-GPU, existent également.

II

Contribution

Dans la première partie de notre mémoire nous avons présenté notre contexte, les systèmes multi-agents et les GPU, et mis en avant l'intérêt de cette nouvelle architecture en termes d'accessibilité et de performance, par rapport aux solutions de parallélisation classiques basées sur la distribution en mémoire partagée ou en mémoire distribuée.

Dans cette seconde partie, nous introduisons et décrivons maintenant nos contributions à cette problématique. Pour cela, nous commençons par étudier la parallélisation sur GPU d'un modèle concret, le modèle proie-prédateur. Cette parallélisation est pour nous l'occasion d'une réflexion sur les différentes représentations en termes de données et de comportements de ce système, et leur impact sur une adaptation GPU.

Les contraintes rencontrées dans l'adaptation de ce modèle ouvrent alors la voie à une définition des principales considérations nécessaires pour le portage d'un système multi-agents sur GPU. Cette présentation est en particulier l'occasion de décrire les différents types de découpages des données ou d'exécution devant être supportés par une bibliothèque pour permettre son utilisation dans les systèmes multi-agents.

Ces contraintes nous servent de préambule pour présenter notre solution, MCMAS, et la manière dont cette bibliothèque répond à ces problématiques d'exécution et de données. Pour cela, nous commençons par présenter les principaux objectifs de MCMAS et leur traduction en termes d'architecture de la bibliothèque. Nous présentons ensuite l'utilisation de MCMAS en tant que bibliothèque de fonctions génériques, puis l'ajout de nouvelles fonctionnalités à cette solution.

L'utilisation de MCMAS est ensuite illustrée de manière expérimentale sur trois modèles :

— Proie-prédateur qui nous a servi de fil rouge dans notre propos.
— MIOR, un modèle d'évolution microscopique de sol s'intégrant dans le cas de la simulation multi-échelles Sworm.
— Collemboles, enfin, un modèle permettant de décrire la diffusion de populations entre parcelles de terrain importées depuis un système d'information géographique.

Une fois ces applications présentées, nous proposons quelques recommandations d'implémentation, de manière à prendre en compte les éventuels différences en traitements des données et en performance associées à chaque matériel d'exécution.

4

PROBLÉMATIQUE

Comme nous l'avons vu dans le chapitre précédent de nombreux systèmes multi-agents ont déjà été adaptés sur GPU. Dans la plupart des cas, ces adaptations sont basées sur une réécriture complète du programme, directement à l'aide d'un modèle de programmation comme OpenCL ou au moyen du formalisme spécialisé proposé par l'environnement FLAME-GPU.

La première approche, l'implémentation directe du modèle avec la programmation GPGPU, permet un contrôle fin sur le programme obtenu mais requiert une expertise et un investissement en temps conséquents pour pouvoir tirer parti efficacement des possibilités offertes par l'architecture matérielle. Ces contraintes s'expliquent par la nécessité d'implémenter la totalité du système agent, sans aucune infrastructure existante pour ce type de simulation.

La seconde approche, l'utilisation de FLAME-GPU, apporte une abstraction totale du mode d'exécution en permettant au concepteur de n'avoir qu'à décrire les opérations de son modèle. Elle repose sur l'utilisation de l'approche FLAME pour générer automatiquement le programme GPU liant ces différentes actions. Cette abstraction présente cependant un coût, en contraignant la structure des modèles multi-agents décrits dans un formalisme existant strict. Ce formalisme peut être aisément utilisé pour la conception de nouveaux modèles mais rend difficile la comparaison des simulations obtenues avec des simulations plus traditionnelles, du fait de la nature radicale des changements apportés à l'algorithme. Il permet toutefois la disponibilité d'un support robuste adapté aux agents, et ne laisse donc pas le concepteur livré à lui-même comme un modèle de programmation plus générique.

Ces deux approches laissent place à une solution intermédiaire qui permettrait au chercheur de disposer d'un support incrémental pour les traitements agents sur GPU, de manière à pouvoir y exécuter tout ou partie de la simulation. Une telle solution doit également être capable de s'intégrer et de complèter une plate-forme multi-agents existante. Cette facilité d'intégration implique la proposition d'une interface orthogonale à la modélisation du système, à même d'être utilisée aussi bien dans des cadres basés sur la décomposition du système en messages et en comportement que dans des simulations agents moins segmentées. Elle permet également de tirer parti des nombreuses facilités d'implémentation et d'exécution déjà offertes par les plates-formes multi-agents existantes.

Dans les sections suivantes nous présentons les propriétés nous paraissant particulièrement importantes pour notre proposition, MCMAS [1].

1. Many-Core Multi-Agent Systems

4.1 Portabilité

Un premier élément nécessaire à l'intégration d'une nouvelle bibliothèque dans des systèmes multi-agents existants est sa portabilité sur plusieurs systèmes d'exploitation. De nombreux chercheurs utilisent en effet différents systèmes d'exploitation pour réaliser leurs simulations, comme l'illustre la portabilité de plates-formes multi-agents telles que Repast, Madkit ou NetLogo.

Une nouvelle bibliothèque doit également être portable en termes d'utilisation, en se basant sur les langages de programmation largement utilisés dans le domaine. Si C, C++ et Java sont présents parmi les plates-formes multi-agents décrites dans notre contexte, c'est surtout ce dernier langage qui est utilisé par de nombreuses plates-formes comme NetLogo, JADE, MadKit ou D-MASON. Cette large utilisation s'explique par plusieurs avantages associés à l'environnement Java :

— La portabilité des programmes sans recompilation. Il est ainsi possible de distribuer une seule version de la plate-forme pour tous les systèmes d'exploitations supportés.
— La disponibilité d'une bibliothèque graphique intégrée, Swing. Cette bibliothèque facilite la réalisation de plates-formes interactives sans dépendances ou binaires externes.
— La possibilité de programmer en objet, de manière à représenter de manière intuitive les différentes entités du modèle agent.
— La disponibilité de nombreux mécanismes d'exécution (threads) ou de communication (RMI, JMS) et de nombreuses structures de données dynamiques directement dans la bibliothèque standard.
— Une gestion de la mémoire automatique, qui facilite à la fois le développement de la plate-forme de simulation et des modèles multi-agents.
— La disponibilité de bibliothèques de gestion de données géographiques comme GIS, permettant un import aisé d'informations externes dans la simulation.

Il est toutefois nécessaire de coupler Java à des couches d'adaptation native comme JOCL pour permettre l'accès au modèle d'exécution GPU. Le choix du modèle de programmation utilisé est également un élément de cette portabilité d'utilisation : si CUDA et OpenCL sont tous deux disponibles sur de nombreux systèmes d'exploitations, OpenCL offre un plus grand choix de périphériques d'exécution CPU ou many-cores que CUDA, qui reste limité à l'utilisation de matériels Nvidia. Au vu de cette large disponibilité, le choix d'OpenCL paraît naturel pour notre solution, en gardant toutefois à l'esprit que chaque architecture d'exécution ne fournira pas forcément les mêmes performances pour les mêmes programmes.

4.2 Réutilisation d'algorithmes et de structures

Comme évoqué au début de notre problématique, l'utilisation directe de modèles de programmation tels que OpenCL et CUDA pour l'adaptation d'un système multi-agents impose le redéveloppement par le concepteur de l'ensemble de ses traitements et de ses structures de données. Ce développement requiert des connaissances poussées en programmation C ou C++ et un travail important en termes d'adaptation de l'exécution et des structures de données.

Mené correctement, ce type d'adaptation permet d'obtenir une solution optimisée, au plus proche des contraintes des données et des algorithmes utilisés par le modèle source. Cette solution encourage toutefois, en imposant leur développement, l'utilisation de structures spécifiques à chaque modèle adapté et difficilement réutilisables. Elle représente également un coût en temps important devant être réinvesti à l'adaptation de chaque nouveau modèle. Dans ces circonstances, l'intérêt d'une bibliothèque est de permettre et favoriser la réutilisation d'algorithmes et de struc-

tures de données.

Cette réutilisation peut être grandement facilitée en fournissant des implémentations optimisées des traitements multi-agents les plus courants, prêtes à être employées par le modèle agent. L'objectif de cette démarche est de proposer, à l'image de bibliothèques comme CUBLAS, des fonctions de haut-niveau déjà adaptées aux principales problématiques rencontrées dans les systèmes multi-agents. Ces problématiques peuvent aussi bien concerner la mise à jour de l'environnement que le calcul de distances entre agents, la génération de nombres aléatoires ou encore le regroupement des données à chaque pas d'exécution, de manière à pouvoir observer la dynamique du modèle.

4.3 Intégration avec l'existant

De nombreuses plates-formes multi-agents ont pour objectif d'être une solution "tout en un" aux problèmes de modélisation du chercheur. Elles comprennent dans cette optique de nombreuses fonctionnalités annexes de conception et d'édition de modèle, d'affichage de l'environnement simulé ou de courbes représentant l'évolution du modèle, ou de production de fichiers résultats.

Une solution pour les GPU visant à remplacer totalement ces plates-formes impliquerait d'intégrer toutes ces fonctionnalités avant de constituer une alternative crédible, et représenterait une nouvelle solution concurrente, ce qui n'est pas l'objectif de notre contribution.

La bibliothèque doit donc être capable de s'intégrer dans ces plates-formes à plusieurs niveaux d'encapsulation du modèle. Elle ne doit notamment pas imposer un paradigme d'exécution particulier, de manière à ne pas entrer en conflit avec le fonctionnement de la simulation. Cette transparence est également importante pour permettre son utilisation dans des plates-formes n'employant pas directement le langage Java, au moyen de modules additionnels (*plugins*) ou d'extension de la syntaxe agent proposée. Pour décharger totalement le modèle de la gestion de cette bibliothèque, il est possible d'envisager son intégration sous la forme d'agents services fournissant l'accès à des services implémentés sur GPU au reste du modèle.

4.4 Extensibilité

La simulation orientée agent est un domaine en perpétuelle évolution, tant au niveau conceptuel, dans le domaine par exemple de l'intelligence artificielle, qu'en termes d'implémentation, avec l'arrivée de nouvelles architectures matérielles et de nouveaux modèles de programmation. Ce dynamisme et cette flexibilité expliquent la popularité de cette approche de modélisation pour la résolution de nombreux problèmes, mais soulignent également l'importance de proposer des solutions modulaires et si possible génériques. Les avantages de cette modularité sont illustrés par le succès des nombreuses plates-formes agents traditionnelles, fournissant un vaste ensemble de fonctionnalités indépendantes pouvant être ou non exploitées par un modèle particulier : c'est ainsi le cas de plates-formes comme D-MASON ou NetLogo [Skl11], qui proposent un découpage des structures de données implicite, de JADE, avec la possibilité d'ajouter de nouveaux composants de manière dynamique au système, ou encore de MadKit qui permet la surcharge de nombreux comportements de la plate-forme. La généricité est également présente dans ces plates-formes sous forme de cadre de modélisation et d'opérations facilitant la conception de nouveaux modèles agents. Ce cadre peut être très souple, dans le cas par exemple de Madkit, ou beaucoup plus strict, dans le cas des plates-formes FLAME ou FLAME-GPU.

Notre solution doit donc, au-delà de la flexibilité d'utilisation, permettre l'ajout aisé de fonctionnalités et d'extensions sans remettre en cause son architecture fondamentale, de manière à pouvoir traiter de nouveaux problèmes agents sur GPU.

4.5 Synthèse

Dans notre présentation du contexte, nous avons eu l'occasion d'évoquer le besoin en ressources toujours plus important rencontré par les simulations en général et en particulier par les simulations multi-agents. S'il est en effet courant de commencer par valider un système à petites échelles, l'observation de certains comportements émergents requiert parfois un nombre minimal d'individus [SN09]. L'amélioration des résultats obtenus par la simulation agent implique, de manière plus générale, une augmentation en taille des modèles, que ce soit en termes de dimension de l'environnement ou de nombres d'agents, et des comportements toujours plus complexes.

La parallélisation de l'exécution en mémoire partagée ou en mémoire distribuée est une réponse à ce besoin toujours croissant en ressources mémoires et de calcul. Les architectures many-core et GPU en particulier offrent l'accès à plusieurs centaines de coeurs d'exécution à des tarifs comparables à ceux d'une machine de bureau, via des modèles de programmation comme CUDA et OpenCL.

L'utilisation de ces architectures pour la simulation multi-agents n'est pas une idée nouvelle et a déjà été explorée pour de nombreux modèles, sous forme de développement de nouvelles implémentations complètes. Ces adaptations requièrent cependant une connaissance approfondie des modèles de programmation et d'exécution GPU pour permettre l'obtention de programmes efficaces. De nombreux concepteurs agents, théoriciens, n'ont pas le temps d'acquérir ces compétences pour la réalisation de nouveaux modèles.

FLAME-GPU permet, a contrario, la génération de modèles agents basés sur CUDA sans connaissance de ce modèle de programmation à partir de modèles FLAME. Cette solution impose cependant l'utilisation du formalisme strict de cette plate-forme, et n'est donc pas accessible à des modèles basés sur d'autres plates-formes agents parallèles ou séquentielles telles que Madkit, NetLogo ou GAMA sans un redéveloppement complet.

Notre thèse est de montrer qu'une bibliothèque de ce type peut être développée de manière indépendante et générique par rapport à ces plates-formes d'exécution multi-agents, pour permettre son utilisation dans un large nombre d'environnements et de modèles existants.

Dans la suite de ce mémoire, nous présentons la démarche qui nous a mené à la conception de MCMAS, une bibliothèque permettant de réaliser tout ou partie d'une simulation multi-agents sur GPU. L'objectif de cette bibliothèque est de permettre une utilisation de cette nouvelle architecture d'exécution de manière indépendante ou en complément de plates-formes de conception et d'exécution multi-agents existantes. Elle reprend en compte, pour cela, les problématiques de portabilité, de réutilisation, d'intégration et d'extensibilité évoquées dans ce chapitre dans un tout cohérent, proposant deux niveaux d'utilisation, l'un adapté à son utilisation sans connaissances GPU, l'autre à une extension aisée des fonctionnalités offertes par la bibliothèque.

5

Adaptation d'un modèle multi-agents sur GPU : Proie-Prédateur

Le modèle proie-prédateur est un modèle multi-agents inclus à titre de démonstration dans de très nombreuses plates-formes agents. Il présente l'intérêt d'offrir un grand choix de niveaux de représentation, mais également de nombreuses implémentations possibles. Dans ce chapitre, nous l'utilisons comme illustration des types de problématiques agents devant être traitées pour une adaptation sur GPU, tant en termes de mémoire que d'exécution.

5.1 Présentation du modèle

Le modèle proie-prédateur a été pour la première fois décrit de manière indépendante par Alfred James Lotka en 1925 et par Vito Volterra en 1926 sous la forme d'un couple d'équations différentielles. Ces équations, basées sur des observations effectuées au 19^{me} siècle sur des populations d'animaux sauvages, permettent d'estimer l'évolution en fonction du temps de deux populations, carnivore et herbivore, en fonction des paramètres définis pour la simulation du modèle :

$$\begin{cases} \dfrac{\mathrm{d}x(t)}{\mathrm{d}t} &= x(t)(\alpha - \beta y(t)) \\ \dfrac{\mathrm{d}y(t)}{\mathrm{d}t} &= -y(t)(\delta - \gamma x(t)) \end{cases}$$

où

— t est le temps ;
— $x(t)$ est le nombre de proies en fonction du temps ;
— $y(t)$ est le nombre des prédateurs en fonction du temps ;
— les dérivées $\mathrm{d}x(t)/\mathrm{d}t$ et $\mathrm{d}y(t)/\mathrm{d}t$ représentent la variation des populations au cours du temps.

Les paramètres suivants décrivent les interactions entre les deux espèces :

— α le taux de reproduction des proies ;
— β le taux de mortalité des proies dû aux prédateurs ;
— γ le taux de mortalité des prédateurs ;
— δ le taux de reproduction des prédateurs.

Pour des paramètres d'entrée évitant la disparition totale de l'une ou l'autre des populations, l'évolution du nombre d'individus de chaque type tend alors rapidement à osciller de manière régulière entre périodes de faste et de famine. La Figure 5.1 illustre ces oscillations dans le temps.

$$\begin{cases} \alpha = 2 \\ \beta = 0.5 \\ \gamma = 0.2 \\ \delta = 6 \end{cases}$$

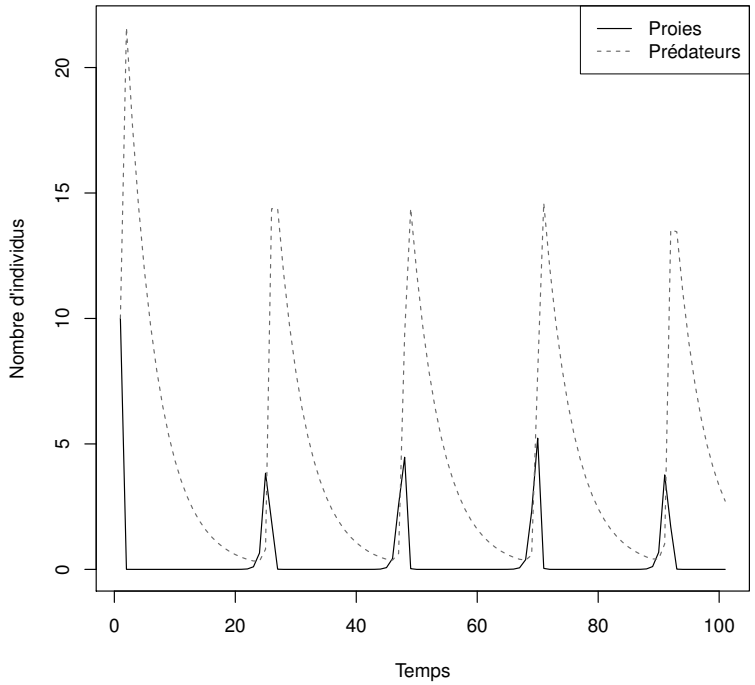

FIGURE 5.1 – Exemple de courbes obtenues par l'application des équations de Lotka-Volterra

Cette première approche, mathématique, ne considère que les nombres d'individus du modèle et des paramètres synthétisant leur évolution générale. Une autre approche de modélisation, orientée agent, reproduit ces tendances en décrivant le comportement des individus mis en présence au moyen d'actions élémentaires de bas niveau décrites sous forme d'algorithme.

Une approche agent possible est ainsi d'isoler trois niveaux d'évolution dans le modèle proie-prédateur :

— L'environnement, modélisant les ressources consommées par les proies (végétation). Il s'agit d'un agent passif, modifié par les autres agents du système. Le seul traitement propre lui étant associé est l'augmentation des ressources d'un facteur fixé entre chaque pas de simulation, de manière à refléter la croissance et le remplacement des végétaux dans le temps.
— Les proies, capables de se déplacer et de se reproduire. Lors de ce second processus, la proie modifie l'environnement, pour répercuter la nourriture consommée.
— Les prédateurs, capables de se déplacer, de consommer des proies et de se reproduire. Ces agents n'interviennent que sur les proies, et ne modifient pas directement l'environnement.

L'ajout d'un environnement basé sur les ressources en végétation s'inscrit dans l'optique de décrire le système selon une approche montante, comme nous l'avons vu dans notre présentation des approches de modélisation, plutôt que de se baser sur une observation de haut niveau pour extrapoler le comportement de chaque individu, avec une approche descendante.

Ce modèle agent se caractérise par une relation stricte entre les populations, à sens unique. Seuls les prédateurs sont à même d'influer sur les proies, qui sont seules à pouvoir influer sur l'environnement. Par rapport à la modélisation mathématique du modèle proie-prédateur, cette re-

présentation agent introduit aussi une prise en compte de l'accès aux ressources de chaque agent, en introduisant une répartition géographique des différentes entités du système. Cet aspect supplémentaire permet non seulement d'obtenir des valeurs de population, comme le modèle mathématique, mais ouvre également l'accès à des résultats plus précis, permettant de mettre en avant l'influence de la répartition des ressources sur la position des agents dans l'environnement.

5.2 Stratégies de déplacement

Un facteur important de l'évolution du système proie-prédateur est l'algorithme de déplacement utilisé. Cet algorithme détermine en effet non seulement la manière dont chaque agent considérera les données de son voisinage, mais également le nombre de branchements et la régularité du nombre d'opérations à effectuer, pour un bon remplissage des ressources GPU.

5.2.1 Déplacement aléatoire

Une première implémentation du déplacement des proies et prédateurs est la sélection d'une destination aléatoire à chaque itération. Cette approche évite le parcours du voisinage de chaque agent pour la recherche d'une cible, et donc de nombreux accès mémoire, mais est peu intéressante à implémenter en termes de modèle et d'adaptation GPU :

— En termes de modélisation, elle est très peu cohérente avec le comportement de nombreuses espèces animales capables de détecter et poursuivre des proies. Elle minimise également artificiellement la population pouvant être supportée par le modèle, en ne laissant qu'une chance minime, en fonction de la densité des ressources dans l'environnement, à chaque individu de disposer des ressources nécessaires à sa survie.
— En termes d'adaptation sur GPU, ce comportement réduit la démarche de chaque individu au simple tirage aléatoire d'un jeu de coordonnées. Ce tirage rend inutile tout parcours de données sur GPU, et ne requiert plus qu'une gestion des conflits et la génération de nombres aléatoires. Ces deux processus sont difficiles à réaliser sans rendre partiellement séquentiels les deux traitements, ce qui limite l'intérêt des centaines de coeurs offerts par l'architecture.

Cette stratégie de déplacement aléatoire est surtout utilisée pour valider le fonctionnement d'une nouvelle simulation.

5.2.2 Proie la plus proche

Cette seconde stratégie implique un parcours pour chaque individu de l'ensemble des cases du modèle qui sont à sa portée, suivi par un déplacement sur la position de la proie la plus proche. Elle requiert un grand nombre d'accès mémoires pour évaluer toutes les cases destinations possibles, ainsi qu'une notion de priorité entre ces cases : une case proche contenant une proie doit être préférée à une autre case solution plus éloignée.

Une manière d'éviter le calcul de la distance de chaque proie est d'effectuer le parcours des cases voisines sous forme d'une spirale comme illustré par la Figure 5.2. Ce parcours en spirale permet non seulement de garantir que les proies seront automatiquement considérées par ordre d'éloignement, mais offre également l'avantage de pouvoir arrêter l'évaluation à la première proie détectée (court-circuit).

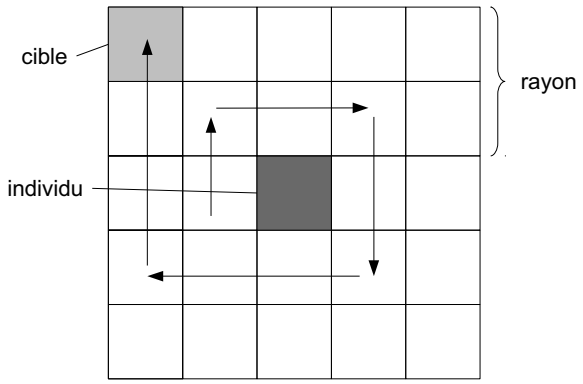

FIGURE 5.2 – Principe de recherche de la proie la plus proche

Cet algorithme est aisément parallélisable sur GPU en attribuant une tâche à chaque individu, puis en effectuant le parcours des cellules voisines sous forme de boucle à l'intérieur de chacune de ces tâches. Il implique néanmoins, du fait de la condition d'arrêt, une forte variabilité du nombre de cellules à parcourir et donc du temps d'exécution de chaque tâche.

Il n'est pas facile de paralléliser davantage ce parcours en spirale en confiant l'examen de chaque case candidate à un thread distinct. Cet algorithme de déplacement requiert en effet la prise en compte des cases dans un ordre particulier, qui n'est pas garanti par l'ordonnanceur GPU. Une solution à ce problème est d'introduire un calcul de la distance pour chaque destination potentielle, suivie d'une réduction pour ne conserver que la cible détectée la plus proche de l'individu.

Cette stratégie minimise la distance de déplacement de chaque prédateur, mais n'est pas forcément la meilleure en termes de survie à moyenne ou longue échéance, car la proie la plus proche n'est pas forcément la plus dotée en énergie.

5.2.3 Proie la plus énergétique

Une autre stratégie de déplacement possible est de sélectionner cette fois systématiquement la proie à portée disposant de la plus grande quantité d'énergie. Cette stratégie, qui permet de toujours sélectionner la proie la plus intéressante immédiatement accessible, garantit la sélection d'une proie équivalente ou supérieure en énergie par rapport à la sélection de la cible la plus proche. Elle impose cependant le parcours de l'ensemble des cases à portée avant de pouvoir tirer une conclusion, tel qu'illustré par la Figure 5.3. Le nombre d'accès à la mémoire est d'autant plus important, en comparaison avec la recherche de la proie la plus proche, que le modèle est dense.

Ce parcours obligatoire de l'ensemble des cellules à portée rend cette stratégie beaucoup plus coûteuse en nombres d'accès mémoire. L'absence de priorité entre les différentes cases voisines permet cependant des accès plus réguliers sur GPU, ligne par ligne ou colonne par colonne, pour mieux tirer parti du chargement des données par paquet ou du regroupement des accès mémoire réalisés à l'exécution.

5.2.4 Compromis et stratégies avancées

En fonction de l'intelligence du prédateur modélisé, il peut être pertinent de combiner une ou plusieurs des stratégies suggérées ci-dessus. Ainsi, si aucune proie n'est à portée d'un individu, un déplacement aléatoire par défaut offre une chance de se rapprocher de densités de proie plus

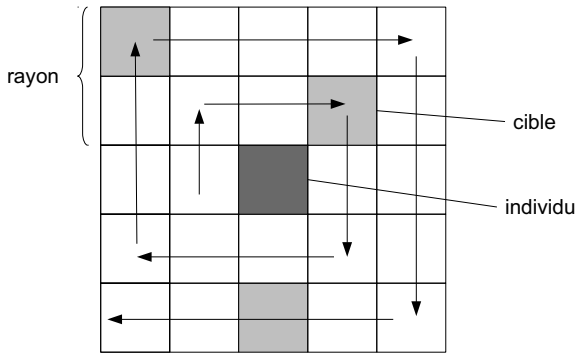

FIGURE 5.3 – Principe de recherche de la proie la plus énergétique

importantes. Ce déplacement comporte cependant également le risque d'éloigner l'individu des ressources recherchées, et son intérêt est donc très dépendant des paramètres du modèle et de la répartition géographique des agents. Nous avons fait le choix dans notre modèle de ne pas déplacer l'individu en l'absence de cible, pour éviter ce problème.

Au-delà de ces compromis instantanés entre plusieurs comportements, la recherche de cible ouvre également la voie à de véritables intelligences, avec mémorisation des proies. Il est ainsi possible d'intégrer la possibilité pour le prédateur de suivre sa proie si celle-ci quitte son champ de vision immédiat. Un autre mécanisme pouvant être implémenté est un comportement de persistance : si plusieurs cibles offrent des caractéristiques très similaires, mais deviennent tour à tour les plus intéressantes du fait de micro-variations du modèle entre chaque itération, il est alors pertinent de pousser l'individu à se concentrer sur une seule cible, plutôt que de dépenser son énergie à courir dans diverses directions.

5.3 Adaptation OpenCL

Après avoir présenté les principales approches de modélisation possibles pour le modèle proie-prédateur et en particulier les différents algorithmes de déplacement et leur influence sur la simulation, tant en termes de modèle que de parallélisation, nous étudions dans cette section les choix d'implémentation effectués pour notre implémentation OpenCL.

5.3.1 Algorithme retenu

Pour évaluer l'efficacité de l'adaptation de ce type d'algorithme sur plate-forme GPGPU, nous avons choisi d'implémenter l'algorithme proie-prédateur décrit dans l'algorithme 1.

Cette implémentation est caractérisée par la simulation de trois populations, dont deux représentent des prédateurs :

— **L'herbe, ou végétation** représente les ressources végétales pouvant être consommées par la population herbivore du modèle.
— **Les proies** représentent la première population prédatrice du modèle, capables de consommer des végétaux, de se déplacer et de se reproduire. Cette population est caractérisée par son abondance, liée à des besoins limités et un taux de fertilité important.
— **Les prédateurs** représentent la seconde population prédatrice du modèle, capable de consommer des proies de la première population. Cette seconde catégorie d'individus est

Algorithme 1 : Algorithme proie-prédateur retenu

Data : $grass$ Grille de flottants représentant la couche herbe
Data : $preys$ Grille de flottants représentant la couche proies
Data : $preds$ Grille de flottants représentant la couche prédateurs
Data : n Largeur de chacune des trois grilles d'entrée
Data : m Hauteur de chacune des trois grilles d'entrée

```
1  for i = 0 to n do
2      for j = 0 to m do
3          grass[i][j] ← grass[i][j] + growth;
4      end
5  end
6  for (i, j) ∈ (proiesX, proiesY) do
7      x, y ← findTarget(i, j);
8      if preys[i][j] vide then
           /* Déplacement de la proie en x, y                          */
9          preys[x][y] ← preys[i][j];
10         preys[i][j] ← 0;
11     end
       /* Consommation des végétaux                                    */
12     preys[x][y] ← clamp(preys[x][y] + grass[x][y], prey_min, prey_max);
       /* Diminution de l'énergie de la proie                          */
13     preys[i][j] ← preys[i][j] − delta;
14     if preys[i][j] < 0 then
           /* Décès de la proie                                        */
15         preys[i][j] ← 0;
16     end
17 end
18 for (i, j) ∈ (predateursX, predateursY) do
19     x, y ← findTarget(i, j);
20     if preds[i][j] vide then
           /* Déplacement du prédateur en x, y                         */
21         preds[x][y] ← preds[i][j];
22         preds[i][j] ← 0;
23     end
       /* Consommation de la proie                                     */
24     preys[x][y] ← 0;
       /* Diminution de l'énergie du prédateur                         */
25     preds[i][j] ← preys[i][j] − delta;
26     if preds[i][j] < 0 then
           /* Décès du prédateur                                       */
27         preds[i][j] ← 0;
28     end
29 end
```

associée à des besoins plus importants, et un taux de fertilité relativement plus faible.

Cet algorithme pose plusieurs problèmes de parallélisation :

— L'évolution de chaque population dépend des mises à jour des populations précédentes.
— Plusieurs individus d'une même population peuvent tenter de se déplacer au même endroit.
— L'algorithme repose sur le parcours des positions des proies et des prédateurs. Cette liste de positions doit être mise à jour au fur et à mesure de la disparition et de l'apparition de nouveaux individus.

5.3.2 Représentation mémoire

Dans notre implémentation OpenCL, nous avons fait le choix de représenter les données globales du modèle sous la forme d'une unique structure mémoire accessible en lecture seule à toutes les tâches GPU. Ces informations comprennent l'ensemble des données d'entrée de la simulation : taux de métabolisme et de croissance des ressources végétales, énergie minimale et maximale pour chaque individu...

Chaque population est stockée sous forme de grille à deux dimensions (Figure 5.4). Toutes ces grilles font la même taille, et peuvent être considérées comme une vision particulière de l'espace de simulation. La présence d'un agent est indiquée par une énergie strictement positive dans une cellule, et l'absence d'individu correspond à une valeur d'énergie négative ou nulle.

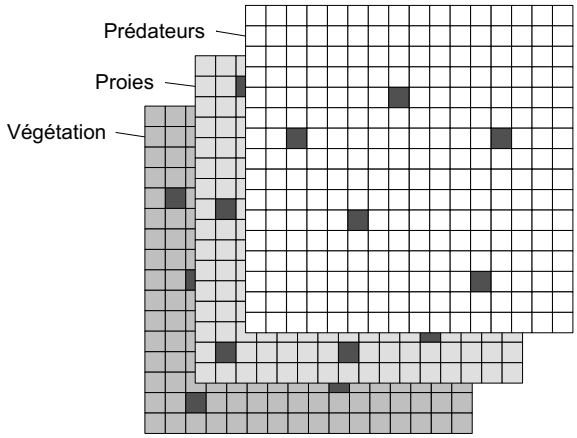

FIGURE 5.4 – Stockage retenu pour les populations et les niveaux d'énergie du modèle proie-prédateur

Pour éviter un parcours complet de chaque grille pour localiser les individus, quatre tableaux statiques indiquant la position des agents proies et prédateurs du modèle sont également spécifiés en paramètres des traitements de mise à jour et de déplacement :

— Un tableau comprenant les positions en x des proies.
— Un tableau comprenant les positions en y des proies.
— Un tableau comprenant les positions en x des prédateurs.
— Un tableau comprenant les positions en y des prédateurs.

Ces tableaux de positions sont mis à jour par le CPU entre chaque itération de la simulation pour prendre en compte l'apparition de nouveaux individus ou les décès d'agents existants. Cette mise à jour sur le processeur hôte permet d'éviter toute problématique de réallocation mémoire non supportée sur GPU.

5.3.3 Extraction de sections parallèles

Il est possible de paralléliser le modèle proie-prédateur de deux manières sur GPU :

— Soit en réalisant l'ensemble de la simulation sur GPU.
— Soit en n'intervenant que sur les traitements coûteux et parallélisables.

La première approche implique de gérer la totalité des problèmes de parallélisation posés par le modèle en OpenCL, et en particulier la gestion des structures de données dynamiques comprenant les positions des proies. Elle implique également le traitement de populations agents très diverses en nombre, qu'il s'agisse de l'herbe, des proies, ou des prédateurs. Cette diversité impose soit une réalisation de la simulation en plusieurs lancements OpenCL, soit une occupation très disparate des threads d'exécution lancés sur la carte graphique aux différents stades de l'exécution.

Nous avons donc fait le choix de privilégier la seconde approche, où la simulation est au départ intégralement réalisée sur CPU, pour en adapter ensuite un ou plusieurs traitements sur GPU. Dans ce cas, un point central de l'adaptation est de sélectionner les portions du programme à même de bénéficier d'une parallélisation sans modification fondamentale de l'algorithme. Cette sélection est favorisée par l'existence dans le modèle d'opération de transformation ou de réductions analogues à celles proposées par des bibliothèques telles que BLAS ou MPI. C'est en particulier le cas de l'opération de mise à jour des ressources végétales, qui correspond à l'application d'une même fonction mathématique à chaque cellule de l'environnement, et le cas de la réduction de la quantité d'énergie présente dans le modèle.

Dans l'algorithme décrit, trois traitements semblent de bons candidats à une telle extraction :

— **La mise à jour des ressources végétales**. L'application d'une transformation mathématique simple à chaque cellule, sans dépendance sur ses voisines, permet l'utilisation d'une tâche OpenCL pour la mise à jour de chaque cellule. L'absence de synchronisation permet ainsi de tirer aisément parti du grand nombre de coeurs du GPU. Le coût de cette section est également directement proportionnel à la taille de l'environnement de simulation, plutôt qu'au nombre d'individus présents dans le modèle, ce qui garantit un temps de calcul important et aisément prévisible en connaissant les dimensions du modèle d'entrée. Cette opération revient à l'application d'une même fonction mathématique à chaque cellule de l'environnement.
— **La recherche des nouvelles positions pour chaque individu.** Si la gestion des individus est malaisée à implémenter de manière intégrale sur GPU, du fait de l'apparition ou de la disparitions des agents, ou des nombreuses conditions rencontrées dans l'algorithme, la recherche de cibles de déplacement implique le parcours d'un nombre de cellules d'autant plus important que le nombre et la portée de chaque agent augmentent. Réaliser ce calcul sur GPU permet le parcours de cellules en parallèle pour chaque position d'individu du modèle, et laisse ensuite la possibilité au CPU d'intervenir au niveau de la prise en compte de cette valeur brute, en l'ignorant ou l'adaptant si nécessaire.
— **Totalisation des quantités d'énergie présentes dans le modèle.** Le choix de représentation de chaque population dans une grille spécialisée, enfin, permet aisément d'effectuer la somme des valeurs de l'ensemble des cellules de la grille pour obtenir l'énergie totale à une itération donnée représentée sous forme de végétaux, de proies ou de prédateurs. Ce traitement revient à une forme de réduction cas de la quantité d'énergie présente dans chaque grille du modèle.

5.4 Synthèse

Dans ce chapitre, nous avons évoqué une parallélisation possible du modèle proie-prédateur reposant sur la délégation de certaines fonctions au GPU. La parallélisation d'une partie du modèle est motivée par deux observations :

— Certains traitements correspondent directement à des opérations déjà parallélisées par des bibliothèques telles que BLAS ou MPI. C'est en particulier le cas de l'opération de mise à jour des ressources végétales, qui revient à l'application de la même fonction mathématique à chaque cellule de l'environnement, et le cas de la réduction de la quantité d'énergie présente dans le modèle.
— D'autres traitements, comme le déplacement, impliquent la réalisation d'un grand nombre d'itérations de boucle pour chaque agent dans le modèle. Dans ce cas, cette opération de recherche peut être parallélisée, de manière à permettre à tous les agents de consulter l'environnement simultanément.

Au vu de notre contexte et de cette première démarche de parallélisation d'un système multi-agents, il est maintenant possible de présenter les différentes approches pouvant être utilisées pour l'adaptation de ce type de système sur GPU.

6

Méthodes d'adaptation SMA sur GPU

Au vu de notre contexte et de notre démarche de parallélisation du modèle proie-prédateur en OpenCL, il est possible de dégager trois méthodes principales d'utilisation du GPU pour des simulations multi-agents : l'adaptation de la totalité du modèle, la parallélisation de certaines opérations, ou le recours à des opérations parallélisées existantes.

La première approche, l'adaptation de la totalité du modèle, implique l'implémentation de l'ensemble de l'algorithme sur GPU par le concepteur. Dans ce cas, l'essentiel de la simulation utilise le langage et les structures de données OpenCL pour son exécution, et la gestion de l'allocation, de la copie et de la manipulation des structures de données est directement effectuée par le programme. Cette solution permet également un contrôle total de l'exécution par le programme.

La seconde approche, la parallélisation de certaines opérations seulement sur GPU, vise à maximiser le gain en performance obtenue tout en conservant une démarche de parallélisation incrémentale. Cette approche partielle permet de conserver les aspects complexes de la simulation sur le CPU, comme dans le cas de la gestion des listes de positions de chaque individu dans le modèle proie-prédateur. Elle requiert elle aussi une expertise en programmation pour implémenter ces opérations parallélisées.

La troisième approche, enfin, est de réutiliser des opérations de haut niveau déjà parallélisées sur GPU. Cette approche permet l'utilisation de la puissance des cartes graphiques dans une simulation multi-agents sans connaissance particulière de l'architecture. Elle implique toutefois de se conformer à l'interface de ces fonctions existantes en visant une certaine généricité des traitements. Dans le modèle proie-prédateur, il est ainsi intéressant de se ramener à des opérations matricielles, plutôt que d'effectuer le traitement de chaque case de manière indépendante.

Dans les sections suivantes, nous présentons dans un premier temps comment l'espace de simulation utilisé par le système multi-agents est susceptible d'orienter le choix du concepteur vers l'une ou l'autre de ces approches. Nous évoquons ensuite, pour chacune de ces trois solutions, les contraintes associées pour permettre une utilisation efficace de la parallélisation sur GPU.

6.1 Gestion de la dimension spatiale

L'algorithme d'évolution d'un système multi-agents ne fixe généralement pas de bornes particulières au nombre d'individus ou à la taille de l'environnement utilisé dans le modèle. Ces deux paramètres sont déterminés par le scénario de la simulation.

Une connaissance même approximative de ces paramètres au moment de la conception d'un programme permet cependant au développeur de privilégier a priori certaines structures de données. Le choix d'utiliser un dictionnaire se justifie ainsi pour indexer un grand nombre d'éléments,

mais s'avère au contraire pénalisant pour une très petite quantité de données, du fait des traitements supplémentaires requis par la gestion de la structure.

Cette connaissance préalable est également importante dans le cadre de l'implémentation d'un système multi-agents : si certaines portions de l'algorithme présentent un coût constant, comme l'initialisation des variables globales, d'autres sont directement liées à ces paramètres d'exécution. La mise à jour des agents peut ainsi présenter un coût linéaire par rapport au nombre d'agents présents dans le modèle. Au contraire, la recherche dans un voisinage dans un espace de simulation en deux dimensions représente un coût évoluant de manière quadratique.

Suivant les modèles, cette topologie spatiale est présente sous forme de structure de données explicite, dans le cas du modèle proie-prédateur, ou de manière plus abstraite. Ce second type est illustré par le graphe d'accessibilité utilisé pour le modèle MIOR plus loin dans ce mémoire.

Cette variation des coûts en fonction des paramètres d'entrée influence directement les performances obtenues par une implémentation particulière, et en particulier par une adaptation sur le GPU, en application de la loi d'Amdhal. Si les portions adaptées sur GPU sont parallélisables et deviennent de plus en plus coûteuses avec l'augmentation du nombre d'agents ou de la taille de l'espace de simulation, l'utilisation de cette plate-forme sera alors d'autant plus avantageuse en regard au CPU que ces deux paramètres augmentent. Au contraire, si le coût de ces portions parallélisées reste constant ou très limité dans la plage de paramètres qui intéresse le chercheur, les gains offerts par telle adaptation seront plus limités.

Dans toute expérimentation, il apparaît donc prometteur d'identifier les traitements dont le coût augmente rapidement en regard de l'espace de simulation du système, puis d'évaluer l'impact sur les performances de ces traitements sur différentes tailles du système, si possible avec un facteur de mise à l'échelle assurant un comportement identique de la simulation.

6.2 Développement d'un modèle sur GPU

La première approche de parallélisation d'un système multi-agents sur GPU est de réaliser l'ensemble du modèle sur ce support. De nombreuses réalisations de ce type ont été présentées dans la section 3.3.1. Cette approche requiert toutefois la maîtrise du modèle de programmation GPU, ainsi que la gestion de nombreuses problématiques par le concepteur du modèle. Nous détaillons ces différentes problématiques dans la suite de cette section.

6.2.1 Implémentation des structures de données

OpenCL ne spécifie pas, en tant que standard basé sur le langage C, de structures telles que les listes chaînées ou les grilles : les seules structures de données gérées directement par le langage sont les types primitifs, les structures et les tableaux statiques. Pour pouvoir adapter un modèle multi-agents sur GPU, il est donc nécessaire de convertir toutes les structures décrites dans le modèle en combinaison d'un ou plusieurs de ces types de données.

Cette restriction limite le modèle à l'utilisation de tableaux statiques à une dimension ou à des structures spécifiques telles que les textures pour représenter les grilles souvent rencontrées dans les systèmes multi-agents. Cette conversion en structures de données implique des choix de représentation. La Figure 6.1 illustre ainsi deux manières possibles de représenter une matrice en OpenCL, sur la base d'une linéarisation ligne par ligne ou colonne par colonne. Chacune de ces alternatives est adaptée à un mode particulier d'accès aux données. La linéarisation par ligne permet

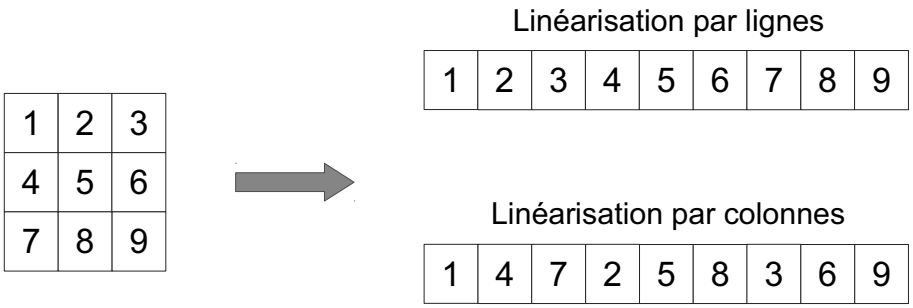

FIGURE 6.1 – Linéarisation par lignes ou par colonnes d'une grille à deux dimensions

de tirer un meilleur parti de la localité des données en cas de traitements sur des lignes complètes de matrice, et en particulier du chargement des données par paquet par la carte graphique. Elle est également adaptée à la répartition des cellules de chaque ligne entre plusieurs traitements, en permettant à la carte graphique de grouper l'accès aux données voisines en mémoire. Ces deux avantages disparaissent dans le cas d'un traitement de la matrice colonne par colonne où il sera alors nécessaire de préférer une linéarisation par colonnes, pour les mêmes raisons.

La conversion de ces structures est rendue plus difficile en OpenCL, par rapport au C, par l'impossibilité d'utiliser des pointeurs à l'intérieur des structures de données GPU [ope]. Toute structure basée sur ce concept (liste chaînée, graphe...) doit également être représentée sur GPU sous forme d'un ou plusieurs tableaux statiques ou d'objets texture avec une perte en flexibilité d'utilisation.

La gestion de ces représentations alternatives doit être prise en charge à la fois au niveau du système hôte, pour permettre l'allocation et l'initialisation de ces structures de données, et au niveau du périphérique d'exécution. La plupart des bibliothèques OpenCL décrites dans notre contexte se focalisent sur l'utilisation de structures et d'opérations du côté hôte uniquement, en délégant à l'implémentation le soin de gérer les données indiquées, même si des bibliothèques comme ELMO [FVSS13] facilitent certains traitements au niveau de l'exécution GPU.

6.2.2 Allocation et gestion de la mémoire

OpenCL requiert également une gestion totalement manuelle de la mémoire de la part du développeur, au contraire de langages comme Java ou de nombreuses plates-formes multi-agents qui intègrent un mécanisme de ramasse-miettes chargé de la libération automatique des ressources.

Cette gestion manuelle de la mémoire est basée sur un mécanisme de comptage des références à chaque objet natif OpenCL, où la mémoire est libérée une fois que plus aucune référence n'existe sur l'objet. Ce mécanisme implique une vigilance importante de la part du développeur pour s'assurer que les ressources natives sont effectivement libérées après utilisation.

Une dernier aspect important de la gestion en mémoire est que son allocation repose, comme en C, sur la demande d'un espace de taille fixée au système. La zone mémoire obtenue n'est pas typée, et peut être utilisée indifféremment pour n'importe quel type de donnée. Si cette généricité permet d'envisager la réutilisation du même espace mémoire pour plusieurs données du modèle, elle empêche également toute vérification automatique de son utilisation par le compilateur ou la plate-forme d'exécution.

Les espaces mémoires obtenus sont également caractérisés, comme en C, par l'absence de toute vérification de la validité des accès. Il est ainsi aisément possible, en manipulant des tableaux

de données, d'écrire ou de lire à des adresses mémoires invalides. Ce type d'erreur n'est pas nécessairement détecté par la plate-forme d'exécution, et impose une grande vigilance dans l'accès aux structures de données pour éviter de corrompre silencieusement les données de la simulation.

6.2.3 Gestion de l'exécution

Au-delà de la gestion de la représentation des données du modèle, la réalisation d'un modèle multi-agents directement sur GPU pose la question de la granularité de parallélisation de l'exécution et du type de distribution des données à retenir : est-il préférable d'effectuer la totalité de la simulation en un seul lancement de kernel ? Vaut-il mieux, au contraire, découper l'exécution en plusieurs kernels correspondants à des données et des traitements différents ? Comment découper l'algorithme utilisé en threads ?

Granularité de parallélisation

La décomposition du traitement à effectuer en nombreux threads est un pré-requis important pour une exécution efficace du nombre important de coeurs fournis par l'architecture matérielle GPU. L'exécution d'un agent à la fois n'a ainsi aucun intérêt si l'exécution de cet agent n'est pas parallélisable et coûteuse en soi, du fait des coûts de transfert, du faible taux d'occupation et des temps d'exécution obtenus.

Une manière d'assurer ce découpage est d'exécuter tous les agents du modèle de manière simultanée (parallélisation "en largeur"), en associant chaque agent à un thread d'exécution. Une alternative est une parallélisation dite "en profondeur", dans laquelle chaque agent réalise un ou plusieurs calculs coûteux à même d'être largement parallélisés, de manière à justifier son exécution indépendante sur GPU.

Le choix du nombre de kernels devant être utilisés pour la réalisation de la simulation dépend de plusieurs facteurs :

L'algorithme a un impact primordial sur le type de parallélisation retenue. Il peut être aisément parallélisable, s'il s'agit par exemple d'un traitement indépendant sur chaque élément d'une grille d'entrée, ou au contraire imposer de nombreuses synchronisations entre agents. Si tous les threads d'une exécution doivent conclure les mêmes opérations avant de pouvoir continuer l'exécution de la simulation, des barrières d'exécution deviennent nécessaires. Le découpage en programmes distincts est un autre moyen d'obtenir implicitement ce type de synchronisation.

Le nombre de threads (work-items) devant être lancés. Le nombre et l'organisation des threads associés à un kernel sont fixés au niveau du lancement et ne peuvent pas être modifiés en cours d'exécution. Si plusieurs populations d'agents de tailles variées, ou des structures de taille très variées, existent dans le modèle, il est préférable d'utiliser un découpage adapté à chaque population, plutôt qu'un unique découpage peu adapté, de manière à maximiser l'utilisation des threads lancés. Le lancement en un seul kernel d'une simulation proie-prédateur comprenant 2000 proies et 100 prédateurs implique ainsi l'utilisation de 2000 works-items, si chaque population doit être traitée en parallèle. Si ce découpage est optimal en regard du nombre de proies, seuls 5% des threads seront effectivement utilisés pour la simulation les prédateurs, ce qui induit une irrégularité importante en temps de traitement sur GPU.

Les dépendances de données. Le type des données manipulées et leur utilisation dans les différentes phases de l'exécution ont également leur importance en termes de découpage. Ainsi, les objets textures ne sont accessibles qu'en lecture ou en écriture au sein d'un même lancement.

Les dépendances de transfert. Certaines données peuvent également demander des traitements intermédiaires sur CPU avant de poursuivre l'exécution sur GPU. Dans ce cas, l'application de ces traitements requiert une récupération des données par le CPU, un calcul, puis une nouvelle copie sur GPU, et les coûts de transferts associés.

La fréquence et le nombre des transferts devant être réalisés. Le fait de déléguer une partie de la simulation au GPU impose l'échange régulier de données entre les deux plates-formes d'exécution, pour la mise à jour de la simulation ou la récupération de résultats. S'il est plus intéressant d'effectuer tous les traitements en un minimum de lancements, pour limiter les coûts associés à ces transferts, des contraintes en termes de découpage d'exécution, évoqués dans le paragraphe précédent, ou en termes de visualisation de la simulation peuvent imposer le découpage de la simulation en plusieurs étapes de traitement.

Distribution des données

L'exécution en OpenCL est basée sur le découpage de la tâche en une grille à une, deux ou trois dimensions de threads (ou work-items) exécutant le même traitement pour l'accès aux données. Chacun de ces threads est associé à une position dans cette grille. Dans ces circonstances, il est naturel de souhaiter adapter ce découpage d'exécution au découpage retenu pour les données, de manière à pouvoir utiliser ces positions dans les traitements. Pour les systèmes multi-agents, cette association revient souvent à associer un thread à chaque agent en cours de traitement du système. Cette association entre index dans la grille d'exécution et agent ouvre la voie à deux possibilités de représentation des données des agents dans le modèle, illustrées par la Figure 6.2.

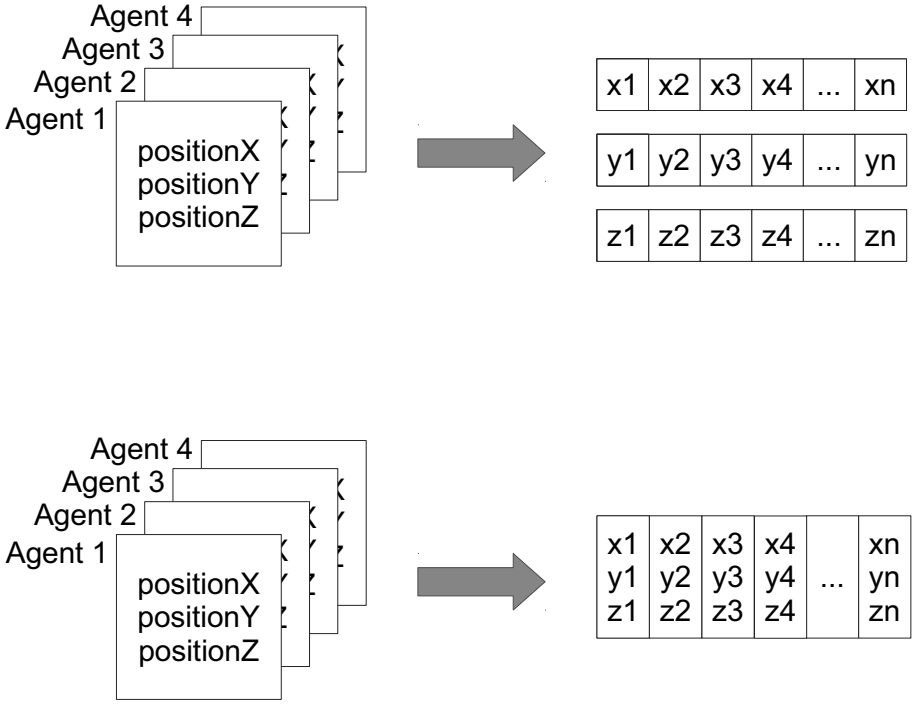

FIGURE 6.2 – Représentations possibles des propriétés multi-agents en termes de structures OpenCL

La première approche pour le stockage de ces données est d'associer à chaque ensemble de propriétés son propre tableau à une dimension. Dans le cas où chaque agent est associé à un triplet de coordonnées réelles (x, y, z) représentant sa position dans l'espace de simulation, ce processus se traduit par le stockage de ces informations sous forme de trois tableaux de réels, un par

coordonnées. Le nombre de tableaux à une dimension obtenu est directement lié au nombre de propriétés différentes associées à chaque agent : si ce nombre augmente, le nombre de tableaux nécessaires, et donc le nombre de paramètres devant être passés au programme, augmente également. Si le modèle agent décrit plusieurs types d'individus différents, dotés ou non de certaines propriétés, un choix se pose alors au concepteur du stockage mémoire : soit d'effectuer l'union de tous les ensembles de propriétés stockés dans le modèle, soit de dédier à chaque type d'agent son ensemble de tableau, sans réutilisation. Le premier cas permet de limiter l'explosion du nombre de paramètres quand de nombreuses informations sont communes à toutes les catégories d'agents, comme une position. Elle signifie cependant que chaque tableau de propriétés n'est plus applicable à l'ensemble des agents, et comprendra donc des "lacunes" correspondant aux agents pour lesquels cette propriété n'est pas définie. Ces lacunes peuvent avoir à être ignorées au moyen de tests nuisant à l'utilisation effective des coeurs d'exécution en forçant le matériel à évaluer les deux branches de la condition. Le second cas évite ce problème, chaque propriété étant définie pour l'ensemble du type d'agent décrit, au prix d'une multiplication du nombre de structures de données devant être gérées dans la programmation du modèle.

Une seconde approche est une programmation "objet", où chaque ensemble de propriétés est stocké dans sa propre structure spécialisée. Il est dans ce cas important de prendre en compte l'augmentation de l'espace mémoire engendré par l'alignement des attributs de chaque structure. Une règle simple pour minimiser ces pertes mémoires est, autant que possible, d'organiser les propriétés par ordre de taille dans la description de la structure. Cette opération n'est pas effectuée automatiquement par la plupart des compilateurs pour éviter de produire des représentations binaires différentes, et donc des incompatibilités, en fonction du logiciel utilisé pour compiler chaque portion de programme.

Le choix de l'une ou l'autre de ces solutions est à la fois gouverné par les considérations décrites en termes de stockage (nombre de paramètres nécessaires, de types distincts, recouvrement ou non de nombreuses propriétés entre les agents) et par l'algorithme lui-même, et plus particulièrement l'ordre et le mode d'accès aux données. Ainsi, dans le cadre d'un kernel où tous les agents accèdent à une seule propriété, un stockage des propriétés par tableaux permet de récupérer cette information pour tous les agents voisins en une seule requête, du fait des lectures par paquet effectuées par le GPU. Si l'exécution de chaque agent est au contraire basée sur l'accès à ses propres données uniquement, l'utilisation de tableaux de structures permet dans ce cas de récupérer toutes les propriétés associées à l'agent de manière simultanée.

6.2.4 Diagnostic des erreurs

Le développement et le diagnostic des erreurs de modèles complets sur GPU sont rendus difficile par l'impossibilité d'écrire des informations de diagnostic dans une sortie ou dans un fichier pendant l'exécution OpenCL.

La récupération des erreurs est également rendue ardue par la nature asynchrone de l'exécution sur GPU. Cette asynchronisme se traduit par une vérification des erreurs sur de nombreuses implémentations d'OpenCL au moment de la soumission de la prochaine opération seulement. Cette soumission peut n'avoir aucun rapport avec l'exécution fautive ou être éloignée, ce qui retarde la découverte du dysfonctionnement.

Enfin, la remontée des erreurs est effectuée, comme en C, sous forme de codes de retours prédéfinis n'indiquant pas la source et la position précise de l'erreur dans le programme. Ces codes d'erreurs sont de plus susceptibles de varier entre implémentations d'OpenCL.

Il est possible de pallier, dans une certaine mesure, à ces difficultés de diagnostic en examinant régulièrement l'évolution des structures de données de la simulation au cours du processus de développement, ou en écrivant des informations de diagnostic dans des structures de données de sorties spécialisées. Cette solution ne fonctionne cependant pas en cas d'interruption du kernel d'exécution, et permet surtout la validation du bon fonctionnement de la simulation.

6.3 Parallélisation de certains traitements

Une seconde approche de parallélisation d'un système multi-agents sur GPU est de ne réaliser que certains traitements sur carte graphique et de conserver le reste de la simulation sur CPU.

Cette approche permet de conserver les aspects complexes de la simulation sur le CPU, comme dans le cas de la gestion des listes de positions de chaque individu dans le modèle proie-prédateur. Elle permet de tirer parti de la puissance du GPU pour des traitements parallélisables et coûteux en temps de calcul, et donc d'accélérer l'exécution du modèle dans son ensemble, mais requiert elle aussi une expertise en programmation pour implémenter ces opérations. L'utilisation de deux langages et architecture d'exécution différentes dans la même simulation impose également de pouvoir transformer les structures de données utilisées sur CPU en structures équivalentes sur GPU, et inversement, pour les informations communes aux deux modes d'exécution.

6.3.1 Adaptation des structures de données CPU

Le langage Java est associé à une très vaste bibliothèque de structures de données prédéfinies à la disposition des développeurs. Cette base commune permet au concepteur de nouvelles bibliothèques Java de disposer des structures les plus courantes, comme les dictionnaires, les listes ou des files d'attente sans avoir besoin d'en développer sa propre implémentation ou de recourir à des bibliothèques externes. Cette standardisation des structures de données facilite également leur partage et leur adaptation sous forme de structure OpenCL.

6.3.2 Exécution synchrone ou asynchrone

Les simulations multi-agents sont basées sur le découpage de leur évolution en étapes discrètes déclenchées de manière régulière (pas de temps) ou par certains événements. L'exécution du modèle ou de la plate-forme est également souvent synchrone, pour faciliter la gestion de la mise à jour de la vue ou des résultats de la simulation.

Le modèle d'exécution offert par OpenCL est, pour sa part, basé sur une exécution asynchrone par le biais d'un mécanisme de file d'attente. L'utilisation d'opérations asynchrones favorise la réalisation de tâches en parallèle de l'exécution de traitements et l'indication de dépendances entre tâches, tandis que l'exécution synchrone permet un blocage implicite de l'exécution du modèle multi-agents dans l'attente de résultat.

Dans ces circonstances, une première approche est d'interrompre le déroulement de la simulation sur CPU pour chaque traitement sur GPU en effectuant des soumissions synchrone. Le déroulement de l'exécution obtenue est illustré par la Figure 6.3. Cette approche revient à bloquer l'exécution de la totalité du modèle tant que le traitement GPU n'est pas terminé, alors qu'il serait par exemple possible d'exploiter le CPU pour réaliser des opérations d'affichage ou encore la mise à jour ou l'exécution d'autres parties du modèle.

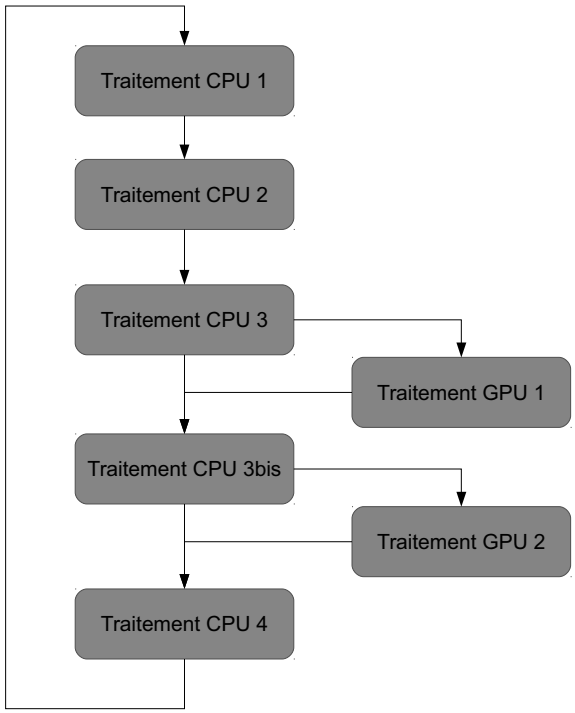

FIGURE 6.3 – Synchronisation bloquante de chaque traitement GPU

Une seconde approche est de lancer l'ensemble des traitements GPU en une seule fois pour obtenir le type de déroulement de l'exécution illustré par la Figure 6.4. Dans ce cas, les opérations GPU sont lancées à la suite, au moyen du système de dépendances fourni par OpenCL, et le programme CPU attend la fin de l'ensemble des traitements plutôt que chaque opération intermédiaire. Cette approche permet de tirer parti du CPU et du reste des ressources matérielles de la machine pendant l'exécution sur GPU, au prix toutefois d'une certaine désynchronisation des traitements à l'intérieur de l'itération, du fait de l'exécution de plusieurs traitements consécutifs en arrière plan, avant leur réintégration dans la simulation.

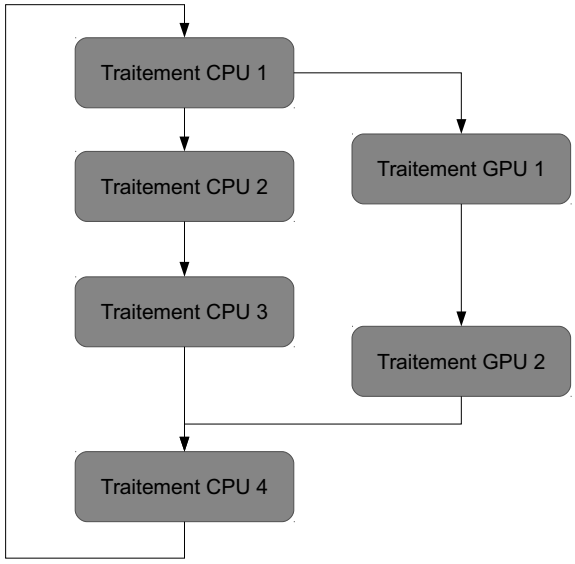

FIGURE 6.4 – Synchronisation bloquante à la fin des traitements GPU

Ces deux approches offrent comme avantage la conservation de l'ensemble des traitements au sein de la même itération. Dans les deux cas, l'ensemble des traitements GPU est terminé et réintégré à chaque pas de temps. Elles montrent cependant leurs limites dans deux situations :

— Pour des traitements de fond très longs. Dans ce cas, la poursuite de l'itération est susceptible d'être bloquée pendant un temps important.
— Si le recours au GPU à chaque itération ne permet pas une exécution efficace. Il est dans ce cas possible de recourir à un système de mise en attente des traitements pour grouper par lots les lancements à effectuer.

Une troisième solution est donc de désynchroniser l'exécution des itérations de la simulation sur CPU et des traitements GPU. De cette manière, il devient possible d'exécuter plusieurs itérations avant de récupérer les données résultats, ou encore de rassembler les travaux de plusieurs itérations avant exécution sur GPU. Cette approche est la plus délicate à gérer, et demande une connaissance fine du modèle simulé, pour déterminer le nombre d'itérations de décalage pouvant être admises entre traitements CPU et GPU, puisque le découpage en pas de temps n'est plus respecté.

Ce choix du niveau de synchronisation implique également un choix du nombre d'étapes de lancements utilisées pour effectuer les traitements, en fonction du taux de contrôle et des différents découpages de parallélisation pour chaque portion de calculs, comme nous l'avons vu dans le choix de la granularité de parallélisation pour l'implémentation d'un modèle sur GPU.

Dans des modèles qui ne requièrent que des mises à jours partielles ou sporadiques du système, il peut être difficile de rassembler d'assez grandes quantités de traitements pour bénéficier d'une exécution sur GPU. La mise en place d'un mécanisme de file d'attente est alors nécessaire, de manière à rassembler les traitements en attente et les exécuter par lot de manière asynchrone, plutôt que sous forme de lancements indépendants. Cette approche, illustrée par la Figure 6.5, permet ensuite la récupération des résultats par la simulation au moyen d'une autre file de données.

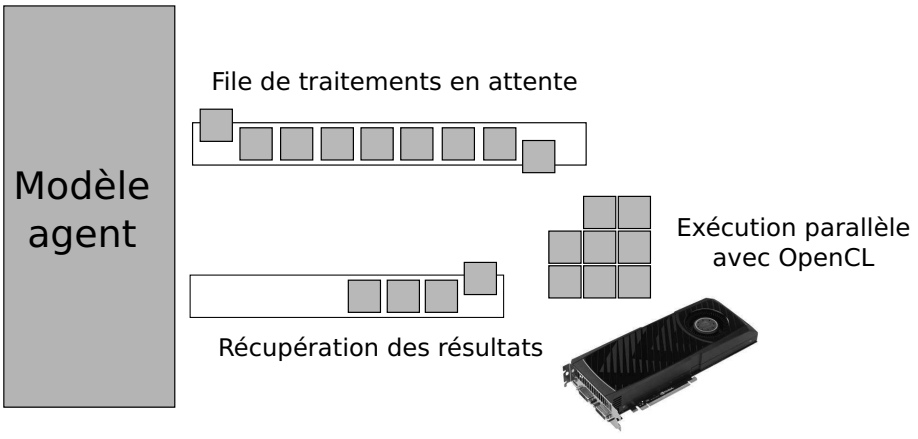

FIGURE 6.5 – Regroupement des traitements en attente pour une exécution par lot sur GPU

6.3.3 Gestion de la mémoire

Dans le cas où seuls certains traitements sont parallélisés sur GPU, il est possible de distinguer trois cycles de vie distincts pour les structures mémoires OpenCL :

— Tout la durée d'utilisation du GPU. Ce cycle de vie comprend les structures de données, telles que le contexte d'exécution OpenCL ou les files de soumission de traitement, qui sont utilisées pour toute la durée de la simulation. La création d'un nouveau contexte est effectuée soit au moment de l'initialisation du modèle ou de la plate-forme, soit au moment du premier lancement OpenCL. Sa libération peut être gérée de la même manière, soit implicitement par la plate-forme ou la sortie du programme, soit explicitement par la simulation. La libération explicite du contexte d'exécution est importante dans le cas où le même programme créerait d'autres contextes par la suite, pour éviter toute perte de ressources liée aux anciens contextes d'exécution GPU encore présents en mémoire.

— Le lancement d'un ensemble de fonctions apparentées ou de la même fonction à plusieurs reprises. La même opération peut être utilisée à chaque itération du modèle, ou plusieurs fois dans une même itération pour des fonctions différentes. C'est le cas, par exemple, dans le modèle proie-prédateur, de la fonction de déplacement utilisée à la fois pour les proies et les prédateurs. Dans ces circonstances, il est intéressant de ne pas avoir de nouveau à préparer et compiler les mêmes programmes GPU à chaque appel de la fonction. Cette réutilisation impose cependant, une fois de plus, la gestion de la mémoire par la plate-forme ou la simulation. Une solution est de permettre au développeur d'indiquer s'il est souhaitable de conserver les programmes entre chaque exécution, ou si la totalité des structures nécessaires au lancement doit être de nouveau préparée à chaque traitement.

— Le lancement d'un seul traitement. Comme nous l'avons vu dans notre présentation du modèle de programmation GPU, l'exécution du lancement d'un programme implique une copie des données d'entrée sur le périphérique avant l'exécution, suivi d'une récupération des résultats une fois l'exécution terminée. Cette démarche donne lieu à l'utilisation de nombreux objets temporaires, susceptibles d'être alloués et libérés automatiquement par l'implémentation du traitement sur GPU, de manière transparente pour le reste de la simulation.

La prise en compte de ces trois cycles de vie est importante pour permettre une gestion correcte de l'allocation et de la libération de la mémoire native, tout en minimisant la quantité de gestion manuelle de la mémoire devant être effectuée par le reste de la simulation, souvent basée sur un mécanisme de ramasse-miettes. L'utilisation de ce mécanisme n'est hélas pas possible pour automatiser toutes ces libérations, car son invocation n'est pas garantie par de nombreux langages et de nombreuses plates-formes tant qu'il reste de la mémoire disponible. Les structures natives n'étant que partiellement situées dans les données visibles par le programme, la saturation des ressources GPU ne déclenche pas ce mécanisme de récupération mémoire. La prise en compte du ramasse-miettes peut cependant venir en complément de cette gestion du cycle de vie des différentes structures GPU, en filet de sécurité supplémentaire permettant la libération de la mémoire.

6.4 Utilisation de traitements parallélisés existants

Les deux approches précédentes correspondent à un premier scénario de parallélisation, où le concepteur dispose d'une expertise GPU à même de lui permettre de réaliser tout ou partie de la simulation sur cette architecture. Elles ne permettent cependant pas directement un second scénario d'utilisation, le recours au GPU sans connaissance particulière de l'architecture d'exécution ou

d'un langage de programmation GPGPU.

Une troisième approche possible d'adaptation d'une simulation multi-agents est de réutiliser des opérations de haut niveau déjà parallélisées sur GPU, en considérant ainsi la programmation GPGPU comme une boîte opaque permettant d'améliorer les performances du programme. Elle ne requiert pas une gestion directe des ressources ou de l'exécution GPU par la simulation ou la plate-forme multi-agents.

L'utilisation de fonctions génériques implique cependant, comme dans le cas de l'utilisation de bibliothèques matricielles ou d'algèbre linéaire, de ramener les portions de la simulation à paralléliser à des opérations et des structures de données standards pour pouvoir utiliser un traitement générique.

Dans le modèle proie-prédateur, il est ainsi intéressant de ramener la mise à jour des végétaux ou la somme des énergies à des opérations matricielles, plutôt qu'à un traitement ponctuel par chaque agent dans l'algorithme.

6.4.1 Contrôle du contexte d'exécution

OpenCL sélectionne par défaut automatiquement une périphérique d'exécution local présent sur la machine, CPU ou GPU, si aucun type de matériel n'est indiqué. Ce comportement facilite le lancement de calculs sans avoir à choisir explicitement un support parmi les solutions d'exécution disponibles.

Il est toutefois important, dans le l'utilisation de traitements parallélisés pré-implémentés, que le concepteur de modèle ait la possibilité de contrôler le type de périphérique d'exécution utilisé, pour plusieurs raisons :

— Dans le cas d'une parallélisation hybride, pour garantir que l'exécution des opérations aura bien lieu sur GPU, Xeon Phi, ou FPGA plutôt que sur le processeur déjà utilisé pour le reste de la simulation.
— Pour permettre la comparaison entre différents matériels, de manière à mesurer l'impact sur les performances de différentes alternatives d'exécution.

Pour faciliter au maximum l'utilisation de ces traitements agents parallélisés, et assurer leur portabilité sur de nombreuses machines, quelles que soient les ressources locales disponibles de manière générale, il est toutefois important de laisser la possibilité d'une sélection implicite du type de périphérique utilisé. Ce choix par défaut peut alors être celui du premier périphérique disponible, ou reposer sur des heuristiques plus complexes, privilégiant la solution disponible considérée comme la plus rapide en calcul.

6.4.2 Gestion de la mémoire

L'utilisation de traitements parallélisés existants libère l'utilisateur de la responsabilité de la gestion des structures mises en jeu de manière interne par les traitements, mais requiert toujours la prise en compte des structures persistantes entre plusieurs traitements.

Le fait de devoir préparer à nouveau un contexte d'exécution et compiler un programme GPU pour chaque opération peut poser problème dans le cas de calculs courts. Ces préparations prennent alors en effet un temps important en regard du temps effectivement passé en calcul sur le périphérique.

Une solution à ce problème est de regrouper les traitements de même type sous forme de mo-

dule, à même d'être préparé une seule fois et réutilisé pour lancer plusieurs fois le même traitement. Dans ce cas, le lancement d'un traitement parallélisé sur GPU peut être décomposé très simplement :

— Création d'un contexte.
— Instantiation d'un module d'exécution.
— Lancement de *n* traitements fournis par ce module.
— Libération du module après utilisation.
— Libération du contexte.

Il est possible d'envisager une libération automatique de toutes les ressources associées à ce contexte au moment de sa destruction. Cette démarche simplifie alors encore davantage ce type d'utilisation dans une simulation ou une plate-forme multi-agents existants.

6.4.3 Conception de nouveaux traitements génériques

Si l'utilisation de traitements génériques permet l'utilisation du GPU sans expertise de l'architecture, elle requiert également une démarche de conception particulière pour le développement de ces traitements. Si ce processus se rapproche de la seconde approche de parallélisation d'une simulation multi-agents, une adaptation partielle sur GPU, elle s'en différencie toutefois par la volonté de proposer une interface générique, découplée de tout modèle spécifique.

Cette volonté de proposer une opération utilisable le plus largement possible se retrouve à la fois dans le choix des structures de données, de type grille, vecteur ou scalaires plutôt qu'objet, et dans l'interface d'appel. Cette dernière est conçue de manière à permettre au concepteur d'indiquer tous les paramètres d'entrée et de sortie de l'exécution sur GPU en une seule fois, en minimisant autant que possible les transformations ou préparations à effectuer. De cette manière, la totalité de l'exécution (préparation, copie des données, exécution et récupération des résultats) peut être prise en charge une seule fois par la fonction générique, plutôt que par le concepteur.

Il est possible d'illustrer cette démarche de généralisation sur la mise à jour des ressources végétales de l'environnement dans le modèle proie-prédateur.

Une première approche de mise à jour de cette grille est d'indiquer directement les facteurs de croissance devant être appliqués sous forme de constantes dans l'implémentation OpenCL. Cette solution lie cependant fortement cette mise à jour au modèle proie-prédateur : pour pouvoir utiliser ce traitement dans un autre modèle, il est nécessaire d'effectuer une copie du code associée à ce traitement, et de modifier la valeur de ces constantes.

Une manière de rendre ce traitement générique et directement utilisable par d'autres simulations multi-agents est d'indiquer ces facteurs de croissance dans l'appel de la fonction. L'opération de mise à jour devient alors un moyen d'appliquer une transformation affine sur une grille quelconque, plutôt qu'uniquement sur un environnement proie-prédateur.

Ce type de traitement générique peut être rapproché des opérations proposées par des bibliothèques comme CuBLAS : il peut alors être exploité sans connaissance particulière du fonctionnement de l'algorithme ou du GPU avec une simple structure de grille et deux facteurs indiquant l'opération à appliquer à chaque cellule.

6.5 Synthèse

Ce chapitre nous a permis de mettre en avant deux scénarios d'utilisation du GPU dans des modèles et des plates-formes multi-agents existants.

Le premier scénario repose sur une utilisation directe des concepts de programmation GPU pour implémenter tout le modèle agent sur cette architecture. Cette utilisation de bas niveau impose cependant de nombreuses contraintes, tant en termes de représentation des données qu'en termes d'exécution, pour lesquelles il est toutefois possible de fournir des mécanismes venant en aide au développeur.

Le second scénario repose sur l'utilisation de traitements agents parallélisés existants ou implémentés par le concepteur. L'utilisation du GPU est alors totalement abstraite pour le reste de la simulation, ce qui facilite son intégration dans des modèles ou des plates-formes multi-agents pour accélérer des portions de traitements. Cette seconde utilisation, de beaucoup plus haut niveau, pose également ses propres problématiques pouvant être en partie résolues par une bibliothèque adaptée.

L'étude de ces deux scénarios a conduit notre réflexion lors de la conception de la bibliothèque MCMAS, conçue pour permettre ces deux types d'utilisations.

7

MCMAS, une bibliothèque d'exécution générique

Dans le chapitre précédent, nous avons mis en avant les deux méthodes de parallélisation de modèles multi-agents sur GPU, le portage de la totalité du modèle sur cette architecture d'exécution ou la seule adaptation de certains traitements. Dans ce chapitre, nous présentons notre bibliothèque MCMAS, qui vise à répondre à ces deux scénarios en facilitant à la fois l'utilisation du GPU sans connaissance approfondie de la plate-forme, et la réutilisation d'algorithmes et de structures pour des modèles qui souhaiteraient utiliser OpenCL.

Dans une première section, nous détaillons les objectifs auxquels répond notre bibliothèque. Nous étudions ensuite l'influence de ces objectifs sur l'architecture de MCMAS, basée sur une interface de programmation haut niveau, ne mettant pas en jeu de connaissances GPU, et une interface plus bas niveau permettant un accès complet à OpenCL. Nous présentons ensuite l'implémentation par MCMAS de la gestion du contexte d'exécution et des principales structures de données rencontrées dans les systèmes multi-agents. Nous abordons également la manière donc la bibliothèque permet le recours à des types plus spécifiques spécialisés pour l'exécution sur GPU. Enfin, nous présentons l'utilisation de MCMAS par le biais de son interface de haut niveau, puis par le biais de son interface de bas niveau pour développer de nouvelles fonctionnalités de manière indépendante ou au sein de la bibliothèque.

7.1 Présentation générale

MCMAS [1] est une bibliothèque Java développée pour permettre la parallélisation efficace de systèmes multi-agents sur GPU. Son exécution est basée sur le modèle de programmation OpenCL, de manière à permettre l'utilisation une variété de supports la plus large possible, allant d'architectures many-cores comme les GPU aux processeurs traditionnels le cas échéant.

La volonté centrale derrière la conception de MCMAS est d'offrir deux niveaux d'utilisation au développeur, basés sur des interfaces de programmation distinctes :

— Une interface d'utilisation OpenCL accessible en Java incluant tous les éléments nécessaires à l'accès au modèle de programmation, dans le cadre d'une programmation objet, ainsi que des structures de données et des mécanismes d'aide à l'exécution prêts à être utilisés.
— Une interface de haut niveau permettant l'appel d'implémentations optimisées des traitements multi-agents les plus courants. Cette interface permet la parallélisation aisée de modèles existants sans connaissance GPU, tant depuis des modèles d'agents directement

1. Many-Core Multi Agent Systems

réalisés en Java, que depuis des plates-formes existantes telles que MadKit, NetLogo ou GAMA. L'utilisation de ces fonctions est conçue pour être la moins intrusive possible et réduire au minimum le nombre de structures de données manipulées pour réaliser un traitement.

Cette volonté de fournir deux interfaces d'utilisation a une influence directe sur l'architecture retenue pour la bibliothèque, que nous présentons dans la section suivante.

7.2 Architecture

L'architecture de MCMAS peut être vue comme l'empilement de deux couches applicatives distinctes et complémentaires en termes d'utilisation :

— Une base commune, MCM [2]. Cette couche permet l'accès au modèle de programmation OpenCL par le biais d'une interface objet. Elle comprend de nombreux outils de gestion de l'exécution et des structures de données communément utilisées pour l'implémentation de modèles et de traitements multi-agents utilisables sur GPU. Elle représente l'interface de bas niveau de MCMAS, permettant la réalisation de nouveaux traitements avec la bibliothèque.
— Un ensemble de plugins fournissant des traitements génériques réutilisables. Cette couche applicative repose sur l'interface de programmation MCM pour proposer des traitements multi-agents déjà implémentés sous la forme de fonctions de haut niveau, comme la mise à jour de l'environnement ou le calcul de déplacements. Cet ensemble de plugins est extensible au moyen de la couche applicative MCM ou en encapsulant l'appel à d'autres plugins, de manière à ajouter le support de nouvelles opérations et de nouvelles structures de données à la bibliothèque.

Ces deux couches applicatives sont représentées dans la Figure 7.1, qui illustre également les concepts de plugins et de contexte d'exécution proposés par notre bibliothèque. Des fonctions différentes sont dévolues à chacune de ces deux couches, comme nous le présentons par la suite.

7.2.1 Une interface de bas niveau : MCM

MCM représente la base de la bibliothèque MCMAS, et offre l'accès à l'interface de programmation de bas niveau de notre bibliothèque et à tous les mécanismes de l'exécution sur GPU. Elle reprend la gestion d'un grand nombre de problématiques liées au développement et au lancement de programmes OpenCL.

MCM facilite la gestion de la mémoire native en intégrant la gestion du ramasse-miettes et en permettant de manière optionnelle d'associer toutes les structures mémoires et applicatives à leur contexte d'allocation, de manière à assurer leur libération automatique au moment de la destruction de ce contexte.

Cette couche applicative fournit les structures de données communes de type grille, vecteur ou objet proposées par MCMAS, à la fois du côté Java et OpenCL de l'exécution. La conversion de ces structures MCMAS vers et depuis les types de données Java les plus courants est également incluse à ce niveau de la bibliothèque.

MCM repose sur la couche d'adaptation JOCL pour l'accès à l'interface de programmation

2. Many-Core Manager

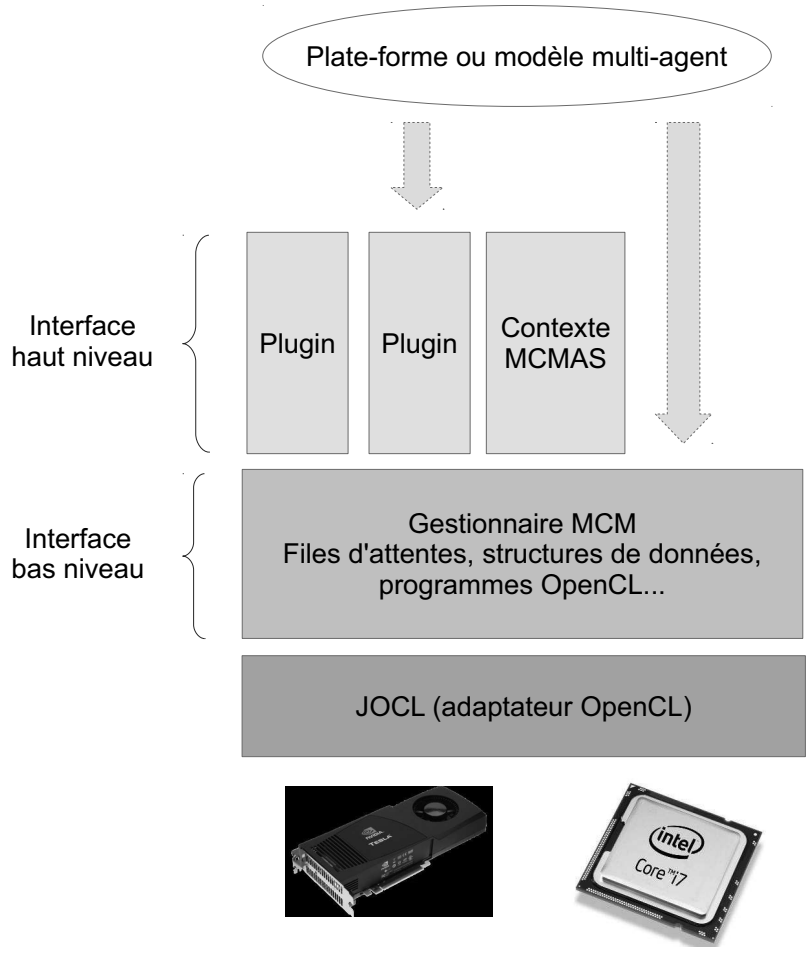

FIGURE 7.1 – Architecture de MCMAS

native. Le choix de cette solution est motivée par son minimalisme illustrée par une interface de programmation identique à OpenCL transposée en Java. D'autres bibliothèques, telles que LWJGL, proposent des implémentations OpenCL déjà orientées objets, sur lesquelles il est cependant plus malaisé de redéfinir une autre gestion de l'allocation ou de la libération de la mémoire.

7.2.2 Une interface de haut niveau basée sur des plugins

MCMAS propose, au dessus de la couche logicielle MCM, une interface de haut niveau basée sur l'utilisation d'un contexte d'exécution abstrait MCMAS et de plugins regroupant des fonctions multi-agents apparentées.

Un contexte d'exécution abstrait

Le contexte d'exécution MCMAS représente un environnement d'exécution doté de tous les mécanismes nécessaires au lancement d'un traitement OpenCL. Ce contexte vient répondre au besoin de personnaliser l'exécution des traitements en permettant au concepteur de modèle multi-agents de sélectionner le type de périphérique d'exécution souhaité et d'activer différents mécanismes à l'exécution tels que l'enregistrement du temps consacré à chaque opération (*profiling*). En l'absence d'indication, MCMAS favorise par défaut l'utilisation des cartes graphiques présentes

en local, et recourt, le cas échéant, à l'utilisation du processeur traditionnel.

Ce contexte offre également l'accès aux objets MCM sous-jacents, de manière à permettre la combinaison des deux interfaces de programmation offertes par MCMAS dans un même environnement d'exécution.

Des plugins spécialisés

Les opérations de haut niveau proposées par MCMAS sont regroupées en plugins spécialisés par thématiques, de manière à faciliter la découverte de l'interface de haut niveau de la bibliothèque et d'assurer un découpage clair des responsabilités de chaque module.

Ces plugins gèrent la réutilisation des ressources mémoires entre fonctions apparentées de manière transparente, et implémentent des classes de traitements agents pouvant être parallélisées.

Cette décomposition en modules indépendants facilite l'ajout de nouvelles fonctionnalités à MCMAS, pour gérer de nouveaux traitements ou de nouvelles structures de données rencontrés dans les systèmes multi-agents.

Une grande partie des plugins proposés avec MCMAS est incluse dans la même archive que la bibliothèque logicielle. Les plugins MCMAS peuvent également être distribués de manière indépendante : l'enregistrement de ces nouvelles opérations n'impose pas de contrainte particulière, de manière à faciliter leur intégration dans tout programme ou plate-forme multi-agents existant.

7.3 Implémentation

Après ce panorama de l'architecture proposée par MCMAS, nous présentons dans la suite quelques points d'implémentation de cette bibliothèque et nous justifions les choix correspondants qui ont été effectués.

7.3.1 Contexte d'exécution

La première fonction essentielle pour MCMAS est la création d'un contexte d'exécution et des structures correspondantes. Ce processus implique le choix d'un périphérique d'exécution sur la machine, qui est susceptible d'offrir de manière simultanée l'accès à des architectures matérielles many-core ou à des architectures multi-coeurs CPU plus traditionnelles.

Le choix de ce périphérique d'exécution peut être effectué de trois façons différentes avec MCMAS :

— En l'absence d'indication, MCMAS sélectionne automatiquement une solution d'exécution. Dans ce cas, les cartes graphiques sont retenues en priorité, avant de considérer à défaut l'exécution sur le processeur local.
— En indiquant un type de matériel d'exécution. Dans ce cas, MCMAS recherche tous les périphériques OpenCL de ce type présents sur la machine, et sélectionne par défaut le premier matériel rencontré. Une erreur est retournée le cas échéant par la bibliothèque. Il est possible d'indiquer plusieurs types de matériel à rechercher, pour reproduire une recherche en cascade similaire à celle effectuée en l'absence d'indication.
— En indiquant une implémentation et un matériel OpenCL particulier. Cette dernière approche permet de contrôler le périphérique d'exécution effectivement utilisé par MCMAS.

Elle est indispensable pour exploiter plusieurs cartes graphiques qui seraient présentes sur la même machine.

En parallèle du choix du périphérique, de nombreux paramètres de l'exécution OpenCL peuvent également être contrôlés, parmi lesquels :

— Le niveau d'optimisation à utiliser pour la compilation des programmes OpenCL.
— L'activation du support du profiling, pour permettre l'obtention de statistiques de temps sur chaque opération.
— L'utilisation par défaut d'opérations flottantes en double ou simple précision.
— Le respect strict ou non du standard IEEE sur les opérations flottantes, pour permettre l'obtention de meilleures performances sur les opérations les plus courantes.

MCMAS permet le contrôle simple de la disponibilité et de l'activation de ces fonctionnalités indépendamment des spécificités liées aux différentes versions d'OpenCL et aux extensions proposées par les implémentations du standard de chaque fabriquant. Par défaut, les fonctionnalités ne pénalisant ni les performances ni la précision des calculs sont activées, pour permettre un meilleur diagnostic au moment de la compilation d'erreurs de syntaxe ou des ressources consommées sur GPU.

7.3.2 Structures de données agents

De nombreuses structures de données, telles que les vecteurs ou les grilles, sont très couramment utilisées par les systèmes multi-agents.

Si ces structures sont fournies de manière standard sous forme de collection ou aisément réalisables par le développeur sous forme de tableaux statiques à plusieurs dimensions en Java, l'absence de support objet et les restrictions sur les types de données imposées par OpenCL rendent moins aisée leur représentation sur GPU, particulièrement en l'absence de véritable bibliothèque de données standard.

OpenCL ne permet en effet le passage et l'utilisation sur GPU que de trois catégories de données :

— Des données scalaires de type primitif ou structure.
— Des tableaux statiques à une dimension.
— Des textures.

Dans ces circonstances, il est nécessaire pour supporter les principaux types de données agents de pouvoir les convertir et les manipuler sous la forme d'une combinaison d'une ou plusieurs des structures ci-dessus.

L'approche retenue par MCMAS est de tirer parti des outils de conversion déjà fournis par JOCL entre buffers binaires Java NIO et buffers OpenCL, en facilitant la conversion et la récupération des autres types scalaires à partir de ce format. Pour ce faire, de nombreuses classes constituées de fonctions statiques de conversion sont fournies.

Dans les sous-sections suivantes, nous allons présenter quelques structures de données fournies par MCMAS, ainsi que leur implémentation.

Types primitifs

Le langage OpenCL étant directement basé sur le standard C99, l'ensemble des types primitifs communs à C et à Java sont directement utilisables dans MCMAS, à l'exception notable du type booléen.

Les types non signés disponibles en C et en OpenCL doivent cependant être manipulés avec précaution, Java ne gérant que les types signés. L'interprétation automatique du premier bit comme un indicateur de signe signifie en effet que leur valeur sera interprétée de manière différente entre le système hôte et les kernels d'exécution.

Types objets

Il est possible de passer des objets à l'exécution d'un programme OpenCL en les représentant sous forme de structure. Ces structures reprennent alors les propriétés de l'objet devant être accessibles sur le GPU.

Ces structures sont générées automatiquement par introspection à partir de toute classe Java héritant de la classe *Struct*. Seuls les attributs publics associés à la classe sont pris en compte, OpenCL ne proposant pas de contrôle d'accès. Le Listing 7.2 illustre un exemple d'objet Java et de la structure équivalente associée au niveau d'OpenCL.

```
1  public class Coord extends
       Struct {
2
3    public int x;
4    public int y;
5    public int z;
6
7    public Coord(int x, int y, int
         z) {
8      this.x = x;
9      this.y = y;
10     this.z = z;
11   }
12 }
```

```
1  struct {
2    int x;
3    int y;
4    int z;
5  } Coord;
```

FIGURE 7.2 – Exemple d'objet Java et de sa représentation en OpenCL avec MCMAS

Vecteurs

Le langage OpenCL propose un type de tableau statique hérité du langage C. Ces tableaux peuvent non seulement stocker les types communs à ces deux langages (primitifs et structures), mais également les types vectoriels introduits par l'architecture GPU. Ces tableaux, contrairement à leur équivalent Java, ne stockent cependant aucune information de taille. Dans ces conditions, deux choix sont possibles :

— L'utilisation directe de tableaux OpenCL. Dans ce cas, la taille du tableau doit être stockée et indiquée aux différentes fonctions de manière indépendante. Cette approche similaire à celle employée en C/C++ est grandement facilitée dans le cas de problèmes où la taille des structures est directement liée aux propriétés du modèle, et peut donc être déduite de manière aisée à partir de ces informations. Le nombre de positions d'individus à déplacer peut

ainsi facilement être déduit, dans le cadre du modèle proie-prédateur, à partir du nombre total de threads lancés, puisqu'un thread correspond à un agent à déplacer. Dans le cas où plusieurs structures de données partageraient une même taille, comme l'addition de deux vecteurs, le passage de la taille de la structure comme donnée importante permet de n'avoir à spécifier qu'une seule fois l'information.

— L'utilisation d'une structure de type vecteur, représentant un tableau natif OpenCL associé à une information entière de taille.

Dans ce dernier cas, une structure OpenCL peut être utilisée pour le stockage des méta-données du vecteur associée à un tableau contenant les données brutes du vecteur. Une autre solution est de stocker cette information de taille au début ou à la fin des données du tableau, sous forme d'élément supplémentaire : cette seconde approche requiert cependant que les éléments stockés dans le tableau soient d'un type compatible avec la représentation de cette taille, ainsi qu'une vigilance particulière lors du parcours des données du vecteur pour ignorer cet élément supplémentaire. La définition de fonctions d'accès spécifiques, ou d'une condition de terminaison particulière, devient alors nécessaire.

Le fait qu'OpenCL interdise l'emploi de pointeurs dans les structures de données [ope] rend l'utilisation d'une structure séparée stockant les méta-données peu pratique. Deux approches sont possibles pour contourner cette limitation, avec cependant certains problèmes :

— La déclaration de structures de tableaux dont la taille est déclarée à la compilation. Cette solution requiert la connaissance de toutes les tailles de vecteur au moment du chargement du programme, et la génération d'une structure pour chaque taille de vecteur du programme. Une alternative est la définition d'une taille maximale pour les vecteurs utilisés, et la définition à la compilation d'une seule structure de données. Cette approche peut cependant, en fonction des variations de tailles de données, occasionner des pertes mémoires importantes.

— L'utilisation de deux variables, l'une associée aux méta-données, l'autre au tableau comprenant les éléments du vecteur. Cette second approche, plus simple, ne réduit pas le nombre de variables distinctes nécessaire par rapport à un stockage séparé de la taille du tableau dans l'algorithme, et peut rapidement favoriser une explosion du nombre de paramètres requis pour le fonctionnement du programme.

Au vu de ces problématiques, deux implémentations des vecteurs sont proposées par défaut par MCMAS, dont la représentation mémoire est illustrée sur la Figure 7.3 :

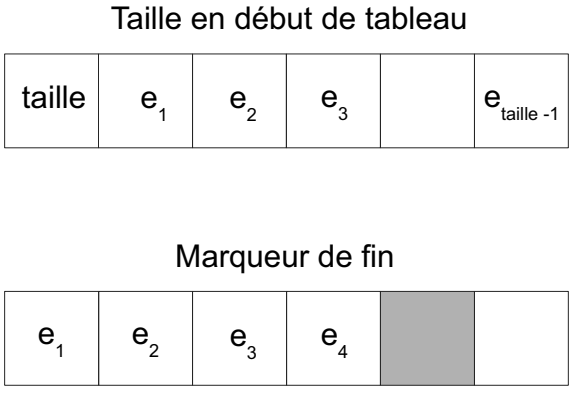

FIGURE 7.3 – Représentations de vecteurs proposées par MCMAS

— Une représentation basée sur un tableau statique contenant comme premier élément le nombre d'éléments stockés. Ce stockage au début du tableau est motivé par la nécessité

de disposer de cette information à un emplacement aisément déductible sans connaître la taille du vecteur.

— Une représentation basée sur un marqueur de fin. Cette représentation ne permet plus l'accès direct à la taille du tableau, mais requiert un parcours du tableau pour obtenir cette information. Ce parcours est rendu nécessaire dans le cas de l'application d'un traitement à chaque élément ou de la copie des données, et n'est donc pas pénalisant pour ce type d'opération. Il pose toutefois problème pour insérer ou récupérer un élément particulier du tableau, s'il est nécessaire de procéder à la validité de l'indice indiqué par rapport aux limites du tableau.

Collections

En complément des tableaux statiques, le langage Java propose un vaste ensemble de collections, correspondant aux structures de données les plus connues et utilisées dans le domaine de l'algorithmique :

— Listes simplement et doublement chaînées
— Dictionnaires
— Matrices
— Piles, files...

Ces structures de données sont représentées en Java sous la forme de type objets dérivés de la classe de base Collection, qui assure la disponibilité de nombreuses méthodes communes d'insertion, de suppression, de parcours des éléments ou de récupération de la taille.

Cette généricité des opérations facilite la conversion de ces collections vers et depuis des tableaux statiques au niveau de MCMAS à l'aide d'un ensemble de primitives de conversion intégrées à MCM. Les positions des individus dans le modèle proie-prédateur sont ainsi, dans notre implémentation, basées sur une collection d'entiers convertie en tableau statique au niveau du GPU.

Grilles

Une structure souvent rencontrée dans les systèmes multi-agents est la grille. Suivant le type de modèle représenté, elle peut constituer l'ensemble des données (cas d'une implémentation du modèle à base d'automates cellulaires) ou un simple moyen de discrétiser un espace de simulation (cas du modèle proie-prédateur).

La solution la plus directe pour représenter une telle structure dans de nombreux langages est l'utilisation de tableaux de tableaux, ou tableaux à plusieurs dimensions. Dans ce cas, l'accès aux données est effectué au moyen d'une double indexation du contenu, du type `tableau[x][y]`.

Cette implémentation n'est cependant pas directement possible en OpenCL du fait de la limitation des tableaux à une seule dimension. Dans ces circonstances, plusieurs implémentations alternatives sont proposées par MCMAS, en fonction du type de grille et d'accès souhaités sur GPU.

La solution la plus simple à cette limitation en nombre de dimensions est la linéarisation de la grille sous forme d'un tableau à une dimension.

Dans ce cas, l'accès à l'élément situé aux coordonnées (x, y) du tableau revient au calcul d'un unique index basé sur ces deux dimensions, et la largeur ou hauteur de la grille, suivant le sens de

linéarisation retenu, en lignes ou en colonnes. Si x correspond à l'abscisse, y à l'ordonnée, l à la largeur et L à la hauteur de la grille, cet index est calculé de la manière suivante :

```
1 // Linéarisation en ligne
2 i = y * l + x
3
4 // Linéarisation en colonne
5 i = x * L + y
```

Le calcul de cet index est pris en charge de manière transparente par les objets grilles offerts par MCMAS du côté CPU, et par un ensemble de directives de macro-processeur du côté de l'exécution OpenCL.

La linéarisation des données est très efficace du point de vue des accès mémoire GPU car elle permet de s'assurer que des éléments adjacents seront effectivement stockés à des emplacements mémoires contigus ou à intervalles réguliers, de manière à pouvoir regrouper et profiter de la largeur des lectures mémoires sur cette architecture. Cette proximité des données permet d'optimiser l'exploitation des caches L1 et L2 intégrés aux matériels GPU récents, en assurant le stockage de ces données voisines dans la même ligne de cache.

Cette linéarisation montre cependant ses limites dans le cas de grilles de faible densité, où un grand nombre de cellules ne sont pas utilisées :

Consommation mémoire. La linéarisation de grilles de grande dimension impose au périphérique de disposer d'assez de mémoire contigüe pour stocker tous les éléments de la grille, même inutilisés, là ou des implémentations Java peuvent être basées sur des structures creuses comme des collections.

Nombre d'accès. Si ce type de grille est très performant en accès, du fait de la simple nécessité de calculer un index supplémentaire par rapport à un tableau statique et de la proximité des données en mémoire, des opérations comme le calcul du nombre de cellules utilisées dans la grille ou tout traitement sur les cellules imposent un parcours de l'ensemble de la grille.

Cette utilisation inefficace de la mémoire pour des structures de faible densité est rendue obligatoire par l'impossibilité d'allouer de la mémoire depuis un programme OpenCL, ce qui impose un dimensionnement de la structure très défensif, à même de gérer le pire des scénarios.

Une solution, dans ce cas, est d'employer d'autres représentations pour les grilles, où seules les cases effectivement utilisées seront stockées de manière contigüe, plutôt que l'ensemble de la grille. Un grand nombre de formats [BG09] ont déjà été proposés pour la littérature pour ce type de matrices à faible densité, typiquement optimisés soit pour la création (DOK [3]), soit pour le parcours et la modification de matrices (LIL [4], COO [5], CSR [6]). Ces formats sont actuellement en cours d'implémentation dans MCMAS.

7.3.3 Structures spécifiques

A côté des types courants en Java, MCMAS permet l'accès à deux types de données spécifiques au GPU, les textures et les types vectoriels. Dans les sections suivantes, nous présentons leur principe et leur fonctionnement.

3. Dictionnaire de clés
4. Liste de listes
5. Liste de coordonnées
6. Lignes creuses compressées

Textures

Ces structures, à l'origine graphique, sont représentées en OpenCL par des objets image associés à des dimensions et un format de stockage des pixels. Ce format de stockage indique le nombre de canaux stockés pour chaque pixel (rouge, vert, bleu, alpha...) ainsi que le type de donnée utilisé pour la représentation de chaque canal (entier 8bits, 16bits ou flottant...)

Une même texture peut être employée dans plusieurs kernels OpenCL, mais ne peut être utilisée qu'en lecture ou en écriture par un même kernel. Cette limitation empêche son utilisation pour des données agents qui seraient accessibles en entrée/sortie, mais permet son utilisation dans le cadre de mises à jour non destructives d'un paramètre d'entrée en lecture seule, où les modifications sont stockées dans un buffer résultat en écriture, comme c'est par exemple le cas dans le jeu de la vie.

Une autre restriction de ces textures tient au nombre limité de formats supportés, pour le stockage des données, chaque canal étant prévu pour le stockage d'une composante entière ou flottante. Ces limitations de format empêchent le stockage de données en double précision, de structures ou de plus de trois informations par pixel, contrairement aux types vectoriels ou au tableaux de structures permis par OpenCL.

L'utilisation de textures présente cependant plusieurs avantages :

— L'accès à ces objets est optimisé par un cache spécialisé sur toutes les architectures GPU supportant OpenCL, comme nous l'avons vu dans la présentation de l'architecture mémoire GPU. Cet avantage est particulièrement employé dans le cas de modèles comme celui de proie-prédateur ou des automates cellulaires, où un nombre limité de propriétés doit être stocké par chaque case mais un accès rapide est indispensable.
— Contrairement aux tableaux, il est possible de déclarer des images à deux ou trois dimensions. Cet aspect multi-dimensionnel facilite l'accès à un élément particulier de la texture sans utilisation d'opérations supplémentaires évoquées dans le cas des vecteurs, et permet au concepteur d'associer directement la dimension des textures et le découpage de l'exécution des données, de manière à simplifier son programme et exploiter au mieux la localité des données dans chaque unité de traitement.
— Un dernier intérêt des textures, enfin, est la possibilité de les afficher directement de manière graphique à l'aide des primitives d'intégration OpenGL offertes par OpenCL. Cette fonctionnalité est particulièrement intéressante dans le cadre de systèmes multi-agents, où une forme d'affichage est souvent attendue pour faciliter le suivi de la simulation. L'absence de transformation particulière est un avantage important dans le cas de modèles dotés de grands nombres d'individus, où tout traitement représenterait une perte en temps d'exécution pouvant être consacré à la simulation elle-même.

Types vectoriels

OpenCL complète l'ensemble de types primitifs hérités de C par des types vectoriels permettant de stocker dans une seule variable 2, 3, 4, 8 ou 16 données primitives de même type. Ces types vectoriels, également présents sur d'autres modèles de programmation GPGPU comme CUDA, sont nativement supportés par de nombreuses opérations arithmétiques fournies par le modèle de programmation. Ce support permet, par exemple, d'effectuer un calcul de distances euclidiennes ou une normalisation de vecteur sur GPU en stockant chaque élément sous la forme d'un seul paramètre.

L'accès aux membres de ces types vectoriels est possible en OpenCL à l'aide des champs x, y, z et w pour les quatre premiers éléments ou sous forme d'index numérique, tel qu'illustré par le

Listing 7.1.

Listing 7.1 – Initialisation de données vectorielles OpenCL

```
1  float4 pos = (float4) (0, 0, 0, 0);
2  pos.x      = 1.0;
3  pos.s3     = 4.0;
```

Le standard OpenCL impose le support de ces alternatives vectorielles pour les types char, short, int, float et long et leurs variantes non signées. Chaque implémentation peut également fournir un support vectoriel pour les nombres réels de type double, en déclarant une extension, de manière analogue au mécanisme proposé par OpenCL.

Au-delà de la possibilité de stocker un ensemble de coordonnées dans une même variable, un autre intérêt important de ces types vectoriels est la facilité avec laquelle ils peuvent être composés ou décomposés en OpenCL, en combinant les noms ou les index de champs de chaque côté de l'opérateur d'affectation. De telles facilités rendent le changement de nombre de dimensions d'une coordonnée, requis pour certaines opération agent, trivial en OpenCL, tel qu'illustré par le Listing 7.2.

Listing 7.2 – Transtypage de données vectorielles OpenCL

```
1  float2 pos1 = (float2) (1, 2);
2  float4 pos2 = (float4) (1, 3, 5, 0);
3  float2 pos4 = (float4) (0, 0, 0, 0);
4  pos4 = (float4) (pos1.xy, pos2.z}
```

L'utilisation de ces types dans MCMAS est rendue possible par le biais de la bibliothèque jocl-structs, proposée par les développeurs de JOCL, offrant l'accès à ces types sous forme d'objets Java.

7.3.4 Exécution synchrone ou asynchrone

La plate-forme OpenCL est basée sur une soumission de l'ensemble des opérations à réaliser sur le périphérique en file d'attente. Ce mode de fonctionnement permet au programme principal de poursuivre son exécution, et de consulter les résultats de sa soumission de manière ultérieure, grâce à un "ticket" retourné lors de la soumission.

Ce mécanisme est également intégré dans l'interface de bas-niveau de MCMAS, sous forme d'objets événements Java implémentant l'interface standard `Future`, en plus des opérations OpenCL spécialisées. Ces événements peuvent être alors manuellement utilisés pour choisir le moment où synchroniser l'exécution des traitements, ou directement comme paramètres pour la soumission de nouveaux traitements, de manière à créer un enchaînement d'actions OpenCL. Ce mécanisme de dépendances peut par exemple être utilisé pour effectuer la copie de données avant et après le lancement d'un programme sans synchronisation intermédiaire, comme illustré par le Listing 7.3.

Ce mécanisme, indispensable pour tirer parti du recouvrement permis par OpenCL entre opérations de copie des données et d'exécution, est exploité par de nombreux plugins offerts par MCMAS. Il peut également être mis à disposition au niveau de l'interface par le biais de variantes asynchrones des opérations proposées, de manière analogue à l'interface proposée par MPI pour la communication en mémoire distribuée.

Listing 7.3 – Utilisation asynchrone de MCM

```
1  // Préparation des arguments et lancement d'un programme OpenCL
2  kernel.setArguments(vector, radius, xPositionsMem, yPositionsMem,
       xResultsMem, yResultsMem);
3
4  MCMEvent finished = q.enqueue1DKernel(kernel, vector.length);
5
6  // Mise en file d'attente de la récupération des données résultats,
7  // une fois l'exécution du programme terminée (événement finished)
8  MCMEvent r1 = q.enqueueReadBuffer(xResultsMem, Pointer.to(xResults), 0,
       xResultsMem.getSize(), finished);
9  MCMEvent r2 = q.enqueueReadBuffer(yResultsMem, Pointer.to(yResults), 0,
       yResultsMem.getSize(), finished);
10
11 // Autres traitements Java
12
13 // Attente bloquante de la fin du graphe d'exécution
14 MCMEvent.waitFor(r1, r2);
```

7.4 Utilisation de l'interface de haut niveau

Après nous être intéressés à son architecture, nous présentons dans cette section l'utilisation de l'interface de haut niveau de notre bibliothèque, sans connaissance particulière de la programmation GPU. Cette utilisation est rendue possible à l'aide de deux structures principales : le contexte d'exécution et un ou plusieurs plugins offrant l'accès aux fonctions de haut niveau de MCMAS.

7.4.1 Initialisation de MCMAS

L'interface de haut niveau de MCMAS est basée sur l'utilisation d'un contexte d'exécution de type `MCMASContext`. Cet objet contient l'ensemble des structures nécessaires à la soumission d'un traitement OpenCL, et admet différents constructeurs permettant au développeur d'indiquer le type et les paramètres d'exécution souhaités, tel que représenté dans le Listing 7.4.

Une fois instancié, un contexte MCMAS peut être utilisé pour créer et appeler des plugins MCMAS. Ces deux modes d'utilisation peuvent être librement combinés pour un même contexte.

Listing 7.4 – Exemples de création de différents types de contexte MCMAS

```
1  // Aucun argument, sélection automatique de la plate-forme d'exécution par
       MCMAS:
2  // GPU en priorité, puis CPU.
3  MCMASContext context = new MCMASContext()
4
5  // Définition explicite de la prioriété des plate-formes à utiliser:
6  MCMASContext context = new MCMASContext(ContextType.GPU, ContextType.CPU)
7
8  // Création d'un contexte GPU
9  MCMASContext context = new MCMASContext(ContextType.GPU)
10
11 // Création d'un contexte CPU
12 MCMASContext context = new MCMASContext(ContextType.CPU)
13
14 // Création d'un contexte supportant le profiling
15 MCMASContext context = new MCMASContext(MCMAS.PROFILING);
```

7.4.2 Exemples d'appel de fonctions de haut niveau

Une fois un contexte MCMAS obtenu, il est possible de l'utiliser pour appeler de nombreuses fonctions de haut niveau regroupées sous forme de plugins spécialisés inclus dans la bibliothèque.

Chacun de ces plugins propose un ensemble de fonctions classées par thématique d'utilisation (calcul de distance, diffusion...) Ces fonctions admettent un certain nombre de paramètres d'entrée et de sortie correspondants au traitement à effectuer. L'accent est mis, au niveau de ces paramètres d'entrée, sur l'utilisation de tableaux statiques et d'autres structures Java standard, de manière à permettre la plus large utilisation possible de ces fonctions.

Des outils de conversions fournis avec MCMAS facilitent le passage vers ces types depuis les autres types de données référencés dans notre présentation précédente, et en particulier depuis des objets, des buffers de données ou des objets. Nous illustrons dans la suite de cette section l'utilisation de certains des plugins fournis par notre bibliothèque.

Calcul de distances

Une première fonctionnalité rencontrée dans de nombreux systèmes multi-agents, dont le modèle proie-prédateur, est le calcul de distances euclidiennes entre individus. Selon le type de modèle employé, ce calcul peut être effectué en une, deux ou trois dimensions, sur des coordonnées entières (grille) ou réelles.

L'obtention de ces distances implique généralement, sur CPU, le calcul séquentiel de cette distance pour chaque couple d'agents présents dans le modèle, ou le calcul de ces distances à la volée pour les seuls points utilisés.

Le calcul de ces distances pouvant être aisément effectué en parallèle, il est possible de réaliser tous ces calculs en simultané avec MCMAS, en indiquant en entrée les coordonnées à considérer et en récupérant l'ensemble des distances comme résultat.

Les coordonnées des points d'entrée peuvent être spécifiées sous forme d'un tableau par dimension à considérer (array_x, array_y, array_z), sous forme de tableau de structures coordonnées, ou encore sous forme de tableau de type vectoriel OpenCL tel que float2 ou float3, comme évoqué précédemment.

Pour deux ensembles de M et N coordonnées d'entrée, le résultat de ce module est une grille comprenant les distances euclidiennes entre chaque point du premier ensemble et chaque point du second ensemble, de dimension $M \times N$, et de type compatible avec le stockage des coordonnées d'entrée (entier, réel ou flottant).

Diffusion

Une autre opération souvent rencontrée dans les modèles multi-agents est la diffusion d'une quantité au sein d'un vecteur ou d'une grille. Ce mécanisme est souvent utilisé pour des mises à jour de l'environnement, de manière à simuler la diffusion de phéromones par exemple dans le cas du modèle des fourmis, ou de populations dans des modèles de reproduction d'individus dans un nouvel habitat.

Cette diffusion est caractérisée par plusieurs paramètres :

— Le nombre de dimensions considérées : le nombre de cellules voisines vers lesquelles diffuser est directement lié à la configuration du modèle. Dans un modèle à une dimension, avec des cellules en grille, seules deux voisines devront être considérées, contre six dans le cas de deux dimensions et vingt-six dans un cas à trois dimensions.
— La possibilité ou non de diffuser en diagonale : le calcul précédent suppose que l'ensemble des cellules voisines sont considérées. Si seules celles partageant une arête avec la cellule courante sont prises en compte (diffusion "en croix"), alors le nombre de voisins pour chaque cellule est modifié, ce qui peut changer drastiquement le comportement de la simulation.

Un autre choix important pour effectuer cette diffusion est le comportement devant être retenu aux limites de la grille, où certains voisins sont manquants :

— Une première approche est de supposer toute diffusion en dehors de la grille comme perdue. Ces frontières se comportent alors comme un puits sans fond, et peuvent amener une perte de matière dans le modèle (système non clos).
— Une autre approche est d'interdire la diffusion en dehors de la grille (système fermé). Cette approche est préférable dans le cadre d'un véritable environnement fermé, mais décon-

seillée dans le cas d'un modèle ne représentant qu'une fraction d'un environnement plus vaste, comme une parcelle de terrain d'un territoire, car elle introduit des frontières artificielles pouvant influer sur le cycle d'évolution des agents situés à la périphérique du modèle, sans que cette influence n'ait aucune base pratique.

— Une dernière approche, enfin, est de considérer l'environnement agent comme bouclant verticalement et horizontalement. Cette approche, souvent retenue pour des raisons de commodité, revient à représenter l'espace de simulation comme la projection d'un tore.

Le module diffusion de MCMAS fournit les opérations correspondant à ces différents cas. Cette implémentation est également basée sur l'utilisation de programmes OpenCL adaptés à chaque type de données d'entrée (scalaire, structure ou vectoriel, entière ou flottante).

Le lancement d'une diffusion peut être effectué en quelques lignes, à l'aide des paramètres acceptés par la fonction, comme l'illustre le Listing 7.5.

Listing 7.5 – Exemple d'utilisation du plugin diffusion sur une grille de flottants représentée par un tableaux à deux dimensions

```
1  // Déclaration des structures
2  float [][] grille, grilleResultat;
3
4  // Instantiation d'un context MCMAS et du plugin de diffusion
5  MCMASContext  context  = new MCMASContext();
6  DiffusePlugin diffuser = DiffusePlugin.newInstance(context);
7
8  // Préparation des structures de données MCMAS
9  MCMASGrid input = MCMAS.createGridFrom(grille);
10 MCMASGrid output = MCMAS.createGridFrom(grilleResultat);
11
12 // Lancement d'une diffusion et récupération du résultat.
13 // DiffusePlugin.DIMENSION_4 requiert une diffusion verticale et horizontale
       :
14 // DiffusePlugin.DIMENSION_8 permettrait de demander la prise en compte des
       diagonales.
15 diffuser.diffuse(input, output, DiffusePlugin.DIMENSION_4);
16
17 // Recopie de la grille obtenue dans la structure java originale
18 output.write(grilleResultat);
```

De nombreux modèles agents imposent également des limites minimales et maximales particulières à la valeur pouvant être stockée dans chaque cellule. Ces limitations sont prises en compte par le plugin diffusion fourni par MCMAS via la disponibilité de variantes bornées des opérations précédentes, permettant d'indiquer la valeur minimale et la valeur maximale permises pour chaque cellule.

Cette opération de normalisation peut également être effectuée à l'aide d'un plugin spécialisé fourni par MCMAS, de manière indépendante.

Réduction

De très nombreuses simulations agents requièrent la production régulière d'indicateurs associés au modèle, comme la quantité globale d'énergie présente dans le système ou la population dans le modèle proie-prédateur. La dispersion de ces quantités entre les différents éléments du modèle implique, à chaque itération, de pouvoir synthétiser ces quantités dans une variable globale, à même d'être affichée ou utilisée pour interrompre ou non la simulation.

Des modèles de programmation CPU tels que OpenMP ou MPI facilitent cette opération à l'aide de primitives de réduction, permettant au concepteur d'indiquer les données devant être réduites et l'opération à utiliser (somme, minimum, maximum...)

MCMAS reprend cette logique dans un plugin spécialisé permettant d'effectuer cette réduction en parallèle à l'aide des informations suivantes :

— Le champ de la réduction : un vecteur ou une grille contenant la propriété du modèle agents à prendre en compte.
— L'opération de réduction : minimum, maximum, moyenne, addition, multiplication...

Le résultat d'une réduction totale est un type scalaire, et un vecteur pour une réduction partielle. Comme dans le cadre d'une diffusion, le type résultat est compatible avec le type d'entrée : flottant, par exemple, pour une réduction sur des données flottantes.

Fonctions affines

Un autre type de traitement parallélisable est, comme nous l'avons vu dans le cas du modèle proie-prédateur, l'application d'une même fonction affine $a \times x + b$ à chaque cellule d'un vecteur ou d'une grille d'entrée.

Cette opération est directement supportée par MCMAS sur ces deux structures de données, un vecteur pouvant être considéré comme une grille à une dimension. Le Listing 7.6 illustre l'appel du plugin effectué pour la croissance de l'herbe dans le cadre du modèle proie-prédateur.

Listing 7.6 – Application d'une fonction affine sur les éléments d'une grille

```
1  // Déclaration des structures
2  float [][] grille, grilleResultat;
3
4  // Instantiation d'un context MCMAS et du plugin de diffusion
5  MCMASContext  context     = new MCMASContext();
6  DiffusePlugin transformer = AXBPlugin.newInstance(context);
7
8  // Préparation des structures de données MCMAS
9  MCMASGrid input = MCMAS.createGridFrom(grille);
10 MCMASGrid output = MCMAS.createGridFrom(grilleResultat);
11
12 // Lancement de la transformation et récupération du résultat.
13 // a vaut 1.0, car aucun facteur variable de croissance n'est appliqué.
14 // b vaut GRASS_GROWTH, le taux de croissance fixe défini dans la simulation
       .
15 transformer.transform(input, output, 1.0, GRASS_GROWTH);
16
17 // Recopie de la grille obtenue dans la structure java originale
18 output.write(grilleResultat);
```

7.4.3 Utilisation depuis des framework multi-agents existants

La bibliothèque MCMAS est utilisable directement en Java, mais doit également pouvoir être accessible à des plates-formes multi-agents ne permettant pas l'accès direct à ce langage.

Dans ce cas, il est possible de fournir le service assurant le rôle d'interlocuteur et de traducteur entre le formalisme utilisé par la plate-forme et MCMAS. Ce service peut être un agent du système

spécialisé, tel qu'illustré sur la Figure 7.4, ou un système indépendant, accessible par le biais de messages.

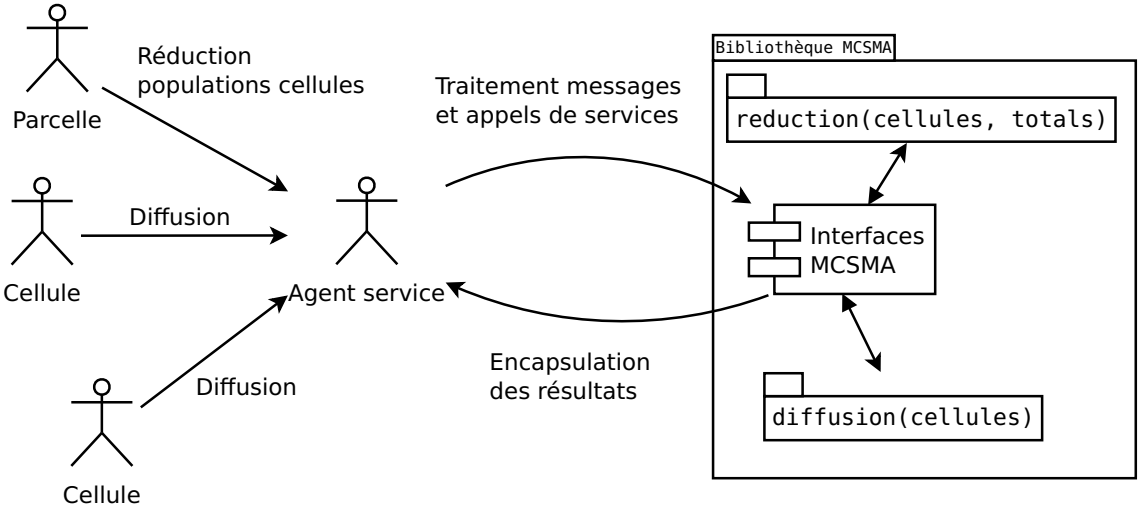

FIGURE 7.4 – Exemple d'utilisation de MCMAS depuis un modèle multi-agents

Chacun de ces messages doit respecter un format bien défini, indiquant le traitement souhaité ainsi que des contraintes d'exécution du calcul (sur GPU, sur CPU...). L'identité de l'expéditeur est mémorisée avant le traitement de chacun de ces messages, de manière à pouvoir lui envoyer les résultats de l'exécution dans un autre message une fois la requête traitée.

Par exemple, dans le cas de Madkit où les agents communiquent à l'aide d'un gestionnaire de messages, ce processus se traduit pas la mise en place d'un protocole d'échange. Dans le cas de GAMA, cette intégration peut être réalisée sous la forme d'un plugin ajoutant des fonctions MCMAS au langage de description utilisé pour les agents.

Cette couche d'adaptation entre MCMAS et la plate-forme multi-agents ciblée permet l'utilisation transparente de la bibliothèque, en conservant l'infrastructure du modèle existant.

L'utilisation d'un agent service permettant la réalisation de traitements MCMAS est une autre solution utilisant les mécanismes de communication du modèle existant. Il est également possible d'utiliser MCMAS directement au niveau de chaque agent, soit à l'aide de Java directement, soit à l'aide d'une couche d'adaptation dans le cas de plates-formes telles que GAMA ou NetLogo.

7.5 Développement de nouveaux plugins

L'ensemble de plugins fourni avec MCMAS ne pourra jamais recouvrir l'ensemble des fonctionnalités pouvant être rencontrées dans la littérature agents. Pour pallier à cette limitation, MCMAS permet l'ajout aisé de nouveaux plugins, de manière à permettre au concepteur de simulations d'étoffer les fonctionnalités offertes par la bibliothèque.

Pour valider le type d'opération à implémenter et l'implémentation à utiliser, plusieurs étapes de réflexion et de conception sont recommandées. Dans cette section, nous décomposons les différentes étapes de cette démarche.

7.5.1 Conception du modèle et parallélisation

La première étape du développement d'un nouveau plugin est la définition du problème rencontré et de sa portée exacte. Un système multi-agents peut représenter un ensemble d'individus et de comportements complexes, se prêtant ou non à une adaptation sur GPU. Il est alors dans ce cas nécessaire de déterminer les parties du modèle pouvant tirer avantage d'une exécution sur GPU, ainsi que la manière de paralléliser ces traitements.

Un élément important pour évaluer les gains pouvant être attendus de la parallélisation avec MCMAS est d'évaluer l'impact en termes de performance des parties du modèle pouvant être accélérées par rapport au temps total d'exécution de la simulation, en application de la loi d'Amdahl. Une conséquence directe de cette loi est que tout gain obtenu par l'utilisation d'une plate-forme est rapidement limité si la portion parallélisée ne représente pas une fraction significative du temps d'exécution de la simulation multi-agents. Le découpage du modèle en sous-parties peut même, dans ce cas, entraîner une perte de performance liée aux échanges de données entre les différentes portions de l'algorithme.

Il est donc nécessaire, en développant un nouveau plugin MCMAS, d'avoir à l'esprit ces limitations et une estimation du gain de performance pouvant être attendu, ainsi que le nombre de recours à cette simulation, de manière à pouvoir quantifier le temps pouvant être consacré à une adaptation GPU du modèle par rapport au temps supplémentaire qui aurait été utilisé avec l'ancienne implémentation.

L'efficacité d'exécution d'un programme sur GPU est directement liée aux opérations et aux structures utilisées, mais n'est pas linéaire en termes d'efforts. La recherche de performance est un processus sans fin tendant toujours vers une limite où les adaptations à effectuer ne permettent plus de gains en performance très importants. Nos cas d'applications illustrent particulièrement qu'un portage à l'identique d'un algorithme ne permet que rarement l'obtention de bonnes performances, pour des raisons d'occupation processeur, de synchronisation ou de mémoire.

Un autre aspect critique de l'efficacité de l'utilisation du GPU, déjà évoqué dans notre présentation du découpage d'une simulation en OpenCL, est le niveau de granularité retenu par la simulation, c'est à dire la taille en temps passé sur chaque plate-forme. Une parallélisation "à gros grains" permet de réduire le nombre d'échanges entre les deux plate-formes, et donc les coûts de synchronisation ou de communication associés ces échanges. Au contraire, une parallélisation dite "à petits grains" implique le lancement de nombreux traitements simultanés sur GPU, pour minimiser l'impact des échanges plus réguliers devant avoir lieu.

7.5.2 Définition de l'interface et des fonctionnalités

Une fois la problématique et la solution précisément connues, il est nécessaire de définir la manière dont la parallélisation pourra être intégrée dans le modèle existant. Cette intégration porte à la fois sur les scénarios d'exécution, et les données manipulées :

— Dans quel ordre les opérations seront-elles appelées ? Est-il possible de se servir de cette information pour rendre les traitements asynchrones ou de changer leur ordre d'exécution ?
— Quelles structures de données sont-elles utilisées ? Doivent-elles être transformées pour une exécution sur GPU ? Peuvent-elles être partagées entre un maximum de traitements, pour éviter des copies ?

Ces deux problématiques, complémentaires, ont une importance vitale pour la définition de l'interface du nouveau plugin, et plus particulièrement sur le nombre et le prototype des opérations

fournies.

7.5.3 Implémentation de la solution retenue

Une fois l'interface du plugin définie, il est possible de réaliser l'implémentation de la solution elle-même. Dans MCMAS, cette implémentation met en jeu deux langages différents :

— **Java** pour l'ensemble des traitements exécutés sur CPU. Dans le cadre d'un plugin, ces traitements comprennent au minimum la gestion de la copie et des lancements des traitements sur la plate-forme OpenCL, ainsi que le suivi de l'exécution et la récupération des résultats. Cette partie peut également inclure des pré-traitements sur les données, soit dans le cadre de conversions de représentations, soit parce que ces traitements sont peu adaptés à une exécution sur GPU. Le reste du modèle multi-agents n'est pas nécessairement réalisé en Java, si une interface d'adaptation MCMAS est utilisée.

— **OpenCL** est utilisé pour tous les traitements ayant lieu sur le périphérique. Comme C et C++, ce langage permet l'inclusion de portions de programmes existantes au moment de la compilation. Cette fonctionnalité est utilisée par MCMAS pour fournir de nombreux raccourcis pour la manipulation des structures de données incluses dans MCMAS, et permet également au plugin de partager des fonctionnalités entre ses traitements natifs. Il est importante de noter que tout lancement de programme OpenCL implique de définir explicitement le point d'entrée (kernel) utilisé : il est donc toujours possible d'implémenter tous les traitements du plugin dans un unique fichier source OpenCL.

Si OpenCL permet le stockage de binaires correspondant à des programmes compilés, les binaires obtenus sont spécifiques à l'implémentation et au matériel courant. Ce mécanisme permet d'éviter de multiples compilations au-delà de la première exécution, et ainsi de réduire le temps de chargement du programme, mais ne dispense pas le développeur du plugin de devoir fournir le code source des portions OpenCL de son programme, en cas de distribution sur de nombreuses plates-formes.

Le plugin obtenu peut être directement inclus dans l'arborescence de fichiers sources du modèle multi-agents y ayant recours, ou empaqueté sous forme d'archive JAR indépendante, de manière à faciliter son partage et sa distribution.

La distribution d'une documentation et de tests associés au nouveau plugin est fortement recommandée. La rédaction de ces tests est facilitée par la disponibilité de nombreuses fonctions de création de contextes et de récupération d'informations, en termes de plate-forme d'exécution et de temps au niveau de MCMAS. La création d'un contexte simple CPU, GPU, avec ou sans activation des fonctionnalités de profiling, est ainsi possible en une ligne.

7.5.4 Validation

Une fois le développement du plugin terminé, une validation de celui-ci est requise. Cette validation regroupe deux aspects :

— **La validation de l'exactitude des résultats.** Cette première étape, critique, consiste à s'assurer que les résultats obtenus sont similaires à ceux prévus par le modèle théorique ou observés dans l'implémentation originale. Elle peut être effectuée de manière formelle, dans le cas où l'équivalence sémantique avant et après adaptation peut être établie. Le recours au parallélisme tend cependant à introduire de nombreuses inconnues dans ce type de démonstration, qu'il est nécessaire de quantifier. Ces difficultés favorisent une validation

expérimentale, bouclant d'une certaine manière le cycle de pensée agent : dans ce cas, les résultats obtenus par les deux implémentations sont comparées dans le cadre d'un protocole expérimental prenant en compte les cas limites, les données et les conditions d'utilisation devant être envisagées.

— **La validation des performances obtenues.** Cette seconde étape permet de quantifier les performances effectivement observées par rapport à l'implémentation originale et aux éventuelles attentes basées sur les caractéristiques de la solution retenue et de la plateforme, comme le nombre de cœurs. Elle ne peut être qu'expérimentale, à l'aide de mesures de temps d'exécution des simulations. La comparaison de ces temps, et l'allure des courbes de performance obtenue, permet de caractériser le type de comportement obtenu en termes de performance et de ressources consommées. Ces résultats peuvent, ou non, conforter les attentes établies au moment de la conception du plugin. Ils permettent également de mettre en avant des parties coûteuses ou mal adaptées de l'algorithme dont l'impact aurait pu être négligé ou sous-estimé, pouvant relancer une itération supplémentaire de parallélisation.

7.6 Synthèse

Dans ce chapitre, nous avons présenté MCMAS, notre bibliothèque dont l'objectif est de prendre en charge deux scénarios d'utilisation, l'utilisation du GPU au moyen de fonctions génériques sans connaissance de l'architecture ou le développement de nouveaux traitements GPU.

Ce double usage est à l'origine du choix d'une architecture modulaire pour MCMAS, basée d'une part sur une interface de bas niveau, MCM, pour l'accès au modèle de programmation OpenCL et d'autre part sur un ensemble de plugins regroupant des traitements agents préimplémentés. Ces plugins reposent également sur l'interface MCM, pour favoriser la combinaison des deux types d'utilisation ou l'expérimentation de nouveaux traitements.

Cette architecture se retrouve également dans l'implémentation de la bibliothèque. Elle est basée sur trois parties fondamentales :

— Un contexte d'exécution encapsulant tout l'environnement d'exécution GPU.
— Un ensemble de structures de données GPU et d'outils de conversion de ces structures vers et depuis des structures de données Java.
— Une interface de programmation MCM reprenant les principaux concepts OpenCL mais en facilitant la gestion dans un environnement objet.

Nous avons ensuite abordé l'utilisation de MCMAS selon chacun de ces deux scénarios d'utilisation :

— Sans connaissance GPU. Dans ce cas, l'utilisation de l'interface de haut niveau MCMAS est possible simplement au moyen de la création d'un contexte d'exécution, suivie de l'instantiation d'un ou plusieurs plugins avec ce contexte. Les fonctions fournies par ces plugins peuvent alors être utilisées pour lancer des opérations comme des calculs de déplacements, de distances, ou de transformations matricielles.
— Avec des connaissances GPU, pour le développement de nouveaux traitements avec MCM. Ce type d'utilisation permet alors, de manière optionnelle, l'encapsulation de ces traitements dans un plugin MCMAS de manière à favoriser leur redistribution et leur réutilisation dans d'autres simulations multi-agents.

8

VALIDATION SUR DES MODÈLES EXISTANTS

Dans le chapitre précédent, nous avons présenté notre bibliothèque MCMAS, ainsi que ses deux scénarios d'utilisation, par le biais d'une interface de bas niveau MCM, ou au moyen de fonctions génériques fournies par un ensemble de plugins. Pour valider son utilisation sur des cas concrets, nous illustrons dans ce chapitre son utilisation sur trois exemples de systèmes multi-agents concrets : le modèle proie-prédateur, qui nous a servi de fil rouge à la présentation de MCMAS, un modèle de simulation de l'évolution microscopique des sols, MIOR, et enfin un modèle de diffusion de populations, le modèle Collemboles.

Dans une première section, nous commençons par présenter les deux modèles n'ayant pas encore été évoqué, ainsi que la manière dont nous avons choisi de les paralléliser avec MCMAS. Dans une seconde section, nous présentons ensuite les performances obtenues sur ces trois adaptations. Pour cela, nous commençons par décrire les plates-formes d'exécution et le protocole utilisés, avant d'analyser les performances obtenues. Nous synthétisons enfin, dans la troisième section, les observations et les recommandations pour une parallélisation sur GPU qu'il nous a été possible de tirer de ces trois adaptations.

8.1 Parallélisation de modèles

L'étude de ces trois modèles a été l'occasion d'appliquer plusieurs approches de parallélisation parmi celles évoquées dans le Chapitre 6. Pour le modèle proie-prédateur, nous avons choisi une implémentation reposant sur l'utilisation de plugins génériques fournis par MCMAS pour paralléliser certains traitements coûteux de la simulation. Pour le modèle MIOR, nous avons choisi de réaliser un nouveau plugin MCMAS pouvant être utilisé pour lancer de nombreuses simulations microscopiques. Enfin, pour le modèle Collemboles, nous avons utilisé l'interface de bas niveau MCM pour implémenter chaque étape de la simulation sur GPU.

8.1.1 Proie-prédateur

Dans la continuité de notre réflexion sur ce modèle, dans le chapitre 5, nous avons choisi d'implémenter la mise à jour des ressources végétales de l'environnement et le déplacement des individus proies et prédateurs à l'aide des plugins génériques de transformation de données et de recherche dans une grille proposés par l'interface de haut niveau de notre bibliothèque.

Cette implémentation est basée sur l'algorithme 1. Deux plugins fournis par MCMAS sont mis à contribution :

— Le plugin de fonction affine est utilisé pour l'ensemble de la mise à jour de la grille re-

présentant les ressources végétales à chaque itération. Dans le cadre de ce traitement, la structure de données grille fournie par MCMAS est automatiquement considérée comme un vecteur.

— Un plugin de recherche de maximums fourni avec MCMAS est utilisé pour le calcul des déplacements des individus vers la cible la plus énergétique. Ce plugin permet la recherche de maximums locaux autour d'une ou plusieurs positions dans une grille. Pour ce faire, trois informations sont indiquées, la grille ainsi que les positions et le rayon de recherche. Ce plugin retourne en résultat le maximum local trouvé pour chaque position, correspondant à la case vers laquelle doit se déplacer l'individu dans le modèle proie-prédateur.

Pour permettre la recherche des nouvelles positions en parallèle à l'échelle de chaque population, l'évolution de chaque type d'individu proie ou prédateur du modèle est effectuée étape par étape, tel qu'illustré par la Figure 8.1 :

— **Préparation des positions**. Toutes les positions en deux dimensions des individus de la population sont synthétisées, en vue de l'appel à MCMAS. Cette étape permet également un comptage du nombre d'agents présents dans le modèle. Cette étape n'est pas parallélisée car elle implique la manipulation de structures de données dynamiques dont la taille n'est pas connue à priori, la liste des positions.

— **Recherche de maximums locaux**. Cette opération, implémentée par un plugin MCMAS, permet le calcul de la nouvelle position des individus en un seul lancement. Elle admet trois arguments d'entrée : un espace de recherche, une liste de positions et un rayon de recherche. Ce rayon de recherche indique la distance maximale autour de chaque position où chercher un couple de coordonnées solution dans l'espace de recherche. La fonction retourne en résultat une liste de coordonnées correspondant aux maximums locaux trouvés pour chaque position. Cette fonction implique un parcours coûteux, en particulier si le rayon de recherche est important. Ce parcours a l'avantage de pouvoir être réalisé simultanément pour tous les individus du modèle, ce qui justifie sa parallélisation.

— **Déplacements**. Une fois les nouvelles positions obtenues, ces informations sont utilisées pour déplacer chaque individu de manière séquentielle. Cette application séquentielle garantit la cohérence des déplacements, en assurant un fonctionnement du type "premier arrivé, premier servi" : si un conflit de destination existe entre plusieurs individus, le premier l'emporte, et les autres restent immobiles pour cette itération. Cette étape n'est pas parallélisée de manière à pouvoir traiter de manière séquentielle les déplacements sur le CPU, et ainsi gérer les conflits où plusieurs individus souhaitent se déplacer au même emplacement.

— **Consommation**. Les ressources présentes à la position de chaque individu sont consommées : dans le cas d'une proie, la quantité de végétaux est réduite. Dans le cas du prédateur, la proie est tuée. Dans les deux cas, le différentiel en énergie est ajouté à l'individu courant, dans une certaine limite correspondant à la vitesse maximale d'absorption des ressources pour ce type d'individu. Cette étape n'est pas parallélisée car elle ne représente qu'une soustraction et une addition simple pour chaque individu qui peut être effectuée dans la continuité du déplacement.

— **Reproduction**. Si l'énergie de l'individu dépasse un certain seuil, un nouvel individu est créé à son ancienne position. La quantité d'énergie minimale prévue par le modèle est retirée de l'individu parent et assignée à ce nouvel arrivant. Cette opération demeure séquentielle car elle implique une modification de la grille pour ajouter de nouveaux individus susceptible d'aboutir à des conflits.

— **Métabolisme**. L'énergie de chaque individu est décrémentée. Si elle devient négative ou nulle, l'individu est supprimé du modèle, pour indiquer son décès. Cette opération est susceptible d'être parallélisée en traitant en parallèle tous les niveaux d'énergie du modèle,

mais nécessiterait en pratique le traitements de nombreuses cellules vides de la grille, ainsi qu'un aller-retour sur GPU. Dans ces circonstances, cette gestion du métabolisme est réalisée dans la même boucle que les trois traitements précédents.

Ce processus d'évolution des individus est appliqué de manière identique à chacune des populations du modèle, en variant les positions et les distances de recherche. Elle permet de réaliser le calcul du déplacement de manière parallèle, plutôt que sous forme de nombreuses boucles séquentielles sur CPU.

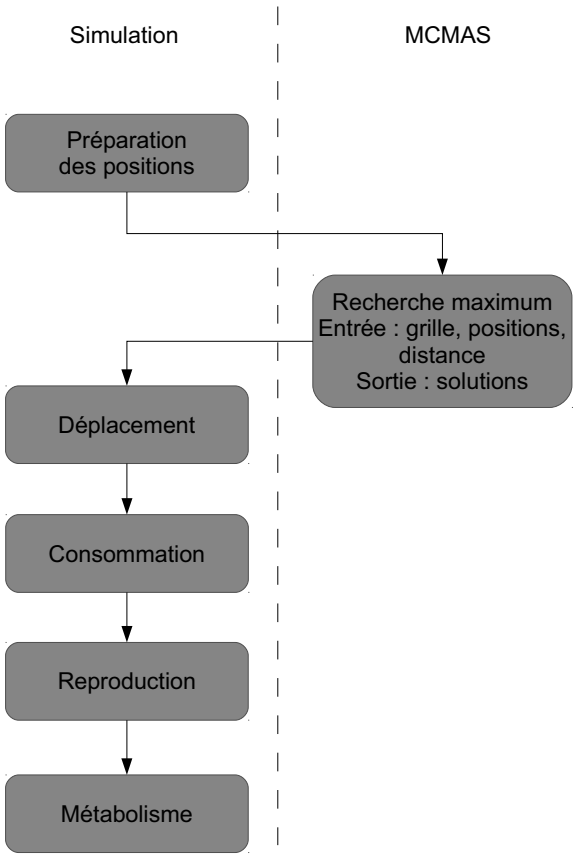

FIGURE 8.1 – Décomposition du cycle de vie des individus avec MCMAS

8.1.2 Une simulation microscopique : MIOR

Le modèle multi-agents MIOR (MIcro-ORganisme) [CCP07] reproduit l'évolution microscopique des sols sous l'effet de l'activité microbienne. Ce modèle permet de simuler l'évolution en terme de biomasse microbienne, de quantité de matière organique et de CO_2 produit d'un cube de sol de de 0.002 mm de côté.

Cette échelle microscopique requiert le lancement d'un grand nombre de simulations MIOR pour traiter des volumes de sols macroscopiques. Dans ce cas, le développement d'un plugin MIOR permettant de lancer des paquets de simulation est intéressant pour permettre la simulation de volumes variables de sol en parallèle, en tirant parti du parallélisme d'exécution offert par l'architecture GPU. Cette multiplication des modèles permet aussi d'augmenter le nombre d'agents en parallèle, de manière à garantir une occupation efficace des coeurs matériels.

Ce lancement peut être effectué de deux manières :

— Soit jusqu'à la stabilisation de l'évolution de l'ensemble des simulations MIOR. De cette manière, le modèle macroscopique dispose de toutes les informations futures de l'état microscopique de cette cellule. Le nombre d'itérations nécessaires à cette stabilisation de toutes les simulations n'est cependant pas forcément aisé à prévoir en fonction des paramètres d'entrée.
— Soit sur un nombre fixe d'itérations. Cette alternative permet d'éviter une attente trop longue pour l'obtention des résultats des simulations MIOR. Il est également possible dans ce cas au modèle Sworm de relancer ces simulations ultérieurement sur GPU, en fonction des besoins du modèle macroscopique. Elle ne garantit pas d'atteindre la stabilisation de la simulation.

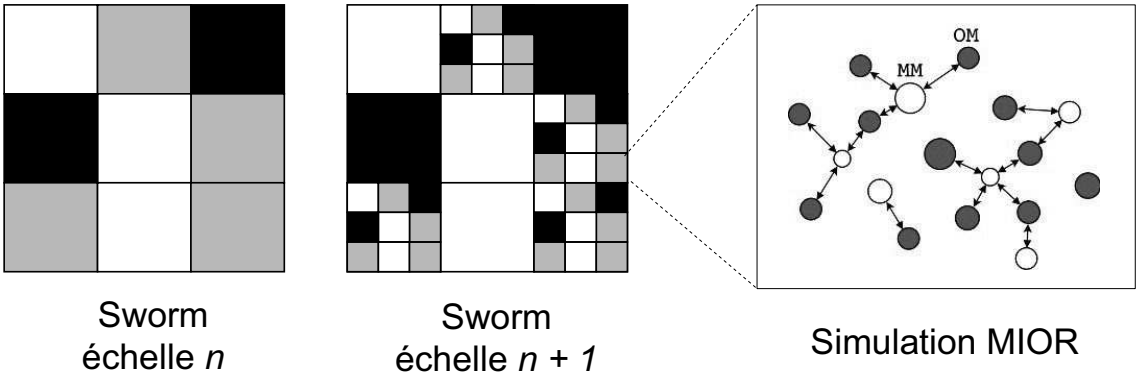

FIGURE 8.2 – Représentation fractale de l'environnement Sworm

Le modèle MIOR repose sur deux espèces d'agents :

— Les Méta-Mior (MM) qui représentent des colonies microbiennes consommatrices de carbone.
— Les dépôts de matière organique (OM) qui caractérisent les dépôts de carbone répartis dans le volume de sol.

Les agents Meta-Mior sont associés à deux comportements distincts :

— *la respiration* : transformation du carbone minéral en dioxyde de carbone CO_2)
— *la croissance* : chaque colonie ayant prélevé suffisamment de carbone dans l'environnement est en mesure de croître en population.

Ces deux comportements sont décrits dans l'algorithme séquentiel 2.

Aucune interaction n'est considérée comme possible entre les colonies microbiennes : les seuls échanges sont réalisés entre dépôts de carbone et colonies, en fonction de leur proximité spatiale (distance en deux ou trois dimensions) par rapport au rayon d'action associé à la colonie microbienne. Ces associations peuvent être représentées sous la forme de lignes liant les agents de chaque type pouvant potentiellement interagir, tel qu'illustré par la Figure 8.3

Algorithme 2 : Algorithme séquentiel d'évolution MIOR

Data : *mmList* Tableau d'agents MM (colonies microbiennes)
Data : *omList* Tableau d'agents OM (dépôts de carbone)
Data : *world* Environnement global de la simulation

```
1  breathNeed ← world.respirationRate × mm.carbon;
2  growthNeed ← world.growthRate × mm.carbon;
3  availableCarbon ← totalAccessibleCarbon(mm);
4  if availableCarbon > breathNeed then
       /* Processus de respiration                                   */
5      mm.active ← true;
6      availableCarbon ← availableCarbon − consumCarbon(mm, breathNeed);
7      world.CO2 ← world.CO2 + breathNeed;
8      if availableCarbon > 0 then
           /* Processus de croissance                               */
9          growthConsum ← max(totalAccessCarbon(mm), growthNeed);
10         consumCarbon(mm, growthConsum);
11         mm.carbon ← mm.carbon + growthConsum;
12     end
13 else
14     mm.active ← false
15 end
```

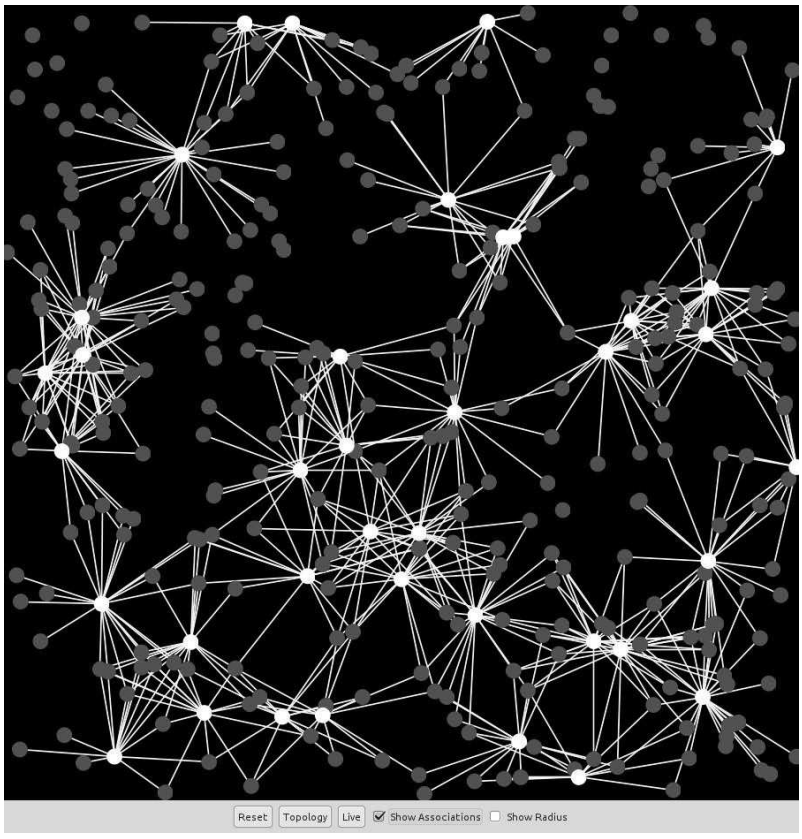

FIGURE 8.3 – Représentation graphique d'une simulation MIOR

Problématiques d'implémentation du plugin

L'ajout d'un nouveau plugin MCMAS permettant de lancer plusieurs simulations MIOR de manière simultanée implique l'utilisation de MCM, puisque l'ensemble de la simulation et non certains traitements génériques doit être parallélisé. Cette approche implique donc l'utilisation d'OpenCL pour implémenter l'algorithme MIOR, ce qui impose la gestion de plusieurs problématiques :

— La parallélisation de l'exécution du modèle. Cette démarche, commune à toute parallélisation de modèle sur GPU, impose en particulier l'identification du grain de parallélisation retenu aux différentes étapes de l'algorithme d'évolution.
— L'adaptation des structures de données. Les principales structures de données utilisées par MIOR sont une grille et un vecteur de structures par population, contenant les informations de chaque agent.
— La gestion de l'accès aux ressources partagées. La parallélisation de l'algorithme séquentiel MIOR implique le partage de nombreux dépôts de carbone entre colonies microbiennes. Il est nécessaire, à ce niveau, de garantir un accès équitable à ces ressources pour ne pas pénaliser certains agents. Cette problématique est l'occasion d'étudier l'application des barrières d'utilisation OpenCL.
— Le choix du nombre d'itérations de la simulation à exécuter. Ce nombre peut être directement indiqué en paramètre du lancement, ou déterminé à partir de l'évolution du modèle. Dans ce second cas se pose alors la question de définir la ou les métriques permettant de déterminer s'il y a lieu d'arrêter l'exécution.

Organisation de l'exécution en parallèle

Le plugin MIOR repose sur l'utilisation d'un bloc, ou work-group, pour traiter chaque simulation MIOR. A l'intérieur de ce bloc, chaque agent de la simulation est associé à un thread GPU. La simulation d'une itération de la simulation est découpée en fonctions OpenCL distinctes, pour permettre leur appel de manière indépendante à des fins de tests ou en un seul lancement pour effectuer une ou plusieurs itérations.

L'utilisation d'un work-group par simulation permet l'exécution de plusieurs modèles en parallèle, tel qu'illustré sur la Figure 8.4. Ce choix permet également de tirer parti des possibilités de recouvrement d'exécution offertes par OpenCL : si l'exécution de un ou plusieurs work-items est bloquée (accès mémoire, opération coûteuse) les ressources disponibles peuvent être allouées à d'autres work-items en attente d'exécution. Cette exécution de multiples simulations permet également de garantir un bon remplissage des coeurs fournis par la plate-forme, ce qui n'est pas toujours possible avec une seule simulation en fonction du nombre d'agents à traiter.

La possibilité de lancer plusieurs simulations MIOR simultanément réduit également le nombre de copies et de données nécessaires pour une même quantité de simulations. Étant donné la rapidité de chaque simulation, ce facteur a un impact direct sur les performances obtenues, tel qu'illustré dans nos expérimentations.

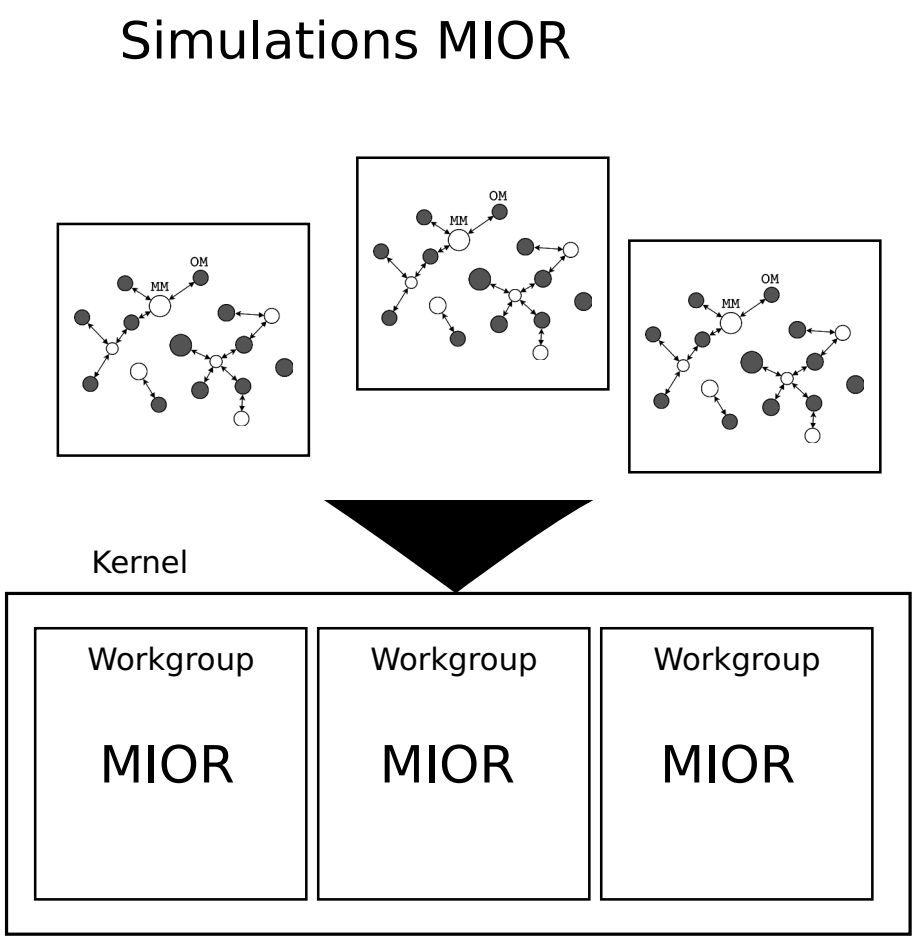

FIGURE 8.4 – Répartition de l'exécution de plusieurs modèles MIOR en OpenCL

Adaptation des structures de données

L'adaptation du modèle MIOR requiert la conversion des données du modèle existant en structures de données OpenCL. Dans le cadre de la réalisation de ce plugin, l'environnement et les agents sont représentés sous forme de tableaux statiques de structures représentant l'état de chaque entité. Le comportement associé à ces entités est, quant à lui, implémenté sous forme de fonctions OpenCL, appelées par les programmes OpenCL exécutés.

Quatre structures de données, illustrées sur le Listing 8.1, sont employées pour représenter chaque simulation MIOR :

— Un tableau d'agents MM, stockant l'état des colonies microbiennes du modèle.
— Un tableau d'agents OM, représentant l'état des dépôts de matière organique.
— Une matrice de topologie, stockant les informations d'accessibilité
— Une structure monde globale, stockant à la fois les paramètres du système (taux de respiration, de métabolisme) et les données résultat (quantité de CO_2 produite).

Listing 8.1 – Structures de données MIOR utilisées en OpenCL

```
// Colonie microbienne
typedef struct MM {
    float   x;         // Position X
    float   y;         // Position Y
    int     carbon;    // Carbone de la colonie
    int     dormancy; // État actuel
} MM;

// Dépôt de carbone
typedef struct OM {
    float   x;         // Position X
    float   y;         // Position Y
    int     carbon;    // Carbone du dépôt
} OM;

// Environnement de simulation
typedef struct World {
    int     nbMM;
    int     nbOM;
    int     RA;        // Rayon d'action
    float   RR;        // Taux de respiration
    float   GR;        // Taux de croissance
    float   K;         // Taux de décomposition
    int     width;     // Taille du modèle
    int     minSize; // Taille minimale d'une colonie microbienne
    int     CO2;       // Quantité totale de CO2 dans le modèle
} World;
```

L'ensemble de ces structures est d'abord alloué et initialisé par le processus principal, puis copié sur le périphérique d'exécution OpenCL.

La topologie du modèle peut être représentée sous deux formes (Figure 8.5) :

— En associant directement à chaque jeu de coordonnées (i, j) de la matrice l'information de voisinage entre la colonie microbienne i et le dépôt de carbone j.
— En représentant ces informations sous forme de structure compacte en nombre d'accès. Notre proposition, basée sur [JGLG09], permet de diminuer le nombre d'accès mémoire devant être effectués pour accéder à tous les voisins associés à un agent particulier. Cette représentation compacte se traduit par le stockage contigu des numéros d'index associés à chaque agent dans chaque ligne de la matrice, mais requiert une duplication de la structure, comme dans le cas des techniques de linéarisation de grille évoquées dans les structures de MCMAS, pour permettre un accès efficace d'un point de vue ligne (index MM connu, recherche des OM associés) et colonne (index OM connu, recherche des MM associés).

L'utilisation d'une représentation compacte (en nombre d'accès) consomme davantage de mémoire mais permet une réduction du nombre d'accès mémoire nécessaires pour le traitement du modèle proportionnelle à la densité de remplissage de la matrice de topologie. Une utilisation de la matrice à 10% permet ainsi de réduire d'autant le nombre d'accès mémoire nécessaires au parcours de toutes les cellules utilisées de la structure dans le cadre de la mise à jour du modèle.

L'allocation dynamique de mémoire n'est actuellement pas possible en OpenCL, et vient seulement d'être introduite dans les dernières versions du standard CUDA. Toutes ces structures de matrices doivent donc être allouées de manière statique sur CPU en prenant en compte le pire des cas possibles, où tous les agents OM du modèle accessibles depuis tous les agents MM. Une allocation moins pessimiste est possible en ajoutant une étape de pré-traitement du modèle, de manière à

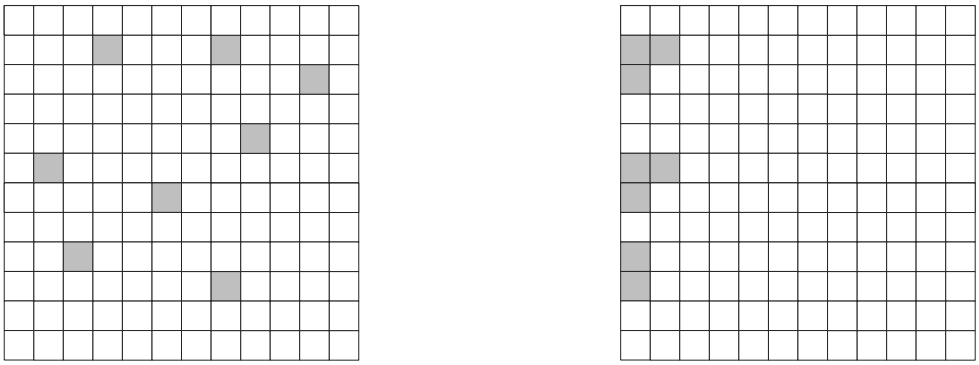

FIGURE 8.5 – Représentation creuse et compacte en accès de la topologie MIOR

compter le nombre de liaisons devant effectivement être représentées, au prix toutefois d'une étape de calcul supplémentaire. Cette piste n'a pas été évaluée dans notre étude.

Gestion de l'accès aux ressources critiques

Deux points critiques du modèle MIOR sont l'équité d'accès aux ressources en carbone pour les colonies microbiennes du modèle et la nécessité d'assurer la cohérence des mises à jour de données, pour éviter toute perte ou gain de matière dans le modèle.

Sur une architecture massivement parallèle telle que les cartes graphiques, ce type de synchronisation peut très rapidement devenir coûteux, et entraîner une séquentialisation de l'algorithme. Dans ce cas extrême, l'ensemble des fils d'exécution est bloqué en attente d'un verrou, et un seul agent peut s'exécuter : les performances obtenues sont alors inférieures à celles d'une simple implémentation séquentielle, du fait des latences et de la complexité introduites par la gestion de la synchronisation. Il est donc critique, dans la parallélisation complète d'un système multi-agents, de s'assurer que les agents seront effectivement capables de s'exécuter de manière indépendante, pour tirer parti du parallélisme.

Pour traiter cette problématique, l'algorithme de la simulation MIOR a été adapté pour permettre un découpage en trois étapes d'exécution parallélisables, séparées par des barrières de synchronisation. Ce fonctionnement est permis par un découpage en parts de carbone des ressources présentes dans le modèle :

1. *distribution* : le carbone disponible dans chaque dépôt de carbone (OM) est partagé en parts équitables entre tous les MM y ayant accès.
2. *simulation du métabolisme* : les différents processus métaboliques associés aux colonies microbiennes (respiration, croissance) sont appliqués en parallèle pour chaque agent sur les parts de carbone qui lui sont associées.
3. *rassemblement* : les parts de carbone restantes non consommées sont réintégrées dans leur dépôt d'origine.

Cette solution permet de réduire le nombre de synchronisations nécessaires à chaque itération à trois barrières, plutôt qu'un grand nombre de verrous, et permet également d'exécuter ces trois étapes en un seul lancement depuis le langage Java.

Détection de la terminaison

Si le modèle MIOR s'intéresse à l'évolution microscopique d'un système, il est tout à fait possible de le coupler avec des modèles s'appliquant à d'autres échelles de taille, notamment macroscopiques, pour obtenir une simulation plus exhaustive du problème : on parle alors de simulation multi-échelles.

Sworm [BMD+09] est un exemple de système multi-agents agent pouvant compléter l'évolution à l'échelle microscopique décrite par MIOR. Ce modèle permet de représenter l'effet de la macrofaune (par exemple les vers de terre) et de la microfaune (les bactéries) sur l'évolution des quantités de matière organique dans le sol. L'implémentation de Sworm est développée en Java sur la plate-forme Madkit [GF00a]. Elle se focalise sur l'effet bioturbant (déplacement des matières minérales et organiques) causé par les vers de terre dans le sol. Cette version ne prenant pas en compte l'activité microbienne, l'objectif du modèle MIOR est de simuler cette activité à l'échelle d'un cube de sol de 0.002 mm, là où le modèle Sworm s'intéresse à des échelles de sols de 20 cm.

La représentation des données sous la forme d'unités de sol est liée à cette intégration : pour ne pas imposer l'instantiation de l'ensemble des cellules du volume de sol représenté, Sworm se base sur une représentation des données de type fractale (Figure 8.2). Cette organisation permet l'allocation et le raffinement de la représentation de chaque cellule de sol à la demande. Elle est totalement transparente du point de vue de la représentation des données du modèle MIOR, qui ne manipule que des cellules de la plus petite échelle de représentation. Elle possède cependant son importance en termes de scénarios d'exécution devant être envisagés pour ce modèle. L'instantiation de nouvelles unités macroscopiques de sol implique en effet le lancement de nombreuses simulations MIOR, qui peuvent être déléguées par lot sur GPU.

Cette utilisation dans le cadre d'un autre modèle pose cependant le problème de pouvoir contrôler le temps d'exécution, et donc la quantité de traitements effectués par ces simulations MIOR. Il est possible de définir deux critères de terminaison pour assurer ce contrôle :

— Stabilisation de l'évolution du modèle sur N itérations
— Exécution d'un nombre fixé d'itérations

Ces deux critères répondent à des optiques différentes, avec toutefois systématiquement la volonté de pouvoir suivre l'historique de l'évolution des principales données du modèle (quantité de carbone, de CO^2).

Stabilisation de l'évolution du modèle sur N itérations

Ce critère de terminaison correspond à une absence d'évolution d'un ensemble de métriques pendant un nombre fixé d'itérations. Cet ensemble peut comprendre un nombre variables de métriques, suivant le niveau d'évolution à surveiller : une simple surveillance de la quantité globale de carbone stockée par les colonies microbiennes permet par exemple de déceler tout arrêt de fixation du carbone sur cette période de temps. Cet état ne garantit pas cependant l'arrêt de l'évolution du modèle, mais simplement l'absence de ressources en carbone suffisantes pour déclencher la moindre croissance microbienne. La poursuite du processus de respiration n'est pas prise en compte. La surveillance des quantités de dioxyde de carbone (CO_2) pallie à ce défaut, du fait de sa production lors du processus de respiration. Une surveillance de ces deux quantités n'est cependant pas nécessaire pour garantir l'arrêt des deux processus d'évolution dans le modèle : la respiration prenant le pas sur toute croissance dans l'algorithme, la cessation de ce processus suffit à garantir l'arrêt du second.

L'utilisation de N itérations de surveillance est rendue nécessaire par l'obligation de prendre

en compte l'éventuel décès d'une ou plusieurs colonies bactériennes, une fois ce type d'équilibre atteint. Ces décès, en diminuant la concurrence d'accès aux dépôts de carbone, augmentent la quantité de carbone utilisable par les autres colonies microbiennes en partageant l'accès. Cette augmentation est susceptible de permettre à ces colonies de sortir de leur état de dormance et de relancer une nouvelle phase d'évolution du système.

L'inconvénient de ce critère d'arrêt est la difficulté d'estimer à priori le nombre d'itérations nécessaires à la stabilisation du système. L'exécution sur GPU ne pouvant pas être interrompue par le programme, il est difficile, même avec une connaissance précise des paramètres de la simulation, de par la nature aléatoire du positionnement des agents, et donc leur accès aux ressources, et de certains processus, de calculer une échéance de temps fiable avant obtention du type de stabilisation de l'évolution recherchée.

Exécution d'un nombre fixé d'itérations

Ce critère de terminaison est totalement agnostique vis-à-vis de l'état du modèle, et considère le seul nombre d'itérations de la simulation écoulé comme indicateur d'arrêt. Cette limite permet, en connaissant la durée moyenne d'une itération, d'estimer le temps total nécessaire pour effectuer le calcul demandé. En variant le nombre d'itérations exécutées pour chaque lancement GPU, cette information permet le contrôle de la latence maximale entre deux retours de résultats.

L'estimation de la durée d'une itération est possible, en dépit de la nature stochastique de certaines portions de l'algorithme (positions, probabilité de décès) en considérant le cas le plus coûteux possible en complexité d'exécution. Dans le cas de MIOR, les seules boucles de l'algorithme sont associées à des parcours de relations inter-agents. Ce coût revient à calculer le nombre maximal de relations pouvant être présentes dans le modèle. Ce nombre est atteint si chaque colonie microbienne a un rayon d'interaction égal ou supérieur à la dimension la plus grande du modèle. Tous les dépôts de carbone présents dans le modèle sont alors accessibles à chaque colonie microbienne, ce qui se traduit, pour n colonies microbiennes et m dépôts de carbone, par un total de $n * m$ relations possibles.

Les deux approches de terminaison sont rendues possibles par le plugin MIOR fourni avec MCMAS, de manière à permettre le choix de l'une ou l'autre des approches par le modèle Sworm, en fonction du nombre de simulations à lancer et des impératifs en temps du modèle macroscopique.

8.1.3 Un modèle macroscopique : Collemboles

Après avoir présenté la parallélisation du modèle MIOR, nous nous intéressons à présent à un autre système multi-agents, macroscopique cette fois, Collemboles. Ce modèle nous permet de mettre en avant un exemple de parallélisation complète de modèle sur GPU au moyen de plusieurs kernels d'exécution lancés de manière asynchrone.

Présentation du modèle

Le modèle Collemboles est un système multi-agents conçu pour modéliser la diffusion d'arthropodes, des collemboles, entre des parcelles de plusieurs types naturelles, forestières ou artificielles, en vue d'étudier leur impact sur la biodiversité. Il est basé sur le chargement de données depuis un système d'information géographique pour obtenir un espace en deux dimensions découpé en parcelles de terrain de forme polygonale, tel qu'illustré par la Figure 8.6. Cet environnement continu

est ensuite décomposé en cellules, ou patchs, correspondants à une aire de sol fixe, qui sont utilisés comme unités de base de modélisation. L'implémentation de référence de cet algorithme a été réalisée en NetLogo, de manière à permettre une visualisation aisée de l'évolution de la répartition géographique et de la densité de population des individus.

FIGURE 8.6 – État initial d'une simulation Collemboles - Implémentation NetLogo

L'évolution de la simulation est découpée en quatre étapes, appliquées au niveau de chaque cellule :

1. L'*Arrivée* de nouveaux individus. Cette opération correspond à la distribution équitable d'une fraction de la population de chaque parcelle à toutes les cellules la constituant. Toutes les populations du modèle étant entières, ce processus n'a d'effet au niveau de chaque parcelle que si cette fraction de nouveaux individus représente un nombre supérieur à la quantité de cellules de la parcelle.
2. La *Reproduction*, qui consiste à mettre à jour la population de chaque parcelle pour correspondre à la somme des populations de toutes les cellules la composant, pour permettre un suivi de l'évolution de chaque parcelle.
3. La *Diffusion*. Cette opération consiste à diffuser une portion de la population de chaque cellule à ses huit voisines, comme évoqué dans nos exemples de plugins MCMAS. Cette diffusion se différencie de celle fournie par notre bibliothèque par le fait qu'elle n'a lieu que si le terrain de la parcelle courante est adapté aux collemboles et si la population globale de la parcelle atteint un certain seuil.
4. La *Mort* des individus. A la fin de chaque itération, toute population non nulle de collemboles au niveau de parcelles inadaptées à ces individus est fixée à zéro pour indiquer sa disparition.

Cet algorithme, relativement simple, est cependant composé de plusieurs opérations coûteuses en temps de calcul mais parallélisables : la diffusion et la mise à jour des populations. L'application

de ces traitements à chaque cellule est cependant largement conditionnelle, en fonction d'informations externes telles que le type ou la population de la parcelle, ce qui rend difficile l'utilisation des primitives de haut niveau de diffusion ou de réduction fournies par MCMAS. Dans la suite de cette section, nous évoquons un autre moyen de paralléliser cette exécution à l'aide de l'interface bas niveau donnant accès à OpenCL.

Implémentation

Comme nous venons de le voir, les différents traitements mis en jeu par le modèle Collemboles sont parallélisables, mais mettent en jeu de nombreuses conditions externes qui rendent difficile l'utilisation de primitives de haut niveau MCMAS. Dans ces conditions, nous avons retenu une autre approche d'implémentation basée sur l'utilisation de l'interface de bas niveau pour décomposer le déroulement de la simulation en quatre traitements distincts lancés sans synchronisation intermédiaire sur GPU, tel qu'illustré par la Figure 8.7 :

— Un kernel responsable de la gestion des nouveaux arrivants.
— Un kernel chargé de la reproduction des individus.
— Un kernel responsable de la diffusion des populations sur les parcelles propices.
— Un dernier kernel gérant la disparition des populations situées sur des parcelles inadaptées.

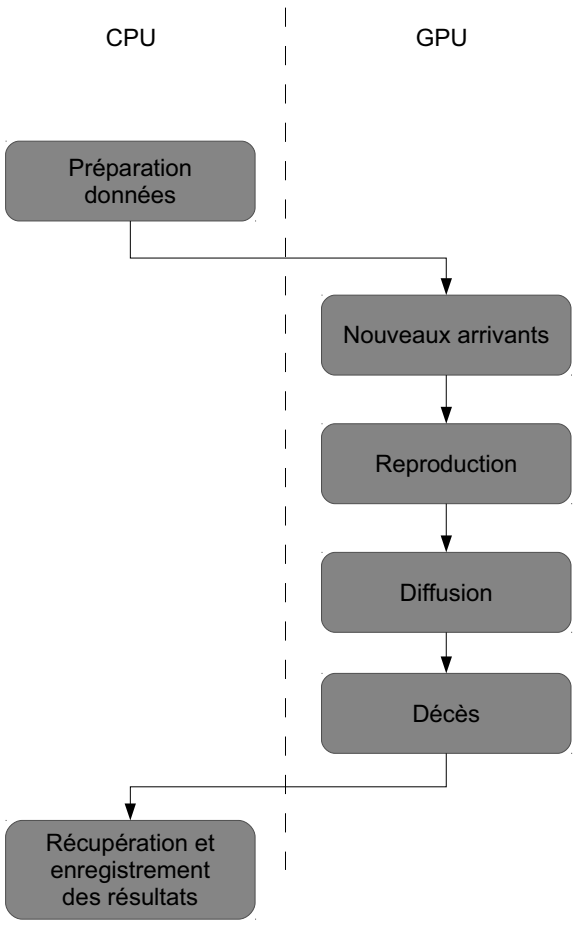

FIGURE 8.7 – Découpage d'une itération collembole entre CPU et GPU

Structures de données

Deux structures principales sont mises en jeu dans le modèle Collemboles :

— Les parcelles, associées à une surface, une population et un type de terrain.
— La grille représentant l'environnement de simulation, dont chaque cellule comprend une indication de parcelle et une population.

La représentation des parcelles en OpenCL peut être effectuée, comme nous l'avons vu dans notre réflexion sur les structures de données agents, sous forme de tableaux de structures ou d'un tableau pour chaque propriété.

Nous avons choisi, dans le cas du modèle Collembole, d'avoir recours à un tableau par propriété pour représenter les parcelles pour plusieurs raisons :

— Le faible nombre de propriétés associées à chaque parcelle limite l'explosion du nombre de paramètres sur GPU.
— Seules une ou deux de ces propriétés sont utilisées à chaque étape des traitements. Le découpage des propriétés en structures distinctes permet donc de récupérer certaines informations du modèle en cours d'exécution à des fins de vérification ou de stockage sans perturber l'exécution des traitements n'y ayant pas recours.
— Les traitements des cellules étant toujours basés sur les mêmes propriétés dans un même traitement, ce découpage maximise la localité de ces propriétés en mémoire, à la fois pour optimiser les accès et l'occupation des éventuels caches L1 et L2 présents sur le matériel.

Ce raisonnement s'applique également à l'environnement, pour lequel l'utilisation d'une grille par propriété, assez similaire à celle de l'implémentation proie-prédateur, a été retenue. Dans le cas de la grille d'environnement, ce découpage en grilles de propriétés offre un avantage supplémentaire, en permettant de limiter au strict minimum les données devant être présentes en mémoire GPU à un moment donné.

Nouveaux arrivants

L'arrivée de nouveaux arrivants est traitée au niveau de chaque cellule. Pour cela, chaque instance du kernel récupère la population et le nombre de cellules de la parcelle associée à la cellule courante, pour déterminer le nombre d'individus devant être répartis sur chaque unité de sol. Cette opération revient à calculer de nombreuses fois la même fraction de population à répartir, mais permet à chaque thread de ne mettre à jour que sa cellule locale, ce qui élimine tout problème de synchronisation de l'écriture des données.

Reproduction

Ce traitement est également effectué pour chaque cellule de l'environnement. La réduction de la population au niveau de la parcelle est effectuée sous forme d'addition atomique, pour garantir la cohérence des totaux obtenus.

Le choix d'un traitement basé sur les parcelles, plutôt que les cellules, éviterait l'utilisation d'opérations atomiques, mais impose de disposer d'une liste des cellules associées à chaque parcelle pour ne pas avoir à parcourir l'ensemble de la grille, structure qui n'est pas présente dans notre modèle.

Le code final obtenu est très proche d'une opération de diffusion MCMAS, si ce n'est que les résultats sont réduits en plusieurs sous-totaux sur la base de parcelles plutôt que sous la forme d'une unique valeur scalaire.

Diffusion

La réalisation de la diffusion sur GPU des populations du modèle est effectuée en deux étapes, séparées par une barrière d'exécution.

— Une quantité d'individus à diffuser est calculée au niveau de chaque cellule.
— Chaque cellule récupère un huitième de la quantité de chaque cellule voisine et l'ajoute à sa propre population.

L'application des mises à jour sur la cellule associée à chaque thread, plutôt que d'effectuer directement les mises à jour sur les cellules voisines, permet comme dans le cas de l'arrivée de nouveaux individus de garantir l'absence d'écriture de la même donnée par plusieurs threads différents, et ainsi d'éviter la synchronisation des modifications associées dans le modèle.

Mort des individus

Le dernier traitement exécuté sur GPU pour chaque cellule est la mise à zéro de la population de chaque cellule inadaptée du fait du type de terrain aux individus collemboles. Cette dernière opération ne requiert pas de considération particulière en termes de synchronisation de l'exécution, la seule donnée utilisée en écriture est la cellule courante. Elle peut donc être directement réalisée sans adaptation particulière en OpenCL.

8.2 Etudes de performances

8.2.1 Supports d'exécution

Dans le cadre de nos études de performances, nous avons eu recours à une variété de supports d'exécution, tant grand public que orientés vers le calcul haute performance. Dans cette section, nous présentons les caractéristiques de chacun de ces matériels, pour les replacer dans le contexte de l'évolution de l'exécution sur GPU.

L'objectif de cette variété de supports est de permettre la comparaison entre gammes professionnelles et grand public d'une part, entre anciennes et nouvelles générations d'autre part, de différentes solutions matérielles d'exécution. De cette manière, il est possible de quantifier, pour un chercheur, le bénéfice pouvant être obtenu par l'utilisation de matériel spécialisé par rapport à celle de son poste personnel. Cette variété permet également de valider les performances obtenues de manière indépendante par rapport à une génération de matériel ou une implémentation OpenCL donnée.

Il est important de noter que le nombre de coeurs n'est pas directement comparable entre matériel NVIDIA et AMD, les unités d'exécution proposées par ce second fabriquant étant plus nombreuses mais également plus spécialisées.

Voici la liste de ces supports, classés par ordre chronologique.

NVIDIA Geforce 8800GT

Le premier matériel sur lequel nous avons eu l'occasion de réaliser des essais est une carte graphique milieu de gamme grand public de NVIDIA, la Geforce 8800GT, sortie en octobre 2007. Cette carte dispose de 112 unités d'exécution, soit 14 multi-processeurs, cadencés à 1.5 GHz et accompagnés de 512 Mo de mémoire vive, et offre une puissance théorique de 504 Gflops en

simple précision. Elle ne supporte pas matériellement la gestion des nombres en double précision. Dans nos expérimentations, elle est associée à un processeur Intel Core 2 Q9300 fonctionnant à 2.5 GHz. Elle ne propose pas de mécanisme de cache L1 et L2.

NVIDIA Tesla S1070

Le second matériel utilisé est un châssis graphique dédié au calcul GPU proposé par NVIDIA en 2009. Il est constitué de quatre cartes graphiques Tesla C1060 dotées de 240 unités d'exécution, soit 30 multiprocesseurs cadencés à 1.3 Ghz pour une puissance théorique de 933 Gflops par carte. Chaque carte est associée à 4 Go de mémoire vive. Dans le cadre de nos tests, un seul de ces GPU est utilisé, couplé à un processeur Intel Xeon X5550 cadencé à 2.67 Ghz. L'architecture matérielle de cette solution est très similaire à la Geforce 8800GT présentée précédemment. Elle se différencie par la quantité de mémoire disponible, 4 Go, ainsi que le support de la correction des erreurs mémoires (ECC) et des calculs en nombres flottants double précision. Elle ne propose pas de mécanisme de cache L1 et L2.

AMD Radeon HD6870

La carte graphique AMD Radeon HD6870 est une carte graphique grand public de milieu de gamme sortie en octobre 2010. Elle se caractérise par l'utilisation d'un mécanisme de cache L1 et L2 similaire à celui rencontré sur les cartes NVIDIA récentes, mais ne supporte pas toutefois le traitement matériel de nombres flottants en double précision. Elle est constituée, au niveau du matériel, de 1120 coeurs cadencés à une fréquence de 900 MHz, pour une puissance théorique de 2016 Gflops. Ces unités d'exécution sont associées à 1Go de mémoire vive intégrés à la carte. Dans nos expérimentations, elle est associée à un processeur AMD Phenom II X6 1090T cadencé à 3.2 GHz.

NVIDIA Geforce 560Ti

La Geforce 560Ti est une carte graphique grand public de milieu de gamme sortie en janvier 2011. Elle est basée sur l'architecture Fermi. Cette carte propose 384 unités d'exécution cadencées à 822 MHz et 1 Go de mémoire vive. Elle dispose d'un mécanisme de cache L1 et L2 et supporte les calculs en double précision, mais limite le débit d'opérations obtenu à un douzième de celui des traitements en simple précision. Dans notre configuration de test, elle est associée à un processeur Intel Core i7 2600K cadencé à 3.4 GHz.

NVIDIA Tesla K20

La carte graphique Tesla K20 est basée sur l'architecture matérielle Kepler et propose donc un cache L1 et un cache L2. Sortie fin novembre 2012, elle est destinée spécifiquement au calcul scientifique, et supporte matériellement le traitement de nombres flottants en double précision ainsi que la correction des erreurs mémoires. Elle est constituée au niveau matériel de 2496 coeurs graphiques cadencés à 706 MHz, pour une puissance théorique de 3520 Gflops. Ces coeurs d'exécution sont associés à 5120 Mo de mémoire vive. Les performances en double précision offertes par cette carte sont de l'ordre du tiers des performances obtenues en simple précision. Cette carte est associée dans nos expérimentations à un processeur Intel Xeon CPU E5-2609v2 cadencé à 2.50 GHz.

8.2.2 Protocole expérimental

L'évolution du matériel GPU est très rapide. Nous avons donc eu l'occasion, entre le début et la fin de nos recherches, de tester des supports très différents. Certains d'entre eux, comme la carte Geforce 8800GT, n'ont pas pu être utilisés pour l'ensemble de nos tests. Nous avons cependant inclus les courbes correspondantes pour permettre une comparaison avec nos autres plates-formes d'exécution.

Tous les tests ont été lancés sur des systèmes d'exploitation Linux 64 bits. Les courbes de la Geforce 8800GT ont été réalisées avec la version 3.2 de l'environnement CUDA fourni par la société NVIDIA. Toutes les autres courbes mettant en jeu du matériel de ce fabriquant ont été réalisées avec la dernière version stable, la version 5.2. Pour les tests de performance sur matériel AMD ou CPU, nous avons utilisé l'implémentation OpenCL proposée par l'environnement AMD APP en version 2.7.

Dans le cadre de nos expérimentations, nous avons choisi d'évaluer le temps d'exécution en fonction de la quantité de traitements. Nous avons pour cela fait varier la taille et le nombre d'agents du modèle dans le cas des modèles MIOR et Collemboles. Dans le cas de proie-prédateur, cette variation de la quantité des traitements est obtenue en modifiant le rayon de recherche de chaque individu.

Toutes les valeurs indiquées sont basées sur une moyenne des temps d'exécution obtenus sur plusieurs dizaines d'exécution, de manière à minimiser l'impact du système d'exploitation et du pilote graphique sur les temps observés.

8.2.3 Résultats obtenus

Proie-prédateur

Pour mesurer l'impact de l'utilisation du GPU sur les performances obtenues, nous avons choisi de faire varier le rayon de recherche de nouvelles positions sur GPU, en maintenant toutes les autres données constantes. De cette manière, il est possible de mesurer l'impact du nombre de cases parcourues et du nombre d'accès mémoires sur les performances obtenues. Le rayon de recherche est directement appliqué aux proies, et majoré de 50% pour les prédateurs, de manière à conserver un rapport fixe entre les champs de vision de chaque population. Nous avons utilisé pour nos expérimentation un environnement de dimension 1000, capable de tenir en mémoire sur tous nos supports d'évolution. Pour éviter de donner l'accès à l'ensemble de l'environnement à chaque individu, nous nous sommes arrêtés à un rayon de recherche de 100, correspondant à 1% de l'espace total simulé. Chaque courbe compare les performances obtenues entre la carte graphique et le CPU présent sur le même support d'exécution, ce qui explique l'allure différente de chaque courbe CPU.

Le temps moyen d'une itération est pris pour référence pour mesurer l'impact de ce rayon de recherche sur les performances du modèle. Ce temps est à chaque fois comparé entre l'implémentation basée sur MCMAS, et une implémentation reprenant exactement le même algorithme mais utilisant des équivalents réalisés en Java de ces opérations génériques, de manière à disposer de deux décompositions du programme équivalentes. La moyenne en temps d'exécution d'une itération sur cinquante itérations de la simulation est retenue comme référence.

La Figure 8.8 illustre les temps obtenus sur carte Kepler et sur le processeur Xeon correspondant. Tant que le rayon de recherche configuré pour les individus proies demeure inférieur à 30, l'implémentation GPU présente des performances très similaires à la version CPU. Cette proximité

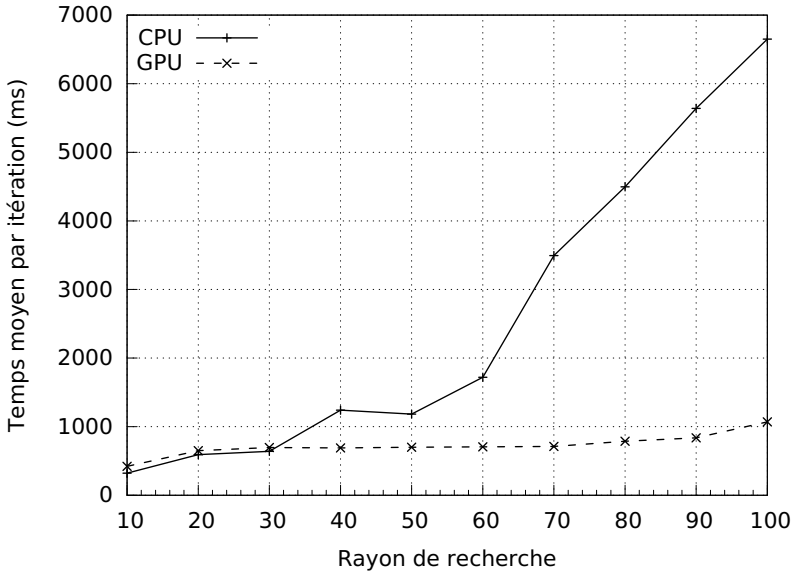

FIGURE 8.8 – Temps moyen d'exécution d'une itération du modèle proie-prédateur sur carte Kepler K20m et processeur Xeon CPU E5-2609v2

s'explique par le coût fixe du reste de l'itération, mais permet toutefois d'illustrer que le recours au GPU ne pénalise pas les performances du modèle, même à petite échelle. La version de la simulation s'exécutant sur carte Kepler prend l'avantage à partir d'un rayon de recherche supérieur à 30, et cet avantage devient particulièrement marqué à partir d'un rayon de recherche de 60, où les performances CPU présentent un important ralentissement. La différence de performance atteint un facteur 7 en faveur de la carte GPU quand le rayon de recherche approche de 100.

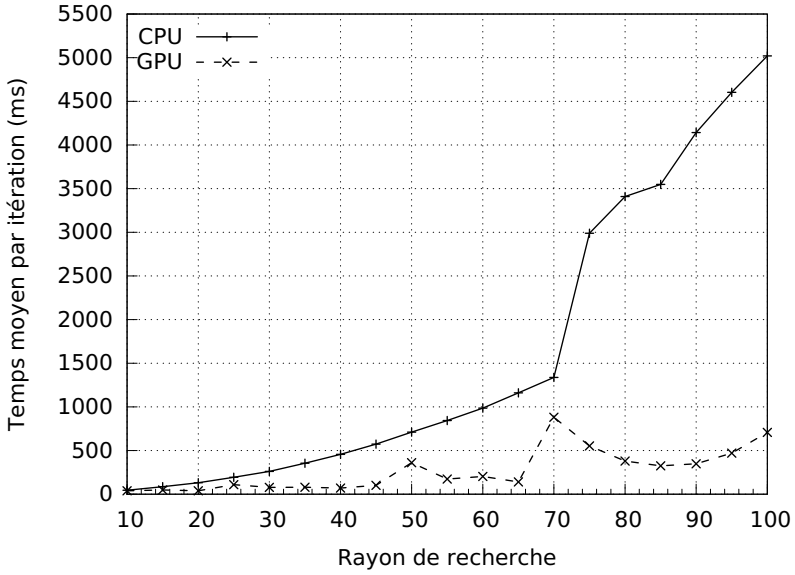

FIGURE 8.9 – Temps moyen d'exécution d'une itération du modèle proie-prédateur sur carte Radeon HD6870 et processeur AMD Phenom II X6 1090T

La Figure 8.9 illustre les performances obtenues sur la Radeon HD6870 décrite dans nos supports d'exécution. L'implémentation GPU prend l'avantage dès un rayon de recherche de 10, et cet avantage va ensuite en s'accentuant irrégulièrement jusqu'à un rayon de recherche de 70, au-delà

duquel la courbe CPU indique une brusque dégradation des performances. Si le comportement du CPU par rapport à l'augmentation du rayon de recherche demeure très régulière, il est possible de remarquer que les performances obtenues par la carte Radeon manifestent des variations marquées. Les temps obtenus sur GPU sont meilleurs qu'avec notre carte Kepler, ce qui tend à confirmer l'excellente réputation des cartes AMD en exécution GPU et en support d'OpenCL.

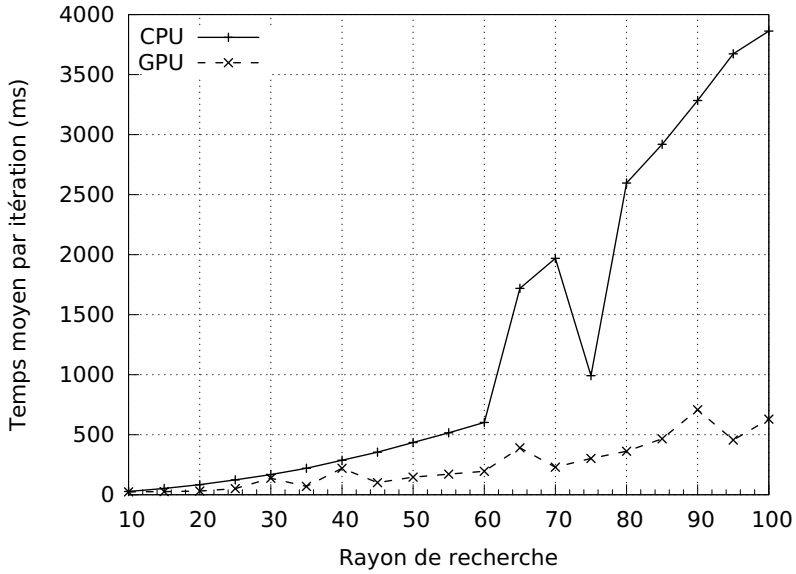

FIGURE 8.10 – Temps moyen d'exécution d'une itération du modèle proie-prédateur sur carte Geforce 560Ti et sur CPU

La Figure 8.10 montre les performances obtenues sur la plate-forme matérielle Geforce 560Ti. Elle se caractérise, comme dans le cas de la carte Kepler, par des performances initialement très proches entre version CPU et GPU, avant que les temps ne tournent nettement à l'avantage de cette dernière à partir d'un rayon de recherche de 40. Cet avantage devient encore plus marqué à partir d'un rayon de 60. La courbe GPU présente également, quoique de manière moins accentuée, les pics observés sur les temps de la carte Radeon. L'accélération finale obtenue entre CPU et GPU est du même ordre qu'avec la carte Kepler, un facteur sept, au rayon de recherche 100.

Si l'objectif de ces trois courbes est de mettre en avant les gains permis par l'utilisation de deux ressources de calculs différentes, CPU et GPU, sur une même machine, il est également intéressant de comparer les résultats obtenus entre matériels CPU et matériels GPU.

— Entre matériels GPU, la carte Kepler se caractérise par une très grande régularité des performances obtenues, cependant légèrement en deçà des cartes grand public récentes. Cet handicap relatif peut être mis sur le compte de la fréquence de fonctionnement moins élevée de chaque coeur graphique (700 MHz contre 800 MHz ou plus). L'utilisation des ressources est à contrario plus stable sur cette carte, comme l'indique la courbe plus régulière. Les ressources en mémoire plus importantes offertes par la carte permettent également d'envisager des scénarios de taille supérieure.
— Entre CPU, les comportements en termes de performance sont très similaires, avec une première portion linéaire, suivie d'un point charnière avant que les performances ne se dégradent plus rapidement. Ces courbes mettent en avant le temps supérieur utilisé par l'implémentation correspondant au CPU associé à la carte Tesla K20. S'il s'agit du processeur le plus récent de nos tests, ce retard en temps est aisément expliqué par le fait que l'exécution CPU ne tire pas parti de tous les coeurs d'exécution dans nos tests. Dans

ces conditions, la fréquence joue un rôle très important, directement visible sur la courbe correspondant au processeur AMD (3.2 GHz) et encore plus sur la courbe correspondant au processeur Core i7 2600K (3.4 GHz). Par comparaison, le processeur Xeon testé ne fonctionne qu'à 2.5 GHz.

MIOR

Pour évaluer les performances de ce nouveau processus de distribution/rassemblement des ressources du modèle sur GPU, nous avons choisi de comparer les performances obtenues par l'implémentation sur plusieurs modèles de cartes graphiques. Pour illustrer l'impact des changements incrémentaux apportés à l'algorithme original, nous avons également testé cinq implémentations successives, comprenant un nombre croissant d'adaptations pour l'architecture GPU. Dans tous les cas, le temps d'exécution moyen de 50 simulations a été retenu comme indicateur de performance.

Voici les caractéristiques des implémentations comparées :

— L'implémentation **GPU 1.0** est une adaptation directe de l'algorithme et des structures de données présentées, incluant seulement la suppression des dépendances d'accès aux données par le biais du mécanisme de distribution/rassemblement décrit dans l'implémentation du modèle.
— L'implémentation **GPU 2.0** ajoute à l'algorithme de la première implémentation l'utilisation d'une représentation alternative, plus compacte en termes d'accès, pour le stockage de la topologie du modèle. Cette représentation exige toutefois des quantités plus importantes de mémoire vive.
— L'implémentation **GPU 3.0** introduit l'utilisation de la mémoire locale du périphérique (voir présentation de l'architecture GPU), au moyen de copies manuelles de données les plus utilisées (parts de carbone) au début et à la fin de chaque pas de calcul. L'algorithme est par ailleurs identique à celui de la seconde implémentation.
— L'implémentation **GPU 4.0** introduit la possibilité de résoudre plusieurs simulations en parallèle pour chaque lancement de kernel. L'algorithme de chacune de ces simulations est identique à la première implémentation.
— L'implémentation **GPU 5.0** permet, de manière similaire à l'implémentation **GPU 4.0**, le lancement de plusieurs implémentations **GPU 2.0** de manière simultanée.

Les deux dernières implémentations présentées, **GPU 4.0** et **GPU 5.0**, ont pour objectif de permettre au modèle de profiter des possibilités de recouvrement d'exécution offerts par l'ordonnanceur GPU, en s'assurant qu'il existe toujours des agents à exécuter en cas de blocage (barrière) d'une simulation particulière, comme évoqué dans nos bonnes pratiques. Le fait de disposer de nombreux threads permet en effet dans ce cas au matériel de traiter d'autres agents MIOR, pendant que certains agents sont en attente de ressources.

Pour permettre la comparaison de ces implémentations avec la version séquentielle originale, une version **CPU** réalisée en Java est également incluse.

Les figures 8.11, 8.12 et 8.13 donnent le temps d'exécution de 50 simulations sur nos supports Tesla S1070, Geforce 560Ti et Tesla K20. Pour permettre une mesure de l'impact de la taille du modèle et du nombre d'agents sur les performances, un facteur d'échelle est appliqué horizontalement : à l'échelle 1, le modèle comprend 38 colonies microbiennes (MM) et 310 dépôts de carbone (OM). Ces nombres sont multipliés par 6 à l'échelle 6, et la taille de l'environnement est également modifiée pour conserver la même densité moyenne d'agents dans le modèle.

La Figure 8.11 met tout d'abord en évidence les performances obtenues sur une carte graphique Tesla C1060 dénuée de cache. Ces courbes illustrent l'avantage initial en performance marqué en faveur de l'implémentation GPU 2.0. Les autres variantes GPU sont plus lentes à prendre l'avantage sur le CPU, et ne se détachent réellement en performance qu'à partir de l'échelle 4, pour ensuite continuer à offrir des performances très proches. L'amélioration de performance obtenue entre l'implémentation la plus rapide, GPU 2.0, et l'exécution sur CPU devient de l'ordre de 10 à l'échelle 10.

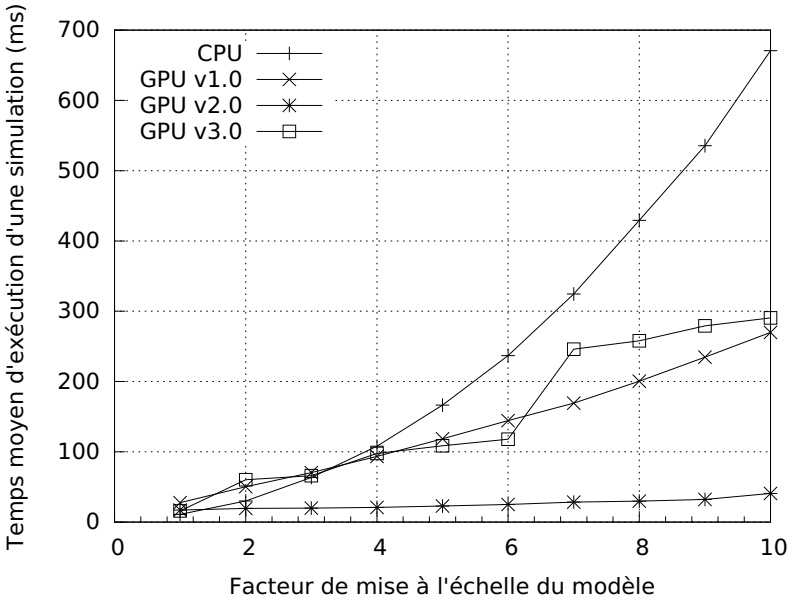

FIGURE 8.11 – Performances CPU et GPU MIOR sur carte Tesla C1060

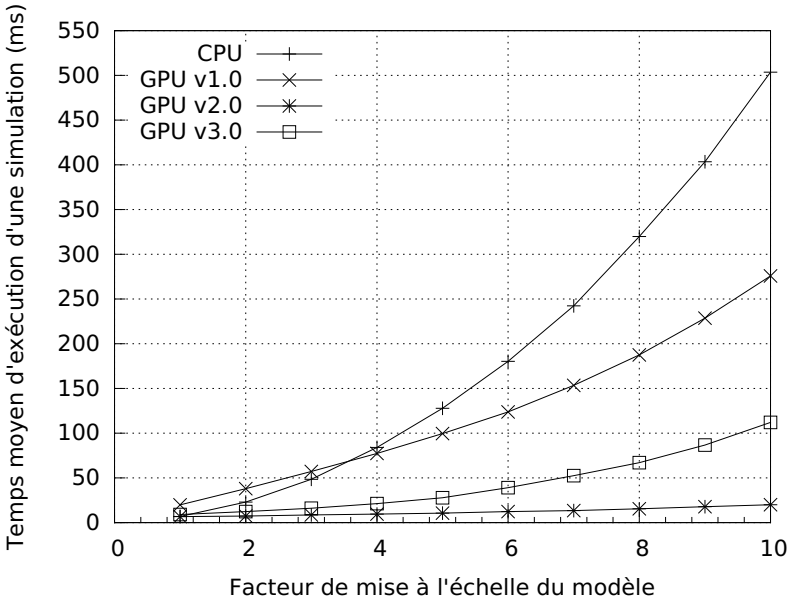

FIGURE 8.12 – Performances CPU et GPU MIOR sur carte Geforce 560Ti

La Figure 8.12 illustre les performances obtenues sur un matériel grand public beaucoup plus récent, et illustre en particulier les importants progrès réalisés par la gestion de la mémoire locale. Les courbes possèdent, en dehors de cet élément, une évolution très similaire, avec des temps d'exécution absolus toutefois de l'ordre de deux fois plus rapides, du fait de la fréquence et du nombre de cœurs plus importants sur cette plate-forme. Les courbes se caractérisent de manière générale par leur évolution très régulière, sans la rupture en performance rencontrée par la carte Tesla. Cette régularité peut être expliquée par la présence de cache d'exécution sur cette nouvelle architecture, et d'un meilleur algorithme de regroupement des accès mémoires, plus à même de gérer les accès à la topologie effectués par la simulation MIOR. Ces courbes illustrent une nouvelle fois un avantage de l'implémentation GPU 2.0 sur les autres implémentation. Elle est ainsi cinq

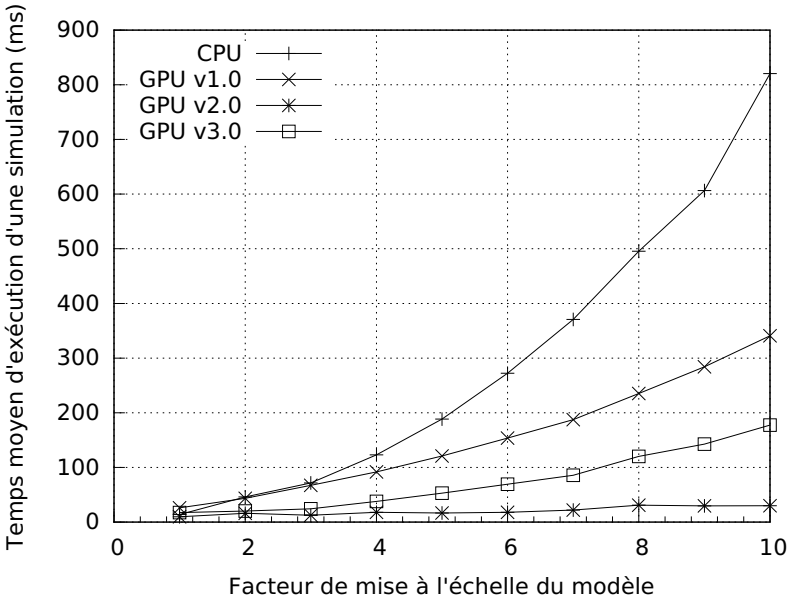

FIGURE 8.13 – Performances CPU et GPU MIOR sur carte Kepler K20m

fois plus rapide que l'implémentation GPU 3.0 à l'échelle 10. Cet avantage sur l'implémentation GPU 3.0 est toutefois moins marqué que pour la carte Tesla C1060, ce qui montre l'intérêt de nombreuses optimisations des accès mémoires effectués sur le coût de recopie des données en mémoire locale. Les courbes GPU 1.0 et CPU illustrent des tendances et des temps d'exécution très similaires au support d'exécution plus ancien.

La Figure 8.13 correspondant à la carte Tesla K20 est extrêmement similaire aux courbes obtenues sur Geforce 560Ti, ce qui s'explique aisément par la proximité dans le temps et en termes d'architecture, Kepler contre Fermi, entre ces deux cartes. Kepler se caractérise ici par des performances en léger retrait, comme dans le cas du modèle proie-prédateur. Ce retrait peut encore une fois être expliqué par la différence de fréquence de fonctionnement entre les deux cartes, dédiées à des utilisations différentes. Ces courbes confirment également encore une fois l'avantage en termes de fréquence de processeurs comme le Core i7 2600K sur des processeurs plus récents mais moins véloces comme le Xeon, dans le cadre d'une exécution séquentielle. L'implémentation GPU 2.0 permet de nouveau l'obtention des meilleures performances, en étant approximativement six fois plus rapide à l'échelle 10 que l'implémentation GPU 3.0. L'implémentation GPU 1.0 demeure la plus lente des implémentations GPU, avec un facteur 10 par rapport à l'implémentation GPU 2.0 à l'échelle 10.

Le plugin MIOR est conçu pour pouvoir réaliser un grand nombre de simulations microscopiques dans le cadre de la simulation multi-échelles Sworm. Dans ces circonstances, il est intéressant de mesurer le coût d'un lancement MIOR sur GPU, ainsi que l'évolution des performances obtenues en fonction du nombre de simulations demandées, de manière à évaluer la taille de lot la plus efficace. L'objectif, de cette manière, est d'amortir les coûts de transferts liés à l'exécution sur GPU, tout en étant capable de connaître le temps d'exécution total du lot de simulation, de manière à éviter de bloquer d'autres traitements Sworm.

Au vu de ces éléments, il existe plusieurs manières de mesurer les performances de l'exécution en parallèle de plusieurs simulations :

— En mesurant les performances d'un seul lancement et en variant le nombre de simulations

(Figure 8.14). Cette approche permet de mesurer l'évolution du temps total d'exécution de l'ensemble des simulations, théoriquement linéaire (loi d'accélération). Une stagnation de la courbe indique une amélioration de l'efficacité d'exécution sur GPU, et une augmentation de sa pente illustre au contraire une augmentation du coût de la parallélisation. Une mauvaise efficacité peut être compensée, à plus grand échelle, par le recouvrement des accès et des calculs proposés par la carte graphique, possible uniquement quand le nombre de threads d'exécution en attente est suffisant.

— En mesurant le temps d'exécution total pour effectuer un nombre fixe de simulations (Figure 8.15) en faisant varier le nombre de simulations lancés simultanément. Cette approche permet de mettre en évidence les coûts associés aux transferts et aux lancements, par rapport au nombre et à la durée des calculs utilisés.

La Figure 8.14 illustre les temps d'exécution obtenus pour le lancement d'un nombre variable de simulations en une seule fois. Les courbes montrent que, pour des petits nombres de simulations, l'implémentation compacte de la topologie mémoire est plus performante que la représentation sous forme de matrice pleine à deux dimensions. Cette tendance s'inverse au-delà de 50 simulations exécutées en parallèle, ce qui s'explique soit par une progression non linéaire des coûts de synchronisation, soit par la consommation mémoire supplémentaire imposée par l'utilisation de la représentation optimisée en accès.

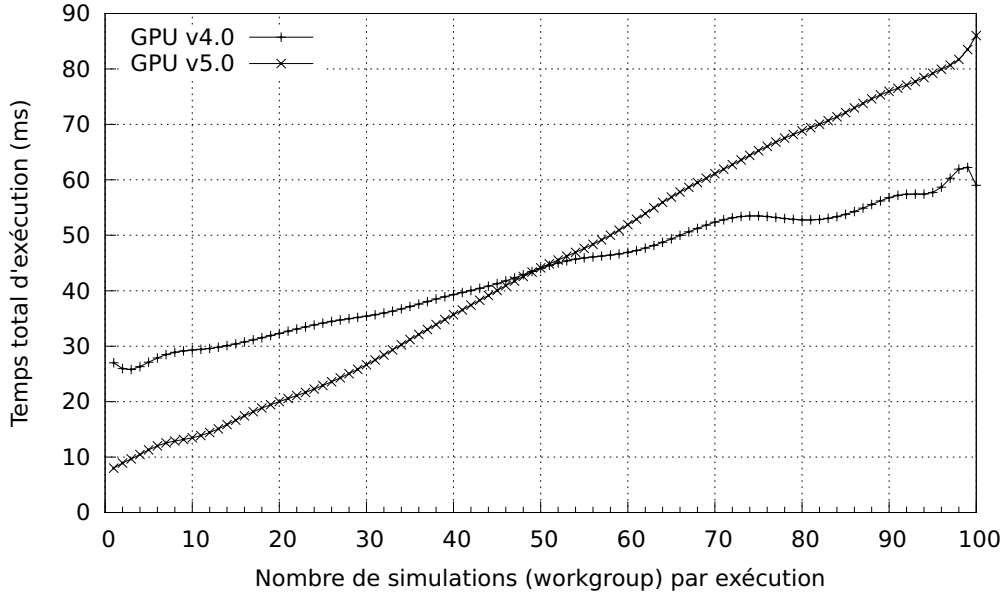

FIGURE 8.14 – Temps d'exécution par simulation MIOR sur Tesla C1060, en fonction du nombre total de simulations

La Figure 8.15 illustre les temps d'exécution obtenus pour l'exécution d'un même nombre total de simulations, en variant le quantité de systèmes lancés en simultané. Les courbes illustrent cette fois les coûts résultants de l'exécution sur GPU pour des lancements de petite taille. Ces coûts comprennent notamment la préparation du programme et la copie des données vers et depuis la carte, entre chaque lot de simulations. Ces coûts sont masqués une fois que le nombre de simulations devient suffisamment important, et que le pilote OpenCL peut ainsi effectuer un recouvrement des temps de communication par des calculs pour conserver les unités d'exécution en activité. Cette pénalité à l'exécution sur GPU est davantage marquée dans le cas de l'implémentation optimisée en accès, mais est visible sur les deux courbes. Au-delà de 30 simulations par lancement, le temps

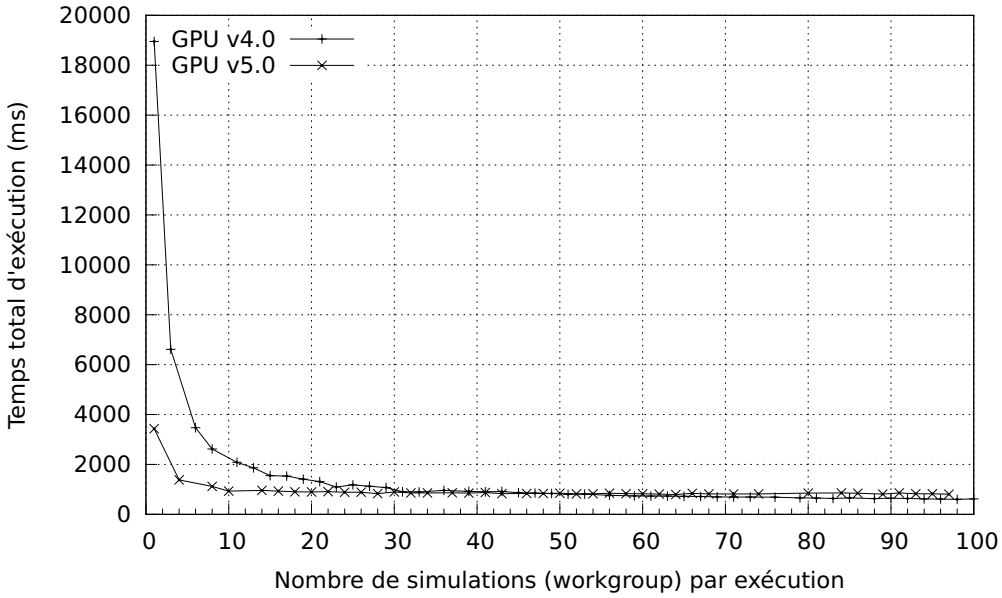

FIGURE 8.15 – Temps d'exécution total pour 1000 simulations MIOR sur plate-forme Tesla C1060, en variant le nombre de simulations exécutées de manière simultanée.

d'exécution total de toutes les simulations stagne, indiquant qu'un remplissage optimal de la carte est atteint pour cette implémentation.

Collemboles

Pour mesurer l'impact de l'utilisation du GPU sur les performances obtenues, nous avons choisi de faire varier le nombre de cellules présentes dans le modèle. Cette mise à l'échelle de l'environnement impose cependant une réflexion particulière, le découpage des parcelles étant assuré sur la base de données géographiques externes, associées à une réalité géographique particulière. Dans ces conditions, l'augmentation du nombre de cellules entraîne une augmentation de la précision du modèle, plutôt qu'un agrandissement de l'espace de simulation.

Le temps d'exécution de 500 itérations du modèle Collemboles est pris en compte pour mesurer l'impact de ce rayon de cellules sur les performances du modèle. Ce temps est comparé sur trois catégories de supports d'exécution :

— Les GPU accessibles au grand public, représentés par les supports Geforce 560Ti et Radeon HD6870.
— Les GPU destinés à une utilisation professionnelle, représentés par les cartes Tesla C1060 et Tesla K20.
— Deux CPU grands publics, pour évaluer les performances pouvant être attendues en l'absence de GPU.

La même implémentation basée sur l'utilisation d'OpenCL est utilisée pour toute les courbes, qui illustrent donc les performances pouvant être obtenues en utilisant tous les coeurs d'exécution disponible sur le matériel.

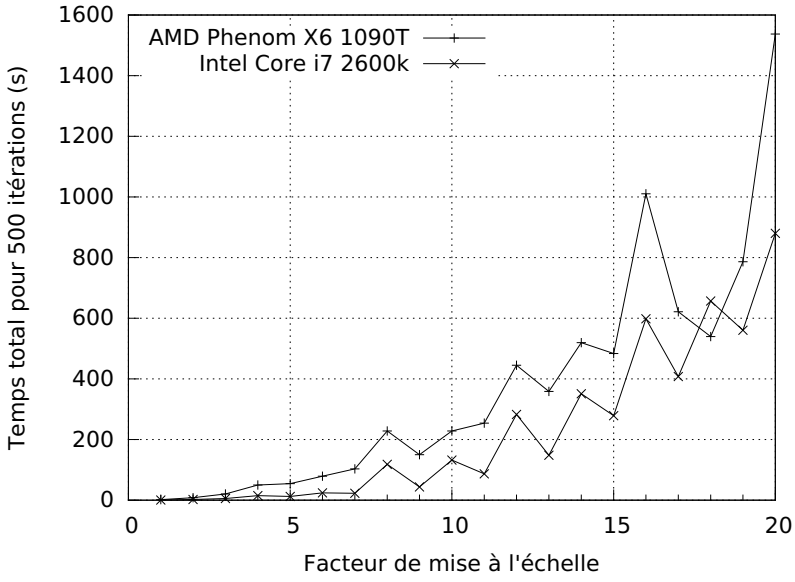

FIGURE 8.16 – Temps d'exécution de 500 itérations du modèle Collembole sur CPU

La Figure 8.16 illustre tout d'abord les performances obtenues en OpenCL sur deux CPU grand public, un AMD Phenom X6 1090T et un Intel Core i7 2600K. Ces courbes mettent en évidence un comportement très irrégulier des performances observées, avec toutefois un avantage de l'ordre de 40% en faveur du processeur Intel.

La Figure 8.17 illustre l'exécution sur deux solutions graphiques grand public, et met en avant l'intérêt de paralléliser la simulation sur GPU, avec des gains de l'ordre d'un facteur 2 par rapport à une exécution sur CPU.

La Figure 8.18 enfin, oppose deux matériels professionnels, une carte Tesla C1060 et une carte

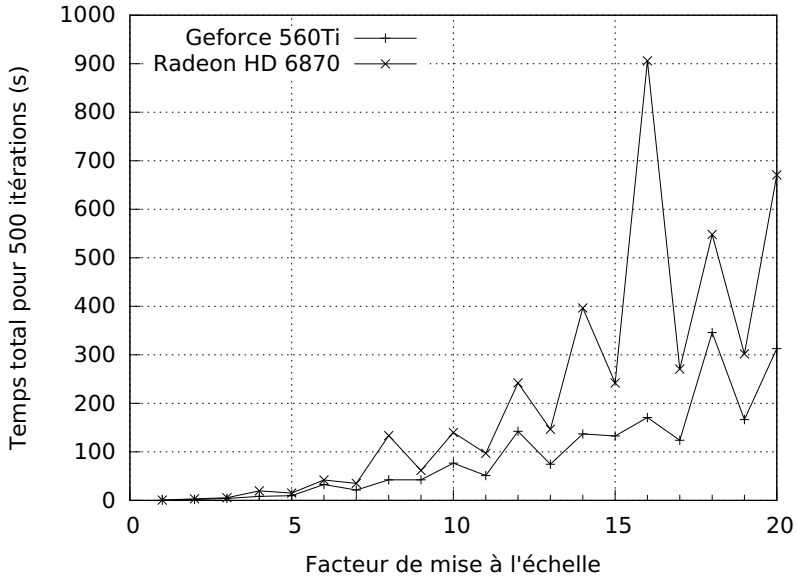

FIGURE 8.17 – Temps d'exécution de 500 itérations du modèle Collembole sur GPU grand public

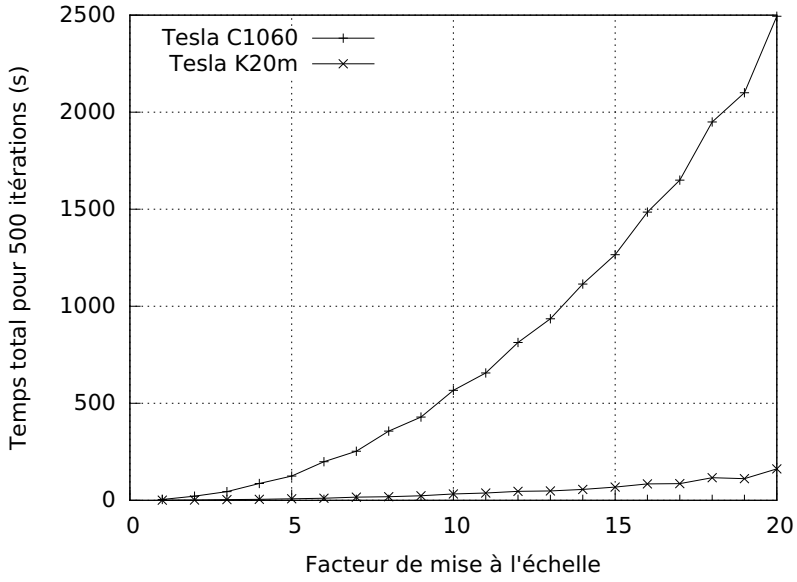

FIGURE 8.18 – Temps d'exécution de 500 itérations du modèle Collemboles sur GPU professionnels

Kepler K20m, et permet de mettre en avant les énormes progrès réalisés en termes de noeuds de calculs GPGPU en quatre années par des mécanismes comme les caches L1 et L2. Les temps obtenus pour la carte Kepler sont sur cette figure proches des résultats sur cartes grand public récentes illustrées par la Figure 8.17, alors que la carte Tesla C1060 se caractérise par des performances jusqu'à quinze fois inférieures à l'échelle 20.

Collemboles est un exemple de modèle multi-agents bénéficiant de manière moins nette d'une parallélisation sur GPU grand public, avec des gains de l'ordre d'un facteur 2 entre l'exécution sur le CPU le plus rapide, le Core i7 2600K, et la Geforce 560Ti à l'échelle 20. Ce gain est toutefois plus marqué dans le cas de la carte graphique professionnelle Tesla K20, qui permet d'obtenir des traitements 4 fois plus rapide à l'échelle 10. La comparaison des cartes graphiques dotées de caches d'une part, et de la Tesla C1060 dénuée de ce mécanisme d'autre part, permet de mettre

en avant l'impact de ce mécanisme dans le cas du modèle Collemboles. Parmi les GPU dotés de ce mécanisme, les performances obtenues sont au contraire proches entre cartes graphiques grand public et matériel professionnel.

8.3 Synthèse

Les résultats obtenus montrent un gain en performance sur GPU par rapport au CPU pour les trois modèles multi-agents adaptés. Ces gains sont particulièrement visibles sur les modèles MIOR et proie-prédateur, où l'utilisation du GPU permet d'obtenir un gain de l'ordre d'un facteur 10 en performance.

Les résultats observés illustrent également l'importance de l'architecture matérielle, et en particulier de la disponibilité ou non de mécanisme de cache L1 et L2, sur les performances observées. Les cartes dotées de caches permettent l'obtention de meilleures performances dans tous les cas. Cet avantage est d'autant plus marqué, dans le cas du modèle MIOR, sur des implémentations ayant recours à de nombreux accès mémoires. Ils montrent l'intérêt du GPU pour déléguer certains traitements normalement effectués sur CPU, en particulier dans le cas du système proie-prédateur.

Dans cette section, nous présentons quelques recommandations d'implémentation au sujet de la parallélisation de traitements sur GPU, sur la base des modèles multi-agents parallélisés et de l'implémentation de la plate-forme MCMAS.

En parallèle aux problématiques de découpage de la représentation et de l'exécution de la simulation sur architecture GPU, certains défis d'implémentations propres à OpenCL s'appliquent aux traitements pouvant être réalisés.

8.3.1 Allocations dynamique de mémoire

Si ce support est présent dans les toutes dernières révisions du modèle de programmation CUDA, OpenCL ne permet pas, au moment de notre rédaction, l'allocation de mémoire depuis le code GPU en cours d'exécution.

Cette limitation est particulièrement problématique pour la manipulation de la structure de données dynamiques sur GPU, dans le cas où la taille des structures n'est pas connue au moment du lancement du traitement.

Dans ce cas, la seule solution est une allocation défensive de la mémoire à priori, basée sur l'hypothèse la plus pessimiste de la taille mémoire requise pour l'exécution du modèle. Elle nécessite un sur-dimensionnement des ressources allouées par rapport aux ressources effectivement consommées, particulièrement dans le cas de modèles de taille importante, et donc une perte de mémoire significative.

La seule alternative pour éviter ce type de perte est une interruption régulière des traitements pour permettre des réallocations sur CPU, avec les coûts associés aux échanges de données et au changement de contexte.

Ces deux approches requièrent des sacrifices soit en mémoire, soit en performance.

8.3.2 Variabilité du support matériel

Un autre défi d'implémentation posé par l'exécution sur GPU est l'existence de multiple générations de plates-formes matérielles, chacune associée à ses propres valeurs limites d'exécution ou support de certaines fonctionnalités.

Des cartes anciennes telles que la plate-forme Tesla C1060 imposent ainsi des contraintes très strictes en termes d'accès mémoires pour permettre une exécution efficace, alors que les CPU ou les cartes graphiques plus modernes minimisent ces contraintes par la présence de logiques de prédictions et de cache gommant ces latences. L'impact de ces attentes est particulièrement visible dans le cas de nombreux accès aléatoires (MIOR) ou de petites tailles (Collemboles) à la mémoire globale GPU sur des cartes dénuées de cache implicite.

Dans un souci de transparence, Nvidia propose la notion de Computing Capabilities (capacité de calcul) pour chacune de ses cartes, indiquant les fonctionnalités CUDA et OpenCL supportées par le matériel, et les limitations associées en termes d'exécution sur ces deux plates-formes :

— Nombre de registres utilisables par work-item et au total.
— Taille maximale de work-group et de grille d'exécution.
— Taille maximale pour chaque type de structure de données.
— Taille maximale totale de chaque espace mémoire.

Ces deux derniers paramètres sont particulièrement importants pour des simulations manipulant de grands buffers de données. Nos expériences ont mis en avant le fait que, si sur les architectures NVIDIA testées, la taille maximale utilisable pour des données en mémoire globale est directement liée à la mémoire physique du périphérique, ce n'est pas le cas sur des implémentations OpenCL proposées par AMD, où cette limite est fixée à une valeur inférieure par l'interface logicielle.

Ces limitations associées au matériel n'ont pas seulement une influence sur la possibilité ou non d'exécuter un traitement OpenCL, mais également sur les performances attendues. Il est alors nécessaire d'adapter l'algorithme ou le découpage de l'exécution utilisé à ces paramètres, pour une efficacité maximale, comme évoqué dans la section suivante.

8.3.3 Adaptation aux paramètres de la plate-forme

Si le standard OpenCL impose à toutes les implémentations le support d'une base commune de primitives et d'opérations, chaque implémentation reste libre, comme nous venons de le voir, de définir ses propres limites au niveau des ressources disponibles.

Le respect du standard permet de garantir le fonctionnement du programme, mais n'assure pas l'obtention automatique des meilleures performances possibles tant sur le nombre limité de coeurs polyvalents offerts par un CPU que sur les centaines de coeurs d'un GPU.

La prise en compte dynamique du type et des limites effectivement offertes par un matériel au moment de l'exécution est donc importante pour permettre une utilisation optimale du matériel disponible [SFSV13].

L'adaptation la plus critique à ce niveau est celle de la taille des paquets d'exécution utilisés, dont la valeur devra être aussi proche que possible, soit du nombre de coeurs effectivement disponibles sur CPU, soit de l'unité de découpage d'exécution ou warp sur GPU.

La taille d'un warp est de 32 threads ou work-items sur la plupart des plates-formes GPU. Les bonnes pratiques OpenCL [Cor12] proposées par NVIDIA recommandent une taille minimale de 64 work-items pour les blocs OpenCL, de manière à permettre un recouvrement des accès

mémoires. Elle recommandent également l'utilisation de blocs de 128 à 256 work-items pour des premières expérimentations, pour ensuite ajuster cette valeur en fonction du taux d'occupation obtenu et des performances obtenues. La société fournit une feuille de calcul permettant d'estimer cette occupation en fonction du modèle de carte utilisé, pour faciliter le choix d'un découpage optimal.

Un exemple de traitement particulièrement impacté par ce choix de découpage est celui de la réduction en parallèle. Si l'utilisation d'un algorithme basé sur plusieurs passes est plus performante sur GPU, il est plus efficace sur CPU d'effectuer un simple découpage du tableau en autant de parties que de coeurs disponibles, et de réaliser l'ensemble de l'opération en un seul lancement.

Pour faciliter cette démarche, MCMAS recommande par défaut une valeur adaptée à l'architecture sous-jacente (CPU, GPU ou autre). Cette valeur est également utilisée pour l'exécution de fonctions de haut niveau par les plugins, en l'absence d'intervention de l'utilisateur. La valeur optimale effective pour ce découpage est cependant très dépendante de l'algorithme exécuté et de l'occupation résultante des ressources matérielles.

8.3.4 Différents espaces mémoires pour différentes utilisations

Comme évoqué dans notre présentation de l'architecture, les cartes graphiques disposent, contrairement aux processeurs traditionnels, de multiples espaces mémoires spécialisés. La sélection de l'espace mémoire dans lequel stocker chaque donnée est effectuée de manière explicite en OpenCL, au moyen de qualificateurs utilisés pour la déclaration de la variable :

— `private int data` : entier en mémoire privée (accessible uniquement au work-item)
— `local int data` : entier en mémoire locale (accessible uniquement au work-group)
— `global int data` : entier en mémoire globale
— `constant int data` : entier en mémoire constante

En l'absence de qualificateur, l'espace privé est utilisé pour le stockage de la variable.

La copie de données entre ces espaces mémoire n'est pas automatique et doit être explicitement effectuée par le programme. La mémoire locale, en particulier, ne peut de plus être initialisée que depuis le périphérique, et requiert donc au moins une copie des données. Elle est généralement employée pour stocker des données intermédiaires souvent utilisées par chaque membre d'un work-group, de manière à éviter son calcul ou sa récupération depuis la mémoire globale à plusieurs reprises.

La copie entre ces types de mémoire n'a pas nécessairement besoin d'être effectuée de manière totalement manuelle : des bibliothèques telles que ELMO [FVSS13] permettent de définir des associations entre structures de données globales et mise en cache en mémoire locale, au moyen de nombreux raccourcis définis sous forme de code OpenCL.

8.3.5 Précision des données et respect des standards

Les cartes graphiques ont initialement été conçues pour effectuer des rendus en deux ou trois dimensions, avant rastérisation sur une grille graphique correspondant au périphérique d'affichage. En termes d'architectures, ce scénario d'utilisation se traduit par une forte optimisation du matériel pour le traitement de nombres flottants, suffisant pour ce type de rendu. D'éventuelles erreurs de précision liées à ce format sont en effet mitigées par deux facteurs propres au rendu graphique : la résolution limitée d'un écran, qui contraint une projection géométrique dans une résolution

ne dépassant pas les quelques milliers de pixels, et le nombre important d'images affichées par seconde, qui estompe toute erreur de rendu ne survenant que dans une image précise.

Cette préférence matérielle pour les flottants est particulièrement marquée, dans le cas des cartes graphiques grand public, par un bridage volontaire des traitements sur des nombres en double précision à une fraction de la fréquence du reste du GPU, pour encourager l'achat de matériel professionnel.

Ces considérations sur les performances sont particulièrement importantes pour des modèles multi-agents, où l'utilisation de nombres double précision est critique pour obtenir des résultats valides. Dans de tels systèmes multi-agents, l'utilisation de réels simple précision peut conduire à des erreurs croissantes, susceptibles de modifier les résultats ou d'empêcher la convergence de la simulation.

Les modèles agents employant souvent des données entières peuvent également être impactés par ce type de disparité matérielle, ce type d'opération étant moins optimisé sur les architectures matérielles les plus anciennes.

En parallèle à ce choix de précision des données se pose la question du mode de calcul souhaité, compatible ou non avec le standard l'IEEE 754. Le non respect strict de ces standards permet de simplifier les calculs dans les cas les plus courants. Il ne garantit cependant plus la prise en compte correcte de valeurs telles que Nan ou l'infini dans les opérations, ou encore des valeurs non normalisées.

Le standard à utiliser pour les calculs flottants peut être configuré pour l'ensemble de l'exécution au moment de la compilation. Dans le cas où de telles opérations devraient être limitées à des portions spécifiques de l'algorithme, OpenCL fournit des versions préfixées des opérations intégrées telles que `fast_sqrt`, potentiellement plus rapide que `sqrt`.

Les compromis liés à cette recherche de performance sont dépendants de chaque plate-forme OpenCL, et peu détaillés par les fabriquants. Dans ces circonstances, l'utilisation du mode standard IEEE et des opérations par défaut est recommandé pour tout nouveau modèle, au moins en attente de validation, de manière à pouvoir évaluer ensuite l'impact de l'utilisation de flottants simple précision ou d'autres modes de calculs sur les résultats de la simulation.

8.3.6 Capacités mémoires physiques

La dernière limitation imposée par l'architecture GPU est la quantité de mémoire offerte par la plate-forme, de l'ordre de 6 à 8 Go sur les solutions graphiques Kepler les plus récentes. Cette quantité demeure plus limitée que sur CPU pour deux raisons :

— Le coût de la mémoire classique pour CPU, et la possibilité pour le chercheur d'étendre aisément la capacité disponible sur sa machine. 4Go de mémoire RAM est un minimum sur de nombreuses machines actuelles, et l'achat de 16 ou 32Go de mémoire est aujourd'hui possible pour quelques centaines d'euros. Au contraire, la mémoire graphique doit être très performante en accès, ce qui augmente son coût, et en adressage, ce qui limite sa taille.

— L'impossibilité d'utiliser, sur GPU, le disque dur ou la mémoire du système pour déléguer de manière transparente le stockage des données supplémentaires. Si l'exploitation de la mémoire du système est facilitée par les dernières révisions des modèles d'exécution GPGPU, il n'existe pas d'analogue à la mémoire d'échange ou au stockage permanent sur cette architecture.

Cette limitation peut poser problème pour des modèles multi-agents de taille importante, particulièrement dans le cas d'utilisation de structures de données surdimensionnées pour compenser l'impossibilité d'effectuer des allocations dynamiques dans un programme OpenCL.

9

Conclusion et perspectives

9.1 Conclusion

Dans ce mémoire, nous avons étudié les manières d'exécuter efficacement des systèmes multi-agents sur cartes graphiques. Cette étude a été l'occasion de mettre en avant l'intérêt des plates-formes parallèles dans le cadre des simulations multi-agents, tant en termes de ressources qu'en termes de performances. Elle nous a également permis de présenter les trois axes de parallélisation possibles pour un système multi-agents, au niveau de l'ordonnanceur, de l'environnement ou des traitements. Cette parallélisation du système prend deux formes principales sur GPU :

— Une adaptation complète de l'exécution sur cette plate-forme, sous forme de nouveau programme.
— Une utilisation du GPU pour ne déléguer qu'une partie de la simulation, et en particulier des traitements coûteux à même d'être parallélisés.

Ces deux approches requièrent une connaissance de la plate-forme GPU. Notre étude bibliographique a montré qu'il n'existait pas de plate-forme générique de parallélisation de systèmes multi-agents permettant un libre choix entre ces deux approches. Notre solution pour permettre l'utilisation la plus large possible de type de matériel est de proposer une nouvelle bibliothèque d'exécution multi-agents, MCMAS.

Notre bibliothèque fournit de nombreux traitements multi-agents de haut niveau prêts à être utilisés sans aucune connaissance en parallélisation. Elle facilite l'ajout de nouveaux traitements au moyen d'une interface de bas niveau fournissant de nombreuses structures de données et facilités d'exécution au développeur. Cette double approche permet une utilisation simple de MCMAS et des cartes graphiques, sans pour autant imposer de cadre de modélisation ou d'exécution particulier, de manière à pouvoir aisément compléter les fonctions offertes par des environnements multi-agents existants.

L'application de ces deux formes de parallélisation sur plusieurs exemples concrets de modèles multi-agents nous a permis de mettre en avant les avantages et les inconvénients associés à chacune de ces approches.

La réalisation de la totalité de la simulation multi-agents permet un contrôle fin de la modélisation et de l'exécution. Elle offre la possibilité d'effectuer la totalité de la simulation en un seul lancement, comme illustré dans le cas du modèle MIOR. Cette flexibilité vient toutefois au prix d'adaptations importantes en termes de données et d'exécution pour tirer pleinement parti de l'architecture GPU. Ces adaptations requièrent une expertise du modèle de programmation et d'exécution pour aboutir à un résultat efficace adapté à l'architecture matérielle. La validation du modèle n'est possible qu'une fois celui-ci fonctionnel sur la nouvelle architecture.

La délégation d'une partie de la simulation permet au contraire un recours plus incrémental au GPU, facilitant des validations intermédiaires du fonctionnement ou des résultats. Elle requiert elle aussi une expertise en parallélisme pour identifier et implémenter les traitements pouvant bénéficier d'une délégation. Cette isolation et cette encapsulation des traitements facilitent leur réutilisation dans d'autres modèles, soit directement, soit dans le cadre de plates-formes multi-agents existantes.

Dans ce mémoire, nous avons apporté une analyse des différentes méthodes de parallélisation de systèmes multi-agents sur GPU, ainsi qu'une illustration de leur utilisation avec MCMAS. Cette illustration nous a permis de détailler ces différentes démarches et leur impact sur les performances obtenues, ainsi que de formuler des recommandations pour la réalisation de calculs sur GPU.

9.2 Perspectives

Si la bibliothèque MCMAS est fonctionnelle et fournit déjà de nombreuses opérations utilisées dans nos trois modèles d'études, elle représente une solution appelée à être améliorée pour répondre à un domaine très dynamique. Au moins trois pistes d'amélioration peuvent à l'heure actuelle être envisagées.

Une première piste d'amélioration consiste à faciliter l'accès à la bibliothèque depuis de nouveaux langages et environnements de simulation agents. De nombreux modèles ont recours à des langages ou à des environnements particuliers pour leur exécution. C'est en particulier le cas du modèle Sworm réalisé avec Madkit pour lequel la simulation MIOR a été conçue. Cet accès peut prendre deux formes, le développement d'une couche d'adaptation permettant l'accès direct à l'interface de MCMAS, ou le développement d'un agent service GPU s'intégrant de manière conceptuelle au modèle. Cet agent est alors à même de répondre à des requêtes d'exécution et de favoriser la réalisation de calculs parallèles de manière transparente pour le reste de la simulation.

Une autre piste, complémentaire, est d'ajouter de nouvelles structures de données et opérations à MCMAS pour répondre aux besoins de nouvelles simulations multi-agents. Si de nombreuses structures de données rencontrées dans les systèmes agents ont déjà été implémentées, certains modèles agents reposent sur des graphes de données encore absents de notre bibliothèque. Des thématiques agents, comme la recherche de chemin, n'ont également pas encore été implémentées. Le développement de nouvelles opérations, et l'enrichissement des opérations existantes, est un élément important pour assurer que MCMAS soit non seulement accessible, mais également utile à de nombreux modèles agents. Le développement de nouveaux traitements agents peut également être facilité par l'intégration de bibliothèques comme ELMO [FVSS13] pour faciliter la gestion de la mémoire partagée en OpenCL.

Une dernière piste d'amélioration concerne la découverte et l'utilisation des ressources matérielles offertes par la machine. Si, à l'heure actuelle, MCMAS permet l'utilisation de plates-formes GPU ou CPU de manière transparente, ces architectures ne sont que quelques représentantes des architectures dites many-core, basées sur de nombreux coeurs d'exécution. Leur exploitation reprend de nombreuses problématiques d'ordonnancement et de décomposition des tâches, et ouvre la voie à l'utilisation de plusieurs matériels locaux. Ces problématiques permettent également d'envisager la mise en place de mécanismes capables d'optimiser le passage des traitements sur les ressources disponibles, de manière à pouvoir simultanément tirer parti du CPU et GPU. De tels mécanismes pourraient dans ce cas introduire une intelligence supplémentaire au niveau de chaque traitement, pour choisir l'algorithme et les types de données les plus adaptés à chaque matériel d'exécution.

Publications

Chapitre de livre

— Guillaume Laville, Christophe Lang, Bénédicte Herrmann, Laurent Philippe, Kamel Mazouzi, and Nicolas Marilleau. **Implementing Multi-Agent Systems on GPU.** In Raphaël Couturier, editor, *Designing Scientific Applications on GPUs*, Numerical Analysis and Scientific Computing, chapter 18, pages 413–439. Chapman and Hall/CRC, 2013.

Conférences

— Guillaume Laville, Christophe Lang, Nicolas Marilleau, Kamel Mazouzi, and Laurent Philippe. **Using GPU for Multi-agent Soil Simulation.** In *PDP 2013, 21st Euromicro International Conference on Parallel, Distributed and Network-based Computing*, Belfast, Ireland, pages 392–399, February 2013. IEEE Computer Society Press.

— Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, Bénédicte Herrmann, and Laurent Philippe. **MCMAS : a toolkit to benefit from many-core architecture in agent-based simulation.** In *PADAPS 2013, 1st Workshop on Parallel and Distributed Agent-Based Simulations, in conjunction with EuroPar 2013*, volume 8374 of LNCS, Aachen, Germany, pages 544–554, August 2013. Springer.

— Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, and Laurent Philippe. **Using GPU for Multi-agent Multi-scale Simulations.** In *DCAI'12, 9-th Int. Conf. on Advances in Intelligent and Soft Computing*, volume 151 of Advances in Intelligent and Soft Computing, Salamanca, Spain, pages 197–204, March 2012. Springer.

Communications

— Présentation d'un poster **"MCSMA : A library for multi-agent simulations on many-core architectures"** au séminaire FEMTO-ST du 26 juin 2013 à la CCI du Doubs.

— Présentation **"Accélération d'une simulation de sol sur GPU"** aux journées Région Grand Est (RGE) du 14 février 2013 à Belfort.

— Présentation **"Portage d'une simulation multi-agents sur GPU"** aux journées Région Grand Est (RGE) du 9 juin 2011 à Metz.

BIBLIOGRAPHIE

[AG13] Nevena Ilieva-Litova Alan Gray, Anders Sjöström. Best Practice mini-guide accelerated clusters. Using General Purpose GPUs. http://www.prace-project.eu/IMG/pdf/Best-Practice-Guide-GPGPU.pdf, 2013. [En ligne ; vérifié le 27 avril 2014].

[amd] AMD Accelerated Parallel Processing Math Libraries (APPML). http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/. [En ligne ; vérifié le 26 avril 2014].

[Amd67] Gene M. Amdahl. 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, 1967. ACM.

[APS10] B. G. Aaby, K. S. Perumalla, and S. K. Seal. 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, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[ASÁ01] Elaini S Angelotti, Edson E Scalabrin, and Bráulio C Ávila. PANDORA : a multi-agent system using paraconsistent logic. In *Computational Intelligence and Multimedia Applications, 2001. ICCIMA 2001. Proceedings. Fourth International Conference on*, pages 352–356. IEEE, 2001.

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Pays-Bas, August 2009.

[BBMC+10] Arnaud Banos, Annabelle Boffet-Mas, Sonia Chardonnel, Christophe Lang, Nicolas Marilleau, and Thomas Thévenin. Simuler la mobilité urbaine quotidienne : le projet MIRO. In Arnaud Banos and Thomas Thévenin, editors, *Mobilités urbaines et risques des transports - approches géographiques*, chapter 2, pages 51–86. Hermès, 2010.

[BCC+11] E. Blanchart, C. Cambier, C. Canape, B. Gaudou, T.-N. Ho, T.-V. Ho, C. Lang, F. Michel, N. Marilleau, and L. Philippe. EPIS : A Grid Platform to Ease and Optimize Multi-agent Simulators Running. In *PAAMS*, volume 88 of *Advances in Intelligent and Soft Computing*, pages 129–134. Springer, 2011.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[BDM04] Jacques M. Bahi, Stéphane Domas, and Kamel Mazouzi. Jace : A Java Environment for Distributed Asynchronous Iterative Computations. In *PDP*, pages 350–357. IEEE Computer Society, 2004.

[BETVG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110(3) :346–359, June 2008.

[BG09] Nathan Bell and Michael Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18 :1–18 :11, New York, NY, USA, 2009. ACM.

[Ble09] A. Bleiweiss. Multi Agent Navigation on the GPU. *GDC09 Game Developers Conference 2009*, 2009.

[BMD+09] E. Blanchart, N. Marilleau, A. Drogoul, E. Perrier, JL. Chotte, and C. Cambier. SWORM : an agent-based model to simulate the effect of earthworms on soil structure. *EJSS. European Journal of Soil Science*, 60 :13–21, 2009.

[BPL+06] Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, Karl-Heinz Krempels, and Peer-Oliver Woelk. A generic time management service for distributed multi-agent systems. *Applied Artificial Intelligence*, 20(2-4) :229–249, 2006.

[Bra00] G. Bradski. *Dr. Dobb's Journal of Software Tools*, 2000.

[BRT11] B. Beresini, S. Ricketts, and M.B. Taylor. Unifying manycore and FPGA processing with the RUSH architecture. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 22–28, June 2011.

[CCC+12] M. Carillo, G. Cordasco, R. De Chiara, F. Raia, V. Scarano, and F. Serrapica. Enhancing the Performances of D-MASON - A Motivating Example. In *SIMULTECH*, pages 137–143, 2012.

[CCDCS11] B Cosenza, G Cordasco, R. De Chiara, and V. Scarano. Distributed Load Balancing for Parallel Agent-based Simulations. In *19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Ayia Napa, Cyprus, 2011.

[CCM+11] Gennaro Cordasco, Rosario Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A Framework for Distributing Agent-Based Simulations. In *Euro-Par 2011 : Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 460–470, 2011.

[CCP07] M. Bousso C. Cambier, D. Masse and E. Perrier. An offer versus demand modelling approach to assess the impact of micro-organisms spatio-temporal dynamics on soil organic matter decomposition rates. *Ecological Modelling*, pages 301–313, 2007.

[CDD+13] Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov. clMAGMA : High Performance Dense Linear Algebra with OpenCL. 2013.

[CDFD10] F Chuffart, N Dumoulin, T Faure, and G Deffuant. SimExplorer : Programming Experimental Designs on Models and Managing Quality of Modelling Process. *IJAEIS*, 1(1) :55–68, 2010.

[CDJM01] Brahim Chaib-Draa, Imed Jarras, and Bernard Moulin. Systèmes multi-agents : principes généraux et applications. *Edition Hermès*, 2001.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CGH+12] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework. In Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefevre, editors, *HPCC-ICESS*, pages 538–545. IEEE Computer Society, 2012.

[CGU+11] José M. Cecilia, José M. Garcia, Manuel Ujaldon, Andy Nisbet, and Martyn Amos. Parallelization Strategies for Ant Colony Optimisation on GPUs. In *IPDPS Workshops*, pages 339–346. IEEE, 2011.

[CKQ+07] Jean-Christophe Castella, Suan Pheng Kam, Dang Dinh Quang, Peter H. Verburg, and Chu Thai Hoanh. Combining top-down and bottom-up modelling approaches of land use/cover change to support public policies : Application to sustainable management of natural resources in northern Vietnam. *Land Use Policy*, 24(3) :531 – 545, 2007. Integrated Assessment of the Land System : The Future of Land Use.

[clp] clpp : OpenCL Data Parallel Primitives Library. https://code.google.com/p/clpp/. [En ligne ; vérifié le 26 avril 2014].

[CN11] Nicholson Collier and Michael North. *Repast HPC : A platform for large-scale agent-based modeling*. Wiley, 2011.

[Cor12] Nvidia Corporation. OpenCL Best Practices Guide. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, 2012. [En ligne ; vérifié le 26 avril 2014].

[cud09] *NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3*. NVIDIA Corporation, 2009.

[DARG12] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. Programming CUDA and OpenCL : A Case Study Using Modern C++ Libraries. *CoRR*, abs/1212.6326, 2012.

[Del13] Audrey Delévacq. *Métaheuristiques pour l'optimisation combinatoire sur processus graphiques (GPU)*. Thèse de doctorat, Université de Reims Champagne-Ardenne, France, February 2013.

[Dem] D. Demidov. VexCL : Vector Expression Template Library for OpenCL. http://www.codeproject.com/Articles/415058/VexCL-Vector-expression-template-library-for-OpenCL. [En ligne ; vérifié le 27 avril 2014].

[DLMK09] Roshan M. D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. 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, 2009. Society for Computer Simulation International.

[DLR07] R. M. D'souza, M. Lysenko, and K. Rahmani. Sugarscape on Steroids : Simulating Over a Million Agents at Interactive Rates. In *Proceedings of the Agent 2007 Conference*, 2007.

[DP93] J.E. Doran and M. Palmer. 'Contrasting models of upper palaeolithic social dynamics : a distributed artificial intelligence approach'. In Aarhus University Press., editor, *J. Andresen, T. Madsen and I. Scollar (Eds.) : Computing the Past - Proceedings of Computer Applications and Quantitative Methods in Archaeology Conference 1992 (CAA 92)*, pages pp.251–262,, 1993.

[DSJD02] Luis T. Da Silva Joao and Y. Demazeau. Vowels co-ordination model. In *AAMAS*, pages 1129–1136, Italy, 2002.

[DvdHD08] Christophe Deissenberg, Sander van der Hoog, and Herbert Dawid. EURACE : A massively parallel agent-based model of the European economy. *Applied Mathematics and Computation*, 204(2) :541 – 552, 2008. Special Issue on New Approaches in Dynamic Optimization to Assessment of Economic and Environmental Systems.

[Fer95] J. Ferber. *Les systèmes multi-agents : vers une intelligence collective*. InterEditions, Paris, 1995.

[fip] FIPA Specifications Published in 1997. http://www.fipa.org/repository/fipa97.html. [En ligne ; vérifié le 26 avril 2014].

[FSN09] L. Fischer, R. Silveira, and L. Nedel. GPU Accelerated Path-Planning for Multi-agents in Virtual Environments. In *Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '09, pages 101–110, Washington, DC, USA, 2009. IEEE Computer Society.

[Fuj03] Richard M Fujimoto. Parallel simulation : distributed simulation systems. In *Proceedings of the 35th conference on Winter simulation : driving innovation*, pages 124–134. Winter Simulation Conference, 2003.

[FVSS13] Jianbin Fang, Ana Lucia Varbanescu, Jie Shen, and Henk Sips. ELMO : A User-Friendly API to Enable Local Memory in OpenCL Kernels. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 0 :375–383, 2013.

[Gar70] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223 :120–123, October 1970.

[GBHS11] Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, 2011.

[GCK+09] S. J. Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming C. Lin, Dinesh Manocha, and Pradeep Dubey. ClearPath : Highly Parallel Collision Avoidance for Multi-Agent Simulation. In *ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON COMPUTER ANIMATION*, pages 177–187. ACM, 2009.

[GF00a] O. Gutknecht and J. Ferber. MadKit : a generic multi-agent platform. In *Proceedings of the fourth international conference on Autonomous agents*, AGENTS '00, pages 78–79, New York, NY, USA, 2000. ACM.

[GF00b] Olivier Gutknecht and Jacques Ferber. The MADKIT Agent Platform Architecture. In *In Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.

[GGDK09] Stéphane Galland, Nicolas Gaud, Jonathan Demange, and Abderrafiaa Koukam. *Environment Model for Multiagent-Based Simulation of 3D Urban Systems*. 2009.

[Gut01] Olivier Gutknecht. *Proposition d'un modèle organisationnel générique de systèmes multi-agents.* PhD thesis, Universite de Montpellier II, Montpellier, France, 2001.

[Hag73] P. Haggett. Analyse spatiale en géographie humaine. *Armand Colin, Paris, 1973.*, 62(1) :125–127, 1973.

[HCS06] Mike Holcombe, Simon Coakley, and Rod Smallwood. A General Framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European Conference on Complex Systems*, 2006.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2) :100–107, July 1968.

[Jac98] Henrik Klinge Jacobsen. Integrating the bottom-up and top-down approach to energy-economy modelling : the case of Denmark. *Energy Economics*, 20(4) :443 – 461, 1998.

[JGLG09] J.-I. Benavides J. Gómez-Luna, J.-M. González-Linares and N. Guil. Parallelization of a Video Segmentation Algorithm on CUDA—Enabled Graphics Processing Units. In *15th Euro-Par Conference*, pages 924–935, Berlin, Heidelberg, 2009. Springer-Verlag.

[JOF03] H. Van Dyke Parunak J. Odell and M. Fleischer. Software engineering for large-scale multi-agent systems. chapter The role of roles in designing effective agent organizations, pages 27–38. Springer-Verlag, Berlin, Heidelberg, 2003.

[Joh12] Haakan Johansson. Volume Raycasting Performance Using DirectCompute. http://hgpu.org/?p=9050, 2012. [En ligne ; vérifié le 26 avril 2014].

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[KRH+10] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough. FLAME : simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 1 - Volume 1*, AAMAS '10, pages 1633–1636, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[KRR10] T. Karmakharm, P. Richmond, and D. Romano. Agent-based Large Scale Simulation of Pedestrians With Adaptive Realistic Navigation Vector Fields. In *Theory and Practice of Computer Graphics (TPCG) 2010*, pages 67–74, 2010.

[KSL+12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL : An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[Mae90] P. Maes. *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and Back*. A Bradford book. MIT Press, 1990.

[MBF02] Fabien Michel, Pierre Bommel, and Jacques Ferber. Simulation distribuée interactive sous MadKit. In *JFSMA*, pages 175–178, 2002.

[McF87] D. McFarland. *The Oxford companion to animal behaviour*. Oxford Paperback Reference. Oxford University Press, 1987.

[MCM12] Longfei Ma, Xue Chen, and Zhouxiang Meng. A performance Analysis of the Game of Life based on parallel algorithm. *CoRR*, abs/1209.4408, 2012.

[Mes09] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard, Version 2.2. Specification, September 2009.

[MFD09] Fabien Michel, Jacques Ferber, and Alexis Drogoul. Multi-Agent Systems and Simulation : a Survey From the Agents Community's Perspective. In Adelinde Uhrmacher Danny Weyns, editor, *Multi-Agent Systems : Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, page 47. CRC Press - Taylor & Francis, May 2009.

[MGR+11] Perhaad Mistry, Chris Gregg, Norman Rubin, David Kaeli, and Kim Hazelwood. Analyzing Program Flow Within a Many-kernel OpenCL Application. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 10 :1–10 :8, New York, NY, USA, 2011. ACM.

[Mic02] Fabien Michel. Introduction to Turtlekit : A Platform for Building Logo Based Multi-Agent Simulations with Madkit. Technical Report 9557, 2002.

[Mic13] Fabien Michel. Intégration du calcul sur GPU dans la plate-forme de simulation multi-agent générique TurtleKit 3. In Salima Hassas and Maxime Morge, editors, *JFSMA*, pages 189–198. Cepadues Editions, 2013.

[net] NetLogo Models Library : Life. http://ccl.northwestern.edu/netlogo/models/Life. [En ligne ; vérifié le 26 avril 2014].

[NSL+11] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[ope] OpenCL 1.2 Reference Pages - Restrictions. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/restrictions.html. [En ligne ; vérifié le 26 avril 2014].

[pan] Pandora : An HPC Agent-Based Modelling framework. https://www.bsc.es/computer-applications/pandora-hpc-agent-based-modelling-framework. [En ligne ; vérifié le 26 avril 2014].

[rep] Repast HPC Manual. repast.sourceforge.net/docs/RepastHPCManual.pdf. [En ligne ; vérifié le 26 avril 2014].

[Rey87] Craig W. Reynolds. Flocks, Herds and Schools : A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[RHK06] Sebastian Rodriguez, Vincent Hilaire, and Abder Koukam. A Holonic Approach to Model and Deploy Large Scale Simulations. In Luis Antunes and Keiki Takadama, editors, *MABS*, volume 4442 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2006.

[Ric11] Paul Richmond. FLAME GPU Technical Report and User Guide (CS-11-03). Technical report, Department of Computer Science, University of Sheffield, 2011.

[RR08] P. Richmond and D. Romano. A High Performance Framework For Agent Based Pedestrian Dynamics on GPU hardware. *European Simulation and Modelling*, 2008.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[RWCR10] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, 2010.

[RWR10] Karl Rupp, Josef Weinbub, and Florian Rudolf. Automatic Performance Optimization in ViennaCL for GPUs. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 6 :1–6 :6, New York, NY, USA, 2010. ACM.

[SFF+10] Renato Silveira, Leonardo Fischer, José Antônio Salini Ferreira, Edson Prestes, and Luciana Nedel. Path-planning for RTS games based on potential fields. In *Proceedings of the Third international conference on Motion in games*, MIG'10, pages 410–421, Berlin, Heidelberg, 2010. Springer-Verlag.

[SFS10] Antoine Spicher, Nazim A. Fatès, and Olivier Simonin. Translating Discrete Multi-Agents Models into Cellular Automata, Application to Diffusion-Limited Aggregation. *CCIS 67 Communications in Computer and Information Sciences series*, 67 :270–282, January 2010.

[SFSV13] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance Traps in OpenCL for CPUs. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 38–45, Washington, DC, USA, 2013. IEEE Computer Society.

[Skl11] E. Sklar. NetLogo, a multi-agent simulation environment. *Artificial Life*, 13(3) :303–311, 2011.

[SMH+10] Aamir Shafi, Jawad Manzoor, Kamran Hameed, Bryan Carpenter, and Mark Baker. Multicore-enabling the MPJ Express Messaging Library. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 49–58, New York, NY, USA, 2010. ACM.

[SN09] D. Strippgen and K. Nagel. Multi-agent traffic simulation with CUDA. *2009 International Conference on High Performance Computing Simulation*, pages 106–114, 2009.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. Task Management for Irregular-parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[UIN12] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. *2013 International Conference on Computing, Networking and Communications (ICNC)*, 0 :94–102, 2012.

[VQC02] G. Vitaglione, F. Quarta, and E. Cortese. Scalability and Performance of JADE Message Transport System. http://jade.tilab.com/papers/Final-ScalPerfMessJADE.pdf?, 2002. [En ligne ; vérifié le 27 avril 2014].

[WD92] Eric Werner and Yves Demazeau. The design of multi-agent systems. *Decentralized AI*, 3 :3–30, 1992.

[Wei13] Robin M. Weiss. Accelerating Swarm Intelligence Algorithms with GPU-Computing. In David A. Yuen, Long Wang, Xuebin Chi, Lennart Johnsson, Wei Ge, and Yaolin Shi, editors, *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, pages 503–515. Springer Berlin Heidelberg, 2013.

[Wil74] A.G. Wilson. *Urban and Regional Models in Geography and Planning*. A Wiley-Interscience publication. John Wiley & Sons Incorporated, 1974.

[WLL+] John Wawrzynek, Mingjie Lin, Ilia Lebedev, Shaoyi Cheng, and Daniel Burke. 1 Rethinking FPGA Computing with a Many-Core Approach.

[WRC12] Peter Wittek and Xavier Rubio-Campillo. Scalable agent-based modelling with cloud HPC resources for social simulations. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 355–362. IEEE, 2012.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC : First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[ZG12] L. Zaoralek and P. Gajdos. CUDA code support in multiagent platform JADE. In *Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on*, pages 546–551, Nov 2012.

Résumé :

Ces dernières années ont consacré l'émergence du parallélisme dans la plupart des branches de l'informatique. Au niveau matériel, tout d'abord, du fait de la stagnation des fréquences de fonctionnement des unités de calcul. Au niveau logiciel, ensuite, avec la popularisation de nombreuses plates-formes d'exécution parallèle. Une forme de parallélisme est également présente dans les systèmes multi-agents, qui facilitent la description de systèmes complexes comme ensemble d'entités en interaction. Si l'adéquation entre ce parallélisme d'exécution logiciel et conceptuel semble naturelle, la parallélisation reste une démarche difficile, du fait des nombreuses adaptations devant être effectuées et des dépendances présentes explicitement dans de très nombreux systèmes multi-agents.
Dans cette thèse, nous proposons une solution pour faciliter l'implémentation de ces modèles sur une plate-forme d'exécution parallèle telle que le GPU. Notre bibliothèque MCMAS vient répondre à cette problématique au moyen de deux interfaces de programmation, une couche de bas niveau MCM permettant l'accès direct à OpenCL et un ensemble de plugins utilisables sans connaissances GPU. Nous étudions ensuite l'utilisation de cette bibliothèque sur trois systèmes multi-agents existants : le modèle proie-prédateur, le modèle MIOR et le modèle Collemboles. Pour montrer l'intérêt de cette approche, nous présentons une étude de performance de chacun de ces modèles et une analyse des facteurs contribuant à une exécution efficace sur GPU. Nous dressons enfin un bilan du travail et des réflexions présentées dans notre mémoire, avant d'évoquer quelques pistes d'amélioration possibles de notre solution.

Mots-clés : Framework de simulation, Système multi-agents, Many-core, GPU, Calcul haute performance

Abstract:

These last years have seen the emergence of parallelism in many fields of computer science. This is explained by the stagnation of the frequency of execution units at the hardware level and by the increasing usage of parallel platforms at the software level. A form of parallelism is present in multi-agent systems, that facilitate the description of complex systems as a collection of interacting entities. If the similarity between this software and this logical parallelism seems obvious, the parallelization process remains difficult in this case because of the numerous dependencies encountered in many multi-agent systems.
In this thesis, we propose a common solution to facilitate the adaptation of these models on a parallel platform such as GPUs. Our library, MCMAS, provides access to two programming interface to facilitate this adaptation: a low-level layer providing direct access to OpenCL, MCM, and a high-level set of plugins not requiring any GPU-related knowledge. We study the usage of this library on three existing multi-agent models : predator-prey, MIOR and Collembola. To prove the interest of the approach we present a performance study for each model and an analysis of the various factors contributing to an efficient execution on GPUs. We finally conclude on a overview of the work and results presented in the report and suggest future directions to enhance our solution.

Keywords: Simulation framework, Multi-agents system, Many-core, GPU, High-performance computing

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ