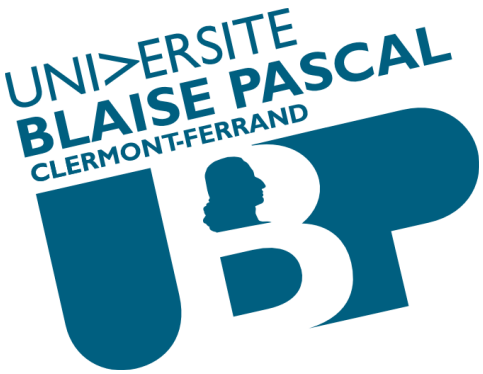
Parallélisation et optimisation d'un simulateur de morphogénèse d'organes. Application aux éléments du rein

Jonathan Caux

N° d'ordre D. U. : 2299
EDSPIC : 587



THESE

pour obtenir le grade de

Docteur de L'université Blaise Pascal
École Doctorale des Sciences pour l'Ingénieur

Discipline : Informatique

présentée par

Jonathan Caux

Parallélisation et optimisation d'un simulateur de morphogénèse d'organes
Application aux éléments du rein

---

sous la direction du professeur

David R.C. Hill

Soutenue publiquement le 30/10/2012

Rapporteurs :	Marc Daumas, Professeur, PROMES, Perpignan
	Stéphane Vialle, Professeur, SUPELEC, Metz
Examinateurs :	Alexandre Muzy, Docteur, LISA, Corte
	Pridi Siregar, Docteur, Integrative BioComputing, Rennes
	Lydia Maigne, Maître de Conférences, LPC, Clermont-Fd
Directeur :	David R.C. Hill, Professeur, LIMOS, Clermont-Fd
Invité :	Nathalie Julen, Docteur, Integrative BioComputing, Rennes

Résumé

Depuis plusieurs dizaines d'années, la modélisation du vivant est un enjeu majeur qui nécessite de plus en plus de travaux dans le domaine de la simulation. En effet, elle ouvre la porte à toute une palette d'applications : l'aide à la décision en environnement et en écologie, l'aide à l'enseignement, l'aide à la décision pour les médecins, l'aide à la recherche de nouveaux traitements pharmaceutiques et la biologie dite « prédictive », etc. Avant de pouvoir aborder un problème, il est nécessaire de pouvoir modéliser de façon précise le système biologique concerné en précisant bien les questions auxquelles devra répondre le modèle. La manipulation et l'étude de systèmes complexes, les systèmes biologiques en étant l'archétype, pose, de façon générale, des problèmes de modélisation et de simulation. C'est dans ce contexte que la société Integrative BioComputing (IBC) développe depuis le début des années 2000 un prototype d'une Plateforme Générique de Modélisation et de Simulation (la PGMS) dont le but est de fournir un environnement pour modéliser et simuler plus simplement les processus et les fonctions biologiques d'un organisme complet avec les organes le composant.

La PGMS étant une plateforme générique encore en phase de développement, elle ne possédait pas les performances nécessaires pour permettre de réaliser la modélisation et la simulation d'éléments importants dans des temps suffisamment courts. Il a donc été décidé, afin d'améliorer drastiquement les performances de la PGMS, de paralléliser et d'optimiser l'implémentation de celle-ci ; le but étant de permettre la modélisation et la simulation d'organes complets dans des temps acceptables.

Le travail réalisé au cours de cette thèse a donc consisté à traiter différents aspects de la modélisation et de la simulation de systèmes biologiques afin d'accélérer les traitements de ceux-ci. Le traitement le plus gourmand en termes de temps de calcul lors de l'exécution de la PGMS, le calcul des champs physicochimiques, a ainsi fait l'objet d'une étude de faisabilité de sa parallélisation. Parmi les différentes architectures disponibles pour paralléliser une telle application, notre choix s'est porté sur l'utilisation de GPU (Graphical Processing Unit) à des fins de calculs généralistes aussi couramment appelé GPGPU (General-Purpose computation on Graphics Processing Units). Ce choix a été réalisé du fait, entre autres, du coût réduit du matériel et de sa très grande puissance de calcul brute qui en fait une des architectures de parallélisation les plus accessibles du marché. Les résultats de l'étude de faisabilité étant particulièrement concluant, la parallélisation du calcul des champs a ensuite été intégrée à la PGMS. En parallèle, nous avons également mené des travaux d'optimisations pour améliorer les performances séquentielles de la PGMS. Le résultat de ces travaux est une augmentation de la vitesse d'exécution d'un facteur 18,12x sur les simulations les plus longues (passant de 16 minutes pour la simulation non optimisée utilisant un seul cœur CPU à 53 secondes pour la version optimisée utilisant toujours un seul cœur CPU mais aussi un GPU GTX500). L'autre aspect majeur traité dans ces travaux a été d'améliorer les performances algorithmiques pour la simulation d'automates cellulaires en trois dimensions. En effet, ces derniers permettent aussi bien de simuler des comportements biologiques que d'implémenter des mécanismes de modélisation tels que les interactions multi-échelles. Le travail de recherche s'est essentiellement effectué sur des propositions algorithmiques originales afin d'améliorer les simulations réalisées par IBC sur la PGMS. L'accélération logicielle, à travers l'implémentation de l'algorithme Hash-Life en trois dimensions, et la parallélisation à l'aide de GPGPU ont été étudiées de façon concomitante et ont abouti à des gains très significatifs en temps de calcul.

Remerciements

Avant toute chose, je tiens à remercier tous ceux sans qui ce travail n'aurait jamais pu être mené à bien :

En premier lieu, **David Hill**, mon directeur de thèse, pour son suivi et sa disponibilité dont j'ai pu disposer durant ces trois ans passés à ses côtés.

Pridi Siregar et Nathalie Julen pour m'avoir permis de réaliser cette thèse dans leur entreprise dans le cadre d'un dispositif CIFRE, **Alexandre Muzy** plus particulièrement pour la participation à l'encadrement ainsi que tous mes collègues.

L'ensemble des membres du laboratoire au cœur duquel j'ai effectué ma thèse, le **LIMOS**, pour m'avoir accueilli chaleureusement mais aussi pour toutes les interactions que j'ai pu avoir avec eux.

Toutes les personnes qui m'ont permis de m'aérer l'esprit tous les jours à 16h tapante : **Jonathan, Baraa, Mohieddine, Nathalie, Guillaume, Faouzi et Romain** pour ne citer qu'eux.

Philippe Mahey pour m'avoir permis de dispenser des cours dans son établissement. Je remercie également **Vincent Barra, Christophe Duhamel, Loïc Yon, Bruno Bachelet** et encore une fois **David Hill** sans qui je n'aurais pu dispenser ces enseignements.

Les membres de l'administration, **Béatrice Bourdieu, Françoise Toledo et Isabelle Villard** en tête, qui m'ont beaucoup aidé dans mes tâches administratives.

Corinne Peymaud et Susan Arbon pour leur sympathie et leur bonne humeur communicatrice.

Toute ma famille et tous mes amis qui m'ont soutenu et conseillé durant ces trois ans. Un remerciement particulier à trois d'entre elles : **ma mère, Sylvie et Thérèse** pour avoir pris le temps de relire mon manuscrit.

Je tiens tout particulièrement à remercier mon collègue de bureau et ami **Luc Touraille**, qui ne m'a jamais fait défaut, dans les hauts comme dans les bas.

Je tiens enfin à remercier l'ensemble **des membres de mon jury** qui m'ont fait l'honneur de bien vouloir y participer.

Sommaire

Table des figures

Table des tableaux

Table des codes

Introduction générale

Introduction générale

Depuis plusieurs dizaines d'années, la modélisation du vivant est un enjeu majeur du domaine de la simulation. En effet, la modélisation du vivant ouvre la porte à toute une palette d'applications :

- L'aide à l'enseignement à travers l'utilisation de patients virtuels pour les élèves de médecine par exemple.
- L'aide à la décision lors de traitements comme lors du dosage personnalisé de celui-ci à la physiologie d'un patient.
- L'aide à la recherche de nouveaux remèdes pour les sociétés pharmaceutiques.
- Etc.

Mais avant de pouvoir traiter ces problèmes, il est donc nécessaire de pouvoir modéliser de façon précise les systèmes biologiques. Or la complexité du vivant est particulièrement importante. Ainsi, le fonctionnement de l'organisme humain est contrôlé par un nombre de gènes de l'ordre de 30 000. Chacun de ces gènes permet la fabrication de protéines qui vont interagir entre elles en fonction de leur structure. Au niveau cellulaire, ces protéines vont permettre l'apparition de mécanismes complexes tels que les voies métaboliques. Enfin, à des niveaux supérieurs, ces cellules permettent la modélisation des organes et des différents systèmes biologiques.

Tous ces mécanismes font des systèmes biologiques des systèmes complexes qu'il est impossible de modéliser et de simuler en prenant en compte l'ensemble des caractéristiques existantes (la modélisation d'un organe ne peut être réalisée en prenant en compte l'expression des gènes dans chaque cellule de celui-ci) (White, Peng et Demir 2009). Une simulation multi-échelle est alors souvent nécessaire pour permettre de prendre en compte des interactions fines au sein de la simulation de comportements plus généraux. Mais même dans ce cas, la modélisation et la simulation des systèmes vivants peuvent être longues et ardues à mettre en œuvre.

C'est la raison pour laquelle, au début des années 2000, la société Integrative BioComputing a développé le prototype d'une Plateforme Générique de Modélisation et de Simulation : la PGMS (P. Siregar 2009). Le but de cette plateforme est de fournir un environnement pour modéliser et simuler les processus et les fonctions biologiques d'un organisme. La PGMS est

donc un outil qui a pour but de faciliter et d'accélérer la modélisation et la simulation du vivant dans le cadre de la recherche et de l'enseignement.

Mais la PGMS étant une plateforme générique encore en phase de développement, elle ne possédait pas en début de thèse les performances nécessaires pour permettre de réaliser la modélisation et la simulation d'éléments importants en des temps suffisamment courts sur des machines de table. Ainsi, la simulation d'un néphron du rein nécessitait plus d'une minute trente ; alors même qu'un rein complet comprend plusieurs centaines de milliers de ces néphrons. Il a donc été décidé, afin d'améliorer drastiquement les performances de la PGMS, de paralléliser et d'optimiser l'implémentation de celle-ci afin de pouvoir envisager la modélisation et la simulation d'organes complets en des temps acceptables.

Afin d'améliorer les performances de la simulation, plusieurs pistes ont été suivies de façon concurrentes. Paradoxalement, la première piste n'a pas eu pour but d'optimiser ou de paralléliser l'application mais a consisté à traiter l'amélioration du code de l'application. En effet, les principaux efforts de développement s'étaient jusque-là concentrés sur la création d'une application la plus complète possible (plus de 200 000 lignes de C++), en limitant les travaux de modifications du code permettant d'améliorer l'implémentation de celle-ci. Ceci a conduit, d'ajout de fonctionnalités en ajout de fonctionnalités, à un code quelques fois difficile à appréhender pour une personne extérieure au projet, en particulier du fait de l'utilisation d'outils obsolètes. Or le bon sens et de nombreuses études montrent que plus le code d'une application est de bonne qualité, facile à lire et à comprendre, plus le nombre d'erreurs qu'il contient est faible et plus il est facile de le faire évoluer (Basili, Briand et Melo 1996).

Au-delà du problème de lisibilité, plusieurs points importants de la PGMS ont été choisis pour être traités de façon individuelle. Ainsi, le calcul des champs physicochimiques ayant un poids très important dans la PGMS actuelle, celui-ci a fait l'objet d'une attention toute particulière. En effet, la valeur de chaque champ doit être en permanence remise à jour en fonction de l'évolution de la simulation. Couplé au fait que ce calcul est réalisé pour chaque cellule d'un espace discrétisé, il en résulte que celui-ci représentait, après étude et « profilage », près de 95% du temps d'exécution total de la simulation. Il s'agissait donc bien d'un point d'amélioration crucial de la simulation. Le second point particulièrement étudié

traite des automates cellulaires. Si ceux-ci ne sont pas encore présents au sein de la PGMS, leur utilisation future pour résoudre certains problèmes de la modélisation multi-échelle devrait les amener à jouer un rôle central. Il est donc nécessaire que la simulation de ceux-ci soit réalisée de la manière la plus efficace possible afin qu'un minimum de temps soit accaparé par celle-ci.

Pour améliorer ces points précis mais aussi les autres goulots d'étranglement de l'application, plusieurs axes ont été privilégiés. Le premier, et le plus important, est la parallélisation de l'application. Le but était ici d'améliorer les performances d'une exécution de l'application et pas de paralléliser les réplications comme cela peut se faire pour les simulations de Monte-Carlo (Reuillon, et al. 2006). La tâche est donc bien différente et certaines architectures parallèles telles que les grilles de calcul sont moins indiquées pour ce genre de travail. Les solutions restent malgré tout nombreuses, il est ainsi possible de s'appuyer sur des fermes de calcul ou, à plus petite échelle, sur des SMP (Symmetric Multi-Processing), des FPGA (Field-Programmable Gate Array) ou encore des GPU (Graphical Processing Unit).

Le second axe d'approche a consisté à chercher à améliorer les performances de l'application séquentielle. Pour cela, la solution la plus efficace, et souvent la plus simple, est de remplacer un algorithme existant par un algorithme plus performant permettant de réaliser la même opération. Bien évidemment, il n'est pas toujours possible de trouver de meilleurs algorithmes. Dans ce cas, il est encore souvent possible d'améliorer les performances de l'algorithme en travaillant sur son implémentation. Il existe en effet différentes techniques permettant d'améliorer les performances d'une application (Kennedy et Allen 2001). Lorsque celles-ci sont mises en pratique sur les parties du code les plus souvent appelées, une modification qui peut sembler limitée peut avoir un impact important sur les performances.

L'objectif principal du travail a donc été de proposer, d'une part, des solutions afin d'améliorer la vitesse d'exécution des simulations réalisées à l'aide de la PGMS, et d'autre part, de réaliser un travail prospectif d'amélioration des performances sur différents problèmes pouvant être plus tard intégrés à la PGMS. Pour cela, le travail a consisté :

- À reprendre dans un premier temps le code de la PGMS pour le standardiser, en le débarrassant de dépendances obsolètes.
- À étudier l'amélioration des performances de différents mécanismes, tels que le calcul de champs physicochimiques et la simulation d'automates cellulaires, en implémentant de nouveaux algorithmes ou en utilisant des architectures parallèles.
- À intégrer les travaux théoriques de parallélisation du calcul des champs physicochimiques à la nouvelle version de la PGMS afin de diminuer le temps d'exécution des simulations.
- À optimiser le code de la PGMS afin d'améliorer encore les performances de l'application.

Ce manuscrit expose dans un premier chapitre les problématiques de la modélisation de systèmes complexes à l'aide de la modélisation multi-échelle ainsi que la solution proposée par la PGMS qui fut utilisée tout au long de cette thèse et enfin le contexte originale de l'application ainsi que les différents champs d'amélioration de celle-ci, que ce soit en génie logiciel ou en optimisation.

Le second chapitre introduit les solutions existantes pour réaliser du calcul à haute performance en introduisant les différentes architectures parallèles disponibles ainsi que la simulation d'automate cellulaire et une optimisation spécifique au problème de la simulation de ceux-ci à travers l'algorithme Hash-Life.

Le troisième chapitre propose une explication à visée pédagogique de l'API retenue pour la parallélisation sur GPU de l'application : l'API CUDA (Compute Unified Device Architecture) proposée par NVIDIA.

Le quatrième chapitre présente l'implémentation de solutions logicielles et matérielles aux problèmes de la simulation des automates cellulaires et aux calculs des champs physicochimiques. Les solutions logicielles porteront sur l'implémentation de l'algorithme Hash-Life en trois dimensions et les solutions matérielles sur la parallélisation d'algorithmes sur GPU.

Le cinquième et dernier chapitre présente l'intégration à la PGMS de différentes solutions permettant son optimisation et la parallélisation d'une partie de son exécution. Pour cela, le

chapitre commence par présenter l'état initial de la PGMS et les travaux de refactorisation nécessaires en amont de tout travail d'amélioration des performances, puis les travaux de parallélisation et d'optimisation proprement dit.

Enfin, le bilan et les perspectives sont proposés dans la conclusion générale.

Modélisation et implémentation des systèmes biologiques

1 Modélisation et implémentation des systèmes biologiques

Introduction

Parmi les différents champs d'étude en modélisation, l'étude des systèmes complexes fait partie de ceux qui ont pris un essor particulièrement important ces quinze dernières années. La Figure 1-1 nous donne une idée de l'évolution du nombre de publications dans le domaine des systèmes complexes (source : Microsoft Academic Research). Un système complexe est un système faisant intervenir un grand nombre d'entités mais qui possède de plus une caractéristique importante : de fortes interactions entre ces entités. Ces interactions font qu'il n'est pas possible de prévoir à priori le résultat de l'évolution d'un tel système même en connaissant précisément le comportement de chaque entité. Pour connaître son comportement, il est ainsi nécessaire de simuler le comportement du système dans son ensemble (ou d'expérimenter son comportement lorsque c'est possible) (Hassas et Nautibus 2005).

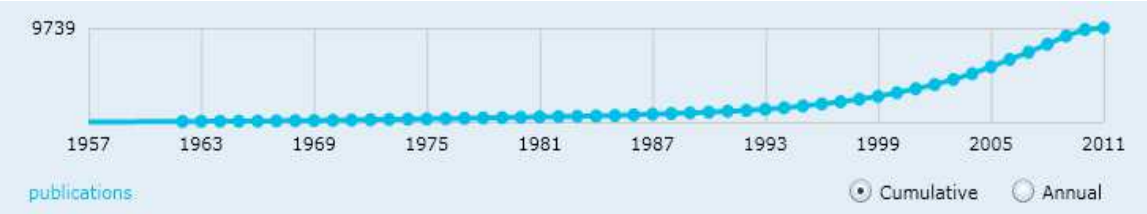


Figure 1-1 : Évolution du nombre de publications dans le domaine des systèmes complexes (source : Microsoft Academic Research)

Des systèmes complexes se retrouvent dans un grand nombre de domaines (Bar-Yam 2003) :

- En finance, la bourse, par exemple, évolue en fonction de multiples facteurs, aussi bien les transactions effectuées par les courtiers que les performances des entreprises ou l'actualité géopolitique (Lillo et Mantegna 2003).
- En sociologie, l'étude, par exemple, des comportements au sein d'une entreprise doit prendre en compte les interactions entre les individus mais également les interactions avec l'environnement (personnes extérieures…) (Sawyer 2005).
- En éthologie, l'étude, par exemple, d'une colonie de fourmis nécessite d'étudier les interactions entre l'ensemble des fourmis à l'aide de phéromones ainsi que celles réalisées avec l'environnement (Boccara 2004).

La discipline la plus encline à l'étude des systèmes complexes est sans doute la biologie. En effet, dès que l'on sort du cadre de l'étude des organismes unicellulaires ou pluricellulaires simples, les mécanismes les plus importants mettent en œuvre un grand nombre d'interactions : entre les organes au plus haut niveau mais aussi entre les cellules (à l'aide de messages chimiques et d'interactions mécaniques entre autres) et à l'intérieur même des cellules entre leurs différents éléments (Southern, et al. 2008).

Ces systèmes comprenant d'importantes interactions, leurs modélisations informatiques sont souvent difficiles à mettre en œuvre. Ainsi, lorsque les interactions se situent à différents niveaux de modélisation, comme ce peut être le cas lors de la modélisation de processus biologiques faisant intervenir par exemple le niveau cellulaire et le niveau intracellulaire, une modélisation informatique particulière permettant de représenter ces interactions est nécessaire. On parle dans ce cas de modélisation multi-échelle. Cette technique consiste à modéliser autant qu'il est possible les différents niveaux d'un problème séparément en n'oubliant pas régulièrement de faire évoluer par simulation chaque niveau de la modélisation en fonction de l'évolution des niveaux supérieurs et inférieurs.

Ce chapitre présente dans une première partie les différentes problématiques de la modélisation multi-échelle dans le cadre de la biologie. Puis, une seconde partie introduit la Plateforme Générique de Modélisation et de Simulation (PGMS) qui est la plateforme de simulation multi-échelle utilisée tout au long des travaux de cette thèse. Enfin sont présentés le contexte original du code de l'application, les différents outils logiciels utilisés pour améliorer, de manière fiable, celui-ci ainsi que différentes techniques d'optimisation logicielle permettant d'accélérer l'exécution d'une application.

1.1 Modélisation multi-échelle de mécanismes biologiques

Depuis le début des années 2000, les évolutions en microbiologie ainsi que l'augmentation des capacités informatiques et techniques ont permis une augmentation des données biologiques recueillies au niveau microscopique. À partir de ces données, les biologistes ont pu s'intéresser à des systèmes et à des processus biologiques, d'une part de plus en plus fins, et d'autre part de plus en plus généraux, pour résoudre un problème donné (White, Peng et Demir 2009).

Mais l'expansion du domaine d'étude n'est pas sans problème. Ainsi, les nouveaux systèmes et processus étudiés peuvent opérer en même temps à plusieurs échelles de tailles ou de temps différents et devenir alors des systèmes très complexes. Le problème majeur est dans ce cas de réussir à faire communiquer les différentes échelles entre elles, bien qu'elles puissent être très différentes. C'est le cas par exemple des échelles biologiques où l'on peut aller de l'atome à la population, en passant par les molécules, les cellules, les tissus, les organes, l'organisme et par bien d'autres niveaux intermédiaires (Southern, et al. 2008).

1.1.1 Modélisation de processus microscopiques

1.1.1.1 Modélisation multi-niveau

La modélisation multi-échelle est donc utilisée en biologie intégrative afin d'harmoniser les différents mécanismes entrant en jeu dans un processus biologique donné. Le cas le plus courant d'utilisation est la modélisation d'un comportement mettant en jeux des facteurs présents à différentes échelles. Ainsi, pour permettre de prendre en compte l'ensemble de ces paramètres, et donc de disposer d'une modélisation la plus fidèle possible, il est nécessaire de modéliser de façon interdépendante chacun de ces niveaux.

C'est le cas pour la simulation de la croissance de nouveaux vaisseaux sanguins (Qutub, et al. 2009). En effet, la création de nouveaux vaisseaux est fonction d'un grand nombre de paramètres présents à différents niveaux :

- Au niveau moléculaire, l'expression des gènes, la synthèse d'enzymes ou encore la présence d'oxygène participent fortement à l'ensemble du processus.
- Au niveau cellulaire, les liaisons ligands-récepteurs peuvent induire une modification de la densité et de la capillarité d'une cellule, modifiant ainsi la perception de son environnement et donc de ses interactions.
- Et, bien évidemment, au niveau des tissus, leurs compositions et leurs activités paracrines sont des facteurs déterminants dans l'évolution du réseau de vaisseaux sanguins.

La modélisation a donc lieu sur trois niveaux permettant de représenter l'ensemble des mécanismes mis en jeux dans cette simulation (voir Figure 1-2).

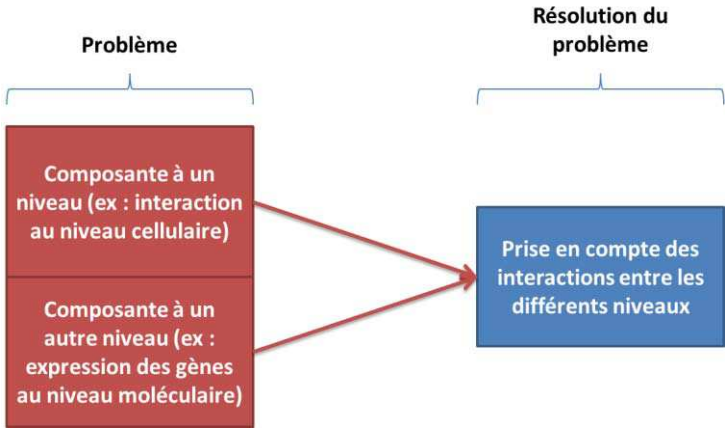


Figure 1-2 : Représentation de la modélisation multi-échelle spatialisée.

En prenant l'exemple de la modélisation du comportement des myocytes ventriculaires (fibres musculaires des ventricules) chez les rongeurs, (Lu, et al. 2009) fait également interagir des éléments à différents niveaux. En effet, pour bien appréhender le fonctionnement de ces cellules, il est nécessaire d'étudier les échanges d'ions entre le milieu intracellulaire et le milieu extracellulaire qui sont la cause de la polarisation et de la dépolarisation des cellules. La modélisation comprend donc deux niveaux bien distincts : la cellule (ou plus exactement la membrane de celle-ci par laquelle vont transiter les ions de l'extérieur de la cellule vers l'intérieur et inversement) et les ions proprement dits (aussi bien dans la cellule qu'à l'extérieur de celle-ci).

Les travaux de (Beyer et Meyer-Hermann 2009) et (Kapellos, Alexiou et Payatakes 2007) entrent également dans cette catégorie. Beyer présente en effet la modélisation des mécanismes cellulaires liés à l'organisation des tissus ayant lieu à deux niveaux différents. Le premier étant le niveau cellulaire auquel ont lieu les changements internes des cellules (propriété de surface par exemple). Le second est le niveau moléculaire auquel vont évoluer des molécules sécrétées par les différentes cellules pour influencer à distance l'évolution d'autres cellules. Kapellos traite quant à lui de la diffusion dans une membrane cellulaire. Il commence par une première évaluation au niveau de la membrane avant de descendre à un niveau plus fin afin de prendre en compte la structure hétérogène de l'espace extracellulaire ainsi que les propriétés des polymères extracellulaires.

1.1.1.2 Modélisation multi-résolution

La modélisation multi-échelle peut également être utilisée pour modéliser de manière différente un mécanisme selon la finesse recherchée.

Cette technique est utilisée par exemple dans (Pivkin, Richardson et Karniadakis 2009) pour l'étude de l'effet des globules rouges sur la coagulation du sang. En effet, selon la taille et l'intérêt des vaisseaux étudiés, la circulation du sang peut être représentée à l'aide d'un modèle 1D assez sommaire pour les éléments les moins intéressants jusqu'à une représentation 3D fine pour les zones les plus intéressantes (voir Figure 1-3).

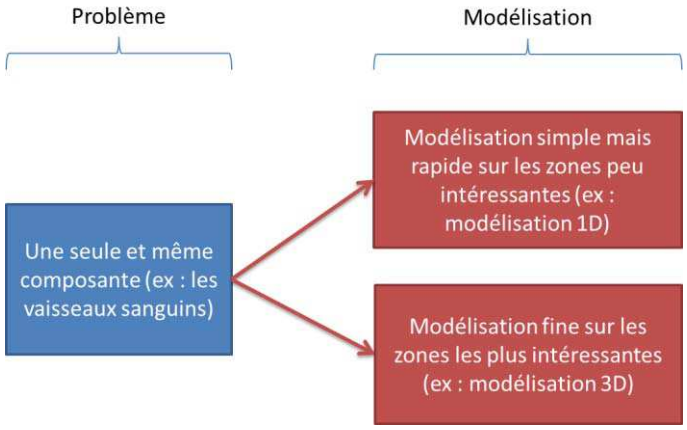


Figure 1-3 : Représentation de la modélisation multi-échelle liée à la finesse recherchée

Dans le même esprit, de façon à pouvoir dériver des lois de la physique moléculaire les principes de la microbiologie, (Ortoleva, et al. 2009) ont eu recours à la simulation multi-échelle. En effet, le temps nécessaire à la simulation d'un petit virus, en se plaçant au niveau de l'atome, prendrait des centaines d'années sur une architecture distribuée. Or certaines macromolécules ne vont pas évoluer durant de longues phases de la simulation, il n'est alors pas nécessaire de disposer d'une simulation aussi fine. En s'intéressant également à l'échelle des molécules, il est possible de différencier les différents éléments étudiés et d'appliquer une simulation plus ou moins fine à chacun d'eux.

1.1.1.3 Modélisation multi-algorithmique

Dans le cas des travaux de (Taufer, et al. 2009), la modélisation multi-échelles est encore une fois utilisée de façon différente. Dans le but de calculer les capacités de liaison entre les protéines et les ligands, plusieurs algorithmes plus ou moins efficaces, selon le couple protéine-ligand étudié, sont disponibles (voir Figure 1-4). La modélisation multi-échelle proposée permet donc de tirer parti de ces différents algorithmes pour sélectionner l'algorithme le plus adéquat pour un couple protéine-ligand donné.

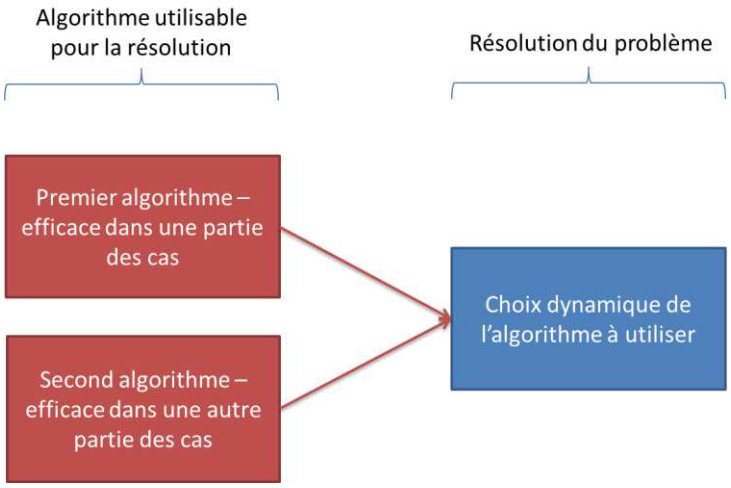


Figure 1-4 : Représentation de la modélisation algorithmiquement multi-échelle

L'utilisation de la modélisation multi-échelle au niveau microscopique peut donc prendre plusieurs formes. La modélisation de comportements situés à des niveaux de modélisation différents permet de prendre en compte, à chaque niveau de la modélisation, les comportements émergents aux niveaux supérieurs et inférieurs. La possibilité de modéliser un problème de manière plus ou moins fine, selon la zone étudiée, permet quant à elle de disposer d'un modèle suffisamment pertinent sur les zones du modèle les plus intéressantes, en conservant une représentation réaliste des zones les moins pertinentes, sans pour autant nécessiter l'attribution d'une importante charge de calcul (pour ces zones). Enfin, la modélisation algorithmiquement multi-échelle permet, lorsqu'une seule et unique solution n'est pas applicable à l'ensemble d'une classe de problèmes de disposer de différentes solutions qui seront utilisées lorsqu'elles seront les plus efficaces. Bien entendu, cette classification n'est pas exhaustive. La simulation multi-échelle étant en pleine évolution (voir Figure 1-5), de nouvelles propositions de modélisations multi-échelles voient le jour régulièrement.

Figure 1-5 : Évolution des publications dans le domaine de la modélisation multi-échelle ces 30 dernières années. (Source : Microsoft Academic Search)

1.1.2 Modélisation de processus macroscopiques

S'il est possible de modéliser finement des processus microscopiques à l'aide de la modélisation multi-échelle, cette technique est tout aussi utile pour la modélisation de processus macroscopiques.

En effet, ces activités biologiques sont également fortement multi-échelles ; aussi bien car elles prennent en compte plusieurs niveaux macroscopiques : les tissus, les organes et les systèmes d'organes par exemple mais également car il est souvent nécessaire de prendre en compte également des processus microscopiques. Les échelles microscopiques prises en compte sont par exemple les niveaux intercellulaires, extracellulaires et multicellulaires pour la cellule.

Or, il se trouve que chacun de ces niveaux est influencé par les autres (Tawhai, et al. 2009) :

- Les forces présentes aux niveaux des plus hautes échelles influencent le comportement des éléments des plus petites échelles.
- Inversement, les propriétés existantes aux niveaux des plus petites échelles influencent les réponses des éléments aux plus hautes échelles.

La modélisation de processus macroscopiques va donc très souvent nécessiter l'étude des processus microscopiques qui entrent en jeux. On retrouve ce phénomène dans tous les systèmes dits complexes et on parle parfois de simulations et modélisations holistiques dans l'étude du climat ou de l'environnement et ce pour préciser que l'on souhaite prendre en compte les niveaux micro, méso et macro (Sato 2004) (voir Figure 1-6).

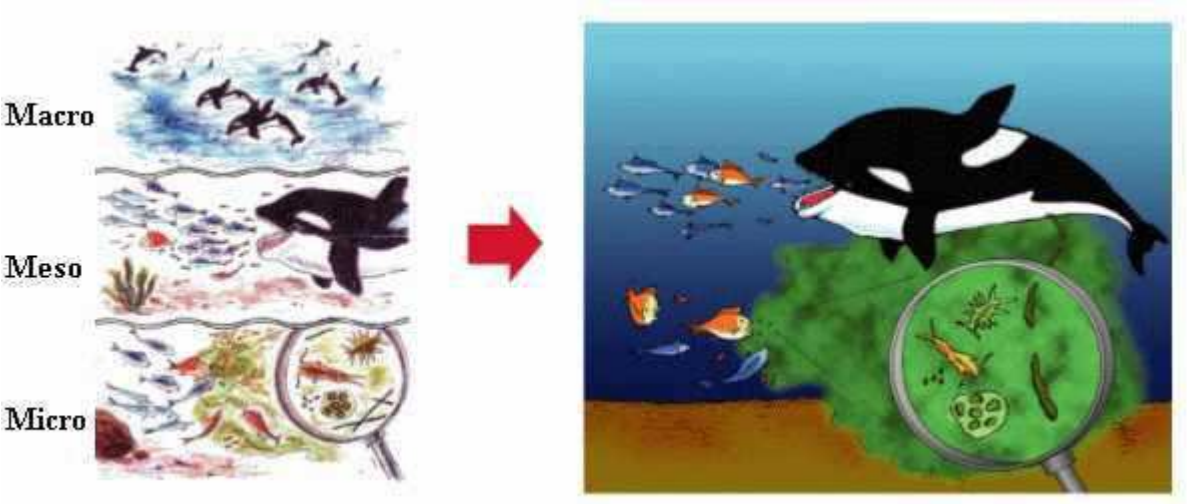


Figure 1-6 : Base des concepts de l'approche holistique (source : Multiscale Simulation & Modeling Research Group - http://www.jamstec.go.jp/esc/research/Mssg/index.en.html).

C'est également le cas pour les travaux de (Zhou, et al. 2009) qui traitent de la modélisation de la respiration à l'exercice. Cette modélisation nécessite de prendre en compte aussi bien la respiration externe que la respiration cellulaire. Il est alors nécessaire de modéliser l'évolution de la concentration en oxygène aux niveaux des tissus et des organes (aussi bien les poumons que l'ensemble des organes nécessitant de l'oxygène à l'exercice) ; mais également de modéliser l'utilisation de cet oxygène au niveau des cellules pour en déduire leur consommation.

Cependant, comme pour la modélisation de processus microscopiques, la simulation multi-échelle peut être utilisée pour faire varier la qualité de la simulation selon l'intérêt de la zone étudiée. (Sander, et al. 2009) utilisent cette technique pour permettre la modélisation des comportements des tissus en fonction de l'évolution de l'environnement à des échelles différentes selon la finesse recherchée. Ainsi, au niveau macroscopique, la modélisation est réalisée à l'aide de la méthode des éléments finis. Mais lorsque le tissu est soumis à des contraintes ayant pour conséquence une déformation significative du tissu, la résolution des éléments finis s'appuie sur la résolution préalable d'un problème de réseau au niveau microscopique. Ce dernier prend donc en compte des éléments plus fins pour permettre une modélisation réaliste de ces phénomènes complexes. Le tissu est alors modélisé en trois dimensions de manière à mieux prendre en compte les points d'interactions du modèle à éléments finis.

La simulation multi-échelle est donc un outil indispensable dans le cas de problèmes trop larges ou trop complexes pour être résolus avec le degré de finesse souhaité dans le laps de temps voulu. C'est par exemple le cas de la simulation de la circulation de gaz dans les poumons d'une personne : générer et simuler avec une maille fine et en trois dimensions le poumon de la trachée jusqu'aux bronchioles demande des capacités de calcul très importantes. (Lin, et al. 2009) tirent donc parti de la modélisation multi-échelle : l'ensemble des poumons est dans un premier temps modélisé en deux dimensions. Puis dans un deuxième temps, les parties que l'on souhaite tout particulièrement étudier sont modélisées en trois dimensions. Ces parties peuvent aussi bien correspondre à l'ensemble des branches de la $n^{ième}$ génération ou à l'ensemble des branches d'un chemin bien précis à travers un poumon. La simulation est alors possible sur l'ensemble des poumons en utilisant des algorithmes plus ou moins fins selon que la partie est modélisée en deux ou trois dimensions.

Si de nombreux systèmes complexes existent, nécessitant pour certains une modélisation multi-échelle, l'implémentation de ces mécanismes peut s'avérer plus compliquée qu'il n'y parait. En effet, le principe d'interactions multi-échelles n'est pas un problème simple du point de vue de la modélisation informatique du mécanisme. Pour permettre une modélisation et une simulation plus aisée de ces mécanismes, la société Integrative BioComputing (IBC) a proposé un outil permettant de réaliser de manière générique ces deux tâches.

1.2 Des systèmes complexes multi-échelles et de la modélisation « générique »

La simulation de systèmes complexes, que ce soit en biologie ou dans d'autres domaines, est rendue difficile par les caractéristiques intrinsèques de ces systèmes. Ils comprennent généralement de nombreux éléments aux propriétés et aux fonctions diverses qui peuvent interagir ensemble aussi bien à la même échelle qu'à des échelles différentes. On parle dans ce second cas de systèmes multi-échelles. Pour permettre, à l'aide d'une seule application, la génération et la simulation de différents systèmes complexes à différentes échelles, ces systèmes étant potentiellement différents les uns des autres, il est donc nécessaire de

disposer d'un outil fortement générique. C'est le but de ce que la société IBC a proposé à travers sa Plateforme Générique de Modélisation et de Simulation (PGMS) (P. Siregar, Simulation of complex systems 2009). Si le nom de cette plateforme est académiquement trop ambitieux, il s'agit néanmoins, comme son nom l'indique, de réaliser la modélisation ainsi que la simulation de processus (biologiques entre autres) de la façon la plus générique possible.

Pour permettre la simulation de systèmes très différents sans une modification en profondeur des mécanismes intrinsèques de l'application, la PGMS ambitionne à terme de se reposer sur l'utilisation deux mécanismes majeurs (Figure 1-7) :

- D'une part une base de connaissances qui permet de déterminer, lors de l'exécution de l'application, les caractéristiques des différents éléments présents dans la simulation.
- Et d'autre part des modules de raisonnement qui permettent quant à eux d'introduire des raisonnements en fonction des éléments présents. Il peut s'agir de moyens de communication, d'évolution, etc.

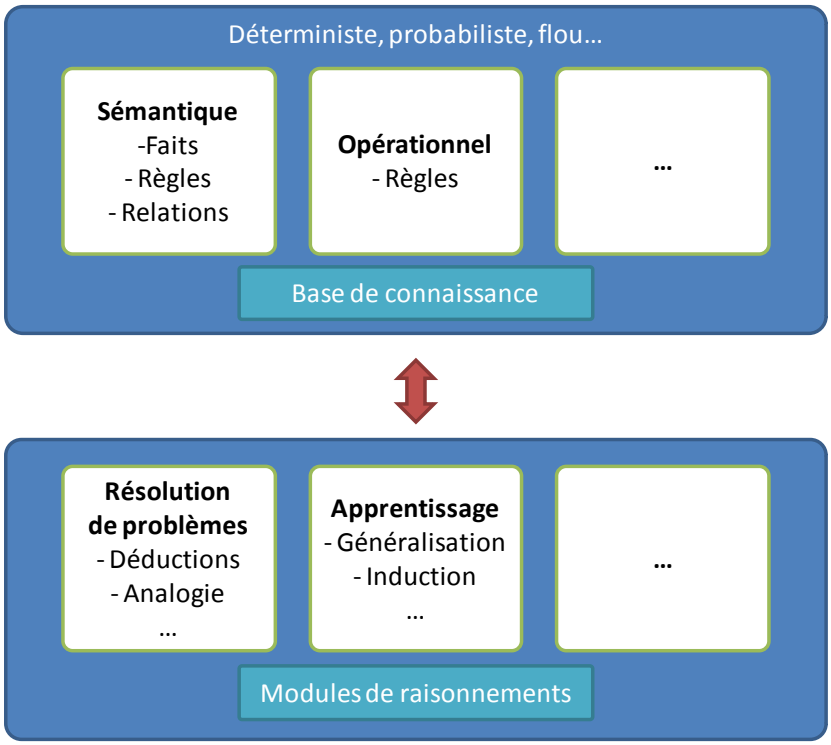


Figure 1-7 : Représentation des deux mécanismes majeurs de la PGMS : base de connaissance et modules de raisonnements.

La base de connaissances comprend toutes les données permettant de décrire les modèles le plus fidèlement possible. Ces données sont par ailleurs structurées de façon à permettre un traitement aisé par l'application. En s'appuyant sur une telle base de connaissances, il est possible de fortement impacter l'issue d'une simulation en ne modifiant que quelques éléments représentant l'initialisation de la simulation par exemple.

La version de la PGMS sur laquelle ont été effectués mes travaux de thèse n'est pas une version finale. Il s'agit d'un prototype, plus simple, ne disposant pas de tout son potentiel de généricité mais gardant les mêmes principes de base. Dans mon cas, j'ai tout particulièrement travaillé sur la morphogénèse de certains éléments du rein : les néphrons (Figure 1-8) et les pyramides rénales (Figure 1-9). Pour autant, mon travail devait être intégré dans la version générique de la PGMS et ce but n'a jamais été perdu de vue.

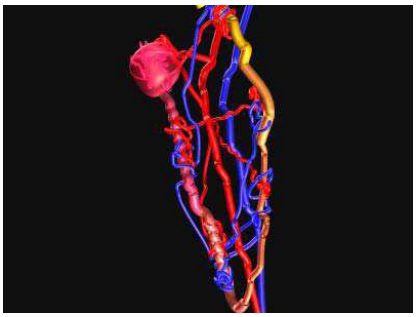


Figure 1-8 : Modélisation d'un néphron.

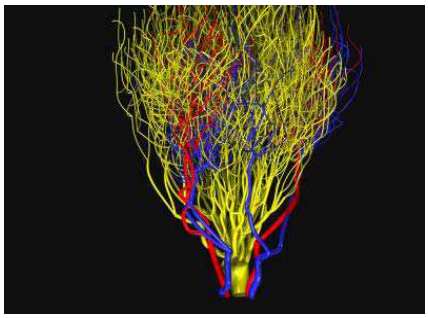


Figure 1-9 : Modélisation d'une pyramide rénale.

1.2.1 Principe général de la simulation au sein de la Plateforme Générique de Modélisation et de Simulation (PGMS)

La génération d'un organe par la PGMS comporte deux étapes majeures (voir Figure 1-10) : l'initialisation de la simulation et la génération de l'organe à proprement parler. Le temps d'exécution de ces étapes est logiquement très variable en fonction de la simulation. Il est évident qu'une pyramide rénale, composée dans la réalité d'environ 200 000 néphrons, va mettre beaucoup plus de temps à être initialisée et générée qu'un néphron unique. Un élément plus pertinent est la proportion du temps passé dans une étape par rapport au temps passé dans l'autre et son évolution en fonction de la taille des problèmes traités. Ce point est étudié plus en détail dans la partie 5.5 - Étude des performances sur d'autres simulations.

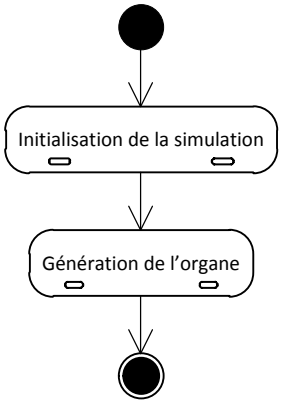


Figure 1-10 : Diagramme d'états représentant les deux grandes étapes de simulations réalisées à l'aide de la PGMS.

La complexité de ces deux étapes est significative et elle porte sur des aspects différents. Lors de l'initialisation de la simulation, un nombre important d'étapes se succèdent, les trois étapes principales sont les suivantes (voir Figure 1-11):

- L'initialisation de l'espace de simulation. Cette étape consiste à allouer l'espace de simulation nécessaire pour manipuler l'ensemble des éléments de la simulation.
- L'initialisation des cellules de type « souche » de départ. Au début de la simulation, seules quelques cellules sont présentes et vont, par division, différentiation, déplacement et apoptose cellulaire, former le futur organe. Cette phase a pour but de rechercher dans la base de connaissance quels sont les types cellulaires, leurs transitions d'état, ainsi que le nombre nécessaire de champs à initialiser la simulation.
- L'initialisation des champs qui peuvent être de nature chimique, électrique ou mécanique. Au début de cette étape, la valeur des champs présents dans l'espace de simulation est nulle en tout point. La valeur initiale est donc calculée en chaque point et pour chaque champ en fonction des cellules déjà présentes dans la simulation.

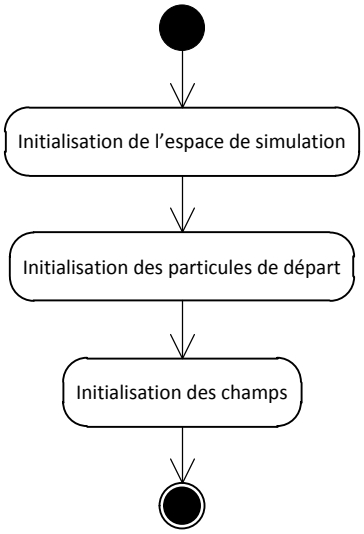


Figure 1-11 : Diagramme d'états de l'initialisation d'une simulation par la PGMS.

À la fin de cette étape, le modèle est donc instancié en mémoire et initialisé avec les paramètres requis pour la simulation choisie.

Une fois l'initialisation du modèle terminée, la génération de l'organe à proprement parler peut commencer (voir Figure 1-12). Celle-ci est effectuée par itérations successives. À chaque itération, l'ensemble des cellules évolue en fonction des différentes valeurs de champ auxquelles elles sont soumises. Les cellules peuvent alors se diviser, se déplacer… À la fin de cette phase, les cellules peuvent donc être en plus grand nombre, de natures différentes, et dans des positions différentes. Il est donc nécessaire de recalculer la valeur des différents champs en fonction de l'ensemble des cellules mises à jour.

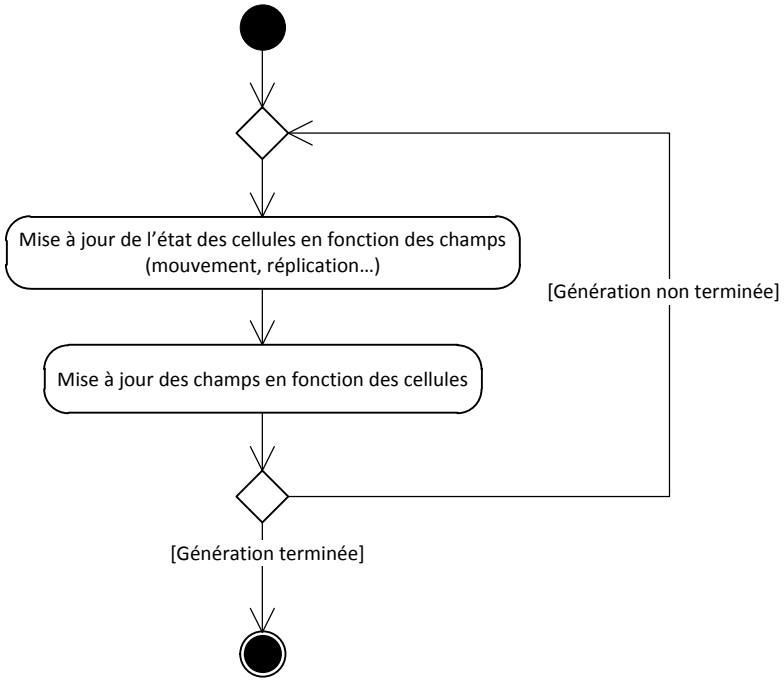


Figure 1-12 : Diagramme d'états de la génération de l'organe par la PGMS.

Comme on peut l'entrevoir ici, les champs sont donc une des composantes principales du mécanisme de génération de l'organe. Tout au long de la simulation, ils impactent très fortement sur le comportement des cellules qui impactent elles-mêmes sur les différents champs chimiques (en émettant des signaux chimiques), permettant ainsi une évolution différente au fur et à mesure de l'avancement de la simulation.

Mais si la modélisation des processus est une part très importante d'un projet de simulation, il ne faut pas minimiser l'importance du développement de l'application correspondante. Ainsi, si un programme extrêmement performant n'est rien sans un modèle correct, une implémentation correcte et performante est nécessaire pour utiliser au mieux un modèle pertinent. Toute la difficulté du développement logiciel est d'arriver à établir ce qu'est une application « correcte » et de l'implémenter. Car, au-delà de disposer d'une application fonctionnelle, davantage de facteurs entrent en compte. Bien entendu, dans une application de simulation, les performances de l'application sont très souvent importantes. Mais d'autres facteurs tels que la facilité d'adaptation du logiciel à de nouvelles problématiques,

l'accessibilité du code pour une personne extérieure[1], etc. sont tout aussi importants lorsque l'application développée dépasse le statut de prototype.

[1] L'accessibilité du code correspond à la facilité de compréhension de l'architecture logicielle utilisée aussi bien qu'à la qualité du code de l'application en tant que tel.

1.3 Génie Logiciel, refactorisation et optimisation.

1.3.1 Contexte original de l'application

Afin de bien appréhender les choix et le travail réalisé, il est important de connaître l'état du développement de la PGMS lorsque j'ai commencé à effectuer mes travaux. En raison de priorités autres au sein de l'entreprise, l'application n'avait que très faiblement évolué depuis le premier prototype fonctionnel présenté en 2006. Celui-ci, permettant déjà la simulation de tous les modèles qui seront présentés dans ce manuscrit, a été développé en Borland C++ et comportait plus de 200 000 lignes de code. Borland C++ est une implémentation du C++ bénéficiant d'extensions au langage ainsi que de bibliothèques proposant des outils complémentaires. En particulier, tous les conteneurs utilisés dans l'application étaient des conteneurs Borland et aucun conteneur standard appartenant à la bibliothèque standard du C++ n'était utilisé. Par ailleurs, l'application bénéficiait d'une interface graphique (permettant de choisir la simulation à réaliser et de la voir évoluer) entièrement développée à l'aide de la bibliothèque proposée par Borland.

Il a donc été nécessaire, avant même de pouvoir travailler et améliorer les performances, de modifier le code de l'application afin qu'il soit possible de plus facilement réaliser des modifications. Ainsi, en découpant tout le code de la PGMS lié à l'interface graphique de celui lié au code métier, les modifications apportées à l'application métier ont une portée beaucoup plus limitée et la propagation d'erreurs est moins importante. Mais des modifications qui peuvent sembler beaucoup moins primordiales, telles que la réorganisation d'une classe, peuvent également fortement impacter la compréhension d'un programme pour un nouveau développeur. Cet aspect du développement informatique est au cœur du génie logiciel qui est introduit dans la partie suivante.

Une fois la mise à jour de l'application effectuée, il a alors été possible de travailler sur l'amélioration des performances globales de l'application. Ici encore, le code de l'application n'ayant que peu évolué depuis le premier prototype fonctionnel, une analyse fine des processus les plus gourmands en temps de calcul a permis de mettre en lumière les points à optimiser dans le programme.

Que ce soit pour améliorer la qualité du code d'un programme ou sa vitesse à travers des optimisations, il est nécessaire de limiter au maximum les risques d'intrusions d'erreurs au

sein du code. Si cette tâche a été quelques temps sous-évaluée, elle correspond maintenant à une discipline de plus en plus connue et étudiée : le « refactoring » (aussi appelée refactorisation en français).

1.3.2 Génie logiciel

L'étude des bonnes techniques de programmation permettant de tendre vers le meilleur logiciel possible est ainsi devenue une discipline à part entière : le Génie Logiciel (souvent abrégé GL). Le but est d'établir des règles plus ou moins strictes qui, lorsqu'elles sont appliquées intelligemment, permettent d'améliorer la qualité du logiciel créé (McConnell 1993) (Sommerville 2001). Le problème est quelque peu différent dans le cas de l'amélioration d'une application déjà existante. Dans ce cas, l'amélioration de la qualité du code développé dans l'application ne peut se faire au détriment de la qualité de l'application développée. Poussée à l'extrême, l'idée est qu'il est inutile de disposer d'un code d'excellente qualité si celui-ci ne permet plus de faire fonctionner l'application. Il faut donc généralement réaliser des opérations d'amélioration ponctuelle du code sans introduire de défauts dans l'application.

Ainsi, si un code fonctionnel de la PGMS existait avant le début de ma thèse, le choix a été fait de retravailler celui-ci en profondeur afin de faciliter les développements futurs. Le code ayant été développé par accrétion de fonctionnalités durant un laps de temps important, la qualité de l'architecture de l'application et celle du code développé s'en est donc ressentie. Or, comme nous allons le voir par la suite, un nombre important d'études ont montré l'intérêt d'appliquer de bonnes méthodes de génie logiciel afin de maximiser l'efficacité des développements et de minimiser les coûts de production.

Dans un programme orienté objet, une partie importante du travail est relative à la conception des classes. Cette partie traite particulièrement de ce problème. Rendre une classe la plus fonctionnelle possible n'est pas une tâche aussi facile que cela peut sembler au premier abord. Un nombre important de points sont à prendre en compte lors de la conception. L'objectif général recherché va avoir une influence importante sur la façon de concevoir les classes de l'application. En fonction de cet objectif, les classes ne vont pas être conçues de la même façon. Ainsi, si l'on cherche à améliorer la robustesse d'une application, c'est-à-dire à améliorer sa capacité à continuer de fonctionner correctement en cas de

mauvaise utilisation, un plus grand nombre de tests seront réalisés au détriment de la vitesse d'exécution du programme.

Au-delà des choix stratégiques qui peuvent exister entre la création d'un programme robuste, fiable, juste, adaptable et/ou sécurisé…, certaines notions restent majoritairement communes à l'ensemble des classes. Ainsi, une étude de (Woodfield, Dunsmore et Shen 1981) a montré qu'un programme structuré en utilisant les types de données abstrait (Abstract Data Type) était plus facilement compréhensible par un programmeur extérieur qu'un programme ayant été implémenté au plus proche des problématiques fonctionnelles.

Une classe est d'autant plus compréhensible, et donc d'autant plus facile à modifier, si elle est organisée en un ensemble cohérent d'attributs et de méthodes. Un accent tout particulier est mis sur la conception de l'interface publique d'une classe. L'idée d'une telle approche est de conserver une interface la plus homogène possible. Le terme homogène est ici utilisé pour une interface utilisant le même niveau d'abstraction pour tous ses éléments. Dans son excellent livre « Code Complete » (McConnell 1993), Steve McConnell donne plusieurs exemples de types de données abstraits ainsi que les opérations correspondantes :

- Pour un type « Lumière », il serait possible de disposer des opérations : allumer et éteindre.
- Pour un autre type : « Pile », les opérations disponibles pourraient être par exemple : initialiser, ajouter un élément en haut de la pile, retirer l'élément en haut de la pile et lire l'élément en haut de la pile.

Ces deux interfaces sont parfaitement homogènes. Dans le premier cas, on traite uniquement de la mise sous ou hors tension d'une lampe. Dans le second cas, on manipule les éléments d'une pile.

À l'inverse, une interface non-homogène pourrait, dans le cas d'une classe représentant un panier de produits pour de la vente en ligne, traiter à la fois de l'ajout et de la suppression de produits (ajouter un produit, supprimer un produit…) mais aussi du stockage dans une base de données d'un tel panier (génération de la requête de création du tuple dans la base de données…). Ici, deux abstractions différentes se mélangent : la gestion des produits d'un panier et le stockage des instances de panier. Pour ne pas compliquer la classe inutilement

et ainsi diminuer les risques d'erreurs, la création d'une seconde classe, ne se chargeant que des manipulations liées aux bases de données, pourrait être une bonne initiative. Cette classe pourrait par ailleurs se retrouver fortement couplée à la première.

Au-delà d'avoir une interface homogène, il est utile de conserver des classes simples. Lorsque l'on parle de classes simples, plusieurs points peuvent être abordés. Ici, nous nous focaliserons sur les points pour lesquels des statistiques ont prouvé, depuis déjà de nombreuses années, un avantage en terme de diminution du nombre d'erreurs générées et donc en terme de temps de conception (Basili, Briand et Melo 1996).

L'interface d'une classe gagne tout d'abord à être limitée en nombre de méthodes. Durant leurs travaux sur le sujet, (Basili, Briand et Melo 1996) ont fait développer une solution informatique à un même problème par plusieurs groupes de personnes. Les solutions utilisant dans leurs classes un plus grand nombre de méthodes ont aussi été celles pour lesquelles le nombre d'erreurs lors de la conception et lors de la livraison de la solution était le plus grand. Toujours dans son livre « Code Complete » (McConnell 1993), Steve McConnell propose le nombre de sept méthodes (plus ou moins deux) par classe. Ce nombre de sept plus ou moins deux est tiré de l'article du psychologue George Miller (Miller 1956) dans lequel il explique que ce nombre correspond au nombre d'éléments différents qu'une personne peut utiliser lorsqu'elle exécute une tâche sans avoir à faire d'efforts supplémentaires pour se rappeler de l'utilisation de chacun. Bien entendu, selon les personnes et selon l'entrainement de celles-ci à la tâche qu'elles sont en train d'effectuer, ce nombre peut être différent. Il n'en reste pas moins qu'il s'agit là d'une bonne base lorsqu'il est nécessaire de réfléchir au nombre de méthodes qu'une classe doit proposer, au nombre de paramètres maximal de ces méthodes ou à la profondeur maximale d'héritage d'une classe (ce qui est souvent très important comme nous le verrons pas la suite), etc.

Maintenant qu'il devrait être acquis que la limitation du nombre de méthodes d'une classe induit un plus faible risque d'erreurs, il est important de bien noter qu'il ne s'agit pas là du seul élément ayant un impact statistique important sur le nombre d'erreurs qui peuvent apparaître lors de l'écriture d'un programme. La conception des méthodes est également très importante. L'étude de (Basili, Briand et Melo 1996) montre ainsi que le nombre d'erreurs est statistiquement plus important lorsque le nombre de fonctions et de méthodes

appelées par l'ensemble des méthodes d'une classe augmente. De la même façon, plus le couplage (nombre d'utilisations de méthodes et d'instances d'autres classes par les méthodes d'une classe) entre classes est grand, plus les chances de rencontrer des erreurs augmenteront.

Une bonne homogénéité de l'interface de la classe permet souvent de limiter le premier point. En effet, on se rend facilement compte que dans l'exemple non homogène du panier présenté précédemment, toutes les méthodes traitant du stockage dans la base de données vont utiliser un jeu de méthodes et de fonctions proches les unes des autres mais très éloignées des autres méthodes qui gèrent les éléments du panier. La conception de deux classes résoudrait une nouvelle fois ce problème : chaque classe utiliserait un jeu de méthodes très différent par rapport à l'autre classe mais au sein de chacune des classes, les méthodes seraient beaucoup réutilisées ou du moins très homogènes.

Le couplage est une source d'erreur car l'augmentation des interactions entre les classes rend ces dernières beaucoup plus difficiles à appréhender dans leur ensemble pour les développeurs. Il faut se rappeler que le nombre d'éléments qu'une personne peut manipuler à un instant donné sans effort supplémentaire, et donc sans augmenter le risque de faire des erreurs, est limité. Ici encore, le découpage en classes plus simples est encore une bonne solution. L'idée est d'augmenter la cohésion de la classe. Si une grande partie de la classe est couplée à des instances de la classe SQLManager, ce n'est pas un problème car la classe en question doit sans doute gérer les connexions aux bases de données. Le problème n'apparait que lorsqu'une classe est couplée à plusieurs classes n'ayant aucun ou peu de rapport entre elles. Il serait alors judicieux de penser à découper la classe en plusieurs classes qui traiteraient chacune une fonctionnalité. Ce problème de cohésion a également été mis en valeur lors de l'implémentation d'une méthode. Si une méthode traite de plusieurs problèmes différents, la probabilité d'apparition d'erreurs est plus importante. Il est alors nécessaire d'être homogène pour l'ensemble des méthodes qui constituent l'interface de la classe comme nous l'avons vu précédemment mais tout autant dans l'implémentation de chacune d'entre elles.

Une autre source importante d'erreurs lors de la conception d'un programme orienté objet est à chercher du côté de l'héritage. Il s'agit d'un concept très puissant pour modéliser

certains problèmes et comme tout concept, une utilisation erronée mène souvent à des erreurs. La première chose à intégrer est la règle suivante : l'héritage public correspond à la relation de généralisation / spécialisation que l'on représente par une relation ontologique de type « est un ». Cette règle est suffisamment importante pour que Meyers parle dans l'une des sections de son livre « Effective C++ » (Meyers 1992) de « la règle la plus importante lors de la conception de programmes orientés objet en C++ ». Dès 1987, Liskov énonce le principe de substitution qui porte son nom et qui définit que n'importe quelle instance « `instanceFille` » d'une classe « `Fille` » ayant pour classe mère une classe de base « `Mere` » doit pouvoir être passée en paramètre de n'importe quelle fonction ou méthode acceptant une instance « `instanceMere` » de la classe mère (Liskov 1987). Cette règle est reformulée par Hunt et Thomas (Hunt et Thomas 2000) dans leur livre « The pragmatic programmer - from journeyman to master » comme le fait qu'une classe fille doit pouvoir être utilisable à travers l'interface de la classe mère sans que l'utilisateur ait besoin de connaitre la différence. Ainsi, lorsqu'une classe ne peut être considérée comme une classe fille du point de vue de ces définitions, il s'agit souvent d'une mauvaise conception de la hiérarchie d'héritage qui nécessite un nouveau découpage.

Afin d'illustrer ce problème, l'exemple d'une classe oiseau possédant une méthode `voler()` est souvent utilisée (voir Figure 1-13). Si l'on souhaite modéliser différentes races d'oiseaux, il est possible d'utiliser l'héritage pour implémenter la classe Faucon ou la classe Pigeon. Mais l'intégration de la classe Autruche pose alors problème : une autruche est bien entendu un oiseau mais celle-ci ne peut pas voler. Il est donc anormal de pouvoir utiliser une instance de la classe Autruche à travers l'interface de sa classe mère Oiseau : le principe de substitution de Liskov est transgressé. Une solution consiste alors à créer deux nouvelles classes : `OiseauVolant` et `OiseauNonVolant`. La méthode `voler()` ne sera alors plus présente dans la classe `Oiseau` mais seulement dans sa sous classe `OiseauVolant` (voir Figure 1-14). Cette solution, qui n'est pas la seule possible, n'est pas exempte de défauts car elle augmente le nombre de classes (cinq classes au lieu de quatre) ainsi que la profondeur d'héritage (trois niveaux d'héritage au lieu de deux) ; ce qui n'est pas sans poser problème comme nous le verrons par la suite.

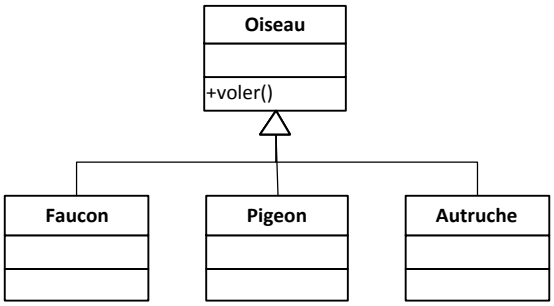


Figure 1-13 : Diagramme de classe original des différents oiseaux.

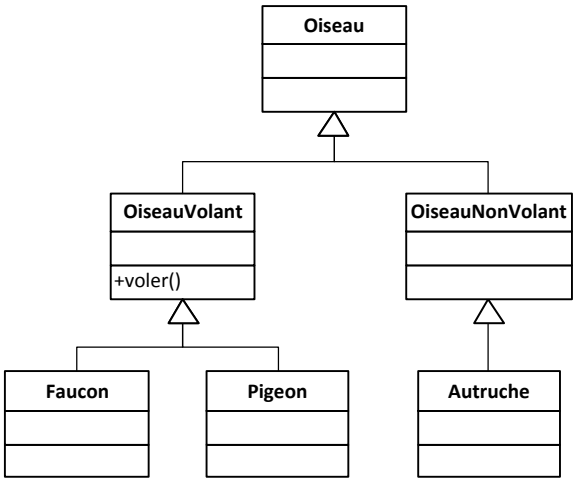


Figure 1-14 : Diagramme de classe des différents oiseaux différenciant les oiseaux volants et non volants.

Lorsque le principe de substitution de Liskov est transgressé, il peut également s'agir d'une erreur de conception. Le concept que l'on souhaite implémenter n'est alors pas « est un » mais souvent « a un » ou « est implémenté en termes de » que Meyers préconise d'implémenter à l'aide de la composition ou de l'agrégation dans une autre section de son livre.

Mais au-delà du problème de conception de l'héritage en tant que « est un », d'autres problèmes peuvent apparaître. L'utilisation de l'héritage a pour désavantage d'augmenter la complexité d'une classe : la connaissance de l'implémentation de la classe fille n'est plus suffisante pour un développeur qui souhaiterait connaitre l'interface complète de cette classe ou bien la façon dont sont implémentées toutes les méthodes. Beaucoup de ces informations vont ainsi appartenir à la classe mère. Et ceci est un problème majeur car cela va avoir un gros impact sur l'encapsulation de la classe, qui peut être brisée. Ainsi, il sera souvent nécessaire de connaitre l'implémentation d'une ou plusieurs méthodes de la classe

mère pour garantir qu'une fonctionnalité de la classe fille est implémentée correctement. Dans son livre « Effective Java » (Bloch 2001), Joshua Bloch propose l'exemple suivant pour illustrer ce problème :

```
public static class InstrumentedHashSet<E> extends HashSet<E> {

   public int addCount = 0;

   @Override
   public boolean add(E a) {
      addCount += 1;
      return super.add(a);
   };

   @Override
   public boolean addAll(Collection<? extends E> c) {
      addCount += c.size();
      return super.addAll(c);
   }
}
```

Le but est ici de créer une nouvelle classe représentant un hash set qui fonctionnerait exactement comme la classe `HashSet` présente dans l'API Java mais qui permettrait en plus de compter le nombre d'ajouts réalisés au `set`. Pour compter les ajouts, l'auteur a ainsi décidé d'incrémenter le compteur d'ajout `addCount` d'une unité dans la méthode `add` qui prend en paramètre un objet à ajouter et également de mettre à jour le nombre d'éléments ajoutés dans la méthode `addAll` qui prend en paramètre une collection d'objets à ajouter. Le problème intervient lorsque l'on sait que la méthode `addAll` de `HashSet` réutilise en fait la méthode `add` sur chacun des objets de la collection : l'utilisation de `addAll` compte alors deux fois chaque ajout. Par le fait qu'il soit nécessaire, pour implémenter correctement cette sous-classe, de connaître l'implémentation de la classe mère `HashSet` (qui, en vertu de l'encapsulation, ne devrait pas avoir à être connue), il est possible dans ce cas de dire que l'héritage brise l'encapsulation.

C'est sans doute pour ces raisons que l'étude de (Basili, Briand et Melo 1996) montre également que l'augmentation de la profondeur de l'héritage a une forte influence sur l'augmentation du nombre d'erreurs dans un programme.

Il serait possible de s'épandre beaucoup plus sur la conception des routines (fonctions et méthodes), de discuter du choix du nom des classes, des routines et des variables,

d'introduire les différentes méthodes permettant la détection d'erreurs le plus efficacement possible (revue du code par un collègue, tests unitaires, tests fonctionnels…) et beaucoup d'autres points. Le choix a été fait de s'arrêter à la conception des classes, qui n'est qu'une branche parmi bien d'autres du génie logiciel (Sommerville 2001), car il s'agissait du problème de conception le plus important constaté dans la reprise du premier prototype de la PGMS.

1.3.3 Optimisation logicielle

Lorsqu'il n'est pas primordial de disposer d'un programme développé le plus « proprement » possible, il est souvent possible de réaliser des compromis entre la qualité du code et les performances de l'application (c'est le cas des applications nécessitant les meilleures performances possibles). En effet, la majeure partie des optimisations qui peuvent être réalisées vont avoir tendance à complexifier le code et à le rendre moins facilement compréhensible et modifiable à l'avenir. Il est donc nécessaire de bien mesurer le gain de chaque optimisation de façon à ne conserver que les optimisations significatives.

Pour cela, en amont du travail d'optimisation, il est nécessaire d'établir précisément la liste des goulots d'étranglement de l'application. Or, parmi tous les outils disponibles pour le développeur, l'intuition n'est généralement pas le plus efficace. Ainsi, dans l'ouvrage « Refactoring: Improving the Design of Existing Code » de Fowler (M. Fowler 1999), Garzaniti rapporte que lors de l'optimisation d'une application, une étude de profilage avait montré que le plus gros goulot d'étranglement était la création de grandes chaines de caractères (jusqu'à 12 000 caractères) utilisées pour les entrées-sorties alors même qu'aucun développeur sur le projet n'avait avancé cette hypothèse lors des réflexions sur les axes d'optimisations possibles. Pour retrouver efficacement les goulots d'étranglement, il est donc obligatoire d'utiliser des outils mesurant précisément les performances des différentes parties de l'application : des profileurs. Lorsque les fonctions de l'application les plus gourmandes en termes de temps de calcul sont connues, il est possible de réaliser différentes formes d'optimisation.

Élimination des sous-expressions communes

Une première optimisation possible revient à supprimer certaines parties de code dupliquées dans des expressions différentes et à les factoriser dans une nouvelle expression

(Cocke 1970). Dans le Code 1-1 par exemple, l'expression permettant le calcul de la TVA sur un produit `coutHT * tauxTVA` est utilisé aussi bien pour le calcul du prix total `coutTTC` que pour le calcul du remboursement sur la TVA `remboursementTVA`. Il est donc possible d'extraire ce calcul dans une variable `TVA` puis d'utiliser cette variable en lieu et place de la sous-expression.

Mise en place, cette technique d'optimisation permet, lorsque le compilateur est incapable de la mettre en place seul, de diminuer les calculs à effectuer (dans notre exemple, la valeur de la TVA n'est ainsi calculée qu'une seule fois) mais aussi parfois de clarifier le code (les calculs sont plus clairs avec l'introduction d'une variable intermédiaire). L'inconvénient majeur de cette technique est qu'elle nécessite généralement l'utilisation de registres supplémentaires pour stocker les variables intermédiaires. Ceci peut poser de gros problèmes, en particulier sur des architectures où le nombre de registres est très limité telles celles des GPU.

```
// Avant
coutTTC = coutHT + coutHT * tauxTVA;
remboursementTVA = tauxRemboursmentTVA * coutHT * tauxTVA;

// Après
TVA = coutHT * tauxTVA
coutTTC = coutHT + TVA;
remboursementTVA = tauxRemboursmentTVA * TVA;
```

Code 1-1 : Exemple d'élimination d'une sous-expression commune.

Échange de boucles

Une bonne partie des optimisations les plus courantes traitent des boucles car celles-ci peuvent répéter un très grand nombre de fois les mêmes opérations. L'échange de boucles fait, comme son nom l'indique, partie de ces optimisations et traite du cas des boucles imbriquées. Ainsi, lorsque deux boucles sont imbriquées l'une dans l'autre et qu'il est possible, sans modifier le comportement final de l'application, de les inverser, il est parfois très avantageux de le faire (Allen et Kennedy 2001). En effet, si cette inversion permet d'accéder aux données dans un ordre plus proche de celui dans lequel elles sont stockées, les mécanismes de cache peuvent rendre l'opération beaucoup plus rapide. Le risque d'une telle optimisation est que l'ordre initialement prévu peut avoir un plus grand sens du point de vue applicatif. Il peut donc arriver que la compréhension du code soit alors plus difficile.

Dans l'exemple du Code 1-2, la distance à l'origine est calculée pour chaque point d'un tableau à deux dimensions (les coordonnées des points correspondent à leurs indices dans le tableau). La boucle extérieure parcourt les coordonnées en x et la boucle intérieure les coordonnées en y. Or, le tableau est stocké en mémoire de telle façon que l'on accède d'abord à l'indice y : `distanceOrigine[ y ][ x ]`. Le tableau est donc stocké ligne par ligne : les cellules possédant une même valeur de `y` se trouvant les unes à la suite des autres, de `0` à `maxX - 1`. Le parcours choisi ici ne reproduit donc pas ce stockage en parcourant tout d'abord toutes les cellules ayant l'indice x à 0 puis toutes celles ayant l'indice x à 1, etc. En inversant les deux boucles, on conserve un comportement similaire (la distance calculée sera la même) mais les accès seront faits dans l'ordre du stockage en mémoire.

```
// Avant
for ( int x = 0; x < maxX; ++x )
{
   for ( int y = 0; y < maxY; ++y )
   {
      distanceOrigine[ y ][ x ] = sqrt( x * x + y * y );
   }
}

// Après
for ( int y = 0; y < maxY; ++y )
{
   for ( int x = 0; x < maxX; ++x )
   {
      distanceOrigine[ y ][ x ] = sqrt( x * x + y * y );
   }
}
```

Code 1-2 : Exemple d'échange de boucle.

Fragmentation de boucles

La fragmentation de boucles s'applique lorsqu'une boucle contient, dans le but de gérer un ou plusieurs cas particuliers en début ou en fin de boucle, des instructions supplémentaires. Il est alors parfois possible de supprimer ces instructions au prix d'une séparation des cas particuliers et des cas généraux, principalement l'initialisation du premier élément ou un traitement particulier sur le dernier élément (Cannings, Thompson et Skolnick 1976). Dans le Code 1-3, l'objectif de remplir un tableau (`tabDecale`) avec le contenu d'un premier tableau (`tabBase`) décalé de trois indices est réalisé en affectant, pour l'indice `i`, la valeur de l'indice `i + 3`. Afin de correctement accéder aux trois premiers éléments du premier tableau lorsque l'on affecte les trois derniers éléments du tableau résultat, le calcul d'un

modulo est nécessaire sur l'ensemble de la boucle. En séparant le cas des sept premiers éléments de celui des trois derniers, il est possible de se contenter d'une addition (`i + 3`) dans les sept premiers cas et d'une soustraction (`i - 7`) dans les trois derniers. Si le gain de cette optimisation est assez évident étant donné que le nombre d'opérations effectuées diminue, l'inconvénient majeur, la complexification du code, l'est quasiment autant. En effet, là où il était assez aisé de comprendre que le code donné en exemple correspondait à un décalage de trois éléments dans la première implémentation, ceci est beaucoup plus difficile pour la seconde implémentation et des commentaires seraient nécessaires.

```
// Avant
for ( int i = 0; i < 10; ++i )
{
   tabDecale[ i ] = tabBase[ ( i + 3 ) % 10 ];
}

// Après
for ( int i = 0; i < 7; ++i )
{
   tabDecale[ i ] = tabBase[ i + 3 ];
}
for ( int i = 7; i < 10; ++i )
{
   tabDecale[ i ] = tabBase[ i - 7 ];
}
```

Code 1-3 : Exemple de fragmentation de boucles.

Boucle unswitching

L'optimisation de « boucle unswitching » correspond à la suppression d'un branchement dans une boucle. Pour que cela soit possible, ce branchement ne doit pas évoluer dynamiquement dans la boucle. Il est alors possible d'extraire le branchement de la boucle en dupliquant le code de la boucle dans chaque branche. Cette optimisation permet de passer d'un test par itération à un unique test. Le problème majeur de cette optimisation est la duplication du code de la boucle. Si une modification est nécessaire dans ce code, il sera alors obligatoire de la répercuter dans les deux copies.

Le Code 1-4 calcule le coût total toutes taxes comprises (TTC) d'une commande. Pour cela, une solution est de parcourir tous les produits et d'ajouter au coût total le coût TTC du produit courant. Mais l'on peut imaginer que les taxes varient selon le type de commande (et non pas selon le type de produit, sans quoi cette optimisation est impossible). On a alors un test au sein de la boucle qui n'évolue pas au cours de celle-ci. L'optimisation consiste

donc à commencer par ce test et à dupliquer la boucle dans les deux branches du `if`. Le code des deux boucles perd alors toute présence de tests.

```
// Avant
float coutTotal = 0;
for ( int i = 0; i < 10; ++i )
{
   if ( commandeAEmporter == true )
   {
      coutTotal += 1.05 * coutHT[ i ];
   }
   else
   {
      coutTotal += 1.19 * coutHT[ i ];
   }
}

// Après
float coutTotal = 0;
if ( commandeAEmporter == true )
{
   for ( int i = 0; i < 10; ++i )
   {
      coutTotal += 1.05 * coutHT[ i ];
   }
}
else
{
   for ( int i = 0; i < 10; ++i )
   {
      coutTotal += 1.19 * coutHT[ i ];
   }
}
```

Code 1-4 : Exemple de boucle unswitching.

Invariant de boucle

L'invariant de boucle est une des optimisations les plus simples à trouver et à mettre en œuvre. C'est la raison pour laquelle les compilateurs sont très performants pour la mettre automatiquement en place. Mais il est toujours intéressant de savoir de quoi il s'agit lorsque le compilateur ne la détecte pas seul. Le but est d'extraire des boucles tous les calculs qui ne varient pas au cours de celle-ci. En les extrayant, on diminue le nombre d'instructions et cela permet même parfois d'accélérer les accès aux données en conservant la constante dans un registre (Aho, Sethi et Ullman 1986). L'inconvénient est, comme cela était déjà le cas pour l'élimination de sous-expression commune qui est une optimisation très proche de celle-ci, que ce mécanisme peut augmenter le nombre de registres utilisés.

Dans le Code 1-5, le calcul du taux de TVA non remboursé ne varie jamais au cœur de la boucle. Il est donc possible de sortir ce calcul de la boucle et de diminuer ainsi le nombre de calculs effectués.

```
// Avant
for ( int i = 0; i < 10; ++i )
{
   coutTotal += coutHT[ i ] +
                coutHT[ i ] * tauxTVA * remboursementTVA;
}

// Apres
tauxTVANonRembourse = tauxTVA * remboursementTVA;
for ( int i = 0; i < 10; ++i )
{
   coutTotal += coutHT[ i ] +
                coutHT[ i ] * tauxTVANonRembourse;
}
```

Code 1-5 : Exemple d'invariant de boucle.

Fission / fusion de boucles

La fission de boucles (la fusion de boucles est le mécanisme inverse) consiste à séparer dans deux boucles deux composantes d'une même boucle. Bien qu'il soit alors nécessaire de réaliser deux fois le parcours au lieu d'un, dans certains cas, lorsque le corps de la boucle est important, il est possible que les mécanismes de cache soient plus performants en réalisant les deux boucles séparément (Kennedy et Allen 2001). Dans l'exemple du Code 1-6, deux tableaux (`tabA` et `tabB`) sont manipulés dans une même boucle. La fission de la boucle consiste à dupliquer la boucle pour que chacune ne s'occupe que d'un tableau.

```
// Avant
for ( int i = 0; i < 10; ++i )
{
   tabA[ i ] = 1;
   tabB[ i ] = 2;
}

// Après
for ( int i = 0; i < 10; ++i )
{
   tabA[ i ] = 1;
}
for ( int i = 0; i < 10; ++i )
{
   tabB[ i ] = 2;
}
```

Code 1-6 : Exemple de fission de boucle.

Déroulement de boucle

Le déroulement de boucle (unrolling en anglais) consiste à réécrire une boucle entièrement ou en partie de manière séquentielle (Aho, Ullman et Biswas 1977). L'objectif est ici de diminuer le nombre d'instructions liées au contrôle de la boucle. Le principal inconvénient est une complexification importante du code avec un éloignement du code de la problématique métier ainsi qu'un exécutable dont la taille peut significativement augmenter.

Dans le Code 1-7, qui présente un exemple de déroulement partiel, la boucle initiale parcourt les 300 valeurs du tableau une à une pour initialiser la valeur d'un pointeur à NULL. Le déroulement de la boucle est effectué en réalisant à chaque pas dans la boucle trois affectations et en incrémentant de 3 la variable i à chaque itération. Cette exemple montre bien les difficultés de maintenance qui peuvent survenir. En effet, si le tableau évolue vers une taille non multiple de 3, il sera nécessaire de modifier le déroulement en rajoutant une fragmentation de boucle pour gérer les derniers éléments par exemple.

```
// Avant
for ( int i = 0; i < 300; ++i )
{
   tabPtr[ i ] = NULL;
}

// Après
for ( int i = 0; i < 300; i += 3 )
{
   tabPtr[ i     ] = NULL;
   tabPtr[ i + 1 ] = NULL;
   tabPtr[ i + 2 ] = NULL;
}
```

Code 1-7 : Exemple de déroulement de boucle.

Métaprogrammation template

La métaprogrammation template tire parti du mécanisme des templates en C++ pour réaliser une partie des instructions lors de la phase de compilation plutôt que lors de l'exécution du programme. En évaluant une expression template, il est ainsi alors possible de réaliser le calcul de constantes de manière à éliminer ces calculs lors de l'exécution ou de réaliser des opérations sur des types afin de faire de la vérification de type par exemple (Touraille, Traoré et Hill 2010). Si cette technique peut permettre de grandement améliorer les temps d'exécutions de l'application, elle peut également grandement complexifier le code de l'application. Par ailleurs, l'ensemble des calculs réalisés à la compilation font croître

le temps total de compilation qui peut ainsi devenir très significatif et rédhibitoire dans certains cas.

L'exemple le plus classique d'utilisation de la métaprogrammation template pour le calcul de constantes est celui du calcul d'une factorielle (voir Code 1-8). De manière classique, la factorielle de `n` est calculée de manière récursive en retournant `1` si `n` vaut `0` et `n * factorielle( n - 1 )` sinon. L'utilisation du mécanisme de programmation template implique de demander au compilateur d'évaluer une expression à la compilation. Plusieurs mécanismes sont possibles parmi lesquels on trouve l'évaluation lors de l'initialisation d'un énumérateur (`valeur` dans notre cas) d'une énumération (non nommée dans notre cas). Pour différentier les appels selon les valeurs de `n`, l'idée est d'avoir une spécialisation template pour chaque valeur différente. La fonction `Factorielle` est donc paramétrée par cette valeur `n`. Dans le cas général (n différent de 0), l'énumérateur est donc initialisé à la valeur de `n` multiplié par la valeur de l'énumérateur pour la classe `Factorielle` paramétré par `n - 1 : n * Factorielle< n - 1 >::valeur`. Lorsque `n` vaut `0`, on spécialise la fonction template pour initialiser `valeur` à `1`. L'appel `Factorielle< 4 >::valeur` réalise alors l'ensemble des calculs à la compilation.

```
// Avant
int factorielle( int n )
{
   if ( n == 0 )
   {
      return 1;
   }
   else
   {
      return n * factorielle( n - 1 );
   }
}

// Appel :
factorielle( 4 ) ;

// Après
template < int n >
struct Factorielle
{
   enum { valeur = n * Factorielle< n - 1 >::valeur };
};

template <>
struct Factorielle< 0 >
{
   enum { valeur = 1 };
};
```

```
// Appel :
Factorielle< 4 >::valeur ;
```

Code 1-8 : Exemple de programmation template

Accélération de calcul à base de GP-GPU

Avec l'objectif d'améliorer les performances de la PGMS sur des machines de tables, l'utilisation de GP-GPU s'est rapidement imposée. Ceci dit, les spécificités de l'architecture des GPU impactent fortement sur les méthodes d'optimisations. Ainsi, si certaines des optimisations précédentes sont toutes aussi valables sur GPU que sur CPU (déroulement de boucle, boucle unswitching par exemple), certaines peuvent être contre productives. C'est le cas des optimisations nécessitant l'utilisation de variables supplémentaires pour être implémentées (invariant de boucle et élimination des sous-expressions par exemple). En effet, le nombre de registres disponibles pour chaque thread exécuté est très limité. Pour pouvoir tirer parti du nombre maximum de threads ordonnancés en même temps, il est ainsi nécessaire de ne pas utiliser plus de trente-deux registres pour chaque thread. Si ce nombre est dépassé, il est alors probable que les performances de l'application diminuent. Il va ainsi parfois être nécessaire, pour accélérer les calculs, de réaliser l'optimisation inverse de l'invariant de boucle ou de l'élimination des sous-expressions en dupliquant des calculs afin de diminuer le nombre de variables et par conséquent le nombre de registres utilisés.

Les performances liées à la mémoire du GPU, que ce soit lors de la copie par le CPU ou lors de l'accès à une variable par les threads, impactent également fortement sur les optimisations à réaliser. Une importante partie du travail d'optimisation sur GPU va chercher à diminuer l'ensemble des coûts liés aux accès mémoire. Pour cela, plusieurs solutions existent. Tout d'abord, il est préférable, autant que possible, de réaliser les accès à la mémoire de façon coalescente. De manière simplifiée, il s'agit de faire accéder les threads à des zones mémoire se succédant (cette notion est reprise plus en détail dans la partie 3.2.1 - Mémoire globale (global memory)). Il est également possible d'utiliser les différents types de mémoires disponibles sur un GPU : mémoire constante, mémoire de texture et surtout la mémoire partagée dont les caractéristiques sont différentes de la mémoire globale (ces caractéristiques sont détaillées dans la partie 3.2 - Mémoire des GPU). Enfin, lors de la copie de quantité de données très importante entre la mémoire du CPU et celle du GPU, il est possible de réaliser celle-ci de façon asynchrone afin de pouvoir utiliser le temps

de copie pour la réalisation d'autres calculs (voir partie 3.3.3 - Utilisation des flux asynchrones (stream)).

Analyse de l'activité

De manière plus complexe, l'optimisation d'un processus peut également être réalisée en traitant différemment l'information. Il est ainsi possible, dans un espace discrétisé, de ne pas traiter l'ensemble des cellules mais uniquement celles dont l'information doit évoluer. L'idée est donc de diminuer les calculs en focalisant ces calculs sur les éléments les plus actifs de la simulation. Ce concept de l'activité est fortement lié aux approches permettant de faire évoluer une simulation. Balci a proposé une classification des processus de simulation en quatre approches distinctes (Balci 1988). Si les deux premières approches sont sans liens avec la notion d'activité, l'approche activité et l'approche en trois phases se basent sur l'activité pour faire évoluer les simulations.

L'approche activité, introduite par Buxton et Laski dans le langage CSL (Buxton et Laski 1962), est fondée sur la scrutation d'activité et est également appelée « approche deux phases ». Elle se décompose fort logiquement en deux phases : la gestion du temps dans la simulation pour la première et la scrutation des activités pour la seconde. Une activité est alors définie d'une part par une ou plusieurs conditions qu'il est nécessaire de remplir pour que cette activité soit valide et d'autre part par une ou plusieurs opérations à effectuer lorsque celle-ci est valide.

L'approche trois phases, proposée par Tocher (Tocher 1963), a été conçue pour améliorer les performances de l'approche précédente. Pour permettre de simplifier la scrutation d'activité, cette nouvelle approche dissocie les activités inconditionnelles des activités conditionnelles. De cette façon, il est possible de traiter séparément les activités inconditionnelles et la scrutation de l'activité à proprement parler. Le traitement des activités inconditionnelles est ainsi effectué en premier lieu en ne tenant compte que de l'horloge de la simulation. Dans un second temps, la scrutation de l'activité permet de déterminer quelles activités conditionnelles sont valides à cet instant de la simulation.

Si l'utilisation de telles approches n'est pas la solution à l'ensemble des problèmes de performance des simulations, son application à des systèmes où la majorité de l'activité du

modèle se trouve dans une zone restreinte permet d'éviter l'évaluation de l'ensemble du domaine à chaque pas de temps (Coquillard et Hill 1997) (Muzy, Nutaro, et al. 2008).

1.3.4 Refactoring

Qu'il s'agisse de modifier un code existant pour améliorer sa conception en s'appuyant sur le génie logiciel ou pour améliorer les performances de l'application à travers des optimisations ou la parallélisation des calculs, il est essentiel d'éviter l'introduction de nouvelles erreurs dans le programme. Réaliser ces modifications sans modifier le comportement de l'application est le but de la refactorisation (le terme anglais « refactoring » est plus couramment employé). Cette discipline, plus complexe qu'il peut y paraître, a également largement été étudiée (M. a. Fowler 1999) (Feathers 2005). Le succès de cette approche s'explique par l'important volume des codes patrimoniaux (« legacy code ») dans les entreprises.

Avant toute chose, une pièce maitresse du dispositif de développement informatique est la présence de tests automatisés (Feathers 2005). Au sein du cycle de développement d'une application, les tests automatisés peuvent être utilisés de différentes façons : afin de vérifier les performances de l'application, sa robustesse ou plus simplement si les fonctionnalités implémentées l'ont été correctement. Ce sont ces derniers tests qui permettent de réaliser des modifications de l'application en minimisant le risque d'intrusion de nouvelles erreurs de programmation. Une fois que de tels tests ont été mis en place, il est possible de vérifier que le comportement d'une application est toujours correct en exécutant simplement les tests et en vérifiant que ceux-ci sont bien toujours réussis.

L'idée majeure de la refactorisation est de vérifier, aussi souvent que possible, que les modifications apportées au code de l'application n'ont pas altéré son comportement. De cette façon, lorsqu'une erreur est détectée, il est beaucoup plus facile de la corriger ou de revenir à une version précédente au fonctionnement adéquat. Pour faciliter ces corrections, les modifications doivent être découpées en étapes les plus élémentaires possibles permettant de tester aussi souvent que possible le bon comportement de l'application. Ainsi, s'il devient utile de séparer une classe comprenant deux catégories de comportements distincts en deux classes, cette séparation se fera en étapes aussi simples que possible. (M. Fowler 1999) indique qu'une bonne solution consiste, après avoir créé la nouvelle classe vide

et créé un lien entre les deux classes, a tout d'abord déplacer un à un les attributs souhaités en compilant et testant après chaque déplacement. Dans un second temps, chaque méthode à déplacer sera prise une à une sans oublier de compiler et de tester après chaque déplacement. Enfin, une fois l'ensemble des attributs et des méthodes dans la bonne classe, il est nécessaire de nettoyer l'interface de chacune des classes pour ne conserver que les éléments essentiels. Cette méthode peut paraitre fastidieuse car elle nécessite plus de travail qu'un découpage réalisé en une seule fois mais tout le temps passé à réaliser ces étapes une à une est largement compensé par le temps gagné sur la correction d'erreurs. Ici, si une erreur survient, celle-ci apparaitra immédiatement après le déplacement de l'attribut ou de la méthode incriminée ; la correction de celle-ci en sera donc largement facilitée.

Conclusion

Parmi les problématiques existant en modélisation informatique, la modélisation des systèmes complexes est sans doute de celles ayant connu le plus grand intérêt ces dernières années. En effet, dans beaucoup de domaines et en particulier la biologie, un nombre important de modèles peuvent être représentés à l'aide de nombreuses entités interdépendantes ; c'est-à-dire à l'aide d'un système complexe.

Comme cela a été vu dans cette partie, la modélisation de tels systèmes nécessite généralement la mise en place de modélisation multi-échelle pouvant prendre différentes formes :

- La modélisation multi-niveau dans le cas où il est possible de découper le système en sous-systèmes ayant des interactions aussi limitées que possible entre eux.
- La modélisation multi-résolution lorsqu'il n'est pas souhaitable de traiter aussi finement un problème en tout point de l'espace de simulation.
- La modélisation multi-algorithme lorsque l'utilisation d'un algorithme pour résoudre un problème donné peut évoluer dynamiquement.

C'est pour permettre la simulation de tels systèmes le plus facilement possible qu'a été développée la Plateforme Générique de Modélisation et de Simulation (PGMS). Cet outil a donc pour but de rendre la modélisation et la simulation les plus génériques possible de façon à rendre la conception de nouveaux modèles rapide. Cet objectif très ambitieux est rendu possible par l'utilisation d'une base de connaissances permettant de réaliser des

opérations différentes sur les cellules biologiques présentes en fonction de leurs types et de l'état du milieu où elles se trouvent. Et c'est ce calcul de l'état du milieu environnant, tout particulièrement le calcul des champs physicochimiques en présence, qui nécessite une importante quantité de calculs à chaque itération de la simulation.

Mais l'application, qui représente plus de 200 000 lignes de code, a été développée à l'aide d'une variante du C++ non maintenu. Il a donc été nécessaire, avant même de parler d'amélioration des performances, de la mettre à jour afin qu'elle fonctionne avec des logiciels de développement récents. Pour mettre à jour l'application, des techniques inspirées du génie logiciel ont été mises en place dans le projet. Une fois ces méthodes appliquées, nous verrons comment il est possible d'améliorer les performances de l'application.

Mais si des problèmes existaient au sein du code original de la PGMS, la qualité du code seule ne peut être tenue responsable pour le temps d'exécution important de la simulation. En effet, le meilleur code, d'aussi bonne qualité soit-il, s'exécutera toujours pendant le même temps si les modèles sont sophistiqués et nécessitent de nombreux calculs. La PGMS permet la simulation d'organes humains faisant pour cela interagir des processus complexes. Il est donc logique que ces processus (qui sont bien entendu nécessaires) pénalisent les performances de l'application. De manière à augmenter significativement les performances de l'application, nous avons donc considéré d'une part la parallélisation de celle-ci, et d'autre part, la proposition d'algorithmes plus efficaces pour diminuer significativement le temps d'exécution tout en conservant le même comportement du modèle. Dans le chapitre suivant, nous présenterons d'une part, les éléments de calcul parallèle nécessaires dans notre contexte et d'autre part, nous aborderons la simulation d'automates cellulaire en détaillant l'algorithme Hash-Life en deux dimensions qui sera repris et adapté en trois dimensions dans la suite de notre manuscrit.

Du calcul parallèle aux automates cellulaires

2 Du calcul parallèle aux automates cellulaires

Introduction

Comme cela a été décrit dans le chapitre précédent, les techniques de simulation de systèmes complexes permettent de modéliser une grande variété de problèmes : du réseau de gènes à la colonie d'insectes, en passant par les interactions sociales et bien d'autres domaines. Mais pour prendre en compte l'ensemble des interactions propres à un système complexe, il est souvent nécessaire que la solution retenue tire parti de mécanismes multi-échelles permettant de gérer les interactions au sein du modèle de la façon la plus simple possible.

L'implémentation de tels mécanismes multi-échelles n'est pas anodine. Ces implémentations doivent à la fois prendre en compte les interactions entre les différents niveaux de la modélisation et celles, au sein d'un même niveau, entre les différents éléments. Pour implémenter de telles solutions, plusieurs outils sont disponibles. Les automates cellulaires sont ainsi largement utilisés. Ils permettent aussi bien d'implémenter certaines notions d'interactions multi-échelles (Hoekstra, et al. 2008) que les processus biologiques eux-mêmes (P. a. Siregar 1997).

Bien entendu, lorsque les niveaux sont nombreux et les composants à chaque niveau en nombre important, la puissance de calcul requise pour permettre à un ordinateur de modéliser ces systèmes peut-être très importante (Masumoto, et al. 2004). L'utilisation de matériels de calcul, toujours plus puissants, est donc nécessaire afin de pouvoir travailler sur des modèles les plus complets possibles, prenant en compte un maximum d'éléments et d'interactions. L'étude des systèmes complexes est ainsi fortement liée au calcul intensif car il est courant qu'une application nécessite d'être implémentée de façon parallèle pour conserver des temps d'exécution corrects (Bagrodia, et al. 1998).

Ce chapitre présente, dans une première partie, les solutions existantes pour paralléliser une application, en commençant par l'historique des supercalculateurs jusqu'aux nouvelles approches hybrides de ces dernières années. Puis, dans une seconde partie nous discutons des algorithmes de simulation d'automates cellulaires avec notamment une présentation de l'algorithme Hash-Life en deux dimensions.

2.1 Parallélisme logiciel et matériel

Dès ses débuts, l'informatique avait pour but de résoudre des problèmes dans les temps les plus courts possibles. De plus, afin de disposer de modèles les plus « fidèles » et précis possibles, les scientifiques ne cessent d'affiner leurs calculs pour les rapprocher le plus possible de la réalité des systèmes complexes. De plus, l'augmentation du nombre de facteurs étudiés sur ces modèles et la mise en œuvre de plans d'expériences les plus complets possibles impliquent des calculs de plus en plus lourds. C'est pour essayer de résoudre ces problèmes de performance que, dès les années 1960, le calcul à haute performance est devenu un domaine de recherche à part entière (High Perfomance Computing ou HPC en anglais).

2.1.1 Introduction à la notion de « superscalculateurs »

De ces travaux sont apparus les premiers supercalculateurs, des ordinateurs uniquement dédiés à l'exécution de tâches de calcul intensif. Afin d'améliorer les performances des calculateurs de l'époque, plusieurs solutions sont apparues au cours du temps. La première d'entre elles consistait simplement à toujours disposer de processeurs séquentiels possédant une capacité de calcul de plus en plus élevée. Cependant, l'augmentation des capacités des processeurs à très vite été jugée trop lente pour satisfaire les utilisateurs les plus exigeants. Afin de continuer à augmenter les performances, les scientifiques se sont donc tournés vers la notion de parallélisation, sous toutes ses formes. Ainsi, les premiers supercalculateurs possédaient déjà des processeurs supplémentaires permettant de traiter séparément les entrées/sorties de façon à ne conserver pour le CPU que les tâches de traitement des données. Il s'agit des prémisses de la programmation parallèle, et en particulier de processeurs dédiés qui sont par exemple les prémisses des futurs GPU (Graphical Processing Units).

Une décennie plus tard, au cours des années 70, les supercalculateurs s'enrichissent d'instructions vectorielles qui permettent aux ordinateurs de traiter une même instruction sur plusieurs données. Dans la classification de Flynn (Flynn 1972), il s'agit du principe connu en anglais sous l'acronyme SIMD (pour Single Instruction, Multiple Data) (voir Figure 2-1). Cette technique permet de fortement accélérer certains calculs ; elle a notamment été pendant des années le fer de lance de la marque Cray Computer fondée par Seymour Cray. Dans les années 80, Danny Hillis a développé au Massachusetts Institute of Technology (MIT)

des machines cellulaires (Hillis 1989) utilisant ce modèle. Il s'agissait ici d'utiliser un très grand nombre de processeurs (plusieurs dizaines de milliers) très simples ne pouvant traiter qu'un bit à la fois. Elle est encore aujourd'hui largement utilisée au travers, par exemple, des instructions SSE[2] (Streaming SIMD Extensions) des microprocesseurs actuels. Pour autant, si le calcul n'est pas parallélisable par rapport aux flux de données, cette technique est inopérante.

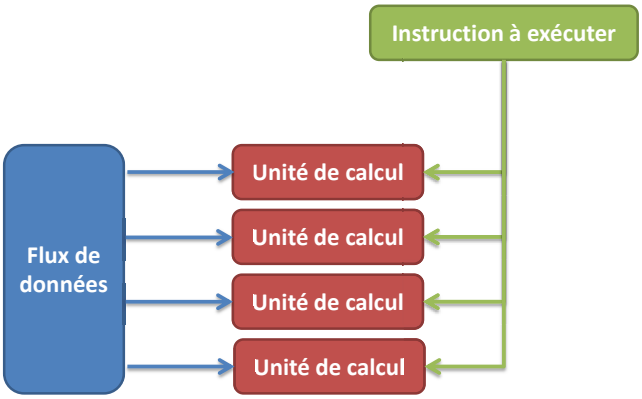


Figure 2-1 : Représentation du principe SIMD : une seule et même instruction est réalisée en parallèle sur différentes données.

D'autre moyens pour paralléliser des calculs plus généralistes ont donc était mis en place et les supercalculateurs se sont rapidement dotés de plusieurs processeurs travaillant en parallèle (Hoffman et Traub 1989). De quatre à seize processeurs dans les années 80, on passera à 140 processeurs en 1994 (Top500 - 1994[3]) pour atteindre plus de 100 GFlops/s en double précision (100 milliards d'instructions à virgules flottantes par seconde). Le net avantage de cette méthode de parallélisation sur la précédente est qu'il est possible d'exécuter sur les différents processeurs des instructions différentes voire des programmes n'ayant pas de lien entre eux. On parle alors de mode MIMD (pour Multiple Instructions, Multiple Data) : des instructions différentes peuvent à un même instant s'appliquer à des données différentes (voir Figure 2-2). Cependant, cette méthode de parallélisation n'est pas parfaite. Ici encore, tous les problèmes ne peuvent pas exploiter pleinement cette architecture : suivant les algorithmes, des communications peuvent être nécessaires pour

[2] Les instructions SSE correspondent à un jeu de 70 instructions supplémentaires permettant aux microprocesseurs d'architecture x86 de réaliser des calculs similaires sur des données différentes en parallèle.

[3] Top500 - 1994 : http://www.netlib.org/benchmark/top500/reports/report94/main.html. Dernier accès le 27/12/2011.

synchroniser les différents processus, mais dans tous les cas, il faut repenser l'application pour qu'elle puisse être découpée en plusieurs parties aussi indépendantes que possible.

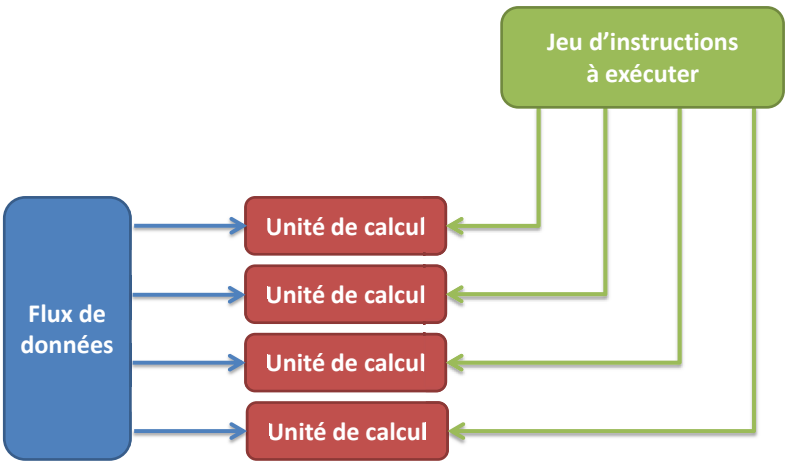


Figure 2-2 : Représentation du principe MIMD : des instructions différentes sont réalisées en parallèle sur différentes données.

Plusieurs architectures spécialisées ont essayé de tirer le meilleur du mode MIMD. Dans les années 80, la société Inmos a ainsi produit les premiers transputers. Cette architecture a été conçue de manière à ce que plusieurs transputers puissent être connectés ensemble le plus aisément possible, sans nécessité de carte mémoire ou de bus complexe. Couplés à l'utilisation du langage Occam, les transputers permettaient de simplement réaliser des applications fortement parallèles (Hoare 1985).

2.1.2 L'arrivée des clusters

Dans les années 90, la manière de penser le calcul hautes performances a dû être complètement revue. En effet, avec l'avènement des micro-ordinateurs, le prix des microprocesseurs standards devient très bas par rapport aux processeurs extrêmement puissants des supercalculateurs traditionnels. De plus, l'apparition, au début des années 90, des processeurs Intel de type 80486, intégrant sur une même puce les unités de calcul en nombres entiers et flottants (FPU – Floatting Point Unit de type '387' à l'époque) a révolutionné les performances des micro-ordinateurs personnels pour le calcul scientifique. C'est la raison pour laquelle apparaissent des systèmes massivement parallèles composés d'un nombre très important de CPU classiques ne partageant pas la même mémoire (Van der Steen et Dongarra 1996). Avant l'apparition des fermes ou grappes de PC dédiés, constituant les premiers « clusters » avec des communications réseau optimisées, le concept de machine parallèle virtuelle constituée éventuellement d'ordinateurs de type PVM

(Parallel Virtual Machine) utilisait simplement les réseaux locaux ou internet. On peut considérer qu'il s'agit d'ailleurs de l'ancêtre des grilles de calcul. Les clusters ont l'avantage de proposer un rapport puissance de calcul sur coût extrêmement bas par rapport aux supercalculateurs traditionnels. Le principal frein à leur utilisation est une lenteur « relative » des communications interprocessus par rapport à des machines de type SMP (Symmetric Multi-Processing) qui sont des machines à mémoire partagée. Ce dernier aspect est tout relatif, car l'accroissement des performances au niveau des composants réseaux interconnectant les éléments du cluster est très significatif. Cet outil, plus encore que le précédent, nécessite l'utilisation d'algorithmes conçus spécifiquement pour une utilisation fortement distribuée et disposant de processus les plus indépendants possibles les uns des autres.

Le début des années 2000 est marqué par l'arrêt brutal de l'augmentation des performances des CPU mono-cœurs avec notamment la stagnation de la progression des performances basées sur l'augmentation de la fréquence qui cadence les micro-processeurs. Les constructeurs s'approchent également du nombre limite de transistors qu'il est possible de graver au m² sur une plaque de silicium en technologie CMOS (Complementary Metal–Oxide Semiconductor). Cela va avoir un impact considérable sur les microprocesseurs standards et donc sur la composition des clusters qui se servent massivement de ces processeurs. Ainsi, si dans un premier temps, les constructeurs ont cherché à reproduire le système des multiprocesseurs déjà existants dans les supers ordinateurs, très vite c'est l'utilisation de plusieurs cœurs sur un même microprocesseur qui a pris le dessus. Tout d'abord, les constructeurs ont introduit la notion de « Simultaneous MultiThreading » (nommé « HyperThreading » par Intel) (Koufaty et Marr 2003), avec des augmentations de performances modestes (x1,2 ou x1,3). Cette approche a convergé vers de vrais processeurs multi-cœurs qui ont l'énorme avantage de partager complètement leur mémoire et donc de permettre des communications bien plus rapides entre cœurs que ce qu'il est possible d'avoir entre processeurs (chaque processeur multi-cœur est en fait une petite machine SMP). Depuis, le nombre de cœurs par microprocesseurs n'a pas cessé d'augmenter tandis que la fréquence de chacun d'eux a plutôt eu tendance à diminuer, au moins dans un premier temps. En effet, la dernière génération des processeurs mono-cœurs Pentium 4 d'Intel, sortie en 2006, était proposée avec des fréquences allant jusqu'à 3,6 GHz de manière

native lorsque la première génération de processeurs dual cœurs sortie la même année ne proposait pas plus de 2.33 GHz. De nos jours, les derniers processeurs Intel Westmere possèdent jusqu'à 10 cœurs permettant d'exécuter chacun deux threads différents et les processeurs AMD de type Magny-Cours proposent jusqu'à 12 cœurs de calcul. Les fréquences ont pu augmenter grâce à des gravures plus fines pour atteindre jusqu'à 3,46 GHz tout comme les meilleurs processeurs de 2003.

L'arrivée de cette technologie a révolutionné le monde de l'algorithmie parallèle : en effet, en proposant plusieurs cœurs au grands public, le parallélisme se voit adapté à un grand nombre de logiciels conventionnels nécessitant d'utiliser toute la puissance de l'ordinateur (décodeur vidéo, navigateur web, etc.). De plus, l'utilisation d'une mémoire commune pour l'ensemble des cœurs permet de travailler des implémentations parallèles différentes. Les clusters possèdent ainsi plusieurs échelles : il y a tout d'abord le parallélisme au sein de chaque processeur au travers de ses cœurs, pour lequel, les échanges mémoires ne sont pas couteux, la synchronisation est rapide, etc., et un parallélisme entre les différents processeurs qui forment le cluster qui eux possèdent toujours les problèmes de communication inhérents aux clusters. Lorsque les clusters agrègent des machines SMP, on parlait de constellations dans le début des années 2000, ce terme tombe en désuétude au profit du terme cluster qui reste générique.

2.1.3 Quelques nouvelles approches pour accéder à des ressources de calcul intensif

En parallèle de l'évolution classique des calculateurs, de nouvelles approches plus innovantes ont vu le jour ces vingt dernières années.

2.1.3.1 Les grilles de calculs

Les années 90 voient le réseau Internet se développer considérablement. C'est grâce à lui que la possibilité de réaliser simplement et de manière fiable des communications à distance entre les ordinateurs apparait. C'est en s'appuyant sur son architecture que vont pouvoir se développer au cours des années 2000 les grilles de calcul. L'idée des grilles de calcul est de relier entre eux des clusters de plus ou moins grande taille (de quelques nœuds à plusieurs centaines) (I. Foster 2002) (I. a. Foster 2004) de manière à disposer d'une puissance de calcul mutualisée très importante (voir Figure 2-3).

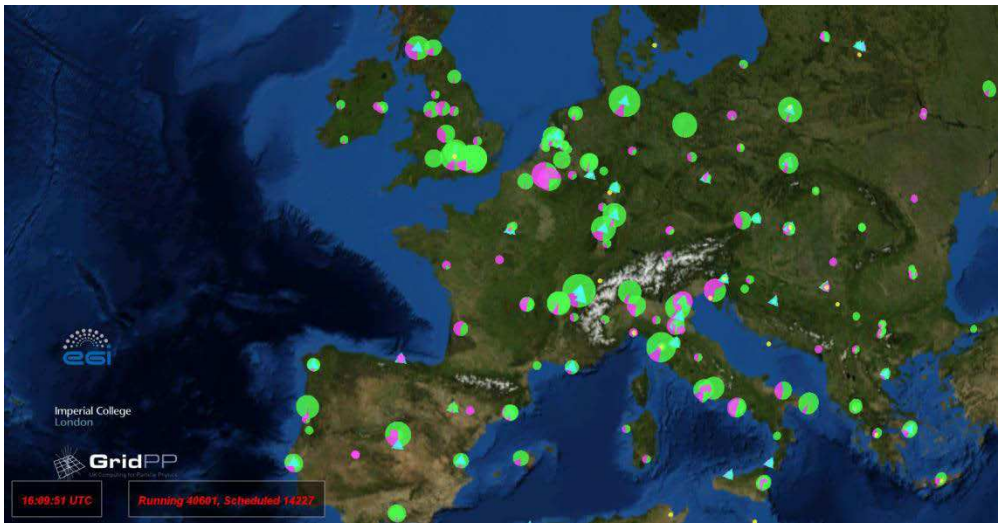


Figure 2-3 : Activité de la grille en Europe. Capture d'écran provenant du Real Time Monitor (RTM) - Imperial College London. http://rtm.hep.ph.ic.ac.uk/. Dernier accès le 29/03/2012.

Cette puissance est ensuite partagée par l'ensemble des utilisateurs du réseau qui peuvent soumettre des demandes de traitement à travers une interface unifiée qui se charge de répartir les taches sur les différents nœuds de la grille. Étant donné que l'utilisateur n'a pas connaissance à priori de l'emplacement sur lequel vont être effectués ses calculs, toute communication interprocessus peut aboutir à des baisses importantes de performances. De même, le processus de soumission peut s'avérer long et il est donc nécessaire que le temps de calcul effectif d'un traitement soit suffisamment important pour que cette latence de soumission ne soit pas trop significative par rapport au temps total d'exécution. À l'inverse, lorsque les traitements peuvent être découpés en plusieurs traitements indépendants, de durée suffisamment importante pour que la latence de soumission ne soit pas significative, comme c'est le cas par exemple lors de l'utilisation de plans d'expérience pour des simulations de taille importante, cet outil se montre d'une redoutable efficacité. Ainsi pour réaliser de la reconstruction d'images médicales, (Bitar, et al. 2006) utilisait la grille de calcul afin de permettre l'exécution de milliers de processus en parallèle. Un autre exemple lié à la bioinformatique, concerne l'implémentation de l'algorithme BLAST (Altschul, et al. 1990) sur grille de calcul. Il permet de réaliser la comparaison d'une séquence ADN inconnue avec une base de données de plusieurs milliers de séquences ADN déjà répertoriées. Chacune des comparaisons étant parfaitement indépendante et gourmande en temps de calcul, et les nombres de comparaisons étant souvent énormes dans les applications actuelles de la bioinformatique, la puissance des grilles de calcul se révèle parfaitement adaptée à cette

problématique dans la mesure où les données sur lesquelles on travaille sont bien réparties sur la grille.

2.1.3.2 Les FPGA (Field-Programmable Gate Array).

La stagnation de performances brutes des microprocesseurs au début des années 2000 a également mis en valeur les autres outils disponibles pour réaliser des calculs intensifs. Une première solution pour pallier cette limite de puissance de calcul est d'utiliser des microprocesseurs programmables : les FPGA (Field-Programmable Gate Array).

À l'inverse des processeurs x86 classiques qui fournissent une gamme d'instructions standard, ces circuits vont pouvoir être programmés spécifiquement pour n'exécuter qu'une tâche. En somme, le programme à exécuter n'est plus exécuté sur le processeur mais est directement implanté dans un circuit intégré dédié. Si l'utilisation des FPGA rajoute une certaine difficulté dans la conception du programme et nécessite de maîtriser des éléments de beaucoup plus bas niveau d'architecture que lors de la conception d'un programme classique, les gains de performances peuvent être très significatifs. Ainsi, dans un projet traitant de la comparaison de séquences ADN, l'utilisation des FPGA a permis de réaliser sur un seul nœud équipé de telles cartes un travail équivalent à celui fourni par un cluster de 50 nœuds de calcul à l'aide de l'algorithme BLAST (Lavenier, Xinchun et Georges 2006).

2.1.4 Calcul hybride, approches matérielles et logicielles

L'essoufflement des performances des microprocesseurs a également mis en lumière la puissance de calcul d'une autre forme de processeurs : les processeurs graphiques. Ces derniers sont les héritiers des processeurs supplémentaires traitant les entrées/sorties sur les premiers supercalculateurs. En effet, les processeurs graphiques ont été incorporés aux ordinateurs au début des années 80 lorsque l'utilisation des interfaces graphiques s'est répandue. Les traitements liés aux données graphiques sont en général très longs mais aussi très répétitifs : il faut souvent appliquer une opération simple à l'ensemble des pixels d'une image. De plus, ces traitements ne souffrent peu ou pas d'erreurs d'approximations : si une erreur d'arrondi mène à un pixel possédant une composante verte 0,3% plus élevée, cela ne pose aucun problème dans l'immense majorité des cas. Les GPU ont donc été développés pour réaliser des calculs très rapides en sacrifiant si nécessaire un peu de précision. Cela a conduit à la création d'une architecture massivement parallèle avec une architecture, d'une

part proche des multiprocesseurs (MIMD), et d'autre part couplée à une architecture SIMD car chaque « processeur » présent dans un GPU possède plusieurs cœurs vectoriels (SIMD).

Au début des années 2000, l'augmentation de la puissance des GPU, mais surtout l'apparition d'API (OpenGL (Woo, et al. 1999), DirectX (Bargen et Donnelly 1998)) permettant de programmer dans les « shaders » du GPU des instructions plus généralistes a amené de nouvelles utilisations pour les GPU. Dès le début des années 2000, les développeurs ont donc commencé à utiliser la puissance des GPU aussi bien pour la résolution d'équations aux dérivés partielles (Rumpf et Strzodka 2001) que pour l'implémentation d'automates cellulaires (Harris, et al. 2002) ou encore pour réaliser des opérations rapides sur d'importantes bases de données (Govindaraju, et al. 2004). Au vu des succès rencontrés, des réflexions ont également été menées sur les méthodes de programmation et les langages utilisés (Buck, et al. 2004) qui ont amené les constructeurs à travailler sur des API orientées vers le calcul généraliste.

En 2007, NVIDIA et AMD, les deux principaux constructeurs de cartes graphiques non intégrées, ont ainsi chacun proposé une nouvelle API pour permettre le développement d'applications généralistes : respectivement CUDA et ATI Stream. Un an plus tard, la première version du standard ouvert OpenCL était proposée (Khronos OpenCL Working Group and others 2008). Si l'API d'AMD est restée en retrait du fait de sa faible maturité et de l'absence de produit dédié au calcul généraliste chez AMD, les deux autres API (CUDA et OpenCL) ainsi que la production par NVIDIA de GPU dédié au calcul généraliste ont permis l'essor de la programmation d'applications généralistes sur carte graphique. Enfin, dans un autre registre, Microsoft a également proposé en 2009 Direct Compute, une API faisant partie de la collection d'API de DirectX 11. Cette dernière, de par son intégration à DirectX cible davantage les applications tirant parti de Direct3D. Si la première génération de carte graphique dédiée à la programmation généraliste sur GPU possédait de nombreux désavantages (mémoire rapide de taille insuffisante, absence de caches généralistes et lenteur des calculs en double précision entre autre) (Lindholm 2008), de nettes améliorations ont été réalisées avec l'architecture Fermi de NVIDIA (Wittenbrink 2011) tout en améliorant les performances brutes de la carte. La prochaine génération de carte graphique, nommée Kepler chez NVIDIA et Tahiti chez AMD sera gravée plus finement (28nm) et devrait fournir de l'ordre de 40% de performances supplémentaires. NVIDIA a par

ailleurs déjà annoncé vouloir produire dès 2014 des GPU gravés en 20nm (la famille Maxwell) qui pourraient encore significativement améliorer les performances.

Pour permettre aux développeurs d'exploiter le plus simplement possible de la puissance des GPU, de nouvelles approches, surcouche des API précédentes ont été développées. L'idée principale de ces solutions est de masquer toutes les contraintes liées à l'architecture particulière des GPU, en particulier l'utilisation d'une mémoire propre au GPU.

Pour les applications MATLAB, une bibliothèque de la « Parallel Computing Toolbox » permet de réaliser les calculs matriciels sur GPU sans se soucier des mécanismes mis en place. La simple utilisation de la fonction `gpuArray` pour créer une matrice implique la réalisation des calculs sur le GPU. Ainsi, dans le Code 2-1, la matrice `matriceGPU` sera automatiquement créée en mémoire sur le GPU avec les données de la première matrice créée sur le CPU. La bibliothèque propose ensuite nativement un nombre important de fonctions pouvant être réalisées sur le GPU. Mais cette bibliothèque officielle de MATLAB n'est pas la seule existante. Il en existe également deux autres : GPUMat qui est, elle, libre et gratuite et Jacket qui est propriétaire.

```
matriceCPU = rand(1000);
matriceGPU = gpuArray(matriceCPU);
```

Code 2-1 : Exemple d'initialisation de matrice sur le GPU avec la fonction gpuArray.

Ces solutions se fondent sur l'API CUDA de NVIDIA et limitent donc pour le moment l'utilisation des opérations sur GPU aux cartes graphiques NVIDIA. Mais des solutions, permettant de tirer parti de l'interopérabilité de l'API OpenCL, existent également. On trouve ainsi, entre autre, des solutions en Java, Scala ou encore en C++ (respectivement JavaCL (Chafik, 2009), ScalaCL (Chafik, 2011) et QtOpenCl (TheQtProject 2010)). À chaque fois, l'idée directrice est toujours la même : proposer une expérience utilisateur la plus proche possible de ce que pourrait être le développement de l'application sur un CPU. Dans l'exemple suivant, l'inclusion de la bibliothèque ScalaCL (première ligne du code) couplée à l'appel `.cl` lors de la création de l'intervalle `range` suffit à permettre l'utilisation de GPU pour réaliser les calculs. Cet exemple permet également de mettre en valeur la puissance de l'API OpenCL qui permet de choisir, à chaud, le meilleur outil de parallélisation (CPU ou GPU) à travers l'instruction `Context.best` dans le code suivant (Code 2-2).

```
import scalacl._
import scala.math._

object Test {
   def main(args: Array[String]): Unit = {
      // Choix du meilleur contexte : CPU ou GPU
      implicit val context = Context.best
      // Tableau ScalaCL de 100 à 100 000
      val range = (100 until 100000).cl
      // Copie vers le GPU de range automatique
      val sum = range.map(_ * 2)
            .zipWithIndex.map(p => p._1 * p._2).sum
      // Copie depuis le GPU automatique
      println("sum = " + sum)
   }
}
```

Code 2-2 : Exemple simple d'utilisation de ScalaCL (source : site officiel - http://code.google.com/p/scalacl/).

Une approche quelque peu différente a été utilisée par une équipe de Microsoft pour développer la solution C++ AMP (C++ Accelerated Massive Parallelism) (Microsoft, 2011). En effet, le choix a été fait de fournir, en plus d'une intégration complète à Visual Studio (l'IDE produit par Microsoft), une spécification complète plutôt qu'une bibliothèque. L'idée principale est de s'appuyer sur le mécanisme de boucle basée sur les intervalles introduits dans la dernière version du standard du C++ (nommé C++ 11). Ce mécanisme permet de simplement parcourir l'ensemble des éléments d'un conteneur. Dans l'exemple du Code 2-3, chacune des cinq valeurs du tableau va être parcourue une fois et sera manipulée à travers la référence « x ». Il s'agit donc simplement de multiplier le contenu du tableau par deux.

```
int mon_tableau[ 5 ] = { 1, 2, 3, 4, 5 };
for ( int & x : mon_tableau )
{
   x *= 2;
}
```

Code 2-3 : Exemple d'utilisation d'une boucle basée sur les intervalles.

Le standard C++ AMP propose d'utiliser d'une façon très semblable la fonction `parallel_for_each`. Le Code 2-4 illustre cette utilisation. Pour additionner deux tableaux `pA` et `pB` dans un troisième tableau `pC`, il est donc nécessaire d'encapsuler dans un premier temps ces trois tableaux dans des structures manipulables pour le GPU, les `array_view`. Une fois ceux-ci créés, il suffit de paralléliser sur autant d'éléments que contiennent les tableaux à l'aide de l'instruction `sum.grid` (il aurait été possible d'utiliser indifféremment `a.grid` ou `b.grid`). Enfin, au sein de la boucle, une fonction va manipuler l'indice courant à travers la variable `i` : `sum[i] = a[i] + b[i]`.

```
#include <amp.h>

using namespace concurrency;

void AddArrays(int n,
        int * pA, int * pB, int * pC)
{
   array_view<int,1> a(n, pA);
   array_view<int,1> b(n, pB);
   array_view<int,1> sum(n, pC);

   parallel_for_each(
      sum.grid,
      [=](index<1> i) restrict(direct3d)
      {
         sum[i] = a[i] + b[i];
      }
   );
}
```

Code 2-4 : Exemple d'utilisation de C++ AMP.

Si ces différentes approches sont très intéressantes, elles permettent rarement de tirer le maximum de la puissance des GPU. Lorsqu'il est nécessaire d'obtenir des performances optimales, il est donc avantageux de réaliser l'ensemble des travaux dans des API plus bas niveau tel que CUDA, ATI Stream ou OpenCL.

L'utilisation des GPU pour réaliser du calcul scientifique à l'aide de l'API de programmation CUDA étant au cœur de ma thèse, l'architecture des GPU sera détaillée dans la partie 2.1.4.3 - L'architecture des processeurs graphiques à but générique~~L'architecture des processeurs graphiques à but générique~~ et les détails de l'API dans la partie 3 - Développement sur architecture hybride à l'aide de l'API CUDA.

2.1.4.1 Les machines hybrides

Les GPU peuvent donc être très utiles pour réaliser des tâches très importantes et répétitives (réaliser un calcul complexe sur l'ensemble des points d'un espace 3D par exemple). Mais les problématiques liées à l'architecture, et en particulier la latence des accès mémoire, interdisent à l'heure actuelle de pouvoir efficacement porter tous les codes scientifiques sur GPU. Plus spécifiquement, dans les applications les plus importantes, certaines parties de l'application vont posséder des propriétés algorithmiques différentes : certaines seront plus aptes à tirer le maximum des architectures des CPU lorsque d'autres seront plus efficaces sur GPU.

C’est la raison pour laquelle, à partir de la fin des années 2000, l’utilisation de machines hybrides possédant à la fois des processeurs très puissants et des cartes GPGPU s’est généralisée. Ces machines laissent ainsi la possibilité de développer des applications qui exploitent le meilleur des CPU et des GPU. Ces machines hybrides ont eu un tel succès que dès le classement de novembre 2010 du top500 des calculateurs les plus puissants, trois des cinq premiers supercalculateurs au monde comportaient une composante GPU en plus de la composante CPU traditionnelle. Le prochain ordinateur prévu à Los Alamos aux États-Unis pour remplacer la machine « Jaguar » (la première à proposer plus d’un PetaFlops/s) sera une machine hybride délivrant plus de 20 PetaFlops/s. La marche vers la performance exaflopique a été sensiblement accélérée par l’arrivée des derniers GP-GPU. Récemment, la possibilité de proposer, au sein des serveurs lames, de racks 1U équipés de huit GPU a encore permis de faciliter l’intégration des GPU aux serveurs existants.

Pour réaliser des applications utilisant des GPU à une échelle plus petite que celle des supercalculateurs, il est maintenant possible de disposer de la puissance des GPU à bas coûts. Une station de travail équipée d’une carte graphique dédiée au calcul sur GPU ne requiert ainsi qu’un surcoût réduit grâce aux connectiques standards utilisées (PCI Express 16x) pour des performances qui peuvent être grandement améliorées (jusqu’à plus de 200x). Devant l’intérêt croissant des professionnels, les assembleurs ont commencé à proposer des stations de travail dédiées aux GPU et équipées de quatre cartes GPU (voir Figure 2-4).

Figure 2-4 : Machine cible : station de travail dédiée au calcul sur GPU.

2.1.4.2 L'approche Single Instruction Multiple Thread (SIMT)

La programmation sur carte graphique utilise une approche très proche de celle des composants SIMD (Single Instruction Multiple Data). Mais pour rendre la tâche la plus aisée possible aux développeurs, NVIDIA a introduit la notion de SIMT (Single Instruction Multiple Thread). Le principe de base est assez simple : plutôt que de manuellement choisir les opérations que l'on souhaite voir effectuer de manière vectorielle comme c'est le cas avec des instructions SSE par exemple, la programmation sur GPGPU ne nécessite que la création d'une fonction qui sera exécutée sur le GPU et qui dans la terminologie CUDA se nomme kernel. En CUDA, les fonctions faisant office de kernel sont préfixées par le mot clé `__global__` (voir Code 2-5). Ce n'est que lors de l'appel de ce kernel que l'on va préciser combien de threads doivent exécuter cette fonction en parallèle à l'aide de triples chevrons (dans l'exemple suivant, le kernel est appelé dix fois – l'explication de la valeur 1 vient juste après).

```
__global__ void monPremierKernel()
{
}

int main( int, char *[] )
{
    monPremierKernel<<< 1, 10 >>>();
}
```

Code 2-5 : Déclaration et appel d'un kernel en CUDA.

Pouvoir lancer plusieurs fois un même kernel en parallèle n'est intéressant que si l'on peut réaliser des traitements différents dans chaque kernel. L'idée de la programmation sur GPGPU est d'attribuer à chaque thread un identifiant unique qui peut être calculé dynamiquement à l'exécution. En ayant cet identifiant, il est alors possible de réaliser des tâches différentes mais surtout de permettre de traiter des données différentes. Dans le code suivant (Code 2-6), le kernel est appelé quatre fois en parallèle. Dans chaque kernel, l'identifiant du thread est retrouvé (calculé) à l'aide d'une variable proposée par l'API CUDA : `threadIdx`. Cette variable, qui possède trois composantes (x, y et z –qui sont héritées de l'usage original des GPU : le rendu de scène 3D) (voir Code 2-7) permet d'identifier de manière unique les quatre threads. Il est alors possible à chacun des threads de n'effectuer qu'un seul des quatre calculs à réaliser.

```
__global__ void foisDeux( float * inoutTab )
{
   int indiceX  = threadIdx.x;
   inoutTab[ indiceX ] = inoutTab[ indiceX ] * 2;
}

float tableau[] = { 2, 4, 6, 8 };
foisDeux<<< 1, 4 >>>( tableau );
```

Code 2-6 : Utilisation de l'API CUDA pour calculer un identifiant unique pour chaque thread.

```
struct uint3
{
   unsigned int x;
   unsigned int y;
   unsigned int z;
};

typedef uint3 dim3;
```

Code 2-7 : Définition du type dim3.

Bien entendu, l'utilisation de quatre threads serait contreproductive et il est souvent utile de lancer des milliers de threads en parallèle pour tirer le maximum d'un GPU. Ces milliers de threads ne seront pas pour autant tous exécutés en même temps mais beaucoup seront ordonnancés en même temps. En effet, comme cela a été expliqué précédemment, un GPU possède plusieurs éléments assez proches des processeurs, les Streaming Multiprocessors (leurs explications détaillées seront données plus tard), ils permettent d'exécuter des threads de façon vectorielle. Chacun d'eux ne peut ordonnancer qu'un nombre limité de threads en même temps (avec un maximum de 1536 threads dans l'architecture actuelle au moment de la rédaction de ce manuscrit) et c'est la raison pour laquelle les threads sont découpés en blocs de threads (voir Figure 2-5).

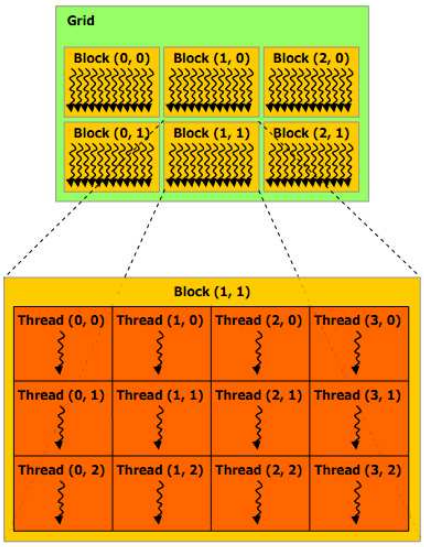


Figure 2-5 : Représentation d'une grille de 6 blocs (x=3, y=2) contenant chacun 12 threads (x=4, y=3 et z=1) pour un total de 72 threads exécutés (source : documentation officielle NVIDIA).

Un bloc est un conteneur à trois dimensions (x, y et z) de threads (dans la figure précédente, la dimension z vaut 1 donc les blocs ne sont qu'en deux dimensions). Tous les threads d'un bloc ont la garantie d'être ordonnancés en même temps, ce qui n'est pas le cas des différents blocs entre eux (ne serait-ce que dans le cas où il y a trop de blocs pour la capacité du GPU, certains blocs devront attendre la fin de l'exécution des autres). Pour contenir les blocs, CUDA utilise une grille qui n'est autre qu'un conteneur à deux dimensions (x et y) de blocs. Au lancement d'un kernel sur le GPU, on configure donc la taille des blocs dans les trois dimensions et la taille de la grille dans les deux dimensions. Dans l'exemple suivant (voir Code 2-8), la grille est maintenant de taille 10 par 10 (elle contient 100 blocs –z doit toujours valoir 1 pour conserver une grille en deux dimensions) et les blocs sont des conteneurs de 32 par 4 par 2 (256) threads. L'appel ne se fera donc plus à l'aide de 72 threads comme dans le cas de la figure précédente mais avec 25600 threads.

```
dim3   dimensionBloc( 32, 4, 2 );      // 256 threads par bloc
dim3   dimensionGrille( 10, 10, 1 );  // 100 blocs (, 1 optionnel)
monTroisiemeKernel<<< dimensionGrille, dimensionBloc >>>( ... );
```

Code 2-8 : Appel d'un kernel en CUDA sur un grand nombre de threads répartis en blocs et grille.

Bien entendu, des variables permettent de calculer l'indice absolu d'un thread au sein de l'ensemble de la grille. `threadIdx` donne la position selon x, y et z du thread dans le bloc et `blocIdx` fait de même pour le bloc dans la grille. De plus, afin d'éviter le passage de

paramètres supplémentaires, les dimensions des blocs et de la grille sont données respectivement par les variables `blocDim` et `gridDim`.

À l'aide de ces variables, il est donc possible d'implémenter des kernels pouvant être fortement parallélisés en s'exécutant sur un grand nombre de threads en parallèle. Ces threads peuvent alors soit exécuter des instructions différentes en fonction de l'identifiant du thread, soit traiter des données différentes. Il est très important, autant que possible, que le parallélisme se trouve au niveau des données. En effet, comme cela a été dit plus haut, une partie de la puissance de calcul des GPU suit une approche SIMD. C'est la raison pour laquelle, les threads sont regroupés en « warp ». Un warp est actuellement composé de 32 threads (ce nombre pourrait évoluer à l'avenir mais la notion de warp subsisterait) et son objectif est d'exécuter les mêmes instructions pour ses 32 threads mais sur des données différentes (SIMT). Si une divergence de traitement, c'est-à-dire une exécution de code différente apparait sur un thread dans un warp, tous les traitements différents sont alors réalisés de façon séquentielle sur le GPU. Ce cas de figure a un impact très important sur les performances globales de l'application : si les 32 threads d'un warp, qui devaient s'exécuter en parallèle, s'exécutent en séquentiel, il est évident que les performances seront amoindries. Nous verrons par contre comment nous avons proposé de tirer bénéfice des warps pour combiner les approches MIMD et SIMT (Passerat, et al. 2011).

D'autres spécificités des GPU rentrent également en compte lorsqu'il s'agit de développer une application. Afin de bien comprendre les autres problématiques de la programmation selon l'approche SIMT sur carte graphique, il est nécessaire de comprendre les spécificités de l'architecture.

2.1.4.3 L'architecture des processeurs graphiques à but générique

Comme cela a été introduit précédemment, l'architecture des GPGPU est un mélange subtil d'architecture vectorielle (SIMD) et de parallélisme pur[4] (MIMD).

Le parallélisme pur, autorisant des calculs indépendants, se caractérise par la présence de « Streaming Multiprocessors » (SM dans la suite) selon la terminologie de NVIDIA. Selon les cartes graphiques, le nombre de SM présents ainsi que la structure de chacun d'eux évolue

[4] Le parallélisme pur suppose plusieurs processeurs ou cœurs traitant des instructions différentes.

mais l'organisation générale reste la même. Ainsi, chaque SM qui compose la carte graphique possède au moins un ordonnanceur de warps qui lui est propre.

Il est donc possible pour deux SM différents d'exécuter deux instructions différentes à un moment donné. Il s'agit donc bien de parallélisme pur entre les SM d'une même carte qui peuvent exécuter des instructions différentes. Dans la Figure 2-6, représentant l'architecture d'une carte graphique Fermi, chaque SM possède deux ordonnanceurs qui permettent d'exécuter deux instructions différentes lors d'un même cycle d'horloge dans un même SM. Il est alors possible d'exécuter au maximum un nombre d'instructions différentes égales à deux fois le nombre de SM. Étant donné que le nombre de SM reste limité au sein d'un GPU (on en trouve entre 14 et 16 dans les cartes d'architecture Fermi proposé par NVIDIA et 30 dans les cartes d'architecture Tesla précédente), la parallélisation selon cette composante l'est aussi. La fréquence des processeurs des GPU étant bien plus faible que celle des CPU (de l'ordre de quatre fois plus faible par rapport à un processeur actuel performant) et les accès mémoire étant encore très lents, une parallélisation qui utiliserait uniquement cette possibilité serait rarement suffisante.

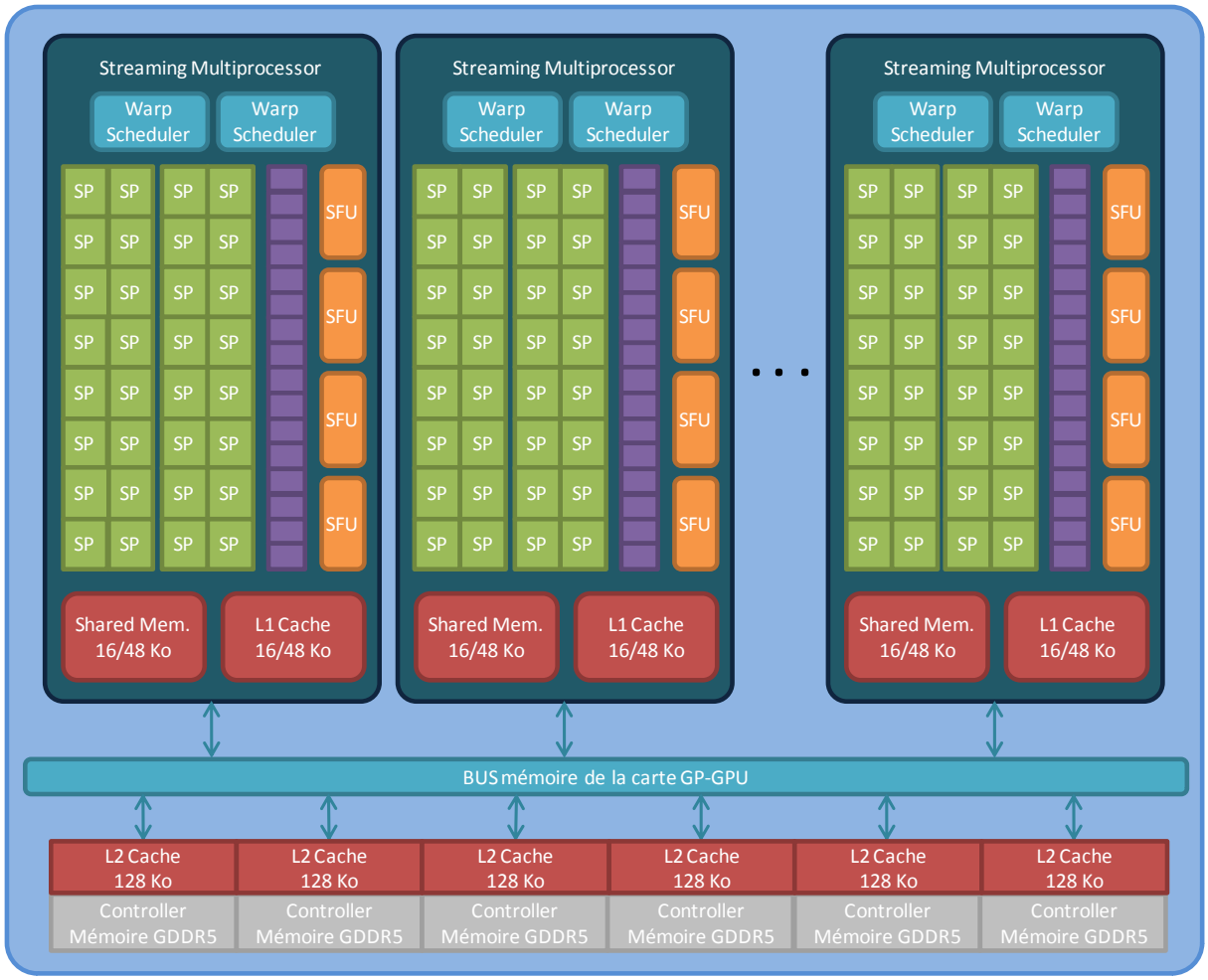


Figure 2-6 : Représentation simplifiée de l'architecture d'un GP-GPU (modèle Fermi).

Une deuxième composante a donc été introduite au sein de chaque SM pour permettre d'accélérer les calculs. Ainsi, au lieu de disposer d'une unité de calcul par ordonnanceur, chaque SM dispose d'un grand nombre d'entre elles. Dans l'architecture Fermi, un SM comporte 32 unités de calcul flottant : 2 groupes de 16 unités. À chaque cycle, il est alors possible pour une instruction d'être exécutée en parallèle sur 16 unités de calcul à la fois, afin de traiter des données différentes : 448 instructions (14 SM comportant chacun 2x16 unités de calcul) peuvent donc être exécutées en parallèle, ce qui est beaucoup plus intéressant.

En plus de l'aspect calculatoire des GPGPU, un autre aspect très important pour l'obtention de bonnes performances concerne les accès mémoire. Comme cela a été dit plus haut, les accès sont beaucoup plus lents sur GPU que sur CPU. La latence pour un accès en mémoire globale du GPU (la RAM du GPU) peut atteindre plusieurs centaines de cycles d'horloges. Plusieurs mécanismes sont présents pour pallier le plus possible ces latences importantes. Le

premier est l'ordonnancement des warps. En effet, les GPGPU bénéficient d'ordonnanceurs très performants pouvant très rapidement changer de contexte. Ainsi, si certains warps sont en attente d'accès mémoire, l'ordonnanceur donne la main à des warps prêts à exécuter des instructions. Ceci a un impact important sur la conception d'applications sur GPU car cela signifie qu'il est nécessaire de disposer d'un nombre de threads à exécuter sur le GPU bien supérieur au nombre d'unités de calcul présentes sur celui-ci. Mais la capacité d'un SM à ordonnancer les warps n'est pas sans limite et il n'est pas possible sur les cartes de dernière génération d'ordonnancer plus de 48 warps (1536 threads). Il n'est donc pas toujours possible de compenser ces fortes latences par la simple utilisation de l'ordonnanceur du GPU. En particulier, si le rapport entre le nombre d'opérations arithmétiques et le nombre d'accès mémoire est petit, l'ordonnancement seul ne permettra pas de masquer la latence. On trouve dans la littérature le ratio minimum de 10 opérations arithmétiques pour 1 accès mémoire (Kirk, Wen-mei et Hwu, Programming massively parallel processors: a hands-on approach 2010) lorsque les SM sont chargés autant que possible pour pouvoir masquer la latence.

Pour traiter les cas où la latence est trop importante, les constructeurs de GPU ont progressivement introduit différentes modifications architecturales afin d'améliorer les accès mémoire. La génération précédente de GPGPU comportait ainsi, en plus de la mémoire globale accessible en lecture et en écriture, d'autres types de mémoires. Les deux plus anciennes sont la mémoire constante et la mémoire de texture (voir Figure 2-7). Pour être précis, il faudrait plutôt parler de cache constant et de cache de texture car ces deux mémoires sont implémentées au sein même de la mémoire globale. La différence se trouve dans la présence d'un cache de taille limitée qui permet d'accéder plus rapidement à la mémoire lorsque celle-ci est déjà présente dans le cache (de quelques cycles de latences pour le cache constant, jusqu'à une centaine pour la mémoire de texture contre plusieurs centaines lors d'accès non cachés). Contrairement à la mémoire globale qui est de grande taille (plusieurs Go sur les cartes actuelles) et permet un accès en lecture comme en écriture, ces deux caches sont de taille limitée (plusieurs dizaines de Ko maximum) et ne sont accessibles qu'en lecture. Tous les accès à la mémoire globale ne peuvent donc être substitués par des accès passant par un de ces deux caches, que ce soit dû à la nécessité d'utiliser une taille de mémoire trop importante ou à des accès en écriture obligatoire. De

manière générale, lors d'un calcul scientifique sur une carte graphique, les résultats ne peuvent être copiés vers la mémoire du CPU que depuis la mémoire globale, l'utilisation exclusive de cache accessible en lecture seule est donc impossible.

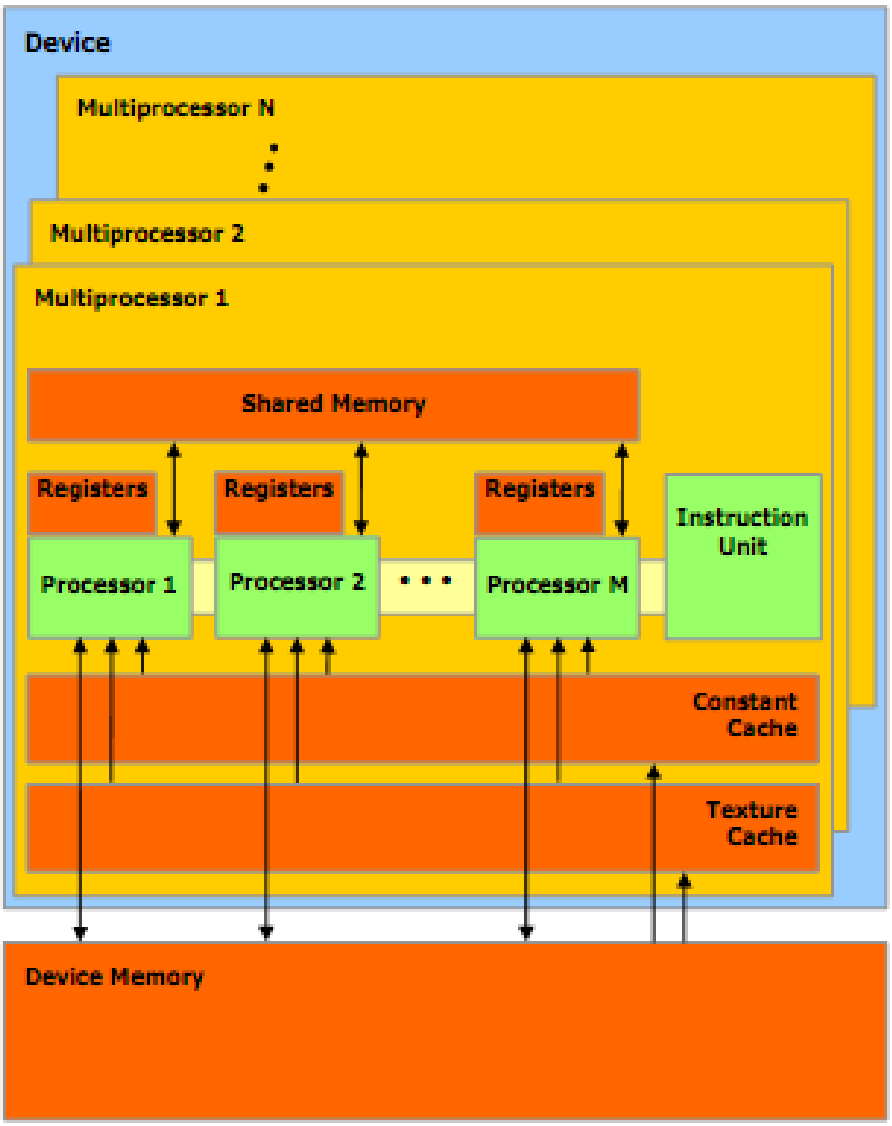


Figure 2-7 : Organisation des mémoires des GPU (source : documentation officielle NVIDIA).

Une autre mémoire permet des améliorations significatives de la performance des programmes s'exécutant sur GPU, il s'agit de la mémoire dite partagée (au sein d'un même SM). Cette mémoire, contrairement aux précédentes, n'est pas commune à l'ensemble de la carte mais est propre à un Streaming Multiprocessor. Ainsi, seul les threads appartenant à un même bloc de threads (et donc étant ordonnancés en même temps sur un SM) vont pouvoir partager cette mémoire. Si ce dernier point est indéniablement un énorme

inconvénient, cette mémoire permet de grandement accélérer les applications car elle possède des caractéristiques d'accès particulièrement rapides : un accès, en lecture ou en écriture, ne subit ainsi aucune latence. Mais cette mémoire, présente au niveau des SM n'est pas accessible depuis le CPU, il est donc généralement nécessaire, pour tirer parti de ses excellentes caractéristiques, de tout d'abord charger des données depuis la mémoire globale vers cette mémoire partagée. Or, une nouvelle fois, les accès en mémoire globale sont très lents. Pour que le chargement en mémoire partagée soit rentable, il est donc nécessaire que la même donnée chargée en mémoire soit utilisée par plusieurs threads d'un même bloc. Ainsi, les threads partageant des accès à une même donnée ne la chargent (lentement) qu'une fois en mémoire partagée depuis la mémoire globale puis réalisent de multiples accès en mémoire partagée sans aucune latence. À la fin du traitement, il est possible que la donnée mise à jour en mémoire partagée soit écrite (lentement une nouvelle fois) en mémoire globale. Mais cette écriture en mémoire globale ne sera faite qu'une fois par bloc de threads.

La mémoire partagée, si elle n'est pas utilisable efficacement pour tous les problèmes, permet donc très souvent de fortement diminuer le coût global des accès mémoire pour l'application. Malheureusement, sa petite quantité (de l'ordre de quelques dizaines de Ko une nouvelle fois) limite le type des applications qui pourront en tirer bénéfice et il est parfois nécessaire d'accéder directement à la mémoire globale.

Il existe de plus des accès à la mémoire globale « obligatoires », ne serait-ce que pour charger les données en mémoire partagée puis écrire les résultats, les constructeurs cherchent à en minimiser l'impact sur le temps d'exécution globale d'une application. C'est la raison pour laquelle la dernière architecture développée par NVIDIA (l'architecture Fermi) a rajouté des systèmes de cache généraliste (L1 et L2) permettant d'améliorer ces accès. Comme le montre la Figure 2-6, ces caches se trouvent d'une part au niveau des contrôleurs de mémoires pour le cache L2 et implémenté en commun avec la mémoire partagée au niveau des SM pour le cache L1. Ces deux caches permettent, lorsque les données s'y trouvent de grandement diminuer la latence. Le cache L1 en particulier possède des performances similaires à la mémoire partagée et, hors défaut de cache, les accès se font donc sans aucune latence.

Cette partie n'ayant pour but que d'introduire la programmation sur GPU, une description plus complète de l'API CUDA sera développée dans le chapitre 3. La dernière version du kit de développement CUDA utilise par exemple un compilateur basé sur LLVM (Low Level Virtual Machine). Il s'agit non pas d'un simple compilateur mais de toute une infrastructure de compilation modulable. Ce changement de politique, qui peut paraître anecdotique, pourrait permettre de réaliser beaucoup plus simplement la compilation de code CUDA vers d'autres architectures comme les cartes graphiques AMD.

Par ailleurs, des outils de plus en plus performants apparaissent pour faciliter la parallélisation sur GPU de code existant. À ce titre, il est possible de citer deux initiatives françaises : Par4All[5] et HMPP[6]. Par4all est un outil de parallélisation automatique de code C et Fortran en code CUDA, OpenCL et OpenMP développé dans le cadre du projet HPC Project. Si les performances de cette approche ne sont pas encore équivalentes à celle d'un code parallélisé manuellement, elles s'en rapprochent de plus en plus (Amini, et al. 2011). HMPP est une implémentation proposée par l'entreprise CAPS du standard OpenHMPP, qui se veut le pendant d'OpenMP pour la parallélisation à l'aide d'accélérateurs matériels (GPU en tête). L'idée n'est pas ici de paralléliser un code automatiquement mais de proposer des directives qui permettent de spécifier la façon dont l'on souhaite utiliser les accélérateurs matériels sur certaines parties du code.

Enfin, l'arrivée des APU (Accelerated Processing Unit) qui sont des processeurs proches des microprocesseurs classiques auxquels sont adjoints des unités de calculs supplémentaires, généralement sous la forme de GPU. Le fort potentiel de cette technologie réside dans l'utilisation de mémoire beaucoup plus proche entre le CPU et le GPU permettant de grandement diminuer les temps de copie entre CPU et GPU (Daga, Aji et Feng 2011).

Si l'utilisation d'architecture parallèle et de matériels plus puissants permettent souvent d'améliorer les performances d'une application, ce n'est pas pour autant la seule solution. Il

[5] Site officiel de Par4All : http://www.par4all.org/. Dernier accès le 21/03/2012.
[6] Site officiel d'HMPP : http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36. Dernier accès le 21/03/2012.

arrive parfois que l'utilisation d'un algorithme différent pour résoudre un même problème donne des résultats spectaculaires. C'est le cas par exemple des automates cellulaires et de l'algorithme Hash-Life.

2.2 Algorithmique à haute performance et automates cellulaires

2.2.1 Historique des automates cellulaires

Un automate cellulaire consiste en un espace (d'une ou plusieurs dimensions) discrétisé en cellules identiques connectées localement (qui peuvent prendre différents états) et d'un ensemble de règles sur ces cellules (permettant l'évolution de l'automate) (Wolfram, A new kind of Science 2002). Le calcul de l'état, à l'itération suivante, d'un automate cellulaire depuis un état courant est réalisé en calculant le nouvel état de chaque cellule indépendamment. Le nouvel état de chacune des cellules est fonction de leur état courant et de celui de toutes les cellules présentes dans son voisinage. À partir des données sur ces différents états, les règles propres à l'automate permettent de déterminer le nouvel état de la cellule. Parmi les voisinages les plus utilisés et les plus connus, on trouve le voisinage de von Neumann et le voisinage de Moore (voir Figure 2-8~~Figure 2-8~~). Le premier ne comprend que les cellules ayant une arête commune avec la cellule étudiée et le second comporte toutes les cellules qui possèdent au moins un sommet en commun avec la cellule étudiée.

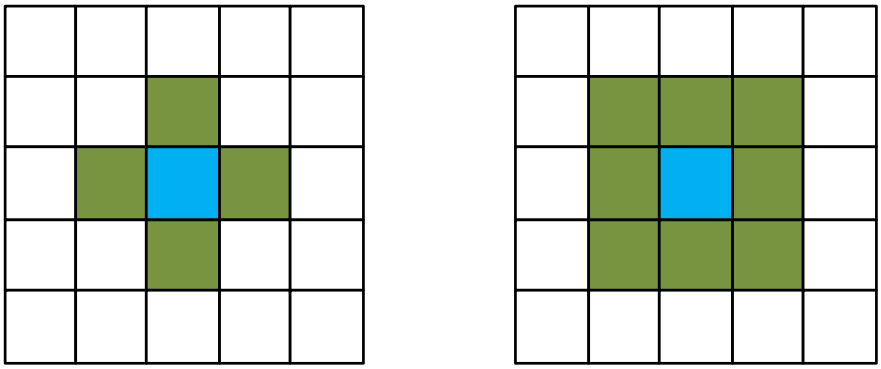


Figure 2-8 : Voisinage de von Neumann (à gauche) et de Moore (à droite).

Le concept d'automate cellulaire est fortement associé à von Neumann. Dans le but de modéliser facilement des systèmes auto-réplicatifs, il crée, sur les conseils de son collègue Ulam travaillant sur une modélisation à base de grille, ce qui se rapproche le plus du premier automate cellulaire (Von Neumann 1966). Il s'agit d'un automate en deux dimensions dont

les cellules possèdent 29 états. À partir d'une configuration bien précise de cellules, celui-ci était capable de se reproduire lui-même indéfiniment.

Dès 1969, Heldlung pose les bases de l'étude mathématique des automates cellulaires à l'aide d'une compilation des articles les plus pertinents déjà parus sur le sujet (Hedlund 1969). Les automates cellulaires ont par la suite été grandement popularisés par le « Jeu de la Vie » (Game of Life ou Life en anglais) créé par Conway et publié par Gardner (Gardner 1970). Cet automate est beaucoup plus simple que celui proposé par von Neumann. Chaque cellule ne comprend que deux états : en vie ou morte, et l'automate n'utilise que quelques règles couplées à un voisinage de Moore :

- Si une cellule vivante possède 2 ou 3 voisines vivantes, elle reste en vie.
- Si une cellule morte possède exactement 3 voisines vivantes, elle devient vivante.
- Dans tous les autres cas, la cellule reste ou devient morte.

Si le jeu de la vie peut paraitre simple, voire simpliste au premier abord, il n'en est pas moins une puissant métaphore : il a ainsi été montré que le Jeu de la Vie était Turing complet (Berlekamp, Conway et Guy 2004).

En plus du Jeu de la Vie, beaucoup d'automates cellulaires permettent de simuler des comportements plus ou moins complexes. Silverman a ainsi proposé un automate cellulaire « WireWorld » en 1987 permettant de simuler le comportement des circuits électriques (Dewdney 1990). Les cellules possèdent quatre états et des règles de transition simples :

- Vide : une cellule vide reste vide indéfiniment.
- Tête de l'électron : une cellule à l'état « tête de l'électron » passe dans l'état « queue de l'électron » à l'itération suivante.
- Queue de l'électron : une cellule à l'état « queue de l'électron » passe dans l'état « conducteur » à l'itération suivante.
- Conducteur : une cellule à l'état « conducteur » passe dans l'état « tête d'électron » si une cellule voisine est dans l'état « tête de l'électron » (le voisinage Moore est une nouvelle fois utilisé).

Un autre phénomène simulé à l'aide des automates cellulaires qui va nous intéresser par la suite est le milieu excitable. Un milieu excitable est un système dynamique non linéaire qui permet à une onde de se propager à un instant 't' mais interdit la propagation d'une

nouvelle onde au même endroit avant qu'un certain laps de temps ne soit écoulé (Ermentrout et Edelstein-Keshet 1993). Pour simuler ce phénomène à l'aide d'un automate cellulaire, seuls trois états sont nécessaires :

- Active : une cellule « active » devient « réfractaire » après un certain laps de temps.
- Réfractaire : une cellule « réfractaire » devient « au repos » après un certain laps de temps.
- Repos : une cellule « au repos » reste dans cette état jusqu'à qu'une de ses cellules voisines devienne active ; auquel cas, elle devient active à son tour.

Plusieurs phénomènes réels peuvent ainsi être modélisés. C'est en particulier le cas de la modélisation de l'activité électrique du cœur humain (Siregar, Sinteff, et al. 1998). L'état « actif » correspond alors à la dépolarisation de la cellule électrique, l'état « réfractaire » correspond à la repolarisation et l'état « repos » correspond à la cellule au repos en attente d'un signal électrique.

Les automates cellulaires peuvent de nos jours être utilisés pour simuler des comportements beaucoup plus complexes grâce aux puissances de calcul actuelles. Il est ainsi possible de simuler des mécanismes biologiques plus complexes, tels que la croissance d'une tumeur dans un organe (Alarcon, Byrne et Maini 2004) ou la représentation en trois dimensions dans un milieu anisotrope de la dépolarisation et de la repolarisation du myocarde (Siregar, Sinteff, et al. 1998). Le comportement émergent majeur de ce type de modèle est le phénomène de réentrée qui est à l'origine d'un grand nombre de troubles du rythme.

Mais les automates à plusieurs dimensions ne sont pas les seuls à avoir été étudiés en détail. Wolfram s'est ainsi focalisé sur un type d'automate à une seule dimension (Wolfram, Universality and complexity in cellular automata 1984) et a montré que des comportements émergents complexes pouvaient apparaître. De nombreux problèmes ont ainsi pu être modélisés à l'aide d'automates cellulaires à une dimension (Mitchell 1996) tel que le « Firing Squad Problem » (Moore 1964) ou le calcul arithmétique en parallèle (Steiglitz, Kamal et Watson 1988).

2.2.2 L'algorithme Hash-Life

Pour simuler le comportement d'un automate cellulaire, plusieurs algorithmes peuvent être utilisés. Le plus simple consiste à parcourir l'ensemble des cellules et, pour chacune d'elles,

calculer son prochain état en fonction de son état courant et de celui de ses cellules voisines. Cette solution peut être efficace pour certaines catégories de problèmes. Elle peut également être parfaitement acceptable lorsque les problèmes sont de petite taille ou lorsque les temps de calcul liés à l'automate cellulaire ne sont pas significatifs par rapport au temps d'exécution global de l'application. Dans ces cas, il est alors inutile de perdre du temps à améliorer un comportement qui est déjà satisfaisant. Mais ce n'est pas toujours le cas et il est parfois très intéressant de diminuer le temps de calcul nécessaire pour réaliser les calculs sur les automates cellulaires de façon à diminuer le temps d'exécution global de l'application.

Lorsque c'est le cas, plusieurs solutions s'offrent au développeur et dépendent généralement de l'utilisation qui est faite des automates cellulaires et en particulier du type d'automate cellulaire utilisé. Ainsi, si l'algorithme utilisé est « WireWorld », l'évolution bien particulière des états permet de grandement limiter les calculs car les cellules vides n'évolueront jamais vers un autre état et ne seront jamais impliquées dans le changement d'état d'une autre cellule. Parcourir l'ensemble des cellules n'est donc pas forcément opportun dans ce cas. Conserver la liste des cellules non vides ainsi que celle de leurs voisins non vides peut permettre de minimiser les calculs.

Si des algorithmes spécifiques peuvent ainsi être développés pour des automates cellulaires particuliers, il existe aussi des algorithmes généralistes permettant dans certains cas l'amélioration des performances. C'est le cas de l'algorithme Hash-Life qui va nous intéresser ici.

L'algorithme Hash-Life se fonde sur le principe de la mémoïzation. Ce principe a été introduit par Michie en 1968 (Michie 1968). Il introduit ainsi son article :
« Il serait intéressant que les ordinateurs puissent apprendre de leurs expériences et ainsi automatiquement améliorer l'efficacité de leurs propres programmes à l'exécution. »

Cette phrase résume très bien le principe de la mémoïzation qui est de conserver le résultat d'un calcul afin qu'il soit réutilisé directement si celui-ci venait à être de nouveau nécessaire.

La suite de Fibonacci implémentée récursivement[7] est un bon exemple de fonction pouvant tirer profit de la mémoïzation. Pour rappel, la suite de Fibonacci `fibonacci( n )` vaut 0 pour « n » valant 1, 1 pour « n » valant 2 et vaut `fibonacci( n - 1 ) + fibonacci( n - 2 )` pour toutes autres valeurs de n > 2 . Une implémentation récursive est la suivante (Code 2-9) :

```
int fibonacci( int n )
{
   if ( n == 0 )
   {
      return 0 ;
   }
   else if ( n == 1 )
   {
      return 1;
   }
   else
   {
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   }
}
```

Code 2-9 : Implémentation récursive de la fonction de Fibonacci.

On s'aperçoit très facilement que cette implémentation peut appeler un grand nombre de fois la fonction `fibonacci` avec les mêmes paramètres pour réaliser un seul calcul de la valeur de `fibonacci`, pour un `n` donné. Ainsi, l'appel de la fonction avec comme paramètre n=7 va réaliser :

- 1 appel à `fibonacci( 7 )` (l'appel initial obligatoire).
- 1 appel à `fibonacci( 6 )`.
- 2 appels à `fibonacci( 5 )`.
- 3 appels à `fibonacci( 4 )`.
- 5 appels à `fibonacci( 3 )`.
- 8 appels à `fibonacci( 2 )`.
- 13 appels à `fibonacci( 1 )`.
- 8 appels à `fibonacci( 0 )`.

Soit un total de 41 appels de la fonction `fibonacci`. De manière générale, cela produit `2 x fibonacci( n ) - 1` appels pour tout n > 0 (Robertson 1999). Or, en utilisant le

[7] Une implémentation récursive de la suite de Fibonacci n'est pas, et de loin, l'implémentation la plus efficace mais ce n'est pas le sujet abordé ici.

principe de la mémoïzation pour sauver le résultat de la fonction `fibonacci` pour une valeur de « n » donnée, un seul appel pour un « n » donné est nécessaire. Si le gain peut paraitre minime pour de petites valeurs de « n », il faut garder en mémoire que la fonction de Fibonacci croît très vite. Ainsi, pour n = 20, 13529 appels seront nécessaires sans mémoïzation et seulement 20 avec.

Bien entendu, l'utilisation de la mémoïzation a un coût, aussi bien en empreinte mémoire qu'en temps de calcul. Le coût mémoire est lié au stockage du résultat de la fonction. Le coût en temps de calcul, qui peut très vite être rentabilisé, vient des traitements supplémentaires nécessaires dans la fonction pour vérifier si le calcul a déjà été effectué et pour stocker le résultat (après l'avoir calculé normalement) dans le cas contraire (voir Code 2-10).

```
int fibonacci( int n )
{
   if ( n a déjà été calculé )
   {
      return la valeur stockée;
   }
   else
   {
      if ( n == 0 )
      {
         return 0 ;
      }
      else if ( n == 1 )
      {
         return 1;
      }
      else
      {
         stocker fibonacci( n - 1 ) + fibonacci( n - 2 );
         return la valeur stockée;
      }
   }
}
```

Code 2-10 : Pseudocode de l'implémentation à l'aide de la mémoïzation de la fonction de Fibonacci.

Transposée aux automates cellulaires, c'est la fonction de calcul de l'état suivant qui va utiliser la mémoïzation. Afin d'accélérer le plus possible les calculs, l'algorithme Hash-Life (Gosper 1984) ne se contente pas de limiter l'utilisation de la mémoïzation à une fonction qui prendrait l'état d'une cellule et de son voisinage et retournerait le nouvel état de la cellule. L'idée est de sauvegarder l'évolution de la configuration de cellules de tailles diverses (la Figure 2-9~~Figure 2-9~~ montre plusieurs configurations différentes de taille 8x8), pouvant aller jusqu'à la totalité de l'espace de simulation. De cette manière, si une

configuration de cellules est souvent retrouvée dans la simulation (comme ce pourrait être le cas de la configuration n°1 qui est vide ci-dessous), beaucoup de temps de calcul peut être économisé ; en particulier si les configurations retrouvées sont de tailles importantes.

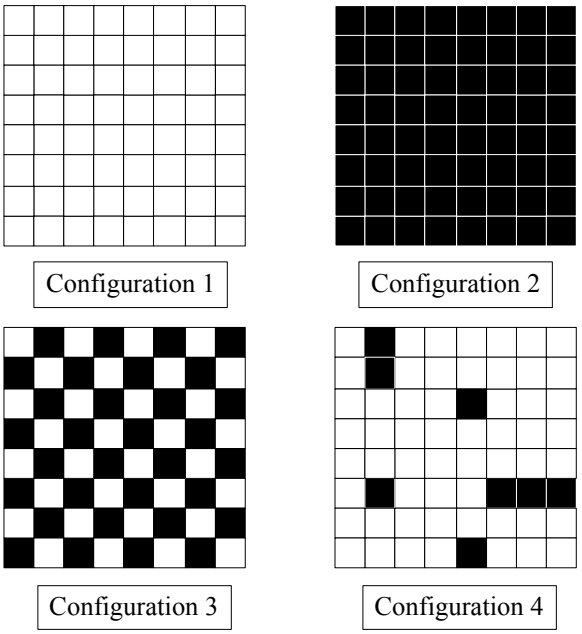


Figure 2-9: Différentes configurations de cellules possibles

Mais le mécanisme de mémoïzation n'est pertinent que si les configurations déjà connues sont retrouvées rapidement. Pour arriver à un tel résultat, deux mécanismes sont mis en place :

- Une organisation de l'espace permettant de facilement retrouver les configurations de cellules déjà rencontrées.
- Une organisation des données permettant un accès très rapide aux configurations.

Pour retrouver les configurations, l'algorithme Hash-Life organise l'espace à l'aide de quadtrees complets. Un quadtree est une structure de données sous forme d'arbre où chaque nœud est soit une branche qui possède entre 1 et 4 nœuds fils (4 dans le cas d'un quadtree complet –voir Figure 2-10~~Figure 2-10~~), soit une feuille terminale.

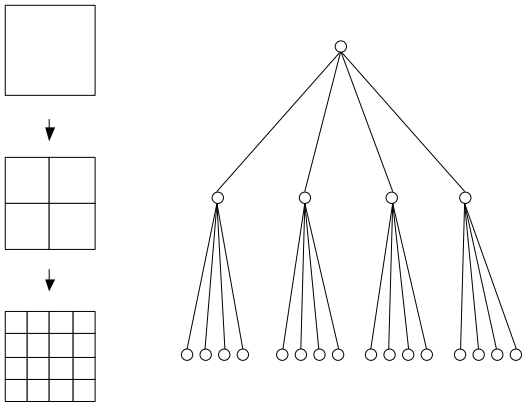


Figure 2-10 : Représentation en quadtree complet.

Pour tirer parti de cette structure de données, une configuration n'est instanciée qu'une seule fois. Il faut donc pouvoir la retrouver très rapidement, c'est la raison de l'utilisation de tables de hachage (Maurer et Lewis 1975) qui ont donné leur nom à l'algorithme. Deux tables de hachage au minimum sont nécessaires :

– La première stocke l'adresse des différentes branches terminales (composées de quatre feuilles). Ainsi, à partir de la valeur de quatre cellules (la cellule supérieure gauche, la cellule supérieure droite, la cellule inférieure gauche et la cellule inférieure droite), il est possible de récupérer l'adresse mémoire de la configuration équivalente. Dans l'exemple de la Figure 2-11, la recherche d'une branche terminale comportant uniquement des cellules vides sauf dans le coin inférieur gauche retournera la configuration de cellule d'adresse « c ».

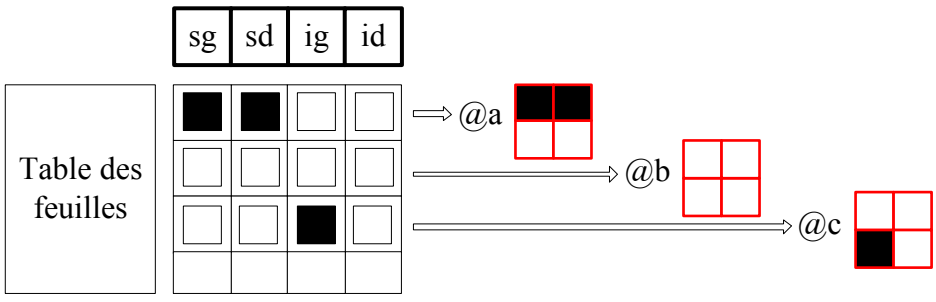


Figure 2-11 : Table de hachage des branches terminales : la clé est composée de l'état des quatre cellules et la valeur est l'adresse de la configuration correspondante. (sg : supérieur gauche ; sd : supérieur droit ; ig : inférieur gauche ; id inférieur droit)

– La seconde stocke l'adresse des différentes branches non terminales (composées de quatre branches). Il est alors possible, à l'aide de l'adresse mémoire des quatre branches qui composent une branche de retrouver son adresse en mémoire.

Dans l'exemple de la Figure 2-12, si l'on recherche la configuration 8x8 composée uniquement de cellules vides, on va d'abord chercher l'adresse mémoire de la branche finale possédant quatre cellules vides dans la première table de hachage (Figure 2-11), ce qui va retourner l'adresse « b ». On recherche ensuite l'adresse de la configuration composée de quatre configurations de 2x2 cellules vides, donc on vient de trouver l'adresse en mémoire (« b »), ce qui retourne la configuration d'adresse mémoire « e » dans la table des branches. Enfin, la recherche de la configuration comportant quatre fils de 4x4 (d'adresse mémoire « e ») retourne la configuration d'adresse mémoire « f ».

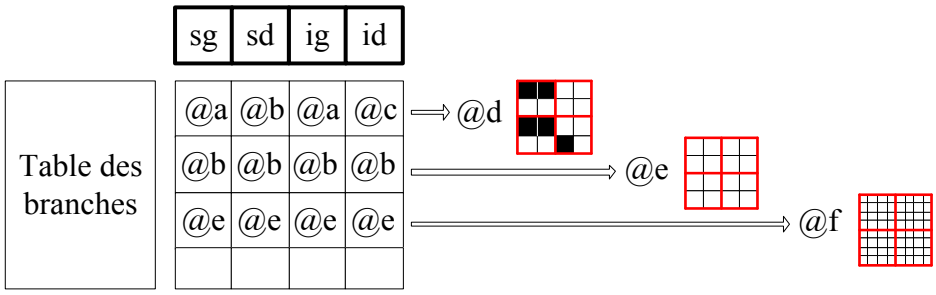


Figure 2-12 : Table de hachage des branches non terminales : la clé est composée de l'adresse des quatre branches filles et la valeur est l'adresse de la configuration correspondante. (sg : supérieur gauche ; sd : supérieur droit ; ig : inférieur gauche ; id inférieur droit)

Bien entendu, lorsqu'une configuration n'est pas disponible en mémoire, celle-ci est créée et ajoutée dans la table de hachage correspondante. Ainsi, au démarrage de l'application, les deux tables sont vides et s'enrichissent peu à peu de configurations différentes.

Cette stratégie permet de diminuer l'espace mémoire nécessaire car un nombre limité de configurations est généralement très souvent répété. En particulier, lorsque l'on traite des espaces très vastes comportant une grande proportion de cellules vides, le coût de stockage de ces cellules vides peut être rendu négligeable à l'aide d'une telle structure de données.

À l'aide de cette stratégie de stockage qui permet de retrouver les configurations déjà manipulées, il est alors possible (en sauvegardant pour chaque configuration la future configuration, résultat d'une itération) de n'effectuer qu'une fois ce calcul (voir Code 2-11).

```
class Branche : public Noeud
{
    // Pointeur de Noeud correspondant aux 4 fils.
    Noeud *   supGauche_;
    Noeud *   supDroit_;
    Noeud *   infGauche_;
    Noeud *   infDroit_;

    // Pointeur vers la configuration résultat.
    Noeud *   resultat_;
};
```

Code 2-11 : Extrait de la déclaration de la classe Branche dans l'algorithme Hash-Life en 2D.

De cette façon, l'algorithme, pour trouver le résultat d'une itération sur une configuration donnée, est très simple :

- Si le résultat a déjà été calculé, le retourner.
- Sinon, le calculer, le sauvegarder et le retourner.

Ainsi, le résultat d'une itération pour une configuration donnée n'est calculé qu'une fois. Pour les configurations peu courantes, ce mécanisme n'est pas rentable en termes de temps de calcul mais dès lors que la configuration réapparait souvent dans une simulation, les gains en temps peuvent être très importants.

Le problème est qu'en l'absence de la connaissance des cellules voisines d'une configuration, il est impossible de connaitre le nouvel état des cellules sur le bord de la configuration. La taille de la configuration de cellules sauvegardées comme résultat correspond donc au cœur de la configuration initiale (voir Figure 2-13~~Figure 2-13~~).

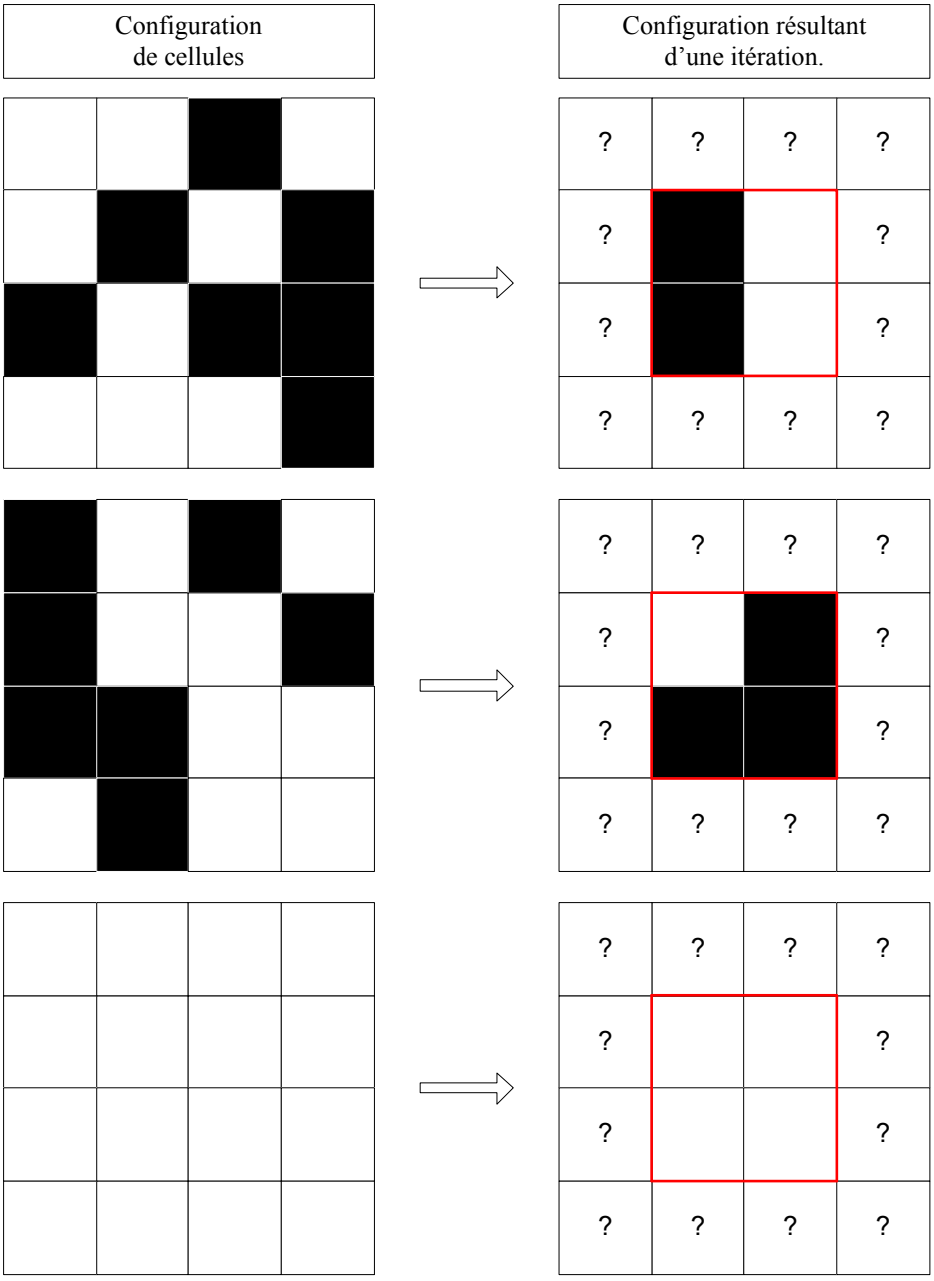


Figure 2-13 : Configurations de cellules avec leurs configurations résultantes sauvegardées après une itération.

Pour retrouver le nouvel état d'une configuration de cellules d'un automate (cadre rouge dans l'exemple de la Figure 2-14), il va donc être nécessaire de travailler sur une configuration sensiblement plus grande (cadre bleu dans l'exemple suivant). Étant donné que l'on utilise un quadtree, la configuration supérieure possédera toujours un côté deux fois plus grand (dans l'exemple suivant, la configuration supérieure est de taille 8x8 pour calculer une configuration de 4x4).

À partir de cette configuration de taille supérieure (la bleue), il n'est pas possible de retrouver le cœur en réalisant un simple calcul par récursion sur les quatre fils car celui-ci ne mène pas au résultat souhaité. En appliquant un algorithme séquentiel à l'automate de la Figure 2-15~~Figure 2-15~~, les quatre fils (en vert) fourniraient dans un premier temps comme résultat leurs cœurs respectifs (en orange). Or ces quatre zones en orange ne permettent pas de reconstituer le cœur de l'automate initial (en rouge), elles sont disjointes et trop éloignées du centre.

Pour disposer du résultat souhaité, il est nécessaire de créer quatre nouvelles configurations temporaires de cellules, de la même taille que la configuration que l'on cherche à étudier (en rouge), décalées respectivement en haut et à gauche, en haut et à droite, en bas et à gauche, et en bas et à droite (dans l'exemple suivant, il s'agit des quatre configurations de cellules rouges dans la deuxième étape du schéma). Une fois ces quatre configurations accessibles (créées ou récupérées dans la table de hachage), la fusion de leur calcul à l'itération suivante (numérotés de un à quatre dans l'exemple suivant) permet de former la configuration de cellules à l'itération suivante pour la configuration de départ.

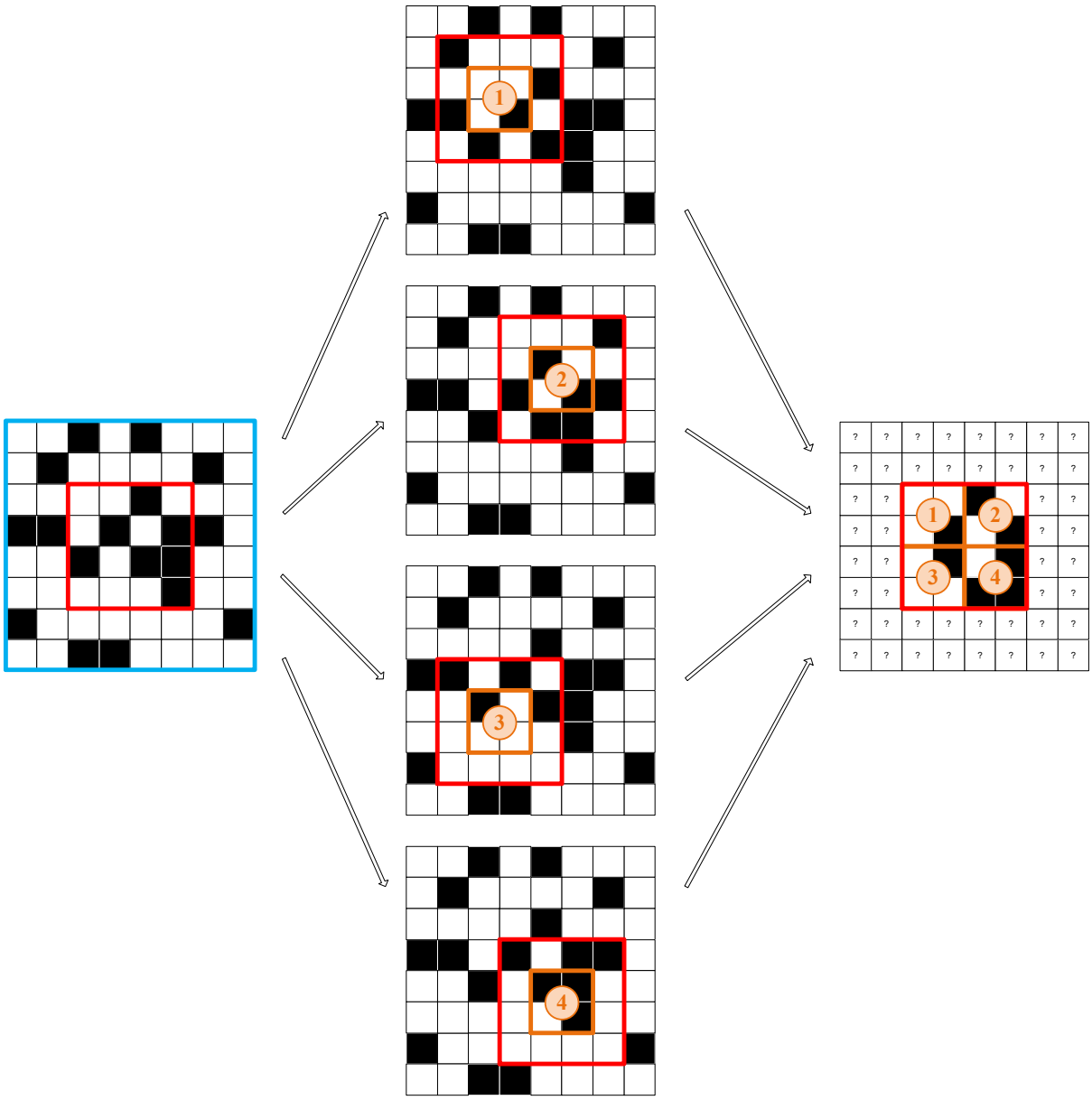


Figure 2-14 : Création de quatre configurations temporaires (au milieu) pour disposer, après fusion du résultat d'une itération sur chacune de ces configurations du nouvel état de la configuration (à droite).

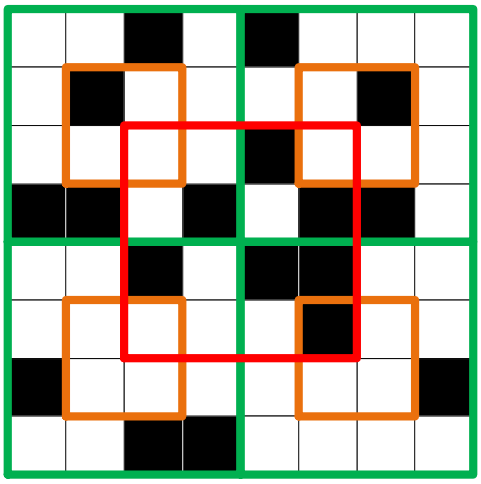


Figure 2-15 : Échec d'une récursion simple sur les quatre fils.

Le surcoût de ces différents mécanismes lors du calcul de l'itération suivante (lorsque ce résultat n'est pas déjà connu) est clairement apparent. Pour autant, le fait de pouvoir accéder au résultat directement (une fois qu'il a déjà été calculé) permet de contrebalancer tous ces coûts dans le cas où l'on retrouve suffisamment souvent les mêmes configurations de cellules (ce point sera développé plus en détail par la suite).

Mais l'algorithme Hash-Life ne se contente pas de sauvegarder le résultat de l'itération suivante, il peut sauvegarder le résultat de plusieurs itérations. Bien entendu, pour chaque itération, le cœur du cube va être un peu plus petit. Dans l'exemple de la Figure 2-14, étant donné que l'on ne s'intéresse qu'au centre du cube (de 4 par 4) et que nous avons travaillé avec un contour de deux cellules en utilisant le parent (de 8 par 8), il est alors possible de sauvegarder le résultat après deux itérations.

De manière générale, étant donné que l'on manipule un quadtree, le cœur sera toujours d'une taille deux fois moindre que l'espace initial (dans l'exemple de la Figure 2-14, un espace de 8x8 devient un espace de 4x4). Il est alors possible d'effectuer N / 4 itérations à l'avance sans perdre d'informations où N est le côté de l'espace initial. Cette technique est très puissante (pour un cube de 1024 par 1024, on pourra réaliser 256 itérations d'un coup), elle permet de réaliser très vite le calcul d'un grand nombre d'itérations. Pour autant, cette technique n'est viable que lorsque les résultats intermédiaires ne sont pas nécessaires.

Une explication complète (en anglais) de l'implémentation de l'algorithme est accessible sur le site DrDobbs[8] ou à travers l'implémentation libre du logiciel Golly[9].

2.2.3 Limites de l'algorithme Hash-Life

Selon les cas d'utilisation, l'algorithme peut-être d'une grande efficacité, ou peu rentable, pénalisant en terme de vitesse de calcul voire même inutilisable.

Son efficacité va dépendre de la réapparition de configurations de cellules. Si celles-ci sont souvent retrouvées, le surcoût du traitement initial pour le calcul de l'itération suivante (ou des itérations suivantes) est rapidement compensé. Plus il y a de nouvelles configurations qui apparaissent régulièrement et plus l'efficacité de l'algorithme va chuter.

La réapparition des combinaisons va donc avoir un effet primordial sur l'efficacité d'un tel algorithme. La proportion de réapparitions, si elle ne peut pas être établie à l'avance sans connaitre l'état initial de l'automate peut être estimée. Ainsi, plusieurs facteurs connus à priori vont avoir un impact sur la réapparition des configurations de cellules :

- Le nombre d'états possibles pour une cellule.

  Il est bien évident que, plus le nombre d'états acceptables pour une cellule lors d'une simulation est faible, plus le nombre de combinaisons est limité, et donc plus les chances de retrouver une configuration déjà rencontrée sont grandes.

  Un espace de 16 cellules possédera ainsi N^{16} configurations différentes (où N est le nombre d'états différents que peut prendre une cellule). On a donc 65 536 configurations différentes dans le cas d'un automate à deux états (comme le Jeu de la Vie). Si le nombre d'états augmente, le nombre de configurations différentes augmente très rapidement : pour N = 4, on a déjà plus de quatre milliards de configurations différentes (4 294 967 296).
- Les règles de l'automate cellulaire.

  Si le nombre d'états possibles pour une cellule influe fortement la réapparition des configurations, les règles sont toutes aussi importantes. En effet, selon les règles utilisées, le nombre de combinaisons pouvant apparaître au cours de la simulation

[8] http://drdobbs.com/high-performance-computing/184406478?pgno=1 : Dernier accès le 07/11/2011 (les liens vers les figures ne sont plus correct, accéder à celles-ci via les pages suivantes en bas de l'article).
[9] Golly - Page officielle: golly.sourceforge.net : Dernier accès le 07/11/2011.

peuvent être limitées et donc les réapparitions de configurations de cellules déjà connues seront plus courantes.

Pour reprendre une nouvelle fois l'exemple du jeu de la vie, hormis à l'initialisation, une configuration 4x4 ne pourra jamais comprendre une totalité de cellules vivantes. Pour l'algorithme WireWorld, la queue de l'électron ne sera jamais suivie de la tête de l'électron, ces contraintes diminuant d'autant le nombre de configurations possibles.

Par ailleurs, un inconvénient majeur de l'algorithme actuel est qu'il n'est pas adapté aux modèles d'automates cellulaires stochastiques. En effet, Hash-Life repose sur l'utilisation de la mémoïzation et sauve le résultat du calcul d'une ou plusieurs itérations. Or, avec un modèle stochastique, ce résultat peut être différent à chaque fois qu'il devrait être calculé. La mémoïzation et donc l'algorithme Hash-Life sont donc inutilisables, en l'état, avec des modèles stochastiques.

Conclusion

L'étude de la modélisation des systèmes complexes pose de nombreux problèmes aux informaticiens. Les contraintes de la modélisation, qu'elles soient multi-échelles ou non, se retrouvent en bout de chaîne lors de l'implémentation informatique. Ainsi, les différentes formes de modélisation des problèmes nécessitent des solutions différentes. Pour pouvoir résoudre plus simplement ces problématiques variées, il est ainsi nécessaire de disposer d'un outil de modélisation et de simulation le plus générique possible. La PGMS a été développée dans ce but. Si la PGMS est pour le moment cantonnée à des problèmes simples, l'objectif d'un tel outil est de permettre la simulation d'un grand nombre de systèmes, en particulier les systèmes complexes.

Même lorsqu'une application bénéficie d'un développement optimal, les contraintes matérielles limitent ses performances. Il est alors parfois obligatoire, pour simuler des systèmes plus importants, d'avoir recours à la parallélisation de l'application pour disposer de performances raisonnables. Si l'utilisation de clusters ou de grilles de calcul permet de disposer d'immenses puissances de calcul, l'utilisation des GPU pour réaliser des calculs scientifiques est une alternative de plus en plus séduisante. La puissance de ces derniers ne cesse en effet de croire alors même que celle des microprocesseurs conventionnels a

tendance à stagner. D'autre part, l'apparition d'API permettant d'effectuer des calculs généralistes simplement sur de telles cartes (CUDA et OpenCL en particulier) à l'aide de l'approche SIMT (Single Instruction Multiple Thread) a grandement accéléré le développement d'applications généralistes sur GPU. Dans notre contexte nous avons retenu cette solution qui se révèle tout à fait adaptée à la problématique de performance obtenue sur une machine de table.

Pour autant, la parallélisation de l'application n'est pas toujours la seule solution existante pour améliorer les performances d'une application. Pour bien des problèmes, un changement d'algorithme peut fortement faire varier les performances. C'est par exemple le cas des automates cellulaires où l'algorithme Hash-Life permet dans de très nombreux cas une amélioration drastique des performances.

Dans le contexte de la PGMS, la simulation d'automates en trois dimensions est essentielle, nous avons cependant constaté que les meilleures implémentations n'ont malheureusement été étudiées que dans le cas de modèles en deux dimensions. En effet, les recherches sur des modèles en trois dimensions sont rares.

Mon travail a donc porté sur plusieurs axes, l'enjeu restant toujours l'amélioration des performances de l'application. Des travaux ont ainsi été effectués sur les automates cellulaires et le calcul de champs afin d'améliorer les performances de modèles utilisant ces outils. Dans ce but et pour permettre à l'application d'être la plus efficace possible, différentes options ont été étudiées : optimisation du code existant, la modification de l'algorithme utilisé ou la parallélisation sur GPU.

Étant donné l'usage privilégié dans mes travaux de la parallélisation à base de GPU pour améliorer les performances de la PGMS, nous avons souhaité présenter l'API CUDA de façon pédagogique dans le chapitre qui suit. L'explication du choix de cette API plutôt qu'une autre et la présentation complète de celle-ci sont le sujet du prochain chapitre.

Développement sur architecture hybride à l'aide de l'API CUDA

3 Développement sur architecture hybride à l'aide de l'API CUDA

Introduction

Si le potentiel des cartes graphiques en termes de performances brutes n'est plus à démontrer, il est cependant nécessaire de disposer des bons outils et d'une bonne formation pour pouvoir en tirer le meilleur. Dans le but de faciliter la tâche des développeurs, un nombre important d'API permettant la réalisation de calculs sur GPU est aujourd'hui disponible. Au début de ma thèse, en 2009, deux API majeures étaient disponibles pour la réalisation de calculs sur architecture hybride. Il s'agissait de l'API CUDA proposée par NVIDIA et de l'API OpenCL proposée par Khronos. Dans notre contexte de thèse CIFRE, le choix s'est porté sur la parallélisation sur GPU à l'aide de l'API proposée par NVIDIA.

Les raisons de ce choix sont multiples. Tout d'abord, une des données les plus importantes a bien entendu été la performance des deux API. Or, pour une même application, les performances des applications réalisées à l'aide de CUDA étaient nettement plus performantes que celles réalisées à l'aide d'OpenCL (qui étaient parfois jusqu'à 60% plus lente) (Karimi, Dickson et Hamze 2010). Cette différence était majoritairement due à une volonté de NVIDIA de mettre en avant son API et donc d'optimiser l'utilisation de ses cartes graphiques pour son API. Cette mentalité a fortement évolué depuis le début de ma thèse et les nouveaux « drivers » OpenCL pour les cartes graphiques NVIDIA sont beaucoup plus performants.

La production par NVIDIA de cartes graphiques très performantes dédiées au calcul généraliste a également largement contribué au choix final. En effet, aussi performante qu'elle soit, l'API CUDA n'est pour le moment utilisable qu'avec des GPU NVIDIA. Il est donc nécessaire de disposer du matériel le plus adapté pour obtenir les meilleures performances. En 2008, NVIDIA proposait déjà du matériel très performant spécialisé dans le calcul généraliste : la carte Tesla M1060 pour les serveurs et la carte C1060 pour les stations de travail fournissaient déjà 240 unités de calcul en virgules flottantes cadencées jusqu'à 1,44GHz.

De plus, un plus grand nombre d'outils étaient disponibles pour aider au développement d'applications à l'aide de l'API CUDA alors que pour OpenCL l'offre était moins étoffée à

l'époque. En particulier, la présence d'un débogueur inspiré de gdb et d'un profileur a été très précieuse lors de l'optimisation de la parallélisation de notre application. Enfin, la popularité plus importante de l'API CUDA a permis de disposer de davantage de ressources ; ce qui est loin d'être négligeable lorsque l'on travaille sur une API encore très jeune dans un contexte limité par le temps.

Ce sont ces raisons qui ont guidé le choix de l'API à utiliser. Pour autant, il est important de noter que le but a toujours été de rester le plus générique possible en limitant l'utilisation de toutes les fonctions propres à l'API CUDA. Par exemple, les fonctions proposées par CUDA afin de simplifier le traitement des erreurs n'ont pas été utilisées afin de conserver un découpage maximum par rapport à l'API CUDA. Ainsi, si le choix venait un jour à être fait de porter la parallélisation depuis CUDA vers OpenCL, un minimum d'efforts serait nécessaire.

La suite de ce chapitre va donc détailler les principes de la programmation sur GPU à l'aide de l'API CUDA. Une première partie traitera de l'utilisation des threads, des blocs de threads et des grilles de blocs au sein de l'API CUDA avant d'introduire assez simplement l'ordonnancement de tous ces éléments par le GPU. La seconde partie décrira les différentes mémoires disponibles sur un GPU et la façon de les utiliser à l'aide de CUDA. Enfin, dans une dernière partie, des éléments complémentaires de l'API CUDA seront présentés : la gestion des erreurs, le calcul en nombres flottants et les flux asynchrones.

3.1 Gestion des threads

Pour ne pas avoir à se soucier des détails d'implémentation du paradigme SIMT (présentée dans la partie 2.1.4.2 - L'approche Single Instruction Multiple Thread (SIMT)~~L'approche Single Instruction Multiple Thread (SIMT)~~) et d'architecture (présentée dans la partie 2.1.4.3 - L'architecture des processeurs graphiques à but générique~~L'architecture des processeurs graphiques à but générique~~), une API la plus simple possible a donc été proposée par NVIDIA. Pour permettre une utilisation simple, CUDA s'appuie sur le préprocesseur nvcc qui permet de générer, depuis un fichier écrit en CUDA, un fichier C++ qu'il est possible de compiler avec g++ (NVIDIA 2008). Cette stratégie a l'avantage de permettre à CUDA d'utiliser des mots clés et des syntaxes particulières pour la conception et l'appel de threads CUDA.

3.1.1 Principe des Grilles, Blocs et Threads

Pour exploiter les nombreuses unités de calcul présentes sur le GPU, il n'est pas nécessaire de créer manuellement autant de *threads* que l'on souhaite en exécuter et d'affecter chacun d'eux à une unité de calcul. Il suffit, lors de l'appel d'une fonction à exécuter sur le GPU (appelée *kernel* dans la terminologie CUDA), de préciser quel est le nombre de *threads* qui doit se charger de son exécution. Ce nombre est fourni en renseignant la dimension des *blocs* (le nombre de *threads* compris dans un *bloc*) et la dimension de la *grille* (le nombre de *blocs* compris dans la *grille*).

3.1.1.1 Threads

Les deux notions de *bloc* et de *grille* seront détaillées plus tard, il suffit pour le moment de comprendre qu'un *bloc* contient un certain nombre de *threads* et que la *grille* contient un certain nombre de *blocs*.

Les appels sont donc de la forme suivante :

```
ma_fonction<<< dimension_de_la_grille, dimension_des_blocs >>>(
mes_parametres );
```

Code 3-1 : Pseudocode de l'appel d'un kernel.

Ainsi, pour créer 10 *threads* exécutant en parallèle la fonction `kernelExemple1`, il suffit de réaliser l'appel de la fonction en passant en pseudo-paramètre la dimension de la grille (ici 1 car il n'est pas nécessaire de disposer de plus d'un bloc pour utiliser 10 *threads*) et la taille du bloc (10 pour le nombre de *threads* souhaité) :

```
kernelExemple1<<< 1, 10 >>>(); // Appel du kernel sur 10 threads
```

Code 3-2 : Appel d'un kernel utilisant dix threads.

Pour que la fonction `kernelExemple1` soit exécutée sur le GPU, il est nécessaire que celle-ci ait été définie en tant que *kernel*. Un *kernel* est une fonction exécutable sur le GPU. Pour différencier les différents types de fonction, « C for CUDA » a rajouté trois autres mots clés : `__global__`, `__device__` et `__host__`. Ces trois mots clés permettent de définir la portée d'une fonction :

- `__global__` : Ce qualificatif permet de spécifier que la fonction est un *kernel*. C'est-à-dire qu'il s'agit d'une fonction qui ne pourra être appelée que depuis l'hôte (le CPU) et sera exécutée sur le dispositif (le GPU).

Dans l'exemple précédent, la fonction `kernelExemple1`, qui est donc un *kernel*, sera définie à l'aide du mot clé `__global__` :

```
__global__ void kernelExemple1()
{
   [...] // Code de la fonction
}
```

Code 3-3 : Exemple de déclaration d'un kernel.

De façon générale, toutes les fonctions qui doivent être exécutées sur le GPU et qui sont appelées depuis l'hôte (et donc comprennent dans leurs appels la dimension de la grille et la dimension des blocs) doivent être définies à l'aide du mot clé `__global__`.

- `__device__` : Ce qualificatif est utilisé pour définir des fonctions qui seront appelées depuis le GPU ; c'est-à-dire des fonctions appelées depuis d'autres fonctions qualifiées par `__device__` ou encore des fonctions qualifiées par `__global__`.
  Il serait par exemple possible dans le code de kernelExemple1() d'appeler une fonction qualifiée par le mot clé `__device__` :

```
__device__ void fonctionDevice1()
{
   [...] // Code de la fonction
}
__global__ void kernelExemple1()
{
   [...]
   fonctionDevice1(); // Appel de la fonction
   [...]
}
```

Code 3-4 : Exemple de déclaration et d'utilisation d'une fonction "device".

- `__host__` : Ce qualificatif est utilisé pour définir les fonctions appelées depuis l'hôte et exécutées sur celui-ci. Cela revient tout simplement à ne mettre ni `__global__`, ni `__device__` et à considérer la fonction comme une fonction standard du C.

Pour permettre l'appel d'un grand nombre de *threads* exécutant le même *kernel*, les *threads* doivent être regroupés en *blocs*, eux-mêmes regroupés dans une *grille*. La répartition entre la *grille* et les *blocs* est laissée libre aux développeurs. Dans l'exemple précédent exécutant 10 *threads*, le choix correspond à un seul bloc contenant 10 *threads* ; il aurait aussi été possible de choisir de disposer d'une grille contenant 2 blocs et dont chacun des blocs n'aurait plus contenu que 5 *threads* chacun (ou à l'inverse 5 *blocs* de 2 *threads*). Ces

questions ne sont pas complètement anodines et peuvent avoir un impact conséquent sur les performances.

Warps

Avant d'en venir à la notion très importante des *blocs*, il est nécessaire de réaliser un court aparté sur la notion de *warps* présente chez NVIDIA. Un *warp* est une notion logicielle qui correspond à un groupement de 32 *threads* qui exécutent la même instruction en même temps sur un *streaming multiprocessor* (pour être exact, étant donné qu'un *streaming multiprocessor* ne comprend que 8 unités de calcul flottant, il faudra 4 cycles consécutifs). Le fait qu'il s'agisse d'une notion logicielle signifie ici qu'il n'y a pas d'équivalent matériel au *warp*. Malgré cela, il n'est pas moins vrai que cette notion a un fort impact sur l'utilisation du GPU et en particulier sur les accès mémoire et l'ordonnancement des *threads*. En effet, les accès mémoire d'un *warp* peuvent être mutualisés. Ainsi, si tous les threads d'un *warp* accèdent à des emplacements mémoire proches, uniquement deux accès seront nécessaires (cf. Mémoire globale (global memory)). De plus, les dernières versions (avant Fermi) de GP-GPU de NVIDIA ne peuvent pas ordonnancer plus de 32 w*arps* en même temps sur un même *streaming multiprocessor*. Un *warp* contenant lui-même 32 *threads*, il est donc impossible de dépasser 1024 *threads* ordonnancés en même temps. Ainsi, le fonctionnement optimal d'un GPU ne sera généralement atteint que lorsque les *blocs* seront composés d'un nombre de *threads* multiple de 32. La notion de *warp* a donc un impact important sur la programmation des GPU.

3.1.1.2 Blocs

Les *blocs* permettent de regrouper les threads dans une première unité logique. Le nombre de *threads* maximum qu'il est possible de contenir dans un *bloc* dépend de la carte GPU utilisée (512 threads maximum par bloc sur les GPU actuels). En effet, l'exécution de l'ensemble des threads d'un même bloc sera effectuée sur un même *streaming multiprocessor* (à l'inverse, des *threads* présents sur différents *blocs* peuvent être exécutés sur différents *streaming multiprocessors*).

Or le fait d'être exécuté sur un même *streaming multiprocessor* n'est pas anodin et va permettre aux *threads* d'un même *bloc* de pouvoir tirer parti de la mémoire partagée propre à chaque *streaming multiprocessor* (pour communiquer mais aussi pour mutualiser les accès

mémoire). Il va également être possible de synchroniser les *threads* à un endroit précis d'un *kernel*. La mémoire partagée et la synchronisation seront détaillées plus loin dans ce document.

Par ailleurs, afin de faciliter les calculs sur des structures multidimensionnelles, il est possible de créer des *blocs* possédant deux ou trois dimensions. Pour cela, il est nécessaire d'utiliser la structure `dim3` fournie par l'API. Les dix *threads* de l'exemple précédent auraient alors pu être représentés dans des blocs de dimension 2 par 5 par 1 :

```
dim3  dimension_bloc( 2, 5, 1 );
```

Code 3-5 : Exemple de déclaration de dimension d'un bloc.

Il existe malgré tout une restriction sur la taille des blocs dans chaque dimension : la taille des dimensions x et y (la première et la deuxième) est limitée à 512 alors que la dimension z (la troisième) ne peut dépasser 64. Il est par ailleurs impossible pour le moment d'utiliser des blocs de plus de 512 *threads* ; il est donc impossible en particulier d'utiliser un bloc de 512x512x64 car cela ferait bien plus de 512 *threads* (16 777 216 pour être précis). De plus, lorsque la taille d'une dimension n'est pas spécifiée (comme les dimensions y et z dans la première ligne de l'exemple suivant), sa valeur est mise par défaut à 1. Ainsi, les trois lignes de code suivantes sont équivalentes et vont toutes les trois créer des *blocs* de taille 10 à une dimension :

```
dim3   dimension_bloc1( 10 );
dim3   dimension_bloc2( 10, 1 );
dim3   dimension_bloc3( 10, 1, 1 );
```

Code 3-6 : Exemple de déclarations équivalentes de bloc contenant dix threads.

Une fois la dimension du bloc choisie, l'appel d'un *kernel* en passant cette dimension en paramètre permet d'appeler ce *kernel* de façon à ce que son code soit exécuté plusieurs fois en parallèle sur le GPU. Ainsi, le code ci-dessous lancera automatiquement l'exécution de 12 *threads* en parallèle (voir Figure 3-1).

```
dim3   dimension_bloc( 4, 3 );

// Appel du kernel sur 12 threads (4x3)
kernelExemple2<<< 1, dimension_bloc >>>();
```

Code 3-7 : Appel d'un kernel prenant une instance de dim3 en paramètre.

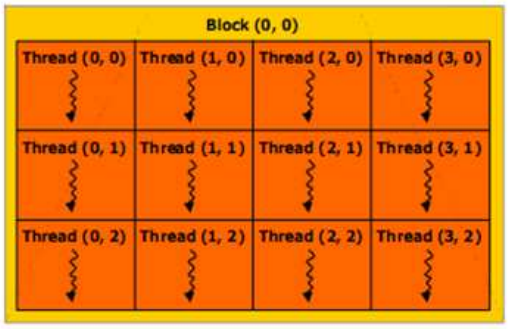


Figure 3-1 : Exécution de 12 threads (4x3) dans un bloc. [source NVIDIA]

Mais il est parfaitement inutile de disposer de plusieurs *threads* exécutant le même *kernel* si tous les *threads* ne peuvent être différenciés de manière unique et ainsi travailler sur des données différentes. Pour permettre la différenciation des différents *threads*, « C pour CUDA » fournit un ensemble de variables accessibles depuis un *kernel* et permettant de connaitre ses coordonnées aussi bien au sein de son *bloc* que les coordonnées du *bloc* dont il fait partie au sein de la *grille*.

Tout d'abord la variable `threadIdx` permet d'identifier un *thread* au sein d'un *bloc*. Elle est de type `uint3` avec 3 coordonnées entières pour se positionner dans un bloc. L'accès à l'indice du thread courant dans chacune des trois dimensions x, y et z se fera donc de la façon suivante :

```
int   indiceX     = threadIdx.x;      // Indice en dimension x
int   indiceY     = threadIdx.y;      // Indice en dimension y
int   indiceZ     = threadIdx.z;      // Indice en dimension z
```

Code 3-8 : Récupération des identifiants x, y et z d'un thread dans un bloc.

En considérant le bloc précédent de dimension 4x3, on obtiendrait alors 12 *threads* ayant tous la valeur 0 pour z mais un couple unique de valeur (x, y) variant respectivement de 0 à 3 et de 0 à 2 pour chacun des indices. Le fait de se repérer dans un bloc avec trois coordonnées est historique et vient du fait que les GPU traitaient des cubes graphiques. Pour simplifier les traitements, ces trois coordonnées ont été introduites. Il n'est nullement obligatoire de les utiliser toutes les trois ; la coordonnée x peut suffire à n'importe quel traitement.

Ainsi, un *kernel* où chaque *thread* devrait doubler la valeur de chaque élément d'un tableau à une dimension passé en paramètre prendrait la forme suivante :

```
__global__ void kernelExemple3( float * inTab, float * outTab )
{
   int   indiceX  = threadIdx.x;
   outTab[ indiceX ] = inTab[ indiceX ] * 2;
}
```

Code 3-9 : Kernel permettant de doubler chaque valeur d'un tableau.

Dans le corps du *kernel*, on récupère l'identifiant unique `threadIdx.x`. On ne réalise alors les calculs que sur l'élément du tableau possédant l'indice correspondant : le premier *thread* (d'indice 0) va s'occuper de multiplier le premier élément du tableau (d'indice 0 également), le deuxième *thread* (d'indice 1) va multiplier le deuxième élément du tableau (d'indice 1 également) et ainsi de suite. Sans oublier qu'il est nécessaire de stocker le résultat dans le tableau de retour toujours à l'indice correspond au numéro du *thread*.

Aucun test n'est réalisé dans ce code pour vérifier que l'on n'essaie pas d'accéder à un indice de tableau supérieur à celui du dernier élément du tableau. Il est donc nécessaire de réaliser l'appel de ce *kernel* avec autant de *threads* qu'il y a d'éléments dans le tableau :

```
void appelExemple3()
{
float *    tabCPU;
float *    tabGPU_Input;
float *    tabGPU_Output;
const int  tailleTab = 25;

[...] // On passe sous silence pour le moment l'allocation de la
      // mémoire sur l'hôte et le GPU ainsi que sa copie sur le GPU.

dim3       dim_bloc( tailleTab );
dim3       dim_grille( 1 );

// Appel du kernel précédent avec 25 threads
kernelExemple3<<< dim_grille, dim_bloc >>>( tabGPU_Input,
tabGPU_Output );

[...] // On ne s'intéresse pas non plus □ a copie retour.
}
```

Code 3-10 : Appel d'un kernel avec le bon nombre de threads.

En allouant un tableau de taille `tailleTab` et en utilisant cette même variable pour définir le nombre de *threads* présents dans chaque *bloc* dans la dimension x, on s'assure d'avoir exactement un *thread* pour chaque cellule du tableau.

L'utilisation de deux ou trois dimensions ne diffère que très légèrement :

```
__global__ void kernelExemple4( float *** inTab, float *** outTab )
{
   int   iX   = threadIdx.x;
   int   iY   = threadIdx.y;
   int   iZ   = threadIdx.z;

   outTab[ iX ][ iY ][ iZ ]  = inTab[ iX ][ iY ][ iZ ] * 2;
}

void appelExemple4()
{
   float ***   tab;
   float ***   tabResult;
   int         tailleTabX   = 25;
   int         tailleTabY   = 12;
   int         tailleTabZ   = 3;

   [...] // On passe encore sous silence l'allocation et la copie
         // de la mémoire.

   dim3     dim_bloc( tailleTabX, tailleTabY, tailleTabZ );
   dim3     dim_grille( 1 );

   kernelExemple3<<< dim_grille, dim_bloc >>>( tab, tabResult );

   [...] // On ne s'intéresse pas non plus a la copie retour.
}
```

Code 3-11 : Utilisation d'un bloc à plusieurs dimensions.

Mais l'utilisation seule des *blocs* ne permet pas de tirer le maximum de la puissance des GP-GPU. En effet, il est possible pour un GPU d'exécuter plusieurs *blocs* en même temps et donc d'augmenter le nombre de calculs réalisés en parallèle. Il est également souvent beaucoup plus simple de programmer dans un premier temps les calculs en affectant à chaque *thread* une seule cellule d'un tableau (même si cette version n'est pas toujours la plus rapide). Il est alors nécessaire de disposer d'un grand nombre de *threads*, nombre qui peut facilement arriver à dépasser les 512 *threads* maximum d'un *bloc*. Dans ce cas, il est nécessaire de travailler avec plusieurs *blocs* dans la *grille*.

3.1.1.3 Grilles

Le principe de fonctionnement de la grille est très proche de celui des blocs. Mais là où un *bloc* comprend un certain nombre de *threads*, la grille comprend un certain nombre de *blocs* (voir Figure 3-2).

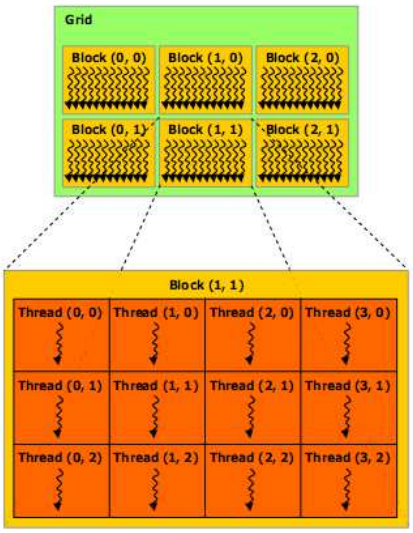


Figure 3-2 : Grille de 3x2 blocs comprenant chacun 4x3 threads. [source NVIDIA]

Une *grille* peut ainsi comporter jusqu'à 65535x65535 *blocs* soit 2^{32}-1 blocs (4 294 836 225). À la différence des *blocs*, il n'est pas possible de disposer de grilles à trois dimensions, la valeur de la dimension z doit impérativement rester à 1.

```
dim3   dimension_grille1( 1, 1, 1 );        // OK
dim3   dimension_grille2( 25, 25 );         // OK
dim3   dimension_grille3( 65535, 65535 );   // OK
dim3   dimension_grille4( 80000, 1 );       // pas OK : x > 65535
dim3   dimension_grille5( 1, 80000 );       // pas OK : y > 65535
dim3   dimension_grille6( 1, 1, 2 );        // pas OK : z > 1
```

Code 3-12 : Exemple de dimensions, de grilles, valides et invalides.

Multiplié par les 512 *threads* qu'un bloc peut contenir, on obtient un nombre de *threads* potentiel de l'ordre de mille milliards[10].

Tout comme pour les *threads* d'un *bloc*, il est possible de connaître dans un *kernel* les indices x et y (z vaut toujours 1) d'un *bloc* au sein de la *grille*. Ceci est réalisé à l'aide de la variable `blockIdx` qui est en tout point similaire pour les *blocs* à ce qu'est `threadIdx` pour les *threads*. Par ailleurs, de manière à ne pas avoir à passer en paramètre de chaque *kernel* les dimensions des *blocs*, la variable `blockDim` fournit cette information. Ainsi, le calcul de l'indice global du *thread* en x doit prendre en compte : l'indice du thread dans son bloc, l'indice du bloc dans la grille ainsi que la dimension en x des blocs :

[10] À l'inverse des *threads* d'un *bloc* dont l'exécution est garantie d'être effectuée sur un même *streaming multiprocessor*, rien ne permet de garantir que deux *blocs* seront exécutés sur un même *streaming multiprocessor*, même si les *blocs* sont suffisamment petits pour le permettre.

```
__global__ void kernelExemple5( float * inTab, float * outTab )
{
   int  iX  = threadIdx.x + blockIdx.x * blockDim.x;

   outTab[ iX ]  = inTab[ iX ] * 2;
}
```

Code 3-13 : Utilisation des variables blockIdx et blockDim.

Le calcul de l'indice en y est parfaitement similaire en remplaçant x par y. À noter toutefois qu'il n'est pas possible de disposer de grilles à trois dimensions. Si l'on veut malgré tout utiliser l'indice z de manière homogène aux indices x et y, celui-ci ne nécessitera pas d'effectuer un calcul à l'aide de blockIdx.z * blockDim.z car le résultat sera toujours égal à 0. L'indice z est donc uniquement égal à la valeur de la variable *threadIdx.z*. Une autre solution, lorsque l'on souhaite travailler à l'aide de tableaux tridimensionnels de taille importante, consiste à n'utiliser que la dimension x et à retrouver les indices x, y et z souhaités à l'aide de trois calculs simples :

```
__global__ void kernelExemple5( float * inTab, float * outTab,
                    int inTailleX, int inTailleY, int inTailleZ )
{
   int  iXGlobal  = threadIdx.x + blockIdx.x * blockDim.x;
   int  iX   = iXGlobal % inTailleX;
   int  iY   = ( iXGlobal / inTailleX ) % inTailleY;
   int  iZ   = ( iXGlobal / ( inTailleX * inTailleY ) ) % inTailleZ;

   [...]
}
```

Code 3-14 : Récupération des indices x, y et z à partir d'un indice x global.

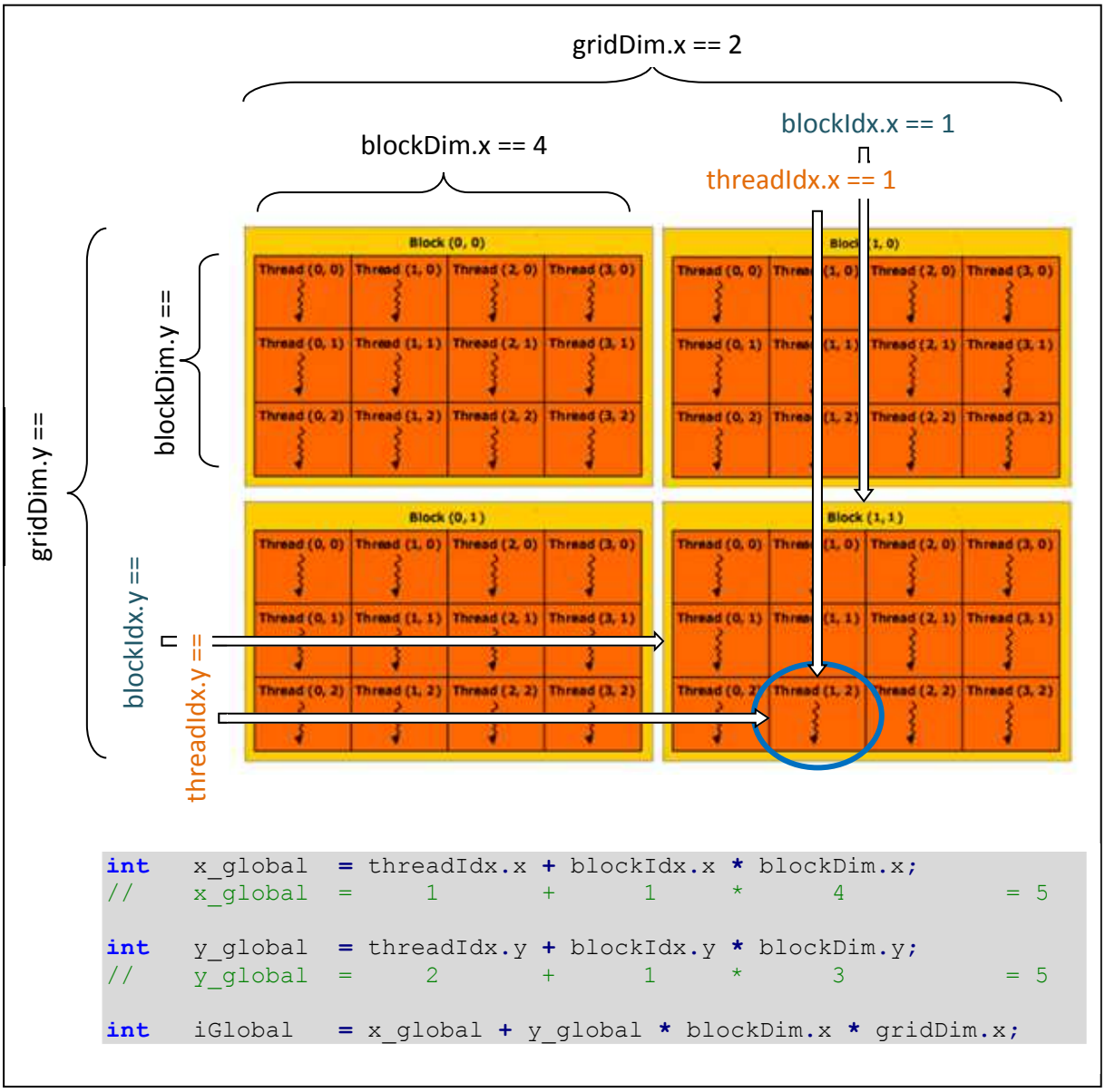
Une autre variable intéressante est la variable `gridDim` qui fournit les dimensions de la grille en nombre de blocs. Cette fonction est utile pour calculer l'indice global d'un thread toutes dimensions confondues ; c'est-à-dire son indice en x plus son indice en y multiplié par le nombre de threads présents dans la dimension x :

```
   int   iX              = threadIdx.x + blockIdx.x * blockDim.x;
   int   iY              = threadIdx.y + blockIdx.y * blockDim.y;
   int   nbThreadsDimX   = gridDim.x * blockDim.x;
   int   global          = iX + iY * nbThreadsDimX;
```

Code 3-15 : Utilisation de la variable grimDim.

Cet indice global est très utile lorsque l'on travaille sur d'immenses tableaux à une seule dimension (qui représentent souvent des tableaux à plusieurs dimensions où les trois indices peuvent être recalculés comme dans l'exemple précédent). La limite de la dimension x de 512x65535 (≈$3.4x10^7$) peut alors être pénalisante. Utiliser les dimensions x et y (la dimension

z ne permet pas d'augmenter la capacité en nombre de *threads*) permet alors de travailler sur des tableaux de taille bien supérieure à ce qu'il est possible de stocker en mémoire sur le GPU.




```
int   x_global  = threadIdx.x + blockIdx.x * blockDim.x;
//    x_global  =     1        +      1     *      4            = 5

int   y_global  = threadIdx.y + blockIdx.y * blockDim.y;
//    y_global  =     2        +      1     *      3            = 5

int   iGlobal   = x_global + y_global * blockDim.x * gridDim.x;
```

Figure 3-3 : Variables intégrées pour les dimensions et les indices des blocs et de la grille.

Premier exemple concret

Pour illustrer concrètement l'ensemble des notions présentées dans ce paragraphe, nous allons étudier l'exemple très simple du seuil d'une image en niveaux de gris. La valeur de chaque pixel de l'image est comprise entre 0 (noir) et 255 (blanc). Le but est ici de fournir une fonction permettant d'effectuer un seuil sur les pixels de l'image. Cela correspond à

parcourir l'ensemble des pixels de l'image et à remplacer tous les pixels dont la valeur est inférieure au seuil par la valeur seuil.

Par souci de compréhension, trois simplifications vont être utilisées :

- La valeur de chaque pixel sera stockée dans un `int` bien qu'elle ne nécessite que 8 bits et qu'il serait donc possible de stocker plusieurs valeurs de pixel dans un même `int`.
- Un *thread* sera affecté à un et un seul pixel. Cette méthode permet d'implémenter plus facilement le parallélisme mais perd en efficacité lorsque la taille de l'image augmente.
- Aucune image ne sera chargée. À la place, on saisira les dimensions de l'image et des valeurs seront générées pseudo-aléatoirement entre 0 et 255.

Regardons tout d'abord la fonction `main` :

```
int main( int argc, char ** argv )
{
   int *    image_host;
   int *    image_device;
   dim3     dimGrille;
   dim3     dimBloc;

   int      tailleX              = 6000;
   int      tailleY              = 8000;
   int      seuil                = 50;

   int      tailleTotale         = tailleX * tailleY;
   int      tailleTotaleOctets   = tailleTotale * sizeof( int );
```

Code 3-16 : Fonction main - déclaration des variables.

Dans la première partie de la fonction `main`, il est nécessaire de déclarer les tableaux qui serviront à stocker les pixels sur l'hôte (`image_host`) et sur le GPU (`image_device`) ainsi que les variables stockant les dimensions de la grille (`dimGrille`) et des blocs (`dimBloc`). De plus, le choix a été fait de saisir les dimensions de l'image (`tailleX` et `tailleY`) et le seuil (`seuil`) en dur dans l'application par souci de clarté. Enfin, les variables `tailleTotale` et `tailleTotaleOctets` permettent de calculer une seule fois le nombre de pixels de l'image ainsi que la taille en octets pour un tableau de `int` de cette taille.

```
   // Allocation de la mémoire, sur le CPU, puis sur le GPU.
   image_host   = new int[ tailleTotale ];
   cudaMalloc( (void **) &image_device, tailleTotaleOctets );

   // Initialisation pseudo-allocation (voir remarque).
   for ( int i = 0; i < tailleTotale; ++i )
   {
      image_host[ i ] = rand() % 256;
   }

   // Copie sur le GPU du tableau.
   cudaMemcpy(
               image_device,
               image_host,
               tailleTotaleOctets,
               cudaMemcpyHostToDevice
             );
```

Code 3-17 : Fonction main - allocation et copie de la mémoire.

Dans un second temps, il est obligatoire d'allouer la mémoire nécessaire sur le CPU ainsi que sur le GPU. Ceci est dû au fait qu'il n'est pas possible pour le CPU d'accéder directement à la mémoire du GPU et inversement. Il est donc nécessaire d'utiliser un tableau de taille identique sur chacun des processeurs et de les copier entièrement. La gestion de la mémoire est expliquée en détail dans la partie 3.2 - Mémoire des GPU, elle ne sera donc pas détaillée ici. Les valeurs du tableau sont générées pseudo-aléatoirement le plus simplement possible à l'aide de la fonction `rand` et d'un modulo. Ce n'est pas la meilleure solution en termes de qualité statistique du générateur mais elle suffit largement dans notre cas. Une fois le tableau d'entiers initialisé, celui-ci est copié sur le GPU.

```
   calculDimensionGrilleBloc( dimGrille, dimBloc, tailleTotale );
   seuil_gpu<<< dimGrille, dimBloc >>>(
                                        image_device,
                                        tailleTotale,
                                        seuil
                                      );

   cudaMemcpy(
               image_host,
               image_device,
               tailleTotaleOctets,
               cudaMemcpyDeviceToHost
             );
   verifImage( image_host, tailleTotale, seuil );
}
```

Code 3-18 : Fonction main - appel du kernel et récupération du résultat.

Dans la dernière partie du `main`, on calcule tout d'abord, en fonction de la taille totale du tableau, les dimensions de la grille et des blocs (la fonction est détaillée ci-après). Une fois ces dimensions calculées, il est possible de lancer le calcul du seuil sur le GPU (`seuil_gpu`).

Enfin, on réalise la copie du tableau mis à jour sur le GPU et on vérifie, succinctement dans cet exemple, que le seuillage a bien été réalisé à l'aide de la fonction `verifImage` de façon à valider nos calculs :

```
void verifImage( int * inImage, int inTailleTotale, int inSeuil )
{
   for ( int i = 0; i < inTailleTotale; ++i )
   {
      if ( inImage[ i ] < inSeuil )
      {
         std::cerr << "Erreur, le seuil n'a pas été réalisé
                   << "comme il se doit pour l'indice " << i << "\n";
         exit( 1 );
      }
   }
}
```

Code 3-19 : Fonction de vérification des résultats.

La vérification se fait par simple comparaison entre chaque valeur des pixels de l'image et la valeur du seuil. Si un pixel a une valeur inférieure au seuil, on en informe l'utilisateur et l'on quitte le programme. Pour disposer d'un test plus complet, il serait nécessaire de copier le tableau original et de vérifier que les pixels dont la valeur était supérieure au seuil n'ont pas été modifiés. (Cette vérification n'est nécessaire que lors de la validation des calculs.)

```
void calculDimensionGrilleBloc(
                                dim3 &  outDimGrille,
                                dim3 &  outDimBloc,
                                int     inTailleTotale
                               )
{
   static const int  NbThreadMaxParBloc     = 512;
   static const int  NbBlocMaxParDimGrille  = 65535;
   if ( inTailleTotale <= NbThreadMaxParBloc )
   {
      outDimBloc.x  = inTailleTotale;
   }
   else
   {
      outDimBloc.x  = NbThreadMaxParBloc;
      int  nbBloc  = inTailleTotale / NbThreadMaxParBloc +
             ( inTailleTotale % NbThreadMaxParBloc == 0 ? 0 : 1 );
      if ( nbBloc <= NbBlocMaxParDimGrille )
      {
         outDimGrille.x  = nbBloc;
      }
      else
      {
         outDimGrille.x  = NbBlocMaxParDimGrille;
         outDimGrille.y  = nbBloc / NbBlocMaxParDimGrille +
                  ( nbBloc % NbBlocMaxParDimGrille == 0 ? 0 : 1 );
      }
   }
}
```

Code 3-20 : Fonction de calcul de la dimension des blocs et de la grille.

Pour le calcul de la dimension de la *grille* et des *blocs*, le choix a été fait ici de disposer de *blocs* de 512 *threads* maximum. Ce choix est tout à fait discutable et des études de performances permettraient de l'affiner. Quoi qu'il en soit, l'algorithme est ensuite très simple : si l'image comprend moins de pixels que le nombre de *threads* maximum par *bloc*, on utilise un seul *bloc* avec le nombre exact de *threads* nécessaires. Sinon un *bloc* comprend 512 *threads* et l'on dimensionne la *grille* de façon à ce que le nombre total des *blocs* permette de contenir le nombre de *threads* souhaités ou plus (le nombre de threads doit être inférieur ou égal à 512 fois le nombre de *blocs*). Attention, cela veut dire qu'il peut arriver que l'on utilise plus de *threads* que nécessaires. Par exemple si l'image comprend 700 pixels, on va utiliser 2 *blocs* de 512 *threads*, soient 1024 *threads*. Il va donc falloir veiller dans le *kernel* à ce que les *threads* « en trop » n'accèdent pas à de la mémoire hors des limites du tableau.

Le *kernel* se présente donc comme ceci :

```
__global__ void seuil_gpu( int * inImage,
                           int inTailleTotale,
                           int inSeuil )
{
   int   iX  = threadIdx.x + blockIdx.x * blockDim.x;
   int   iY  = blockIdx.y; // car on utilise des blocs à une dimension
   int   nbThreadsDimX  = gridDim.x * blockDim.x;
   int   iGlobal  = iX + iY * nbThreadsDimX;

   if ( iGlobal < inTailleTotale && inImage[ iGlobal ] < inSeuil )
   {
      inImage[ iGlobal ] = inSeuil;
   }
}
```

Code 3-21 : Kernel de seuillage.

Le *kernel* est d'une très grande simplicité. Il se contente dans un premier temps, et comme à chaque fois, de calculer l'indice global du *thread*. Puis, si celui-ci est inférieur au nombre total de pixels (c'est-à-dire que l'on ne fait pas partie des *threads* « en trop »), on seuille si cela est nécessaire.

Le code du *kernel* est donc extrêmement simple dans cet exemple. Bien évidemment, ceci est dû à l'algorithme implémenté qui est lui aussi extrêmement simple. Mais c'est également le cas car nous utilisons d'une part une méthode basique pour paralléliser (un thread par pixel) et d'autre part un stockage entièrement réalisé en mémoire globale sans optimisation. Or, les accès en mémoire globale, comme c'est le cas ici, possèdent une forte latence et pénalisent donc l'application : il serait avantageux de diviser la taille du tableau par 4 en stockant 4 pixels dans chaque `int` de manière à diviser le nombre d'accès par 4 par exemple. De plus, un grand nombre de *threads* implique un grand nombre de *warps* (groupement de 32 *threads*) et donc un grand nombre de *blocs*, ce qui pose la question de l'ordonnancement sur le GPU : tous les threads ne pourront pas être exécutés en même temps si l'on travaille sur des images de grande taille.

3.1.2 Ordonnancement sur le GPU

Même s'il n'est pas obligatoire de comprendre parfaitement le fonctionnement interne du GPU pour tirer parti de ses performances, il est avantageux de le connaitre pour en tirer le maximum.

Ordonnancement des warps

Comme cela a été précisé dans la partie 2.1.4.3 - L'architecture des processeurs graphiques à but générique~~L'architecture des processeurs graphiques à but générique~~, un *streaming multiprocessor* est composé de 8 unités de calculs flottants (*thread processors* dans la terminologie CUDA). Or, le *streaming multiprocessor* travaille sur des *warps* comprenant 32 *threads*. Il faut donc 4 cycles d'horloges de l'ordonnanceur du *streaming multiprocessor* pour que les 8 unités de calcul aient le temps d'effectuer les 32 opérations nécessaires. Étant donné que l'ordonnanceur du GPU possède une cadence diminuée de moitié par rapport à celle des *streaming multiprocessors*, les 32 opérations sont effectuées en 2 cycles d'horloge du GPU.

Par ailleurs, les unités de calcul réalisent leurs opérations sur de la mémoire qu'il est bien évidemment nécessaire de charger au préalable. Ces accès mémoire nécessaires à l'exécution d'une instruction par l'ensemble d'un *warp* sont réalisés par demi-*warp* (16 accès mémoire à la fois). Ces 16 accès étant réalisés en 1 cycle d'horloge du GPU, ils sont réalisés aussi rapidement que sont exécutées 16 instructions sur le *streaming multiprocessor* (1/2 cycle du GPU permet l'exécution d'une instruction sur les 8 unités de calcul). Si les accès mémoire ne sont pas sujets à une latence supérieure à 1 cycle d'horloge, les traitements ne sont pas ralentis par ceux-ci. Bien évidemment, si l'accès à la mémoire est sujet à une latence supérieure, les instructions ne peuvent être exécutées tant que la mémoire n'a pas été chargée et les performances s'en trouvent diminuées.

Ordonnancement des blocs

Un autre niveau d'ordonnancement a lieu au niveau du GPU pour ordonnancer les blocs. Un GP-GPU possède plusieurs *streaming multiprocessors*, il est donc possible d'exécuter en même temps plusieurs *blocs* sur différents *streaming multiprocessors*. Un ordonnancement est nécessaire lorsque le nombre de *blocs* est trop grand pour permettre leurs exécutions simultanées. Dans ce cas, les blocs seront exécutés en plusieurs fois. Il va alors être possible pour le processeur graphique d'ordonnancer jusqu'à 8 *blocs* différents sur un même *streaming multiprocessor* tant que la limite du nombre de *threads* maximum pouvant être ordonnancé sur un streaming multiprocessor n'est pas atteinte. L'ordonnancement des blocs permet dans une certaine mesure de diminuer l'impact des accès mémoire en exécutant d'autres *threads* lorsque certains attendent un accès mémoire.

Ce mécanisme explique que, dans certains cas, les temps d'exécution n'évoluent que très peu avec l'augmentation du nombre de *blocs* utilisés : tant que l'exécution des *blocs* peut toujours être effectuée simultanément, le temps total d'exécution n'augmente pas proportionnellement au nombre de threads utilisés (même s'il augmente généralement car l'augmentation du nombre de *threads* va généralement de pair avec une augmentation de la taille des données traitées et donc des accès mémoire). À l'inverse, lorsque la charge maximum du GPU (en termes de nombre de *blocs* qu'il est possible d'exécuter simultanément) est atteinte, l'ajout d'un seul bloc peut nécessiter d'attendre la fin de l'exécution de tous les autres blocs et d'exécuter dans un second temps ce nouveau bloc ; entraînant ainsi une augmentation du temps de calcul bien supérieur à celle que pourrait laisser présager l'augmentation d'un *bloc*.

3.2 Mémoire des GPU

La question des accès à la mémoire est loin d'être insignifiante. Des accès nombreux à des mémoires comportant une latence importante peuvent grandement détériorer les performances de l'application. Il est donc nécessaire de bien maîtriser les différents types de mémoire pour pouvoir utiliser celle qui permettra de maximiser les performances de l'application.

3.2.1 Mémoire globale (global memory)

Description

La première de ces mémoires est la mémoire globale. Il s'agit de la seule mémoire accessible en lecture et en écriture qui soit de taille importante (2Go sur les cartes récentes) et commune à l'ensemble du GPU (et donc accessible depuis n'importe quel *thread* de la *grille* exécuté sur n'importe quel *streaming multiprocessor*). Enfin, cette mémoire est persistante entre les appels de *kernels* : il est ainsi possible d'appeler un premier *kernel* pour réaliser un traitement sur un tableau stocké en mémoire globale puis, à la fin de l'exécution de celui-ci, d'appeler un second *kernel* qui réalisera un traitement sur le résultat du premier traitement.

Cette mémoire est loin d'être parfaite. En particulier, ne disposant pas de cache généraliste avant l'architecture Fermi, l'accès aussi bien en lecture qu'en écriture subit une très importante latence d'environ 400 cycles. Il est donc nécessaire de limiter au maximum les accès à cette mémoire en exploitant les autres types de mémoire.

De plus, pour maximiser l'accès à la mémoire, il est nécessaire de respecter quelques règles un peu pointues. Comme cela a été vu plus haut, la mémoire nécessaire aux 16 threads d'un demi-*warp* est chargée simultanément. Or, CUDA est capable de réaliser en une transaction le chargement d'un segment de mémoire de 32, 64 et 128 octets. La lecture de segments de 16 mots mémoire de 4, 8 et 16 octets peut ainsi être réalisée en seulement une ou deux transactions (une transaction pour les mots de 4 et 8 octets et deux pour les mots de 16 octets). Mais ceci n'est possible que si l'accès à la mémoire est coalescent.

De manière simplifiée, un accès coalescent à la mémoire correspond à un accès à des emplacements mémoire contigus. Cette notion est importante car l'accès non coalescent à de la mémoire peut entrainer des baisses de performances significatives. Pour les diminuer, la notion de coalescence de la mémoire a évolué avec les versions des GPU de manière à être de plus en plus permissive et ainsi permettre un développement plus souple :

- Dans les versions 1.0 et 1.1 des GPU, la lecture de mots mémoire de 4, 8 et 16 octets ne peut être réalisée qu'à partir, respectivement, d'un segment de 64 octets, d'un segment de 128 octets et de deux segments de 128 octets. De ce fait, l'ensemble des 16 threads doit accéder à des adresses mémoires présentes dans une zone mémoire ne pouvant dépasser celle du segment (la mémoire doit être parfaitement contigüe).
  De plus, l'adresse mémoire accédée par le premier thread doit être un multiple de la taille des segments. Par exemple, la lecture de mots mémoire de 4 octets ne pourra commencer qu'aux adresses 0, 64, 128, 192, 256...
  Enfin, la dernière contrainte (et pas des moindres) est que le $i^{ème}$ thread ne peut accéder qu'au $i^{ème}$ mot de la zone mémoire en question.
  Si une seule de ces trois règles est violée, il est alors réalisé autant de transactions qu'il y a d'accès mémoire : si les 16 threads accèdent à de la mémoire, il y aura alors 16 transactions !
- Dans les versions 1.2 et 1.3, les critères ne sont plus si restrictifs. Un seul accès mémoire sera nécessaire si tous les mots mémoire lus par les 16 *threads* d'un demi-*warp* sont dans le même segment de mémoire :
  - ✓ de 32 octets pour les mots de 1 octet,
  - ✓ de 64 octets pour les mots de 2 octets et
  - ✓ de 128 octets pour les mots de 4 ou 8 octets.

  Il est donc possible pour plusieurs threads d'accéder à un même mot mémoire ou encore d'accéder aux mots mémoire de façon désordonnée sans pénaliser le temps d'accès. De plus, afin de limiter la consommation de bande passante, c'est le segment

de mémoire le plus petit qui sera utilisé : si un seul mot mémoire de 4 octets est lu, un segment de 32 octets sera utilisé et non pas un segment de 128 octets.

La contrainte de début d'adresse des segments mémoire qui oblige l'accès à des segments commençant par des adresses mémoires qui sont des multiples de sa taille est, elle, toujours présente : l'accès aux mots d'adresse mémoire 28 et 32 ne pourra jamais être réalisé à l'aide d'une seule transaction de 32 octets.

Enfin, lorsque la lecture d'un seul segment n'est pas possible, un nombre minimum de transactions de la plus petite taille possible est maintenant effectué (à l'inverse des 16 transactions automatiques précédemment).

Implémentation

La gestion de la mémoire globale est réalisée à l'aide d'un jeu de fonctions très semblables à ce qui existe dans le langage C : `cudaMalloc` pour l'allocation de la mémoire sur le GPU, `cudaFree` pour libérer la mémoire allouée et enfin `cudaMemcpy` pour copier de la mémoire depuis le GPU vers le CPU et inversement. Tout comme en C, `cudaMalloc` requiert une taille en octets. Ainsi, pour allouer un tableau de 30 `float` sur le GPU, il ne faut pas oublier de multiplier la taille du tableau par la taille en octets d'un `float` :

```
int             tabSize          = 30;
size_t          tabSizeInBytes   = tabSize * sizeof( float ) ;
float *         tab;
cudaError_t     retCode;

retCode    = cudaMalloc( (void **) &tab, tabSizeInBytes );
```

Code 3-22 : Utilisation de la fonction cudaMalloc.

À noter qu'il ne s'agit pas d'une erreur et qu'il est bien nécessaire de passer l'adresse du pointeur sur le tableau car celui-ci sera initialisé par la fonction ; le tout étant transtypé (casting) en `(void **)`.

La copie est réalisée à l'aide de `cudaMemcpy`. Avant l'appel d'un *kernel*, elle est généralement réalisée depuis le CPU vers le GPU en spécifiant `cudaMemcpyHostToDevice` comme dernier paramètre. Lorsque l'exécution d'un *kernel* est terminée, le rapatriement des données est réalisé à l'aide de la même fonction en spécifiant cette fois `cudaMemcpyDeviceToHost` pour copier les données du GPU vers le CPU. La syntaxe de la libération de l'espace mémoire est identique à celle du C.

```
float *    tab_cpu;

tab_cpu   = new float[ tabSize ];

[...] // Initialisation des valeurs contenues dans tab_cpu

// Copie des valeurs du tableau présent sur le CPU
// vers le tableau sur le GPU
retCode   = cudaMemcpy( tab, tab_cpu, tabSizeInBytes,
                        cudaMemcpyHostToDevice );

[...] // Appel d'un kernel

// Copie des valeurs mises à jour sur le GPU
// par l'appel de kernel vers le CPU
retCode   = cudaMemcpy( tab_cpu, tab, tabSizeInBytes,
                        cudaMemcpyDeviceToHost );

cudaFree( tab );
delete tab_cpu;
```

Code 3-23 : Utilisation de cudaMemcpy et cudaFree.

À noter que les fonctions d'allocation et de copie retournent un code erreur. Il est très important de bien vérifier que ces opérations ont été réalisées avec succès (le retour vaut `cudaSuccess` dans ce cas). Pour plus de détails sur le prototype et les différentes valeurs de retour possible de ces fonctions, une documentation en ligne très complète est disponible sur le site web de NVIDIA[11]. La partie 3.3.1 - Gestion des erreurs traite ce point plus en détail.

3.2.2 Mémoire partagée (shared memory)

Description

À l'inverse de la mémoire globale, la mémoire partagée dispose d'une faible capacité (de l'ordre de la dizaine de kilooctets) mais l'accès en lecture et en écriture ne subit aucune latence.

Cette mémoire est située sur les streaming processors. De ce fait, la même zone de mémoire partagée n'est accessible qu'aux threads qui sont exécutés dans un même bloc. Cette mémoire est généralement utilisée pour mutualiser, au sein d'un même bloc, la lecture de données. Pour l'utiliser sans risquer d'importantes erreurs, il est très souvent nécessaire de

[11] Documentation en ligne des fonctions de gestion de la mémoire (dernier accès le 09/12/2010) : http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/online/group__CUDART__MEMORY.html

synchroniser les threads d'un même bloc après l'écriture dans la mémoire partagée pour garantir une lecture cohérente des données par l'ensemble des threads du bloc.

Implémentation

La mémoire partagée est déclarée dans le *kernel* à l'aide du mot clé `__shared__`.

Si la taille nécessaire de mémoire partagée est connue à la compilation, la déclaration d'un tableau en mémoire partagée est identique à la déclaration d'un tableau statique que l'on aurait préfixé par le mot clé `__shared__` :

```
__global__ void kernel_memoire_partagee1()
{
   __shared__ float  memoire_partagee[ 1000 ];
   [...]
}
```

Code 3-24 : Déclaration de mémoire partagée statique.

Si par contre la taille de la mémoire est calculée de façon dynamique, l'implémentation est légèrement différente :

- Il faut d'une part déclarer le tableau dans le *kernel* sans préciser sa taille :

```
__global__ void kernel_memoire_partagee2()
{
   __shared__ float  memoire_partagee[];
   [...]
}
```

Code 3-25 : Déclaration de mémoire partagée dynamique.

- Et d'autre part, lors de l'appel du *kernel*, il est nécessaire de passer en troisième pseudo-paramètre la taille en octets de mémoire partagée souhaitée (après les dimensions de la grille et les dimensions des blocs) :

```
void ma_fonction( const int inValeur )
{
   dim3   dimGrille( 100, 100 );
   dim3   dimBloc( 512 );
   int    taille_memoire_partagee   = inValeur * sizeof( float );

   kernel_memoire_partagee2<<< dimGrille,
                               dimBloc,
                               taille_memoire_partagee >>>();
}
```

Code 3-26 : Appel d'un kernel avec une taille de mémoire partagée.

L'accès au tableau est par la ensuite réalisé de la même façon qu'un accès à n'importe quel tableau.

Illustration

Pour illustrer l'utilisation de la mémoire partagée, nous allons étudier l'exemple d'un filtre moyenneur très simple. Celui-ci devra seulement réaliser la moyenne de ses quatre voisins les plus proches et de lui-même (voir Figure 3-4).

Figure 3-4 : Exemple de l'application d'un filtre moyenneur sur un tableau de 10 entrées.

Le *kernel* suivant permet de simplement réaliser cette opération :

```
__global__ void moyenneur_gpu(
                                float *   inTableau,
                                float *   outTableau
                              )
{
   int    iX    = threadIdx.x + blockIdx.x * blockDim.x;
   float  total = 0;

   iX    += 2;
   for ( int i = -2; i <= 2; ++i )
   {
      total  += inTableau[ iX + i ];
   }
   outTableau[ iX ]  = total / 5;
}
```

Code 3-27 : Moyenneur – Kernel basique.

Le *kernel* est des plus simples. Il calcule tout d'abord l'indice courant du thread auquel il rajoute 2 car il n'est pas possible de calculer la moyenne sur les bords. Puis il réalise la somme des 2 éléments le précédant, de lui-même et des 2 éléments suivants, avant de

stocker la moyenne dans la cellule du tableau de sortie correspondante : le premier thread (d'indice 0) va donc calculer la moyenne à l'indice 2 et ainsi de suite.

Ce code ne réalise aucun test pour savoir si l'indice du thread est supérieur à celui de la mémoire, il devra donc être appelé avec exactement le nombre de threads nécessaires. Pour notre tableau précédent de 10 valeurs, il aurait fallu 6 threads (de manière générale, le nombre de threads nécessaires est égal au nombre de valeurs du tableau moins la taille du filtre diminuée de 1 — 10 - (5 - 1) dans notre cas).

Cette méthode fonctionne correctement mais la majorité des cellules du tableau (à l'exception des bords) est chargée 5 fois depuis la mémoire globale. En effet, `inTableau` et `outTableau` sont en mémoire globale, chaque accès à ces tableaux correspond donc à de la lecture ou de l'écriture en mémoire globale. Dans l'exemple précédent, ce n'est le cas que des cellules d'indice 4 et 5 (qui stockent respectivement 5 et 7) mais ceci est dû à la petite taille de notre tableau. Avec un tableau plus grand, on conserverait toujours nos 8 cellules qui réalisent moins de 5 chargements sur les bords du tableau mais l'ensemble des autres cellules réaliseraient 5 chargements. On peut donc très vite avoir un nombre de chargements très proche de 5 fois la taille du tableau (4980 chargements pour un tableau de 1000 éléments).

L'idée est donc de mutualiser ces chargements qui sont très lents de manière à se rapprocher le plus possible d'un seul chargement en lecture en mémoire globale par élément du tableau. Pour cela, on va réaliser le chargement en mémoire partagée de chaque valeur nécessaire aux threads d'un *bloc*. Celle-ci ne sera réalisée qu'une fois et permettra aux threads d'un même *bloc* de n'avoir plus qu'à accéder aux valeurs en mémoire partagée (l'accès y est beaucoup plus rapide).

Pour tirer parti au maximum de la mémoire partagée, il est nécessaire de travailler avec des *blocs* assez grands. Le choix ici est de travailler avec des *blocs* de 512 *threads* (tous présents dans l'axe x). Ce choix est discutable une fois de plus et des tests plus poussés permettraient de l'affiner en fonction du matériel utilisé. Chacun des 512 *threads* d'un *bloc* remontera une valeur depuis la mémoire globale vers la mémoire partagée mais seuls 508 d'entre eux seront utiles pour effectuer le calcul de la moyenne. En effet, il est nécessaire de disposer des deux valeurs précédentes et des deux valeurs suivantes pour effectuer la moyenne. Or,

pour les deux cellules à chaque extrémité, il va manquer certaines de ces données en mémoire partagée (voir Figure 3-5).

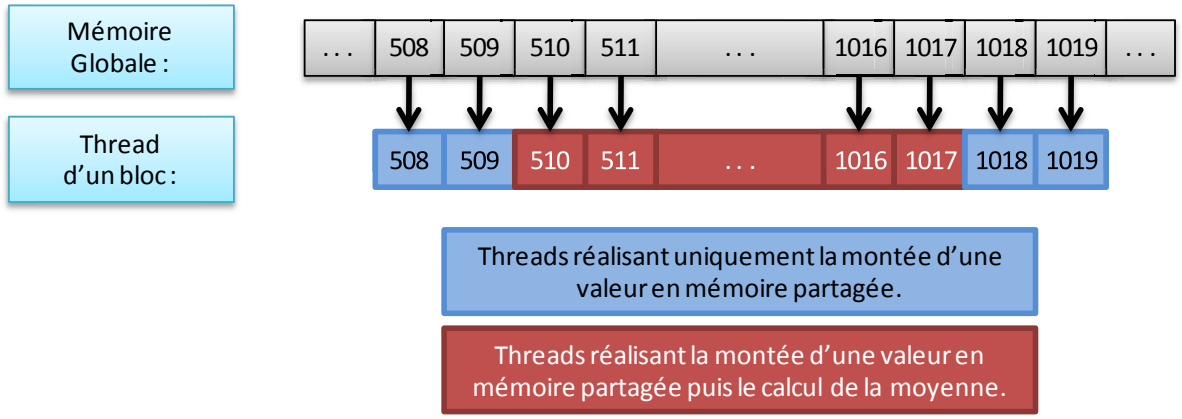


Figure 3-5 : Action réalisée par les différents threads d'un bloc.

Pour simplifier encore l'exercice, nous allons de nouveau partir du postulat que la taille du tableau et le nombre de blocs utilisés concordent et qu'il n'est donc pas nécessaire de tester si l'on dépasse l'indice maximum du tableau. Étant donné que l'on n'utilise que 508 threads sur les 512 d'un bloc, le tableau doit donc faire une taille égale au nombre de blocs multiplié par 508 auquel il est nécessaire d'ajouter les deux premiers et les deux derniers éléments du tableau qui ne seront jamais calculés :

```
tailleTableau    = nbBloc * 508 + 4;
```

Code 3-28 : Moyenneur - Calcul du nombre de threads nécessaires.

Pour bien comprendre le code, celui-ci va être expliqué en plusieurs étapes.
La première consiste à définir quelques variables permettant de mieux comprendre la suite.

```
const int   taille_block_x   = 512;
const int   nb_cell_moy      = 5;
const int   nb_cell_tampon   = 2;
const int   nb_thd_calc_x    = 508;
```

Code 3-29 : Moyenneur – Variables utilisées pour simplifier les calculs.

Ces variables sont assez explicites :

- `taille_bloc_x` correspond au nombre de threads présents dans un bloc : 512 dans notre cas.
- `nb_cell_moy` correspond au nombre de cellules du tableau utilisées pour le calcul d'une moyenne : nous avons choisi ici de calculer une moyenne sur cinq cellules.

- `nb_cell_tampon` correspond au nombre de cellules inutilisées sur les bords (cela correspond à nb_cell_moy / 2 mais le choix a été fait de le rentrer en dur ici pour plus de lisibilité).
- `nb_thd_calc_x` correspond au nombre de threads d'un bloc qui vont effectivement réaliser le calcul de la moyenne. Nous avons vu précédement que seuls 508 threads sont utilisés sur les 512 présents (encore une fois, cette valeur peut être calculée simplement : `taille_bloc_x - 2 * nb_cell_tampon`).

Nous allons maintenant voir la première partie du kernel implémentant le moyenneur :

```
__global__ void moyenneur_gpu(
                                float *   inTableau,
                                float *   outTableau
                              )
{
   int    thX      = threadIdx.x;
   int    iX       = threadIdx.x + blockIdx.x * nb_thd_calc_x;
   float  total    = 0;

   __shared__ float  sh_mem[ taille_block_x ];

   sh_mem[ thX ]   = inTableau[ iX ];

   __syncthreads();
```

Code 3-30 : Moyenneur – Kernel avec mémoire partagée 1/2.

On commence dans cette partie par récupérer l'indice local du thread `thX` (ce n'est utile que pour la lisibilité du code) et calculer l'indice de calcul `iX` (différent de l'indice global car l'indice du *bloc* est multiplié par le nombre de thread réalisant un calcul — `nb_thd_calc_x` : 508 — et non pas la largeur d'un bloc — `blockDim.x` : 512). On alloue ensuite statiquement un segment de mémoire partagée de la taille du bloc à l'aide du mot clé `__shared__`. Puis chacun des threads remonte une valeur depuis la mémoire globale vers la mémoire partagée. Enfin, et il est très important de ne pas l'oublier, il est nécessaire de réaliser une synchronisation entre l'ensemble des threads du bloc pour s'assurer qu'aucun thread ne réalisera la suite des calculs avant que l'ensemble de la mémoire partagée n'ait été chargée. Cette synchronisation s'effectue par l'appel de la fonction `__syncthreads()` qui place une sorte de rendez-vous à tous les threads d'un bloc à cet endroit précis du code. Aucun thread du bloc ne peut alors exécuter de code après cette synchronisation tant que l'ensemble des threads n'ont pas atteint cette instruction.

```
  if ( thX > nb_cell_tampon && thX < taille_block_x - nb_cell_tampon )
  {
     for ( int i = -nb_cell_tampon; i < nb_cell_tampon; ++i )
     {
        total    += sh_mem[ thX + i ];
     }
     outTableau[ iX ]   = total / nb_cell_moy;
  }
}
```

Code 3-31 : Moyenneur – Kernel avec mémoire partagée 2/2.

Une fois la mémoire partagée chargée complétement, il suffit pour 508 des 512 threads de réaliser un calcul de moyenne (les *threads* d'indice 2 à 509 — ni les deux premiers, ni les deux derniers). Chacun d'eux réalise alors la moyenne des deux éléments le précédant, de lui-même et des deux élements lui succédant en accèdant à la mémoire partagée puis affecte le résultat en mémoire globale. Par exemple, le *thread* numéro 2 du *bloc* 1 va faire la somme des cinq premières valeurs présentes en mémoire partagée (indices 0 à 4) puis diviser le résultat par 5 et enfin l'affecter à l'indice 510 de la mémoire globale.

Et ce code est suffisant pour le *kernel* utilisant la mémoire partagée. Il est plus compliqué que le précédent mais aussi plus efficace. On réalise moins de chargements en mémoire globale (même si l'on n'atteint pas la barre de l'unique chargement par élément). On réalise par contre 5 chargements en mémoire partagée et on utilise un plus grand nombre de threads mais en revanche le gain de près de 4 chargements en mémoire globale est bien plus important.

Pour clore cet exemple, voici le contenu du programme principal (`main` ci-dessous). Une de ses particularités vient du fait que l'on décide de la taille du tableau en fonction du nombre de *blocs* utilisés. Encore une fois, il s'agit là d'une simplification ; en temps normal, c'est bien sûr l'inverse qui est réalisé mais il faut alors rajouter des tests dans le *kernel* pour ne pas accéder à des indices supérieurs à ceux du tableau. Pour le reste, comme l'immense majorité des programmes CUDA, il commence par les allocations mémoire, suivi de la copie du tableau en mémoire globale. L'appel du *kernel* a lieu une fois la copie terminée et le programme se termine avec la copie du tableau de résultats.

```
int main( int argc, char ** argv )
{
   float *  tab_host;
   float *  tab_device_in;
   float *  tab_device_out;

   int      nbBloc              = 3;
   int      tailleX             = nbBloc * 508 + 4;

   dim3     dimGrille( nbBloc, 1, 1 );
   dim3     dimBloc( taille_block_x, 1, 1 );

   int      tailleTotale        = tailleX;
   int      tailleTotaleOctets  = tailleTotale * sizeof( float );

   // Allocation mémoire
   tab_host       = new float[ tailleTotale ];
   cudaMalloc( (void **) &tab_device_in, tailleTotaleOctets );
   cudaMalloc( (void **) &tab_device_out, tailleTotaleOctets );

   // Initialisation du tableau
   for ( int i = 0; i < tailleTotale; ++i )
   {
      tab_host[ i ] = rand() % 255;
   }

   // Copie aller
   cudaMemcpy(
               tab_device_in,
               tab_host,
               tailleTotaleOctets,
               cudaMemcpyHostToDevice
             );

  // Appel du kernel
   moyenneur_gpu<<< dimGrille, dimBloc >>>(
                                             tab_device_in,
                                             tab_device_out
                                           );

   // Copie retour
   cudaMemcpy(
               tab_host,
               tab_device_out,
               tailleTotaleOctets,
               cudaMemcpyDeviceToHost
             );

   cudaFree( tab_device_in );
   cudaFree( tab_device_out );
   delete[] tab_host;
}
```

Code 3-32 : Moyenneur – main.

3.2.3 Mémoire constante (constant memory) et mémoire des textures (texture memory)

Description

Ces deux mémoires sont des mémoires dont l'accès ne peut être réalisé depuis le GPU qu'uniquement en lecture. De plus, tout comme la mémoire globale, ces mémoires sont accessibles à l'ensemble des threads de la grille et elles sont persistantes entre différents appels de *kernel*.

La mémoire constante est « cachée » : elle nécessite donc une lecture en mémoire lors d'un défaut de cache (cache miss) et une lecture dans le cache constant dans le cas contraire. Ce cache est réalisé de telle manière que si l'ensemble des threads d'un demi-*warp* (16 *threads* sur les 32) accèdent à la même adresse en mémoire cachée, l'accès est aussi rapide que l'accès à un registre. Cependant, pour optimiser les calculs, NVIDIA recommande que l'ensemble des 32 threads du *warp* accèdent à la même adresse mémoire car les prochaines versions le requerront pour une utilisation optimale.
Dans les versions actuelles, la mémoire constante est composée de 64ko pour l'ensemble de la carte et chaque *streaming multiprocessor* possède 8ko de cache.

Tout comme la mémoire constante, la mémoire des textures est cachée : elle nécessite donc une lecture en mémoire lors d'un défaut de cache et une lecture dans le cache de texture dans le cas contraire. À l'inverse de la mémoire constante, qui est optimisée pour la lecture de la même zone mémoire par l'ensemble des *threads* d'un *warp*, la mémoire de texture est optimisée pour un accès en deux dimensions : l'accès est donc optimisé si les threads d'un même *warp* accèdent à des adresses mémoires proches les unes des autres.
La mémoire de texture est mappée sur la mémoire globale. Elle peut donc potentiellement atteindre des tailles très importantes. Mais le cache est limité à quelques kilooctets par *streaming multiprocessor* ; l'utilisation de textures trop grandes n'est donc pas efficace.

Implémentation

L'utilisation de la mémoire constante est assez simple, il suffit dans le code de l'hôte (CPU) de créer une variable préfixée par le mot-clé CUDA `__constant__` puis de copier la mémoire depuis le CPU vers le GPU à l'aide de la commande `cudaMemcpyToSymbol`. Cette

fonction est équivalente à la fonction `cudaMemcpy` à l'exception du sens de copie qui n'est pas précisé : il s'agit de mémoire en lecture seule, la copie retour n'est donc jamais utile.

```
// Allocation mémoire
__constant__ float tabConst_GPU[256];
             float tab_CPU[256];

// Initialisation du tableau

// Copie aller
cudaMemcpyToSymbol( tabConst_GPU, tab_CPU, sizeof( tab_CPU ) );

// Appel du kernel
[...]
```

Code 3-33 : Utilisation de la mémoire constante.

L'utilisation de la mémoire de texture est plus compliquée. Elle est réalisée en allouant de la mémoire globale puis en liant cette mémoire à un cache défini préalablement et par lequel seront réalisés les accès mémoire. Nous allons étudier ici uniquement l'implémentation de la texture la plus simple : il s'agit de la texture sur un tableau à une seule dimension.

Le masque permettant de « cacher » la mémoire globale pour en faire de la mémoire de texture est de type `texture< Type, Dim, ReadMode >`. Il doit être déclaré de manière globale. Il s'agit d'un type générique qui prend trois paramètres génériques :

- Le type de donnée contenu.
- Le nombre de dimensions de la texture. (1 par défaut.)
- Le mode de lecture du cache : `cudaReadModeElementType` ou `cudaReadModeNormalizedFloat`. Si le type est entier et que ce mode de lecture vaut `cudaReadModeNormalizedFloat`, les nombres retournés ne seront alors pas des entiers mais des nombres à virgules flottantes compris en 0 et 1 ou -1 et 1 selon qu'il s'agit d'entiers non signés ou signés. (`cudaReadModeElementType` par défaut.)

```
// Definition d'une texture de flottant à une dimension
texture< float, 1, cudaReadModeElementType > texRef;
```

Code 3-34 : Définition d'une texture.

D'autres paramètres (des attributs de `texture`) peuvent être spécifiés à l'exécution par l'hôte :

- `normalized` permet de spécifier si l'accès se fait par coordonnées normalisées entre 0 et 1 (lorsque `normalized` est différent de 0) ou à partir des valeurs classiques entre 0 et N - 1 pour une texture de taille N (lorsque `normalized` vaut 0).
- `addressMode[ 3 ]` permet de spécifier pour chaque dimension le traitement réalisé lorsque les valeurs d'adressage ne sont pas dans l'intervalle possible. Si la variable

prend la valeur `cudaAddressModeClamp`, les valeurs inférieures à 0 valent 0 et celles supérieures ou égales à N valent N - 1. Si la variable prend la valeur `cudaAddressModeWrap` et que `normalized` est différent de 0 (adressage normalisé), seule la partie fractionnée est utilisée. Elle est conservé intacte si le nombre était positif et on la soustrait au nombre 1 si elle était négative (1.25 devient 0.25 ou encore -1.25 devient 0.75).

- `filterMode` permet de spécifier si l'on souhaite que soit retourné l'élément le plus proche des coordonnées voulues (`cudaFilterModePoint`) ou une interpolation linéaire des éléments proches (`cudaFilterModeLinear`).

```
texRef.normalized  = 0; // Non normalisé
texRef.addressMode[ 0 ] = cudaAddressModeClamp; // Entre 0 et N-1
texRef.filterMode = cudaFilterModePoint; // Valeur la plus proche
```

Code 3-35 : Spécification des paramètres d'une texture.

Il faut ensuite lier cette texture à la mémoire sur le GPU à l'aide de la fonction `cudaBindTexture`. Il est nécessaire de passer à cette fonction la texture à lier, la mémoire sur laquelle elle sera liée et la taille de la partie de la mémoire liée. À la fin de l'utilisation de la texture, il est nécessaire de délier la texture manuellement.

```
float *   tab_CPU;
float *   tab_GPU;

// Allocation des tableaux
tab_CPU = new float[ tailleTotale ];
cudaMalloc( (void **) &tab_GPU, tailleTotaleOctets );

// Initialisation du tableau sur le CPU
for ( int i = 0; i < tailleTotale; ++i )
{
   tab_CPU[ i ]  = rand() % 255;
}

// Copie vers le GPU
cudaMemcpy( tab_GPU, tab_CPU, tailleTotaleOctets,
cudaMemcpyHostToDevice );

// Liaison de la texture
cudaBindTexture( 0, texRef, tab_GPU, tailleTotaleOctets );

// Appel du kernel utiliant la texture
kernel_texture<<< dimGrille, dimBloc >>>();

// La mémoire de texture est déliée
cudaUnbindTexture( texRef );
```

L'accès à la mémoire par la texture se fait dans le *kernel* à l'aide de la fonction `tex1Dfetch` qui prend en paramètre la texture et l'indice voulu :

```
__global__ void kernel_texture()
{
   int    iX   = threadIdx.x + blockIdx.x * blockDim.x;
   float  val  = tex1Dfetch( texRef, iX );
   [...]   // Traitement de la valeur
}
```

Code 3-36 : Utilisation d'une texture.

3.2.4 Registres (registers)

Les registres sont des mémoires accessibles en lecture et en écriture, propres à chaque thread et qui possèdent généralement des temps d'accès sans latence.

Malgré tout, dans certains cas (vraie dépendance - read before write - et les conflits de banque), une latence peut apparaître. L'utilisation de plus de 192 threads par streaming multiprocessor permet de masquer cette latence. De même, pour une utilisation optimale des registres, chaque bloc doit posséder un nombre de threads multiple de 64.

Les variables automatiques (≈variables locales - ex : "`int thX;`") définies dans un kernel sont stockées dans un registre dans la limite des registres disponibles. Et c'est bien la limite du nombre de registres disponibles qui limite leurs utilisations : seulement 16384 registres sont disponibles par *streaming multiprocessor* pour les cartes graphiques de version 1.3. Cela correspond à 32 registres par threads si un seul *bloc* comprenant 512 *threads* est ordonnancé par *streaming multiprocessor*. Dans le cas de *kernels* complexes, il n'est pas rare d'avoir à utiliser davantage de variables automatiques qui ne pourront donc être toutes stockées dans les registres ce qui va avoir pour effet de ralentir l'exécution. Ces variables peuvent alors être stockées en mémoire partagée. Si il n'y a plus d'espace mémoire disponible en mémoire partagée, elles seront alors stockées en mémoire locale (décrite ci-après). Cependant l'utilisation de la mémoire locale est à éviter au maximum car cela ralentit grandement le traitement (les temps d'accès sont les mêmes que ceux de la mémoire globale).

3.2.5 Mémoire locale (local memory)

La mémoire locale possède des caractéristiques proches de la mémoire globale : l'accès, possible en lecture et en écriture, possède, du fait de l'absence de cache, une très forte latence. La différence majeure avec la mémoire globale est que la mémoire locale est propre à chaque thread. Elle est utilisée pour stocker les variables automatiques représentant des structures de tailles importantes ou des tableaux qui nécessiteraient trop de registres pour

réaliser leurs stockages. À noter qu'à la différence de la mémoire globale, le fait que la mémoire locale soit propre à un thread garantit des accès coalescents à la mémoire.

3.3 Autres éléments de l'API CUDA

3.3.1 Gestion des erreurs

« C for CUDA » étant très proche du C, la remontée des erreurs est donc réalisée à l'aide de codes d'erreur qui sont soit retournés par les fonctions lors de leurs appels (lors de la copie de la mémoire par exemple), soit stockés sur le GPU et mis à la disposition de l'hôte s'il les demande (dans le cas des appels de *kernels* par exemple).

Ces codes d'erreur sont très importants car ils permettent de déceler toute une catégorie de problèmes qu'il serait très difficile de déceler sans leurs supports. Parmi ces problèmes, on trouve par exemple l'absence de carte GPGPU utilisable ou encore un appel de *kernel* impossible à réaliser.

3.3.1.1 Le type erreur : cudaError_t

Les erreurs sont de type `cudaError_t`. Ce type n'est autre que la définition d'un type (`typedef`) sur l'énumération `cudaError`. La description de l'ensemble des valeurs que peut prendre cette énumération est disponible sur le site internet de NVIDIA[12]. Mais le plus important est de savoir que l'ensemble de ces valeurs à l'exception de `cudaSuccess` correspond à une erreur, pour un motif ou pour un autre. Pour s'assurer que les opérations se sont déroulées normalement, le minimum suffisant est donc de tester le code erreur pour vérifier si la valeur correspond bien à `cudaSuccess`. Mais le nombre important de codes d'erreur (48 codes différents en plus du succès) permet d'avoir une vision plus large de l'erreur et il est souvent très utile de connaître précisément l'erreur survenue. Ainsi, deux erreurs très courantes sont :

- `cudaErrorLaunchFailure` lorsque l'appel d'un *kernel* échoue. Ceci est généralement dû aux dimensions des *blocs* ou/et de la *grille* qui n'appartiennent pas aux intervalles acceptés par la carte.
- `cudaErrorNoDevice` lorsque qu'aucune carte n'est visible par l'hôte sur lequel est lancé l'exécution de l'application. Cela peut aussi bien dire que la carte n'est pas

[12]Définition de l'énumération cudaError sur le site de NVIDIA (dernier accès le 20/03/2012) : http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__TYPES_g3f51e3575c2178246db0a94a430e0038.html#g3f51e3575c2178246db0a94a430e0038

présente physiquement que signifier une absence ou une mauvaise installation des pilotes de la carte.

3.3.1.2 *Récupération des codes d'erreur*

La récupération des codes d'erreur est effectuée de deux façons différentes selon qu'il est possible ou non pour les fonctions de les renvoyer.

Aussi souvent que cela est possible, pour les fonctions de l'API, c'est la fonction appelée qui retourne le code erreur. C'est le cas par exemple de la fonction `cudaMalloc` ou encore de la fonction `cudaMemcpy`. La récupération du code d'erreur est alors des plus classiques :

```
cudaError_t   retCode;

retCode   = cudaMalloc( (void **) &tab, tabSizeInBytes );
if ( retCode != cudaSuccess )
{
   // Traitement de lerreur.
}
```

Code 3-37 : Utilisation des codes erreurs de façon synchrone.

Mais le cas d'un appel de kernel est différent. Le code erreur est alors récupéré à l'aide de la fonction `cudaGetLastError`. Comme son nom l'indique, cette fonction retourne le code erreur correspondant au dernier appel de *kernel*. L'appel de cette fonction après l'appel du *kernel* permet donc de récupérer et traiter l'erreur :

```
cudaError_t   retCode;

kernelExemple1<<< 1, 10 >>>();   // Appel du kernel
retCode   = cudaGetLastError();  // Recuperation de lerreur
if ( retCode != cudaSuccess )
{
   // Traitement de lerreur.
}
```

Code 3-38 : Utilisation des codes erreurs de façon asynchrone.

3.3.2 Calcul en nombres flottants

Sur les architectures précédant Fermi, l'implémentation par défaut sur les GPU NVIDIA des calculs sur les nombres flottants ne suit pas précisément la norme IEEE-754 (NVIDIA 2007). Ceci est dû au but original des GPU qui ne nécessitait pas une précision rigoureuse pour un rendu graphique réaliste. Ainsi, pour améliorer la vitesse d'exécution, quelques petites libertés ont été prises sur le standard.

Pour autant, l'API de CUDA dispose de fonctions réalisant les arrondis de manière fidèle à la norme IEEE-754. Ces fonctions sont les suivantes :

- `__fmaf_r{n,z,u,d}( float, float, float )` : multiplication et addition.
- `__frcp_r{n,z,u,d}( float )` : inverse.
- `__fdiv_r{n,z,u,d}( float, float )` : division.
- `__fsqrt_r{n,z,u,d}( float )` : racine carrée.
- `__fadd_r{u,d}( float, float )` : somme.
- `__fmul_r{u,d}( float, float )` : multiplication.

Les quatre modes d'arrondi spécifiés par le standard IEEE sont donc disponibles pour la majorité de ces fonctions :

- `n` : arrondi au plus près.
- `z` : arrondi vers zéro.
- `u` : arrondi vers plus l'infini.
- `d` : arrondi vers moins l'infini.

Exemple :

```
float    a   = 10;
float    b   = 3;
float    c   = 10 / 3;                 // 3.33333349227905

float    cn  = __fdiv_rn( a, b );      // 3.33333325386047
float    cz  = __fdiv_rz( a, b );      // 3.33333325386047
float    cu  = __fdiv_ru( a, b );      // 3.33333349227905
float    cb  = __fdiv_rd( a, b );      // 3.33333325386047
```

Code 3-39 : Différentes versions de la division de flottant.

L'utilisation de ces différentes fonctions d'arrondi conformes au standard IEEE-754 n'est pas innocente. S'il a été choisi d'utiliser par défaut des arrondis non conformes, c'est que ces derniers sont plus rapides à exécuter sur le GPU. Par conséquent, l'utilisation d'implémentations conformes au standard IEEE entraîne généralement une diminution des performances du programme sur les architectures pré-Fermi. Il est donc important de ne les utiliser que lorsqu'elles sont nécessaires et de ne pas les substituer de manière automatique aux implémentations par défaut.

3.3.3 Utilisation des flux asynchrones (stream)

Les échanges mémoires entre le CPU et le GPU peuvent être un vrai goulot d'étranglement pour l'application. Lorsque ceux-ci sont trop importants et que les algorithmes employés le permettent, il est alors possible de réaliser la copie de la mémoire de façon asynchrone. L'idée est donc de réaliser une partie de la copie de la mémoire en même temps que l'on

exécute le *kernel* sur une autre partie de la mémoire copiée précédemment (Qiang-qiang, et al. 2011).

Dans l'exemple du diagramme de séquence suivant (Figure 3-6), on réalise la copie d'une première partie de la mémoire (partie1). Une fois cette copie terminée, on lance le *kernel* sur cette partie de la mémoire alors que dans le même temps, on réalise la copie de la deuxième partie de la mémoire (partie2). Une fois la copie de la deuxième partie de la mémoire finie, on exécute le *kernel* sur cette partie de la mémoire.

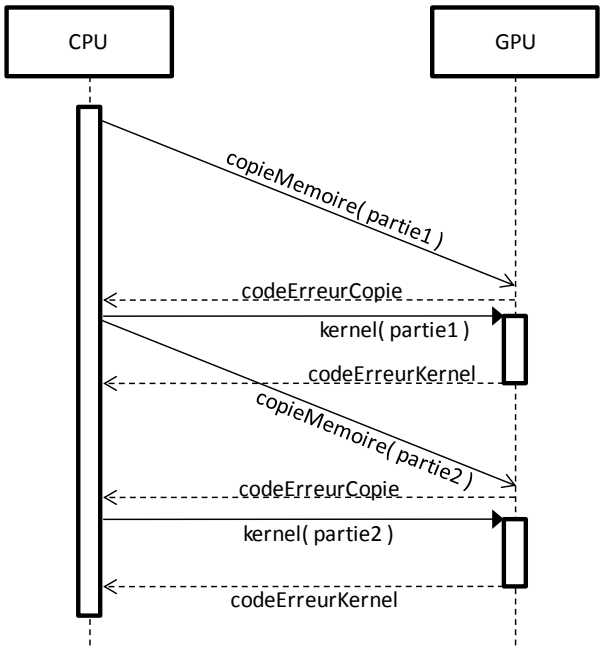


Figure 3-6 : Diagramme de séquence de l'utilisation de flux asynchrones.

Plus la taille de la mémoire utilisée est importante et plus cette méthode peut permettre d'accélérer les calculs. Pour autant, elle nécessite que les calculs effectués par le *kernel* sur des éléments d'une zone mémoire puissent être réalisés indépendamment du reste de la mémoire.

Pour permettre ce mécanisme, le type `cudaStream_t` est utilisé pour déclarer des flux qui sont créés à l'aide de la fonction `cudaStreamCreate` :

```
int           nbFlux   = 5;
cudaStream_t  tabStream[ nbFlux ];
for ( int i = 0; i < nbFlux; ++i )
{
    cudaStreamCreate( &tabStream[ i ] );
}
```

Code 3-40 : Initialisation de flux CUDA.

On réalise ensuite la copie des différentes parties de la mémoire à l'aide des flux. Pour cela, il est nécessaire d'utiliser la fonction `cudaMemcpyAsync` qui requiert, en plus des paramètres de copie classique (destination, source, quantité d'octets copiée et direction de la copie), un flux. Le passage d'un flux en paramètre de la copie va permettre de connaître l'évolution de la copie.

```
int      tailleParFlux  = 100;
int      taille         = tailleParFlux * nbFlux;
float *  tab_host;
float *  tab_device;

tab_host   = new float[ taille ];
cudaMalloc( (void **) & tab_device, taille * sizeof( float ) );

for ( int i = 0; i < nbFlux; ++i )
{
    cudaMemcpyAsync(
                    tab_device + i * tailleParFlux * sizeof( float ),
                    tab_host + i * tailleParFlux * sizeof( float ),
                    tailleParFlux * sizeof( float ),
                    cudaMemcpyHostToDevice,
                    tabStream[ i ]
                  );
}
```

Code 3-41 : Utilisation des flux CUDA avec cudaMemcpyAsync.

L'appel de chaque *kernel* est lui aussi paramétré par un flux. De cette manière, l'appel ne sera effectué que lorsque le transfert de mémoire correspondant au flux sera terminé.

```
for ( int i = 0; i < nbFlux; ++i )
{
    monKernel<<< 1, tailleParFlux, 0, tabStream[ i ] >>>
        (
          tab_device + i * tailleParFlux * sizeof( float ),
          tailleParFlux
        );
}
```

Code 3-42 : Utilisation des flux CUDA avec un kernel.

Conclusion

Parmi les interfaces de programmation existantes au début de mes travaux, le choix a été fait de travailler avec l'API proposée par NVIDIA : CUDA. Cette API, alors strictement réservée à la programmation sur carte NVIDIA (la dernière version du kit de développement laisse entrouverte la porte à son utilisation sur d'autres matériels), était plus performante que l'API OpenCL. Ces meilleures performances étaient rendues possibles par le choix fait

par NVIDIA de développer des cartes graphiques dédiées au calcul généraliste ce qui lui a permis de garder une longueur d'avance sur la concurrence.

Une fois l'API choisie, une bonne connaissance de celle-ci est nécessaire pour obtenir le maximum de puissance du GPU. Il est ainsi important de comprendre le système de gestion des threads à travers des blocs et une grille et en particulier les mécanismes d'ordonnancements de ces différents éléments. Une fois ces concepts intégrés, il est possible de paralléliser assez simplement une application. La conception d'une application très performante sur GPU est beaucoup plus compliquée.

Les accès mémoire sur GPU étant beaucoup moins performants que ce qui se fait sur CPU, une attention particulière doit être donnée à l'utilisation de la mémoire. Pour améliorer les performances des accès mémoire, plusieurs types de mémoires, aux caractéristiques différentes, sont disponibles. Parmi celles-ci, la mémoire globale est ainsi accessible à l'ensemble des threads mais subit une latence importante. Les mémoires de texture et constante sont également accessibles à l'ensemble des threads mais l'utilisation d'un cache permet des accès (en lecture uniquement) plus efficaces. Au sein d'un bloc, la mémoire partagée, qui ne subit aucune latence, peut quant à elle être utilisée pour mutualiser des accès.

L'API CUDA permet également de traiter efficacement les erreurs produites par les appels sur le GPU, que ce soit des appels synchrones ou asynchrones. Un traitement fin des opérations en virgules flottantes permet de plus de régler, selon l'application, entre précision des calculs et vitesse d'exécution de ceux-ci. L'API CUDA peut également être utilisée pour maximiser l'utilisation du GPU en réalisant la copie de mémoire entre le CPU et le GPU en même temps que sont réalisés des traitements. Pour cela, en plus de la copie classique de mémoire, une copie asynchrone est proposée par l'API. Cette copie asynchrone, couplée à l'utilisation de flux permettant de suivre l'évolution des copies, permet de paralléliser les deux traitements.

Le lecteur intéressé par plus de détail pourra également consulter un des ouvrages suivants : « Programming Massively Parallel Processors - A Hands-on Approach » (Kirk, Wen-mei et Hwu 2010) ou « CUDA by example » (Sanders et Kandrot 2010) qui introduisent la programmation sur GPU à l'aide de l'API CUDA de façon simple et très pédagogique.

Maintenant que nous avons expliqué l'API CUDA, nous sommes en mesure de développer nos propositions en matière d'algorithmique parallèle. Le prochain chapitre traitera des solutions apportées dans trois domaines : le traitement rapide de grands automates cellulaires en trois dimensions, le calcul des champs physicochimiques sur GPU et la parallélisation des réplications de simulations stochastiques en général.

Simulation d’automates cellulaires en 3D et champs de force

4 Simulation d'automates cellulaires en 3D et champs de force

Introduction

Nous avons vu précédemment différentes solutions pour améliorer les performances d'une application. On trouve, d'une part des solutions logicielles à travers l'utilisation de nouveaux algorithmes ou l'amélioration de l'implémentation d'algorithmes existants et, d'autre part, des solutions matérielles à travers l'utilisation d'architectures parallèles, telles que les GPU, qui peuvent être développées à l'aide de l'API CUDA. Afin d'améliorer les performances des simulations réalisées à l'aide de la PGMS, je me suis donc employé à proposer et à mettre en œuvre différentes solutions. Ces solutions procèdent d'une démarche d'intégration tirant parti des différents outils disponibles, aussi bien matériels que logiciels, pour améliorer les performances des différentes composantes de l'application.

Au registre des moyens matériels utilisés pour accélérer les calculs, j'ai privilégié l'utilisation de processeurs graphiques à des fins de calculs généralistes (GP-GPU) pour les calculs les plus lourds. Cette technologie a été utilisée pour paralléliser d'une part un algorithme permettant de faire évoluer les automates cellulaires à trois dimensions et d'autre part le calcul des champs physicochimiques. Cette seconde parallélisation a par la suite été intégrée à la PGMS afin d'améliorer ses performances.

L'accélération logicielle a également nécessité un investissement significatif. Ceci a amené aussi bien à réaliser des modifications profondes en implémentant un nouvel algorithme (l'algorithme Hash-Life en trois dimensions), qu'à réaliser des modifications plus fines, plus proches du génie logiciel, mais pouvant amener également des résultats tout à fait significatifs. Ces modifications ont permis d'optimiser l'implémentation de la PGMS afin de supprimer, autant faire se peut, les goulots d'étranglement en améliorant, en particulier, les accès mémoire.

Deux idées majeures ont guidé mes choix. D'une part, il était important de ne pas s'enfermer dans une catégorie d'améliorations mais plutôt d'essayer de trouver la combinaison la mieux adaptée à un problème donné. D'autre part, l'objectif à terme est de permettre à la PGMS de pouvoir fonctionner de manière rapide et efficace sur une machine autonome, tel qu'un simple poste de travail. Cela a donc parfois conduit à écarter certaines technologies comme, par exemple, l'utilisation de fermes ou de grilles de calcul.

Pour satisfaire ces deux contraintes, notre objectif a été de permettre à la simulation une exécution la plus rapide possible sur une machine de bureau équipée d'une carte GPU conçue pour effectuer des calculs généralistes.

La première partie de ce chapitre détaillera l'implémentation sur GPU du traitement des automates cellulaires ainsi que l'utilisation de l'algorithme Hash Life dans un modèle en trois dimensions (la présentation de l'algorithme pour un modèle en deux dimensions est présente dans la partie 2.2.2 - L'algorithme Hash-Life~~L'algorithme Hash-Life~~). La seconde partie traitera de l'implémentation du calcul de champs physiques (gravitationnel, électrique et magnétique). Enfin, une dernière partie présentera l'utilisation de la puissance des GPU afin de réaliser efficacement des réplications de simulations stochastiques.

4.1 Automates cellulaires 3D sur GPU et CPU (Hash-Life)

Lorsque l'on s'intéresse au monde du vivant, il est très fréquent, pour disposer de modèles très précis, que ces derniers soient des modèles en trois dimensions. Si les algorithmes appliqués aux automates cellulaires ont été largement étudiés pour des modélisations en deux dimensions, l'étude des algorithmes en trois dimensions est beaucoup moins fournie. Il a donc été nécessaire pour utiliser des algorithmes traitant de problèmes en trois dimensions, soit de porter des algorithmes déjà existants en deux dimensions (c'est le cas pour l'implémentation de l'algorithme Hash-Life en trois dimensions (Caux, Siregar et Hill 2010)), soit d'implémenter un nouvel algorithme (c'est le cas pour l'implémentation sur GPU (Caux, Siregar et Hill 2011)).

4.1.1 L'algorithme Hash-Life en 3D

4.1.1.1 Explication de l'algorithme

Le portage en trois dimensions de l'algorithme Hash-Life (Gosper 1984) engendre deux principales conséquences. La première est bien évidemment une modification assez importante de l'algorithme, même si l'idée de base, qu'est l'utilisation de la mémoïzation (Michie 1968), reste la même. La seconde est une modification tout aussi importante des caractéristiques de l'algorithme et donc du champ d'application.

L'algorithme

La modification la plus importante de l'algorithme est bien entendu le passage d'une représentation des données en deux dimensions sous forme de quadtrees à son équivalent en trois dimensions avec des octrees (Meagher 1982). Tout comme le quadtree, l'octree est une structure de données sous forme d'arbre où chaque nœud est soit une branche, soit une feuille terminale. La différence repose dans le fait que les branches ne possèdent plus entre 1 et 4 nœuds fils mais entre 1 et 8 nœuds fils : les 4 nœuds avant et les 4 nœuds arrière (8 dans le cas d'un octree complet – voir Figure 4-1~~Figure 4-1~~).

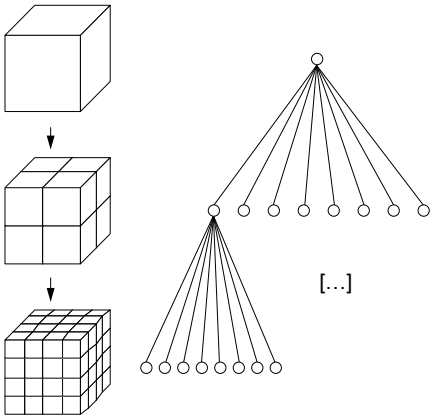


Figure 4-1: Représentation en octree complet.

Bien entendu, ce changement de représentation a un impact majeur sur l'ensemble de l'implémentation. Les tables de hachage, élément principal de la solution, se trouvent ainsi modifiées, passant d'une clé comportant quatre éléments (les éléments supérieur gauche, supérieur droit, inférieur gauche et inférieur droit) à une clé à huit éléments (les quatre éléments précédents sont répétés à l'avant et à l'arrière). En effet, les tables stockent une nouvelle entrée pour chaque configuration différente, c'est-à-dire pour chaque configuration qui ne comprend pas les huit mêmes éléments fils. Dans la Figure 4-2, la table des feuilles comprend donc une nouvelle entrée pour chaque configuration de huit feuilles différentes des configurations déjà présentes (c'est-à-dire lorsque les huit feuilles ne possèdent pas les mêmes états). Ainsi la configuration stockée à l'adresse '@a' qui correspond à la première entrée diffère de la troisième entrée (adresse '@c') par la valeur de deux cellules. Il est donc nécessaire que les deux entrées soient présentes. L'idée est semblable pour le stockage dans la table des branches à la différence que l'on ne retrouve plus des états comme clé mais l'adresse d'autres branches. Une configuration complètement

vide de 4^3 cellules correspond ainsi à une branche comprenant la même branche fille pour ses huit fils (‘@b’). Lors de l’apparition d’une branche différente, c'est-à-dire une branche ne possédant pas les mêmes fils, il est nécessaire de rajouter une nouvelle entrée. C’est le cas pour une configuration de 4^3 cellules possédant des branches différentes mais aussi pour une configuration de taille plus importante (8^3 cellules dans l’exemple).

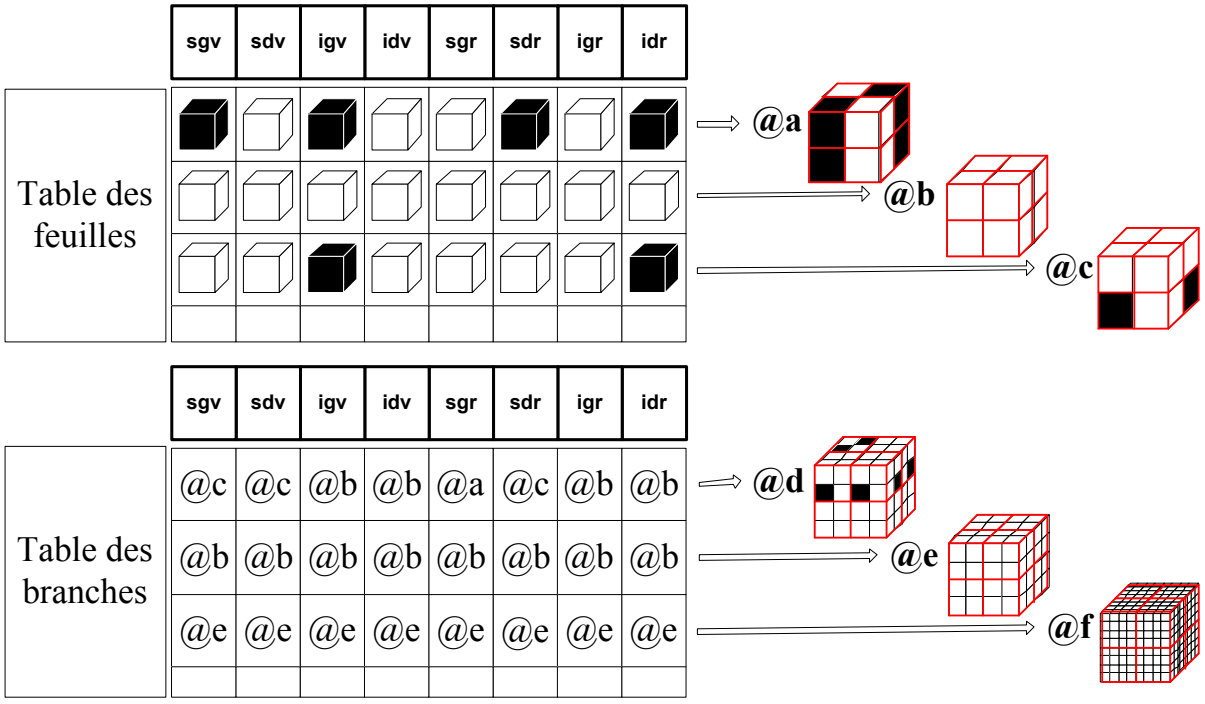


Figure 4-2 : Table de hachage des branches non terminales (en haut) et terminales (en bas). (sgv : supérieur gauche avant ; sdv : supérieur droit avant ; igv : inférieur gauche avant ; idv inférieur droit avant ; sgr : supérieur gauche arrière ; sdr : supérieur droit arrière ; igr : inférieur gauche arrière ; idr inférieur droit arrière)

Cette figure permet de commencer à entrevoir le principal problème de l’algorithme Hash-Life en 3D : le grand nombre de combinaisons possibles. Le nombre de configurations de cellules différentes pour une structure cubique de deux cellules de côté (2x2x2), lorsque chaque cellule peut posséder deux états différents, passe ainsi à 2^8=64 (contre 2^4=16 pour les configurations de cellules de 2x2). Mais nous reviendrons plus en détails sur ce point dans la partie 4.1.1.1 - Intérêts et limites en 3D.

Tout comme pour l’algorithme en deux dimensions, une branche est donc composée de huit nœuds (pouvant être des feuilles ou des branches selon qu’il s’agisse, respectivement, d’une branche terminale ou non).

En stockant également l'adresse de la configuration de cellules résultant d'une ou plusieurs itérations sur la branche courante à l'aide de la variable `resultat_`, il est alors possible, si l'on a déjà manipulé la configuration, de connaitre, sans calcul supplémentaire, le résultat d'une ou plusieurs itérations sur cette configuration (voir Code 4-1).

```
class Branche : public Noeud
{
   // Pointeur de Noeud correspondant aux huit fils.
   Noeud *   supGaucheAvant_;
   Noeud *   supDroitAvant_;
   Noeud *   infGaucheAvant_;
   Noeud *   infDroitAvant_;
   Noeud *   supGaucheArriere_;
   Noeud *   supDroitArriere_;
   Noeud *   infGaucheArriere_;
   Noeud *   infDroitArriere_;

   // Pointeur vers la configuration résultat.
   Noeud *  resultat_;
};
```

Code 4-1 : Extrait du code de la classe Branche de l'algorithme Hash-Life en 3D.

La partie la plus importante du portage de l'algorithme Hash-Life en trois dimensions est bien entendu liée au calcul de l'itération suivante lorsque celle-ci n'est pas connue. Alors que, dans le cas de l'algorithme initial en deux dimensions, il suffisait de manipuler quatre configurations de cellules temporaires pour déterminer le cœur d'une configuration de cellules à l'étape suivante (voir Figure 4-3 pour mémoire et la partie 2.2.2 - L'algorithme Hash-Life~~L'algorithme Hash-Life~~ pour plus de détails), il va maintenant être nécessaire de manipuler huit de ces configurations temporaires (on va retrouver les même décalages selon les axes haut/bas et gauche/droite auxquels va se rajouter l'axe avant/arrière).

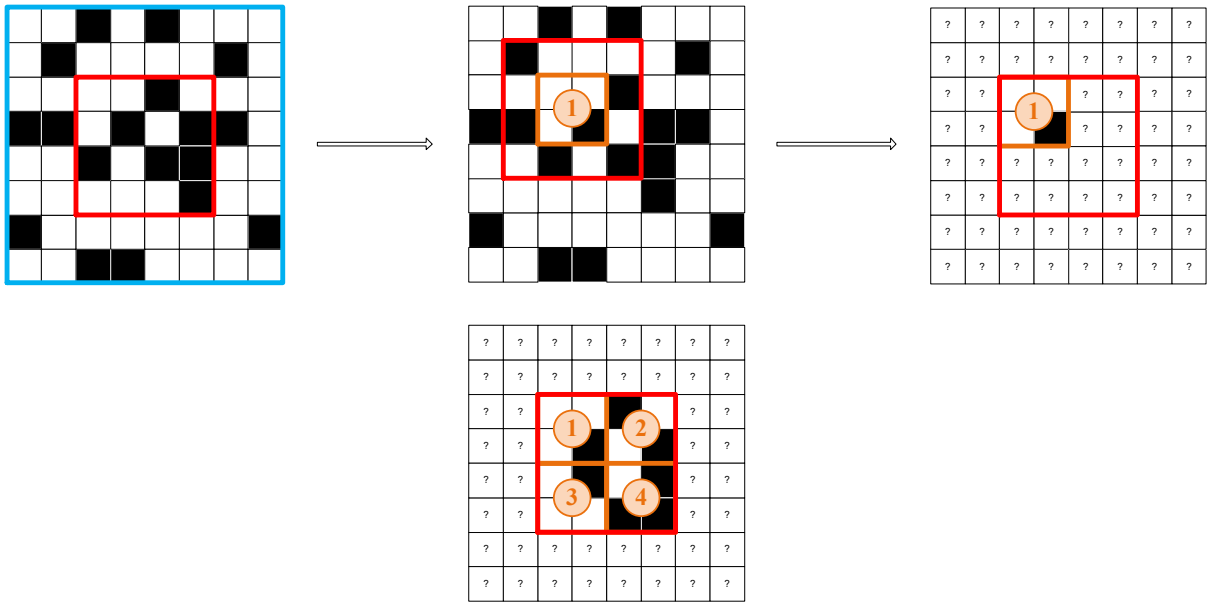


Figure 4-3 : Exemple de la création d'une configuration temporaire en 2D (première ligne) et résultat une fois les quatre configurations créées (deuxième ligne au centre).

Tout d'abord, il est important de rappeler que le travail en trois dimensions va impliquer de retrouver le cube au cœur du cube original. Ainsi, dans le schéma de la Figure 4-4, le cube que l'on souhaite voir évoluer est le cube blanc de 8^3 cellules (il a été découpé ici pour permettre de voir le cœur de celui-ci). Le cœur du cube blanc, en orange dans la figure suivante et composé de 4^3 cellules, est le cube correspondant au résultat d'une itération sur le cube blanc que l'on souhaite récupérer.

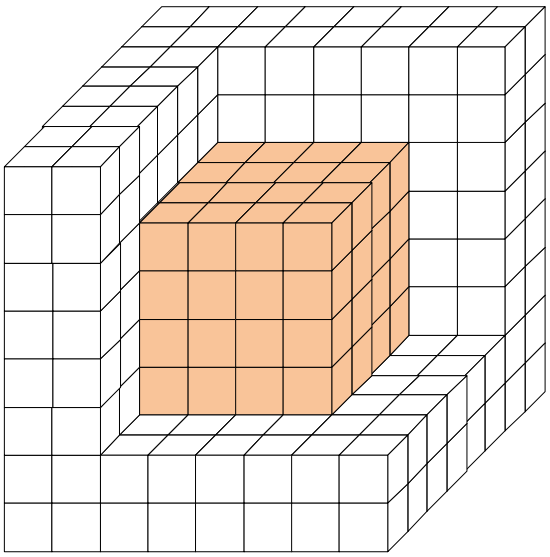


Figure 4-4 : Cube original de 8^3 cellules (en blanc) et cœur de 4^3 cellules qu'il va être nécessaire de calculer (en orange).

Pour cela, il est donc nécessaire de travailler avec huit configurations temporaires. Ces huit configurations correspondent à l'ensemble des combinaisons possibles de décalage du cœur du cube : vers le haut ou vers le bas, à droite ou à gauche, et enfin en avant et en arrière. Ainsi, les quatre premières configurations sont décalées d'un rang vers l'avant par rapport au cœur du cube (cubes n°1 à n°4 dans la Figure 4-5) et les quatre dernières sont décalées d'un rang vers l'arrière (cubes n°5 à n°8). De la même façon, les cubes n°1, 2, 5 et 6 sont décalés vers le haut lorsque les cubes n°3, 4, 7 et 8 sont décalés vers le bas. Les cubes n°1, 3, 5, 7 sont décalés vers la gauche et les cubes n°2, 4, 6 et 8 vers la droite.

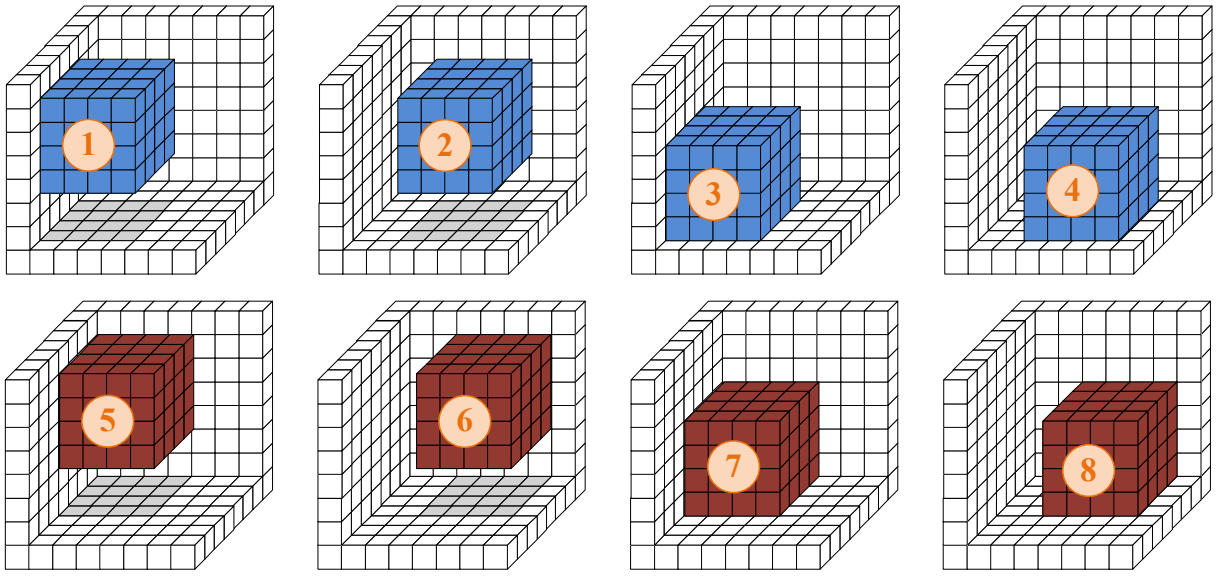


Figure 4-5 : Création des huit configurations temporaires de 4^3 cellules à partir des cellules du cube blanc de 8^3 cellules.

Tout comme c'était le cas avec l'algorithme en deux dimensions, à partir de ces huit configurations, il est alors possible de fusionner le résultat d'une ou plusieurs itérations sur chacune d'elles pour obtenir le nouvel état du cœur de la configuration de base (voir Figure 4-6 et Figure 4-7).

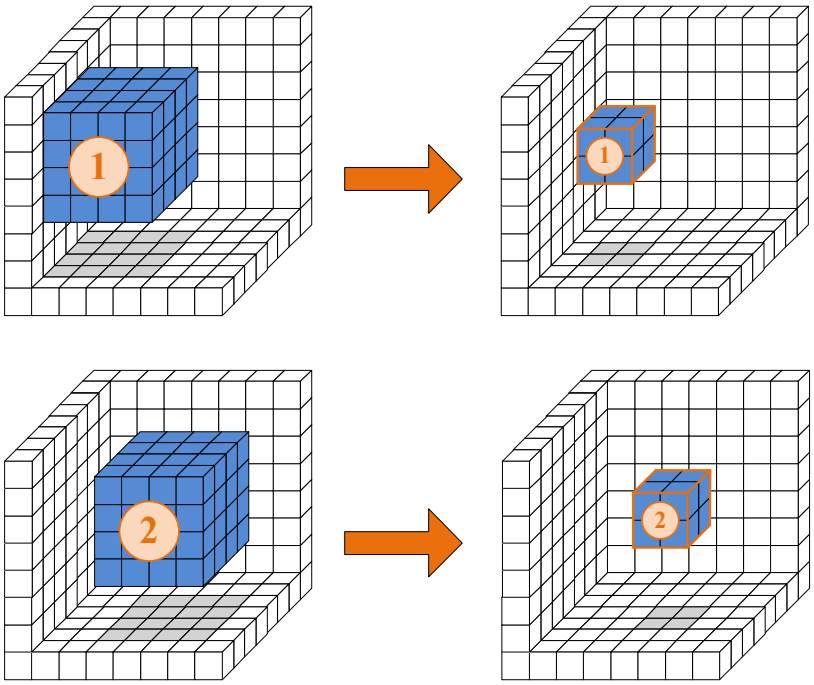


Figure 4-6 : Calcul d'une ou plusieurs itérations afin d'obtenir le cœur des sous-cubes.

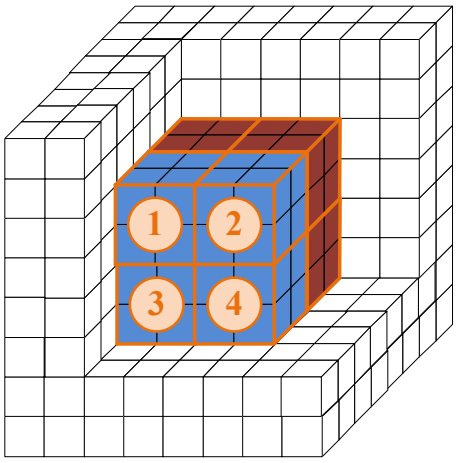


Figure 4-7 : Cube résultant de la concaténation du résultat d'une ou plusieurs itérations sur les huit configurations temporaires.

Intérêts et limites en 3D

Tout comme l'algorithme en deux dimensions, l'utilité et les limites de l'algorithme en trois dimensions vont fortement dépendre de la réapparition des configurations de cellules identiques. Si celles-ci sont nombreuses, l'algorithme est alors très performant. Mais si une majorité de nouvelles configurations apparaissent, le surcoût induit par la mémoïzation aura tendance à pénaliser l'ensemble de l'application et à amoindrir les performances générales.

La principale différence avec l'algorithme en deux dimensions est l'augmentation du nombre de configurations de cellules différentes pour un niveau donné de l'arbre. Ainsi pour un quadtree de niveau 2, on va considérer un domaine de 4x4 soit 16 cellules, mais lorsque l'on

considère un octree 4x4x4 nous avons 64 cellules. Même si nous avons le même nombre d'états possibles par cellule, il est bien évident que le nombre de combinaisons de cellules pour l'octree de niveau 2 va être bien supérieur à celui que nous obtenions avec des quadtrees de même niveau. L'impact est très significatif ; pour un quadtree à deux niveaux nous avons 65536 combinaisons, mais pour un octree à deux niveaux, nous avons 2^{64} combinaisons possibles, ce de qui donne un nombre supérieur à 10^{19}.

Le nombre de configurations possibles croît donc très rapidement en trois dimensions et cela va avoir un impact direct très significatif sur les performances. Tout comme nous l'avons déjà expliqué pour l'algorithme Hash-Life en deux dimensions, davantage de configurations équivalent, pour beaucoup de modèles d'automates (fonction des règles), à moins de réapparition de configurations et donc à une utilisation moins efficace, voir peu ou pas efficace, de la mémoïzation.

L'algorithme Hash-Life en trois dimensions possède bien évidemment les mêmes lacunes intrinsèques que l'algorithme de mémoïzation utilisé pour l'algorithme Hash-Life en deux dimensions : à savoir son application à des automates stochastiques ou à des milieux non-isotropes[13] n'est pas possible en l'état. En effet, une configuration de cellules sauvegarde l'état résultant. Or, dans le cas d'un automate stochastique ou de milieux non- isotropes, ce résultat peut varier selon les conditions (respectivement l'état du générateur de nombres pseudo-aléatoires ou la structure du milieu dans lequel on se trouve).

Une implémentation en trois dimensions à l'aide d'octrees n'est donc pas forcément une solution à tous les problèmes de performances liés aux automates cellulaires en trois dimensions, même si cela permet d'en résoudre une partie. Pour les cas où l'approche « Hash-Life en trois dimensions » échouerait, d'autres pistes ont été étudiées.

4.1.1.2 Performances de l'algorithme Hash-Life 3D sur les milieux excitables

Le modèle, utilisé pour tester les performances de l'algorithme Hash-Life en trois dimensions sur les milieux excitables, utilise un voisinage de von Neumann de degré un[14]. Ce modèle reprend les trois états classiques d'un automate cellulaire représentant un milieu excitable :

[13] Un milieu est dit isotrope lorsque ses propriétés sont constantes dans toutes les directions. Dans le cas d'un automate cellulaire, un milieu isotrope possède un voisinage semblable dans toutes les directions.
[14] Une cellule a pour voisine toutes les cellules disposant d'une face en contact. Pour un cube possédant six faces, une cellule a donc six voisines.

un état activé, un état réfractaire et un état de repos. Mais, pour permettre une modélisation plus fine, les transitions entre l'état activé et l'état réfractaire, et entre l'état réfractaire et l'état de repos ne sont pas réalisées en une seule itération mais en plusieurs (Gerhardt, Schuster et Tyson 1990). Dans le cas de la Figure 4-8, aussi bien dans le cas en deux dimensions que dans le cas en trois dimensions, les cellules sont actives pendant trois itérations et réfractaires pendant deux itérations.

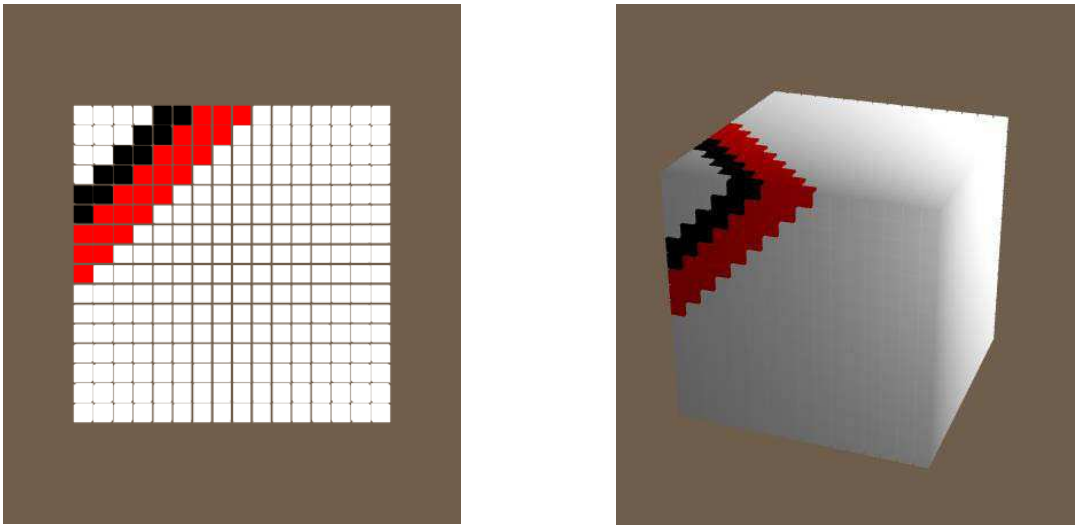


Figure 4-8 : Exemple de propagation dans un milieu excitable (automate à deux dimensions à gauche, automate à trois dimensions à droite). Les cellules blanches sont au repos, les cellules rouges sont actives (pendant trois itérations) et les cellules noires sont réfractaires (pendant deux itérations).

Dans les résultats présentés dans la suite, les cellules du modèle nécessitent cinq itérations pour évoluer de l'état activé à l'état réfractaire et trente itérations pour passer de l'état réfractaire à l'état de repos. Ces valeurs ont été utilisées pour représenter de façon simpliste la dépolarisation et la repolarisation du myocarde. Pour modéliser plus précisément le comportement du myocarde, un découpage plus fin, permettant de différencier les différentes périodes réfractaires (absolue, effective et relative), serait nécessaire (Bardou, et al. 1996) (Siregar, Sinteff, et al. 1998). Ici, le but est d'analyser les performances de l'algorithme Hash-Life lorsqu'un nombre important de sous-états entre en jeu. Cela peut paraitre anodin, mais pour permettre à l'algorithme Hash-Life, qui utilise la mémoïzation, de bien appréhender les différents sous-états de l'état activé et de l'état réfractaire (état activé depuis une itération, état activé depuis deux itérations…), il faut considérer chaque sous-état (un état avec un compteur) comme un état différent. Sans un tel mécanisme, l'algorithme ne pourra faire la différence lorsqu'il retrouvera une cellule dans un état activé depuis deux

itérations et une cellule dans un état activé depuis quatre itérations. Pour l'algorithme Hash-Life, cela conduit à ne pas manipuler trois mais trente-six états différents pour les cellules de l'automate. Or, étant donné que cet algorithme se base sur la découverte de configurations de cellules déjà existantes, cette augmentation du nombre d'état sera préjudiciable aux performances.

Pour en finir avec les conditions initiales des tests réalisés, il est important de noter que ceux-ci étaient effectués à partir d'un cube complètement au repos (toutes les cellules à l'état de repos) auquel il était apporté un stimulus à une seule cellule dans un coin du cube qui passait alors dans l'état activé. Une fois cette cellule activée, cette activation se propage à l'ensemble du cube jusqu'au côté opposé (le cube gauche de la Figure 4-9 représente l'état du cube cinq itérations après l'initialisation). La simulation prend alors fin lorsque la cellule opposée à la cellule activée au départ retrouve un état de repos (le cube droit de la Figure 4-9 représente l'état du cube lors de l'activation de la dernière cellule, c'est-à-dire cinq itérations avant la fin de la simulation).

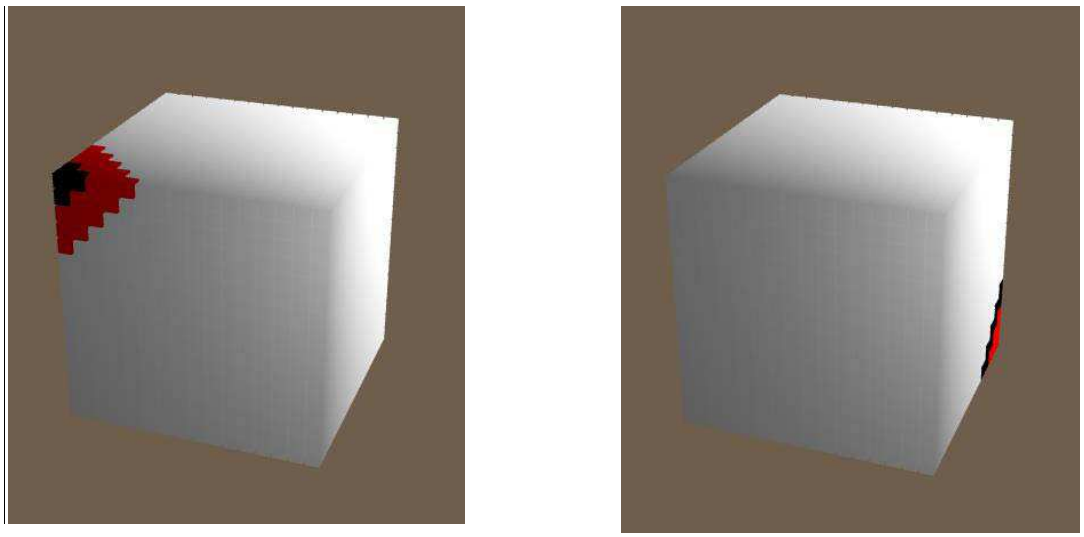


Figure 4-9 : Début et fin de la propagation dans un milieu excitable (cinq itérations après l'initialisation de la simulation à gauche). Activation de la dernière cellule à droite (cinq itérations avant la fin de la simulation).

Le nombre d'itérations nécessaires pour réaliser une telle simulation est fonction de la taille du côté du cube. L'utilisation d'un voisinage de von Neumann de degré un implique une propagation de proche en proche. La propagation complète d'une onde prend donc le temps nécessaire à l'onde pour aller activer la cellule dans l'angle opposé du cube : trois fois la taille du côté du cube auquel il faut rajouter le temps nécessaire pour cette cellule pour

passer à l'état réfractaire puis à l'état de repos (respectivement, cinq et trente itérations). Cela correspond donc à N x 3 + 35 itérations où N est la taille du côté du cube. Cette variation du nombre d'itérations réalisées au cours de la simulation en fonction de la taille du cube explique la raison pour laquelle le temps total d'exécution sera souvent remplacé par la mesure, plus pertinente, du temps moyen d'exécution d'une itération. C'est-à-dire, le temps total de simulation rapporté au nombre total d'itérations réalisées.

4.1.1.2.1 Performances, itération par itération

Lorsque les résultats intermédiaires sont nécessaires, l'algorithme est alors utilisé dans une approche itération par itération permettant de traiter l'automate cellulaire à la fin de chaque itération (ainsi, à chaque fois que l'on souhaite faire évoluer l'automate, celui-ci n'évolue que d'une itération). Dans ce cas, le temps moyen nécessaire pour réaliser une itération a tendance à diminuer avec l'augmentation du nombre de cellules dans le cube jusqu'à atteindre un palier (voir Figure 4-10). Ce résultat s'explique par le plus grand nombre d'itérations réalisées pour les cubes de plus grande taille (et donc avec davantage de cellules). En effet, le principe de la mémoïzation est de sauvegarder les résultats d'un calcul lorsqu'il est effectué pour la première fois. Or cette opération est coûteuse. Mais, plus les configurations de cellules seront retrouvées, plus le gain de l'utilisation de la mémoïzation sera grand. Dans le cas de milieux excitables, les configurations sont souvent retrouvées, le coût initial est donc petit à petit compensé par le fait que les configurations sont retrouvées. Lorsque les cubes traités sont plus grands, le nombre d'itérations nécessaires pour propager un signal sur l'ensemble du cube est plus grand et donc le coût initial de la mémoïzation est plus facilement amorti. C'est pour cette raison que le temps moyen d'une itération tend à diminuer jusqu'à atteindre un palier. Ce palier correspond au temps nécessaire à l'algorithme pour fonctionner lorsque les configurations sont connues ou que peu de nouvelles configurations apparaissent.

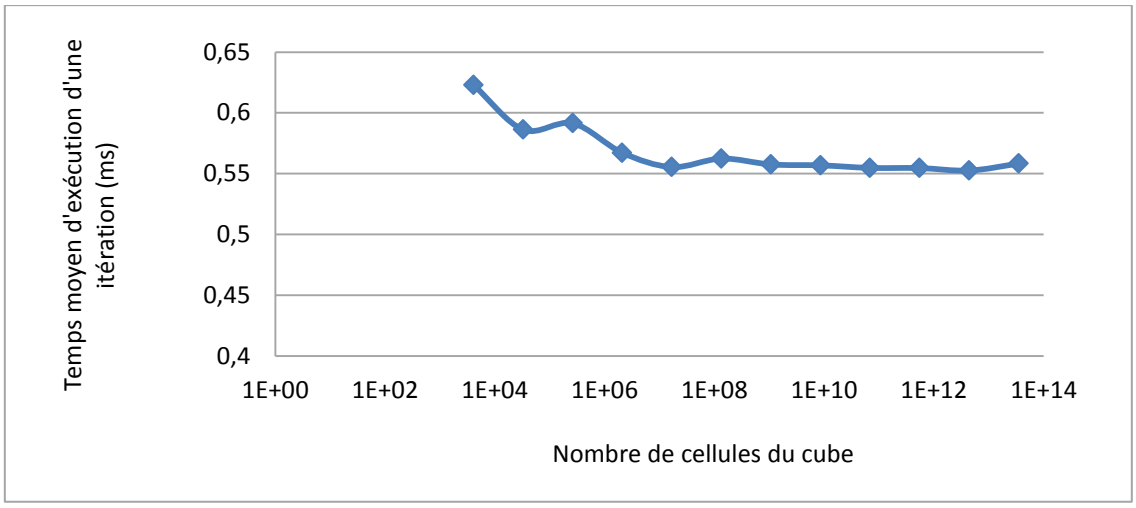


Figure 4-10 : Temps moyen d'une itération en fonction du nombre de cellules du cube.

Fort logiquement, le temps total d'exécution est donc linéairement proportionnel au nombre d'itérations nécessaires pour propager la vague d'un coin à l'autre du cube (voir Figure 4-11).

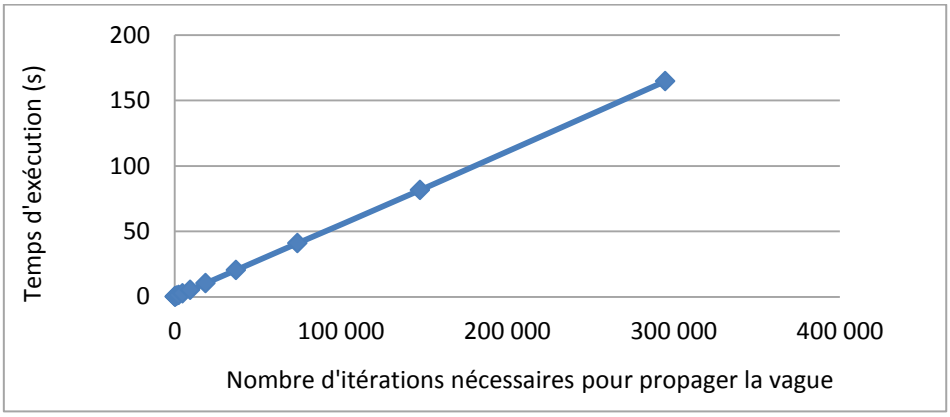


Figure 4-11 : Temps total d'exécution par rapport au nombre d'exécutions nécessaires pour propager la vague à l'ensemble du cube.

Au-delà du temps d'exécution, un élément intéressant, lors de l'étude d'un algorithme utilisant la mémoïzation, est son empreinte mémoire. En effet, il est nécessaire de stocker l'ensemble des configurations qui vont apparaître lors de la simulation. Plus le nombre de configurations découvertes est important, plus l'empreinte mémoire de l'application sera importante. La Figure 4-12, qui présente la mémoire utilisée en fonction de la taille, en nombre de cellules, du côté du cube, montre que celle-ci est proportionnelle au côté du cube et non pas au nombre de cellules présentes au total, qui correspond fort logiquement à la taille du côté du cube élevé au cube. Pour mémoire, un cube possédant 10 000 cellules sur

un côté possède ainsi mille milliards de cellules au total (10^4 au cube = 10^{12}). Ceci montre que, même si le nombre de configurations différentes découvertes est plus important pour les cubes de plus grandes tailles, un nombre limité de nouvelles configurations apparait. Ceci permet à l'empreinte mémoire de ne pas évoluer aussi vite que le nombre de cellules du cube.

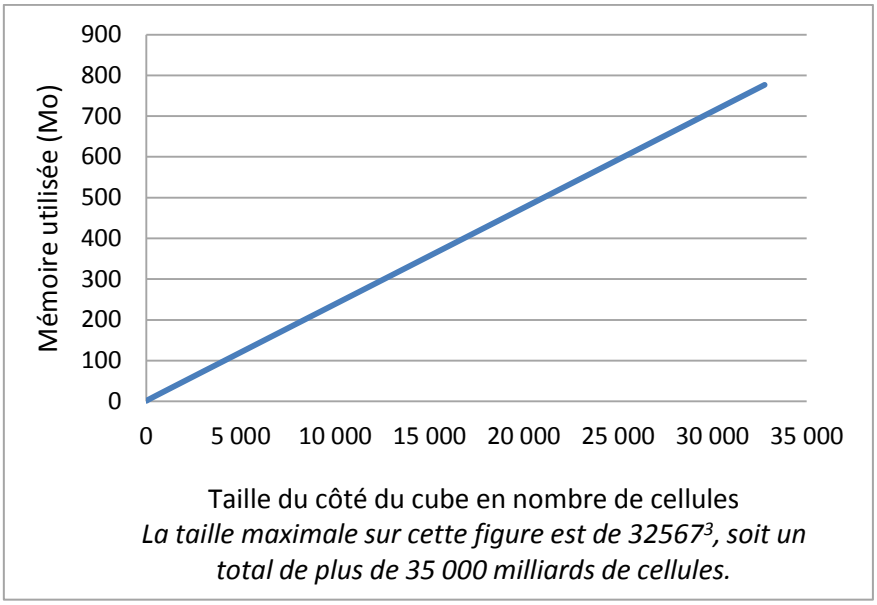


Figure 4-12 : Mémoire utilisée en fonction de la taille du côté du cube en nombre de cellules (nombre de cellules totales = côté^3).

Un cube de 32 768 cellules de côté, soit 35x10^{12} cellules au total (32 768^3), ne nécessite qu'un peu moins de 800Mo de mémoire vive. Ceci peut être beaucoup trop important pour un grand nombre d'applications, mais reste très accessible, y compris sur les micro-ordinateurs contemporains. De plus, cette quantité de mémoire nécessaire est très proche de doubler lorsque la taille du côté du cube double elle aussi. Il n'est donc pas possible de traiter des automates de très grande taille sans disposer d'une quantité de mémoire importante (les plus de 750Mo nécessaires précédemment ne sont pas acceptables pour toutes les simulations où l'empreinte mémoire est critique). Pour pallier à cette augmentation de mémoire importante, il serait possible de chercher à supprimer les configurations les moins retrouvées. Par exemple, il est possible, lorsque le nombre maximal de configurations que l'on souhaite stocker est dépassé, de mettre en place un algorithme pour supprimer les configurations qui n'ont pas été retrouvées depuis le plus longtemps. Cependant, ce processus pénaliserait les temps d'exécution et n'a pas été mis en place dans notre implémentation de l'algorithme Hash-Life.

Si les résultats de l'algorithme Hash-Life itération par itération sont intéressants, utilisé de cette façon, l'algorithme ne permet pas encore de disposer de performances exceptionnelles. En particulier, les cubes de grande taille nécessitent un temps d'exécution et une quantité de mémoire importante.

4.1.1.2.2 Performances en calculant plusieurs itérations d'un coup

Lorsque l'algorithme Hash-Life est utilisé en calculant à chaque pas de temps le résultat d'un maximum d'itérations d'un seul trait (c'est-à-dire qu'en faisant évoluer l'automate une seule fois, celui-ci se trouve dans un état qui correspondrait à l'état après plusieurs itérations conventionnelles), les performances de l'application évoluent.

La Figure 4-13 montre que le temps d'exécution nécessaire, pour que l'ensemble des cellules du cube passent à l'état actif puis reviennent à l'état de repos, n'évolue pas linéairement avec le côté du cube (graphe supérieur). Le graphe inférieur, qui possède une échelle logarithmique montre que l'évolution du temps d'exécution est linéaire au logarithme de la taille du côté du cube. La propagation d'une vague ne prend ainsi qu'environ 1,1s pour un cube comprenant 10^{27} cellules[15] (ceci n'est visible que sur le deuxième graphique utilisant une échelle logarithmique). En comparaison, avec l'approche itération par itération, 164s sont nécessaires pour réaliser la propagation dans un cube de 10^{13} cellules. Pour se donner un ordre d'idée, le temps d'exécution est 150 fois moins important pour un cube de 32 768 cellules de côté à l'aide de cette approche.

[15] La valeur pour une taille de côté de 10^9 cellules n'est pas présente au sein des graphes utilisant une échelle linéaire afin de conserver une bonne lisibilité des valeurs pour les plus petits cubes.

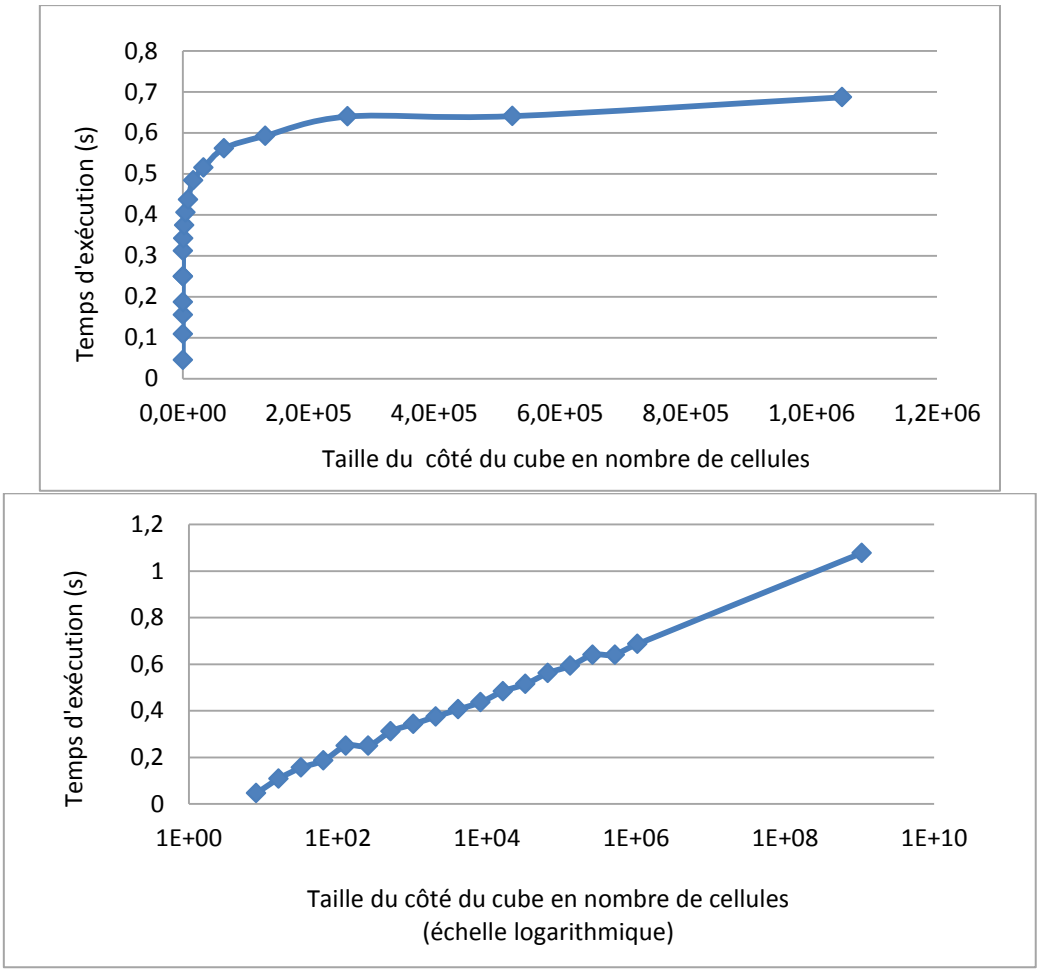


Figure 4-13 : Temps d'exécution de l'algorithme Hash-Life en réalisant plusieurs itérations à chaque pas (échelle linéaire pour le graphe supérieur et échelle logarithmique pour le graphe inférieur).

Si le temps d'exécution est moindre, il serait logique que la quantité de mémoire utilisée le soit également, étant donné que l'algorithme est pénalisé lors de la création de nouvelles instances en mémoire. La Figure 4-14 confirme ce point : la quantité de mémoire utilisée est en effet très faible ; environ 6Mo sont utilisés pour un cube de 10^9 cellules de côté lorsque plus de 750Mo était nécessaire pour un cube de 32 768 cellules de côté avec l'approche précédente. À taille de cube équivalente, l'empreinte mémoire est donc environ mille fois moins importante avec cette approche qu'avec la précédente. Ceci a pour conséquence que la quantité nécessaire de mémoire ne sera que très rarement un frein à la modélisation d'automates de grandes tailles.

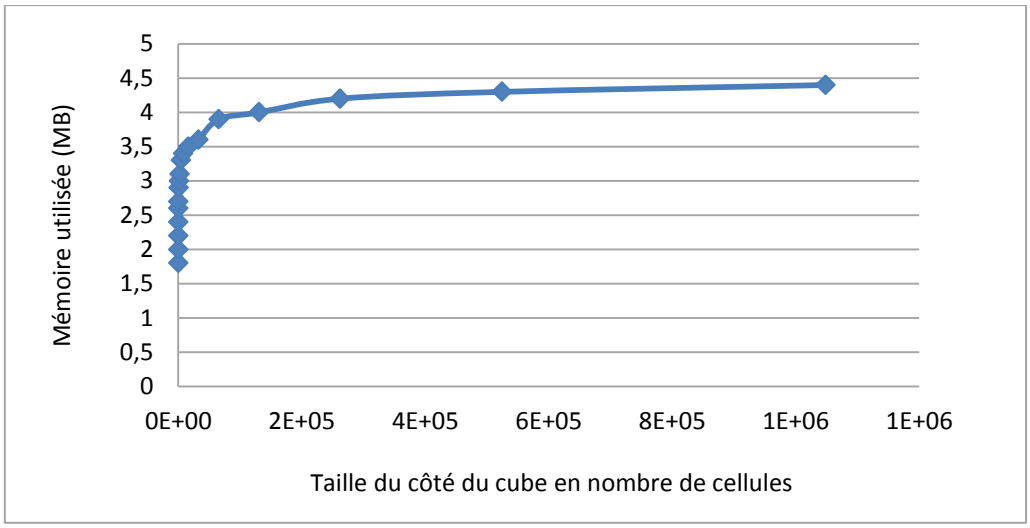


Figure 4-14 : Mémoire utilisée par Hash-Life en réalisant plusieurs itérations à chaque pas.

Mais ces excellents résultats ne doivent pas faire oublier l'importante limitation d'une telle approche. En effet, celle-ci n'est valide que s'il est acceptable de ne pas disposer de l'ensemble des étapes intermédiaires. Dans le cas contraire, il n'est pas possible de tirer parti de celle-ci et l'utilisation de l'approche itération par itération est obligatoire et entrainera une baisse significative des performances.

4.1.1.3 Performances de l'algorithme Hash-Life 3D à la lumière de l'activité

L'étude de l'algorithme Hash-Life à la lumière de l'activité permet de mieux cerner les caractéristiques de l'algorithme. Notre étude (Muzy, Varenne, et al. 2012) a porté sur deux simulations d'un même milieu excitable possédant les mêmes règles que les modèles précédents (cellules actives durant cinq itérations et réfractaires durant trente itérations). Mais à la différence des simulations précédentes où l'on étudiait l'évolution d'un signal à partir d'un unique stimulus depuis un coin d'un cube jusqu'à sa disparition dans le coin opposé, de nouveaux stimuli sont injectés dans le modèle à intervalles de temps aléatoire. Les deux simulations se différencient par l'intervalle moyen entre deux excitations et la manière dont sont choisies les cellules correspondant aux nouveaux stimuli. Dans le cas de la première simulation (noté n°1 dans les graphiques qui vont suivre), un stimulus a lieu n'importe où dans le cube en moyenne toutes les 10 itérations suivant une loi exponentielle négative[16]. Dans le cas de la seconde simulation (noté n°2), un stimulus a lieu plus

[16] La densité de probabilité d'une distribution exponentielle vaut $f(x) = \lambda e^{-\lambda x},\ \mathrm{x} > 0$ où la moyenne vaut $\frac{1}{\lambda}$.

régulièrement (toutes les 8 itérations en moyenne, toujours selon une loi exponentielle) mais toujours au niveau de la même cellule dans un coin du cube).

Si l'on s'intéresse à l'activité (Muzy et Hill 2011) au niveau cellulaire, l'activité correspond alors aux changements d'états d'une cellule. Dans le cas d'un milieu excitable, les changements d'état sont réguliers pour toutes les cellules qui ne sont pas au repos, l'activité peut donc être représentée par le nombre de cellules qui ne sont pas dans cet état de repos. La Figure 4-15, qui représente le pourcentage d'activité cellulaire (nombre de cellules au repos sur nombre de cellules totales) tout au long de la simulation montre que cette forme d'activité est beaucoup plus forte dans le cas de la deuxième simulation, où un plus grand nombre de stimuli est appliqué en un même endroit du cube.

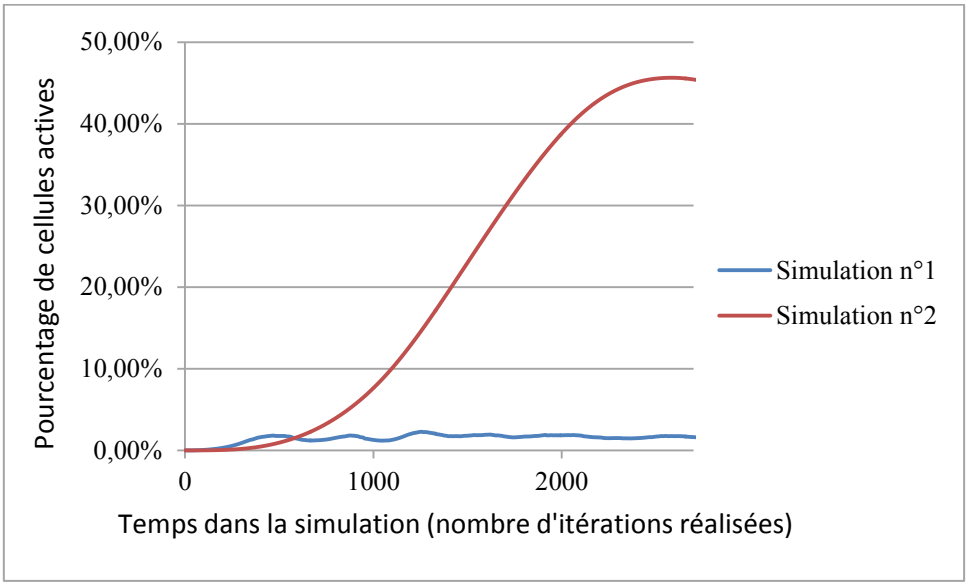


Figure 4-15 : Pourcentage de l'activité cellulaire sur l'ensemble du cube à chaque itération de la simulation.

Mais lorsque l'on étudie le temps d'exécution de chaque simulation (voir Figure 4-16), on s'aperçoit aisément qu'il n'existe pas de corrélation entre l'activité au niveau cellulaire et le temps d'exécution. En effet, la première simulation, pour laquelle l'activité cellulaire ne dépasse jamais 3% est de loin la plus gourmande en terme de temps de calcul.

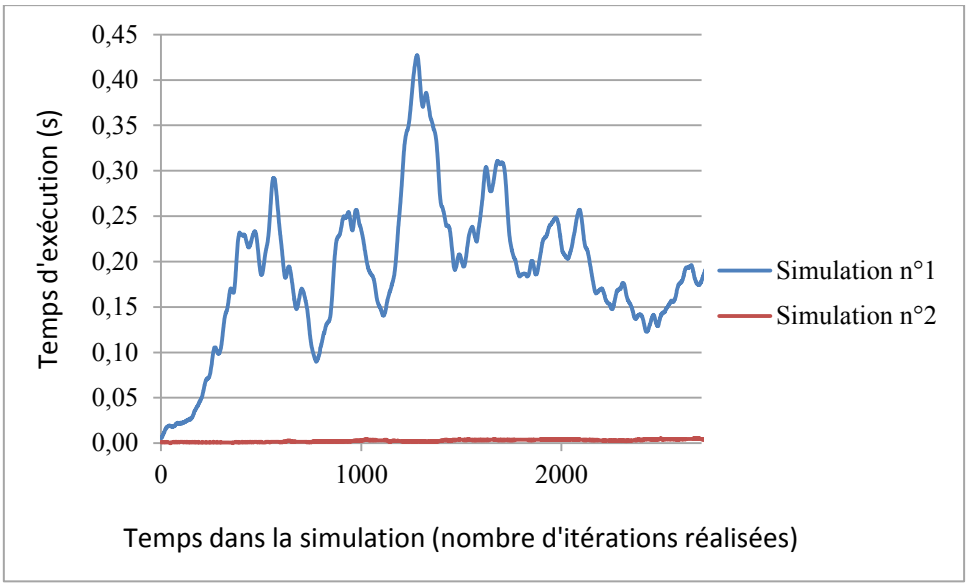


Figure 4-16 : Temps d'exécution de chaque itération.

Cela prouve bien que l'approche utilisant la mémoïzation de l'algorithme Hash-Life n'apporte que peu d'importance à l'état individuel de chaque cellule. L'activité d'un système serait donc davantage liée à l'apparition de nouvelles configurations de cellules. Ainsi, quel que soit l'état des cellules qui les composent, des configurations déjà connues de mêmes tailles seraient équivalentes. Dans l'exemple de la Figure 4-17, chaque configuration serait ainsi équivalente, qu'elle contienne uniquement des cellules actives, aucune ou n'importe quel nombre entre ces deux.

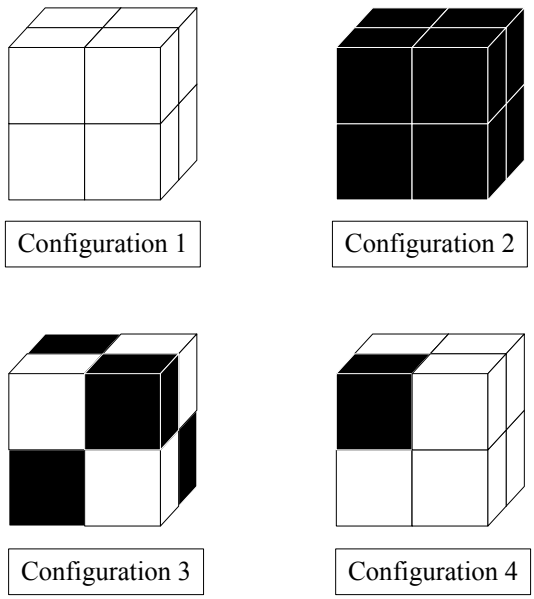


Figure 4-17 : Exemple de configurations équivalentes.

L'activité ne se mesure alors plus à travers le pourcentage de cellules actives mais en comptant le nombre de nouvelles configurations apparues à chaque itération. Le résultat, proposé en Figure 4-18, montre bien une correspondance très forte avec le temps d'exécution des simulations présentées en Figure 4-16 (la comparaison des deux courbes peut être vue en Figure 4-19). Ainsi, l'évolution du temps d'exécution suit bien l'évolution du nombre des nouvelles configurations de cellules trouvées. Dans le cas de la première simulation, l'apparition d'un plus grand nombre de nouvelles simulations induit un temps d'exécution plus long alors que l'inverse se produit dans la seconde simulation (moins de nouvelles configurations et un temps d'exécution plus faible).

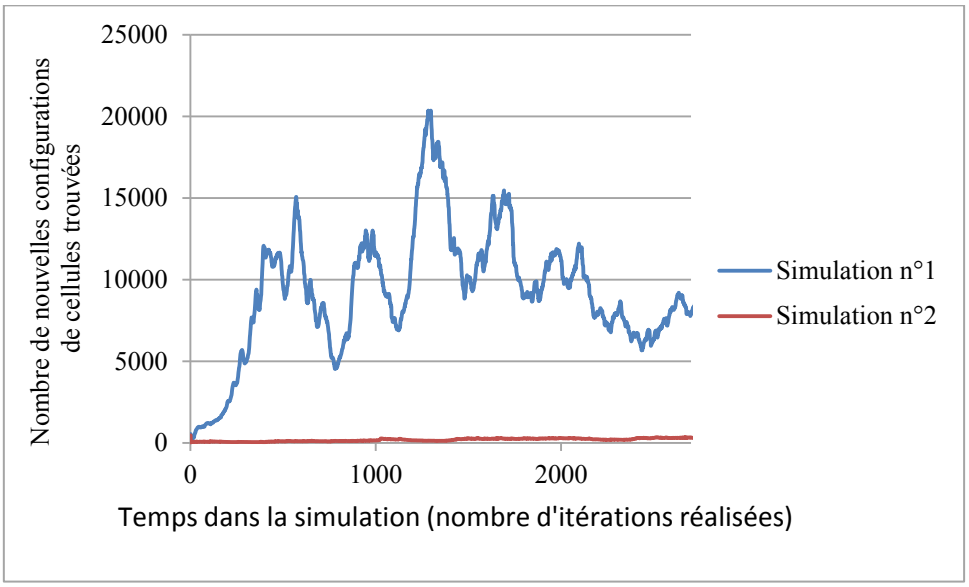


Figure 4-18 : Nombre de nouvelles configurations découvertes à chaque itération.

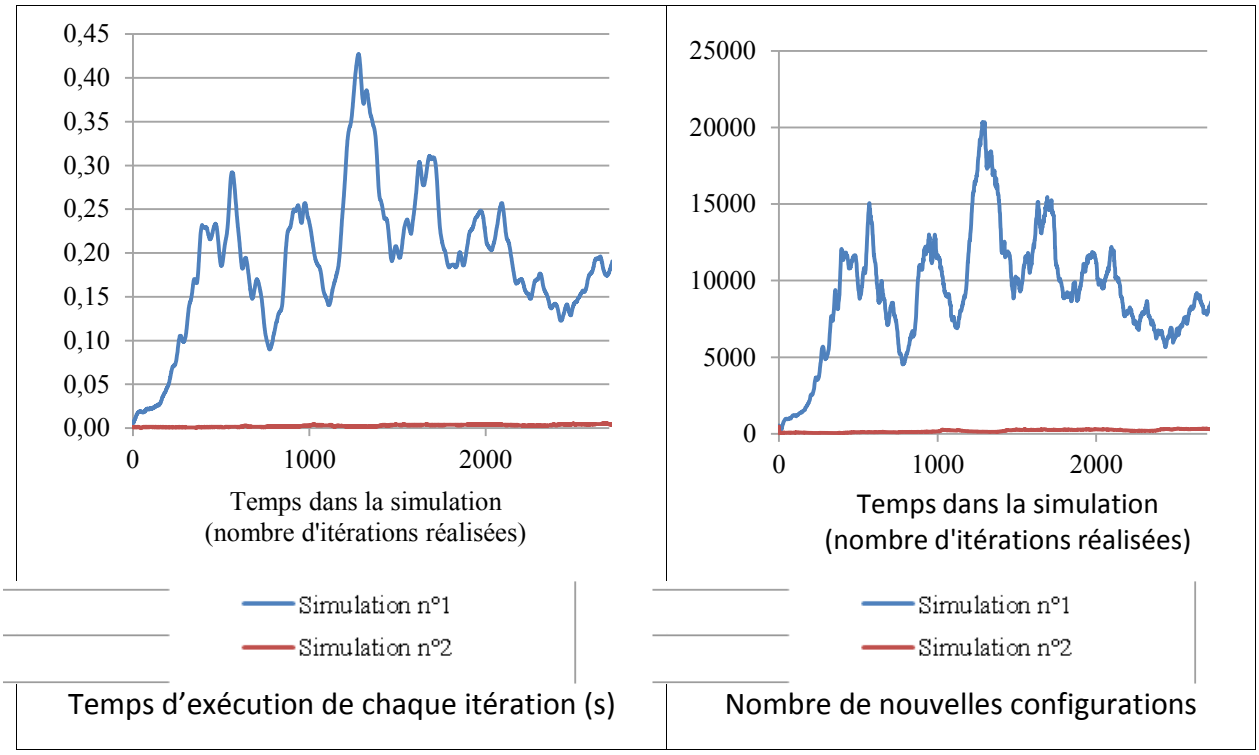


Figure 4-19 : Comparaison de l'évolution du temps d'exécution avec celle du nombre de nouvelles configurations.

Le taux d'activité, et donc le temps d'exécution, d'un tel algorithme ne dépend ainsi pas de l'état de chaque cellule prise individuellement mais de la fréquence d'apparition de nouvelles configurations.

4.1.2 Implémentation d'automates cellulaires sur GPU

Après avoir étudié l'applicabilité de l'algorithme Hash-Life en trois dimensions, nous avons cherché à voir s'il était possible d'accélérer le calcul de l'évolution d'automates cellulaires lorsque cet algorithme se montrait inadapté. Toujours dans le but de proposer une solution pouvant être adaptée à un ordinateur de bureau ou une station de travail, la parallélisation de ce processus a été réalisée sur GPU (Caux, Siregar et Hill 2011).

4.1.2.1 L'implémentation sur GPU

Idée directrice et implémentation

Comme cela a été présenté dans la partie précédente, l'utilisation de GPU permet de paralléliser de façon très importante les processus. Lorsque l'on parallélise sur quelques cœurs en utilisant le CPU d'un seul ordinateur, on dispose de plusieurs centaines d'unités de calcul pouvant travailler en parallèle sur un GPU. Pour tirer le maximum des performances

d’un GPU, il est donc nécessaire de découper les processus en un très grand nombre de sous-processus les plus indépendants les uns des autres.

Dans le cas des automates cellulaires, tant que l’on ne souhaite réaliser que le calcul d’une itération à la fois, le problème d’interdépendance est inexistant. En effet, le résultat à l’itération t+1 n’est dépendant que de l’état de l’automate à l’itération t. Le développement d’un algorithme simulant une itération à la fois permet donc de traiter indépendamment chaque cellule de l’automate.

Le choix a donc été fait d’utiliser un thread GPU pour chaque cellule de l’automate. Bien entendu, le nombre de threads pouvant être ordonnancés en même temps sur un GPU n’est pas illimité et dans le cas d’un espace de simulation trop important, les threads seront ordonnancés sur le GPU par groupes. Mais les GPU ont été conçus pour pouvoir changer de contexte très rapidement, cela n’a donc que très peu d’incidence sur l’efficacité global du programme. L’algorithme réalisé par chaque thread sur le GPU est donc des plus simples :

- Dans un premier temps, le thread calcule son identifiant unique afin de travailler sur la cellule qui lui est affectée.
- Dans un second temps, il réalise le calcul de la valeur de la cellule à l’itération suivante.

Pour le Jeu de la Vie en deux dimensions, un exemple d’implémentation pourrait être le Code 4-2 suivant (l’explication du code est détaillée plus bas).

```
__global__ void calculGenerationSuivante(
                                Cellule *     inMatrice,
                                Cellule *     outMatrice,
                                unsigned int  inNbCellParLigne,
                                unsigned int  inNbCellParSurface
                                      )
{
/* Calcul de l'indice du thread et donc de la cellule a traiter */
   unsigned int   indiceThread   = threadIdx.x +
                                   blockIdx.x * blockDim.x;

/* Calcul du nouvel état de la cellule en fonction des règles */


   // Calcul du nombre de cellules voisines vivantes
   // de la cellule ayant pour indice global indiceThread
   unsigned int   nbVoisinesVivantes = ...;

   if ( inMatrice[ indiceThreadX ] == CelluleVivante )
   {
      if ( nbVoisinesVivantes == 2 || nbVoisinesVivantes == 3 )
      {
         outMatrice[ indiceThreadX ] = CelluleVivante;
      }
      else
      {
         outMatrice[ indiceThreadX ] = CelluleMorte;
      }
   }
   else // inMatrice[ indiceThreadX ] == CelluleMorte
   {
      if ( nbVoisinesVivantes == 3 )
      {
         outMatrice[ indiceThreadX ] = CelluleVivante;
      }
      else
      {
         outMatrice[ indiceThreadX ] = CelluleMorte;
      }
   }
}
```

Code 4-2 : Implémentation simplifiée du kernel permettant de réaliser le calcul d'une itération sur un automate cellulaire.

La première étape est donc le calcul de l'identifiant global du thread à partir des variables proposées par l'interface de programmation CUDA : `threadIdx, blockIdx et blockDim`.

Une fois cet identifiant calculé, il ne reste qu'à suivre les règles de l'automate cellulaire. Ici, pour l'exemple du jeu de la vie, il faut tout d'abord calculer le nombre de cellules voisines qui sont vivantes. Puis, en fonction de l'état de la cellule courante et du nombre précédemment calculé, il est possible de trouver le nouvel état de la cellule.

Le goulot d'étranglement de cet algorithme se trouve dans la partie de code qui n'est pas présente dans le code précédent. En effet, pour calculer le nombre de cellules voisines

actives, il va être nécessaire d'accéder à l'état de chacune des cellules voisines. Dans le cas d'un automate à trois dimensions utilisant un voisinage de Moore de degré 1 (sont voisines toutes les cellules qui comportent au moins un coin en commun), on obtient donc un code de la forme du Code 4-3 :

```
unsigned int   nbVoisinesVivantes =
   // Cellules sur la même coupe en z et la même ligne en y
   inMatrice[ indiceThread - 1 ] +
   inMatrice[ indiceThread + 1 ] +

   // Cellules sur la même coupe en z et la ligne précédente
   inMatrice[ indiceThread - inNbCellParLigne - 1 ] +
   inMatrice[ indiceThread - inNbCellParLigne     ] +
   inMatrice[ indiceThread - inNbCellParLigne + 1 ] +

   // Cellules sur la même coupe en z et la ligne suivante
   inMatrice[ indiceThread + inNbCellParLigne - 1 ] +
   inMatrice[ indiceThread + inNbCellParLigne     ] +
   inMatrice[ indiceThread + inNbCellParLigne + 1 ] +

   // Cellules sur la coupe précédente
   [...] // 9 accès supplémentaires (3 pour chaque ligne)

   // Cellules sur la coupe suivante
   [...] // 9 accès supplémentaires (3 pour chaque ligne)
```

Code 4-3 : Extrait de l'implémentation du calcul de nombre de voisins vivants.

Il est ainsi nécessaire de réaliser 27 accès en mémoire (3x3x3) (voir Figure 4-20), l'état de la cellule courante étant également nécessaire plus loin dans l'algorithme. Or les accès à la mémoire d'un GPU, aussi bien en lecture qu'en écriture, peuvent être très couteux. C'est le cas en particulier si ces accès ne sont pas réalisés de façon contiguë en mémoire. En effet, des accès très éloignés en mémoire rendent l'utilisation des mémoires cache généralistes (introduites sur les dernières architectures de GPU) beaucoup moins efficaces.

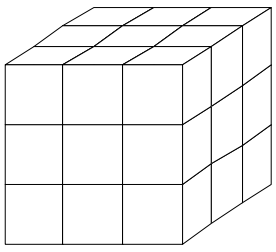


Figure 4-20 : Accès nécessaires pour une cellule (cette cellule se retrouve au centre du cube).

Pour éviter de multiplier les accès à la RAM du GPU, il est possible de les mutualiser. En effet, il est aisé de se rendre compte que deux cellules côte à côte vont partager 18 de leurs 27 accès (voir Figure 4-21). Si ces accès sont mutualisés, le nombre d'accès total passe de 54

à 36, soit 33% moins d'accès en RAM du GPU. Si ce concept est reproduit pour un cube de plus grande taille, les avantages sont encore plus clairs. Ainsi, les cellules d'un cube de 6x6x6 (216) cellules n'auront à connaitre l'état que de 8x8x8 (512) cellules (le cube englobant) pour pouvoir calculer la nouvelle valeur de chacune des cellules. On passe alors de 5832 à 512 accès.

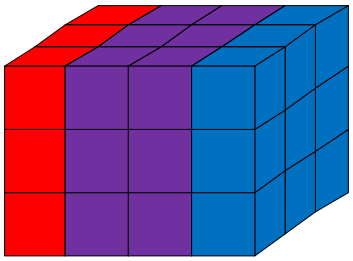


Figure 4-21 : Accès nécessaires pour deux cellules côte à côte (rouge : accès propres à une cellule, bleu : accès propres à l'autre cellule, violet : accès en commun).

La solution pour mutualiser les accès mémoire est simple : il s'agit de ne charger l'état de chaque cellule qu'une seule fois en mémoire partagée. De façon à diminuer ce temps de chargement depuis la RAM du GPU vers la mémoire partagée, celui-ci est réparti entre tous les threads travaillant sur des données communes (D. Kirk 2007). Une fois la mémoire partagée chargée, les threads peuvent alors accéder à l'état des cellules non plus directement dans la RAM du GPU mais en mémoire partagée.

Le code résultant (voir Code 4-4) est alors très semblable au code précédent. Il faut donc tout d'abord charger la mémoire partagée puis ne pas oublier de synchroniser les threads avant d'utiliser cette mémoire. Étant donné que chaque thread charge une partie de la mémoire (dans le code suivant, il charge l'état d'une seule cellule), il faut bien attendre que tous les threads aient fini le chargement de la mémoire partagée pour l'utiliser. Il suffit ensuite d'accéder à l'état des cellules depuis cette mémoire partagée plutôt que depuis la mémoire globale pour en tirer parti. L'écriture finale du nouvel état d'une cellule se faisant, quant à lui, toujours dans la mémoire globale de façon à pouvoir être connu par tous les threads lors du calcul de l'itération suivante et par le CPU si l'on souhaite récupérer les données.

```
__global__ void calculGenerationSuivante(
                                Cellule *     inMatrice,
                                Cellule *     outMatrice,
                                unsigned int  inNbCellParLigne,
                                unsigned int  inNbCellParSurface
                                       )
{
/* Calcul de lindice du thread et donc de la cellule a traiter */
   unsigned int   indiceThreadLocal    = threadIdx.x;
   unsigned int   indiceThreadGlobal   = threadIdx.x +
                                         blockIdx.x * blockDim.x;

/* Chargement de la mémoire partagée */
   memoirePartagee[ indiceThreadLocal ] =
                                inMatrice[ indiceThreadGlobal ];

/* Synchronisation des threads */
   __syncthread();

/* Calcul du nouvel état de la cellule en fonction des règles */

   // Calcul du nombre de cellules voisines vivantes
   // en utilisant la mémoire partagée
   unsigned int   nbVoisinesVivantes =
            memoirePartagee[ indiceThreadLocal - 1 ] +
            memoirePartagee[ indiceThreadLocal + 1 ] +
            [...]

   if ( memoirePartagee[ indiceThreadLocal ] == CelluleVivante )
   {
      if ( nbVoisinesVivantes == 2 || nbVoisinesVivantes == 3 )
      {
         outMatrice[ indiceThreadGlobal ] = CelluleVivante;
      }
      else
      {
         outMatrice[ indiceThreadGlobal ] = CelluleMorte;
      }
   }
   else // inMatrice[ indiceThreadX ] == CelluleMorte
   {
      [...]
   }
}
```

Code 4-4 : Implémentation simplifiée du kernel utilisant la mémoire partagée pour réaliser le calcul d'une itération sur un automate cellulaire.

4.1.2.2 Simulation d'automate cellulaire 3D sur GP-GPU

Des tests ont été effectués sur plusieurs architectures. À l'époque où ont été réalisés ces travaux, l'architecture Fermi n'était pas encore apparue, un large travail a donc été réalisé sur une carte Tesla C1060. Plus tard, il a été possible d'établir des comparaisons avec la nouvelle architecture en utilisant une carte Fermi GTX590. Pour établir la comparaison avec un algorithme séquentiel, des processeurs Core 2 Xeon et Nehalem Xeon cadencé à 2,5GHz ont été utilisés.

Coût de la copie mémoire

Comme cela a été expliqué dans la partie précédente, l'algorithme sur GPU se décompose en trois grandes étapes : l'envoi de l'état de l'automate initial sur le GPU, le calcul de l'évolution des cellules pour le nombre d'itérations souhaitées et la récupération des données lorsque l'ensemble des itérations a été effectué. Si le calcul de l'évolution des cellules n'est pas suffisamment important en terme de temps de calcul, le coût des copies mémoires aller et retour peut impacter fortement sur le temps total d'exécution (Hovland 2008). La Figure 4-22 permet ainsi de se rendre compte que dans le cas d'un cube de 10^3 cellules, le temps d'exécution, rapporté au nombre d'itérations réalisées, est significativement plus important lorsque le nombre d'itérations est faible. Cela représente le poids des copies entre la mémoire du CPU et celle du GPU. Cette figure permet aussi de se rendre compte que dès 200 itérations, le poids des copies est très limité et qu'après 400 itérations, ce poids n'est plus significatif par rapport au temps d'exécution de calcul sur le GPU et un palier est alors atteint.

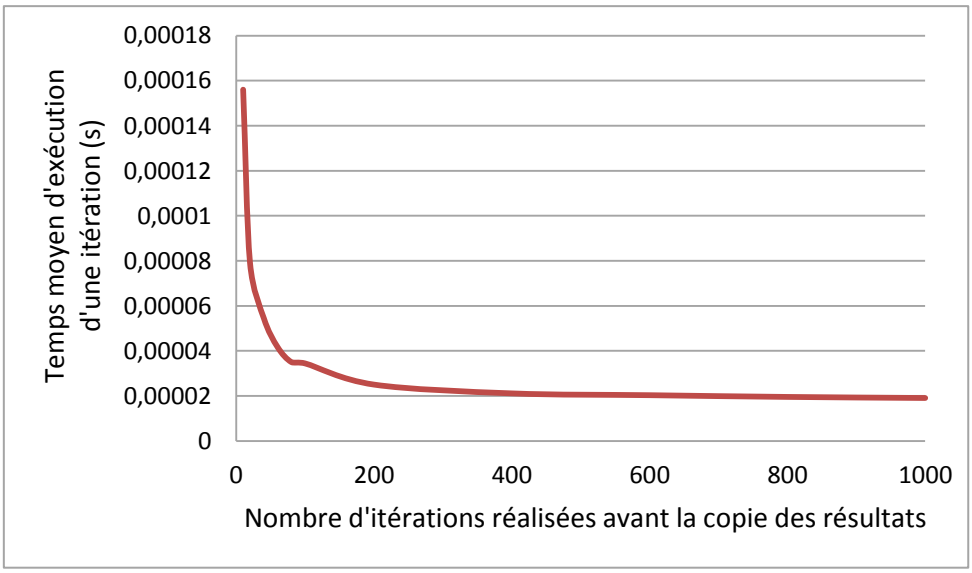


Figure 4-22 : Temps d'exécution moyen d'une itération pour un cube de 10^3 cellules.

Comparaison d'une implémentation basique sur GPU avec une implémentation sur CPU séquentielle

Les résultats de la Figure 4-22 n'impliquent cependant pas que le temps d'exécution sur GPU soit supérieur à celui sur CPU, même dans le cas d'un petit nombre d'itérations. La Figure 4-23 permet de comparer la simulation d'un automate de 20^3 cellules à l'aide, d'une part d'une implémentation basique (sans utilisation de la mémoire partagée) exécutée sur une

carte Tesla C1060, et d'autre part d'une implémentation séquentielle simple (n'utilisant qu'un seul cœur) exécutée sur un Core2 Q9300 à 2,5GHz. Les résultats montrent que si le temps moyen d'exécution d'une itération sur GPU est nettement plus important lorsque l'on réalise un petit nombre d'itérations totales, celui-ci est toujours nettement inférieur à l'équivalent séquentiel sur CPU. De plus, cette figure montre que le nombre minimum d'itérations nécessaires, pour que la copie des données ne soit plus significative, diminue avec l'augmentation du nombre de cellules : l'utilisation d'un cube de côté deux fois plus important permet d'atteindre un palier dès la réalisation de 200 itérations.

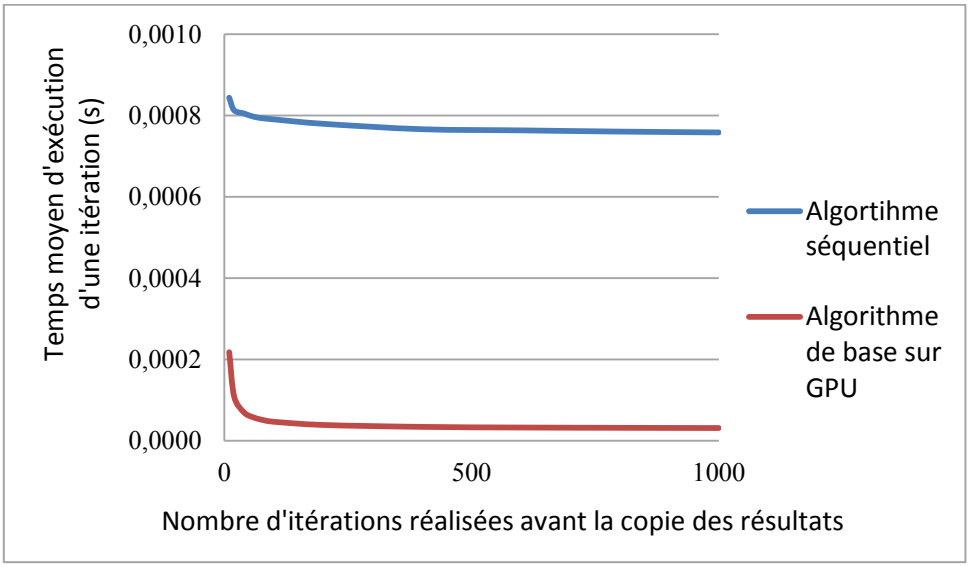


Figure 4-23 : Comparaison du temps d'exécution moyen d'une itération sur GPU et sur CPU pour un cube de 20^3 cellules.

Un autre aspect important est l'évolution des performances en fonction de la taille du cube étudié. La durée moyenne d'une itération pour une simulation de 1000 itérations en fonction de la taille du cube, lors d'exécutions sur le CPU et sur le GPU, est donnée par la Figure 4-24. Cette figure permet de montrer la différence importante de temps d'exécution entre le CPU et le GPU ainsi que la croissance plus rapide du temps d'exécution CPU.

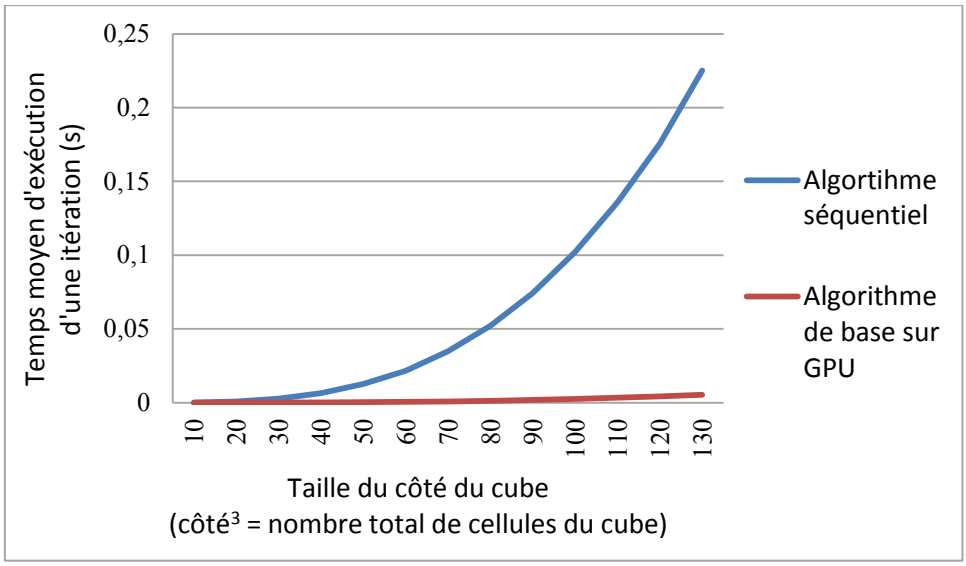


Figure 4-24 : Comparaison du temps moyen d'exécution d'une itération en fonction de la taille du cube sur GPU et sur CPU.

Pour autant, elle semble laisser croire que le temps d'exécution sur GPU n'augmente pas ou peu, ce qui est bien entendu faux. Si celui-ci n'augmente pas aussi vite, la Figure 4-25 montre que le temps d'exécution croît de plus en plus rapidement avec l'augmentation du côté du cube. La différence notable est l'échelle utilisée en abscisse. La courbe croît ainsi beaucoup plus rapidement dans le cas de l'application séquentielle. Sur GPU, il faut attendre que l'ensemble de la carte soit chargé pour voir cette forte croissance apparaître. Ainsi, il est possible de simuler sur GPU des cubes beaucoup plus grands que ce qu'il est possible de réaliser en séquentiel dans le même temps.

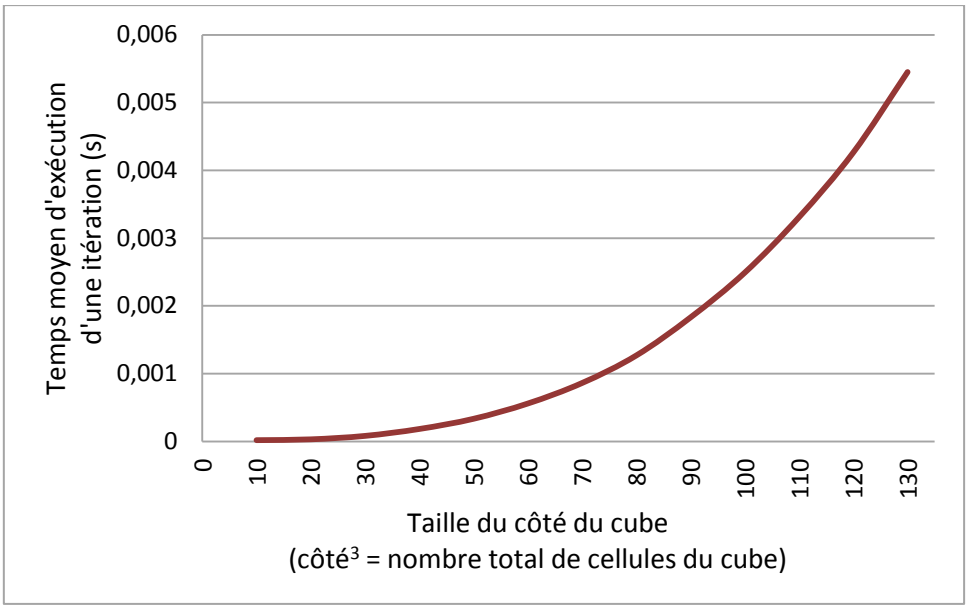


Figure 4-25 : Temps moyen d'exécution sur GPU d'une itération en fonction de la taille du cube.

Du fait de la forte différence d'échelle, il est donc difficile d'analyser les améliorations sur une figure comparant les deux temps d'exécution. Il est par contre beaucoup plus simple d'étudier le gain. La Figure 4-26 présente ainsi le gain de l'algorithme basique sur GPU par rapport à l'algorithme séquentiel sur CPU, et ceci pour différentes tailles de cubes. Cette figure fournit et confirme plusieurs informations. Tout d'abord, elle confirme le coût significatif de la copie mémoire lors de l'utilisation de GPU sur un petit nombre d'itérations. Pour une taille de cube donnée, le gain croît donc avec l'augmentation du nombre d'itérations réalisées pour atteindre un palier entre 200 et 400 itérations selon la taille de cube étudiée. L'enseignement nouveau est l'augmentation du gain avec la taille du cube étudié jusqu'à l'atteinte d'un palier. Le gain n'est ainsi que d'un facteur 5 entre les implémentations GPU et les implémentations CPU pour un cube de 10^3. Ce gain croît très rapidement pour atteindre 32x pour un cube de 30^3. Le palier correspondant à un gain de 40x est, quant à lui, atteint dès l'utilisation d'un cube de 80^3. L'apparition de ce palier est plus nette sur la Figure 4-27 qui représente l'évolution du gain entre implémentation GPU et implémentation CPU pour 1000 itérations réalisées en fonction de la taille du cube.

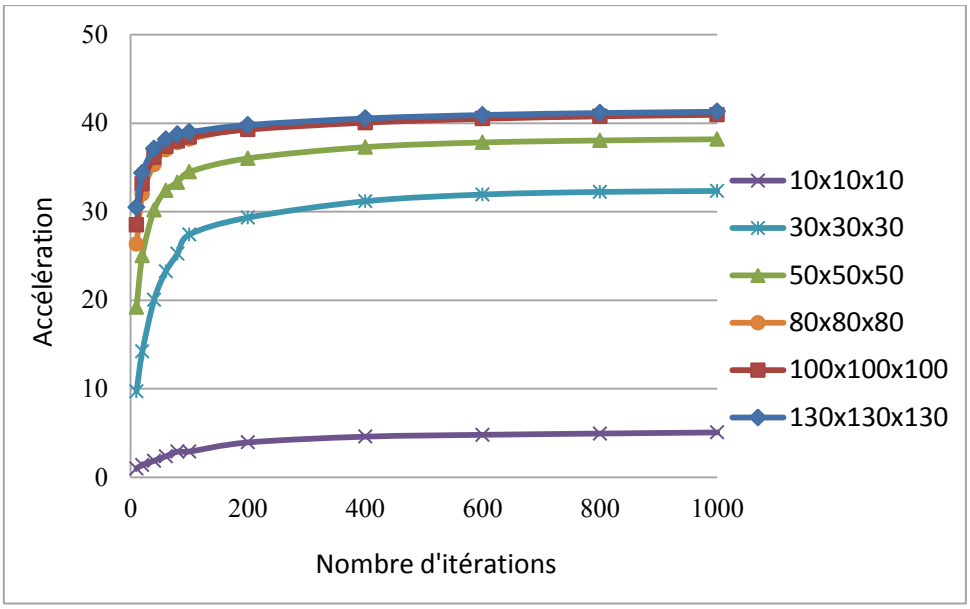


Figure 4-26 : Gain de performance, pour différentes tailles de cubes, de l'application basique sur GPU par rapport à l'application séquentielle sur CPU.

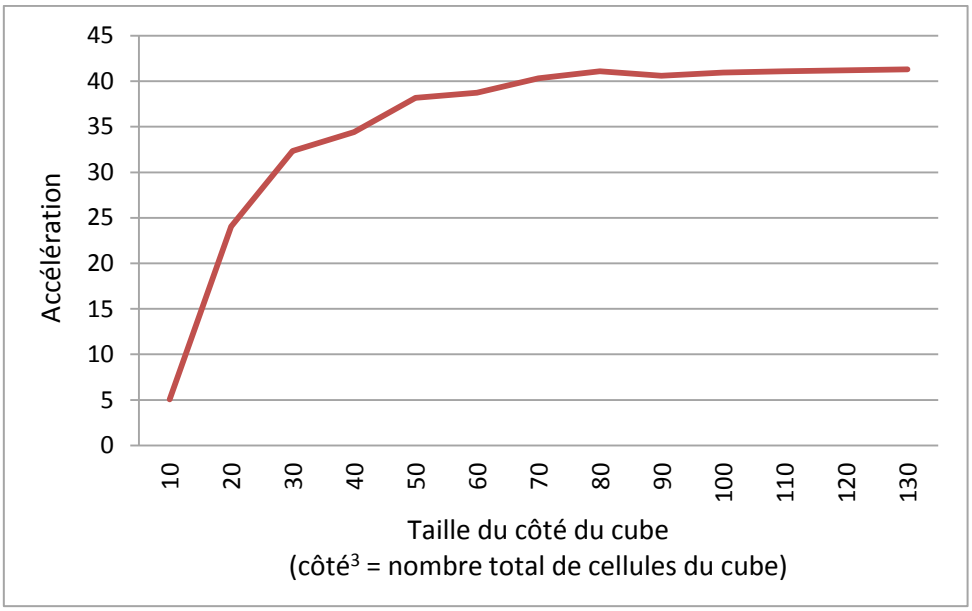


Figure 4-27 : Évolution du gain pour 1000 itérations en fonction de la taille du cube étudié.

Le gain moyen pour un cube de taille supérieure ou égale à 80^3, et dans le cas où le nombre d'itérations réalisées est supérieur à 200, est donc de l'ordre de 40. Ceci représente le gain entre une implémentation basique sur GPU et une implémentation séquentielle sur CPU. Nous avons proposé une autre implémentation sur GPU, qui exploite la mémoire partagée, qui est extrêmement performante sur les cartes GPU.

Comparaison de l'implémentation sur GPU avec utilisation avancée de la mémoire partagée

Exécutée sur une carte C1060, cette implémentation est beaucoup plus rapide. Ainsi, pour un cube de 100^3, les temps d'exécution de l'algorithme classique et de l'algorithme utilisant la mémoire partagée possèdent des tendances très proches (voir Figure 4-28). Comme cela a été précédemment présenté, la copie de la mémoire affecte le temps moyen lorsque le nombre d'itérations est faible avant que ce coût ne soit plus significatif (aux alentours de 200 itérations dans le cas d'un cube de cette taille). Mais si les deux courbes possèdent une tendance commune, le temps d'exécution de l'implémentation utilisant la mémoire partagée est nettement plus faible.

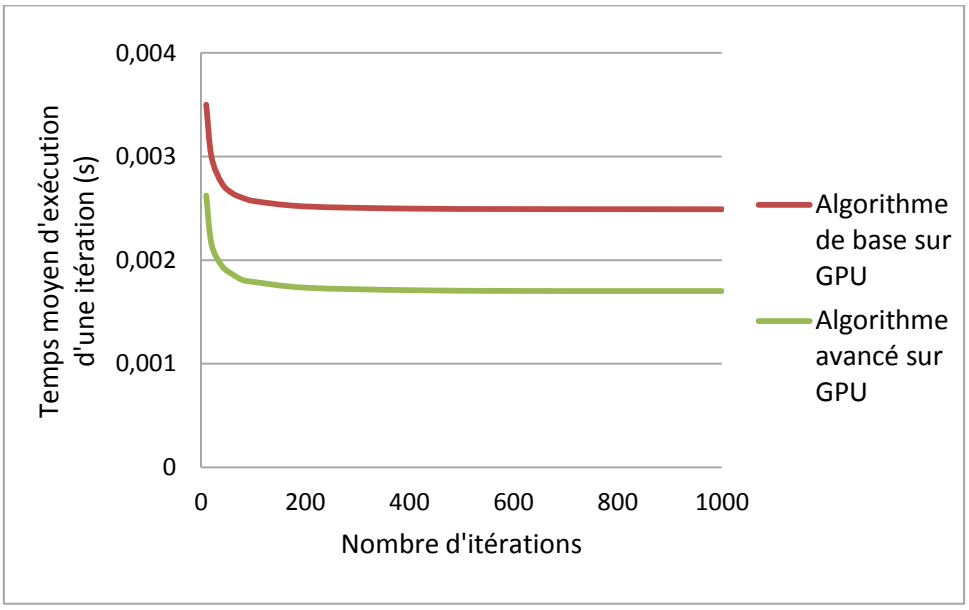


Figure 4-28 : Comparaison des temps d'exécution moyens d'une itération entre les deux algorithmes sur GPU pour un cube de 100^3.

Lorsque l'on compare la mesure du gain entre les deux implémentations (voir Figure 4-29), on s'aperçoit que pour des cubes de taille importante, le gain est significatif. Ainsi, lorsque les paliers, en termes de gain, sont atteints pour les deux implémentations, l'approche utilisant la mémoire partagée propose un gain (par rapport au CPU Core2) 50% plus important (en moyenne 60x contre 40x) que l'approche basique. Il est intéressant également de constater que le gain maximal est atteint pour des cubes plus grands dans le cas de l'algorithme utilisant la mémoire partagée.

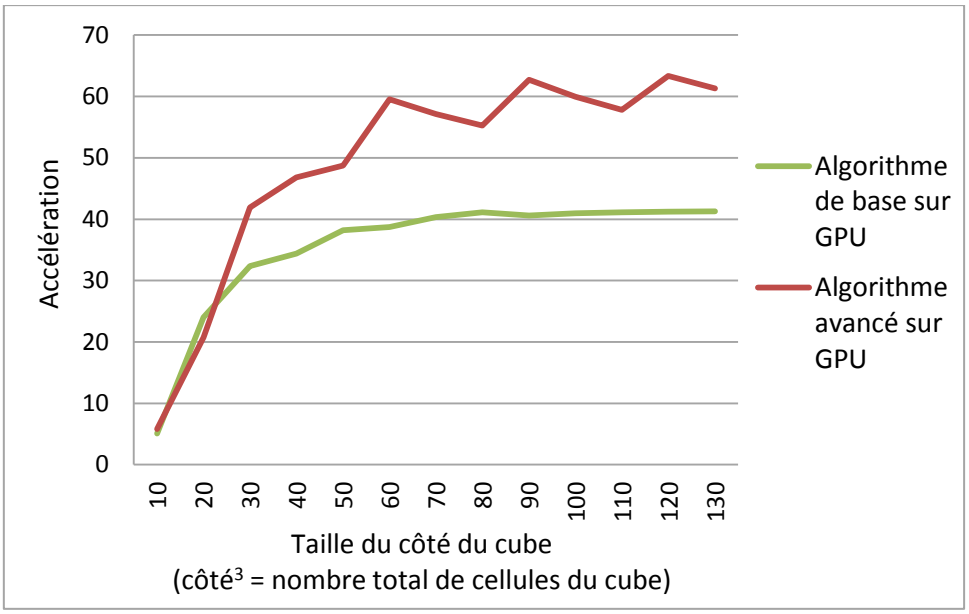


Figure 4-29 : Comparaison de l'évolution du gain pour 1000 itérations en fonction de la taille du cube.

Comparaison Core2/Nehalem et Tesla C1060/GTX590

Ces bonnes performances ne doivent pas occulter le fait que ces tests ont été réalisés sur du matériel appartenant à la génération de processeurs disponible au début de ma thèse. Aussi bien pour ce qui est du processeur que de la carte graphique, de nouveaux matériels beaucoup plus performants sont apparus.

Ainsi, le gain de l'implémentation sur GPU est moindre lorsque l'on ne compare plus à un processeur Core2 Xeon mais à un processeur de type Nehalem 5500 (voir Figure 4-30). Le gain moyen, qui est supérieur à 60x lors de l'utilisation d'un Core2 Xeon, diminue à 20x lorsque l'on utilise un processeur Nehalem de fréquence équivalente. Cette différence est majoritairement due à la présence de caches plus performants dans les nouveaux processeurs qui permettent des accès beaucoup plus rapides à la mémoire (Molka, et al. 2009).

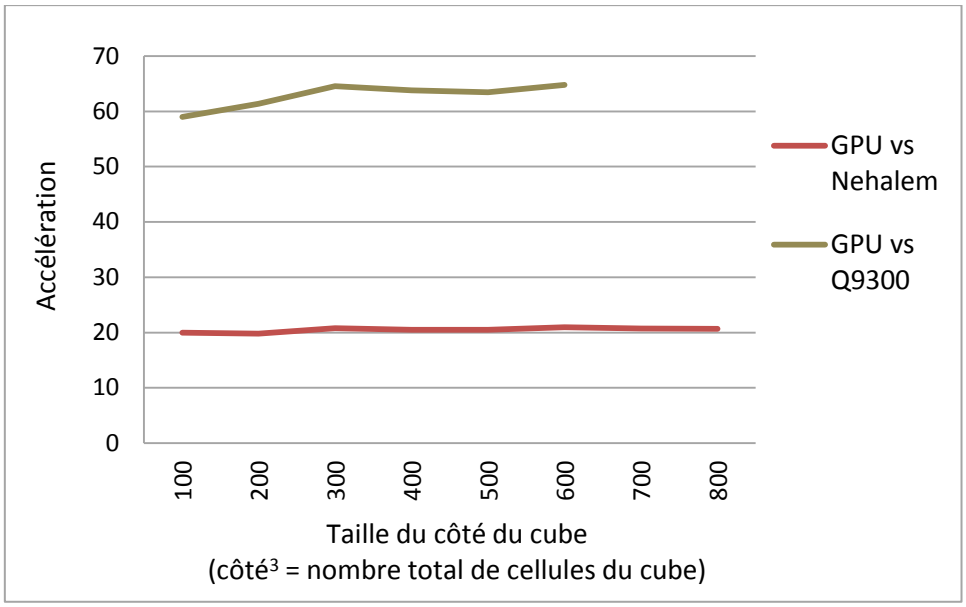


Figure 4-30 : Comparaison du gain de l'implémentation sur GPU en fonction du modèle de CPU utilisé.

Ce gain limité peut être expliqué par différents facteurs déjà abordés dans la partie précédente. Le fond du problème vient de l'implémentation sur le GPU. Afin de disposer de l'implémentation la plus simple et donc la plus réutilisable possible de manière à facilement pouvoir l'adapter à un nouveau problème, aucune optimisation spécifique à l'algorithme utilisé n'a été réalisée. Cela conduit à des accès mémoire beaucoup plus nombreux que nécessaire : l'état binaire (vie ou mort) est codé sur un entier (32 bits) là où pourraient être stockés 32 états, divisant dans le même temps le nombre d'accès par 32. De la même façon, un stockage plus dense permettrait de diminuer les temps de copie entre la mémoire du CPU et celle du GPU.

Il est également très intéressant de s'attarder sur les résultats d'une simulation sur un GPU d'architecture Fermi. Pour ce faire, des tests ont été réalisés à l'aide d'un ordinateur équipé d'un processeur Intel i7-2600K et d'un GPU NVIDIA GTX 590 où seuls 512 cœurs étaient dédiés à la simulation de l'automate cellulaire sur GPU. La Figure 4-31 montre les résultats réalisés pour des simulations de 1000 itérations selon la taille du côté du cube (le nombre total de cellules de la simulation correspond au côté du cube élevé au cube) pour les deux algorithmes implémentés. Les gains varient donc, à partir de plus de $100^3 (=10^6)$ cellules, entre 27x et 30x pour la parallélisation basique sur GPU et autour de 20x pour la parallélisation avancée sur GPU. Ce résultat est très intéressant car il signifie que l'ajout de caches généralistes dans l'architecture Fermi a permis d'améliorer les accès mémoire au

point de rendre certaines optimisations obsolètes dans certains cas. Dans notre cas, lorsque l'architecture Fermi est utilisée, la mutualisation des accès aux cellules est effectuée nativement par les caches du GPU. Le traitement supplémentaire requis pour gérer la mémoire partagée n'est alors plus qu'un handicap pour la version avancée, ce qui explique ses performances plus faibles. Il serait maintenant intéressant de voir si cette tendance se confirme avec la prochaine architecture des GPU qui devrait diminuer le rapport vitesse d'accès aux données sur puissance de calcul.

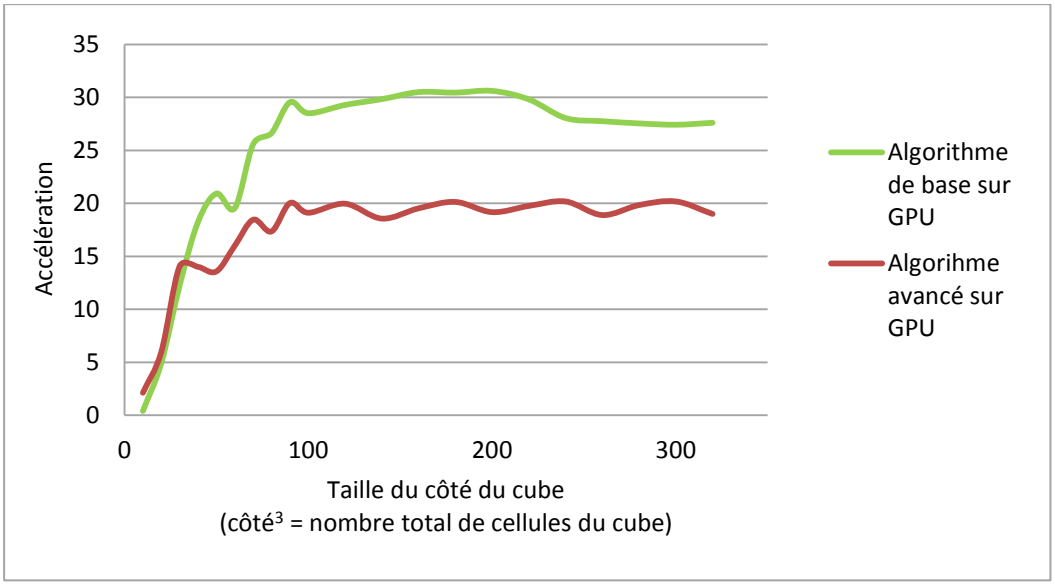


Figure 4-31 : Comparaison de l'évolution du gain pour 1000 itérations en fonction de la taille du cube avec une carte GTX 590.

4.1.2.3 Avantages et désavantages de l'implémentation sur GPU

Cet algorithme permet de tirer parti de la très grande puissance de calcul parallèle des GPU. Il est ainsi possible d'effectuer en parallèle plus de 500 calculs de nouvel état pour plus de 500 cellules différentes. Cette très grande capacité de calcul est donc un avantage indéniable des GPU. Bien évidemment, cette méthode n'a pas que des avantages. Pour être particulièrement efficace, le code généraliste qui a été développé et présenté ici devrait être personnalisé pour un jeu de règles et un environnement bien précis.

Le voisinage, tout d'abord, impacte fortement l'implémentation utilisée. Selon ce voisinage, les cellules utiles à l'évolution d'une cellule ne seront pas les mêmes. Ainsi, les montées en mémoire partagées seront plus importantes si le voisinage est important : un voisinage de von Neumann de degré 2 (cellules en orange et en vert dans la Figure 4-32) nécessitera de

mettre en mémoire partagée des cellules éloignées de deux cases de la cellule étudiée (seules celles possédant une arête complète en commun avec la cellule ou le voisinage de von Neumann de degré 1 sont considérées).

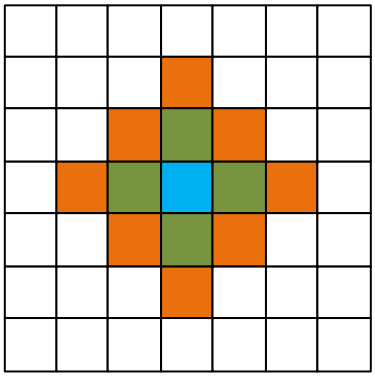


Figure 4-32 : Voisinage de von Neumann de degré 1 (en vert) et de degré 2 (en vert et en orange)

Par ailleurs, le voisinage à utiliser peut être très différent d'un voisinage conventionnel tel que le voisinage de von Neumann ou le voisinage de Moore. Dans le cas de la simulation des mécanismes électriques du cœur par exemple, le signal se propage plus facilement dans l'axe principal des fibres du myocarde que dans les directions orthogonales à celui-ci (Siregar, Sinteff, et al. 1998). Le voisinage utilisé se trouve alors être un voisinage plus long que large. Ceci pose un problème car à chaque voisinage différent, l'algorithme doit être modifié et les accès mémoire optimisés pour celui-ci.

De la même façon, selon les règles de l'automate, le nombre d'états possibles d'une cellule peut varier de façon importante et selon ce nombre d'états, des optimisations d'accès sont possibles. Ainsi, si, comme c'est le cas pour le jeu de la vie, une cellule ne peut posséder que deux états, il sera possible, à l'aide d'opérations binaires, de stocker huit états dans un seul entier et par conséquent de réduire par huit le nombre d'accès en mémoire globale. Encore une fois, si ces optimisations sont intéressantes, elles requièrent la personnalisation du code implémenté à un algorithme donné. Cette tâche peut être longue et fastidieuse, surtout si le modèle d'automate tend à évoluer régulièrement et qu'il devient nécessaire de modifier son implémentation régulièrement.

4.1.2.4 Discussion et perspectives

Deux solutions ont été implémentées pour améliorer les performances des simulations d'automates cellulaires. La première consiste à utiliser un nouvel algorithme tirant parti de

la mémoïzation : Hash-Life. Les performances de cet algorithme dépendent fortement de l'automate cellulaire manipulé. Si sa simulation implique la réapparition régulière des configurations déjà étudiées, cet algorithme peut devenir extrêmement intéressant avec des gains supérieurs à 99%. Ainsi, il est possible en quelques secondes de connaitre l'état après plusieurs milliers d'itérations d'un système de plusieurs milliards de cellules.

Mais ces excellentes performances ne sont effectives que lorsque le principe de mémoïzation est rentable. Lorsque l'évolution de l'automate n'est pas assez régulière pour que des configurations de cellules réapparaissent régulièrement, l'algorithme Hash-Life ne permet plus de disposer de temps d'exécution satisfaisants. La seconde implémentation propose donc une alternative à cet algorithme. Cette implémentation parallèle sur GPU permet de disposer de gain variant entre 20x et 60x par rapport à une implémentation séquentielle selon le processeur utilisé pour l'implémentation séquentielle.

Le but serait ainsi de choisir l'algorithme le plus efficace pour un problème donné : Hash-Life (qui est implémenté sur CPU) ou l'implémentation sur GPU. Mais l'intégration des optimisations liées aux automates cellulaires au sein de la PGMS ou d'une autre application de biologie intégrative n'est pas encore à l'ordre du jour du fait de contrainte temporelle. En effet, les automates cellulaires ne seront introduits dans la PGMS que lors d'une prochaine version et les autres applications de l'entreprise ont quelque peu été mises en retrait pour favoriser l'évolution plus rapide de la PGMS.

Les automates cellulaires n'ont pas été les seuls systèmes à avoir été parallélisés sur GPGPU. Une autre partie importante du travail effectué dans ce domaine a consisté à traiter les calculs de champs physicochimiques qui sont particulièrement gourmands en terme de temps de calcul dans la PGMS.

4.2 Calcul de force sur GPU

Comme cela a été introduit précédemment, la PGMS fait une utilisation importante du calcul de champs physicochimiques. Pour étudier ce problème, le choix a été fait tout d'abord de traiter le cas généraliste des champs de gravitation, électriques et électromagnétiques puis d'adapter les résultats aux cas présents au moment de mes travaux dans le logiciel PGMS.

4.2.1 Formule des champs

Avant d'aller plus en avant dans les algorithmes mis en place, voici un rappel sommaire des équations utilisées pour calculer ces trois champs.

- Le champ gravitationnel, présent à une position r_2, généré par une masse m_1 en position r_1 est donné par l'équation (1)~~(1)~~ :

$$\vec{g}(r_2) = -G\frac{m_1}{r_{12}^2}\hat{r}_{12} \qquad (1)$$

où :

- G est la constante de gravitation et vaut 6,67 x 10^{-11} $Nm^2.kg^{-2}$.
- r_{12} est la distance euclidienne entre les points r_1 et r_2.
- $\hat{r}_{12}$ est la norme du vecteur $\vec{r}_{12}$.

- Le champ électrique à une position r_2 et dû à une charge q_1 en position r_1 est donné par l'équation (2)~~(2)~~ :

$$\vec{E}(r_2) = \frac{q_1}{4\pi\varepsilon_0 r_{12}^2}\hat{r}_{12} \qquad (2)$$

où :

- ε_0 est la permittivité du vide et vaut 8.845 x 10^{-12} Fm^{-1}.
- r_{12} est la distance euclidienne entre les points r_1 et r_2.
- $\hat{r}_{12}$ est la norme du vecteur $\vec{r}_{12}$.

- Le champ magnétique présent en position r_2 dû à un magnéton de moment magnétique μ positionné en r_1 est donné par l'équation (3)~~(3)~~ :

$$\vec{B}(r_2) = \frac{\mu_0}{4\pi}\frac{3r_{12}(\mu.r_{12} - \mu.r_{12}^2)}{r_{12}^5} \qquad (3)$$

où :

- μ_0 est la perméabilité magnétique du vide et vaut 4 π x 10^{-7}.

Aussi bien pour le champ gravitationnel que pour le champ électrique, le champ généré en un point par plusieurs éléments est égal à la somme des champs générés. Pour le champ

gravitationnel par exemple, le champ généré en position r_j par les masses m_i en position r_i pour i allant de 1 à n est donné par l'équation (4)~~(4)~~ :

$$\vec{g}(r_2) = -G\sum_{i=1}^{n}\frac{m_i}{r_{ij}^2}\hat{r}_{ij} \qquad (4)$$

4.2.2 Algorithme séquentiel

L'algorithme séquentiel utilisé pour calculer les différents champs en tout point de l'espace est donc élémentaire : il parcourt l'espace discrétisé et pour chaque point de l'espace, il calcule la somme des champs générés par chacun des éléments (voir Code 4-5 et Code 4-6) :

```
void CalculChampGravitation(
                        std::vector< Particule * > & inParticules,
                        Espace * inEspace
                            )
{
   const double   G  = 0.0000000000667;

   for ( int x = 0; x < inEspace->Sx ; x++ )
   {
      for ( int y = 0; y < inEspace->Sy; y++ )
      {
         for ( int z = 0; z < inEspace->Sz; z++ )
         {
            CalculChampGravitationCellule(
                                inParticules,
                                inEspace->Treillis[ x ][ y ][ z ],
                                inEspace->ChampGravite[ x ][ y ][ z ]
                                          );
         }
      }
   }
}
```

Code 4-5 : Code séquentiel de calcul de la valeur du champ de gravitation en tout point de l'espace.

```
void CalculChampGravitationCellule(
                      std::vector< Particule * > & inParticules,
                      Cellule *                    inCellule,
                      VecteurF                     outChampResultat
                                )
{
   outChampResultat  = 0;

   for ( unsigned int i = 0; i < inParticules.size(); i++ )
   {
      Particule * particuleCourante = inParticules[ i ];
      float       m1                = particuleCourante->Masse;
      VectorF     r12               = inCellule->PositionGlobale -
                                    particuleCourante->PositionGlobale;

      if ( r12.Norme() != 0. )
      {
         outChampResultat +=
           -G * m1 / ( r12.Norme() * r12.Norme() ) * r12.Normalize();
      }
   }
}
```

Code 4-6 : Code séquentiel de calcul de la valeur du champ de gravitation en un point de l'espace.

La structure de l'algorithme de calcul du champ électrique est identique, seule l'équation change. Le champ magnétique est légèrement différent car il ne provient que d'une seule source, il n'est donc pas nécessaire de parcourir une liste de sources. Le parcours de l'ensemble des cellules discrétisées de l'espace est quant à lui logiquement bien présent dans tous les cas.

4.2.3 Utilisation des GPU

Comme on le voit très clairement dans le code ci-dessus, le calcul d'un champ pour une cellule est indépendant du calcul du champ des autres cellules. De plus, chaque cellule va exécuter exactement le même code, en travaillant sur des données différentes uniquement pour ce qui est de la position de la cellule traitée. De plus, pour chaque cellule, un nombre assez important de calculs est nécessaire : les cellules se trouvant dans un espace à trois dimensions, la moindre multiplication entre deux vecteurs nécessite plusieurs multiplications et plusieurs additions de nombres flottants. Toutes ces caractéristiques font que ces algorithmes sont parfaitement adaptés à l'utilisation de GPU pour paralléliser les calculs.

Pour ce qui est de l'implémentation sur GPU, la différence majeure va être la disparition des trois boucles « for » de l'algorithme séquentiel permettant de parcourir l'ensemble de l'espace. À leur place, le kernel `KernelCalculChampGravitation`, qui sera exécuté sur le

GPU, sera appelé autant de fois qu'il y a de cellules (voir Code 4-7). On obtient ainsi une parallélisation en autant de threads qu'il y a de cellules dans l'espace considéré. Excepté dans le cas d'un espace très réduit, cette parallélisation est largement suffisante pour utiliser tous les cœurs du GPU.

```
void CalculChampGravitation(
                        std::vector< Particule * > & inParticules,
                        Espace * inEspace
                            )
{
   dim3  dimGrille;
   dim3  dimBloc;

   Particule *  particules_gpu;
   VecteurF *   treilli_gpu;
   VecteurF *   champ_gpu;

   // Copie des donnees du cpu vers les tableaux gpu
   [...]

   calculDimensionGrilleEtBloc(
                        inEspace->Sx * inEspace->Sy * inEspace->Sz,
                        dimGrille,
                        dimBloc
                               );



   KernelCalculChampGravitation<<< dimGrille, dimBloc >>>(
                                                 particules_gpu,
                                                 inParticules.size()
                                                 treilli_gpu,
                                                 champ_gpu
                                                           );

   // Copie retour des donnees du gpu vers les tableaux cpu
   [...]
}
```

Code 4-7 : Extrait simplifié de l'appel du kernel de calcul du champ de gravitation sur le GPU.

Le kernel correspondant est par ailleurs très simple (voir Code 4-8). Il calcule tout d'abord l'indice du thread. Puis, en fonction de cet indice, il appelle le calcul du champ de gravité pour la bonne cellule. Ce calcul est réalisé de façon semblable à celui de la fonction `CalculChampGravitationCellule` vu précédemment (voir Code 4-6).

```
__global__ void KernelCalculChampGravitation(
                        Particule *  inParticules,
                        unsigned int inNbParticules,
                        VecteurF *   inTreilli,
                        VecteurF *   inChamp
                                                )
{
   unsigned int   threadIdGlobal    = calculerIdGlobal();

   CalculChampGravitationCellule_gpu(
                               inParticules,
                               inNbParticules,
                               inTreilli[ threadIdGlobal ],
                               inChamp[ threadIdGlobal ],
                                      );
}
```

Code 4-8 : Kernel permettant le calcul du champ de gravitation.

L'appel est donc très semblable à ce qui se fait en séquentiel. Le point le plus enclin à introduire des erreurs est la réalisation des copies de mémoire entre la mémoire du CPU et celle du GPU avant l'appel du kernel (pour les données d'entrées) et une fois celui-ci terminé (pour les résultats).

Le problème majeur de l'utilisation des GPU est, comme cela a été dit précédemment, la latence des accès mémoire. Or, pour calculer le champ en chaque point de l'espace, il est nécessaire à chaque fois de parcourir l'ensemble des particules pour ensuite sommer le résultat du champ produit par chaque particule en ce point de l'espace. Si le nombre de particules est grand, par exemple une centaine, le calcul du champ de gravitation, qui nécessitera toujours quatre fois plus d'accès à des entiers (un pour la masse et trois pour la position –voir Code 4-9), réalisera donc 400 accès. Ces accès vont alors avoir un impact très important sur le temps d'exécution du code.

```
void CalculChampGravitationCellule_gpu(
                      Particule *  inParticules,
                      unsigned int inNbParticules,
                      VecteurF &   inCellule,
                      VecteurF &   outChamp
                                    )
{
   outChamp  = 0;

   for ( unsigned int i = 0; i < inNbParticules; i++ )
   {
      Particule & particuleCourante = inParticules[ i ];
      // Un accès pour la masse
      float       m1                = particuleCourante.Masse;
      // Trois accès pour la position
      VectorF     r12               = inCellule.PositionGlobale -
                                    particuleCourante->PositionGlobale;

      if ( r12.Norme() != 0. )
      {
         outChamp +=
           -G * m1 / ( r12.Norme() * r12.Norme() ) * r12.Normalize();
      }
   }
}
```

Code 4-9 : Accès réalisés aux particules durant le calcul du champ de gravitation d'une cellule.

Pour éviter ces accès en mémoire globale, il est possible de mutualiser les accès aux données des particules. En effet, chaque cellule de l'espace a besoin des données de toutes les particules. Il est donc possible qu'elles partagent cette donnée à travers la mémoire partagée du GPU.

Pour cela, il est nécessaire que chaque cellule issue d'un même bloc (qui partage la même mémoire partagée) charge les données d'une particule (on partira ici du principe que le nombre de particules est inférieur au nombre de threads maximum d'un bloc pour simplifier) (voir Code 4-10). Une fois ces montées en mémoire réalisées, il est impératif de synchroniser tous les threads qui partagent la mémoire car sans cela, un thread pourrait commencer le calcul du champ pour sa cellule alors même que toutes les particules n'ont pas été chargées en mémoire partagée.

Une fois cela fait, il suffit alors d'appeler la fonction `CalculChampGravitationCellule_gpu` en prenant en paramètre, non plus le tableau de particules stocké en mémoire globale, mais celui stocké en mémoire partagée. Les quatre accès en mémoire ne sont plus alors réalisés en mémoire globale mais sont réalisés en mémoire partagée.

```
__global__ void KernelCalculChampGravitation(
                        Particule *  inParticules,
                        unsigned int inNbParticules,
                        VecteurF *   inTreilli,
                        VecteurF *   inChamp,
                                              )
{
   unsigned int   threadIdGlobal    = calculerIdGlobal();
   unsigned int   threadIdBloc      = calculerIdBloc();

   // Definition de la mémoire partagée
   __shared__ Particule    particulesPartagees[];

   // Chargement de la mémoire partagée
   if ( threadIdBloc < inNbParticules )
   {
      particulesPartagees[ threadIdBloc ] =
                               inParticules[ threadIdBloc ];
   }

   // Synchronisation des threads
   __syncthread();

   CalculChampGravitationCellule_gpu(
                              particulesPartagees,
                              inNbParticules,
                              inTreilli[ threadIdGlobal ],
                              inChamp[ threadIdGlobal ]
                                     );
}
```

Code 4-10 : Kernel permettant le calcul du champ de gravitation avec l'utilisation de mémoire partagée.

4.2.4 Étude du calcul de forces gravitationnelles, électriques et magnétiques

Afin de préparer les travaux à venir sur le calcul de champs physicochimiques de la PGMS, un travail important a été réalisé pour étudier l'opportunité de la parallélisation du calcul de champs physiques (gravitationnel, électrique et magnétique). En plus du calcul de ces trois champs, un modèle permettant de simuler le déplacement des particules en fonction de leur charge, de leur vitesse initiale et des champs auxquels elles sont soumises a été étudié. Dans tous ces cas, il est important de noter que seules des opérations en simple précision étaient nécessaires. Le GPU utilisé lors de ces tests, un GTX500 de NVIDIA qui possède 512 cœurs, est plus performant pour les opérations en simples précisions que pour celles en doubles précisions, il s'agit donc là d'un avantage important. Le CPU utilisé est quant à lui un processeur Intel Sandy Bridge i7-2600K cadencé à 3,4GHz.

4.2.4.1 Calcul du champ magnétique à partir d'une source

Dans notre étude de cas, le calcul du champ magnétique ne fait intervenir qu'une seule source à la différence des calculs des champs gravitationnels et électriques. Ceci a un effet

important sur les performances car les calculs effectués pour chaque cellule vont être plus limités. Or il est nécessaire de réaliser le plus de calcul possible pour maximiser l'efficacité des GPU (Kirk et Hwu 2010). Malgré cela, comme le montre la Figure 4-33, le temps d'exécution nécessaire au calcul du champ magnétique évolue plus rapidement sur CPU que sur GPU lorsque le nombre de cellules de l'espace augmente.

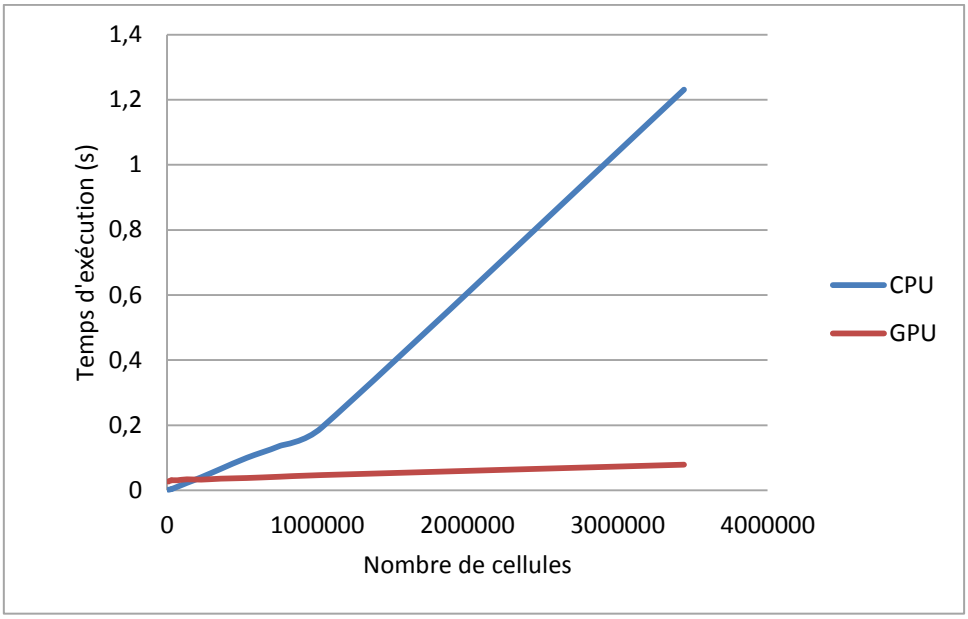


Figure 4-33 : Comparaison des temps d'exécutions du calcul du champ magnétique sur CPU et sur GPU

Pour autant, même si le temps d'exécution à l'aide du GPU semble évoluer très lentement, le gain entre les deux simulations n'est pas très important, même pour des espaces de grande taille. Ce gain n'atteint ainsi que 15,58x dans le cas de plus de 340 000 cellules. La Figure 4-34, qui montre l'accélération de l'exécution de l'implémentation sur GPU par rapport à celle n'utilisant que le CPU, permet tout de même de noter que l'accélération croît rapidement avec le nombre de cellules. Il est donc possible que ce gain continue de croître avec la taille de l'espace. Mais pour coller au plus proche de l'application cible et ainsi permettre une intégration plus aisée, un extrait de celle-ci a été utilisé pour étudier ces optimisations. Cela a pour conséquence de fortement impacter l'empreinte mémoire de l'application (plus de 1Ko de données sont utilisées pour instancier chaque cellule) et d'empêcher la réalisation de tests sur des espaces plus importants, alors même que la mémoire au sein du GPU est loin d'être saturée.

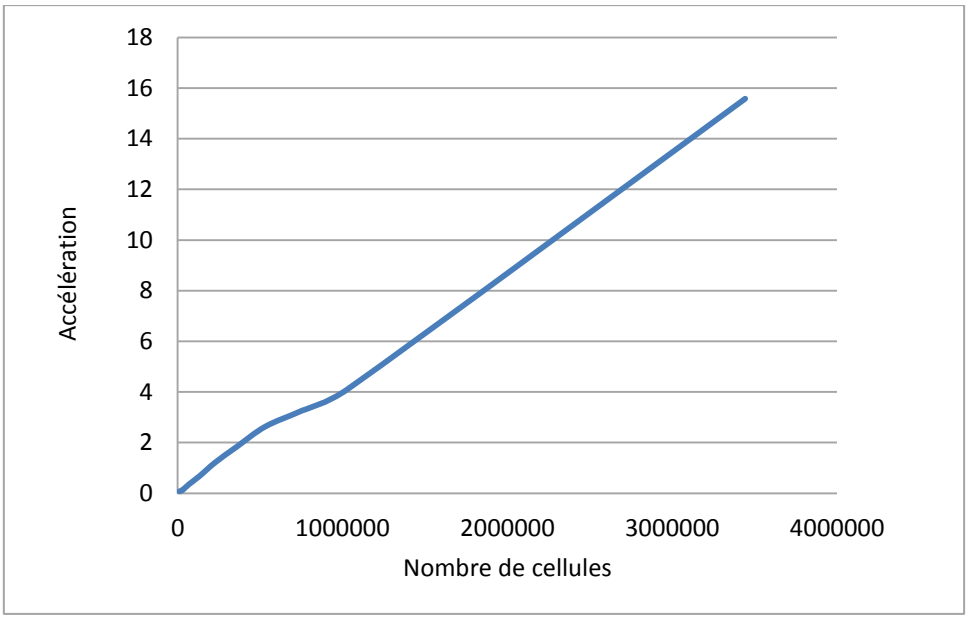


Figure 4-34 : Accélération du temps de calcul du champ magnétique en fonction du nombre de cellule grâce à une parallélisation sur GPU

4.2.4.2 Calcul du champ gravitationnel et du champ électrique

Le calcul du champ de gravitation et celui du champ électrique sont algorithmiquement très similaires. Ainsi, pour un cube de 51 cellules de côté (132 651 cellules au total), le temps nécessaire, en séquentiel, pour le calcul du champ de gravité en fonction du nombre de particules est donné par la Figure 4-35. Le temps nécessaire, dans les mêmes conditions, pour le calcul du champ électrique est donné par la Figure 4-36. Il est évident que les temps d'exécution sont extrêmement proches. L'étude des courbes de tendances des deux calculs de champs montre également que les temps de calcul, qui évoluent logiquement quasiment linéairement par rapport au nombre de particules, croissent à un rythme très similaire (avec une pente respectivement de 0,018 et de 0,0183). Les résultats sur GPU étant également extrêmement proches entre les deux calculs de champs, les documents présentés dans la suite de cette partie ne porteront que sur le calcul du champ gravitationnel, leurs équivalents pour le calcul du champ électrique sont disponibles dans l'annexe 1.

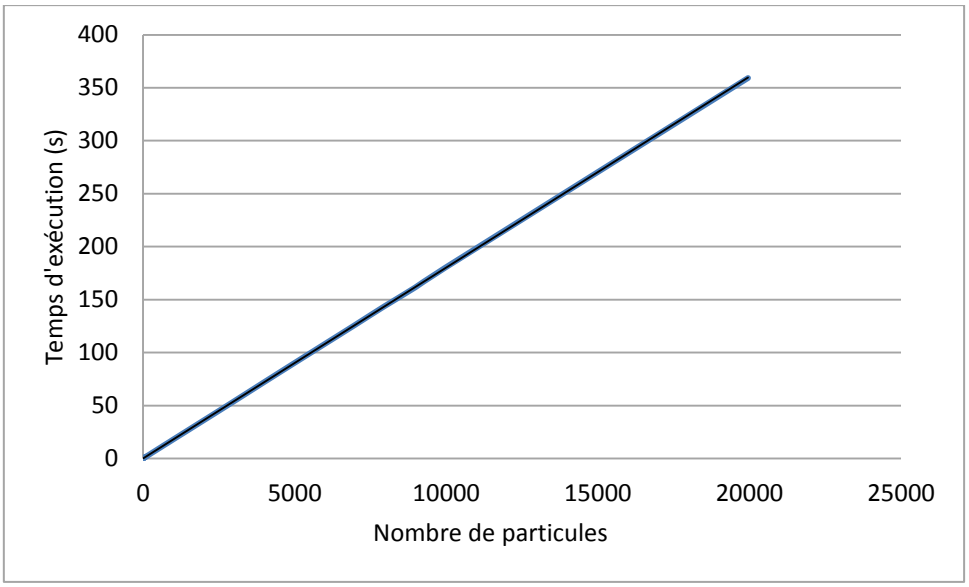


Figure 4-35 : Temps d'exécution du calcul du champ de gravité en séquentiel en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

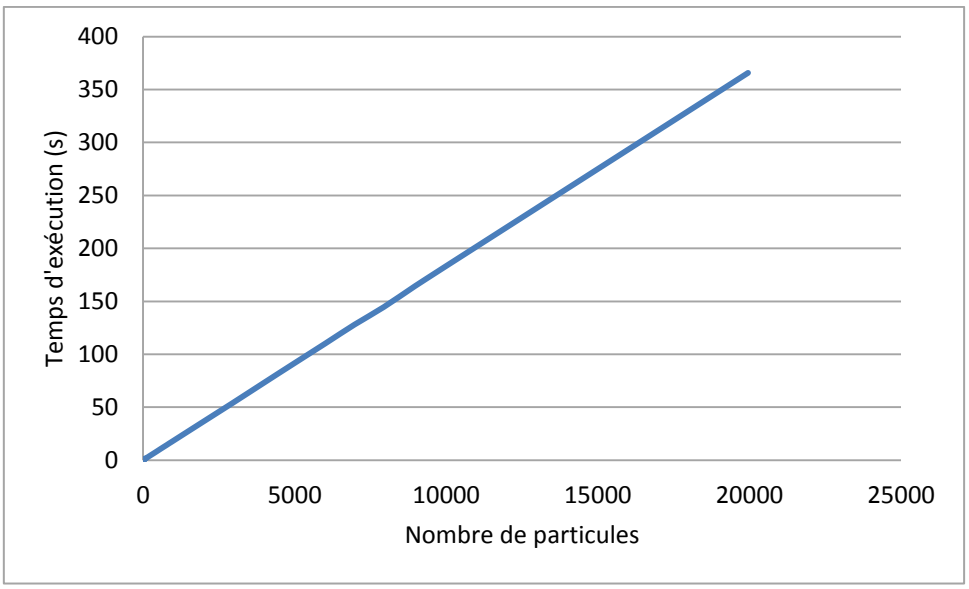


Figure 4-36 : Temps d'exécution du calcul du champ de électrique en séquentiel en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

La Figure 4-37 présente le temps d'exécution nécessaire pour effectuer le calcul du champ de gravité sur GPU selon le nombre de particules présentes. Le temps de calcul suit une courbe moins linéaire que la précédente. On peut malgré tout calculer une courbe de tendance assez proche. Et là où la pente de cette courbe de tendance valait $1{,}83 \times 10^{-2}$, soit une seconde supplémentaire toutes les 55 nouvelles particules, dans le cas de l'application séquentielle, elle vaut dans le cas de l'application parallèle $7{,}03 \times 10^{-5}$, soit une seconde

supplémentaire toutes les 14225 particules. L'application séquentielle a donc un temps d'exécution qui croît de manière 250 fois plus importante que l'application parallèle.

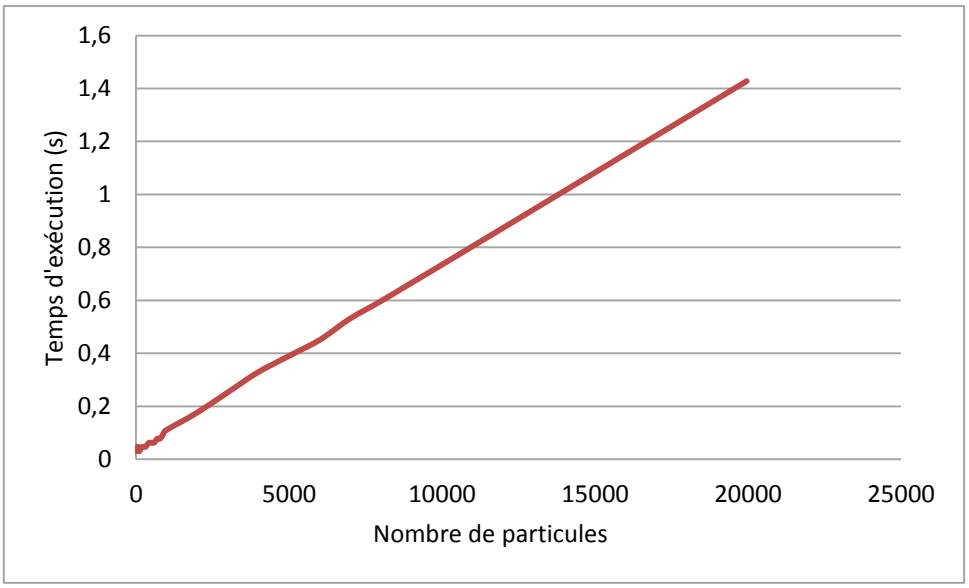


Figure 4-37 : Temps d'exécution du calcul du champ de gravité sur GPU en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

Ce rapport est confirmé par l'étude de l'accélération fournie par la parallélisation du calcul de champ sur GPU. Comme le montre la Figure 4-38, l'accélération induite par la parallélisation a tendance à se stabiliser autour d'un gain de l'ordre de 250x. Avant cette stabilisation, l'accélération croît progressivement. Ceci est dû à certains mécanismes liés à l'utilisation des GPU : principalement la copie des éléments générant le champ depuis le CPU vers le GPU ainsi que la copie du tableau correspondant au champ magnétique à la fin du kernel. Ces opérations représentent des coûts importants mais globalement fixes. Plus le nombre de particules utilisées est important et moins ces coûts fixes sont significatifs, ce qui explique cette courbe.

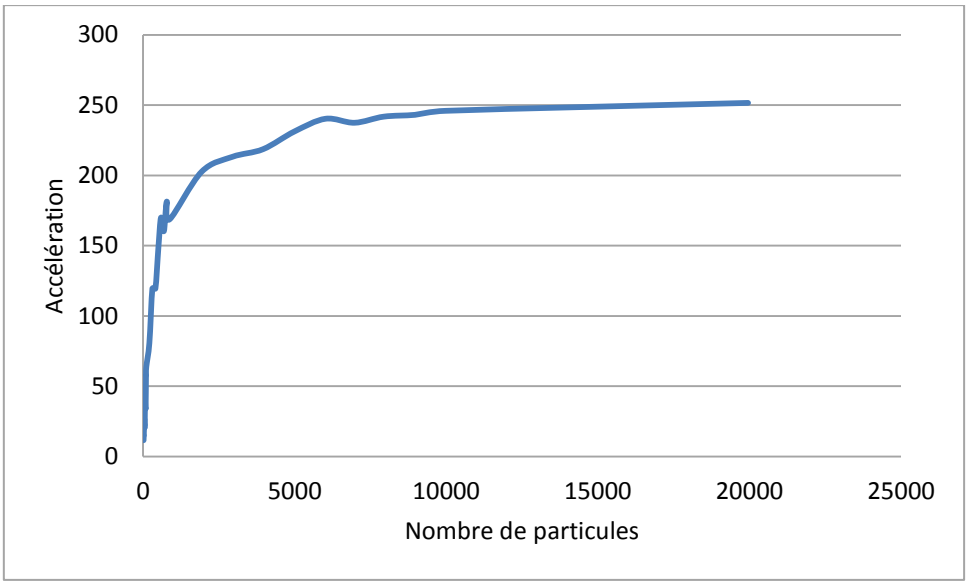


Figure 4-38 : Accélération du temps d'exécution du calcul du champ de gravité réalisée grâce à la parallélisation sur GPU en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

Tout comme pour le calcul du champ magnétique, le temps d'exécution nécessaire au calcul des champs gravitationnel et électrique évolue également en fonction de la taille de l'espace. La Figure 4-39, qui présente les temps d'exécution en séquentiel et lors d'une parallélisation sur GPU pour un nombre fixe de particules (200), montre bien l'écart très important entre les temps de simulation. L'augmentation de la pente à partir de plus d'un million de cellules dans le cas de l'implémentation séquentielle peut être expliquée par la saturation du cache du processeur.

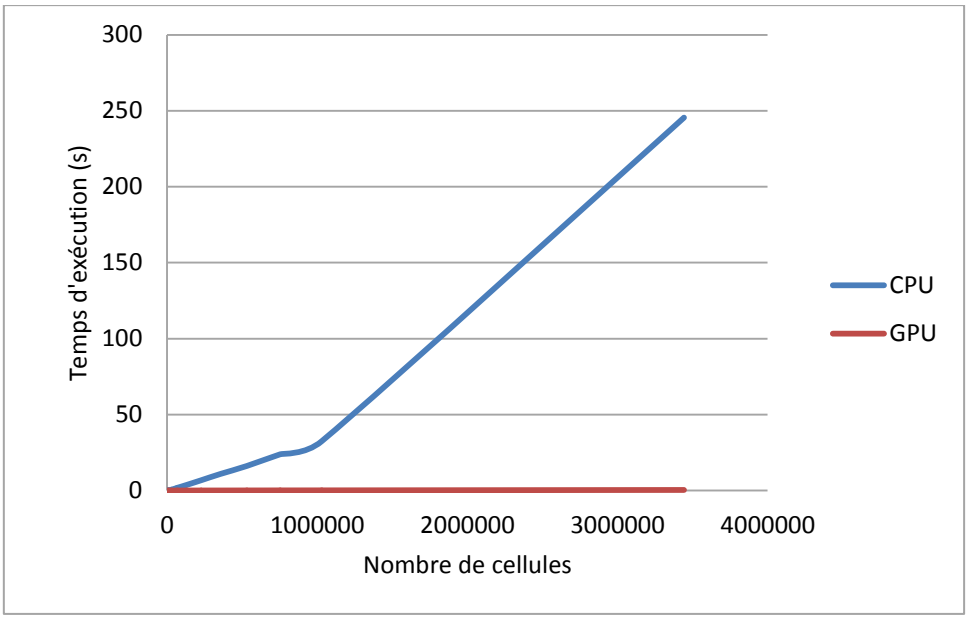


Figure 4-39 : Comparaison du temps d'exécution du calcul de champ de gravité en fonction du nombre de cellules présentes sur CPU et sur GPU (200 particules, 20 itérations).

La Figure 4-40, qui ne représente plus que le temps d'exécution sur GPU permet de se rendre compte que si le temps d'exécution semble ne pas évoluer dans la figure précédente, il est bien en constante augmentation. Mais à l'inverse de l'implémentation séquentielle, l'augmentation est très faible et reste sensiblement la même pour l'ensemble de l'intervalle étudié. La carte graphique étant déjà complètement utilisée à la fin de cet intervalle, rien ne tendrait à faire accélérer ou diminuer l'évolution de cette courbe.

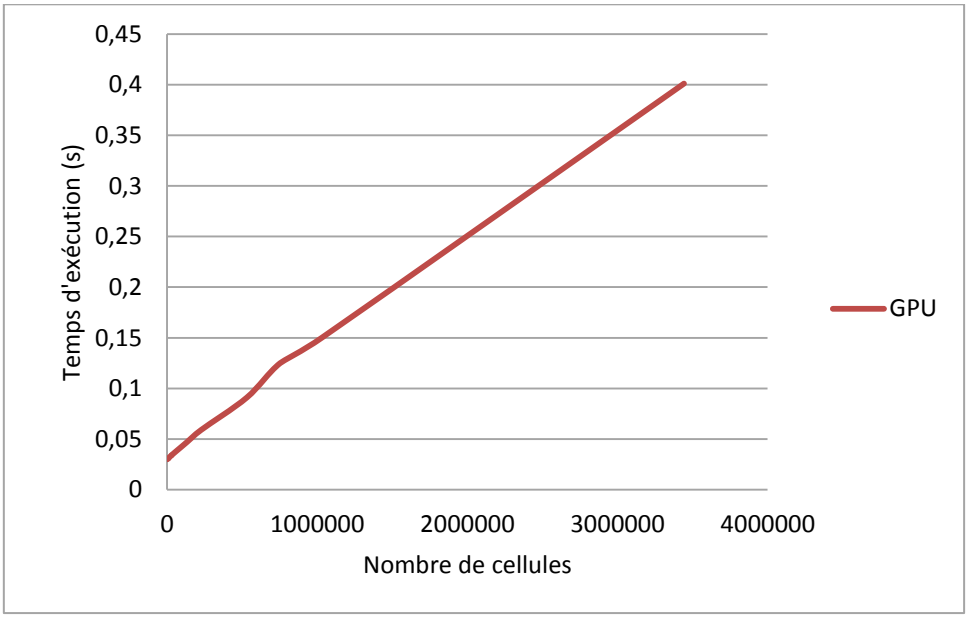


Figure 4-40 : Temps d'exécution du calcul de champ de gravité en fonction du nombre de cellules présentes sur GPU (200 particules, 20 itérations).

La différence de temps d'exécution entre l'implémentation séquentielle et l'implémentation parallèle étant très importante, le gain obtenu entre les deux simulations est donc significatif (voir Figure 4-41). Le traitement d'un espace de 100^3 cellules (soit un million de cellules) est ainsi plus de 215 fois plus rapide lorsque le calcul est parallélisé sur une carte GPU possédant 512 cœurs (GTX 500). Pour un cube de 151^3 cellules (plus de 3,4 millions de cellules), le gain atteint plus de 610x.

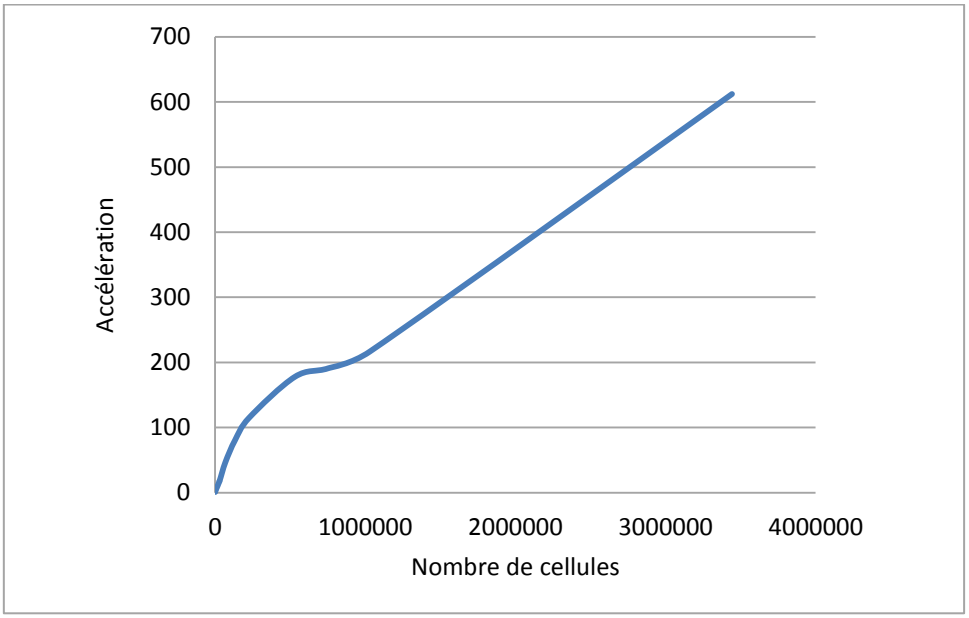


Figure 4-41 : Accélération du temps d'exécution du calcul du champ de gravité réalisée grâce à la parallélisation sur GPU en fonction du nombre de cellules (200 particules, 20 itérations).

L'utilisation de GPU lors du calcul des champs gravitationnel et électrique est donc particulièrement efficace pour les cubes de très grandes tailles et pour un nombre important de particules générant ces champs.

4.2.4.3 Calcul du mouvement de particules

Le calcul des champs physiques n'est pas actuellement utilisé au sein de la PGMS et n'est donc pas utile en soit pour l'application. Ces champs sont par contre nécessaires pour permettre de faire évoluer des particules dans l'espace et dans le temps en fonction des différentes forces en présence (magnétique, électrique et de gravité). Les champs vont alors devoir être calculés de nouveau à chaque itération de la simulation pour permettre de calculer la nouvelle position d'une particule.

Étant donné que les calculs des champs gravitationnel et électrique sont significativement plus longs que le calcul du champ magnétique, les résultats sont très proches de ceux présentés pour le calcul des champs gravitationnel et électrique. Ainsi, lorsque l'on étudie le gain produit par la parallélisation sur GPU en fonction du nombre de particules présentes (voir Figure 4-42), on s'aperçoit rapidement que celui-ci est très proche de ce qui existe pour le calcul des champs gravitationnel et électrique (voir Figure 4-38). On retrouve ainsi un gain qui se stabilise autour de 250x lorsque le nombre de particules devient suffisamment grand. Du fait de la présence du calcul du champ de gravité et du calcul du champ électrique, ce palier est atteint plus tôt : autour des 3000 particules.

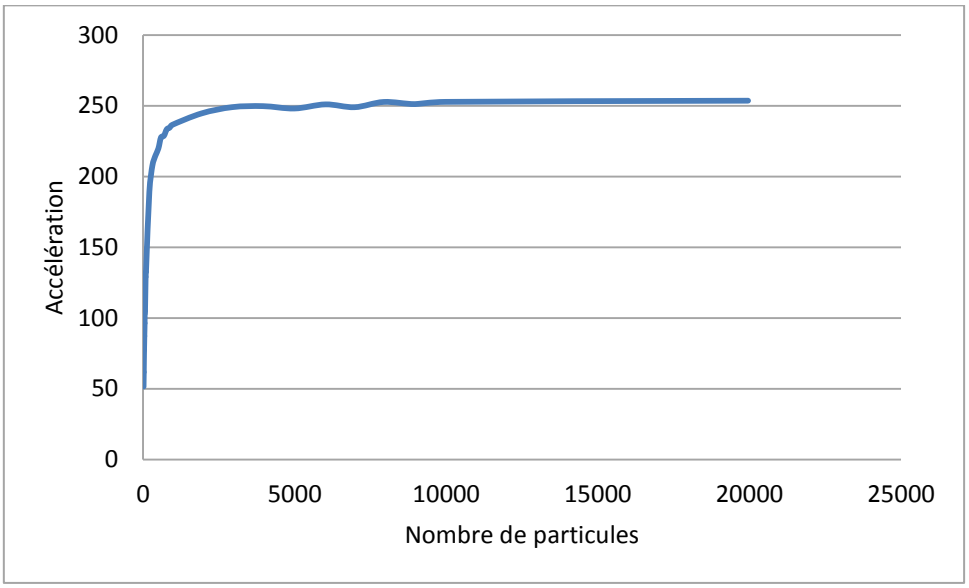


Figure 4-42 : Accélération du temps d'exécution du calcul de l'évolution des particules réalisée grâce à la parallélisation sur GPU en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

De la même façon, l'évolution du temps d'exécution en fonction du nombre de cellules étudiées (voir Figure 4-43) se rapproche de celle du calcul du champ de gravité présentée sur la Figure 4-41. Ainsi, l'accélération croît (globalement) avec le nombre de cellules pour atteindre près de 630x pour un cube de 151^3 cellules. L'évolution plus lente de l'accélération pour un nombre limité de cellules est toujours due à deux facteurs. D'une part, l'application séquentielle est plus efficace lorsque le nombre de cellules est suffisamment limité pour maximiser l'utilisation des caches présents au niveau du microprocesseur. D'autre part, les coûts fixes liés aux communications mémoires entre l'hôte et le GPU sont de moins en moins significatifs pour ce qui est de l'application parallèle. L'impact de ces deux facteurs, une

application séquentielle proportionnelle au nombre de cellules (plus rapide pour les petites simulations) et une application parallèle proportionnellement moins rapide pour ces mêmes simulations, expliquent l'évolution de la courbe.

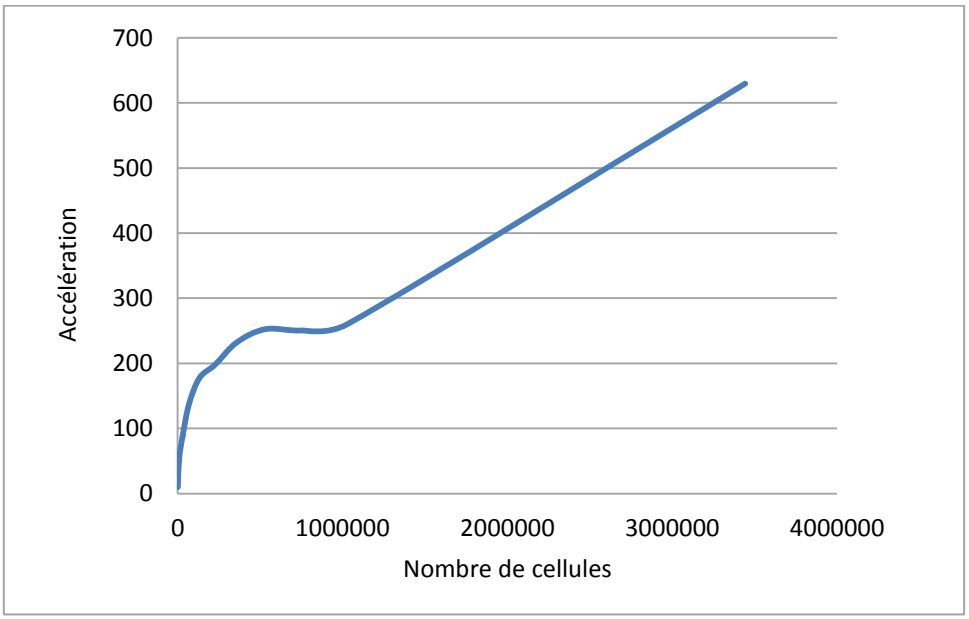


Figure 4-43 : Accélération du temps d'exécution du calcul de l'évolution des particules réalisée grâce à la parallélisation sur GPU en fonction du nombre de cellules (200 particules, 20 itérations).

Mais la parallélisation des calculs de champs physiques n'est pas intégrée actuellement aux simulations de la PGMS. Cette parallélisation visait donc à faciliter la parallélisation du calcul des champs chimiques déjà présents dans la PGMS ainsi que l'intégration du calcul des champs physiques bientôt incorporés à la PGMS.

Ce travail a donc permis la parallélisation sur GPGPU du calcul de champs physiques. Mais ces champs ne sont pas pour le moment utilisés par les modèles implémentés au sein de la PGMS. Il s'agissait donc d'un travail préalable à l'implémentation au sein de la PGMS de la parallélisation des champs chimiques.

4.3 Parallélisation au niveau des warps

Si l'efficacité des GPU pour réaliser de manière fortement parallèle (plusieurs centaines de threads pouvant s'exécuter en même temps sur un même GPU) des calculs identiques sur des données différentes n'est plus à prouver, nous avons cherché à étudier l'opportunité d'utiliser les GPU pour réaliser, de façon parallèle, des calculs différents.

4.3.1 Idée directrice

L'idée n'est pas ici d'exécuter plusieurs fonctions différentes en parallèle sur un GPU mais une même fonction qui, en fonction de branchements conditionnels, pourrait exécuter des instructions différentes. Ces divergences entre les exécutions sont généralement présentes dans les applications stochastiques où le flot d'exécution est conditionné par les nombres aléatoires générés. Il est par ailleurs obligatoire, lors de l'étude de modèles stochastiques, de réaliser plusieurs réplications de la même simulation. En effet, sans celles-ci, le résultat ne correspondrait qu'à une exécution particulière et ne représenterait pas une moyenne de l'ensemble des possibilités. Une image simple de l'utilité du processus de réplication est la simulation d'un jeu de « pile ou face ». Une seule exécution de la simulation donnerait l'impression qu'un côté de la pièce survient dans 100% des cas et l'autre jamais. En répétant suffisamment l'expérience, on arrive très vite à se rapprocher des 50% de probabilité pour chaque côté de la pièce (une trentaine d'itérations est généralement suffisante pour avoir une précision acceptable). C'est ce mécanisme de réplication que nous avons cherché à paralléliser sur GPU.

Pour ce faire, il n'est plus possible d'exploiter le parallélisme SIMD présent au sein des GPU et qui nécessite de réaliser la même instruction sur un groupe de trente-deux threads (appelé un warp). En effet, lorsque des threads d'un même warp souhaitent exécuter des instructions différentes (comme c'est le cas si des branches différentes sont parcourues), ces instructions sont réalisées de façon séquentielle. Cette opération nécessite un traitement supplémentaire du GPU pour réorganiser le warp de façon à isoler les threads traitant les mêmes instructions de ceux qui divergent. Notre idée est de ne plus effectuer une exécution par thread mais une exécution par warp afin de ne plus nécessiter ce traitement supplémentaire (voir Figure 4-44). De cette façon, quelles que soient les divergences, aucun traitement supplémentaire ne sera nécessaire car seul un thread sera actif dans chaque warp. Ainsi, si l'on souhaite paralléliser cinquante réplications d'une simulation, il ne suffira pas d'utiliser autant de threads que de réplications sur le GPU mais bien trente-deux fois plus, soit 50x32=1600 threads.

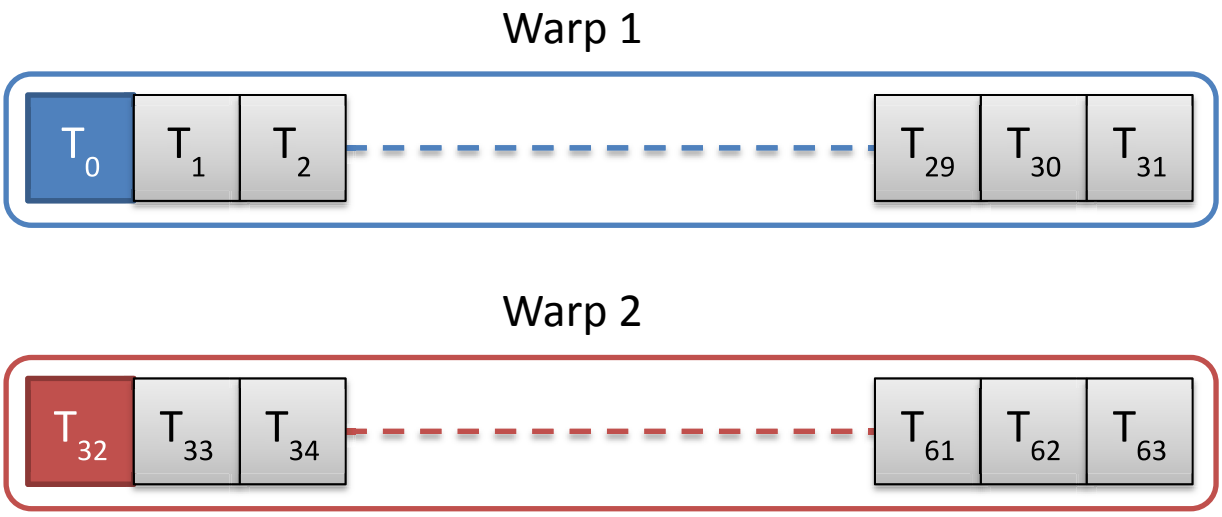


Figure 4-44 : Représentation de l'unique thread actif (en couleur) au sein d'un warp (les autres threads sont grisés).

4.3.2 Implémentation

Pour permettre le traitement d'une réplication par un seul thread du warp, deux mécanismes sont nécessaires :

- Il faut tout d'abord que chaque thread puisse identifier l'indice du warp auquel il appartient.
- Il faut ensuite pouvoir restreindre l'exécution de la fonction à un seul thread de chaque warp.

L'identification du warp est réalisée en calculant l'indice global du thread puis en divisant cet indice par la taille d'un warp (voir Code 4-11).

```
const unsigned int warpIdx = (
         threadIdx.x + blockDim.x * (
            threadIdx.y + blockDim.y * (
               threadIdx.z + blockDim.z * (
                  blockIdx.x + gridDim.x *
                     blockIdx.y
         ) ) ) ) / warpSize;
```

Code 4-11 : Calcul de l'indice du warp.

Le calcul de l'indice global n'est pas anodin et doit suivre l'organisation utilisée par CUDA. Au sein d'un bloc, c'est donc l'indice en « x » qui est considéré en premier, suivi de l'indice en « y » puis de l'indice en « z ». Le mécanisme est le même pour les blocs au sein de la grille où seuls les indices x et y sont utilisés (Kirk et Hwu 2010). La variable `warpSize`, qui correspond

au nombre de threads dans un warp, est quant à elle fournie par l'API CUDA, ce qui permet de rendre le code générique.

Pour restreindre l'utilisation d'un seul thread par warp, il n'est plus nécessaire de calculer l'identifiant global mais seulement l'identifiant propre au bloc (car tous les threads d'un warp appartiennent au même bloc). Une fois cet identifiant calculé, le calcul de son modulo par rapport à la taille d'un warp permet d'identifier ce thread dans le warp. Pour ne conserver qu'un thread actif par warp, il suffit alors de ne traiter que les threads possédant un indice précis au sein de leur warp (le premier qui est d'indice 0 dans notre cas) (voir Code 4-12).

```
if ( ( threadIdx.x + blockDim.x *
          ( threadIdx.y + blockDim.y * threadIdx.z )
       ) % warpSize == 0 )
```

Code 4-12 : Restriction des calculs à un thread par warp.

Encapsulé dans des macro-instructions (respectivement `WARP_INIT` pour l'initialisation de `warpIdx` et `WARP_BEGIN` pour le test de l'indice du thread dans le warp), ces deux mécanismes sont extrêmement simples à mettre en place. Il suffit de commencer le kernel par le test de l'indice (`WARP_BEGIN`) puis d'initialiser la variable `warpIdx` (`WARP_INIT`) (voir Code 4-13).

```
__global__ void replication_gpu( ... )
{
   WARP_BEGIN
   {
      WARP_INIT
      [...] // Traitement de la simulation en utilisant warpIdx
            // lorsqu'un indice unique à la réplication est nécessaire
   }
}
```

Code 4-13 : Exemple d'utilisation des macro-instructions WARP_BEGIN et WARP_END.

4.3.3 Résultats

Les tests, que nous avons effectués, ont été réalisés sur un ordinateur équipé d'un processeur Intel Westmere cadencé à 2,527GHz et d'une carte graphique NVIDIA C2050. Nous avons cherché à comparer une approche séquentielle sur CPU à notre approche utilisant la parallélisation par les warps ainsi que notre approche à l'approche conventionnelle sur GPU utilisant les threads.

Le premier modèle étudié a été celui d'une simulation de Monte Carlo permettant d'approximer la valeur de Pi. Pour ce faire, la simulation tire au hasard une succession de coordonnées x et y comprises entre 0 et 1 et incrémente un compteur pour chaque point se trouvant dans le quart d'un cercle unitaire. La valeur de Pi est approximée, en fin de simulation, en calculant le ratio entre les points trouvés dans le cercle et la totalité des points.

La Figure 4-45 montre fort logiquement que le temps de l'application séquentiel croît linéairement avec le nombre de réplications. La parallélisation par rapport au warp évolue quant à elle par palier. Ce comportement n'a rien d'anormal car tant que toute la capacité de la carte GPU n'est pas utilisée, ajouter une itération supplémentaire n'a pas d'impact sur les performances. Mais lorsque celle-ci est complètement utilisée, l'ajout d'une réplication nécessite qu'une partie de la carte réalise deux réplications séquentiellement. Ceci explique le palier après la 64ème réplication lorsque toute la capacité disponible est utilisée. Ce mécanisme explique également que l'utilisation des GPU ne soit pas rentable en dessous de 30 réplications mais le devienne de plus en plus avec l'augmentation du nombre de réplications.

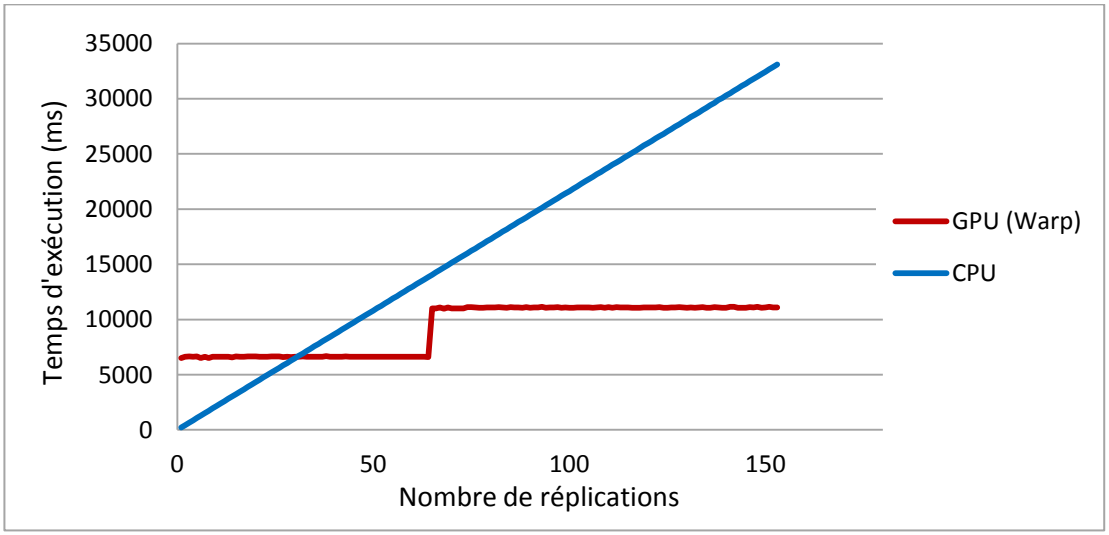


Figure 4-45 : Comparaison pour la simulation de Monte Carlo des temps d'exécution de l'approche séquentielle et de notre approche par rapport au nombre de réplications.

Un second modèle a servi à comparer la parallélisation sur GPU à l'aide de l'approche traditionnelle et notre approche fondée sur l'utilisation des warps. Ce modèle est une adaptation du « random walk » (Vattulainen et Ala-nissila 1995) qui consiste à simuler des

marcheurs se déplaçant aléatoirement sur un damier. À la fin de la simulation, chaque marcheur a effectué un même nombre de mouvements et l'on comptabilise le nombre de marcheurs dans chaque zone du damier. Si le flux stochastique est de bonne qualité, la répartition des marcheurs dans chaque zone est très proche. Pour complexifier le modèle existant, nos marcheurs ne disposaient pas de quatre possibilités de déplacement comme dans l'algorithme original mais de trente-deux, soit le nombre de threads dans un warp.

La premier graphique de la Figure 4-46 montre que le temps d'exécution nécessaire pour l'exécution avec l'approche classique sur GPU est près de six fois plus important que pour notre approche. Il n'est ainsi rentable de paralléliser sur GPU à l'aide de l'approche classique qu'à partir de plus de 90 réplications contre 20 réplications pour notre approche basée sur les warps. Mais le temps d'exécution du programme en utilisant notre approche finit par rattraper le temps d'exécution de l'approche classique à partir de 700 réplications avant de le dépasser (deuxième graphique de la Figure 4-46). L'utilisation optimale de notre approche se situe donc, pour ce modèle, entre 20 et 700 itérations. Mais ces résultats peuvent varier de façon importante en fonction du modèle utilisé (avec le modèle précédent, notre approche ne permettait aucun gain pour moins de 30 itérations par exemple).

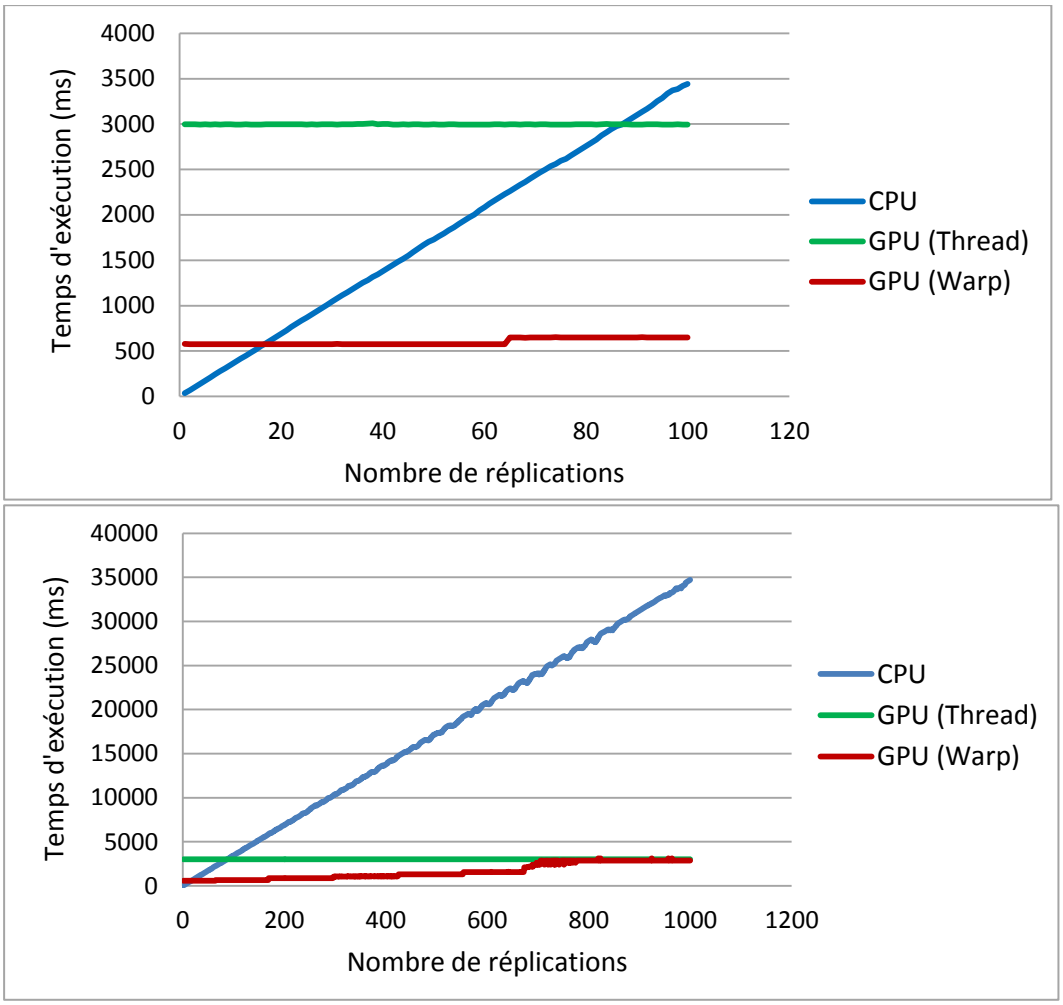


Figure 4-46 : Comparaison pour le modèle de « random walk » des temps d'exécution de l'approche séquentielle, de l'approche GPU classique et de notre approche par rapport au nombre de réplications.

Conclusion

Afin d'améliorer les performances de la PGMS, plusieurs problèmes ont été étudiés. Le premier problème étudié a été l'implémentation de simulateurs d'automates cellulaires 3D. Ceux-ci ont une double utilité en modélisation biologique. Ils permettent d'une part de modéliser des mécanismes biologiques mais également d'implémenter des mécanismes de simulation multi-échelle. Afin d'autoriser la modélisation des systèmes biologiques complets, il est nécessaire que la simulation de ces automates cellulaires soient la plus rapide possible. Dans ce but, l'implémentation de l'algorithme Hash-Life en 3D a permis de mettre en évidence son excellent rendement sur certains modèles d'automates cellulaires. Il a ainsi été possible de simuler des espaces de plus de 10^{27} cellules de manière extrêmement rapide (environ 1,1 secondes pour parcourir tout l'espace dans un milieu excitable).

Une étude de l'algorithme Hash-Life, en considérant la notion d'activité au sein d'un automate, a par ailleurs mis en évidence le poids essentiel de la réapparition des configurations de cellules dans les performances de celui-ci. L'algorithme tirant massivement parti de la mémoïzation, le temps d'exécution de la simulation des automates cellulaires à l'aide de Hash-Life évolue de façon très proche du pourcentage de réapparition des configurations de cellules déjà rencontrées au cours de la simulation. Ces résultats impliquent que l'algorithme n'est pas efficace pour toutes les formes d'automates cellulaires et que des solutions autres sont à chercher pour les formes ne permettant pas une utilisation efficace de l'algorithme Hash-Life.

Afin d'améliorer les performances de la simulation d'automates cellulaires 3D, lorsque le choix de l'algorithme Hash-Life n'est pas pertinent, l'implémentation d'un simulateur sur GPU a également été étudié (Caux, Siregar et Hill 2011). La parallélisation de l'évolution de chaque cellule dans un thread propre permet de disposer d'un nombre très important de threads pouvant s'exécuter en parallèle. La parallélisation sur GPU a ainsi permis d'améliorer la vitesse d'un facteur 60 par rapport à un cœur de microprocesseur moyenne gamme et d'un facteur 20 par rapport à un très bon microprocesseur de même génération (un cœur Nehalem 5500 vs. Tesla C1060). Sachant, d'une part que la nécessité d'accéder à un voisinage peut être très importante dans un modèle en trois dimensions (vingt-six voisins dans le cas d'un voisinage de Moore de degré un) et d'autre part que le choix qui a été fait de conserver un code extrêmement générique sans l'optimiser pour un automate cellulaire en particulier a limité les performances de la simulation, il parait envisageable d'améliorer encore ces performances dans le cadre d'une optimisation pour un automate cellulaire bien précis.

Mais pour améliorer les performances de la PGMS, la parallélisation des processus les plus gourmands en termes de temps de calcul était nécessaire. Parmi ces processus se trouvait le calcul des champs physicochimiques. En vue de l'intégration dans la PGMS, l'étude de leur parallélisation a donc été réalisée en s'appuyant sur trois champs physiques bien connus : le champ de gravité, le champ électrique et le champ magnétique. Dans notre modèle, le champ de gravité et le champ électrique étaient générés par l'ensemble des particules présentes dans le système ; le champ magnétique étant quant à lui généré par un seul élément. La parallélisation choisie, centrée sur le découpage existant de l'espace de

simulation, a permis de réaliser le calcul de la valeur d'un champ en chaque cellule de l'espace à l'aide d'un thread. Dans le cas des champs gravitationnel et électrique, la mise en commun des caractéristiques des particules (coordonnées, masse et charge) a permis d'améliorer les performances de l'application.

Les résultats de ces parallélisations sont très proches pour le champ de gravité et le champ électrique avec des gains de plus en plus importants lorsque l'on augmente la taille de l'espace de simulation ou le nombre de particules présentes. Ces gains, particulièrement importants, peuvent ainsi varier entre 215x et 610x lors du traitement de 200 particules selon que l'espace comprend de 1 à 3,4 millions de cellules (un cœur Sandy Bridge i7-2600K à 3,4GHz vs. GTX500). Dans le cas du champ magnétique, une seule source est présente, le gain est donc facteur uniquement de la taille de l'espace de simulation. Si le gain croît encore une fois avec cette taille, il croît beaucoup plus faiblement et ne dépasse pas les 15x pour un espace de 3,4 millions de cellules.

Lorsque l'on utilise ces trois champs pour faire évoluer dans l'espace les particules (elles-mêmes influençant à leur tour la nouvelle valeur des champs à chaque itération), les résultats se rapprochent de ceux obtenus lors des calculs des champs gravitationnel et électrique. Ceci s'explique simplement par le poids beaucoup plus important du calcul de ces deux champs sur le temps d'exécution de l'application (le calcul de chacun de ces deux champs en présence de 200 particules est plus de cent fois plus long que le calcul du champ magnétique). Les gains croissent donc une nouvelle fois avec le nombre de particules présentes ainsi qu'avec la taille de l'espace, avec des valeurs très proches de ce qui existait pour les champs gravitationnel et électrique (de l'ordre de 600x pour le traitement de 200 particules dans un espace de 3,4 millions de cellules).

Mais ces solutions de parallélisations sur GPU sont propres à un problème donné. Afin d'exploiter au mieux la puissance des GPU pour n'importe quelle application stochastique, la parallélisation, en suivant une approche MIMD basée sur les *warps* (groupe de trente-deux threads exécutés en même temps selon une approche SIMD sur le GPU), a été étudiée. Cette approche, qui consiste à ne travailler que sur un seul thread par warp, permet d'éviter les problèmes de la parallélisation sur GPU lorsque plusieurs threads d'un même warp parcourent des branches différentes. Bien entendu, le fait de ne jamais utiliser l'approche

SIMD des GPU, présente au niveau des threads d'un même warp, implique une limitation des performances. Pour autant, lorsque la parallélisation est limitée, comme c'est le cas lors du calcul de réplications (entre 30 et 700 réplications pour le modèle que nous avons testé), la parallélisation via l'approche warp est plus pertinente.

Tous ces développements, plus ou moins théoriques, ont eu pour but d'aider à l'amélioration des performances de la PGMS. Si toutes n'ont pas pu être intégrées à l'heure actuelle dans le code de la PGMS, certaines ont permis d'obtenir des gains très intéressants. C'est l'intégration de ces développements ainsi que toutes les autres améliorations apportées au code de la PGMS (refactorisation et optimisations) qui sont le thème du prochain chapitre.

Intégration des solutions à la PGMS et optimisations

5 Intégration des solutions à la PGMS et optimisations

Introduction

Si certains travaux peuvent sembler avoir été réalisés hors du cadre de la PGMS, cela a toujours été fait sans perdre le but d'améliorer les performances de la PGMS. Ainsi, le travail sur les automates cellulaires a été réalisé en vue d'une intégration ultérieure à la PGMS, lorsque celle-ci se sera enrichie des éléments nécessitant des automates cellulaires. Pour ce qui est du calcul des champs physicochimiques, l'application est plus directe. Car s'il est vrai que l'incorporation des champs physiques n'est pas encore réalisée au sein de la PGMS, l'utilisation intensive de champs chimiques, dont les calculs sont approximés en utilisant des formules analytiques semblables à celle des champs physiques permet de tirer parti du travail présenté dans le chapitre précédent.

Mais le projet de la PGMS, avant que les premiers travaux ne soient effectués, nécessitait d'être repris en partie. En effet, le code de la PGMS, en raison d'un développement ancien, comprenait certains éléments devenus obsolètes. On trouvait ainsi parmi les API utilisées l'API Borland C++ qui fournissait entre autre la totalité des conteneurs utilisés ainsi que tous les éléments de l'interface graphique. Un travail a donc dû être mené afin d'extraire le code de cette dépendance devenue obsolète. Le projet manquait quant à lui d'outils aujourd'hui très courants dans les projets récents. La gestion des versions se faisait par exemple de manière manuelle à travers des commentaires dans le code. L'incorporation au projet d'outils performants a permis de faciliter les travaux d'amélioration des performances. Une fois ces travaux réalisés, il a donc été possible de travailler efficacement sur l'amélioration des performances de l'application, à travers la parallélisation du calcul de champs chimiques et l'incorporation d'optimisations du code existant permettant d'améliorer sensiblement les performances.

La première partie de ce chapitre présentera donc le profilage de la PGMS originale qui a servi de base à l'amélioration de l'application. Une seconde partie traitera des modifications apportées au projet en amont de sa parallélisation et de son optimisation afin d'améliorer celui-ci. La troisième partie traitera de la parallélisation des champs chimiques à l'aide des techniques introduites dans le chapitre précédent pour les champs physicochimiques. Une quatrième partie présentera les optimisations réalisées afin d'améliorer l'exécution

séquentielle de l'application. Enfin, une dernière partie étudiera l'effet des modifications apportées au code de la PGMS sur le temps d'exécution de deux autres simulations.

5.1 L'existant

Afin de bien concentrer les efforts d'amélioration des performances sur les points bloquants et de maximiser les gains obtenus, il est nécessaire que ceux-ci soit connus. Il faut donc réaliser le profilage de l'application à l'aide d'outils spécialisés. C'est ce qui a été fait avec la version de base (sans optimisations) de la PGMS puis avec toutes les autres versions.

Le profileur utilisé fournit comme résumé les résultats de la Figure 5-1. À noter que tous les résultats dans la suite ont été réalisés sur une machine correspondant à la machine cible pour la PGMS : une machine de bureau performante équipée d'un excellent processeur graphique : processeur Intel Core i7-2600K, 8Go de mémoire (RAM) cadencée à 1866MHz et carte graphique GTX 590 (qui intègre deux GPU GTX 500 à 512 cœurs chacun). Par ailleurs, sauf lorsque le contraire est précisé, la simulation que nous présentons est celle de la génération d'un unique néphron. L'étude des résultats des différentes simulations est présentée à la fin de cette partie du manuscrit.

Fonctions faisant le plus de travail individuel

Fonctions présentant les échantillons les plus exclusifs

Nom	% d'échantillons exclusifs
Field::ComputeSemaphorinField(class MaterialEntity *,floa	51,33
ChemField::ComputeDiffusion(class UniVector<class Fielc	14,41
zzz_AsmCodeRange_Begin	10,16
zzz_AsmCodeRange_Begin	6,07
__load_CW	3,67

Figure 5-1 : Résumé du profilage de l'application PGMS de base.

La Figure 5-1 propose les cinq fonctions réalisant le plus de travail individuellement, c'est-à-dire les cinq fonctions dont le code spécifique prend, en cumulé, le plus de temps à être exécuté. Pour autant, ce tableau peut être trompeur car le pourcentage affiché n'est que le pourcentage propre à la fonction, il ne prend pas en compte les appels aux fonctions standards souvent codés en langage d'assemblage (`exp`, `log`…) et les appels à d'autres fonctions définies par l'utilisateur. Or, comme le montre la Figure 5-2 qui représente la composition d'une des deux méthodes, les deux méthodes assembleurs

`zzz_AsmCodeRange_Begin` (`exp` et `log`), qui présentent respectivement à 10,16% et 6,07%, sont en fait en majeure partie appelées depuis les deux premières méthodes. La dernière instruction qui représente 3,67% est quant à elle partie intégrante des deux autres fonctions en langage d'assemblage (chargement des indicateurs –flags en anglais– avant les calculs). Ainsi, lorsque l'on s'intéresse au temps d'exécution des méthodes `ComputeSemaphorinField` et `ComputeDiffusion` en comptant également les appels en leurs seins aux codes assembleurs et aux autres méthodes, le pourcentage de temps d'exécution est respectivement de 71,39% et 23,30% soit 94,69% pour le cumul des deux.

Figure 5-2 : Découpage du temps d'exécution de la méthode ComputeSemaphorinField avec l'implémentation de base de la PGMS.

Il est donc, dans un premier temps, quasiment inutile d'essayer d'optimiser une partie du code qui n'aurait pas d'influence sur la vitesse d'exécution de ces deux méthodes. Le seul gain qu'il serait alors possible d'espérer serait au mieux de l'ordre de 5% s'il était possible de réaliser tous les autres calculs instantanément. La première optimisation a donc consisté à essayer d'améliorer les performances de ces deux fonctions à travers une implémentation parallèle sur GPU. Mais la parallélisation sur GPU nécessitant toujours un accès aux données, d'autres optimisations, telles que le stockage contigu couplé à l'accès séquentiel et au stockage sous forme de tableau de certains attributs ont donc été implémentées. Une fois ces optimisations réalisées, des fonctions, auparavant mineures en terme de temps de calcul, ont vu leur impact augmenter. Ceci a conduit à réfléchir à d'autres optimisations

telles que celles liées à l'utilisation d'un vecteur conservant l'ordre d'insertion mais interdisant tout ajout d'un élément déjà présent.

5.2 Refactorisation de l'application PGMS

La première étape du travail sur la PGMS n'a été ni l'optimisation, ni la parallélisation du code mais une phase de refactorisation afin de pouvoir travailler à partir d'une application fonctionnelle. Cette phase avait deux buts principaux. Le premier était de rendre le code le plus standard possible en abandonnant complètement l'API proposé par Borland. Le second but était de découpler autant que possible le code métier de l'application de façon à rendre celui-ci standard et l'interface graphique qui serait développée à l'aide du cadriciel Qt[17].

Pour faciliter le développement de l'application, la première étape a été de versionner l'ensemble du code de l'application afin de disposer de toutes les possibilités fournies par les outils de gestion de versions : conservation de l'historique des modifications, conservation de l'identité des développeurs ayant effectués des modifications, possibilité de revenir à une ancienne version du code… Par ailleurs, ces outils permettent de créer des branches différentes pour un même projet. Le code peut alors évoluer en parallèle sur les différentes branches avant que les modifications apportées depuis la création des branches soient fusionnées. Ceci fut particulièrement utile lors des travaux d'optimisation et de parallélisation du code qui ont pu être séparés du développement principal de l'application.

Comme cela a été expliqué précédemment, modifier un code sans disposer de mécanismes permettant de réaliser des tests automatisés du code peut s'avérer très compliqué. C'est pour cette raison qu'il a très vite était mis en place un cadriciel de test permettant de vérifier le comportement adéquat du programme. Le choix du cadriciel s'est porté sur celui de la bibliothèque boost : Unit Test Framework. Ce cadriciel a l'avantage d'être très simple d'utilisation et de faire partie de boost, une des bibliothèques les plus complètes existantes en C++. Le Code 5-1 est un extrait de code utilisé dans l'application. Une suite de tests est tout d'abord créée pour la simulation « Single Nephron 2 » à laquelle on rajoute les différents tests. Enfin, la suite de test est elle-même rajoutée à la suite principale qui sera exécutée automatiquement du fait de l'utilisation d'un « `main` » propre au cadriciel de test.

[17] Qt est un framework libre, disponible en licence gratuite ou payante selon l'utilisation qui en est faite. Il permet entre autre la conception d'interfaces graphiques. Site officiel : http://qt.nokia.com/.

```
// Creation de la suite de test pour Single Nephron 2
test_suite *  tsSingleNephron2 =
                      BOOST_TEST_SUITE( "Test_Single_Nephron2" );

// Ajout des differents tests de la suite
tsSingleNephron2->add( BOOST_TEST_CASE( &nephron2::testSimulation ) );
tsSingleNephron2->add( BOOST_TEST_CASE( &nephron2::testBioCells ) );
tsSingleNephron2->add( BOOST_TEST_CASE( &nephron2::testBioDucts ) );
tsSingleNephron2->add( BOOST_TEST_CASE( &nephron2::testBioFluids ) );
[...]

// Ajout de la suite de test pour Single Nephon 2
// a la suite principale
framework::master_test_suite().add( tsSingleNephron2 );
```

Code 5-1 : Extrait du code de test de la simulation "Single Nephron 2".

Mais le gros du travail n'a pas été la mise en place d'un gestionnaire de version ou celle de tests automatisés mais bien le découplage entre le code métier et l'interface graphique d'une part et la standardisation du code métier en éliminant les références à l'API proposée par Borland d'autre part.

Le travail de découplage du code métier et de l'interface graphique a consisté tout d'abord à retirer les différentes entités du programme traitant de l'interface graphique. Il s'agissait ainsi de classe de l'API tel que `TColor` pour stocker une couleur ou `TRect` pour représenter une zone de dessin rectangulaire ainsi que des fonctions et des méthodes dédiées à ces traitements. Une partie conséquente des classes étant correctement découpée, cette tache a également consisté à retirer certaines classes du projet principal. Le but de cette décomposition était de disposer le plus rapidement possible de la partie traitant du code métier afin que les travaux d'optimisations et de parallélisation puissent commencer en parallèle de la création d'une nouvelle interface graphique implémentée à l'aide de la bibliothèque Qt.

Mais le travail le plus conséquent a été le portage de tous les composants de l'API Borland (hors interface graphique qui a été reprise de zéro) en C++ standard. Les conteneurs proposés par Borland étaient particulièrement utilisés (aucun conteneur de la bibliothèque standard du C++ n'étant utilisé). Or, les conteneurs Borland ne proposant pas la même interface que ceux de la bibliothèque standard du C++, il a fallu d'une part changer les conteneurs utilisés vers des conteneurs standards et d'autre part adapter leurs utilisations. Certains conteneurs Borland, sans équivalant dans la bibliothèque standard ont du faire l'objet d'une implémentation spécifique. En plus de ces travaux majeurs sur les conteneurs,

un grand nombre de modifications plus simples (mais en quantité importante) ont également dû être effectuées (utilisation du mot clé `HUGE`, utilisation d'exceptions nommées différemment dans le standard…).

5.3 Application du calcul des champs physicochimiques à la PGMS

La PGMS, et plus spécifiquement la simulation des éléments du rein, nécessite de calculer à chaque itération de la simulation différents champs chimiques présents dans la simulation. Ces champs chimiques peuvent être considérés comme des champs scalaires obéissant à des équations aux dérivées partielles (EDP) de type diffusion.

5.3.1 Intégration du calcul des champs physicochimiques

Le calcul d'un tel champ en tout point de l'espace est donc différent de celui présenté ci-dessus pour les champs physiques. Cependant les calculs ont été simplifiés en s'affranchissant de résoudre le problème de diffusion par une EDP en l'approximant par une fonction analytique qui par conséquent nous rapproche du cas des calculs de champs physiques vu plus tôt. L'algorithme de calcul pour une cellule de l'espace est ainsi le suivant :

Pour chaque cellule biologique présente dans l'espace de simulation, **faire**

- Calculer la distance euclidienne entre la cellule biologique et la cellule de l'espace que l'on traite.
- Calculer la valeur du champ en fonction de cette distance, de paramètres de la simulation (constante) et éventuellement de paramètres de la cellule biologique elle-même (dynamique) et ajouter cette valeur à la valeur déjà présente.
- Calculer le vecteur gradient du champ correspondant et l'additionner au gradient déjà existant tant que les sources du champ sont actives.

Fin pour.

Il est bien clair ici que le calcul est dans sa structure quasi identique à celui d'un champ physique :

- Calcul de la distance entre le point traité et l'objet générant le champ.

- Calcul de la valeur du champ et l'ajouter à la valeur courante.
- Addition des champs générés par tous les objets présents dans l'espace.

La différence majeure est que les champs chimiques, dans cette approximation, se cumulent entre deux itérations. Le champ présent dans une cellule n'est donc pas nul au début du calcul, il est égal à la valeur qu'il avait à la fin de l'itération précédente.

Pour autant, la parallélisation du calcul de champs chimique sur GPU est réalisée de la même façon qu'a été réalisée, de façon plus théorique, celle des champs physiques. Mais à la différence des champs physiques, il peut cohabiter dans la simulation un nombre important de champs chimiques, chacun étant généré par des cellules biologiques différentes. Pour le moment, la mise à jour des différents champs est séquentielle : chaque champ est mis à jour l'un après l'autre. Nous reviendrons plus tard sur ce point précis dans les perspectives.

La mise à jour d'un champ est donc parallélisée pour chaque cellule de l'espace, un thread se chargeant de calculer la valeur du champ pour cette cellule. Le Code 5-2 est clairement très proche de celui étudié précédemment pour les champs physiques, il commence par envoyer les données sur le GPU, puis, après avoir calculé la taille de la grille et des blocs en fonction du nombre de threads nécessaires (le détail de cette fonction est présent dans le Code 5-3), appelle le kernel avant de récupérer le résultat de la simulation.

```
// Envoie de la mémoire sur le GPU
VolumGPU    champGPU    = envoiChampSurGPU( inChamp );

calculDimBlocGrille( dimBloc, dimGrille, maxX * maxY * maxZ );

// Appel du kernel avec 1 thread par cellule de l'espace
ComputeSemaphorinField_GPU<<< dimGrille, dimBloc >>>(
                                            tabCellulesBiologiques,
                                            inNbCellBiologiques,
                                            champGPU.fieldValue_,
                                            champGPU.fieldVector_,
                                            ...
                                                );

// Récupération de la mémoire depuis le CPU
recupChampDepuisGPU( inChamp );
```

Code 5-2 : Fragment de l'implémentation du calcul de champs sur GPU dans la PGMS.

```
void calculDimBlocGrille( dim3 & inoutDimBloc,
                          dim3 & inoutDimGrille,
                          const int inTaille,
                          const int inNbThreadMax = 256,
                          const int inNbBlocMaxParDim = 65535 )
{
   int nbBloc;

   // Affectation de la taille d'un bloc
   inoutDimBloc.x    = inNbThreadMax;
   // Calcul du nombre de blocs nécessaires
   nbBloc            = inTaille / inNbThreadMax
             + static_cast< int >( inTaille % inNbThreadMax != 0 );

   // Calcul de la taille de la grille en fonction du nombre de blocs
   // nécessaires et de la taille d'un bloc
   if ( nbBloc <= inNbBlocMaxParDim )
   {
      inoutDimGrille.x  = nbBloc;
   }
   else
   {
      inoutDimGrille.x  = inNbBlocMaxParDim;
      inoutDimGrille.y  = nbBloc / inNbBlocMaxParDim
            + static_cast< int >( nbBloc % inNbBlocMaxParDim != 0 );
   }
}
```

Code 5-3 : Calcul de la dimension des blocs et de la grille.

Le kernel est également très proche de ce qui a été fait précédemment (voir Code 5-4). L'identifiant du thread est tout d'abord calculé. Puis, pour chaque cellule biologique, la valeur du champ et du gradient est ajoutée à la valeur existante.

```
void ComputeSemaphorinField_GPU(
                              float *  inCoordBioCell,
                        const int      inNbBioCell,
                              float *  inFieldValue,
                              float *  inFieldVector,
                              ...
                                    )
{
   const int      indiceGlobal   = calculerIdGlobal();

   for ( unsigned int i = 0; i < inNbBioCell; i++ )
   {
      float   bioCellPos[ 3 ];
      bioCellPos[ 0 ]  = inCoordBioCell[ i * 3 ];
      bioCellPos[ 1 ]  = inCoordBioCell[ i * 3 + 1 ];
      bioCellPos[ 2 ]  = inCoordBioCell[ i * 3 + 2 ];

      // Calcul de la distance en fonction de bioCellPos et
      // la cellule courante
      float   distance  = ...;

      // Calcul de la valeur du champ
      inFieldValue  += inStrengthCoeff * expf( -inK * distance );

      if ( distance > 0.f )
      {
         // Mise à jour du vecteur de champ
         inFieldVector[ 0 ]  += ...;
         inFieldVector[ 1 ]  += ...;
         inFieldVector[ 2 ]  += ...;
      }
   }
}
```

Code 5-4 : Extraits de l'implémentation du kernel ComputeSemaphorinField_GPU.

Et bien entendu, il est donc possible d'implémenter l'optimisation permettant de partager des données auxquelles tous les threads accèdent en mémoire partagée. Exactement de la même façon que pour les particules dans le cas du champ de gravité, les coordonnées des cellules biologiques peuvent être stockées puis accédées en mémoire partagée. De cette façon, les accès mémoire sont minimisés et le rapport temps de calcul sur temps d'accès à la mémoire minimisé.

Afin de diminuer les accès mémoire, il est également possible d'utiliser une variable locale au kernel sur laquelle est calculé le champ de la cellule de l'espace traitée (valeur et gradient). Une fois toutes les cellules biologiques parcourues, le calcul de la valeur du champ est donc terminé et il est possible d'ajouter en mémoire globale le résultat au champ déjà présent pour cette cellule. Un seul accès à la mémoire globale par variable est donc effectué alors qu'il en fallait un par cellule biologique précédemment. La contrepartie de cette solution est l'augmentation du nombre de variables locales à un kernel. Les registres

permettant le stockage efficace des variables locales étant en nombre très limité, cela peut impacter (indirectement via le mécanisme d'ordonnancement) très fortement les performances de l'application. L'utilisation de quatre variables de plus (une pour la valeur du champ et trois pour le vecteur champ) n'est donc pas aussi anodine qu'elle le serait pour un programme s'exécutant sur un CPU.

Un point intéressant est que le calcul des champs dans la PGMS est répété à chaque itération. Il ne serait donc pas opportun d'allouer la mémoire, de copier les valeurs courantes du champ depuis le CPU vers le GPU et de libérer l'ensemble de la mémoire à chaque itération. En effet, ces trois opérations n'ont pas à être effectuées plus d'une fois dans toute la vie de l'application tant que le GPU dispose de suffisamment de mémoire. La solution consiste donc, lors du premier appel d'un calcul sur un champ donné, à allouer la mémoire nécessaire sur le GPU, à associer cette mémoire au champ correspondant et à copier l'état courant du champ (il s'agit généralement d'une initialisation à zéro) (voir Code 5-5). Lors des appels suivants, la mémoire stockée sur le GPU, correspondant au champ, est retrouvée et les calculs sont effectués sur celle-ci sans surcoût supplémentaire. Enfin, à la fin de l'exécution de la simulation, la mémoire peut être désallouée une seule fois.

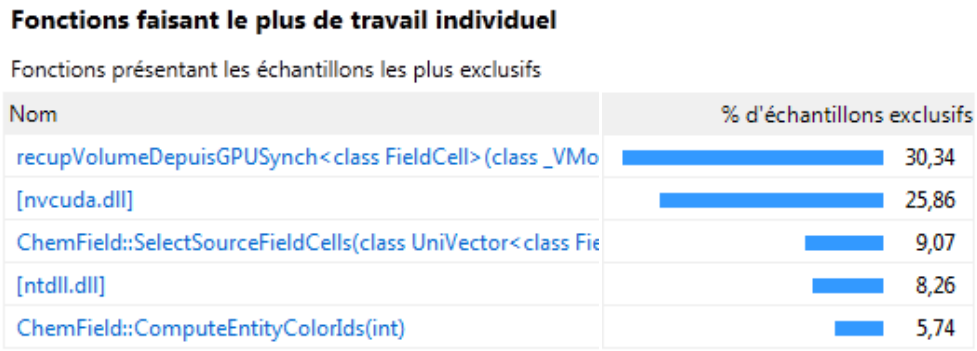
```
// Allocation memoire et copies
if ( ! estDejaCharge( inChamp ) )
{
   envoiVolumeSurGPU( inChamp );
}

// Recuperation de la memoire sur le GPU correspondant au champ
ChampGPU    champGPU    = mapChamp[ inChamp->GetAdresseMemoire() ];
```

Code 5-5 : Extrait du code de la PGMS permettant, pour un champ donné, l'allocation, la copie et la récupération de la mémoire stockée sur le GPU.

5.3.2 Résultat du calcul des champs physicochimiques à l'aide des GPU

La parallélisation sur GPU consiste à réaliser tous les calculs de champs sur GPU. La Figure 5-3 montre le résumé du profilage de l'application une fois les modifications effectuées. Une majorité du temps est toujours passée à traiter le calcul de champs (`recupVolumeDepuisGPUSynch` et `nvcuda.dll` correspondent à des appels liés à la parallélisation). Lorsque l'on prend également en compte les appels internes, c'est toujours plus de 77% du temps d'exécution qui est occupé par le calcul sur GPU. Pour autant, le temps d'exécution n'en est pas moins fortement diminué, passant de 132 secondes en moyenne à moins de 11,3 secondes en moyenne, soit un facteur d'accélération de 11,7x.

Pour ce qui est des nouveaux appels de fonctions fortement consommateurs de temps de calcul (voir Figure 5-3), on s'aperçoit que les fonctions qui n'étaient pas significatives précédemment prennent maintenant une place beaucoup plus prépondérantes. Ainsi dans la version GPU, la fonction `SelectSourceFieldCells` compte comme la 3ème plus importante fonction en termes de temps de calcul cumulé (9,07%). Dans la version basique, elle est reléguée très loin dans le classement et ne représente que 0,47% du temps de calcul pour un travail équivalent (voir Figure 5-4).



Fonctions faisant le plus de travail individuel

Fonctions présentant les échantillons les plus exclusifs

Nom	% d'échantillons exclusifs
recupVolumeDepuisGPUSynch<class FieldCell>(class _VMo	30,34
[nvcuda.dll]	25,86
ChemField::SelectSourceFieldCells(class UniVector<class Fie	9,07
[ntdll.dll]	8,26
ChemField::ComputeEntityColorIds(int)	5,74

Figure 5-3 : Résumé du profilage de l'application PGMS parallélisé sur GPU.

Nom de la fonction	% d'échantillons i...	% d'échantillons e...
__set_statfp	1,15	1,15
__checkTOS_withFB	0,98	0,98
__CIexp	0,79	0,79
__math_exit	6,56	0,63
ChemField::SelectSourceFieldCells(class UniVector<class Fiel	0,47	0,47

Figure 5-4 : Extrait des appels de fonction de la PGMS de base.

Pour analyser plus en détail ces résultats, un profiler pouvant interpréter les appels à l'API CUDA, permet de fournir davantage d'informations. Son utilisation nous apprend que le temps passé à réaliser les calculs (appel de fonctions –kernels– sur le GPU) ne représente que 3,1% du temps d'exécution total (voir Figure 5-5) lorsque la copie de données entre le GPU et le CPU représente 20,6% (voir Figure 5-6). C'est donc la copie qui prend la majeure partie du temps de l'utilisation de l'API CUDA (`nvcuda.dll` dans le tableau de la Figure 5-3).

3,1 % [2881] ComputeSemaphorinField_GPU(...)	
Name	Value
Duration	
Session	11,382 s
Timeline	11,328 s
Kernel	347,884 ms
Utilization	3,1 %
Invocations	2881

Figure 5-5 : Détail du temps d'exécution du kernel sur le GPU

MemCpy (DtoH)	
Name	Value
Duration	
Session	11,382 s
Timeline	11,329 s
Memcpy	2,348 s
Invocations	5762
Total Bytes	9,659 GB

Figure 5-6 : Détail du temps d'exécution de la copie mémoire entre le GPU et le CPU.

Mais la fonction `recupVolumeDepuisGPUSynch`, sans prendre en compte la copie du GPU vers le CPU via l'API CUDA correspondant aux deux fonctions `recuperationMemoire` dans la Figure 5-7, prend elle aussi beaucoup de temps (plus de 30%). Ceci est dû au fait que la copie de la mémoire ne se fait pas directement vers les objets utilisés par le code séquentiel mais dans des tableaux temporaires dont les valeurs doivent par la suite être recopiées (partie « Organisation de la mémoire » dans le code suivant).

```
                template < typename T >
               void recupVolumeDepuisGPUSynch( _VModel< T > * inVolumModel )
                {
                    ...

                    // Recuperation de la memoire
    5,5 %           recuperationMemoire( fieldValue_CPU, volumGPU.fieldValue_, nbCellGrid );
   13,1 %           recuperationMemoire( fieldVector_CPU, volumGPU.fieldVector_, nbCellGrid * 3 );

                    // Organisation de la memoire
                    int     i   = 0;
  < 0,1 %           for ( int z = 0; z < maxZ; ++z )
                    {
    0,2 %              for ( int y = 0; y < maxY; ++y )
                       {
  < 0,1 %                 for ( int x = 0; x < maxX; ++x, ++i )
                          {
    2,2 %                    FieldCell *     curr_field_cell     = &( volume[ z ][ y ][ x ] );

    1,6 %                    curr_field_cell->FieldValue         = fieldValue_CPU[ i ];
    0,7 %                    curr_field_cell->FieldVector.x      = fieldVector_CPU[ i * 3     ];
    0,1 %                    curr_field_cell->FieldVector.y      = fieldVector_CPU[ i * 3 + 1 ];
   25,4 %                    curr_field_cell->FieldVector.z      = fieldVector_CPU[ i * 3 + 2 ];
                          }
                       }
                    }

                    ...

                }
```

Figure 5-7 : Extrait de la méthode recupVolumeDepuisGPUSynch.

Or, la raison pour laquelle la recopie est si longue n'est pas seulement qu'il est nécessaire de réaliser beaucoup de copies. Elle est aussi liée à la structure de données utilisées. C'est la raison pour laquelle un travail a été effectué pour améliorer celle-ci.

La parallélisation sur GPGPU des phases les plus gourmandes en temps de calcul (le calcul des champs chimiques dans notre cas) permet d'obtenir des gains très importants. Pour autant, d'autres modifications plus fines du code existant permettent d'améliorer sensiblement les performances de l'application.

5.4 Optimisations du code

Les optimisations présentées jusqu'ici ont été de deux formes. Dans le cas de l'algorithme Hash-Life, il s'agissait de remplacer l'algorithme existant par un nouvel algorithme plus efficace, ayant le même but (la simulation d'automates cellulaires). Dans le cas de l'implémentation sur GPU, il s'agit de tirer parti d'une plus grande puissance de calcul afin de

réaliser les calculs de façon plus rapide. Pour cela, l'utilisation d'une architecture parallèle, dans notre cas à base de GPU, a été nécessaire. Cependant, toutes les améliorations ne nécessitent pas un remplacement de l'algorithme ou l'utilisation d'architectures parallèles. Il est en effet également possible d'améliorer l'implémentation des algorithmes séquentiels existants.

Dans notre contexte, il est possible d'obtenir des gains significatifs en travaillant sur les éléments du programme qui sont les plus gourmands en temps de calcul. La difficulté d'un tel travail est de bien savoir mesurer le rapport entre le gain offert par une optimisation et le coût engendré par celle-ci sur la qualité logicielle afin de valider ou de refuser l'optimisation. Par exemple, l'implémentation en assembleur d'une partie critique de code peut être acceptable si les gains sont très significatifs. Mais s'il s'agit de gagner 5% de temps d'exécution sur cette fonction et de disposer par la suite d'un code très difficile à maintenir et à comprendre, l'opération ne sera pas rentable sur le long terme.

Dans le cas de la simulation d'un morceau de rein à l'aide de la PGMS et comme cela a été dit plus tôt, c'est le calcul de champs qui représente la partie la plus importante du temps de calcul global. Si l'optimisation pour effectuer plus rapidement les calculs sur GPU est efficace, certaines optimisations peuvent permettre de réduire significativement le temps d'exécution de l'application en séquentielle. Ainsi, la Figure 5-8 illustre les importants temps d'accès aux éléments du champ dans une fonction de calcul de champ. Pour améliorer ces performances, deux solutions ont été proposées, la première est de stocker de façon contigüe les tableaux et la deuxième est d'accéder aux éléments du tableau dans leur ordre de stockage pour maximiser l'utilisation des caches.

```
< 0,1 %     for ( i=0; i<imax; i++ )
              for ( j=0; j<jmax; j++ )
0,2 %             for ( k=0; k<kmax; k++ )
                  {
0,2 %                 FieldCell* curr_field_cell = &(fieldMatrix[k][j][i]);

13,2 %                fieldcell_wp       = curr_field_cell->CorrespMatterCell->WorldPos;
```

Figure 5-8 : Extrait du profilage du code du calcul de champs pour la PGMS de base.

5.4.1 Stockage contigu en mémoire des tableaux.

Le premier point négatif de l'accès aux cellules de l'espace 3D était la structure de données utilisée pour stocker les cellules : `Array3D`.

Existant

Une classe générique comprenait un tableau de pointeur (`data3D_`) vers des instances de type `Array2D` (voir Code 5-6).

```
template <class _Element> class Array3D
{
   public:
      Array3D( int inSz = 0, int inSy = 0, int inSx = 0 );
      [...] // Reste de l'interface publique

   private :
      int                     sizeZ_;  // Dimensions
      int                     sizeY_;  // du
      int                     sizeX_;  // volume
      Array2D< _Element > **  data3D_; // Zone d'allocation
};
```

Code 5-6 : Extrait de la déclaration originale de la classe Array3D.

Le constructeur allouait donc dans un premier temps un tableau de pointeur de taille `inSz`. Puis, dans un second temps, le constructeur instancie un objet de type `Array2D` pour chaque valeur de z valide (voir Code 5-7).

```
Array3D( int inSz = 0, int inSy = 0, int inSx = 0 )
   : sizeZ_( inSz ), sizeY_( inSy ), sizeX_( inSx )
{
   data3D_  = new Array2D< _Element > *[ inSz ];

   for( int z = 0; z < sizeZ_; z++ )
   {
      data3D_[ z ]  = new Array2D< _Element >( sizeY_, sizeX_ );
   }
}
```

Code 5-7 : Extrait de l'implémentation originale du constructeur de la classe Array3D.

`Array2D` est également une classe générique, elle contient un tableau dynamique d'objet du type paramètre de la classe (`data2D_`) (voir Code 5-8).

```
template <class _Element> class Array2D
{
   public:
      Array2D( int inSz = 0, int inSy = 0 );
      [...] // Reste de l'interface publique

   private :
      int         sizeY_;  // Dimensions
      int         sizeX_;  // du volume
      _Element *  data2D_; // Zone d'allocation
};
```

Code 5-8 : Extrait de la déclaration originale de la classe Array2D.

L'allocation du tableau est simplement réalisée pour correspondre à la taille souhaitée (voir Code 5-9).

```
Array2D( int inSy = 0, int inSx = 0 )
   : sizeY_( inSy ), sizeX_( inSx )
{
   data2D_  = new _Element[ inSy * inSx ];
}
```

Code 5-9 : Extrait de l'implémentation originale du constructeur de la classe Array2D.

Cette implémentation a l'avantage de bien différencier le tableau à trois dimensions des tableaux à deux dimensions qui le compose. Même si cela peut s'avérer utile par moment, cette implémentation de la classe `Array3D` peut induire de forte diminution des performances. La raison est que la mémoire n'est pas allouée de façon contigüe en mémoire. La Figure 5-9 montre bien que l'on va traiter plusieurs blocs de mémoire alloués à divers endroit de la mémoire. D'une part, le tableau de pointeur `Data3D_` est alloué en premier de manière isolée (de taille z=6 dans notre exemple) puis chaque instance d'`Array2D` est alloué séparément (de taille 8 –x=2 et y=4– dans notre exemple). Aucune garantie n'est donc donnée quant au fait que ces sept tableaux soient alloués de manière contigüe en mémoire.

Figure 5-9 : Ancienne représentation de la mémoire d'une instance de Array3D (x=2, y=4 et z = 6).

Cette approche souple est cependant pénalisante sur les architectures récentes des microprocesseurs où la présence de mécanismes de cache permet d'accélérer les accès à la mémoire lorsque ceux-ci sont faits sur des données consécutives en mémoire (Denning 2005). Le principe utilisé par les microprocesseurs modernes est le suivant, lors d'un accès à une valeur en mémoire, le gestionnaire de cache ne va pas récupérer uniquement cette

valeur mais également toute une partie des valeurs suivantes. Ainsi, en cas d'accès successifs à des données contigües en mémoire, ceux-ci vont pouvoir être réalisés de façon beaucoup plus rapide.

L'implémentation décrite précédemment ne tire pas la pleine mesure des mécanismes modernes de gestion des mémoires caches. En effet, si les accès successifs au sein d'un même bloc de données à deux dimensions (`Array2D`) sont bien contigus, il n'en est pas de même pour les accès au sein d'un `Array3D` lorsque la dimension z évolue.

Solution proposée

Pour remédier à ce problème, la solution proposée est d'allouer en un seul bloc toute la mémoire nécessaire au tableau à trois dimensions. La classe Array3D ne comprend plus alors qu'un tableau dynamique de l'élément générique (voir Code 5-10 et Code 5-11).

```
template < class _Element >
class Array3D
{
   public :
      Array3D( int inSz = 0, int inSy = 0, int inSx = 0 );
      [...] // Reste de l'interface publique

   private :
      int          sizeZ_;  // Dimensions
      int          sizeY_;  // du
      int          sizeX_;  // volume

      _Element *   data3D_;  // Zone d'allocation
};
```

Code 5-10 : Extrait de la nouvelle déclaration de la classe Array3D.

```
Array3D( int inSz = 0, int inSy = 0, int inSx = 0 )
   : sizeZ_( inSz ), sizeY_( inSy ), sizeX_( inSx )
{
   data3D_  = new _Element[ inSz * inSy * inSx ];
}
```

Code 5-11 : Extrait de la nouvelle implémentation du constructeur de la classe Array3D.

De cette façon, l'ensemble de la mémoire est alloué de façon contigüe (voir Figure 5-10).

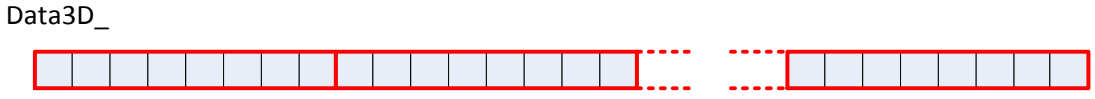


Figure 5-10 : Nouvelle représentation de la mémoire d'une instance de Array3D (x=2, y=4 et z = 6).

Si les accès sont plus rapides, un inconvénient qui peut sembler important est l'interface publique de la classe. Précédemment, pour accéder à une ligne ou un élément d'un Array3D, il suffisait d'appeler une ou plusieurs fois l'opérateur « [] » (voir Code 5-12).

```
Array3D< int >    espace( 6, 4, 2 );  // ( z, y, x )

espace[ 1 ][ 0 ][ 1 ]      = 28;      // [ z ][ y ][ x ]
int *    ligne    = espace[ 3 ][ 0 ]; // [ z ][ y ]
```

Code 5-12 : Méthodes d'accès originales aux éléments d'une instance de Array3D.

Sans passer par une classe proxy qui rajouterait de la complexité, il n'est plus possible d'utiliser l'opérateur « [] » aussi simplement. Deux méthodes ont donc dû être rajoutées à la classe : `getLigne` et `getElement`. Ces méthodes permettent de réaliser exactement les mêmes opérations que précédemment (voir Code 5-13).

```
Array3D< int >  espace( 6, 4, 2 );           // ( z, y, x )

espace.getElement( 1, 0, 1 )  = 28;          // ( z, y, x )
int *    ligne    = espace.getLigne( 3, 0 ); // ( z, y )
```

Code 5-13 : Nouvelles méthodes d'accès aux éléments d'une instance de Array3D.

Comme cela est bien visible lorsque l'on compare ces deux classes, hormis l'utilisation de méthodes classiques plutôt que la surcharge d'opérateur, le code est parfaitement semblable. L'implémentation des deux méthodes est quant à elle très simple. La différence majeure est qu'il est nécessaire de faire davantage d'arithmétique de pointeurs manuellement à l'intérieur de ces méthodes (voir Code 5-14).

```
template <class _Element>
inline
_Element * Array3D< _Element >::getLigne( const int inZ,
                                           const int inY )
{
   return data3D_ + ( ( inZ * sizeY_ ) + inY ) * sizeX_;
}

template <class _Element>
inline
_Element & Array3D< _Element >::getElement( const int inZ,
                                            const int inY,
                                            const int inX )
{
   return data3D_[ ( ( ( inZ * sizeY_ ) + inY ) * sizeX_ ) + inX ];
}
```

Code 5-14 : Implémentation des nouvelles méthodes d'accès aux éléments de la classe Array3D.

Cette solution n'est pas la seule envisageable pour résoudre ce problème. S'il avait paru nécessaire de conserver une structure utilisant des `Array2D` afin de conserver le mécanisme de l'opérateur « [] », il aurait été possible d'implémenter une solution différente conservant ces caractéristiques. Il aurait fallu, lors de l'appel de l'opérateur « [] » sur un `Array3D`, créer un proxy `Array2D` avec en paramètre l'adresse du premier élément du tableau qui lui correspond. Le comportement d'Array2D aurait alors pu rester semblable à celui qui était le sien auparavant.

Le choix a été fait de ne pas implémenter cette solution car Array2D n'avait qu'un rôle très mineur dans l'application. Il permettait principalement d'implémenter le mécanisme de l'opérateur « [] ». Ce rôle semble bien léger pour justifier la conservation d'une classe entière. Le nouveau code s'en trouve donc simplifié. L'appel aux méthodes `getLigne` et `getElement` est par contre moins proche de l'utilisation d'un tableau classique avec un opérateur « [] ». À l'inverse, l'interface de la classe est maintenant plus explicite.

Résultat

Le stockage contigu consiste à allouer l'ensemble des espaces 3D de façon contigu en mémoire sous la forme d'un tableau à une seule dimension. Les gains sont alors de l'ordre de 7 secondes pour une simulation qui prend 126 secondes, soit un gain de l'ordre de 5% (facteur d'accélération de 1,05x). Cela peut paraitre un gain assez maigre en comparaison du gain de 96% de l'implémentation sur GPU mais cette optimisation permet surtout l'utilisation très efficace d'un accès séquentiel présenté par la suite.

Toujours dans le cas de la même fonction, la Figure 5-11 permet de rendre compte d'un gain lors de l'accès aux données des tableaux. Ainsi, le pourcentage de temps nécessaire, dans cette fonction, à l'accès à la position de la cellule passe de 13,2% à 11,6% (le gain global de l'ordre de 5% s'explique par un impact non limité à cette seule fonction).

```
  0,3 %                 FieldCell* curr_field_cell = &(fieldMatrix.getElement( k, j, i ) );

 11,6 %                 fieldcell_wp       = curr_field_cell->CorrespMatterCell->WorldPos;
```

Figure 5-11 : Extrait du profilage du code du calcul de champs pour la PGMS avec stockage contigu.

5.4.2 Accès séquentiel aux tableaux.

Comme nous l'avons vu précédemment, les champs chimiques sont recalculés à chaque itération. Pour cela, un parcours complet du tableau de champ est nécessaire. Or si celui-ci

n'est pas fait de façon optimale, cela peut-être plus coûteux en temps de calcul que nécessaire. La mise à jour des champs étant une part importante du temps de calcul global de l'application, le gain sur une partie locale de l'algorithme peut être significatif au niveau du temps global d'exécution de l'application.

Existant

Ce parcours était originalement réalisé de la même façon dans plusieurs zones du programme. Le code était systématiquement le suivant celui du Code 5-15.

```
for ( int x = 0; x < xMax; ++x )
{
   for ( int y = 0; y < yMax; ++y )
   {
      for ( int z = 0; z < zMax; ++z )
      {
         Cell     cellule     = espace[ z ][ y ][ x ];
         [...] // Traitement
      }
   }
}
```

Code 5-15 : Ancienne méthode de parcours d'un tableau 3D.

Le problème de cette approche vient de l'ordre d'imbrication des boucles. Mais pour bien appréhender le problème, il est nécessaire de se souvenir que les tableaux à trois dimensions sont stockés sous forme de tableaux à une dimension. Ainsi, le tableau de 5x3x4 de la Figure 5-12 sera simplement stocké à l'aide d'un tableau à une dimension de taille 60 comme cela est schématisé dans la Figure 5-13.

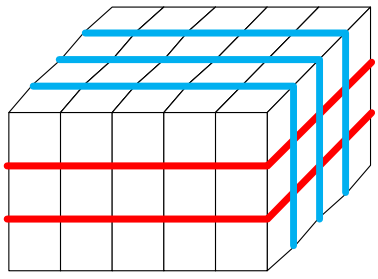


Figure 5-12 : Représentation 3D d'un tableau 3D de 5x3x4.

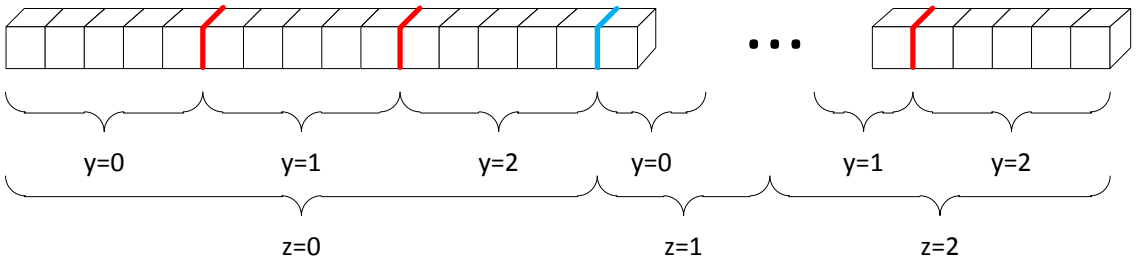


Figure 5-13 : Représentation 1D d'un tableau 3D de 5x3x4.

Dans l'exemple du Code 5-15 (qui utilise volontairement l'interface de la classe Array3D originale), on s'aperçoit que c'est l'indice z qui va évoluer le plus souvent étant donné qu'il s'agit du dernier imbriqué. Or la figure précédente montre bien que les tableaux sont stockés ligne par ligne puis coupe par coupe. Les cellules possédant une même valeur pour x et y mais une valeur de z différente sont donc éloignées en mémoire les unes des autres d'autant de cellules qu'il y en a dans une coupe en z (dans la figure précédente, elles sont donc éloignées de 15 cellules). Une nouvelle fois, sans mécanismes de cache, cela n'aurait que peu d'influence sur la simulation. Mais à l'aide des mécanismes de cache, accéder à des données stockées se succédant en mémoire est bien plus efficace (10 fois plus rapide qu'un accès complètement aléatoire[18]) (Kennedy et Allen 2001).

Étant donné que la simplification utilisée pour calculer la valeur du champ chimique dans une cellule rend le calcul de la valeur du champ en un point complètement indépendant (voir Code 5-16), le parcours peut s'effectuer dans n'importe quel ordre. Il est donc avantageux de réaliser ce calcul dans l'ordre permettant les meilleures performances.

Solution proposée

Dans notre cas, cela correspond à parcourir les cellules dans l'ordre dans lequel elles sont stockées en mémoire. Pour cela, l'ordre des boucles est inversé :

[18] http://www.sisoftware.net/?d=qa&f=ben_mem_latency : Tests de performance réalisés par la société SiSoftware afin de mesurer la latence des accès mémoire à travers un cache. Dernier accès le 24/02/2012.

```
for ( int z = 0; z < zMax; ++z )
{
   for ( int y = 0; y < yMax; ++y )
   {
      for ( int x = 0; x < xMax; ++x )
      {
         Cell     cellule     = espace[ z ][ y ][ x ];
         [...] // Traitement
      }
   }
}
```

Code 5-16 : Nouvelle méthode de parcours d'un tableau 3D.

Ainsi, les cellules seront bien accédées ligne par ligne et coupe par coupe (dans le même ordre qu'elles sont stockées en mémoire) de façon à pouvoir tirer parti du mécanisme de cache.

Optimisation supplémentaire au parcours de tableau

Dans ce même code, une autre optimisation est possible. En effet, la copie de la cellule n'est pas nécessaire. Il est ainsi possible d'accéder à l'élément « par référence » (voir Code 5-17) ou « par adresse » sans modifier le comportement du code. Ainsi, seule l'adresse de la cellule est manipulée et le coût de l'appel du constructeur de copie est complètement éliminé.

```
         Cell &   cellule     = espace[ z ][ y ][ x ];
```

Code 5-17 : Récupération de la cellule par référence.

Mais le fait d'accéder aux données de manière séquentielle en mémoire permet également une autre forme d'optimisation à travers l'utilisation d'arithmétique de pointeur pour le parcours du tableau. Ainsi, au lieu d'accéder aux données comme cela a été fait précédemment, il est possible de manipuler un pointeur vers la cellule courante que l'on avance à chaque itération (voir Code 5-18).

```
// Récupération de l'adresse mémoire de la première cellule
Cell *      ptrCelluleCourante    = &( espace[ 0 ][ 0 ][ 0 ] );

for ( int z = 0; z < zMax; ++z )
{
   for ( int y = 0; y < yMax; ++y )
   {
      for ( int x = 0; x < xMax; ++x )
      {
         // Utilisation de ptrCelluleCourante pour le traitement
         [...] // Traitement

         // Incrémentation de ptrCelluleCourante
         ++ptrCelluleCourante;
      }
   }
}
```

Code 5-18 : Nouvelle méthode de parcours d'un tableau 3D utilisant l'arithmétique de pointeur.

Résultat

L'accès séquentiel aux données consiste à accéder à celles-ci dans l'ordre dans lequel elles sont stockées autant que possible. L'implémentation de cette fonctionnalité couplée à celle du stockage contigu (sans quoi le gain à travers l'utilisation de cache est bien moindre) permet de diminuer le temps de calcul en séquentiel de plus de 23% (correspondant à un facteur d'accélération de 1,31x), faisant passer le temps d'exécution de 132 secondes à 101 secondes. L'accès aux données lors du même calcul de champs en séquentiel pris en exemple précédemment ne représente plus que 8,3% de ce qu'il représentait initialement (1,1% contre 13,2%) (voir-ci-dessous la Figure 5-14 par rapport au pourcentage de base de 13,2% sur la Figure 5-8).

```
  0,3 %     FieldCell* curr_field_cell = &(fieldMatrix.getElement( k, j, i ) );

  1,1 %     fieldcell_wp       = curr_field_cell->CorrespMatterCell->WorldPos;
```

Figure 5-14 : Extrait du profilage du code du calcul de champs pour la PGMS avec stockage contigu et accès séquentiel.

5.4.3 Implémentation sous forme de tableau des vecteurs mathématiques

Une grande partie des données sont stockées dans l'application à l'aide de vecteurs mathématiques à deux et trois composantes (respectivement x et y, et x, y et z).

Existant

Ceux-ci étaient implémentés à l'aide de deux ou trois attributs (selon qu'il s'agissait de vecteurs de taille deux ou trois). Cette implémentation posait un vrai problème lors de

l'appel de l'opérateur [] pour ces deux classes. En effet, comme le montre la Figure 5-15, il était nécessaire de réaliser plusieurs tests pour renvoyer le bon attribut. Or cette fonction étant très souvent appelée, la succession de ces tests représentait un vrai poids pour l'application.

```
          template <class Type>
          inline Type& TPnt3D<Type>::operator[] ( int index )
  0,4 %   {
  0,5 %     if( index==_X )
                return(x);
            else
            {
< 0,1 %        if(index==_Y)
                  return(y);
               else
< 0,1 %           return(z);
            }
  0,3 %   }
```

Figure 5-15 : Implémentation basique de l'opérateur [] pour la classe TPnt3D.

Proposition

Leur implémentation sous forme de tableau (respectivement un tableau de 2 et 3 éléments) permet de supprimer l'ensemble des tests réalisés par l'opérateur [] de ces deux classes.

Le remplacement de ce code par un simple retour du tableau contenant les composantes à l'indice `index` permet ainsi de simplifier le code de cette fonction (voir Figure 5-16). Mais l'introduction des méthodes `getX`, `getY` et `getZ` en lieu et place de `[0]`, `[1]` et `[2]` (et leurs équivalents `setX`, `setY` et `setZ`) (voir Figure 5-17) accélère également le calcul, favorise l'intégration directe du code de la fonction au code appelant (inlining) et facilite ainsi les optimisations du compilateur en lui fournissant plus d'informations.

```
template <class Type>
inline const Type& TPnt3D<Type>::operator[] ( int index ) const
{
   return coordinates_[ index ];
}
```

Figure 5-16 : Implémentation optimisée de l'opérateur [] pour la classe TPnt3D.

```
Type getX() const { return coordinates_[ 0 ]; }
Type getY() const { return coordinates_[ 1 ]; }
Type getZ() const { return coordinates_[ 2 ]; }

void setX( const Type inX ) { coordinates_[ 0 ] = inX; }
void setY( const Type inY ) { coordinates_[ 1 ] = inY; }
void setZ( const Type inZ ) { coordinates_[ 2 ] = inZ; }
```

Figure 5-17 : Implémentation des getters et des setters pour la classe TPnt3D.

Résultat

Le résultat est un code beaucoup plus rapide réalisant la simulation en 112 secondes en moyenne, soit un gain de plus de 15% sur le temps de calcul initial (132 secondes).

Mais ces optimisations sont particulièrement utiles car elles sont également efficaces dans le cas de l'implémentation GPU. Il est donc possible et même conseillé de les coupler toutes ensemble.

5.4.4 Optimisations couplées à l'implémentation GPU

Lorsque l'on couple les trois optimisations à la parallélisation du GPU, le temps moyen d'exécution de la simulation passe à 10,4 secondes en moyenne contre 11,3 secondes sans optimisation. Le gain par rapport à l'implémentation de base est donc de plus de 92%, soit un facteur d'accélération supérieur à 12,5x entre les deux temps. Le gain entre l'implémentation utilisant le GPU avec les optimisations et celle ne les utilisant pas est donc égal à 7,2%.

Par rapport à la version n'incorporant que la parallélisation sur GPU, cette solution accélère principalement les calculs réalisés sur le CPU. Il est donc logique que le pourcentage de temps passé à réaliser des calculs liés à la parallélisation sur GPU ait augmenté (voir Figure 5-18) : +13% pour `recupVolumeDepuisGPUSynch` et +18% pour les appels à l'API CUDA (nvcuda.dll).

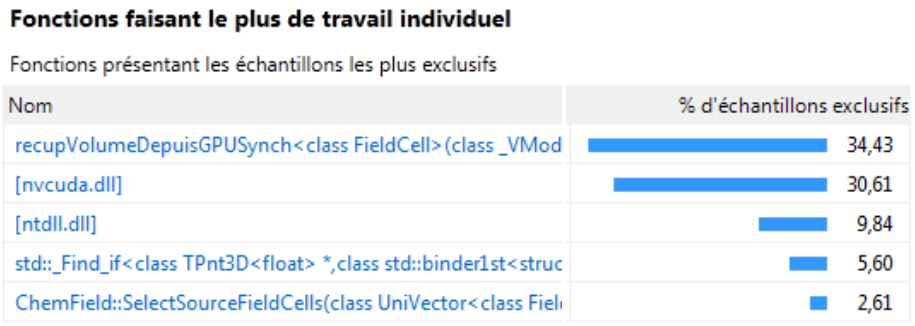


Figure 5-18 : Résumé du profilage de l'application PGMS optimisée et parallélisée sur GPU

5.4.5 Autres optimisations

Hormis les calculs à proprement parler, il est de plus en plus visible que l'allocation de mémoire devient pénalisante pour l'application (la bibliothèque ntdll.dll se charge entre autre choses de l'allocation mémoire –voir Figure 5-18). Or l'allocation mémoire est connue pour être gourmande en temps de calcul (Fog 2008) (Detlefs, Dosser et Zorn 1994).

En conservant d'un appel sur l'autre les tableaux alloués en RAM et utilisés lors de la parallélisation sur GPU, il est possible de diminuer cet impact lié à l'allocation dynamique. Cette optimisation a pour conséquence d'augmenter l'empreinte mémoire de l'application. Mais pour un espace de 225 000 cellules, le stockage de dix champs ne prend que 9Mo. Ce qui est négligeable au vu de la quantité de mémoire RAM disponible actuellement. À l'inverse, cette optimisation permet de ramener le temps de calcul global de l'application à 8,6 secondes lorsque toutes les optimisations et la parallélisation sont appliquées (gain de 16,5% par rapport à cette même application sans cette dernière optimisation).

La mise à jour des champs n'est pas pour autant la seule opération consommant du temps de calcul de façon importante. Ainsi, lors de l'initialisation, certaines étapes consomment une part importante du temps de calcul. La difficulté, lors de l'optimisation de ces parties de codes est de savoir dans quelle mesure ces portions de code resteront les goulots d'étranglement des simulations de taille plus importantes ou si ces goulots sont propres à une simulation. Le travail qui a été réalisé en ce sens concerne l'implémentation d'un conteneur générique, non présent dans la bibliothèque standard, permettant de conserver l'ordre d'insertion mais ne permettant pas l'insertion multiple du même élément.

Afin d'implémenter ce conteneur, la version originale utilisait un vecteur de la bibliothèque standard du C++. L'insertion d'un élément était donc réalisée si et seulement si l'élément

n'était pas déjà présent dans le vecteur. La conséquence étant bien évidemment que le parcours du vecteur était réalisé à chaque insertion pour vérifier l'absence de l'élément. Pour remédier à ce problème, deux solutions ont été envisagées.

La première solution consiste à adjoindre au vecteur un set (qui est un conteneur trié garantissant l'unicité de ses éléments – il serait aussi possible d'utiliser un vecteur trié à la place). Ce set n'est pas utilisé pour le stockage à proprement parler, même s'il stocke également les éléments. Son rôle est de garantir l'unicité des éléments. Ainsi, lors de l'insertion d'un nouvel élément, le test d'unicité n'est plus fait à travers un parcours exhaustif du vecteur mais à travers une recherche dans le set. Le set étant trié, la recherche est beaucoup plus rapide. Bien entendu, cette solution n'est pas parfaite. Elle requiert tout d'abord davantage de mémoire pour le set qui duplique le stockage. Dans le cas d'éléments de tailles importantes, cela peut avoir une incidence. Dans le cas de la PGMS, la diminution du temps d'exécution prévaut sur l'empreinte mémoire du programme, ce défaut n'est donc pas critique. Le deuxième problème est la complexification du code. En effet, l'ajout d'un set est source d'erreur. Ainsi, dans l'implémentation de la classe, lorsque le vecteur est modifié (ajout ou suppression d'un élément, de plusieurs éléments…), il est nécessaire de toujours conserver le set à jour.

La deuxième solution est différente dans le sens où elle ne conserve pas les propriétés du conteneur. Après avoir analysé l'utilisation qui est faite du conteneur, il en est ressorti que la conservation de l'ordre d'insertion n'était pas nécessaire. À partir de ce constat, il a été possible de proposer une solution se passant complètement de cette classe pour n'utiliser qu'un set.

5.5 Étude des performances sur d'autres simulations

L'ensemble des tests de la PGMS présentés jusqu'ici ont été réalisés à partir d'une simulation permettant la morphogénèse d'un néphron. Cette simulation qui correspond à un usage réel de la PGMS a été utilisée comme référence lors des travaux d'optimisation et de parallélisation. Dans le même temps, une autre équipe de l'entreprise travaillait à la création de nouvelles simulations. Lorsque ces travaux furent terminés, il a été possible de fusionner les deux projets afin d'appliquer à tous les modèles disponibles les optimisations développées pour le premier modèle de référence. Comme cela a été expliqué

précédemment, l'idée directrice des choix d'optimisation a toujours été de disposer des améliorations les plus généralistes possibles afin qu'elles s'appliquent sans modifications aux nouveaux modèles. Les deux nouveaux modèles qui ont été développés récemment n'ont donc pas disposé de nouvelles optimisations propres à leurs exécutions, ils ont juste été exécutés via l'implémentation optimisée et parallélisée de la PGMS.

La parallélisation sur GPU étant activable ou désactivable à la compilation, trois mesures différentes existent pour la simulation de chaque modèle :

1. La version originale de la PGMS avant l'incorporation des optimisations.
2. La version optimisée de la PGMS avec la parallélisation sur GPU désactivée.
3. La version optimisée et parallélisée sur GPU de la PGMS.

Les temps de la simulation d'un néphron ne sont pas exactement les mêmes que ceux présentés précédemment. Ces écarts sont dus à la version de la PGMS utilisée. Dans le cas des résultats précédents qui vous ont été présentés, la version de PGMS utilisée ne permettait alors que de faire fonctionner la simulation d'un néphron simple. Dans le cas des résultats suivants, il s'agit de la première version de la PGMS optimisée et parallélisée pouvant simuler les trois types de modèles. Entre ces deux versions, des modifications ont été réalisées sur l'algorithme de la PGMS qui explique les différences de temps d'exécution.

Le Tableau 1 présente les temps d'exécution des trois simulations en fonction de la version de la PGMS utilisée. L'enseignement principal est l'ordre de grandeur des différentes simulations. Dans la version originale, la simulation d'un néphron simple (Single Nephron2) prend de l'ordre de 2 minutes lorsque la simulation de plusieurs néphrons (Renal Juxtamedullar Section) prend plus de 11 minutes et la simulation d'une pyramide rénale plus de 16 minutes. Les deux dernières simulations sont donc significativement plus importantes.

Temps moyen d'exécution des simulations de la PGMS			
	Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
Version originale	129,07s	671,00s	971,65s
Version optimisée	85,42s	468,86s	735,42s
Version optimisée et parallélisée sur GPU	9,13s	54,56s	53,63s

Tableau 1 : Temps d'exécution moyen des trois simulations de la PGMS testées.

Afin de comparer les temps d'exécution les uns aux autres, il est plus intéressant d'étudier le gain moyen et l'accélération des différentes versions pour chaque simulation par rapport à la version originale. Dans ce but, le Tableau 2 présente le facteur d'accélération des trois simulations en fonction de la version de la PGMS utilisée.

Accélération moyenne des simulations par rapport à la version originale de la PGMS			
	Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
Version optimisée	1,51x	1,43x	1,32x
Version optimisée et parallélisée sur GPU	14,14x	12,30x	18,12x

Tableau 2 : Accélération (speed-up) moyenne par rapport à la version originale de la PGMS pour les trois simulations.

Cela permet de mettre en valeur les facteurs d'accélération obtenus à l'aide des optimisations séquentielles (« Version optimisée » –première ligne– dans le tableau). En mettant en œuvre des modifications simples, mais précisément choisies, du code il est ainsi possible d'accélérer les temps d'exécution d'un facteur compris entre 1,32x et 1,51x. Mais l'amélioration la plus significative est logiquement à porter au crédit de la parallélisation sur GPU. Couplée aux optimisations mises en place, les facteurs d'accélération de 14,14x (néphron simple), 12,30x (multi-néphron) et 18,12x (pyramide rénale). L'accélération atteint donc 18,12x dans le cas de la simulation initialement la plus longue. Afin de bien établir le gain propre à la parallélisation, le facteur d'accélération moyen obtenu par la parallélisation du code optimisé est donné par le Tableau 3.

Accélération moyenne par rapport à la version optimisée de la PGMS			
	Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
Version optimisée et parallélisée sur GPU	9,36x	8,59x	13,71x

Tableau 3 : Accélération (speed-up) moyenne par rapport à la version optimisée de la PGMS pour les trois simulations.

Plusieurs sources peuvent expliquer les différences de gains entre les différentes simulations:

1. Le rapport entre initialisation et génération des simulations :
   Le Tableau 4 et le Tableau 5 montrent le pourcentage de temps passé par chaque simulation dans l'initialisation et la génération de l'organe. Or l'initialisation a été moins impactée par les modifications apportées (optimisation et parallélisation). Il est donc logique que plus l'initialisation constitue une part importante du processus total, plus les optimisations auront un impact faible.

Pourcentage du temps passé en initialisation			
	Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
Version originale	18,57%	7,32%	7,23%

Tableau 4 : Part du temps d'initialisation dans la simulation.

Pourcentage du temps passé en génération			
	Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
Version originale	81,43%	92,68%	92,77%

Tableau 5 : Part du temps de génération dans la simulation.

2. Le nombre de cellules utilisées pour discrétiser l'espace :
   Les trois simulations utilisent des espaces discrétisés de taille différente (voir Tableau 6). Or, comme cela a été expliqué plus haut, les copies de mémoire deviennent le principal goulot d'étranglement lors de la parallélisation sur GPU. Un plus petit nombre de cellules implique logiquement des copies plus petites et donc un rapport plus faible calculs sur copie mémoire.

Nombre de cellules de l'espace discrétisé		
Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
25x60x150 225000 cellules	25x80x100 200000 cellules	25x60x80 120000 cellules

Tableau 6 : Taille de l'espace discrétisé.

3. Le nombre de cellules biologiques présentes dans la simulation :
   Comme l'ont montré les tests réalisés sur les calculs de champs physiques, plus le nombre de particules est important, plus le gain croît. La Tableau 7 montre que le nombre de cellules biologiques est bien plus faible dans le cas de la simulation la plus courte (néphron simple) et également plus important pour le modèle de pyramide rénale que pour le modèle de multi-néphrons.

Nombre de cellules biologiques générées		
Single Nephron2	**Renal Juxtamedullar Section**	**Renal Pyramid**
2 855 cellules	15 888 cellules	17 616 cellules

Tableau 7 : Nombre de cellules biologiques.

Ces résultats sont très encourageants car sans avoir à réaliser des adaptations spécifiques, les optimisations et la parallélisation effectuées en étudiant le modèle de néphron simple ont amélioré significativement les performances de l'exécution des deux autres modèles. Ainsi, il est réaliste d'attendre un comportement similaire pour d'autres simulations utilisant le même mécanisme des champs physicochimiques. De plus, la parallélisation réalisée tire précisément parti des exécutions plus longues et plus complexes. Cela permet d'imaginer des gains conséquents sur des modèles nécessitant encore plus de puissance de calcul.

Conclusion

Le travail d'amélioration de la PGMS, qui était au cœur de cette thèse réalisée au sein d'une entreprise, a donc commencé par un travail de refactorisation et d'amélioration du projet sans lien direct avec l'amélioration des performances. Ainsi, la mise en place d'un gestionnaire de version, de tests automatisés ou encore le portage de l'application en C++ standard ont précédé toutes formes d'améliorations des performances. Le but de ces travaux étant de permettre l'implémentation et l'intégration des améliorations futures beaucoup plus facilement. Il a ainsi été possible, à l'aide du gestionnaire de versions, qu'une équipe continue à enrichir le simulateur de la PGMS pendant que les travaux sur l'amélioration des performances étaient effectués en parallèle. La fusion des deux travaux étant réalisée de manière semi automatisée.

Une fois ces modifications apportées au projet, l'analyse d'une exécution standard de la PGMS a servi de référence. Le choix s'est porté sur la simulation de la morphogénèse d'un néphron (rein) à l'aide d'un profileur. Cette approche a permis de mettre en lumière les différents goulots d'étranglements de l'application. Le plus important d'entre eux étant le calcul de champs chimiques pour chaque cellule de l'espace discrétisé de la simulation (près de 95% du temps d'exécution total), la première tâche a été de paralléliser ce calcul. L'implémentation sur GPU de ce calcul se rapproche fortement de celle des champs physiques étudiée préalablement, en utilisant un thread sur le GPU pour chaque cellule de

l'espace discrétisé. Tout comme dans le cas des champs physiques, les gains résultants de cette parallélisation ont été importants : le temps d'exécution de la simulation d'un néphron est ainsi passé de 132 secondes dans la version originale (utilisant seulement un cœur Sandy Bridge i7-2600K à 3,4GHz) à 11,3 secondes en moyenne dans la version parallélisée (utilisant encore un cœur Sandy Bridge i7-2600K à 3,4GHz pour les calculs séquentiels mais aussi une carte GTX500 pour les parties parallélisées), soit un gain de 11,7x.

Cependant, le profilage de l'application fait apparaître, en plus des appels de fonctions très gourmands en temps de calcul, des instructions et des appels au sein de ces fonctions qui les rendent particulièrement lentes. Dans ce cas, une optimisation du code permet souvent d'améliorer les temps d'exécution des fonctions. Dans ce but, plusieurs travaux ont été menés, entre autres sur le stockage et l'accès des tableaux représentant l'ensemble de l'espace discrétisé et sur l'implémentation des vecteurs mathématiques.

Le stockage continu des tableaux représentant l'ensemble de l'espace discrétisé a ainsi permis un gain de 5% sur le temps d'exécution total de la simulation. Mais ce stockage continu permet d'optimiser l'accès aux éléments du tableau en exploitant les mécanismes de cache de la mémoire devenus très performants au sein des microprocesseurs récents. En accédant aux éléments dans l'ordre de stockage, les mécanismes de cache mémoire permettent ainsi de réduire encore de 23% le temps d'exécution par rapport à l'application originale.

L'implémentation de vecteurs mathématiques, de dimension deux ou trois à l'aide de tableaux de deux ou trois éléments plutôt que de deux ou trois attributs, couplée à l'utilisation d'accesseurs, permettant d'accéder directement à une composante du vecteur, a permis de limiter certains calculs mais aussi de fournir davantage d'informations au compilateur, lui permettant ainsi d'optimiser davantage le code. Le résultat est qu'une modification qui peut elle aussi sembler mineure a permis un gain supplémentaire de 15% du temps d'exécution.

Mis bout à bout, l'ensemble des optimisations seules a permis un gain de 1,5x pour la simulation d'un néphron. Lorsque l'on couple les optimisations à la parallélisation sur GPU, le gain pour cette même simulation atteint 14,14x. Un point très intéressant est que toutes ces améliorations ont été réalisées de façon à ce qu'elles soient les plus génériques

possibles. Ainsi, même si la simulation d'un néphron a été utilisée tout au long des travaux pour mesurer les résultats de chaque amélioration, les autres simulations sont également accélérées. La simulation de plusieurs néphrons (qui prenait plus de 11 minutes originellement) est 1,43x plus rapide en séquentiel et 12,30x plus rapide lorsque la parallélisation sur GPU est activée (54,5 secondes dans ce dernier cas). La simulation d'une pyramide rénale, à l'origine la plus longue des simulations (plus de 16 minutes) est, quant à elle, 1,32x plus rapide grâce aux optimisations en séquentiel et 18,12x plus rapide grâce à la parallélisation sur GPU (53,6 secondes dans ce cas, soit moins que pour la simulation de multi-néphron). Ces résultats sont très encourageants car ils montrent que les améliorations apportées à l'application sont bien de nature à impacter des simulations variées au sein de la PGMS.

Bilan général et perspectives

Bilan général et perspectives

Au cours des dix dernières années, la modélisation des systèmes complexes a pris un essor particulièrement important parmi les disciplines de modélisation. La popularité de l'étude des systèmes complexes est due au fait que ces systèmes permettent de modéliser un nombre important de systèmes réels difficilement appréhendables avec une autre technique de modélisation (White, Peng et Demir 2009). Ils permettent ainsi la modélisation des flux boursiers (Lillo et Mantegna 2003), des comportements sociaux (Sawyer 2005) ou des comportements animaux (Boccara 2004) par exemple. Mais les caractéristiques même des systèmes complexes en font des systèmes généralement très difficiles à simuler fidèlement du fait des nombreuses interactions entre les éléments du système.

Si des méthodes de modélisation efficaces existent, leurs implémentations ne sont pas triviales et nécessitent souvent beaucoup de travail. C'est pour permettre de plus facilement disposer d'une modélisation multi-échelle d'un problème que la Plateforme Générique de Modélisation et de Simulation (PGMS) a été développée (P. Siregar 2009). Son objectif, comme son nom l'indique, est de proposer une plateforme suffisamment générique pour permettre de modéliser et de simuler le plus simplement possible un système biologique multi-échelle. Mais l'utilisation d'un outil générique implique logiquement des temps d'exécution plus longs que ceux d'un outil spécialement conçu pour un système donné. La modélisation d'un système complexe étant déjà très souvent gourmande en temps de calcul, la PGMS s'est retrouvée dans une situation où les simulations prenaient beaucoup trop de temps : plus d'une minute pour la morphogénèse d'un néphron ; sachant qu'un rein est composé de plusieurs centaines de milliers de néphrons. Ces vitesses d'exécutions étant incompatibles avec l'objectif de simulations d'éléments beaucoup plus important du corps humain sur une machine de table, une reprise du code de l'application était nécessaire.

L'objectif des travaux menés a donc été d'améliorer l'implémentation de la PGMS à travers trois grands facettes : la standardisation du code de l'application (initialement implémentée en Borland C++), l'optimisation du code séquentiel existant et la parallélisation sur une architecture parallèle des mécanismes de l'application les plus lourds. En parallèle de ces travaux intégrés à la PGMS, des travaux plus théoriques ont également été menés sur les automates cellulaires et sur la programmation sur GPU en anticipation d'une intégration de ceux-ci dans une future version de la PGMS.

La standardisation du code de la PGMS ainsi que l'amélioration de la gestion du projet étaient devenues des préalables obligatoires à l'amélioration des performances. En effet, la PGMS avait été développée dans une extension du C++ proposé par Borland : Borland C++. Cette extension proposait, en plus de mot-clés n'appartenant pas au standard, plusieurs bibliothèques permettant, entre autre, de traiter de la conception d'interfaces graphiques ou de disposer d'un panel complet de conteneurs. Cela avait conduit les développeurs à utiliser ces deux bibliothèques de manière extensive. Afin de se passer complètement des outils proposés par Borland C++ et de fournir un code fidèle au standard du C++, il a donc était nécessaire de reprendre le code de l'application en profondeur.

En amont de ces travaux et afin d'améliorer le processus de modification de l'application, deux outils ont été mis en place : un gestionnaire de versions et un cadriciel de test. L'utilité première d'un gestionnaire de versions est de conserver l'historique de chaque modification apportée au projet, afin de pouvoir comparer la version actuelle d'un code à une version antérieure et même de revenir à une version antérieure en cas de problème. Une autre caractéristique très utile dans notre cas a été la possibilité de travailler en parallèle sur deux versions d'un même programme et de fusionner les deux travaux plus tard. L'ajout de fonctionnalités et l'amélioration des performances ont ainsi pu être réalisés par deux équipes en parallèle. Un cadriciel de test permet quant à lui de produire des tests automatisés. L'automatisation des tests est un outil précieux du développeur car elle permet de vérifier qu'une modification apportée au code (une optimisation aussi bien que l'ajout d'une fonctionnalité) n'a pas introduit de bogues dans le code (Feathers 2005).

Une fois la reprise de l'application en C++ standard effectuée, il a été possible de travailler à l'amélioration des performances. Parmi les goulots d'étranglement pointés par le profilage de l'application, le calcul des champs chimiques représentait très largement la part la plus importante du temps d'exécution (près de 95% du temps d'exécution y était consacré). L'objectif a donc été dans un premier temps d'étudier l'opportunité de paralléliser ces calculs particulièrement importants, la puissance de calcul des processeurs en séquentiel étant limitée.

La parallélisation des champs chimiques sur GPU a d'abord commencé par une étude de performance préliminaire, en traitant de façon plus général les champs physicochimiques dans une application dédiée. Parmi les trois champs choisis pour cette étude (gravitationnelle, électrique et magnétique), les champs gravitationnel et électrique se calculent de la même façon en cumulant le champ créé par chaque particule en présence dans la simulation. Le champ magnétique ne provient dans notre cas que d'une unique source. Pour atteindre notre objectif, qui est d'obtenir la valeur de chaque champ dans chaque cellule d'un espace discrétisé, l'algorithme séquentiel parcourt pour chaque champ l'ensemble des cellules de l'espace. Dans le cas des champs gravitationnel et électrique, pour chacune de ces cellules, il parcourt alors chacune des particules et additionne le champ généré. Le champ magnétique ne possédant qu'une source, il se contente pour chaque cellule de calculer le champ généré.

Nous avons choisi de paralléliser le calcul de champ sur GPU en affectant un thread s'exécutant sur le GPU à chaque cellule de l'espace. Ainsi, pour un espace de 50^3 cellules et quelques soit le nombre de particules présentes, le parallélisation sur GPU se fera sur 125 000 threads. Cette stratégie, à l'inverse d'une parallélisation basée sur les particules, permet d'éviter tout problème d'accès parallèle à une même donnée et fournit très vite un nombre de threads très important (le nombre de particules traitées par la PGMS peut être inférieur à 1 000, ce qui ne permettrait pas de charger les GPU récents au maximum). Un thread calculant le champ gravitationnel ou électrique n'a donc qu'à retrouver sa position en fonction de l'indice du thread puis à parcourir l'ensemble des particules afin de pouvoir calculer le champ résultant. La liste de ces particules étant commune à l'ensemble de l'espace, afin de mutualiser l'accès à ces données, celles-ci ont été stockées en mémoire partagée, ce qui a permis de réduire les temps d'accès mémoire.

Le succès de la parallélisation n'est pas le même pour le calcul des champs gravitationnel et électrique, et celui du champ magnétique. Les deux premiers étant particulièrement gourmands en temps de calcul (plus de cent fois plus long lorsque 200 particules sont présentes), il est logique que la parallélisation soit plus profitable dans leurs cas. Ainsi, lors de la parallélisation sur GPU (GTX500 de NVIDIA) d'un code exécuté séquentiellement sur un seul cœur CPU (Intel i7-2600k cadencé à 3,4GHz), si le gain croît, avec la taille de l'espace de simulation, jusqu'à plus de 15x dans le cas du champ magnétique, il croît jusqu'à plus de

600x dans le cas des champs gravitationnel et électrique (lorsque 200 particules sont présentes). Le gain de la parallélisation des champs gravitationnel et électrique évolue également avec le nombre de particules. Ainsi, pour un cube de 50^3 cellules (125 000), un palier de 250x n'est atteint que lorsque plus de 1 000 particules sont utilisées.

Au sein de la PGMS, le calcul des champs chimiques est réalisé à l'aide d'une méthode analytique très proche de celle utilisée pour le calcul des champs gravitationnel et électrique. Le champ présent en un point de la simulation est ainsi calculé en additionnant le champ déjà présent à celui généré par les différentes cellules. La très forte proportion du temps de calcul des champs sur l'ensemble du temps d'exécution de l'application couplé au gain important apporté par la parallélisation a permis d'obtenir un facteur d'accélération de 11,7x pour la simulation d'un néphron du rein (la simulation utilisée comme référence lors de ces travaux), faisant passer le temps d'exécution de 132 à 11,3 secondes.

Une fois ces travaux de parallélisation terminés et afin d'augmenter encore les performances de l'application, le choix a été fait d'optimiser les opérations les plus courantes. À l'inverse de la parallélisation ou d'un changement d'algorithme, l'optimisation d'un code n'a pas pour but de modifier en profondeur la façon de réaliser une opération mais d'améliorer son implémentation. Un grand nombre de techniques existent dans ce but dont beaucoup sont maintenant implémentées dans les compilateurs. Mais les compilateurs n'ont pas toujours suffisamment d'informations pour réaliser seuls les optimisations. Il est alors nécessaire d'implémenter manuellement les optimisations les plus pertinentes après profilage de l'application.

Bien entendu, une optimisation est d'autant plus utile qu'elle affecte un code appelé un grand nombre de fois. C'est la raison pour laquelle beaucoup de ces optimisations ciblent les boucles (Allen et Kennedy 2001). C'est le cas par exemple de l'échange de boucles, qui consiste à inverser deux boucles imbriquées l'une dans l'autre afin de maximiser l'utilisation des caches mémoires. Mais d'autres techniques ciblent également les expressions indifféremment d'où elles se situent. C'est le cas par exemple de l'élimination des sous-expressions communes qui consiste, lorsque plusieurs expressions possèdent une même sous-expression, à évaluer cette sous-expression en amont des expressions afin de ne la calculer qu'une seule fois (Cocke 1970).

Au sein de la PGMS, le profilage de l'application a montré que les plus gros gains pouvaient être espérés de l'amélioration des accès aux grands tableaux représentant tout l'espace de la simulation (l'ensemble des valeurs des différents champs chimiques entre autre) et des accès aux différentes composantes des vecteurs mathématiques à deux ou trois dimensions (représentant souvent une position). Pour améliorer les accès aux grands tableaux, une première étape a consisté à modifier la façon donc la structure de données allouait l'espace mémoire. En effet, pour une instance d'espace en trois dimensions, rien ne garantissait que l'espace soit alloué de façon contiguë. Afin d'exploiter au maximum les mécanismes de cache de la mémoire, une première optimisation a consisté à allouer l'ensemble de ces données de manière continue dans un tableau possédant une seule dimension. L'accès à un élément se fait alors à l'aide d'une méthode qui calcule, à partir des indices x, y et z, quel est l'indice correspondant dans le tableau. L'impact d'une telle optimisation sur la PGMS est limité à un gain de 5% du temps d'exécution par rapport à la PGMS originale. Mais cette optimisation permet de pouvoir également optimiser l'ordre d'accès aux différents éléments.

En effet, les accès aux éléments du tableau dans les boucles étaient réalisés en parcourant celui-ci selon l'indice z puis l'indice y et enfin l'indice x. Or le stockage est réalisé selon x, puis y et enfin z. Un parcours suivant ce stockage est donc beaucoup plus performant grâce à l'utilisation des mécanismes de cache mémoire. Ceci est d'autant plus vrai lorsque l'optimisation précédente est réalisée car l'ensemble du tableau est alors contigu et non plus que certaines parties. Cette optimisation consiste donc simplement à appliquer l'échange de boucles (Allen et Kennedy 2001) sur les trois boucles parcourant x, y et z de façon à avoir comme boucle intérieur la boucle parcourant x, la boucle intermédiaire parcourant y et la boucle extérieur parcourant z. Couplée à l'optimisation précédente, cette optimisation permet de réduire le temps d'exécution de 23% sans influer sur le résultat de la simulation (le calcul de chaque élément dans la boucle étant indépendant).

Un autre goulot d'étranglement de l'application était l'accès aux éléments des vecteurs mathématiques à l'aide de l'opérateur `[]`. Ceci était dû à l'utilisation d'un attribut par élément du vecteur et d'enchainement de « if … else if … » pour retrouver quel élément retourner. L'utilisation d'un tableau comprenant l'ensemble des éléments du vecteur a permis de réduire l'implémentation de l'opérateur `[]` à sa plus simple expression (retourner

l'élément du tableau de l'indice passé en paramètre). Pour accéder de manière encore plus explicite à un élément précis du conteneur, des accesseurs ont également était ajoutés. Ceci a eu pour conséquences de favoriser l'intégration directe du code de la fonction au code appelant (inlining) et favoriser les optimisations du compilateur. Le résultat est un gain important de 15% par rapport au temps original de la PGMS.

En plus des trois optimisations récapitulées ici, d'autres optimisations, aux portées plus ou moins limitées, ont également été implémentées. La suite logique de ce travail serait de continuer à profiler l'application afin de trouver de nouveaux goulots d'étranglements et de chercher à les supprimer. Le problème de cette stratégie est qu'elle ne peut s'attaquer au plus gros goulot d'étranglement actuel qui est la copie des résultats des calculs de champs entre le GPU et le CPU.

Une fois la parallélisation et toutes les optimisations mises bout à bout, la simulation d'un néphron ne prend plus que 9,1 secondes soit un gain de plus de 14x par rapport à la simulation originale. Ces améliorations sont toutes aussi efficaces sur d'autres simulations pour lesquelles elles n'avaient jamais été testées durant leurs élaborations. Ainsi, la simulation d'un modèle de multi-néphron est réalisée en 54,5 secondes lorsqu'elle nécessitait plus de 11 minutes précédemment (gain de 12,3x). La simulation d'une pyramide rénale, qui prenait plus de 16 minutes, prends également moins d'une minute (53,6 secondes), soit un gain de 18,1x. Ces résultats sont particulièrement intéressants car ils signifient que les améliorations réalisées sont suffisamment génériques pour s'appliquer à d'autres simulations. Il reste malgré tout à tester ces améliorations sur des simulations traitant d'autres éléments que des éléments du rein mais celles-ci ne sont pas encore disponibles au sein de la PGMS.

En plus des travaux qui ont pu être intégrés à la PGMS pendant cette thèse, d'autres travaux ont été menés en vue d'une intégration future à celle-ci. Ainsi, une partie importante des travaux réalisés a eu pour thème les automates cellulaires (Von Neumann 1966). Cet outil de modélisation, très utilisé pour la modélisation du vivant, consiste en un espace discrétisé en cellules identiques, interconnectées localement, sur lesquelles s'appliquent un ensemble de règles (Wolfram 2002). Les automates cellulaires ont été popularisés par le « Jeu de la Vie » inventé par Conway (Gardner 1970), un automate cellulaire très simple qui permet de

simuler une forme de vie de façon simpliste. Mais cet outil permet de simuler des comportements beaucoup plus complets et beaucoup plus réalistes comme la croissance d'une tumeur (Alarcon, Byrne et Maini 2004) ou l'activité électrique du myocarde (Siregar, Sinteff, et al. 1998).

Notre but était encore une fois d'améliorer les performances de la simulation de tels automates. Pour cela, l'algorithme Hash-Life (Gosper 1984) a retenu notre attention. Cet algorithme tire parti de la mémoïzation pour améliorer les performances de la simulation d'automates cellulaires. Appliqué à une fonction, le principe de mémoïzation (Michie 1968) consiste à sauvegarder le résultat de celle-ci pour un jeu de paramètres de façon à pouvoir retourner ce résultat directement si le même appel survient. Pour un automate cellulaire, cela revient à sauvegarder, pour une configuration de cellules données, la configuration résultante après la simulation d'une ou plusieurs itérations. L'algorithme Hash-Life sauvegarde ainsi le résultat de la simulation d'une ou plusieurs itérations sur chaque configuration qu'il découvre. Lorsqu'une configuration déjà apparue est retrouvée par l'algorithme, il se contente alors de retourner le résultat déjà calculé auparavant. En utilisant également un découpage de l'espace qui permet de ne conserver en mémoire qu'une instance de chaque configuration apparue, Hash-Life réussi à disposer d'excellentes performances aussi bien en terme de temps d'exécution que d'empreinte mémoire. Nos travaux sur les automates cellulaires ont donc porté dans un premier sur l'implémentation de l'algorithme Hash-Life en trois dimensions et l'étude de son efficacité dans cette configuration.

Au-delà de l'aspect technique du portage de l'algorithme Hash-Life en trois dimensions, ce qui nous a intéressés est la capacité de l'algorithme à conserver ses excellentes performances dans un espace en trois dimensions (Caux, Siregar et Hill 2010). L'utilisation d'un espace en trois dimensions n'est pas anodine. L'algorithme utilise une structure en arbre complet qui impose de manipuler des cubes d'une largeur égale à une puissance de 2 (en nombre de cellules). Ainsi, lorsqu'en deux dimensions il n'était possible de n'avoir que 65 536 configurations pour une surface de $2^{ème}$ niveau (4^2 cellules) et deux états possible par cellules, ce nombre monte à 10^{20} en trois dimensions (4^3 cellules). Un nombre plus grand de configurations possibles implique logiquement des chances plus faibles de retrouver une

configuration déjà apparue (si toutes ont une probabilité équivalente d'apparaître) et donc moins de chance de tirer parti du mécanisme de mémoïzation.

Selon le degré de réapparition des configurations, l'algorithme Hash-Life en trois dimensions n'est pas toujours performant. Ainsi, lorsque l'on simule certaines variantes du « Jeu de la Vie » en trois dimensions, les performances sont moins bonnes qu'un algorithme séquentiel. À l'inverse, lorsque l'on s'intéresse à un automate cellulaire où l'on retrouve plus souvent des configurations identiques, tel qu'un milieu excitable, les performances sont bien meilleures. En traitant une itération à la fois, le temps d'exécution et la mémoire nécessaire pour exciter l'ensemble des cellules de l'espace sont ainsi linéaires au côté du cube. Et ceci, bien que le nombre de cellules totales du cube est égale au côté élevé au cube et que le nombre d'itérations nécessaires pour exciter l'ensemble des cellules vaut trois fois ce même côté du cube. Une itération dure ainsi approximativement aussi longtemps quel que soit la taille du cube étudié. Les résultats sont bien supérieurs lorsque l'on s'autorise à réaliser plusieurs itérations à la fois (mais dans ce cas, le résultat intermédiaire ne peut plus être retrouvé). Avec cette approche, le temps d'exécution et la mémoire nécessaires sont linéaires au logarithme du côté du cube. Il est alors possible d'exciter toutes les cellules d'un automate cellulaire de plus de 10^{27} cellules en moins de 1,2 secondes (le résultat final correspond à 10^{9} itérations). Les bons résultats de l'algorithme sur un milieu excitable s'expliquent parfaitement lorsque l'on considère l'algorithme du point de vue de l'activité (Muzy et Hill 2011). Nos travaux montrent que l'algorithme, via son mécanisme de mémoïzation, peut s'apparenter à un algorithme se focalisant sur l'activité du système, celle-ci étant l'apparition de nouvelles configurations dans le cas de Hash-Life (Muzy, Varenne, et al. 2012).

Lorsque l'algorithme Hash-Life n'est pas utilisable, que ce soit pour des raisons de temps d'exécution ou d'impossibilité dans le cas d'automates cellulaires stochastiques, il est nécessaire de disposer d'un autre outil pour améliorer le temps d'exécution de la simulation. Notre choix a été d'implémenter un simulateur d'automates cellulaires sur GPU (Caux, Siregar et Hill 2011). Pour cette parallélisation, l'idée a été encore une fois d'utiliser un thread sur le GPU pour chaque cellule de l'espace discrétisé. Chaque thread se charge donc de faire évoluer une cellule de l'automate cellulaire. De plus, afin de réduire les coûts de la copie entre le GPU et le CPU, l'algorithme implémenté permet de réaliser plusieurs

itérations successives avant que le résultat ne soit récupéré sur le CPU. Le problème de l'implémentation d'un simulateur d'automates cellulaires en trois dimensions sur GPU est qu'il est nécessaire pour chaque cellule de connaitre la valeur de chacun de ses voisins, qui peuvent être nombreux (26 avec un voisinage de Moore de degré un). Or les accès mémoire ont une grosse influence sur les performances des GPU. Ainsi même en mutualisant les accès des cellules proches dans une mémoire partagée, ceux-ci ont un coût important.

Ces travaux ayant été réalisés en début de thèse, les résultats complets de cette parallélisation ont été effectués sur un Core 2 Xeon et un Nehalem Xeon équipé d'une carte Tesla C1060. La copie aller et retour de l'espace discrétisé avant et après la simulation d'une ou plusieurs itérations possédant un coût fixe quel que soit le nombre d'itérations effectuées, il est logiquement plus rentable d'effectuer plusieurs itérations sur le GPU avant de récupérer les données. Ainsi, pour un nombre de cellules donné, le gain croît avec le nombre d'itérations jusqu'à atteindre un palier, qui est atteint d'autant plus vite que le nombre de cellules est grand (celui-ci apparait dès 100 itérations dans le cas d'un automate de 50^3 cellules lorsqu'il n'apparait qu'après 200 itérations dans le cas d'un automate de 10^3 cellules). Ce problème de coût de la copie entre le CPU et le GPU mis à part, le gain proposé par le GPU est de l'ordre de 60x par rapport au processeur Core 2 et 20x par rapport au processeur Nehalem. Ce gain est bien évidemment beaucoup plus faible que ce que permet l'algorithme Hash-Life lorsqu'il traite d'un cas favorable mais il permet, lorsque le cas n'est pas favorable à l'algorithme Hash-Life de disposer d'un gain conséquent. Par ailleurs, l'algorithme proposé ici est très générique. Si les tests ont été effectués avec une variante du « Jeu de la Vie » en trois dimensions, ce n'est pas l'objectif final. Une fois l'automate cellulaire auquel il devra être appliqué parfaitement connu, il sera possible d'optimiser le stockage de l'état de chaque cellule ainsi que l'accès au voisinage afin d'améliorer les performances.

De façon plus transverse à l'ensemble des travaux effectués, nous avons également cherché une solution permettant d'exploiter plus facilement la puissance de calcul des GPU pour les simulations stochastiques. L'idée étant ici d'utiliser la puissance des GPU, non pas pour réaliser plus rapidement une simulation, mais pour accélérer l'exécution des réplications d'une simulation stochastique. Les réplications d'une simulation stochastique pouvant parcourir des branches différentes du programme lors de leurs exécutions, l'approche

purement SIMD utilisée généralement lors de la parallélisation sur GPU n'est pas toujours judicieuse. Nous avons donc proposé une nouvelle approche (Passerat, et al. 2011) basée sur l'utilisation des warps plutôt que des threads. Un warp représente un groupe de threads (trente-deux à l'heure actuelle) devant exécuter la même instruction et donc utiliser une approche SIMD pour réaliser des traitements identiques sur des données différentes. Lorsque des threads d'un même warp doivent effectuer des instructions différentes, le GPU doit créer un nouveau warp pour chaque branche en désactivant tous les threads réalisant des instructions différentes. En n'utilisant qu'un thread par warp, le parcours de branches différentes n'impacte pas sur le temps d'exécution de la simulation.

Bien entendu, cela n'est possible qu'au prix d'une utilisation partielle de la puissance du GPU car toute la composante SIMD du GPU est inutilisée (un seul des trente-deux threads d'un warp traite des données). Nos résultats montrent que malgré cela, les performances fournies par un GPU sont supérieures à celles d'une implémentation séquentielle sur CPU dès que le nombre de réplications est conséquent (entre 20 et 30 réplications minimum pour les modèles que nous avons essayés). Lorsque l'on compare notre approche à une parallélisation des réplications sur GPU réalisée de façon traditionnelle (en laissant à la charge du GPU de séparer les threads d'un même warp parcourant des branches différentes), les temps d'exécutions sont également plus faibles tant que le nombre de réplications n'est pas trop important (moins de 700 réplications). Ces résultats, couplés à une implémentation permettant de masquer au maximum les détails d'implémentation de notre approche, permettent de proposer une solution efficace pour réaliser les réplications de simulations stochastiques possédant un nombre important de branchements.

Si des avancées significatives ont pu être réalisées lors de ces travaux, des limites sont encore présentes à une utilisation généralisée. Ainsi, si la parallélisation sur GPU du calcul de champs dans la PGMS a permis de réduire très significativement le temps d'exécution du calcul de champs, elle a également introduit une copie des données du GPU vers le CPU qui est devenue une part très importante du temps d'exécution de la PGMS. Or si des pistes existent pour réduire ce temps (nous reviendrons sur ce point dans les perspectives juste après), ces copies sont pour le moment un goulot d'étranglement important sur lequel nous

ne disposons pas d'une marge de manœuvre très importante, la copie par le bus PCI de la carte mère étant obligatoire.

De la même façon, si l'algorithme Hash-Life en trois dimensions peut s'avérer très performant lorsque le cas d'utilisation est adéquat, le mécanisme de mémoïzation peut également introduire des ralentissements si le taux de réapparition des configurations de cellules est trop faible. Étant donné le voisinage plus important de chaque cellule en trois dimensions, la réapparition des configurations est plus limitée avec l'algorithme en trois dimensions que pour celui en deux dimensions. L'algorithme en trois dimensions est donc utilisable dans un nombre plus réduit de cas. Mais l'aspect le plus limitant pourrait être que la solution proposée à l'heure actuelle ne permet pas de simuler les automates stochastiques. En effet, si l'état d'une configuration après une itération varie aléatoirement, il n'est pas possible de le sauvegarder et de le réutiliser tel quel par la suite.

La parallélisation au niveau des warps est intéressante mais possède également des limites importantes. La plus importante est que son utilisation ne garantit pas automatiquement un gain. Pour que l'utilisation de notre approche soit intéressante, il est nécessaire que le code parallélisé dispose d'un nombre important de branchements et que le nombre de réplications soit dans une fourchette donnée ; fourchette qu'il est très difficile de prédire à priori pour une simulation donnée. Par ailleurs, l'apparition des nouvelles architectures GPU, qui semblent augmenter davantage la puissance de calcul parallèle pur (SIMD), pourrait rendre notre approche moins pertinente.

Pour autant, toutes les perspectives de parallélisation du calcul de champs n'ont pas pu être explorées. L'importance des temps de copie entre le GPU et le CPU laisse supposer que les copies pourraient être réalisées de manière asynchrone de façon à réaliser en parallèle la copie du résultat du calcul d'un champ pendant que le calcul du champ suivant est effectué. Cette solution a pour le moment été laissée de côté car elle nécessiterait une modification plus en profondeur du code de la PGMS que la parallélisation simple d'une fonction (le calcul sur GPU se fait pour le moment simplement en appelant la fonction `computeSemaphorinField_gpu` en lieu et place de la fonction originale `computeSemaphorinField_cpu`) et impacterait beaucoup plus largement l'architecture

de l'application. Une autre piste pourrait être l'utilisation de plusieurs GPU pour réaliser les calculs. Le code du calcul de chaque champ étant parfaitement indépendant, il serait ainsi assez simple d'envisager l'utilisation de plusieurs cartes GPU. Le temps global d'exécution des kernels pourrait ainsi être diminué mais il serait également possible de réaliser les copies retour en parallèle sur des ports PCI Express différents. À ce sujet, le support par les nouvelles cartes graphiques de la norme PCI Express 3 pourrait avoir un effet important sur le temps d'exécution sans nécessiter la moindre modification du code de l'application. Enfin, le développement par NVIDIA de cartes toujours plus puissantes (architecture Kepler puis Maxwell) permet d'envisager une diminution du temps de calcul en faisant simplement évoluer le matériel.

Un autre aspect important, qui permettrait de gagner encore en temps d'exécution, est bien entendu l'utilisation des multiples cœurs proposés par les microprocesseurs actuels. En effet, si le choix a été fait de se concentrer sur la parallélisation sur GPU pour les calculs de champs, c'est dû au fait qu'il représentait près de 95% du temps de calcul. Une parallélisation minutieuse sur GPU a ainsi permis de diminuer de manière très importante le temps d'exécution. Cette parallélisation effectuée, aucun processus aussi important n'est présent. Une parallélisation, plus grossière, sur CPU, d'autres parties de l'application permettrait de diminuer encore le temps d'exécution. Malheureusement, la prise en main du code de la PGMS n'a pu se faire que tardivement et ces travaux n'ont pas pu être menés à bien dans le cadre de cette thèse.

Bibliographie

Aho, A.V., J.D. Ullman, and S. Biswas. *Principles of compiler design.* Addison-Wesley Longman Publishing Co., Inc., 1977.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

Alarcon, T., HM Byrne, and PK Maini. "Towards whole-organ modelling of tumour growth." Edited by Elsevier. *Progress in biophysics and molecular biology* 85, no. 2-3 (2004): 451-472.

Allen, Randy, and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., 2001.

Altschul, Stephen F., Warren Gish, Webb C. Miller, Eugene W. Myers, and David J. Lipman. "Basic local alignment search tool." *Journal of Molecular Biology* (Elsevier) 215, no. 3 (1990): 403-410.

Amini, M., F. Coelho, F. Irigoin, and R. Keryell. "Static Compilation Analysis for Host-Accelerator Communication Optimization." *LCPC'2011: The 24th International Workshop on Languages and Compilers for Parallel Computing*, 2011.

Bagrodia, R., et al. "Parsec: A parallel simulation environment for complex systems." *IEEE Computer* 31, no. 10 (1998): 77-85.

Balci, Osman. "The implementation of four conceptual frameworks for simulation modeling in high-level languages." *WSC '88 Proceedings of the 20th conference on Winter simulation* , 1988: 287-295.

Bardou, A.L., P.M. Auger, P.J. Birkui, and J.L. Chassé. "Modeling of cardiac: Electrophysiological mechanisms: From action potential genesis to its propagation in myocardium." *Critical Reviews in Biomedical Engineering* (CRC Press) 24, no. 2-3 (1996): 141-221.

Bargen, B., and P. Donnelly. *Inside DirectX: in-depth techniques for developing high-performance multimedia applications.* Mircrosoft Press, 1998.

Bar-Yam, Y. *Dynamics of complex systems.* Westview Press, 2003.

Basili, Victor R., Lionel C. Briand, and Walcélio L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators." *IEEE Transactions on Software Engineering* 22, no. 10 (1996): 751-761.

Berlekamp, Elwyn R., John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays.* Peters, 2004.

Beyer, T., and M. Meyer-Hermann. "Multiscale modeling of cell mechanics and tissue organization." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 2 (2009): 38-45.

Binstadt, E.S., et al. "A comprehensive medical simulation education curriculum for emergency medicine residents." *Annals of emergency medicine* (Elsevier) 49, no. 4 (2007): 495-504.

Bitar, Ziad El, Delphine Lazaro, Christopher Coello, Vincent Breton, David Hill, and Irene Buvat. "Fully 3D Monte Carlo image reconstruction in SPECT using functional regions." *Nuclear Instruments & Methods in Physics Research Section A-accelerators Spectrometers Detectors and Associated Equipment* 569, no. 2 (2006): 399-403.

Bloch, J. *Effective java: programming language guide.* Addison-Wesley Professional, 2001.

Boccara, N. *Modeling complex systems.* Springer Verlag, 2004.

Buck, Ian, et al. "Brook for GPUs: stream computing on graphics hardware." *ACM Transactions on Graphics* 23, no. 3 (2004): 777-786.

Buxton, J. N., and J. G. Laski. "Control and simulation language." *The Computer Journal* (Br Computer Soc) 5, no. 3 (1962): 194-199.

Cannings, C., EA Thompson, and HH Skolnick. "The recursive derivation of likelihoods on complex pedigrees." *Advances in Applied Probability* (JSTOR) 8, no. 4 (1976): 622-625.

Caux, J., P. Siregar, and D. Hill. "Hash-life algorithm on 3D excitable medium application to integrative biology." *Summer Simulation Multiconference* (Society for Computer Simulation International), 2010: 387-393.

Caux, J., P. Siregar, and D. R.C. Hill. "Accelerating 3D Cellular Automata Computation with GP GPU in the Context of Integrative Biology." In *Cellular Automata - Innovative Modelling for Science and Engineering*, by Alejandro Salcido. InTech, 2011.

Chafik, O. *JavaCL : Opensource Java wrapper for OpenCL library.* 2009. http://code.google.com/p/javacl/ (accessed 03 26, 2012).

—. *ScalaCL : Faster Scala : optimizing compiler plugin + GPU-backed collections (OpenCL).* 2011. http://code.google.com/p/scalacl/ (accessed 03 26, 2012).

Cocke, John. "Global common subexpression elimination." *Sigplan Notices* 5, no. 7 (1970): 20-24.

Coquillard, P., and D. Hill. *Modélisation et Simulation des Ecosystèmes.* Masson, 1997.

Daga, M., A.M. Aji, and W. Feng. "On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing." *2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011: 141-149.

Denning, P.J. "The locality principle." *Communications of the ACM* (ACM) 48, no. 7 (2005): 19-24.

Detlefs, D., A. Dosser, and B. Zorn. "Memory allocation costs in large C and C++ programs." *Software: Practice and Experience* (Wiley Online Library) 24, no. 6 (1994): 527-542.

Dewdney, AK. "The cellular automata programs that create Wireworld, Rugworld and other diversions." *Scientific American* 262, no. 1 (1990): 146-149.

Ermentrout, G.B., and L. Edelstein-Keshet. "Cellular automata approaches to biological modeling." Edited by Academic Press. *Journal of Theoretical Biology* 160 (1993): 97-133.

Feathers, M.C. *Working effectively with legacy code.* Prentice Hall Professional Technical Reference, 2005.

Flynn, Michael J. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers* 21, no. 9 (1972): 948-960.

Fog, A. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.* Copenhagen University College of Engineering, 2008.

Foster, I. and Kesselman, C. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann, 2004.

Foster, I. "What is the grid? a three point checklist." *GRID today*, 2002.

Fowler, M. and Beck, K. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

Fowler, Martin. *Refactoring: Improving the Design of Existing Programs.* 1999.

Gardner, M. "Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'." *Scientific American* 223, no. 4 (1970): 120-123.

Gerhardt, M., H. Schuster, and J.J. Tyson. "A cellular automation model of excitable media including curvature and dispersion." *Science* 247, no. 4950 (1990): 1563-1566.

Gosper, R. Wm. "Exploiting regularities in large cellular spaces." *Physica D: Nonlinear Phenomena* 10, no. 1-2 (1984): 75-80.

Govindaraju, Naga K., Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. "Fast Computation of Database Operations using Graphics Processors." *International Conference on Management of Data*, 2004: 215-226.

Harris, Mark J., Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. "Physically-based visual simulation on graphics hardware." *SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware*, 2002: 109-118.

Hassas, Salima, and Bat Nautibus. "Engineering Complex Adaptive Systems Using Situated Multi-agents." *Engineering Societies in the Agent World*, 2005: 125-141.

Hedlund, G.A. "Endomorphisms and automorphisms of the shift dynamical system." Edited by Springer. *Theory of Computing Systems* 3, no. 4 (1969): 320-375.

Hillis, Daniel. *The Connection Machine.* MIT Press Series in Artificial Intelligence, 1989.

Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall International, 1985.

Hoekstra, A., J.L. Falcone, A. Caiazzo, and B. Chopard. "Multi-scale modeling with cellular automata: The complex automata approach." *Cellular automata* (Springer), 2008: 192-199.

Hoffman, A.R., and J.F. Traub. *Supercomputers: directions in technology and applications.* National Academies Press, 1989.

Hovland, R.J. *Latency and Bandwidth Impact on GPU-systems.* Norwegian University of Science and Technology (NTNU), 2008, 61.

Hunt, A., and D. Thomas. *The pragmatic programmer: from journeyman to master.* Addison-Wesley Professional, 2000.

Kapellos, G, T Alexiou, and A Payatakes. "A multiscale theoretical model for diffusive mass transfer in cellular biological media." *Mathematical Biosciences* 210, no. 1 (2007): 177-237.

Karimi, K., N.G. Dickson, and F. Hamze. "A performance comparison of CUDA and OpenCL." 2010: 1-10.

Kennedy, K., and J.R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2001.

Khronos OpenCL Working Group and others. *The OpenCL specification.* A. Munshi, Ed, 2008.

Kirk, D. "NVIDIA CUDA software and GPU parallel computing architecture." *International Symposium on Memory Management*, 2007: 103-104.

Kirk, D., W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann, 2010.

Kirk, David Blair, and Wen-mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach.* 2010.

Koufaty, David, and Deborah T. Marr. "Hyperthreading Technology in the Netburst Microarchitecture." *IEEE Micro* 23, no. 2 (2003): 56-65.

Lavenier, D., L. Xinchun, and G. Georges. "Seed-based genomic sequence comparison using a FPGA/FLASH accelerator." *IEEE International Conference on Field Programmable Technology*, 2006: 41-48.

Lillo, F., and R.N. Mantegna. "Power-law relaxation in a complex system: Omori law after a financial market crash." *Physical Review E* 68, no. 1 (2003).

Lin, Ching-long, M. H. Tawhai, G. Mclennan, and E. A. Hoffman. "Computational fluid dynamics." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 3 (2009): 25-33.

Lindholm, J., Nickolls, J., Oberman, S., and Montrym, J. "Nvidia tesla : A unified graphics and com-puting architecture." *IEEE Micro* 28, no. 2 (2008): 39-55.

Liskov, Barbara. "Keynote Address - Data Abstraction and Hierarchy." *ACM Sigplan Notices* 23, no. 5 (1987): 17-34.

Lu, S., et al. "Multiscale modeling in rodent ventricular myocytes." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 2 (2009): 46-57.

Masumoto, Y., et al. "A fifty-year eddy-resolving simulation of the world ocean: Preliminary outcomes of OFES (OGCM for the Earth Simulator)." *J. Earth Simulator* 1 (2004): 35-56.

Maurer, W.D., and T.G. Lewis. "Hash table methods." *ACM Computing Surveys (CSUR)* (ACM) 7, no. 1 (1975): 5-19.

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction.* 1993.

Meagher, D. "Geometric modeling using octree encoding." *Computer Graphics and Image Processing* (Elsevier) 19, no. 2 (1982): 129-147.

Meyers, Scott. *Effective c++: 50 specific ways to improve your programs and designs.* Amsterdam: Addison-Wesley Longman, 1992.

Michie, D. "Memo functions and machine learning." *Nature* 218, no. 1 (1968): 19-22.

Microsoft - MSDN Blogs. *C++ AMP in a nutshell.* 2011. http://blogs.msdn.com/b/nativeconcurrency/archive/2011/09/13/c-amp-in-a-nutshell.aspx (accessed 03 27, 2012).

Miller, George A. "The magical number seven, plus or minus two: some limits on our capacity for processing information." *Psychological Review* 63, no. 2 (1956): 81-97.

Mitchell, M. "Computation in cellular automata: A selected review." *Non-Standard Computation*, 1996: 95-140.

Molka, D., D. Hackenberg, R. Schone, and M.S. Muller. "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system." *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, 2009: 261-270.

Moore, E.F. *Sequential machines: Selected papers.* Addison-Wesley Pub. Co., 1964.

Muzy, A., and D. Hill. "What Is New With The Activity World View In Modeling And Simulation? Using Activity As A Unifying Guide For Modeling And Simulation." *Proceedings of the 2011 Winter Simulation Conference*, 2011: 13.

Muzy, A., et al. "Refounding of Activity Concept? Towards a Federative Paradigm for Modeling and Simulation." *Simulation: transactions of the society of modeling and simulation international, Society for Modeling and Simulation (SCS)*, 2012: 36.

Muzy, A., J.J. Nutaro, B.P. Zeigler, and P. Coquillard. "Modeling and simulation of fire spreading through the activity tracking paradigm." *Ecological Modelling* 219, no. 1-2 (2008): 212-225.

NVIDIA. *NVIDIA CUDA compute unified device architecture programming guide.* 2007.

—. *NVIDIA: Nvidia CUDA Compiler Driver NVCC.* 2008.

Ortoleva, P., P. Adhangale, S. Cheluvaraja, M. Fontus, and Z. Shreif. "Deriving principles of microbiology by multiscaling laws of molecular physics." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 2 (2009): 70-79.

Passerat, J., J. Caux, P. Siregar, C. Mazel, and D.R.C. Hill. "Warp-Level Parallellism: Enabling Multiple Replications In Parallel on GPU." *Proceding of the 2011 EuroPean Simulation and Modelling Conference*, 2011: 76-83.

Pivkin, I., P. Richardson, and G. Karniadakis. "Effect of red blood cells on platelet aggregation." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 2 (2009): 32-37.

Qiang-qiang, H., J. Shun-liang, X. Shao-ping, and D. Tian-wen. "The Simulation and Implementation of Discrete Particle System based on CUDA." *2011 International Conference on Business Computing and Global Informatization (BCGIN)*, 2011: 501-504.

Qutub, A.A., F. Mac Gabhann, E.D. Karagiannis, P. Vempati, and A.S. Popel. "Multiscale models of angiogenesis." *Engineering in Medicine and Biology Magazine, IEEE* (IEEE) 28, no. 2 (2009): 14-31.

Reuillon, R., D. RC Hill, C. Gouinaud, Z. El Bitar, V. Breton, and I. Buvat. "Monte Carlo simulation with the GATE software using Grid computing." *Proceedings of the 8th international conference on New technologies in distributed systems*, 2006: 201-204.

Robertson, John S. "How many recursive calls does a recursive function make?" *ACM Sigcse Bulletin* 31, no. 2 (1999): 60-61.

Rumpf, Martin, and Robert Strzodka. "Level set segmentation in graphics hardware." *International Conference on Image Processing*, 2001: 1103-1006.

Sander, E.A., T. Stylianopoulos, R.T. Tranquillo, and V.H. Barocas. "Image-based biomechanics of collagen-based tissue equivalents." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 3 (2009): 10-18.

Sanders, J., and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

Sato, T. "The earth simulator: Roles and impacts." *Nuclear Physics B-Proceedings Supplements* (Elsevier) 129 (2004): 102-108.

Sawyer, R.K. *Social emergence: Societies as complex systems.* Cambridge University Press, 2005.

Siregar, P. and Sinteff, J.P. and Julen, N. and Lebeux, P. "Spatio-temporal reasoning for multi-scale modeling in cardiology." *Artificial Intelligence in Medicine* (Elsevier) 10, no. 1 (1997): 41-57.

Siregar, P. Simulation of complex systems. Europe Patent EP20080290535. 12 16, 2009.

Siregar, P., JP Sinteff, N. Julen, and P. Le Beux. "An Interactive 3D Anisotropic Cellular Automata Model of the Heart." Edited by Elsevier. *Computers and biomedical research*, 1998: 323-347.

Sommerville, I. *Software engineering (6th edition).* 2001.

Southern, J., et al. "Multi-scale computational modelling in biology and physiology." *Progress in biophysics and molecular biology* (Elsevier) 96, no. 1-3 (2008): 60-89.

Steiglitz, K., I. Kamal, and A. Watson. "Embedding computation in one-dimensional automata by phase coding solitons." Edited by IEEE. *Computers, IEEE Transactions on* 37, no. 2 (1988): 138-145.

Taufer, M., R. Armen, Jianhan Chen, P. Teller, and C. Brooks. "Computational multiscale modeling in protein-ligand docking." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 2 (2009): 58-69.

Tawhai, M., J. Bischoff, D. Einstein, A. Erdemir, T. Guess, and J. Reinbolt. "Multiscale modeling in computational biomechanics." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 3 (2009): 41-49.

TheQtProject. *QtOpenCL : OpenCL in Qt Labs.* 2010. http://qt.gitorious.org/qt-labs/opencl (accessed 03 26, 2012).

Tocher, K.D. "The Art Of Simulation." *English Universities Press*, 1963.

Touraille, L., M.K. Traoré, and D.R.C. Hill. "Enhancing DEVS simulation through template metaprogramming: DEVS-MetaSimulator." *2010 Summer Simulation Multiconference*, 2010: 394-402.

Van der Steen, A.J., and J.J. Dongarra. "Overview of recent supercomputers." Department of Computer, University of Tennessee, 1996.

Vattulainen, I., and T. Ala-nissila. "Mission Impossible: Find a Random Pseudorandom Number Generator." *Computers in Physics* 9, no. 5 (1995): 500-510.

Von Neumann, John. *Theory of self-reproducing automata (edited and completed by A. W. Burks).* Urbana: University of Illinois Press, 1966.

White, R.J., G.C.Y. Peng, and S.S. Demir. "Multiscale modeling of biomedical, biological, and behavioral systems (Part 1)[Introduction to the special issue]." *Engineering in Medicine and Biology Magazine, IEEE* (IEEE) 28, no. 2 (2009): 12-13.

Wittenbrink, C., Kilgariff, E., and Prabhu, A. "Fermi gf100 gpu architecture." *IEEE Micro* 32, no. 2 (2011): 50-59.

Wolfram, S. *A new kind of Science.* Wolfram Media Champaign, 2002.

Wolfram, S. "Universality and complexity in cellular automata." Edited by Elsevier. *International Symposium on Physical Design* 10, no. 1-2 (1984): 1-35.

Woo, M., J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: the official guide to learning OpenGL, version 1.2.* Addison-Wesley Longman Publishing Co., Inc., 1999.

Woodfield, Scott N., Hubert E. Dunsmore, and Vincent Yun Shen. "The effect of modularization and comments on program comprehension." *International Conference on Software Engineering*, 1981: 215-223.

Zhou, Haiying, N. Lai, G. Saidel, and M. Cabrera. "Multiscale modeling of respiration." *IEEE Engineering in Medicine and Biology Magazine* 28, no. 3 (2009): 34-40.

Annexe

Annexe 1 : Graphiques des temps d'exécution du calcul du champ électrique.

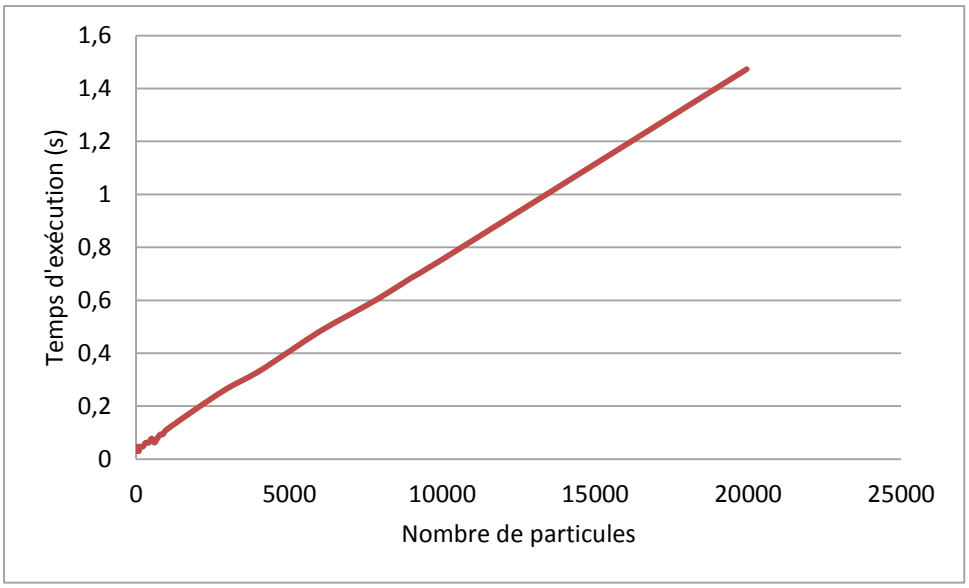


Figure 0-1 : Temps d'exécution du calcul du champ électrique sur GPU du nombre de particules (cube de 51 cellules de côté, 20 itérations).

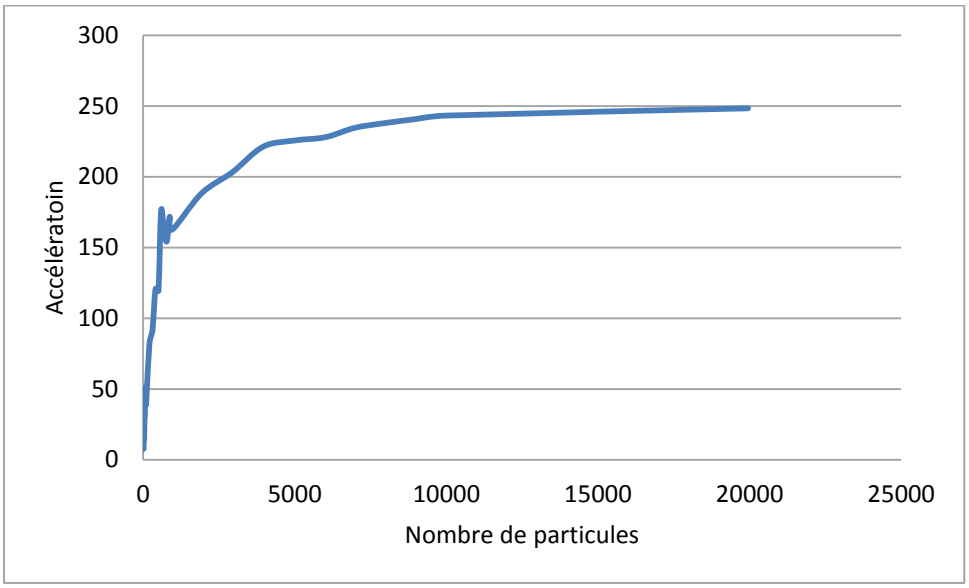


Figure 0-2 : Accélération du temps d'exécution du calcul du champ électrique réalisée grâce à la parallélisation sur GPU en fonction du nombre de particules (cube de 51 cellules de côté, 20 itérations).

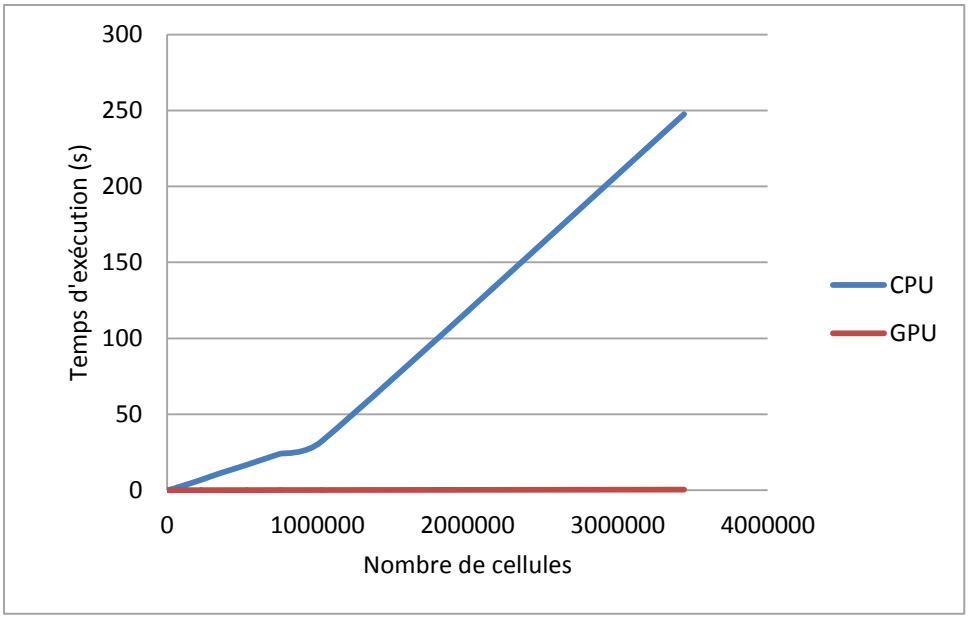


Figure 0-3 : Comparaison du temps d'exécution du calcul de champ électrique en fonction du nombre de cellules présentes sur CPU et sur GPU (200 particules, 20 itérations).

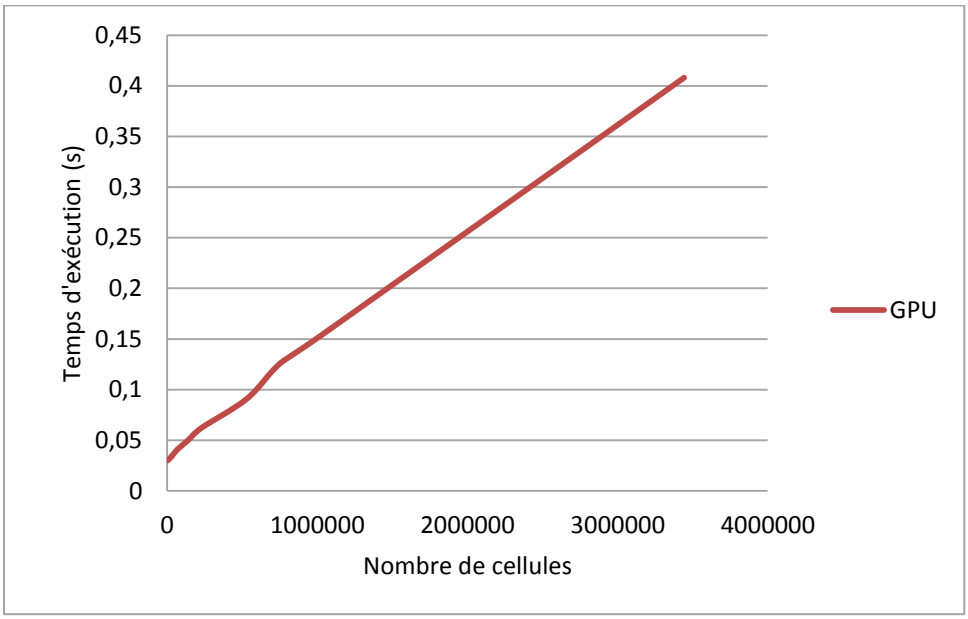


Figure 0-4 : Temps d'exécution du calcul de champ électrique en fonction du nombre de cellules présentes sur GPU (200 particules, 20 itérations).

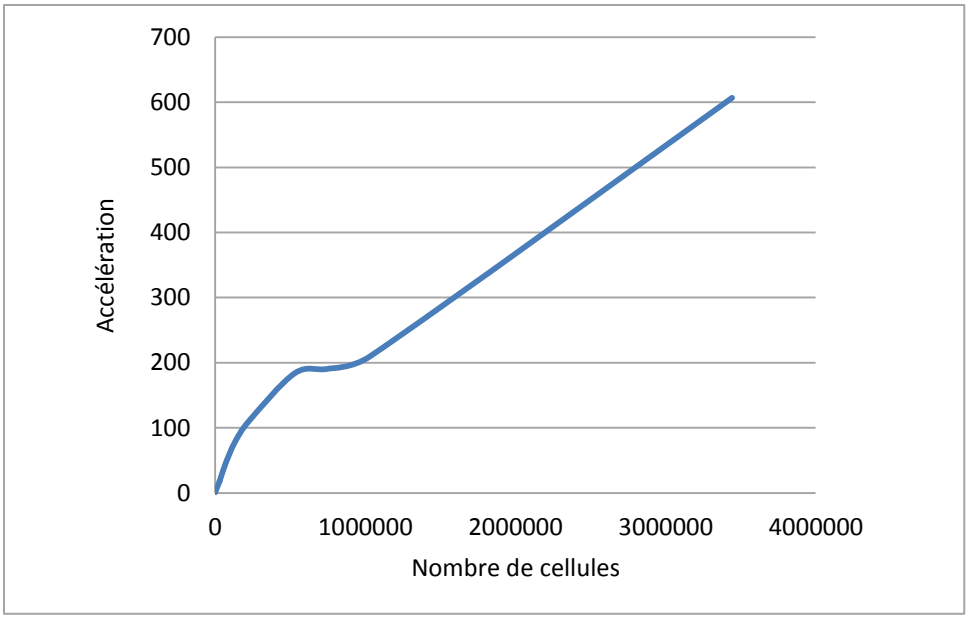


Figure 0-5 : Accélération du temps d'exécution du calcul du champ électrique réalisée grâce à la parallélisation sur GPU en fonction du nombre de cellules (200 particules, 20 itérations).