Accélération des calculs en chimie théorique : l'exemple des processeurs graphiques

Gaëtan Rubez

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

École Doctorale Sciences du Numérique et de l'Ingénieur

THÈSE

Pour obtenir le grade de :

Docteur de l'Université de Reims Champagne-Ardenne

Discipline : **Informatique**

Spécialité : **Chimie**

Présentée et soutenue publiquement par :

Gaëtan RUBEZ

le 6 décembre 2018

Accélération des calculs en Chimie théorique : l'exemple des processeurs graphiques

---

Sous la direction de :
Éric HÉNON, Professeur des Universités
Michaël KRAJECKI, Professeur des Universités

JURY

Éric HÉNON	Professeur à l'Université de Reims Champagne-Ardenne	Directeur
Michaël KRAJECKI	Professeur à l'Université de Reims Champagne-Ardenne	Co-directeur
Laetita JOURDAN	Professeur à l'Université de Lille	Rapporteur
Jean-Philip PIQUEMAL	Professeur à Sorbonne Université	Rapporteur
Xavier VIGOUROUX	Directeur du Centre pour l'Excellence en Programmation Parallèle	Examinateur
Jean-Matthieu ÉTANCELIN	Maître de Conférences à l'Université de Pau et des Pays de l'Adour	Examinateur
Jean-Charles BOISSON	Maître de Conférences à l'Université de Reims Champagne-Ardenne	Examinateur

---

Institut de Chimie Moléculaire de Reims UMR CNRS 7312, URCA
Centre de Recherche en Sciences et Technologies de l'Information et de la Communication EA 3804, URCA

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

École Doctorale Sciences du Numérique et de l'Ingénieur

THÈSE

Pour obtenir le grade de :

Docteur de l'Université de Reims Champagne-Ardenne

Discipline : **Informatique**

Spécialité : **Chimie**

Présentée et soutenue publiquement par :

Gaëtan RUBEZ

le 6 décembre 2018

Accélération des calculs en Chimie théorique : l'exemple des processeurs graphiques

---

Sous la direction de :
Éric HÉNON, Professeur des Universités
Michaël KRAJECKI, Professeur des Universités

JURY

Éric HÉNON	Professeur à l'Université de Reims Champagne-Ardenne	Directeur
Michaël KRAJECKI	Professeur à l'Université de Reims Champagne-Ardenne	Co-directeur
Laetita JOURDAN	Professeur à l'Université de Lille	Rapporteur
Jean-Philip PIQUEMAL	Professeur à Sorbonne Université	Rapporteur
Xavier VIGOUROUX	Directeur du Centre pour l'Excellence en Programmation Parallèle	Examinateur
Jean-Matthieu ÉTANCELIN	Maître de Conférences à l'Université de Pau et des Pays de l'Adour	Examinateur
Jean-Charles BOISSON	Maître de Conférences à l'Université de Reims Champagne-Ardenne	Examinateur

---

Institut de Chimie Moléculaire de Reims UMR CNRS 7312, URCA
Centre de Recherche en Sciences et Technologies de l'Information et de la Communication EA 3804, URCA

Remerciements

Je souhaite remercier mes directeurs de thèse Éric Hénon et Michaël Krajecki qui m'ont permis d'apprendre énormément scientifiquement et humainement au cours de ces trois années. Ce fut un réel plaisir de travailler au quotidien avec vous. Merci pour vos conseils avisés et ce partage d'expérience que je garde avec moi pour la suite. Et surtout merci pour le temps précieux que vous m'avez consacré.

Je tiens à remercier Xavier Vigouroux pour la confiance qu'il m'a accordé au cours de cette thèse, ainsi que pour l'opportunité professionnelle qu'il m'a permis d'obtenir grâce à son soutien. Ce fut un plaisir d'échanger avec toi et je n'oublie pas que je dois te recontacter pour un restaurant. Merci pour ta participation à mon Jury.

Je souhaite vivement remercier Jean-Charles Boisson pour son accompagnement au cours de ces trois années. Ses multiples conseils furent avisés et prirent toujours en compte les fluctuations émotionnelles qui sont à l'œuvre au cours d'un tel travail. Merci pour ta participation à mon Jury.

Je remercie Jean-Matthieu Étancelin pour son partage de compétences et d'expérience sur l'utilisation des cartes graphiques et des outils de profilage en général, de même pour ses conseils appréciables sur la ville de Grenoble, ainsi que pour sa participation appréciée à mon jury de thèse.

Je souhaite remercier mes rapporteurs de thèse Laetitia Jourdan et Jean-Philip Piquemal pour leur travail, leurs conseils et avis de valeur ainsi que pour cette décision appréciée de m'octroyer, pour mes travaux de recherche, le rang de docteur.

Merci à Fabien Bérini pour son aide et sa sympathie au centre de calcul ROMEO avec qui cette thèse a pu être réalisé dans de bonnes conditions.

Merci à Arnaud Renard et Florian Mazière pour avoir partagé de bons moments, un soutien quotidien ainsi qu'un bureau à l'université de Reims Champagne-Ardenne.

Merci à Julien Loiseau pour ces moments de décompression et de franche rigolade au cours d'événements qui ont été rudes.

Merci à Corentin Lefebvre car ce fut un plaisir de travailler avec toi et pour ton utilisation de cuNCI qui m'a permis d'identifier des bogues.

Merci à Christophe Jaillet pour sa bonne humeur. Ce fut un plaisir d'échanger avec toi au cours de ma thèse.

Christelle Anstet, Ida Lenclume, Rezak Ayad et Ségolène Buffet pour l'encadrement administratif qui fut efficace au cours de cette thèse. Mes remerciements vont à George-Emmanuel Moulard et Romain Dolbeau pour le repas partagé à la Ferme à Dédé mais surtout pour vos conseils techniques qui m'ont permis d'avancer dans la réalisation de mon travail.

Je remercie mes parents Armelle et Laurent Rubez, ma sœur et son conjoint Mélanie Rubez et Clément Lerat, mes grands-parents Anita Rubez, Daniel Rubez et Micheline Delpech, mes cousins, cousines, oncles et tantes, Alexis Carlu, Amandine Gommard, Catherine Carlu, Denis Delpech, Dimitri Delpech, Emilien Delpech, Léa Rubez, Lionel Gommard, Loïs Rubez, Nadège Delpech, Philipe Carlu, Rémi Carlu, Sarah Delpech, Simon Gommard, Sofian Rubez, Stéphane Gommard, Stéphanie Rubez, Thierry Rubez, Thomas Carlu, Xavier Rubez, Yaël Rubez, en y incluant Houssni Janah, Jean-Noël Gommard, Lyes Naji, Marise Gommard, Marjolaine Naji, Virginie Janah et ma belle-famille Alain Vérita, Cédric Carré, Fabrice Carré, Marie-Paule Carré-Vérita, Patricia Vérita, Renaud Carré et Yves Carré.

Je souhaite aussi remercier mes amis Antony Hoarau, Avila Pardon, Benjamin Bieri, Benjamin Diez, Benjamin Vo-dinh, Camille Mangelinck, Cécile Héliot, Charlotte Lecerf,

Corentin Conart, Elise Esquerre Pourtere, Fabien Lamret, Florine Ceccantini, Gin-Pamp Erlon, Jean-Baptiste Henry, Lbee Erlon, Lucie Ambroise, Lucille Aoustin, Nicolas Simons, Olivier Pico Renard, Pierre Boussagol, Quentin Leclerc, Romuald Ory, Sarah Ortonovi, Sébastien Dauvergne, Sevde Celik, Simon Mielcarek, Teddy Bruy, Théo Grillot, Victor Artufel.

Je remercie considérablement Cécile Aurore Carré pour son soutien quotidien durant cette épreuve. Un soutien qui je le souhaite durera une vie.

Résumé

Nous nous intéressons aux architectures *manycore* proposées par les cartes graphiques dans le cadre de la chimie théorique. Nous soutenons la nécessité pour ce domaine d'être capable de tirer profit de cette technologie. Nous montrons la faisabilité et les limites de l'utilisation de cartes graphiques en chimie théorique par le portage sur GPU de deux méthodes de calcul en modélisation moléculaire. Ces deux méthodes peuvent potentiellement être intégrées au programme de *docking* moléculaire AlgoGen. L'accélération et la performance énergétique ont été examinées au cours de ce travail.

Le premier programme NCIplot implémente la méthodologie NCI qui permet de détecter et de caractériser les interactions non-covalentes dans un système chimique. L'approche NCI se révèle être idéale pour l'utilisation de cartes graphiques comme notre analyse et nos résultats le montrent. Le meilleur portage que nous avons obtenu, a permis de constater des facteurs d'accélération allant jusqu'à 100 fois plus vite par rapport au programme NCIplot. Nous diffusons actuellement librement notre portage GPU : cuNCI.

Le second travail de portage sur GPU se base sur GAMESS qui est un logiciel complexe de portée internationale implémentant de nombreuses méthodes quantiques. Nous nous sommes intéressés à la méthode combinée DFTB/FMO/PCM pour le calcul quantique de l'énergie potentielle d'un complexe. Nous sommes intervenus dans la partie du programme calculant l'effet du solvant. Ce cas s'avère moins favorable à l'utilisation de cartes graphiques, cependant nous avons su obtenir une accélération.

Mots-clefs : Chimie théorique, Informatique, GPU, NCIplot, GAMESS, NCI, DFTB, FMO, PCM

Speed up computations in theoretical chemistry : The example of graphics processors

Abstract

In this research work we are interested in the use of the manycore technology of graphics cards in the framework of approaches coming from the field of Theoretical Chemistry. We support the need for Theoretical Chemistry to be able to take advantage of the use of graphics cards. We show the feasibility as well as the limits of the use of graphics cards in the framework of the theoretical chemistry through two usage of GPU on different approaches.

We first base our research work on the GPU implementation of the NCIplot program. The NCIplot program has been distributed since 2011 by Julia CONTRERAS-GARCIA implementing the NCI methodology published in 2010. The NCI approach is proving to be an ideal candidate for the use of graphics cards as demonstrated by our analysis of the NCIplot program, as well as the performance achieved by our GPU implementations. Our best implementation (VHY) shows an acceleration factors up to 100 times faster than the NCIplot program. We are currently freely distributing this implementation in the cuNCI program.

The second GPU accelerated work is based on the software GAMESS-US, a free competitor of GAUSSIAN. GAMESS is an international software that implements many quantum methods. We were interested in the simultaneous use of DTFB, FMO and PCM methods. The frame is less favorable to the use of graphics cards however we have been able to accelerate the part carried by two K20X graphics cards.

Keywords : Theoritical Chemistry, Computer Science, GPU, NCIplot, GAMESS, NCI, DFTB, FMO, PCM

Sommaire

Introduction générale

Le cadre général de ma thèse se situe à l'interface disciplinaire entre la chimie théorique et l'informatique. Les algorithmes étudiés, implémentés et accélérés ici proviennent de la chimie théorique tandis que les technologies mises en œuvre (en particulier les cartes graphiques) proviennent du domaine de l'informatique. Cette thèse étant à l'interface disciplinaire, l'encadrement des travaux a été réalisé par deux laboratoires rémois : l'ICMR (pour la chimie) et le CReSTIC (pour l'informatique). Ce travail de thèse est le produit d'un financement CIFRE par la filiale Bull du groupe ATOS. Et mon sujet de thèse a été proposé en 2015 au moment où le monde du HPC (*High Performance Computing*) voit se démocratiser des supercalculateurs hybrides utilisant des accélérateurs graphiques pour améliorer les performances des algorithmes hautement parallélisables.

L'hypothèse que nous soutenons dans ce travail de recherche est que la chimie théorique peut tirer profit de l'utilisation d'architectures dites *manycore*. Les architectures *manycore* mettent à disposition plusieurs centaines (voire milliers) de cœurs de calcul. Nous utiliserons des cartes graphiques comme exemple d'architecture *manycore* dans le cadre de cette thèse : elles sont au cœur du supercalculateur ROMEO.

Dans le premier chapitre, afin de justifier les motivations de ce travail, nous aborderons l'évolution des architectures de calcul et la tendance actuelle à tendre vers des architectures hybrides : processeurs multi-cœurs, accélérés par l'utilisation de cartes graphiques. En effet, la nécessité d'augmenter la puissance de calcul pour répondre à la demande croissante, combinée aux difficultés physiques pour augmenter la fréquence des processeurs, font que l'augmentation du nombre de cœurs de calcul s'avère nécessaire. Nous justifions l'évolution des architectures vers ce modèle de calcul hybride par des contraintes énergétiques limitantes. Ce passage d'architectures multi-cœurs à des architectures *manycore* nécessite de définir les notions fondamentales pour évaluer les performances d'un algorithme parallèle, comme par exemple l'accélération.

Afin de refléter l'évolution de l'algorithmique et des modèles de programmation, l'objectif du chapitre 2 est de fournir au lecteur toutes les notions informatiques pour pouvoir comprendre les portages réalisés par la suite dans les chapitres 4 et 5. Nous abordons les langages de programmation ainsi que les outils permettant d'analyser les performances d'une exécution, que nous utilisons dans ce travail de recherche. En effet, pour être efficace lorsque l'on souhaite accélérer un algorithme il est intéressant d'utiliser des outils d'analyse pour déterminer rapidement les fonctions qui constituent la majorité du temps d'exécution. Ce chapitre permet aussi de définir les modèles de parallélisme : algorithmique en mémoire partagée (mis en œuvre avec OpenMP) et par échange de messages ou à mémoire distribuée (mis en œuvre avec MPI). Ces modèles sont abordés car ils reflètent correctement une grande partie des codes parallèles actuels. L'augmentation du nombre de cœurs de calcul fait que le développeur doit être en mesure d'orchestrer ces milliers de cœurs de calcul (dans le cas du *manycore).* Nous définirons donc aussi les notions nécessaires pour comprendre le portage d'un algorithme sur cartes graphiques. Nous nous limiterons cependant aux notions pour utiliser les cartes graphiques NVIDIA par le biais

de l'interface de programmation CUDA[1]. Pour terminer ce chapitre 2 nous détaillerons l'architecture utilisée au cours de cette thèse grâce au centre de calcul ROMEO.

En constatant l'évolution des architectures HPC, il semble nécessaire d'être capable d'en tirer parti en chimie théorique. Comme il n'est pas possible de s'intéresser à toutes les pratiques de la chimie théorique, nous avons fait le choix de nous concentrer sur deux d'entre elles : l'approche NCI[2] (*Non-Covalent Interaction*) et l'approche combinée DFTB / FMO / PCM[3]. Néanmoins, un bref tour d'horizon des méthodes quantiques est présent dans le chapitre 3. La théorie de la fonctionnelle de la densité est abordée pour introduire la méthodologie DFTB (*Density-Functional Tight Binding*) qui est au centre de l'approche combinée DFTB / FMO / PCM étudiée. L'approche FMO (*Fragment Molecular Orital*) permet d'accélérer les calculs et tend à rendre le temps d'exécution linéaire avec la taille du système chimique étudié. L'approche PCM (*Polarizable Continuum Model*) qui permet de simuler l'effet d'un solvant sur le système chimique étudié est aussi abordée. L'approche NCI (*Non-Covalent Interaction*) est elle aussi évoquée : NCI est basée sur une approche topologique de la densité électronique et est utilisée pour localiser et caractériser dans l'espace les interactions dites non-covalentes (principalement entre les molécules). Ce chapitre se termine sur le modèle IGM (*Independent Gradient Model*) qui est apparu pendant ce travail de recherche et est similaire sur de nombreux point à l'approche NCI, comme nous le verrons.

Le travail de portage sur GPU réalisé sur NCIplot est décrit dans le chapitre 4. Dans un premier temps nous détaillerons l'existant : le code implémenté au sein du logiciel actuel NCIplot. Un jeu de systèmes chimiques est décrit. Ce jeu permet d'évaluer les performances obtenues par les différentes implémentations de l'approche NCI. Dans le but de réaliser une évaluation juste des performances des portages GPU un code de référence CPU est défini en analysant les performances des différentes installations CPU de l'approche NCI. En effet, il est important d'avoir une implémentation de référence CPU aussi performante que possible pour pouvoir faire une analyse des performances aussi juste que possible. L'ensemble des portages GPU réalisés de l'approche NCI est détaillé et met en avant certaines pratiques permettant d'obtenir de bonnes performances. Pour terminer ce chapitre, les performances obtenues sur les temps d'exécution ainsi que sur la consommation énergétique sont analysées.

Le second travail de portage sur GPU réalisé sur GAMESS (*General Atomic and Molecular Electronic Structure System*) est décrit dans le chapitre 5. GAMESS est un concurrent *open source* du logiciel GAUSSIAN, l'un des plus utilisés dans le domaine de la chimie quantique. Le logiciel GAMESS est de portée internationale et implémente un grand nombre de méthodes quantiques (RHF, ROHF, UHF, DFT, DFTB...). Nous nous sommes focalisés dans ce travail sur l'approche combinée DFTB / FMO / PCM. Nous décrivons le travail réalisé sur GAMESS dans l'ordre historique. Ce format permet de retranscrire les motivations et les réflexions plus amplement. Le premier objectif du travail réalisé sur GAMESS est de déterminer la fonction algorithmique pouvant tirer le plus profit d'une accélération par portage sur GPU. Une partie de ce chapitre décrit donc l'analyse réalisée pour obtenir cette fonction cible. Cette fonction cible est alors décrite, avant de considérer les portages GPU qui ont été réalisés. Pour terminer les performances des temps d'exécution et énergétique sont détaillés.

C'est donc par ces deux exemples de portage GPU (NCIplot décrit dans le chapitre 4 et GAMESS décrit dans le chapitre 5) que cette thèse tente d'apporter une contribution par l'utilisation de l'architecture *manycore* contemporaine (GPU) sur deux exemples d'approche du domaine de la chimie théorique (l'approche NCI et l'approche combinée DFTB / FMO / PCM). Nous avons obtenu une accélération allant jusqu'à un facteur 100

sur l'approche NCI en comparant notre meilleur portage GPU (utilisant deux cartes graphiques K20X) à l'implémentation CPU NCIplot (disponibles librement aux utilisateurs). Le travail sur le logiciel GAMESS permet quant à lui de montrer certaines difficultés pouvant se présenter lors de la mise en œuvre de cartes graphiques : comme la complexité pour identifier dans un code de plus d'un million de lignes les zones susceptibles d'être accélérées ainsi que la nécessité d'adapter l'approche à l'architecture utilisée.

Les approches NCI et IGM étant voisines, l'expérience acquise au cours du portage de NCI pourra aisément être bénéfique pour un portage sur GPU de IGM. De même, les temps obtenus sur NCI par l'accélération sur GPU permettent d'envisager la perspective d'une visualisation en temps réel d'interactions ligand-protéine.

Chapitre 1

L'évolution des architectures de calcul

Dans ce premier chapitre de thèse nous allons aborder l'évolution des architectures de calcul. Nous justifierons historiquement le passage des architectures mono-cœurs à des architectures multi-cœurs puis à l'ère actuelle des architectures *manycore*.

L'industrie informatique et en particulier des processeurs connait à ses débuts une progression constante, souvent connue sous le nom de loi de Moore. En 2004 cette industrie se heurte à une stagnation de la fréquence de calcul liée à des problématiques physiques provenant de la finesse de gravure, mais aussi à des difficultés de dissipation thermique. Pour contourner cette stagnation de la fréquence de calcul, les industriels décident d'augmenter le nombre de cœurs de calcul présents sur les processeurs.

Cette évolution vers des systèmes informatiques avec de plus en plus de cœurs de calcul amène la nécessité de réinventer les algorithmes pour pouvoir tirer partie du parallélisme apporté par ces systèmes.

L'objectif de ce chapitre est de justifier succinctement, par les contraintes énergétiques, l'augmentation du nombre de cœurs de calcul des architectures informatiques. Nous aborderons les implications du passage à des architectures multi-cœurs (jusqu'à une dizaine de cœurs de calcul) puis à des architectures dites "*manycore*" (des milliers de cœurs de calcul). Ce propos est illustré par un exemple concret (d'une somme de données) qui permet d'introduire les différentes approches (calcul en parallèle et communications) lors de l'utilisation de processeurs multi-cœurs et *manycore*. Cet exemple permet d'introduire la notion d'accélération que nous utiliserons dans les chapitres 4 et 5 pour évaluer nos portages sur cartes graphiques (architecture *manycore*). La qualité de nos portages s'appuie sur deux axes, l'extensibilité forte (*strong scaling*) et l'extensibilité faible (*weak scaling*), qui sont introduits dans ce chapitre par les lois d'Amdahl et de Gustafson.

1.1 L'arrivée des architectures multi-cœurs

Depuis les années 80 jusqu'en 2004 l'amélioration de la puissance de calcul est liée à l'augmentation de la fréquence d'horloge des processeurs. Le géant Intel sort en 2004 le processeur Pentium 4 avec, à l'origine, l'ambition d'une fréquence de 4 GHz. Le constat est alors fait : l'ère de l'amélioration de la puissance de calcul des processeurs par l'augmentation de la fréquence arrive à son terme. En effet, le refroidissement à air grand public ne permettra pas de faire dépasser les 3,8 GHz au Pentium 4.

Une solution doit alors être apportée afin de répondre à la demande croissante de puissance de calcul. Les industriels décident d'accroître le nombre de processeurs par puce

permettant d'augmenter la puissance de calcul sans augmenter la fréquence d'horloge. La diminution de la chaleur à dissiper provenant du passage des architectures mono-cœurs aux architectures multi-cœurs peut s'expliquer par le biais de l'équation suivante :

$$P = cV^2 f \tag{1.1}$$

où P représente la puissance nécessaire pour faire fonctionner un processeur : directement liée à la chaleur à dissiper. c représente la capacité électrique, V le voltage et f la fréquence de fonctionnement. Si nous considérons un processeur mono-cœur fonctionnant à une fréquence f et un processeur bi-cœur fonctionnant à une fréquence $\frac{f}{2}$. La capacité électrique passe alors à 2,2c et le voltage à 0,6V. Nous avons alors le même nombre d'instructions par seconde pour une puissance P sur le processeur mono-cœur et une puissance 0,396P pour le processeur bi-cœurs.

Nous avons donc là l'argument majeur en faveur de l'augmentation du nombre de cœurs de calcul au sein des processeurs : obtenir une puissance de calcul supérieure avec une consommation moindre. C'est donc pour continuer d'améliorer les performances dans un cadre énergétique restreint que les processeurs multi-cœurs apparaissent.

1.2 L'intérêt croissant des architectures *manycore*

Dans le domaine de la chimie théorique (domaine utilisant des ordinateurs pour simuler la matière), à ce jour, la majorité des algorithmes est capable de tirer profit de plusieurs cœurs de calcul d'un processeur (multi-cœur). De même il est fréquent, qu'une implémentation d'un algorithme de chimie théorique puisse tirer profit de la puissance disponible sur un *cluster* de calcul (ensemble d'ordinateurs indépendants appelés nœuds de calcul).

Les architectures HPC (*High Performance Computing*) évoluent vers des architectures dites *manycore*, comme les processeurs graphiques (GPU de l'anglais *Graphics Processing Unit*), car ils permettent une augmentation de la puissance de calcul via un accroissement du nombre de cœurs, limitant ainsi la chaleur à dissiper comme nous l'avons vu dans la section précédente.

Pour illustrer cette tendance, nous pouvons évoquer le classement TOP500[4] qui répertorie les supercalculateurs les plus puissants au monde. Au TOP500 de juin 2018, la machine américaine d'IBM (Summit) est en première position avec 4 608 nœuds de calcul chacun disposant de 6 cartes graphiques. Le cadre énergétique justifie cette évolution des architectures de calcul comme nous avons pu le voir précédemment dans la section 1.1. Nous pouvons voir cette contrainte énergétique illustrée par l'apparition du classement GREEN500 pour la première fois en novembre 2007, là où le TOP500 apparaît en juin 1993. Le GREEN500 est un classement qui liste les 500 supercalculateurs les plus efficaces énergétiquement (consommant le moins d'énergie par opération flottante).

Plusieurs différences fondamentales sont à énoncer lorsque l'on compare des architectures CPU et GPU. La différence principale est quantitative, le nombre de cœurs au sein d'un CPU est restreint (8 cœurs pour les processeurs utilisés dans cette thèse), tandis que pour un GPU le nombre de cœurs GPU (2 688 pour les cartes graphiques K20X utilisée dans cette thèse) se quantifie en milliers.

Chaque cœur de calcul d'un CPU est capable de réaliser des tâches complexes à haute fréquence (2,6 GHz dans notre cas) de manière indépendante des autres cœurs de calcul. Les cœurs de calcul d'un GPU travaillent en groupe pour exécuter des tâches similaires sur des données différentes. Le modèle qui nous intéresse dans cette thèse est celui fourni

dans le cadre de l'utilisation de cartes NVIDIA, par CUDA[5] (*Compute Unified Device Architecture*). Nous détaillons plus amplement CUDA dans la section 2.4 du chapitre suivant.

Une autre différence fondamentale entre ces architectures est qu'un CPU est autonome dans son fonctionnement alors qu'un GPU est dépendant, en général d'un CPU. En effet l'utilisation d'un GPU passe, en général, par le port PCIe (*Peripheral Component Interconnect Express*) et est alors contrôlable par un CPU. Dans cette configuration, le CPU est couramment nommé *hôte*, là où le GPU est appelé *device*. Il faut garder en tête qu'un CPU et un GPU fonctionnant conjointement possèdent tous deux leurs mémoires propres et que les transferts de données passent par le port PCIe, ce qui peut être une des difficultés lors de la recherche de performances. La version PCIe 3.0[6] (introduit en 2010) possède des débits allants de 984,6 Mo/s (x1) à 15,8Go/s (x16). La version PCIe 4.0[7] (introduite en 2017) est plus performante avec des débits allants de 1 969 Mo/s à 31,5 Go/s.

Notons qu'il existe deux grands constructeurs de GPU au monde, AMD et NVIDIA, et que dans le cadre de cette thèse nous nous intéressons uniquement aux cartes provenant du constructeur NVIDIA. Ce choix est déterminé par le matériel accessible sur le centre de calcul ROMEO.

Les cartes graphiques servent à l'origine pour obtenir les rendus 3D et tendent depuis plusieurs années à être utilisées dans de nombreux contextes plus généraux ce qui est communément appelé le *general-purpose processing on graphics processing units* (GPGPU). Une des premières tentatives de la communauté scientifique fut l'opération de multiplication matricielle[8]. Un des premiers programmes scientifiques à s'exécuter plus rapidement sur GPU que sur CPU est une implémentation de la factorisation LU[9]. Le GPGPU fonde la base du sujet de recherche de cette thèse, en appliquant les architectures des GPU aux algorithmes de la chimie théorique. Il est important pour les scientifiques d'être capable de tirer profit de l'évolution des architectures informatiques, afin de pouvoir améliorer la précision des résultats mais aussi de travailler sur des systèmes de plus en plus importants et complexes. Ce travail de thèse cherche à apporter une contribution dans ce sens, par l'utilisation des GPU sur des approches de la chimie théorique.

1.3 Les architectures informatiques selon Flynn

Les architectures des machines informatiques ont grandement évolué depuis le début du domaine et expliquer les différentes familles d'architectures est une tâche ardue. C'est pour cela que nous nous appuierons sur la taxonomie de Flynn[10], afin de faire le tour des modèles existants jusqu'à celui des processeurs graphiques qui nous intéresse particulièrement dans le cadre de cette thèse.

En 1972, Michael Flynn publie une classification des architectures d'ordinateur[10]. Cette classification comporte quatre catégories en fonction des flux de données et d'instructions.

- SISD : *Single Instruction on Single Data.* correspondant à l'architecture de von Neumann, c'est un ordinateur séquentiel sans parallélisme, que ce soit pour la mémoire ou les instructions. La figure 1.1 représente ce modèle.
- MISD : *Multiple Instructions on Single Data.* correspond à une architecture où une donnée est utilisée par plusieurs unités fonctionnelles pour des opérations différentes. L'exemple couramment évoqué pour cette architecture sont les ordinateurs de contrôle de vol de la navette spatiale américaine[11]. Un des principes

d'utilisation est de répliquer des tâches pour détecter et masquer les erreurs. La figure 1.2 représente ce modèle.

- SIMD : *Single Instruction on Multiple Data.* correspond à une machine qui exécute une seule instruction sur plusieurs données à la fois. La figure 1.3 représente ce modèle.
- MIMD : *Multiple Instructions on Multiple Data.* dans cette architecture, plusieurs processeurs exécutent des instructions différentes sur des données différentes. Plusieurs sous-modèles sont distinguables, mémoire partagée ou locale avec communications. La figure 1.4 représente ce modèle.

La plupart des architectures du TOP500[4] sont de la catégorie MIMD[12]. Des subdivisions peuvent être faites :

- SPMD : *Single Program on Multiple Data.* Plusieurs processeurs autonomes exécutent simultanément le même programme sur différentes données. SPMD est le modèle de programmation parallèle le plus courant.
- MPMD : *Multiple Programs on Multiple Data.* Plusieurs processeurs autonomes exécutent au moins deux programmes indépendants. Ce système se trouve classiquement dans le modèle maître/esclaves, où un programme gère globalement le travail à effectuer en l'envoyant aux esclaves, les esclaves retournant le résultat au maître.

Nous nous intéressons à une dernière catégorie majeure dans le cadre de cette thèse :

- SIMT : *Single Instruction on Multiple Threads.* Ce modèle est introduit par NVIDIA et disponible depuis 2006[13] grâce aux puces GPU G80. SIMT correspond à la combinaison du modèle SIMD avec du *multithreading*. Ces processeurs utilisent plusieurs processus légers simultanément sur différentes données. C'est ce modèle qui nous intéresse principalement dans le cadre de cette thèse. Son utilisation par le biais de la pile logicielle mise à disposition par NVIDIA est décrite dans la section 2.4.

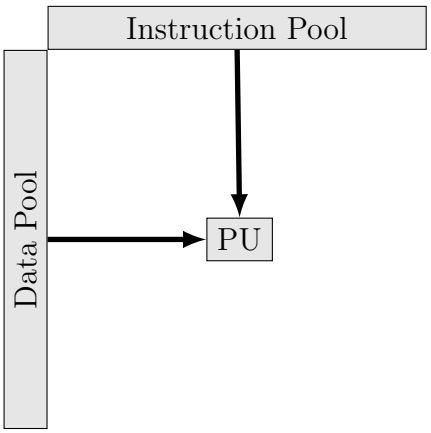


FIGURE 1.1 – SISD : *Single instruction stream single data stream*

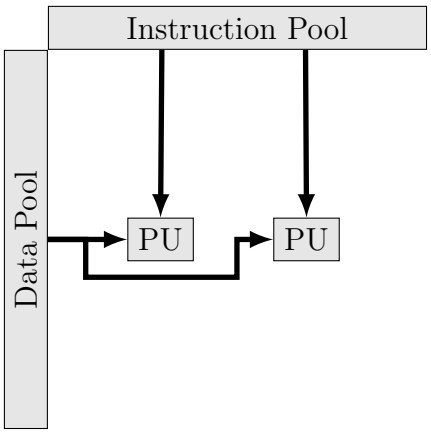


FIGURE 1.2 – MISD : *Multiple instructions stream single data stream*

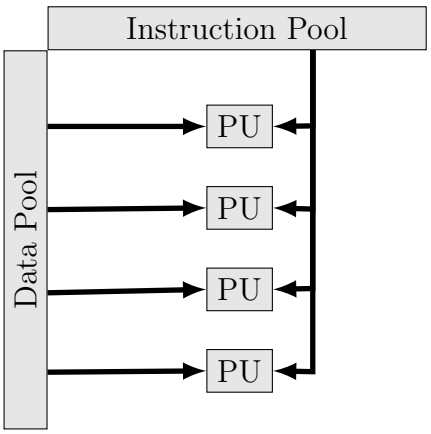


FIGURE 1.3 – SIMD : *Single instruction stream multiple data streams*

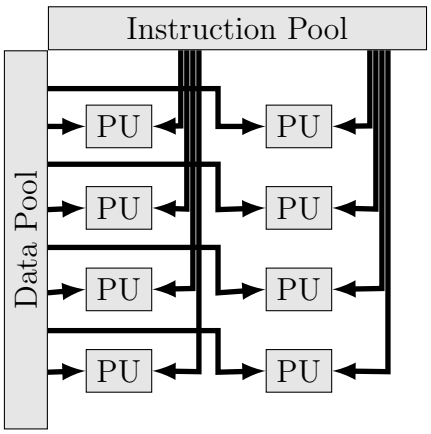


FIGURE 1.4 – MIMD : *Multiple instructions stream multiple data streams*

Nous pouvons voir grâce la Taxonomie de Flynn la diversité et l'évolution des architectures dans le domaine de l'informatique. Pour cette thèse, nous allons dans la section suivante nous intéresser à un exemple concret afin d'illustrer les différences fondamentales entre l'utilisation d'un unique cœur de calcul contre plusieurs cœurs de calcul.

Illustration des implications sur un exemple concret

Nous allons dans cette section nous intéresser à l'exemple de sommation de données pour illustrer les implications de l'utilisation de plusieurs cœurs de calcul. Pour simplifier le propos, nous omettrons dans cette section de parler de la précision des calculs.

Approche séquentielle

Nous souhaitons par exemple, dans un premier temps, sommer les nombres de 0 à 15 stockés en mémoire. Une approche classique avec un processeur mono-cœur est de parcourir à l'aide d'une boucle les 16 éléments de la mémoire (allant de 0 à 15), et de réaliser la somme terme à terme dans une variable qui contiendra le résultat final à la fin de la boucle. Ce traitement séquentiel est illustré dans la figure 1.5. Un seul cœur de calcul est utilisé pour toute la somme.

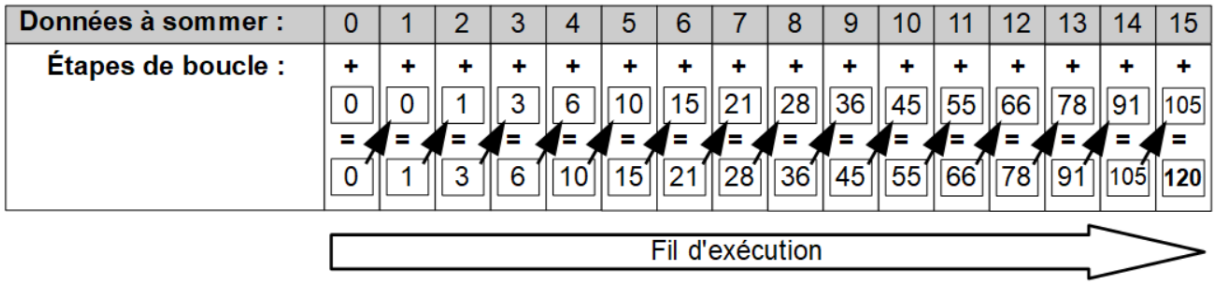


FIGURE 1.5 – Illustration d'une somme de 0 à 15 réalisée avec un processeur mono-cœur.

Approche parallèle

La mise en œuvre conjointe des ressources parallèles demande une expertise particulière de la part des développeurs pour pouvoir tirer profit des multiples cœurs de calcul disponibles dans ces architectures.

Nous pouvons envisager pour notre exemple d'augmenter le nombre de cœurs utilisés. La figure 1.6 illustre une somme des nombres de 0 à 15 avec 8 cœurs. Chaque calcul intermédiaire de la somme peut être calculé par un cœur du processeur. L'augmentation du nombre de cœurs permet d'augmenter la puissance de calcul. Nous pouvons voir dans notre exemple (illustré figure 1.6) que des communications pour les résultats intermédiaires sont nécessaires. En effet, à chaque étape des résultats doivent être communiqués pour préparer l'étape suivante.

Nous pouvons aussi voir qu'à chaque étape, le nombre de cœurs capables d'effectuer des calculs en parallèle diminue. Ainsi le gain de temps via le parallélisme (engendré par l'utilisation de plusieurs cœurs) diminue aussi proportionnellement.

Données à sommer :	0 + 1	2 + 3	4 + 5	6 + 7	8 + 9	10 + 11	12 + 13	14 + 15
Étape 1 :	Cœur 1	Cœur 2	Cœur 3	Cœur 4	Cœur 5	Cœur 6	Cœur 7	Cœur 8
Données restantes:	1 + 5		9 + 13		17 + 21		25 + 29	
Étape 2 :	Cœur 1		Cœur 2		Cœur 3		Cœur 4	
Données restantes:	6 + 22				38 + 54			
Étape 3 :	Cœur 1				Cœur 2			
Données restantes:	28 + 92							
Étape 4 :	Cœur 1							
Résultat :	**120**							

FIGURE 1.6 – Illustration d'une somme de 0 à 15 réalisée avec un processeur possèdant huit cœurs.

Dans la réalité, un grand nombre de cas de figures peuvent arriver. Il est fréquent que le nombre de calculs à réaliser soit très nettement supérieur aux ressources de calcul utilisables. Ce cas de figure est en général favorable à l'utilisation de plusieurs cœurs de calcul, car le parallélisme tire son plein potentiel dans les phases où tous les cœurs de calcul peuvent calculer simultanément.

Cet exemple, plutôt simple, permet d'appréhender certains aspects de la programmation parallèle, comme le fait de diminuer le temps d'exécution global par l'utilisation de plusieurs cœurs simultanément ou encore la nécessité de pouvoir réaliser des communications entre les cœurs de calcul.

Comme nous avons pu le constater dans l'exemple de la somme de cette section, le nombre de cœurs mis en œuvre influence la performance d'un algorithme parallèle. Nous allons dans la section suivante définir des notions importantes lorsque l'on souhaite comparer des implémentations pouvant être sur des architectures différentes.

1.4 Mesure de l'efficacité du parallélisme

Une notion capitale lorsque l'on souhaite évaluer un algorithme parallèle par rapport à un algorithme séquentiel est l'accélération communément appelée *speed-up*. Le rapport

entre le temps t_{seq} de l'algorithme séquentiel optimal par le temps $t_{par}(n_p)$ de l'algorithme parallèle étudié (en fonction du nombre de processeurs n_p) définit l'accélération de algorithme parallèle, soit :

$$Acc_{par} = \frac{t_{seq}}{t_{par}(n_p)} \tag{1.2}$$

Cette définition amène plusieurs conséquences notables :

- L'accélération obtenue dépend du nombre de processeurs utilisés pour l'évaluation de l'algorithme parallèle.
- L'accélération maximale d'un algorithme parallèle est égale au nombre de processeurs utilisés pour évaluer cet algorithme.
- L'algorithme séquentiel doit être le plus optimisé pour une comparaison honnête.

Nous utilisons pour la suite une définition similaire de l'accélération pour évaluer un portage GPU. En calculant le rapport entre, le temps $t_{par}(n_p)$ de l'algorithme parallèle optimal, par le temps $t_{gpu}(n_{gpu})$ de l'algorithme du portage GPU étudié (dépendant du nombre n_{gpu} cartes graphiques utilisées), soit :

$$Acc_{gpu} = \frac{t_{par}(n_p)}{t_{gpu}(n_{gpu})} \tag{1.3}$$

L'accélération obtenue dépend donc de l'algorithme parallèle de référence, ainsi que du nombre de cœurs utilisés pour exécuter cet algorithme de référence. De même, la carte graphique utilisée pour évaluer le temps d'exécution de l'algorithme GPU influence l'accélération. Dans cette thèse nous avons aussi des accélérations qui utilisent des temps d'algorithmes accélérés par plusieurs GPU.

En général, un code parallélisé contient encore des parties séquentielles. La loi d'Amdahl[14] est utile dans ce cadre : lorsqu'il s'agit d'estimer la performance pouvant être obtenue par l'utilisation de ressources parallèles.

Loi d'Amdahl

La loi d'Amdahl[14] énoncée en 1967 par Gene AMDAHL permet de prédire l'accélération théorique d'une exécution lors de l'utilisation de ressources parallèles. Une formulation connue de l'accélération de la loi d'Amdahl est :

$$Acc_{\text{Amdahl}} = \frac{1}{(\text{f} + \frac{1-\text{f}}{n_p})} \tag{1.4}$$

Comme précédemment n_p est le nombre de cœurs de calcul utilisés pour traiter les parties parallèles du code. f est la fraction du programme qui reste exécuté en séquentiel (par un seul cœur de calcul).

Concrètement, la loi d'Amdahl[14] s'intéresse à l'estimation de l'accélération sur un problème dont la taille des données est fixe et où seule une portion de l'algorithme est parallélisable.

La figure 1.7 montre que pour un problème de taille fixe avec une portion parallélisable, seule cette portion parallélisable tire profit de l'ajout de cœurs de calcul. La portion séquentielle est incompressible lors de l'ajout de cœurs de calcul.

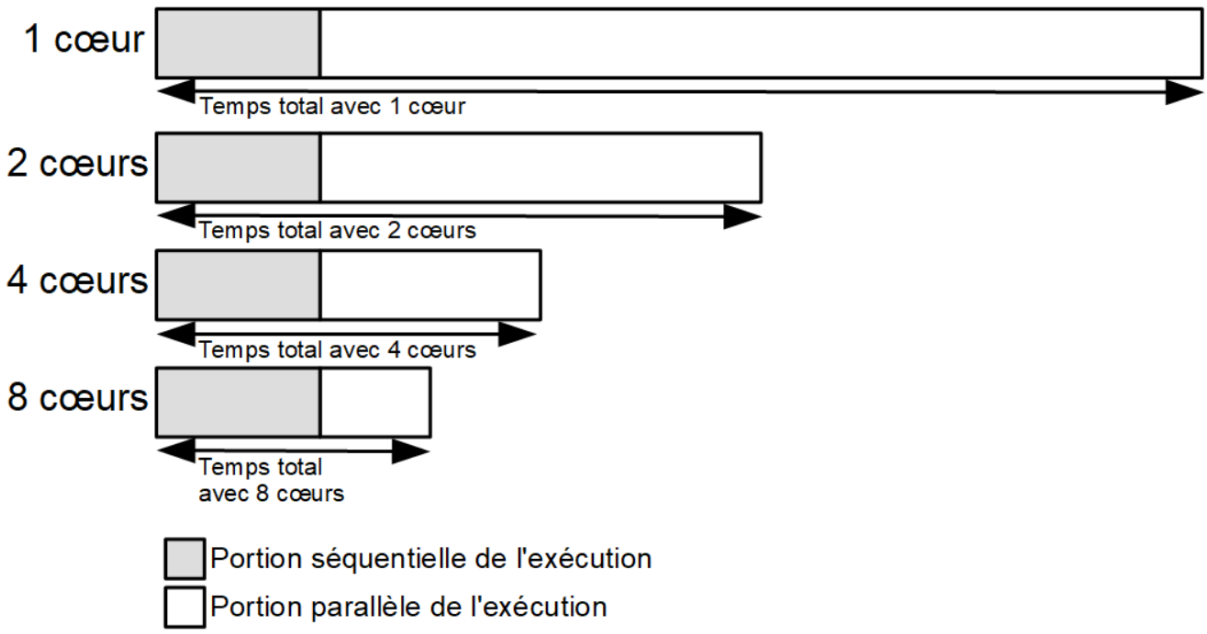


FIGURE 1.7 – Illustration de la loi d'Amdahl

La loi d'Amdahl indique donc que pour un code avec 95% de ses parties parallélisables, le gain maximal que l'on peut obtenir (par rapport à l'utilisation d'un seul cœur de calcul) est une accélération d'un facteur 20 (sur le temps total), peu importe le nombre de cœurs mis en œuvre pour le calcul parallèle. Ce phénomène est illustré par la figure 1.8 avec des codes possédant 95%, 90%, 75% et 50% de parties parallélisables.

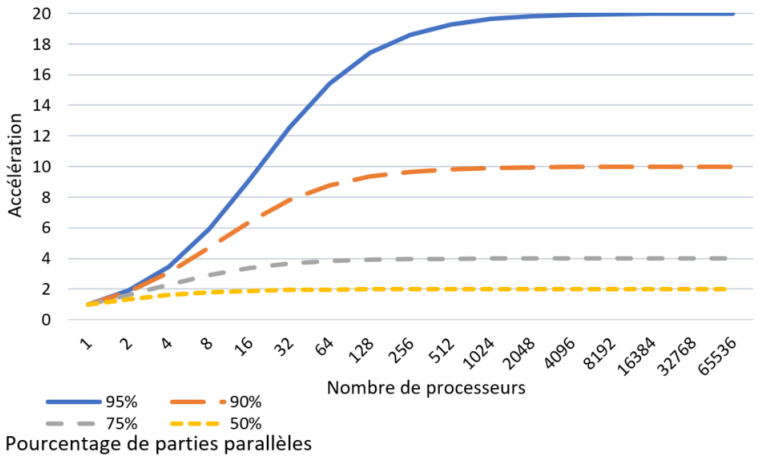


FIGURE 1.8 – Loi d'Amdahl : accélération maximale possible en fonction du nombre de cœurs utilisés pour quatres exemples

La loi d'Amdahl fait le constat que lorsque la quantité de données à traiter ne peut pas être augmentée : l'accélération que l'on peut obtenir est bornée, peu importe le nombre de cœurs mis en œuvre.

Dans notre cas, en chimie théorique, le problème peut en général être augmenté, soit dans la précision du résultat mais surtout dans la taille du système traité. Ce qui fait

que la loi d'Amdahl ne semble illustrer qu'une partie de ce que l'on peut attendre de l'utilisation en parallèle de plusieurs cœurs de calcul.

Loi de Gustafson

Pour tenter de voir jusqu'où peut nous mener l'accroissement du nombre de cœurs des architectures contemporaines, nous nous intéressons maintenant à la loi de Gustafson. John L. Gustafson réévalue en 1988[15] la loi d'Amdahl.

La loi de Gustafson permet de prédire l'accélération théorique que l'on peut obtenir par l'utilisation de plusieurs processeurs, quand il est possible d'augmenter la quantité de données à traiter. C'est un cas qui correspond mieux à cette thèse, car dans le domaine de la chimie théorique les données à traiter peuvent, en général, être augmentées.

Une formulation classique de l'accélération de la loi de Gustafson est :

$$Acc_{\text{Gustafson}} = \text{f} + (1 - \text{f})n_p \tag{1.5}$$

Comme précédemment n_p est le nombre de cœurs de calcul utilisés pour traiter les parties parallèles du code et f est la fraction du programme qui reste exécuté en séquentiel.

Concrètement, dans le cas de la loi de Gustafson la performance d'un algorithme est évaluée pour un temps fixe. Pour obtenir ce temps fixe, la quantité de données calculées par les cœurs augmente proportionnellement à leur nombre. La figure 1.9 illustre donc que le temps d'exécution global reste le même, peu importe le nombre de cœurs de calcul utilisés. En revanche, la quantité de données traitée lors de la portion parallèle augmente. Cela permet de montrer un aspect négligé dans la loi d'Amdahl, à savoir que l'augmentation du nombre de cœurs peut permettre de traiter plus de données pour un même temps. Ce qui implique pour la chimie théorique la possibilité de traiter des systèmes de tailles plus importantes.

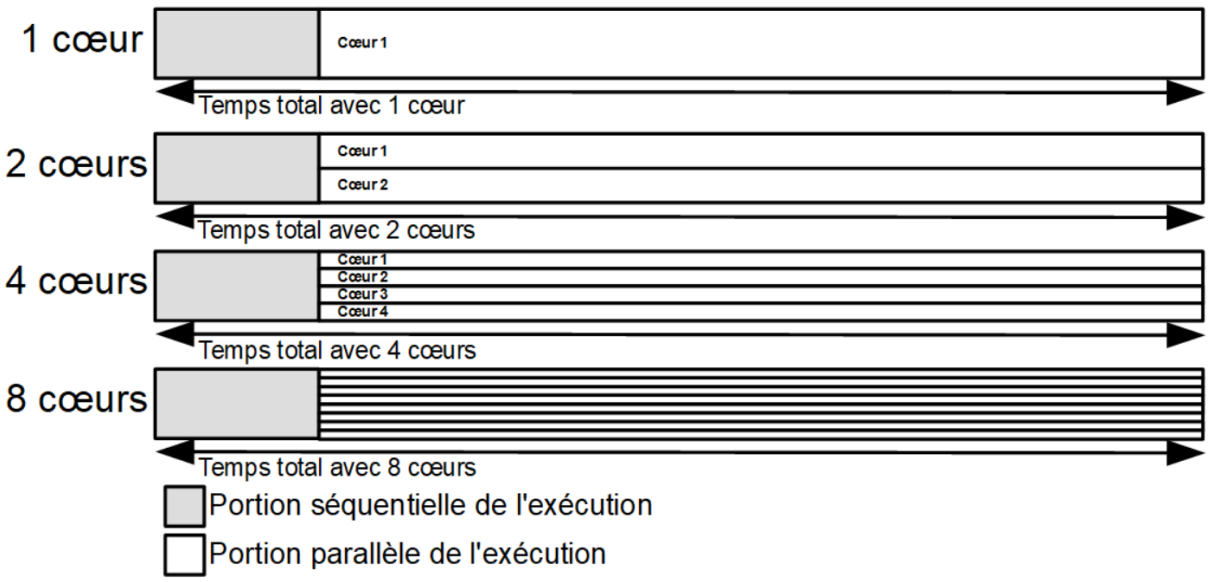


FIGURE 1.9 – Illustration de la loi de Gustafson

La figure 1.9 montre que la partie séquentielle reste la même. La portion parallèle fait s'exécuter plusieurs cœurs de calcul simultanément qui traitent un même volume de

données. Ce traitement implique que le temps global reste fixe, peu importe la quantité de cœurs de calcul utilisée.

La loi de Gustafson (illustrée figure 1.10) permet de voir que l'augmentation du nombre de cœurs de calcul permet, à temps fixe, de traiter une plus grande quantité de données et donc d'accélérer l'algorithme.

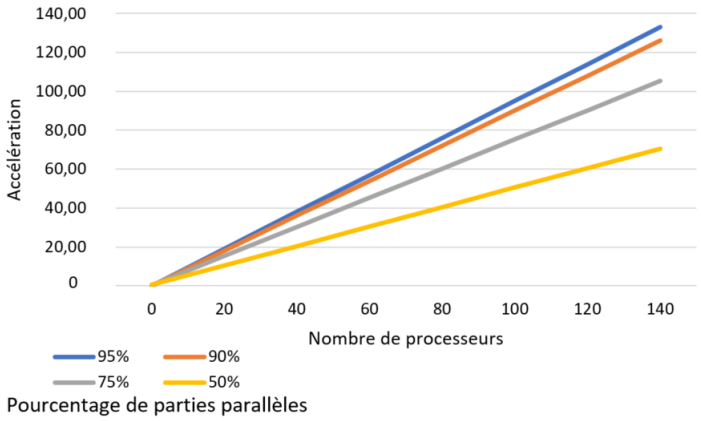


FIGURE 1.10 – Loi de Gustafson : accélération théorique possible en fonction du nombre de cœurs utilisés pour quatres exemples.

Dans le cas de la loi de Gustafson, l'accélération n'est pas bornée en fonction du nombre de cœurs de calcul.

Les lois d'Amdahl et de Gustafson nous permettent de voir qu'il existe plusieurs approches pour évaluer un algorithme parallèle dans le domaine du calcul haute performance. Ces lois permettent de définir deux approches principales :

- **l'extensibilité forte (*strong scaling*)** s'intéresse aux variations de temps d'exécution en fonction du nombre de processeurs utilisés pour un problème de taille fixe. L'extensibilité forte s'apparente donc à la loi d'Amdahl.
- **l'extensibilité faible (*weak scaling*)** s'intéresse aux variations de temps d'exécution en fonction du nombre de processeurs utilisés pour un problème dont la taille est fixe par processeur. L'extensibilité faible s'apparente donc à la loi de Gustafson.

1.5 Conclusion

La notion d'accélération utilisée pour évaluer nos portages GPU des chapitres 4 et 5 est donc la suivante :

$$Acc_{gpu} = \frac{t_{par}(n_p)}{t_{gpu}(n_{gpu})} \tag{1.6}$$

avec

- $t_{par}(n_p)$ le temps de l'algorithme CPU parallèle optimal exécuté avec n_p le nombre de cœurs de calcul mis en œuvre.
- $t_{gpu}(n_{gpu})$ le temps d'exécution du portage GPU, qui peut être réalisé avec une ou deux cartes graphiques dans le cadre de cette thèse.

Les deux notions d'extensibilité (forte et faible) nous intéressent pour ce travail de recherche car lorsqu'un portage sur GPU est réalisé, il est intéressant de voir comment il se comporte sur un problème de taille fixe (extensibilité forte) mais aussi comment le portage supporte la charge (extensibilité faible) quand la quantité de calcul est augmentée.

En général, ces notions s'entremêlent. Le premier point qui nous intéresse est le moment où l'utilisation d'une carte graphique devient "rentable" et accélère le calcul. L'autre point qui nous intéresse est le moment où le facteur d'accélération stagne pour un matériel donné. L'évaluation de la performance d'un algorithme parallèle peut donc être complexe à analyser à cause de ces différentes considérations.

Concluons ce premier chapitre sur le fait que les architectures de calcul évoluent dans un cadre énergétique contraignant et pour continuer d'augmenter la performance, un angle d'attaque est d'augmenter le nombre de cœurs de calcul par processeur. Cette augmentation du nombre de cœurs de calcul par processeur nécessite un changement dans les méthodes de programmation afin de tirer profit des ressources parallèles disponibles sur ces machines, induisant la nécessité de revisiter des algorithmes de la chimie théorique.

Le chapitre suivant s'intéresse aux méthodes multi-cœurs couramment mis en œuvre dans le monde du calcul haute performance et particulièrement en chimie théorique. Le second chapitre traite aussi de la mise en œuvre de GPU par le biais de CUDA, qui est utilisé dans cette thèse.

Chapitre 2

La conception d'applications parallèles

Les architectures multi-cœurs et *manycore* impliquent une algorithmique et une mise en œuvre qui différent du développement séquentiel.

Dans un premier temps, ce chapitre aborde les langages de programmation rencontrés au cours de ce travail de recherche, à la fois pour implémenter les portages GPU mais aussi des codes de chimie théorique existants que nous avons étudiés.

Ces codes de chimie théorique tirent en général profit du parallélisme multi-cœurs. Comme évoqué dans le chapitre précédent, l'accélération de nos portages GPU est définie par rapport à un code CPU parallèle optimal. Nous aborderons dans ce chapitre deux modèles classiques de parallélisation.

La performance d'un code dépend de nombreux critères parmi lesquels la chaîne de compilation qui n'est pas à sous-estimer. En effet, le choix et la configuration du compilateur peut largement impacter l'efficacité du programme. Nous indiquerons pour cette raison les différents compilateurs utilisés au cours de cette thèse.

Débutons ce chapitre avec les langages de programmations rencontrés au cours de ce travail de recherche.

2.1 Langages pour le calcul scientifique

Un langage de programmation est un langage formel qui utilise un lot d'instructions produisant des sorties diverses et variées.

En comparaison au langage courant, un langage de programmation se compose d'une structure particulière composée d'un vocabulaire et de règles de grammaire qui permettent de décrire les structures de données manipulées par le matériel informatique et les manipulations à effectuer pour réaliser l'algorithme souhaité. Les langages de programmation permettent, en somme, aux programmeurs de communiquer avec la machine, afin d'exécuter les opérations nécessaires à la réalisation de l'algorithme escompté. Cette thèse utilise uniquement les langages C et Fortran qui sont des langages compilés, nous n'évoquerons donc pas la notion de langage interprété.

La compilation est une étape clé qui permet à partir de fichiers sources d'obtenir un ou plusieurs exécutables pour une architecture cible. Le compilateur part du code source écrit dans le langage de programmation adapté, compréhensible par une personne érudite, et le traduit en un code binaire incompréhensible par l'homme et, a contrario, fortement adapté aux machines.

En général, les langages compilés permettent de générer des binaires qui s'exécutent plus rapidement que les langages interprétés cependant il faut recompiler le code pour chaque architecture cible. Il semble alors cohérent d'utiliser des langages compilés dans le milieu scientifique qui nous intéresse, où la rapidité est souvent au cœur de la problématique.

Dans cette thèse nous abordons le langage C car ce dernier est efficace[16] puisque de bas niveau, populaire au sein de la communauté scientifique et permettant l'utilisation de CUDA qui est aussi écrit en C. Le Fortran est aussi abordé dans cette thèse car les codes de références qui implémentent les algorithmes de chimie théorique qui nous intéressent dans la suite sont en totalité ou en partie écrits en Fortran.

Le C[16] est inventé en 1972 par Dennis RITCHIE et Ken THOMPSON, c'est un langage de programmation, compilé, généraliste, impératif, procédural, structuré et fondé sur un standard ouvert. Le C est un langage dit de bas niveau, fortement utilisé dans tous les domaines et généralement considéré comme étant une base solide pour toute personne souhaitant comprendre en détail les bases de la programmation informatique.

Ce langage a pour propriété d'utiliser des variables typées ainsi que de permettre l'utilisation de pointeurs mémoires permettant la maximisation des performances.

Un des intérêts, pour cette thèse, du langage C est d'être capable d'utiliser les cartes graphiques de NVIDIA en utilisant CUDA qui est écrit dans ce langage. De plus, l'aide[5] disponible à l'utilisation de CUDA est principalement illustrée grâce à des exemples utilisant le langage C. Notons aussi, de manière subsidiaire, que le support disponible grâce à la communauté autour du langage C est important.

Fortran[17] de *FORmula TRANslator*, à l'origine développé par IBM dans les années 1950 est un langage de programmation, compilé, généraliste et impératif. Fortran apparaît toujours comme une référence dans le monde du *High Performance Computing* (HPC), par exemple par le programme *High-Performance Linpack*[18] (HPL) implémenté en Fortran, qui sert d'évaluation pour classer les supercalculateurs du TOP500[4].

Le Fortran est abordé dans cette thèse car les implémentations des algorithmes de références qui nous intéressent sont en totalités ou en partie en Fortran.

Différences entre les langage C et Fortran

Nous allons prendre le temps maintenant d'évoquer les différences majeures entre le langage C et Fortran car dans le chapitre 5, le programme GAMESS utilise simultanément ces deux langages. Les portages GPU réalisés dans cette partie utilisent conjointement Fortran, C et CUDA.

Les différences de syntaxe entre les deux langages ne présentent que peu d'intérêts, nous les passerons donc sous silence. Ajoutons cependant que le langage C est généralement sensible à la casse contrairement au langage Fortran.

Les tableaux sont stockés en mémoire sous forme uni-dimensionnelle pour les deux langages. En revanche le stockage en mémoire des tableaux diffère entre les deux langages, de même l'indice de départ diffère, en général. En Fortran, l'indice de départ est nativement 1 et le stockage des tableaux est en colonne. En C, l'indice de départ est 0 et le stockage des tableaux est en ligne. Par exemple la matrice A de dimension (2,3) est stockée en fortran comme dans la figure 2.1 et en C comme dans la figure 2.2.

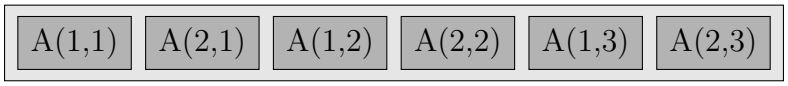


FIGURE 2.1 – Stockage mémoire d'un tableau de dimension (2,3) en fortran.

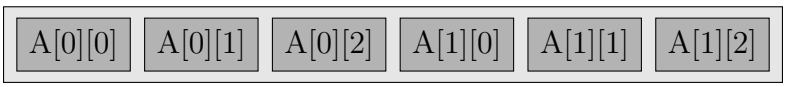


FIGURE 2.2 – Stockage mémoire d'un tableau de dimension (2,3) en C.

Une mauvaise utilisation du stockage amène directement à une perte d'efficacité car le nombre d'accès mémoire augmente dû à l'absence de contiguïté des accès.

Ensuite, nativement, le passage des arguments en C se fait par valeur tandis qu'en Fortran le passage des arguments se fait par référence.

Les différents types de variables sont, en général, transposables d'un code à l'autre, le tableau 2.1 montre les types appropriés pour l'interopérabilité entre Fortran et C.

Fortran 77	C
BYTE var	char var
CHARACTER var	unsigned char var
CHARACTER*n var	unsigned char var[n]
DOUBLE PRECISION var	double var
REAL var	float var
REAL*4 var	float var
REAL*8 var	double var
REAL*16 var	long double var
INTEGER var	int var
INTEGER*2 var	short var
INTEGER*4 var	int var

Tableau 2.1 – Tableau de comparaison des types entre les langages Fortran et C.

Dans ce travail de recherche nous avons rencontré dans deux cas (NCI et GAMESS) ces langages. Tout d'abord avec le logiciel NCIplot (chapitre 4) qui est entièrement écrit en Fortran, puis l'approche NCI qu'implémente le logiciel NCIplot a été réécrite en C au cours de cette thèse. L'utilisation de multiples langages ajoute un niveau de complexité qui rend plus difficile à la fois la compréhension du code et ses modifications.

Le C et le Fortran étant des langages compilés, la sous-section suivante définit les compilateurs utilisés au cours de cette thèse pour compiler les codes des algorithmes de chimie théorique rencontrés.

Choix de compilation

Choisir un compilateur n'est pas une étape anodine lorsque l'architecture ciblée est connue par avance. De plus, une bonne connaissance des capacités de chaque compilateur permet un ratio gain/investissement souvent très intéressant comparé à l'optimisation seule d'un code source. C'est pour cela que nous avons utilisé les compilateurs suivants :

- La *GNU Compiler Collection* est utilisée, gcc pour le langage C, gfortran pour le langage Fortran et g++ pour les parties de GAMESS utilisant du C++. Une référence car libre d'accès.
- Les compilateurs `icc` et `ifort` développés par Intel sont utilisés ici, car le centre de calcul ROMEO est équipé en matériel Intel. Nous pouvons présupposer de

meilleures performances pour ce compilateur développé spécifiquement pour le matériel Intel.
- PGI (Portland Group, Inc.) *compilers* sont aussi utilisés au cours de cette thèse. Ces compilateurs payants permettent d'obtenir des performances sur une large gamme de matériel. Le centre de calcul ROMEO dispose de ces compilateurs.
- NVIDIA CUDA compiler (`nvcc`) est utilisé pour réaliser la compilation des sources utilisant CUDA permettant la mise en œuvre des GPU de chez NVIDIA.

Les performances d'un code dépendent du compilateur et des options de compilation utilisés. C'est un point essentiel car rappelons que pour obtenir l'accélération d'un algorithme (parallèle ou d'un portage GPU), il est nécessaire d'avoir une implémentation de référence (séquentielle ou parallèle). Nous avons fait notre possible au cours de cette thèse, pour obtenir des codes de références les plus performants possibles, afin de réaliser des comparaisons équitables.

Différentes méthodes existent pour analyser un code, la section suivante s'intéresse aux programmes tiers permettant de réaliser une telle analyse qui ont été utilisés au cours de ce travail de recherche.

2.2 Les analyseurs de code

Les analyseurs de code forment une catégorie de programmes permettant aux développeurs d'obtenir des informations précieuses sur les performances d'un code. Avant même d'améliorer les performances d'une application, savoir si des zones d'amélioration existent est une étape clé pour le développeur. L'utilisation d'analyseurs de code permet de conforter le chercheur dans ses intuitions d'optimisation voir même de révéler des comportements inattendus. Dans ce travail de recherche plusieurs analyseurs de code sont utilisés, en voici la liste :

- `gprof` est un outil open source GNU.
- *Modular Assembly Quality Analyzer and Optimizer* (MAQAO)[19] est décrit comme un analyseur de performances libre d'accès. Une particularité de MAQAO est de pouvoir analyser le binaire d'un code pour rendre une analyse fidèle de ce qui est exécuté. C'est un critère très intéressant dans le cas où l'accès à l'architecture ciblée est limitée en dehors de l'exécution même de l'approche (réservation de tâche, coût des tests...).
- *Tuning and Analysis Utilities* (TAU)[20] fournit une suite d'outils statiques et dynamiques permettant une analyse sur des applications parallèles en Fortran, C++, C, Java et Python.
- Nvprof[21] qui est développé par NVIDIA dans l'objectif principal d'analyser des codes utilisant des GPU.

L'utilisation d'analyseurs de code nous a surtout permis de définir les zones d'intérêts concentrant la majorité des calculs du code étudié, permettant ainsi d'estimer les gains maximaux possibles par un portage GPU. Ainsi, il est possible d'éviter de concentrer nos efforts sur une partie qui n'apporterait qu'une accélération mineure.

À ce stade, nous avons défini le contexte et les notions nécessaires pour réaliser le travail préliminaire d'analyse d'un code scientifique séquentiel en vu d'une optimisation ou, dans notre cas, d'un portage GPU. Nous allons dans la section suivante aborder un modèle courant de programmation parallèle qui permet d'utiliser la puissance de plusieurs CPU multi-cœurs simultanément. Puis nous introduirons les notions de CUDA nécessaires à la compréhension des portages GPU des chapitres 4 et 5.

2.3 Modèles classiques de programmation parallèle

La programmation parallèle cherche à tirer profit des architectures multi-cœurs parallèles décrites dans le chapitre 1. Nous avons abordé au cours de la taxonomie de Flynn l'architecture MIMD se décomposant en deux modèles de mémoire : partagée ou distribuée.

Dans le premier cas de modèle mémoire, la mémoire est partagée par les cœurs de calcul. Chaque cœur d'un tel processeur à donc une complète visibilité sur cette mémoire partagée. Deux architectures à mémoire partagée sont couramment définies :

- *Uniform Memory Access* (UMA) : tous les cœurs de calcul partagent physiquement la même mémoire de manière uniforme.
- *Non-Uniform Memory Access* (NUMA) : la mémoire est physiquement distribuée entre les cœurs de calcul et le temps d'accès dépend de la position relative du cœur à la mémoire.

Dans le second cas de modèle mémoire, la mémoire est distribuée entre les nœuds de calcul. Chaque processeur possède sa propre mémoire locale et n'a pas directement accès à la mémoire des autres processeurs. Pour partager des données, cela se fait par le biais de messages entre les processeurs.

Les modèles de programmation utilisés pour exploiter ces architectures dans cette thèse sont :

- *Open Multi-Processing* (OpenMP)[22] pour le calcul au sein d'un processeur multi-cœurs à mémoire partagée. OpenMP est une interface de programmation ou en anglais *application programming interface* (API). OpenMP est utilisable dans plusieurs langages : C, C++ et Fortran. Le modèle d'exécution est de type *Fork-Join*, sous-modèle de la catégorie maître-esclaves. Dans ce modèle, un *thread Maître* fait appel à une équipe de *threads esclaves*, lorsqu'une région est spécifiée parallèle dans le code.
- *Message Passing Interface* (MPI)[23] permet l'échange d'information entre processeurs multi-cœurs. MPI est un protocole de communication pour programmer les ordinateurs parallèles, utilisable dans plusieurs langages : C, C++ et Fortran. Il existe plusieurs implémentations libres d'accès permettant à la fois les communications de type point à point (*Point-to-point*) et les communications collectives.

Le principe global de ce modèle de programmation combinant OpenMP et MPI est illustré par la figure 2.3. MPI gère les processus se situant au niveau des nœuds de calcul tandis que OpenMP est utilisé au sein de chaque processus. En somme, OpenMP permet d'utiliser tous les cœurs des processeurs (multi-cœurs), tandis que MPI permet d'utiliser simultanément plusieurs machines constituées de processeurs (multi-cœurs).

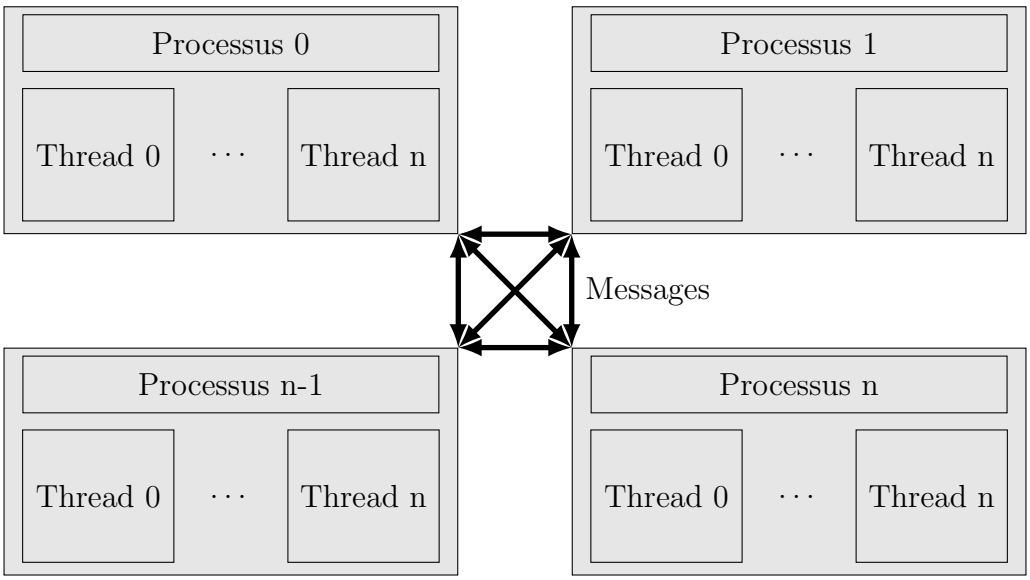


FIGURE 2.3 – Schéma de programmation parallèle hybride classique.

L'utilisation conjointe de OpenMP et MPI est courante dans le domaine du calcul haute performance. Plus spécifiquement ce modèle est utilisé couramment en chimie théorique pour tirer profit des architectures CPU parallèles.

2.4 Notions pour l'utilisation des GPU : CUDA

Compute Unified Device Architecture (CUDA)[1] est un ensemble de bibliothèques fournies et développées par NVIDIA[24] permettant d'utiliser leurs accélérateurs graphiques dans un contexte général, appelé, en anglais *General-Purpose Computing on Graphics Processing Units* (GPGPU). L'idée du GPGPU est d'utiliser les cartes graphiques sur certaines parties hautement parallélisables d'un code pour les accélérer. Nous avons fait le choix dans cette thèse d'utiliser CUDA pour tenter d'obtenir les meilleurs performances possibles sur les cartes graphiques NVIDIA à disposition. Avec pour but d'évaluer au mieux les portages GPU des méthodes de la chimie théorique qui ont été réalisés. Ces portages sont décrits dans les chapitres 4 et 5. L'ensemble des notions nécessaires à la compréhension de ces portages GPU sont décrites dans cette section 2.4.

CUDA est utilisable avec plusieurs langages : C, C++, Fortran, Python et Java par exemple. Un ensemble de bibliothèques scientifiques déjà accélérées pour le GPU sont fournies par NVIDIA :

- cuBLAS : fonctions de la bibliothèque BLAS.
- cuRAND : génération de nombres aléatoires.
- cuSOLVER : solveurs directs, dense et creux.
- cuSPARSE : fonctions d'algèbre linéaire basiques sur les matrices creuses.
- cuFFT : transformés de Fourrier.
- nvGRAPH : bibliothèque d'analyse de graphe.
- cuDNN : bibliothèque de réseau de neurones.

Comme décrit précédemment dans la section 1.2, une carte graphique est une architecture *manycore*, qui doit être utilisée par (au moins) un CPU. Le CPU est alors appelé *hôte* et la carte graphique est nommée *device*. Un code utilisant un ou plusieurs accélérateurs graphiques peut, par exemple, s'illustrer par la figure 2.4. C'est-à-dire que le CPU alterne

entre des instructions exécutées par lui-même, puis un ou plusieurs appels à des fonctions qui sont accélérées sur GPU. Une telle fonction est appelée noyau ou *kernel* en anglais.

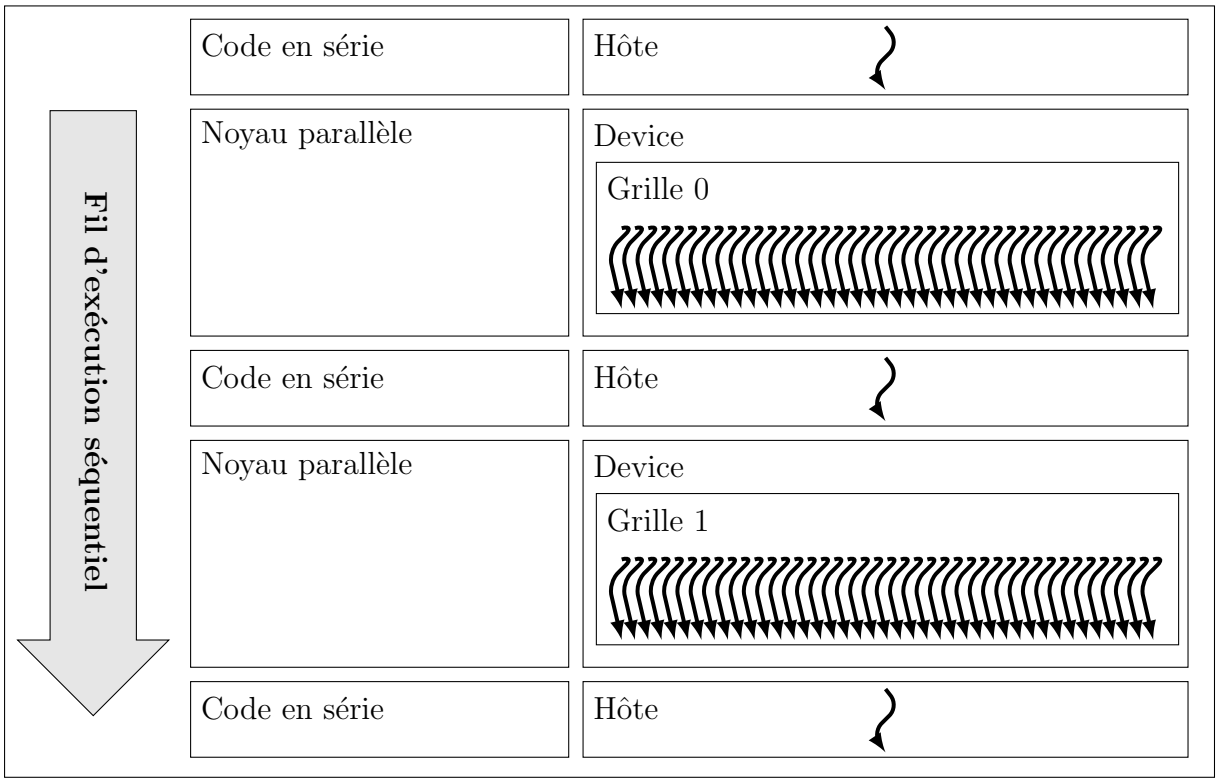


FIGURE 2.4 – Représentation d'une exécution d'un code tirant profit des capacités d'une carte graphique

L'utilisation de CUDA requiert un ensemble de notions qui sont définies dans la sous-section suivante, car nécessaires à la compréhension du travail de portage GPU avenir.

Les éléments constitutifs de l'architecture CUDA

CUDA permet de définir des fonctions en C, appelées noyaux (*kernel*), qui lorsqu'elles sont appelées, sont exécutées en parallèle par autant de processus légers (en anglais *threads*) CUDA. Un noyau se définit par l'utilisation de __global__ dans sa déclaration.

```
__global__ void Nom_du_kernel(variables)
```

L'appelle d'un noyau est caractérisé par l'utilisation de la syntaxe <<<...>>>, un appel de noyau générique ressemble à ceci :

```
Nom_du_kernel<<<Nb_blocs,Nb_threads[,mémoire,stream]>>>(variables);
```

Le noyau s'exécute à l'aide d'une grille de ressources, qui peut être jusqu'à tridimensionnelle. Cette grille est constituée de blocs qui peuvent eux aussi être tridimensionnels. Les blocs sont quant à eux constitués des processus légers qui exécutent le code du noyau. Le programmeur a accès à l'identité des processus légers par le biais des variables *blockIdx* et *threadIdx*, donnant respectivement l'identifiant du bloc contenant le processus léger et l'identifiant du processus léger au sein du bloc. Une grille et un bloc peuvent être jusqu'à

tridimensionnels, les données identifiant un processus léger le sont aussi (tridimensionnelles). Ainsi, les coordonnées d'un processus léger au sein d'un bloc sont disponibles grâce aux trois variables, *threadIdx.x*, *threadIdx.y* et *threadIdx.z*. Comportement similaire pour la variable *blockIdx* des coordonnées du bloc au sein de la grille.

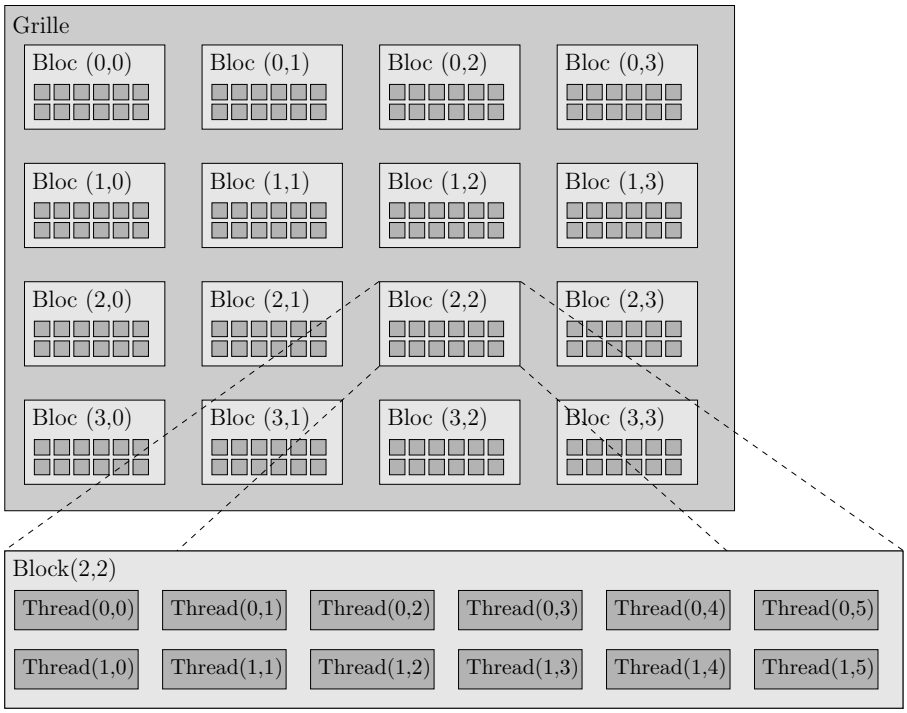


FIGURE 2.5 – Représentation d'un exemple de constitution de la grille d'un noyau en blocs, et des blocs en processus légers ; grille (4,4) et blocs (6,2).

Chaque bloc de processus légers est alors résolu en parallèle de manière asynchrone sur les différents *Streaming Multiprocessor* (SM) que contient la carte graphique. Un bloc résout sa partie d'un noyau en faisant exécuter les tâches à réaliser par *warp* de processus légers. En CUDA, un *warp* est un ensemble de 32 processus légers[25] (depuis les cartes avec une *Compute Capability 3.0*) qui exécutent leurs instructions simultanément.

Plusieurs limitations sont à prendre en compte. En fonction de la carte graphique utilisée le nombre maximum de processus légers que peut contenir un bloc est différent car les ressources mémoires doivent être partagées pour tous les processus légers du même bloc. Sur les cartes graphiques utilisées dans cette thèse la limite est de 1024 processus légers par bloc[25]. La carte graphique utilisée définit aussi les limites des dimensions pour la grille et les blocs.

Il est important de garder à l'esprit que les blocs doivent pouvoir s'exécuter indépendamment les uns des autres et de comprendre que l'exécution se fait par *warp* de 32 processus légers simultanément avec concurrence entre les différents blocs. De même, la taille d'un *warp* étant fixe et définie par la carte graphique, n'utiliser qu'en partie un *warp* est sous-optimal, car certains processus légers se retrouvent inactifs.

La majeure partie des notions primordiales de CUDA (noyau, bloc, processus léger et *warp*) sont maintenant détaillées au lecteur. L'autre aspect des GPU qui peut permettre d'obtenir une performance optimale passe par l'utilisation des différentes mémoires disponibles sur une carte graphique. Ces mémoires sont détaillées dans la section suivante.

Les différentes mémoires d'un GPU

Notons tout d'abord que la mémoire de l'hôte et du *device* sont à différencier, les transferts sont, à la base, à la charge du programmeur. Depuis CUDA 6.0, NVIDIA a introduit la notion de mémoire unifiée, avec pour objectif de simplifier l'utilisation des GPU en automatisant la gestion de la mémoire entre le CPU et le GPU. Pour faire cela, la mémoire unifiée élimine la nécessité d'expliciter les mouvements de la mémoire avec l'utilisation de fonctions (comme *cudaMemcpy*). Les déplacements de la mémoire entre CPU et GPU sont toujours présents, implicitement. Nous nous intéresserons dans cette partie à la mémoire disponible sur le *device*. Le principe de mémoire unifiée n'a pas été utilisé dans le cadre de cette thèse.

Sur une carte graphique, plusieurs types de mémoires sont présentes avec des quantités et des temps d'accès différents. L'obtention de performances sur GPU est en partie conditionnée par la "bonne" utilisation des différents types de mémoires disponibles. De plus, la connaissance des propriétés des mémoires justifie certains choix réalisés dans les chapitres 4 et 5 sur les portages GPU implémentés.

Avec CUDA, chaque processus léger a sa propre mémoire registre. Chaque bloc donne accès à de la mémoire partagée. Les trois mémoires constante, globale et texture sont persistantes entre les noyaux. Le schéma 2.6 illustre la localité des mémoires dans le GPU utilisé dans ce travail de thèse.

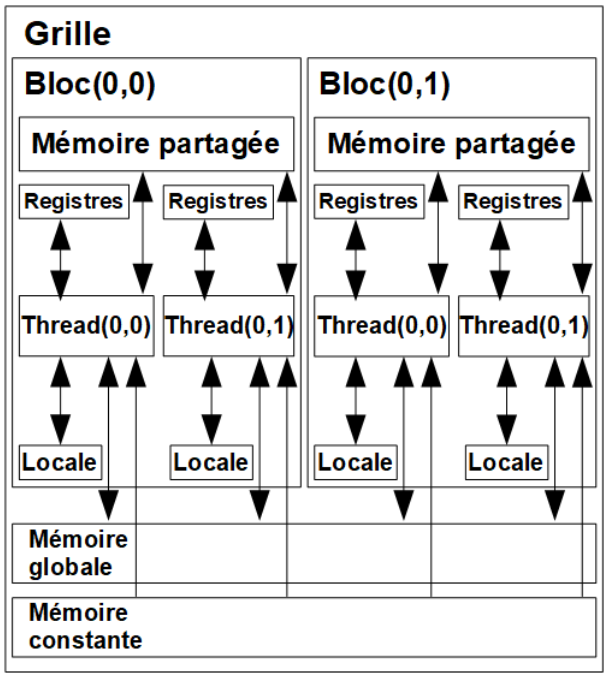


FIGURE 2.6 – Représentation simplifiée des mémoires utilisables en CUDA.

Les mémoires ont donc chacune une localité mais aussi une quantité et une vitesse d'accès qui leurs sont propres. La suite aborde plus en détail les différents types de mémoires :

- globale : accessible par tous les processus légers, en grande quantité cependant la vitesse d'accès est lente.
- constante : en lecture uniquement pour un noyau. Si plusieurs processus légers lisent des données de différents endroits, les accès sont sérialisés. Seul l'hôte peut

écrire en mémoire constante. La quantité de mémoire constante est de 64Ko[25] au maximum.
- partagée : accessible par tous les processus légers d'un même bloc, en quantité limitée, avec un temps d'accès plus efficace que la mémoire globale.
- registre : attribuée par processus léger et est visible uniquement à celui-ci. La quantité de mémoire registre est très limitée mais sa vitesse d'accès est rapide.
- locale : correspond à une abstraction. En effet, il n'existe pas un composant physique dédié au stockage de la mémoire locale, cette dernière est allouée dans une région de la mémoire globale dans le cas, par exemple, où trop de registres sont demandés par un processus léger.
- texture : absente de la figure simplifiée 2.6, car cette dernière n'est pas utilisée au cours de cette thèse. La mémoire texture est accessible par tous les processus légers du *device* en lecture et possède la particularité de pouvoir bénéficier d'opérations d'interpolations spécifiques au niveau matériel.

La mémoire partagée est composée de mots de 4 octets. Chaque mot pouvant être un réel simple précision, un entier de 32 bits, deux entiers courts, la moitié d'un réel double précision... La mémoire partagée est gérée sur les cartes NVIDIA par 32 banques par SM. Chaque mot de mémoire appartient à une banque de tel sorte que le mot numéro 0 appartient à la banque 0, le mot numéro 1 appartient à la banque 1 ainsi de suite jusqu'au mot numéro 31 qui appartient à la banque 31. La numérotation recommence alors, le mot 32 appartient à la banque 0, le mot 33 appartient à la banque 1... La banque 0 possède tous les mots multiples de 32, la banque 1 possède tous les mots dont le modulo par 32 est 1... La figure 2.7 illustre ce propos.

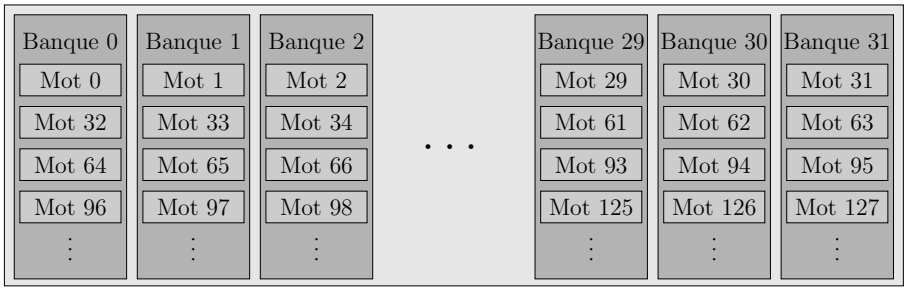


FIGURE 2.7 – Stockage des mots de la mémoire partagée dans les banques.

L'accès à la mémoire partagée peut être perturbé dans certains cas, c'est pour cela que l'annexe D aborde la notion de conflit de banque, problème rencontré au cours de ce travail de recherche.

Toutes les notions informatiques nécessaires à la compréhension des différents portages des chapitres 4 et 5 ont été définies. La section suivante décrit le centre de calcul ROMEO dans lequel ce travail de thèse a été réalisé.

2.5 Centre de calcul ROMEO

Le centre de calcul régional ROMEO est soutenu par la région Champagne-Ardenne depuis 2002. C'est avec les ressources qu'il met à disposition que nous avons réalisé l'entièreté de ces travaux de recherche. Ces ressources sont à la fois matérielles et logicielles.

La pile logicielle mise à disposition des utilisateurs est en constante évolution, en fonction de la demande des différents utilisateurs, allant de logiciels de calcul, aux compilateurs en passant par différents analyseurs de code.

Le cluster ROMEO entre à la 151ème place[26] du TOP500[4] et à la 5ème place[27] du Green500[28] lors de son installation en 2013. Il est composé de 130 nœuds chacun composé de deux GPU (NVIDIA Tesla K20X) et de deux processeurs Intel Ivy Bridge 8 cœurs à 2,6GHz avec 32Go de mémoire. Le centre de calcul ROMEO dispose aussi de l'architecture de pointe vendue par NVIDIA : le DGX-1.

La nouvelle machine du centre ROMEO vient d'être mise à disposition des scientifiques le 1er octobre 2018. Elle est entrée au TOP500 de juin 2018 à la 249éme place. Cette nouvelle machine met à disposition deux types de nœuds de calcul :

- 45 nœuds de calcul CPU avec deux processeurs Intel Skylake (6 132) possédant chacun 14 cœurs de calcul d'une fréquence de 2,6GHz. Chaque nœud possède 192 Go de mémoire.
- 70 nœuds de calcul hybrides avec deux processeurs Intel Skylake (6 132) possédant chacun 14 cœurs de calcul d'une fréquence de 2,6GHz. Chaque nœud possède 96 Go de mémoire. Et surtout, chaque nœud contient quatre cartes graphiques NVIDIA P100.

Ayant travaillé sur la machine du centre de calcul ROMEO disponible depuis 2013, j'ai utilisé les cartes graphiques K20X de NVIDIA. Nous allons donc décrire les cartes graphiques K20X dans la prochaine section.

2.6 Accélérateur NVIDIA K20X

Les cartes graphiques K20X appartiennent à la génération Kepler de NVIDIA se basant sur l'architecture GK110. NVIDIA annonce[29] un pic théorique avec des nombres à virgule flottante double précision à 1,31 Tflops pour les cartes graphiques K20X, et un pic théorique à 3,95 Tflops en simple précision.

La figure 2.8 représente de manière simplifiée la composition d'une carte graphique K20X. Cette carte est composée d'un GPU GK110[30]. Un GPU GK110 est composé de 15 SMX, la carte graphique K20X est construite pour utiliser 14 SMX. L'architecture des GPU GK110 est détaillée dans la suite et représentée par la figure 2.9.

Tous les transferts entre le CPU et le GPU passent par le port PCIe. La distribution du travail est orchestrée par le *Giga Thread Engine* entre les SM. Six contrôleurs de mémoire 64 bits s'assurent des accès aux 6Go de GDDR5 constituants la mémoire totale du *device*. La mémoire totale est utilisée pour le stockage des mémoires constantes, globale, locale (en cas de manque de registres) et texture.

L'antémémoire (ou mémoire cache) L2 de 1,5Mo se situe entre la mémoire totale et chaque SMX. Tous les accès à la mémoire totale passent par le cache L2, incluant aussi les transfères de données en provenance de l'hôte par PCIe.

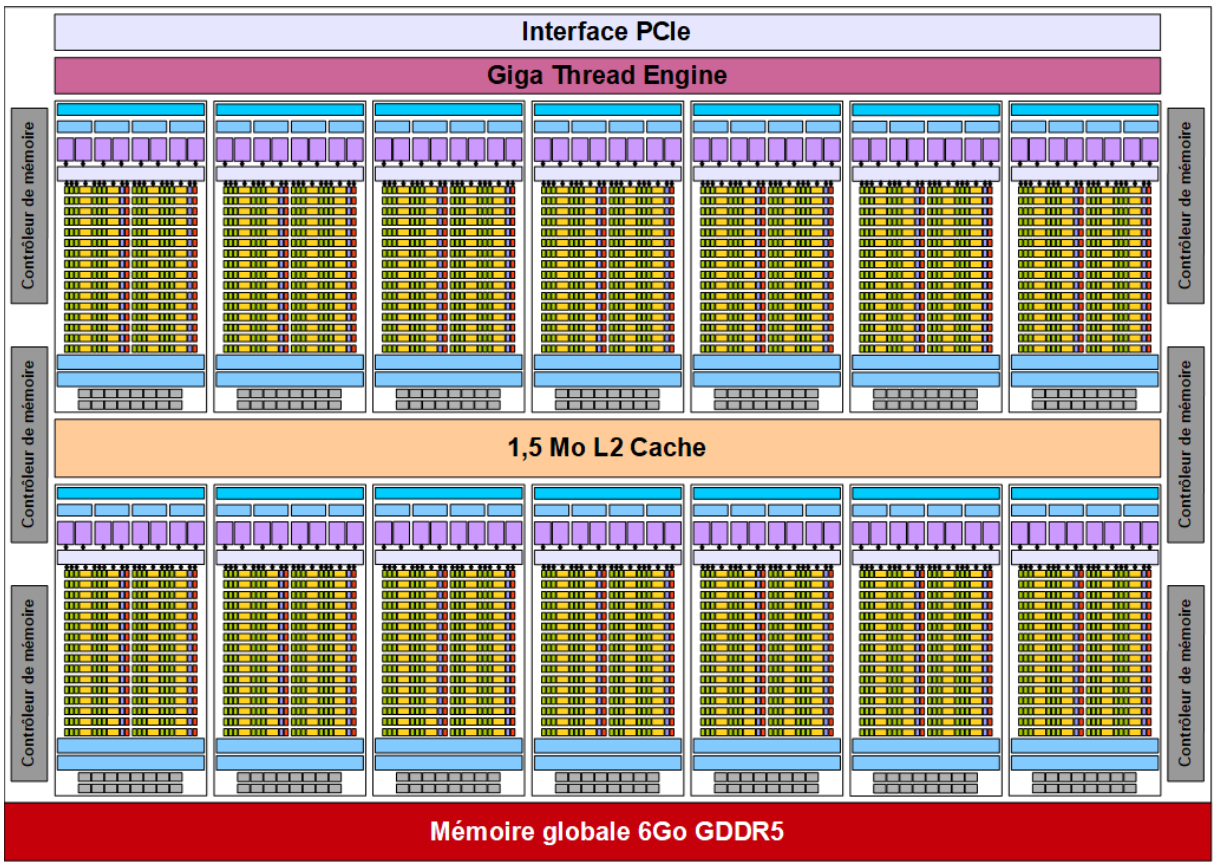


FIGURE 2.8 – Accélérateur NVIDIA K20X.

Chaque SMX contient 192 cœurs de calcul CUDA dédiés aux opérations à virgule flottante simple précision ou arithmétiques entières simples, 64 cœurs dédiés aux opérations à virgule flottante double précision, et 32 unités pour les fonctions spéciales comme des instructions transcendantes (exponentielles, trigonométriques...). Pour servir les unités précédentes, 32 unités dédiées au chargements et au stockages mémoire. La figure 2.9 illustre la constitution d'une SMX.

À cela, s'ajoute 65 536 registres 32 bits par SMX avec accès à 64Ko de mémoire sur puce, configurable par l'utilisation entre mémoire partagée et cache L1, pour la carte graphique K20X trois configurations existent : 16Ko-48Ko, 32Ko-32Ko et 48Ko-16Ko. Une limite maximale de 255 registres par processus léger est aussi imposée par la carte graphique. Chaque SMX possède aussi, sur puce, de 48Ko de cache en lecture seule. Ainsi que 16 unités de filtrage de texture par SMX. Les différentes mémoires sont décrites de manière générale dans la section 2.4.

Quand un noyau est exécuté sur une carte graphique K20X, le *Giga Thread Engine* gère l'envoie des blocs de processus légers aux SMX disponibles. Chaque SMX est capable de gérer jusqu'à 2048 processus légers ou 64 blocs de processus légers, incluant des blocs de noyaux différents, si les ressources le permettent. Tous les processus légers d'un bloc particulier doivent résider sur un seul SMX. Une fois qu'un bloc de processus légers est assigné à un SMX, tous les processus légers contenues dans ce bloc sont exécutés entièrement sur ce SMX.

Au niveau d'un SMX, chaque bloc de processus légers est réduit en pièces de 32 processus légers consécutifs, que sont les *warps*. Une carte graphique K20X émet ses

instructions au niveau des *warps*. Les instructions sont émises par vecteur de 32 processus légers consécutifs simultanément. Ce modèle d'exécution correspond dans la taxonomie de Flynn, section 1.3, au sigle SIMT.

Chaque SMX a quatre ordonnanceurs de *warp*. Quand un bloc est divisé en plusieurs *warps*, chaque *warp* est assigné à un ordonnanceur de *warp*. Les *warps* restent toutes leur existence sur l'ordonnanceur qui leur est assigné. L'ordonnanceur est capable de basculer entre des *warps* concurrents, originaires de n'importe quel bloc de n'importe quel noyau, sans coût. Quand un *warp* stagne, c'est-à-dire que l'instruction ne peut pas être exécutée au cycle suivant, l'ordonnanceur bascule vers une *warp* qui peut exécuter une instruction. Cette capacité de changer de *warp* permet de cacher efficacement la latence des instructions, si suffisamment de *warps*, avec des instructions pouvant être émises, sont sur le SMX.

Chaque ordonnanceur de *warp* a deux unités d'envoi d'instruction. À chaque cycle l'ordonnanceur choisit une *warp*, et si possible, deux instructions indépendantes de ce *warp* seront traitées. Deux cycles sont requis pour traiter des instructions double précision d'un *warp* complet.

La carte graphique K20X fournit un support complet de la norme IEEE 754-2008 pour les opérations arithmétiques à virgule flottante simple et double précision. L'annexe B aborde le principe de stockage des nombres à virgule flottante, le fait important pour nous est que ce stockage n'est pas infiniment précis, faisant que certaines opérations sur les nombres à virgule flottante perdent leur caractère associatif et/ou commutatif.

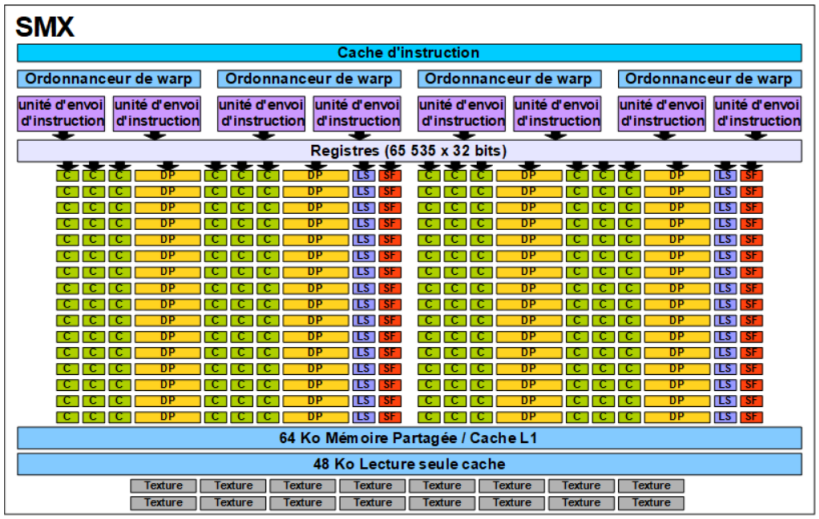


FIGURE 2.9 – Illustration de la composition d'un SMX utilisé dans les cartes graphiques K20X

C'est en utilisant une ou deux cartes graphiques K20X que la majorité de cette thèse a été réalisée. En effet, les temps d'exécutions des portages GPU réalisés dans les chapitre 4 et 5 sont obtenus par l'utilisation des cartes graphiques K20X disponibles au centre de calcul ROMEO.

2.7 Conclusion

Comme nous l'avons vu au cours du premier chapitre : les architectures informatiques évoluent. Les contraintes physiques et énergétiques expliquent l'évolution des architectures

informatiques vers une augmentation du nombre de cœurs pour répondre à une demande croissante de puissance de calcul. Cette évolution vers des solutions dites *manycore* implique la nécessité pour certains domaines de devoir se réinventer et d'innover pour faire correspondre leurs approches aux architectures contemporaines. Ce travail de thèse tente d'apporter une contribution dans ce sens, en portant des algorithmes de pointe de la chimie théorique sur GPU. Notre hypothèse est que l'utilisation d'architectures *manycore* telles que les cartes graphiques peuvent permettre d'accélérer grandement des approches de la chimie théorique vis-à-vis des approches simplement multi-cœurs. L'obtention d'une telle accélération pourrait permettre d'aborder l'étude de systèmes chimiques jusqu'à présent peu, voir pas abordables, à cause des temps de calcul trop importants ou d'une précision des résultats peu fiable.

L'évolution des architectures HPC implique une évolution algorithmique et des modèles de calcul comme nous avons pu le voir au cours de ce chapitre 2. Dans la suite, pour valider notre hypothèse, plusieurs algorithmes de chimie théorique sont étudiés puis portés sur GPU afin d'en évaluer la performance. Les chapitres 4 et 5 traitent des portages GPU réalisés au cours de ce travail de recherche. L'entièreté des notions nécessaires pour comprendre ces chapitres sont maintenant décrites grâce à ce chapitre 2.

Le chapitre 3 suivant nous permet d'aborder des approches majeures de la chimie théorique et justifie le choix des algorithmes accélérés par GPU dans les chapitres qui suivront.

Chapitre 3

Algorithmes abordés en chimie théorique

Ce travail de thèse s'inscrit à l'interface entre la chimie théorique et l'informatique. Le contexte scientifique est celui du logiciel AlgoGen[31] développé à l'URCA par la collaboration entre les laboratoires ICMR et CReSTIC. C'est un outil de *docking* moléculaire dont l'utilité est de prédire la manière dont une molécule appelée ligand, constituée de quelques atomes (une centaine au maximum) peut venir "s'attacher" à son hôte, en général une protéine constituée de plusieurs centaines, voire milliers d'atomes. C'est un outil de modélisation moléculaire couramment employé dans le domaine du *drug design* (conception de médicaments) visant à trouver de nouvelles molécules actives dans le domaine médical. La technique de *docking* nécessite de pouvoir évaluer l'énergie d'interaction (l'affinité) entre le ligand et la protéine. Ainsi, AlgoGen, au moment d'évaluer ce score fait appel à une méthode externe. En chimie théorique on distingue généralement deux manières d'obtenir cette énergie d'interaction. La première est la mécanique moléculaire basée sur la notion de champ de force permettant d'accéder rapidement à un "score" d'interaction grâce à un jeu de paramètres empiriques. La seconde est la chimie quantique, beaucoup plus précise, basée sur le concept de fonction d'onde ; en contrepartie les calculs dans cette dernière sont beaucoup plus coûteux. C'est ce dernier outil (chimie quantique) qui a été choisi par les auteurs du code AlgoGen pour améliorer les résultats des simulations de *docking* moléculaire. Ces simulations de *docking* "quantique" pouvant être coûteuses en temps de calcul, plusieurs centaines d'évaluations quantiques par simulation, il est donc intéressant de voir comment réduire les temps de calcul par l'utilisation des nouvelles architectures. Nous pensons que l'utilisation de cartes graphiques est une voie possible à emprunter pour accélérer des approches de la chimie théorique, en particulier dans le cadre du logiciel AlgoGen. C'est cette hypothèse que nous avons éprouvée au cours de ce travail de recherche. Le code AlgoGen n'est pas abordé dans cette thèse mais plutôt deux méthodes d'évaluation de score qui sont envisagées à moyen terme pour être appelées par le logiciel AlgoGen. Ainsi, AlgoGen s'appuiera à terme sur les deux programmes externes : GAMESS ou NCIplot. Mon travail de thèse a donc consisté à réaliser le portage GPU du programme NCIplot (code relativement petit) et aussi d'une petite routine du logiciel GAMESS (code beaucoup plus important proposant de nombreuses méthodes de chimie quantique). Ces deux programmes externes à AlgoGen s'appuyant sur la notion de mécanique quantique notamment à travers les notions de la fonction d'onde et de densité électronique, il est important de faire quelques rappels dans ce domaine.

Nous décrivons dans ce chapitre 3 de manière générale les algorithmes qui sont au cœur du sujet de recherche. Nous introduirons très sommairement les méthodologies quantiques

DFTB, FMO et PCM pour la partie sur le logiciel GAMESS, et la méthodologie NCI pour la partie sur le logiciel NCIplot. Nous aborderons aussi dans cette partie la méthodologie IGM (*Independent Gradient Model*) qui nous est apparue pendant la recherche dont le portage sur GPU constitue une suite logique à mon travail.

La section 3.1 s'intéresse aux différentes méthodes quantiques telles que les méthodes ab initio et semi-empiriques. Cette section 3.1 décrit succinctement l'approche Hartree-Fock (HF), base des méthodes quantiques. La section 3.2 aborde la méthode DFT (Density Functional Theory) qui est le fondement nécessaire pour la méthode quantique DFTB (Density-Funtional Tight-Binding) qui nous intéresse ensuite pour le chapitre 5. La méthode DFTB est abordée dans la section 3.3. Les méthodologies FMO (Fragment Molecular Orbital) et PCM (Polarizable Continuum Model) sont aussi évoquées respectivement dans les sections 3.4 et 3.5. La méthodologie FMO tend à accélérer les calculs vers une extensibilité linéaire en fonction de la taille du système étudié. La méthodologie PCM permet elle de simuler l'effet d'un solvant sur le système étudié. La section 3.6 traite brièvement de la méthode NCI qui est l'objet du travail réalisé dans le chapitre 4. Avant de conclure, la section 3.7 fait allusion au modèle IGM (Independent Gradient Model) qui est apparu au cours de ce travail de thèse.

3.1 Tour d'horizon des méthodes quantiques

Les deux programmes abordés dans ce travail de thèse s'appuient sur la description quantique des électrons des molécules étudiées par le chimiste. Le premier, NCIplot (méthode NCI), s'appuie sur le concept de densité électronique ρ qui est reliée dans le modèle quantique à la fonction d'onde électronique Ψ du système comme nous allons le voir. Le second GAMESS (méthode PCM) cherche à déterminer l'effet d'un solvant sur cette fonction Ψ.

L'objectif de cette section 3.1 est de poser le contexte des méthodes quantiques majeures, avant de définir le problème qui m'a intéressé lors du portage GPU réalisé sur une partie de GAMESS (méthode PCM). Le modèle quantique considère une molécule comme un ensemble de noyaux et d'électrons. La résolution de l'équation de Schrödinger, $\hat{H}\Psi = E\Psi$ (où $\hat{H}$ représente l'opérateur hamiltonien du système, Ψ la fonction d'onde décrivant ce système et E son énergie qui est la quantité recherchée), base de la chimie quantique, étant impossible pour des systèmes multiélectroniques, des approximations sont faites afin de pouvoir estimer l'énergie E d'une molécule. La première est l'approximation de Born-Oppenheimer qui vise à séparer les électrons des noyaux dans l'expression de la fonction d'onde totale Ψ. La position des noyaux est alors considérée fixe pendant le calcul de l'énergie E, les variables de la fonction Ψ portant uniquement sur la position des électrons. On se ramène à une équation similaire dite équation de Schrödinger "électronique" :

$$\hat{H}_{elec}\Psi_{elec} = E_{elec}\Psi_{elec} \tag{3.1}$$

où l'opérateur hamiltonien électronique $\hat{H}_{elec}$:

$$\hat{H}_{elec} = \hat{H}^{e}_{cin} + \hat{H}^{e-N}_{attrac} + \hat{H}^{e-e}_{repul} \tag{3.2}$$

Avec :

- Ψ_{elec} est la fonction d'onde électronique.
- $\hat{H}^{e}_{cin}$ est l'opérateur énergie cinétique des électrons.
- $\hat{H}^{e-N}_{attrac}$ est l'opérateur énergie d'attraction électron-noyau.
- $\hat{H}_{repul}$ est l'opérateur énergie de répulsion électron-électron.

L'énergie totale U dans l'approximation de Born-Oppenheimer est composée de deux termes :

$$U = E_{elec} + V_{NN} \tag{3.3}$$

- E_{elec} l'énergie électronique évoquée plus tôt.
- V_{NN} l'énergie de répulsion noyau-noyau.

La résolution analytique de cette équation de Schrödinger (3.1) est généralement impossible. La résolution numérique peut être très coûteuse en temps de calcul. Plusieurs niveaux d'approximations sont apparus donnant lieu à différentes méthodes de chimie quantique. Nous introduirons sommairement les méthodes ab initio et semi-empiriques. La section 3.2 aborde quant à elle, une autre catégorie de méthodes quantiques qui nous intéresse pour la suite : la théorie de la fonctionnelle de la densité (DFT). Les méthodes ab initio et DFT ont en commun l'expression de la densité électronique $\rho =< \Psi_{elec}|\Psi_{elec} >$ (intégrale sur tout l'espace du carré de la fonction d'onde électronique). Cette quantité ρ est au cœur de la méthode NCI, objet d'une partie de ce travail de thèse.

Dans les méthodes de chimie quantique, un système électronique est décrit par un ou plusieurs déterminants de Slater. Le déterminant de Slater Ψ_{elec}^{DS} est construit comme suit pour un système à n électrons :

$$\Psi_{elec}^{DS} = \frac{1}{\sqrt{n!}} \begin{vmatrix} \chi_1(1) & \chi_2(1) & \dots & \chi_n(1) \\ \chi_1(2) & \chi_2(2) & \dots & \chi_n(2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(n) & \chi_2(n) & \dots & \chi_n(n) \end{vmatrix} \tag{3.4}$$

où $\chi_i(j)$ représente la spinorbitale i décrivant l'électron j dans la molécule. Une spinorbitale moléculaire χ_i est le produit d'une fonction d'espace ψ_i (dont les trois variables sont les trois coordonnées de l'électron décrit) et d'une fonction de spin (à une seule variable, ne pouvant prendre que deux valeurs). Une fois Ψ_{elec}^{DS} obtenue (après un calcul quantique par le biais le GAMESS par exemple), on peut obtenir les propriétés associées à une observable du système, par exemple l'énergie moyenne :

$$E_{elec} = \frac{< \Psi_{elec}^{DS}|\hat{H}|\Psi_{elec}^{DS} >}{< \Psi_{elec}^{DS}|\Psi_{elec}^{DS} >} \tag{3.5}$$

La notation de Dirac (bra-ket) est utilisée pour exprimer les intégrales correspondantes ici. À noter que dans la pratique, ce déterminant de Slater met en jeu des produits d'orbitales moléculaires (OM) monoélectroniques : $\chi_1(1)\chi_2(2)...\chi_n(n)$ qui décrivent finalement des électrons indépendants les uns des autres, approximation connue sous le nom "d'approximation orbitale". Dans la pratique chacune des OM χ_i est construite comme une combinaison linéaire d'orbitales atomiques (voir ci-après).

Les méthodes *ab initio*

Les méthodes *ab initio* s'affairent à trouver une solution approchée Ψ_{elec}^{DS} de l'équation de Schrödinger à partir de la connaissance des positions des noyaux des atomes, du nombre d'électrons et du spin du système. Les méthodes ab initio reposent sur les premiers principes de la mécanique quantique, donc l'utilisation de données empiriques est normalement proscrite. Les méthodes ab initio peuvent être classifiées en plusieurs catégories.

L'approche Hartree-Fock (HF)

Tout d'abord l'approche Hartree-Fock (HF) aussi appelée méthode du champ auto-cohérent ou encore *self-consistent field method* (SCF). Il en existe plusieurs variantes dont les méthodes *Restricted Hartree-Fock* (RHF) et *Unrestricted Hartree-Fock* (UHF), selon que le système étudié ne possède que des électrons appariés (2 à 2) ou possède un ou plusieurs électrons célibataires.

La méthode HF utilise le principe variationnel pour approximer Ψ_{elec}^{DS} décrite par un seul déterminant de Slater : la meilleure fonction doit être celle qui correspond à un minimum d'énergie. Nous n'entrerons pas dans les détails de cette méthode et nous allons aller à l'essentiel car ce n'est pas le cœur de mon sujet de recherche, cependant l'approche SCF est une méthode fondatrice qui permet d'introduire plusieurs notions importantes pour la suite, comme les intégrales à calculer.

Nous allons nous intéresser au formalisme de la méthode RHF. Dans un système à couches fermées constitué de $2n$ électrons, la fonction d'onde est décrite par un déterminant de Slater construit sur la base de $2n$ spinorbitales moléculaires χ_i, produit d'une fonction d'espace ψ_i et d'une fonction de spin (décrivant le spin de l'électron). Dans la pratique, chaque OM ψ_i est décrite dans l'équation (3.6) comme une combinaison linéaire de N orbitales atomiques (CLOA) où les coefficients (que l'on doit obtenir) $c_{\mu i}$ sont supposés réels :

$$\psi_i = \sum_{\mu=1}^{N} c_{\mu i}\phi_\mu \tag{3.6}$$

Les orbitales atomiques (OA) ϕ_μ sont les solutions connues monoélectroniques d'un atome isolé. Le jeu des N orbitales atomiques utilisées est appelée "base d'orbitale atomique".

L'approche RHF est un processus itératif qui passe par la résolution du système de N équations linéaires et homogènes suivant :

$$\sum_{\eta} c_{\eta i}[F_{\eta\mu} - e_i S_{\eta\mu}] = 0 \tag{3.7}$$

i pouvant prendre toutes les valeurs de 1 à N, nous sommes bien en présence de N équations de ce type. Les coefficients $F_{\eta\mu}$ constituent la matrice dite de Fock :

$$\begin{aligned} F_{\eta\mu} =< \phi_\eta(1)| -\frac{1}{2}\nabla^2(1)|\phi_\mu(1) > - \sum_{k}^{noyaux} Z_k < \phi_\eta(1)|\frac{1}{r_k}|\phi_\mu(1) > \\ + \sum_{\lambda\sigma} P_{\lambda\sigma}[(\eta\mu|\lambda\sigma) - \frac{1}{2}(\eta\lambda|\mu\sigma)] \end{aligned} \tag{3.8}$$

chaque terme à calculer $F_{\eta\mu}$ comprend deux types d'intégrales. Tout d'abord, l'intégrale monoélectronique $< \phi_\eta(1)|\hat{g}(1)|\phi_\mu(1) >$, où $\hat{g}$ est un opérateur monoélectronique qui prend comme argument la fonction atomique monoélectronique de base ϕ_μ :

$$< \phi_\eta(1)|\hat{g}(1)|\phi_\mu(1) >= \int \phi_\eta(1)\hat{g}\phi_\mu(1)d\tau_1 \tag{3.9}$$

Ici, $\hat{g}(1) = -\frac{1}{2}\nabla^2(1) - \sum_{k}^{noyaux} \frac{Z_k}{r_{1k}}$. Les autres intégrales sont biélectroniques $(\eta\mu|\lambda\sigma)$ et $(\eta\lambda|\mu\sigma)$, cette notation signifiant :

$$(\eta\mu|\lambda\sigma) = \int\int \phi_\eta(1)\phi_\mu(1)\frac{1}{r_{12}}\phi_\lambda(2)\phi_\sigma(2)d\tau_1 d\tau_2 \tag{3.10}$$

$$(\eta\lambda|\mu\sigma) = \int\int \phi_\eta(1)\phi_\lambda(1)\frac{1}{r_{12}}\phi_\mu(2)\phi_\sigma(2)d\tau_1 d\tau_2 \tag{3.11}$$

Pour chaque élément $F_{\eta\mu}$ de la matrice de Fock, ces deux intégrales biélectroniques sont à calculer pour toutes les paires possibles d'orbitales atomiques $\lambda\sigma$. Cette étape constitue donc un calcul très lourd dans l'approche HF.

L'équation (3.8) donnant $F_{\eta\mu}$ fait intervenir les coefficients $P_{\lambda\sigma}$ qui constituent la matrice densité électronique. Ces coefficients s'écrivent :

$$P_{\lambda\sigma} = 2 \sum_{i}^{n\ occupees} c_{\lambda i}c_{\sigma i} \tag{3.12}$$

Il est à noter que la boucle de l'équation (3.12) parcourt les n orbitales occupées. Nous pouvons aussi noter que le facteur 2 apparaît car nous nous intéressons au formalisme RHF (chaque orbitale moléculaire est occupée par 2 électrons). Enfin les éléments $S_{\eta\mu}$ prennent la forme suivante :

$$S_{\eta\mu} =< \phi_\eta|\phi_\mu > \tag{3.13}$$

et constituent la matrice de recouvrement S.

Les équations (3.7) sont aussi appelées équations de Roothaan. Ce système de N équations (3.7) n'admet de solutions non toutes nulles que dans le cas où les e_i sont choisis parmi les racines de l'équation séculaire suivante :

$$\begin{vmatrix} F_{11} - eS_{11} & F_{12} - eS_{12} & \dots & F_{1N} - eS_{1N} \\ F_{21} - eS_{21} & F_{22} - eS_{22} & \dots & F_{2N} - eS_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ F_{N1} - eS_{N1} & F_{N2} - eS_{N2} & \dots & F_{NN} - eS_{NN} \end{vmatrix} = 0 \tag{3.14}$$

Ce déterminant de dimension N × N conduit donc à une équation de degré N en e, donc à N valeurs e_i, conduisant chacune aux N coefficients $c_{\mu i}$ recherchés, donnant ainsi l'expression de chaque orbitale ψ_i (équation (3.6)). Notons le paradoxe des méthodes HF, la résolution de ce déterminant passe par le calcul préalable de ses éléments, dont les éléments de la matrice de Fock $F_{\eta\mu}$. Or, leur calcul nécessite de connaître les coefficients $c_{\lambda i}c_{\sigma i}$ présents dans la matrice densité $P_{\lambda\sigma}$(3.12), et qui constitue la solution recherchée. C'est donc par une procédure itérative que l'approche HF converge vers la solution.

Nous pouvons écrire le problème sous la forme matricielle suivante :

$$FC = ESC \tag{3.15}$$

avec :

- F la matrice de Fock.
- C la matrice des coefficients $c_{\mu i}$ rangés en colonne.
- E La matrice diagonale avec les valeurs propres e_i correspondantes (associées aux orbitales moléculaires ψ_i).
- S la matrice de recouvrement.

La première étape est d'orthogonaliser la base des orbitales atomiques passant de ϕ à ϕ' (φ' est une base d'orbitales atomiques orthogonalisée). Dans ce cadre, les matrices F et C deviennent F' et C', S' est la matrice identité, et on a alors à résoudre l'équation suivante :

$$F'C' = EC' \tag{3.16}$$

C'est alors un problème aux valeurs propres classique dont la résolution peut se faire par diagonalisation de la matrice de Fock F' dans cette nouvelle base.

Le déroulement (algorithme global) d'un calcul RHF dans le cas d'une base orthogonale d'orbitales atomiques est définie figure 3.1. Tout d'abord, on choisit notre base d'orbitales atomiques et une géométrie moléculaire de notre système étudié. A partir de là, les intégrales monoélectroniques et biélectroniques peuvent être calculées et stockées en mémoire. Elles ne dépendent que de la géométrie et donc n'ont pas à être recalculées ensuite dans le processus itératif SCF. Ensuite, nous créons une première matrice densité $P^{(0)}$ (par l'équation 3.12) pour pouvoir débuter le processus itératif. Pour cela un premier jeu initial de coefficients $c_{\mu i}$ est utilisé (appelé *guess*). Nous pouvons alors créer une première matrice de Fock avec les intégrales et la matrice densité $P^{(0)}$. En diagonalisant cette matrice de Fock, nous obtenons les valeurs propres e_i. Ces valeurs propres e_i permettent de déterminer de nouveaux coefficients $c_{\mu i}$ pour une nouvelle matrice densité $P^{(1)}$. Dans le cas où cette matrice densité $P^{(1)}$ est suffisamment distincte de $P^{(0)}$, le processus recommence avec une création d'une nouvelle matrice de Fock, cette fois à partir des intégrales (déjà calculées) et de la matrice densité $P^{(1)}$. Ce processus se répète jusqu'à convergence selon des critères que nous n'aborderons pas, car ce n'est pas l'objet de ce travail de recherche.

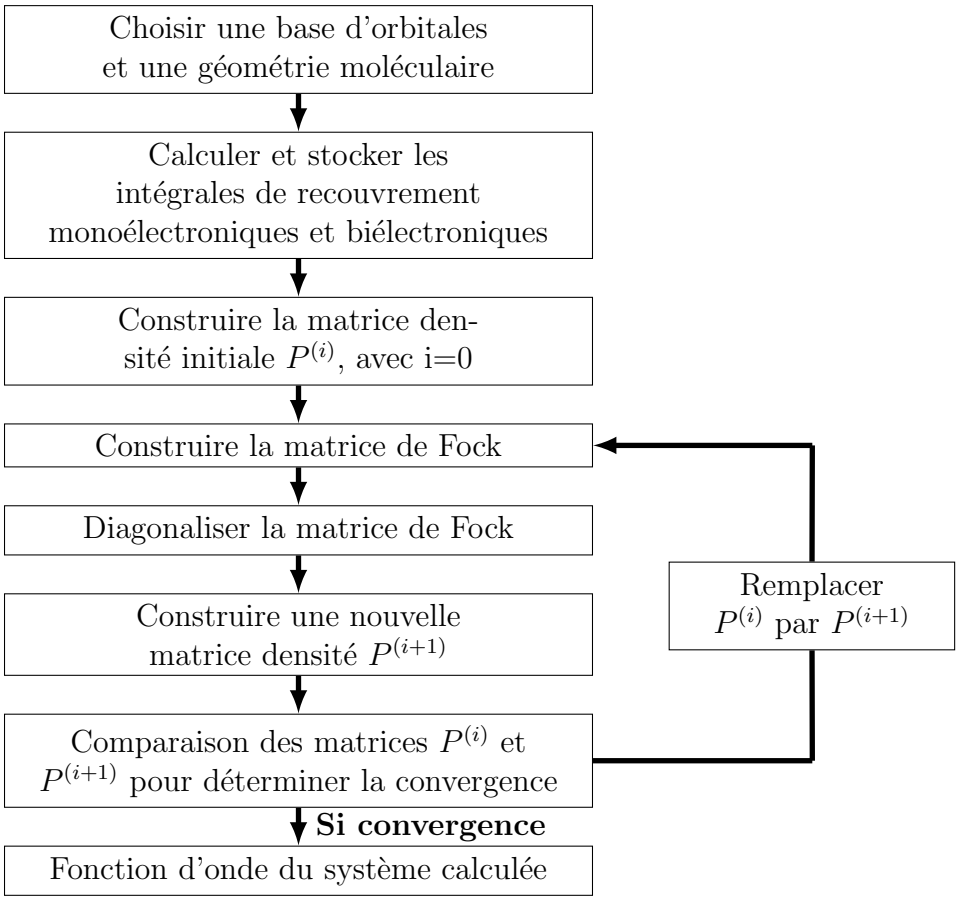


FIGURE 3.1 – Algorithme global de la méthode Hartree-Fock à géométrie fixe.

L'énergie obtenue tend vers une limite (en fonction de la taille de la base) appelée «limite Hartree-Fock» chaque électron étant uniquement affecté par le champ "moyen" créé par les autres électrons du système. Il n'y a aucune influence directe du mouvement

d'un électron sur un autre. Pour décrire l'énergie "exacte" du système, il manque la «corrélation électronique» utilisée pour décrire l'interaction entre les électrons du système.

La complexité algorithmique de la méthode HF est en N^4, le goulot d'étranglement et le calcul des intégrales biélectroniques (équations 3.10 et 3.11) dont le nombre augmente comme N^4 (N, la taille de la base d'orbitales atomiques). Cela rend le calcul couteux en terme de temps de calcul dès que le système étudié augmente en taille.

Les méthodes post-Hartree-Fock

Les méthodes post-Hartree-Fock tentent d'améliorer la méthode Hartree-Fock en prenant en compte une partie de la corrélation électronique, pour mieux décrire la répulsion entre les électrons, permettant ainsi de passer outre la limite HF.

Ces méthodes décrivent la fonction d'onde totale par une combinaison plus ou moins grande de déterminants de Slater. Elles sont importantes dans le cas où l'effet de la corrélation électronique est important. Plusieurs méthodes sont dans cette classification comme la méthode de champ multi-configurationnel auto-cohérent (*Multi-Configurational Self-Consistent Field* : MCSCF), la méthode Møller–Plesset de second ordre (MP2) ou encore d'interaction de configuration multi-référence (*Multi-Reference Configuration Interaction* : MRCI). Dans ce cadre les méthodes dépassent N^4 en complexité algorithmique, rendant les calculs limités à des systèmes de quelques atomes.

Les méthodes semi-empiriques

Les méthodes quantiques semi-empiriques réduisent le temps de calcul en diminuant le nombre d'intégrales à calculer. Plusieurs approximations sont communes à ces méthodes, dont :

- seuls les électrons de la couche de valence sont explicitement décrits dans les calculs.
- certaines intégrales (biélectroniques) sont supposées nulles.
- une partie des intégrales est estimée grâce aux informations obtenues par les connaissances expérimentales. À ce titre, ces méthodes ne sont pas des méthodes *ab initio*. L'utilisation de paramètres expérimentaux permet cependant, d'inclure une partie des effets de corrélation électronique, absents de l'approche HF.

Ainsi la complexité algorithmique diminue passant à N^3 au lieu de N^4 dans la méthode HF. Le goulot d'étranglement devient la diagonalisation de la matrice de Fock. Il existe plusieurs types de méthodes semi-empiriques, comme par exemple :

- Les méthodes du type NDDO (*Neglect of Diatomic Differential Overlap*)
- Les méthodes du type CNDO (*Complete Neglect of Differential Overlap*)
- Les méthodes du type INDO (*Intermediate Neglect of Differential Overlap*)

Dans le logiciel de *docking* moléculaire AlgoGen, l'énergie des molécules est actuellement obtenue par la méthode semi-empirique *parameterization method 7* (PM7). Les temps de calculs restent cependant importants pour un système de la taille d'une protéine. Plus récemment sont apparues différentes approches permettant de linéariser la complexité du temps de calcul en fonction de la taille de la molécule, technique communément appelé *linear scaling* dont nous reparlerons dans la section 3.4, et qui est utilisée par le logiciel de *docking* AlgoGen.

3.2 La théorie de la fonctionnelle de la densité

Bien que ce travail de thèse ne porte pas directement sur les méthodes théoriques, la seconde partie de ma thèse ayant portée sur l'accélération de calculs quantiques DFTB / FMO, la théorie de la fonctionnelle de la densité (DFT) au cœur de ces calculs est introduite très brièvement ici. En anglais, *Density Functional Theory* (DFT), prend la fonction de la densité électronique ρ du système comme variable.

La densité électronique $\rho(r)$ est une fonction de $\mathbb{R}^3_+$ correspondant à la densité de probabilité de présence d'un électron en r. Dans la théorie de la fonctionnelle de la densité, la fonction d'onde Ψ^{DS}_{elec} est représentée par un seul déterminant de Slater construit à partir de N orbitales moléculaires ψ_k, fonction chacune des trois coordonnées spatiales d'un électron. Nous avons alors la densité électronique sous la forme suivante pour un système à couches fermées :

$$\rho(r) = 2\sum_{k=1}^{N} |\psi_k(r)|^2 \tag{3.17}$$

L'intérêt de l'utilisation de la densité électronique est donc de passer d'un espace à 3N variables (dans l'approche HF) à un espace défini par les 3 variables (x, y, z) de l'espace (dans la DFT). L'avantage de la densité électronique est aussi son caractère observable expérimentalement.

Dans le cadre de la DFT (indépendante des méthodes ab initio), le premier théorème de Hohenberg-Kohn démontre que toutes les propriétés d'un système dans un état fondamental non dégénéré sont complètement déterminées par sa densité électronique ρ. L'énergie est alors entièrement définie par une fonctionnelle sur la densité électronique, soit :

$$E = E[\rho] = F[\rho] + \int V_{ext}(r)\rho(r)dr \tag{3.18}$$

Où :

- $V_{ext}(r)$ représente le potentiel extérieur ressenti par les électrons (potentiel généré par les noyaux atomiques).
- $F[\rho]$ est une fonctionnelle indépendante de ce potentiel externe, représentant l'énergie cinétique d'électrons indépendants et les interactions entre électrons.

Le second théorème de Hohenberg-Kohn indique qu'il suffit de minimiser la fonctionnelle de la densité pour déterminer l'état fondamental du système étudié, soit :

$$E[\rho] \geqslant E[\rho_0] = E_0 \tag{3.19}$$

Ces deux théorèmes de Hohenberg-Kohn forment la base de la résolution de l'équation de Schrödinger par l'utilisation de la densité électronique. Kohn-Sham proposent une méthode de résolution itérative, figure 3.2, revenant à considérer des électrons indépendants évoluant dans le potentiel de Sham V_S avec la même densité que le système réel :

$$V_S = V_{ext} + V_H[\rho] + V_{XC}[\rho] \tag{3.20}$$

Où :

- V_{ext} est le potentiel externe des noyaux.
- V_H est le potentiel d'Hartree (potentiel Coulombien classique).
- V_{XC} est le potentiel d'échange-corrélation, ayant une forme analytique inconnue en général. en plus du terme d'échange (présent aussi dans l'approche HF), il prend notamment en compte le terme de corrélation électronique qui fait tant défaut à la méthode HF, mais aussi la différence entre l'énergie cinétique du système réel et celle d'un ensemble d'électrons indépendants.

Les orbitales moléculaires monoélectroniques ψ_i du système étudié obéissent alors aux équations de Kohn-Sham :

$$[-\frac{1}{2}\nabla^2 + V_S]\psi_i = \epsilon_i\psi_i \tag{3.21}$$

Comme le potentiel V_S dépend de la densité électronique ρ cherchée (notamment pour décrire $V_{XC}[\rho]$) : ces équations sont résolues par une procédure auto-cohérente qui ressemble à celle employée dans l'approche HF, décrite dans la figure 3.1. Concrètement, une base connue d'orbitales atomiques ϕ est utilisée pour construire chaque orbitale moléculaire ψ_i : $\psi_i = \sum_{\mu=1}^{N} c_{\mu i}\phi_\mu$ (équation (3.6)). La différence est qu'à la place d'utiliser ensuite la matrice de Fock F, la matrice de Kohn-Sham est utilisée. Un élément $K_{\eta\mu}$ est défini comme ceci :

$$\begin{aligned} K_{\eta\mu} =& < \phi_\eta(1)| -\frac{1}{2}\nabla^2(1)|\phi_\mu(1) > - \sum_{k}^{noyaux} Z_k < \phi_\eta(1)|\frac{1}{r_{1k}}|\phi_\mu(1) > \\ &+ < \phi_\eta(1)| \int \frac{\rho(r')}{|r_1 - r'|}dr'|\phi_\mu(1) > + < \phi_\eta(1)|V_{XC}|\phi_\mu(1) > \end{aligned} \tag{3.22}$$

Si la densité électronique $\rho(r')$ qui apparaît dans le terme coulombien est construit en utilisant des orbitales moléculaires combinaisons linéaires d'orbitales atomiques : $\rho(r') = \sum_\lambda \sum_\sigma \sum_i 2c_{\mu i}c_{\eta i}\phi_\lambda(r')\phi_\sigma(r')$ (analogue à l'équation (3.17)), alors des intégrales biélectroniques apparaissent dans le calcul de $K_{\eta\mu}$, comme dans l'approche HF : (équation (3.10)) $(\eta\mu|\lambda\sigma)$. Dans la pratique, $\rho(r')$ est plutôt développée sur une base de fonctions auxiliaires : $\rho(r') \simeq \sum_s c_s w_s(r')$, ramenant la combinatoire des intégrales biélectroniques à évaluer à N^2 en DFT au lieu de N^4 en HF. L'approche DFT passe donc aussi par le calcul d'intégrales électroniques sur une géométrie donnée du système chimique. Nous pouvons voir que les deux premières intégrales sont les mêmes pour les équations 3.8 de Fock et 3.22 de Kohn-Sham. En revanche, les différentes façons d'estimer le potentiel d'échange-corrélation V_{XC} conduisent à autant de méthodes DFT. Ce n'est pas l'objet de cette thèse donc nous n'en parlerons pas plus dans notre cadre.

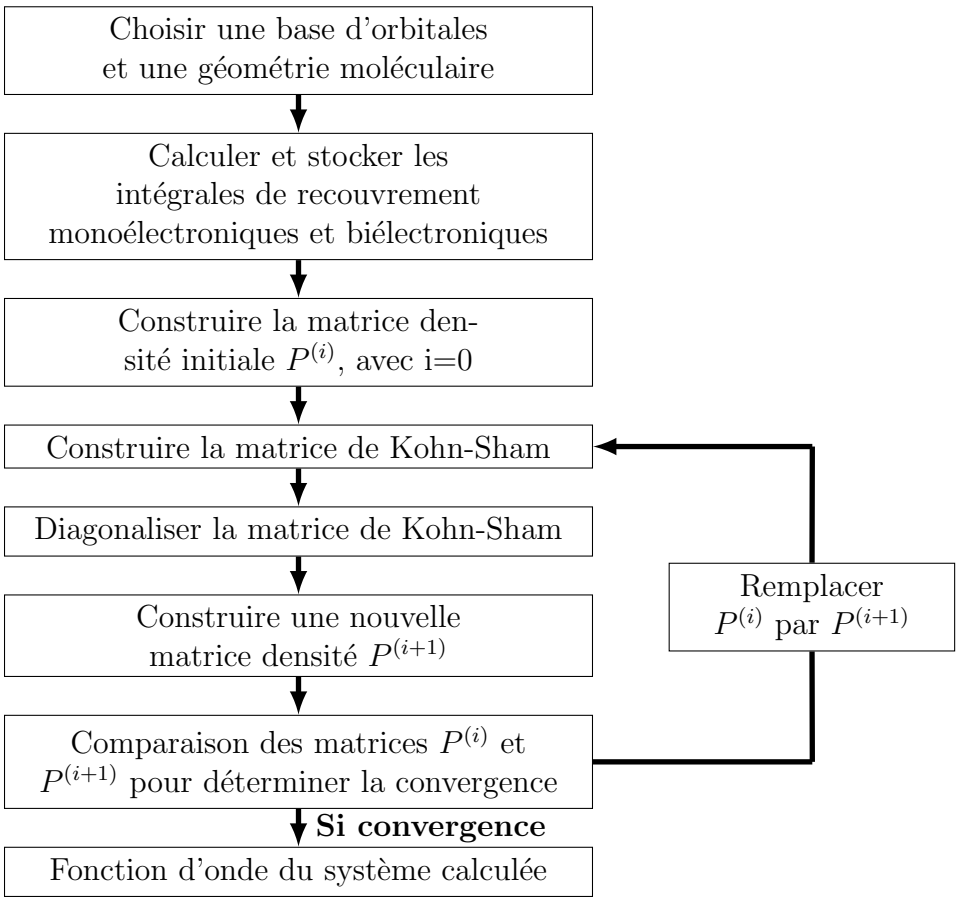


FIGURE 3.2 – Algorithme global de la méthode DFT à géométrie fixe.

L'approche DFT est fondatrice de l'approche DFTB qui nous intéresse dans le cadre de cette thèse. Nous abordons l'approche DFTB dans la section suivante.

En résumé, par rapport à l'approche HF, cette approche DFT permet de mieux décrire les effets de corrélation électronique tout en employant un algorithme analogue mais moins couteux car le goulot d'étranglement porte maintenant sur la diagonalisation de la matrice de KS (N^3).

3.3 Les méthodologies DFTB

Pour les mêmes raisons que pour les méthodes *ab initio*, le temps de calcul et les ressources mémoires nécessaires dans les calculs DFT augmentent rapidement avec la taille du système étudié (nombre d'électrons et taille de la base d'orbitales atomiques). Globalement, les implémentations des méthodes DFT ont une complexité algorithmique en N^3. Comme mentionné précédemment, il existe des stratégies pour réduire le temps de calcul.

L'approche *density-Functional Tight-Binding* (DFTB) est une approximation de la méthode DFT. Les approches DFT et DFTB ont toutes deux une complexité algorithmique en N^3. Tout en ayant une précision similaire, la méthode DFTB coûte cependant beaucoup moins cher en terme de temps de calcul. Cela est dû au fait que la méthode DFTB remplace, comme dans les méthodes semi-empiriques, le calcul de nombreuses intégrales par des quantités pré-calculées. Plus précisément, ces quantités proviennent de

calculs DFT (intégrales dites de Slater-Koster). De plus, seuls les électrons de valences des atomes sont considérés comme dans les méthodes semi-empiriques.

Les calculs quantiques que nous avons réalisés dans cette thèse emploient cette approche DFTB. Cependant, à nouveau, l'objectif à long terme du programme de *docking* moléculaire AlgoGen est de pouvoir appliquer ces calculs quantiques à des systèmes protéiniques pouvant compter des milliers d'atomes. Cette approche DFTB ayant une complexité algorithmique en N^3, malgré toutes ces stratégies de réduction de coût de calcul, demeure trop coûteuse pour aborder de tels systèmes. La solution à ce problème provient de méthodes dites *linear scaling* pour rendre linéaire le temps de calcul avec le nombre d'électrons du système étudié.

3.4 Vers l'extensibilité linéaire (Linear scaling)

Plusieurs stratégies existent pour rendre le temps d'un calcul quantique presque linéaire avec la taille du système. Ces stratégies de chimie théorique ont abouti aux méthodes MOZYME[32], DivCon[33] ou celle qui nous intéresse pour la suite : FMO (Fragment Molecular Orbital)[34]. AlgoGen emploie actuellement l'approche MOZYME[32] couplée à la méthode semi-empirique PM7. À terme, il est souhaitable de tester les capacités de la méthode FMO, d'où le choix d'avoir examiné de près dans ce travail de thèse la méthode DFTB / FMO proposée dans GAMESS.

Fragment Molecular Orbital (FMO) est un schéma général d'accélération développé en 1999[34] et implémenté dans GAMESS, il peut en théorie être appliqué à tout type de méthode quantique, dont la DFTB. L'intérêt de FMO est de permettre le calcul de systèmes moléculaires larges en divisant la molécule en fragments (monomères et dimères). La méthode quantique désirée (HF, MP2, DFT, DFTB...) est alors exécutée sur les monomères et les dimères. Cette méthode permet une accélération du temps de calcul au prix d'une approximation, l'énergie du système étant représentée par la formule :

$$E_{FMO} = \sum_{I} E_I + \sum_{I<J} (E_{IJ} - E_I - E_J) \tag{3.23}$$

Ainsi, le calcul le plus imposant à réaliser se résume maintenant à la taille d'un dimère, au lieu du système total. La figure 3.3 représente un système dans son ensemble tandis que la figure 3.4 représente la fragmentation du système étudié. En contrepartie, une multitude de ces calculs est à réaliser, proportionnellement au nombre de monomères. Ce schéma permet de rendre quasiment linéaire le temps de calcul quantique en fonction du nombre d'atomes[35]. Ce changement de complexité algorithmique provient du fait que l'approche FMO permet le passage d'une diagonalisation d'une matrice de dimension N × N (complexité N^3) à plusieurs diagonalisations de plus petites matrices d'une taille de dimère au maximum. Cette approche est de plus bien adaptée à l'utilisation de matériel parallèle en distribuant les calculs des monomères et dimères sur les ressources informatiques disponibles. Cette approche a rencontré un vif intérêt ces dernières années, car pouvant être appliquée en principe à toutes les méthodes de la chimie quantique.

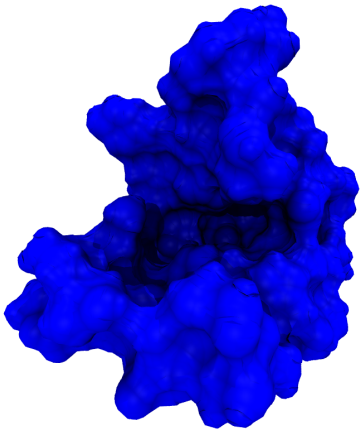


FIGURE 3.3 – Représentation du site d'une protéine.

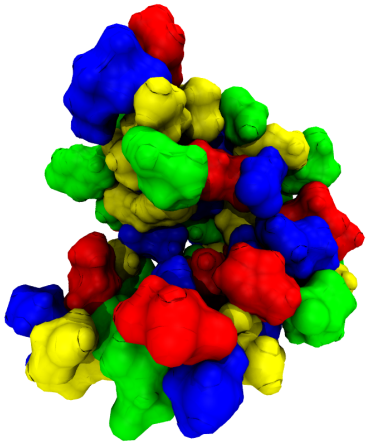


FIGURE 3.4 – Représentation d'une fragmentation des atomes du site d'une protéine.

En particulier, l'approche combinant ce schéma d'accélération à la méthode quantique DFTB est implémentée au sein du logiciel GAMESS. Ce schéma a déjà permis d'aborder des systèmes d'un million d'atomes[36]. Les possibilités pour fragmenter le système étudié sont multiples. Dans le cas des protéines, il est conseillé par les auteurs de placer deux acides-aminés successifs par fragment (monomère) dans GAMESS. C'est cette approche combinée DFTB-FMO, qui nous a intéressé dans ce travail de thèse.

Nos premiers essais ont montré l'importance spécifique dans cette approche DFTB-FMO de prendre en compte l'effet du solvant sous peine d'être confronté à des problèmes de divergence de la procédure itérative du calcul quantique. Fedorov montre en 2016[3] que lorsque l'effet du solvant est intégré, les calculs se stabilisent et convergent. D'un point de vue "effort calculatoire", cette remarque est importante car la prise en compte de l'effet de solvant est coûteuse en temps de calcul.

3.5 Simulation de l'effet d'un solvant : méthode PCM

L'influence du solvant sur les propriétés du système étudié peut être prise en compte de deux manières en chimie quantique. Une première approche est de placer explicitement des molécules autour du système étudié appelé le soluté. L'inconvénient majeur est le coût prohibitif de cet ajout. En effet, pour représenter correctement les effets de solvatation sur les propriétés électroniques, se limiter à la première sphère de solvatation est insuffisant, et ce sont des dizaines voir des centaines de molécules qu'il faut alors ajouter, ce nombre dépendant de la taille du soluté. D'autres problèmes apparaissent, comme la question du placement de ces molécules.

Une alternative à cette première approche qui représente explicitement la distribution de charge du solvant est de remplacer ce dernier par un champ électrique continu représentant une moyenne statistique de tous ses degrés de libertés à une température T. Ce champ dit de "réaction", s'explique par l'orientation concertée des molécules (implicites) du solvant en présence de la distribution de charges du soluté introduit dans ce solvant. L'effet de ce champ est d'accroître la polarité du soluté (soluté à l'origine même de ce champ de réaction). Le processus est donc itératif, le champ de réaction s'accroît sous l'influence du soluté, le champ influençant en retour le soluté. Ce phénomène de po-

larisation mutuelle finit par s'arrêter à cause du coût entropique d'orientation du solvant et surtout, les molécules de solvant interagissent de manière défavorable avec le champ de réaction. Ce modèle continu prend donc en compte explicitement la distribution de charge du soluté, mais de manière implicite le solvant par un champ électrique. La présence de ce champ "extérieur" au soluté implique qu'un nouveau terme doit être inclus dans l'hamiltonien du système. Il traduit l'interaction entre la distribution de charge du soluté et le potentiel électrostatique associé au champ de réaction du solvant. Concrètement dans le *Polarizable Continuum Model* (PCM), associé au calcul quantique de l'énergie du soluté, ce potentiel électrostatique est modélisé par un ensemble de N_T charges "apparentes" explicites discrètes q_i disposées aux centres des N_T triangles constituant la surface d'une cavité construite autour du soluté. Dans un calcul quantique, la densité électronique du soluté est, elle, représentée de manière homogène via le concept de fonction d'onde. La présence de ces charges apparentes (représentant le champ de réaction du solvant) ajoute alors un terme coulombien monoélectronique $W_{\eta\mu}$ à chaque élément de la matrice de Fock vue précédemment (équation (3.8)) :

$$F_{\eta\mu}^{PCM} = F_{\eta\mu} + W_{\eta\mu} \tag{3.24}$$

Ce terme coulombien monoélectronique prend la forme suivante :

$$W_{\eta\mu}(1) = -\sum_{t=1}^{N_T} q_t w_{\eta\mu}^t(1) \tag{3.25}$$

terme représentant l'influence de toutes les charges apparentes q_t sur l'élément $F_{\eta\mu}$ de la matrice de Fock, avec :

$$w_{\eta\mu}^t(1) = < \phi_\eta(1) | \frac{1}{|r_1 - R_t|} | \phi_\mu(1) > \tag{3.26}$$

et où q_t est la charge de surface apparente du triangle t et R_t sont les coordonnées de ce même triangle t. N_T est le nombre total de triangles formant la surface de la cavité de solvatation. On voit donc que pour incorporer l'effet de solvant à chaque élément $F_{\eta\mu}$ de la matrice de Fock, il faut connaître les N_T charges apparentes q_t (les intégrales monoélectroniques $w_{\eta\mu}^t$ étant du même type que les intégrales monoélectroniques de la méthode HF, elles sont "facilement" calculables).

Le calcul de ces charges apparentes suit le protocole suivant[3]. Le soluté (constitué d'électrons et de noyaux chargés positivement) exerce un potentiel électrostatique V à la surface de cavité (induisant les charges de surface apparente cherchées) :

$$V = V^e + V^N \tag{3.27}$$

où V^e et V^N sont respectivement les contributions électronique et nucléaire du soluté (V^e et V^N sont deux vecteurs de N_T composantes) :

$$V_t^e = -Tr(Pw^t) \tag{3.28}$$

où P est la matrice densité électronique du soluté (3.12) et N_{AT} est le nombre d'atomes du système. V_t^e représentant cette fois le potentiel électrique généré sur le triangle t par la densité électronique de tout le soluté. De même :

$$V_t^N = \sum_{\alpha=1}^{N_{AT}} \frac{Z_{AT}}{|R_\alpha - R_t|} \tag{3.29}$$

Pour obtenir le vecteur des charges de surface apparente q_t, l'équation suivante doit être résolue :

$$Cq = g \tag{3.30}$$

où C représente une matrice carrée de dimension N_T contenant l'information sur la géométrie de la cavité ainsi que sur le milieu diélectrique représentant de manière continue le solvant[37]. g représente le vecteur potentiel électrique (N_T composantes) associé au champ de réaction du solvant en chacun des N_T points de la surface. Ce potentiel g dépend de la densité électronique du soluté, donc de la fonction d'onde du système chimique. g prend la forme suivante :

$$g = -\frac{\varepsilon - 1}{\varepsilon} V \tag{3.31}$$

où ε est la constante diélectrique du solvant.

Pour résumer l'impact de l'effet du solvant sur la procédure SCF (HF ou DFT), la figure 3.5 montre l'enchaînement des calculs auxquels nous avons eu affaire dans GAMESS au niveau du portage GPU (chapitre 5).

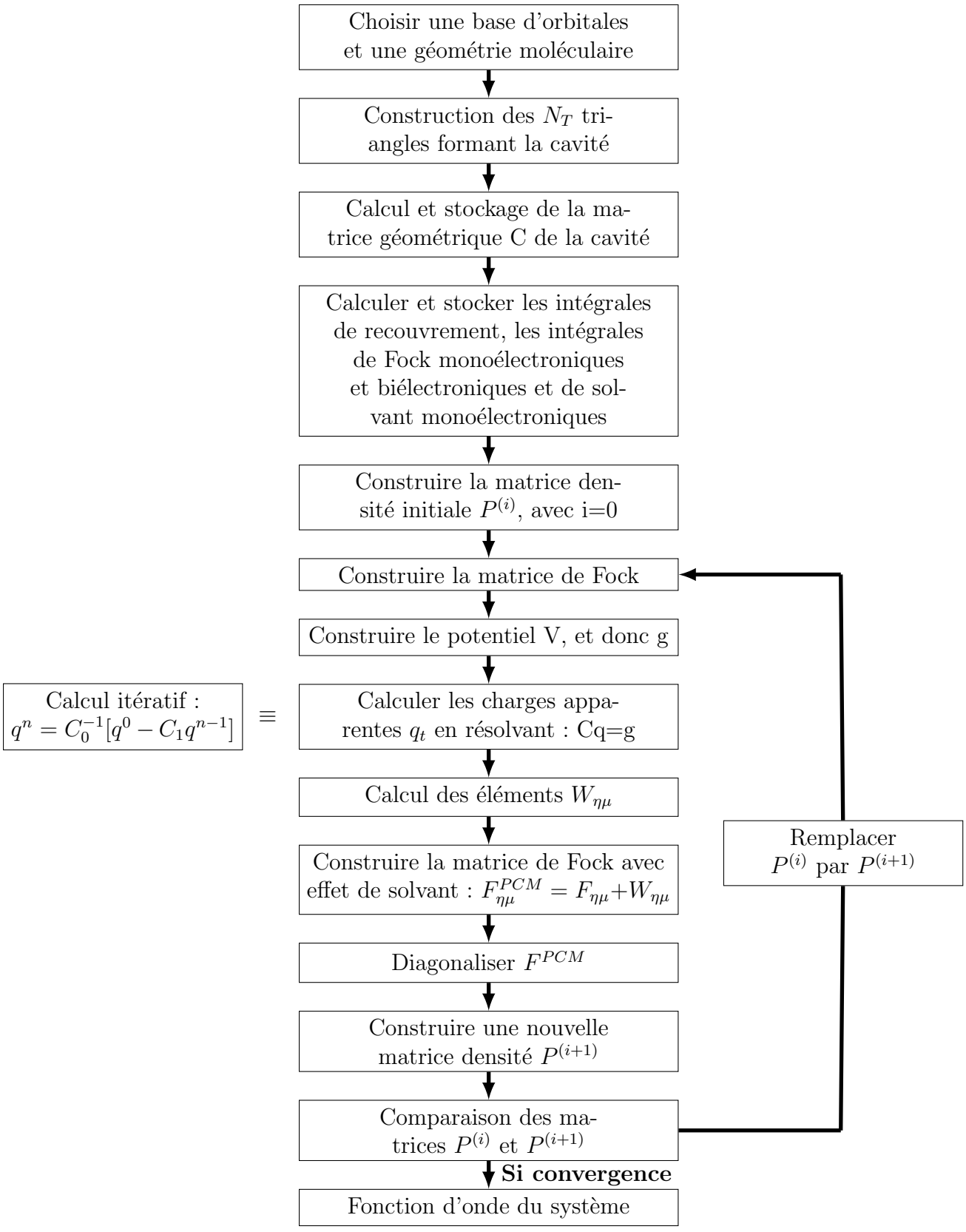


FIGURE 3.5 – Schéma SCF incorporant l'effet de solvant par la méthode PCM ; deux niveaux d'itérations imbriqués.

La densité électronique du soluté sert à dériver un premier jeu de charges apparentes

$q^{(0)}$ à la surface de la cavité, qui servent à obtenir par une procédure itérative, le jeu de charges apparentes finales $q^{(n)}$. Celles-ci servent alors à calculer les termes coulombiens $W_{\eta\mu}$ monoélectroniques rajoutés aux éléments de la matrice de Fock $F_{\eta\mu}$ simulant ainsi l'interaction entre le soluté et le solvant. La diagonalisation de cette matrice de Fock F^{PCM} génère alors une nouvelle fonction d'onde du soluté et donc une nouvelle densité incorporant l'influence du solvant. Cette densité est alors à nouveau utilisée pour décrire un nouveau jeu de charges apparentes. Ainsi de suite.

Le calcul du produit $C_1 q^{(n-1)}$ apparaît à chaque nouvelle itération de la procédure d'inversion de la matrice C. Ajoutons à cela le renouvellement du calcul à chaque itération quantique (procédure SCF).

La résolution requiert l'inversion de la matrice géométrique C qui peut être importante en fonction de la taille du soluté entouré de solvant. C'est pourquoi Pomelli et al.[37] ont proposé une procédure itérative plus rapide de résolution de cette équation, au centre de laquelle le vecteur des charges $q^{(n)}$ calculé à l'itération n est donné par :

$$q^{(n)} = C_0^{-1}(q^{(0)} - C_1 q^{(n-1)}) \tag{3.32}$$

où $q^{(0)}$ est le jeu de charges apparentes de départ dépendant de la densité électronique quantique du soluté, C_0 et C_1 contiennent respectivement les éléments diagonaux et hors-diagonaux de la matrice géométrique C. Mon travail est en partie porté sur le calcul du produit matrice-vecteur $C_1 q^{(n-1)}$, implémenté dans GAMESS.

3.6 La méthode NCI

Une fois la fonction d'onde Ψ du système étudié obtenue par l'une des méthodes quantiques précédentes (ab initio ou DFT, avec ou sans effet de solvant), il existe en chimie théorique des outils de "post-traitement" pour analyser la densité électronique ρ définie par le carré de cette fonction d'onde Ψ. C'est dans ce cadre que s'est placée ma première application de portage GPU consacrée à la méthode *Non-Covalent Interaction* (NCI) décrite brièvement dans cette section. Quelques mots seront dits aussi à propos de la toute nouvelle approche IGM, dérivée de NCI, qui a vu le jour lors de ma période de thèse.

Ces méthodes s'appuient sur l'approche topologique de la densité électronique ρ prenant son origine dans la théorie *Atoms In Molecules* (AIM) qui est un modèle pour les systèmes moléculaires. Le modèle AIM débute avec les travaux de Richard BADER[38], débutant la théorie dans les années 60 et s'appuyant uniquement sur l'analyse topologique de la densité électronique pour définir l'atome et la liaison chimique.

La topologie de la densité électronique s'intéresse à la distribution de la densité électronique dans l'espace autour des noyaux d'un système chimique. L'exemple le plus simple est un système constitué d'un atome d'hydrogène isolé. La distribution est telle que plus l'on s'éloigne de l'atome, plus la densité décroît comme illustré figure 3.6.

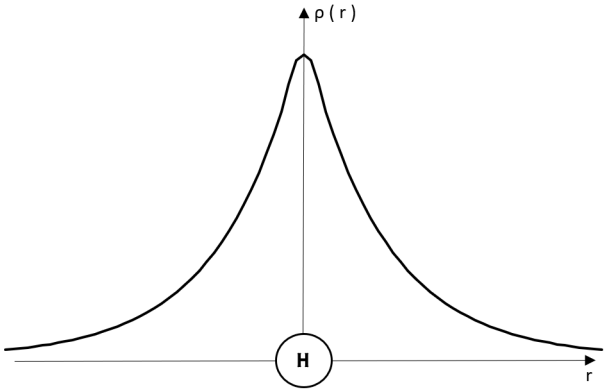


FIGURE 3.6 – Densité électronique d'un atome d'hydrogène isolé en fonction de la distance à cet atome, obtenue à l'aide de calculs quantiques.

Pour le cas de l'hydrogène isolé figure 3.6, la fonction de la densité électronique ρ atteint son maximum lorsque la position est centrée sur l'atome d'hydrogène. Pour simplifier la compréhension, la densité ρ est représentée sur cette figure selon la distance par rapport à l'atome. On peut facilement concevoir que la distance par rapport à un objet dans un espace tridimensionnel peut varier selon les trois axes. La densité électronique est une fonction des trois variables de l'espace pouvant donc produire une hypersurface, avec des points dits critiques. Ces points critiques peuvent être reliés aux notions chimiques d'atomes et de liaison. Mathématiquement, un point critique r_c, de coordonnées spatiales (x_c, y_c, z_c), est défini de telle sorte que toutes les composantes de son vecteur gradient s'annulent :

$$\nabla\rho(r_c) = \begin{pmatrix} \frac{\partial\rho(r_c)}{\partial x} \\ \frac{\partial\rho(r_c)}{\partial y} \\ \frac{\partial\rho(r_c)}{\partial z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.33}$$

faisant d'un tel point un extremum (maximum ou minimum).

Lorsqu'au moins deux atomes constituent le système étudié, plusieurs points critiques apparaissent sur cette hypersurface. C'est sur l'étude de ces points critiques que se concentrent les méthodes comme NCI ou IGM qui s'appuient sur l'approche topologique de la densité électronique. Dans l'exemple de la figure 3.7, le système est constitué de deux atomes d'hydrogènes faisant apparaître trois points critiques de la densité électronique. Les deux premiers (P1 et P2) correspondent aux positions des atomes d'hydrogène du système. Le troisième point critique (P3), se situe quant à lui exactement à mi-distance entre les deux atomes. Comme précédemment, cette notion de point critique peut être généralisée jusqu'aux trois dimensions de l'espace. Afin de déterminer la nature de celui-ci, le calcul des dérivées partielles du second ordre est nécessaire. Pour le cas illustré figure 3.7, les points P1 et P2 possèdent une courbure négative dans toutes les directions, permettant de conclure à des maxima locaux. Le point critique P3 possède à la fois une courbure positive le long de l'axe inter-nucléaire, et deux négatives perpendiculairement à cet axe.

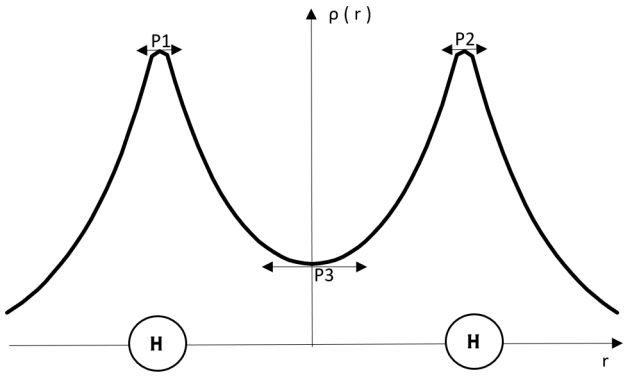


FIGURE 3.7 – Densité électronique quantique le long de l'axe intermoléculaire du système chimique H—H.

La caractérisation de la nature du point critique est généralisée par le nombre de valeurs propres négatives de la matrice Hessienne. Les dérivées partielles d'ordre deux permettent de définir cette matrice Hessienne H de la densité :

$$H(\rho(r)) = \begin{pmatrix} \frac{\partial^2\rho(r)}{\partial x^2} & \frac{\partial^2\rho(r)}{\partial x\partial y} & \frac{\partial^2\rho(r)}{\partial x\partial z} \\ \frac{\partial^2\rho(r)}{\partial y\partial x} & \frac{\partial^2\rho(r)}{\partial y^2} & \frac{\partial^2\rho(r)}{\partial y\partial z} \\ \frac{\partial^2\rho(r)}{\partial z\partial x} & \frac{\partial^2\rho(r)}{\partial z\partial y} & \frac{\partial^2\rho(r)}{\partial z^2} \end{pmatrix} \xrightarrow{diagonalisation} \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \tag{3.34}$$

Dans le cas que l'on étudie, cette matrice Hessienne est carrée de dimension (3,3), symétrique et réelle. La diagonalisation de cette matrice permet d'obtenir trois valeurs propres réelles $\lambda_1 \leq \lambda_2 \leq \lambda_3$. Lorsque les trois valeurs propres sont négatives, on a affaire à un maximum local (position nucléaire, comme P1 et P2 ici). Lorsque deux valeurs propres sont négatives et une valeur propre est positive, on a affaire à un point critique de liaison chimique (comme P3 dans notre exemple). Deux autres cas peuvent survenir en chimie, non dessinés ici. Tout d'abord le cas où une valeur propre est négative et deux valeurs propres sont positives. C'est le cas d'un point critique de cycle (*ring critical point*), comme au centre du benzène. Enfin un point critique possédant trois valeurs propres positives se trouve au centre d'une cage, comme dans le fullerène par exemple.

NCI est une méthodologie relativement récente (2010) se basant sur la densité électronique permettant de localiser et d'identifier la nature des interactions d'un système moléculaire. NCI va au delà du modèle AIM en définissant des régions de l'espace où se produisent des interactions chimiques. Le point de départ de notre implémentation GPU est le programme NCIplot de Julia CONTRERAS-GARCIA[39]. La méthode NCI se base sur le gradient réduit de la densité électronique, $s(r)$:

$$s(r) = \frac{1}{2(3\pi^2)^{\frac{1}{3}}} \frac{||\nabla\rho(r)||}{\rho(r)^{\frac{4}{3}}} \tag{3.35}$$

où $\nabla\rho(r)$ est le vecteur gradient de la densité électronique. Le comportement de cette fonction en fonction de la densité électronique ρ permet d'identifier la présence d'interactions dans un système chimique. Cet aspect prédictif est au cœur de l'intérêt de cette approche NCI, et sera décrit ci-après. Pour obtenir $s(r)$, la densité $\rho(r)$ peut être calculée de deux manières selon le niveau de précision souhaitée. Dans l'absolu, $\rho(r)$ est calculée à partir du carré de la fonction d'onde Ψ du système étudié. Pour de très gros systèmes comme une protéine, ce calcul quantique étant trop coûteux, il a été montré que l'on pouvait se

contenter de la densité "promoléculaire"[40][41] en se limitant à l'étude des interactions non-covalentes, c'est à dire dans des régions de faible densité $\rho(r)$. Ce calcul de densité électronique promoléculaire est au centre de ma première application de portage GPU (chapitre 4).

La densité électronique promoléculaire est la somme des densités atomiques, sans la relaxation induite par l'environnement moléculaire. Elle est obtenue en un point r de l'espace simplement en sommant les densités électroniques sphériques moyennes ρ_i des N atomes du système étudié :

$$\rho(r) = \sum_{i=1}^{N} \rho_i(r_i) \tag{3.36}$$

avec r_i la distance entre le point r de l'espace où est calculé la densité électronique $\rho(r)$, et l'atome i. Chaque densité atomique $\rho_i(r_i)$ est représentée dans le programme NCIplot par la combinaison linéaire de trois fonctions exponentielles :

$$\rho_i(r_i) = \sum_{j=1}^{3} a_{i,j} e^{-b_{i,j} r_i} \tag{3.37}$$

pour les atomes des trois premières rangées de la table périodique. Chaque atome est donc caractérisé par six paramètres (trois coefficients a et trois exposants b) qui ont été ajustés pour reproduire la densité quantique sphérique moyenne de l'atome en question. Les trois premières rangées de la table périodique sont paramétrées dans NCIplot.

Lorsque l'on porte le gradient réduit $s(r)$ comme une fonction de la densité $\rho(r)$ signée par le signe de λ_2 (seconde valeur propre de la matrice Hessienne) on obtient un graphe comme illustré figure 3.8. En l'absence d'interactions (par exemple, la partie du graphe pour $-0,1 \leq signe(\lambda_2)\rho \leq 0,1$, pour une molécule d'eau), $s(r)$ montre une forme générale croissante ou décroissante continue en $\rho^{-1/3}$. Dans ce cas, la variation de cette fonction $s(r)$ est gouvernée par la densité électronique $\rho(r)$ dans l'expression (3.35). En effet, par exemple, pour un atome isolé (donc sans aucune interaction chimique), loin du noyau, $\rho^{4/3}$ (dénominateur) et $\nabla\rho$ (numérateur) sont tous les deux petits. Mais comme le premier approche zéro plus rapidement que le second, leur ratio croît exponentiellement et tend vers l'infini lorsque les valeurs de la densité sont basses. Un raisonnement analogue montre que dans les régions à haute densité, proche des noyaux, $\rho^{4/3}$ gouverne le gradient et en conséquence, $s(r)$ diminue. Pour les systèmes moléculaires (avec interactions chimiques), le tracé 2D de $s(r)$ apporte de nouvelles informations. Des écarts au comportement exponentiel continu peuvent être observés. Par exemple, sur le graphe de gauche figure 3.8, pour une seule molécule d'eau, une chute de $s(r)$ est observée pour $\rho(r)$ signée autour de -0,22 u.a. Un décrochement observé dans la partie négative du graphe correspond à une interaction attractive tandis qu'une chute se produisant dans la partie positive du graphe caractérise la présence d'une interaction répulsive dans le système chimique. Cette information (attractif ou répulsif) est donnée par le signe de λ_2 en chaque point du graphe. En portant dans l'espace réel ces points du graphe 2D (ceux qui apparaissent dans la chute de $s(r)$), on met en évidence les régions 3D associées aux interactions moléculaires par les chimistes. Ces régions 3D peuvent être représentées pour une valeur donnée de $s(r)$ sous la forme d'isosurfaces (par l'utilisation du logiciel VMD par exemple), comme dans la figure 3.9. De plus, ces isosurfaces peuvent être colorées en utilisant la densité électronique signée. Un schéma de couleur RVB est traditionnellement utilisé pour classer ces interactions. Le rouge indique les interactions déstabilisantes, bleu les interactions attractives et vert pour les interactions faibles. Les liaisons covalentes sont identifiées par des creux de $s(r)$ à fortes densités (comme la chute observée vers -0,22 u.a pour la liaison

O-H dans H_2O figure 3.8) tandis que les dépressions à faibles densités révèlent des interactions non covalentes, telles que des contacts de vdW (van der Waals) ou des liaisons hydrogène (comme la dépression observée autour de -0,025 u.a pour le dimère d'eau de la figure 3.8 à droite). Ces dépressions du graphe $s(\rho(r))$ se traduisent alors dans l'espace du système chimique par des isosurfaces d'interaction. Sur la figure 3.9, l'analyse NCI révèle bien l'existence des deux liaisons chimiques O-H de la molécule d'eau sous la forme de "pastilles" bleu foncé, tandis que la liaison hydrogène (interaction faible) se traduit par une pastille verte située entre les deux molécules dans le cadre de droite. Il est à noter que l'emploi de la densité promoléculaire approchée restreint l'utilisation de l'approche NCI à l'étude des interactions faibles (non-covalentes) ; l'étude des liaisons fortes requiert l'emploi de calculs quantiques. Le programme NCIplot permet l'emploi de ces densités quantiques plus précises mais cette partie n'a pas été abordée durant ma thèse.

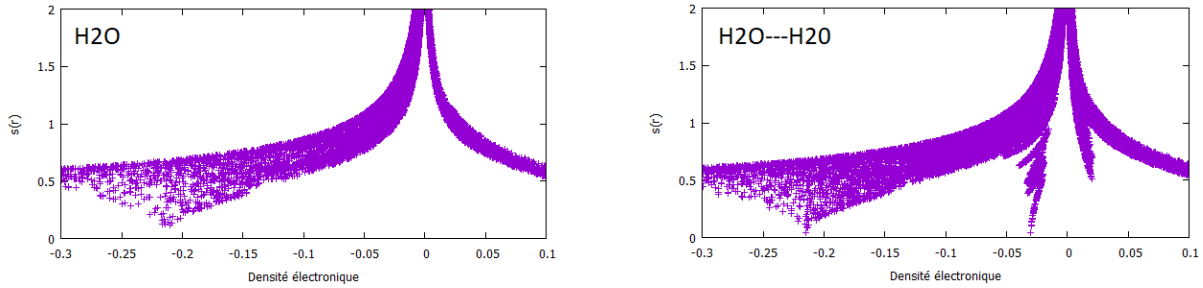


FIGURE 3.8 – Graphiques de $s(r)$ en fonction de la densité électronique promoléculaire ; à gauche, une molécule (H_2O) isolée ; à droite deux molécules (H_2O) en interaction.

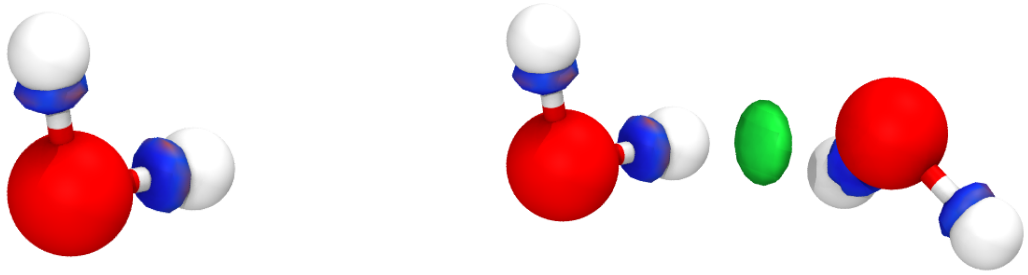


FIGURE 3.9 – Représentation (à l'aide de VMD) des systèmes étudiés figure 3.8 ; à gauche, une molècule (H_2O) isolée ; à droite deux molécules (H_2O) en interaction (isosurface verte) ; isosurfaces $s(r) = 0{,}4$ u.a colorées sur l'échelle : -15 u.a $\leq$ signe$(\lambda_2)\rho \leq 15$ u.a

D'un point de vue pratique, il est important d'avoir en tête les points suivants car ils permettent de comprendre que l'approche NCI peut potentiellement tirer profit de l'utilisation de cartes graphiques pour accélérer les calculs :

1. Une grille de points doit être construite autour du système chimique étudié. La densité de cette grille peut varier en fonction des choix de l'utilisateur qui définit le pas de grille dans les trois dimensions de l'espace. Le nombre de points peut valoir quelques milliers de points à quelques millions de points.
2. Pour chaque point de la grille, plusieurs quantités doivent être calculées :
   - La densité électronique ρ.
   - Le gradient de la densité électronique $\nabla\rho$.
   - La matrice Hessienne de la densité électronique H.

- λ_2 (diagonalisation de la matrice H).
- La densité électronique signée par (le signe de) λ_2.
- Le gradient réduit de la densité électronique $s(r)$.

3. Trois fichiers doivent être écrits :
   - Un fichier "`dat`" contenant deux colonnes qui contiennent les valeurs de la densité électronique signée et du gradient réduit pour tous les nœuds de la grille de calcul. Ce fichier permet de construire le graphe 2D du système chimique étudié (comme sur la figure 3.8).
   - Deux fichiers "`cube`", un pour la densité électronique signée et l'autre pour le gradient réduit de la densité électronique. Ces fichiers "`cube`" permettent (via VMD par exemple) de construire les isosurfaces voulues.

Nous pouvons donc voir par ces étapes majeures de l'approche NCI que la grille de calcul demande un nombre important de calculs indépendants et similaires sur des données différentes : un cadre qui semble favorable à l'utilisation de cartes graphiques. Nous avons travaillé sur cette hypothèse dans la partie 4.

3.7 L'approche IGM

En parallèle du travail réalisé ici sur l'approche NCI, un travail théorique sur l'approche NCI à été développé à Reims par les laboratoires ICMR (chimie) et CReSTIC (informatique). La méthode *Independent Gradient Model* (IGM)[42][43] est apparue pendant cette thèse, fruit du travail des deux laboratoires. Cette approche est dans la lignée de l'outil NCI pour l'identification d'interactions chimiques. La méthodologie IGM est développée pour une identification automatique des interactions intermoléculaires. Comme NCI, IGM utilise la densité électronique promoléculaire ρ précédemment définie dans l'équation 3.36 ou une densité quantique plus précise pour la description de liaisons chimiques. Ce modèle définit une densité de référence où les molécules du système n'interagissent pas. Grâce à cette fonction de référence sans interaction le modèle IGM fournit un moyen de quantifier les chutes observées précédemment dans le graphe 2D de $s(r)$. En utilisant un schéma de découplage intra/inter, un descripteur (δg^{inter}) est proposé qui définit de façon unique les régions d'interaction intermoléculaires. Une caractéristique attrayante de la méthodologie IGM est de générer automatiquement des données composées uniquement d'interactions intermoléculaires pour dessiner les représentations d'isosurface 3D correspondantes.

Les approches NCI et IGM se basent sur des quantités identiques à calculer, à savoir la densité électronique, son gradient, la matrice Hessienne de la densité et les valeurs propres de cette matrice Hessienne. Ces quantités sont, pour les deux approches, à calculer pour tous les nœuds de la grille. Nous pouvons donc voir que tout travail d'accélération par utilisation de cartes graphiques pourra être transposé sur l'approche IGM.

3.8 Conclusion

Nous avons, au cours de ce chapitre 3, pu voir un aperçu de l'étendu des diverses théories, méthodologies et approches dont regorge la chimie théorique, juste en introduisant les algorithmes qui nous intéressent pour ce travail de recherche : les approches DFTB / FMO / PCM et NCI.

L'hypothèse est que ces approches peuvent tirer profit d'une accélération par l'utilisation de la technologie *manycore* des cartes graphiques. Ce travail de recherche cherche donc par le biais de l'étude de l'utilisation des cartes graphiques sur les approches NCI

(chapitre 4) et DFTB / FMO / PCM (chapitre 5) à apporter sa contribution à la chimie théorique par l'accélération sur GPU de ces approches.

Pour l'approche combinée DFTB / FMO / PCM (méthode présente dans le logiciel GAMESS examiné dans ma thèse chapitre 5), la partie FMO de fragmentation du système étudié fait que l'approche tire profit des technologies multi-cœurs et donne une indication sur un possible passage aux technologies *manycore*. Le travail sur cette méthode est détaillé dans le chapitre 5 de cette thèse, par le biais du logiciel GAMESS[44]. La méthode PCM (effet du solvant) est requise pour obtenir la convergence du cycle SCF détaillé dans ce chapitre pour les méthodes HF et DFT. Le chapitre 5 s'intéresse donc à l'approche combinée DFTB / FMO / PCM dans sa globalité.

L'approche NCI (adaptée sur GPU dans le chapitre 4) semble elle, bien adaptée à l'utilisation de cartes graphiques car un grand nombre de calculs similaires et indépendants sont nécessaires au sein de la grille. Ce genre de contexte est a priori favorable à l'architecture d'une carte graphique. Le travail de portage GPU réalisé sur cette approche NCI au cours de cette thèse est détaillé dans le chapitre 4 et tente de vérifier cela. Nous savons aussi maintenant que si des gains sont obtenus par l'utilisation de cartes graphiques sur l'approche NCI, ces gains pourront être réutilisés sur l'approche IGM car la méthodologie des approches NCI et IGM (développée à Reims) est similaire.

Nous pouvons donc maintenant nous intéresser aux travaux de portage GPU qui ont été réalisés au cours de cette thèse sur les différentes approches que nous venons d'évoquer, avec pour but d'obtenir des accélérations grâce aux cartes graphiques par rapport à une implémentation CPU multi-cœurs de référence. Nous commencerons par détailler le travail réalisé sur l'approche NCI (chapitre 4) par le biais du logiciel NCIplot et terminerons par le travail réalisé sur le logiciel GAMESS (chapitre 5) qui implémente l'approche combinée DFTB / FMO / PCM.

Chapitre 4

Accélération de l'approche NCI sur architecture GPU

Dans cette partie nous décrivons dans le détail notre travail de recherche réalisé sur l'algorithme NCI publié en 2010[45]. La méthode NCI est introduite précédemment dans le chapitre 3 de cette thèse, section 3.6. Un article a été publié sur le travail de thèse réalisé dans ce chapitre [2].

Ces travaux se basent sur l'implémentation NCIplot de Julia CONTRERAS-GARCIA et coll. diffusée depuis 2011[39]. Ce code est écrit en Fortran et prévoit l'utilisation d'un processeur multi-cœur par le biais de l'interface de programmation OpenMP. Dans le cadre de ce travail de recherche, il a été complètement ré-écrit en C pour le portage sur GPU.

Dans cette partie notre objectif est de fournir une implémentation GPU optimisée de l'approche NCI.

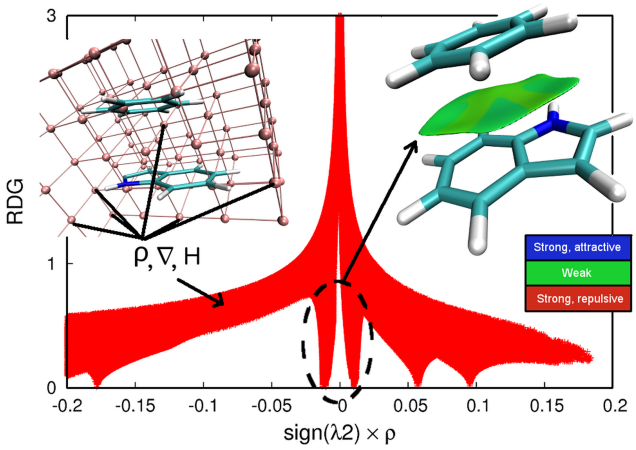


FIGURE 4.1 – Flux des travaux d'un calcul NCI en partant du calcul des densités électroniques ρ en chaque point de la grille (à gauche) jusqu'à la visualisation de l'isosurface du gradient réduit de la densité (à droite). [2]

Le cadre général de l'approche NCI est la topologie de la densité électronique $\rho(r)$ calculée en chacun des nœuds de la grille de travail. La connaissance de la densité électronique et de ses dérivées permet l'identification d'interactions moléculaires dans l'espace réel, basées sur les dépressions qui apparaissent à basse densité ρ dans le tracé du gradient

réduit de la densité $s(\rho)$; pour rappel :

$$s(\rho) = \frac{||\nabla\rho||}{C\rho^{\frac{4}{3}}} \tag{4.1}$$

avec $C = 2(3\pi^2)^{1/3}$ une constante.

4.1 Codes CPU de l'approche NCI

Les grandes étapes de l'algorithme NCI sont illustrées par la figure 4.1 pour le complexe d'indole-benzène. Une grille régulière de points est d'abord construite autour du système. Connaissant les coordonnées cartésiennes de ces points et des atomes : la densité électronique ρ, ses dérivées premières et secondes sont calculées et collectées en chaque point de la grille englobant les molécules en interaction comme représenté sur la figure 4.1. De plus, le calcul de la seconde valeur propre λ_2 de la matrice Hessienne H de la densité électronique est nécessaire pour obtenir le caractère attractif ($\lambda_2 < 0$) ou répulsif ($\lambda_2 > 0$) des interactions en ces points. Un calcul de diagonalisation d'une matrice (3,3) est donc nécessaire en chaque point de la grille. Puis, le programme NCIplot génère trois fichiers de sortie. Le premier fichier (nciplot.dat) contient deux colonnes de données correspondant à la densité électronique signée $signe(\lambda_2) \times \rho$ et $s(\rho)$ (dénommé RDG), respectivement. Il permet de construire le tracé 2D qui révèle les interactions moléculaires lorsque des dépressions apparaissent comme illustrée figure 4.1 dans l'allure générale du graphe $s = f(\rho)$. De plus, deux fichiers cube sont générés (`density.cube` et `rdg.cube`) permettant le rendu des isosurfaces (grâce à un logiciel de visualisation comme VMD[46], par exemple) comme illustré figure 4.1. Ils contiennent des données volumétriques de grille (densité électronique signée et $s(\rho)$ respectivement) ainsi que les positions atomiques. Un fichier cube respecte un certain format[39] permettant le calcul et l'affichage d'isosurfaces par des logiciels comme VMD[46] ou ParaView[47]. Lorsque l'on reporte dans l'espace réel les points localisés dans les dépressions de la représentation 2D précédente, des isosurfaces d'interactions à des valeurs $s(\rho)$ constantes peuvent être tracées dans la représentation 3D du système chimique en utilisant le fichier `rdg.cube`. De plus, ces isosurfaces peuvent être colorées en fonction de la densité électronique signée (issue du fichier `density.cube`). Une échelle de couleurs RGB est traditionnellement utilisée pour classer les interactions où le rouge signifie répulsif, bleu pour attractif et vert pour les interactions faibles (van der Waals en général).

Calcul de la densité électronique

La pierre angulaire de l'analyse NCI est la densité électronique qui doit être déterminée en chaque nœud de la grille. La chimie quantique semble être le meilleur moyen de prédire cette information. Cependant, de tels calculs quantiques coûteux en termes de CPU sont rarement réalisables pour de grands systèmes chimiques comme par exemple une protéine hébergeant un ligand. Heureusement, il a été montré que les caractéristiques topologiques de densité électronique dans ces régions NCI (à faible densité ρ) sont très stables par rapport à la méthode utilisée pour calculer la densité électronique[40][41]. Une approche alternative intéressante est alors de calculer la densité promoléculaire. Elle est calculée en un point (x, y ,z) de la grille en sommant la densité atomique neutre moyenne sphérique ρ_i centrée sur les positions (x_i, y_i, z_i) des N atomes composant le système : $\rho(x, y, z) = \sum_{i=1}^{N} \rho_i(r_i)$ (avec $r_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}$). Les densités atomiques isolées

ρ_i sont obtenues à partir d'une combinaison linéaire de trois fonctions exponentielles simples : $\rho_i(r_i) = \sum_{j=1}^{3} a_{i,j} e^{-b_{i,j} r_i}$ avec $a_{i,j}$ et $b_{i,j}$ ajustés préalablement au calcul NCI (paramètres stockés dans le programme) pour restituer au mieux la densité électronique ab initio moyenne sphérique pour un atome spécifique. Il est à noter que les molécules de ligand et de protéine n'étant composées que d'un nombre limité d'éléments atomiques différents (en général : H, C, N, O, S), les données nécessaires pour calculer la densité électronique sont limitées. Le gradient (trois composantes $\nabla\rho_x$,$\nabla\rho_y$,$\nabla\rho_z$) et la matrice Hessienne H réelle symétrique 3×3 (six composantes à calculer) découlent de la définition de $\rho(r)$ (voir les formules mathématiques en annexe H). Dans la procédure NCI, pour différencier le caractère (attractif ou répulsif) des interactions intermoléculaires le signe de la seconde valeur propre λ_2 de la matrice Hessienne provenant de la densité électronique est nécessaire. Ceci requiert de diagonaliser H en chaque point de la grille.

Taille du problème

Bien que l'utilisation de la densité promoléculaire évite des calculs quantiques très coûteux, cela peut prendre du temps en raison de la taille du système chimique et du pas de grille sélectionné. Dans l'étude de complexes ligand-protéine, protéine-protéine ou même d'une molécule solvatée[48] le nombre d'atomes peut être de plusieurs milliers, la taille de la grille peut aller jusqu'à une douzaine d'angströms et un pas d'espacement de grille aussi petit que 0,02Å, résultant en des calculs numériques intensifs (plusieurs millions de points de grille).

D'un point de vue calcul, l'analyse NCI d'une paire de molécules en interaction est un problème de dimensions 4 : 3 dimensions (K, L, M) pour la grille et une dimension (N) sur les atomes du système, donnant lieu à plusieurs implémentations de GPU possibles.

Algorithme de l'approche NCI

Le chemin général d'exécution de l'approche NCI, illustré figure 4.2, est le suivant : après avoir lu les données des fichiers d'entrée (coordonnées cartésiennes des atomes des deux molécules interagissants et pas de la grille), le programme détermine d'abord automatiquement la taille de la grille englobant la petite molécule, généralement le ligand. En effet, il est possible de limiter le nombre de points de grille à traiter en ne traitant que les interactions dans un rayon de l'une des deux molécules. Ceci n'exclut pas les atomes de la deuxième molécule de la boucle de calcul sur les atomes du système (dimension N dans l'équation (3.36)). Cette procédure est particulièrement recommandée pour les complexes ligand (petit)-protéine (grand système). Il est conseillé de choisir le ligand comme molécule centrale. Tout d'abord, le programme calcule la taille de boîte minimale enfermant le ligand (le plus petit système). Puis NCIplot prévoit une zone tampon de 4Å qui est ajoutée dans les trois directions de la boîte pour pouvoir décrire les interactions avec la seconde molécule. En effet, au delà de 4Å , les interactions deviennent très faibles et aucune dépression n'est alors observée dans le graphe 2D $s(\rho(r))$. Ensuite, le programme calcule la densité électronique $\rho(x, y, z)$ en chaque nœud de cette grille.

Avant de passer aux calculs complémentaires (dérivées premières et secondes puis diagonalisation de la matrice Hessienne) un test doit être effectué pour éviter des calculs inutiles. En vue de porter le calcul de l'approche NCI sur GPU, une autre considération doit en effet être prise en compte : le calcul de la fraction de densité électronique provenant de chaque molécule en interaction. En effet, la visualisation spatiale 3D de NCI entre deux molécules (à l'aide de logiciels comme VMD [46]) nécessite de différencier les

points de grille associés aux situations intermoléculaires des autres points (indésirables) correspondant aux régions intramoléculaires qui ne nécessitent pas d'être étudiées. Pour cela, dans le code original, il est possible de rejeter les nœuds de la grille pour lesquels plus d'une fraction (valeur de seuil par défaut est 0,95) de la densité promoléculaire totale provient d'une seule molécule (A ou B). Quand ce seuil est dépassé on a affaire à un point de la grille "appartenant" au domaine d'une des deux molécules mais pas "entre" les deux molécules. Le résultat de ce test n'est malheureusement connu qu'après avoir calculé la densité $\rho(r)$. On peut éviter toute fois le calcul des dérivées. La densité ρ doit donc être calculée en séparant la somme de la formule (3.36) en deux parties A et B : $\rho(x,y,z) = \sum_{i=1}^{N_A} \rho_i(r_i) + \sum_{i=1}^{N_B} \rho_i(r_i) = \rho_A + \rho_B$ et les calculs suivants (gradient, Hessienne, valeurs propres...) ne seront effectués que si le critère de la fraction est rempli. D'un point de vue algorithmique, ce critère introduit une condition qui va engendrer un traitement irrégulier des nœuds de la grille.

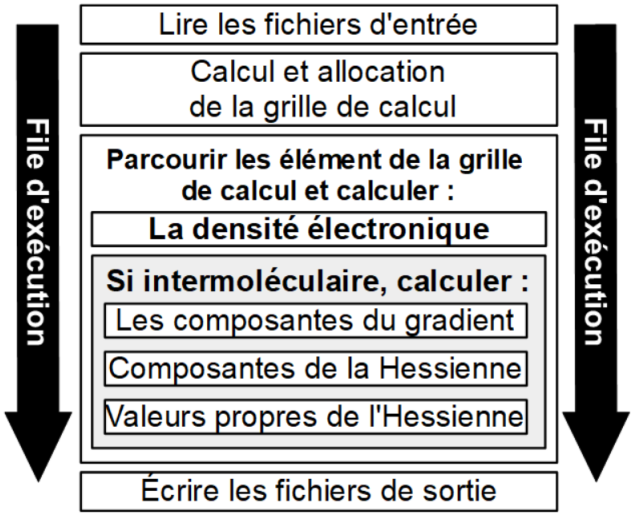


FIGURE 4.2 – Algorithme de l'approche NCI.

L'approche NCI implique des manipulations simples d'un grand nombre de données plutôt que de faire des opérations complexes sur un petit ensemble de données. Un environnement qui semble favorable à l'utilisation de GPU.

La section suivante analyse l'implémentation de l'approche NCI existante : le logiciel NCIplot.

Code fortran NCIplot

Le programme NCIplot de Julia CONTRERAS-GARCIA et coll., diffusé depuis 2011[39], cherche a être le plus générique possible, permettant son utilisation sur la plus large gamme de matériels possible. En conséquence avant portage sur GPU, des gains CPU pourraient être obtenus mais risquent à la fois de nécessiter du temps de développement et d'impacter l'aspect générique du code (ou bien la simplicité d'installation du logiciel).

La parallélisation est actuellement réalisée en insérant une seule instruction OpenMP «parallel do» avant la boucle triple imbriquée sur les nœuds de la grille spatiale. Seule la boucle externe est parallélisée. Aucune parallélisation n'a été envisagée par les auteurs du code pour la boucle sur les atomes, ce qui évite la latence de communication entre les cœurs. Nous avons évalué, sur NCIplot (code Fortran), l'influence de distribuer les

processus légers du CPU sur les boucles imbriquées sur la grille spatiale. Cela a entraîné une légère amélioration des performances d'exécution du processeur de 4%, donnant un indice sur le fait que la performance CPU de NCIplot peut être améliorée en investissant du temps dans le développement.

Écriture d'une version C de NCIplot

Plusieurs raisons nous ont conduits à réécrire le code NCIplot en langage C. Tout d'abord, mon travail s'inscrit dans le projet AlgoGen[31] de simulation de *docking* moléculaire. Les seules interactions moléculaires à considérer sont donc "intermoléculaires" qui surviennent à basse densité électronique. Ainsi, le calcul simplifié de la densité électronique promoléculaire (décrit section 3.6 du chapitre précédent) est suffisant. Or le programme d'origine NCIplot en Fortran est beaucoup plus complet en traitant aussi la densité "quantique", rendant le code complexe par rapport à nos besoins. Afin de disposer d'un code de départ plus simple, avant le portage GPU, nous avons donc pour cette première raison réécrit l'algorithme NCI en C. Ensuite, notons que nous avons fait le choix de changer de langage de programmation dans le but de nous rapprocher des portages GPU écrits en C et utilisant CUDA. Ce changement de langage est la seconde raison de la réécriture du code en langage C. Enfin les fonctions de diagonalisation utilisées pour calculer les trois valeurs propres de la matrice Hessienne de la densité électronique diffèrent dans les deux codes CPU (Fortran et C). Alors que dans le code en C nous considérons une méthode analytique reposant sur la formule de Cardan, le package Fortran EISPACK[49] (algorithme itératif QL) est utilisé dans NCIplot. Ce choix de changer l'algorithme pour obtenir les trois valeurs propres est motivé par le portage à venir sur GPU. En effet une méthode itérative introduirait potentiellement de la divergence au sein d'un GPU, entre les processus légers d'un même bloc. Là où, une méthode analytique n'introduit pas cette divergence entre les processus légers.

Dans la suite, nous comparons notre version CPU en C à la version originale en Fortran ; pour cela plusieurs installations du code C ont été réalisées avec différentes options de compilation et différents compilateurs.

4.2 Compilateurs et options de compilation des codes CPU

Nous disposons du code d'origine en Fortran (NCIplot), mais aussi de notre version C complètement ré-écrite qui peut être compilée en simple ou double précision et en utilisant différents compilateurs (gnu, Intel ou Portland group) et options de compilation. Ces différentes possibilités multiplient les choix envisageables pour obtenir une installation CPU de référence qui servira pour la comparaison aux exécutions sur GPU. D'autres implémentations NCI existent au sein de logiciels (Multiwfn[50] et Jmol[51]), mais ils ne sont pas dédiés qu'au calcul NCI.

Le code C sera comparé au programme Fortran original NCIplot (version 3.0) mis à la disposition des utilisateurs sur le site[52]. Plusieurs compilateurs ont été utilisés pour obtenir une référence CPU la plus efficace possible. Nous comparons les performances obtenues du code C compilé avec trois compilateurs différents (gcc, icc et pgcc), précédemment évoqués section 2.1 et le logiciel NCIplot compilé avec un compilateur (ifort).

Dans le cas de l'implémentation NCIplot (Fortran), nous avons utilisé le compilateur d'Intel (ifort), avec lequel deux installations principales ont été réalisées (chacune tirant

profit du parallélisme d'OpenMP par l'option -qopenmp). La première utilise l'option de compilation (-O2) que préconise NCIplot lors du téléchargement. Après avoir essayé manuellement plusieurs jeux d'options de compilation, il s'avère que nous avons obtenu de meilleurs performances en ajoutant l'option *-ipo* (*Interprocedural Optimisation*), qui permet l'optimisation interprocédurale entre les fichiers (concrètement, le compilateur étend l'utilisation de fonctions *inline* à des fonctions définies dans des fichiers séparés). Le binaire le plus efficace pour NCIplot a donc été obtenu avec :

- **ifort** : `{O2 {openmp {ipo`

Pour notre implémentation en C, trois compilateurs (gcc, pgcc et icc) ont été étudiés. De même, plusieurs jeux d'options ont été testés. Les binaires les plus efficaces (pour chaque compilateur) de notre code C ont été obtenus avec les options suivantes :

- **gcc** : `{O3 {march=native -fopenmp {W {Wall {ansi {pedantic`
- **pgcc** : `{O2 {mp=allcores {Mprefetch {Mfprelaxed {fast {Mipa=fast,inline {Msmartalloc`
- **icc** : `{O2 {openmp {W {Wall {ansi {pedantic march=native {fast`

Ce sont donc ces binaires (un pour NCIplot et trois pour notre code C) qui vont nous permettre de comparer les performances des codes CPU C et Fortran, afin d'obtenir une référence CPU pour évaluer nos portages GPU. De nombreuses combinaisons d'options sur les différents compilateurs ont été testées afin de définir une référence la plus performante possible pour évaluer les performances des portages GPU. Nous présentons uniquement les meilleurs résultats pour chaque compilateur utilisé.

Pour comparer ces binaires (et par la suite les portages GPU), nous avons créé plusieurs systèmes chimiques.

4.3 Systèmes chimiques testés

Pour évaluer les performances des différentes implémentations réalisées de l'algorithme NCI, un jeu de test (systèmes chimiques) a été créé. Ces systèmes chimiques représentent 36 complexes provenant des combinaisons possibles d'assemblage de 6 ligands avec 6 protéines. Les 6 ligands sont nommés L1, L2, L3, L4, L5 et L6 et sont respectivement composés de 3, 6, 12, 24, 48 et 96 atomes. Les ligands ont été choisis pour contenir un nombre d'atomes évoluant d'un facteur 2 entre deux ligands successifs. Également, ils contiennent des éléments chimiques souvent rencontrés dans les molécules du vivant ou d'intérêt thérapeutique, comme l'hydrogène, le carbone, l'azote, l'oxygène ou le fluor. Ce sont des molécules de taille relativement petite par rapport aux six protéines amenant à des complexes ligand-protéine typiquement rencontrés dans les simulations de *docking* moléculaire. Les grilles de calcul ont donc été construites autour de ces ligands. Les 6 protéines sont nommées P1, P2, P3, P4, P5 et P6 et sont respectivement composés de 135, 239, 394, 722, 907 et 1245 atomes. Concernant la protéine, nous avons utilisé le logiciel AlgoGen (et son interface graphique jBox) pour définir une première boîte de *docking* autour du site actif de la protéine phosphodiesterase 4 (isoforme PDE4D, PDB ID : 1MKD, 3310 atomes). Le logiciel AlgoGen conserve alors uniquement les atomes à l'intérieur de cette boîte (et sature chimiquement par des hydrogènes les points de coupure). Cela conduit tout d'abord à la molécule P6 (1245 atomes) en utilisant une grande boîte de *docking*. Partant de là, cette procédure a été répétée avec des boîtes de plus en plus petites, donnant les protéines P5 à P1. La figure 4.3 donne une représentation des 6 ligands et 6 protéines ainsi que le complexe le plus gros L6-P6 constitué du ligand L6 et de la protéine P6. Ces 36 complexes forment les systèmes de référence sur lesquels des évaluations NCI

ont été réalisées afin d'évaluer les performances des différentes implémentations ; de plus quatre pas de grille différents de 0,2 Å, 0,1 Å, 0,05 Å et 0,025 Å ont été utilisés, formant un total de 144 instances d'évaluations.

Il faut savoir que les utilisateurs de la méthode NCI utilisent en général une finesse de grille de 0,1 Å, voir de 0,05 Å tout au plus. Les quatre pas de grille choisis ici couvrent bien les besoins habituels des utilisateurs. La taille des ligands étudiés ici combinée aux différents pas de grille a conduit à des grilles contenant entre 88 704 et 648 873 680 nœuds.

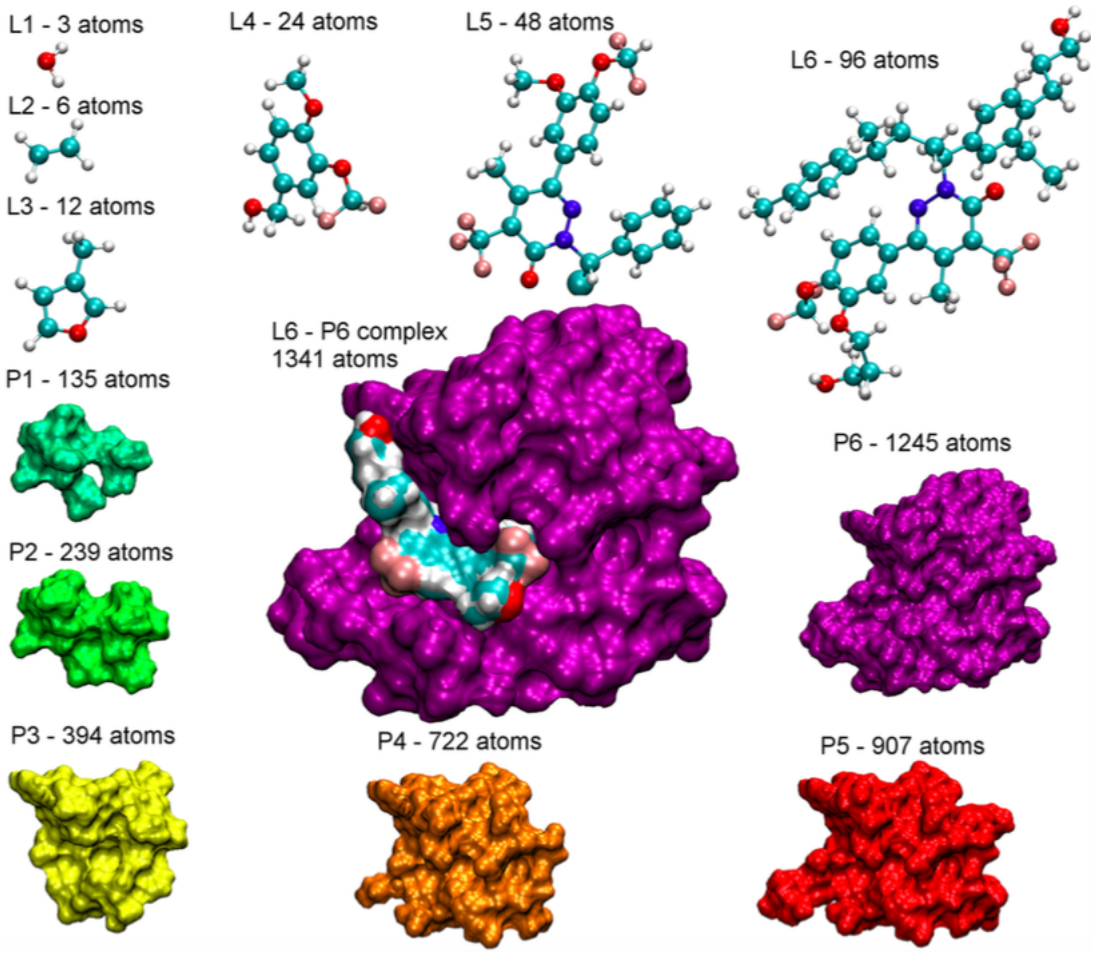


FIGURE 4.3 – Représentation des 6 ligands et 6 protéines constituant les 36 systèmes de référence pour évaluer les implémentations de l'algorithme NCI

4.4 Comparaison des implémentations CPU de NCIplot

Pour comparer les deux implémentations CPU (Fortran et C) que nous avons à notre disposition, plusieurs installations (une pour NCIplot et trois pour notre code C) décrites section 4.2 sont utilisées. Dans cette section nous allons regarder l'évolutivité des deux implémentations CPU (Fortran et C) en fonction du nombre de cœurs de calcul utilisés. L'ensemble des temps d'exécutions avec 2x8 cœurs (Ivy bridge 2,6 GHz) est donné dans l'annexe A. Puis en comparant les performances obtenues des différentes installations CPU nous déterminerons l'installation qui nous servira de référence CPU. Cette référence nous permettra d'évaluer les performances des différents portages GPU. Pour rappel (chapitre

1), il est important d'avoir une référence CPU parallèle aussi performante que possible pour évaluer la performance d'un code GPU.

Les temps d'exécutions CPU de ces quatre installations (ifort, gcc, pgcc et icc) avec 1, 2, 4, 8 et 16 processus légers sont dans le tableau 4.1.

Compilateur/#Cœurs	1	2	4	8	16
Fortran (`ifort -O2 -qopenmp -ipo`) *version Intel 2016*	2512/**1**	1254/**2**	630/**4**	315/**8**	154/**16**
gcc (-O3 -march=native -fopenmp) *version gcc 5.1.0*	4066/**1**	2039/**2**	1041/**4**	530/**8**	273/**15**
pgcc (pgcc –mp=allcores –Mprefetch –Mfprelaxed –fast –Mipa=fast,inline –Msmartalloc) *version pgi 2016*	2225/**1**	1191/**2**	563/**4**	286/**8**	146/**15**
icc (icc –openmp –march=native –fast) *version Intel 2016*	963/**1**	470/**2**	237/**4**	127/**8**	63/**15**

Tableau 4.1 – Performances obtenues sur un nœud de calcul avec différents compilateurs et OpenMP ; Temps total d'exécution de l'algorithme NCI en secondes pour un système chimique de 770 atomes et 37 545 966 nœuds de grille. L'accélération est notée en gras.

Le premier constat que nous pouvons faire est que toutes les installations tirent profit de l'utilisation de plusieurs processeurs, avec des facteurs d'accélération concordant avec les ressources mises en œuvre. L'installation évoluant relativement le moins bien étant notre implémentation avec un facteur d'accélération de 15.

En revanche lorsque nous regardons les temps d'exécution absolus, nous pouvons constater que l'installation de notre code (en C) avec icc est clairement plus performante en prenant entre 963s et 63s pour respectivement 1 et 16 cœurs de calcul contre 2512s et 154s pour NCIplot (ifort). Cette analyse reste vraie pour tous les systèmes chimiques traités.

En effet, le graphe figure 4.4 permet de constater que la version C compilée avec icc (Intel) est toujours clairement plus performante que l'installation la plus performante du logiciel NCIplot avec ifort (Intel).

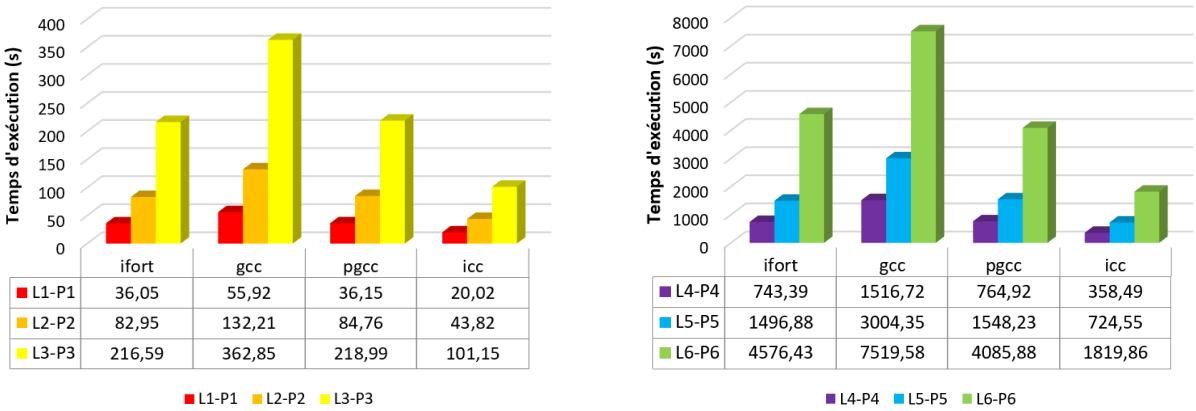


FIGURE 4.4 – Temps en secondes de quatre des installations CPU (ifort, gcc, pgcc et icc) évaluées sur six complexes chimiques avec le pas de grille de 0,025Å

Il peut être ardu de définir une référence pour l'évaluation de performances dans le cadre d'un travail d'optimisation (dans notre cas par l'utilisation de GPU). Nous pouvons voir que les performances peuvent varier drastiquement en fonction du compilateur et des options utilisées. Pour nous, dans le cadre de l'approche NCI, l'installation la plus optimale est donc générée par le compilateur icc sur notre code C pour les processeurs Intel Ivy Bridge comme le montrent le tableau 4.1 et le graphe 4.4.

4.5 Évolutivité du code CPU de référence en C

La figure 4.5 montre que le temps de calcul obtenu en utilisant notre installation de référence CPU croît linéairement à la fois avec le nombre de nœuds de la grille NCI (dimensions K, L, M) et avec le nombre d'atomes (dimension N).

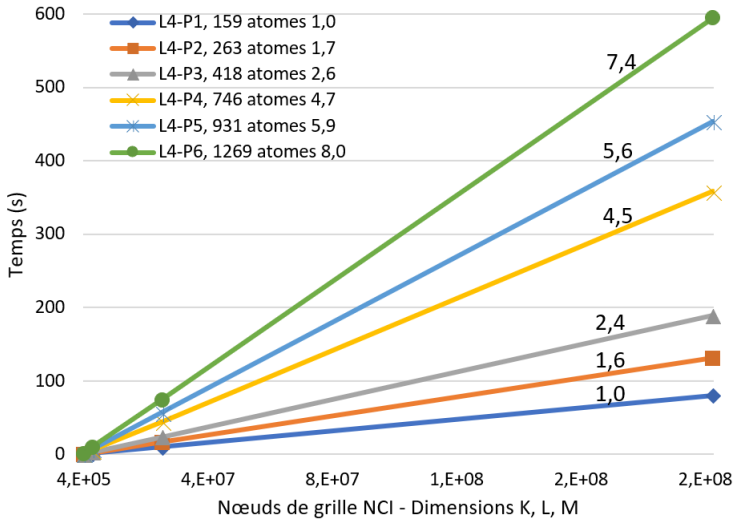


FIGURE 4.5 – Évolutivité de notre code NCI de référence CPU en C (icc, 16 cœurs CPU) ; la pente normalisée est indiquée au-dessus de chaque tracé, elle est à comparer aux rapports normalisés de taille des complexes étudiés (reportés en face du nombre d'atomes) [2]. Six complexes sont examinés, impliquant le ligand L4 et les six protéines P1 à P6.

Nous avons une référence avec un comportement correspondant à nos attentes, avec une évolutivité faible (*weak scaling*) augmentant linéairement avec la taille du problème selon quatre dimensions (taille de grille et nombre d'atomes). Maintenant que notre code de référence CPU est défini, nous allons l'analyser, afin de déterminer les zones du code qui représentent une part importante du temps d'exécution total, en vue du portage sur GPU.

4.6 Analyse préliminaire de l'installation CPU de référence

Commençons par rappeler les étapes majeures de l'approche NCI, illustrées figure 4.2. Nous pouvons en définir quatre principales :

- Lecture des fichiers d'entrée.
- Définition de la grille de calcul.
- Calcul des éléments de la grille.
- Écriture des fichiers de sortie.

La première étape débute par la lecture d'un premier fichier (`param.nci`) au format ASCII contenant le nombre de molécules à traiter (une ou deux, au choix de l'utilisateur), les chemins d'accès aux fichiers d'extension `.xyz` des molécules, ainsi que l'ensemble des paramètres définissant la grille de calcul pour configurer la méthode NCI. Le ou les fichiers de coordonnées spatiales (`.xyz`) des molécules à traiter sont ensuite lus.

La seconde partie (construction de grille) est une étape préliminaire importante au calcul de la méthode NCI, mais n'est pas limitante en terme de temps de calcul. Les paramètres donnés par l'utilisateur définissent à la fois la localisation de la grille dans l'espace (en général autour du ligand) mais aussi la finesse de la grille.

La troisième partie du code regroupe la quasi totalité des calculs à réaliser. Les calculs se portent sur la grille définie juste avant. Pour chaque nœud de la grille (caractérisé par sa position x, y, z dans l'espace) plusieurs quantités sont à calculer :

- La densité électronique promoléculaire.
- Le gradient de la densité.
- La matrice Hessienne de la densité électronique.
- Les valeurs propres de la matrice Hessienne.

Cette troisième étape semble clairement la partie limitante.

Avec l'utilisation de l'analyseur de code MAQAO nous avons confirmé que la partie couteuse en temps de calcul est cette troisième partie avec plus de 99% du temps d'exécution. Un résumé d'une telle analyse est disponible dans le tableau 4.2.

Le calcul des exponentielles (__svml_exp4_e9, __svml_exp4 et __svml_dexp_cout_rare) constitue au final 59% (56,63% + 1,54% + 0,76%) de l'exécution totale. Le calcul des gradients (`computeGradHess`) et des éléments de la matrice Hessienne représentent 14% du temps total d'exécution. 27% de l'exécution totale est composée (`computeGrid`) du parcours des éléments et du calcul des distances.

Nom de la fonction	Temps total (%)	Temps Min (s)	Temps Max (s)	Temps moyen (s)
__svml_exp4_e9	56,63	96,86	126,24	110,63
computeGrid	26,73	50,36	54,24	52,21
computeGradHess	13,99	12,34	42,16	27,32
__svml_exp4	1,54	2,62	3,42	3,00
__svml_dexp_cout_rare	0,76	0,86	2,24	1,49
computeSortLambdas	0,04	0,02	0,16	0,08

Tableau 4.2 – Résumé d'un profilage par le logiciel MAQAO d'une exécution de notre code C de l'approche NCI sur le complexe L4-P4 avec un pas de 0,025Å.

Lorsque l'on regarde dans le détail, nous pouvons constater que le calcul des valeurs propres (fonction *computeSortLambdas*) est finalement une partie mineure du calcul avec 0,04% du temps de calcul total dans l'exemple présenté tableau 4.2.

La quatrième partie du code écrit les trois fichiers contenant les résultats du calcul. C'est donc sur l'évaluation de la grille que les efforts sont portés dans la suite dans ce travail de thèse. L'utilisation de cartes graphiques pour accélérer l'approche NCI semble donc prometteuse. En effet, cette analyse montre que le temps d'exécution est quasiment compris dans le calcul des différentes quantités de la grille de calcul. Or les éléments de cette grille peuvent être calculés indépendamment et les tâches à réaliser (calcul des distances, évaluation des fonctions exponentielles...) sont similaires sur des données différentes, rendant le contexte favorable pour l'utilisation de cartes graphiques.

4.7 Description des portages GPU réalisés pour accélérer l'approche NCI

Un fait qui apparaît immédiatement lorsque l'on cherche à porter un algorithme sur GPU, est le nombre important de façons de s'y prendre. Des solutions sur GPU peuvent même s'avérer moins performantes que l'utilisation de CPU. Il faut parfois, malheureusement rencontrer l'échec pour se rendre compte qu'une implémentation est inefficace.

Une caractéristique importante à prendre en compte lors de utilisation de GPU est qu'il faut gérer au mieux la mémoire : mettre sur le GPU les informations nécessaires à la réalisation du calcul ainsi que rapatrier les résultats sur le CPU lorsque nécessaire. Ceci se traduit dans notre cadre par l'ajout de deux étapes importantes aux parties majeures précédemment évoquées :

- Lecture des fichiers d'entrée.
- Définition de la grille de calcul.
- **Allouer et copier les éléments sur le/les GPU.**
- Calcul des éléments de la grille.
- **Copier les résultats du GPU vers le CPU.**
- Écriture des fichiers de sortie.

Trois principes majeurs de portages sont réalisés et décrits dans cette section. Le dernier (V_{HY}) étant le plus performant des trois versions.

Répartition du calcul des contributions des atomes au niveau du bloc GPU : version V_{AT}

Historiquement ce portage GPU V_{AT} ("AT" pour atomes) a été le premier envisagé et implémenté. Sur un nœud donné de la grille les contributions des atomes (pour les

différentes quantités à calculer, ρ, le gradient de ρ et les coefficients de la matrice Hessienne H) étant indépendantes les unes des autres, paralléliser ce calcul apparaît comme une possibilité intéressante. Le principe de cette première version V_{AT} est donc de créer une grille CUDA tridimensionnelle correspondant à la grille de calcul NCI, où chaque bloc CUDA correspond à un nœud de la grille de calcul NCI. Dans un bloc donné, toutes les contributions (à ρ, au gradient de ρ ou aux coefficients de la matrice Hessienne H) sont alors calculées par les processus légers du bloc en se répartissant les différents atomes constituant le système étudié. Un processus léger calcule donc la contribution d'un atome à l'une des quantités (ρ, $\frac{\partial\rho}{\partial x}$, $\frac{\partial\rho}{\partial y}$, $\frac{\partial\rho}{\partial z}$ et les éléments de la matrice H) à calculer au cours du processus de l'approche NCI. Ce principe est illustré dans la figure 4.6.

Première implémentation de la version V_{AT} en fixant le nombre de processus légers au nombre d'atomes du problème : V_{AT1}

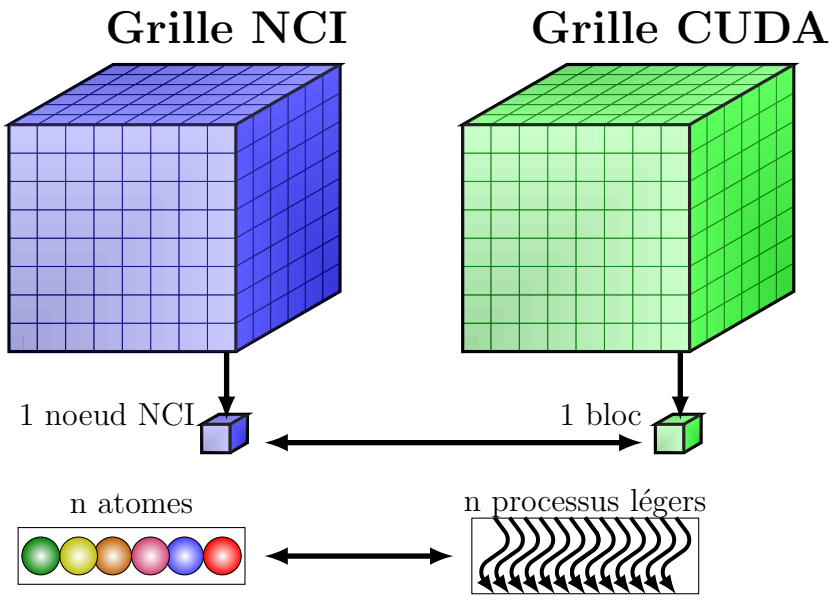


FIGURE 4.6 – Représentation de l'implémentation V_{AT1} d'un calcul NCI sur GPU.

La première implémentation de cette version (V_{AT1}) assigne donc à chaque processus léger t_i un atome de tel sorte que le nombre de processus légers est égal au nombre d'atomes. Chaque processus léger calcule la contribution atomique ρ_i de son atome à la densité électronique totale du nœud (x, y, z) en utilisant les paramètres a_{ij} et b_{ij} stockés en mémoire constante. La contribution atomique ρ_i est alors stockée en mémoire partagée pour être sommée ensuite, une fois toutes les contributions atomiques calculées sur le nœud en cours. Sur GPU, cette somme peut être calculée en parallèle de plusieurs manières différentes, plus ou moins efficaces. Il est important d'y attacher de l'importance car cette somme est effectuée sur chaque nœud de la grille pour chaque quantité calculée (ρ, $\frac{\partial\rho}{\partial x}$, $\frac{\partial\rho}{\partial y}$, $\frac{\partial\rho}{\partial z}$ et les coefficients de la matrice H). La méthode la moins performante mais la plus simple à mettre en place est l'utilisation d'opérations dites "atomiques" (disponibles avec CUDA) qui permettent d'éviter les conflits en sérialisant les accès à la mémoire. La problématique est que sérialiser les opérations revient à perdre l'intérêt majeur des cartes graphique qu'est le parallélisme. Une autre solution pour réaliser cette somme est d'utiliser une opération parallèle de réduction de type somme (inspirée du document NVIDIA[53])

décrite dans l'annexe F. L'avantage est de tirer profit du parallélisme pour une grande partie des sommes à réaliser. Cette somme reste cependant une étape limitante car elle empêche l'utilisation optimale (le nombre d'éléments à calculer diminuant à chaque étape et les ressources disponibles restant les mêmes à chaque étape) du parallélisme disponible sur les cartes graphique.

L'algorithme 1 de la version GPU V_{AT} de NCI présenté ci-dessous résume le noyau exécuté. Le premier processus léger calcule en mémoire partagée (ligne 2) les coordonnées du nœud (x, y, z), qui est traité par le bloc. Avant le début des autres calculs nécessitant les distances entre ce nœud (x, y, z) et chaque atome, les processus légers sont synchronisés (ligne 3) pour attendre le fin du calcul précédent du premier processus léger. Chaque processus léger i du bloc peut alors (ligne 4) calculer en local "sa" distance entre le nœud (x, y, z) et son atome i. Avec cette distance et les paramètres a_{ij} et b_{ij} (stockés en mémoire constante) le processus léger peut calculer (ligne 5) la contribution de cet atome i à la densité électronique ρ_i en ce nœud de grille. Chaque processus léger met (ligne 6) alors sa contribution ρ_i en mémoire partagée. Chaque processus léger met (ligne 7) soit 0, soit ρ_i si l'atome appartient à la molécule A (on prépare ici le calcul de la fraction de densité ρ_A apportée par la molécule A). Les processus légers sont synchronisés (ligne 8) afin d'assurer que chaque processus léger a mis sa contribution en mémoire partagée. Une fois les contributions atomiques chargées en mémoire partagée, ces dernières peuvent être sommées (ligne 9) afin d'obtenir la densité électronique ρ_A de la molécule A et la densité électronique totale ρ en ce nœud par une opération de réduction réalisant simultanément les deux sommes correspondantes. Les processus légers sont synchronisés (ligne 10) pour attendre la fin du calcul de ρ_A et ρ. À ce stade (ligne 11) la densité électronique ρ_A de la molécule A et la densité électronique totale ρ du nœud (x, y, z) sont obtenues permettant le calcul en local du ratio $\frac{\rho_A}{\rho}$. La valeur de ce ratio pour ce nœud de grille détermine si l'on doit poursuivre les calculs, afin d'évaluer les autres quantités (gradient, Hessienne...). Si le critère de seuil est vérifié en ce point de grille ($0,05 < \frac{\rho_A}{\rho} < 0,95$) alors chaque processus léger calcule (ligne 12) successivement les contributions de ses atomes au gradient $\nabla\rho$ et à la matrice Hessienne H. Chaque contribution est stockée en mémoire partagée puis réduite (ligne 13) par une opération de réduction similaire à celle réalisée précédemment pour la densité $\rho(r)$. Ces étapes de stockage puis réduction sont effectuées deux quantités par deux quantités jusqu'à la dernière qui est réduite seule. D'abord les quantités $\frac{\partial\rho}{\partial x}$ et $\frac{\partial\rho}{\partial y}$ sont les premières calculées puis $\frac{\partial\rho}{\partial z}$ et $\frac{\partial^2\rho}{\partial x^2}$ ainsi de suite jusqu'à $\frac{\partial\rho}{\partial y\partial z}$. Les trois valeurs propres de la matrice Hessienne H sont calculées (ligne 15) par le premier processus léger en local. Puis la norme du gradient $\nabla\rho$ est calculée (ligne 16) en local par le premier processus léger. Le gradient réduit de la densité $s(\rho)$ est calculé (ligne 17) en local ainsi que la densité électronique signée à partir de la densité électronique ρ et du signe de la seconde valeur propre λ_2 de la Hessienne par le premier processus léger. Pour terminer le premier processus léger stocke en mémoire globale la densité électronique dans la variable cubeRho et le gradient réduit de la densité dans la variable cubeRDG, terminant ainsi le calcul pour le nœud (x, y, z).

En plus de la limitation provenant de la réduction citée plus haut, une seconde limitation apparaît due au nombre de processus légers maximum constituant un bloc qui est borné à 1024 sur les cartes graphiques K20X utilisées dans ce travail de recherche. Cette limitation est problématique car elle empêche l'étude de systèmes contenant plus de 1024 atomes. Cette première implémentation est naïve mais permet un premier pas vers le processus d'optimisation.

Algorithme 1 Version GPU V_{AT1}

FONCTION COMPUTENCI
    Calculer en mémoire partagée les coordonnées (x, y, z) du noeud traité par le thread 0 ;
    Synchronisation des threads ;
    Calculer la distance entre le nœud traité et l'atome lié au thread ;
    Calculer en registre la contribution ρ_i à la densité du nœud ;
    Mettre en mémoire partagée la contribution ρ_i à l'indice du thread ;
    Mettre en mémoire partagée la contribution ρ_i si l'atome appartient à la molécule A, 0 si l'atome appartient à la molécule B ;
    Synchronisation des threads ;
    Réductions de la mémoire partagée pour obtenir la densité ρ_A de la molécule A et la densité totale ρ du nœud traité ;
    Synchronisation des threads ;
    SI $0,05 < \frac{\rho_A}{\rho} < 0,95$ **ALORS**
        Calculer et mettre en mémoire partagée les trois contributions au gradient $\nabla\rho$ et les six contributions à la matrice Hessienne H du nœud (x, y, z).
        Réductions de la mémoire partagée pour obtenir le gradient $\nabla\rho$ et la matrice Hessienne H du nœud (x, y, z).
        SI le thread est le premier **ALORS**
            Calculer les trois valeurs propres de la Hessienne en les rangeant dans l'ordre croissant($\lambda_1 < \lambda_2 < \lambda_3$) ;
            Calculer la norme du gradient $\nabla\rho$
            Calculer $s(\rho)$ (RDG) et la densité signée ;
            Stocker les résultats en mémoire globale pour le nœud (x, y, z) dans les tableaux cubeRho et cubeRDG ;
        FIN SI
    FIN SI
FIN FONCTION

Cette limitation évoquée précédemment est résolue dans la seconde implémentation V_{AT2} basée sur V_{AT1} en fixant un nombre constant de processus légers. Dans V_{AT2} nous utilisons 128 processus légers par bloc. Ce nombre optimum est déterminé en utilisant conjointement deux méthodes. Tout d'abord l'utilisation de l'outil *occupancy calculator* [54], fourni par Nvidia, nous a permis d'obtenir trois valeurs d'intérêt pour le nombre de processus légers. Dans notre cas ces valeurs sont 64, 128 et 256 processus légers. Ensuite de manière empirique, en testant les trois cas et en comparant la rapidité d'exécution de chacun, nous avons déterminé que 128 processus légers est la valeur la plus optimale pour les cartes graphiques K20X que nous utilisons sur le problème traité.

Cette dernière partie empirique d'évaluation du nombre de 128 processus légers est illustré par le tableau 4.3. Dans ce tableau les temps d'exécution de l'implémentation V_{AT2} (avec 64, 128 et 256 processus légers) sont retranscrits pour six systèmes chimiques (L1-P1, L2-P2, L3-P3, L4-P4, L5-P5 et L6-P6) avec un pas de grille de 0,025Å. Dans ce tableau 4.3, nous pouvons voir que pour le système L1-P1, les temps d'exécution de V_{AT2} (4 s) pour les trois valeurs de nombre de processus légers sont similaires. Puis pour le système L2-P2 l'utilisation de 256 processus légers devient moins rapide (8 s) que pour 64 et 128 processus légers (6 s). Et enfin sur les systèmes L3-P3 à L6-P6, l'utilisation de 128 processus légers se démarque avec des temps significativement inférieurs aux deux autres nombres de processus légers (64 et 256).

Ce simple nombre de processus légers par bloc peut donc avoir un impact très significatif sur les temps de calcul. Dans le cas de V_{AT2} sur le système chimique L6-P6 nous pouvons constater un écart de 60s entre l'utilisation de 256 et 128 processus légers passant donc 218s à 158s, soit une facteur d'accélération de 1,38.

	L1-P1	L2-P2	L3-P3	L4-P4	L5-P5	L6-P6
VAT2 : 64 threads	4s	6s	14s	38s	70s	190s
VAT2 : 128 threads	**4s**	**6s**	**12s**	**32s**	**61s**	**158s**
VAT2 : 256 threads	4s	8s	20s	81s	86s	218s

Tableau 4.3 – Temps en secondes de la version GPU NCI V_{AT2} avec 64, 128 et 256 processus légers sur les six systèmes chimiques L1-P1, L2-P2, L3-P3, L4-P4, L5-P5 et L6-P6 pour un pas de grille de 0,025Å.

Deuxième implémentation de la version V_{AT} en fixant le nombre de processus légers à une valeur fixe en rapport au matériel utilisé : V_{AT2}

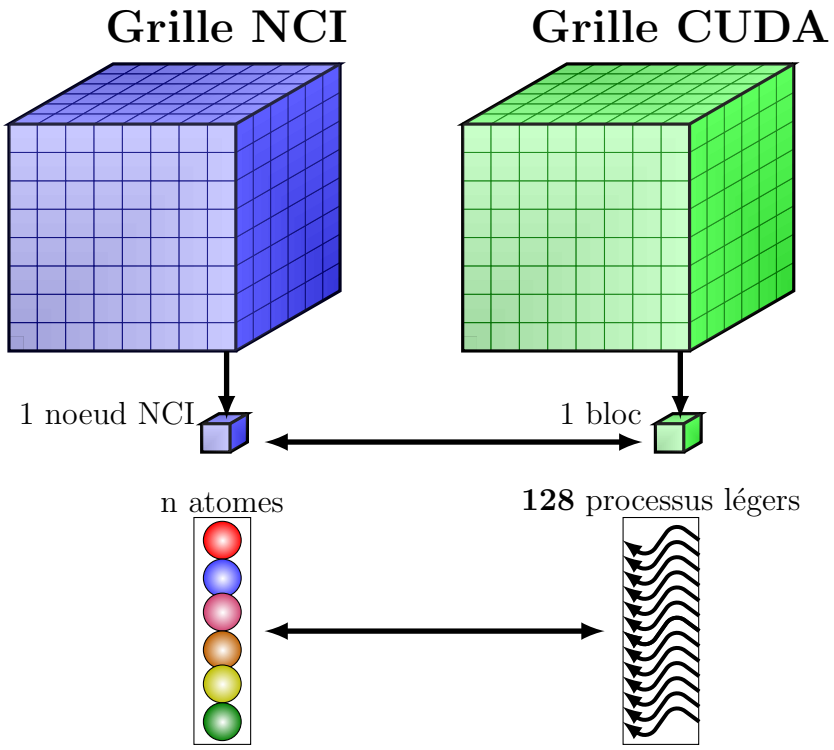


FIGURE 4.7 – Représentation de l'implémentation V_{AT2} d'un calcul NCI sur GPU

La figure 4.7 représente l'implémentation V_{AT2}. Chaque processus léger calcule alors les contributions ρ_i d'une partie des atomes du système étudié et non plus d'un seul atome comme précédemment dans la version V_{AT1}. Concrètement, le processus léger i s'occupe de tous les atomes d'indice égal à i modulo 128. Par exemple pour un système étudié de 256 atomes, le premier processus léger s'occupe des atomes d'indice 0, 128 et 256. Le processus léger 1 des atomes d'indices 1 et 129. le processus léger 2 des atomes d'indices 2 et 130. Ainsi de suite pour les autres processus légers du bloc.

Chaque processus léger somme en registres les différentes contributions ρ_i qu'il calcule, avant de mettre ses contributions partielles en mémoire partagée afin d'y réaliser une réduction équivalente à l'implémentation V_{AT1}.

L'algorithme 2 (reporté ci-dessous) résume le fonctionnement de l'implémentation V_{AT2}. Le premier processus léger calcule en mémoire partagée (ligne 2) les coordonnées du nœud (x, y, z), qui est traité par le bloc. Avant le début des autres calculs nécessitant la distance, les processus légers sont synchronisés (ligne 3) pour attendre le calcul précédent du premier processus léger. Chaque processus léger parcourt (ligne 4) les atomes qu'il doit traiter. Le processus léger calcule (ligne 5) en local la distance entre le nœud (x, y, z) traité et l'atome i en cours. À partir de cette distance, il calcule la contribution ρ_i de cet atome i et l'ajoute à la variable ρ_t (ligne 6) représentant la contribution calculée par le processus léger t. Le processus léger vérifie (ligne 7) à quelle molécule l'atome appartient. Si l'atome appartient à la molécule A, la contribution ρ_i est ajoutée (ligne 9) à la contribution ρ_t^A du processus léger courant. Cette procédure qui assigne à un processus léger plusieurs atomes à l'avantage d'utiliser les registres qui sont rapides d'accès. Toutes les contributions partielles ρ_t et ρ_t^A des différents processus légers sont stockées (ligne 11) en mémoire partagée. Les processus légers du bloc sont synchronisés (ligne 12) pour attendre que toutes les contributions soient en mémoire partagée. Une opération de réduction est alors réalisée (ligne 13) permettant d'obtenir les valeurs de la densité électronique totale ρ et de la densité électronique ρ_A de la molécule A : $\rho = \sum_{threads} \rho_t$, $\rho_A = \sum_{threads} \rho_t^A$. Les processus légers sont synchronisés (ligne 14) afin d'attendre les résultats de l'opération précédente. Le ratio $\frac{\rho_A}{\rho}$ permet (ligne 15) de déterminer s'il faut poursuivre. Chaque processus léger parcourt (ligne 16) alors à nouveau les atomes qu'il doit traiter. La distance (ligne 17) entre l'atome i et le nœud (x, y, z) est recalculée. Puis chaque processus léger calcule (ligne 18) en local sa contribution au gradient $\nabla\rho$ et à la matrice Hessienne H des atomes à traiter. Comme précédemment pour la densité, ces calculs se font en registre, donc bénéficient d'un accès rapide en mémoire. Chacun des processus légers met (ligne 20) en mémoire partagée ses contributions. Une synchronisation (ligne 21) est nécessaire pour la cohérence de la suite. Une fois la mémoire partagée chargée par tous les processus légers, des opérations de réductions (ligne 22) sont effectuées pour obtenir le gradient total $\nabla\rho$ et la matrice Hessienne H totale. Les dernières étapes sont réalisées (ligne 23) par le premier processus léger. Ce dernier calcule d'abord (ligne 24) les valeurs propres de la matrice Hessienne H, puis (ligne 25) la norme du gradient, ensuite (ligne 26) le gradient réduit de la densité $s(\rho)$ ainsi que la densité signée. Pour finir le premier processus léger stocke en mémoire globale $s(\rho)$ et la densité signée dans respectivement les tableaux *cubeRDG* et *cuRHO*. Ce qui termine le noyau.

Algorithme 2 Version GPU V_{AT2}

FONCTION COMPUTENCI
    Calculer en mémoire partagée les coordonnées (x, y, z) du nœud traité par le *thread* 0 ;
    Synchronisation des *threads* ;
    POUR tous les atomes numéros d'indice du *thread* + (0, 128, 256, ...) **FAIRE**
        Calculer la distance entre le nœud (x, y, z) et l'atome i en cours ;
        Ajouter la contribution ρ_i à la densité électronique ρ_t en registre pour le *thread* ;
        SI l'atome appartient à la molécule A **ALORS**
            Ajouter ρ_i au registre contenant ρ_t^A la densité de la molécule A pour le *thread* ;
        FIN SI
    FIN POUR
    Stocker les contributions ρ_t et ρ_t^A en mémoire partagée ;
    Synchronisation des *threads* ;
    Réductions pour obtenir la densité ρ_A de la molécule A et la densité totale ρ du nœud (x, y, z) traité ;
    Synchronisation des *threads* ;
    SI $0,05 < \frac{\rho_A}{\rho} < 0,95$ **ALORS**
        POUR sur les atomes numéros d'indice du *thread* + (1, 129, 257, ...) **FAIRE**
            Calculer la distance entre le nœud (x, y, z) et l'atome i en cours ;
            Calculer la contribution de l'atome i aux composantes du gradient $\nabla\rho$ et de la matrice Hessienne H dans les variables en registre du *thread* ;
        FIN POUR
        Chaque *thread* stocke en mémoire partagée ses contributions ;
        Synchronisation des *threads* ;
        Réductions pour obtenir $\nabla\rho$ et H ;
        SI le *thread* est le premier **ALORS**
            Calculer les valeurs propres ($\lambda_1 < \lambda_2 < \lambda_3$) de la matrice Hessienne H ;
            Calculer la norme du gradient $\nabla\rho$;
            Calculer le gradient réduit de la densité $s(\rho)$ et la densité signée ;
            Stocker en mémoire globale les résultats pour le nœud (x, y, z) ;
        FIN SI
    FIN SI
FIN FONCTION

Plusieurs points font perdre de l'efficacité aux implémentations V_{AT1} et V_{AT2}. Les coordonnées du nœud (x, y, z) étant communes aux processus légers, seul le premier processus léger réalise le calcul de ces coordonnées et les met en mémoire partagée. Lors de ce calcul, seul ce premier processus léger travaille, les autres sont inactifs. De même, le premier processus léger calcule les valeurs propres de la matrice Hessienne associée au nœud (x, y, z). Ces deux étapes perdent l'intérêt principal de l'utilisation des cartes graphiques, à savoir le parallélisme, en sérialisant les calculs sur le premier processus léger.

L'autre section perdant aussi en partie la puissance du parallélisme des cartes graphiques est l'utilisation de réductions pour sommer les différentes contributions, car les étapes finales d'une réduction sérialisent les calculs dans notre cas. On peut cependant noter que contrairement à la version V_{AT1}, V_{AT2} réalise une partie des sommes en registres (variable ρ_t et ρ_t^A), ce qui en théorie est plus efficace que dans la version V_{AT1}.

La version V_{NO} détaillée dans la partie suivante tente de supprimer cette utilisation

des réductions par une organisation différente du travail à effectuer. Cette seconde version rencontre elle aussi son lot de facteurs limitants qui sont détaillés dans la section suivante.

Un atome par noyau et une ligne de nœuds par bloc : version V_{NO}

L'idée de cette version GPU V_{NO} ("NO" pour nœuds) est de se dispenser des opérations de réduction en organisant différemment le calcul par rapport à V_{AT}. Contrairement à V_{AT}, le principe ici est d'appeler à la suite plusieurs noyaux : chaque noyau s'occupe de calculer les contributions d'un atome dans toute la grille de calcul. Dans un premier temps, sur CPU, une boucle sur les atomes est réalisée pour exécuter les noyaux. Chaque noyau s'occupe de traiter un atome, pour obtenir sa contribution à la densité électronique totale ainsi qu'à la densité électronique d'une des deux molécules en chaque point de la grille. Ces contributions atomiques sont stockées en mémoire globale (du GPU) et utilisées ultérieurement pour déterminer si les autres valeurs doivent être calculées. Chaque noyau calcule les contributions d'un atome en répartissant les blocs GPU sur les lignes de la grille de calcul. Les processus légers du bloc se répartissent alors les différentes contributions de la ligne à calculer. L'algorithme global de cette version V_{NO} suit ces étapes majeures :

- Lecture des fichiers d'entrée.
- Définition de la grille de calcul.
- Allouer et copier les données nécessaires aux calculs sur le/les GPU.
- Parcourir les atomes de la molécule A et exécuter séquentiellement sur GPU un noyau par atome pour calculer les contributions à densité électronique pour chaque nœud de la grille de calcul.
- Parcourir les atomes de la molécule B et exécuter sur GPU un noyau par atome pour calculer les contributions de densité électronique pour chaque nœud de la grille de calcul.
- Ajouter les densités électroniques des molécules A et B afin d'obtenir la densité électronique totale pour chaque nœud de la grille de calcul et conserver (en mémoire globale) la densité électronique de la molécule A.
- Exécution d'un dernier noyau permettant de calculer en chaque nœud de la grille : le gradient de la densité électronique, la matrice Hessienne, les trois valeurs propres H, le gradient réduit de la densité électronique et la densité électronique signée.
- Copier les résultats du GPU vers le CPU.
- Écriture des fichiers de sortie.

Première implémentation de la version V_{NO} en fixant le nombre de processus légers au nombre de nœuds d'une ligne de la grille de calcul : V_{NO1}

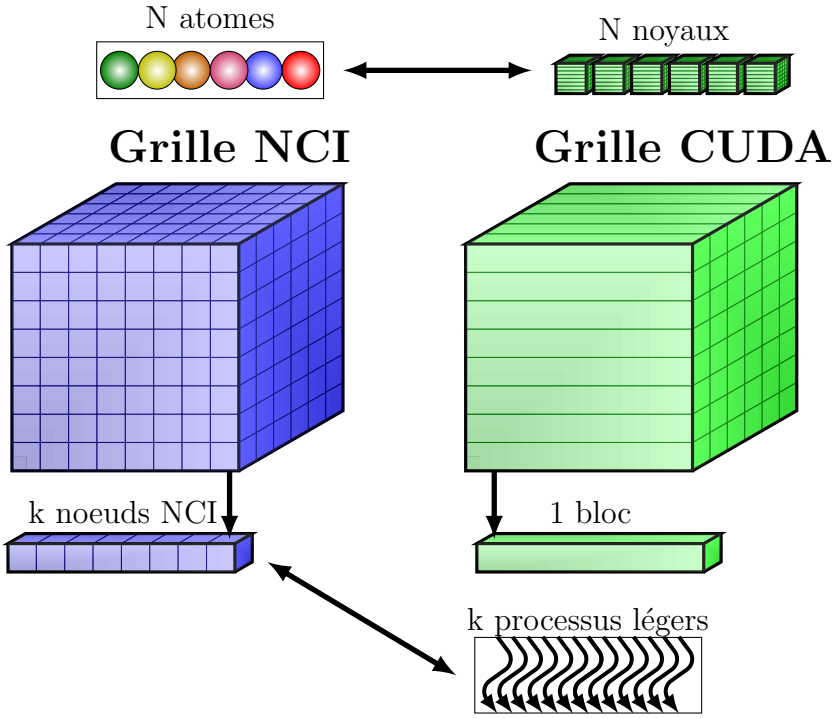


FIGURE 4.8 – Représentation de l'implémentation V_{NO1} d'un calcul NCI sur GPU.

La première implémentation de cette famille, V_{NO1} est illustrée par la figure 4.8. L'algorithme 3 ci-dessous résume le premier type de noyau exécuté par l'implémentation V_{NO1} pour calculer sur les nœuds de la grille la contribution à la densité électronique de l'atome en cours par le noyau. Chaque processus léger calcule (ligne 2 et 3) en local les coordonnées du nœud (x, y, z) qu'il traite à partir des indices du bloc et de l'indice du processus léger. La distance entre le nœud (x, y, z) et l'atome i du noyau peut (ligne 4) alors être calculée. Le processus léger se termine (ligne 5) en calculant la contribution à la densité électronique ρ_i à partir de la distance précédemment calculée et des paramètres a_{ij} et b_{ij} de l'atome de ce noyau. Enfin, la contribution ρ_i de l'atome en cours (noyau courant) au nœud courant est ajoutée à celle des autres calculées précédemment et stockée (ligne 6) dans le tableau *cubeRho* en mémoire globale sur GPU. Ce calcul en mémoire globale est plus long que s'il était réalisé en mémoire registre.

Algorithme 3 Version GPU V_{NO}

1: **FONCTION** COMPUTEPARTRHO
2: Calculer les coordonnées (y, z) communes aux nœuds du bloc (processus léger 0)
3: Calculer la coordonnée x restante du nœud (x, y, z) ;
4: Calculer la distance entre le nœud(x, y, z) et l'atome i du noyau ;
5: Calculer en registre la contribution ρ_i ;
6: Ajouter ρ_i à la densité totale stockée en mémoire globale du GPU (`cubeRho`) ;
7: **FIN FONCTION**

L'algorithme 4 ci-dessous résume le dernier noyau de l'implémentation V_{NO1}. L'organisation CUDA de ce dernier noyau est calquée sur celle du noyau V_{AT1} (algorithme 1 lignes

11 à 18) : nous revenons à un atome qui est traité par un processus léger. Ce dernier noyau n'est pas l'étape limitante car il ne s'applique qu'aux nœuds de la grille qui respectent le critère du ratio de densité $\frac{\rho_A}{\rho}$. Le coût du calcul est porté principalement par l'algorithme 3.

Deux implémentations ont été réalisées pour l'algorithme principal 3 : la première, V_{NO1} utilise autant de processus légers que de nœuds d'une ligne de la grille subissant une limitation due au nombre maximum de 1024 processus légers pour les blocs avec la carte graphique K20X. La seconde V_{NO2} fixe cette limite en rendant constant le nombre de processus légers.

Algorithme 4 Implémentation GPU V_{NO1}

1: **FONCTION** computeRDG
2: **SI** $0.05 < \frac{\rho_A}{\rho} < 0.95$ **ALORS**
3: Calculer en mémoire partagée les coordonnées (x, y, z) du nœud ;
4: Calculer et mettre en mémoire partagée les contributions des atomes aux composantes du gradient $\nabla\rho$ et de la matrice Hessienne H du nœud (x, y, z) ;
5: Synchronisation des processus légers ;
6: Réductions ;
7: **SI** le processus léger est le premier **ALORS**
8: Calculer les valeurs propres en les rangeant dans l'ordre croissant$(\lambda_1 < \lambda_2 < \lambda_3)$;
9: Calculer la norme du gradient $\nabla\rho$
10: Calculer $S(\rho)$ (RDG) et la densité signée ;
11: Stocker les résultats en mémoire globale pour le nœud (x, y, z) dans les tableaux `cubeRho` et `cubeRDG` ;
12: **FIN SI**
13: **FIN SI**
14: **FIN FONCTION**

Deuxième implémentation de la version V_{NO} en fixant le nombre de processus légers à un nombre fixe dépendant du matériel : V_{NO2}

Une réponse à ce problème de limite du nombre de processus légers est de traiter plusieurs nœuds avec un même processus léger. Cette légère variante (algorithme 5) est implémentée dans V_{NO2}. Le nombre de processus légers est alors fixé à 128. Ce nombre de processus légers a été obtenu en utilisant l'outil *occupancy calculator*[54] qui nous a indiqué deux valeurs possibles optimums (64 et 128 processus légers) que nous avons testées et comparées.

Le tableau 4.4 illustre ce propos avec les temps d'exécution de V_{NO2} avec 64 et 128 processus légers sur six systèmes chimiques (L1-P1, L2-P2, L3-P3, L4-P4, L5-P5 et L6-P6) avec un pas de grille de 0,025Å. Notons immédiatement que la version V_{NO} (indépendamment de l'implémentation) est incapable de traiter le système L6-P6 avec un pas de 0,025Å. En effet, la version V_{NO} nécessite le stockage en mémoire globale du GPU d'une variable supplémentaire (densité électronique d'une des molécules du système étudié) faisant que la quantité de mémoire globale disponible est insuffisante (oom : *Out Of Memory*) sur une carte graphique K20X.

	L1-P1	L2-P2	L3-P3	L4-P4	L5-P5	L6-P6
VNO2 : 64 threads	4s	6s	12s	36s	66s	oom
VNO2 : 128 threads	**4s**	**6s**	**13s**	**34s**	**63s**	**oom**

Tableau 4.4 – Temps en secondes de V_{NO2} avec 64 et 128 processus légers sur les six systèmes chimiques L1-P1, L2-P2, L3-P3, L4-P4, L5-P5 et L6-P6 pour un pas de grille de 0,025Å ; oom : *Out Of Memory*, mémoire insuffisante.

Le choix de 128 processus légers est ici un peu plus délicat car pour les systèmes L1-P1 et L2-P2 les temps d'exécutions (4 s et 6 s) sont similaires pour 64 et 128 processus légers. Puis pour le système L3-P3, l'utilisation de 64 processus légers est légèrement plus rapide (12 s) que l'utilisation de 128 processus légers (13 s). Et enfin, et c'est avec ces données que nous avons choisi 128 processus légers, avec les systèmes L4-P4 et L5-P5 l'utilisation de 128 processus légers par bloc est plus rapide de 2s et 3s respectivement. Pour cette seconde implémentation V_{NO2} l'impact du nombre de processus légers n'est pas capital.

La figure 4.9 représente le paradigme de cette implémentation V_{NO2}.

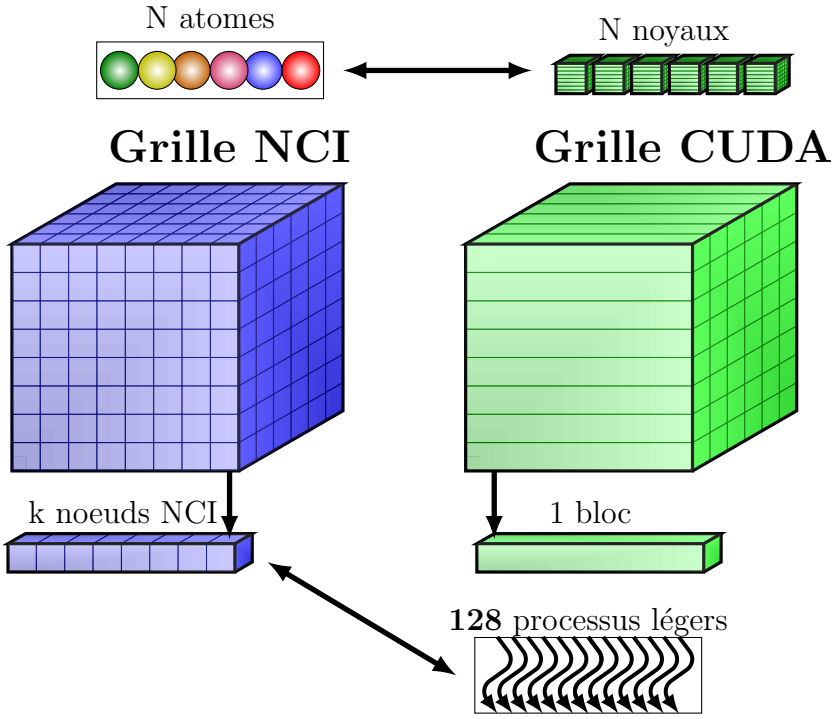


FIGURE 4.9 – Représentation de l'implémentation V_{NO} d'un calcul NCI sur GPU.

Chaque noyau (pour son atome) procède alors comme décrit dans l'algorithme 5. Le premier processus léger calcule (ligne 2) en mémoire partagée les coordonnées (y, z) communes aux nœuds du bloc. Chaque processus léger parcourt (ligne 3) ensuite l'ensemble des nœuds qu'il doit traiter. Pour chaque nœud traité, le processus léger calcule (ligne 4) la coordonnée x manquante du nœud (x, y, z). La distance (ligne 5) entre le nœud (x, y, z) et l'atome i du noyau (en cours) est calculée. Et enfin, la densité électronique ρ_i est calculée et ajoutée (ligne 6) à la mémoire globale.

Algorithme 5 Implémentation GPU V_{NO2}

1: **FONCTION** computePartRho
2: Calculer les coordonnées (y, z) communes aux nœuds du bloc (*thread* 0)
3: **POUR** chacun des nœuds d'indices *IDThread*+(0,128,256,...) **FAIRE**
4: Calculer la coordonnée x restante du nœud (x, y, z) ;
5: Calculer la distance entre le nœud (x, y, z) et l'atome i du noyau ;
6: Calculer et ajouter ρ_i à la densité totale stockée en mémoire globale (`cubeRho`) ;
7: **FIN POUR**
8: **FIN FONCTION**

Le dernier noyau exécuté est détaillé dans l'algorithme 10 qui est reporté en annexe G.

L'idée de V_{NO} est de se dispenser des opérations de réduction en organisant différemment le calcul par rapport à V_{AT}, le soucis étant alors que chaque noyau fait autant d'addition en mémoire globale qu'il y a de contributions de calculées et la mémoire globale du GPU est peu efficace en terme d'accès. De plus, cette version consomme plus de mémoire globale car la densité électronique de la molécule A ρ_A, est elle aussi stockée en mémoire globale avant d'être utilisée dans la suite de l'algorithme.

La version suivante V_{HY} ("HY" pour hybride) règle les problèmes des deux versions (V_{AT} et V_{NO}) et réalise les meilleures performances.

Combinaison des atouts des approches précédentes : version V_{HY}

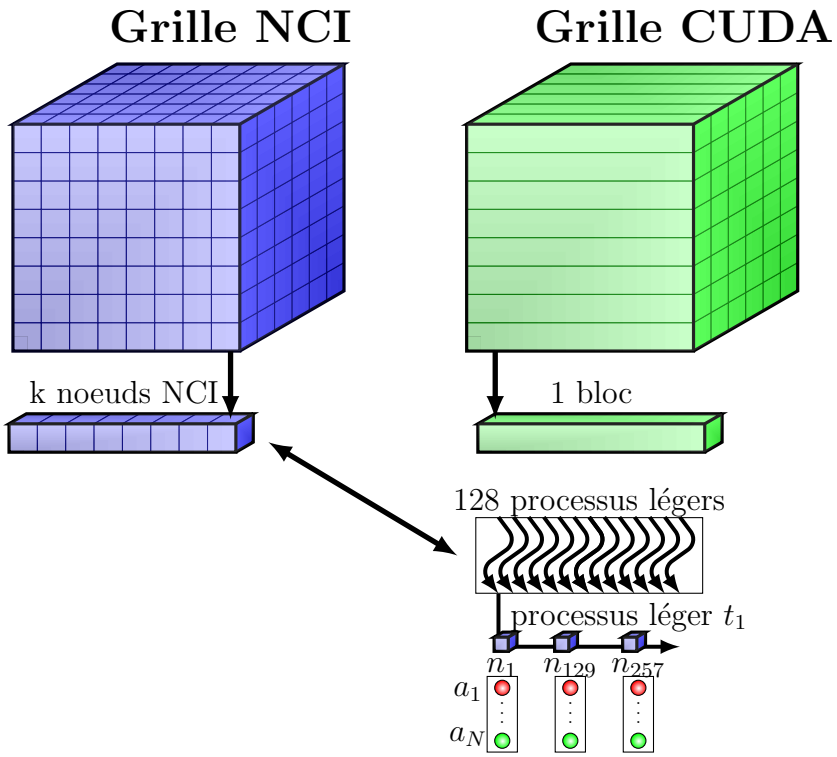


FIGURE 4.10 – Représentation de l'implémentation V_{HY}

Cette troisième approche, hybride, V_{HY} permet de résoudre les problèmes des deux versions (V_{AT} et V_{NO}) précédentes. Concrètement, un unique noyau est appelé réalisant

l'entièreté des calculs de la grille. La figure 4.10 représente le paradigme de cette version. Comme pour la version V_{NO} chaque bloc GPU s'occupe d'une ligne de la grille NCI ; comme pour les implémentations V_{AT2} et V_{NO2}, chaque bloc est constitué d'un nombre fixe de processus légers, pour tout le système traité, ici de 128 processus légers. La différence (avec V_{AT2} et V_{NO2}) est que chaque processus léger s'occupe de calculer toutes les contributions des atomes pour chaque nœud de la grille de calcul, que le processus léger traite. Un bloc GPU traite une ligne de la grille de calcul. Le processus léger d'indice i de ce bloc, s'occupe des nœuds d'indice égal à i modulo 128 de la ligne traitée. Par exemple, le premier processus léger s'occupe des nœuds d'indices 0,128,256... Le processus léger 1 des nœuds d'indices 1, 129, 257... Ainsi de suite. Pour chacun de ses nœuds le processus léger parcourt tous les atomes afin d'obtenir les quantités cherchées : ρ, $\nabla\rho$, H et λ_2.

L'algorithme 6 ci-dessous résume les instructions réalisées par le noyau de la version V_{HY}. Le processus léger t_i (ligne 2) calcule, en local, les coordonnées (y, z) de la ligne calculée par le bloc en cours, en utilisant les indices (L, M) du bloc. Le processus léger (ligne 3) parcourt ensuite les nœuds qu'il doit calculer. Pour chaque nœud, le processus léger calcule en local (ligne 4) la dernière coordonnée x du nœud (x, y, z) en cours en utilisant l'indice du processus léger. Une première boucle intérieure (ligne 5) parcourt les N_A atomes de la molécule A. Pour chaque atome (ligne 6), la distance entre le nœud (x, y, z) et l'atome courant i est calculée en local. La contribution atomique (ligne 7) ρ_i peut alors être calculée en local à partir de la distance et des paramètres a_{ij} et b_{ij} stockés en mémoire constante. ρ_i est sommée en local (ligne 8) afin de tirer profit de la rapidité d'accès à ρ, la variable qui contiendra la densité électronique totale du nœud (x, y, z). La densité de cette première molécule A est aussi stockée (ligne 10) dans la variable ρ_A en registre. Une fois ρ_A stockée, un traitement analogue aux lignes 5 à 9 est effectué pour la molécule B. À ce stade (ligne 16) la densité électronique ρ_A de la molécule A et la densité électronique totale ρ du nœud (x, y, z) sont connues permettant le calcul en local du ratio $\frac{\rho_A}{\rho}$ déterminant s'il faut poursuivre les calculs pour ce nœud. Si oui, les atomes du système sont parcourus (ligne 17). Pour chaque atome (ligne 18), la distance entre le nœud (x, y, z) et l'atome courant i est recalculée en local. Les contributions (ligne 19) de chaque atome aux trois composantes du gradient $\nabla\rho$ et six composantes de la matrice Hessienne H peuvent être calculées en local à partir de la distance et des paramètres a_{ij} et b_{ij}. Une fois la matrice H obtenue, le processus léger calcule (ligne 21) les trois valeurs propres de cette dernière par la méthode de Cardan. La norme du gradient (ligne 22) est ensuite calculée en local. Puis (ligne 23) le gradient réduit de la densité $s(\rho)$ est calculé ainsi que la densité électronique signée. Il est important de noter que les opérations manipulent des valeurs stockées en registres, bénéficiant ainsi d'un accès rapide. Pour terminer le processus léger stocke en mémoire globale la densité électronique dans le tableau *cubeRho* et le gradient réduit de la densité dans le tableau *cubeRDG*, terminant ainsi le calcul pour le nœud (x, y, z) et pouvant passer au suivant.

En procédant ainsi, les opérations de réduction de la version V_{AT} sont évitées car chaque nœud de la grille NCI est traité par un unique processus léger. De plus, contrairement à la version V_{NO} les opérations sont effectuées de manière efficace en registre et un seul accès est nécessaire en mémoire globale, par propriété à calculer ($s(r)$ et la densité signée), pour obtenir les résultats. Rappelons aussi que dans la version V_{HY} (contrairement à V_{AT}) les processus légers calculent tous leurs coordonnées et leurs valeurs propres impliquant une utilisation efficiente du parallélisme des GPU.

La version V_{HY} apparaît à ce stade comme étant la plus efficiente, en théorie. Ceci est vérifié d'après les résultats qui sont présentés dans la suite.

Algorithme 6 Version V_{HY}

1: **FONCTION** COMPUTENCI
2: Calculer les coordonnées (y, z) communes aux k nœuds du bloc ;
3: **POUR** chacun des nœuds de numéros *IDThread*+(0, 128, 256, . . .) **FAIRE**
4: Calculer la coordonnée x restante du nœud k et le stocker dans une variable locale ;
5: **POUR** sur les N_A atomes de la molécule A **FAIRE**
6: Calculer la distance entre le nœud (x, y, z) et l'atome i traité ;
7: Calculer ρ_i la contribution de l'atome i au nœud (x, y, z) ;
8: Ajouter ρ_i au registre contenant ρ la valeur du nœud (x, y, z) ;
9: **FIN POUR**
10: Copier ρ dans le registre ρ_A contenant la contribution de la molécule A au calcul du nœud (x, y, z) ;
11: **POUR** chacun des N_B atomes de la molécule B **FAIRE**
12: Calculer la distance entre le nœud (x, y, z) et l'atome i traité ;
13: Calculer ρ_i la contribution de l'atome i au nœud (x, y, z) ;
14: Ajouter ρ_i au registre contenant ρ la valeur du nœud (x, y, z) ;
15: **FIN POUR**
16: **SI** $0,05 < \frac{\rho_A}{\rho} < 0,95$ **ALORS**
17: **POUR** tous les atomes($N_A + N_B$) **FAIRE**
18: Calculer la distance entre le nœud (x, y, z) et l'atome i traité ;
19: Calculer les trois contributions au gradient $\nabla\rho$ et les six contributions aux composantes de la matrice Hessienne H et les ajouter aux (3+6) registres contenant le total du nœud (x, y, z) ;
20: **FIN POUR**
21: Calculer les valeurs propres en les rangeant dans l'ordre croissant ($\lambda_1 < \lambda_2 < \lambda_3$) ;
22: Calculer la norme du gradient $\nabla\rho$
23: Calculer $s(\rho)$ (`RDG`) et la densité signée ;
24: Stocker les résultats en mémoire globale pour le nœud (x, y, z) dans les tableaux `cubeRho` et `cubeRDG` ;
25: **FIN SI**
26: **FIN POUR**
27: **FIN FONCTION**

4.8 Résultats

Dans cette section nous nous intéressons à évaluer les différents portages GPU précédemment détaillés. Les systèmes chimiques utilisés pour tester les performances de nos portages GPU ont été détaillés dans la section 4.3. L'ensemble des temps d'exécution des différentes implémentations est disponible dans les tableaux de l'annexe A.

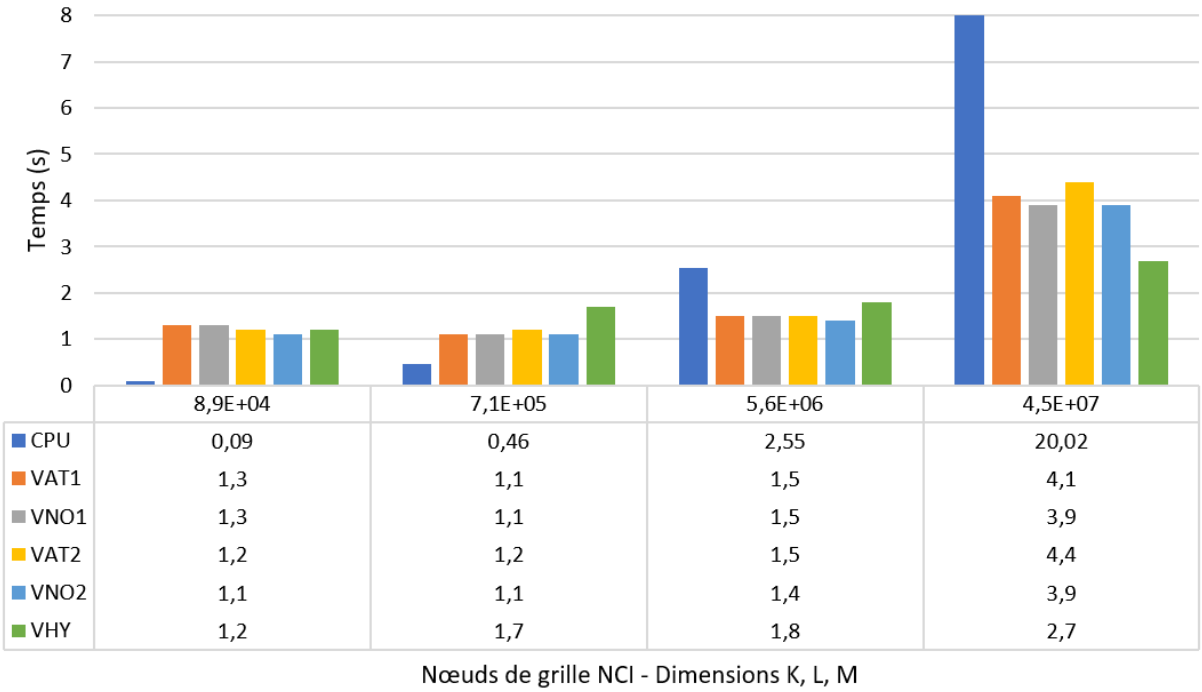
Analyse des temps d'exécutions des portages GPU sur une carte graphique K20X

Dans cette section nous évaluons les performances des cinq portages GPU précédemment décrits : V_{NO1}, V_{NO2}, V_{AT1}, V_{AT2} et V_{HY}. Cette évaluation est réalisée sur les 36 complexes précédemment détaillés avec quatre pas possibles (0,2 Å, 0,1 Å, 0,05 Ået 0,025 Å) de grille

de calcul ; cela représente pour chaque version un jeu de 144 (36×4) calculs à réaliser. La variété des systèmes nous permet de rencontrer de nombreux cas de figures permettant d'évaluer au mieux les performances. Les temps CPU sont obtenus en utilisant notre installation de référence (le code C compilé avec le compilateur d'Intel et les options définies précédemment), ainsi que les 16 cœurs CPU (Ivy Bridge 2,6GHz) disponibles sur un nœud du centre de calcul ROMEO. Les performances des implémentations GPU sont quant à elles obtenues en utilisant une à deux cartes graphiques K20X disponibles sur ce même nœud (du centre de calcul ROMEO).

Nous commençons par détailler les cas défavorables aux portages GPU. La figure 4.11 représente les temps de calcul sur le système L1-P1 avec quatre pas de grille différents (0,2Å, 0,1Å, 0,05Ået 0,025Å), de notre référence CPU (en C) et des cinq portages GPU décrits précédemment (V_{AT1}, V_{AT2}, V_{NO1}, V_{NO2} et V_{HY}).

La figure 4.11 illustre la nécessité d'avoir un calcul comportant suffisamment d'opérations pour tirer profit de la puissance des cartes graphiques. En effet, la figure 4.11 permet de montrer que sur le système L1-P1, les portages GPU ne deviennent plus rapide que la version CPU qu'à partir d'un pas de grille de 0,05 Å. Pour les pas de grille supérieurs (0,2Å et 0,1Å) à 0,05Å le temps d'exécution de notre référence CPU est inférieur aux temps d'exécution des cinq portages GPU. Pour un pas de 0,025 Å, tous les portages GPU (4 s environ pour les version V_{AT1}, V_{AT2}, V_{NO1}, V_{NO2} et 3s pour V_{HY}) s'exécutent significativement plus rapidement que notre référence CPU (20 s). En plus d'une exécution globalement plus rapide des portages GPU, nous pouvons aussi commencer à voir une tendance sur les performances entre les différents portages GPU. Le portage V_{HY} semble, à ce stade, être le plus rapide de tous les portages GPU lorsque la quantité de données à calculer est suffisamment importante (dernière colonne).



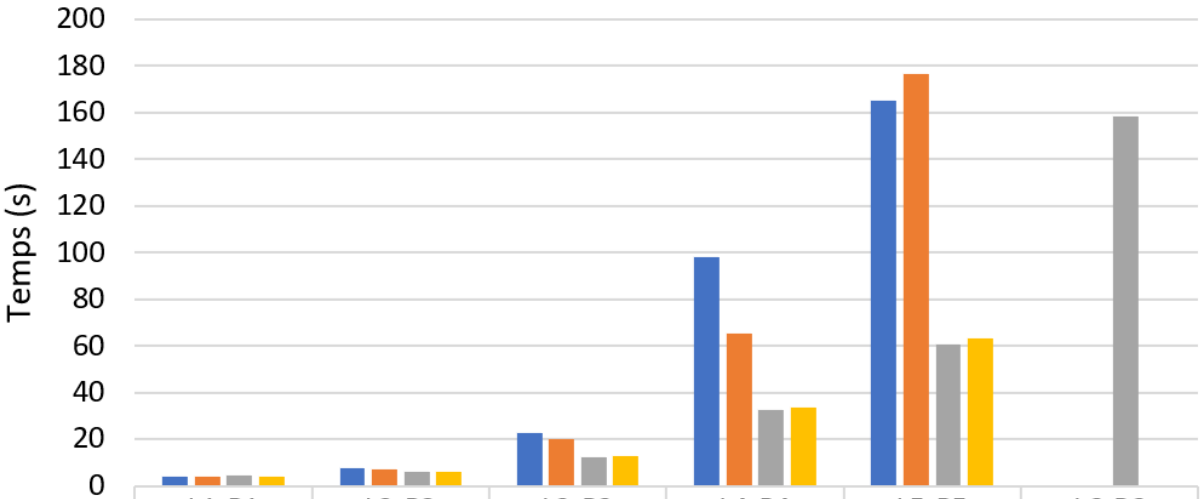
	8,9E+04	7,1E+05	5,6E+06	4,5E+07
CPU	0,09	0,46	2,55	20,02
VAT1	1,3	1,1	1,5	4,1
VNO1	1,3	1,1	1,5	3,9
VAT2	1,2	1,2	1,5	4,4
VNO2	1,1	1,1	1,4	3,9
VHY	1,2	1,7	1,8	2,7

FIGURE 4.11 – Temps d'exécution de la référence CPU et des cinq portages GPU (V_{AT1}, V_{AT2}, V_{NO1}, V_{NO2} et V_{HY}) de calculs NCI sur le petit complexe L1-P1 avec quatre pas de grilles (0,2Å, 0,1Å, 0,05Å et 0,025Å) ; le nombre de points constituant la grille est indiqué en tête de chaque colonne.

Considérons maintenant uniquement le pas de grille le plus fin (0,025Å) impliquant des grilles à traiter beaucoup plus denses en nœuds (figure 4.12). De manière spécifique, les portages V_{NO1} et V_{NO2} sont incapables, avec une carte graphique K20X, de traiter les cas utilisant un pas de grille de 0,025Å et le ligand L6 ("oom" indiqué en dernière colonne). La quantité de mémoire globale requise par ces instances est trop importante par rapport à celle disponible sur une carte graphique K20X. Ce problème peut être réglé en modifiant le code V_{NO} pour effectuer le calcul de la grille en plusieurs étapes.

L'implémentation V_{AT1} est quant à elle limitée par les 1024 processus légers par bloc disponibles sur les cartes graphiques K20X, ce qui limite le nombre d'atomes qui peut être traité : les complexes composés de la protéine P6 (1 245 atomes) sont intraitables car composés de plus de 1024 atomes.

Ces limitations des algorithmes V_{AT1}, V_{NO1} et V_{NO2} montrent une erreur de conception qui doit être évitée. Une implémentation doit être réalisée en prenant en compte l'architecture utilisée ainsi que ses limitations. De ce fait, calquer le problème physique résolu sur les dimensions de la carte graphique amène à limiter la taille des problèmes envisageables. Une telle limitation doit être contournée ce qui amène l'amélioration des versions V_{AT1} et V_{NO1}, en respectivement V_{AT2} et V_{NO2}, en fixant les quantités limitantes dans chacune des implémentations.

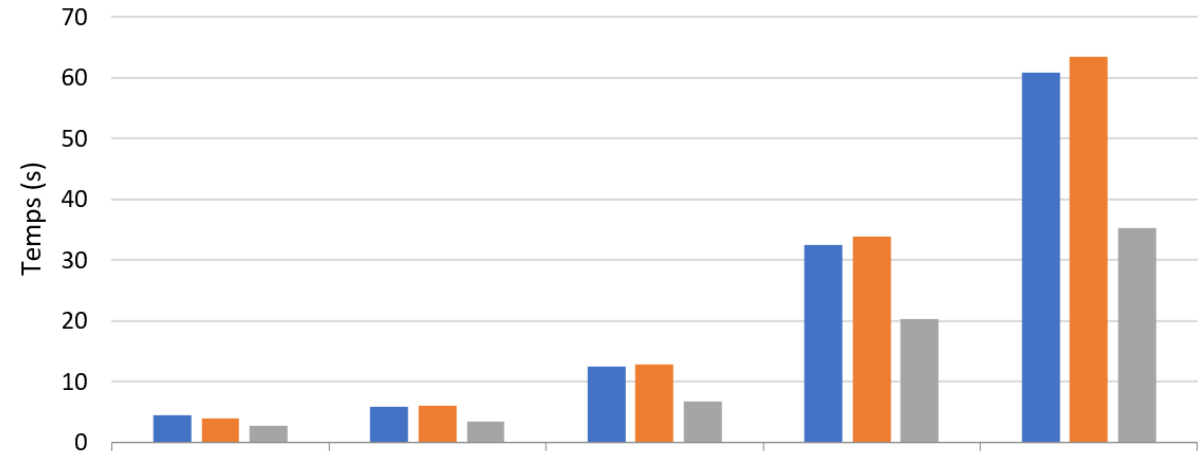


	L1-P1	L2-P2	L3-P3	L4-P4	L5-P5	L6-P6
VAT1	4	8	23	98	165	oot
VNO1	4	7	20	65	177	oom
VAT2	4	6	12	32	61	158
VNO2	4	6	13	34	63	oom

FIGURE 4.12 – Temps en secondes des versions GPU V_{AT1}, V_{NO1}, V_{AT2} (128 processus légers) et V_{NO2} (128 processus légers) d'un calcul NCI sur six complexes L1-P1, L2-P2, L3-P3, L4-P4, L5-P5 et L6-P6 avec un pas de 0,025Å ; oom : *Out Of Memory*, mémoire insuffisante ; oot : *Out Of Thread*, nombre de processus légers insuffisant.

Comme la figure 4.12 le montre, fixer le nombre de processus légers (de manière adéquat) permet d'améliorer significativement les performances par rapport aux implémentations V_{AT1} et V_{NO1}. Pour des systèmes plus grands que 722 atomes (L3-P3), l'inconvénient des versions V_{AT} (ralentissements par l'utilisation de réductions) est contrebalancé par le calcul rapide en registre des différentes quantités lorsque le nombre de processus légers est fixé par bloc (V_{AT2}). Cette amélioration provient d'une meilleure occupation des SMX de la carte graphique par des *warps* complètement utilisés.

Lorsque l'on compare entre elles les implémentations V_{AT2} et V_{NO2}, des temps similaires sont observables figure 4.13. Le traitement des atomes au niveau des processus légers (V_{AT2}) est plus efficace uniquement de 10 à 15% au delà du premier complexe L1-P1. V_{AT2} réalise une performance légèrement supérieure à V_{NO2} qui utilise plus fréquemment la mémoire globale pour réaliser ses calculs. De plus, l'implémentation V_{AT2} peut traiter le système L6-P6 avec un pas de 0,025Å (non reportée sur la figure 4.13).



	L1-P1	L2-P2	L3-P3	L4-P4	L5-P5
VAT2	4,4	5,9	12,4	32,4	60,9
VNO2	3,9	6,1	12,8	33,8	63,4
VHY	2,7	3,4	6,7	20,3	35,3

FIGURE 4.13 – Temps en secondes des versions GPU V_{AT2}, V_{NO2} et V_{HY} d'un calcul NCI avec 128 processus légers sur les complexes L1-P1, L2-P2, L3-P3, L4-P4 et L5-P5 avec un pas de 0,025Å

La meilleure performance est obtenue avec l'implémentation V_{HY} qui apparaît 30 à 50% plus performante que V_{AT2} comme le montre la figure 4.13. Cette performance est obtenue par une meilleure utilisation de la carte graphique en concentrant les calculs dans les registres et en tirant le plus possible profit du parallélisme fourni par les cartes graphiques, comme expliqué dans la section 4.7.

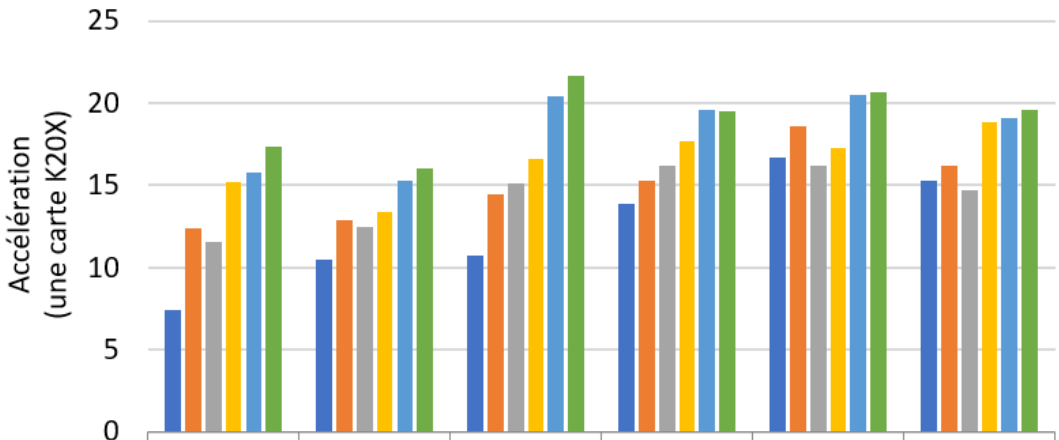
Accélérations avec une carte graphique K20X

Nous savons donc que la version V_{HY} est la plus rapide par rapport aux autres portages (V_{AT1}, V_{AT2}, V_{NO1} et V_{NO2}). Regardons maintenant l'accélération obtenue sur différents systèmes, par rapport à notre installation parallèle de référence (CPU).

36 systèmes chimiques sont utilisés dans la figure 4.14 avec un pas de grille (0,025Å) pour évaluer l'accélération avec une carte graphique K20X de la version V_{HY}. Les temps des portages GPU sont obtenus avec la carte graphique K20X décrite section 2.6. Les temps d'exécution parallèle (CPU) sont obtenus à l'aide de deux processeurs Intel Ivy Bridge (2,6 GHz) composés chacun de huit cœurs. L'ensemble des temps d'exécution est disponible annexe A.

Nous pouvons constater à l'aide de la figure 4.14 que pour un GPU (avec des grilles de calcul suffisamment denses) le facteur d'accélération va de 7 à 22. Plus le système étudié est grand, plus le facteur d'accélération est grand, jusqu'aux systèmes contenant les ligands L3 à L6 et les protéines P5 ou P6, où le facteur d'accélération est compris entre 19 et 22. Le meilleur facteur d'accélération (22) est atteint avec le système L3-P6.

L'approche NCI tire donc profit de la puissance du parallélisme *manycore* apportée par la carte graphique K20X.



	L1	L2	L3	L4	L5	L6
P1	7	10	11	14	17	15
P2	12	13	14	15	19	16
P3	12	13	15	16	16	15
P4	15	13	17	18	17	19
P5	16	15	20	20	21	19
P6	17	16	22	20	21	20

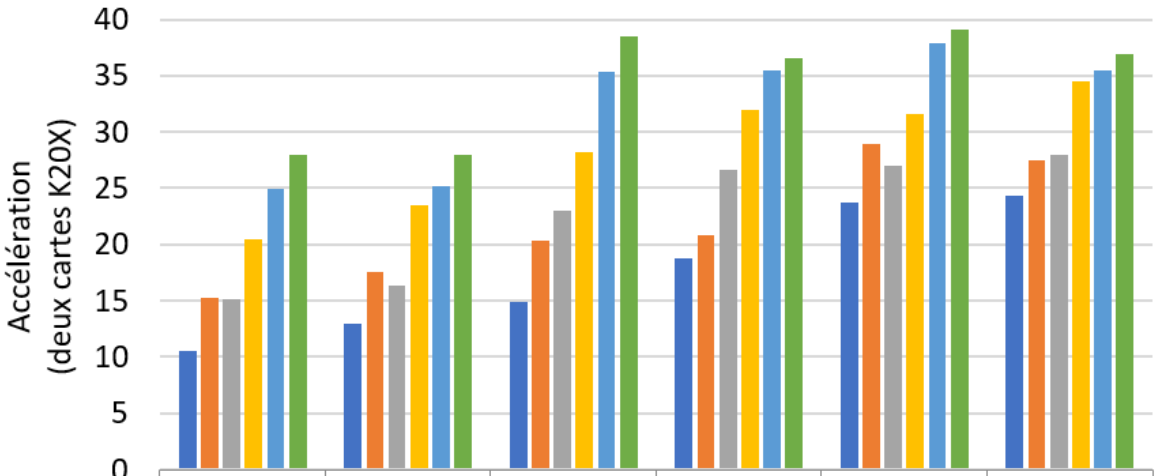
FIGURE 4.14 – Facteurs d'accélérations de la version GPU V_{HY} d'un calcul NCI avec une carte graphique K20X sur les 36 complexes avec un pas de 0,025Å en rapport à la référence CPU ; les complexes sont formés par l'interaction des six ligands L1-L6 avec six modèles de protéines P1-P6.

Accélérations avec deux cartes graphiques K20X

Chaque nœud du centre de calcul ROMEO contient deux cartes graphiques K20X. Nous avons donc écrit des variantes des codes V_{AT2}, V_{NO2} et V_{HY} capables de tirer profit de deux cartes graphiques. Ces variantes répartissent le calcul en découpant la grille de calcul NCI en deux, afin que chaque carte graphique puisse réaliser une partie des calculs.

Pour toutes les instances prenant plus de 10 secondes, l'utilisation de deux cartes graphiques K20X permet d'obtenir une accélération significative de 40 à 49%, pour toutes les versions en les comparant à leur version utilisant un seul GPU. Les meilleures performances sont obtenues avec V_{HY} utilisant deux GPU. Comme la figure 4.15 permet de le constater, l'approche NCI peut tirer significativement profit de la puissance apportée par le parallélisme des cartes graphiques octroyant un facteur d'accélération allant de 11 à 39 dans ces exemples. Le facteur d'accélération obtenu dépend grandement du système chimique étudié, et ce facteur se stabilise autour de 35 pour de grands systèmes chimiques et de grandes grilles. Nous constatons que le facteur d'accélération cesse d'augmenter après des systèmes de 1341 atomes (L3-P6) et une grille de 10^8 nœuds.

Il est donc possible de tirer profit de l'utilisation de plusieurs cartes graphiques pour l'approche NCI, même pour des systèmes de très grande taille.



	L1	L2	L3	L4	L5	L6
P1	11	13	15	19	24	24
P2	15	18	20	21	29	27
P3	15	16	23	27	27	28
P4	20	23	28	32	32	35
P5	25	25	35	35	38	35
P6	28	28	39	37	39	37

FIGURE 4.15 – Facteurs d'accélérations de la version GPU V_{HY} pour un calcul NCI avec deux cartes graphiques K20X sur les 36 complexes avec un pas de 0,025Å en rapport à la référence CPU

Analyse comparative de l'utilisation de réels flottants simple ou double précision

Les versions GPU présentées jusqu'ici ont été écrites en utilisant des nombres à virgule flottante simple précision. En effet, l'utilisation de ces nombres plutôt que des nombres à double précision est motivée par la recherche de performances optimales lors de l'utilisation ultérieure de GPU. Il faut savoir que les cartes graphiques K20X dont nous disposions possèdent 2688 cœurs simple précision contre seulement 896 cœurs double précision (comme précédemment décrit section 2.6). L'utilisation d'opérations simple précision apparaît comme plus optimisée lorsque la puissance de calcul est recherchée, à condition que les résultats puissent être obtenus avec une précision satisfaisante d'un point de vue scientifique. L'implémentation V_{HY} a aussi été modifiée pour réaliser les calculs en double précision.

Comme nous pouvons tout d'abord le constater dans la figure 4.16, le problème de la quantité de mémoire globale est de nouveau présent lors de l'utilisation de la double précision (nécessitant deux fois plus de mémoire) pour le traitement du grand système L6-P5 avec un GPU ("oom" en dernière colonne). L'utilisation de deux GPU répartissant les données traitées entre les deux cartes graphiques permet de contourner immédiatement ce problème ($V_{HY}(DP, 2GPU)$, 66,6 s). Notons qu'une telle découpe est généralisable (automatisable) pour faire passer un problème qui serait de trop grande taille pour tenir sur un GPU.

Le tableau 4.5 illustre la différence entre les résultats obtenus par un calcul GPU

simple précision et un calcul CPU du code NCIplot double précision. Pour mesurer cette différence nous avons calculé deux erreurs quadratiques moyennes (RMSD : *Root-Mean-Square Deviation*). Le premier RMSD porte sur la densité électronique ρ. Pour chaque point de la grille de calcul NCI, la différence entre ρ calculé en simple précision et double précision est élevée au carré. Puis la moyenne est faite sur tous les nœuds de la grille, permettant d'obtenir la première erreur quadratique moyenne (sur ρ). De la même manière, l'erreur quadratique moyenne est calculée sur le gradient réduit $s(\rho)$. De plus, deux sommes P sont calculées, pour simple et double précisions, telles que :

$$P = \sum_{\Omega_{NCI}} \rho^{\frac{4}{3}} \tag{4.2}$$

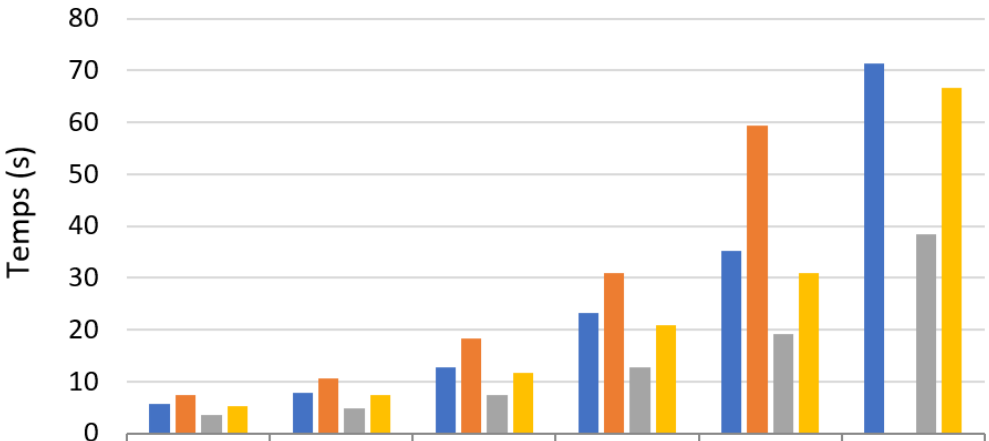
avec Ω_{NCI} l'ensemble des points d'interaction entre le ligand et la protéine et ρ la densité électronique calculée.

La conclusion que nous en avons tirée est que l'erreur relative étant inférieure à $10^{-4}\%$, l'utilisation de la simple précision est tout à fait acceptable pour le type de résultats attendus.

	RMSD ρ	RMSD s(ρ)	P = $\sum_{\Omega_{NCI}} \rho_{SP}^{\frac{4}{3}}$	P = $\sum_{\Omega_{NCI}} \rho_{DP}^{\frac{4}{3}}$	Erreur relative de P(en %)
L1-P1, 0,2Å	5,31 10^{-7}	3,27 10^{-5}	1 402,6528	1 402,6530	2 10^{-5}
L5-P5, 0,05Å	4,21 10^{-3}	7,78 10^{-5}	1 138 856,3	1 138 856,6	3 10^{-5}

Tableau 4.5 – Écarts entre les résultats obtenus avec l'implémentation CPU du progamme NCIplot double précision et avec le portage GPU V_{HY} simple précision.

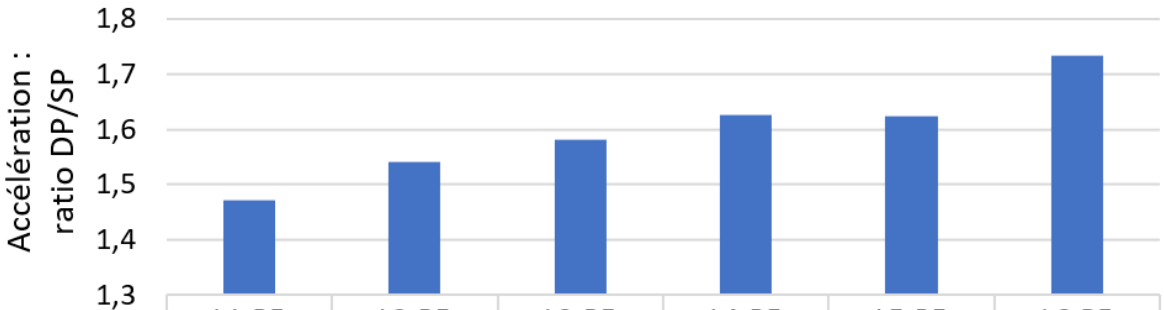
Examinons maintenant les temps de calcul en simple ou double précision. L'utilisation d'unités de calcul simple précision permet de tirer profit de la plus grande quantité de cœurs disponible sur carte graphique pour les calculs simple précision. L'utilisation de la simple précision permet donc d'accélérer les calculs.



	L1-P5	L2-P5	L3-P5	L4-P5	L5-P5	L6-P5
V_{HY} (SP, 1GPU)	5,7	7,9	12,8	23,2	35,3	71,3
V_{HY} (DP, 1GPU)	7,4	10,7	18,4	30,9	59,3	oom
V_{HY} (SP, 2GPU)	3,6	4,8	7,4	12,8	19,1	38,4
V_{HY} (DP, 2GPU)	5,3	7,4	11,7	20,8	31	66,6

FIGURE 4.16 – Temps en secondes des implémentations GPU V_{HY} simple précision (SP) ou double précision (DP) d'un calcul NCI avec une ou deux cartes graphiques K20X (GPU) sur les complexes L1-P5, L2-P5, L3-P5, L4-P5, L5-P5 et L6-P5 avec un pas de 0,025Å ; oom : *Out Of Memory*, mémoire insufisante.

La figure 4.17 permet de voir dans le détail le facteur d'accélération obtenu lors du passage de la double précision (DP) à la simple précision (SP) avec la version GPU V_{HY}. Cette accélération a tendance à augmenter avec la la taille du système étudié comme nous pouvons le voir dans la figure 4.17. Le facteur d'accélération passe de 1,5 pour le système L1-P5 à 1,7 pour le système L6-P5. Nous pouvons donc ajouter que plus le système augmente en taille, plus l'utilisation de la simple précision est efficace.



	L1-P5	L2-P5	L3-P5	L4-P5	L5-P5	L6-P5
V_{HY} (2GPU)	1,5	1,5	1,6	1,6	1,6	1,7

FIGURE 4.17 – Accélération de V_{HY} en simple précision (SP) par rapport à la double précision (DP) pour un calcul NCI avec deux cartes graphiques K20X (GPU) sur les complexes L1-P5, L2-P5, L3-P5, L4-P5, L5-P5 et L6-P5 avec un pas de 0,025Å.

En conclusion, à précision numérique identique, il est préférable d'utiliser des réels à virgule flottante simple précision dans notre cas, car les cartes graphiques K20X disposent de plus d'unités de calcul simple précision que de double précision. L'utilisation de la simple précision permet donc d'avoir accès à plus de puissance de calcul et donc d'accélérer les calculs, lorsque le système chimique étudié est suffisamment dense.

Analyse comparative de l'efficacité énergétique

Les enjeux de la consommation énergétique sont de plus en plus importants dans le monde du calcul haute performance. En Europe, le projet Mont-Blanc vise le développement de systèmes HPC avec des architectures efficaces en terme de consommation d'énergie. Il est important, dans ce cadre, d'évaluer la consommation d'énergie et l'efficacité énergétique du portage GPU en comparaison des CPU. Sur les nœuds de calcul que nous avons utilisés, les consommations énergétiques "théoriques" données sont de 2×235W pour les deux cartes GPU et de 2×77W pour les deux processeurs CPU (chacun constitué de huit cœurs). Comme souligné précédemment, les deux cartes graphiques K20X disposent de beaucoup plus de cœurs de calculs (5 376 au total) en comparaison des cœurs CPU disponibles (2x8) sur les processeurs Intel IvyBridge. Mais une des différences en terme de consommation théorique provient du fait que les cœurs du GPU possèdent une fréquence moins élevée (732 MHz) que les cœurs CPU (2,6 GHz).

Pour réaliser cette analyse énergétique comparative, la version GPU V_{HY}(SP) a été utilisée sur six complexes représentatifs (avec un pas de 0,025Å). Les mesures de consommation sont réalisées par un logiciel tiers (nommé POWMON, développé par le centre de calcul ROMEO) qui utilise les bibliothèques Intel RAPL (*Running Average Power Limit*) et NVML (*NVIDIA Management Library*). Le mode de fonctionnement de cet outil utilise un serveur client pour éviter d'influencer les performances du code analysé et retourne directement les valeurs obtenues par les bibliothèques, en Joules pour RAPL et en milliwatts pour NVML (la fonction puissance a alors été intégrée sur le temps d'exécution pour obtenir la consommation en Joules). Les performances sont collectées à une fréquence de 10Hz pour le CPU et le GPU.

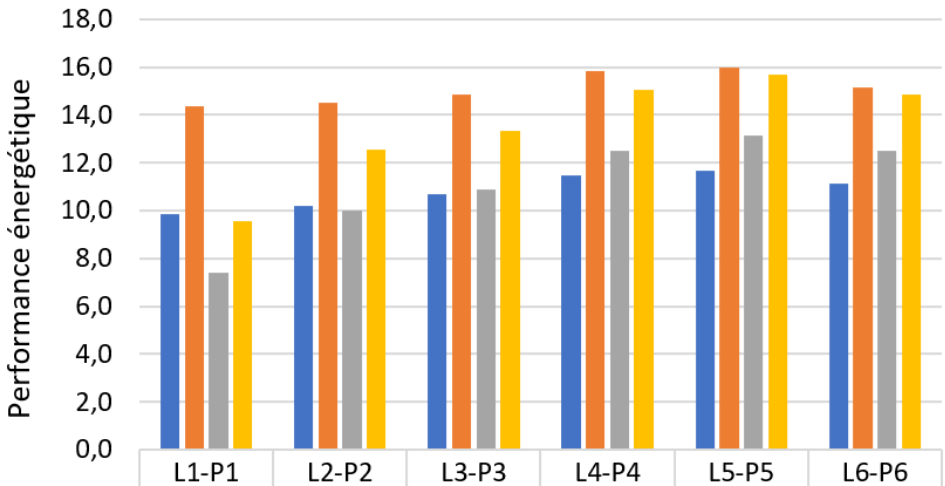
L'énergie consommée pour nos tests ainsi que les temps d'exécution sont répertoriés dans le tableau 4.6. Pour les tests GPU, deux consommations d'énergie sont distinguées, énergie1 et énergie2. La première inclut la consommation du CPU hôte, la seconde n'inclut pas la consommation du CPU hôte. Dans l'absolu, l'énergie consommée par le CPU hôte contribue significativement à la consommation globale du système même lorsque celui-ci n'est pas mis à contribution pour les calculs. Pour une exécution avec un unique GPU, le CPU consomme entre 27% et 31% de la consommation totale. Cette contribution est approximativement divisée par deux si deux GPU sont utilisées sur le même nœud, tombant à 16% pour la plus grande instance. Ces résultats caractérisent bien nos tests. Le CPU exécute le système d'exploitation ainsi que ses parties du calcul de l'application. Dans la version GPU utilisée (V_{HY}), la partie calcul est complètement accélérée sur GPU, ce faisant l'hôte est principalement dans l'attente de la fin du noyau exécuté par le GPU, et gère le lancement du noyau et le transfert des données entre le GPU et la mémoire de l'hôte. Nous pouvons remarquer que les prochaines architectures hybrides CPU/GPU ne seront pas nécessairement équipées d'un CPU haut de gamme. Actuellement, des solutions pour les systèmes embarqués, comme le Nvidia Jetson TX1, combinent un GPU optimisé avec un CPU basse consommation.

Système	**L1-P1**	**L2-P2**	**L3-P3**	**L4-P4**	**L5-P5**	**L6-P6**
16 cœurs CPU						
Énergie(J)	1 619	3 685	9 044	32 238	61 691	158 767
Temps(s)	20,0	43,8	101,2	358,5	724,6	1 819,9
1 GPU						
Énergie1(J)	164	361	847	2 814	5 284	14 260
Énergie2(J)	113	254	609	2 039	3 856	10 474
Temps(s)	2,7	3,4	6,7	20,3	35,3	92,8
2 GPU						
Énergie1(J)	218	368	830	2 578	4 691	12 704
Énergie2(J)	169	294	678	2 140	3 934	10 698
Temps(s)	1,9	2,5	4,4	11,2	19,1	49,3

Tableau 4.6 – Énergie consommée (J) par le code de référence (icc, 16 cœurs) et par la version GPU (V_{HY}) sur un calcul NCI avec une ou deux cartes graphiques K20X ; énergie1 inclut la consommation du CPU hôte contrairement à énergie2 ; temps en secondes ; pas de grille 0,025Å

Les portages GPU de l'algorithme NCI consomment beaucoup moins d'énergie que l'équivalent CPU. Les exécutions avec un ou deux GPU obtiennent des performances énergétiques similaires (voir les deux lignes énergie2 du tableau 4.6), même si l'utilisation de deux GPU demeure légèrement plus coûteuse. Cette tendance est inversée lorsque la consommation de l'hôte est prise en compte ; utiliser deux GPU devient légèrement plus intéressant. En revanche, La prise en compte de la consommation de l'hôte n'impacte pas dramatiquement le ratio CPU/GPU de consommation d'énergie figure 4.18. Par exemple, toujours sur le cas L6-P6, ajouter le coût du CPU hôte dans la consommation un GPU, fait passer le ratio énergétique CPU/GPU de 15,2 à 11,1 seulement et de 14,8 à 12,5 pour deux GPU.

Il est notable de constater que la taille du système (nombre d'atomes et taille de grille) influence la performance énergétique (consommation CPU/consommation GPU). Ce ratio augmente pour les systèmes de plus grande taille avec un maximum atteint pour le système L5-P5 composé de 955 atomes (ratio de 16). D'une manière générale, sur les exemples traités ici, le ratio se situe entre 7 et 16. Cependant, le ratio de la consommation n'est pas aussi important que l'accélération obtenue qui se situe entre 11 et 39 (dans le cas de l'utilisation de deux GPU). Même si en absolu, les portages GPU consomment moins d'énergie en comparaison des exécutions CPU, les exécutions GPU sont moins efficientes d'un point de vue énergétique pour de grandes instances. Globalement, pour des applications hautement parallélisables et denses en calculs, l'implémentation GPU est plus performante sur les deux axes : temps d'exécution et consommation énergétique. Ces deux avantages justifient l'intérêt des architectures *manycore* dans ce cas de figure.



	L1-P1	L2-P2	L3-P3	L4-P4	L5-P5	L6-P6
Une carte K20X (avec hôte)	9,9	10,2	10,7	11,5	11,7	11,1
Une carte K20X (sans hôte)	14,4	14,5	14,8	15,8	16,0	15,2
Deux cartes K20X (avec hôte)	7,4	10,0	10,9	12,5	13,2	12,5
Deux cartes K20X (sans hôte)	9,6	12,5	13,3	15,1	15,7	14,8

FIGURE 4.18 – Performance énergétique (consommation CPU/consommation GPU) pour un calcul NCI réalisé sur six complexes chimiques en utilisant une ou deux cartes graphiques K20X ; les ratios sont donnés en prenant ou pas en compte la consommation de l'hôte.

4.9 Bilan et perspectives

Dans ce chapitre 4, nous avons décrit une nouvelle implémentation rapide de la méthode NCI qui utilise la puissance de calcul des accélérateurs graphiques de NVIDIA. Ce code GPU (cuNCI[55]) est disponible sur le site igmplot.univ-reims.fr et permet la détection et visualisation des interactions intermoléculaires en tirant profit de l'accélération des cartes NVIDIA. La partie la plus coûteuse de l'approche NCI est le calcul des densités électroniques promoléculaires, et en second lieu le calcul des gradients et de la matrice Hessienne. Ces fonctions sont réparties sur les différentes ressources (blocs/processus légers) des GPU, afin de tirer complètement bénéfice de cette architecture ; plusieurs versions ont été optimisées et testées avec une variété de combinaisons de ligands, protéines et de pas de grille. L'utilisation d'une taille de bloc mémoire appropriée, une organisation adéquat des données ainsi que la modification des schémas d'accès en mémoire ont permis d'améliorer la performance du code. Nous avons aussi implémenté des versions capables d'utiliser deux GPU ce qui permet d'améliorer encore l'accélération.

Le ratio d'accélération entre les exécutions CPU (icc, OpenMP, 16 cœurs) et GPU augmente avec le nombre d'atomes du système à traiter jusqu'à atteindre 39 pour 1341 atomes et 10^8 nœuds de grille. Globalement, notre implémentation GPU V_{HY} avec deux cartes graphiques K20X surpasse la version originale NCIplot CPU en Fortran disponible librement (OpenMP, 16 cœurs). Nous avons alors pu constater un facteur d'accélération allant jusqu'à 99 fois.

La précision de notre implémentation de l'approche NCI est évaluée en comparaison des résultats obtenus par le programme d'origine NCIplot. Elle donne complète satisfaction des traitements, aussi bien qualitativement que quantitativement.

Lorsque le sujet est la parallélisation, il est important de se poser les questions d'évolutivité (*strong scaling* et *weak scaling*, définies au chapitre 1), et de consommation d'énergie pour déterminer si l'approche GPU est réellement intéressante. Cela représente un facteur actuellement de plus en plus important lors du choix de l'architecture pour accélérer les simulations. Les mesures de consommation d'énergie réalisées sur les implémentations CPU et GPU de NCI montrent que l'approche GPU est très largement avantageuse pour l'économie d'énergie. Quantitativement, le ratio énergétique CPU/GPU va de 7 à 16 pour les molécules testées. De plus, les GPU sont efficaces d'un point de vue énergétique sur des codes hautement parallélisables dû à l'accélération significative qui peut être obtenue, comme nous l'avons montré sur l'approche NCI. Tout cela fait que l'approche NCI est particulièrement bien adaptée à l'utilisation de GPU.

Nous pensons qu'une perspective intéressante est d'accélérer l'approche NCI d'origine par un pré-calcul sur des critères géométriques provenant de la chimie pour rapidement identifier les nœuds de la grille qui sont en dehors des régions NCI. En effet, de très nombreux calculs sont effectués inutilement sur beaucoup de nœuds de grille éloignés de la zone d'interaction. Ce pré-traitement géométrique pourra permettre d'accélérer la méthode NCI en se concentrant uniquement sur les nœuds entre les deux molécules réalisant les interactions. Il semble évident qu'uniquement les régions géométriquement à l'interface entre les deux molécules devraient être calculées. Dans l'approche d'origine de NCI, ces régions sont déterminées à partir de la fraction $\frac{\rho_A}{\rho}$. Toutefois, ce test nécessite le calcul des densités électroniques pour chaque nœud de la grille, ce qui est très coûteux en terme de calculs. Des alternatives existent dans le domaine du *docking* moléculaire[56] pour identifier rapidement les "zones de contact" entre deux molécules. L'idée serait par exemple de ne s'intéresser qu'aux nœuds de grilles compris dans la zone de recouvrement des rayons des sphères de van der Waals des atomes des deux molécules en interaction.

Ce travail d'accélération sur GPU est une contribution à un effort en cours dans nos groupes de recherche pour développer une utilisation de l'approche NCI dans le domaine du *docking* moléculaire. La plus grande instance utilisée dans ce chapitre, utilise un pas de grille plus petit que nécessaire (0,025Å) ; un ligand de 96 atomes, un site de 1341 atomes nous semblent aussi être une limite supérieure pour les problèmes d'interaction ligand-protéine. Cette limite supérieure ne prend actuellement que 49 s sur un seul nœud en utilisant notre code V_{HY_2GPU}. Revenir à un pas plus raisonnable de 0,1 Å amène à un temps de 2 s pour cette plus grosse instance testée. La perspective par pré-identification géométrique des nœuds d'interaction grâce aux rayons de van der Waals laisse penser que l'on peut encore réduire d'un ordre de grandeur ce temps de calcul compte-tenu de ces temps de calcul déjà beaucoup réduits grâce à deux cartes GPU. Exploiter plus de cartes GPU encore (sur d'autres nœuds distants) en utilisant une bibliothèque MPI ne nous a donc pas semblé utile pour une application dans le domaine du *docking* moléculaire. Une version est actuellement distribuée (cuNCI[55]), permettant d'utiliser tous les GPU d'un nœud. Cette version GPU de NCI est adaptée pour accélérer les problèmes nécessitant de calculer plusieurs instances. Par exemple, plusieurs *snapshots* d'une trajectoire de dynamique moléculaire pour suivre l'évolution des interactions d'un système chimique dans le temps ou encore le calcul NCI d'une population de ligands se liant à une protéine pour un algorithme génétique réalisant du *docking* moléculaire. Alors, tous ces calculs indépendants, s'il respectent les limites mémoires, peuvent être distribués sur plusieurs nœuds de calcul.

Des instances plus grandes (par exemple d'interaction protéine-protéine) pourraient tirer profit d'une exécution sur plusieurs nœuds. Toutefois, une grille de calcul plus grande serait un obstacle lors du post-traitement de la méthode NCI. En effet, les deux fichiers

résultats (`.cube`) pourraient ne pas tenir en mémoire pour la visualisation des régions d'interactions moléculaire lors de l'utilisation d'applications graphiques comme VMD ou ParaView. L'étape limitante devient clairement la mémoire pour ces grandes grilles. En revanche, nous pouvons noter que les cartes graphiques NVIDIA de dernières générations, Pascal et Volta, possèdent des configurations de la mémoire allant respectivement à 16 Go et 32 Go au maximum. Ces quantités de mémoire permettent de calculer à la fois des fichiers cube plus grands ainsi que de les visualiser. L'objectif premier du code actuel est d'être utilisable sur un seul nœud.

Trois autres perspectives sont à envisager. Tout d'abord, comme mentionné section 3.7, le transfert à la nouvelle approche IGM de nos connaissances acquises lors du portage GPU de la méthode NCI. Comme évoqué précédemment, les mêmes propriétés issues de la densité électronique sont employées dans cette approche récente IGM[42][43]. L'avantage de cette méthode IGM étant l'automatisation de l'identification des interactions moléculaires, son portage GPU permettrait de traiter encore plus rapidement des séquences de calculs constituant des trajectoires de dynamique moléculaire ou des populations d'algorithme génétique. Cette perspective en amène une autre : pouvoir réaliser des calculs NCI/IGM en temps réel. Les temps de calculs obtenus avec notre meilleure version 2 GPU/V_{HY} ($\simeq$ 1 s) et la perspective de disposer de 16 cartes GPU sur les dernières architectures (DGX-2), combiné à un pré-traitement géométrique des grilles nous autorise à penser que ce type de projet est tout à fait réalisable. Enfin, une dernière perspective pourrait être de recommencer le travail effectué sur l'approche NCI (ou IGM) utilisant une densité non pas promoléculaire mais quantique. Ces calculs, s'appuyant sur l'expression d'une fonction d'onde (développée aussi sur une base de fonctions exponentielles) sont très couteux et répétés sur tous les nœuds de la grille. Ils pourraient aussi bénéficier de l'architecture des GPU.

Chapitre 5

Accélération du logiciel GAMESS sur architecture GPU

Dans ce chapitre nous nous intéressons au travail de recherche réalisé sur une partie du logiciel GAMESS (*General Atomic and Molecular Electronic Structure System*) disponible librement [44]. GAMESS est un logiciel de chimie théorique mettant à disposition un certain nombre de méthodes de chimie quantique. Il a commencé a être écrit en 1982 et son développement est encore actif aujourd'hui. Le choix de s'intéresser au logiciel GAMESS est principalement motivé par plusieurs raisons. La première est que GAMESS est un logiciel de portée internationale. De plus, le logiciel GAMESS est libre. Enfin, le logiciel GAMESS implémente un grand nombre de méthodes, en particulier la méthode quantique DFTB, qui combinée aux méthodes FMO et PCM peut servir de fonction de score dans le logiciel AlgoGen développé par nos deux laboratoires ICMR et CReSTIC. La méthode DFTB est actuellement une méthode quantique performante en matière de temps de calcul et de précision des résultats pour les raisons évoquées dans la section 3.3 de cette thèse. L'utilisation de l'approche FMO (section 3.4) en combinaison de la DFTB permet d'accélérer les temps de calculs et de tendre vers une extensibilité linéaire, soit de rendre le temps de calcul linéaire avec le nombre d'atomes traités ce qui en chimie Quantique est un véritable défi. Pour assurer la convergence des calculs, l'ajout de l'effet de solvant est nécessaire dans notre cas, condition soulignée dans la littérature [3]. Cette combinaison DFTB / FMO / PCM possède donc tous les atouts pour être utilisée dans le cadre de simulations de *docking* moléculaire traitant des systèmes ligand-protéine possédant plusieurs milliers d'atomes. Elle est actuellement envisagée pour remplacer la méthode quantique MOZYME[32] couplée à l'approche semi-empirique PM7 utilisée actuellement dans le logiciel de *docking* moléculaire AlgoGen. Considérer l'accélération de l'approche combinée DFTB / FMO / PCM fait donc partie des priorités de nos laboratoires.

Plusieurs difficultés ont été rencontrées. Contrairement au travail sur l'approche NCI, la difficulté la plus générale provient du fait que nous avons du intervenir dans un code déjà écrit (sans le ré-écrire). Le logiciel GAMESS est composé d'un grand nombre de fonctionnalités écrites par un ensemble d'auteurs différents. De plus, la volonté sous-jacente des auteurs du logiciel GAMESS est d'être le plus universel possible vis-à-vis des différentes machines disponibles sur le marché. Ces faits rendent le logiciel GAMESS difficile d'accès quand il s'agit "d'entrer" dans le code. Une autre difficulté provient du fait qu'en fonction de l'installation réalisée, le paradigme de parallélisation change. Ces paradigmes sont décrits dans la suite.

Ajoutons à cela, que certaines installations de GAMESS sont dépendantes de bibliothèques externes, comme le montre la figure 5.1.

Le travail réalisé a suivi le plan suivant :

- Installer GAMESS de la manière la plus simple possible afin d'exécuter le cas test sur un seul cœur CPU.
- Passer à l'utilisation de plusieurs cœurs CPU d'un nœud de calcul sur le cas test.
- Identifier les points chauds du calcul réalisé, portables sur GPU.
- Porter et optimiser le code correspondant sur GPU.
- Analyser les performances du portage GPU.

Ce travail s'est avéré plus difficile à réaliser que prévu pour les raisons évoquées précédemment. L'investissement initial en matière de temps et d'effort fut important et risqué. Contrairement à l'approche NCI, problème de grille totalement adapté à un portage sur GPU, nous n'avions concernant GAMESS aucune idée du bénéfice possible de l'utilisation de cartes graphiques. La forme de ce chapitre 5 suit le déroulement chronologique du travail réalisé sur le logiciel GAMESS pour obtenir un portage GPU et l'analyser. L'objectif est d'accélérer si possible une partie du logiciel GAMESS par l'utilisation de la technologie *manycore* des GPU ainsi que de s'intéresser à la performance énergétique obtenue.

Dans la suite sont décrits les installations possibles de GAMESS et les analyses de performance réalisées avec principalement le logiciel MAQAO[19] sur un cas test que nous présenterons. La fonction "limitante" `ascpot` a été identifiée suite aux analyses : nous décrivons aussi le portage GPU réalisé.

5.1 Étapes préliminaires d'installations et d'exécutions

Nous nous sommes intéressés à la version du logiciel GAMESS publiée le 30 septembre 2017. L'installation du logiciel GAMESS sur ROMEO fut compliquée et la compréhension précise du mode de fonctionnement interne de ce logiciel a demandé beaucoup d'efforts.

De manière globale, cette phase d'installation et de compréhension a pris plus de temps que le travail de portage GPU lui-même. Nous avons rencontré plusieurs difficultés pour "rentrer" dans le code :

- En comparaison du code NCI (3 314 lignes), GAMESS est un gros code de chimie quantique (environ 1,5 millions de lignes).
- Le code source de GAMESS est implémenté en Fortran, C et C++. Le Fortran est souvent utilisé dans le domaine de la chimie. En revanche l'utilisation combinée de langages de programmation ajoute une difficulté.
- Le logiciel GAMESS fournit un script de soumission, cependant ce script ne prévoit pas l'utilisation du gestionnaire SLURM utilisé au centre de calcul ROMEO que nous utilisons.
- Le programme GAMESS peut être configuré de multiples façons ; un aperçu des différentes configurations est reporté figure 5.1. Ajoutons qu'en fonction de l'installation, le modèle de parallélisation utilisé par GAMESS peut changer.
- Les auteurs de GAMESS encouragent l'utilisation de la bibliothèque LIBCCHEM (interne au logiciel GAMESS) pour deux cas de figures qui nous ont été évoqués par le professeur Mark GORDON (auteur principal de GAMESS) :
  1. Développer des codes C++ orientés objet CPU capables de tirer plus de performances des architectures informatiques modernes en comparaison du code Fortran de GAMESS.
  2. Développer des codes pour les architectures GPU. Il est intéressant d'informer le lecteur que lors de nos échanges avec le professeur Mark GORDON ce der-

nier nous a expliqué que de leurs expérimentations (sans indiquer les méthodes testées) l'utilisation de GPU n'apporta qu'un gain mineur par rapport à une implémentation CPU optimisée.

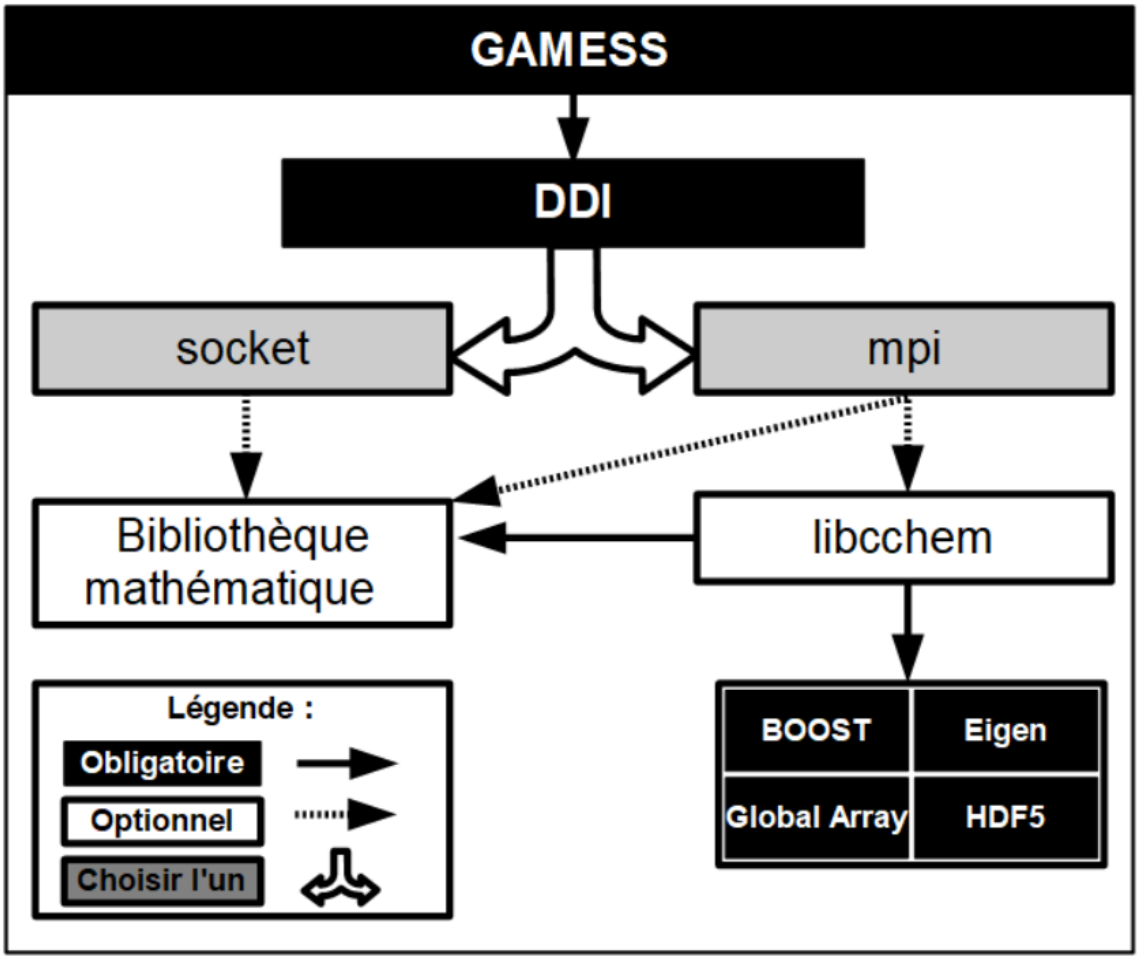


FIGURE 5.1 – Représentation des configurations possibles pour l'installation de GAMESS, où *bibliothèque mathématique* peut être (au choix) ATLAS, PGI BLAS, ACML ou MKL.

La figure 5.1 illustre la diversité des installations pouvant être réalisées. Plusieurs installations de GAMESS ont été réalisées au cours de cette thèse, avec différentes configurations. Nous détaillerons dans la suite de ce manuscrit les différentes installations réalisées. Avant cela nous allons évoquer les grandes lignes de chaque bibliothèque. Quant aux bibliothèques DDI (Distributed Data Interface)[57] et LIBCCHEM, elles seront détaillées par la suite car définissent des notions importantes de GAMESS : le modèle de parallélisation CPU utilisé.

DDI est une implémentation permettant l'utilisation de mémoire partagée globale, des ressources CPU, par utilisation de messages explicites. C'est une des solutions possibles pour la gestion du parallélisme sur CPU. Nous décrivons cette bibliothèque dans la section 5.1. Peu importe la configuration de LIBCCHEM, la bibliothèque DDI doit être compilée.

Un choix entre deux options s'offre à l'utilisateur souhaitant installer GAMESS :

- Utiliser le mode *socket*.
- Utiliser le mode *mpi*.

Le mode *socket* s'intéresse à l'utilisation des ressources d'un nœud seul de calcul. Le mode *mpi* permet l'utilisation de plusieurs nœuds de calcul.

Ces deux modes (*socket* et *mpi*) permettent l'utilisation d'une bibliothèque mathématique pour accélérer les fonctions classiques. Cette bibliothèque peut être (au choix de l'utilisateur) ATLAS, PGI BLAS, ACML ou MKL.

Le mode *mpi* peut utiliser la bibliothèque LIBCCHEM. L'utilisation de LIBCCHEM est encouragée par les auteurs de GAMESS qui cherchent à réaliser la transition de ce logiciel vers les pratiques contemporaines de l'informatique (programmation orientée objet) ainsi que l'utilisation des GPU. Cette bibliothèque est détaillée dans la section 5.1. Notons tout de même que son utilisation change le modèle de parallélisation CPU par rapport au modèle de la bibliothèque DDI.

Le mode d'installation LIBCCHEM implique nécessairement l'utilisation de plusieurs autres bibliothèques :

- BOOST est un ensemble de bibliothèques gratuites écrites en C++ révisées par des pairs. L'objectif des auteurs de BOOST est de fournir des implémentations de référence pour éviter de "réinventer la roue".
- La bibliothèque Eigen est une bibliothèque écrite en C ++ pour l'algèbre linéaire : matrices, vecteurs, solveurs numériques et algorithmes associés.
- La bibliothèque Global Array (GA) fournit une API pour la programmation de la mémoire partagée sur les ordinateurs à mémoire distribuée pour les tableaux multidimensionnels.
- La bibliothèque HDF5 est un modèle de données, une bibliothèque et un format de fichier pour stocker et gérer des données.

Pour ce travail de recherche, nous nous sommes principalement intéressés au mode *socket* (avec ou sans bibliothèque mathématique) ainsi qu'à l'utilisation du mode *mpi* avec la bibliothèque LIBCCHEM (obligeant l'utilisation d'une bibliothèque mathématique). Nous détaillons ces installations après avoir détaillé la bibliothèque DDI puis la bibliothèque LIBCCHEM.

Distributed Data Interface (DDI)

Distributed Data Interface (DDI) est une couche permettant la transmission de messages. Cette bibliothèque, écrite en C, permet l'exécution parallèle de GAMESS (en dehors de l'utilisation de LIBCCHEM). DDI a été développée à l'origine pour GAMESS mais peut être utilisée séparément. DDI contient les fonctions de programmation parallèles habituelles, telles que l'initialisation/fermeture de communication, l'envoi de messages point à point, et des opérations collectives de somme globale et de diffusion.

DDI tente d'exploiter de manière extensible, l'entièreté du système informatique disponible. Le concept de mémoire distribuée est contenu dans la partie *Remote Memory Access* de MPI-2 [58]. À l'origine, ce concept de mémoire distribuée est implémenté dans la bibliothèque Global Array écrit par Pacific Northwest National Laboratory. Concrètement l'idée est de fournir trois fonctions pour accéder à la mémoire des autres processeurs (locaux ou éloignés) : PUT, GET et ACCUMULATE.

Comme dans DDI les accès mémoire aux autres CPU sont explicites par les appels aux fonctions, le développeur est conscient qu'un message doit être transmis. Cette pratique encourage le transfère de multiple données dans un seul message. L'utilisation d'un appel à une fonction pour accéder à la mémoire éloignée est aussi une reconnaissance de la nature hétérogène (non-uniform memory access : NUMA) des ordinateurs parallèles.

Pour qu'un CPU puisse communiquer des données à un deuxième CPU quand il en a besoin et sans un délai significatif, le processus de calcul du premier CPU doit s'interrompre brièvement pour fournir les données. Ce type de communication est connue en

informatique sous la dénomination "*one sided messages*".

Dans notre cas, lors de l'utilisation du mode *socket*, le modèle mémoire implique l'utilisation de deux processus par cœur de processeur, comme présenté dans la figure 5.2. La première moitié des processus fait de la chimie quantique. Juste après l'exécution, la deuxième moitié des processus appelle une routine de service DDI qui consiste en une boucle infinie afin de réaliser les requêtes GET, PUT et ACC jusqu'à la fin du travail. L'interruption du programme est confiée au système d'exploitation.

Quand le processus p=1 a besoin de données se situant sur le processus 0, une requête est envoyée au serveur de données p=2 pour transférer l'information vers le processus de calcul p=1. Le processus de calcul p=0 n'est absolument pas affecté par une telle transaction.

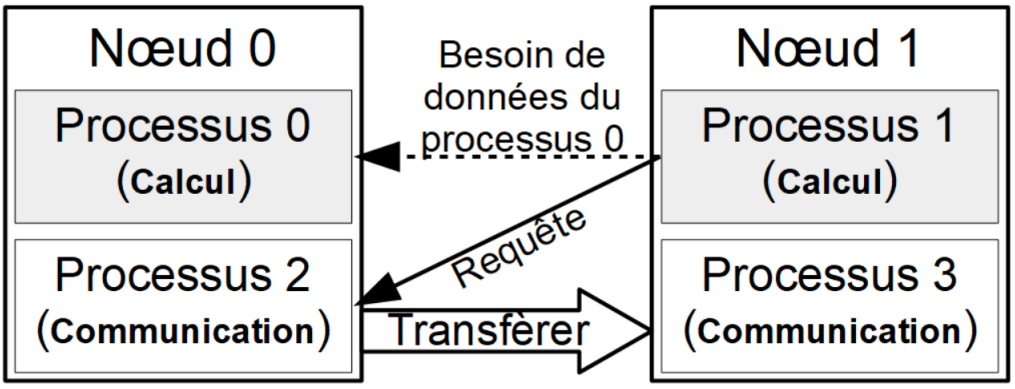


FIGURE 5.2 – Modèle mémoire de GAMESS.

Ce comportement est celui réalisé lors de l'utilisation du code GAMESS sans la bibliothèque LIBCCHEM qui possède un autre paradigme. La section suivante décrit la bibliothèque libcchem utilisable conjointement à GAMESS.

La bibliothèque LIBCCHEM

La bibliothèque LIBCCHEM est écrite en C++ avec pour but de ramener la programmation dans GAMESS à des standards plus récents, notamment par le biais de la programmation orientée objet ou encore avec l'utilisation de GPU NVIDIA par le biais de CUDA.

La bibliothèque LIBCCHEM a commencé à être écrite entre 2010 et 2012 par Andrey Asadchev au département de chimie de l'université d'état de l'Iowa. Cette bibliothèque comprend déjà des fonctions de chimie quantique utilisant des GPU. Trois noyaux GPU sont fournis :

- The closed shell SCF energy.
- The closed shell MP2 energy.
- The closed shell CCSD(T) energy.

Les auteurs de GAMESS incitent fortement l'implémentation des nouvelles méthodes dans cette bibliothèque.

Le paradigme de parallélisation diffère de l'utilisation normale de GAMESS et passe par la bibliothèque *Global Arrays* (GA). Cette dernière est utilisée pour distribuer la mémoire. Dans les faits GA fonctionne en utilisant MPI. MPI doit débuter en exécutant un seul processus par nœud, avec LIBCCHEM exécutant les processus légers sur les CPU

et les GPU disponibles lorsqu'un noyau a besoin d'être calculé. Un binaire combinant l'utilisation de GAMESS et LIBCCHEM doit être réalisé afin de pouvoir utiliser les trois noyaux précédemment évoqués. Tous les algorithmes de GAMESS ne sont pas retranscrits dans LIBCCHEM. Les implémentations existantes dans LIBCCHEM utilisent des variables double précision, dans le but de garder la même précision que le code de référence.

Nous nous sommes intéressés, pour ce travail de recherche, à l'utilisation de la bibliothèque LIBCCHEM car cette dernière implémente certaines fonctionnalités en les accélérant sur GPU avec CUDA, ce qui nous permet de nous inspirer de leurs codes pour réaliser le notre. De plus, les auteurs de GAMESS incitent fortement l'utilisation de cette approche.

5.2 Analyse des algorithmes implémentés par GAMESS

Mon travail, pour rappel, s'inscrit dans un contexte de recherche plus global, qu'est le logiciel de *docking* moléculaire AlgoGen, développé conjointement par les laboratoires ICMR et CReSTIC. AlgoGen permet de faire des simulations de *docking* moléculaire en se basant sur une fonction de score qui est une méthode quantique semi-empirique (PM7). AlgoGen fait donc appel actuellement au logiciel de chimie quantique MOPAC[59]. Ce dernier propose la méthode MOZYME[32] pour rendre linéaire les temps de calcul quantique avec la taille du système. La nouvelle combinaison de méthodes quantiques DFTB / FMO / PCM proposée dans le logiciel de chimie quantique GAMESS est une alternative de qualité envisagée pour AlgoGen. C'est donc avec cet objectif que nous nous intéressons à GAMESS qui implémente les méthodes DFTB, FMO et PCM et permet une utilisation combinée de ces dernières. Ces méthodes ont été introduites précédemment dans les sections 3.3, 3.4 et 3.5. La méthode DFTB est une méthode quantique se basant sur la méthode DFT.

En combinant la méthode DFTB avec l'approche FMO, l'extensibilité des temps de calcul en fonction de la taille du système est quasi linéaire. La méthode PCM permet de simuler l'effet d'un solvant sur le système étudié, et l'effet du solvant s'est révélé nécessaire[3] pour faire converger les calculs de l'approche combinée DFTB / FMO.

Nous avons concentré nos efforts pour cette partie de thèse sur l'utilisation de l'approche combinée DFTB / FMO / PCM dans GAMESS. Cette utilisation ne sollicite qu'une partie du logiciel GAMESS. L'objectif est d'étudier ce type d'exécution (DFTB / FMO / PCM) afin de déterminer si des gains par l'utilisation des GPU sont possibles. Le matériel pour réaliser cette partie reste celui disponible sur le centre de calcul ROMEO. De même, les termes pour la compréhension d'un portage GPU sont définis dans le chapitre 2 de cette thèse.

Pour commencer, une installation simple de GAMESS a été réalisée en utilisant le mode *socket* et les compilateurs d'Intel (ifort et icc). Pour ce faire, la procédure d'installation proposée par les auteurs de GAMESS a été suivie. C'est avec le binaire obtenu que nous avons testé les exemples fournis au moment du téléchargement de GAMESS. Puis nous avons constitué un système chimique test qui correspond à une instance type rencontrée dans les simulations de *docking* moléculaire. La section qui suit présente immédiatement ce système chimique.

Constitution du système chimique test

Notre cas test est basé sur la protéine P6 étudiée précédemment dans le portage de NCI, section 4.3. Cette molécule est constituée de 1244 atomes. Comme nous utilisons la méthode FMO pour linéariser le temps de calcul, cette molécule a du être fragmentée, dans notre cas en 75 fragments. Les auteurs de GAMESS conseillent de réaliser cette fragmentation en groupes d'acides aminés. Chaque fragment représente deux acides aminés, soit environ 15 atomes (en fonction du type des acides aminés). Cette étape a été effectuée manuellement. À terme, dans le cadre de l'intégration de l'approche combinée DFTB / FMO / PCM au *docking* moléculaire via le code AlgoGen, cette tâche de fragmentation pourrait être automatisée. Pour l'approche PCM (simulation de l'effet du solvant), nous avons choisi d'utiliser l'eau comme solvant : c'est le solvant qui caractérise le mieux le milieu d'étude des interactions ligand-protéine. La méthode DFTB a été choisie pour réaliser le calcul de l'énergie du système. Un maximum de 50 itérations a été fixé comme seuil de convergence pour le processus itératif SCF.

Nous avons créé ce cas test afin d'avoir un exemple significatif en matière de taille pour une utilisation dans le logiciel de *docking* moléculaire AlgoGen.

Profilage CPU

Profilage sans bibliothèque mathématique

C'est donc avec l'installation de GAMESS en mode *socket* (exécution limitée à un seul nœud) et le compilateur d'Intel que nous avons pu réaliser nos premières exécutions du code. Les nœuds de calcul de ROMEO disposent chacun de 16 cœurs CPU. Nous avons tout d'abord regardé l'extensibilité des temps d'exécution totale en fonction du nombre de cœurs CPU.

Processus légers	1	2	4	8	16
Socket ifort (s)	154	94,2	63,8	51,2	43,0
Accélération	1,0	1,6	2,4	3,0	3,6

Tableau 5.1 – Tableau des temps d'exécution d'un calcul d'énergie DFTB / FMO / PCM en secondes pour 1, 2, 4, 8 et 16 processus légers de GAMESS (*socket* et ifort) sur notre cas test (1244 atomes).

Dans le tableau 5.1, nous pouvons voir les temps en secondes de l'installation *socket* avec ifort obtenus avec 1, 2 ,4 ,8 et 16 processus légers. L'ajout de ressources amène un gain, qui n'est cependant pas optimal en fonction du nombre de cœurs CPU utilisés.

Afin d'avoir des informations sur les goulots d'étranglement du calcul, plusieurs profilages ont été réalisés avec le logiciel MAQAO[19].

Fonction	Info	Temps moyen (%)	Temps min (s)	Temps max (s)	Temps moyen (s)
ascpot	3252@pcmief.f	77,28	119,8	119,28	119,28
ddot	164@blas.f	6,06	9,36	9,36	9,36
daxpy	66@blas.f	2,15	3,32	3,32	3,32
einvit	886@eigen.f	2,03	3,14	3,14	3,14
__brk_limit		1,63	2,52	2,52	2,52
dftb_pcmpot	5268@dftbfo.f	1,40	2,16	2,16	2,16
eqlrat	714@eigen.f	1,31	2,02	2,02	2,02
dspmv	1010@blas.f	1,27	1,96	1,96	1,96
pcmnup	1860@fmoprp.f	1,00	1,54	1,54	1,54

Tableau 5.2 – Extrait de l'analyse d'une exécution DFTB / FMO / PCM (calcul d'énergie) de GAMESS (socket et ifort) avec 1 processus léger par MAQAO, sur notre cas test (1244 atomes).

Les temps de passage dans chaque routine et leur poids relatif ont été obtenus avec l'utilisation d'un unique processus léger et de l'installation *socket* avec ifort. Une partie de ces informations est présente dans le tableau 5.2. Nous pouvons voir qu'une fonction se distingue. Cette fonction c'est `ascpot` qui représente 77% du temps de calcul. Cette fonction `ascpot` est appelée au moment de simuler l'effet du solvant (méthode PCM). Dans la méthode quantique combinée DFTB / FMO / PCM, c'est donc la dernière (PCM) qui est limitante sur notre cas test. Une telle fonction peut nous mettre sur la piste d'un cas de portage GPU très favorable si cette fonction s'y prête. Dans le cas contraire, 77% du temps de calcul ne pourra pas être amélioré, produisant un portage potentiel très peu intéressant si l'on s'en réfère à la loi d'Amdahl précédemment évoquée section 1.4. Si nous regardons les autres fonctions qui ressortent du profilage tableau 5.2, nous pouvons voir que les deux fonctions qui suivent (ddot et daxpy) ne représentent que 8% du temps de calcul et sont des fonctions mathématiques classiques (respectivement produit scalaire, et constante fois un vecteur plus un vecteur). Il va donc être difficile dans ce cas d'obtenir des performances en comparaison de ces bibliothèques optimisées disponibles (ATLAS, PGI BLAS, ACML ou MKL).

Regardons le profil d'une exécution de cette même installation de GAMESS (*socket* et ifort) avec 16 processus légers afin d'avoir un premier avis sur les fonctions actuellement parallélisées.

Fonction	Info	Temps moyen (%)	Temps min (s)	Temps max (s)	Temps moyen (s)
ddot	164@blas.f	25,96	8,82	10,60	9,52
ascpot	3252@pcmief.f	20,24	7,28	7,54,	7,42
daxpy	66@blas.f	10,10	3,24	4,06	3,70
einvit	886@eigen.f	9,31	2,64	3,80	3,41
eqlrat	714@eigen.f	4,87	1,52	2,16	1,78
dspmv	1010@blas.f	4,48	1,20	2,04	1,64
etrbk3	1476@eigen.f	3,06	0,84	1,32	1,12
dspr2	1256@blas.f	1,68	0,44	0,80	0,62
dsyrk	2299@blas.f	1,55	0,40	0,84	0,57
einmgs	1303@eigen.f	1,44	0,32	0,86	0,53
__intel_memset		1,19	0,30	0,62	0,44
dftb_esp	241@dftbfo.f	1,07	0,28	0,64	0,39

Tableau 5.3 – Résumé d'une exécution DFTB / FMO / PCM (calcul d'énergie) de GAMESS (socket et ifort) avec 16 processus légers par MAQAO, sur notre cas test (1244 atomes).

Le tableau 5.3 résume une exécution de GAMESS (*socket* et ifort) avec 16 processus légers. Comparé à l'exécution sur un cœur, ici quatre fonctions ressortent avec des poids relatifs compris entre 9% et 26%. On peut voir que la fonction `ascpot` est cette fois en deuxième position, passant de 77% (précédemment avec un cœur CPU) à 20% du temps total de l'exécution avec 16 cœurs. Avec cette information, nous savons que la fonction `ascpot` est parallélisée, un bon signe pour un portage sur GPU.

Profilage avec bibliothèque mathématique

Nous allons regarder le comportement d'une exécution DFTB / FMO / PCM sur notre cas test lorsqu'une bibliothèque mathématique est utilisée. Le tableau 5.4 ajoute une ligne au tableau 5.1 précédent donnant les temps en secondes pour l'installation de GAMESS socket, ifort et la bibliothèque mathématique MKL d'Intel. Nous pouvons constater une légère amélioration (3 s) du temps de calcul passant de 154 s pour 1 processus léger à 151 s. Ce léger gain se retrouve à chaque fois, peu importe le nombre de processus légers utilisés. Cette installation étant plus performante nous allons donc la profiler avec MAQAO, avec un processus léger dans un premier temps, afin de voir l'impact de la bibliothèque MKL.

Processus légers	1	2	4	8	16
Socket ifort (s.)	154	94,2	63,8	51,2	43
Socket ifort MKL (s.)	151	91,4	61,3	46,5	39,4

Tableau 5.4 – Tableau des temps en secondes (calcul d'énergie DFTB / FMO / PCM) pour 1, 2, 4, 8 et 16 processus légers de GAMESS (*socket*, ifort et MKL) sur notre cas test (1244 atomes).

Le tableau contenu dans la figure 5.5 résume l'analyse (du logiciel MAQAO[19]) d'un calcul d'énergie DFTB / FMO / PCM par GAMESS (socket, ifort et la bibliothèque MKL) avec 1 processus léger. Cette analyse est à comparer à celle du tableau 5.2 (sans bibliothèque MKL). Cette analyse nous permet de confirmer que la fonction `ascpot` prédomine le calcul à près de 79%. `ascpot` ne semble donc pas tirer profit de l'utilisation de cette bibliothèque.

Fonction	Info	Temps moyen (%)	Temps min (s)	Temps max (s)	Temps moyen (s)
ascpot	3252@pcmief.f	78,64	119,22	119,22	119,22
mkl_blas_avx_xddot		2,65	4,02	4,02	4,02
mkl_blas_avx_xdaxpy		2,65	4,02	4,02	4,02
einvit	886@eigen.f	2,39	3,62	3,62	3,62
__brk_limit		2,03	3,08	3,08	3,08
dftb_pcmpot	5268@dftbfo.f	1,54	2,34	2,34	2,34
mkl_blas_avx2_dgemmt_nobufs		1,35	2,04	2,04	2,04
eqlrat	714@eigen.f	1,12	1,70	1,70	1,70
pcmnup	1860@fmoprp.f	1,06	1,60	1,60	1,60

Tableau 5.5 – Profil d'une exécution DFTB / FMO / PCM (calcul d'énergie) de GAMESS (socket, ifort et bibliothèque MKL) avec 1 processus léger réalisé avec le logiciel MAQAO[19] sur notre cas test (1244 atomes).

Regardons maintenant le profilage réalisé avec le logiciel MAQAO[19], figure 5.6, de GAMESS (socket, ifort et la bibliothèque MKL) avec 16 processus légers. La fonction `ascpot` se retrouve en première place avec 22% du temps de calcul total, et les fonctions

daxpy et ddot juste derrière avec 12%. La situation est un peu comparable à celle du tableau 5.3 (sans bibliothèque MKL). Dans les deux cas de figure la fonction `ascpot` semble donc la plus propice à l'obtention de gains de temps par portage GPU. En revanche, ces gains seront limités par le fait que d'autres fonctions impactent aussi le calcul significativement.

Fonction	Info	Temps moyen (%)	Temps min (s)	Temps max (s)	Temps moyen (s)
ascpot	3252@pcmief.f	22,06	7,32	7,56	7,42
mkl_blas_avx_xdaxpy		11,84	3,42	4,58	3,98
mkl_blas_avx_xddot		11,79	3,48	4,42	3,96
einvit	886@eigen.f	10,83	3,28	4,02	3,64
mkl_blas_avx2_dgemmt_nobufs		6,25	1,56	2,74	2,10
eqlrat	714@eigen.f	5,13	1,42	2,06	1,73
etrbk3	1476@eigen.f	4,02	1,10	1,62	1,35
mkl_blas_avx_dspmv		3,48	0,96	1,38	1,17
mkl_blas_avx_dspr2		1,91	0,46	0,88	0,64

Tableau 5.6 – Profil d'une exécution DFTB / FMO / PCM (calcul d'énergie) de GAMESS (socket, ifort et la bibliothèque MKL) avec 16 processus légers réalisé avec le logiciel MAQAO[19] sur notre cas test (1244 atomes).

Il ressort de toutes nos analyses MAQAO que la routine `ascpot` contribue significativement au goulot d'étranglement d'un calcul quantique DFTB / FMO / PCM sur 1244 atomes. Contrairement aux autres fonctions comme ddot(produit scalaire), `ascpot` n'est, a priori, pas une fonction mathématique standard. Elle a été écrite par les auteurs de GAMESS pour simuler l'effet du solvant (partie PCM du calcul). À ce titre, il est intéressant de l'examiner pour voir si elle peut tirer profit d'une accélération sur carte graphique.

Il résulte également de nos analyses, qu'au mieux, lors d'une exécution sur 16 coeurs, nous ne pourrons accélérer, par portage sur GPU de la fonction `ascpot` qu'un quart du temps de calcul DFTB / FMO / PCM. Nous avons donc à faire à une situation beaucoup moins favorable que lors du travail de portage de l'approche NCI, détaillé chapitre 4 de cette thèse.

Nous sommes donc entrés dans le code afin de voir ce que représente la fonction `ascpot` dans GAMESS.

5.3 Au coeur de l'effet de solvant : la fonction `ascpot`

La fonction ayant attiré notre attention est `ascpot` car représentant environ 77% des temps de calcul avec 1 processus léger et 22% avec 16 processus légers. Cette fonction réalise une partie de l'algorithme PCM qui permet comme décrit section 3.5, de simuler l'effet d'un solvant sur le système étudié. Le nom `ascpot` signifie "*Apparent Surface Charge Potential*". Il s'agit ici de calculer en N_T points de la surface de la cavité moléculaire, le potentiel électrique résultant de différentes charges électriques fictives (apparentes) q_i placées au centre des N_T éléments géométriques constituant cette surface. Ce jeu de charges joue un rôle clé dans l'équation (3.25) qui traduit l'effet du solvant sur la densité électronique du soluté. La procédure (décrite section 3.5) implémentée dans GAMESS pour obtenir le jeu de charges apparentes est itérative. $q^{(n)}$ à l'itération n prend la forme suivante :

$$q^{(n)} = C_0^{-1}(q^{(0)} - C_1 q^{(n-1)})$$

C_0 et C_1 contiennent respectivement les éléments diagonaux et hors-diagonaux d'une matrice géométrique C de dimension $N_T \times N_T$. Notons de suite que le nombre N_T de

triangles formant la surface de la cavité de solvatation peut être grand (8 946 dans le plus petit cas que nous étudierons). $q^{(0)}$ est le jeu (un vecteur) de N_T charges apparentes de départ dépendant de la densité électronique quantique du soluté. Afin que le propos soit limpide au lecteur, la fonction `ascpot` réalise le produit matrice-vecteur $C_1 q^{(n-1)}$. C'est ce produit matrice-vecteur que nous avons porté sur GPU : $V = C_1 q^{(n-1)}$ Le vecteur $q^{(n-1)}$ est un vecteur dont chacune des N_T composantes représente une charge électrique. C_1 représente une matrice carrée ($N_T \times N_T$), dont chaque élément représente l'inverse de la distance d_{ik} entre deux éléments (triangles) différents i et k de la surface de la cavité. À ce titre, les éléments diagonaux de C ne doivent pas être considérés dans le calcul. Par exemple, dans le cas d'une cavité moléculaire constituée de 100 éléments de surfaces, C_1 représente alors une matrice de dimension (100, 100) sans élément diagonal et $q^{(n-1)}$ un vecteur colonne de 100 éléments. Il faut donc réaliser le produit matrice-vecteur $C_1 q^{(n-1)}$ en omettant la contribution des éléments diagonaux. Ceci introduit normalement une condition dans le traitement. Le vecteur résultat V possède bien alors 100 composantes, et chacune (V_k) s'obtient par une somme de 99 produits et non de 100 comme dans un produit matrice-vecteur classique :

$$V_k = \sum_{i \neq k}^{N} V_k^i \tag{5.1}$$

avec :

$$V_k^i = C_1[k,i] q^{(n-1)}[i] = \frac{q^{(n-1)}[i]}{d_{ki}} \tag{5.2}$$

d_{ki} étant la distance entre les deux éléments de surface k et i. Un de ces produits élémentaires $V_k^i : C_1[k,i] q^{(n-1)}[i]$ représente d'un point de vue chimique le potentiel électrique V_k^i au centre de l'élément k engendré par la charge électrique située sur l'autre élément de surface i.

En toute rigueur, le produit $C_1 q^{(n-1)}$ nécessite donc une boucle sur les N_T éléments de surfaces (boucle balayant les N_T composantes du vecteur résultat V), et pour le calcul de chaque composante V_k une boucle interne sur les N_T-1 autres éléments (équation (5.1)). Cela résulte en un calcul de dimension environ N_T^2. N_T peut être grand, cela dépend de la taille du système chimique étudié. Par exemple, le notre (1244 atomes) conduit à la construction de 8946 éléments de surface (en utilisant les paramètres par défaut de GAMESS). Dans cette double boucle, la distance d_{ki} égale à d_{ik}, est calculée deux fois : une fois pour la composante V_k et une fois pour la composante V_i du vecteur résultat. Cela a amené les auteurs de GAMESS à proposer un algorithme plus efficace du calcul de V dénommé ci-après : "parcours triangulaire de la matrice C_1".

Par ailleurs, d'un point de vue chimique, le potentiel électrique varie inversement avec la distance d_k entre deux éléments de surface. Cette distance pouvant être très grande entre deux éléments de la surface de cavitation, les auteurs de GAMESS ont donc décidé de tirer profit de cette propriété pour accélérer le calcul via une approximation décrite ci-après sous le nom "d'approximation régionale".

Approximation du calcul par régions : approximation régionale

Les auteurs de GAMESS répartissent l'ensemble des éléments de surface en plusieurs régions, comme illustré figure 5.3. Chaque région est constituée d'un ensemble de charges proches les unes des autres. Au sein d'une région A, pour l'élément k, comme les autres charges sont proches, le calcul de toutes les contributions de la région est nécessaire car générant un potentiel en k : aucune approximation n'est possible.

De même, illustré figure 5.4, lorsque deux régions (A et B) sont considérées proches, à nouveau, aucune approximation n'est faite dans le calcul du potentiel en un élément k de la région par une charge de la région B. Ainsi, pour un élément k, toutes les contributions des éléments des régions proches sont calculées.

En revanche (figure 5.5), lorsque deux régions (A et C) sont considérées éloignées, une approximation est réalisée en utilisant non pas la distance entre les éléments de surface, mais la distance entre l'élément de surface calculé appartenant à la région A et un point de référence représentant toutes les charges de la région éloignée C. Cette approximation réduit d'un ordre de grandeur le calcul entre deux régions éloignées. Cette approximation ensuite appelée "approximation régionale" se justifie par le fait qu'à longue distance les potentiels électroniques décroissent en $\frac{1}{r}$.

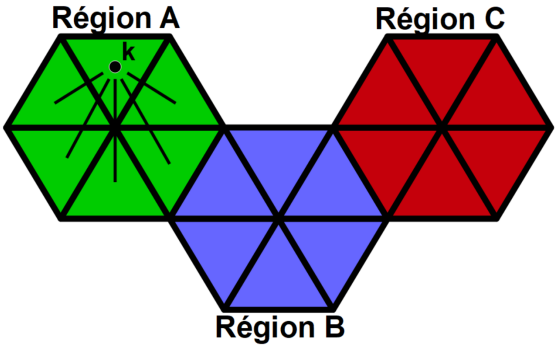


FIGURE 5.3 – Illustration des interactions générées par les charges d'une région A sur un point k de la même région.

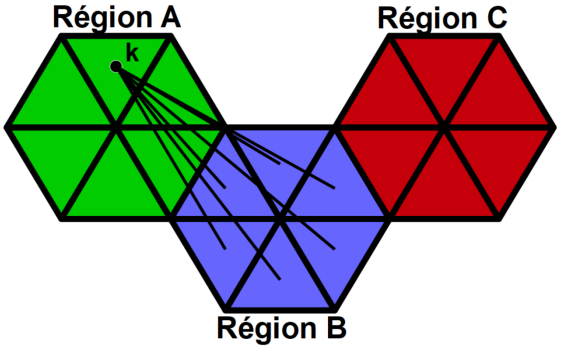


FIGURE 5.4 – Illustration des interactions générées par les charges d'une région B proche sur un point k d'une région A.

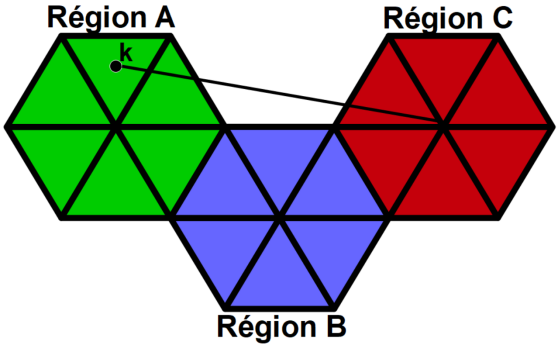


FIGURE 5.5 – Illustration des interactions générées par les charges d'une région C éloignée sur un point k de la région A.

Parcours triangulaire de la matrice C_1

Comme indiqué précédemment, en plus de l'approximation régionale précédemment décrite, une optimisation algorithmique est implémentée dans la routine `ascpot` de GAMESS pour le calcul du produit matrice-vecteur $C_1 q^{n-1}$ afin d'améliorer les performances du CPU. Cette optimisation s'applique uniquement sur les calculs internes d'une région (cas de la figure 5.3). Un parcours astucieux des charges permet de se passer de l'opération

conditionnelle ($i \neq k$) présente dans la relation 5.1. Ce parcours triangulaire astucieux permet aussi d'éviter de calculer deux fois une même distance. Ce traitement est réalisé en parcourant uniquement la partie triangulaire supérieure de la matrice C_1 traitée.

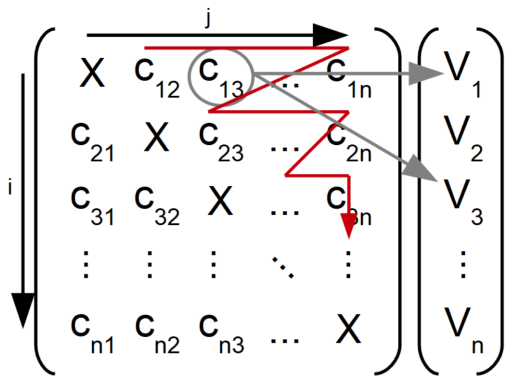


FIGURE 5.6 – Illustration du parcours triangulaire (en rouge) de la matrice pour le calcul $V = C_1 q^{(n-1)}$ dans la routine `ascpot` ; exemple (gris) de double utilisation de la distance.

Lors de ce parcours (illustré en rouge figure 5.6), chaque élément c_{ij} fait appel à la distance d_{ij} égale par symétrie à la distance d_{ji}. Le parcours triangulaire permet de tirer profit de cette symétrie en calculant simultanément deux contributions i et j qui vont alimenter deux éléments du vecteur V à chaque étape du calcul, évitant de calculer deux fois la distance d_{ij}.

Algorithme de la fonction `ascpot`

Nous allons dans cette section détailler l'algorithme implémenté dans la fonction `ascpot` au sein de GAMESS. Cet algorithme est reporté ci-dessous (algorithme 7).

Avant d'arriver dans cette routine, GAMESS a déjà construit la surface de la cavité de solvatation en N_T triangles. Ces N_T triangles ont déjà été regroupés en un certain nombre de régions r. La fonction `ascpot` débute (ligne 1) par calculer le seuil S servant à évaluer si des régions sont proches. Le seuil S est calculé en utilisant une donnée fournie par l'utilisateur.

Le premier cas (5.3) des interactions à l'intérieur d'une même région est d'abord traité. Toutes les régions (ligne 3) sont parcourues pour être traitées. Les indices des éléments de début et de fin de la région r en cours sont recueillis (lignes 4 et 5) respectivement dans LISTI et LISTIP. Deux boucles sont alors réalisées (lignes 6 et 7) sur les éléments de la région r en cours. La première boucle d'indice i parcourt tous les éléments de la région r tandis que la seconde parcourt avec j les éléments supérieurs à i de la même région r en cours, c'est le premier cas (figure 5.3). Ces deux boucles permettent le parcours efficace des éléments d'une région pour minimiser le calcul des distances (parcours triangulaire). La distance d_{ij} entre les éléments i et j (ligne 8) est calculée. À partir de cette distance d_{ij} sont calculées (lignes 9 et 10) deux contributions $\frac{q_j}{d_{ij}}$ et $\frac{q_i}{d_{ij}}$ et ajoutées respectivement aux potentiels $V(i)$ et $V(j)$. C'est le parcours optimisé de la figure 5.6 qui vient d'être réalisé.

Une fois la matrice triangulaire de la région parcourue, les coordonnées (ligne 13) de la région r en cours (x_r, y_r, z_r) sont récupérées. Une boucle d'indice s parcourt (ligne 14) toutes les régions différentes de la région r en cours. Les coordonnées (x_s, y_s, z_s) de la région s (ligne 15) sont récupérées. La distance d_{rs} (ligne 16) entre les régions r et

s est calculée. Les indices des éléments de début et de fin de la région s en cours sont recueillis (lignes 17 et 18) respectivement dans LISTJ et LISTJP. Deux traitements sont alors envisagés en fonction du test (ligne 19) $d_{rs} < S$.

Dans le cas de deux régions limitrophes (figure 5.4) ($d_{rs} < S$, lignes 20 à 27), une boucle d'indice i parcourt (ligne 20) les éléments de la région r. La variable V_r est initialisée (ligne 21) et contiendra la contribution d'une région s à l'élément V_i. Une boucle d'indice j parcourt alors (ligne 22) les éléments de la région s. La distance d_{ij} (ligne 23) entre l'élément i de la région r et l'élément j de la région s est calculée. À partir de la distance d_{ij}, la contribution $\frac{q_j}{d_{ij}}$ est calculée et ajoutée à la variable intermédiaire V_r. Une fois toutes les contributions des éléments de la région s calculées, V_r est ajouté à $V(i)$.

Nous arrivons enfin au troisième cas (approximation régionale, figure 5.5) lorsque le test $d_{rs} < S$ est invalidé (lignes 29 à 32) le calcul de la région s est approché par le traitement suivant. Une boucle d'indice i (ligne 29) parcourt les éléments de la région r. La distance d_{is} (ligne 30) entre l'élément i et la région s est calculée. À partir de la distance d_{is}, l'approximation de la région s à l'élément i est faite.

Algorithme 7 fonction *ascpot* du logiciel GAMESS

```
1: FONCTION ASCPOT
2:     Obtenir le seuil S ;
3:     POUR r parcourant les régions FAIRE
4:         Récupérer l'indice LISTI du premier élément de surface de la région r ;
5:         Récupérer l'indice LISTIP du dernier élément de surface de la région r ;
6:         POUR i allant de LISTI à LISTIP FAIRE
7:             POUR j allant de i+1 à LISTIP FAIRE
8:                 Calculer la distance d_ij ;
9:                 Calculer et ajouter au potentiel V(i) la contribution q_j/d_ij ;
10:                Calculer et ajouter au potentiel V(j) la contribution q_i/d_ij ;
11:            FIN POUR
12:        FIN POUR
13:        Récupération des coordonnées (x_r, y_r, z_r) de la région r ;
14:        POUR  s parcourant les régions avec r ≠ s FAIRE
15:            Récupération des coordonnées (x_s, y_s, z_s) de la région s ;
16:            Calculer d_rs la distance les deux régions r et s ;
17:            Récupérer l'indice LISTJ du premier élément de surface de la région s
18:            Récupérer l'indice LISTJP du dernier élément de surface de la région s
19:            SI d_rs < S ALORS
20:                POUR i parcourant les éléments de la région r FAIRE
21:                    Initialiser la variable intermédiaire V_r à 0 ;
22:                    POUR j parcourant les éléments de la région s FAIRE
23:                        Calculer la distance d_ij entre l'élément k et l ;
24:                        Ajouter à V_r la contribution q_j/d_ij ;
25:                    FIN POUR
26:                    Ajouter V_r à V(i) ;
27:                FIN POUR
28:            SINON
29:                POUR i parcourant les éléments de la région r FAIRE
30:                    Calculer la distance d_is entre l'élément i et la région (x_s, y_s, z_s) ;
31:                    Ajouter à V(i) la contribution de toute les charges de la région s
   considérée à une même distance de i ;
32:                FIN POUR
33:            FIN SI
34:        FIN POUR
35:    FIN POUR
36: FIN FONCTION
```

La parallélisation réalisée par GAMESS sur cette implémentation est obtenue en répartissant le calcul des différentes régions sur les différentes ressources CPU de calcul. Chaque région est entièrement traitée par un unique processus léger.

C'est donc en partant de ce code (fonction `ascpot`) que nous avons considéré le portage GPU d'une partie de la méthode PCM par le biais du portage GPU de la fonction `ascpot`. Comme les auteurs de GAMESS nous ont demandé d'incorporer tout travail par le biais de la bibliothèque LIBCCHEM, le portage GPU devra être installé dans cette bibliothèque, ce qui nécessite de réaliser une installation de GAMESS compilée avec la bibliothèque LIBCCHEM.

5.4 Considération préliminaire au portage GPU

Nous avons reçu de la part des auteurs de GAMESS la recommandation de faire apparaître toute nouvelle fonction dans la bibliothèque LIBCCHEM. Ce cahier des charges a pour but d'améliorer à terme la lisibilité du code GAMESS.

Les auteurs de GAMESS ont déjà porté certaines fonctionnalités quantiques sur GPU. Ces implémentations sont placées dans la bibliothèque LIBCCHEM. De manière à nous placer dans le contexte d'une exécution GAMESS utilisant une carte graphique nous avons donc réalisé une nouvelle installation de GAMESS en spécifiant l'utilisation de la bibliothèque LIBCCHEM dans laquelle nous avons écrit notre propre implémentation GPU de la fonction `ascpot`. L'objectif, ici est donc de détailler l'installation réalisée de GAMESS avec LIBCCHEM. Cette installation est compilée avec ifort et bénéficie de la bibliothèque mathématique MKL. Le modèle de parallélisation CPU lors de l'utilisation de LIBCCHEM diffère des installations précédentes (utilisant le mode *socket*). Le modèle de LIBCCHEM est plus classique, en utilisant MPI pour la gestion des différents nœuds de calcul disponibles et OpenMP pour les différents cœurs CPU.

Le tableau 5.7 ajoute donc une ligne de résultats par rapport au tableau de résultats 5.4 précédent :

Processus légers	1	2	4	8	16
socket ifort	154	94,2	63,8	51,2	43
socket ifort MKL	151	91,4	61,3	46,5	39,4
LIBCCHEM ifort MKL	153	92,2	63,2	52,9	65,1

Tableau 5.7 – Tableau des temps en secondes d'un calcul d'énergie par la méthode combinée DFTB / FMO / PCM pour 1, 2, 4, 8 et 16 processus légers de plusieurs installations de GAMESS sur notre cas test (1244 atomes).

Si seule la performance en matière de temps nous intéresse, la deuxième installation présentée (socket ifort MKL) est la plus performante. En effet, l'emploi de la bibliothèque MKL permet d'améliorer les performances cependant l'utilisation de la bibliothèque LIBCCHEM diminue les performances, significativement pour 8 et 16 cœurs.

Dans le cas où, nous souhaitons comparer dans la suite, un portage GPU à une version CPU, l'installation la plus proche en matière de code est la troisième (LIBCCHEM ifort MKL). Le paradigme de programmation d'une installation avec la bibliothèque LIBCCHEM est différent d'une installation en mode *socket*. Nous pouvons ajouter que cette bibliothèque (LIBCCHEM) oblige l'utilisation d'une bibliothèque mathématique (nous avons choisi la bibliothèque MKL) et qu'elle force aussi la bibliothèque mathématique à être utilisée en série, expliquant en partie, les moins bonnes performances.

C'est donc en partant de la base d'une installation de GAMESS avec LIBCCHEM que nous avons tenté un portage GPU de la fonction `ascpot`. L'objet de la section qui suit est ce portage GPU ainsi que les différentes optimisations qui ont été tentées.

5.5 Portages GPU de la fonction `ascpot` de GAMESS

Cette section traite donc du travail de portage GPU réalisé sur GAMESS avec pour objectif d'accélérer l'utilisation combinée de DFTB, FMO et PCM. Comme nous l'avons décrit section 5.2, nous concentrons nos efforts sur la fonction `ascpot` de GAMESS. L'algorithme de cette fonction (`ascpot`) est détaillé précédemment section 5.3.

Chronologiquement, ce travail est réalisé après celui sur NCI, qui fut une expérience significative pour moi dans le domaine de l'utilisation de GPU, permettant d'éviter un certain nombre de considérations précédemment évoquées dans le chapitre 4 (portage GPU du code NCIplot). Par exemple, nous avons immédiatement fait le choix de fixer le nombre de processus légers en fonction de la carte graphique utilisée, cela par l'usage préalable de l'occupancy calculator[54].

Portage GPU naïf de la fonction `ascpot`

La première approche fut de réaliser un portage GPU en utilisant le même paradigme de parallélisme que celui déjà implémenté par GAMESS. Nous conservons le parcours triangulaire de la matrice C_1 et l'utilisation de l'approximation régionale, chaque région étant traitée par un processus léger. Nous supposons que cette approche sera moins performante que le code CPU. En effet, dans notre exemple de taille moyenne, les 8946 charges apparentes sont réparties dans 1073 régions. Les processus légers du GPU se répartissent donc 1073 tâches, or la carte graphique K20X décrite section 2.6 est composée de 14 SMX capables chacune de gérer 2048 processus légers simultanément, soit un total de 28 672 processus légers. Nous pouvons donc supposer une carence en calculs par rapport aux ressources disponibles. La deuxième raison de s'attendre à une faible performance GPU provient des désynchronisations (entre les processus légers d'un même *warp*) produites par les régions. En effet, les régions de la cavité moléculaire possèdent un nombre différent d'éléments de surface : d'une région à l'autre le parcours des contributions est donc différent (donc d'un processus léger à l'autre).

Cette version d'essai GPU a permis la mise en place des appels de fonction qui sont communs aux portages GPU, comme les transferts de données d'entrées (positions et valeurs des charges) et de sorties (potentiel de l'itération k : q^k) entre le CPU et le GPU. Rappelons que cette étape normalement triviale est complexifiée dans GAMESS par la grandeur du programme et surtout le fait de l'utilisation combinée de plusieurs langages de programmation (Fortran et C).

Cette étape m'a semblé être nécessaire pour simplifier la réflexion qui va suivre pour obtenir une version adaptée au GPU me permettant ensuite de concentrer mes efforts sur l'algorithme exécuté sur le GPU.

Les maigres performances obtenues se révèlent confirmées par un temps d'exécution total de 156 s (contre 39 s pour le meilleur temps CPU sur 16 cœurs).

C'est de l'hypothèse de carence précédemment évoquée dans cette sous-section, qu'un nouveau portage GPU adapté a été réalisé. Il est présenté dans la sous-section suivante.

Portage GPU de la fonction `ascpot`

Afin d'augmenter le nombre de processus légers actifs au moment du calcul du produit matriciel $C_1 q^{n-1}$ nous avons eu l'idée de supprimer l'approximation régionale, et donc le regroupement des éléments de surface en régions. Cette approximation, efficace sur CPU car permettant de diminuer le nombre de calculs à réaliser, met, possiblement, en carence le GPU sur notre exemple. Supprimer les régions permet d'augmenter le nombre potentiel de processus légers à calculer simultanément tout en supprimant une légère approximation. Cela peut à première vue être contreproductif, mais l'efficacité d'un algorithme sur GPU ne suit pas les mêmes règles que sur CPU.

L'idée est donc de faire traiter un élément de surface par un processus léger. Autrement dit, un processus léger va se charger du calcul d'une des composantes V_k du vecteur V

(équation 5.1). Nous pouvons alors aisément voir qu'il semble difficile de tirer profit du parcours triangulaire astucieux implémenté par GAMESS permettant que le calcul d'une distance soit attribué à deux contributions simultanément.

Nous nous retrouvons dans notre exemple avec 8 946 éléments de surface de la cavité moléculaire à répartir sur le GPU. Comme évoqué dans la section précédente, une carte graphique K20X peut gérer sur ses différentes SMX un total de 28 672 processus légers. Nous pouvons donc anticiper que l'accélération restera encore en dessous du maximum possible pour ce système test.

L'algorithme 8 ci-dessous illustre le contenu du noyau permettant de porter sur GPU la fonction `ascpot` de GAMESS.

Algorithme 8 *kernel* portant la fonction *ascpot* du logiciel GAMESS

```
1: FONCTION ASCPOT DEVICE
2:     POUR i parcourant les éléments de surface t à traiter FAIRE
3:         Initialiser en registre la variable V_t à 0 pour le triangle t ;
4:         POUR j parcourant toutes les charges avec i ≠ j FAIRE
5:             Calculer la distance d_ij entre les éléments i et j ;
6:             Calculer la contribution q_j/d_ij de l'élément j et l'ajouter à V_t ;
7:         FIN POUR
8:         Ajouter à V(i) le contenu de la variable V_t ;
9:     FIN POUR
10: FIN FONCTION
```

Par la suppression des régions et du parcours triangulaire de la matrice C_1 de la cavité moléculaire, le code apparaît plus concis que précédemment. Les temps obtenus par ce portage vont maintenant être détaillés dans la section suivante. Le choix initial a été de conserver l'utilisation de nombres double précision par rapport au code quantique de GAMESS.

Résultats du portage GPU de la fonction `ascpot`

Pour comparer les résultats obtenus par cette première version du code GPU, les deux premières lignes du tableau 5.8 reprennent deux installations CPU précédentes de GAMESS. La première installation (socket, ifort avec MKL) est la plus performante en matière de temps en s'exécutant sur 16 cœurs CPU en 39 s cependant la deuxième installation (LIBCCHEM, ifort avec MKL) se rapproche plus (avec 52,9 s) de l'installation du portage GPU (54,2 s). La troisième ligne du tableau 5.8 contient les temps du portage en utilisant une seule carte graphique K20X. Cette installation est réalisée avec LIBCCHEM, ifort et MKL. À noter que nous avons reporté les temps obtenus avec des nombres de cœurs différents, même pour la version GPU. En effet, rappelons qu'une seule partie du calcul (fonction `ascpot`) est portée sur GPU mais que d'autres parties du code tirent profit des ressources CPU, comme par exemple le calcul quantique DFTB (procédure SCF).

Processus légers	1	2	4	8	16
Socket ifort MKL	151	91,4	61,3	46,5	39,4
LIBCCHEM ifort MKL	153	92,2	63,2	52,9	65,1
Portage GPU (une K20X)	55,2	54,2	54,5	59,6	75,3

Tableau 5.8 – Temps en secondes d'un calcul d'énergie par l'approche combinée DFTB / FMO / PCM pour 1, 2, 4, 8 et 16 processus légers obtenus pour plusieurs installations de GAMESS sur notre cas test (1244 atomes) ainsi que pour le portage GPU.

Nous constatons que la première installation CPU (socket, ifort avec MKL), est la plus rapide avec un temps pour 16 processus légers de 39 s tandis que le portage GPU conduit à un temps de 54 s au mieux. Gardons à l'esprit que la première installation approxime (par le fractionnement en régions) les contributions considérées éloignées, ce que ne fait pas le portage GPU.

En comparant le portage GPU à la deuxième installation (LIBCCHEM, ifort avec MKL), nous voyons que les meilleurs temps sont similaires, 54 s et 53 s respectivement pour notre version GPU et la version CPU(LIBCCHEM, ifort avec MKL).

Ayant la possibilité d'utiliser deux cartes graphiques K20X sur les nœuds de calcul utilisés, j'ai modifié le code pour pouvoir tirer partie de cette puissance supplémentaire disponible. L'utilisation de deux GPU se fait dans ce cas en répartissant le calcul des éléments de surface de la cavité moléculaire sur les deux GPU, et en dupliquant sur les deux GPU les données d'entrées (positions et valeurs des charges).

Processus légers	1	2	4	8	16
socket ifort MKL	151	91,4	61,3	46,5	39,4
LIBCCHEM ifort MKL	153	92,2	63,2	52,9	65,1
Portage GPU avec une K20X	55,2	54,2	54,5	59,6	75,3
Portage GPU avec deux K20X	46,7	45,6	46,1	51,2	67

Tableau 5.9 – Tableau des temps en secondes d'un calcul d'énergie par l'approche combinée DFTB / FMO / PCM pour 1, 2, 4, 8 et 16 processus légers de deux installations CPU de GAMESS ainsi que le portage GPU avec une ou deux cartes graphiques K20X sur notre cas test (1244 atomes)

L'utilisation de deux GPU permet d'améliorer encore les résultats comme nous pouvons le voir dans le tableau 5.9, peu importe le nombre de processus légers CPU utilisés. L'utilisation de deux GPU permet, en général, de gagner environ 8 s sur le temps total d'exécution. Même si ce gain est faible, cela permet ainsi au portage GPU d'être plus rapide que la deuxième installation de GAMESS (LIBCCHEM, ifort avec MKL) en s'exécutant en 46 s contre 53 s. Cependant en comparaison de la première installation CPU (39 s) n'utilisant pas la bibliothèque LIBCCHEM, le portage avec deux GPU reste moins efficace.

Variantes du portage GPU

Deux modifications du code GPU ont été considérées pour améliorer ses performances. Tout d'abord, nous avons tenté l'utilisation de réels simple précision (ce qui accroît le nombre de cœurs de calcul disponibles) à la place de double précision. Cette approche a

été abandonnée car la précision des résultats quantiques est impactée, amenant l'absence de convergence de la méthode PCM dans le cadre de notre exemple.

En revanche, une utilisation de la mémoire partagée adaptée permet d'améliorer les performances, en passant de 54 s à 44 s avec une seule carte graphique K20X et de 46 s à 41 s avec deux cartes graphiques K20X. Cette optimisation est implémentée par *blocking* en mettant les coordonnées et la valeur des charges apparentes en mémoire partagée. Le tableau 5.10 ajoute donc deux lignes avec les temps totaux des exécutions pour une ou deux cartes graphiques avec *blocking* avec 1, 2, 4, 8 et 16 cœurs de calcul CPU.

Processus légers	1	2	4	8	16
socket ifort MKL	151	91,4	61,3	46,5	39,4
LIBCCHEM ifort MKL	153	92,2	63,2	52,9	65,1
Portage GPU avec une K20X	55,2	54,2	54,5	59,6	75,3
Portage GPU avec deux K20X	46,7	45,6	46,1	51,2	67
Portage GPU (*blocking*) avec une K20X	45,9	44,4	44,8	49,9	65,4
Portage GPU (*blocking*) avec deux K20X	41,8	41,4	40,9	46,0	61,8

Tableau 5.10 – Tableau des temps en secondes d'un calcul d'énergie par l'approche combinée DFTB / FMO / PCM pour 1, 2, 4, 8 et 16 processus légers de deux installations CPU de GAMESS ainsi que les portages GPU (avec ou sans *blocking*) avec une ou deux cartes graphiques K20X sur notre cas test (1244 atomes)

Le *blocking* est une technique utilisée dans notre cadre pour permettre un accès efficace à la mémoire partagée. Cette technique consiste à utiliser manuellement la mémoire partagée comme un cache afin de diminuer le nombre d'accès à la mémoire globale lorsque cette dernière est utilisée par plusieurs processus légers d'un même bloc à des étapes différées. Dans notre cas, nous avons implémenté le noyau avec 128 processus légers par bloc.

Cette implémentation est décrite dans l'algorithme 9. Le noyau débute (ligne 2) par une boucle d'indice i qui parcourt tous les éléments à calculer pour le processus léger courant. Le registre intermédiaire V_t est initialisé (ligne 3) à 0. Une deuxième boucle d'indice j parcourt (ligne 4) les éléments modulo 128, soit les éléments 0, 128, 256... Car dans notre cas le *blocking* est réalisé avec 128 processus légers par bloc. Afin d'éviter d'obtenir un résultat possiblement faux, les processus légers sont synchronisés (ligne 5). Chaque processus léger met (ligne 6) en mémoire partagée depuis la mémoire globale les coordonnées et la charge de l'élément j+IDThread. Les processus légers sont synchronisés (ligne 7) afin que la mémoire partagée soit bien remplie avant la suite du traitement. Une fois que la mémoire partagée est chargée avec les coordonnées et la charge des 128 éléments ces derniers sont accessibles par tous les processus légers du bloc permettant de diminuer les accès à la mémoire globale, ce qui constitue le *blocking*. Les calculs peuvent maintenant être réalisés à partir de la mémoire partagée à la place de la mémoire globale. Une boucle d'indice k est alors réalisée (ligne 8) qui parcourt les 128 éléments stockés en mémoire partagée afin de calculer chaque contribution. La distance d_{ik} (ligne 9) entre l'élément i traité (stocké en mémoire registre) et l'élément k (stocké en mémoire partagée) est calculée. Cette distance d_{ik} est utilisée (ligne 10) pour calculer la contribution $\frac{q_k}{d_{ik}}$ et l'ajouter à V_t. Enfin, une fois V_t calculé avec toutes les contributions, V_t est mis dans V(i).

Algorithme 9 Noyau portant la fonction *ascpot* du logiciel GAMESS avec *blocking*

```
1: FONCTION ASCPOT DEVICE BLOCKING
2:     POUR i parcourant les éléments de surface à traiter FAIRE
3:         Initialiser en registre la variable V_t à 0 ;
4:         POUR j parcourant les éléments 0, 128, 256... FAIRE
5:             Synchronisation des processus légers du bloc ;
6:             Mise en mémoire partagée des coordonnées et de la charge de l'élément
   j+IDThread à la coordonnée IDThread du tableau correspondant ;
7:             Synchronisation des processus légers du bloc ;
8:             POUR k parcourant 0 à 127 FAIRE
9:                 Calculer la distance d_ik entre l'élément i et l'élément k stocké en mémoire
   partagée ;
10:                Calculer la contribution q_k/d_ik de l'élément k stocké en mémoire par-
   tagée et l'ajouter à V_t ;
11:            FIN POUR
12:        FIN POUR
13:        Ajouter à V(i) le contenu de la variable V_t ;
14:    FIN POUR
15: FIN FONCTION
```

Cette dernière version GPU (40,9 s) rejoint la performance de la meilleure version (39,4 s) que nous avons de GAMESS en CPU. Il est important de noter que notre version GPU la plus optimisée ne fait aucune approximation régionale dans le calcul du produit matrice-vecteur $C_1 q^{n-1}$ pour estimer l'effet du solvant. Elle réalise donc beaucoup plus d'opérations que la version CPU la plus efficace (sans LIBCCHEM) qui se sert de l'approximation des régions pour réduire le nombre de calculs (comme décrit section 5.3).

De plus, la comparaison CPU/GPU s'appuie jusqu'ici sur le temps total d'exécution. Or, seule une partie du code (la fonction `ascpot`) a été portée sur GPU. Pour une comparaison plus juste, nous allons maintenant comparer les temps d'exécution CPU/GPU de la fonction `ascpot` seulement.

5.6 Analyse de la performance sur le temps d'exécution de la fonction `ascpot`

Nous avons remarqué que lors d'un calcul d'énergie par l'approche combinée DFTB / FMO / PCM sur 1244 atomes, la fonction `ascpot` est appelée 343 fois. Nous allons concentrer nos efforts dans cette section sur le temps d'exécution de la fonction `ascpot`. L'objectif est d'évaluer le portage GPU le plus performant réalisé actuellement. Le portage GPU évalué est donc celui avec *blocking*. Nous allons comparer les temps de ce portage GPU pour une ou deux cartes K20X avec la version CPU la plus efficace (socket-intel-mkl). Dans un premier temps nous allons nous intéresser à l'extensibilité forte du code CPU en fonction du nombre de cœurs de calcul utilisés pour l'exécution de la fonction `ascpot` toute seule.

Extensibilité forte du code CPU en fonction du nombre de cœurs de calcul

Nous avons déjà précédemment étudié l'extensibilité du code CPU de GAMESS en reportant le temps d'exécution total. Dans cette section nous regardons l'extensibilité de la fonction `ascpot` en fonction du nombre de processeurs utilisés. La figure 5.7 illustre l'extensibilité forte du code existant. Nous pouvons voir sur notre exemple (calcul d'énergie DFTB / FMO / PCM sur 1244 atomes et 8946 éléments de surface de cavité de solvatation) que le temps d'exécution suit une progression linéaire en fonction du nombre de cœurs de calcul avec un facteur 15 d'accélération lorsque 16 processeurs sont utilisés.

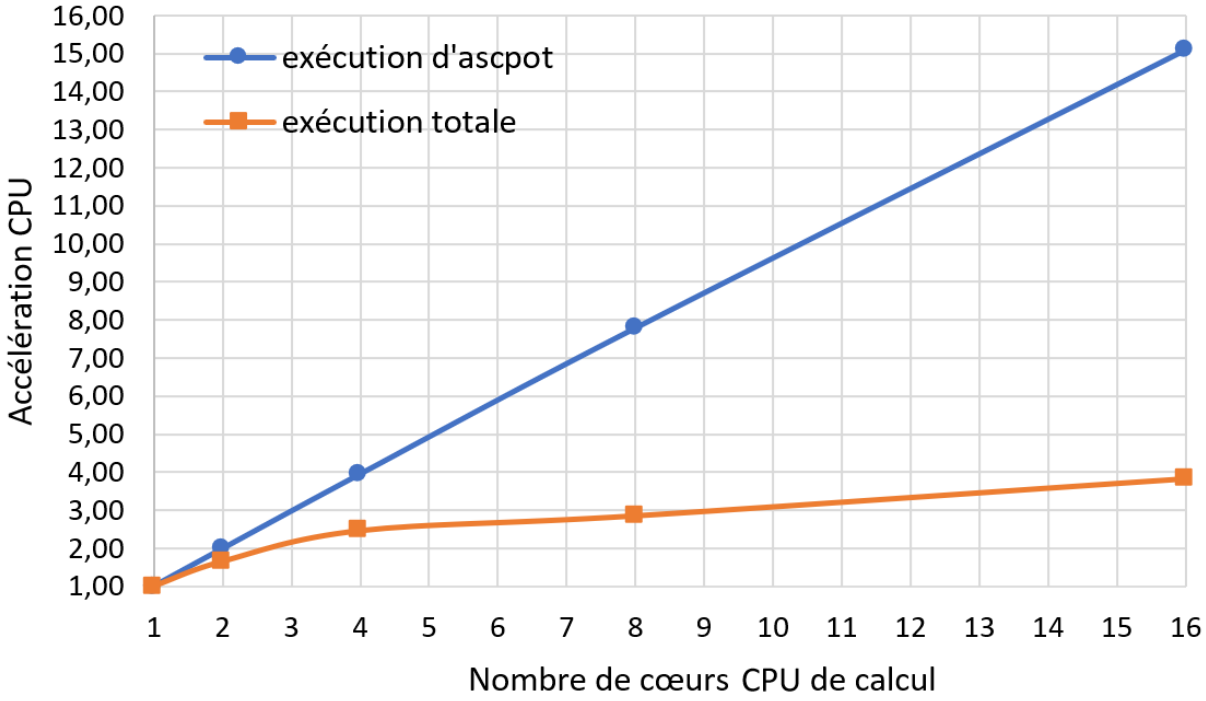


FIGURE 5.7 – Extensibilité de la fonction `ascpot` en fonction du nombre de cœurs CPU de calcul

Le code actuel implémenté dans GAMESS permet donc de tirer correctement profit des ressources parallèles disponibles pour la fonction `ascpot`. Ce constat sur extensibilité de la fonction `ascpot` est rassurant par rapport à l'extensibilité du temps d'exécution total.

Nous pouvons donc prendre comme référence pour évaluer le portage GPU les temps de calcul de l'installation socket-intel-mkl avec 16 cœurs. Gardons tout de même en tête plusieurs différences :

- Le code CPU utilise des régions pour accélérer le calcul de la fonction `ascpot`, ce que ne fait pas le portage GPU.
- Le code CPU utilise l'installation en mode *socket* tandis que le portage GPU utilise la bibliothèque LIBCCHEM.

Ces différences peuvent impacter l'évaluation de la performance du portage GPU.

Comparaison des temps de calcul de la fonction `ascpot` du portage GPU (*blocking*) à l'installation CPU (*socket-intel-mkl*)

Dans le but de simplifier le propos, nous allons nous concentrer ici sur le portage GPU avec *blocking*. Nous allons comparer les temps de ce portage GPU avec deux versions CPU provenant de l'installation socket-intel-mkl, l'une avec l'optimisation des régions et l'autre sans. En effet au même titre qu'il est important d'avoir une référence aussi optimisée que possible, il est aussi important pour être honnête de comparer des algorithmes similaires. Dans notre cas le portage GPU ne tire pas profit de l'approximation des régions donc avoir une référence CPU qui ne tire pas profit de cette approximation des régions est intéressant pour analyser les performances du portage GPU.

La fonction `ascpot` est appelée plusieurs fois au cours d'une exécution. Pour évaluer ses performances deux valeurs sont récupérées :

- La somme de l'ensemble des temps d'exécution des appels à la fonction `ascpot`.
- La moyenne de l'ensemble des temps d'exécution des appels à la fonction `ascpot`.

Afin d'évaluer le temps d'exécution passé dans la fonction `ascpot` trois exemples sont considérés. La description de la surface de la cavité de solvatation avec plus de triangles est possible via une option de GAMESS. Chaque exemple possède un certain nombre d'éléments de surface, ici : 8 964, 35 694 et 142 795. L'intérêt d'augmenter la taille du problème est d'évaluer comment se comporte le matériel en fonction de la quantité de données à traiter (l'extensibilité faible du chapitre 1).

Utilisation d'une seule carte graphique K20X

Les sommes des temps de la fonction `ascpot` sont illustrées dans la figure 5.8. Trois temps sont reportés par exemple. Pour le premier exemple (8 946 éléments) le temps CPU avec l'approximation régionale est le plus court (7,9 s), puis le code CPU sans l'approximation régionale (10,1 s) et enfin le portage GPU (11,8 s). Le deuxième exemple (35 694 éléments) produit des temps ordonnés de manière similaire au premier exemple. Quant au troisième exemple (142 795 éléments) le portage GPU (3 441 s) est plus rapide que le code CPU sans l'approximation des régions (3 686 s) mais le portage GPU reste plus lent que le code CPU avec l'approximation des régions (2 055 s).

Nous pouvons constater qu'avec une carte graphique K20X, l'installation CPU (socket-intel-mkl) reste plus performante sur tous les exemples. L'utilisation d'une carte graphique K20X ne permet donc pas d'accélérer les performances par rapport au code avec l'approximation des régions. En revanche, si l'on compare l'utilisation d'une seule carte graphique au code CPU sans approximation, nous pouvons constater que les performances sont similaires.

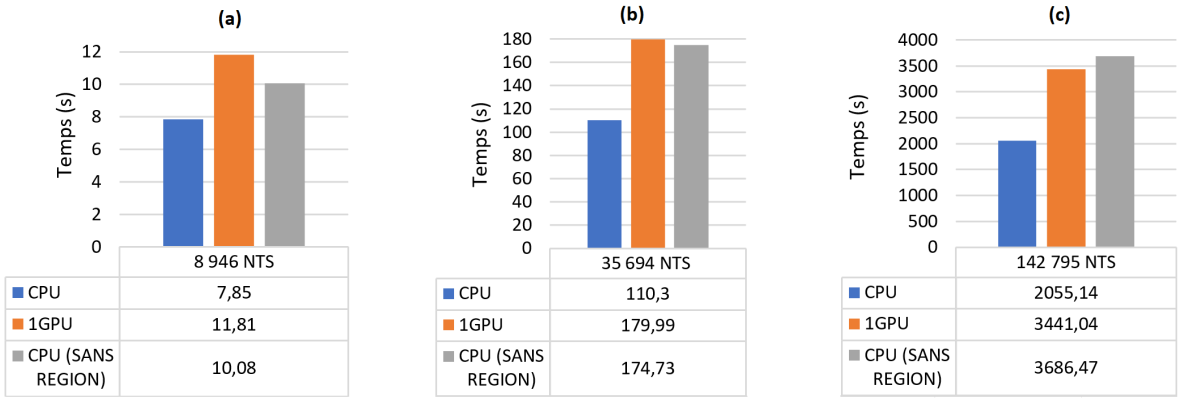


FIGURE 5.8 – Temps d'exécution de la fonction `ascpot` d'un calcul d'énergie DFTB / FMO / PCM sur un système comportant 1244 atomes et trois maillages différents pour la cavité de solvatation : 8 946 (a), 35 694 (b) et 142 795 (c) triangles. Installations CPU (socket-intel-mkl) avec ou sans l'approximation des régions et le portage GPU (*blocking*) sur une carte graphique K20X.

La performance décrite précédemment sur les sommes des temps de la fonction `ascpot` pour une carte graphique K20X se retrouve (figure 5.9) lorsque nous regardons le temps moyen des exécutions rencontrées sur les trois instances : 8 946, 35 694 et 142 795 éléments. Le code CPU avec approximation s'exécute plus rapidement sur ces trois instances, avec par exemple, un temps d'exécution moyen de la fonction `ascpot` de 4,14 s pour le code CPU (socket-intel-mkl) avec l'approximation des régions et de 6,91 s pour le portage GPU (*blocking*) sur l'exemple avec 142 795 éléments tandis que le code CPU sans approximation exécute en moyenne en 7,40 s les appels à la fonction `ascpot`.

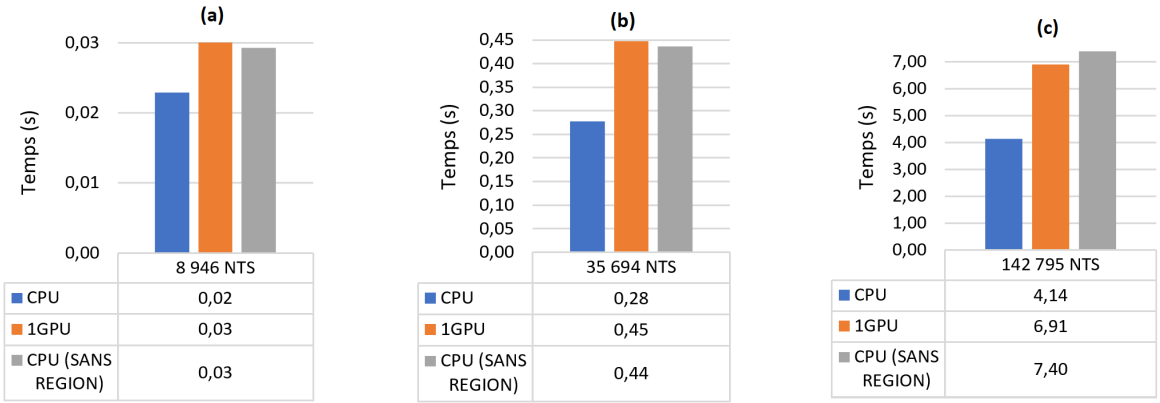


FIGURE 5.9 – Moyennes des temps d'exécutions de la fonction `ascpot` d'un calcul DFTB / FMO / PCM sur un système comportant 1244 atomes et trois maillages différents pour la cavité de solvatation : 8 946 (a), 35 694 (b) et 142 795 (c) triangles. Installations CPU (socket-intel-mkl) avec ou sans l'approximation des régions et le portage GPU (*blocking*) sur une carte graphique K20X.

L'algorithme porté est itératif et utilise des variables flottantes double précision ce qui n'est pas un environnement favorable à l'utilisation de cartes graphiques. Nous pouvons cependant constater qu'à approximation régionale égale, les temps d'exécution du portage GPU et de l'implémentation CPU sont similaires.

Maintenant que nous avons vu les performances du portage GPU (*blocking*) avec une carte graphique sur la partie portée (fonction `ascpot`) du logiciel GAMESS, regardons les performances lors de l'utilisation de deux cartes graphiques K20X.

Utilisation de deux cartes graphiques K20X

Pour utiliser deux cartes graphiques, les données à calculer sont réparties sur les deux GPU simplement en découpant en deux le vecteur de données à calculer et en dupliquant les données d'entrées (coordonnées et charges des éléments) sur les GPU.

Dans ce cadre, pour le premier exemple avec 8 946 éléments, les performances de l'installation CPU avec l'approximation régionale et du portage GPU (*blocking*) sont similaires avec 8 s pour la somme des temps de la fonction `ascpot`. L'installation CPU sans l'approximation des régions prend elle plus de temps : 10 s.

Pour les deux autres exemples (35 694 et 142 795 éléments) le code CPU sans l'approximation régionale reste le code le plus lent (respectivement 175 s et 3 687 s) puis vient le code CPU avec l'approximation (respectivement 110 s et 2 055 s) et enfin le plus rapide est le portage GPU (93 s et 1 718 s) utilisant deux cartes graphiques K20X.

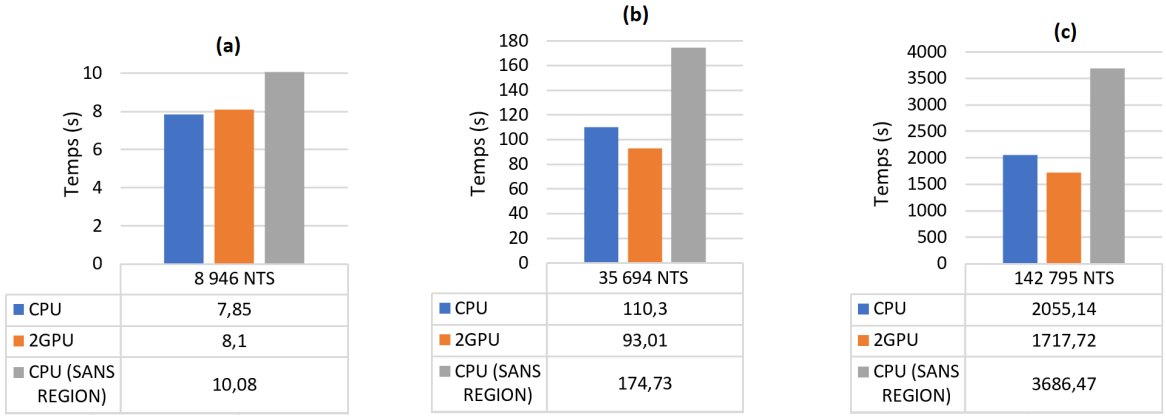


FIGURE 5.10 – Temps d'exécution de la fonction `ascpot` d'un calcul d'énergie DFTB / FMO / PCM sur un système comportant 1244 atomes et trois maillages différents pour la cavité de solvatation : 8 946 (a), 35 694 (b) et 142 795 (c) triangles. Installations CPU (socket-intel-mkl) avec ou sans l'approximation des régions et le portage GPU (*blocking*) sur deux cartes graphiques K20X.

Nous pouvons voir que l'utilisation de deux GPU permet dans deux de nos exemples d'améliorer significativement la performance en comparaison au code CPU socket-intel-mkl, sans l'approximation des régions. L'augmentation de la quantité de cartes graphiques est un des axes de développement des centres de calcul à ce jour. Nous pouvons donc constater par le biais de cette section 5.6 que ce type de fonction (fonctions similaires à `ascpot`) peut tirer profit de l'évolution des architectures en cours.

Nous allons dans la section suivante regarder la performance énergétique des différents codes afin de pouvoir conclure l'analyse du portage GPU réalisé sur le logiciel GAMESS.

5.7 Performance énergétique du portage GPU

Nous nous intéressons dans cette section à la consommation énergétique totale de trois codes :

- L'installation CPU (socket-intel-mkl) avec l'approximation régionale.
- L'installation CPU (socket-intel-mkl) sans l'approximation régionale.
- Le portage GPU (*blocking*), sans l'approximation régionale par définition.

La consommation énergétique du portage GPU est obtenue avec une ou deux cartes graphiques K20X et en prenant ou non en compte la consommation de l'hôte. En revanche, la consommation des codes CPU est obtenue en sommant la consommation des 16 cœurs de calcul. Les mesures de consommation sont réalisées par un logiciel tiers (nommé POWMON, développé par le centre de calcul ROMEO) qui utilise les bibliothèques Intel RAPL (*Running Average Power Limit*) et NVML (*NVIDIA Management Library*). Le mode de fonctionnement de cet outil utilise un serveur client pour éviter d'influencer les performances du code analysé et retourne directement les valeurs obtenues par les bibliothèques, en Joules pour RAPL et en milliwatt pour NVML (la fonction puissance a alors été intégrée sur le temps d'exécution pour obtenir la consommation en Joules). Les performances sont collectées à une fréquence de 10Hz pour le CPU et le GPU.

L'ensemble des consommations est représenté dans la figure 5.11.

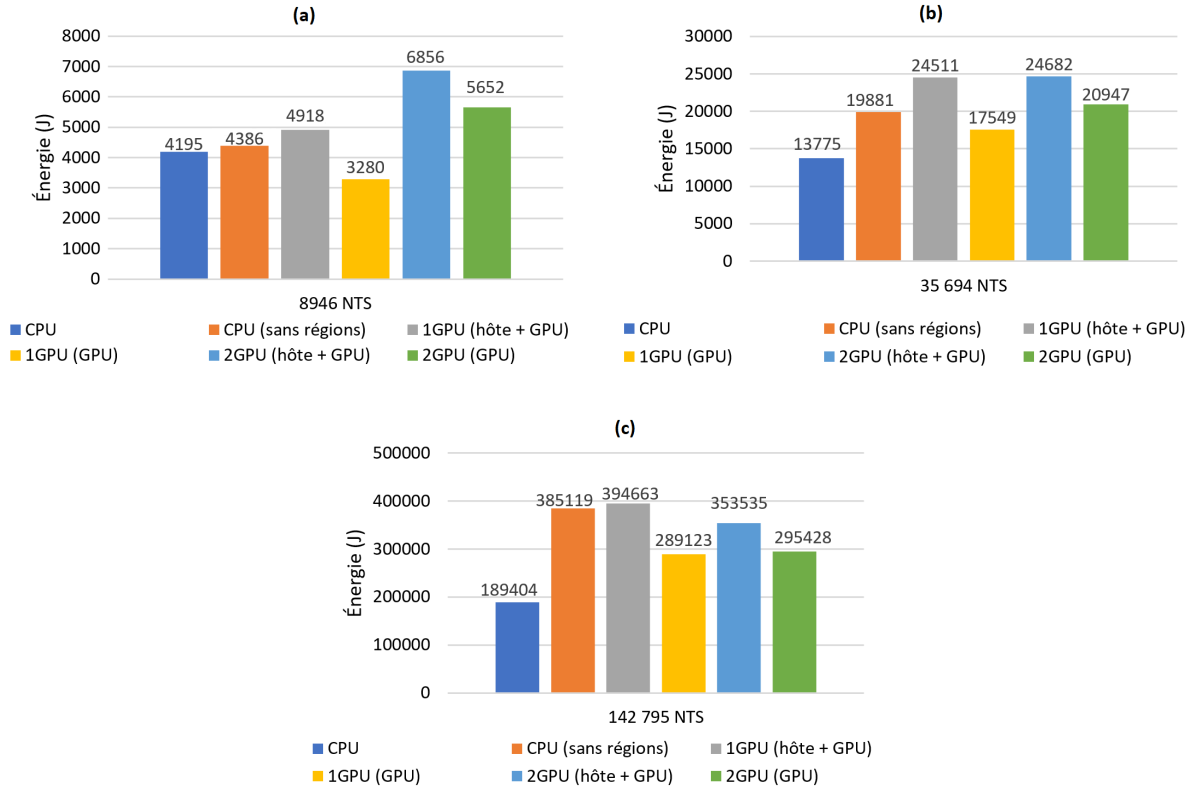


FIGURE 5.11 – Consommations énergétiques (J) de l'exécution totale pour les deux installations CPU (socket-intel-mkl avec et sans régions) et le portage GPU (*blocking*) sur une ou deux cartes graphiques K20X (avec ou sans la consommation de l'hôte). Calcul d'énergie DFTB / FMO / PCM sur 1244 atomes et trois maillages : 8 946 (a), 35694 (b) et 142 795 (c) triangles.

Premier constat naturel que nous pouvons faire, la prise en compte de la consommation de l'hôte augmente la consommation totale, nous pouvons par exemple faire ce constat dans le deuxième exemple (b) : lors de l'utilisation d'une carte graphique K20X la consommation sans l'hôte est de 17 549 J contre 24 511 J avec la consommation de l'hôte.

La consommation énergétique dans l'exemple (a) est supérieure pour deux cartes graphiques K20X que pour une carte graphique K20X. En revanche, pour les exemples (b) et (c) la consommation est similaire pour une ou deux cartes graphiques.

Dans tous les cas, si la consommation de l'hôte est prise en compte, l'implémentation CPU avec approximation est plus efficiente. Nous pouvons par exemple regarder le deuxième exemple (35 964 éléments) où le code CPU avec approximation consomme 13 775 J pour s'exécuter tandis que le portage GPU consomme (avec l'hôte) 24 511 J avec une carte graphique K20X et 24 682 J avec deux cartes graphiques K20X.

L'utilisation de GPU dans ce cadre quantique particulier ne semble donc pas permettre d'obtenir de gains sur la consommation énergétique de l'exécution du code. L'accélération par l'utilisation de cartes graphiques n'est pas suffisamment importante pour tirer pleinement profit du matériel à disposition : cela entrave la performance énergétique.

5.8 Bilan et perspectives du travail réalisé sur GAMESS

Par le biais de cette partie de mon travail de recherche, plusieurs points peuvent être mis en exergue.

Tout d'abord, la mise en œuvre des GPU dans un code complexe par sa taille, la diversité des auteurs et l'usage cumulé de langages peut être difficile. En effet tous ces éléments augmentent la quantité de connaissances et le temps nécessaire pour modifier un tel code.

Ajoutons qu'obtenir une référence CPU clairement comparable est complexe car les architectures CPU et GPU sont si différentes que des optimisations pour l'une peuvent être contreproductives pour l'autre. Dans notre cas, l'optimisation régionale implémentée par GAMESS se prête mal à l'utilisation des GPU. Sa suppression permet d'améliorer les performances du portage GPU comme nous avons pu le constater au cours de ce chapitre 5 (section 5.5).

L'obtention d'une accélération par le biais de GPU n'est pas toujours garantie. Cela dépend du caractère indépendant des tâches à réaliser au cours de l'algorithme porté, mais aussi de la taille du problème. En effet, comme nous avons pu le voir dans certains cas de figure, l'utilisation des GPU ne permet pas d'obtenir une accélération et l'absence de cette accélération empêche la diminution de la consommation énergétique. Dans le cadre de GAMESS, la difficulté provient du caractère itératif de l'algorithme porté. La fonction `ascpot` n'est pas constituée d'un grand nombre de calculs à réaliser contrairement au travail de portage GPU réalisé dans le chapitre 4 sur l'approche NCI. Il est aussi important de constater que nous retrouvons dans le travail réalisé dans ce chapitre 5 la limitation provenant de la loi d'Amdahl (décrite section 1.4), en effet la fonction `ascpot` ne prend dans notre analyse qu'une portion (entre 22% et 79%) du temps total d'exécution de notre exemple de taille fixe. Donc l'accélération maximale que nous pouvons obtenir est théoriquement limitée (d'après la loi d'Amdahl).

Nous pouvons aussi constater avec ce chapitre 5 qu'une accélération peut être obtenue lorsque plusieurs cartes graphiques peuvent être utilisées. L'ajout de cartes graphiques pose un autre problème d'ordre financier, cependant dans notre cas, la machine ROMEO tend dans ce sens en augmentant la quantité de cartes graphiques disponibles dans la nouvelle génération du calculateur. En effet la nouvelle machine met à disposition quatre cartes graphiques P100 par nœud de calcul contre deux cartes graphiques K20X par nœud de calcul sur l'ancienne machine que nous avons utilisé pour cette thèse. Nous pouvons

aussi constater la tendance plus générale qui va dans ce sens. Par exemple le premier superordinateur au TOP500 de juin 2018 est la machine hybride Summit dont chaque nœud est constitué de six cartes graphiques NVIDIA V100. Une autre solution fournit par NVIDIA est le DGX, la première génération (DGX-1) permet d'utiliser 8 cartes graphiques tandis que la deuxième génération (DGX-2) permet d'utiliser 16 cartes graphiques. Nous pouvons donc voir que l'utilisation de plusieurs cartes graphiques au sein d'un même nœud de calcul s'annonce comme étant une solution fréquente dans les prochaines années. Cette tendance vient soutenir le travail qui a été réalisé dans cette thèse en motivant le fait que la chimie théorique doit être capable de s'adapter aux architectures actuelles.

En conclusion, l'utilisation des GPU peut être ardue et inadaptée dans certaines applications. Il est souvent nécessaire que le développeur sache tirer profit de l'architecture qu'il a à sa disposition en étant capable d'analyser le code existant et le cas échéant, d'adapter la méthode de résolution utilisée à l'architecture utilisée. Nous avons dans ce chapitre 5 un exemple de code où il semble complexe d'obtenir une accélération significative pour les cas qui nous intéressent sans quitter le cadre existant. C'est pour cela que nous avons dans un premier temps supprimé les approximations existantes dans le code CPU qui se révélaient inefficaces (dans l'état actuel) sur GPU.

La première perspective que nous pouvons donc voir serait d'adapter l'approximation régionale (décrite section 5.3) de GAMESS (CPU) présent dans la fonction `ascpot` (méthode PCM : effet de solvant) afin que cette approximation soit performante sur GPU. Pour faire cela il faudrait par exemple équilibrer la quantité de triangles dans chaque région afin que le traitement puisse être similaire pour chaque processus léger du GPU. La limitation que nous avons eue dans notre cas fut que notre exemple était constitué de trop peu de triangles (8 946) par rapport aux cartes graphiques utilisées pour se permettre de partitionner le calcul dans des régions. Dans le cadre où l'approche nécessiterait plus de triangles, une approximation régionale sur GPU peut se révéler efficace.

Dans le cadre de ce chapitre 5 nous nous sommes cantonnés à rester proche de l'algorithme implémenté sur CPU et nous avons pu constater une accélération lors de l'utilisation de deux cartes graphiques K20X. Dans ce travail notre portage GPU tire profit uniquement de la puissance disponible sur le GPU. Dans ce cadre, il peut être intéressant d'envisager comme perspective de répartir le calcul des charges du solvant à la fois sur les ressources GPU et sur le CPU hôte afin de maximiser la puissance de calcul utilisée. Cela soulèvera la question de la répartition de la charge de calcul entre CPU et GPU mais peut permettre d'accélérer le calcul.

Toujours le cadre de la méthode PCM (effet de solvant), une autre perspective intéressante serait de résoudre l'équation Cq=g (équation (3.30)) non pas en portant le code CPU (de GAMESS) sur GPU mais en utilisant une méthode adaptée à l'utilisation de GPU. Par exemple en utilisant une décomposition QR pour résoudre ce système ou bien en inversant la matrice C par le biais de décompositions LU[60]. En effet, l'utilisation d'une méthode plus adaptée aux GPU peut potentiellement permettre d'obtenir de meilleurs performances.

Conclusion générale

Au cours de ce travail de recherche nous nous sommes intéressés à l'utilisation de l'architecture *manycore* des cartes graphiques dans le cadre de la chimie théorique. Nous avons aussi pu voir au cours du premier chapitre que l'évolution des architectures informatiques provient des contraintes physiques fortes qui font passer l'augmentation de la puissance de calcul par une augmentation du nombre de cœurs de calcul. Ce choix d'utiliser les cartes graphiques est justifié dans le premier chapitre par l'évolution des architectures HPC vers ce type d'architectures *manycore* ainsi que la disponibilité dans le cadre de cette thèse de cartes graphiques K20X sur le centre de calcul ROMEO. Le domaine de la chimie théorique doit selon nous être capable de tirer profit de cette évolution. L'objet de cette thèse est donc d'apporter une contribution en portant deux exemples d'applications de la chimie théorique sur GPU afin de montrer la faisabilité ainsi que les limites d'une telle démarche.

Dans notre cas, nous nous sommes concentrés sur des approches pouvant à moyen terme être utilisées dans le logiciel de *docking* moléculaire AlgoGen développé à Reims par la collaboration des laboratoires ICMR et CReSTIC.

Le premier travail de portage GPU sur l'approche NCI (Non-covalente interactions) décrit dans le chapitre 4 permet d'avoir un cas extrêmement favorable à l'utilisation des cartes graphiques. L'approche NCI se base sur la topologie de la densité électronique et permet la localisation et la caractérisation dans l'espace des interactions non-covalentes, en général entre des molécules. Pour ce travail de portage sur GPU, nous nous sommes intéressés à l'implémentation NCIplot de l'approche NCI. Ce choix est justifié par la disponibilité du code aux utilisateurs ainsi que le fait que NCIplot n'implémente que l'approche NCI simplifiant ainsi l'accessibilité du code. La résolution de l'approche NCI passe par le calcul d'éléments indépendants d'une grille tridimensionnelle. Comme nous le détaillons dans le chapitre 4, le caractère indépendant des calculs rend l'approche NCI hautement parallélisable et bien adaptée à l'utilisation de GPU. De plus, la quasi totalité (99%) de l'exécution de l'approche NCI se révèle être le calcul des éléments de la grille (réalisable en simple précision) donnant ainsi un cadre idéal à une accélération par l'utilisation de cartes graphiques. Pour avoir une évaluation aussi complète que possible nous avons créé un jeu de 36 complexes provenant de la combinaison de 6 ligands et de 6 protéines. Ces 36 complexes ont été utilisés avec 4 pas de grilles différents constituant ainsi 144 cas tests différents ce qui nous a permis d'évaluer au mieux l'évolutivité des différentes implémentations de NCI. Pour évaluer les portages GPU réalisés de NCI nous avons fait notre possible pour obtenir une version parallèle CPU aussi efficace que possible ainsi qu'utiliser une technologie de CPU comparable aux GPU utilisés. Ces efforts sont justifiés par la nécessité d'avoir une évaluation juste des portages GPU réalisés. Notre meilleure version (V_{HY}) a un facteur d'accélération (avec deux cartes graphiques K20X) allant jusqu'à 39 fois plus vite que notre référence CPU sur 16 cœurs de calcul et un facteur 99 par rapport au logiciel NCIplot mis à disposition des utilisateurs actuellement. Nous constatons quand même la nécessité lors de l'utilisation de cartes graphiques d'avoir

un problème d'une taille suffisamment conséquente pour pouvoir tirer profit de la puissance de calcul disponible sur GPU. La nécessité de tirer profit de la puissance de calcul se retrouve aussi dans notre analyse de la consommation énergétique. En effet, dans le cadre de l'approche NCI, nous avons pu constater une diminution de la consommation énergétique (en considérant la consommation de l'hôte) jusqu'à un facteur 11,7 pour un GPU et facteur 13,2 pour deux GPU par rapport à notre installation CPU de référence. Nous avons fait le constat que cette diminution de consommation absolue par rapport au CPU provient principalement de l'accélération obtenue. Ce premier exemple d'application des cartes graphiques (sur l'approche NCI) aide à voir que cette technologie *manycore* peut permettre d'accélérer certaines problématiques de la chimie théorique, lorsque ces dernières sont constituées d'un grand nombre de calculs indépendants réalisable en parallèle.

Dans le chapitre 5 nous nous sommes intéressés à une situation moins favorable à l'utilisation de cartes graphiques que précédemment : l'approche quantique combinée DFTB / FMO / PCM du logiciel GAMESS. Le code du logiciel GAMESS est très grand en étant constitué de plus d'un million de lignes écrites dans plusieurs langages de programmation. Un tel code s'avère délicat à modifier comme nous en avons évoqué dans le chapitre 5. Nous n'avions que peu d'informations au moment de débuter le travail de portage sur GPU dans ce chapitre 5. L'analyse de plusieurs exécutions de l'approche DFTB / FMO / PCM de GAMESS par le logiciel MAQAO nous a permis de déterminer la fonction *ascpot* comme étant une candidate intéressante pour un portage sur GPU. Cette fonction *ascpot* ne constitue qu'une partie (77% avec un seul cœur de calcul CPU et 22% pour 16 cœurs de calcul) du temps de calcul total, limitant l'accélération totale que nous pouvons obtenir. De plus, la fonction *ascpot* fait partie du processus itératif permettant de déterminer l'effet d'un solvant (PCM) sur le système étudié. Un tel processus itératif partitionne la charge de calcul en plusieurs sections et augmente d'autant la partie transfert de données, situation qui n'est pas idéale pour l'utilisation de GPU. Nous pouvons aussi ajouter comme point défavorable à la mise en œuvre de GPU, le fait que l'utilisation de réels double précision est nécessaire pour obtenir la convergence des résultats. Or, sur la carte graphique K20X que nous avons utilisée il y a trois fois plus de cœurs de calcul simple précision que double précision. Tous les points évoqués précédemment montrent bien le cadre défavorable à l'utilisation de GPU lors de cette partie du travail réalisé. Nous avons aussi pu mettre en exergue avec le travail réalisé sur l'approche DFTB / FMO / PCM qu'il peut se révéler délicat de définir une référence CPU claire afin de comparer les performances des portages GPU réalisés. De mon point de vue, la référence CPU doit être la plus efficace possible tant que la précision du résultat est satisfaisante pour l'utilisation souhaitée. Pour moi, ce qui est capital est de répondre à la problématique en adaptant la méthode de résolution à l'architecture utilisée. Pour conclure ce propos, je pense que l'approche utilisée sur CPU peut être radicalement différente de celle utilisée sur GPU si les deux approches répondent à la problématique à résoudre. Il peut s'avérer même nécessaire dans certains cas d'avoir une approche fondamentalement différente afin d'utiliser la puissance de l'architecture *manycore* utilisée. Dans le cas de GAMESS (calcul DFTB / FMO / PCM, sur un complexe ligand-protéine), nous nous sommes éloignés du code CPU en supprimant l'approximation régionale car malgré son efficacité sur CPU, dans notre cas s'est révélée contreproductive sur GPU. Nous avons ensuite pu obtenir notre portage GPU avec une utilisation efficace de la mémoire partagée (par *blocking*), capable de mettre en œuvre une ou deux cartes graphiques pour résoudre la fonction *ascpot*. La performance de ce portage GPU a donc été étudiée afin de pouvoir voir si malgré un cadre défavorable, il est tout de même possible d'obtenir une diminution du

temps d'exécution. Dans le cas de l'utilisation d'une carte graphique K20X (12s, 180s et 3441s) nous avons constaté que le code CPU avec l'optimisation régionale (8s, 110s et 2055s) était plus rapide sur nos trois cas tests. Lors de l'utilisation de deux cartes graphiques K20X, nous avons constaté que sur le premier exemple (8 946 éléments) les temps étaient similaires (8s). En revanche pour les deux autres exemples (35 694 et 142 795 éléments), l'utilisation de deux cartes graphiques permet d'améliorer le temps d'exécution de la fonction *ascpot* (passant respectivement de 175 s et 3 687 s sur CPU à 93 s et 1 718 s sur GPU). Néanmoins, l'accélération obtenue n'est pas suffisante pour diminuer la consommation énergétique des exécutions. Le travail réalisé sur GAMESS permet de voir que l'obtention d'une accélération peut nécessiter un travail d'optimisation du code important afin d'utiliser un algorithme adapté à l'architecture *manycore* utilisée.

En somme, ce travail de recherche montre que la chimie théorique est capable de tirer profit de la technologie *manycore* des GPU dans certains cas favorables, tout en montrant qu'il est nécessaire d'adapter la méthode de résolution aux nouvelles architectures parallèles. De même, nous avons pu constater que l'efficacité énergétique des cartes graphiques est principalement liée à l'accélération obtenue.

Dans le cadre de cette thèse nous avons évalué l'utilisation des cartes graphiques sans considérer une utilisation conjointe CPU et GPU. Ce type d'approche peut permettre d'obtenir dans certains cas une meilleure performance des temps de calcul en nécessitant un investissement supplémentaire dans l'équilibrage des charges de calcul.

Dans le contexte de l'approche NCI, un projet de visualisation en temps réel des surfaces d'interactions semble être une perspective réalisable à présent. En effet, nous pensons que nous avons actuellement la puissance de calcul pour permettre un tel projet. D'autant que certaines optimisations peuvent encore être envisagées pour diminuer le nombre de calculs à réaliser dans le cadre de l'approche NCI, par exemple par le biais d'un pré-traitement géométrique. Un tel projet pourrait prendre la forme d'un logiciel interactif où l'utilisateur serait capable de déplacer "à la main" le ligand afin d'explorer le site d'une protéine et d'avoir en temps réel les surfaces d'interactions.

L'approche IGM abordée section 3.7 peut tirer profit du travail de portage réalisé sur NCI car ces approches sont proches, un facteur d'accélération similaire peut être attendu. Le travail réalisé sur NCI est actuellement diffusé au travers du programme cuNCI[55] écrit au cours de cette thèse.

Concernant les perspectives du travail de portage GPU réalisé sur GAMESS, l'approximation régionale semble adaptable à l'utilisation des GPU pour des cas d'une taille suffisante pour charger les GPU. Mais une perspective qui nous semble plus intéressante serait de changer complètement la méthode de résolution, afin de passer par une méthode plus adaptée aux GPU.

Liste des tableaux

Table des figures

Bibliographie

[1] NVIDIA Corporation : CUDA Zone. https://developer.nvidia.com/cuda-zone, consulté le 15 septembre 2018.

[2] Gaëtan Rubez, Jean-Matthieu Etancelin, Xavier Vigouroux, Michael Krajecki, Jean-Charles Boisson et Eric Hénon : GPU accelerated implementation of NCI calculations using promolecular density. *Journal of Computational Chemistry*, 38(14):1071–1083, 2017.

[3] Yoshio Nishimoto et Dmitri G. Fedorov : The fragment molecular orbital method combined with density-functional tight-binding and the polarizable continuum model. *Phys. Chem. Chem. Phys.*, 18:22047–22061, 2016.

[4] TOP500.org : Site internet du top 500. Disponible à l'adresse : https://www.top500.org/, consulté le 15 septembre 2018.

[5] NVIDIA Corporation : Cuda toolkit documentation. Disponible à l'adresse : http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, consulté le 15 septembre 2018.

[6] FAQ PCIe 3.0. Disponible à l'adresse : https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2, consulté le 15 septembre 2018.

[7] FAQ PCIe 4.0. Disponible à l'adresse : https://web.archive.org/web/20140518224913/http://www.pcisig.com/news_room/faqs/FAQ_PCI_Express_4.0/#EQ3, consulté le 15 septembre 2018.

[8] E. Scott Larsen et David McAllister : Fast Matrix Multiplies Using Graphics Hardware. *In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55, New York, NY, USA, 2001. ACM.

[9] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson et Jack Dongarra : From CUDA to OpenCL : Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012. APPLICATION ACCELERATORS IN HPC.

[10] M. J. Flynn : Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[11] Alfred Spector et David Gifford : The Space Shuttle Primary Computer System. *Commun. ACM*, 27(9):872–900, septembre 1984.

[12] Hans Werner Meuer : The TOP500 Project : Looking Back Over 15 Years of Supercomputing Experience. *Informatik-Spektrum*, 31(3):203–222, Jun 2008.

[13] E. Lindholm, J. Nickolls, S. Oberman et J. Montrym : NVIDIA Tesla : A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[14] Gene M. Amdahl : Validity of the single processor approach to achieving large scale computing capabilities. *In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[15] John L. Gustafson : Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[16] Brian W. Kernighan et Dennis Ritchie : *The C Programming Language*. Dunod, 1988.

[17] John Backus : The history of fortran i, ii, and iii. *SIGPLAN Not.*, 13(8):165–180, août 1978.

[18] TOP500.org : The Linpack Benchmark on TOP500. Disponible à l'adresse : `https://www.top500.org/project/linpack/`, consulté le 15 septembre 2018.

[19] Site internet de MAQAO. Disponible à l'adresse : `http://www.maqao.org/`, consulté le 15 septembre 2018.

[20] Department of Computer and Information Science University of Oregon : Site internet de TAU. Disponible à l'adresse : `https://www.cs.uoregon.edu/research/tau/home.php`, consulté le 15 septembre 2018.

[21] NVIDIA Corporation : Documentation en ligne de NVprof. Disponible à l'adresse : `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`, consulté le 15 septembre 2018.

[22] Site internet d'openMP. Disponible à l'adresse : `https://www.openmp.org/`, consulté le 15 septembre 2018.

[23] Forum MPI. Disponible à l'adresse : `https://www.mpi-forum.org/`, consulté le 15 septembre 2018.

[24] NVIDIA Corporation : Site internet de NVIDIA. Disponible à l'adresse : `https://www.nvidia.fr/page/home.html`, consulté le 15 septembre 2018.

[25] NVIDIA Corporation : CUDA Toolkit Documentation : Features and Technical Specifications. Disponible à l'adresse : `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications`, consulté le 15 septembre 2018.

[26] TOP500.org : Liste du TOP500 de novembre 2013. Disponible à l'adresse : `https://www.top500.org/list/2013/11/?page=2`, consulté le 15 septembre 2018.

[27] TOP500.org : Liste du Green500 de novembre 2013. Disponible à l'adresse : `https://www.top500.org/green500/lists/2013/11/`, consulté le 15 septembre 2018.

[28] TOP500.org : Site internet du Green500. Disponible à l'adresse : `https://www.top500.org/green500/`, consulté le 15 septembre 2018.

[29] NVIDIA Corporation : Accélérateurs GPU Kepler Tesla. Disponible à l'adresse : `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`, consulté le 15 septembre 2018.

[30] NVIDIA Corporation : Kepler GK110 Whitepaper. Disponible à l'adresse : `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, consulté le 15 septembre 2018.

[31] Chantal Barberot, Jean-Charles Boisson, S Gérard, Hassan K. Khartabil, E Thiriot, Gérald Monard et Eric Hénon : AlgoGen : A tool coupling a linear-scaling quantum method with a genetic algorithm for exploring non-covalent interactions. *Computational and Theoretical Chemistry*, 1028:7–18, 01 2014.

[32] Stewart James J. P. : Application of localized molecular orbitals to the solution of semiempirical self-consistent field equations. *International Journal of Quantum Chemistry*, 58(2):133–146, 1996.

[33] James J. Vincent, Steven L. Dixon et Kenneth M. Merz Jr. : Parallel implementation of a divide and conquer semiempirical algorithm. *Theoretical Chemistry Accounts*, 99(4):220–223, Jul 1998.

[34] Kazuo Kitaura, Eiji Ikeo, Toshio Asada, Tatsuya Nakano et Masami Uebayasi : Fragment molecular orbital method : an approximate computational method for large molecules. *Chemical Physics Letters*, 313(3):701 – 706, 1999.

[35] Stephan Mohr, Marc Eixarch, Maximilian Amsler, Mervi J. Mantsinen et Luigi Genovese : Linear scaling dft calculations for large tungsten systems using an optimized local basis. *Nuclear Materials and Energy*, 15:64 – 70, 2018.

[36] Yoshio Nishimoto, Dmitri G. Fedorov et Stephan Irle : Density-Functional Tight-Binding Combined with the Fragment Molecular Orbital Method. *Journal of Chemical Theory and Computation*, 10(11):4801–4812, 2014. PMID : 26584367.

[37] Christian Silvio Pomelli, Jacopo Tomasi et Vincenzo Barone : An improved iterative solution to solve the electrostatic problem in the polarizable continuum model. *Theoretical Chemistry Accounts*, 105(6):446–451, May 2001.

[38] Richard F. W. Bader : *Atoms in Molecules : A Quantum Theory*. Clarendon Press, 1990.

[39] Julia Contreras-García, Erin R. Johnson, Shahar Keinan, Robin Chaudret, Jean-Philip Piquemal, David N. Beratan et Weitao Yang : NCIPLOT : A Program for Plotting Noncovalent Interaction Regions. *Journal of Chemical Theory and Computation*, 7(3):625–632, 2011. PMID : 21516178.

[40] Gabriele Saleh, Carlo Gatti et Leonardo Lo Presti : Non-covalent interaction via the reduced density gradient : Independent atom model vs experimental multipolar electron densities. *Computational and Theoretical Chemistry*, 998:148 – 163, 2012. Non-covalent interactions and hydrogen bonding : commonalities and differences.

[41] Julia Contreras García : *Revealing Non Covalent Interactions. Analysis and development of the reduced density gradient in molecules and solids*. Habilitation à diriger des recherches en chimie, Université Pierre et Marie Curie : Paris, Octobre 2015. 71 pages.

[42] Corentin Lefebvre, Gaëtan Rubez, Hassan Khartabil, Jean-Charles Boisson, Julia Contreras-García et Eric Hénon : Accurately extracting the signature of intermolecular interactions present in the NCI plot of the reduced density gradient versus electron density. *Phys. Chem. Chem. Phys.*, 19:17928–17936, 2017.

[43] Lefebvre Corentin, Hassan K. Khartabil, Jean-Charles Boisson, Julia Contreras-García, Jean-Philip Piquemal et Eric Hénon : The Independent Gradient Model : a new approach for probing strong and weak interactions in molecules from wave function calculations. *ChemPhysChem*, 19, 12 2017.

[44] Site internet du logiciel GAMESS. Disponible à l'adresse : `http://www.msg.ameslab.gov/gamess/`, consulté le 15 septembre 2018.

[45] R. Johnson, Erin, Shahar Keinan, Paula Mori-Sánchez, Julia Contreras-García, Aron J. Cohen et Weitao Yang : Revealing Noncovalent Interactions. *Journal of the American Chemical Society*, 132(18):6498–6506, 2010. PMID : 20394428.

[46] William Humphrey, Andrew Dalke et Klaus Schulten : VMD : Visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33 – 38, 1996.

[47] Utkarsh Ayachit : *The ParaView Guide : A Parallel Visualization Application*. Broché, 2015.

[48] Pan Wu, Robin Chaudret, Xiangqian Hu et Weitao Yang : Noncovalent Interaction Analysis in Fluctuating Environments. *Journal of Chemical Theory and Computation*, 9(5):2226–2234, 2013.

[49] Burton S. Garbow : EISPACK — A package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179 – 184, 1974.

[50] Multiwfn : A Multifunctional Wavefunction Analyzer. Disponible à l'adresse : `http://sobereva.com/multiwfn/`, consulté le 15 septembre 2018.

[51] Jmol : an open-source Java viewer for chemical structures in 3D. Disponible à l'adresse : `http://www.jmol.org/`, consulté le 15 septembre 2018.

[52] Site internet du logiciel NCIplot. Disponible à l'adresse : `http://www.lct.jussieu.fr/pagesperso/contrera/nciplot.html`, consulté le 15 septembre 2018.

[53] Harris Mark : Document de NVIDIA sur l'optimisation de l'opération de réduction sur GPU. Disponible à l'adresse : `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`, consulté le 15 septembre 2018.

[54] Feuille excel CUDA Occupancy calculator. Disponible à l'adresse : `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, consulté le 15 septembre 2018.

[55] Gaëtan Rubez, Xavier Vigouroux, Michael Krajecki, Jean-Charles Boisson et Eric Hénon : Page de téléchargement du programme cuNCI. Disponible à l'adresse : `http://www.lct.jussieu.fr/pagesperso/contrera/nci-GPU.html`, consulté le 15 septembre 2018.

[56] E. Katchalski-Katzir, M. Eisenstein I. Shariv, A. A. Friesem, C. Aflalo et I. A. Vakser : Molecular surface recognition : determination of geometric fit between proteins and their ligands by correlation techniques. *Proc Natl Acad Sci U S A*, 89(6):2195–2199, Mars 1992.

[57] Graham D. Fletcher, Michael W. Schmidt, Brett M. Bode et Mark S. Gordon : The Distributed Data Interface in GAMESS. *Computer Physics Communications*, 128(1):190 – 200, 2000.

[58] Ping Lai, Sayantan Sur et Dhabaleswar K. Panda : Designing truly one-sided mpi-2 rma intra-node communication on multi-core systems. *Computer Science - Research and Development*, 25(1):3–14, May 2010.

[59] James J. P. Stewart : Mopac2016. Disponible à l'adresse : `HTTP://OpenMOPAC.net`, consulté le 15 septembre 2018.

[60] Andrzej Chrzeszczyk et Jacob Anders : Matrix computations on the gpu. Disponible à l'adresse : `https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf`, consulté le 15 septembre 2018.

[61] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, août 2008.

Annexe A

Temps des versions de NCI

	1lig 3At				2lig 6At				3lig 12At				4lig 24At				5lig 48At				6lig 96At			
1prot 135																								
FortranX16 -O2	0,19	0,69	4,41	36,1	0,19	0,69	4,41	52,2	0,22	1,5	10,7	82,6	0,39	2,74	21,7	167	0,71	5,13	38,4	301	1,77	13,5	104	813
FortranX16	0,22	1,98	5,05	39,5	0,22	1,06	7,36	56,7	0,43	1,69	12,4	94,1	0,64	3,18	23,7	188	0,85	5,68	42,6	334	2,11	15,1	116	907
gcc -O3	0,11	0,92	7,1	55,9	0,18	1,37	9,9	78,4	0,27	2,14	16,9	131	0,6	4,41	33,7	269	1,07	8,3	63,2	493	2,47	19,3	144	1150
openMPX16 gcc	0,18	1,03	8,2	65,4	0,2	1,59	11,6	92,5	0,32	2,48	19,7	153	0,71	5,15	39,4	315	1,24	9,66	73,6	574	2,88	23	177	1389
pgcc -O2	0,15	0,58	4,54	36,2	0,13	0,87	6,38	50,3	0,19	1,35	10,7	83,1	0,39	2,72	20,8	171	0,67	5,05	38,7	303	1,56	11,5	89,9	719
openMPX16 pgcc 14,10	0,11	0,6	4,65	36,9	0,15	0,9	6,53	51,4	0,2	1,38	11	84,8	0,38	2,82	21,3	170	0,68	5,18	39,5	402	1,56	11,5	92	735
icc -openmp -O2	0,09	0,46	2,55	20	0,2	0,53	3,92	27,3	0,17	0,74	5,39	41,9	0,21	1,32	10,1	80,6	0,35	2,43	18,7	147	0,72	5,03	39,3	314
openMPX16 icc	0,27	1,05	6,09	44,9	0,42	1,26	8,18	62,4	0,42	1,68	13,7	103	0,63	3,57	25,8	207	0,83	6,29	48,1	377	2,13	14,3	113	903
VAT1	1,3	1,1	1,5	4,1	1,2	1,2	1,7	5,4	1,1	1,2	1,9	7,8	1,1	1,3	2,6	13	1,1	1,5	3,9	23,1	1,2	2	8,3	58,3
VNO1	1,3	1,1	1,5	3,9	1,1	1,2	1,6	5,1	1,1	1,2	1,8	7,3	1,1	1,3	2,4	11,9	1,1	1,4	3,5	21,3	1,2	1,8	7,1	oom
VAT1_2GPU	1,3	1,5	1,5	3,1	1,3	1,6	3,8	12	1,3	1,7	4,8	13	1,3	1,3	2,1	7,8	1,3	1,4	2,7	13	1,3	1,7	5	31,5
VNO1_2GPU	1,3	1,3	1,5	2,9	1,3	1,3	1,6	3,8	1,2	1,3	1,7	4,8	1,3	1,3	2,1	7,9	1,3	1,4	2,7	13,3	1,3	1,6	4,5	28,4
VAT2_64threads	1,1	1,1	1,5	4,1	1,1	1,2	1,6	5,3	1,1	1,2	2	7,9	1,1	1,3	2,6	13,5	1,1	1,4	3,5	20,1	1,2	1,8	6,9	47,3
VAT2_128threads	1,2	1,2	1,5	4,4	1,1	1,2	1,6	5,5	1,1	1,2	2	8,3	1,1	1,3	2,8	14,5	1,1	1,4	3,7	21,6	1,2	1,8	6,5	44
VAT2_256threads	1,1	1,1	1,5	4,1	1,1	1,2	1,6	5,3	1,1	1,2	2	7,7	1,1	1,5	3,9	23,2	1,1	1,5	3,9	23,2	1,2	2	8,3	58,3
VAT2_64threads_2GPU	1,3	1,3	1,5	3,1	1,3	1,3	1,6	3,8	1,2	1,3	1,7	4,9	1,3	1,4	2,1	8,1	1,3	1,4	2,6	11,5	1,3	1,6	4,4	26,4
VAT2_128threads_2GPU	1,2	1,3	1,5	3,2	1,2	1,3	1,6	3,9	1,3	1,3	1,7	5,1	1,3	1,4	2,2	8,6	1,3	1,4	2,6	12,4	1,3	1,6	4,1	24,5
VNO2_64threads	1,1	1,1	1,4	3,6	1,1	1,2	1,6	4,7	1,1	1,2	1,8	6,8	1,1	1,3	2,5	11,5	1,1	1,4	3,3	18	1,2	1,8	6,2	oom
VNO2_128threads	1,1	1,1	1,4	3,9	1,2	1,1	1,6	5,2	1,2	1,2	2	7,6	1,2	1,3	2,5	12,8	1,2	1,4	3,4	20,4	1,2	1,8	6,4	oom
VNO2_64thread_2GPU	1,3	1,3	1,4	2,7	1,2	1,3	1,5	3,2	1,3	1,3	1,7	4,5	1,3	1,4	2,1	7,4	1,3	1,4	2,6	11,2	1,3	1,7	4,1	23,1
VNO2_128thread_2GPU	1,3	1,4	1,5	2,7	1,2	1,3	1,5	3,2	1,3	1,4	1,7	4,4	1,3	1,4	2	7,5	1,3	1,4	2,6	11,2	1,3	1,6	4	23,2
VHY	1,2	1,7	1,8	2,7	1,3	1,8	2,6	3,9	1,4	1,7	3,9	5,8	1,1	1,2	1,8	5,8	1,1	1,2	2,2	8,8	1,2	1,5	3,9	20,5
VHY_2GPU	1,3	1,3	1,4	1,9	1,3	1,4	2,1	2,8	1,2	1,3	1,5	2,8	1,3	1,3	1,7	4,3	1,3	1,4	1,9	6,2	1,3	1,5	3,3	12,9
VHY_DP	1,1	1,2	1,3	2,5	1,1	1,1	1,4	3,7	1,1	1,2	1,8	5,7	1,1	1,3	2,2	9,7	1,1	1,3	2,7	12,8	1,2	1,6	4,9	oom
VHY_DP_2GPU	1,6	1,5	1,6	2,5	1,5	1,7	3,1	1,5	1,8	4,2	1,5	1,6	2,2	6,8	1,5	1,6	2,7	10,4	1,5	1,8	4,3	20,2	1,6	1,6
2prot 239																								
FortranX16 -O2	0,25	1,17	7,45	58,2	0,24	1,46	10,5	83	0,34	2,25	17,6	136	0,62	4,45	33,7	269	1,39	7,7	58,8	460	2,46	19,3	148	1161
FortranX16	0,22	1,27	8,2	63,4	0,43	1,69	11,6	91,2	0,43	2,54	19,3	149	0,85	4,85	36,8	293	1,29	8,62	64,1	501	2,74	21	161	1265
gcc -O3	0,21	1,53	12,1	96,8	0,29	2,29	16,9	132	0,46	3,7	29,5	227	1,16	8,3	65,3	512	1,89	14,9	114	889	3,62	28,2	218	1746
openMPX16 gcc	0,22	1,87	14,6	113	0,34	2,68	19,8	158	0,54	4,33	34,4	265	1,24	9,04	69,2	552	2,08	16,5	126	984	4,32	34	262	2052
pgcc -O2	0,13	1,03	7,85	62,5	0,19	1,47	10,8	84,8	0,3	2,29	18,2	141	1,11	4,49	34,3	274	1	7,93	61	478	2,24	16,4	130	1041
openMPX16 pgcc 14,10	0,13	1,03	8	63,8	0,19	1,5	11,1	86,6	0,3	2,36	18,6	144	0,63	4,6	35	280	1,04	8,1	62,3	489	2,26	16,7	133	1064
icc -openmp -O2	0,14	0,81	4,29	33,5	0,2	1,02	5,81	43,8	0,52	1,13	8,93	69,3	0,3	2,21	16,6	131	0,54	3,85	29,4	231	0,97	7,04	56	447
openMPX16 icc	0,41	1,47	9,87	77,3	0,42	2,11	13,7	105	0,62	2,94	22,7	175	0,84	5,66	42,6	341	1,25	9,86	75,8	594	2,96	20,6	164	1310
VAT1	1,1	1,2	1,7	5,8	1,1	1,2	1,9	7,6	1,1	1,2	2,4	11,6	1,1	1,4	4	23,8	1,2	1,7	5,6	36,6	1,3	2,5	12,8	94,8
VNO1	1,1	1,2	1,6	5,3	1,1	1,2	1,8	7	1,1	1,2	2,3	10,9	1,2	1,4	3,5	20,9	1,1	1,6	5	34,3	1,2	2,2	10,2	oom
VAT1_2GPU	1,3	1,3	1,6	4	1,2	1,3	1,7	5,1	1,3	1,4	1,9	6,9	1,3	1,4	2,8	13,9	1,3	1,5	3,6	19,9	1,3	2	7,3	49,3
VNO1_2GPU	1,2	1,3	1,6	3,8	1,3	1,3	1,7	4,5	1,3	1,3	1,9	6,6	1,3	1,4	2,7	12,9	1,3	1,5	3,5	20,4	1,3	1,8	6	42,1
VAT2_64threads	1,2	1,1	1,5	4,8	1,1	1,2	1,7	6,3	1,1	1,2	2,1	9,4	1,1	1,3	3,2	17,9	1,1	1,5	4,5	28,4	1,2	2,1	8,8	62,5
VAT2_128threads	1,1	1,2	1,6	4,6	1,1	1,2	1,7	5,9	1,1	1,2	2,1	8,9	1,1	1,3	3,1	17,4	1,1	1,5	4,4	27,7	1,2	2	7,9	57
VAT2_256threads	1,1	1,2	1,7	5,8	1,1	1,2	1,9	7,6	1,1	1,3	2,4	11,6	1,2	1,9	7,6	33,0	1,2	1,9	7,6	53	1,3	2,7	14	104
VAT2_64threads_2GPU	1,2	1,3	1,5	3,5	1,2	1,3	1,6	4,3	1,2	1,3	1,8	5,7	1,3	1,4	2,4	10,5	1,3	1,5	3	15,4	1,3	1,8	5,3	33,6
VAT2_128threads_2GPU	1,2	1,3	1,6	3,3	1,2	1,3	1,6	4,1	1,2	1,4	1,8	5,4	1,3	1,4	2,4	10,2	1,3	1,5	3	15	1,3	1,7	4,8	30,7
VNO2_64threads	1,2	1,2	1,5	4,5	1,1	1,2	1,7	5,8	1,1	1,2	2,1	8,9	1,2	1,4	3	15,4	1,2	1,5	4,2	24,4	1,2	2	7,8	oom
VNO2_128threads	1,1	1,1	1,6	4,6	1,1	1,2	1,7	6,1	1,1	1,7	2,6	9,6	1,2	1,4	3	16	1,2	1,5	4,2	25,5	1,2	2	7,9	oom
VNO2_64thread_2GPU	1,3	1,3	1,5	3,2	1,3	1,3	1,6	3,8	1,3	1,3	1,8	5,5	1,3	1,4	2,3	9,5	1,3	1,5	3	14,5	1,3	1,7	4,9	29,6
VNO2_128thread_2GPU	1,3	1,4	1,5	3,2	1,3	1,3	1,6	3,8	1,3	1,4	1,8	5,5	1,3	1,4	2,3	9,5	1,3	1,5	3	14,5	1,3	1,8	4,9	29,7
VHY	1,2	1,3	1,7	2,7	1,1	1,2	1,4	3,4	1,1	1,2	1,6	4,8	1,1	1,2	2,3	8,6	1,1	1,3	2,7	12,4	1,2	1,6	4,8	27,6
VHY_2GPU	1,3	1,3	1,8	2,2	1,3	1,3	1,4	2,5	1,3	1,3	1,6	3,4	1,3	1,3	1,9	6,3	1,3	1,4	2,3	8	1,3	1,5	3,4	16,3
VHY_DP	1,1	1,1	1,5	3,1	1,1	1,1	1,6	4,2	1,1	1,3	1,6	4,2	1,1	1,3	2,8	11,1	1,1	1,4	3,3	17,4	1,2	1,7	6,2	oom
VHY_DP_2GPU	1,5	1,5	1,7	2,9	1,5	1,7	3,7	1,5	1,5	5,3	2	1,6	1,5	1,6	2,5	8,8	1,5	1,7	3,1	13,4	1,5	2	5,2	30,7

		1lig 3At				2lig 6At				3lig 12At				4lig 24At				5lig 48At				6lig 96At			
3prot 394	FortranX16 -O2	0,27	1,56	11,9	94	0,39	2,29	16,8	133	0,48	3,56	27,9	217	0,98	6,9	53	423	1,58	11,7	89,1	699	3,59	27,9	215	1684
	FortranX16	0,43	1,69	12,8	101	0,44	2,54	18,1	144	0,64	4,01	30,3	234	1,06	7,57	56,9	454	1,69	12,8	96,6	753	4,02	30,3	232	1820
	gcc -O3	0,27	2,16	17,2	137	0,42	3,32	24,4	194	0,75	5,9	46,9	363	1,75	12,7	97,4	768	2,71	21,5	168	1289	5,52	43,8	337	2652
	openMPX16 gcc	0,32	2,56	20,4	163	0,5	3,94	28,9	231	0,86	6,88	54,8	423	1,94	13,9	106	848	2,93	23,3	178	1392	6,56	50,3	394	3066
	pgcc -O2	0,18	1,37	12,3	86,9	0,27	2,1	15,4	123	0,46	3,56	28,4	219	1,06	6,84	52,3	418	1,41	11,2	84,8	663	3,06	24,4	182	1449
	openMPX16 pgcc 14,10	0,18	1,4	11,1	88,8	0,27	2,15	15,7	125	0,46	3,64	29	224	0,96	6,98	53,4	427	2	11,4	86,7	682	3,09	27,3	187	1482
	icc -openmp -O2	0,19	0,93	5,07	39,4	0,23	0,89	7,29	53,8	0,25	1,66	13,1	101	0,44	3,1	23,8	189	0,65	4,8	37,1	286	1,33	9,37	74,6	595
	openMPX16 icc	0,63	1,89	13,9	109	0,41	2,73	19,5	154	0,84	4,85	35,5	273	1,25	8,6	65,3	522	1,93	14,1	106	832	4	30,2	232	1839
	VAT1	1,1	1,2	2,1	9,1	1,2	1,3	2,5	12,5	1,1	1,4	3,8	22,8	1,2	1,7	6,1	41,4	1,2	2,1	8,9	63,2	1,3	3,2	17,7	134
	VNO1	1,1	1,2	1,9	7,3	1,1	1,2	2,2	10,4	1,1	1,4	3,4	20,3	1,2	1,6	5,2	35,2	1,2	1,9	7,6	56,7	1,3	2,7	14,4	oom
	VAT1_2GPU	1,2	1,3	1,8	5,4	1,3	1,3	2	7,1	1,3	1,4	2,7	12,8	1,3	1,6	4	23,4	1,3	1,8	5,3	33,7	1,4	2,3	9,9	70,1
	VNO1_2GPU	1,3	1,3	1,7	4,6	1,3	1,3	1,9	6,1	1,3	1,4	2,4	11,1	1,3	1,5	3,5	19,7	1,3	1,7	4,8	31,5	1,4	2,1	8,1	59,1
	VAT2_64threads	1,1	1,2	1,7	5,9	1,1	1,2	1,9	8	1,1	1,3	2,6	13,5	1,1	1,4	3,9	23,8	1,2	1,7	5,6	36,7	1,2	2,3	11	80,8
	VAT2_128threads	1,1	1,2	1,6	5,4	1,1	1,2	1,9	7,2	1,1	1,3	2,5	12,4	1,1	1,4	3,7	22,2	1,2	1,6	5,3	34,7	1,2	2,2	9,7	69,9
	VAT2_256threads	1,1	1,2	2	8,2	1,1	1,3	2,4	11,2	1,1	1,4	3,5	19,9	1,2	1,9	7,9	55,3	1,2	1,9	7,9	55,3	1,3	2,8	14,7	109
	VAT2_64threads_2GPU	1,2	1,3	1,6	3,8	1,3	1,3	1,7	4,9	1,2	1,3	2,1	7,8	1,3	1,4	2,8	13,5	1,3	1,5	3,6	19,6	1,3	1,9	6,4	42,2
	VAT2_128threads_2GPU	1,3	1,3	1,5	3,6	1,2	1,2	1,7	4,5	1,3	1,4	2	7,2	1,3	1,4	2,7	12,6	1,3	1,5	3,4	18,7	1,3	1,8	5,7	36,7
	VNO2_64threads	1,1	1,2	1,6	5,4	1,1	1,2	1,9	7,1	1,1	1,3	2,6	12,2	1,2	1,5	3,9	21,7	1,2	1,7	5,4	33,8	1,3	2,3	10,3	oom
	VNO2_128threads	1,2	1,2	1,6	5,2	1,1	1,2	1,8	7,1	1,1	1,7	3	12,8	1,2	1,5	3,7	21,6	1,2	1,6	5,3	34,1	1,2	2,2	10,1	oom
	VNO2_64thread_2GPU	1,3	1,3	1,6	3,5	1,2	1,3	1,7	4,4	1,3	1,4	2	7,1	1,3	1,4	2,7	12,4	1,3	1,6	3,6	18,9	1,3	1,9	6,1	39,3
	VNO2_128thread_2GPU	1,2	1,3	1,6	3,5	1,3	1,3	1,7	4,4	1,3	1,4	2	7,1	1,3	1,5	2,7	12,4	1,3	1,6	3,6	18,9	1,4	1,9	6,1	38,7
	VHY	1,2	1,1	1,4	3,4	1,2	1,1	1,5	4,3	1,1	1,2	1,9	6,7	1,1	1,3	2,6	11,7	1,2	1,4	3,6	17,7	1,2	1,8	6,4	40,4
	VHY_2GPU	1,3	1,3	1,5	2,6	1,2	1,3	1,5	3,3	1,3	1,3	1,7	4,4	1,3	1,4	2,1	7,1	1,3	1,5	2,5	10,6	1,3	1,6	4,1	21,3
	VHY_DP	1,1	1,2	1,5	4,7	1,1	1,2	1,8	5,7	1,1	1,2	1,8	5,7	1,1	1,5	3,6	19,3	1,2	1,6	4,9	30,1	1,2	2	8,2	oom
	VHY_DP_2GPU	1,4	1,5	1,7	3,4	1,5	1,5	1,8	4,5	1,5	1,6	2,2	6,8	1,5	1,7	2,9	11,6	1,5	1,8	3,7	17,6	1,6	2,2	6,3	38,8
4prot 722	FortranX16 -O2	0,68	2,93	21,4	169	0,61	4,12	30,1	239	0,92	6,32	49,9	387	1,87	12,2	94,7	743	2,59	20,2	154	1208	5,99	46,3	356	2795
	FortranX16	0,64	2,95	22,7	181	0,65	4,64	32,4	257	1,06	6,94	53,8	414	1,91	13,1	99,3	794	2,95	21,9	165	1291	6,34	49,8	382	3001
	gcc -O3	0,5	3,96	31,5	251	0,76	6,07	45,1	353	1,35	10,8	85,8	661	3,49	25	190	1517	4,48	35,7	273	2142	9,57	71	566	4523
	openMPX16 gcc	0,59	4,68	41	298	0,92	7,2	52,7	421	1,57	12,6	100	770	3,96	28,5	216	1729	5,18	41,3	315	2465	11,2	89,6	688	5370
	pgcc -O2	0,32	2,5	19,8	158	0,5	3,82	28	223	0,83	6,32	50,3	390	2,05	12,6	95,6	765	2,43	19,2	146	1150	5,33	39,5	316	2524
	openMPX16 pgcc 14,10	0,32	2,55	20,3	162	0,5	3,93	28,6	228	0,83	6,46	51,4	398	1,78	12,9	97,6	781	2,48	19,6	149	1175	5,42	40,4	323	2580
	icc -openmp -O2	0,28	1,23	9,23	71,5	0,24	1,61	12,7	96,2	0,38	2,99	22,9	178	0,83	5,93	44,8	358	1,17	8,29	63,2	514	2,39	17,8	142	1132
	openMPX16 icc	0,63	3,36	25	199	0,84	5,05	35,3	280	1,47	7,98	63	487	2,3	15,7	119	954	3,39	24,2	184	1444	7,16	50	398	3181
	VAT1	1,1	1,4	3,5	20	1,2	1,5	4,5	28	1,2	1,9	7,6	52,8	1,3	2,6	13,3	98,1	1,4	3,6	20,8	158	1,7	5,9	39,1	304
	VNO1	1,1	1,3	2,4	11,9	1,1	1,3	3	17,2	1,2	1,6	5,3	35,4	1,2	2,1	8,9	65,2	1,4	3,7	22,2	175	1,7	5,6	37,5	oom
	VAT1_2GPU	1,3	1,4	2,5	11	1,3	1,5	3	14,9	1,3	1,7	4,5	27,4	1,3	2,1	7,5	51,6	1,4	2,5	11,3	81,9	1,6	3,9	22,2	169
	VNO1_2GPU	1,3	1,3	2	7,1	1,3	1,4	2,3	9,7	1,3	1,5	3,4	18,8	1,3	1,8	5,4	35,4	1,4	2,6	12	90,1	1,5	3,6	20,3	158
	VAT2_64threads	1,1	1,2	2,1	8,8	1,1	1,3	2,4	11,9	1,1	1,4	3,6	20,7	1,2	1,7	5,8	38,1	1,2	2	8,7	61,2	1,3	3	16,6	125
	VAT2_128threads	1,1	1,2	1,9	7,7	1,1	1,2	2,3	10,3	1,1	1,3	3,2	17,6	1,2	1,6	5	32,4	1,2	1,9	7,5	52,7	1,3	2,8	14,6	109
	VAT2_256threads	1,1	1,2	2,3	10,5	1,2	1,3	2,8	14,4	1,1	1,6	4,2	25,7	1,2	2,4	11,1	81	1,2	2,4	11,1	81	1,4	3,7	22	168
	VAT2_64threads_2GPU	1,2	1,3	1,7	5,2	1,2	1,3	2	6,8	1,3	1,4	2,5	11,4	1,3	1,6	3,6	20.2	1,3	1,7	5,2	32,5	1,4	2,3	9,7	70,5
	VAT2_128threads_2GPU	1,3	1,3	1,7	4,6	1,2	1,3	1,9	6	1,3	1,4	2,3	9,7	1,3	1,5	3,2	17,3	1,3	1,7	4,6	28,2	1,4	2,2	8,6	59,9
	VNO2_64threads	1,1	1,2	2	8,3	1,1	1,3	2,5	11,1	1,1	1,5	3,5	19,4	1,2	1,8	5,7	35,7	1,2	2,1	8,5	56,4	1,3	3,1	15,6	oom
	VNO2_128threads	1,1	1,2	2	7,6	1,2	1,3	2,3	10,7	1,1	1,9	3,7	18,9	1,2	1,7	5,3	33,8	1,2	2	7,9	55	1,3	2,8	15,1	oom
	VNO2_64thread_2GPU	1,3	1,3	1,7	5	1,3	1,4	1,9	6,4	1,3	1,4	2,5	10,6	1,3	1,6	3,7	19,5	1,3	2	5,1	29,6	1,4	2,3	8,8	61,1
	VNO2_128thread_2GPU	1,3	1,3	1,7	5	1,3	1,4	2	6,4	1,3	1,4	2,5	10,6	1,3	1,6	3,8	19,5	1,3	1,8	5	29,7	1,4	2,3	8,8	60,3
	VHY	1,1	1,2	1,6	4,7	1,5	1,2	1,9	7,2	1,1	1,3	2,5	10,7	1,2	1,5	3,8	20,3	1,2	1,7	5,1	29,8	1,3	2,3	9,6	60,1
	VHY_2GPU	1,4	1,7	2,1	3,5	1,2	1,3	1,6	4,1	1,3	1,4	2	6,3	1,3	1,5	2,7	11,2	1,3	1,5	3,4	16,3	1,4	1,9	5,8	32,8
	VHY_DP	1,1	1,2	2	6,2	1,1	1,2	2,1	10,2	1,1	1,2	2,1	10,2	1,2	1,7	4,5	25,3	1,2	1,8	6,3	39	1,4	2,5	15,2	oom
	VHY_DP_2GPU	1,5	1,5	1,9	4,6	1,5	1,5	2,1	6,3	1,5	1,6	2,7	10,1	1,5	1,8	3,9	18,2	1,5	1,9	5	27	1,6	2,5	9	57,5

		1lig 3At				2lig 6At				3lig 12At				4lig 24At				5lig 48At				6lig 96At			
5prot 907	FortranX16 -O2	0,53	3,47	26,8	212	0,72	5,14	37,7	299	1,16	7,93	62,4	484	2,15	15,2	117	927	3,26	25,1	191	1497	7,26	56,8	437	3427
	FortranX16	0,64	3,79	28,6	226	0,85	5,68	40,3	320	1,29	8,42	67	516	2,32	16,2	124	987	3,59	26,9	204	1596	7,79	60,9	468	3673
	gcc -O3	0,64	4,99	39,6	316	0,96	7,6	55,9	444	1,86	14,9	111	921	3,38	29,3	223	1780	6,32	46,6	386	3004	11,8	88,4	706	5639
	openMPX16 gcc	0,75	5,89	46,9	375	1,14	9,1	66,3	529	2,16	17,2	137	1051	4,61	33,6	256	2043	7,22	57,5	441	3436	13,8	110	849	6681
	pgcc -O2	0,41	3,13	24,9	199	0,9	4,79	35,1	279	1,06	8,43	62,9	515	1,85	15,3	117	935	3,3	24	199	1548	6,45	48	383	3063
	openMPX16 pgcc 14,10	0,43	3,2	25,5	203	0,62	4,89	35,9	286	1,08	8,6	64,2	526	1,88	15,6	119	954	3,34	24,5	203	1581	6,58	49	391	3132
	icc -openmp -O2	0,29	1,44	11,3	89,9	0,42	2	15,9	121	1,11	4,29	32,1	261	1	7,68	57	454	1,57	11,3	93,7	725	2,94	21,3	170	1361
	openMPX16 icc	0,85	3,99	31,7	250	0,84	6,08	44,5	351	1,47	10,5	78,3	640	2,51	19,3	146	1168	4,19	30	248	1930	8,41	60,7	484	3869
	VAT1	1,1	1,4	3,8	22,3	1,2	1,6	4,9	31,2	1,2	2	8,2	57,4	1,3	2,7	14,1	104	1,4	3,7	21,7	165	1,8	6,4	43,3	339
	VNO1	1,1	1,4	3,5	19,9	1,1	1,5	4,5	29,1	1,2	2	8,4	60,3	1,3	2,7	14	106	1,4	3,7	22,3	177	1,7	5,8	39,7	oom
	VAT1_2GPU	1,3	1,4	2,6	12,2	1,3	1,5	3,2	16,5	1,3	1,7	4,8	29,6	1,4	2,1	7,9	54,4	1,4	2,6	11,7	84,4	1,6	4,1	24	183
	VNO1_2GPU	1,3	1,4	2,6	11,6	1,3	1,5	3,1	40,3	1,3	1,8	5,4	35	1,4	2,1	7,9	55,2	1,4	2,7	12,9	97,4	1,6	3,7	21,4	168
	VAT2_64threads	1,1	1,2	2,3	10,5	1,1	1,3	2,7	14,3	1,2	1,5	4	24,6	1,2	1,8	6,5	44,6	1,2	2,2	9,7	70,1	1,4	3,4	19,6	149
	VAT2_128threads	1,1	1,2	2,1	9,3	1,1	1,3	2,5	12,3	1,1	1,4	3,6	21	1,2	1,7	5,7	38,3	1,2	2	8,6	60,9	1,3	3	16,5	124
	VAT2_256threads	1,1	1,3	2,5	24,9	1,2	1,3	3,1	17,1	1,1	1,6	4,7	29,9	1,3	2,4	24,2	86,4	1,3	2,4	24,2	86,4	1,4	3,8	22,7	175
	VAT2_64threads_2GPU	1,3	1,3	1,9	6,1	1,3	1,4	2,1	8	1,3	1,4	2,8	13,3	1,3	1,6	4	23,4	1,3	1,8	5,7	37,1	1,4	2,5	11,1	80,2
	VAT2_128threads_2GPU	1,3	1,4	1,8	5,3	1,2	1,3	2	7	1,3	1,4	2,5	11,5	1,3	1,5	3,6	20,2	1,3	1,7	5	31,8	1,4	2,3	9,5	67,1
	VNO2_64threads	1,1	1,3	2,2	9,9	1,2	1,3	2,8	13,3	1,1	1,5	4	23	1,2	1,9	6,6	41,9	1,3	2,3	9,7	65,7	1,4	3,5	18,3	oom
	VNO2_128threads	1,1	1,3	2,1	9	1,2	1,3	2,5	12,8	1,2	1,5	3,7	21,7	1,2	1,8	6	39,3	1,3	2,1	9	63,4	1,4	3,1	17,4	oom
	VNO2_64thread_2GPU	1,3	1,3	1,8	5,8	1,3	1,4	2,1	7,5	1,3	1,5	2,8	12,8	1,3	1,7	4	22	1,3	1,9	5,7	34,8	1,4	2,5	10,2	73,4
	VNO2_128thread_2GPU	1,3	1,4	1,8	5,8	1,3	1,4	2,1	7,5	1,3	1,5	2,8	12,8	1,3	1,6	4	22,1	1,3	1,9	5,7	34,8	1,4	2,5	10,2	70,9
	VHY	1,1	1,6	2,2	5,7	1,1	1,7	2	7,9	1,1	1,4	2,8	12,8	1,2	1,6	4,3	23,2	1,2	1,8	5,9	35,3	1,3	2,5	11,2	71,3
	VHY_2GPU	1,3	1,3	1,6	3,6	1,3	1,3	1,7	4,8	1,3	1,4	2,2	7,4	1,3	1,5	2,9	12,8	1,3	1,6	3,7	19,1	1,4	2	6,5	38,4
	VHY_DP	1,1	1,2	2,3	7,4	1,1	1,2	2,3	10,7	1,1	1,2	2,3	18,4	1,2	2,5	6,5	30,9	1,2	2,2	8,3	59,3	1,4	3,3	16,9	oom
	VHY_DP_2GPU	1,5	1,5	2	5,3	1,5	1,6	2,2	7,4	1,5	1,7	3	11,7	1,5	1,9	4,3	20,8	1,5	2	5,6	31	1,6	2,7	10,3	66,6
6prot 1245	FortranX16 -O2	0,72	4,71	36,5	290	0,97	7,04	51,4	408	1,53	10,8	85,2	660	2,88	20,7	159	1261	4,42	33,9	258	2024	9,57	75,8	583	4576
	FortranX16	0,85	5,06	38,9	308	1,07	7,58	54,8	437	1,5	11,6	91,2	703	3,17	22,2	168	1341	4,64	36,4	276	2152	10,3	81,1	623	4892
	gcc -O3	0,86	6,86	54,6	436	1,33	10,5	77	613	2,54	20	150	1243	4,45	38,6	293	2340	8,46	62,3	517	4031	15,6	117	936	7520
	openMPX16 gcc	1,02	8,13	64,7	516	1,57	12,5	91,3	728	2,94	23,1	185	1433	6,04	44,5	366	2698	9,66	76,8	592	4610	18,4	146	1132	8864
	pgcc -O2	0,54	4,29	34,1	272	0,86	6,56	48	383	1,42	11,3	84,6	698	2,48	20,4	155	1243	4,41	32,1	267	2075	8,49	64	511	4086
	openMPX16 pgcc 14,10	0,56	4,39	34,9	278	0,85	6,7	49,1	391	1,45	11,5	86,3	713	2,53	20,8	159	1270	4,48	32,8	273	2121	8,69	65,5	523	4180
	icc -openmp -O2	0,29	1,96	15,5	123	0,57	2,75	21,6	165	0,76	5,73	42,5	362	1,13	9,87	74,5	595	2,07	15,1	125	968	3,81	28,7	228	1820
	openMPX16 icc	0,85	5,46	43,1	342	1,26	8,49	60,5	481	2,1	14,3	105	868	3,15	25,6	195	1557	5,67	40,1	333	2592	11	81,1	646	5165
	VAT1	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err
	VNO1	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err
	VAT1_2GPU	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err
	VNO1_2GPU	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err	Err
	VAT2_64threads	1,2	1,3	2,7	13,4	1,2	1,4	3,3	18,3	1,1	1,6	4,9	31,4	1,2	1,9	8	56,7	1,3	2,5	12,6	92,8	1,6	4,1	24,7	190
	VAT2_128threads	1,1	1,3	2,4	11,3	1,1	1,3	2,9	15,4	1,1	1,6	4,3	26,2	1,2	1,8	6,9	47,2	1,3	2,3	10,6	77,3	1,5	3,6	20,7	158
	VAT2_256threads	1,2	1,3	2,8	14,7	1,2	1,4	3,5	20,3	1,2	1,6	5,4	35,7	1,3	2,8	14,5	108	1,3	2,8	14,5	108	1,7	4,5	28,3	218
	VAT2_64threads_2GPU	1,3	1,4	2	7,6	1,3	1,4	2,3	10,1	1,3	1,5	3,2	16,7	1,3	1,7	4,9	30,1	1,3	2	7,1	47,8	1,5	2,8	13,6	100
	VAT2_128threads_2GPU	1,3	1,4	2	6,4	1,3	1,4	2,2	8,5	1,3	1,4	2,9	14	1,3	1,6	4,3	25,2	1,3	1,9	6,1	39,9	1,5	2,5	11,6	84,2
	VNO2_64threads	1,1	1,3	2,6	12,8	1,2	1,4	3,3	17,4	1,2	1,7	5	29,9	1,3	2,1	8,3	54,2	1,3	2,7	12,4	86	1,6	4,2	23,5	oom
	VNO2_128threads	1,1	1,4	2,5	11,4	1,1	1,4	3	16,4	1,2	1,6	4,5	27,7	1,3	2	7,4	49,9	1,3	2,5	11,3	81,3	1,6	3,7	22,2	oom
	VNO2_64thread_2GPU	1,3	1,4	2	7,3	1,3	1,4	2,4	9,6	1,3	1,5	3,3	16,4	1,3	1,8	4,9	28,3	1,3	2,1	7,1	45,3	1,6	2,8	12,8	93,4
	VNO2_128thread_2GPU	1,3	1,4	2	7,3	1,3	1,4	2,4	9,5	1,3	1,6	3,3	16,4	1,3	1,8	4,9	28,3	1,3	2,1	7,1	45,3	1,6	2,8	12,8	91,2
	VHY	1,1	1,3	2	7,1	1,1	1,3	2,3	10,3	1,2	1,4	3,4	16,7	1,2	1,7	5,4	30,5	1,3	2	7,4	46,9	1,4	2.9	14.3	92,8
	VHY_2GPU	1,3	1,3	1,7	4,4	1,3	1,3	1,9	5,9	1,3	1,4	2,4	9,4	1,3	1,6	3,5	16,3	1,3	1,7	4,5	24,8	1,4	2,2	8,1	49,3
	VHY_DP	1,1	1,3	2,7	9,5	1,1	1,4	3,1	16,2	1,1	1,4	3,1	16,2	1,2	2,2	8,3	50,5	1,3	2,4	10,5	62,4	1,5	4,1	22,1	oom
	VHY_DP_2GPU	1,5	1,6	2,3	6,6	1,5	1,6	2,5	9,3	1,5	1,8	3,5	15	1,5	2	5,2	26,5	1,6	2,2	6,9	40,4	1,7	3	12,9	84,3

Annexe B

Nombre à virgule flottante

L'encodage en informatique d'un nombre réel peut être fait selon différents paradigmes, à virgule fixe ou encore à virgule flottante par exemple. Au cours de l'histoire plusieurs normes d'encodages sont apparues et ont été utilisé pour représenter les nombres réels. Notons tout de même que la représentation la plus couramment utilisée suivie la norme IEEE 754 [61] définissant cinq formats basiques d'encodage pour les nombres à virgule flottante :

- Trois formats binaires, encodés sur 32, 64 et 128 bits.
- Deux formats décimaux, encodés sur 64 et 128 bits.

Un format de nombre à virgule flottante se constitue :

- d'un triplet (signe, exposant, mantisse) dans une base b, le nombre à virgule flottante représenté par ce triplet est $(-1)^{signe} \times mantisse \times b^{exposant}$
- $+\infty$,$-\infty$
- NaNs (Not a Number)

Le format de la norme IEEE sur 32 bits est parfois utilisé par les compilateurs du langage C pour les variables de type float et la version 64 bits pour les variables de type double. Sur 32 bits, le format est défini par un bit de signe (0 pour une nombre positif, 1 pour un nombre négatif), 8 bits pour l'exposant (L'exposant est biaisé) et 23 bits pour la mantisse. La valeur du nombre prend donc cette forme avec le format 32 bits : $(-1)^{signe} \times mantisse \times 2^{exposant-127}$. Le format 64 bits est constitué d'un bit de signe, de 11 bits pour l'exposant et de 52 bits pour la mantisse. La valeur du nombre prend donc cette forme au format 64 bits : $(-1)^{signe} \times mantisse \times 2^{exposant-1023}$.

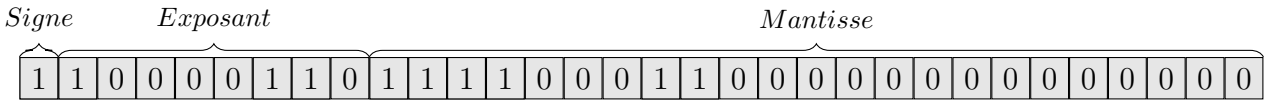


FIGURE 2.1 – Représentation binaire de -248,75 avec la norme IEEE-754, 32 bits

Dans un intervalle réel il existe une infinité de nombre réel hors le stockage est fait sur un nombre fini de bits, en conséquence les nombres à virgule flottante sont une approximation des nombres réels. De ce constat découle une perte de certaines propriétés mathématiques, tel que l'associativité de l'addition, dans certains cas. Avoir ce fait en tête est primordial pour s'assurer une utilisation des nombres à virgule flottante minimisant l'erreur produite.

Annexe C

Méthode de Cardan

La méthode de Cardan permet d'obtenir une expression analytique des solutions de l'équation :

$$z^3 + pz + q = 0 \tag{C.1}$$

Ainsi, permet d'obtenir les racines du polynôme :

$$ax^3 + bx^2 + cx + d \tag{C.2}$$

par le changement de variable suivant :

$$x = z - \frac{b}{3a} \tag{C.3}$$

Un polynôme du troisième degrés comme décrit C.2 admettant trois racines réelles peut écrire ses trois racines sous cette forme :

$$x_1 = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{\arccos(\frac{3q}{2p}\sqrt{\frac{-3}{p}})}{3}\right) - \frac{b}{3a} \tag{C.4}$$

$$x_2 = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{\arccos(\frac{3q}{2p}\sqrt{\frac{-3}{p}}) + 2\pi}{3}\right) - \frac{b}{3a} \tag{C.5}$$

$$x_3 = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{\arccos(\frac{3q}{2p}\sqrt{\frac{-3}{p}}) + 4\pi}{3}\right) - \frac{b}{3a} \tag{C.6}$$

C'est à partir de cette formulation de les racines du polynôme caractéristique de la matrice Hessienne H sont calculées dans l'implémentation en C ainsi que dans les différents portages sur GPU.

Annexe D

Conflit de banque

Des conflits de banque peuvent arriver lors de l'utilisation de la mémoire partagée d'un GPU, par exemple.

Un conflit de banque subvient lorsqu'au moins deux *threads* demandent à la même banque des mots différents. Dans ce cas les accès sont sérialisés perdant l'intérêt du parallélisme tant recherché lors de l'utilisation de GPU. Si deux *threads* ou plus demandent l'accès au même mot de la même banque, le contenu est diffusé évitant ainsi le conflit de banque. Les quatre exemples de la figure4.1 sont sans conflits, tandis que les deux exemples de la figure 4.2 sont avec des conflits de banque.

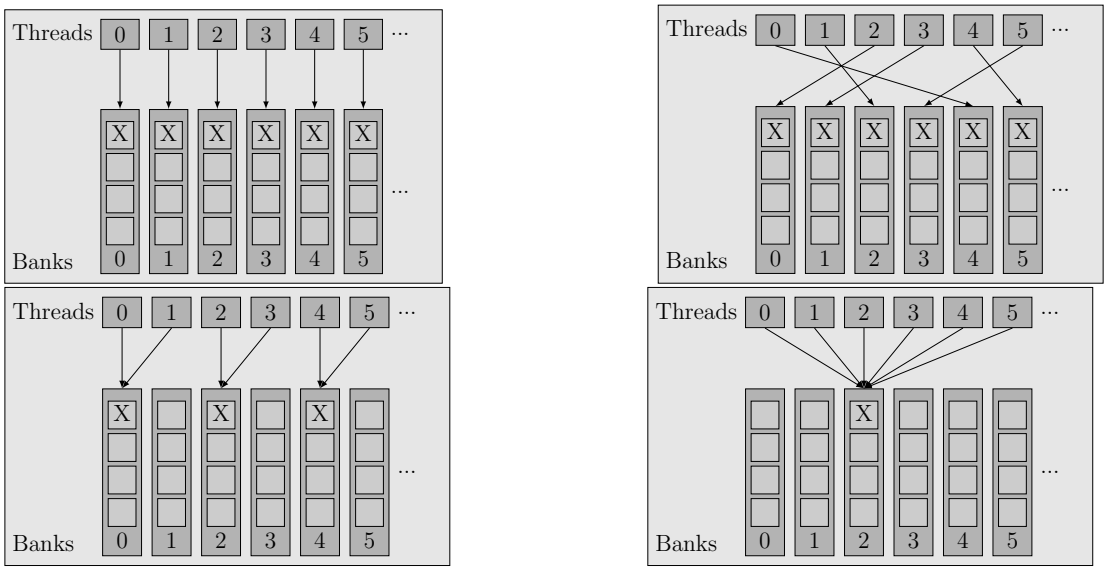


FIGURE 4.1 – Quatre exemples d'accès en mémoire sans conflit de banque

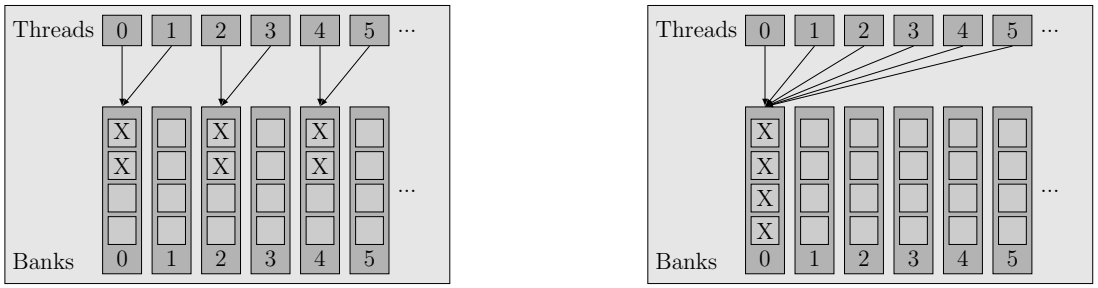


FIGURE 4.2 – Deux exemples d'accès en mémoire avec conflit de banque

L'objectif pour une utilisation optimale de la mémoire partagée est donc d'éviter autant que possible les conflits de banque. Une stratégie utilisable pour se faire s'appelle le padding.

Annexe E

Padding

Le *padding*, en français le remplissage, est une technique permettant d'éviter les conflits de banque en remplissant de manière astucieuse la mémoire. Le coût en quantité de mémoire pouvant être un élément limitant à cette technique dans certains cas, car une utilisation de mémoire supplémentaire est parfois inenvisageable. Les gains en terme de temps d'accès sont parfois significatifs car permettent de tirer profit du parallélisme propre à l'architecture des cartes graphiques.

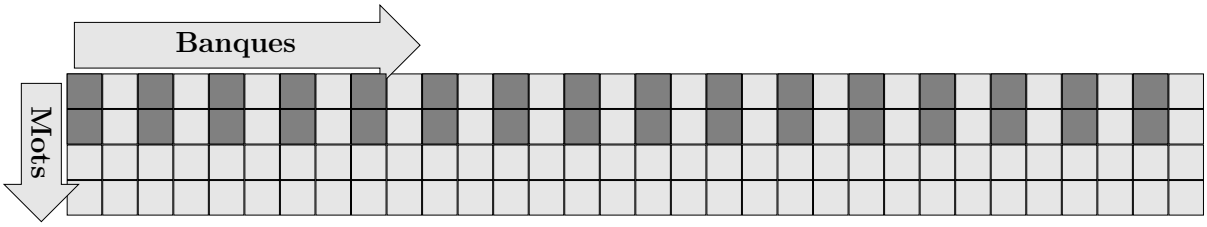


FIGURE 5.1 – Exemple de 2-way bank conflict

L'exemple présenté figure 5.1 peut arriver lors de l'utilisation de variables de type double, car ces dernières prennent actuellement deux mots en mémoire. L'accès en mémoire partagé par le biais de $shared\_memory[threadIdx.x]$ fait apparaître un conflit d'accès, par exemple, entre les threads numéro 0 et 16 sur la banque numéro 0. L'envoie des données se fera donc en deux temps, ce qui est appelé un 2-way bank conflict.

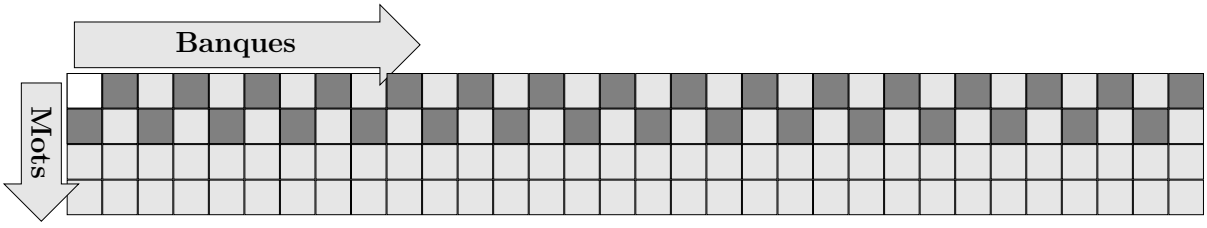


FIGURE 5.2 – Exemple de solution par padding du 2-way bank conflict

La figure 5.2 montre comment résoudre ce problème par padding. En ajoutant un mot de mémoire supplémentaire et en débutant le stockage à l'indice 1 plutôt qu'à l'indice 0. Les accès sont alors tous réalisés en parallèle évitant ainsi les conflits de banque.

Annexe F

Réduction en parallèle

Une réduction en programmation est une fonction portant sur un jeu de données et s'affaire à les réduire en un ensemble plus petit. Par exemple la somme d'un ensemble de nombre est une opération de réduction spécifique. Ce genre d'opérations peuvent être accélérées par l'utilisation de parallélisme, en particulier avec les cartes graphiques. Notons que la quantité de données à réduire doit être significative pour que l'utilisation d'une carte graphique soit efficace. En effet, il faut garder à l'esprit la nécessité de transférer les données de l'hôte au device et de récupérer, si nécessaire, le résultat.

Dans l'exemple de la figure 6.1 est illustré le principe d'une réduction réalisée par 8 threads. Dans cette exemple la réduction est une somme de 16 termes. Pour la première étape, les 8 threads réalisent la somme de deux éléments, le premier élément correspondant à l'indice du thread le second étant 8 éléments plus loin. Les étapes suivantes sont réalisées de manière similaire sur 8,4 et 2 éléments avec respectivement 4,2 et 1 threads. Notons que cette partie de la réduction constitue le point critique d'une réduction, car assimilable à la fin où le nombre d'éléments devient moins grand que le nombre de threads induisant une perte d'efficacité car certains threads sont inactifs.

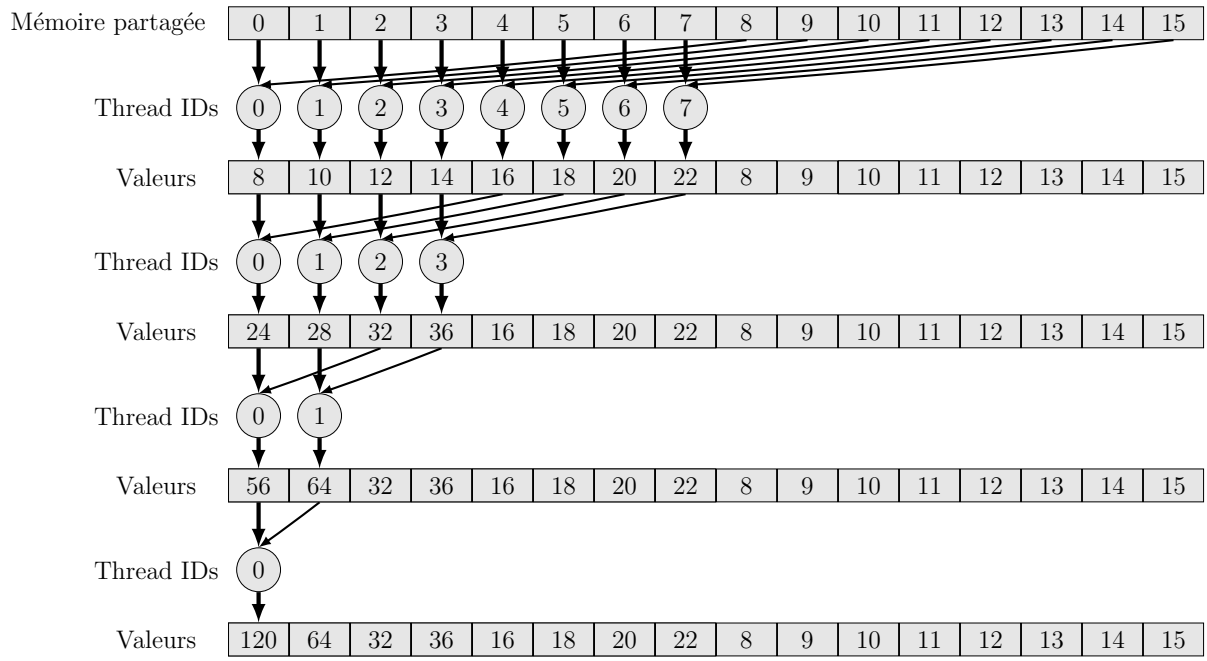


FIGURE 6.1 – Exemple de réduction

Annexe G

Dernier noyau de l'implémentation V_{NO2}

Le ratio $\frac{\rho_A}{\rho}$ permet (ligne 2) de déterminer les à traiter, diminuant ainsi le nombre de calculs. Chaque thread parcourt (ligne 3) l'ensemble des atomes qu'ils doivent traiter. Les threads calculent (ligne 4) alors pour tous leurs atomes la distance entre le noeud (x, y, z) et l'atome i courant. A partir de cette distance (ligne 5) les contributions au gradient $\nabla\rho$ et à la matrice Hessienne H sont calculées en locale par chaque thread. Chaque thread met en mémoire partagée (ligne 7) ses contributions une fois qu'il les a calculé. Tous les threads attendent (ligne 8) que la mémoire partagée soit chargée pour pouvoir réaliser (ligne 9) les opérations de réductions. Le premier thread (ligne 10) peut alors terminer le traitement du noeud. Tout d'abord le thread 0 calcule (ligne 11) les valeurs propres de la matrice Hessienne H. Le thread 0 calcule (ligne 12) ensuite la norme du gradient, puis (ligne 13) le gradient réduit de la densité ainsi que la densité signée. Pour terminer le thread 0 stocke en mémoire globale les résultats du noeud (x, y ,z).

Algorithme 10 Implémentation GPU V_{NO2}

1: **FONCTION** computeRDG
2: **SI** $0.05 < \frac{\rho_A}{\rho} < 0.95$ **ALORS**
3: **POUR** tous les atomes d'indices *IDThread*+(0, 128, 256, ...) **FAIRE**
4: Calculer la distance entre le nœud (x ,y ,z) et l'atome i courant ;
5: Calculer les contributions aux composantes du gradient $\nabla\rho$ et de la matrice Hessienne H dans les variables locales du *thread* ;
6: **FIN POUR**
7: Chaque *thread* stocke en mémoire partagée les contributions précédentes ;
8: Synchronisation des *threads* ;
9: Réductions ;
10: **SI** le premier *thread* **ALORS**
11: Calculer les valeurs propres ($\lambda_1 < \lambda_2 < \lambda_3$) de la matrice Hessienne H ;
12: Calculer la norme du gradient $\nabla\rho$;
13: Calculer le gradient réduit de la densité $s(\rho)$ et la densité signée ;
14: Stocker en mémoire globale (`cubeRDG` et `cubeRHO`) les résultats pour le nœud (x, y, z) ;
15: **FIN SI**
16: **FIN SI**
17: **FIN FONCTION**

Annexe H

Composantes du gradient et de la matrice Hessienne de la densité promoléculaire

$$\nabla\rho = \begin{pmatrix} -\sum_{i=1}^{N_{at}} \left(\frac{x-x_i}{r_i}\right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} \\ -\sum_{i=1}^{N_{at}} \left(\frac{y-y_i}{r_i}\right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} \\ -\sum_{i=1}^{N_{at}} \left(\frac{z-z_i}{r_i}\right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} \end{pmatrix} ; \; H = \begin{pmatrix} \frac{\partial^2\rho}{\partial x^2} & \frac{\partial^2\rho}{\partial x \partial y} & \frac{\partial^2\rho}{\partial x \partial z} \\ \frac{\partial^2\rho}{\partial x \partial y} & \frac{\partial^2\rho}{\partial y^2} & \frac{\partial^2\rho}{\partial y \partial z} \\ \frac{\partial^2\rho}{\partial x \partial z} & \frac{\partial^2\rho}{\partial y \partial z} & \frac{\partial^2\rho}{\partial z^2} \end{pmatrix}$$

Avec les six composantes (matrice réelle symétrique) :

$$\frac{\partial^2\rho}{\partial x^2} = \sum_{i=1}^{N_{at}} \frac{1}{r_i^2} \left[\left(\frac{(x-x_i)^2}{r_i} - r_i \right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + (x-x_i)^2 \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

$$\frac{\partial^2\rho}{\partial y^2} = \sum_{i=1}^{N_{at}} \frac{1}{r_i^2} \left[\left(\frac{(y-y_i)^2}{r_i} - r_i \right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + (y-y_i)^2 \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

$$\frac{\partial^2\rho}{\partial z^2} = \sum_{i=1}^{N_{at}} \frac{1}{r_i^2} \left[\left(\frac{(z-z_i)^2}{r_i} - r_i \right) \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + (z-z_i)^2 \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

$$\frac{\partial^2\rho}{\partial x \partial y} = \sum_{i=1}^{N_{at}} \frac{(x-x_i)(y-y_i)}{r_i^2} \left[\frac{1}{r_i} \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

$$\frac{\partial^2\rho}{\partial x \partial z} = \sum_{i=1}^{N_{at}} \frac{(x-x_i)(z-z_i)}{r_i^2} \left[\frac{1}{r_i} \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

$$\frac{\partial^2\rho}{\partial y \partial z} = \sum_{i=1}^{N_{at}} \frac{(y-y_i)(z-z_i)}{r_i^2} \left[\frac{1}{r_i} \sum_{j=1}^{3} a_{i,j} b_{i,j} e^{-b_{i,j} r_i} + \sum_{j=1}^{3} a_{i,j} b_{i,j}^2 e^{-b_{i,j} r_i} \right]$$

Accélération des calculs en Chimie théorique : l'exemple des processeurs graphiques

Nous nous intéressons aux architectures *manycore* proposées par les cartes graphiques dans le cadre de la chimie théorique. Nous soutenons la nécessité pour ce domaine d'être capable de tirer profit de cette technologie. Nous montrons la faisabilité et les limites de l'utilisation de cartes graphiques en chimie théorique par le portage sur GPU de deux méthodes de calcul en modélisation moléculaire. Ces deux méthodes peuvent potentiellement être intégrées au programme de *docking* moléculaire AlgoGen. L'accélération et la performance énergétique ont été examinées au cours de ce travail.

Le premier programme NCIplot implémente la méthodologie NCI qui permet de détecter et de caractériser les interactions non-covalentes dans un système chimique. L'approche NCI se révèle être idéale pour l'utilisation de cartes graphiques comme notre analyse et nos résultats le montrent. Le meilleur portage que nous avons obtenu, a permis de constater des facteurs d'accélération allant jusqu'à 100 fois plus vite par rapport au programme NCIplot. Nous diffusons actuellement librement notre portage GPU : cuNCI.

Le second travail de portage sur GPU se base sur GAMESS qui est un logiciel complexe de portée internationale implémentant de nombreuses méthodes quantiques. Nous nous sommes intéressés à la méthode combinée DFTB/FMO/PCM pour le calcul quantique de l'énergie potentielle d'un complexe. Nous sommes intervenus dans la partie du programme calculant l'effet du solvant. Ce cas s'avère moins favorable à l'utilisation de cartes graphiques, cependant nous avons su obtenir une accélération.

Mots-clefs : Chimie théorique, Informatique, GPU, NCIplot, GAMESS, NCI, DFTB, FMO, PCM

Speed up computations in theoretical chemistry : The example of graphics processors

In this research work we are interested in the use of the manycore technology of graphics cards in the framework of approaches coming from the field of Theoretical Chemistry. We support the need for Theoretical Chemistry to be able to take advantage of the use of graphics cards. We show the feasibility as well as the limits of the use of graphics cards in the framework of the theoretical chemistry through two usage of GPU on different approaches.

We first base our research work on the GPU implementation of the NCIplot program. The NCIplot program has been distributed since 2011 by Julia CONTRERAS-GARCIA implementing the NCI methodology published in 2010. The NCI approach is proving to be an ideal candidate for the use of graphics cards as demonstrated by our analysis of the NCIplot program, as well as the performance achieved by our GPU implementations. Our best implementation (VHY) shows an acceleration factors up to 100 times faster than the NCIplot program. We are currently freely distributing this implementation in the cuNCI program.

The second GPU accelerated work is based on the software GAMESS-US, a free competitor of GAUSSIAN. GAMESS is an international software that implements many quantum methods. We were interested in the simultaneous use of DTFB, FMO and PCM methods. The frame is less favorable to the use of graphics cards however we have been able to accelerate the part carried by two K20X graphics cards.

Keywords : Theoritical Chemistry, Computer Science, GPU, NCIplot, GAMESS, NCI, DFTB, FMO, PCM

Institut de Chimie Moléculaire de Reims
UMR CNRS 7312, URCA
Centre de Recherche en Sciences et Technologies de l'Information et de la Communication
EA 3804, URCA