Vers une simulation par éléments finis en temps réel pour le génie électrique

van Quang Dinh

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Génie Electrique**

Arrêté ministériel : 7 août 2006

Présentée par

Van Quang DINH

Thèse dirigée par **Yves MARECHAL**

préparée au sein du **Laboratoire de Génie Electrique de Grenoble** dans **l'École Doctorale Electronique, Electrotechnique Automatique & Traitement du signal**

Vers une simulation par éléments finis en temps réel pour le génie électrique

Thèse soutenue publiquement le **15 décembre 2016**, devant le jury composé de :

M. Zhuoxiang REN
Professeur à l'Université Pierre et Marie CURIE, Rapporteur
M. Yvonnick LE MENACH
Professeur à l'Université Lille 1, Président
M. Yves MARECHAL
Professeur à Grenoble INP, Directeur de thèse
M. Brahim RAMDANE
Maître de conférences à Grenoble INP, Membre

REMERCIEMENTS

Ce travail a été réalisé au sein de l'équipe MAGE du laboratoire de Génie Electrique de Grenoble (G2Elab) dans le cadre d'une bourse A.R du Ministère de l'Enseignement Supérieur et de la Recherche.

J'adresse mes plus sincères remerciements à Monsieur **Yvonnick LE MENACH**, Professeur à l'Université Lille 1, qui m'a fait le grand honneur de présider le jury de cette thèse et d'en être l'un des rapporteurs ; à Monsieur **Zhuoxiang REN**, Professeur à l'Université Pierre et Marie CURIE qui m'a également fait l'honneur d'être rapporteur de cette thèse; à Monsieur **Brahim RAMDANE**, Maître de Conférences à Grenoble INP, pour avoir accepté d'être membre du jury, et pour l'intérêt qu'ils ont manifesté à l'égard de mes travaux.

Je tiens à remercier très chaleureusement Monsieur **Yves MARECHAL**, Professeur à Grenoble INP, mon directeur de thèse, pour m'avoir proposé ce sujet de recherche très intéressant, pour ses conseils et son encadrement dynamique, sa disponibilité et pour la confiance qu'il m'a accordée tout au long de cette thèse. J'ai beaucoup appris en travaillant avec lui. Qu'il trouve ici l'expression de ma profonde reconnaissance.

Mes remerciements s'adressent également à tous les professeurs, ingénieurs de l'équipe MAGE, plus particulièrement Monsieur **Gérard MEUNIER**, Monsieur **Jean-Louis Coulomb**, Monsieur **Olivier CHADEBEC**, Monsieur **Jean-Michel GUICHON**, Monsieur **Patrice LABIE** et Monsieur **Brahim RAMDANE** pour tous leurs conseils scientifiques qui m'a permis d'effectuer mes travaux de recherche dans les meilleures conditions. Je remercie particulièrement à **Bertrand BANNWARTH** qui a dépensé sans compter beaucoup de son temps, pour ses suggestions et ses aides en matière de programmation. J'adresse également mes sincères remerciements à tous les personnels du service informatique et administratif du G2Elab pour leurs aides.

J'aimerais remercier tous les doctorants et les amis internationaux avec qui j'ai des moments précieux: **Vincent, Zaki, Diego, Douglas, Lucas, Quentin, Arnaud, Thiago, Oussama, Melissa, Jonathan, Mateus, Thomas, Lyes, Sokchea**… et la liste est longue. Mes sincères remerciements s'adressent en particulier à **Vincent** pour ses aides à résoudre multiple problèmes et ses gentillesses. J'adresse mes remerciements particuliers à mes amis Vietnamiens pour tous les moments inoubliables passés ensemble. Je remercie plus particulier A. **Trung,** A. **Sang,** E**. Binh,** A. **Phuong,** A. **Hoang Anh,** A. **Hai,** A. **Trung Son,** A. **Hoa,** A. **Phong** pour ses aides lors des premiers jours de la thèse.

J'écris les dernières lignes de cette thèse en pensant à mes parents et à ma famille. Ce travail n'aurait pas pu s'accomplir sans leurs soutiens et encouragements. Je leur dois une profonde gratitude et je les remercie pour tout.

Con xin dành đoạn cuối để cám ơn gia đình. Con cám ơn bà ngoại đã chăm sóc từ những ngày đầu con đi học ở Hà Nội, đã tin tưởng rất nhiều khi con đi du học. Con xin cám ơn cha mẹ, anh chị em đã luôn bên con, động viên, tin tưởng vào con. Con biết cha mẹ đã hi sinh rất nhiều để tạo điều kiện thuận lợi nhất có thể cho con học tập và trưởng thành. Anh xin cám ơn vợ vì tất cả những lúc khó khăn hay vui vẻ đều có em ở bên, cám ơn đã chia sẻ, cảm thông và vì tình yêu em dành cho anh.

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

INTRODUCTION GENERALE

Les ordinateurs permettent de simuler les phénomènes physiques réels complexes pour mieux les comprendre. Ces phénomènes sont modélisés par des modèles mathématiques exacts ou approchés. La résolution de ces phénomènes repose sur des méthodes numériques dédiées. Dans le domaine du génie électrique, la simulation des phénomènes physiques en électromagnétisme sont basés sur les équations de Maxwell. Il s'agit d'équations aux dérivées partielles dont les solutions intègrent également le comportement des matériaux et les conditions aux limites du domaine d'étude. Parmi les méthodes numériques disponibles, la méthode des éléments finis (MEF) est couramment utilisée pour la simulation des dispositifs électromagnétiques.

La MEF consiste à chercher une solution approchée de la solution exacte sur un espace discrétisé constitué par des éléments. Les éléments sont obtenus par une décomposition du domaine, appelé le maillage. La quantité ainsi que la qualité des éléments du maillage jouent un rôle essentiel dans la précision de la simulation. Plus le maillage est fin, plus la précision de la solution peut être élevée. Cependant, les coûts calcul qui en résultent augmentent rapidement avec la taille du maillage et les temps de simulations constituent des freins à la créativité des concepteurs, même sur des ordinateurs performants.

D'autre part, le calcul parallèle sur GPU (Graphic Processor Unit) présente de nos jours un potentiel important de performances. Le calcul sur GPU consiste à utiliser un processeur graphique en complément du CPU pour accélérer les applications de scientifiques. Le calcul sur GPU amène une capacité de parallélisation massive des tâches et ainsi, il est possible de demander au GPU d'accélérer les portions de code les plus gourmandes en ressources de calcul, le reste de l'application restant affecté au CPU. Le confort d'utilisation des codes de simulation s'en trouve amélioré.

Cette thèse s'inscrit dans le contexte de la modélisation dans le domaine de l'électromagnétisme par la méthode des éléments finis. L'objectif de la thèse est d'améliorer la performance de la MEF, voire d'en changer les stratégies d'utilisation, en profitant des performances du calcul parallèle sur GPU. En effet, si grâce aux GPUs, le calcul parvenait à s'effectuer en quasi temps réel, les outils de simulation deviendraient alors des outils de conception intuitifs, qui permettraient par exemple de faire « ressentir » immédiatement la sensibilité d'un dimensionnement à la modification de paramètres géométriques ou physiques. Un nouveau champ d'utilisation des codes de simulation s'ouvrirait alors. C'est un peu le fil conducteur de ce travail, qui tente, en abordant les différentes phases d'une simulation par la MEF, de les accélérer au maximum, pour rendre l'ensemble quasi instantané.

Ainsi donc, ce travail se compose de quatre chapitres :

Le premier chapitre présente tout d'abord le contexte et les motivations quant à l'utilisation du calcul parallèle sur GPU dans le cadre de la MEF. Les concepts de base, les terminologies et les spécifications d'un GPU seront également introduits. Nous présenterons l'architecture CUDA,

une plateforme matérielle et logicielle très utilisée, qui permet de grandement simplifier l'utilisation des GPUs pour le calcul scientifique. CUDA devrait en particulier permettre de booster de manière remarquable la simulation par la MEF.

Le deuxième chapitre présente une réflexion pour paralléliser sur GPU la phase de maillage, en utilisant pour cela un algorithme adapté, la méthode de maillage par bulles. Cette méthode est capable de donner des maillages de très haute qualité y compris sur des domaines géométriques complexes. En prenant en compte le coût calcul des étapes successives de la méthode, nous proposerons un algorithme adapté à l'architecture massivement parallèle des GPUs.

Le troisième chapitre introduit le noyau de calcul FEM dans une version massivement parallèle destinée aux GPUs. Il s'agit de prendre en compte les deux phases de la MEF, l'intégration et d'assemblage d'une part et la résolution d'autre part. Le cadre des formulations magnétostatiques sert de terrain d'expérimentation. Les algorithmes proposés dans la littérature pour l'intégration et l'assemblage parallèles sont développés dans un premier temps. Ensuite, nous proposerons une nouvelle approche d'intégration et d'assemblage plus performante, qui prend bien en compte les forces et les contraintes de ce type de plateforme. La phase de résolution des équations algébriques consiste en l'utilisation des solveurs itératifs parallélisés sur GPU. Une analyse des performances obtenues clôt ce chapitre.

Le quatrième chapitre montre une implantation parallèle de l'exploitation des résultats sur GPU, avec en particulier la construction des iso-valeurs de grandeurs scalaires. Puis, un environnement couplant CPU/GPU pour réaliser une plateforme de calcul hétérogène est décrit et sa performance est évaluée.

Enfin une conclusion générale viendra clore cette présentation, introduisant également un ensemble de perspectives qui pourraient être l'objet de prolongement de ces travaux.

CHAPITRE I

Calcul parallèle au service de la simulation numérique par la méthode des éléments finis

SOMMAIRE

Ce chapitre présente le contexte général de la thèse, à savoir la simulation par la méthode des éléments finis (MEF) en électromagnétisme. Il introduit également les motivations qui sont le fil conducteur de ce travail sur le calcul haute performance à base de GPU pour assister la simulation MEF. Dans ce chapitre, nous décrivons également l'architecture générale des GPUs et la plateforme CUDA qui est l'environnement de programmation utilisé durant cette thèse.

I.1. Contexte et motivations

I.1.1. MEF dans notre monde

La MEF est reconnue comme une méthode générale destinée à résoudre des équations aux dérivées partielles qui sont couramment utilisées dans des domaines comme la mécanique, la mécanique des fluides, la thermique et l'électromagnétisme. Dans tous les cas, un système d'équations aux dérivées partielles représente totalement le comportement du système physique : par exemple, la loi de l'élasticité en mécanique ou les équations de Maxwell pour l'électromagnétisme.

La méthode consiste à découper le domaine considéré en petites parties, appelés éléments finis et à simplifier la formulation du problème sur chaque élément, permettant de transformer le système d'équations aux dérivées partielles en un système d'équations algébriques. La procédure de discrétisation du domaine géométrique s'appelle le **maillage**, c'est un prérequis de la méthode des éléments finis.

Schématiquement, tout part de la méthode des résidus pondérés qui permet de construire une formulation intégrale à partir d'équations aux dérivées partielles. En utilisant les fonctions de forme des éléments, la formulation intégrale est discrétisée et transformée en une expression matricielle. Pour cette étape, il est classique d'introduire des matrices et vecteurs élémentaires qui sont ensuite assemblés dans la matrice globale et dans le vecteur global. En fait, les matrices et vecteurs élémentaires représentent la contribution de l'élément tandis que la matrice et le vecteur global représentent le calcul sur le domaine entier. Au final, simuler le problème physique revient à résoudre un système $Ax = b$ avec A une matrice carrée généralement creuse, b le vecteur des sources et x la vecteur des inconnues. La résolution de ce système algébrique linéaire ou non-linéaire donne la solution approchée du problème.

Accompagnant le développement des ordinateurs, la méthode des éléments finis fait partie de nos jours des méthodes numériques les plus utilisées dans la simulation des phénomènes physiques. On peut citer quelques logiciels largement utilisés dans l'industrie s'appuyant sur la MEF :

- ABAQUS : logiciel pluridisciplinaire développé par la société Dassault Systèmes
- ANSYS : logiciel pluridisciplinaire développé par ANSYS
- ASTER : logiciel pluridisciplinaire libre français développé par EDF
- COMSOL Multiphysics : logiciel pluridisciplinaire développé par Comsol
- Et bien sûr **Flux2D/3D** le logiciel de simulation MEF développé par **CEDRAT/ALTAIR** en collaboration avec le **G2Elab** permettant les calculs magnétiques, électriques ou thermiques en régimes permanents, harmoniques et transitoires, avec des fonctionnalités d'analyse multiparamétrique étendue, les

couplages circuit et cinématique. La Figure I.1 montre une simulation du champ magnétique par Flux3D.

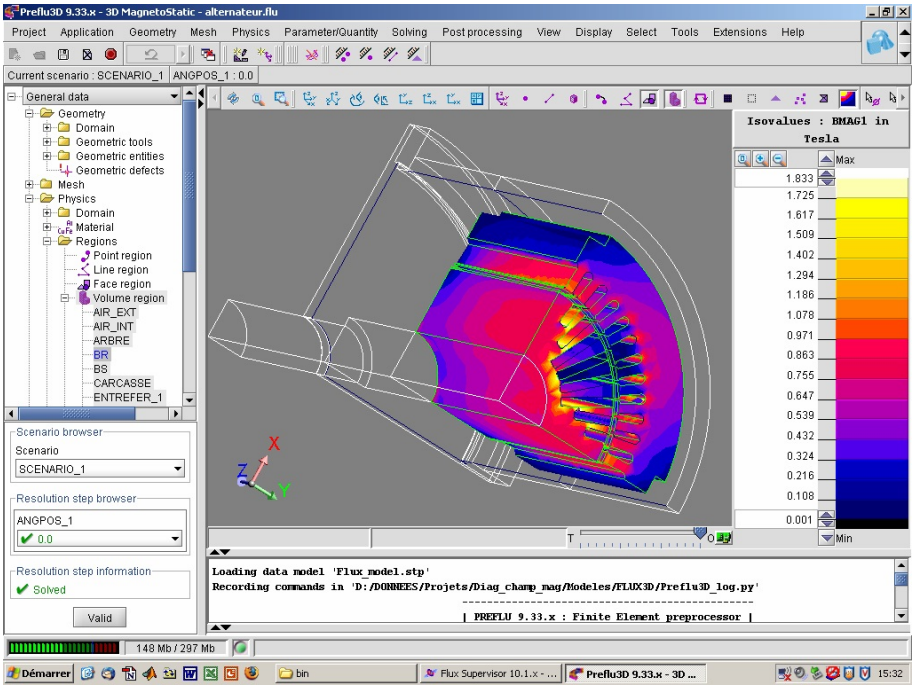


Figure I.1 Modélisation du champ magnétique par le logiciel FLUX 3D

I.1.2. Le parallélisme massif pour le calcul scientifique

Le calcul parallèle est largement répandu en informatique en général, y compris sur les ordinateurs de bureau classiques qui possèdent plusieurs cœurs capables d'agir en parallèle. Il convient de distinguer deux notions concernant le calcul parallèle : Le « parallélisme » qui repose généralement sur une plateforme informatique (par ex. un ordinateur ou un système des ordinateurs) se composant de **plusieurs** unités de processeurs conventionnels (CPU ou **C**entral **P**rocessing **U**nit) d'une part et le « parallélisme massif » qui est généralement lié à l'utilisation de **milliers** d'unités de calcul en parallèle pour exécuter une tâche unique sur des jeux de données multiples et très nombreux d'autre part. Dans ce dernier cas, les petites tâches unitaires sont appelées **thread** et leur nombre peut atteindre des millions.

Figure I.2 GPU NVIDIA Tesla C1060 possède 240 processeurs, une puissance de calcul de 933 GFLOPS en simple précision (32 bits), une bande passante mémoire de 102 GB/s

Contrairement aux super-ordinateurs couteux et encombrants, de nos jours la capacité de parallélisme existe naturellement dans tout ordinateur de bureau par le biais des cartes graphiques

avec leur processeur GPU si particulier. La Figure I.2 présente un GPU NVIDIA Tesla C1060 dont la performance de calcul arithmétique théorique est équivalant à plusieurs dizaines de fois celle d'un CPU conventionnel.

Grâce à un coût relativement modique pour une performance très conséquente, les GPUs constituent des plateformes prometteuses et intéressantes pour le calcul numérique. Désormais, le calcul parallèle sur GPU est facilement accessible et pourrait permettre de réduire drastiquement le coût calcul sur ordinateur. L'alliance CPU/GPU offre au développeur de logiciel un moyen de conduire efficacement les calculs en chargeant la part de calcul la plus lourde sur GPU tandis que le reste s'exécute toujours sur CPU.

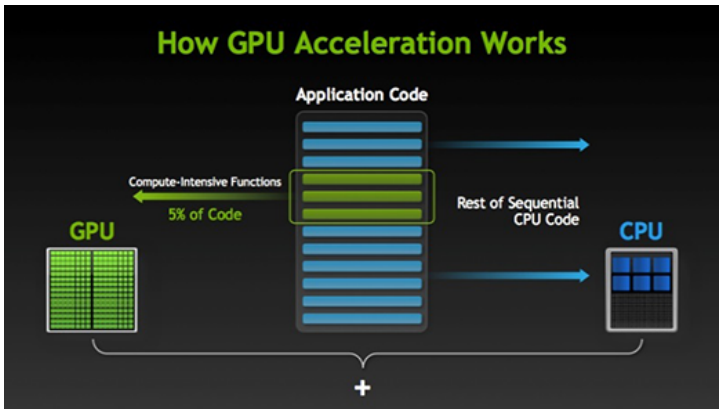


Figure I.3 Modèle de calcul hétérogène par l'alliance CPU/GPU

Les modèles de GPUs se développent rapidement. Ils offrent une croissance beaucoup plus rapide des capacités de calcul arithmétique et des vitesses d'accès à la mémoire que les modèles de CPU. La capacité de calcul se mesure par le nombre d'opérations en virgule flottante par seconde (**FLOPS**). Il en existe deux catégories principales selon le Standard IEEE-754, la simple précision (32 bits, FP32) et la double précision (64 bits, FP64). La Figure I.4 compare la performance en GFLOPS (1GFLOPS = 10^9 FLOPS) du GPU par rapport à celle du CPU au cours du temps [1]. On peut constater que tant en simple qu'en double précision, la capacité de calcul des GPU a évolué très rapidement au fil des années récentes par rapport à celle des CPU. En particulier, la performance en simple précision sur GPU dépasse de plus de dix fois celle des CPU. Ce simple rapport de performance permet de comprendre pourquoi maintenant, on commence à trouver des GPUs pour accélérer le calcul dans de nombreux domaines applicatifs.

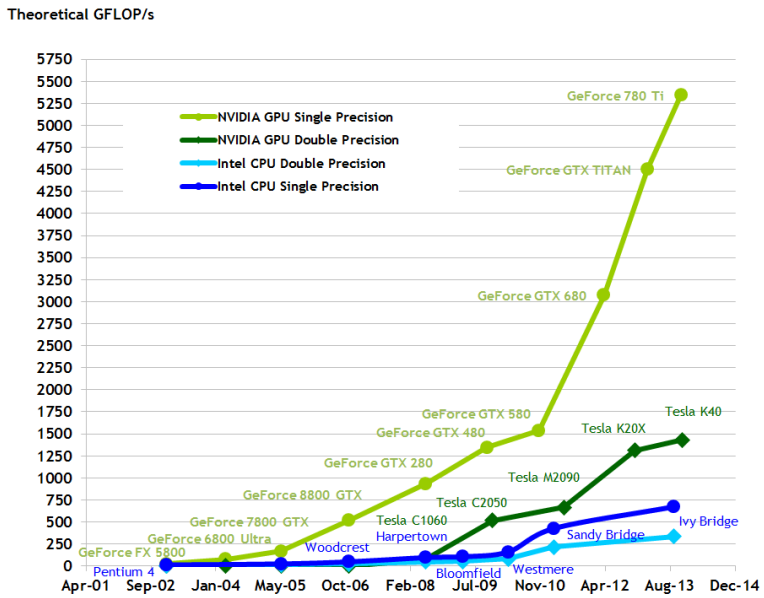


Figure I.4 Croissance de la performance brute théorique des GPU par rapport CPU

La Figure I.5 fournit une comparaison sur la vitesse d'accès de la mémoire globale entre GPU et CPU [1], cette vitesse se mesurant par la bande passante en giga bytes par seconde (**GB/s**). A nouveau, les GPUs ont un net avantage par rapport aux CPU conventionnels. Pour quantifier la comparaison, considérons une carte GPU typique avec un CPU traditionnel d'ordinateur du bureau :

- D'une part, le GPU NVIDIA Tesla C1060, qui est dotée de 30 multiprocesseurs (en anglais **S**treaming **M**ultiprocessors, SM) avec 8 cœurs de calcul par SM. Il possède donc 30 x 8 = 240 cœurs de calculs fonctionnant à la fréquence de 1,3 GHz. Le GPU est capable de d'atteindre 933 GFLOPS en simple précession (32 bits) et sa bande passante mémoire atteint 102 GB/s avec une mémoire de 4GB [2].
- D'autre part, le CPU i7-3770 d'Intel comportant 4 cœurs (ou 8 threads) à la fréquence habituelle de 3.4 GHz. Il est capable de produire 108,8 GFLOPS avec une bande passante mémoire (RAM) de 25,6 GB/s [3].

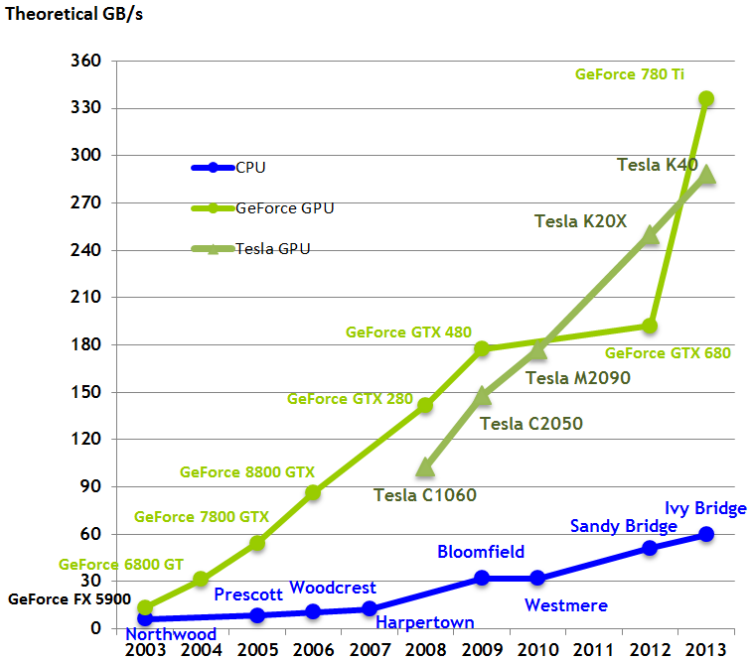


Figure I.5 Comparaison de la bande passante sur la mémoire GPU et CPU

Le GPU fournit également un moyen de calcul de plus en plus important pour la communauté de la recherche scientifique et cela explique que CUDA, introduit en 2007 par NVIDIA, soit une plateforme largement utilisée par les développeurs. Il existe désormais une communauté qui, utilisant ce langage, produit régulièrement des algorithmes qui sont parallélisés et adaptés à ce type d'architecture.

I.1.3. Motivations

La méthode des éléments finis et le calcul parallèle assisté par GPU sont des sujets attractifs pour la communauté scientifique. De fait, ils reposent sur une même approche, la subdivision. La MEF divise l'espace en éléments finis, exprime le problème physique à résoudre sur ces parties finies du domaine, puis rassemble ces informations dans le système global. De manière similaire, le calcul parallèle divise la tâche en petit travaux et les distribue vers les cœurs de calcul qui fonctionnent parallèlement et simultanément. Cette analogie de démarche permet de comprendre immédiatement le potentiel de parallélisation de la MEF sur des processeurs GPU. En pratique, on retrouve l'application du calcul parallèle à la MEF dans de nombreux articles scientifiques, des projets de recherche académique ainsi que des produits commerciaux. Il y a donc de multiples travaux visant à concevoir une implémentation MEF parallèle efficace sur plusieurs modèles d'architectures parallèles [4] [5] [6] [7] [8] [9] [10] [11].

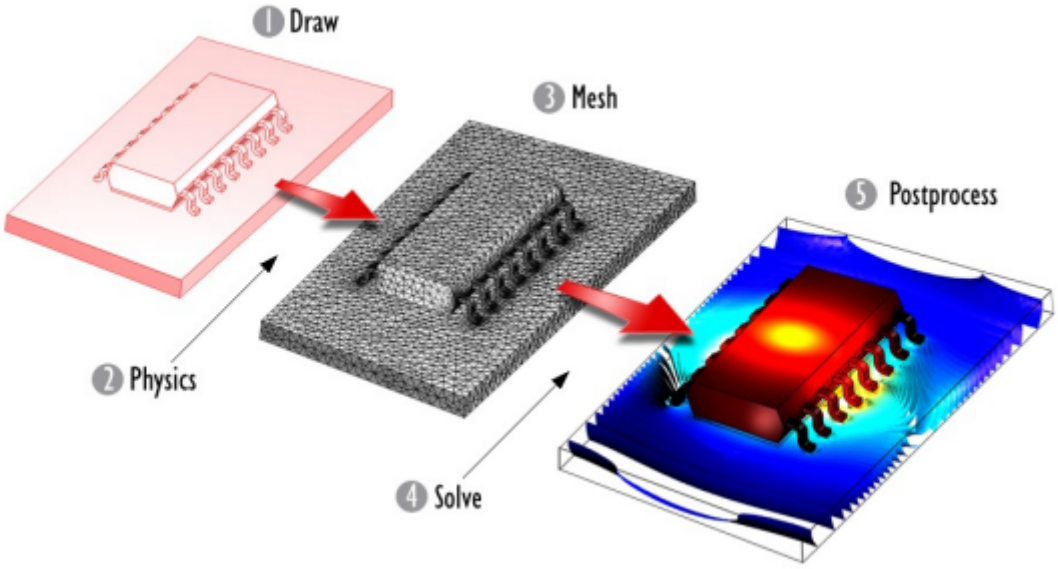


Figure I.6 Procédure pour la simulation par la MEF

La Figure I.6 présente les 5 étapes essentielles de la modélisation par la MEF :

- Construction de la géométrie
- Définition du problème physique
- Maillage
- Résolution des équations algébriques
- Exploitation ou post-traitement

Tandis que les deux premières étapes sont particulièrement interactives et ne sont que peu consommatrices de calcul, les trois dernières étapes correspondent à des besoins de calculs lourds.

La plupart des générateurs de **maillage** pour la MEF produisent des maillages non structurés et se basent sur une triangulation de Delaunay en 2 et 3D. La génération automatique de maillage non structuré se compose de deux étapes principales : **Génération des nœuds** et **triangulation de Delaunay**. La génération des nœuds doit respecter la géométrie du domaine et quelques contraintes qui conditionnent la forme et la taille des éléments, en fonction de la densité requise. En conséquence, cette procédure suit normalement des étapes successives, qui nécessitent donc un traitement séquentiel. On peut citer les quelques méthodes de génération de nœuds les plus utilisées : l'insertion de Delaunay, la méthode frontale, la méthode de Quadtree / Octree. Même s'il existe quelques travaux de recherches qui proposent des algorithmes parallèles sur des architectures CPU multi-cœurs [12], il n'existe que très peu de travaux pour les architectures GPU que ce soit en 2D [13] ou en 3D [14]. Pour ce type d'architecture, d'autres approches de maillage semblent bien mieux adaptées, en particulier les mailleurs par bulles. Cette méthode permet de générer des maillages de haute qualité en 2 et 3 dimensions et surtout, elle est quasi intrinsèquement compatible avec le calcul parallèle. Cependant, il faut noter qu'il n'existe actuellement aucun algorithme de parallélisation massive efficace pour le maillage par bulles.

La seconde étape calculatoire concerne la phase projection de la physique sur le maillage par **l'intégration et l'assemblage des contributions élémentaires** dans la matrice globale. La parallélisation massive du calcul d'intégration ne pose aucun problème particulier, par contre, pour l'assemblage, plusieurs difficultés surgissent :

- Premièrement, la dispersion en mémoire des données relatives à des éléments d'un maillage non-structuré constitue un frein à la performance, car elle entraine de nombreux chargements de mémoire.

- Deuxièmement, une condition de conflit ou de concurrence peut se produire lors de l'assemblage de milliers de calculs parallèles qui souhaitent réaliser la mise à jour simultanée de mêmes données (par ex. des termes de la matrice de rigidité).

Une approche d'assemblage par des éléments colorés a été proposée [4]. Le principe de cette méthode consiste à séparer les éléments en paquets indépendants à qui on attribue une même couleur. Couleur après couleur, les contributions des éléments sont calculées en parallèle. Bien que cette méthode réduise la dépendance entre les threads exécutés en parallèle, sa performance est limitée par la phase d'accès aux données. Plusieurs alternatives visant à regrouper les éléments dans des threads parallèles sont proposées dans [5] [15]. Cependant, la quasi-totalité de ces méthodes demande une manipulation supplémentaire, soit sur le maillage comme le coloriage des éléments, soit sur les données. Le coût de ce traitement est même potentiellement plus important que le gain de temps lié au calcul parallèle. Une nouvelle approche doit être proposée afin de réduire ce coût.

Enfin, **la résolution** s'appuie sur des solveurs pour résoudre **le système d'équations algébriques**. Parmi les deux catégories de solveurs, directs ou itératifs, les solveurs itératifs sont de nos jours les plus efficaces pour traiter les systèmes matriciels creux générés par la MEF. La version parallélisée sur GPU des solveurs itératifs se base généralement sur la famille Krylov, comme le gradient conjuguée [16], BiCG, BiCGStab [17], GMRES [18], ou AMG algébrique multi grille [5]. Le cœur d'un solveur itératif est le produit de matrice-vecteur et vecteur-vecteur, dont le calcul parallèle sur GPU est réalisé avec succès [19] [20] et est bien supporté par les librairies natives de CUDA comme Cublas, Cusparse et CUSP.

En résumé, l'usage de la plateforme de calcul parallèle CUDA pour assister la simulation par la MEF semble prometteur, en particulier pour les problèmes de grande taille. Dans le cadre de cette thèse, nous explorerons et développerons les algorithmes qui sont les mieux adaptés à l'architecture massivement parallèle des GPU, en cherchant à atteindre un haut niveau de parallélisme sans conflit de concurrence. Ces travaux se rapportent sur toutes étapes calculatoires essentielles de la MEF : Maillage, Intégration numérique et Assemblage, Résolution, Exploitation.

I.1.4. Contributions

Les contributions de cette thèse visent à proposer une simulation MEF quasi instantanée en s'appuyant sur une plateforme à base de GPU. Elle se compose de travaux sur :

- La parallélisation du maillage : nous chercherons à développer un générateur de maillage par bulles qui donne à la fois une haute qualité de maillage et une rapidité d'exécution par une implémentation parallèle efficace sur GPU.

- Le développement de l'intégration et de l'assemblage en parallèle sur GPU d'un problème physique, en particulier dans le domaine de la magnétostatique. Nous exploiterons les algorithmes d'assemblage par coloriage, par pièce (patch) et proposerons une nouvelle technique d'assemblage par tri et réduction globale.
- L'application des solveurs itératifs parallèles sur GPU en nous concentrant sur les solveurs itératifs de la famille de Krylov comme CG, BiCG, GMRES, BiCGStab avec ou sans préconditionneurs qui utilisent des opérateurs de multiplication de matrice-vecteur assistée par GPU.
- Le développement de calculs parallèles sur GPU pour booster l'exploitation des résultats.

I.2. Généralités sur le calcul parallèle sur GPU

A la base, l'exécution en parallèle d'une tâche doit diviser le temps d'exécution et ainsi donner l'opportunité de résoudre des problèmes plus grands et plus complexes dans un temps qui reste raisonnable.

En réalité, la possibilité de parallélisation d'un problème dépend de sa nature et quelques problèmes n'ont pas facilement de solution parallèle : c'est par exemple le cas des problèmes d'évolution dans le temps où le calcul de l'étape courante dépend du résultat de l'étape précédente. A contrario, il y a des problèmes qui peuvent naturellement être divisés en une multitude de petits problèmes traités en parallèle, comme par exemple, l'addition d'un ensemble de chiffres. Il existe également des problèmes qui peuvent être traités en parallèle en adoptant un algorithme spécifique, comme le problème du tri.

D'une manière générale, la parallélisation d'un problème conduit à se poser les questions suivantes : à **quel type de problème a-t-on à faire, « data-parallèle » ou « tâche-parallèle » ? Quelle architecture utiliser pour résoudre ce problème ? Quel langage informatique utiliser ? Comment évaluer l'efficacité d'un algorithme parallèle ? Comment optimiser le calcul parallèle ?** Dans cette section, nous présentons quelques concepts de base concernant les familles, la performance et l'architecture de calculs parallèles pour amener des éléments de formalisme au regard de ces questions.

I.2.1. Data-parallèle, tâche-parallèle

Définition 1 : Le calcul parallèle est l'action de résolution d'un problème de taille n en divisant son domaine en $k \geq 2$ parties pour le résoudre avec $p \geq 2$ processeurs physiques simultanément.

Nous distinguons deux catégories de calcul parallèle, **data-parallèle et tâche-parallèle,** car ils sont adaptés à deux classes de calculs parallèles. Supposons que l'on considère un problème P_D dans le domaine (ou sur les données) D. Si P_D est susceptible d'être parallélisé, le domaine D peut se décomposer par k petit domaines :

$$D = d_1 \oplus d_2 \oplus ... \oplus d_k \tag{I.1}$$

Définition 2 : Le problème est dit **data-parallèle** si la même exécution s'effectue simultanément sur les différents domaines décomposés.

$$f(D) = f(d_1) \oplus f(d_2) \oplus ... \oplus f(d_k) \quad \text{(I.2)}$$

Définition 3 : Le problème est dit **tâche-parallèle** si les exécutions différentes s'effectuent simultanément sur le domaine complet.

$$f(D) = f_1(D) \oplus f_2(D) \oplus ... \oplus f_3(D) \quad \text{(I.3)}$$

Le problème de type « data-parallèle » est bien adapté à l'architecture des GPUs car sa conception vise à maximiser la performance parallèle d'une portion identique de code sur des milliers de threads, tandis que celui de type « tâche-parallèle » est mieux adapté à l'architecture multithreads des CPU car l'architecture du CPU permet d'exécuter simultanément des tâches complexes sur ses threads parallèles.

I.2.2. Limite de performance

L'un des points importants dans le calcul parallèle, c'est d'évaluer l'efficacité de l'algorithme parallèle. En pratique, il s'agit de comparer le temps de l'implantation parallèle par rapport à celle séquentielle. Cette mesure est connue sous le terme de « speed-up » ou bien d'accélération.

La loi d'Amdahl « donne l'accélération théorique en latence de l'exécution d'une tâche à charge d'exécution constante que l'on peut attendre d'un système dont on améliore les ressources ». D'après la loi d'Amdahl, l'accélération S_p peut être formulée de la façon suivante :

$$S_p = \frac{1}{r_s + \frac{r_p}{p}} \quad \text{(I.4)}$$

où $r_s + r_p = 1$ et r_s , r_p correspondent à la fraction de code respectivement séquentielle et parallèle dans le programme de taille fixé. p représente le nombre de processeurs. Si le système possède un grand nombre de processeurs ($p \approx \infty$ comme dans un super-ordinateur ou un GPU moderne), alors **l'accélération maximale ne dépend que la portion séquentielle de l'algorithme** : $S_p = 1/r_s$ Concrètement, cela veut dire que si 90% du problème est traité en parallèle alors l'accélération maximale atteinte sera de 10 et ce malgré le nombre quasi infini de processeurs.

Au lieu de considérer un problème de taille fixée, **la loi de Gustafson** considère un problème où le temps d'exécution est fixé et où la charge par processeur est constante en augmentant le nombre de processeur p. Supposons que le temps total du problème parallélisé se compose d'un temps d'exécution du code séquentiel s et d'un temps d'exécution du code parallèle c: Le temps total est, dans le cadre parallèle, $T_p = s + c$ et celui du problème séquentiel vaut $T_s = s + cp$. Dans ce cas, l'accélération maximale est donnée par:

$$S_p = \frac{T_s}{T_p} = \frac{s + cp}{s + c} = \frac{s}{s + c} + p\frac{c}{s + c} = \alpha + p(1 - \alpha) = p - \alpha(p - 1) \quad \text{(I.5)}$$

Ainsi, **l'accélération maximale augmente linéairement avec le nombre de processeurs** lorsque que la charge par processeur reste fixe.

FLOPS (floating-point operation per second)

Le nombre de FLOPS représente la performance arithmétique brute d'un matériel, mesurée en nombre d'opérations par seconde. Notons F_t la performance théorique brute en virgule flottante du hardware et F_r la performance réelle mesurée de notre algorithme avec ce matériel, alors l'efficacité de calcul numérique dans ce cas est $E_p = F_t / F_r$ Théoriquement, la valeur $E_p = 1$ correspond à l'usage maximal de la performance de calcul arithmétique du hardware. En pratique, les GPUs possèdent généralement une performance brute en FLOPS très élevée par rapport à un CPU traditionnel mais il est rare et difficile de profiter à 100% de sa capacité à cause des limites en bande passante de la mémoire.

Bande passante mémoire

La bande passante mémoire représente la vitesse (mesurée par GB/s) de transfert des données entre les processeurs et la mémoire principale. Les CPUs modernes possèdent une bande passante maximale de 40 à 100 GB/s tandis que celle des GPUs modernes peut atteindre 200 à 300GB/s. Par exemple, le processeur Intel Xeon E5-4620 avec 16M mémoire cache à la fréquence 2.20 GHz accède à la mémoire RAM avec une bande passante maximale de 42.6 GB/s. Celle du GPU NVIDIA Tesla K20 est 208 GB/s.

Atteindre la bande passante mémoire maximale sur GPU est en réalité extrêmement difficile à réaliser du fait de la structure des données. Cela demande une organisation des données par groupe pour faciliter l'accès des threads parallèles simultanément. Les données interdépendantes, désordonnées ou les structures orientées objet réduisent la bande passante.

I.2.3. Classification

L'architecture du système matériel peut être classifiée selon la taxonomie de Flynn présentée dans le Tableau I-1.

Tableau I-1 La taxonomie de Flynn

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

Les SISD représentent les processeurs classiques qui peuvent exécuter qu'une instruction sur une unité de données à un moment. On ne parle pas de parallélisme pour ce type d'architecture.

Les SIMD appelés aussi architectures vectorielles ou à parallélisme de données peuvent traiter simultanément plusieurs jeux de données avec une même instruction. Les GPUs sont des exemples de ce type d'architecture.

Les MISD sont des architectures peu répandues qui peuvent traiter des tâches différentes sur la même base de données.

Les MIMD permettent de traiter plusieurs unités de données avec plusieurs différentes instructions en parallèle. Il s'agit d'un système hétérogène très flexible. La complexité physique de l'architecture est importante. Quelques modèles de CPU modernes entrent dans cette catégorie (Intel, AMD multi-cœurs).

De plus, selon au regard de l'organisation de la mémoire, il existe également deux architectures, la **mémoire partagée et la mémoire distribuée**.

Dans un système à **mémoire partagée**, tous les processeurs partagent une mémoire commune et l'accès aux données s'effectue à tout moment nécessairement par un module commun unique. Dans un GPU, tous les threads parallèles peuvent accéder à la mémoire globale embarquée dans la carte graphique.

Dans l'architecture à **mémoire distribuée**, chaque processeur possède pour lui-même une mémoire propre et indépendante, les processeurs communiquent entre eux par message à l'aide d'un réseau. Un serveur de calcul comportant des ordinateurs classiques connectés par le réseau est un exemple pour ce type d'approche.

I.3. Plateforme CUDA

Le terme GPGPU (General-Purpose Computing on Graphics Processing Units) désigne un système hétérogène composé d'un processeur traditionnel (CPU) assisté par des processeurs graphiques (GPU) afin de bénéficier de leur capacité de traitement massivement parallèle. CUDA (Compute Unified Device Architecture) est une plateforme de GPGPU.

NVIDIA a présenté CUDA pour la première fois en 2006. La GeForce 8800 GTX est le premier GPU construit avec l'architecture CUDA. De nos jours, les GPUs qui supportent CUDA possèdent une capacité de calcul toujours plus importante. CUDA est devenu un standard industriel pour le calcul de haute performance. La différence entre GPU et le processeur traditionnel CPU porte sur deux aspects :

- Le nombre des microprocesseurs intégrés sur GPU varie entre quelques centaines et des milliers tandis sur CPU, il est de l'ordre de quelques unités.

- La mémoire sur GPU est hiérarchisée pour répondre aux besoins du programmeur tandis qu'il s'agit d'une notion quasi implicite pour un CPU.

Un GPU est constitué de plusieurs **multiprocesseurs** (au moins de 2), appelé Streaming Multiprocessor ou **SM**. Chaque SM est doté d'un nombre fixé des microprocesseurs, appelés les **processeurs CUDA.** Chaque microprocesseur joue le rôle d'une unité de calcul. Le nombre des processeurs CUDA par SM varie selon le modèle de GPU et tend à augmenter au fil du temps. La Figure I.7 illustre l'architecture d'un multiprocesseurs de GPU selon les modèles Tesla (1^{er} génération), Fermi (2^{eme} génération) et Kepler (3^{eme} génération), qui contiennent de 8, 32 (ou 48) et 196 processeurs CUDA respectivement.

GPU Tesla GPU Fermi GPU Kepler

Figure I.7 Schéma d'un multiprocesseur SM (Streaming Multiprocessor) de GPU NVIDIA selon le modèle Tesla, Fermi, Kepler

Dans la classification des architectures parallèles, CUDA est une plateforme **SIMT** (single instruction multiple threads) avec une **mémoire hiérarchique et partagée** à plusieurs niveaux par ses nombreux processeurs.

Le code CUDA est accessible nativement en C/C++ et en Fortran. De nos jours, seuls les GPUs NVIDIA supportent CUDA. Cependant, le développement rapide des GPU amène un nouveau standard ouvert appelé OpenCL destiné à devenir un langage commun pour toute plateforme GPU mais également pour les CPU multi-cœurs. Concernant la partie GPU, son modèle de programmation est similaire à CUDA.

I.3.1. Terminologies CUDA

I.3.1.1 Kernel CUDA

Le **kernel CUDA** est une fonction ou une portion de code parallèle à exécuter sur le GPU, caractérisé par le mot-clé **__global__** et stockée dans un fichier spécifique **.cu**.

- Un kernel est normalement écrit en langage C/C++.
- Le kernel est compilé par le compilateur spécifique : **nvcc**, abréviation de « NVIDIA CUDA Compiler ».
- Le lancement d'un kernel est réalisé par le **CPU**, qui est appelé **hôte**.
- L'exécution du kernel s'effectue sur le **GPU**, qui est appelé **périphérique**.
- Le lancement d'un kernel suit une syntaxe spéciale de la forme :

kernel <<< (paramètres du kernel) >>> (données du kernel)

- Lors de son lancement, le code du kernel CUDA est multiplié à de nombreuses instances. Chaque instance du kernel est un **thread** CUDA

Il existe 3 types de fonctions spécifiques CUDA que l'on distingue par leur préfixe :

__global__ correspond à un kernel exécuté par le GPU et appelé par le CPU.

__device__ correspond à une fonction exécutée et appelée par le GPU. Normalement, c'est une partie d'un __global__ kernel.

__host__ correspond à une fonction exécutée et appelée par le CPU. Il est utilisé en combinant avec __device__ (et pas avec __global__) pour définir une fonction qui peut être exécutée sur GPU et aussi sur CPU.

I.3.1.2 Modèle de programmation

Le lancement d'un kernel CUDA s'appuie sur un espace de calcul qui est gérée selon trois niveaux : **grille**, **bloc** et **thread** comme illustré par la Figure I.8. Il s'agit une spécification du modèle de programmation de CUDA. Une grille est constituée des blocs, tandis que chaque bloc se compose de threads. Le thread est l'unité de calcul de CUDA. Tous les threads effectueront un calcul identique qui est décrit dans le code du kernel correspondant.

La structure grille–bloc–thread est directement en lien avec l'architecture physique du GPU :

- Chaque **grille** correspond au lancement d'un kernel CUDA sur un **GPU**.
- Chaque **bloc** de la grille est lancé sur un **multiprocesseurs** SM disponible du GPU.
- Chaque **thread** du bloc est exécuté par un **processeur** CUDA du SM.

Il est à noter que les SMs du GPU fonctionnent d'une manière indépendante, donc les blocs CUDA sont également indépendants. La coopération entre blocs CUDA est impossible. Contrairement aux blocs, les threads dans un bloc peuvent coopérer les uns avec les autres si nécessaire car ils partagent des ressources du SM (par ex. la mémoire). La communication entre les threads CUDA est un élément clef dans un scénario d'optimisation du CUDA.

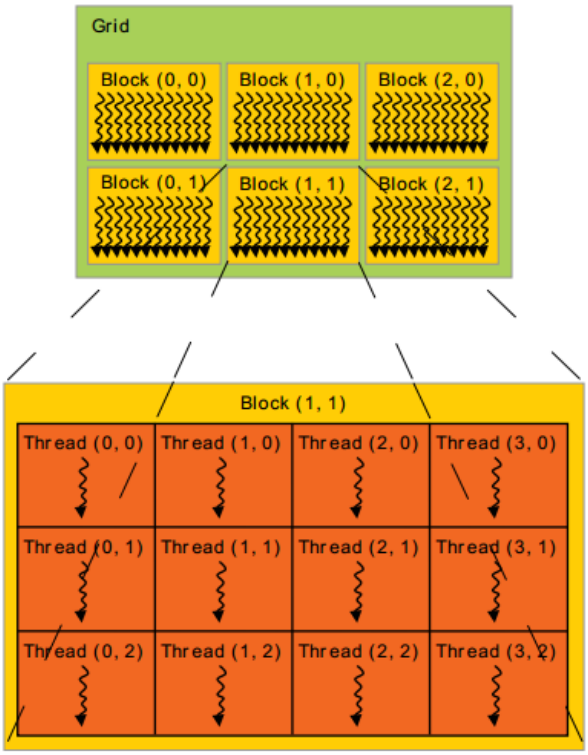


Figure I.8 Structure hiérarchie de la modèle de calcul parallèle de CUDA

I.3.1.3 Paramètres de lancement d'un kernel

En rappelant le lancement d'un kernel CUDA « __global__ » dans la syntaxe:

kernel <<< gridDim, blockDim, sharedMemSize, streamCUDA >>> (données du kernel)

Les paramètres suivants sont requis :

- gridDim : ce paramètre définit la dimension et la taille de la grille. Il est de type dim3, ça veut dire 3 composants gridDim.x, gridDim.y, gridDim.z. Le nombre des blocs de la grille est le produit de gridDim.x * gridDim.y * gridDim.z. En particulier, gridDim.z doit valoir 1 pour les GPUs de capacité de calcul 1.x.
- blockDim : ce paramètre précise la dimension et la taille de chaque bloc. il est de type dim3 avec trois composantes blockDim.x, blockDim.y, blockDim.z. Le nombre des threads du bloc est égal à blockDim.x * blockDim.y * blockDim.z et identique pour tous les blocs.
- sharedMemSize : ce paramètre est facultatif par défaut. Il montre la taille de la mémoire partagée (pour les variables avec __shared__) utilisée par le kernel, mesuré par le nombre de bytes. Ce paramètre est impératif dans le cas de l'utilisation de la mémoire partagée dynamique du kernel.
- streamCUDA : ce paramètre facultatif spécifie le CUDA « stream » au lequel le kernel s'associe.

Le choix des paramètres du kernel CUDA dépend essentiellement du fonctionnement du kernel, du scénario d'optimisation du calcul mais aussi de limites du matériel.

La dimension de la grille est limitée par la capacité matérielle spécifique du GPU. Si le paramètre **gridDim** dépasse cette limite, le lancement du kernel échoue. De manière similaire, le nombre de threads dans un bloc (cf. **blockDim**) dépend des ressources utilisées par le kernel et des ressources du SM disponibles. En fait, le SM alloue à chaque thread une quantité de registres et de mémoire partagée pour son exécution, ces ressources étant rendues au SM au moment où le thread termine son calcul. Les ressources totales de chaque SM sont fixes et dépendent de la version du GPU. Ainsi, on comprend que plus un thread utilise de ressources, moins le nombre de threads lancés simultanément est important. Si la quantité de ressources utilisées dépasse la limite disponible pour le SM, les variables du kernel sont allouées sur la mémoire locale temporaire dont l'accès est très lent. Pour faciliter le choix du paramètre **blockDim**, le nombre total de registres et la quantité totale de mémoire partagée allouée pour un bloc sont documentés dans l'outil de **« CUDA Occupancy Calculator »** fourni dans « **CUDA Software Development Kit** ».

Le SM découpe les threads du bloc par paquets de 32 **threads** (appelé **warp)** qui s'exécutent simultanément à un moment. Un warp se compose toujours de **32 threads quel que soit la version de GPU** et même si le bloc est composé de milliers threads. Le SM s'occupe de deux tâches à la fois : pousser les nouveaux warps dans une pile et exécuter une ou plusieurs warps simultanément jusqu'à l'épuisement de tous les warps. Donc pour optimiser son code, le programmeur doit choisir le paramètre **blockDim** tel qu'il soit un multiple de 32.

Prenons l'exemple du calcul de l'addition de deux vecteurs de taille N de nombres en virgule flottante « float* », le résultat sera stocké dans un troisième vecteur. L'exécution normale sur CPU effectue simplement une boucle de calcul selon le code ci-dessous :

```
// CPU code
void vectorAddHost(const float *A, const float *B, float *C, int N)
{
    for(int i=0; i<N; i++)
       {
              C[i] = A[i] + B[i];
       }
}
```

Avec le calcul parallèle sur GPU, nous divisons le calcul en N opérations dont chacune correspond à un calcul sur un élément. Le code parallèle est donc similaire à tous N threads parallèle. Le **kernel** CUDA C est le suivant :

```
// CUDA Kernel Device code
__global__ void
vectorAdd(const float *A, const float *B, float *C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < N)
    {
        C[i] = A[i] + B[i];
    }
}
```

Il est à noter que l'indice du thread « i » sert à désigner l'élément traité. Le **lancement** du kernel a la syntaxe suivante :

```
// lancement du CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid =(N - 1) / threadsPerBlock + 1;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

Les paramètres ci-dessous sont choisis par le programmeur selon son schéma d'optimisation :

« `threadsPerBlock` » est le nombre de threads par bloc, choisi à 256 (multiple de 32) par l'outil calculateur d'occupation de CUDA.

« `blocksPerGrid` » est le nombre de blocs qui dépend de la taille du vecteur et du nombre de threads par bloc.

« `d_A, d_B, d_C, N` » sont les données du kernel, les trois vecteurs et la taille/le nombre d'éléments du vecteur. Le mot « d_ » signifie que les trois vecteurs sont initialisés sur la mémoire de GPU.

I.3.1.4 Condition de concurrence

Il s'agit d'une situation de conflit entre les threads CUDA lorsque plusieurs threads modifient une même valeur en mémoire au même moment. Le résultat dans ce cas est inconnu car la valeur finale dépend de l'ordre précis des threads. En conséquence, la validité des données n'est pas assurée. C'est pourquoi, la synchronisation ou l'opération atomique sont nécessaires pour éviter cette condition.

I.3.1.5 Synchronisation des threads du bloc

La **Synchronisation** est une action de coopération entre les threads d'un bloc pour synchroniser l'exécution qui vise à coordonner l'accès à la mémoire. Un thread déclare un point de synchronisation dans un kernel par le mot-clé **__syncthreads()**. Dans ce cas, la synchronisation fonctionne comme une barrière virtuelle que tous les threads doivent atteindre avant de continuer. A noter que **__synchthreads()** n'agit que sur les threads d'un même bloc. C'est le mécanisme ad-hoc pour manipuler les données sur la mémoire partagée.

I.3.1.6 Opération « Atomique »

Une opération **atomique** signifie une action d'un thread qui permet de lire-modifier-enregistrer un mot de 32 bits ou 64 bits en une seule opération. Contrairement à la synchronisation, la fonction atomique a une portée soit globale pour les threads de tous les blocs, soit locale dans un bloc. Par exemple, la fonction **atomicAdd()** lit un nombre existant sur la mémoire globale ou mémoire partagée, additionne avec un autre nombre, et enregistre le résultat sur la même adresse. En conséquence, aucun autre thread ne peut accéder à cette adresse jusqu'à ce que l'opération soit terminée et ainsi la fonction atomique rend l'exécution séquentielle dans le cas où deux threads ou plus tentent d'accéder à cette adresse à la fois. C'est pourquoi, l'usage abusif des fonctions atomiques peut réduire considérablement la performance du kernel CUDA. Cependant, l'opération atomique fournit un outil de synchronisation sur la mémoire globale et est indispensable dans certains cas. En pratique, tous les GPUs ne donnent pas accès aux opérations atomiques, cela dépend de la capacité de calcul CUDA et aussi de la mémoire affectée.

I.3.1.7 Capacité de calcul

La **capacité de calcul** d'un GPU représente ses caractéristiques et ses spécifications techniques. La capacité de calcul s'exprime par 2 chiffres, dits majeur et mineur. Le chiffre majeur représente son architecture et le chiffre mineur exprime la présence de caractéristiques supplémentaires de cette version de GPU par rapport son architecture standard. Par exemple l'architecture Tesla a une capacité 1.x (1.0, 1.1, 1.2, 1.3), quant à l'architecture Fermi, elle a une capacité 2.x (2.0, 2.1)… Chaque GPU possède donc une capacité de calcul spécifique.

I.3.2. La hiérarchie des mémoires

La mémoire d'un GPU se compose de plusieurs niveaux qui se caractérisent par leur accessibilité, leur vitesse d'accès et leur volume [1]. La Figure I.9 présente la distribution des mémoires **globale**, **partagée** et **locale,** avec, en vis-à-vis, la structure hiérarchique des structures de calculs du GPU.

- Chaque **thread** possède ses variables **propres** qui sont réservées soit dans la mémoire de **registres** soit dans la **mémoire locale**.
- Tous les threads d'un **bloc** peuvent accéder à une **mémoire partagée** du SM.
- Tous les threads d'une grille peuvent accéder la **mémoire globale** du GPU. Chaque type de mémoire est optimisé pour un usage spécifique.

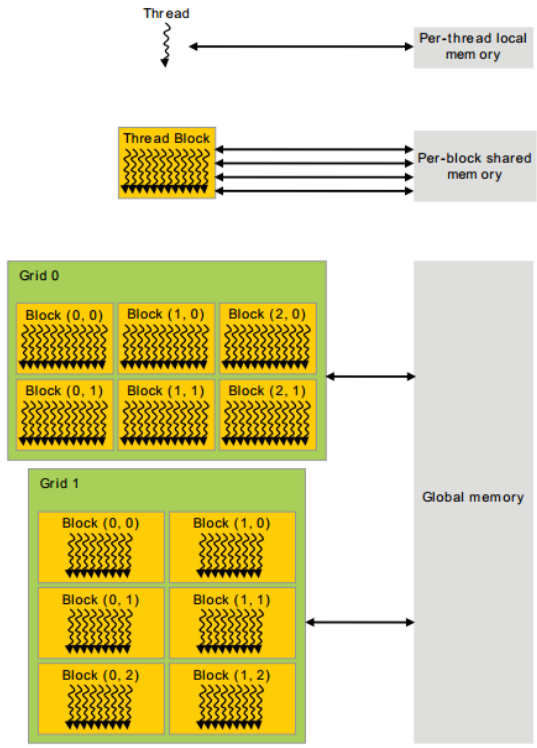


Figure I.9 Distribution de la mémoire correspond au niveau de calcul grille-bloc-thread

La **mémoire globale** est la mémoire principale du GPU et indépendante de la mémoire de CPU. La mémoire globale est accessible pour tous les kernels CUDA avec une latence de lecture/écriture d'environ **400 à 600 cycles**. Pendant ce temps d'accès, le calcul sur le SM est inactif. C'est la mémoire la plus importante en volume (quelques **GB**) mais aussi la plus lente parmi toutes les mémoires du GPU. L'accès mémoire se base sur des transactions de **32, 64** ou **128** bytes, ce qui revient à dire que l'accès mémoire est de type coalescent (en groupe). Quand un **warp** (32 threads) accède à une séquence sur la mémoire globale, l'accès est regroupé dans une ou plusieurs transactions en fonction de la taille des données demandées par thread et la localisation des données sur la mémoire. Ce mode d'accès est abordé dans le paragraphe I.3.5.

La **mémoire partagée** est la mémoire intégrée sur le chipset de chaque SM du GPU, ce qui lui permet d'être beaucoup rapide que la mémoire globale, mais son volume est nécessairement limité (quelques **dizaines** de **KB**). La bande passante élevée de la mémoire partagée vient de sa structure composée de plusieurs modules de même taille, appelés **banques**. Les banques sont organisées pour que des mots successifs de 32 bits soient assignés à des banques successives. Les banques peuvent être accédées simultanément. Par exemple, le GPU Fermi se compose de 48KB de mémoire partagée comportant 32 banques de 32 bits ayant la bande passante de **deux cycles** d'horloge. Les variables sont assignées à la mémoire partagée en utilisant le mot-clé **__shared__** dans leur déclaration.

Le **registre** est une mémoire sur le chipset du SM. La vitesse d'accès est donc très rapide avec un seul cycle par instruction. Chaque SM possède quelques dizaines de KB de registres 32-bits. Chaque thread possède ses propres registres et ne les partage pas avec d'autres threads. Le nombre maximal de registres par threads est également limité (par ex. 128 registres par threads pour un GPU Tesla 1.x, 63 registres pour un GPU Fermi).

La **mémoire locale** est un emplacement spécial de mémoire conçue pour contenir les registres virtuels (en anglais « spilled register »). Les registres virtuels apparaissent lorsqu'un thread CUDA requiert plus de registres que ce qui est disponible dans le SM. Les registres « virtuels » ont une **latence très élevée** contrairement aux registres « normaux ». En règle générale, une variable réside automatiquement dans un registre « normal » à l'exception des quelques cas suivant :

- Un tableau initialisé de manière dynamique dont le compilateur ne peut pas déterminer la taille.
- Un grand tableau qui consomme trop de stockage de registres.

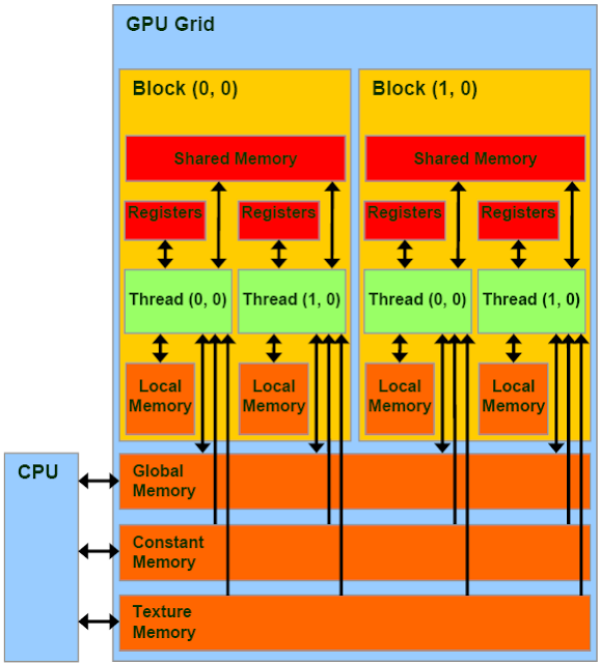


Figure I.10 Accessibilité de mémoire hiérarchie du GPU. Les flèches en double sens signifient la lecture et aussi l'écriture, alors les flèches de sens unique ne signifient que la lecture

La **mémoire constante** réside sur la mémoire globale du GPU mais elle utilise un cache dédié qui en facilite l'accès. On dispose de 64KB de mémoire constante en utilisant le mot-clé **__constant__**. Comme son nom indique, elle est non modifiable par les threads, mais modifiable par l'hôte (CPU) (cf. Figure I.10). L'avantage de la mémoire constante, c'est que la lecture ne coûte qu'un cycle par instruction si tous les threads d'un warp accèdent à la même adresse. En revanche, l'accès à des adresses différentes par les threads est sérialisé, donc le coût augmente linéairement avec le nombre d'adresses différentes. Elle est à privilégier pour diffuser **une donnée** aux multiples threads.

La **mémoire texture** possède aussi un cache spécifique et le coût de lecture est donc très faible. Cette mémoire est optimisée pour des données à deux dimensions comme c'est le cas dans le traitement des images. Les threads d'un même warp qui lisent des informations à des adresses de texture proches l'une de l'autre auront des performances optimales. La mémoire texture est à privilégier dans le cas d'un besoin important de lecture de données qui sont rarement mises à jour, ou le besoin de visualiser les données et traiter à la fois.

I.3.3. Intégration du code avec CUDA

Il existe 3 approches de base pour tirer parti de l'accélération du GPU avec CUDA :

- **Intégrer** les packages de bibliothèque ou libraires optimisés de CUDA.

Les packages et libraires de CUDA peuvent facilement être intégrées. On peut en particulier citer ici quelques bibliothèques qui seront très utilisées dans la thèse :

Thrust est une puissante bibliothèque d'algorithmes parallèles et de structures de données. Thrust fournit une interface de programmation flexible, de haut niveau, qui améliore considérablement de la productivité des développeurs.

cuBLAS (CUDA Basic Linear Algebra Subroutines) cette bibliothèque est une version accélérée pour GPU de la très complète bibliothèque BLAS classique qui offre de 6 à 17x plus de performance que la dernière MKL BLAS.

cuSPARSE (Sparse Matrice NVIDIA CUDA) Cette bibliothèque fournit une collection de sous-programmes d'algèbre linéaire de base utilisés pour les matrices creuses qui fournit jusqu'à 8x plus de performances que la dernière MKL.

CUDPP est une bibliothèque d'algorithme parallèle de fonctions de base telles que le préfixe-somme parallèle («scan»), le tri en parallèle, et la réduction parallèle.

CUSP est une bibliothèque pour les calculs d'algèbre linéaires creux et de graphes à base de **Thrust**. Cusp fournit une interface flexible de haut niveau pour la manipulation de matrices creuses et la résolution de systèmes linéaires creux.

- **Utiliser les compilateurs automatiques** pour paralléliser le code, comme OpenAcc.
- **Programmer** une portion de code parallèle **(kernel CUDA)** en utilisant l'extension en langage C/C+ et Fortran. De plus, le programmeur en Python ou Java peut profiter CUDA via des librairies supplémentaires comme pyCUDA pour Python, JCuda via JNI (Java native interface) pour Java.

I.3.4. Evaluation des performances du kernel CUDA

Pour développer un kernel CUDA et optimiser le code, il est nécessaire de comprendre la façon de mesurer correctement la performance du kernel. Normalement, un problème parallélisé sur GPU est limité soit par la bande passante, soit par la capacité de calcul arithmétique. Il faut donc comprendre comment atteindre au mieux ces limites.

Timing

La durée d'exécution d'un kernel peut être mesurée par soit par des chronomètres **CPU**, soit par des chronomètres **GPU**. Les développeurs doivent également être conscients des problématiques de **synchronisation** de CPU-GPU car de nombreuses fonctions CUDA sont asynchrones, ce qui veut dire que ces dernières rendent le contrôle au CPU avant de terminer leur travail. En particulier, tous les lancements du **kernel** sont **asynchrones**. Par conséquent, pour mesurer avec précision le temps écoulé pour un appel ou une séquence particulière d'appels

CUDA, il est nécessaire de **synchroniser** le thread CPU avec le GPU avant le démarrage et l'arrêt du chronomètre.

Analyse de bande passante

L'analyse de la bande passante est indispensable dans un problème qui est limité non pas par la performance arithmétique du processeur, mais par la lecture ou l'enregistrement des données. La bande passante, en **GB/s**, est mesurée par le débit d'accès à la mémoire globale (en **GB**, 1GB = 10^9 bytes) pendant le temps d'exécution du kernel (en **seconde**). La durée d'exécution du kernel est obtenue par le **timing** ci-dessus. La bande passante représente l'efficacité de lecture et d'enregistrement d'un kernel sur la mémoire globale. En comparant deux kernels différents, la bande passante effective est une métrique permettant de mesurer la performance et l'optimisation de l'implantation de chaque kernel. Idéalement, le kernel approprié possèdera la bande passante plus proche de la valeur théorique du GPU.

La bande passante effective d'un kernel CUDA se mesure par

$$\left(B_r + B_w\right) \div 10^9 \div t$$

avec

B_r , B_w: le total en *bytes* des lectures et des enregistrements effectués sur la mémoire globale

t : la durée de temps d'exécution du kernel (en *seconde*)

Analyse de GFLOPS

Contrairement à l'analyse de la bande passante, l'analyse des GFLOPS sert pour un problème limité par la capacité de calcul arithmétique du GPU. Il s'agit de la mesure du nombre d'opérations en virgule flottante (1 GFLOP = 10^9 FLOP) en fonction du temps (seconde). Les GLOPS représentent la performance réelle du GPU lors de l'exécution d'un kernel. En comparant deux kernels correspondant à deux implantations différentes, le kernel à conserver est celui qui possède les GFLOPS les plus grands et les plus proches de la valeur maximale du GPU.

Prenons un exemple d'un kernel CUDA pour le calcul SAXP comme suit :

```
// kernel CUDA for SAXP y(i) = a*x(i) + y(i)
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
       y[i] = a*x[i] + y[i];
}
```

On constate que SAPX manipule 12 bytes par élément (3 float avec 1 float = 4 bytes) tandis que le calcul est de 2 opérations (une addition et une multiplication = 2 flops). Le problème est clairement limité par la bande passante car le ratio de flops / bytes dans ce cas égal à 1/6 ce qui est beaucoup plus petit que les dizaines ou centaines des GPU. Dans ce cas, l'analyse de la bande passante est à privilégier pour optimiser la performance.

En pratique et dans nombreux cas, l'évaluation des GLOPS est plus difficile que la bande passante. Heureusement, la plupart des problèmes sont limités par la bande passante mémoire.

Ces deux indicateurs peuvent être évalués par un profileur CUDA adapté comme « NVIDIA Visual Profiler ». Cependant, dans certains cas comme le portage de code jCuda entre JAVA et CUDA C/C++, ces évaluations sont effectuées par une approximation théorique. L'outil le plus efficace dans ce cas pour évaluer un kernel est le timing. L'optimisation du kernel se base sur une comparaison du temps d'exécution de deux versions du kernels : avant et après son optimisation.

I.3.5. Optimisation du kernel CUDA

L'optimisation de la performance d'un **kernel** se construit autour de trois stratégies de base :

- **Maximiser l'exécution en parallèle** pour atteindre une utilisation maximale. En effet, le calcul doit être structuré de manière à ce qu'il utilise autant que possible le parallélisme et le distribue aux différents cœurs de calcul pour les tenir occupés la plupart du temps.

- **Optimiser l'utilisation de la mémoire** pour obtenir le débit maximum de mémoire. L'objectif est de minimiser les transferts de données à faible bande passante entre l'hôte et le périphérique et de maximiser l'utilisation de la mémoire on-chip comme la mémoire partagée et les caches.

- **Optimiser le choix des instructions** pour obtenir le débit d'instruction maximal. L'objectif est de minimiser l'utilisation des instructions arithmétiques à faible débit (par exemple, la division est plus lente que la multiplication) et, en tenant compte du compromis entre la précision et la vitesse, de faire un bon usage de la simple précision au lieu de la double précision. Il faut également minimiser les chaînes divergentes qui sont liées à des conditions « if » et réduire le nombre d'instructions.

I.3.5.1 Accès coalescent à la mémoire

L'optimisation de mémoire est la priorité principale pour atteindre la performance maximale du kernel CUDA. Il faut rappeler bien sûr l'usage efficace de la hiérarchie des mémoires du GPUs et la coalescence de la mémoire globale en tant que première priorité. Grâce à la mémoire globale, l'accès (lecture ou enregistrement) des threads dans un warp peut être réalisé en quelques transactions si les données sont **contiguës** [1], comme illustré dans la Figure I.11. Autrement dit, un accès non coalescent qui aura lieu avec des données fragmentées, mène à une utilisation inefficace et lente de la mémoire et en conséquence, à une performance très dégradée.

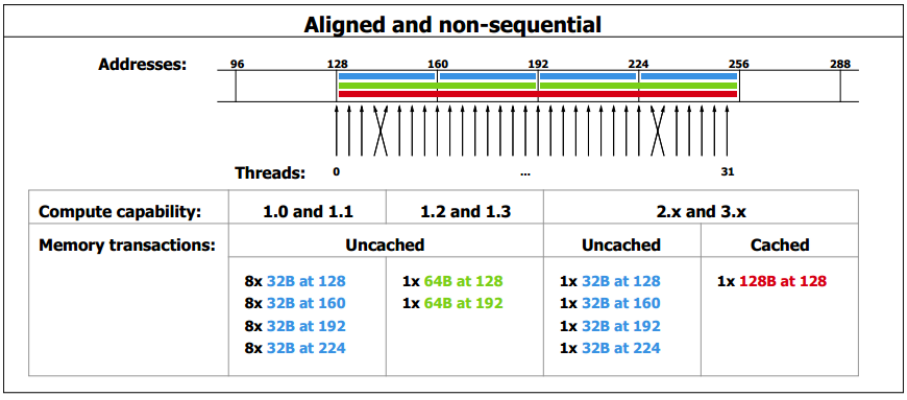


Figure I.11 Accès coalescent sur la mémoire globale selon la capacité de calcul du GPU

I.3.5.2 Conflit de la mémoire partagée

La mémoire partagée a une bande passante beaucoup plus élevée et une latence plus faible que la mémoire locale et globale s'il n'y a pas de conflits entre ses banques [1]. L'accès efficace est montré par la Figure I.12. Le conflit aura lieu si plusieurs threads accèdent simultanément à une banque de mémoire. Dans ce cas, ces accès sont mis en série et cela diminue la bande passante.

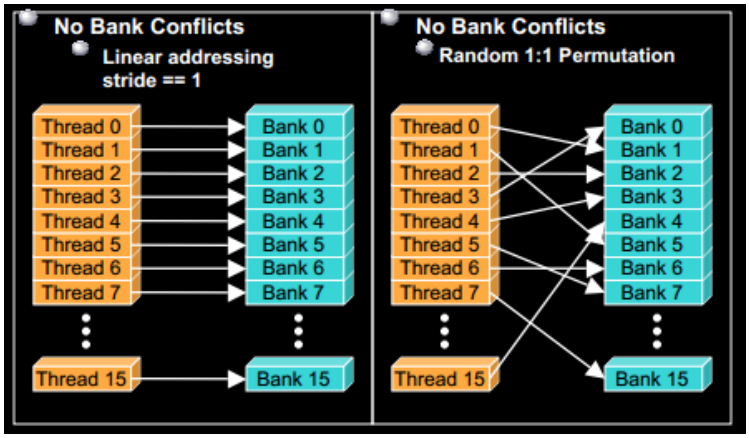


Figure I.12 Accès sans conflit sur les banques de mémoire partagée

I.3.5.3 Utilisation des Registres

Le nombre de registres utilisable dans un GPU est très limité selon sa modèle. Pour éviter cela, les algorithmes de GPU doivent être scindés en des kernel plus petits et l'usage de registres par kernel doit être le plus bas possible. Il est parfois nécessaire de se reporter sur une autre mémoire comme la mémoire partagée pour remplacer les registres. Le nombre de registres utilisés par un thread bride l'occupation du kernel. L'**Occupation** est définie comme étant le rapport du nombre de warp actifs (réel) par rapport au nombre maximum de warp (admissible). Pour une performance optimale, le taux d'occupation doit être maximisé.

I.3.5.4 Utilisation de l'allocation dynamique et le structure d'objet

L'algorithme conçu pour le CPU utilise souvent l'allocation dynamique et la notion d'objet le cas échéant sans que cela suscite des problèmes de performance. En particulier, la structuration orientée objet est largement utilisée dans la MEF pour représenter les éléments, les nœuds, la géométrie… CUDA supporte l'allocation de mémoire dynamique à l'intérieur des kernels. Toutefois, elle est très **limitée** et affecte **négativement les performances** [22]. En outre, les approches objet font que les données sont réparties de façon aléatoire dans la mémoire globale est cela devient très inefficace en raison de l'accès mémoire non coalescent, comme nous l'avons expliqué précédemment.

I.3.5.5 Divergence de « warp »

Tous les threads d'un warp s'exécutent simultanément. En pratique, le contrôle d'exécution avec des mots-clés « *if* », « *switch* », « *do* », « *for* », « *while* » peut provoquer la divergence des threads, ce qui signifie qu'ils doivent suivre des chemins d'exécution différents [22]. Si cela se produit, les voies d'exécution différentes doivent être **mises en série**. Pour éviter cette situation, les algorithmes sur GPU doivent être développés de telle sorte que la branche de divergence entre les threads adjacents soit réduite au minimum possible.

I.3.6. JNI et JCuda

Tandis que CUDA est nativement écrit en langage C/C++, il y a eu pas mal d'efforts pour porter la puissance de calcul des GPU vers d'autres langages. JAVA est un des langages de programmation les plus utilisés du fait de ses avantages. JCuda est un projet de portage de CUDA en JAVA. En principe, JCuda fonctionne sous JAVA en interagissant directement avec CUDA C à travers la Java Native Interface (JNI) qui fait le pont entre JAVA et C. En conséquence, les runtime CUDA et les APIs de CUDA, ainsi que les librairies de CUDA peuvent être exploitées en JAVA avec très peu modification par rapport leur approche originale en C/C++. Dans la cadre de cette thèse, les études s'effectuent sur un programme développé en langage JAVA. Donc, nous utiliserons le package JCuda au lieu de CUDA C.

Du point de vue du programmeur en JAVA, il y a deux types de code : un code « **JAVA** » et un code « **natif** ». Le code « JAVA » est compilé en *bytecode* et s'exécute à l'intérieur de la machine virtuelle JAVA (en anglais, JAVA Virtual Machine, JVM). La JVM permet une application JAVA de fonctionner et produire les mêmes résultats quelle que soit la plate-forme. Le code « natif » est compilé en *binaire* et concerne l'application native ou les bibliothèques logicielles basées sur d'autres langages comme C/C++. C'est le cas du code CUDA C. Le code CUDA C peut être utilisé en JAVA comme une « méthode native ». En général, il y a trois étapes principales pour créer et utiliser une méthode native en JAVA :

1. **déclarer** la méthode native dans une classe JAVA avec le mot-clé `native`. Lorsque le compilateur JAVA détecte une déclaration de méthode native dans le code source Java, il enregistre le nom et les paramètres de la méthode. La machine JVM peut ensuite résoudre la méthode correctement lorsqu'elle est appelée.

2. **exécuter** la méthode native. Les méthodes natives sont exécutées sous forme de points d'entrée externes dans une bibliothèque binaire chargeable (DLL). Le contenu d'une bibliothèque native est spécifique à la plateforme d'exécution. Lorsque la JVM appelle les méthodes natives par JNI, elle lui passe en paramètre un pointeur qui contient l'interface vers la JVM. Ce dernier inclut toutes les fonctions nécessaires pour interagir avec la JVM et travailler avec les objets Java. La méthode native est exécutée sur cette structure.

3. **charger** le code de la méthode native à utiliser par la JVM. Outre la déclaration de la méthode native, la bibliothèque native qui contient la méthode doit être chargée lors de l'exécution. JAVA fournit deux interfaces à charger les bibliothèques natives :

`java.lang.System.load()` ou `java.lang.System.loadLibrary()`

Généralement, une classe qui déclare des méthodes natives charge la bibliothèque native lors de son initialisation. Par exemple ci-dessous, la classe `DemoNative` utilise une méthode native `foo()`, elle doit charger une bibliothèque externe native qui contient le code d'exécution pour la méthode `foo()`. Le détail de JNI et la méthode de portage de code en JAVA se trouve dans [23].

```
public class DemoNative {
    static{
        System.loadLibrary("native"); // native.dll pour window
    }
    // declarer une méthode native
    public static native void foo() ;
}
```

Quant à la **performance** de CUDA en JAVA, *Docampo et al.* ont analysé le timing des kernels en comparant plusieurs approches différentes, y compris JCuda, CUDA C/C++ [24]. Les expériences sont effectuées d'abord sur des exemples donnés dans « CUDA Toolkit » pour être dans une situation de code natif CUDA C [25]. Les résultats montrent que la performance du kernel CUDA en C/C++ est meilleure qu'en JCuda. Parmi toutes les solutions de portage CUDA en JAVA, JCuda atteint la performance la plus grande. La différence de performance entre JCuda et CUDA C dépend du problème. JCuda fournit une performance de calcul d'environ 85-90% par rapport l'original en CUDA C pour le problème de multiplication des matrices en utilisant les bibliothèques natives de CUDA comme Cublas, Cusparse, Cufft. Le coût de 10-15% est le chargement des données entre la JVM et le code natif CUDA C/C++. En général, JCuda est évalué comme le package de CUDA en JAVA le plus efficace, à la fois en termes de son potentiel de productivité et de performance.

I.4. Conclusion

Dans ce chapitre :

- Nous avons présenté le contexte général de la thèse et la motivation d'application du calcul parallèle sur GPU afin d'améliorer la performance de la simulation MEF. En prenant en compte la haute performance de calcul à un faible coût des GPU, ils peuvent être considérés comme de puissants accélérateurs de simulation dans le domaine de l'électromagnétisme. En outre, nous avons constaté que le calcul parallèle sur GPU est bien adapté aux besoins de la MEF sur les aspects suivants :
  - La procédure de génération des nœuds du maillage peut être accélérée par une approche parallèle. En assignant chaque thread parallèle à un nœud, le calcul sur GPU peut réduire le coût du maillage.
  - La MEF nécessite un calcul quasi identique pour tous les éléments. Cela est très cohérent avec le modèle de calcul des GPUs qui vise à paralléliser des tâches similaires dans des milliers threads simultanément.
  - La simulation MEF génère un système d'équations algébriques de très grande taille. La résolution de ce système demande un solveur itératif performant. En utilisant la puissance de calcul arithmétique des GPUs, il est possible de réduire considérablement le temps de calcul.
  - En outre, tous les calculs de MEF basés sur le maillage peuvent être traités sur GPU, y compris le post-traitement.
- Nous avons introduit les notions de base sur le calcul parallèle en général et le calcul massivement parallèle sur GPU. Nous savons comment classifier le problème selon le type de parallélisme et évaluer le potentiel du calcul parallèle.
- Nous prenons en compte l'architecture CUDA, une plateforme matérielle et logicielle qui permet de simplifier la prise en main du calcul parallèle sur GPU. L'utilisation de CUDA est rendue assez aisée en particulier grâce aux bibliothèques intégrées. L'efficacité d'un calcul parallèle sur GPU dépend entièrement de la compatibilité de l'algorithme avec l'architecture GPU. Cela demande au développeur d'avoir des connaissances sur l'architecture GPU et sur les techniques d'optimisation des algorithmes de calcul parallèle.

CHAPITRE II

Parallélisation du Maillage

SOMMAIRE

La méthode des éléments finis est une méthode numérique largement utilisée pour la conception et la simulation dans de nombreux domaines. Cette méthode nécessite la discrétisation du domaine en un ensemble d'éléments, appelé le maillage. Les formes les plus courantes d'éléments sont les triangles en 2D et les tétraèdres en 3D. Dans ce chapitre, nous nous limiterons aux maillages 2D en éléments triangulaires même si nous verrons par la suite qu'il est assez direct d'étendre la méthode proposée au 3D [26]. Dans l'approximation par éléments finis, la qualité du maillage joue un rôle important au regard de la précision des solutions [27]. Un maillage avec de mauvais éléments tels que des éléments plats, longs ou minces peut conduire à une matrice de rigidité mal conditionnée, et par conséquent, peut ralentir le solveur ou réduire la précision de la solution. Dans la littérature, de nombreuses méthodes pour la génération automatique de maillage ont été proposées [28]. La plupart des méthodes sont basées sur la triangulation de Delaunay. La parallélisation de cette méthode est assez rare car la génération des points intérieurs du domaine et des éléments est une procédure séquentielle dont les opérations successives dépendent des précédentes. Par ailleurs, un procédé utilisant un système dynamique de bulles a montré une grande capacité à générer des maillages de très haute qualité en limitant drastiquement les éléments dégradés en taille ou en forme [29]. Son principal inconvénient est le temps d'exécution de cette méthode qui est long et qui augmente très significativement avec le nombre de bulles utilisées. Cependant, au contraire des mailleurs de Delaunay, nous constatons que le mailleur par bulles s'adapte bien à l'algorithmique parallélisée. Dans ce chapitre, nous allons utiliser la plateforme de calcul parallèle CUDA sur GPU pour améliorer considérablement le maillage par bulles.

II.1. Introduction du maillage pour MEF

Le maillage non structuré est essentiel pour la méthode des éléments finis car il est capable de paver un domaine irrégulier et complexe. Il y a plusieurs formes d'éléments : triangle, quadrilatère, tétraèdre, hexaèdre, prisme comme le montre la Figure II.1 mais la forme utilisée pour un mailleur automatique est généralement le triangle en 2D et le tétraèdre en 3D.

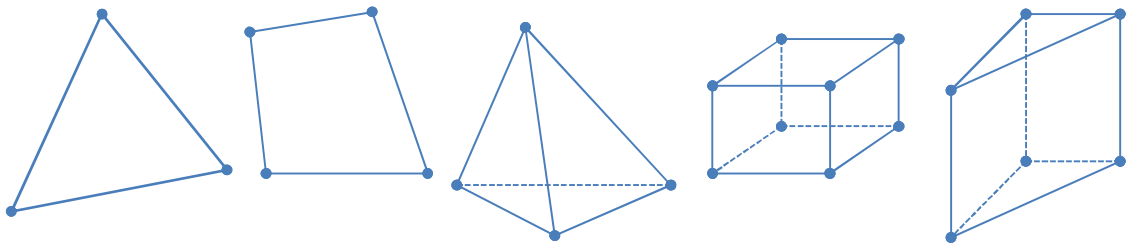


Figure II.1 Quelques types d'élément : triangle, quadrilatère, tétraèdre, hexaèdre, prisme

Conditions de validité des maillages non structurés pour la MEF

La méthode des éléments finis demande quelques prérequis aux maillages non structurés [28]:

- le maillage ne doit pas contenir de trou ni de chevauchement entre éléments. En outre, dans le cas des éléments triangulaires en 2D, aucun élément ne peut partager deux ou plus de deux arêtes avec un autre élément. De façon similaire en 3D, aucun élément tétraédrique ne peut partager deux ou plus de deux facettes avec un autre élément.

- le maillage doit se conformer à la géométrie, c'est-à-dire respecter le bord de la géométrie.

- la densité du maillage doit être contrôlable par une fonction de taille, qui varie de manière souple et continue sur le domaine. Soit cette fonction de taille est attachée à un estimateur d'erreur locale, soit elle est définie par l'utilisateur.

- la forme des éléments doit satisfaire quelques conditions pour atteindre un bon niveau de qualité : un maillage de bonne qualité possède un maximum d'éléments équilatéraux (triangles équilatéraux ou tétraèdres équilatéraux).

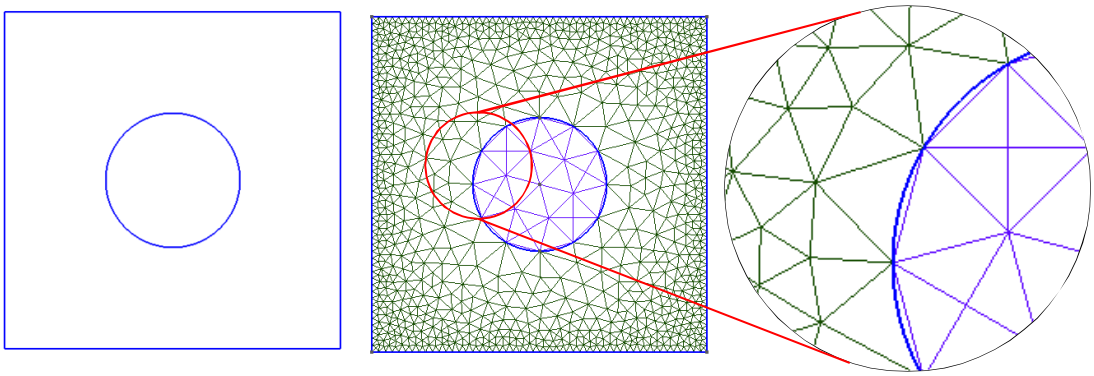


Figure II.2 Illustration de maillage triangulaire d'un domaine comportant un quadrilatère et un trou circulaire adapté à la géométrie.

Le maillage par triangles/tétraèdres facilite la discrétisation d'un bord irrégulier. La Figure II.2 illustre un maillage conforme d'un domaine en 2D avec des éléments triangulaires. La condition de taille et de forme permet de limiter les mauvais éléments qui sont à la source d'une dégradation de la solution. Dans cette section, nous présentons rapidement les trois méthodes les plus utilisées pour le maillage non structuré, à savoir les méthodes de Quadtree/Octree, d'avancement de front et de Delaunay.

II.1.1. Méthode de maillage basée sur une grille

Dans cette méthode, une grille rectangulaire ou triangulaire simple est utilisée pour découper le domaine, y compris au-delà du bord de la géométrie. Ensuite, la grille est tronquée par le bord du domaine et les points d'intersection de la grille avec ces bords sont déplacés afin de créer les éléments. Le maillage triangulaire est obtenu finalement en divisant en deux les éléments rectangulaires. Le maillage par grille est illustré par la Figure II.3. Cette méthode est simple et rapide mais la qualité des éléments est faible surtout pour les éléments du bord alors que la MEF demande assez systématiquement une bonne discrétisation sur ces contours [30].

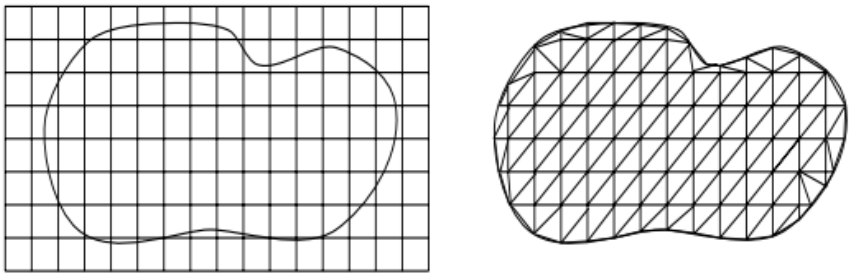


Figure II.3 : Illustration de maillage par une grille qui découpe le domaine

L'usage d'une grille régulière conduit également à des contraintes de densité car elle ne génère que des éléments uniformes en taille. Les quadtree en 2D et les octree en 3D permettent de dépasser cette limitation, en apportant des grilles à taille variable (Figure II.4). On constate cependant que la discrétisation au bord reste une limitation de cette méthode [30].

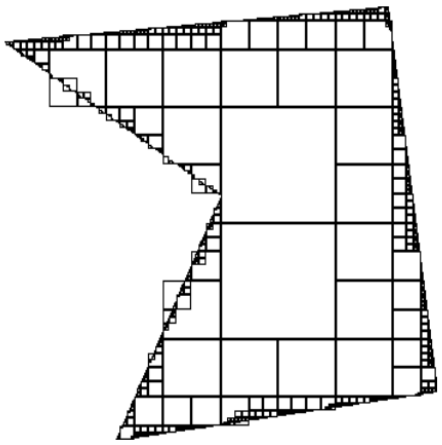


Figure II.4 Illustration de maillage rectangulaire par quadtree en 2D

II.1.2. Méthode frontale ou de l'avancement de front

Le principe de cette méthode est simple :

- Elle débute par une discrétisation du bord de domaine qui constitue un front initial (un ensemble des points qui créent une courbe/surface fermée de la géométrie). Le front est théoriquement la courbe/surface qui sépare la partie déjà maillée du domaine avec la partie non encore maillée.

- Suit une procédure d'insertion d'un élément (triangle en 2D, tétraèdre en 3D) dans le maillage avec au moins une arête/une facette sur le front. Le front est mis à jour avec le nouvel élément.

- la génération des éléments continue ainsi jusqu'à ce que le front soit vide et le maillage terminé.

La Figure II.5 ci-dessous illustre le maillage à l'aide l'avancement de front sur un domaine 2D.

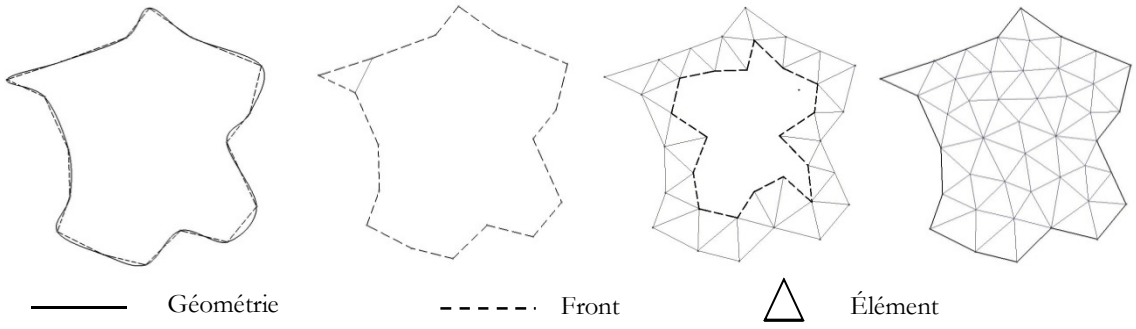


Figure II.5 Illustration de la méthode de l'avancement de front

Si son principe est simple, la procédure de progression de cette méthode est compliquée à mettre au point à cause des règles qui régissent la création des nouveaux éléments. Ces règles

peuvent être basées sur le critère de la cercle (la sphère) vide de Delaunay et / ou d'autres conditions concernant le front ou la forme des éléments afin de garantir la convergence de l'algorithme ainsi que la qualité du maillage [28] [31] [32]. Par contre, la méthode de l'avancement sait naturellement s'adapter à la frontière du domaine et aux contraintes de taille. Le front initial joue un rôle important dans cette méthode. En 3D, la discrétisation des surfaces aux bords du domaine devient très sensible et coûteuse : si cette première triangulation crée de mauvais éléments de départ, l'avancement de front peut construire un maillage volumique de très mauvaise qualité.

L'avantage de la méthode frontale réside dans la bonne qualité du maillage obtenu mais au prix d'une implantation complexe et d'un coût calcul important.

II.1.3. Méthode de maillage basée sur la triangulation de Delaunay

La triangulation de Delaunay d'un ensemble de points est très courante et est présentée dans de nombreux ouvrages de maillage [28] [33] [34]. Dans la pratique, la plupart des logiciels de maillage actuels utilisent cette technique. Le maillage d'un domaine se réalise généralement en deux étapes : l'insertion successive de nœuds intérieurs au domaine et la régularisation de Delaunay. Les méthodes de maillage actuelles génèrent souvent les nœuds internes en même temps que la triangulation.

Triangulation de Delaunay

La triangulation de Delaunay d'un ensemble P de points du plan consiste à mettre en place des triangles DT(P) tels qu'aucun point de P n'est à l'intérieur de cercle circonscrit d'un quelconque des triangles de DT(P) [35]. La Figure II.6 illustre la triangulation d'un ensemble de dix points en 2D.

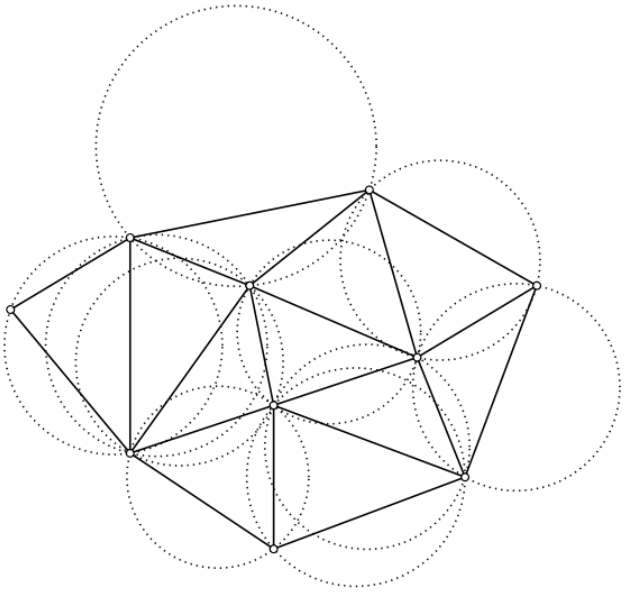


Figure II.6 Triangulation de Delaunay en insistant sur le critère du cercle vide. Il n'existe aucun sommet de triangle intérieur aux cercles en pointillé

La décomposition de Voronoï est un dual de la triangulation de Delaunay. Le diagramme de Voronoï d'un ensemble de points P est la division V(P) de l'espace en régions autour de chaque

point p Є P qui définissent la partie de l'espace le plus proche de ce point par rapport aux autres [35]. La Figure II.7 illustre la décomposition de Voronoï et sa triangulation duale de Delaunay.

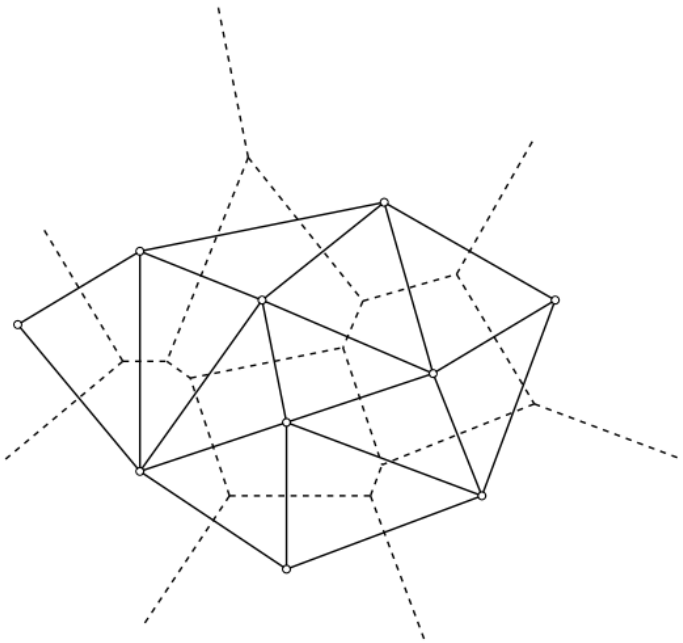


Figure II.7 Diagramme de Voronoï (ligne en pointillés) et son dual sous forme de triangulation de Delaunay

La triangulation de Delaunay possède deux caractéristiques intéressantes pour le maillage :

- Aucun point n'est intérieur à un quelconque cercle circonscrit de n'importe quel triangle. Cette propriété est appelée le critère du cercle vide en 2D et de la sphère vide en 3D. Cette condition est utilisée dans plusieurs méthodes de maillage.

- en 2D, la triangulation de Delaunay **maximise le plus petit angle** de l'ensemble des angles des triangles. Cette condition est importante car elle assure que la triangulation est le meilleur maillage possible pour un ensemble de points donnés. Cependant, cette propriété « max-min » n'est pas obtenue en 3D et de mauvais éléments peuvent être formés.

Il y a plusieurs algorithmes de triangulation de Delaunay d'un ensemble de points [32] [35]. La complexité théorique est en $\Theta(N\log N)$ en 2D, cependant, plusieurs articles en proposent une implantation en $\Theta(N)$ tant en 2D qu'en 3D, avec N le nombre de points. En pratique, l'algorithme le plus utilisé et applicable à toutes les dimensions est **l'algorithme de Bowyer-Watson** [35] [36] [37] [38].

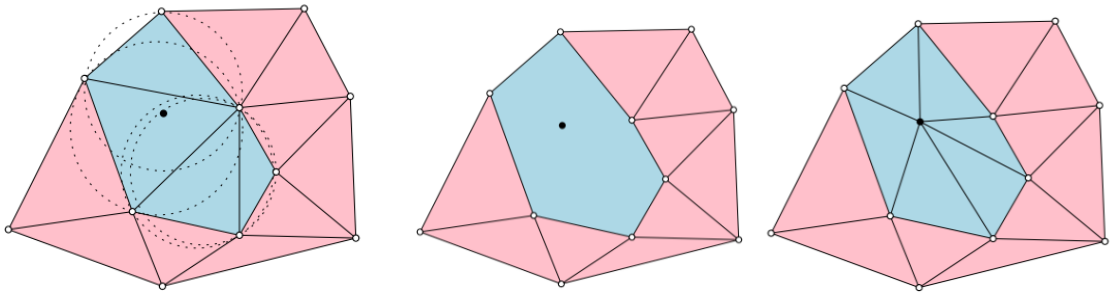


Figure II.8 Illustration de l'algorithme de Bowyer-Watson en 2D

La procédure de cet algorithme est présentée en Figure II.8 ci-dessus :

- ajout un point dans la triangulation existant.

- recherche du triangle qui contient le point. La recherche coûte approximativement Θ(logN) si une technique d'optimisation est appliquée (algorithme de « marche » [39]), sinon dans le cas général, elle coûte Θ(N).

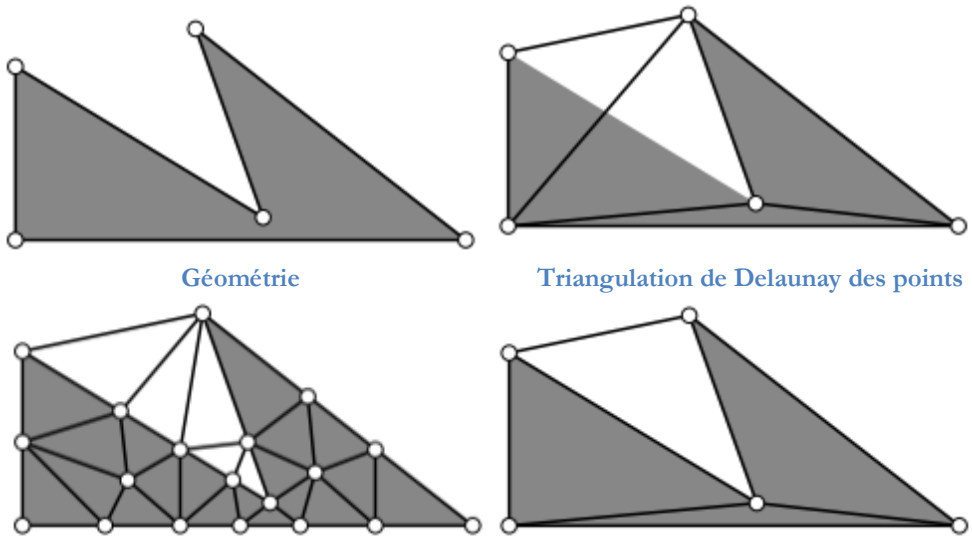
- recherche des triangles voisins dont le cercle circonscrit contient le point. Ce sont les triangles ne satisferont pas la règle du cercle vide si le point est réellement ajouté.

- suppression de ces triangles, ce qui crée (toujours) une cavité convexe.

- rattachement du nouveau point à tous les sommets sur la frontière de la cavité

Conformation à la géométrie

Le maillage doit respecter la géométrie. Malheureusement, cette condition est parfois en conflit avec la propriété max-min de la triangulation de Delaunay. La Figure II.9 illustre le maillage une géométrie concave : en satisfaisant le critère de cercle vide, la triangulation des points de la géométrie peut générer les éléments plats, longs ou ne pas être conforme à la géométrie.



Géométrie | Triangulation de Delaunay des points

Insertion des points supplémentaires | Triangulation modifiée CDT

Figure II.9 Illustration du maillage en 2D par la triangulation de Delaunay

En 2D, il est possible de forcer la triangulation à se conformer à la géométrie. Il y a pour cela deux techniques :

- Ajout des points supplémentaires sur les arêtes non respectées de la géométrie en maintenant le critère de cercle/sphère vide.
- L'autre technique ne maintient plus la condition de cercle/sphère vide de Delaunay en introduisant une triangulation modifiée, dénote CDT (en anglais **Constrained Delaunay Triangulation**). La CDT consiste à construire d'abord

une triangulation de Delaunay à partir des points de la géométrie et puis à traiter à part des arêtes non respectées via un algorithme de permutation.

En 3D, la création de CDT devient plus difficile car non seulement les arêtes mais aussi les facettes des éléments peuvent intersecter des entités de géométrie. Dans la plupart des cas, une combinaison de permutations d'arêtes et de facettes avec une insertion de points supplémentaires établira une CDT qui sera conforme à la géométrie en 3D [40].

Conformation aux contraintes de taille et de forme

La triangulation de Delaunay ne constitue que 50% de la problématique de maillage pour la MEF : Elle construit un maillage en connectant un ensemble de points connus sur une géométrie connue. Mais selon la position des points dans l'espace, cette première triangulation du domaine comprend généralement des éléments trop grands, trop longs ou trop plats. C'est pourquoi il faut introduire une procédure permettant d'éliminer ces éléments par ajout de points intérieurs au domaine. Cette procédure est appelée le **raffinement de triangulation** et guidée par des contraintes de taille et de forme.

Les contraintes de taille et de forme sont définies par l'utilisateur ou s'inscrivent dans le cadre d'une approche de remaillage auto-adaptatif. La carte de taille définit la densité du maillage en tout point de l'espace. La contrainte de forme représente le seuil minimal de qualité que les éléments générés doivent respecter. Plus l'élément est équilatéral, plus sa qualité est grande. La qualité des éléments peut être évaluée de plusieurs façons [28]. Par exemple, la Figure II.10 considère le rapport entre le rayon du cercle inscrit (dénote r) et circonscrit (dénote R): plus le rapport est grand, plus la qualité de l'élément est bonne.

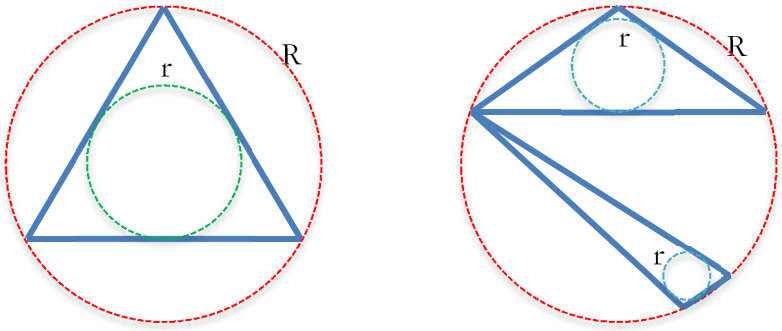


Figure II.10 Indicateur de qualité via le rapport de rayon du cercle inscrit et circonscrit du triangle : r/R = 0.5 pour le triangle équilatéral et r/R<0,5 pour le triangle long, plat

Le raffinement de Delaunay le plus connu est l'algorithme de Ruppert [41]. Paul Chew [42] a apporté quelques modifications qui permettent d'améliorer la qualité de la triangulation finale. Les alternatives peuvent être retrouvées dans plusieurs rapports et articles [43], [32], [28].

Algorithme de raffinement de Ruppert

L'algorithme de raffinement de Ruppert se base sur deux règles :

- La première règle consiste en l'ajout d'un point au centre du cercle circonscrit du « mauvais » triangle. Un triangle est considéré de « mauvaise qualité » si la taille ou l'angle le plus petit du triangle sont en infraction avec les contraintes de taille et de forme [43].

- La deuxième règle repose sur l'ajout d'un point au milieu d'une arête considérée « mauvaise », c'est à dire telle que le cercle diamétral de l'arête contient un point de la triangulation [43]. En outre, la deuxième règle est prioritaire par rapport la première règle. Les deux règles sont illustrées par la Figure II.11 et la Figure II.12.

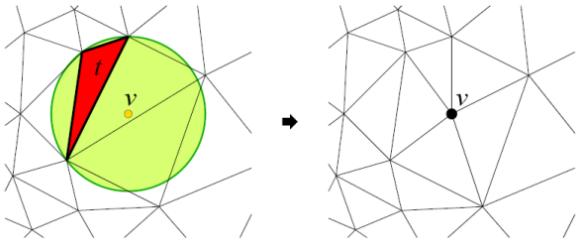


Figure II.11 Insertion du point centre du cercle circonscrit peut éliminer le mauvais triangle

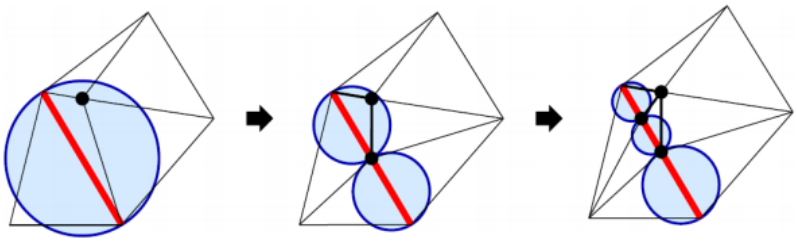


Figure II.12 Insertion du point de milieu de l'arête élimine le mauvais triangle

L'algorithme de Ruppert donne un guide pour ajouter des points à la triangulation de Delaunay et éliminer de mauvais triangles au point de vue des contraintes de taille et de forme. Cet algorithme assure un maillage dont l'angle le plus petit des triangles est supérieur de 20.7^0. Il est à noter que le traitement des mauvais éléments repose sur une démarche séquentielle qui parcourt les éléments parce que la suppression d'un triangle existant et l'ajout des nouveaux triangles vont modifier la structure de la triangulation. Cette propriété est peu compatible avec un traitement des éléments en parallèle.

II.1.4. Etat de l'art sur la parallélisation du maillage

Les algorithmes séquentiels pour le maillage décrits ci-dessus ont été développés il y a plusieurs années et sont désormais quasi stables. Ce sont des techniques largement connues et utilisées dans la communauté scientifique. Au contraire, le maillage parallèle est aujourd'hui un problème ouvert et actif. Les premières tentatives de parallélisation du maillage ont été effectuées sur des plateformes de CPU.

Parallélisation du maillage sur la plateforme parallèle de CPU

Dans [44], *Chrisochoides et al.* présentent une synthèse sur les techniques de parallélisation du maillage jusqu'en 2005. L'auteur se concentre sur la triangulation de Delaunay parallèle et le maillage non structuré : il est possible de classifier la parallélisation du maillage selon deux axes : la technique de triangulation et la technique de calcul parallèle. Du point de vue de la technique de calcul parallèle au moment où l'article a été présenté, il existait deux plateformes connues: la **plateforme de mémoire partagée** (« open multi-processing » ou openMP) et la **plateforme de mémoire distribuée** (autrement dit « message passing interface » ou MPI). Du côté de la triangulation de Delaunay [45], les propositions se basent sur la technique de **décomposition du**

domaine. Le domaine est divisé en plusieurs sous-domaines ce qui sont ensuite distribuées vers les multiples processeurs (CPU) et chaque sous-domaine est maillé indépendamment selon une approche séquentielle. La décomposition du domaine joue alors un rôle important dans l'efficacité de la parallélisation du maillage sur la plateforme de multiprocesseurs à CPU.

Il existe quelques implantations parallèles qui traitent du **raffinement de la triangulation** de Delaunay afin d'améliorer la qualité des éléments. Dans [74], *Chernikov et al.* présentent une technique du raffinement parallèle de Delaunay qui est calculée sur un système de mémoire distribuée. L'algorithme se compose de deux étapes : une triangulation effectuée en séquentiel afin de créer un maillage initial et puis, un raffinement parallèle du maillage. Le raffinement parallèle se base sur deux processus : création d'un ensemble de point indépendant pour les insérer, et la mise à jour parallèle de la triangulation. La même approche est utilisée par *Chrisochoides et al.* [46] [12].

Parallélisation du maillage sur GPU

La plateforme de calcul parallèle sur GPU ouvre une nouvelle voie pour la parallélisation du maillage. Le challenge principal du maillage non structuré sur cette plateforme est la communication régulière entre les threads parallèles qui doivent vérifier et échanger des informations concernant les éléments adjacents. Ce problème est plus important dans le cas de grandes structures de données. En effet, la performance du GPU est lourdement pénalisée si ses threads parallèles sont dépendants et requièrent une synchronisation ou une communication fréquente. De plus, la nature non structurée du maillage entraine une grande dispersion des données conduisant à un accès non optimal en mémoire des GPUs. Jusqu'à aujourd'hui, la tentative du portage de maillage sur GPU ne se concentre que sur la triangulation de Delaunay à partir d'un ensemble de points connus.

Ainsi, GPU-DT est une implantation parallèle de la triangulation sur GPU dont le principe se base sur la permutation des arêtes. L'exécution de GPU-DT est proposée par *Rong et al.* dans [13] en utilisant un diagramme de Voronoï discrétisé afin de faciliter l'insertion des points en parallèle. Les données du diagramme de Voronoï sont réservées sur la mémoire de texture du GPU. Une version récente de cet algorithme, appelé GPU-CDT par *Cao et al.* [47] est capable de mailler un domaine 2D simple dont la géométrie se compose des lignes droites selon le format PSLG (« Planar Straight Line Graph »). Une triangulation parallèle en 3D se trouve dans [14]. Le code est développé sur une plateforme mode de calcul hétérogène avec la plupart du calcul effectué en parallèle sur GPU et quelques portions de code exécutées sur CPU. D'une manière similaire, *Navaro et al.* dans [48] ont proposé un algorithme de permutation d'arêtes en parallèle pour une triangulation de Delaunay exécuté sur GPU.

La triangulation de Delaunay parallèle sur GPU présente de bonnes performances par rapport sa version séquentielle en termes de réduction du temps d'exécution. Cependant, elle se base sur un **ensemble de points prédéfinis** ce qui n'est qu'une des deux étapes d'un maillage automatique. Il convient également de produire un algorithme parallèle pour générer cet ensemble de points, selon une distribution adaptée à la géométrie, respectant des critères de taille et de forme. Donc, au regard de la littérature actuelle, il n'existe aucune implantation parallèle sur GPU d'un mailleur automatique de Delaunay qui effectue à la fois la génération des points et la triangulation.

Dans la section suivante, nous proposons notre méthode de maillage automatique : le maillage par bulles, qui est non seulement capable de générer des maillages de très bonne qualité mais également bien adaptée à la parallélisation sur GPU.

II.2. Méthode de maillage par Bulles

II.2.1. Etat de l'art

Le maillage par bulles a été proposé initialement par *Shimada et al.* [29] en partant de l'analogie entre les polyèdres de Voronoï et des cercles. En se rapportant à la configuration de Voronoï, on constate qu'un ensemble de cercles/sphères, appelé « bulles » peut remplir le domaine avec une densité dont la variation suit la densité de nœuds. En s'appuyant sur une distribution adaptée de ces bulles dans l'espace, pour obtenir la triangulation de Delaunay, il suffit de relier tous les nœuds centraux des bulles pour créer le maillage. Les auteurs ont introduit un modèle de forces de répulsion-attraction entre bulles adjacentes et également un facteur de frottement pour achever de caractériser un système dynamique. Les bulles, initialement en position quelconque non optimale, sont mises en mouvement et une fois que le système a atteint un état stable, la position des nœuds au centre des bulles est connue. Une densité locale de bulles est calculée par un ratio de chevauchement pour contrôler et optimiser le nombre de bulles. Le système de bulles dynamiques peut s'adapter au maillage à densité variable à l'aide d'une carte de taille définie sur tout le domaine.

Cingoski et al. ont proposé une variante de maillage par bulles en 2D et une nouvelle fonction pour estimer la taille des bulles dans [49]. Pour créer les bulles initiales, un maillage de fond est construit à partir de la géométrie du domaine. Chaque zone géométrique est divisée selon une grille de points candidats et les bulles sont générées sur ces candidats en satisfaisant la condition de densité locale. *Yokoyama et al.* étendent cette méthode avec une fonction exponentielle en 3D dans [26].

Kim et al. ont suggéré une technique automatisée de maillage adaptatif par bulles pour les problèmes éléments finis en 2D et 3D [50]. Un estimateur d'erreur est utilisé pour piloter la densité de maillage et ainsi que la taille des bulles. Les résultats démontrent des capacités de génération de maillage de qualité, mais le temps pour faire déplacer les bulles est toujours long et suit en $O(N^2)$ le nombre des bulles.

En utilisant une technique adaptative similaire, *Ebene et al.* ont proposé des solutions pour améliorer la qualité du maillage de Delaunay par le système de bulles [51]. Les auteurs ont proposé une modification de la carte de taille des bulles en fonction d'un estimateur d'erreur normalisé. Avec une procédure locale d'ajout et de suppression de bulles, le maillage adaptatif possède une qualité bien meilleure que le maillage original de Delaunay.

Nobuyama et al. [55] a présenté une implantation parallèle sur GPU du maillage par bulles. Cependant, les résultats ne semblent pas totalement convaincants et la performance de calcul est faible.

Notre expérience montre que la performance d'un calcul parallèle sur GPU atteint son efficacité maximale quand l'algorithme s'adapte à l'architecture des GPUs. Dans cette section, nous exprimerons tout d'abord les principes de la méthode de maillage par bulles. Ensuite, nous

proposons un nouvel algorithme parallèle reposant sur une structure de données bien adaptée pour le système de bulles.

II.2.2. Initialisation des bulles

La méthode doit être initialisée avec un premier ensemble de bulles couvrant le domaine. Une bonne configuration initiale de bulles permettra de réduire le temps de convergence de la méthode [29]. La qualité de la configuration initiale dépend à la fois du nombre et de la position des bulles dans le domaine. La procédure de génération des bulles initiales est la suivante :

- discrétiser les points géométriques par des bulles. Des bulles sont créés aux points d'extrémité des lignes ou points d'angle. Ces points deviennent le centre des bulles. La taille des bulles est contrôlée par une fonction de taille qui est définie préalablement.

- discrétiser les lignes par des bulles : l'objectif est de remplir les lignes sans créer de trous. La Figure II.13 illustre la discrétisation des lignes par les bulles dans deux cas : avec trous et sans trous. Un trou est présent lorsque le nombre des bulles générées sur la ligne est insuffisant. En présence d'un trou sur la ligne, l'espace libre entre les deux bulles va être occupé par une bulle intérieure qui va venir se coller à proximité de la ligne. Cette disposition va conduire à créer des éléments plats (Figure II.13, gauche). Idéalement, les bulles doivent se toucher l'une l'autre ou se chevaucher légèrement ce qui favorise la création d'éléments équilatéraux (Figure II.13, droite).

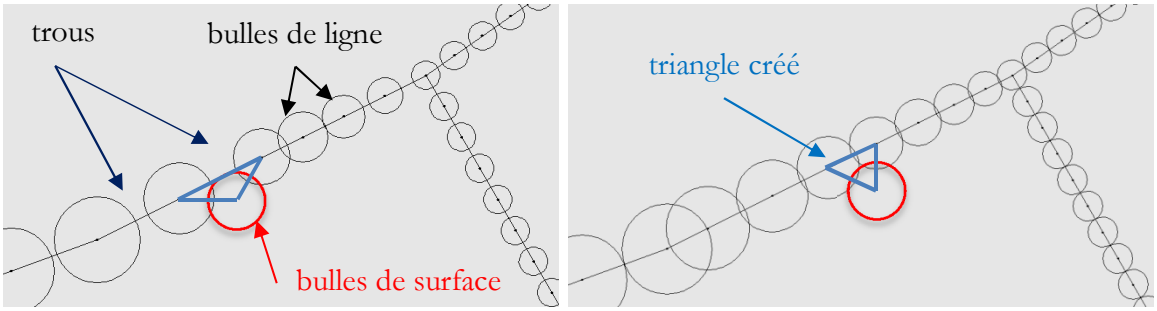


Figure II.13 Illustration de discrétiser des ligner par les bulles

- générer les bulles intérieures à la surface (2D)/volume (3D). Contrairement à la génération des bulles aux points ou sur la ligne, il est difficile de prévoir directement le nombre et la position appropriée des bulles intérieures de la surface ou du volume. Deux techniques sont proposées:

Subdivision par Quadtree/Octree oblique

Le Quadtree en 2D/Octree en 3D réalise une décomposition hiérarchique de la géométrie telle que la taille des cellules de la grille suive une carte de taille. Les bulles sont ensuite placées au centre des cellules [50]. Il est à noter que des cellules obliques ont été introduites, en remplacement des traditionnelles cellules carrées/cubiques pour obtenir directement la bonne configuration hexagonale de centre de bulles, ce qui est bien illustré à la Figure II.14. car son efficacité a été prouvée.

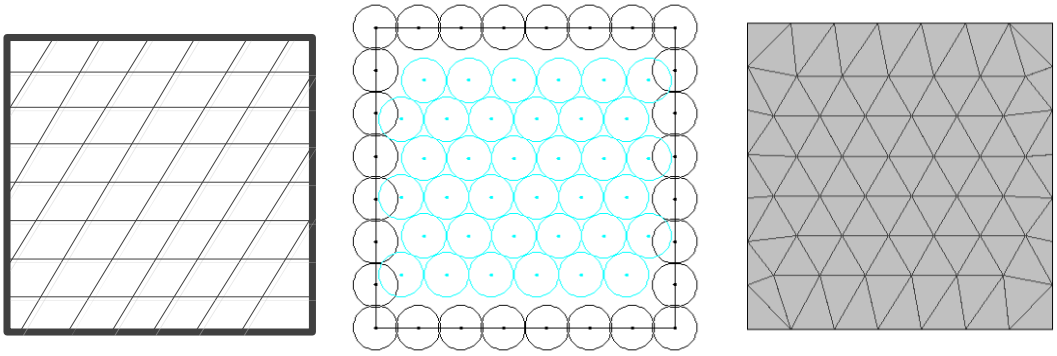


Figure II.14 : Illustration de l'initialisation des bulles par Quadtree en 2D

Maillage grossier

Un maillage grossier est construit sur la géométrie et les bulles sont générées à l'intérieur de chaque élément du maillage grossier. La subdivision d'un élément est effectuée selon une grille. Pour cela, les éléments dans l'espace réel sont transformés dans un espace de référence pour subir une décomposition. La Figure II.15 montre la transformation d'un élément réel du maillage vers l'élément de référence, transformation classique en éléments finis. Les points d'intersection de la grille sont les points candidats où placer les bulles si la condition de densité locale est satisfaite. Un facteur de densité (*alpha*) peut être utilisé pour gérer la densité de la grille ainsi que le nombre des points candidats ([26] [49]).

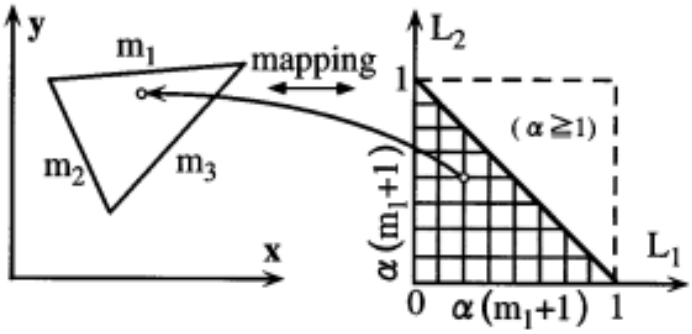


Figure II.15 : Création des bulles intérieures d'un triangle à l'aide d'une grille des points de candidat

II.2.3. Système de bulles dynamiques

Définition des bulles

Un système dynamique est défini par l'ensemble des bulles qui se déplacent dans la frontière de la géométrie. Le mouvement des bulles est guidé et contrôlé par des forces d'interaction entre toutes les bulles. Chaque bulle est caractérisée par sa position a_i, son rayon r_i, sa masse m_i et sa vitesse v_i. L'équation de mouvement d'une bulle dynamique est considérée comme un système du 2ème ordre [52]:

$$m_i \frac{d^2 a_i}{dt^2} + c_i \frac{da_i}{dt} = f_i \qquad \text{(II.1)}$$

où c_i le coefficient de frottement, f_i est la force agissant sur la bulle.

Modèle de force

Il existe plusieurs types de force utilisables mais tous partent d'un même principe. Il s'agit de force attractive-répulsive de Van der Waals [29] ou assimilable à la force du ressort. En pratique, nous définissons un modèle de force entre des bulles proches sous la forme suivante :

$$f_i = \begin{cases} \left(1 - w^4\right)e^{-w^4} & 0 \le w = \dfrac{d_{ij}}{r_i + r_j} \le 1.5 \\ 0 & \mathrm{w} > 1.5 \end{cases} \qquad \text{(II.2)}$$

où d_{ij} est la distance, r_i et r_j représentent le rayon des deux bulles. La force totale qui agit sur une bulle est la somme de toutes les forces causées par les bulles voisines.

Intégration

Nous intégrons l'équation (II.1) à l'aide de la méthode de Runge-Kutta [1] du quatrième ordre [53] et mettons à jour la position et la vitesse des bulles à chaque itération. . Le processus d'intégration itère jusqu'à ce que le système atteigne un état d'équilibre défini par une énergie cinétique totale du système de bulles en-dessous d'un seuil prédéterminé.

Les paramètres m_i, c_i, f_i doivent être optimisés pour obtenir un bon compromis entre le temps de réponse et la stabilité du système. Selon [29] et [51], l'optimum est atteint lorsque la force est représentée par une constante k de ressort:

$$k = \frac{1.47}{2r_i} \qquad \text{(II.3)}$$

Et le coefficient d'amortissement [2] est fixé à 0,7 pour déterminer la valeur du coefficient de viscosité pour une masse donnée :

$$\zeta = \frac{c_i}{2\sqrt{m_i k}} = 0.7 \qquad \text{(II.4)}$$

Le pas de temps d'intégration est choisi en fonction du temps de réponse [3] du système linéaire du 2ème ordre.

---

[1] Equation (II.1) équivaut à 2 équations différentielles: dai/dt = vi et dvi/dt = (fi – ci vi)/mi.

En mettant y = {ai, vi}, nous menons à la formulation générale : dy/dt = f(t, y) et à l'intégration de Runge-Kutta 4eme ordre est formulée comme suivante :

k1 = dt * f(tn, yn)

k2 = dt * f(tn + dt/2, yn + k1/2)

k3 = dt * f(tn + dt/2, yn + k2/2)

k4 = dt * f(tn + dt, yn + k3)

y_{n+1} = yn + (k1 + 2 * k2 + 2 * k3 + k4)/6 + O(dt^5)

[2] Avec un system de 2ème ordre: $m\ddot{x} + c\dot{x} + kx = u$, nous convertissons vers une forme standard comme : $\ddot{x} + \frac{c}{m}\dot{x} + \frac{k}{m}x = \frac{1}{m}u$. Donc, la pulsation propre et le coefficient d'amortissement sont $w_n = \sqrt{k/m}$ et $\zeta = \frac{c}{2\sqrt{km}}$

[3] Temps de réponse est le temps nécessaire pour une sortie d'atteindre et de rester dans une bande d'erreur donnée (souvent 2% ou 5%) de la valeur finale. Le temps de réponse 5% dans le cas ζ < 1 est donné par : $T_s \approx \frac{3}{\zeta w_n}$

Critère d'arrêt

Le mouvement s'arrête lorsque toutes les bulles atteignent une position équilibre au regard des interactions avec leurs voisins. Cependant, l'interaction entre les bulles converge asymptotiquement et des mouvements de très faible amplitude peuvent persister durablement. Donc, il est nécessaire de définir un critère afin d'arrêter le mouvement des bulles lorsqu'elles sont quasi stables. *Ebene et al.* ont proposé une limite qui se base sur l'énergie cinétique maximale théorique du système de bulles [54].

$$E_{\max} = \frac{1}{2}\left(\frac{qR_m}{dt}\right)^2 \qquad \text{(II.5)}$$

avec R_m le rayon moyen des bulles, dt la valeur du pas de calcul caractéristique du système (II.1) et q la fraction maximale de mouvement à chaque itération. Par expérience, q est choisi à 10%, ce qui revient à une énergie cinétique équivalant à un déplacement de 10% de R_m pendant un pas de calcul. La Figure II.16 illustre l'amortissement de l'énergie des bulles dans un mouvement.

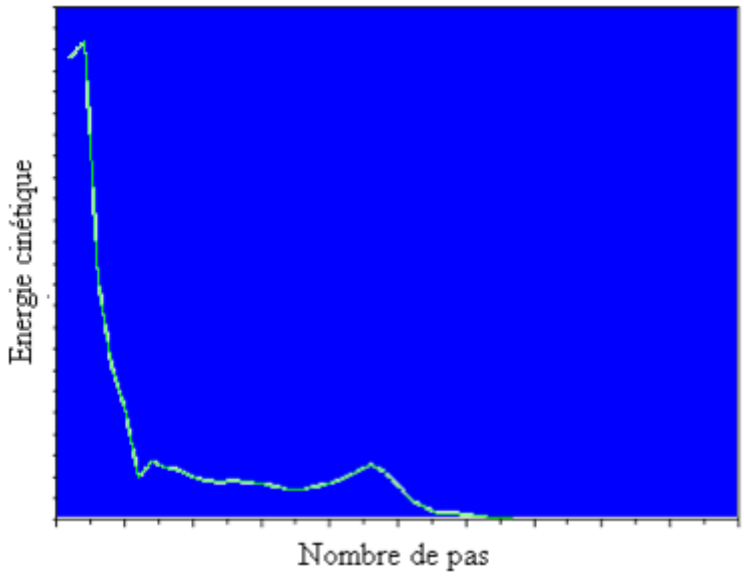


Figure II.16 Variation de l'énergie cinétique mesurée pendant un mouvement

De plus, pour converger encore plus vite, un seuil supplémentaire est appliqué à chaque bulle individuellement, qui définit à partir de quand la bulle peut être considérée comme fixe et sortie des calculs. Ce seuil est fixé à une énergie équivalant à un déplacement d'un centième de son rayon pour un pas de calcul. En deçà de cette limite, la bulle est figée définitivement.

La liste de voisins

Les interactions entre les bulles sont assimilables à un problème de N-corps [55] [56]. Dans le cas de N bulles, N × (N-1) interactions doivent être calculées, le calcul est donc proportionnel à $\Theta(N^2)$. En conséquence, le temps d'exécution dépend fortement du nombre des bulles.

Dans [51], l'auteur constate que le système de bulles possède une mobilité limitée et en réalité, la plupart de bulles se contentent d'osciller autour d'une position. Donc, pour la plupart des bulles, leur liste de bulles voisines ne changent pas ou ne changent que légèrement pendant leur

mouvement. Dans ce cas, en vue d'optimiser les temps de calcul, une liste de bulles voisines de chaque bulle peut être construite avant le calcul d'intégration et n'être mise à jour que périodiquement pendant le mouvement. La liste des voisins permet également de réduire le calcul d'intégration RK4 en supposant une unique liste de voisins pendant les 4 étapes de l'intégration proprement dite. Dans tous les cas, la mise à jour de la liste de voisin ne concerne que les bulles en mouvement et non les bulles figées.

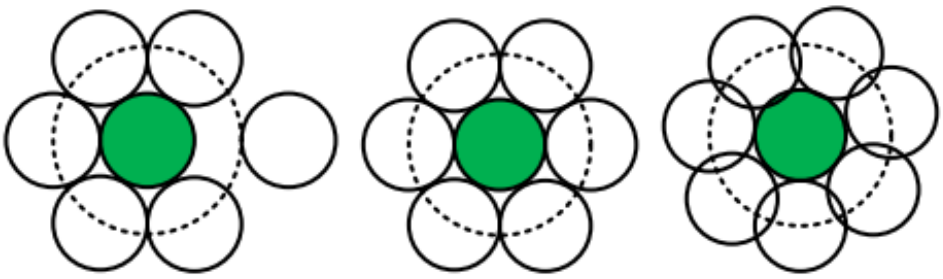
II.2.4. Contrôle de population

Pour obtenir un maillage de qualité, il est nécessaire de contrôler la densité locale de bulles pour obtenir le nombre approprié des bulles, autrement dit, ajouter ou supprimer des bulles au cours du mouvement. *Shimada et al.* [29] suggère un ratio de chevauchement β qui correspond au nombre de voisins pour contrôler la densité locale des bulles :

$$\beta_i = \frac{1}{r_i}\sum_{j=1}^{v}\left(2.r_i + r_j - d_{ij}\right) \qquad \text{(II.6)}$$

Avec v est le nombre des voisins.

La Figure II.17 illustre le cas où une nouvelle bulle doit être insérée autour d'une bulle qui manque de voisins tandis qu'une bulle ayant un excès de voisins doit être supprimée.



Ajout d'une bulle | Nombre optimal | Suppression d'une bulle

Figure II.17 Contrôle du nombre des bulles à l'aide du ratio de chevauchement

Un seuil est utilisé pour déterminer si des bulles manquent ou sont en excès. Dans le cas des bulles de taille uniforme, la valeur standard de β devrait être 2.0 pour des bulles de ligne et 6.0 pour des bulles de surface. Le contrôle de la population de bulles n'est effectué qu'à partir du moment où la dynamique du système de bulles est relativement stable.

II.2.5. Contrôle de taille

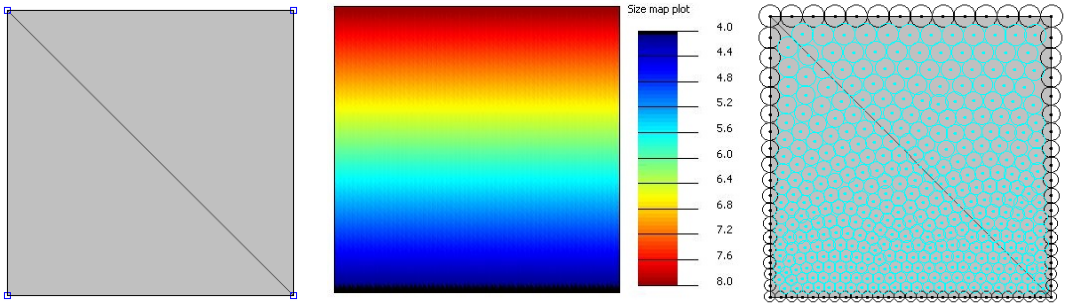
Le maillage doit être capable de répondre aux contraintes de taille définies par l'utilisateur. En conséquence, la taille de bulles (le rayon) doit s'adapter à une carte de taille et être mise à jour pendant leur mouvement. Les contraintes données par l'utilisateur imposent la taille des éléments sur une région du domaine (ex. un point/ ligne/surface/volume). Ensuite, la fonction de taille doit être interpolée sur tout le domaine, de façon régulière et continue à partir de ces valeurs [43].

En pratique, une approche par éléments finis peut résoudre ce problème d'interpolation : Les valeurs imposées sont des conditions de Dirichlet et nous utilisons une interpolation linéaire basée sur un maillage triangulaire grossier du domaine, illustré à la Figure II.18.

En supposant que le maillage se compose d'un ensemble de *n* points {*x1, x2, … xn*} qui correspond aux valeurs de taille données {*v1, v2, … vn*}, la valeur de taille à un point quelconque *x** est donnée par:

$$v(x^*) = \lambda_1 v_1 + \lambda_2 v_2 + \lambda_3 v_3 \qquad \text{(II.7)}$$

Où (*v1, v2, v3*) sont les valeurs de taille connues aux trois sommets du triangle contenant le point *x** et (*λ1, λ2, λ3*) sont les coordonnées barycentriques du point *x** dans le triangle.



Maillage de fond | Carte de taille | Taille de bulles

Figure II.18 : Contrôle de taille de bulles

Il reste donc le problème de localiser le point dans une triangulation du domaine. Evidemment, le balayage de tous les triangles, y compris sur un maillage de fond n'est pas envisageable.

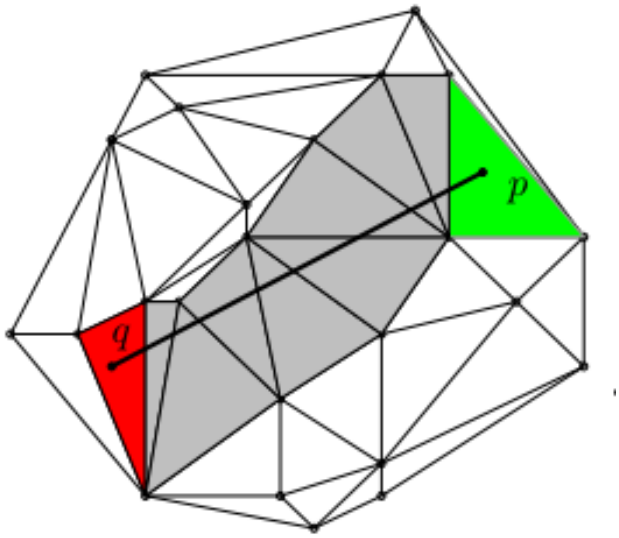


Figure II.19 : Problème de localisation d'un point dans une triangulation

Il existe un algorithme simple à mettre en place, appelé *marche directe* («straight walk» en anglais) [39] illustré à la Figure II.19. On effectue une *marche* à travers la triangulation à l'aide d'une ligne droite entre un point aléatoire *q* et le point de requête *p*. La recherche démarre à partir d'un triangle pris au hasard qui donne le point *q*, l'orientation de chaque arête du triangle est vérifiée au regard du point de requête *p*. Si une arête possède une intersection non nulle avec le segment (*p,q)* on passe au triangle voisin qui partage cette arête et on vérifie si le point est intérieur.

II.2.6. Qualité du maillage

Pour estimer la qualité du maillage triangulaire, il existe de nombreux critères. Pour la méthode de maillage par bulles, nous utilisons l'irrégularité topologique ε_t et l'irrégularité géométrique ε_g qui représentent respectivement la qualité de la distribution des nœuds et de la forme des éléments. Plus l'irrégularité est petite, plus la qualité du maillage est élevée [29].

$$\varepsilon_t = \frac{1}{n}\sum_{i=1}^{n}\left|\delta_i - 6\right| \tag{II.8}$$

où n désigne le nombre de nœuds internes du domaine, δ_i est le nombre de nœuds voisins.

$$\varepsilon_g = \frac{1}{e}\sum_{i=1}^{e}\left(0.5 - \frac{r_i}{R_i}\right) \tag{II.9}$$

où e représente le nombre d'éléments, r_i et R_i sont les rayons inscrit et circonscrit, respectivement.

Le fonctionnement de ces deux critères est montré à la Figure II.20 et la Figure II.21 avec la comparaison entre le maillage généré par notre mailleur par bulles par rapport un raffinement de Delaunay obtenu par l'algorithme de Ruppert [15]. La qualité du maillage est mesurée par la distribution d'angles, ainsi que les valeurs des irrégularités ε_t et ε_g. Le maillage par bulles génère un maillage dont la qualité est très supérieure à l'approche de Ruppert.

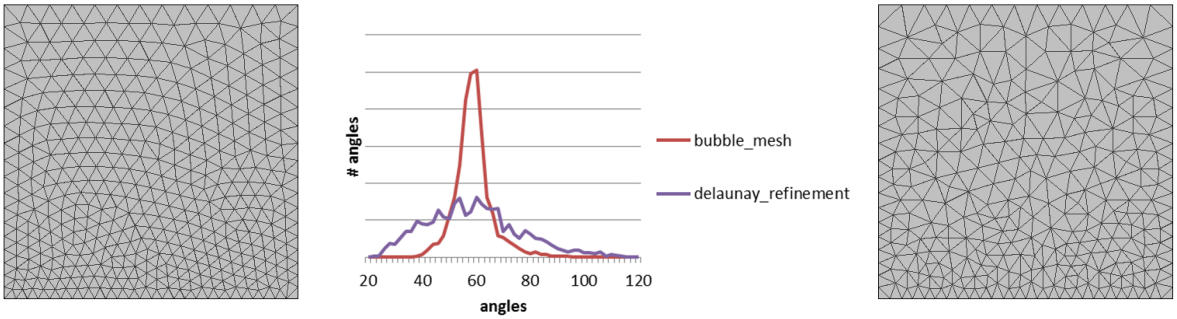


Figure II.20 Comparaison entre notre maillage par bulles (à gauche) et le raffinement de Delaunay (à droite) sur un domaine carré. Les angles sont dans l'intervalle [40, 90] avec notre méthode par rapport à [20, 120] avec le raffinement de Delaunay, la répartition des angles concentrée à 60-deg avec notre méthode, les irrégularités de maillage sont également plus faibles : ε_t 0,16 vs 0,56 et ε_g 0,01 vs 0,06

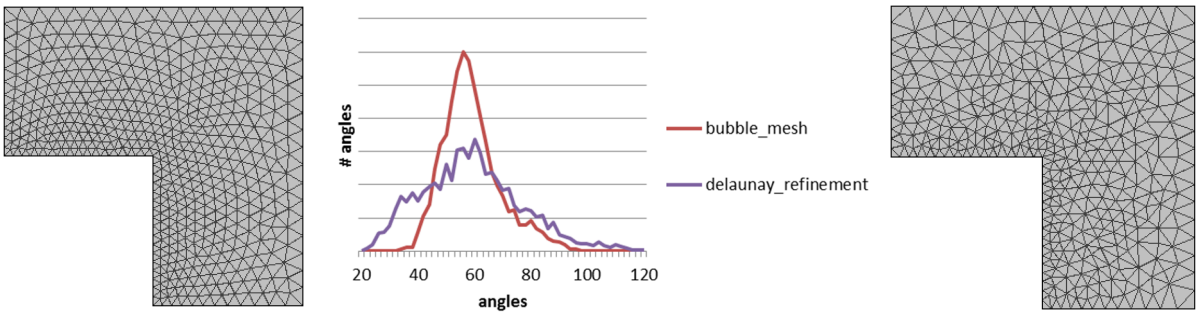


Figure II.21 Comparaison entre notre maillage par bulles (à gauche) et le raffinement de Delaunay (à droite) sur un domaine en forme de L, les angles sont dans l'intervalle [40, 100] avec notre méthode par rapport à [20, 120] avec le raffinement de Delaunay, la répartition des angles concentrée à 60-deg avec notre méthode, les irrégularités de maillage sont également plus faibles : ε_t 0,15 vs 0,64 et ε_g 0,01 vs 0,04

II.2.7. Algorithme de maillage par bulles

En résumé, la procédure générale de maillage par bulles est la suivante :

```
1. Importer la géométrie, les contraintes de taille et de forme
2. Initialisation du système de bulles
3. Pour chaque surface
   a. Récupérer toutes les bulles de la surface (points, lignes, surface)
   b. Fixer les bulles sur la frontière (points et lignes)
   c. Mouvement des bulles intérieures (surfaciques)
   d. Contrôle de population
   e. Vérifier la qualité du maillage, si OUI passer à l'autre surface, si NON
      retourner à 3c.
4. Si le problème en 3D, une même procédure s'exécute pour chaque volume avec des
   bulles volumiques au lieu de bulles surfaciques.

INITIALISATION
1. Génération des bulles sur les points
2. Génération des bulles sur les lignes
3. Génération des bulles dans les surfaces

CONTROLE DE POPULATION
1. Pour chaque bulle intérieur de la surface, vérifier le ratio de chevauchement
   a. Si le ratio est trop bas, ajouter une ou plusieurs bulles
   b. Si le ratio est trop élevé, supprimer la bulle
   c. Eviter de doubler l'ajout ou la suppression

MOUVEMENT
1. Pour chaque bulle dynamique de 1 à N
   a. Construction de la liste de voisins
   b. Intégration RK4 : calcul de la force, le pas adaptif de temps et la mise à
      jour la position et la vitesse
   c. Mise à jour la taille de la bulle avec sa nouvelle position
2. Vérifier le critère d'arrêt du mouvement par l'énergie cinétique totale. Si
   oui, terminer le mouvement. Sinon, continuer le mouvement en retournant à
   l'étape 1.
```

La procédure se compose de deux boucles : Une boucle interne qui concerne le **mouvement** des bulles sous des contraintes de géométrie et de taille. Le critère d'arrêt de cette boucle est basé sur la valeur de l'énergie cinétique totale du système des bulles, il s'agit d'une détection du moment où l'ensemble de bulles est quasi-stable. Une boucle externe est destinée à contrôler la **qualité du maillage** à l'aide des critères sur la qualité locale des éléments et régulariser ce maillage par ajout ou suppression de bulles ni nécessaire. Si un ajout ou une suppression a lieu, un nouvel ensemble des bulles est constitué et il est nécessaire de relancer une nouvelle itération de mouvement. En pratique, la boucle externe se termine après quelques itérations pour une géométrie complexe ou dès la première itération pour une géométrie simple si le nombre de bulles initial est approprié.

II.2.8. Analyse du temps d'exécution

Nous réalisons une procédure du maillage par bulles avec la géométrie en forme de L comme présenté à la Figure II.21. Le nombre des bulles est paramétré par la fonction de taille. La Figure II.22 montre l'évolution du temps d'exécution en fonction du nombre des bulles. On constate bien que la complexité de l'algorithme est située entre les deux lignes $\Theta(N)$ et $\Theta(N^2)$, grâce en particulier à la coupure prématurée reposant sur des extinctions de bulles individuelles d'une part, et d'autre part, par l'usage de liste de voisins dans le calcul de la force.

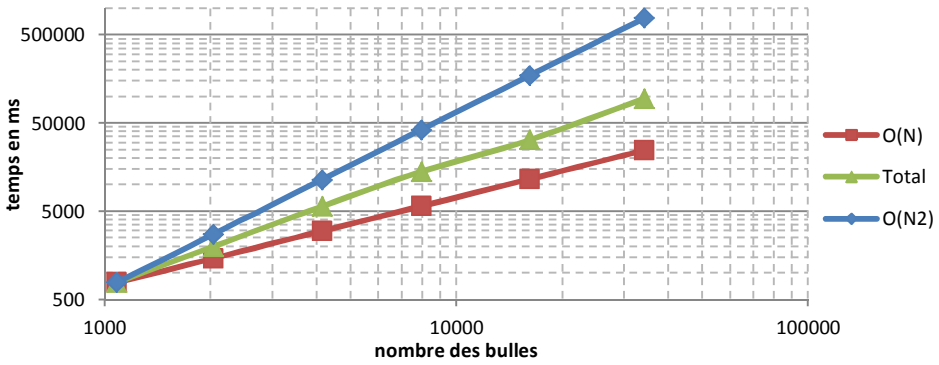


Figure II.22 Evolution de temps d'exécution du mailleur par bulle en fonction du nombre des bulles

La Figure II.23 montre une analyse du temps des étapes du processus. En général, les phases « initialisation » et « contrôle de densité » présentent un impact faible (environ 10%) tandis que le calcul de « mouvement » coûte 90% de la durée totale. Dans le « mouvement », le calcul « liste de voisins » qui nécessaire au calcul des interactions entre les bulles et la construction de la liste de voisins en demande 60%. Le calcul d'intégration de RK4 et le contrôle de taille coûtent environ 30%. Le processus « autre » concernant la vérification de convergence est de l'ordre de 10%. La part du processus de construction des voisins ainsi que le « mouvement » augmente avec le nombre de bulles.

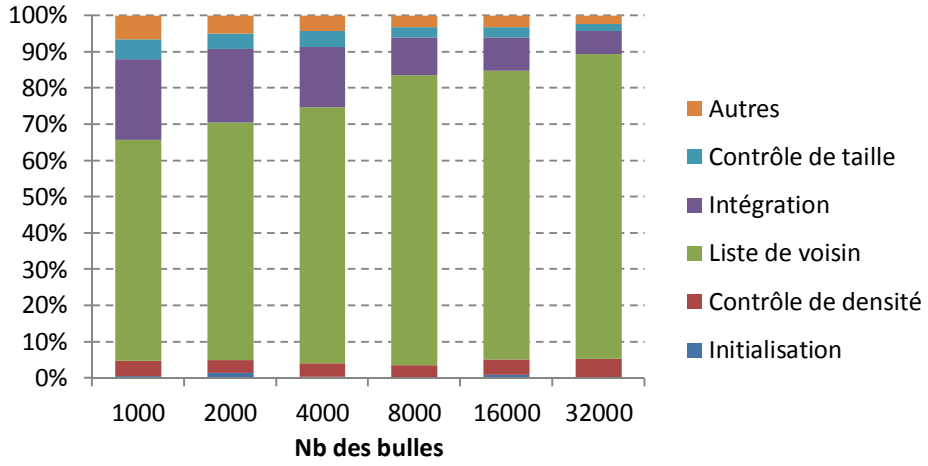


Figure II.23 Répartition du temps d'exécution des processus individuels selon le cas

Nous constatons que le calcul du mouvement des bulles est très prépondérant dans le temps total du maillage. Rappelons que le mouvement des bulles est un calcul itératif dont le calcul d'une itération demande une boucle de N évaluations où N est le nombre de bulles. Le calcul séquentiel sur CPU s'effectue par une simple boucle sur les bulles de 1 à N. En supposant que le calcul individuel de chaque bulle soit indépendant, il est parait naturel d'attribuer le calcul des N bulles à N threads de calcul parallèle simultanés. Ce modèle est parfaitement approprié au calcul parallèle sur GPU à l'aide de la plateforme CUDA et son implantation est présentée dans la section suivante.

II.3. Parallélisme du maillage par bulles

Le calcul parallèle sur GPU est bien approprié à la parallélisation de données dans laquelle plusieurs threads exécutent le même travail, mais sur des données différentes.

Les GPUs ont une puissance de calcul théorique très élevée et, d'une manière générale, une exécution sur GPU est souvent limitée par la bande passante mémoire. Dans un GPU classique, l'ordre de grandeur du ratio **flops / bytes** est de plusieurs dizaines à plusieurs centaines, ce qui revient à dire que le temps d'accès un byte équivaut au temps d'exécution de dizaines ou de centaines d'opérations en virgule flottante. Il est donc important de définir une structure de données bien adaptée et dédiée pour fluidifier la lecture et l'écriture des données sur GPU. Concernant le maillage par bulles, nous souhaitons généraliser le calcul parallèle sur toutes les étapes couteuses. Par contre, au regard de l'analyse des coûts présentée précédemment, l'ajout/suppression ou l'initialisation des bulles ne seront pas portés sur GPU. Les codes écrits en langage JAVA sont exécutés sur CPU tandis que les kernels CUDA en langage C/C++ sont exécutés sur GPU.

II.3.1. Structure de données

Le système de bulles comporte un ensemble des bulles. Chaque bulle est définie comme un objet avec ses variables, avec schématiquement la structure suivante :

```
Public class Bubble
{
   float coordinates[] ;
   float radius ;
   float speed[] ;
   float mass;
   float energy ;
}
Public class SystemBubbles
{
   Bubble[] bubbles ;
}
```

Bien que cette structure orientée objet soit évolutive et favorable à la programmation, elle n'est pas optimale pour l'architecture CUDA. La mémoire en CUDA est conçue pour servir plusieurs threads parallèles en même temps. Pour être plus précis, les demandes d'accès de 32 threads sont groupées par warp dans une ou plusieurs tranches de 128 bytes généralement. Dans le cas d'une approche orientée objet, les données des bulles vont être dispersées dans la mémoire, cela aboutit à des accès non coalescent [13] [57]. En conséquence, la bande passante mémoire s'écroule et cela nuit gravement à la performance de calcul du kernel.

Ainsi, au lieu d'une structure fragmentée comportant des bulles séparées, nous considérons l'ensemble des bulles comme un seul objet et introduisons une structure de données contenant un ensemble de vecteurs, illustré par la Figure II.24. Chaque vecteur correspond à une variable de l'ensemble des bulles comme les coordonnées, l'état, le rayon, la vitesse, l'énergie cinétique, tel que présenté ci-dessous :

```
Public class SystemBubble
{
   int number_bubbles;           // is N
   float[] coordinates;          // size N
   float[] radius;               // size N
   float[] speed;                // size N
   float[] mass;                 // size N
   float[] energy;               // size N
   int[] list_neighbors;         // size N * k, k is maximal number of neighbors
                                 // neigbor i of bubble j -> [i*N + j]
   ...
}
```

Le stockage des voisins est un peu particulier, dans la mesure où le nombre de bulles voisines d'une bulle est différent et change selon la position de chaque bulle durant le mouvement. En général, un système de pointeurs dans une liste dynamique est utilisé pour garder l'indice des voisins. Cependant, l'allocation dynamique dans un kernel CUDA est coûteuse et donc d'une utilisation très limitée [56] [58]. Pour éviter le pointeur sur une liste de bulles, nous introduisons « *k* » le maximum de nombre de voisins par bulle, et l'indice des bulles adjacentes sont stockées dans un tableau de *[N * k]* éléments avec N le nombre de bulles. De plus, l'ordre de stockage est optimisé (le voisin *i* de la bulle *j* est *liste_voisin[i*N + j]*) afin de produire un accès coalescent sur la mémoire du GPU par colonne, comme le montre dans la Figure II.24.

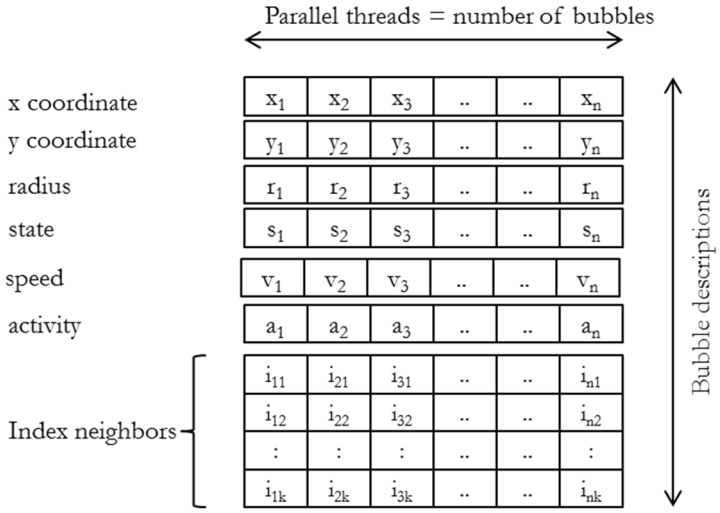


Figure II.24 Structure de données utilisée pour le calcul parallèle sur GPU

II.3.2. Construction de la liste de voisin

La formule (II.2) montre que la force d'interaction entre les bulles est paramétrée par la distance et on remarque en particulier que, si la distance entre deux bulles dépasse 1,5 fois la somme des rayons des deux bulles, la force est considérée comme nulle. Cette troncature est évidemment favorable au temps de calcul et c'est en particulier pour cette raison qu'il est important de maintenir une liste de voisins de chaque bulle. L'absence de pavage de l'espace demande de construire des approches dédiées pour construire et maintenir cette connaissance des bulles proches d'une bulle. Mais toutes les techniques ne sont pas nécessairement adaptées au calcul parallèle.

La liste de Verlet construit une liste de bulles proches à une distance r_m qui est supérieure à la distance de voisinage utile r_{cut}. Elle se base sur une mise à jour périodique et non une mise à jour à chaque itération. Donc le coût de construction de la liste de voisins se trouve divisé par « *m* » avec « *m* » la périodicité de la mise à jour. Celle-ci est déclenchée si le déplacement des voisins dépasse $(r_m - r_{cut})$. Cependant, pour vérifier quelles sont les bulles à la distance r_m d'une bulle, il faut toujours N-1 évaluations avec N-1 bulles, ce qui veut dire que l'évaluation (II.2) est à nouveau proportionnelle en $\Theta(N^2)$ du nombre de bulles. De plus, cette approche demande un stockage plus important car le nombre de voisins dans la liste de Verlet est plus nombreux que dans la liste strictement utile (détails dans [59])

La décomposition par Quadtree/Octree/KDtree peut améliorer l'évaluation de (II.2) en tendant vers un coût en $\Theta(NlgN)$ mais la structure de données de ces méthodes est complexe :

elle comporte un arbre avec une racine, des branches et des feuilles [60] [56]. Et surtout, pour vérifier l'information présente dans les feuilles de voisin, il faut traverser les données dans l'ordre : racine – branche – feuille. Du coup, cette approche augmente la besoin de communication entre les threads parallèles si, comme on souhaite le faire, à chaque bulle est assigné un thread. . Au final, cette approche de décomposition est un véritable défi à développer correctement sur GPU.

La grille uniforme [59] est utilisée dans la simulation dynamique moléculaire qui considère un ensemble d'atomes uniformes (de même taille). L'algorithme utilise une décomposition spatiale en cellules à l'aide d'une grille uniforme. La dimension de la cellule est choisie égale au rayon de l'atome. Chaque cellule possède alors un ou plusieurs atomes et un atome cherche ses voisins dans sa propre cellule et dans les cellules voisines. De cette façon, cette technique peut réduire l'évaluation des interactions entre les atomes en les ramenant à $\Theta(N)$. En plus, la structure de données est indépendante entre les cellules. Du point de vue du parallélisme, cette technique de grille est très favorable à une mise en place sur GPU puisqu'elle limite la communication entre les processeurs.

Dans notre contexte, nous proposons une technique similaire en y apportant les quelques modifications suivantes :

- la taille des bulles est variable selon leur position puisqu'elles doivent respecter la carte de taille définie par l'utilisateur. Elle est également variable pendant le mouvement. Cette différence essentielle influence le choix de dimension des cellules.

- le déplacement des bulles doit respecter la géométrie. Si une bulle sort dehors de la frontière de la géométrie, il faut arrêter de la déplacer et l'invalider pour qu'elle n'influence pas les bulles proches qui sont restées dans le domaine géométrique.

Algorithme général

Etape 1 : Découper le système de bulles par une grille uniforme et associer aux bulles une table d'indices de hachage.

Etape 2 : Trier et réorganiser le système des bulles par l'indice de hachage.

Etape 3 : Récupérer les bulles par cellules, vérifier la distance d'interaction entre les bulles dans la cellule et les cellules voisines pour rechercher les bulles voisines. Construire la liste des voisins.

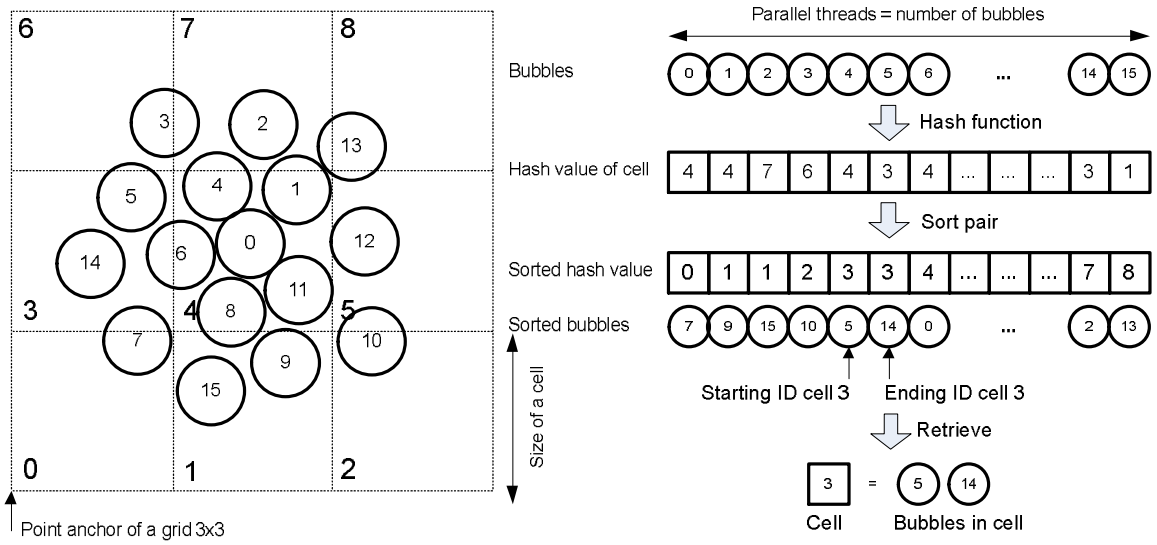


Figure II.25 : Illustration de la construction de la liste de voisins des bulles par une grille uniforme

Etape 1 : Discrétiser par la grille et la fonction de hachage

Le premier concept de cette technique est la discrétisation du système de bulles selon une grille uniforme et l'usage d'une fonction de hachage [61] [62]. La grille doit couvrir toute la géométrie. Pour simplifier, la taille de grille et ses cellules est choisie uniforme quel que soit la direction (carré en 2D ou cube en 3D). Donc, les paramètres de la grille sont le point d'origine, la taille d'une cellule et le nombre de cellules par direction. La Figure II.25 illustre une grille de 3x3 cellules uniformes. Le choix des paramètres de la grille est lié à une stratégie d'optimisation du temps d'exécution : le nombre de cellules de la grille est généralement choisi comme un multiple de 32 car CUDA exécute les threads parallèles en groupe de 32 threads simultanément par warp.

L'objectif de la fonction de hachage est de mettre en correspondance les cellules comportant des bulles intérieures avec une table d'indices. Pour une bulle au point $p = (x,y)$ en 2D, l'indice de la cellule correspondant aux 2 dimensions vaut :

$$(i,j) = \left(\left\lfloor \frac{x - x_0}{cellsize} \right\rfloor, \left\lfloor \frac{y - y_0}{cellsize} \right\rfloor\right) \qquad \text{(II.10)}$$

avec :

`(x₀, y₀)` est le point d'origine de la grille

`cellsize` est la dimension de la cellule

$\lfloor\ \rfloor$ est l'opération d'arrondi « floor »

La fonction de hachage permet de construire une indice h à partir deux indice (i,j) selon $h = f(i,j)$. Il y a plusieurs façons de construire une fonction de hachage, qui doit satisfaire des propriétés de distribution uniforme et unique [63] et [62]). Pour les bulles, on utilise une fonction de hachage simple à partir l'indice de la cellule même si d'autres fonctions comme le z-ordre ou le code Morton pourraient améliorer la performance [64]:

$$f(i,j) = (j \bmod gridSize) * gridSize + (i \bmod gridSize) \qquad \text{(II.11)}$$

avec *gridSize* est le nombre de cellules dans chaque direction de la grille.

Par conséquence, toutes les bulles dans une même cellule ont une même valeur de hachage.

Etape 2 : Trier et réorganiser des données par l'indice de hachage

Ensuite, nous trions la table des indices de hachage et réorganisons les données des bulles selon leur indice de hachage. En conséquence, toutes les données des bulles dans la même cellule seront stockées dans un segment contigu de mémoire. Cela assure également que les données des cellules adjacentes seront stockées à proximité. L'avantage de cette technique est de réordonner les bulles par la proximité, ce qui entraîne un accès mémoire plus cohérent. Au passage, il est à noter que l'algorithme de tri utilisé atteint une excellente performance sur la plupart des cartes graphiques supportant CUDA [19]. La Figure II.26 illustre la réorganisation les données des bulles correspondant à trois cellules i, j, k rendues différentes par leur couleur. Chaque carré représente les données d'une bulle. Les données des bulles avant le tri sont désordonnées à cause de l'initialisation ou du mouvement des bulles. Les bulles après le tri sont regroupées par leur cellule correspondante. Ainsi, l'accès aux données est bien coalescent au sens du GPU.

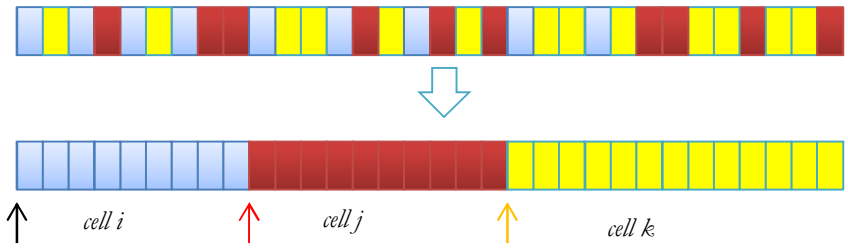


Figure II.26 Illustration de la réorganisation des données des bulles selon leurs cellules

Pour rechercher les bulles dans une cellule, il suffit de 2 pointeurs qui pointent dans la séquence de données des bulles correspondant à cette cellule : un indice de début et un indice de fin. En pratique, on utilise deux tables qui stockent les deux indices pour toutes les cellules, pour gérer plus simplement le cas des cellules vides.

Etape 3 : Rechercher des voisins par cellules et construire la liste de voisins

Les voisins de chaque bulle peuvent se trouver facilement en examinant la cellule comportant la bulle et ses cellules adjacentes, comme illustré par la Figure II.27. Le champ de recherche est différent selon la taille de la bulle : la distance est plus grande pour de grandes bulles et plus petite pour petites bulles. Revenons à la distance caractéristique de la force équivalant à 1,5 fois la somme des rayons des deux bulles. Si les deux bulles ont la même taille, la force s'annule pour une distance de 3,0 fois le rayon de la bulle. En pratique, en prenant en compte la différence de taille entre les bulles, nous élargissons le champ de recherche à 4,0 fois le rayon de la bulle.

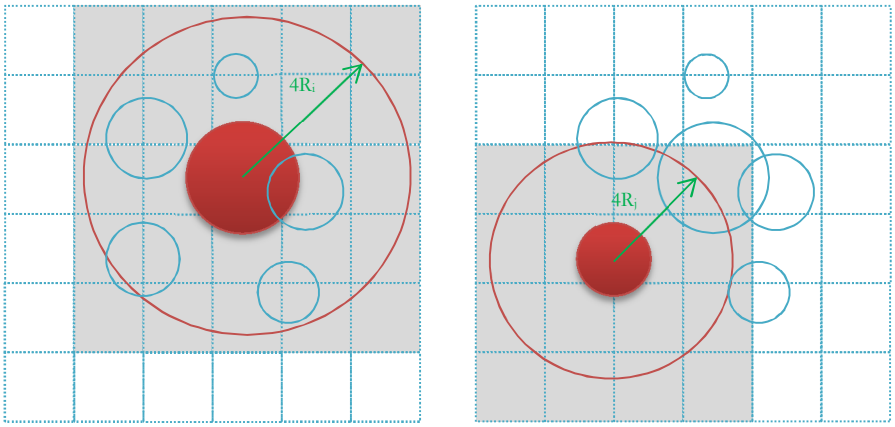


Figure II.27 : Rechercher les voisins d'une bulle par sa cellule et les cellules adjacentes. Le champ de recherche (cercle rouge) dépend du rayon de la bulle. Les bulles dans toutes les cellules correspondant au champ de recherche sont récupérées.

Le rapport entre la taille de la cellule et la taille de la bulle est un paramètre important. Une cellule de grande taille comportant un nombre conséquent de bulles peut augmenter le nombre de bulles à examiner et à prendre en compte dans le calcul de la force par exemple. A contrario, une grille plus fine demande plus de mémoire et augmente aussi le taux des cellules « vides ». Par expérience, nous limitons le nombre maximal admissible de bulles dans une cellule. Ce paramètre est normalement déterminé au lancement de l'exécution pour permettre son optimisation. Il faut contrôler le nombre réel de bulles par cellule pendant le mouvement des bulles. Si le nombre réel de bulles dans une cellule dépasse largement la limite, une grille plus fine doit être recalculée. Dans ce cas, toutes les données concernant la grille (table de hachage, indice de début et de fin) sont réinitialisées en même temps.

Mise en œuvre

Quatre **kernels** CUDA sont impliqués pour la construction de la liste de voisins:

1) **Calculer les indices de hachage**:

```
__global__ void calcHashDevice(d_systemBubbles, d_gridData, d_hashkey, d_index)
```

A chaque thread est assigné à une bulle. Cette méthode charge la localisation des bulles (*d_systemBubbles*) et les données de la grille (*d_gridData*) afin de déterminer la cellule correspondant à la bulle. La sortie est le tableau des indices de hachage de cellules. En pratique, nous utilisons deux vecteurs de taille N avec N le nombre des bulles : *d_hashkey* pour stocker l'indice de hachage et *d_index* pour garder l'indice des bulles (« *d_* » signifiant des données sur la mémoire de GPU).

2) **Trier et réorganiser des données** de bulles par ses indices de hachage:

Il y a deux solutions :

La première solution repose sur une structure qui permet de combiner des données pour synchroniser l'indice de hachage et les données des bulles. Il s'agit de la notion de « tuple » et « zip_iterator » du package Thrust CUDA: Le « tuple » est une structure qui peut combiner des composants différents dans un objet, le « zip_itérator » groupe des vecteurs différents dans un seul vecteur. Dans le cas des bulles, le vecteur des indices de hachage peut être combiné avec les données des bulles (qui sont unifiées à travers le « tuple ») dans un « zip_iterator » de « tuple ». Le

tri se fait sur tous les composants à la fois. Le coût de cette combinaison de structures est élevé. En pratique, nous utilisons la solution suivante.

La deuxième solution utilise un vecteur supplémentaire pour garder l'ordre du tri des indices de hachage et puis guider la suite de la réorganisation par cette table. En pratique, nous utilisons le tri par paire pour deux vecteurs d'indice : *d_hashkey* et *d_index*. En particulier, la table *d_index* est triée en accord avec l'ordre de *d_hashkey*. La méthode de tri par paire utilisé est le « radix-sort » du package CUDPP (CUDA Data-Parallel Primitives) ou le « merge-sort » du package THRUST. Ensuite, la réorganisation des données de bulles est guidée par les indices *d_index* triée. En pratique, c'est une permutation des données des bulles entre ses anciens indices et ses nouveaux indices. Une base de données temporaire (*d_temp*) est utilisée pour assurer le bon résultat de la permutation parallèle. Le kernel à réorganiser est sous la forme :

```
__global__ void reoderDataDevice(d_index, d_systemBubbles, d_temp)
```

où chaque thread correspond à la permutation d'une bulle.

3) **Rechercher l'indice de début et de fin de cellules**:

Deux vecteurs de indice de taille M avec M le nombre de cellules sont utilisées dans ce cas: *d_cellstart* pour l'indice de début et *d_cellEnd* pour l'indice de fin.

```
__global__ void findCellStartEnd(d_hashkey, d_cellStart, d_cellEnd)
```

Chaque thread correspond à une cellule. Pour les cellules vides, les deux indices sont mis à la valeur -1 pour exprimer une exception.

4) **Etablir la liste des voisins**:

```
__global__ void buildListDevice(d_systemBubbles, d_gridData, d_cellStart, d_cellEnd)
```

Ce kernel va construire la liste des voisins pour chaque bulle. Il récupère les bulles sur la cellule de la bulle et sur les cellules voisines, vérifie la relation de voisinage et garde l'indice des bulles voisines dans une liste.

Au lieu de calculer la liste de voisins à chaque itération, nous la reconstruisons périodiquement (toutes les 10 itérations typiquement). Pour ce faire, nous devons étendre la distance de recherche (4 fois le rayon de la bulle au lieu des 3 fois normalement) et enregistrer un plus grand nombre de voisins. Dans la pratique, ce choix nécessite plus de mémoire, mais donne une exécution plus rapide.

II.3.3. Intégration

Nous utilisons une intégration de Runge-Kutta du 4ème ordre où chaque étape nécessite 4 évaluations de forces en s'appuyant sur la liste de voisins. Ce procédé intègre les attributs de la bulle (position et vitesse) pour déplacer la bulle : la vitesse est mise à jour en fonction des forces appliquées, puis la position est mise à jour en fonction de la vitesse.

L'intégration de chaque bulle est calculée par chaque thread CUDA :

```
__global__ void integrationRK4Device(d_systemBubbles, d_temp)
```

Les données des bulles sont utilisées non seulement par le thread correspondant mais aussi par les threads qui correspondent aux bulles voisines. Pour que la mise à jour des données d'une bulle ne perturbe pas ses voisines, la nouvelle valeur de position et de vitesse des bulles sont enregistrés dans une variable temporaire (*d_temp*).

Du point de vue du kernel, la liste des voisins ne change pas pendant l'intégration et il faut évaluer 4 fois les forces à chaque itération. Le kernel demande donc un grand nombre de registres. Dans ce cas, l'usage de la mémoire partagée est recommandé. Les données des voisins seront chargées dans la mémoire partagée durant la première évaluation et restera utilisable pour les trois évaluations suivantes.

De plus, la lecture des données des bulles voisines est évidemment non coalescente. Cependant, ces dernières sont assez proches en mémoire grâce à la réorganisation. Dans ce cas, la mémoire de texture de GPU (tex1Dfetch) peut améliorer la performance (jusqu'à 45% en théorie) car la lecture sur cette mémoire est mise en cache.

II.3.4. Mise à jour la taille des bulles

La mise à jour des tailles de bulles s'appuie sur le maillage triangulaire grossier (décrit par des coordonnées des nœuds, une connectivité de triangles, des indices de triangles voisins) et la valeur de taille prédéfinie aux nœuds. La Figure II.28 illustre cette base de données pour le calcul parallèle sur GPU.

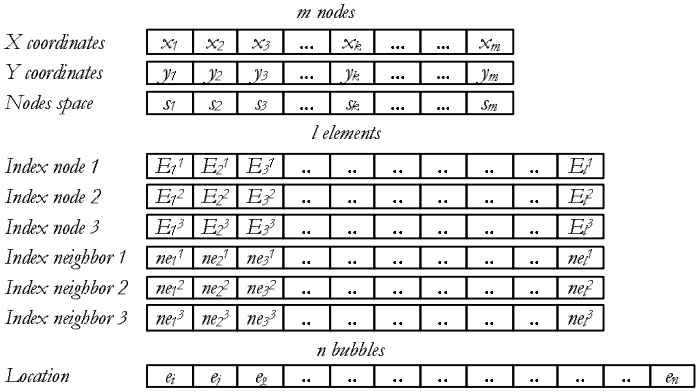


Figure II.28 : Structure de données pour la mise à jour la taille des bulles: le maillage de fond et la valeur prédéfinie aux nœuds

Chaque thread CUDA prend en charge le calcul d'une bulle en parallèle et les opérations élémentaires sont les suivantes : charger les données nécessaires au maillage (*d_meshData*), rechercher la localisation du triangle contenant (*d_location*) et mettre à jour le rayon (*d_systemBubbles*).

```
__global__ void updateRadiusDevice(d_systemBubbles, d_meshData, d_location)
```

L'interpolation de la taille des bulles basée sur un maillage triangulaire est a priori simple et nécessite un minimum de traitement, mais elle présente cependant des défis à surmonter pour des architectures CUDA. Tout d'abord, il est difficile de coalescer la lecture des données des nœuds avec un maillage non structuré. En outre, les triangles voisins partagent des nœuds, la lecture des nœuds par plusieurs threads repose donc sur une redondance des données, qui peut s'avérer couteuse en particulier en cas d'un nombre élevé de triangles.

II.3.5. Mise en œuvre du calcul parallèle sur GPU

La Figure II.29 présente l'exécution hétérogène CPU et GPU dans lequel tous les calculs lourds sont parallélisés et mis en place sur GPU tandis que tous les procédés de contrôle sont exécutés sur CPU. Les données des bulles sont transférées dans la mémoire du GPU une fois quand un nouvel ensemble de bulles est créé dans la boucle interne.

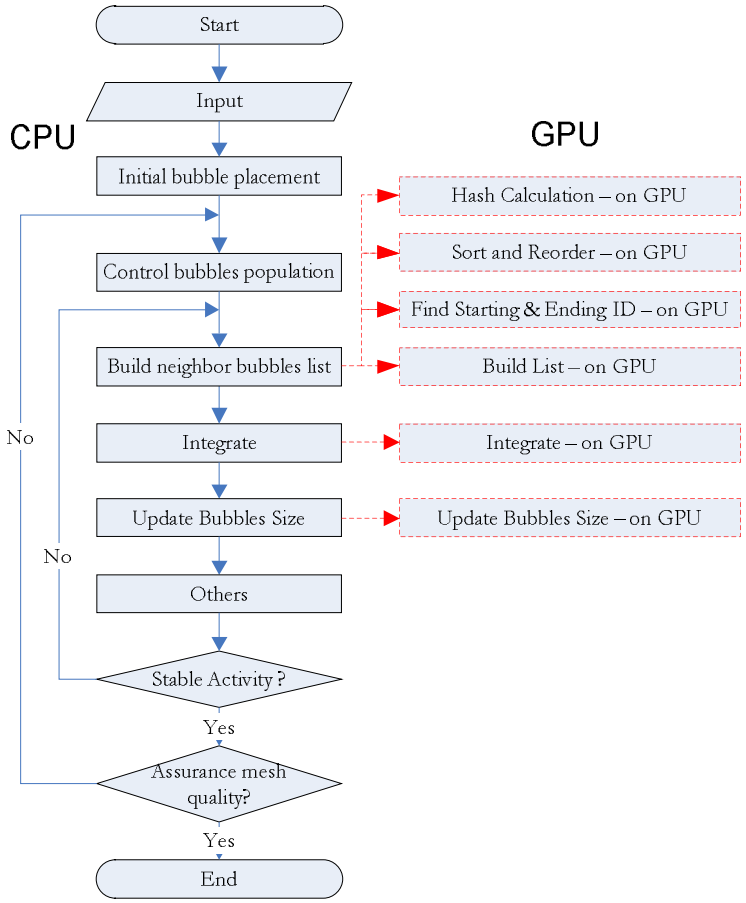


Figure II.29 Schéma d'opérations de maillage par bulles sur une plateforme hétérogène CPU et GPU

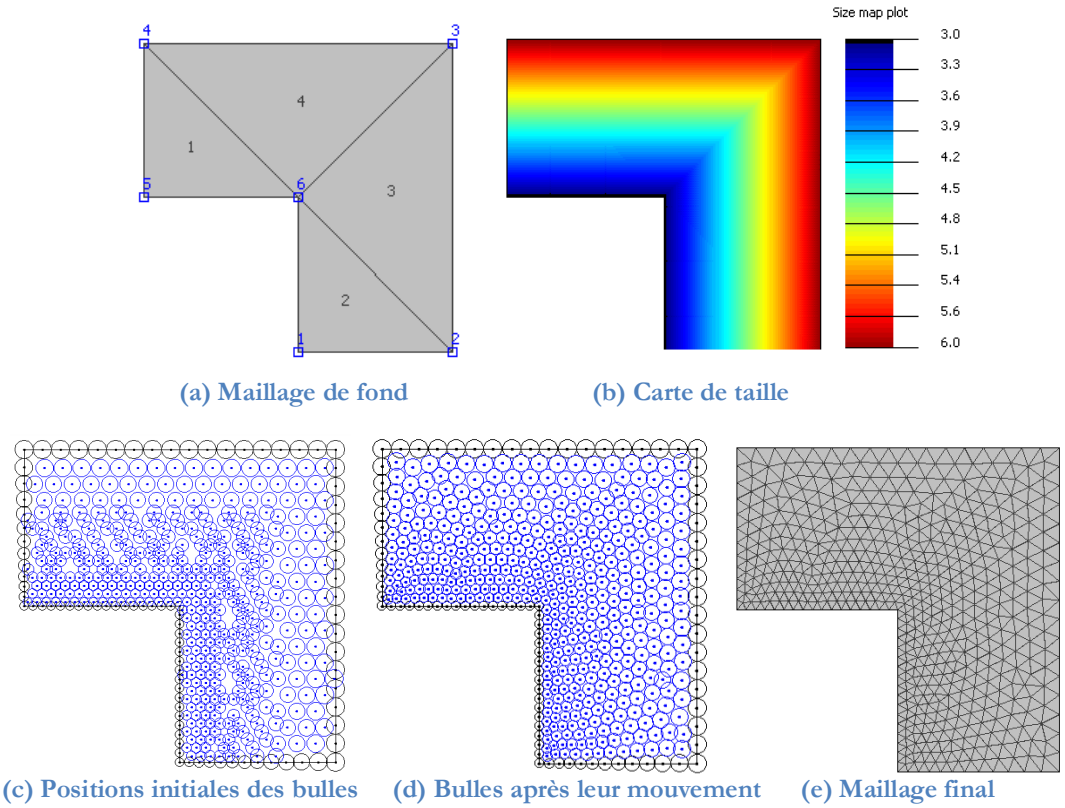
II.4. Evaluation de performance

II.4.1. Spécification du hardware

Pour tester les performances de l'algorithme parallélisé, nous réalisons des expériences sur un ordinateur avec un processeur CPU Intel Xeon 2,67 GHz, 4 Go RAM et un GPU NVIDIA Tesla C1060 240 cœurs, 1,3 GHz, 4 Go de mémoire globale. Le code d'origine est développé en langage JAVA. Le code du kernel CUDA est porté en JAVA en utilisant la package JCUDA v5.5, package JCUDPP v2.1 et JNI v1.6. Cette carte est relativement ancienne (sortie initiale en septembre 2008) et a une capacité de calcul 1.3, pour 933 GFLOPS en simple précession (32 bits) et une bande passante mémoire de 102 GB/s. Cette carte a servi de support au test du mailleur uniquement. Les autres tests de cette thèse seront faits sur une génération plus récente (capacité 3.5), qui sera précisée le moment venu.

II.4.2. Cas d'une géométrie en forme de L

Tout d'abord, on considère le maillage par bulles dans un cas simple avec un domaine comportant une surface en forme de L. Le maillage est illustré par la Figure II.30. La taille des bulles est contrôlée par une fonction linéaire qui se base sur un maillage de fond comme le montre les figures a et b. La technique de quadtree modifié est utilisée, ce qui donne une configuration initiale en hexagone des bulles comme présenté sur la figure c. L'insertion et la suppression automatique des bulles contrôlent le nombre de bulles intérieures du domaine et le mouvement des bulles régularise leur position. Le résultat final des bulles et le maillage obtenu sont présentés aux figures d et e.



(a) Maillage de fond (b) Carte de taille

(c) Positions initiales des bulles (d) Bulles après leur mouvement (e) Maillage final

Figure II.30 Illustration du maillage par bulles dans le domaine en L

Le Tableau II-1 montre la comparaison du temps entre les deux implantations, sur l'hôte (CPU) et sur le processeur graphique (GPU) selon le nombre de bulles. Les deux implantations sont faites sur le même jeu de bulles initiales. Le contrôle de densité s'effectue régulièrement pour assurer le nombre approprié de bulles. Ces deux procédures sont exécutées sur CPU avec un temps comparable. Quant au mouvement, nous avons fixé le même nombre d'itérations d'intégration de bulles pour les deux applications. Cependant, les positions finales des bulles changent un peu parce que les deux applications mettent à jour les données des bulles de manière différente. En effet, le code CPU traite chaque bulle séquentiellement et la met à jour dans la foulée, alors que le code GPU traite toutes les bulles en parallèle en même temps et la mise à jour ne s'effectue que lorsque toutes les bulles sont traitées, ce qui justifie une légère dégradation des indicateurs de qualité.

Tableau II-1 Comparaison de temps de fonctionnement de deux exécutions : sur CPU et sur GPU

Chronomètre	2500 bulles			6500 bulles			10000 bulles		
	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU
Initialisation, ms	16	16	-	16	16	-	31	31	-
Contrôle de population, ms	94	47	-	210	187	-	531	296	-
Mouvement, ms	2730	172	15,87	9930	187	53,10	16505	234	70,53
- construction de liste, ms	1885	15	125,67	5918	15	394,53	12701	15	846,73
- intégration, ms	466	31	15,03	2294	46	49,87	2232	94	23,74
- contrôle de taille, ms	189	30	6,30	767	31	24,74	658	31	21,23
- autres, ms	190	96	-	951	95	-	914	94	-
Temps total, ms	2840	235	12,08	10156	390	26,04	17067	561	30,42
Irrégularité topologique	0,170	0,196	-	0,137	0,16	-	0,149	0,153	-
Irrégularité géométrique	0,012	0,013	-	0,010	0,011	-	0,011	0,011	-

Le résultat montre une accélération significative du calcul sur GPU par rapport à celui sur CPU tandis que la qualité des maillages reste comparable. L'accélération du code parallèle augmente linéairement avec le nombre de bulles. Dans le cas de 2500 bulles, le temps total est réduit de 12 fois. Cette réduction est de 26 fois pour 6500 bulles et de 30 fois pour 10 000 bulles. L'accélération du calcul parallèle dépend également de la procédure. La construction de la liste de voisins parvient à une accélération supérieure au calcul d'intégration ou au contrôle de taille. La Figure II.31 montre l'accélération dans le cas de 10 000 bulles : le temps total de maillage atteint une accélération jusqu'à 30x et le mouvement jusqu'à 70x. Quant à la mise à jour de la liste des voisins, elle présente une accélération extrême de 846x grâce à son algorithme parallèle bien adapté. L'intégration et la mise à jour des tailles ont une accélération plus faible (23x et 21x) en raison de des nombreux registres utilisés et des données non structurées du maillage.

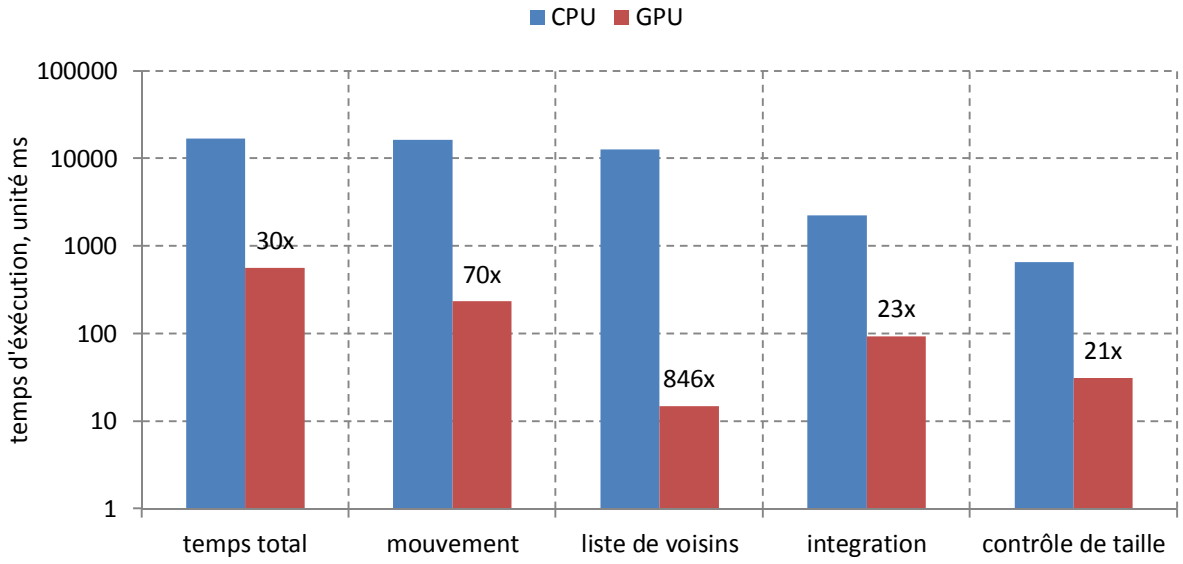


Figure II.31 Accélération du code GPU contre CPU dans le cas de la surface en L avec 10 000 bulles

La qualité du maillage final est très bonne avec une faible irrégularité et l'irrégularité géométrique se maintient à environ 1%. En conclusion, notre algorithme de maillage parallèle par bulles peut générer des maillages avec une qualité comparable à la version d'origine, qui est bien supérieure aux approches conventionnelles de Delaunay.

II.4.3. Cas d'une géométrie complexe

La Figure II.32 montre une géométrie plus complexe, un profil 2D d'un moteur, qui se compose de 25 régions surfaciques. Le maillage de fond possède 196 nœuds, 338 triangles et la carte de taille est construite sur cette discrétisation afin de contrôler la taille de bulles. Nous avons effectué le maillage par bulles tout d'abord aux nœuds, sur les lignes et puis sur toutes les surfaces. Le maillage est exécuté successivement par surface, le maillage final obtenu est présenté à la Figure II.33. Le temps total de traitement sur CPU et GPU est résumé dans le Tableau II-2 .

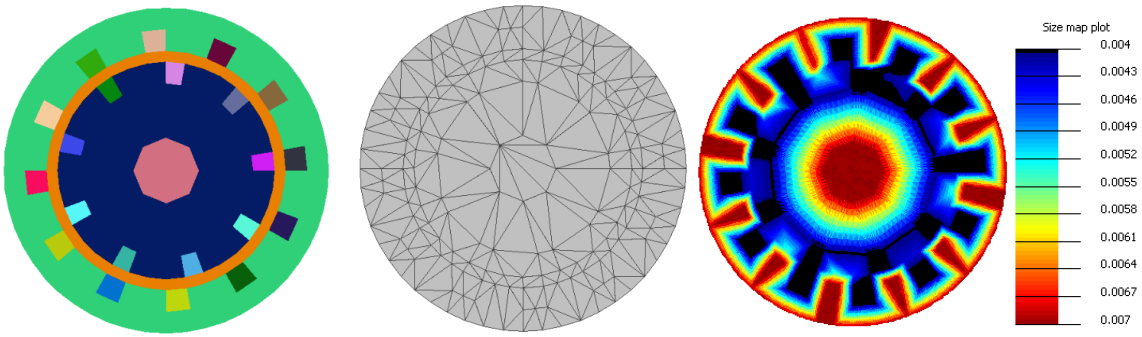


Figure II.32 : Illustration du cas de géométrie complexe à mailler sous forme d'un moteur

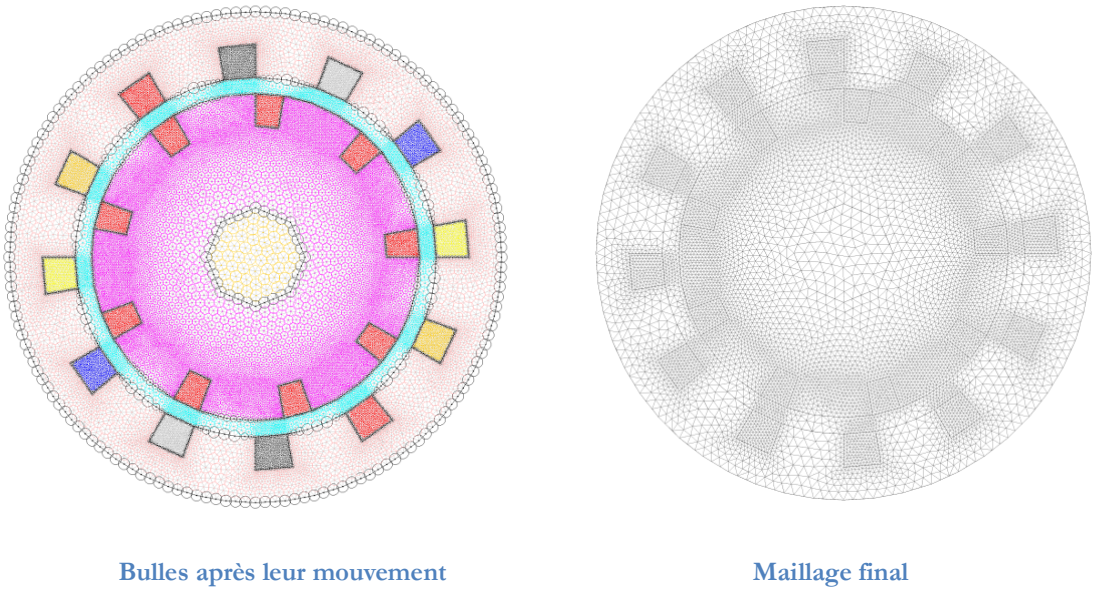


Figure II.33 : Maillage par bulles dans le cas de géométrie complexe

En comparant la durée entre CPU et GPU, on peut remarquer une réduction de temps d'un ordre de magnitude. L'accélération atteint de 9,3x pour le cas de 12 000 bulles, et respectivement 10,1x et 14,1x pour 24 000 et 51 000 bulles. On constate que le bénéfice du code parallèle augmente avec le nombre de bulles mais l'intensité globale de performance est non prévisible et dépend de la complexité de la géométrie. Par rapport au cas de la géométrie en L, l'accélération du code parallèle dans le cas de ce moteur est moins grande. L'accélération pour le cas de 12 000 bulles du profil de moteur peut se comparer à celle obtenue dans le cas précédent (en L) pour 2500 bulles. Pourquoi ? Si l'accélération du calcul parallèle dépend bien du nombre de bulles traitées, pour cette géométrie composée de 25 surfaces et qui est traitée surface par surface, le nombre de bulles de chaque surface est souvent bien inférieur à 1000 bulles. Par expérience,

l'avantage de calcul parallèle est vraiment notable à partir de 1000 bulles. De plus, le coût de réinitialisation la grille pour chaque surface grève la performance totale. La complexité de la géométrie ralentit enfin la procédure de mise à jour de la taille des bulles.

Tableau II-2 Comparaison du temps d'exécution CPU et GPU dans le cas de la géométrie complexe

Chronomètre	12 000 bulles			24 000 bulles			51 000 bulles		
	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU
Initialisation, ms	390	390	-	795	795	-	1404	1404	-
Contrôle de population, ms	408	157	-	1574	497	-	4024	671	-
Mouvement, ms	13236	966	13,70	35176	2496	14,09	93166	4890	19,05
- construction de liste, ms	10078	419	24,05	27387	682	40,16	77614	1882	41,24
- intégration, ms	2138	265	8,07	4919	514	9,57	9654	951	10,15
- contrôle de taille, ms	809	221	3,66	1694	423	4,00	3616	642	5,63
- autres, ms	211	61	-	1176	877	-	2282	1415	-
Temps total, ms	14034	1513	9,27	37545	3696	10,16	98594	6964	14,16
Irrégularité topologique	0,175	0,175	-	0,135	0,138	-	0,098	0,101	-
Irrégularité géométrique	0,016	0,090	-	0,012	0,012	-	0,008	0,009	-

Nous analysons le cas de 51 000 bulles par la Figure II.34. Pour une accélération globale d'environ 14x, le mouvement atteint 19x et la mise à jour des voisins est d'environ 41x.

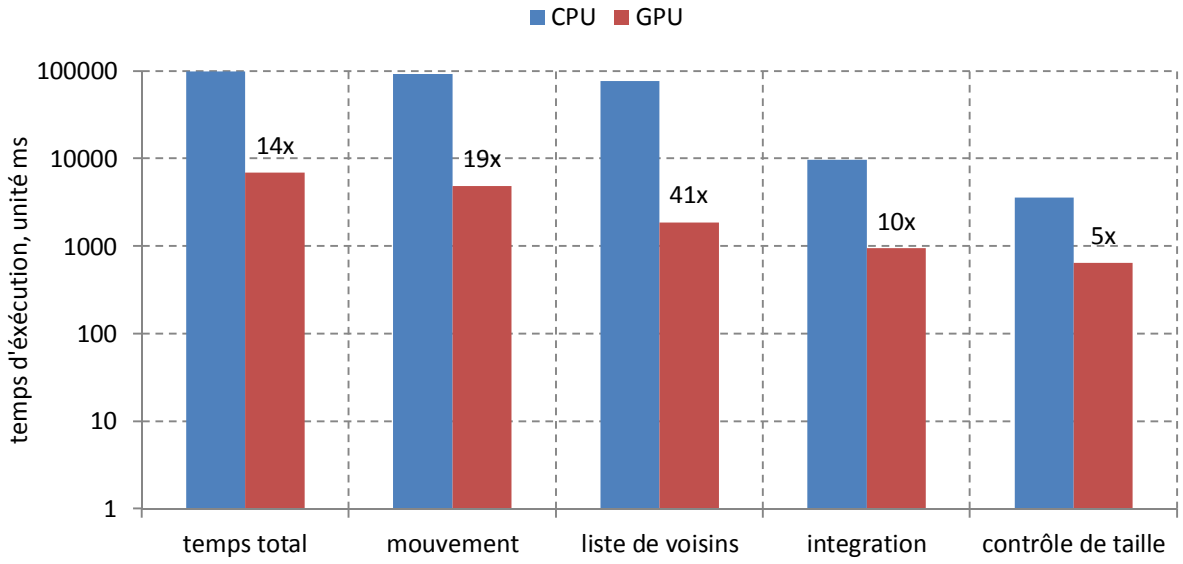


Figure II.34 : Accélération du code GPU contre CPU dans le cas du moteur avec 51000 bulles

En conséquence, le maillage par bulles parallélisé sur GPU peut remplir 51 000 bulles en environ 7 secondes, ce qui équivaut à un maillage d'environ 10^5 éléments triangulaires et $2\ 10^5$ nœuds pour un maillage du second ordre. Cette performance est à ramener à une carte qui a 8 ans d'âge. La carte plus récente avec laquelle nous avons fait nos tests par la suite à des performances trois fois supérieures. L'utilisation de cette nouvelle carte dans ce contexte aurait ramené le temps de calcul à 2 ou 3 secondes.

Tableau II-3 Comparaison du maillage par bulles et maillage de Delaunay par GMSH

Comparaison	Maillage par bulles	Maillage par GMSH
# points	51637	58667
# éléments	104752	116812
50 < # angles	86,14 %	40,50 %
35 < # angles ≤ 50	13,79 %	59,39 %
20 < #angles ≤ 35	00,07 %	00,01%
0 < angle ≤ 20	0 %	0 %
Temps total	6,96 secondes	8,21 secondes
Irrégularité topologique ε_t	0,126	Non évaluable
Irrégularités géométrique ε_g	0,009	0,029

Le Tableau II-3 montre la comparaison avec le maillage de Delaunay du package GMSH[4] .

Avec la même géométrie dans le cas de 51 000 points, le maillage par bulles utilise moins de points (51 637 points vs. 58 667 points de GMSH) et fournit un maillage de meilleure qualité. Les angles des éléments du maillage sont répartis selon 4 catégories : « excellent » pour des angles supérieurs à 50 degrés, « bon » pour des angles de 35 degrés à 50 degrés, « moyen » pour des angles de 20 à 35 degrés et « mauvais » pour des angles inférieurs à 20 degrés. On constate en effet, que les deux méthodes limitent les proportions d'angles « moyens » et « mauvais », mais le maillage par bulles génère une très grande majorité d'angle « excellents » (près de 9 sur10) alors que la méthode de Delaunay de GMSH se concentre plutôt dans la catégorie des angles « bons ». Cela est également vérifié par l'indicateur d'irrégularités. Les irrégularités du maillage créé par les bulles sont plus 3 fois plus petites : irrégularité géométrique 0,009 vs 0,029 pour GMSH. (L'irrégularité topologique n'est pas disponible pour GMSH).

La distribution des angles tracée à la Figure II.35 forme une pointe à 60 degrés pour le maillage par bulles et est plus plate pour le maillage de Delaunay. La plupart des éléments triangulaires sont quasi équilatéraux pour le maillage par bulles. Quant au temps d'exécution, la performance du maillage par bulles parallèle est comparable au maillage de Delaunay de GMSH (6,96 secondes contre 8,21 secondes). Notons que le temps du maillage par bulles ne tient compte pas du temps de triangulation de Delaunay.

[4] Le maillage 2D pour les surfaces plat de GMSH utilise l'algorithme de maillage de Delaunay qui se base sur le travail le travail de l'équipe GAMMA au INRIA [65]

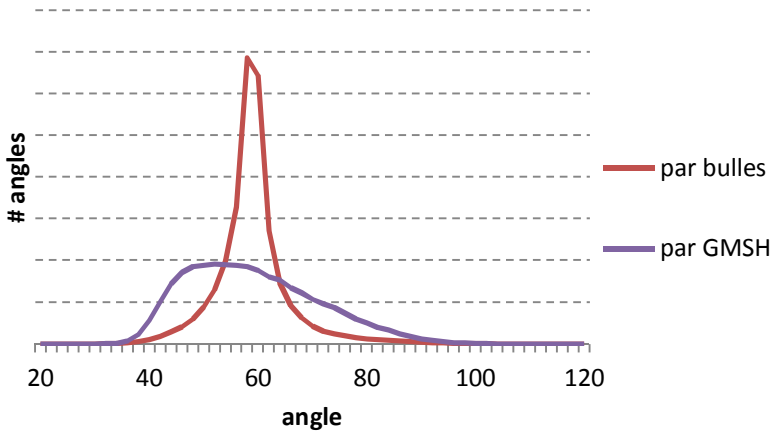


Figure II.35 Distribution des angles des éléments du maillage par bulles et de Delaunay par GMSH

II.5. Conclusion

Nous avons présenté les détails de la méthode de maillage par bulles et son exécution parallèle sur GPU. Lors des tests, nous avons présenté un domaine 2D complexe avec de nombreuses surfaces. Le mailleur par bulles traite chaque surface séquentiellement et la génération des nœuds intérieurs de la surface est parallélisé sur GPU. Le mailleur parallèle par bulles s'adapte bien à la géométrie du domaine en répondant aux conditions de taille et de forme. La qualité du maillage obtenu est meilleure que celle obtenue par un raffinement de Delaunay conventionnel. Cela confirme la capacité de maillage automatique de bonne qualité du mailleur par bulles qui peut répondre aux contraintes des problèmes industriels modélisés par élément finis.

La performance du mailleur par bulles a été améliorée considérablement par le calcul parallèle sur GPU. En général, l'accélération du maillage parallèle atteint des gains de 10 à 30 par rapport sa version en séquentielle et est capable de mailler 100 000 éléments triangulaires d'une géométrie complexe en quelques secondes sur une machine très modeste. Le temps de maillage par bulles parallélisé est comparable au maillage de Delaunay conventionnel. L'intensité d'amélioration du calcul parallèle dépend du problème et tout particulièrement de la complexité de la géométrie (le nombre et la forme des surfaces), de la variété de taille des éléments dans l'espace et de la qualité désirée du maillage. Non seulement l'intégration des bulles mais aussi tous les calculs concernant le mouvement comme la construction de la liste des voisins, le calcul de la force, la mise à jour du rayon, le contrôle de la convergence sont parallélisés. Seul le contrôle de la population des bulles qui traite l'insertion et la suppression n'est pas parallélisable.

Le calcul parallèle sur GPU demande une structure de données adaptée pour les bulles qui facilite l'accès sur la mémoire du GPU. Si la structure orientée objet apporte flexibilité et souplesse en programmation, son stockage est dispersé en mémoire. Cela augmente considérablement le coût d'accès aux données. Au lieu de cette structure, une structure vectorielle plus favorable pour le GPU a été proposée. Les données du système de bulles sont regroupées dans une structure comportant plusieurs vecteurs compacts. Chaque vecteur correspond à un paramètre des bulles. En conséquence, cette structure de données permet aux 32 threads du warp CUDA d'accéder à des données coalescentes en parallèle à un coût réduit.

L'usage de la technique de décomposition par une grille uniforme et le tri des données joue un rôle essentiel pour le calcul parallèle des interactions entre bulles. La construction de la liste de

voisin permet de renforcer le calcul d'intégration des bulles. Le coût du calcul de force entre les bulles actives est le plus lourd (normalement $O(n^2)$ dans la modélisation de n-corps). L'intégration RK4 demande 4 évaluations de la force par itération. En utilisant une liste de voisins commune pour les 4 évaluations au lieu de 4 mises à jour de voisins, le coût de calcul de force est fortement réduit. Cela permet également de négliger la mise à jour des voisins de chaque bulle lorsque le système des bulles atteint sa stabilité avec une mobilité faible. Avec le calcul parallèle sur GPU, la construction de la liste de voisins peut être accomplie à l'aide de la technique de la **grille uniforme**. La technique se base sur la discrétisation du système de bulles dans l'espace par une grille uniforme et le tri des données des bulles selon leur localisation. Cette technique s'adapte bien au système de bulles de tailles variables, tel que sont les maillages qui intéressent l'utilisateur.

Nous avons limité nos travaux au 2D en sachant qu'il est possible d'étendre l'exécution parallèle sur GPU du maillage par bulles au 3D. Il est cependant nécessaire de faire quelques adaptations pour pouvoir mettre en œuvre le calcul 3D, surtout au plan géométrique et concernant la triangulation de Delaunay que nous n'abordons pas dans ce travail. Du point de vue de la parallélisation, les calculs géométriques en 3D sont plus compliqués qu'en 2D mais favorables à une exécution sur GPU car ils reposent sur des calculs d'arithmétique.

CHAPITRE III

Parallélisation de la méthode des éléments finis sur GPU

SOMMAIRE

Dans ce chapitre, nous abordons le portage du calcul de la méthode des éléments finis (MEF), à savoir les trois procédures essentielles : l'intégration numérique, l'assemblage et la résolution, sur une plateforme de calcul parallèle à GPU. L'étude repose sur la formulation magnétostatique bien qu'il soit possible d'appliquer nos algorithmes sur d'autres domaines de la MEF en électromagnétisme, voire sur d'autres physiques. Le contenu de ce chapitre se décompose en trois parties :

- Définition du problème magnétostatique en rappelant les formulations, ainsi que la technique d'assemblage et la résolution des équations algébriques.
- Projection de nombreux algorithmes concernant l'intégration et l'assemblage dans le contexte du calcul massivement parallèle sur GPU. Trois techniques principales sont comparées au regard de leur efficacité.
- Utilisation de solveurs itératifs et de préconditionnement parallèle pour la résolution.

III.1. Magnétostatique et la discrétisation de MEF

III.1.1. Problème de magnétostatique

III.1.1.1 Rappel des équations de Maxwell

Les phénomènes en électromagnétisme à l'échelle macroscopique sont caractérisés par le système des quatre équations de Maxwell. Elles décrivent la relation des champs électrique et magnétique à leurs sources, la densité du courant et la densité de la charge électrique. Les équations sont :

$$\mathbf{rot}\,\mathbf{H} = \mathbf{J} + \partial_t \mathbf{D} \quad \text{(III.1)}$$

$$\mathbf{rot}\,\mathbf{E} = -\partial_t \mathbf{B} \quad \text{(III.2)}$$

$$\text{div}\,\mathbf{B} = 0 \quad \text{(III.3)}$$

$$\text{div}\,\mathbf{D} = \rho \quad \text{(III.4)}$$

Avec

H est le vecteur de champ magnétique (A/m)

B est le vecteur d'induction magnétique (Tesla ou T)

E est le vecteur de champ électrique (V/m)

D est le vecteur d'induction électrique (V/m^3)

J est le vecteur de densité du courant (A/m^2)

ϱ est la valeur scalaire de densité volumique de la charge électrique (C/m^3)

t est le temps (s)

III.1.1.2 Définition du problème magnétostatique

Le problème est défini sur un domaine Ω comportant deux types de région : $\Omega = \Omega_0 \cup \Omega_m$ avec Ω_m la région perméable, et Ω_0 la région d'air. La région perméable est caractérisée par un comportement ferromagnétique. La région air comprend des sources de champ qui sont soit des

bobines inductrices parcourues par un courant continu de densité **J** donnée, soit des aimants permanents d'induction rémanente $\mathbf{B}_r$. Elle se compose d'une boite d'air suffisamment grande.

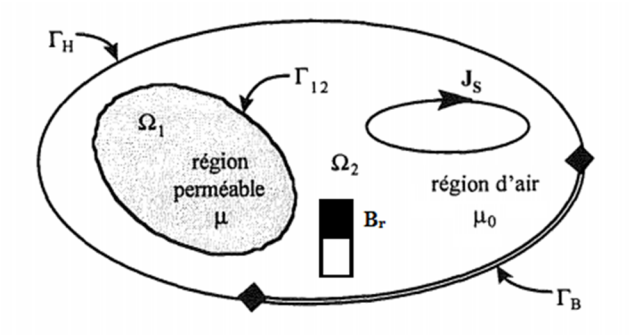


Figure III.1 : Définition du problème magnétostatique

Le problème magnétostatique consiste à étudier les phénomènes électromagnétiques en régime stationnaire. Dans ce cas, les champs électromagnétiques sont indépendants du temps. Les dérivées temporelles sont annulées. Les équations de Maxwell s'écrivent :

$$\mathbf{rot}\,\mathbf{H} = \mathbf{J} \qquad \text{(III.5)}$$

$$\text{div}\,\mathbf{B} = 0 \qquad \text{(III.6)}$$

Le comportement magnétique du matériau s'exprime par les équations constitutives générales :

$$\mathbf{B} = \mu_0\mu_r\mathbf{H} + \mathbf{B}_r = \mu\mathbf{H} + \mathbf{B}_r \qquad \text{(III.7)}$$

Avec

μ_0 est la perméabilité du vide, $\mu_0 = 4\pi 10^{-7}$ H/m

μ_r est la perméabilité relative du matériau, $\mu = \mu_0\mu_r$ est la perméabilité du matériau H/m

$\mathbf{B}_r$ est le vecteur d'induction rémanente de l'aimant permanent en Tesla, ou T.

Le domaine est borné par la frontière Γ_Ω composée de deux parties différentes : $\Gamma_\Omega = \Gamma_B \cup \Gamma_H$, correspondant aux conditions de bord :

$$\mathbf{H} \times \mathbf{n} = \mathbf{0} \quad \text{sur } \Gamma_H \qquad \text{(III.8)}$$

$$\mathbf{B}.\mathbf{n} = 0 \quad \text{sur } \Gamma_B \qquad \text{(III.9)}$$

Avec **n** est la normale extérieure au domaine de la frontière Γ_Ω.

III.1.1.3 Introduction du potentiel scalaire

A partir l'équation (III.5), la formulation en potentiel scalaire magnétique consiste à décomposer le champ magnétique H en deux composantes :

$$\mathbf{H} = \mathbf{H}_s + \mathbf{H}_m \qquad \text{(III.10)}$$

La composante rotationnelle $\mathbf{H}_s$ représente l'effet des sources, ex. le courant parcouru des bobines :

$$\mathbf{rot}\,\mathbf{H}_s = \mathbf{J} \qquad \text{(III.11)}$$

La composante non rotationnelle $\mathbf{H}_m$ représente la réaction du matériau :

$$\mathbf{rot}\,\mathbf{H}_m = \mathbf{rot}\left(\mathbf{H} - \mathbf{H}_s\right) = \mathbf{0} \qquad \text{(III.12)}$$

Le champ magnétique $\mathbf{H}_m$ et l'identité $\mathbf{rot}\left(\mathbf{grad}\ U\right) = \mathbf{0}\ \forall U$ permet d'introduire le potentiel scalaire magnétique ψ tel que :

$$\mathbf{H}_m = -\mathbf{grad}\,\psi \qquad \text{(III.13)}$$

Donc, en introduisant $\mathbf{H} = \mathbf{H}_s - \mathbf{grad}\,\psi$ dans l'équation (III.7), on réécrit l'équation (III.6), $\mathrm{div}\,\mathbf{B} = 0$ pour le potentiel scalaire :

$$div(\mu\,\mathbf{grad}\,\psi) = div(\mu\,\mathbf{H}_s + \mathbf{B}_r) \qquad \text{(III.14)}$$

En cas d'absence d'aimants permanents, la formulation se simplifie :

$$div(\mu\,\mathbf{grad}\,\psi) = div(\mu\,\mathbf{H}_s) \qquad \text{(III.15)}$$

III.1.1.4 Introduction du potentiel vecteur

L'équation (III.6) et $\mathrm{div}\left(\mathbf{rot}\ \mathbf{u}\right) = 0\ \forall\ \mathbf{u}$ permet d'introduire le potentiel vecteur **A** comme :

$$\mathbf{B} = \mathbf{rot}\ \mathbf{A} \qquad \text{(III.16)}$$

On remplace la relation $\mathbf{H} = \frac{1}{\mu}(\mathbf{B} - \mathbf{B_r})$ et (III.16) dans (III.5) et obtient :

$$\mathbf{rot}\left(\nu\ \mathbf{rotA}\right) = \mathbf{J} + \mathbf{rot}(\nu\mathbf{B}_r) \qquad \text{(III.17)}$$

Avec v l'inverse de la perméabilité : $\nu = \frac{1}{\mu}$

En cas d'absence d'aimant permanent, la formulation est écrite comme :

$$\mathbf{rot}\left(\nu\ \mathbf{rotA}\right) = \mathbf{J} \qquad \text{(III.18)}$$

Et la condition au bord (III.8) (III.9) se réécrit par :

$$\nu\ \mathbf{rotA} \times \mathbf{n} = \mathbf{0} \quad \text{sur } \Gamma_H \qquad \text{(III.19)}$$

$$\mathbf{rotA}\ .\ \mathbf{n} = 0 \quad \text{sur } \Gamma_B \qquad \text{(III.20)}$$

En 2D, si on considère une invariance suivant z et le plan Oxy, $\mathbf{J}$ = J(x,y) $\mathbf{e_z}$ et $\mathbf{A}$ = A(x,y) $\mathbf{e_z}$, ce qui revient à dire que le courant source est perpendiculaire au plan Oxy et que le potentiel magnétique ne possède qu'une composante selon la direction z. Dans ce cas, le champ magnétique **H** et l'induction magnétique **B** sont des vecteurs dans le plan Oxy.

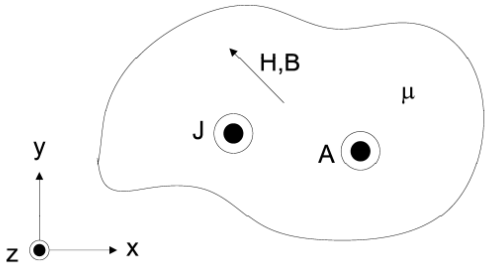


Figure III.2 : Analyse du problème magnétostatique en 2D

L'équation (III.17) s'écrit pour le problème en 2D comme une équation de Poisson :

$$\frac{\partial}{\partial x}\left(v\frac{\partial A}{\partial x}\right)+\frac{\partial}{\partial y}\left(v\frac{\partial A}{\partial y}\right)=-J \qquad \text{(III.21)}$$

III.1.1.5 Discrétisation

La discrétisation repose le triplet (Ω, E_f, Σ_Ω) avec

- Ω le domaine géométrique maillé à travers les éléments: triangle en 2D, tétraèdre en 3D.

- E_f l'espace fonctionnel sur lequel le modèle mathématique de la solution est défini.

- Σ_Ω l'ensemble de N degrés de liberté représentés par N fonctions d'interpolation linéaires w_i, i = 1...N ce qui sont définies dans l'espace fonctionnel.

La solution $u(\mathbf{r})$ est définie comme une fonction dans l'espace fonctionnel E_f qui est associé au domaine géométrie Ω. Elle est approchée par N fonctions d'interpolation $w_i(\mathbf{r})$ qui sont également définies sur le même espace fonctionnel E_f. La formulation est telle que :

$$u(\mathbf{r}) \approx \sum_{i=1}^{N} w_i(\mathbf{r})u_i \quad \text{avec} \quad w_i \in E_f, \mathbf{r} \in \Omega, u_i \in \Sigma_\Omega \qquad \text{(III.22)}$$

Le domaine Ω est maillé avec un ensemble d'éléments finis. Sur chaque élément, il est possible de construire des fonctions d'interpolation $w_i(\mathbf{r})$ selon l'entité géométrique associé : nœud, arête, facette, volume.

Tableau III-1 Fonction d'interpolation utilisée pour MEF

Type d'interpolation	Entité géométrique	Propriété de continuité
nodale	Nœud	Continuité de la fonction interpolée $u_{\|\Gamma_\Omega} = cte$
d'arête	Arête	Continuité de la composante tangentielle $\mathbf{u}\times\mathbf{n}_{\|\Gamma_\Omega} = \mathbf{0}$
surfacique	Facette	Continuité de la composante normale $\mathbf{u}.\mathbf{n}_{\|\Gamma_\Omega} = 0$
volumique	Volume	Discontinuité

Pour clarifier, nous notons N_n, N_a, N_f, N_v respectivement le nombre de nœuds, d'arêtes, de faces, de volume du maillage. Dans l'espace élémentaire, la fonction d'interpolation est identique à la fonction de forme.

Fonction de forme nodale

La fonction de forme est scalaire et associé à tous les nœuds de l'élément.

$$u(\mathbf{r}) \approx \sum_{n=1}^{N_n} w_n(\mathbf{r})u_n \qquad \text{(III.23)}$$

Avec u_n la valeur du degré de liberté et $w_n(\mathbf{r})$ la fonction de forme associée au nœud. La fonction $w_n(\mathbf{r})$ est telle que $w_n(\mathbf{r})=1$ sur le nœud associé et $w_n(\mathbf{r}) = 0$ sur les autres nœuds. La construction de la fonction $w_n(\mathbf{r})$ dépend de la fonction polynomiale utilisée pour approcher $u(\mathbf{r})$ dans l'espace de l'élément.

Fonction de forme d'arête

La fonction de forme d'arête est associée à une arête composée de deux nœuds n_1 et n_2, orientés de n_1 vers n_2. Elle est définie comme un vecteur construit grâce aux fonctions de formes nodales:

$$\mathbf{w}_a = w_{n1}\,\mathbf{grad}\,w_{n2} - w_{n2}\,\mathbf{grad}\,w_{n1} \tag{III.24}$$

La valeur du vecteur $\mathbf{w}_a$ égal à 1 sur l'arête associée et 0 sur les autres arêtes. L'interpolation est donc adaptée à une grandeur vectorielle dont la composante tangentielle est continue :

$$\mathbf{u}(\mathbf{r}) \approx \sum_{n=1}^{N_a} \mathbf{w}_a(\mathbf{r}) u_a \tag{III.25}$$

Fonction de forme de facette

La fonction de forme de facette pour un triangle composé de trois nœuds n1, n2, n3 est définie grâce aux fonctions de forme nodales :

$$\mathbf{w}_f = 2(w_{n1}\,\mathbf{grad}\,w_{n2} \times \mathbf{grad}\,w_{n3} + w_{n2}\,\mathbf{grad}\,w_{n3} \times \mathbf{grad}\,w_{n1} + w_{n3}\,\mathbf{grad}\,w_{n1} \times \mathbf{grad}\,w_{n2}) \tag{III.26}$$

La valeur du vecteur w_f égal à 1 sur la facette associée et 0 sur les autres facettes. L'interpolation d'une grandeur vectorielle est définie pour assurer la continuité de la composante normale à travers la facette :

$$\mathbf{u}(\mathbf{r}) \approx \sum_{n=1}^{N_f} \mathbf{w}_f(\mathbf{r}) u_f \tag{III.27}$$

Fonction de forme de volume

La fonction de forme de volume est définie comme une fonction scalaire:

$$w_v = \frac{1}{V_e} \tag{III.28}$$

avec V_e le volume de l'élément géométrique

La valeur de la fonction égale à 0 dans tous les éléments autres que l'élément e. L'interpolation est définie pour la grandeur scalaire et n'assure aucune continuité entre les éléments :

$$u(\mathbf{r}) \approx \sum_{n=1}^{N_v} w_v(\mathbf{r}) u_v \tag{III.29}$$

III.1.1.6 Formulation en potentiel scalaire

Forme discrète

La projection de Galerkin utilise un ensemble de fonctions tests W_i continues qui sont également N_n fonctions de formes nodales W_i avec N_n le nombre de nœuds du maillage.

$$\psi(\mathbf{r}) \approx \sum_{j=1}^{N_n} W_j(\mathbf{r})\psi_j \qquad \text{(III.30)}$$

Les fonctions de formes nodale W_i respectent la condition :

$W_i = 0$ sur Γ_H

La forme intégrale de (III.14) est donc écrite avec les fonctions W_i, i = 1…N_n, par :

$$\int_\Omega W_i div(\mu\, \mathbf{grad}\, \psi) d\Omega = \int_\Omega W_i div(\mu\, \mathbf{H}_s + \mathbf{B}_r) d\Omega \qquad \text{(III.31)}$$

L'introduction du théorème de Green donne :

$$\int_\Omega \mathbf{grad} W_i\ \mu\, \mathbf{grad}\psi d\Omega + \int_\Gamma W_i(\mu\, \mathbf{grad}\psi).\mathbf{n}\, d\Gamma = \int_\Omega \mathbf{grad} W_i(\mu\, \mathbf{H}_s + \mathbf{B}_r) d\Omega + \int_\Gamma W_i(\mu\, \mathbf{H}_s + \mathbf{B}_r).\mathbf{n}\, d\Gamma \qquad \text{(III.32)}$$

Les deux intégrales surfaciques sur $\Gamma_\Omega = \Gamma_B \cup \Gamma_H$ sont supprimées en vérifiant faiblement les conditions aux limites. Alors, la formulation devient :

$$\int_\Omega \mathbf{grad}\, W_i\ \mu\, \mathbf{grad}\, \psi d\Omega = \int_\Omega \mathbf{grad}\, W_i(\mu\, \mathbf{H}_s + \mathbf{B}_r) d\Omega \qquad \text{(III.33)}$$

Forme matricielle

$$\sum_{j=1}^{N_n} \left[\int_\Omega \mathbf{grad} W_i\ \mu\, \mathbf{grad} W_j\ d\Omega \right] \psi_j = \int_\Omega \mathbf{grad}\, W_i(\mu\, \mathbf{H}_s + \mathbf{B}_r) d\Omega \text{ avec i=1…N}_n \qquad \text{(III.34)}$$

III.1.1.7 Formulation en potentiel vecteur

Forme discrète

La projection de Galerkin consiste à projeter (III.17) sur un ensemble de fonctions tests $\mathbf{W}_i$ qui sont également des fonctions de forme des éléments satisfaisant la condition :

$$\mathbf{n} \times \mathbf{W}_i = \mathbf{0} \text{ sur la frontière } \Gamma_B \text{ où } \mathbf{B}.\mathbf{n} = 0 \qquad \text{(III.35)}$$

Cela permet d'écrire (III.17) comme :

$$\int_\Omega \mathbf{W_i rot}\left(\nu\ \mathbf{rotA}\right) d\Omega = \int_\Omega \mathbf{W_i J} d\Omega \qquad \text{(III.36)}$$

avec i de 1 à N, N le nombre de degrés de liberté

Après l'application du théorème de Green, la forme discrète de la formulation devient :

$$\int_\Omega \mathbf{rotW_i}\ \nu\ \mathbf{rotA}\ d\Omega - \int_\Gamma \mathbf{W}_i(\mathbf{n} \times \nu\ \mathbf{rot}\ \mathbf{A}) d\Gamma = \int_\Omega \mathbf{W_i J} d\Omega \qquad \text{(III.37)}$$

L'intégration surfacique sur $\Gamma = \Gamma_B \cup \Gamma_H$ s'annule à cause de (16) sur Γ_B et (13) sur Γ_H. La formulation est réécrite par :

$$\int_\Omega \mathbf{rotW_i}\ \nu\ \mathbf{rotA}\ d\Omega = \int_\Omega \mathbf{W_i J} d\Omega \qquad \text{(III.38)}$$

En 2D, l'opérateur **rot** se simplifie pour un vecteur à une seule composante suivant z et (III.36) équivaut à :

$$\int_\Omega \mathbf{grad}\, W_i\ \nu\ \mathbf{grad} A\ d\Omega = \int_\Omega W_i J d\Omega \qquad \text{(III.39)}$$

Forme matricielle

En 3D, la discrétisation du potentiel vecteur s'effectue sur des arêtes pour assurer la condition (III.35) :

$$\mathbf{A}(\mathbf{r}) = \sum_{j=1}^{N_a} \mathbf{W}_a(\mathbf{r})\mathrm{A_a} \qquad \text{(III.40)}$$

La formulation (III.36) s'écrit sous forme matricielle ci-dessous :

$$\sum_{j=1}^{N_a}\left[\int_{\Omega} \mathbf{rotW_i}\ \nu\ \mathbf{rotW_j}\ d\Omega\right]A_j = \int_{\Omega} \mathbf{W_i J} d\Omega \ \text{ avec i=1…N}_\text{a} \qquad \text{(III.41)}$$

En 2D, le potentiel vecteur devient le potentiel scalaire A_z car il n'existe que la composante selon la direction z. Dans ce cas, A_z est discrétisé sur des fonctions de forme nodales :

$$\mathbf{A}(\mathbf{r}) = \sum_{j=1}^{N_n} \mathrm{W_n(r)A_n} \qquad \text{(III.42)}$$

La forme matricielle pour la formulation (20) s'écrit alors:

$$\sum_{j=1}^{N_n}\left[\int_{\Omega} \mathbf{grad}\, W_i\ \nu\ \mathbf{grad}\, W_j\ d\Omega\right]A_j = \int_{\Omega} W_i J d\Omega \ \text{ avec i=1…N}_\text{n} \qquad \text{(III.43)}$$

La technique numérique de calcul des formulations matricielles ci-dessus n'est pas détaillée d'ici mais peut se trouver dans les ouvrages généraux classiques [66].

III.1.2. Assemblage de la forme discrétisée

III.1.2.1 Principe d'assemblage par extension des matrices élémentaires

La méthode des éléments finis consiste à calculer exactement les formes intégrales définies ci-dessus par élément, dans l'espace géométrique de l'élément et non dans l'espace global du problème. Les résultats de l'intégration élémentaire sont assemblés dans un système d'équations algébriques qui est représenté par la forme matricielle :

$$[A]\{x\} = \{b\} \qquad \text{(III.44)}$$

Où

[A] est la matrice des coefficients de taille N x N

{x} est le vecteur des variables inconnues, de taille N

{b} est le vecteur de second membre, de taille N

Nous supposons que l'ensemble des N degrés de liberté est numéroté globalement tel que :

$< x > = < x_1 ... x_i ... x_j ... x_N >$

A celui-ci correspond la matrice A et b suivants :

$$\begin{bmatrix} a_{11} & \dots & a_{1i} & \dots & a_{1j} & \dots & a_{1k} & \dots & a_{nn} \\ \vdots & \ddots & \vdots & & \vdots & & \vdots & & \vdots \\ a_{i1} & \dots & a_{ii} & \dots & a_{ij} & \dots & a_{ik} & \dots & a_{in} \\ \vdots & & \vdots & \ddots & \vdots & & \vdots & & \vdots \\ a_{j1} & \dots & a_{ji} & \dots & a_{jj} & \dots & a_{jk} & \dots & a_{jn} \\ \vdots & & \vdots & & \vdots & \ddots & \vdots & & \vdots \\ a_{k1} & \dots & a_{ki} & \dots & a_{kj} & \dots & a_{kk} & \dots & a_{kn} \\ \vdots & & \vdots & & \vdots & & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{n1} & \dots & a_{nj} & \dots & a_{nk} & \dots & a_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_j \\ \vdots \\ x_k \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_j \\ \vdots \\ b_k \\ \vdots \\ b_1 \end{Bmatrix}$$

$$\downarrow \qquad\qquad \downarrow \quad \downarrow$$

$$\mathrm{A}[n \times n] \qquad x[n \times 1] \quad \mathrm{b}[n \times 1]$$

Si chaque élément contient un nombre de degrés de libertés associé, noté n_d, l'intégration par élément produit une matrice élémentaire A^e de taille n_d x n_d et un vecteur second membre de taille n_d. Supposons que l'ensemble des degrés de liberté dans un élément soit numéroté par :

$$< \mathrm{x} >^e = < \mathrm{x_p} \dots \mathrm{x_q} >$$

A celui-ci correspond la matrice élémentaire A^e et b^e suivants :

$$\begin{bmatrix} a_{pp} & \dots & a_{pq} \\ \vdots & \ddots & \vdots \\ a_{qp} & \dots & a_{qq} \end{bmatrix} \begin{matrix} \rightarrow x_p \leftarrow \\ \vdots \\ \rightarrow x_q \leftarrow \end{matrix} \begin{Bmatrix} b_p \\ \vdots \\ b_q \end{Bmatrix}$$

$$\downarrow \qquad\qquad \downarrow \quad \downarrow$$

$$\mathrm{A^e}[n_d \times n_d] \quad x^e[n_d \times 1] \quad \mathrm{b^e}[n_d \times 1]$$

La procédure d'assemblage consiste à construire la matrice globale [A] et le vecteur second membre {b} par extension des matrices élémentaire [A^e] et des vecteurs élémentaires {b^e} :

$$\begin{aligned} [A] &= \sum_e^{N_e} [L^e]^T [A^e] \; L^e \\ \{b\} &= \sum_e^{N_e} [L^e]^T \{b^e\} \end{aligned} \qquad \text{(III.45)}$$

avec [L^e] la matrice d'extension élémentaire de taille n_dxN, qui fait l'extension et la permutation en fonction de la numérotation globale des degrés de liberté (DOF) élémentaires. Le rôle de la matrice L équivaut un mappage M des DOF en fonction la numérotation locale-globale tel que :

M(e, p) = i si $x_p = x_i$

M(e, q) = j si $x_q = x_j$

Donc, la procédure d'assemblage est symbolisée par l'expression ci-dessous :

$$A_{(i,j)} = \sum_{\substack{M(e,p)=i \\ M(e,q)=j}}^{N_e} A^e{}_{(p,q)}$$
$$b_{(i)} = \sum_{M(e,p)=i}^{N_e} b^e{}_{(p)} \qquad \text{(III.46)}$$

III.1.2.2 Introduction de la condition aux limites de Dirichlet

La condition aux limites de type Dirichlet impose une valeur pour certains degrés de liberté connus. C'est pourquoi, il est nécessaire d'éliminer ces degrés de liberté du système d'équations. Il existe de 3 approches différentes [66] :

- la méthode du terme diagonal dominant : Pour chaque condition x_i = Cte, elle change deux termes : l'élément diagonal A_{ii} de la matrice [A] par (A_{ii} + α) et b_i du vecteur {b} par (α.Cte), avec α est un nombre très grand par rapport tous les termes A_{ij}. Cette méthode ne permet d'obtenir qu'une solution approchée et risque de dégrader le conditionnement de la matrice.

- la méthode du terme diagonal unité : Pour la condition x_i = Cte, la matrice [A] et le vecteur {b} sont modifiés ainsi :

$$\begin{aligned} &\overline{A}_{ii} = 1 \\ &\overline{A}_{ij} = \overline{A}_{ji} = 0 \quad \forall j \neq i \\ &\overline{b}_i = Cte \\ &\overline{b}_j = b_j - A_{ji} Cte \quad \forall j \neq i \end{aligned} \qquad \text{(III.47)}$$

- la méthode de suppression des équations: Elle est similaire à la méthode du terme unité sur la diagonale sauf qu'elle supprime l'équation concernant le degré de liberté imposé. La condition x_i = Cte correspond la suppression la colonne « i » et la ligne « i » de la matrice [A], l'élément « i » du vecteur {b}. Le vecteur {b} est modifié comme dans (III.48). Cette méthode a l'avantage d'éliminer les degrés de liberté connus avant l'assemblage et de réduire la taille de la matrice A et du vecteur b.

$$\overline{b}_j = b_j - A_{ji} Cte \quad \forall j \neq i \qquad \text{(III.48)}$$

En développant cela au niveau de l'élément, la formulation devient:

$$\overline{b}_j{}^e = b_j{}^e - A^e_{ji} Cte \quad \forall j \neq i \qquad \text{(III.49)}$$

III.1.2.3 Propriété de la matrice globale

La formulation (III.45) forme la matrice globale à partir des matrices élémentaires. L'opération de transformation de [A^e] à [A], par L^e comporte un grand nombre de zéro. En conséquence, la matrice globale est creuse et de structure de bande diagonale, cf. Figure III.3.

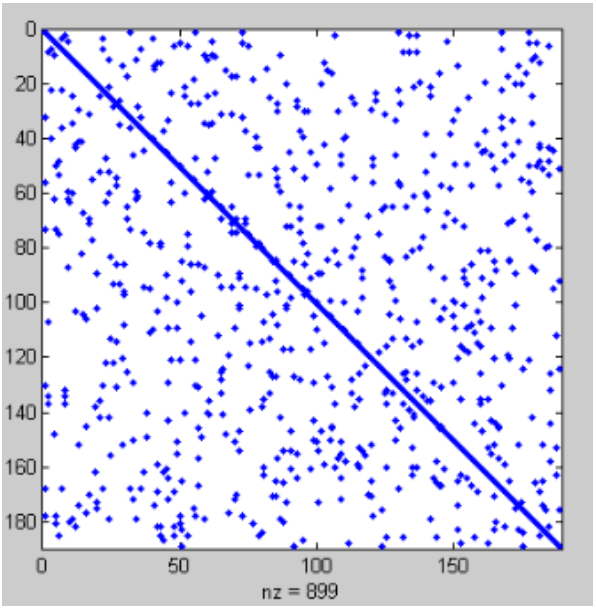


Figure III.3 Topologie d'une matrice de rigidité MEF pour un problème de magnétostatique en 2D avec une numérotation des degrés de liberté arbitraire

Les problèmes magnétostatiques produisent un système d'équations symétrique défini positif. Les formulations en potentiel (III.34) (III.41) (III.43) possèdent cette propriété pour la matrice globale.

III.1.3. Résolution

Le système d'équations obtenu par la MEF se représente sous forme matricielle Ax=b, avec A la matrice carrée de NxN, **symétrique définie positive**. Les solveurs sont généralement classés en deux catégories différentes : directs et itératifs.

Les solveurs **directs**, ex. « Gaussian elimination » ou factorisation LU … permettent d'obtenir la solution exacte après un nombre connu d'opérations qui dépend de l'algorithme. Les solveurs directs sont robustes et plus intéressants que les solveurs itératifs pour **les systèmes de tailles petites ou moyennes** et ils fonctionnent aussi bien pour les matrices denses que creuses. Cependant, si la taille du système augmente, ils deviennent beaucoup moins intéressants à cause du phénomène de **remplissage** (« fill-in » en anglais). Le remplissage vient de la phase de factorisation qui produit des termes non nuls à partir des valeurs initiales de termes matriciels nuls ou non-nuls. En conséquence, la structure de la matrice doit être mise à jour constamment pendant la factorisation et le **coût de stockage** devient important.

Les solveurs **itératifs**, au contraire des solveurs directs, peuvent ne pas modifier la topologie de la matrice pendant le calcul. L'opération essentielle du calcul itératif est la **multiplication matrice-vecteur**. Comme le terme « itératif » l'indique, la résolution est une procédure itérative dans laquelle la solution est approchée successivement à chaque étape. En commençant avec une approximation donnée (normalement à zéro), la solution est améliorée itérativement jusqu'à ce qu'on considère que la solution exacte est obtenue. Cependant, un solveur itératif ne garantit pas une solution quel que soit le système. Il est extrêmement **sensible à la structure de la matrice**, et surtout au conditionnement de la matrice. Du point de vue de la performance, si la solution peut être obtenue, les solveurs itératifs sont capables de la donner plus rapidement que les

solveurs directs. C'est pourquoi les solveurs itératifs sont bien adaptés aux systèmes d'équations de grande taille et creux tels que ceux issus de la MEF.

De plus, la convergence des solveurs itératifs dépend en grande partie de la technique de **préconditionnement**. Cette dernière consiste à chercher une approximation liée à l'inverse de la matrice afin de rendre le système original mieux conditionné.

Dans la section suivante, nous présentons respectivement quelques aspects qui sont particulièrement cruciaux dans cette thèse : le stockage de la matrice creuse, le solveur itératif surtout la famille de Krylov, et la technique de préconditionnement.

III.1.3.1 Stockage de la matrice creuse

Une matrice dense est normalement stockée sous forme d'un tableau à deux dimensions. Par exemple, la matrice A, de m lignes et n colonnes, sera stockée sous la forme A[mxn]. Pour les matrices creuses, il est possible de ne stocker que les termes non nuls pour réduire la mémoire nécessaire. Il existe plusieurs structures différentes pour stocker la matrice creuse. Nous abordons quelques formats utilisés dans cette thèse : COO, CSR, CSC, DIA, même si d'autres formats peuvent être trouvés [19]. En vue de la phase de manipulation des matrices, il est important de comprendre les avantages et les inconvénients de chacun des formats. D'une manière générale, les formats DIA et ELL sont les plus efficaces pour le calcul de produits matrice-vecteur, et donc ils sont les formats les plus rapides de résolution de systèmes linéaires creux avec des méthodes itératives. Les formats COO et CSR sont plus flexibles que DIA et ELL et plus faciles à manipuler.

Format de COO ou Liste des COOrdonnées

Un terme non nul est identifié par un triplet : l'indice de ligne, l'indice de colonne et la valeur. Donc, une matrice creuse est définie par trois tables : une table de nombres entiers COO_row qui contient les indices de ligne, une table de nombres entiers COO_col qui contient les indices de colonne, une table de valeurs réelles COO_val qui contient les valeurs non nulles. Les trois tables sont de taille NNZ, avec NNZ le nombre de termes non nuls. En général, les termes non nuls sont triés par leurs indices de ligne puis par leurs indices de colonne pour améliorer l'accès aléatoire.

$$A = \begin{bmatrix} 1{,}0 & 4{,}0 & 0{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & 2{,}0 & 3{,}0 & 0{,}0 & 0{,}0 \\ 5{,}0 & 0{,}0 & 0{,}0 & 7{,}0 & 8{,}0 \\ 0{,}0 & 0{,}0 & 9{,}0 & 0{,}0 & 6{,}0 \end{bmatrix}$$

$$\text{COO\_row} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$$

$$\text{COO\_col} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 2 & 4 \end{bmatrix}$$

$$\text{COO\_val} = \begin{bmatrix} 1{,}0 & 4{,}0 & 2{,}0 & 3{,}0 & 5{,}0 & 7{,}0 & 8{,}0 & 9{,}0 & 6{,}0 \end{bmatrix}$$

L'accès à un terme non nul de la matrice A, par exemple A(i,j) est décomposé en 3 opérations :

- rechercher la valeur « i » dans la table de COO_row, et si il est trouvé, noter l'intervalle de données correspondant à la ligne « i » (l'indice de début et de fin)

- rechercher la valeur « j » dans la table de COO_col sur l'intervalle trouvé, noter l'indice d'accès du terme, ex k := mappage (i,j)

- lire/modifier la valeur du terme COO_val[k].

Le format COO est le format de matrice creuse le plus général. Il est capable de décrire tous les types de matrices creuses quel que soient la distribution et le taux de termes non nuls.

Format CSR/CRS « compressed sparse row/compressed row storage »

Comme son nom l'indique, ce format se base sur le format COO dont les indices de lignes sont « compressés ». On peut en effet remarquer que la table d'indice de ligne COO_row contient plusieurs indices inutiles, les indices des termes non nuls sur une même ligne. Un pointeur qui donne l'indice de début d'une ligne est utilisé au lieu de cette table afin d'éliminer les indices inutiles. Donc, la matrice A[mxn] est décrite par trois tables : une table de m+1 nombres entiers CSR_rowPtr, une table de NNZ nombres entiers CSR_col et une table de NNZ nombres réels CSR_val.

$$A = \begin{bmatrix} 1{,}0 & 4{,}0 & 0{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & 2{,}0 & 3{,}0 & 0{,}0 & 0{,}0 \\ 5{,}0 & 0{,}0 & 0{,}0 & 7{,}0 & 8{,}0 \\ 0{,}0 & 0{,}0 & 9{,}0 & 0{,}0 & 6{,}0 \end{bmatrix}$$

$$\text{CSR\_rowptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\text{CSR\_col} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 2 & 4 \end{bmatrix}$$

$$\text{CSR\_val} = \begin{bmatrix} 1{,}0 & 4{,}0 & 2{,}0 & 3{,}0 & 5{,}0 & 7{,}0 & 8{,}0 & 9{,}0 & 6{,}0 \end{bmatrix}$$

En comparaison au format COO, le stockage et le coût d'accès du format CSR sont moins chers. L'accès d'un élément A(i,j) est effectué par 2 opérations :

- rechercher la valeur « j » dans la table CSR_col dans l'intervalle de CSR_rowPtr[i] et CSR_rowPtr[i+1], le résultat est l'indice d'accès du terme « k »

- lire/ modifier la valeur du terme non nul correspondant CSR_val[k]

Format CSC « compressed sparse column »

Ce format est similaire à CSR sauf que les termes non nuls sont triés par leurs indices de colonne en priorité puis par leurs indices de ligne et la table d'indices de colonne est compressée dans une table des pointeurs de début des colonnes. La matrice A dans exemple est en format COO trié par colonne et puis ligne :

$$A = \begin{bmatrix} 1,0 & 4,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 2,0 & 3,0 & 0,0 & 0,0 \\ 5,0 & 0,0 & 0,0 & 7,0 & 8,0 \\ 0,0 & 0,0 & 9,0 & 0,0 & 6,0 \end{bmatrix}$$

$$\text{COO\_col} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 4 & 4 \end{bmatrix}$$

$$\text{COO\_row} = \begin{bmatrix} 0 & 2 & 0 & 1 & 1 & 3 & 2 & 2 & 3 \end{bmatrix}$$

$$\text{COO\_val} = \begin{bmatrix} 1,0 & 5,0 & 4,0 & 2,0 & 3,0 & 9,0 & 7,0 & 8,0 & 6,0 \end{bmatrix}$$

Le format CSC est obtenu par la compression des indices de colonnes tel que :

$$\text{CSC\_colptr} = \begin{bmatrix} 0 & 2 & 4 & 6 & 7 & 9 \end{bmatrix}$$

$$\text{CSC\_row} = \begin{bmatrix} 0 & 2 & 0 & 1 & 1 & 3 & 2 & 2 & 4 \end{bmatrix}$$

$$\text{CSC\_val} = \begin{bmatrix} 1,0 & 5,0 & 4,0 & 2,0 & 3,0 & 9,0 & 7,0 & 8,0 & 6,0 \end{bmatrix}$$

Format DIA ou de Bande Diagonale

Pour une matrice creuse dont les termes non nuls sont peu nombreux et concentrés autour de la diagonale, le format bande diagonale (DIA) est approprié. Le format DIA est composé de :

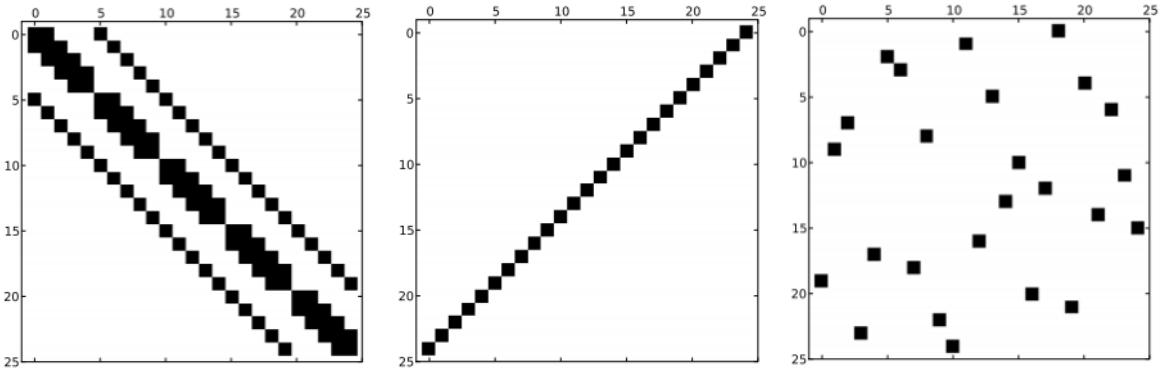
- une matrice dense [mxk] de valeurs réelles qui réserve des termes diagonaux (nuls ou nuls) avec « k » la largeur de la bande diagonale.

- une table de « k » nombres d'entiers, appelé « offset » qui indiquent l'offset de chaque diagonale par rapport la diagonale principale. L'offset 0 correspond à la diagonale principale, l'offset négatif, ex. -2 correspond à la sous diagonale à gauche de 2 éléments, l'offset positif correspond à la sous diagonale à droit. Le symbole * signifie une valeur inutilisée de la matrice, mise à une valeur aléatoire. « k » est appelé également la largeur de bande de la matrice.

$$A = \begin{bmatrix} 1,0 & 4,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 2,0 & 3,0 & 0,0 & 0,0 \\ 5,0 & 0,0 & 0,0 & 7,0 & 8,0 \\ 0,0 & 0,0 & 9,0 & 0,0 & 6,0 \end{bmatrix}$$

$$DIA\_valeur = \begin{bmatrix} * & * & 1,0 & 4,0 & 0,0 \\ * & 0,0 & 2,0 & 3,0 & 0,0 \\ 5,0 & 0,0 & 0,0 & 7,0 & 8,0 \\ 0,0 & 9,0 & 0,0 & 6,0 & * \end{bmatrix}$$

$$DIA\_offset = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$



Favorable au DIA | Défavorable au DIA | Défavorable au DIA

Figure III.4 Illustration d'usage du format DIA favorable à la matrice de bande diagonale

Le format DIA est approprié pour les problèmes où le maillage est structuré. Pour un maillage non structuré, la largeur de bande diagonale est moins maitrisable et le stockage de la matrice sous le format DIA est généralement non optimal car il incorpore de nombreux termes nuls. C'est pour cette raison que par défaut, nous choisissons le format COO et CSR/CSC pour le stockage de la matrice des coefficients.

III.1.3.2 Solveur Itératif

Dans cette section, nous abordons la résolution du système d'équations linéaires représenté par la forme matricielle :

$$Ax = b$$

Avec `A` la matrice non-singulière des coefficients de taille `nxn`, `x` et `f` respectivement le vecteur de solution et le second membre de taille `n`.

En général, le solveur itératif démarre avec une approximation initiale et effectue une séquence d'étapes successives afin de trouver la valeur la plus proche de la solution exacte. Il y a deux catégories de solveurs itératifs : les solveurs itératifs stationnaires, Jacobi et Gauss-Seidel (GS) et les solveurs itératifs non stationnaire, la famille de Krylov, y compris le Gradient Conjugué (CG), deux solveurs qui sont utilisés dans notre travail. La convergence du solveur itératif dépend fortement du spectre de la matrice des coefficients.

Le solveur Gradient Conjugué est à privilégier pour résoudre un système d'équations symétrique défini positif qui est généralement obtenu par la discrétisation MEF en magnétostatique. L'algorithme du solveur itératif CG est résumé par le pseudocode suivant :

Algorithme du solveur Gradient Conjugué (CG)

```
1. Initialiser la valeur initiale de solution x0, calculer le résidu r0 = b – Ax0, p0 = r0

2. Pour i = 1… jusqu'à la convergence, fait :

3.          alpha_i = ri_T * ri / (A*pi)_T * pi ; // valeur scalaire

4.          x_(i+1) = xi + alpha_i * pi ;

5.          r_(i+1) = ri – alpha* A* pi ;
```

```
6.        beta_i = r_(i+1)_T*r_(i+1) / ri_T * ri ; // valeur scalaire
7.        p_(i+1) = r_(i+1) + beta_i * pi
8. fin
```

III.1.3.3 Préconditionneur

Le spectre de la matrice des coefficients peut être amélioré considérablement par un préconditionnement. Il s'agit de trouver une matrice de préconditionnement M non singulière dont l'inverse M^{-1} est proche de A^{-1} et telle que le système des équations de Mx = b soit plus facile à résoudre que Ax = b. Le pseudocode du solveur Gradient Conjugué préconditionné est donné par l'algorithme ci-dessous :

Algorithme du solveur Gradient Conjugué (CG) préconditionné

```
1. Initialiser la valeur initiale de solution x0, calculer le résidu r0 = b – Ax0, z0 = M^-1 r0, p0 = z0
2. Pour i = 1… jusqu'à la convergence, fait :
3.        alpha_i = ri_T * zi / (A*pi)_T * pi ; // valeur scalaire
4.        x_(i+1) = xi + alpha_i * pi ;
5.        r_(i+1) = ri – alpha_i* A* pi ;
6.        z_(i+1) = M^-1 r_(i+1)
6.        beta_i = r_(i+1)_T*z_(i+1) / ri_T * ri ; // valeur scalaire
7.        p_(i+1) = z_(i+1) + beta_i * pi
8. fin
```

Dans cet algorithme, M dénote la matrice de préconditionnement, désormais appelé « préconditionneur ». Il existe plusieurs préconditionneurs différents pour CG. Nous n'abordons que des préconditionneurs utilisés dans la thèse.

Préconditionnement de Jacobi

La matrice des coefficients A se décompose en 3 matrices : une matrice D qui contient les termes diagonaux ; une matrice L triangulaire inférieure et une matrice U triangulaire supérieure :

$$A = L + D + U$$

Le préconditionneur de Jacobi consiste à utiliser la matrice diagonale D:

$$M = D$$

Parmi les préconditionneurs utilisés, le préconditionneur de Jacobi est le plus simple à créer et mettre en œuvre. Cependant, sa performance est limitée.

Approximation de l'inverse de la matrice des coefficients AINV

L'objectif est de définir une matrice M telle que $M^{-1} \approx A^{-1}$. La méthode issue de [67] repose sur la factorisation triangulaire dite « outer produit » formulation qui consiste à chercher deux

matrices triangulaire W et Z telles que : $A^{-1} \approx ZDW^T$. La procédure de construction de W et Z ressemble à une factorisation complète de Cholesky pour la matrice A.

Factorisation Incomplète de Cholesky ILU

Pour une matrice symétrique définie positive, la matrice de préconditionneur est obtenue par la factorisation incomplète LU de Cholesky : $A \approx M = \widetilde{L}\,\widetilde{L}^T$ avec L une matrice triangulaire inférieure. Le système d'équations Ax = b est remplacé par Mx = b ou encore $\widetilde{L}\,\widetilde{L}^T x = b$. La résolution se décompose en 2 étapes de résolutions consécutives en utilisant la matrice triangulaire:

$$u = \widetilde{L}^{-1} b \text{ et } x = \widetilde{L}^{-T} b$$

Le remplissage (« fill-in » en anglais) d'une matrice creuse représente le passage d'une valeur nulle à une valeur non nulle pendant l'exécution d'un algorithme. Pour réduire les besoins supplémentaires en mémoire et en coût de calcul, il est nécessaire de limiter ce phénomène.

La factorisation incomplète ILU(0) qui limite le phénomène de remplissage est présentée dans le pseudocode suivant :

```
0. Donner la matrice A avec S = {(i,j) : A(i,j) ≠ 0 }
1. Pour k = 1 … n, fait
2.       Pour tout (i,j) ∈ S, fait
3.             A(i,j) += -A(i,k) * A(k,j) / A(k,k)
4.       fin
5. fin
```

Ainsi le résultat de la factorisation est stocké directement dans la matrice A. La matrice triangulaire inférieur $\widetilde{L}$ est obtenue par extraction depuis la matrice A factorisée.

Multigrille Algébrique AMG

Le préconditionneur AMG se base sur le principe du solveur de multigrille. Il existe de nombreuses variantes d'algorithme multigrille, mais les caractéristiques communes sont qu'une hiérarchie de discrétisations (grille ou maillage) est considérée. Les étapes importantes sont:

- **Lissage** en réduisant les erreurs à haute fréquence, qui comporte généralement quelques itérations d'un solveur itératif simple comme Gauss-Seidel. L'objectif est d'obtenir une meilleure approximation de la solution qui sera utilisée dans les étapes suivantes.
- **Restriction** en réduisant l'erreur résiduelle, qui consiste à transformer le système correspondant à une grille fine vers un système réduit correspondant à une grille grossière. La taille du système obtenu est plus petite (matrice et vecteur). L'opération de restriction peut être exécutée récursivement en réduisant au fur et à mesure la taille du système. Une phase de résolution sera effectuée sur le système de taille la plus petite par un solveur direct.
- **Interpolation** ou **prolongation** en interpolant une correction calculée d'une grille plus grossière vers une grille plus fine. Chaque opération d'interpolation correspond exactement à une opération de restriction mais en sens inverse.

La méthode de préconditionnement AMG construit un jeu de système à résoudre à partir du système original. Les opérations de réduction ou d'extension sont effectuées directement à partir de la matrice du système. En général, le préconditionnement AMG est coûteux mais robuste et permet de résoudre des systèmes linéaires difficiles.

III.2. Parallélisation de l'intégration et d'assemblage sur GPU

III.2.1. Difficulté du calcul d'intégration et d'assemblage parallèle

L'opération d'assemblage se base sur la formulation (III.46) dont l'algorithme comme ci-dessous :

Algorithme d'assemblage séquentiel

```
   0. Définir le problème MEF, numérotation globale des DOFs {x}
   1. Initialiser la matrice [A] de taille NxN dans un format de stockage de matrice
creuse, ex. COO, et vecteur {b} globale de taille N
   2. Pour chaque élément, numérotation locale des DOFs {x^e}, récupérer le mappage
DOF local-global M(e, n_i), initialiser la matrice élémentaire [A^e] de taille N_d x N_d
et le vecteur {b^e} de taille N_d
   2.a Intégration numérique
   Pour chaque point d'intégration de Gauss

       // Intégration par point de Gauss
       Pour chaque p de 1 à N_d
       {
              Pour chaque q de 1 à N_d , évaluer A^e(p,q)
              Évaluer b^e(p)
       }

       // Condition de Dirichlet
       Pour chaque p de 1 à N_d
       {
              Si p connu = v^p (Dirichlet valeur),
                     Pour tous q de 1 à N_d, et q #p : b^e(q) = b^e(q) - A^e(p,q)*v^p
       }
   2.b Assemblage global
       Pour chaque p de 1 à N_d
       {
              Si p connu (Dirichlet) passer p+1,
              Sinon, assemble b(i) += b^e(p) avec M(e,p) = i ;
              Pour chaque q de 1 à N_d
                     Si q connu (Dirichlet) passer q+1
                     Sinon, assemble A(i,j) += A(p,q) avec M(e,q) = j
       }
```

Chaque élément doit accéder aux données du maillage, récupérer ses données propres, calculer ses sous-matrice et sous-vecteur et assembler le résultat dans la matrice et le vecteur global. Le calcul numérique et l'assemblage s'effectue en itérant sur les tous éléments dans un scénario séquentiel. Un scénario d'exécution parallèle par élément du calcul se présente tel que décrit schématiquement dans la Figure III.5. Clairement, ce changement va se heurter à deux challenges majeurs : la concurrence entre les threads parallèles et l'accès aux données.

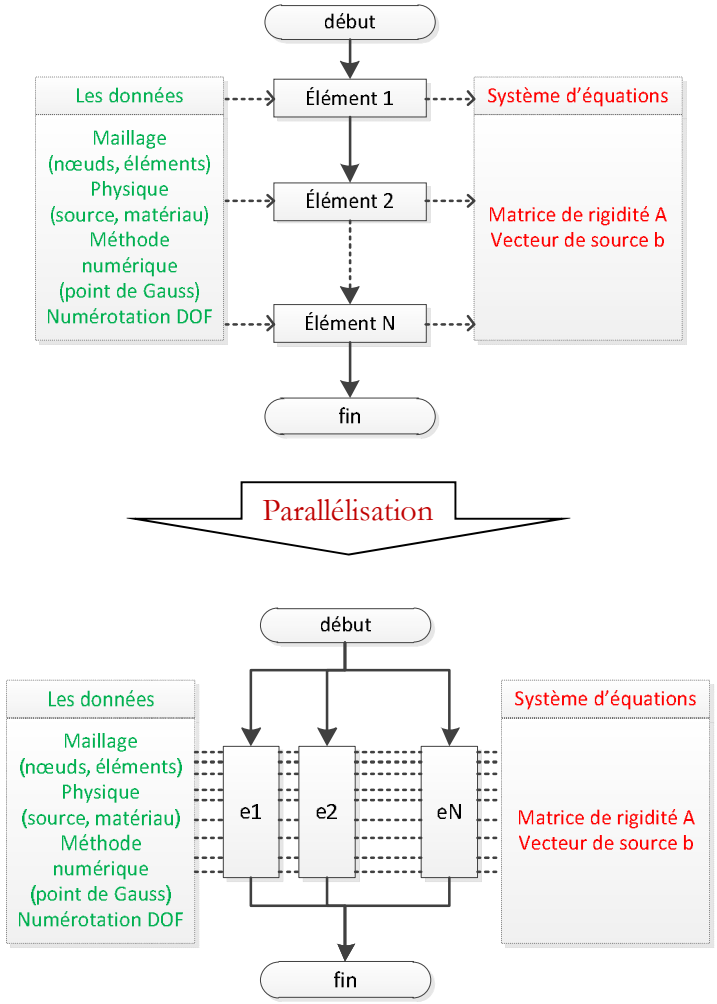


Figure III.5 Passage de calcul d'intégration et d'assemble séquentiel au calcul parallèle sur GPU

Problème de la condition de concurrence

La condition de concurrence décrit une situation où plusieurs threads essaient de modifier une valeur en même temps. C'est le cas de la procédure d'assemblage en parallèle. A titre d'illustration, l'opération de mise à jour du vecteur {b} est faite en concurrence par tous les threads qui sont liés à l'élément comportant le DOF i.

```
b(i) += b^e(p) avec M(e,p) = i
```

D'une manière similaire, l'opération de modification de la valeur A(i,j) est éventuellement effectuée en concurrence par tous les threads qui sont liés aux éléments comportant les deux DOFs i et j.

```
A(i,j) += A(p,q) avec M(e,p) = i et M(e,q) = j
```

En conséquence, les valeurs finales de b(i) et A(i,j) sont indéterminées car l'ordre de l'opération de mise à jour est non prévisible. La solution à ce problème passe globalement par une construction de threads manipulant des listes d'éléments indépendants du point de vue de l'assemblage. Dans la section suivante, on présentera les techniques classiquement utilisées, assemblage parallèle par graphe de coloriage et par regroupement en « patch » et la méthode que nous avons développé basée sur un tri et une réduction globale.

Problème de l'accès à la mémoire GPU

Second problème identifié, l'accès aux données. Pour optimiser l'exécution du kernel, l'accès aux données (en lecture ou en écriture) demande un mode adapté au type de mémoire où les données résident.

Considérons par exemple un problème de MEF nodale, les données du maillage minimales nécessaires sont :

- une table des coordonnées des nœuds (i.e. x, y, z).

- une table qui contient l'indice des nœuds de l'élément.

Le maillage utilisé pour la MEF est généralement non structuré et les données des nœuds de chaque élément sont donc dispersées. Cette approche n'est pas favorable à un passage en mémoire globale pour les kernels CUDA, qui demande une organisation coalescente des données. De plus, si un nœud est partagé par plusieurs éléments, les données du nœud doit être disponible pour tous les threads correspondant aux éléments qui possèdent ce nœud. Cette situation engendre une grande inefficacité de lecture. La solution pour améliorer l'accès à la mémoire globale est de regrouper les données précédemment séparées des nœuds et éléments dans une base de données commune, contiguë et puis, de les transférer vers la mémoire partagée du GPU qui réside on-chip et possède un temps d'accès plus rapide que la mémoire globale.

III.2.2. Etat de l'art

Plusieurs approches ont été développées dans la littérature. Chaque technique présente ses avantages et également ses limitations.

Cecka et al. ont analysé la procédure d'assemblage parallèle en différentiant deux opérations : l'assemblage des termes et le stockage de la matrice élémentaire [15]. Considérons le terme à assembler, 3 approches peuvent être identifiées :

- Assemblage par **terme non nul** : chaque terme non nul A(i,j) est assemblé par un thread parallèle. Même si ce mode facilite l'écriture en agissant directement sur chaque terme de la matrice A, il cause des situations de déséquilibre entre les threads.
- Assemblage par **DOF** : tous les termes non nuls correspondant à un DOF seront assemblés par un thread parallèle. Ce mode réduit considérablement le déséquilibre entre les threads mais l'assemblage demande une procédure de recherche d'indice dont le coût est très élevé.
- Assemblage par **élément** : tous les termes non nuls d'un élément vont être assemblés par un thread parallèle. Le thread utilisé peut exécuter également le calcul d'intégration sur l'élément correspondant afin de profiter du transfert de données. Il assure l'équilibre entre les threads car le calcul de chaque thread est identique. Cependant, plusieurs threads qui partagent un ou plusieurs DOF se heurtent à une condition de concurrence. La solution classique est le coloriage des éléments pour répartir les DOFs dans des groupes disjoints. La procédure parallèle traitera ensuite chaque groupe les uns après les autres. Cependant, la performance de cette technique est limitée par le coût de l'accès non optimal aux données.

Au point de vue du stockage des matrices et vecteurs (qui sont le résultat du calcul d'intégration numérique) il y a également 3 options :

- Stockage en **mémoire globale** du GPU. Puisque cette dernière est accessible pour tous les threads parallèles, ce mode de stockage permet de séparer le calcul d'intégration et la procédure d'assemblage en deux kernels distincts. En conséquence, l'équilibre entre les threads est maximal, tandis que la dépendance entre les threads est réduite au minimum. Cependant, le stockage des intégrales élémentaires nécessite momentanément une grande capacité de mémoire globale ce qui est un point de blocage sans solution. De plus, la grande latence de la mémoire globale ralentit considérablement la vitesse d'assemblage.
- Stockage en **mémoire partagée** du SM. La rapidité de la mémoire partagée est un avantage de ce mode. En pratique, la plupart des techniques utilise la mémoire partagée pour stocker le résultat élémentaire et réduire le nombre de registres utilisés. Cependant, la capacité limitée de la mémoire partagée demande de limiter le nombre d'éléments traités par SM. Il s'agit d'une procédure de décomposition des éléments.
- Stockage en **mémoire de registres** : Le nombre des registres utilisables par un kernel est très limité, donc ce mode n'est que possible que pour des éléments dont la taille de matrice et vecteur élémentaires est faible.

Quatre techniques ont été proposées en combinant les options ci-dessus. L'ordre de l'approximation utilisée est souvent déterminant.

- Pour des ordres d'approximation **faibles**, la technique la plus efficace est d'utiliser la mémoire partagée.
- Pour les ordres **élevés**, l'assemblage en mémoire globale donne la meilleure performance.

La technique qui utilise **la mémoire globale** peut se trouver également dans l'article de *Dziekonski et al.* [68] [69] et [8]. Pour les problèmes de grande taille, la simulation sur une plateforme de plusieurs GPU a été proposée [70]. La procédure générale est divisée par 3 étapes :

- **l'intégration** numérique qui calcule des termes intégraux élémentaires,
- **l'assemblage** de la matrice des coefficients sous le format de **COO**,
- **la réduction** de tous les termes ayant les mêmes indices dans le format de **CSR**.

Les trois étapes sont effectuées en mémoire globale. L'intégration numérique se base la quadrature de Gauss allant jusqu'au $10^{ème}$ ordre avec 81 points Gauss sur des éléments tétraédriques curvilignes. L'expérience de l'assemblage parallèle sur un NVIDIA Tesla C2050 (448 cœurs CUDA) donne une accélération de 80x par rapport à la version sur CPU.

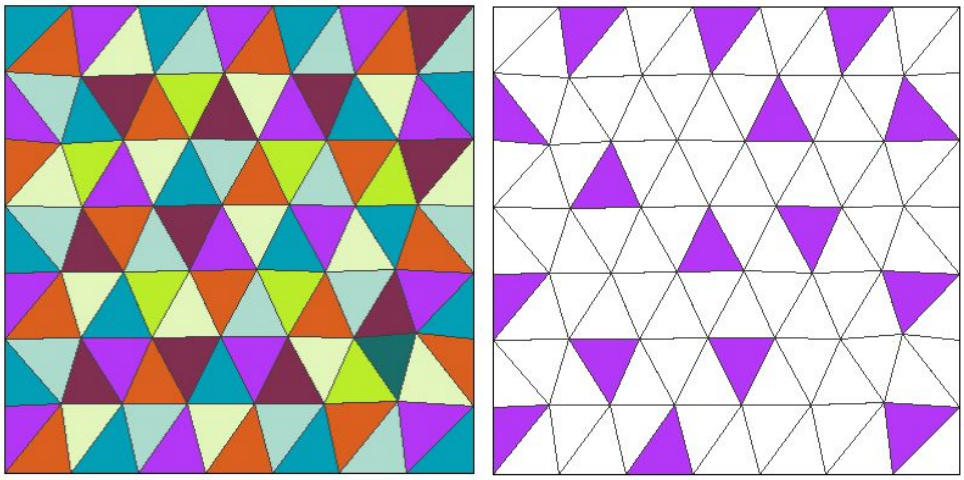
Komatitsch et al. a présenté un technique d'assemblage parallèle par **coloriage des éléments** pour des ordres élevés (4^{eme} ordre) dans le cadre de la simulation de tremblements de terre [4]. Dans cet article, chaque élément contient 125 nœuds et chaque thread est attaché à l'assemblage d'un nœud. L'expérience sur NVIDIA GeForce GTX 280 (240 cœurs CUDA) montre une accélération considérable de 25x du code parallèle par rapport sa version séquentielle.

Fu et al. a présenté un technique qui se base sur la décomposition du maillage par pièce (« patch » en anglais) [5]. Chaque pièce se compose d'éléments qui se touchent en nombre limité dont les données peuvent être stockées en mémoire partagée du GPU. La procédure d'assemblage s'effectue ensuite en mémoire partagée. De plus, les informations sur la géométrie issues de la phase d'assemblage sont également utilisées dans la résolution en créant un préconditionneur spécifique de type multigrille. Les expériences sur un NVIDIA GeForce GTX 580 (512 cœurs CUDA) montrent une accélération de 87x pour la phase d'assemblage et 51x pour la phase de résolution par rapport son exécution sur CPU.

Dans la section suivante, nous présenterons le principe et le code détaillé de 3 implantations différentes : le coloriage des éléments, la décomposition par pièce et notre nouvelle approche qui se passe en mémoire globale. Un bilan d'évaluation de performance de ces trois approches se trouvera dans la section III.4.

III.2.3. Approche par coloriage

Cette approche consiste à diviser le maillage en groupes caractérisés par une couleur dont les éléments sont séparés au sens de la géométrie. Le coloriage d'un maillage triangulaire 2D est illustré par la Figure III.6. Le passage en 3D suit une procédure similaire. Dans le maillage coloré, chaque élément possède une couleur différente de tous ses voisins. De cette manière, les éléments de même couleur ne partagent aucune entité géométrique (nœud ou arête). Ils appartiennent à un groupe de couleur.



Maillage coloré Un groupe des éléments de même couleur

Figure III.6 Illustration du processus de coloriage des éléments du maillage

III.2.3.1 Coloriage

La séparation des éléments d'un maillage se base sur un algorithme de coloriage d'un graphe [4] dont le nœud du graphe correspond à un élément et la branche du graphe lie deux éléments partageant une arête. Nous utilisons un algorithme simple dont le pseudocode est le suivant :

Algorithme de coloriage des éléments

```
1. initialiser tous les éléments par une même couleur (souvent par un nombre
     entier, par ex. 0)
2. pour chaque élément :
```

```
2a. chercher la couleur minimale différente par rapport celui de ses éléments
     voisins
2b. imposer cette couleur pour l'élément et passer l'élément suivant
```

Avec cet algorithme simple, le nombre d'éléments par couleur n'est équilibré et, en pratique, les premières couleurs contiennent le plus grand nombre d'éléments. L'équilibrage du nombre d'éléments par couleur et la recherche du nombre minimal des couleurs sont inutilisables car leur coût est très élevé. La Figure III.7 ci-dessous montre une relation non-linéaire du temps de coloriage en fonction du nombre des éléments.

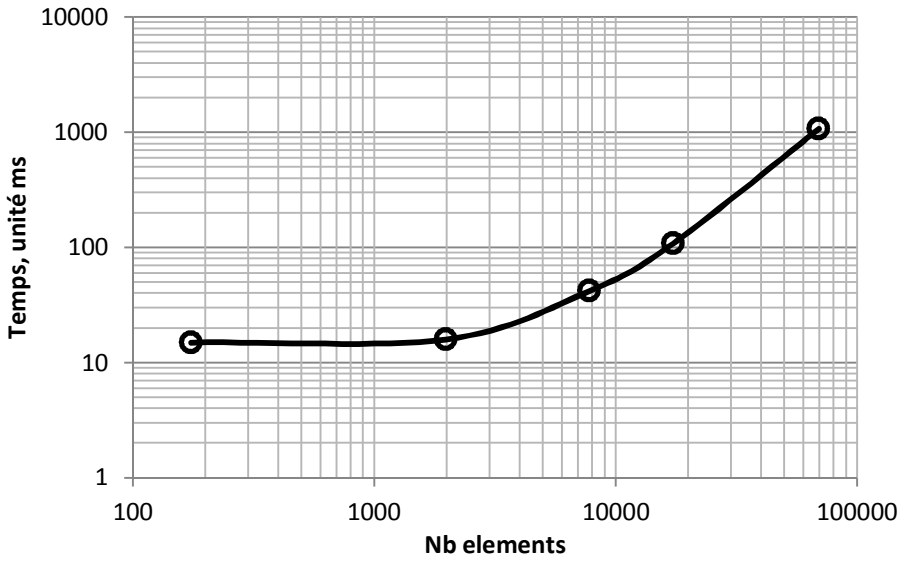


Figure III.7 Evolution du temps de coloriage en fonction du nombre d'éléments

III.2.3.2 Algorithme

Du point de vue de la discrétisation par la MEF, chaque entité géométrique (nœud, arête, ..) est associée à un degré de liberté. L'approche par coloriage limite la condition de concurrence en divisant la procédure d'assemblage en plusieurs étapes traitées séquentiellement, chaque étape correspondant à un groupe d'éléments qui, par construction, ne possèdent aucun degré de liberté en commun. En fait, la procédure d'assemblage est effectué en séquentiel par couleur, et pour chaque groupe de couleur, les éléments sont traités en parallèle [4].

Du point de vue de la génération de la matrice, à chaque groupe d'éléments correspond une matrice virtuelle, et la matrice finale est le résultat superposé à partir de toutes les matrices virtuelles, comme illustré par la Figure III.8.

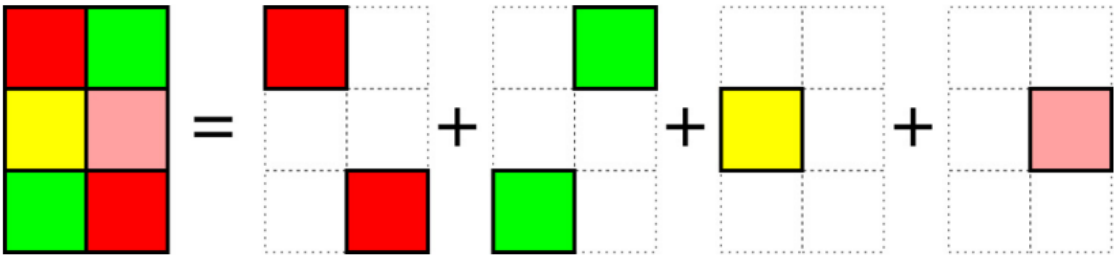


Figure III.8 Illustration du processus d'assemblage par coloriage

La matrice des coefficients `A` et le vecteur second membre `b` sont initialisés en mémoire globale. Par défaut, le format de la matrice `A` est le format CSR. Les tables de pointeur sur l'indice de ligne `CSR_rowPtr` et l'indice de colonne `CSR_col` sont construites. La table des valeurs `CSR_val` est initialisée à zéro.

Algorithme d'assemblage parallèle par coloriage

```
0. Définir le problème MEF, numérotation globale des DOFs {x}
1. Initialiser la matrice [A] de taille NxN au format CSR et vecteur {b} global de
taille N
2. Coloriage des éléments par couleur, regrouper les éléments de même couleurs dans
des groupes différents
3. Pour chaque groupe correspondant à une couleur:
4.    Pour chaque élément/thread, charger les données correspondantes
5.       Calcul d'intégration numérique parallèle : [A^e], {b^e}
6.       Assemblage parallèle des [A^e], {b^e} dans [A], {b} globale
6a.      Pour DOF p = 1 … N_D , avec M(e,p) = i, fait
6b.         b(M(e,p)) += b^e(p)
6c.         pour DOF q = 1 … N_D, avec M(e,q) = j, fait
6d.            chercher k = l'indice de A(i,j) au format CSR
6e.            CSR_val(k) +=  A^e(p,q)
6f.         Fin pour q
6g.      Fin pour p
7.    Fin de tous les éléments
8. Fin de tous les groupes
```

Les étapes de 4 à 7 sont parallélisés par un kernel. Une invocation du kernel correspond à un groupe d'éléments de même couleur. A chaque thread est associé un élément. Le thread doit accéder aux données nécessaires de son élément. Les résultats, à savoir la matrice $[A^e]$ et le vecteur élémentaire $\{b^e\}$, sont gardés en mémoire partagée pour réduire le nombre de registres utilisés. Ensuite, la procédure d'assemblage est effectuée dès que l'intégration se termine.

Alors que la procédure d'assemblage du vecteur $\{b^e\}$ est simple, l'opération d'assemblage de la matrice $[A^e]$ est plus compliquée à cause du mode de stockage de la matrice globale [A]. En effet, le stockage de la matrice [A] sous un format creux demande une recherche de l'indice réel pour chaque terme (voir dans la section III.1.3). Si la matrice élémentaire contient N_D*N_D termes non nul, cela nécessite au final N_D^2 opérations de recherche.

Le coût de la recherche est malheureusement très important à cause de la grande latence de la mémoire globale et il évolue en N_D^2 avec N_D le nombre des DOF par élément. Pour réduire le coût de recherche, [15] a proposé la détermination des indices cibles des termes non nuls avant l'opération d'assemblage. Ils sont conservés dans une table supplémentaire, notée E_k, et utilisée pour guider la phase d'assemblage. Cette table bidimensionnelle E_k est stockée en format de colonne-majeur pour un accès coalescent. La Figure III.9 montre la table E_k pour des éléments triangulaires : le nombre des lignes de E_k est égal au nombre d'éléments à traiter. Chaque ligne contient les 3 indices de nœuds (3 carrés gris sur la figure), les 6 indices des termes non nuls de la matrice élémentaires A^e[3x3] symétrique (6 carrés clairs) et les 3 indices des termes non nuls du vecteur élémentaire b^e[3x1] (3 carrés clairs).

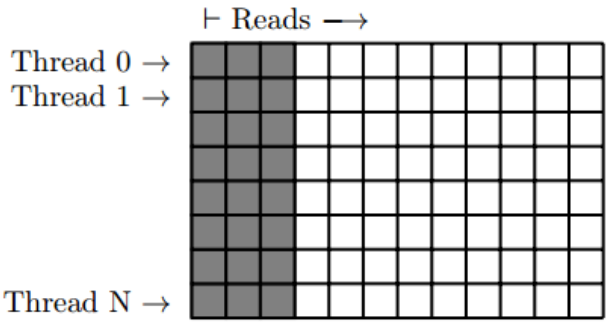


Figure III.9 Table de l'indice cible des termes non nuls de la matrice élémentaire et le vecteur élémentaire dans le cas des éléments triangulaires du premier ordre

Nous proposons une alternative qui exploite la mémoire partagée pour booster la recherche. En fait, tous les indices concernant la recherche sont transférés vers la mémoire partagée. Moyennant quoi, la rapidité de la mémoire partagée compense le coût du transfert des données. L'implantation dépend du format de la matrice creuse. Pour le format CSR, la recherche donne l'indice de colonne si la ligne des termes non nuls est déjà connue. Donc la table de `CSR_col` est transférée vers la mémoire partagée. Pour chaque DOF « i », seule la partie de `CSR_col[CSR_rowptr[i]]` à `CSR_col[CSR_rowptr[i+1]]` est transférée.

III.2.3.3 Exécution

Le maillage se compose de l'ensemble des nœuds et des éléments. Les données des nœuds sont la table des coordonnées (xi, yi) en 2D, ou (xi, yi, zi) en 3D. D'autres données associées aux nœuds peuvent être combinées à ces données initiales, comme par exemple, le mappage des DOF, la valeur des DOF connus (Condition de Dirichlet).

$$NodeCoord = \begin{bmatrix} x_1 & \dots & x_i & \dots & x_n \\ y_1 & \dots & y_i & \dots & y_n \\ \vdots & & \vdots & & \vdots \end{bmatrix} \begin{matrix} \\ \uparrow \\ n_1 \end{matrix} \quad \begin{matrix} \\ \uparrow \\ n_i \end{matrix} \quad \begin{matrix} \\ \uparrow \\ n_N \end{matrix}$$

Figure III.10 Table de coordonnées des nœuds avec N le nombre des nœuds

Les données des éléments sont la table de connectivité qui donne l'indice des nœuds de l'élément. De manière similaire, d'autres données concernant l'élément peuvent être intégrées à la table de connectivité, comme par exemple les paramètres physiques.

$$ElemConnec = \begin{bmatrix} n_1 & \dots & n_1 & \dots & n_1 \\ n_2 & \dots & n_2 & \dots & n_2 \\ \vdots & & \vdots & & \vdots \\ n_D & \dots & n_D & \dots & n_D \end{bmatrix} \begin{matrix} \\ \uparrow \\ e_1 \end{matrix} \quad \begin{matrix} \\ \uparrow \\ e_i \end{matrix} \quad \begin{matrix} \\ \uparrow \\ e_M \end{matrix}$$

Figure III.11 Table de connectivité des éléments avec M le nombre des éléments et D le nombre des nœuds per élément

La table de connectivité est stockée sous forme colonne-majeur pour faciliter la lecture coalescente sur GPU. Pour l'approche d'assemblage par coloriage, les éléments sont réorganisés par couleur ce qui demande une permutation des données sur la table de connectivité. A partir de la table de connectivité, la matrice des coefficients au format CSR sera initialisée, la table `CSR_rowptr` et `CSR_col` sont remplies par la numérotation des DOFs, la table `CSR_val` est initialisé à zéro.

Le kernel CUDA est de forme suivante :

```
__global__ void
assemblyColoringDev(NodeCoord* node,  // <x><y><map><value>
                    ElemConnec* elem, // <n1><...><ND><region>
                    PhysicData* phys, // material, source
                    GaussPointData* gp,  // <u><v><w>
                    int* csrRowPtrA,          // output     [nDof+1]
                    int* csrColIndA,          // output     [nnz]
                    float* csrValA,           // output     [nnz]
                    float* valB)              // output     [nDof]
{
       // one thread one element
       int gtid = blockIdx.x * blockDim.x + threadIdx.x;

       // initializer matElem et vecElem for a block
       __shared__ float matElemBlock[BLOCKDIM * ND * ND];
       __shared__ float vecElemBlock[BLOCKDIM * ND];
       __shared__ int dofNZBlock[BLOCKDIM * MAXNZ];
       init(&matElemBlock, &vecElemBlock, &dofNZBlock) ;

       // load nodes Data
       NodeCoord elemNode[ND];    // ND number of node per element
       loadElem(&elemNode[0], elem, node);

       //INTEGRATE
       for (int k = 0; k < NGP; k++) {
              //for each GP, update the elementary matrix and vector
              matCalc(&matElemBlock[threadIdx.x*ND*ND], phys, gp);
              vecCalc(&vecElemBlock[threadIdx.x*ND], phys, gp);
       }

       //ASSEMBLY
       for(int i = 0; i < ND; i++){
              // load CSR into dofNZBlock
              loadCSR(p[i],csrRowPtrA,csrColIndA,&dofNZBlock[threadIdx.x*ND]);
              for(int j=0; j<ND; j++){
                     // search index of NZ
                     int index = search(&dofNZBlock[threadIdx.x*ND], p[j]);
                     // update value of NZ
                     update(csrValA[index],&matElemBlock[threadIdx.x*ND*ND]);
              }
       }

       // accumulate NZ values at Free Dof (index != -1)
       for(int i=0; i< ND; i++){
              updateVec(&valB[p[i]], &vecElemBlock[threadIdx.x*ND]);
       }
}
```

Dans le kernel, MAXNZ est le nombre maximal de termes non nuls par DOF déterminé à partir de la matrice d'assemblage.

$$MAXNZ = \max(CSR\_rowPtr[i+1] - CSR\_rowPtr[i]) \ , \ i = 1 \ldots N_{DOF}$$

N_D est le nombre de DOF par élément.

III.2.3.4 Discussion

Dans l'assemblage parallèle par coloriage, la coopération et la communication entre les threads est nulle. La mémoire partagée est uniquement utilisée pour réduire le nombre total de registres utilisés ce qui est nécessaire lorsque les matrices et vecteurs élémentaires deviennent importants. Le nombre de threads par bloc (`BLOCKDIM`) est choisi en fonction de la taille de mémoire partagée.

La taille de mémoire partagée utilisée est de :

```
BLOCKDIM * ((ND*ND + ND) * sizeof(value) + MAXNZ * sizeof(index))
```

avec sizeof(value) et sizeof(index) la taille pour le stockage d'une valeur ou d'un indice.

La performance du kernel d'assemblage par coloriage est considérablement freinée par la lecture des données, surtout la table `NodeCoord`. Effectivement, comme les éléments dans un groupe ne partagent aucun nœud, la lecture des données des nœuds est multipliée par le nombre de groupes d'éléments qui contiennent ce nœud. De plus, le transfert de la table `CSR_col` vers la mémoire partagée peut ralentir l'exécution du kernel car il est impossible d'atteindre un accès coalescent en mémoire globale dans ce cas. L'optimisation de la lecture dans ce cas est impossible car les données sont fortement dispersées en mémoire globale. La mise à jour de la valeur de la matrice `CSR_valA` et du vecteur `valB` résidant en mémoire globale est également non coalescente.

III.2.4. Approche d'assemblage par pièce

La dispersion des données en mémoire globale est un défi intrinsèque à la nature de l'approche par coloriage. Il est impossible d'atteindre un accès optimisé sans réorganisation des données. C'est ce que fait l'approche d'assemblage par pièce (« patch » en anglais) ci-dessous. Dans cette section, chaque thread CUDA est associé à un élément, ce qui est un bon choix pour équilibrer le travail des threads dans la phase de calcul d'intégration.

Le principe de cette approche se base sur deux optimisations :

- Optimisation de l'accès aux données en réorganisant les petites données séparées dans une grosse transaction visant à faciliter l'accès de plusieurs threads en parallèle.
- Utilisation des mémoires partagées en remplacement de la mémoire globale à grande latence afin de booster la recherche et la mise à jour des valeurs de la matrice globale. Effectivement, la mise à jour de la valeur des termes non nuls dans un format de matrice creuse, comme CSR, est un challenge car il demande la recherche des indices. Si la recherche s'effectue en mémoire globale, sa grande latence ralentit fortement l'exécution du kernel. La solution est de transférer les données vers la mémoire partagée et d'effectuer la mise à jour en mémoire partagée.

Ces deux optimisations seront partiellement contrebalancées par la nécessité d'utiliser une instruction « *atomique* » pour l'accumulation dans la matrice globale, comme nous le montrerons par la suite.

III.2.4.1 Partition du maillage en pièces

L'accès aux coordonnées des nœuds demande une technique spécifique pour améliorer la performance. Une solution est de regrouper les éléments par pièce (« patch » en anglais). Une pièce est une partie du maillage ce qui se compose d'un ensemble d'éléments connectés. La Figure III.12 illustre la décomposition d'un maillage triangulaire en pièces par le logiciel METIS [71], dont logiciel la vocation est de partitionner des graphes. Dans notre problème, chaque élément joue le rôle d'un nœud du graphe et chaque arête partagée entre deux éléments joue le rôle d'une arête du graphe.

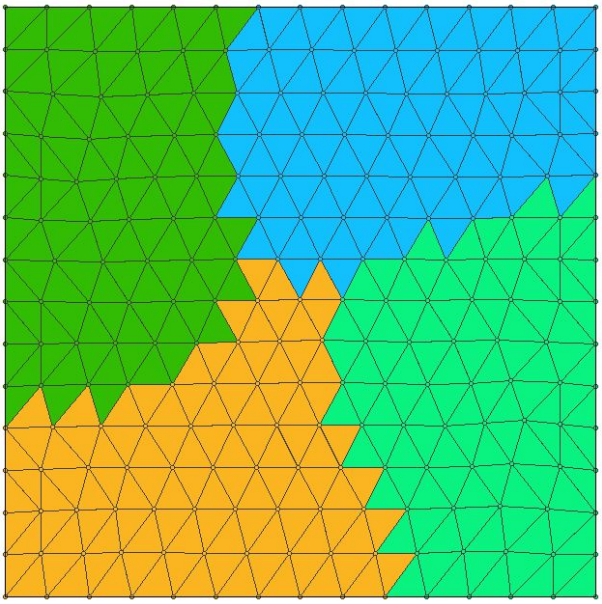


Figure III.12 Illustration de la décomposition d'un maillage 2D par METIS

La taille de la pièce mesurée par le nombre des éléments par pièce est un paramètre important qui influence le coût de décomposition. En effet, le temps de décomposition d'un maillage augmente quasi linéairement avec le nombre des pièces, comme montré dans la Figure III.13. Construire des grandes pièces minimise donc ce coût.

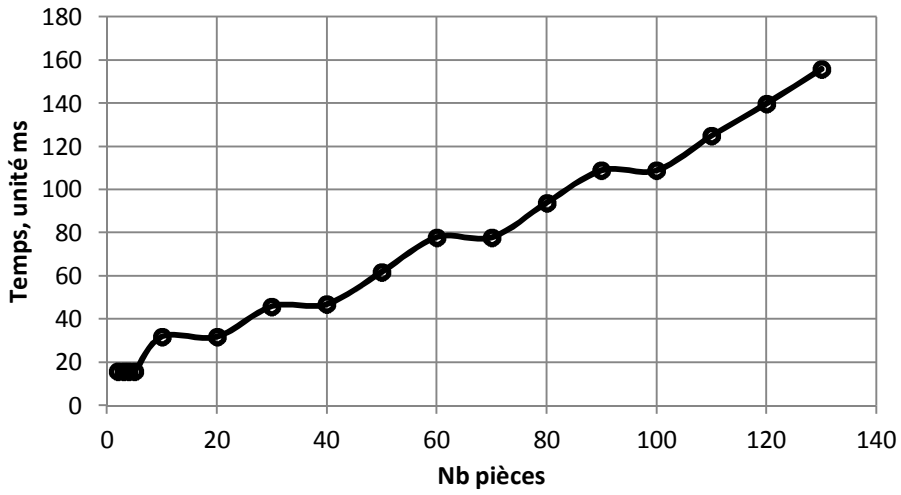


Figure III.13 Evolution du temps de décomposition en fonction du nombre de pièce pour le cas d'un maillage de 17 302 éléments en 2D

Pour estimer le temps de décomposition, nous considérons des maillages différents dont le nombre d'éléments par pièce est fixé respectivement de 32, 64 et 128. Le coût de décomposition augmente linéairement avec le nombre des éléments, cf. Figure III.14.

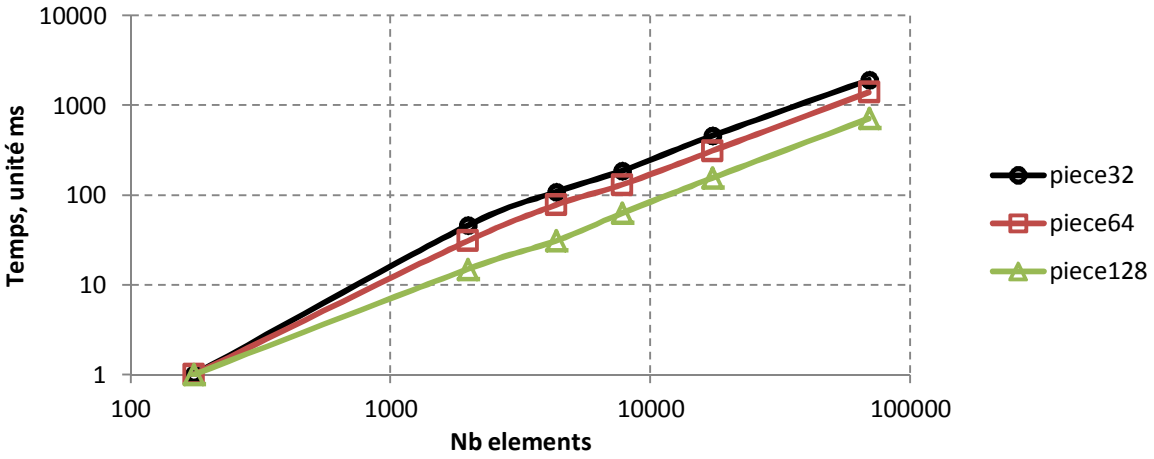


Figure III.14 Evolution du temps de décomposition en fonction du nombre des éléments pour le cas de 32, 64 et 128 éléments par pièce

III.2.4.2 Optimisation de l'accès aux données

Lors de l'intégration numérique, chaque thread est associé à un élément et requiert l'accès aux données des nœuds de cet élément. Pour éviter les redondances d'accès, la démarche retenue consiste à regrouper les données des nœuds par paquets et les transférer vers la mémoire partagée en une fois. En effet, les données en mémoire partagée sont accessible aux threads avec une vitesse est bien meilleure qu'en mémoire globale.

Tout d'abord, la décomposition du maillage en pièces sans chevauchement est effectuée. Chaque pièce peut être considérée comme un sous-maillage dont les éléments partagent un ensemble de nœuds communs (cf. Figure III.12). Le nombre d'éléments par pièce est relativement constant. Il faut limiter le nombre d'éléments dans une pièce pour que les données correspondantes rentrent bien dans la mémoire partagée du GPU. Chaque pièce va être traitée par un bloc CUDA, tandis que chaque élément est associé un thread dans le bloc. En conséquence, le nombre d'éléments par pièce est nécessairement 32 ou un multiple de 32 à cause de la taille du warp.

Ensuite, les données de tous les nœuds d'une pièce doivent être réorganisées en les réunissant en paquets contiguës de mémoire. Cela vaut pour les nœuds intérieurs de la pièce ainsi que pour les nœuds situés sur les bords des pièces. Le regroupement des données des nœuds facilite la lecture en mémoire globale et les transferts en mémoire partagée. Dans ce cas, tous les threads doivent participer à la procédure de transfert pour atteindre la lecture coalescente.

On remarque que les nœuds sur les bords des pièces sont sollicités par plusieurs pièces. Donc, pour optimiser la lecture des données de ces nœuds, la procédure de décomposition doit tendre à minimiser l'interface entre les pièces, i.e., le nombre d'arêtes de l'interface.

Le résultat de la décomposition du graphe par METIS procure un nombre quasi identique d'éléments par pièce et une minimisation des arêtes de l'interface. Par contre, le nombre d'éléments par pièce n'est pas directement paramétrable dans METIS car sa démarche est basée

sur une division récursive d'un graphe en deux. En pratique, nous réglons le nombre de divisions de METIS pour nous approcher de 32 ou d'un multiple de 32 éléments par pièce, en utilisant $|E/E_k,|$ avec $||$ l'opération arrondi supérieur. Si on prend l'exemple d'un maillage comportant 100 éléments et une décomposition de 32 éléments par pièce, $E = 100$, $E_k = 32$ et $|E/E_k| = 4$ pièces, chaque pièces contient plus ou moins $100/4 = 25$ éléments.

Par la suite, il est possible d'ajuster le nombre d'éléments par pièce en échangeant quelques éléments entre les pièces. Dans un exemple de 100 éléments, on le peut diviser en 4 pièces, dont 3 pièces de 32 éléments et 1 pièce de 4 éléments. Dans ce cas, il existe (32-4)=28 threads suspendus dans le bloc correspondant à la pièce de 4 éléments. En conséquence, la situation déséquilibre du nombre d'éléments entre les pièces cause une perte légère de performance de la méthode.

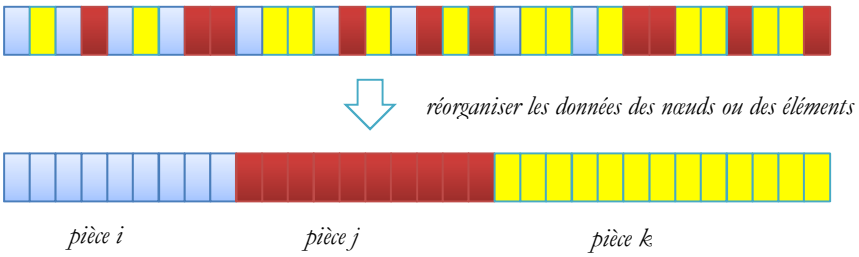
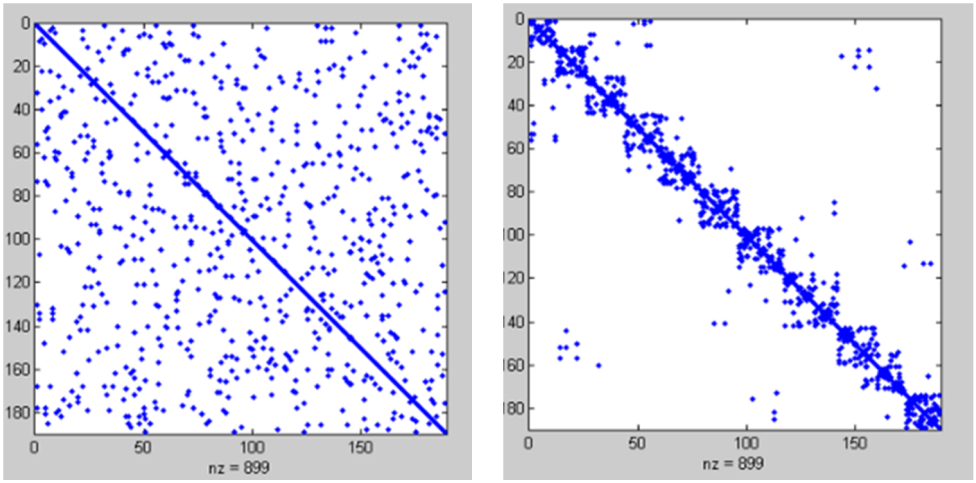


Figure III.15 Réorganisation des données des nœuds et des éléments selon la pièce. Chaque carré représente les données d'un nœud ou d'un élément. La couleur représente la pièce correspondante des nœuds ou des éléments

Après décomposition du maillage, les données des nœuds et des éléments sont réorganisées par pièce. Il s'agit d'une permutation des données, illustrée par la Figure III.15. Au final, les données des nœuds ou des éléments d'une même pièce sont proches d'une séquence contiguë en mémoire globale, donc l'accès aux données est facilement coalescent.

III.2.4.3 Algorithme

D'une manière similaire à la réorganisation des données des nœuds, la numérotation des DOFs associés aux nœuds est réorganisée par pièce. En conséquence, les DOFs d'une pièce sont compressés comme illustré à la Figure III.15. Au final, la topologie de la matrice d'assemblage est divisée en blocs comme le montre la Figure III.16. Du point de vue de la procédure d'assemblage, le regroupement des nœuds par pièce produit une décomposition de la matrice et du vecteur en blocs disjoints. En conséquence, chaque pièce correspond à un bloc de la matrice A et du vecteur b. En pratique, la procédure d'assemblage devient plus simple : pour chaque pièce, transférer une partie correspondante de la matrice A et le vecteur b vers la mémoire partagée, effectuer l'assemblage en mémoire partagée et recopier les résultats vers la mémoire globale.



Numérotation aléatoire Numérotation réorganisé par pièce

Figure III.16 Distribution des termes non nuls de la matrice des coefficients selon la numérotation des DOFs

Enfin, comme avec l'approche d'assemblage par coloriage, il est nécessaire d'initialiser la matrice des coefficients à partir des informations du maillage avant la procédure d'assemblage. Le format CSR est choisi par défaut. La table des pointeurs de ligne `CSR_rowPtr` et la table des indices de colonne `CSR_col` des termes non nuls sont déterminées par la numérotation des DOF associés aux nœuds. La table de valeur `CSR_val` est initialisée à zéro.

Pour synthétiser, la procédure d'assemblage suit les cinq étapes ci-dessous. Une opération de synchronisation des threads est obligatoirement à effectuer entre deux étapes consécutives :

- Etape 1 : Chaque bloc correspond à une pièce de N_k nœuds et de E_k éléments. Transférer les données des nœuds vers la mémoire partagée. Nous utilisons N_k threads parallèles pour réaliser cette étape. Ainsi, la lecture des données en mémoire globale est coalescente. Il est nécessaire de synchroniser les threads pour s'assurer du transfert de l'intégralité du transfert.

- Etape 2 : Chaque thread correspond à un élément : charger la table de connectivité en mémoire globale et charger les données des nœuds en mémoire partagée. Pour les nœuds $n_i \in N_k$, le chargement se base sur le mappage de l'indice local et global des nœuds qui est créé pour chaque pièces. Pour les nœuds sur les interfaces, les données des nœuds ne se situent pas en mémoire partagée, mais en mémoire globale. La matrice élémentaire A^e et le vecteur élémentaire b^e sont en registres. On effectue le calcul numérique des intégrales, le résultat est stocké dans A^e et b^e.

- Etape 3 : Transférer partiellement la matrice globale A de la mémoire globale à la mémoire partagée. Si chaque bloc correspond à D_k degrés de liberté, notons d_i et d_j respectivement le premier et le dernier DOF de la pièce, avec $|D_k| = d_j - d_i$ le nombre des DOF de la pièce. Donc une séquence de la matrice A entre `CSR_rowPtr`[d_i] et `CSR_rowPtr`[d_j+1] est chargée vers la mémoire partagée. Le nombre d'indices ou de valeurs de la séquence est de :

$$NNZ_k = CSR\_rowPtr[d_j+1] - CSR\_rowPtr[d_i]$$

Seules les tables des indices de colonne `CSR_col` et celle des valeurs `CSR_val` sont transférées vers la mémoire partagée tandis que la table de pointeurs de ligne est accessible en mémoire globale. La lecture en mémoire globale est coalescente dans tous les cas.

- Etape 4 : Chaque thread effectue l'assemblage des termes non nuls élémentaires en mémoire partagée. Chaque terme non nul élémentaire A^e_{kl} doit être accumulé dans son pendant global A_{ij} avec M(e,k) = i et M(e,l) = j. Tous les termes non nuls concernant le DOF « i » sont repérés par les pointeurs de `CSR_rowPtr`[i] et `CSR_rowPtr`[i+1], qui correspondent à une séquence d'indice de colonne de `CSR_col[CSR_rowPtr[i]]` à `CSR_col[CSR_rowPtr[i+1]]` et une séquence de valeurs `CSR_val[CSR_rowPtr[i]]` à `CSR_val[CSR_rowPtr[i+1]]`. L'objectif est de trouver l'indice de valeur « j » dans une séquence de colonne et puis d'accumuler la valeur A^e_{kl} vers la valeur A_{ij} avec l'indice trouvé. Pour cela, on utilise une recherche binaire, par exemple « quicksort » puisque l'indice de colonne est déjà trié. L'opération d'addition demande une opération atomique (voir Chap1), ie `atomicAdd`() en mémoire partagée pour assurer le bon résultat car cette valeur est mise à jour par de multiple threads en parallèle, avec accès concurrent. Les DOF associés avec les nœuds de l'interface demandent un traitement particulier. La recherche des indices et l'accumulation sont exécutées en mémoire globale. La performance de la recherche et l'opération atomique en mémoire globale est beaucoup plus lente par rapport à la mémoire partagée mais ceci est acceptable car il y a un très petit nombre de DOFs aux l'interface entre pièces par rapport au nombre de DOFs intérieurs. La valeur du vecteur b^e suit une procédure similaire.

- Etape 5 : La table de valeur `CSR_val` en mémoire partagée est transférée en retour vers la mémoire globale. La séquence de la table correspondante à la pièce est recopiée de la même manière qu'elle a été transférée dans l'étape précédente. Il faut cependant faire attention aux problématiques de concurrences qui peuvent arriver si une pièce recopie la valeur de `CSR_val` en retour pendant qu'une autre pièce accumule les valeurs de `CSR_val` pour ses DOFs de l'interface. Cette situation est évitée en stockant les valeurs non nulles dans une table supplémentaire `CSR_val*`, qui est de même taille avec `CSR_val`. Le résultat final est obtenu par un autre kernel qui fait la somme de `CSR_val*` et `CSR_val` dès que toutes les pièces ont terminé leurs travaux.

L'algorithme pour ce cas est le suivant:

```
0. Définir le problème MEF, numéroter des DOF globales {x}
1. Décomposer le maillage en pièce des éléments, réorganiser les données des nœuds
ainsi des éléments en fonction de la pièce correspondante, renuméroter des DOF
globales
2. Initialiser la matrice [A] de taille NxN au format CSR et vecteur {b} de taille
N à partir l'information du maillage
3. Imposer chaque bloc à une pièce, chaque thread à un élément, pour chaque thread
4.    Transférer les données des nœuds vers la mémoire partagée,
5.    __syncthreads(),
6.   Initialiser et calculer la matrice A^e et vecteur élémentaire b^e
7.    __syncthreads(),
8.    Transférer la matrice A vers la mémoire partagée : La table CSR_col pour la
      recherche de l'indice, la table CSR_val pour accumuler les non zéros,
9.    __syncthreads(),
10.   Effectuer l'assemblage des non zéros
10a.  pour k = 1…Nd, avec M(e,k) = i
10b.    accumuler b^e(k) dans b(i) avec atomicAdd(),
10c.    pour q = 1…Nd, avec avec M(e,k) = j
10b.        chercher   j   dans   l'intervalle   de   CSR_col(CRS_rowPtr(i))   à
            CSR_col(CRS_rowPtr(i+1)), fixer l'indice ciblé (=id)
10c.        accumuler A^e_kl dans CSR_val(id) ou CSR_val*(id) avec atomicAdd()
10d.   fin pour q
10^e.  fin pour q
11.   __syncthreads(),
12.   transférer à retour CSR_val à la mémoire globale.
13. Accumuler CSR_val* dans CSR_val.
```

Dans l'algorithme, `__syncthreads()` dénote la méthode de synchronisation des threads d'un bloc CUDA. Les threads sont synchronisés après chaque chargement/déchargement des données en mémoire partagée. La valeur des termes non nuls de la matrice et du vecteur élémentaire est gardée temporairement dans les registres ou la mémoire locale. Les variables du kernel sont en registres par défaut. Mais si le nombre de variables du kernel dépasse le nombre des registres, elles sont en mémoire locale dont l'accès est beaucoup plus lent que celui des registres. Par exemple, pour l'élément triangulaire de 3ème ordre, le nombre de DOFs des éléments iso-paramétrique est de 10. Donc la taille de la matrice élément est de 10x10=100 tandis que le nombre maximal utilisable de registres par threads est de 63 avec le GPU Fermi. En conséquence, la performance du kernel diminue considérablement avec l'augmentation de l'ordre de l'élément.

III.2.4.4 Exécution

Le premier kernel qui combine le calcul numérique et l'assemblage par pièce est présenté en version simplifiée dans le code suivant :

```
__global__ void
assemblyPatchDev(NodeCoord*  g_node,   // <x><y><position><value> N_n = number of nodes
                 ElemConnec*  g_elem,      // <n1><...><nn><region>  N_e = number of elements
                 PhysicData*     g_phys,       // material, source
                 GaussPointData* g_gp,     // <u><v><w>           N_gp = numbre of Gauss Point
                 PatchData*   g_part,      // <estart><eend><nstart><nend><dofstart><dofend>
                 int*         csr_rowptr,  // output [nDof+1]
                 int*         csr_col,     // output [nnz]
                 float*       csr_val,     // output [nnz]
                 float*       csr_vals,    // output [nnz]
                 float*       valb)        // output [nDof]
{
   //    note: 't_' thread      'b_' bloc        'g_' global        's_' shared

   /******** LOAD NODES COORDINATES AND ELEMENT CONNECTIVITY INTO SM  ************/

   // shared memory requires a computation of enough dimension for all storages
   extern __shared__ float sm[];

   // thread index = element index
   int tid = threadIdx.x;

   // block index = patch index
   int bid = blockIdx.x;

   // nodes coordinates of patch
   NodeCoord* b_nodeCoord = (NodeCoord*) &sm[0];

   // load nodes coordinates
   loadNodeCoord(bid, g_part, b_nodeCoord, g_node);

   __syncthreads();
   /********************************** INTEGRATION *****************************/
   // using registers
   float t_matElem[ND*ND];         // matrix elementary
   float t_vecElem[ND];         // vector elementary

   // init matrix, vector
   init(&t_matElem, &t_matElem);

   // Integration elementary and save results in the vector & matrix elementary

   // get nodes coordinates
   NodeCoord t_nodecoord[ND];
   loadElem(tid, g_elem, bid, g_part, b_nodeCoord, &t_nodecoord);

   //INTEGRATE
   for (int k = 0; k < NGP; k++) {
      //for each GP, update the elementary matrix and vector
      matCalc(&t_matElem[0], phys, gp);
```

```
        vecCalc(&t_vecElem[0], phys, gp);
    }
    __syncthreads();

    /****************************** LOAD CSR MATRIX ******************************/

    int b_NNZ = g_part.getNNZ(bid);        // size = nb_nnz per patch

    // load col-index & value arrays correspond on patch
    int *sm_csr_col = (int*) &sm[0];
    float *sm_csr_val  = (float*) &sm_csr_col[b_NNZ];         // size = nb_nnz

    loadCol(bid, g_part, csr_col,    sm_csr_col);
    loadVal(bid, g_part, csr_val,    sm_csr_val);

    __syncthreads();
    /********************************** ASSEMBLY  ***********************************/

    for(int i=0; i<ND; i++){

        // update vector b by atomic add
        atomicAdd(p[i], &valb, t_vecElem[i]);  // p[i] le map local-global of dof i

        for(int j=0; j<ND; j++){

            // check Dof inside/outside of Patch
            bool flag_dof = insidePatch(p[i], g_part, bid)

            // if dof inside Patch
            if(flag_dof)
            {
                // search p[j] in array col-index (share memory), return index
                int index = searchSM(sm_csr_col, p[i], p[j], g_part, bid);

                // update value array at index-th entry (share memory) by atomic add
                atomicAdd(&sm_csr_val[index], t_matElem[i*ND+j]);
            }
            // if dof boundary
            else
            {
                // search p[j] in csr_col (global memory), return index
                int index = searchGM(csr_col, p[i], p[j], csr_rowptr);

                // update value array at index-th entry (global memory) by atomic add
                atomicAdd(&csr_vals[index], t_matElem[i*ND+j]);
            }

        }

    __syncthreads();

    /********************************** WRITE BACK  ***********************************/

    // write back segment of VALUE array on share memory to global memory
    loadVal(bid, g_part, sm_csr_val, csr_val);

}
```

La deuxième kernel sert à faire l'addition de CSR_val et CSR_val*. On peut agréablement utiliser la fonction `cublas<t>axpy()` du package CUBLAS. Cette fonction multiple le vecteur `x` par une valeur scalaire `alpha`, et fait l'addition avec un vecteur `y`. Le résultat est stocké dans le vecteur `y`. La formulation est : `y[j] = alpha * x[k] + y[j]` avec `k = 1+(i-1)*incx`, `j = 1+(i-1)*incy`, `i=1…n`. L'application de cette fonction pour deux vecteurs est la suivante :

```
cublasSaxpy(nnz, 1.0,CSR_vals, 1, CSR_val,1);
```

avec `nnz` est la taille de deux tables.

III.2.4.5 Dimensionnement de la pièce

Comme les données de la pièce sont transférées vers la mémoire partagée, la taille de la pièce doit s'adapter à la taille de la mémoire. On dénote N l'ensemble des nœuds du maillage, E l'ensemble des éléments du maillage. On définit la pièce « k » comportant N_k nœuds, E_k éléments avec $N_k \subset N$ et $E_k \subset E$. Si les données d'un nœud se composent de n_d coordonnées et n_f quantités scalaires associée à nœud, alors la taille des données des nœuds par pièce est de :

$$|S_n| = N_k(n_d+2*n_f)*\text{sizeof(value)} \text{ et } |S_n| \le |S_{sm}| \quad \forall k \qquad \text{(III.50)}$$

avec $|S_{sm}|$ est la taille de la mémoire partagée du GPU.

La mémoire partagée sert au stockage non seulement des données des nœuds mais également des données de la matrice A. Donc, elle doit respecter une deuxième condition attachée à la taille de la matrice A. La taille de la matrice A d'une pièce tient compte de la taille d'une séquence de `CSR_col` et de ses valeurs `CSR_val` :

$$|S_A| = NNZ_k * (\text{sizeof(index)} + \text{sizeof(value)}) \text{ et } |S_A| \le |S_{sm}| \quad \forall k \qquad \text{(III.51)}$$

avec NNZ_k le nombre des termes non nuls de la matrice A correspondant à la pièce.

Les conditions (III.50) et (III.51) doivent être vérifiées pour toutes les pièces avant la procédure d'assemblage. Si une quelconque des deux conditions n'est pas respectée, la décomposition du maillage doit relancer en diminuant la taille moyenne des pièces.

III.2.4.6 Discussion

L'application de l'algorithme d'assemblage par pièce des éléments est limitée par la taille de la mémoire partagée du GPU. Cette mémoire partagée est utilisée non seulement pour le stockage des données des nœuds, mais également pour les données de la matrice d'assemblage. Pour les éléments d'ordre peu élevé, la taille des données des nœuds ainsi que le nombre des termes non nuls par nœuds est petit. Cela autorise la décomposition du maillage d'origine en pièce comportant de nombreux éléments et ainsi que de nombreux nœuds. Dans ce cas, la performance du kernel d'assemblage augmente considérablement parce que l'algorithme limite au maximum la lecture inutile des nœuds partagés entre éléments et réduit le cout du traitement des DOFS sur l'interface entre pièces. Cependant, pour les éléments d'ordre élevé, le nombre des termes non nuls augmente rapidement avec l'ordre, et il devient obligatoire de diminuer le nombre des nœuds par pièce pour pouvoir insérer la matrice d'assemblage dans la mémoire partagée. En conséquence, le nombre d'éléments per pièce diminue, le maillage est divisé en plus petites pièces et le nombre des DOFs sur l'interface augmente, ce qui ralentit considérablement la procédure d'assemblage.

L'usage des registres pour le stockage du résultat temporaire de l'intégration numérique entraine également un risque de baisse performance avec les éléments d'ordre élevés. Dans ce cas, le nombre des variables dépasse le nombre maximal des registres utilisables du kernel, ce qui conduit à l'utilisation de la mémoire locale plus lente.

III.2.5. Développement d'une nouvelle approche d'assemblage global

Dans cette section, nous présentons une nouvelle approche qui est centrée sur l'usage de la mémoire globale pour la procédure d'assemblage. Elle est différente des deux approches précédentes d'assemblage par coloriage et par pièce. Ces deux dernières approches utilisaient généralement une étape de pré-calcul ou de réorganisation des nœuds et des éléments. En outre, la matrice des coefficients A qui est stockée sous forme creuse nécessite une phase de construction topologique avant la procédure d'assemblage sur GPU. Les coûts de la réorganisation et de cette construction préalable réduisent finalement la performance. Notre méthode présentée ci-dessous ne nécessite pas ces étapes de prétraitement.

L'approche proposée repose sur deux procédures : intégration et assemblage. Toutes deux sont effectuées en parallèle mais par des kernels différents. Pour l'intégration élémentaire, chaque élément se voit attribué un thread. Le résultat de l'intégration est stocké en mémoire globale. Pour la procédure d'assemblage, chaque DOF est associé à un thread. Il accède aux termes non nuls en mémoire globale, fait l'addition des contributions et retourne le résultat de la matrice et du vecteur final. La nouveauté de notre approche par rapport à [68] est que l'on stocke d'abord le résultat de l'intégration numérique dans une matrice au format COO, puis, une procédure de tri et de réduction/accumulation des contributions de termes non nuls s'effectue directement sur le format COO et conduit au format CSR. Toutes ces procédures sont exécutées en mémoire globale du GPU.

III.2.5.1 Algorithme

La procédure générale se compose de trois étapes : l'intégration, le tri des termes non nuls et la réduction/accumulation des termes non nuls, comme illustré à la Figure III.17:

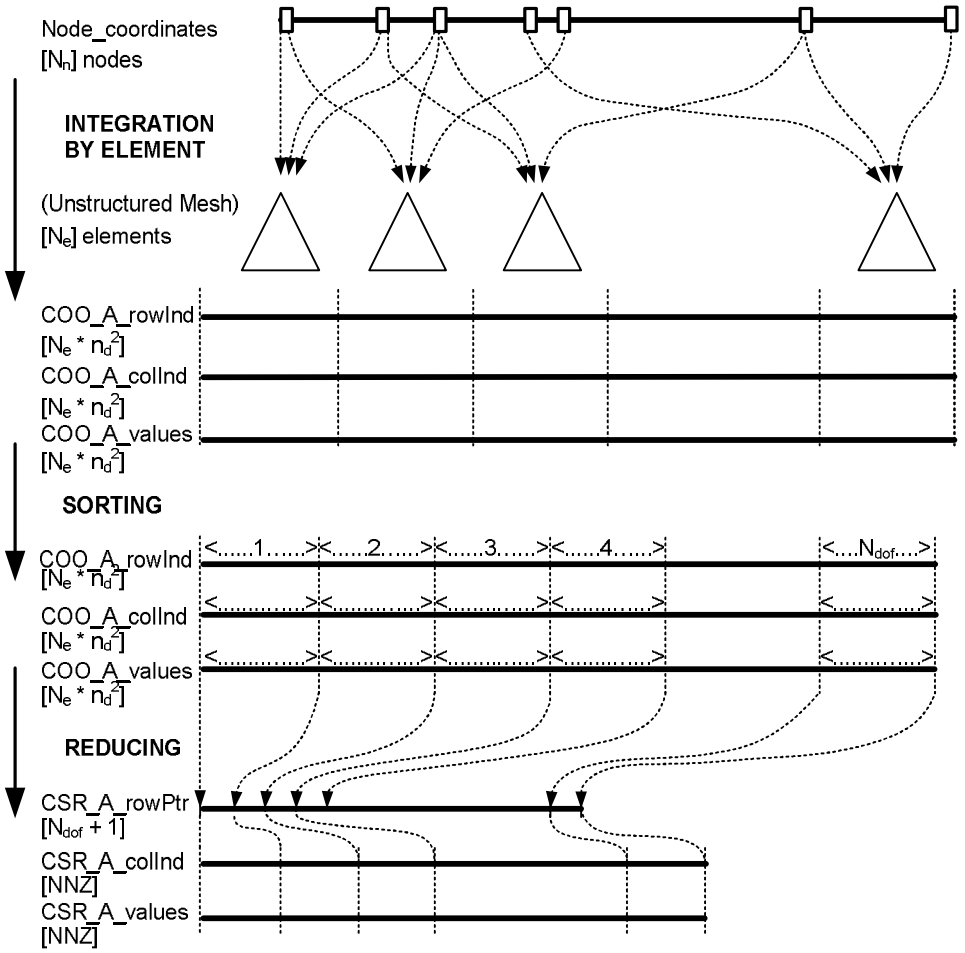


Figure III.17 Algorithme d'assemblage global en trois étapes : l'intégration par l'élément, le tri des termes non nuls et la réduction des contributions

III.2.5.2 Intégration numérique

Tout d'abord, nous attribuons à chaque thread le calcul de l'intégration sur un élément. Le kernel lit la table de connectivité de l'élément d'une manière coalescente, et puis charge les données des nœuds. L'accès aux données est identique à l'approche par coloriage donc nous ne l'abordons pas d'ici. Les variables sont réservées dans les registres. La matrice des coefficients et du vecteur élémentaire est déclarée en mémoire partagée. Quant à la sortie, tous les termes non nuls des matrices élémentaires A^e sont stockés en mémoire globale dans une grande matrice temporaire au format COO, appelée COO_A et dimensionnée pour permettre de recueillir toutes les matrices élémentaires. Trois tables de même taille sont allouées: COO_row pour l'indice de ligne, COO_col pour l'indice de colonne et COO_val pour les valeurs. Pour que l'écriture en mémoire globale soit coalescente, nous utilisons la mémoire partagée comme mémoire intermédiaire et rassembler tous les threads à cette procédure. Une synchronisation des threads est nécessaire à ce point. En pratique, la procédure d'enregistrement est divisée par trois étapes, chaque étape correspond à une table, respectivement à COO_val tout d'abord, et puis COO_row et enfin COO_col.

La taille de la matrice COO_A se déduit simplement à partir de la taille des matrices élémentaires A^e. Supposons que le maillage comporte N^e éléments et que le nombre de DOF est

n_d par élément, la taille de la matrice élémentaire A^e est (n_d * n_d) et la taille d'une table de la matrice A est de (N^e * n_d * n_d).

Dans cette étape, les conditions de Dirichlet doivent être appliquées. Pour chaque DOF connu, tous les termes de la matrice élémentaire concernant ce DOF doivent être ignorés, ce qui se traduit par une mise à zéro tandis que leurs indices de ligne et de colonne sont marqués comme invalide, ex. -1.

Le traitement du vecteur élémentaire est plus simple car l'indice de la valeur est déjà connu. L'addition des valeurs du vecteur élémentaire fait usage de la fonction atomique atomicAdd() en mémoire globale.

III.2.5.3 Tri des termes non nuls

Dans cette étape, nous utilisons une méthode de tri parallèle pour réarranger les valeurs non nulles dans la matrice au format COO par leur indice de ligne et de colonne. Les termes non nuls sont triés tout d'abord par leurs indices de colonne et puis par leurs indices de ligne. Les trois tables sont triées en même temps et selon le même critère. Une matrice au format COO est obtenue à la fin de cette procédure.

En général, l'opération de tri est bien réalisée sur une plateforme parallèle de GPU et il existe plusieurs méthodes de tri efficaces pour cette architecture. Dans notre test, nous avons utilisé un algorithme de tri spécifique, i.e. « radix sort » dans la bibliothèque Cudpp à travers du package JCuda qui fonctionne sur notre environnement de programmation JAVA. L'opération de tri du package Cudpp permet de trier deux tables de données de même taille mais de type différents. Dans le cas du format COO, on a trois tables à trier de manière synchrone. En pratique, nous utilisons une table d'indice supplémentaire qui conserve les indices de permutation d'une des 3 tables table qui est triée par Cudpp. Les deux autres tables sont ensuite triées à l'aide de cette table d'indice. Une autre solution est l'usage la méthode sort_by_key du package Thrust en combinaison avec « thrust :: zip_iterator » et « thrust :: tuple ». En comparant les deux techniques, le package Thrust nous a fourni une solution plus flexible et plus de productivité donc nous avons conservé cette solution comme outil de tri. Pour utiliser Thrust en Java, nous invoquons le JNI (java native interface) et effectuons le tri dans C++.

Désormais, la matrice finale au format COO triée sépare les valeurs en paquets par DOF selon le principe illustré sur la Figure III.17. Chaque zone correspond à un DOF et contient plusieurs contributions de termes non nuls qui possèdent un même indice de ligne et de colonne. Ces valeurs sont stockées en mémoire globale dans une zone contiguë, i.e. très proche l'un à l'autre ce qui facilite leur lecture coalescente dans la phase de réduction suivante. Exceptionnellement, la matrice contient également des termes non nuls marqués comme invalides qui correspondent au DOF connus des conditions de **Dirichlet**. Tous ces termes créent une zone « invalide » qui se situe normalement en tête des trois tables et qui doivent être éliminés avant la procédure de réduction par une simple opération de décalage des pointeurs de données.

III.2.5.4 Réduction des contributions

Dans cette procédure, les contributions de termes non nuls qui ont le même indice de ligne et de colonne sont condensées en une valeur unique en faisant l'addition de leurs valeurs. Le résultat

est stocké dans un format CSR. A chaque thread est assignée la réduction des termes d'un DOF dont le stockage est sur une partie contiguë de mémoire globale.

Avant de la réduction, la taille de la matrice CSR qui correspond au nombre total des termes non nuls, noté NNZ, doit être déterminé. Pour cela, pour chaque terme non nul, on compare son indice de ligne et de colonne avec ceux de ses deux voisins : un terme avant et un terme suivant. Si leurs indices sont différents, on accumule +1 dans NNZ. Un kernel est invoqué dans ce cas pour compter NNZ à l'aide de la mémoire partagée. En pratique, nous utilisons une méthode spécifique du package Thrust, i.e. « thrust ::inner_product() » qui fonctionne selon le même principe.

La matrice au format CSR est dimensionnée avec NNZ termes non nuls pour les indices de colonne CSR_col et les valeurs CSR_val. La table de pointeurs de ligne CSR_rowPtr est de taille (N + 1) avec N est le nombre de DOF. Ensuite, nous utilisons un kernel CUDA sur lequel chaque thread parallèle est affecté à un DOF. Le kernel accède un segment de mémoire correspondant à son DOF et effectue une réduction des termes non nuls adjacents si leurs indices de ligne et de colonne sont identiques. La comparaison et la réduction peut s'exécuter en mémoire partagée afin d'atteindre la meilleure performance. On utilise la fonction « thrust_reduce_by_key()» du package Thrust.

III.2.5.5 Exécution

Le kernel de calcul numérique est le suivant :

```
__global__ void
assemblyGlobalDev(NodeCoord* node, // <x><y><position><value> N_n = number of nodes
             ElemConnec*  elem,        // <n1><...><nn><region>  N_e = number of elements
             PhysicData*     phys,        // material, source
             GaussPointData* gp,       // <u><v><w>       N_gp = numbre of Gauss Point
             int*         coo_row,     // output [N_e * nd * nd]
             int*         coo_col,     // output [N_e * nd * nd]
             float*       coo_val,  // output [N_e * nd * nd]
             float*       valb)     // output [nDof]
{
   // One Thread One Element
   int gtid = threadIdx.x + blockIdx.x * blockDim.x;

   __shared__ float sm[(blockDim.x * ND + 1) * ND];

   // get pointer to shared memory to store the matrix & vector for elementary integration
   float* matElemBlock = (float*) &sm[threadIdx.x * ND * ND];
   float* vecElemBlock = (float*) &sm[blockDim.x * ND * ND];
   int* rowElemBlock = (int*) &sm[threadIdx.x * ND * ND];
   int* colElemBlock = (int*) &sm[threadIdx.x * ND * ND];

   // initialize elementary matrix and vector
   init(matElemBlock, vecElemBlock);

   // load nodes Data
   NodeCoord elemNode[ND];   // ND number of node per element
   loadElem(&elemNode, elem, node);

   //INTEGRATE
   for (int k = 0; k < NGP; k++) {
      matCalc(&matElemBlock[threadIdx.x*ND*ND], elemNode, phys, gp);
      vecCalc(&vecElemBlock[threadIdx.x*ND], elemNode, phys, gp);
   }

   //ASSEMBLY
   updateVec(valb, elemNode, &vecElemBlock[threadIdx.x*ND]);

   //write value NZ in COO format
   updateValue(matElemBlock, coo_val);
   __syncthreads();
```

```
    // load row index to sm
    loadRowIndex(rowElemBlock, elemNode);
    __syncthreads();

    // write row index in COO format
    updateRow(rowElemBlock, coo_row);
    __syncthreads();

    // load column index to sm
    loadColIndex(colElemBlock, elemNode);
    __syncthreads();

    // write column index in COO format
    updateCol(colElemBlock, coo_col);
    __syncthreads();
}
```

Pour le tri des trois tables, en utilisant Thrust

```
    // sort by keys: by column Indexes first and by row Indexes after
    thrust::sort_by_key(coo_col.begin(),coo_col.end(),
    thrust::make_zip_iterator(thrust::make_tuple(coo_row.begin(),coo_val.begin())));
    thrust::sort_by_key(coo_row.begin(),coo_row.end(),
    thrust::make_zip_iterator(thrust::make_tuple(coo_col.begin(),coo_val.begin())));
```

Pour déterminer le nombre de termes non nuls :

```
    // compute unique number of nonzeros in the output
    int num_entries = thrust::inner_product(
    thrust::make_zip_iterator(thrust::make_tuple(new_row.begin(), new_col.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(new_row.end (), new_col.end())) - 1,
    thrust::make_zip_iterator(thrust::make_tuple(new_row.begin(),
    new_col.begin())) + 1,
    int(0),
    thrust::plus<int>(),
    thrust::not_equal_to< thrust::tuple<int,int> >()
    ) + 1;
```

Pour la réduction de valeur des termes non nuls :

```
    // sum values with the same (i,j) index
    thrust::reduce_by_key(
    thrust::make_zip_iterator(thrust::make_tuple(coo_row.begin(), coo_col.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(coo_row.end(), coo_col.end())),
    coo_val.begin(),
    thrust::make_zip_iterator(thrust::make_tuple(csr_row.begin(),csr_col.begin())),
    csr_val.begin(),
    thrust::equal_to< thrust::tuple<int,int> >(),
    thrust::plus<float>());
```

Pour convertir COO à CSR :

```
    // convert uncompressed row indices into compressed row offsets
    thrust::lower_bound(csr_row.begin(),
                        csr_row.end(),
                        thrust::counting_iterator<int>(0),
                        thrust::counting_iterator<int>(csr_rowptr.size()),
                        csr_rowptr.begin());
```

III.2.5.6 Discussion

L'approche d'assemblage global combine les avantages des deux autres approches, l'assemblage par coloriage et par pièce. L'intégration numérique par élément aide à équilibrer la masse de travail vers des threads en parallèle. La séparation de l'intégration numérique par élément et de l'assemblage par DOF permet d'éviter la condition de concurrence. Cette condition demandait une procédure permettant de répartir les éléments par couleurs dans l'approche par

coloriage et elle demandait une fonction atomique dans l'approche par pièce. De plus, la division en deux parties de la démarche globale aide à diminuer l'usage des registres par chaque partie ce qui est un paramètre essentiel de la performance du kernel. Rappelons que le nombre de registres utilisable par thread est également limité par l'architecture du GPU (cf. I.3.5.3). De plus, la mémoire globale est suffisamment grande pour réserver la matrice et le vecteur élémentaire même si l'ordre de l'élément est élevé ce qui était impossible en mémoire partagée dans l'approche de pièce.

III.3. Parallélisation sur GPU de la résolution

Dans cette section, nous mettons au point l'usage de packages spécifiques de CUDA pour paralléliser le calcul de la résolution. Ces packages définissent les formats habituels de stockage de matrices denses et creuses. Ils fournissent également un ensemble de fonctions de calcul sur la plateforme parallèle CUDA, pour les opérations principales des solveurs itératifs ainsi que du préconditionnement.

Le solveur itératif de type Gradient Conjugué se base sur deux opérations essentielles :

- la multiplication d'une matrice creuse avec un vecteur dense ce qui peut exploiter le package Cusparse, CUSP et cuBlas,

- l'opération de calcul entre les vecteurs denses et les scalaires comme le produit scalaire de deux vecteurs denses, l'addition de deux vecteurs avec le package cuBlas, CUSP

Le préconditionnement propose plusieurs opérations différentes selon son type.

- le préconditionneur ILU demande un solveur triangulaires ce qui peut se trouver dans le package cuSparse.

- le préconditionneur AINV, AMG et d'autres préconditionneurs se trouvent dans le package CUSP.

Tous les packages sont présents au sein de CUDA runtime pour assurer la productivité du code. Pour utiliser ces packages, le programmeur doit allouer les matrices et les vecteurs nécessaires en mémoire du GPU, appeler la fonction désirée dans le package et enfin, recopier le résultat retourné en mémoire de CPU. Les packages fournissent également les méthodes pour créer et récupérer les données sur GPU.

III.3.1. CuBlas, CuSparse et CUSP

Le package cuBlas fournit des portages du code BLAS (« Basic Linear Algebra Subprogram » en anglais) sur la plateforme de CUDA runtime. Les fonctions s'exécutent automatiquement sur un GPU seul mais pas en multi-GPU. Le package fournit les fonctions à trois niveaux :

- Niveau 1 de BLAS qui effectuent des opérations entre des valeurs scalaires et des vecteurs,

- Niveau 2 de BLAS qui exécutent des opérations entre les matrices et des vecteurs,

- Niveau 3 de BLAS qui effectuent des opérations de matrices à matrices.

Le package cuSparse contient un ensemble de sous-routines d'algèbre linéaire utilisés pour le traitement des matrices creuses qui fonctionne sur la plateforme CUDA. Les fonctions du package peuvent être classées en quatre catégories:

- Niveau 1: opérations entre un vecteur au format creux et un vecteur au format dense,

- Niveau 2: opérations entre une matrice en format creux et un vecteur au format dense,

- Niveau 3: opérations entre une matrice en format creux et une matrice dense,

- Conversion: opérations qui permettent la conversion entre les différents formats de matrice.

En JAVA, il est possible d'utiliser les fonctions de cuBlas et cuSparse à travers leur portage JCuBlas et JCusparse.

Le package CUSP est une bibliothèque pour les calculs d'algèbre linéaires creux et de graphe à base de Thrust. CUSP fournit une interface flexible de haut niveau pour la manipulation de matrices creuses et la résolution de systèmes linéaires creux.

III.3.2. Exécution

CUSP

Le package CUSP fournit une solution très flexible et rapide grâce son code propre en C++ qui est construit à base de runtime de CUDA et Thrust. L'utilisateur peut choisir des solveurs et préconditionneurs dans un ensemble disponible de CUSP, y compris le solveur de Gradient Conjugué (CG). Par exemple, la fonction permettant de lancer le solveur CG avec le préconditionneur est la suivante :

```
void cusp::krylov::cg
(   LinearOperator & A,
    Vector & x,
    Vector & b,
    Monitor & monitor,
    Preconditioner & M
)
```

avec

`A` est la matrice symétrique, définie positive du système,
`x` est la solution approchée,
`b` est le vecteur de source,
`monitor` est le vecteur qui suit la convergence,
`M` est le préconditionneur.

L'exemple du code CUSP en C++ pour l'emploi du solveur CG avec le préconditionnement diagonal (Jacobi) est présenté à la Figure III.18 : Toutes les données résident en mémoire globale. La matrice A est stockée au format CSR avec NNZ le nombre de termes non nuls de la matrice. CUSP fonctionne en coopération avec Thrust en utilisant une structure spécifique pour gérer les données.

```
typedef typename cusp::array1d_view< thrust::device_ptr<int>   > IndexView;
typedef typename cusp::array1d_view< thrust::device_ptr<float> > ValueView;

typedef cusp::csr_matrix_view<IndexView, IndexView, ValueView> MatrixView;

// solve the system Ax = b with CUSP, the matrix A in CSR format, all data on GPU mem
int solveCUSP(int n, int nnz, int* dev_csr_rowptr, int* dev_csr_col, float* dev_csr_val,
      float* dev_b, float* dev_x)
{
   // *NOTE* raw pointers must be wrapped with thrust::device_ptr!
   thrust::device_ptr<int> dev_csr_rowptr(csr_rowptr);
   thrust::device_ptr<int> dev_csr_col(csr_col);
   thrust::device_ptr<float> dev_csr_val(csr_val);
   thrust::device_ptr<float> dev_b(rhs);
   thrust::device_ptr<float> dev_x(sol);

   // use array1d_view to wrap the individual arrays
   IndexView csrRowPtrA(dev_csr_rowptr, dev_csr_rowptr + n + 1);
   IndexView csrColIndA(dev_csr_col, dev_csr_col + nnz);
   ValueView csrValA(dev_csr_val, dev_csr_val + nnz);
   ValueView x(dev_x, dev_x + n);
   ValueView b(dev_b, dev_b + n);

   // combine the three array1d_views into a coo_matrix_view
   MatrixView A(n, n, nnz, csrRowPtrA, csrColIndA, csrValA);

   // set stopping criteria:
   // iteration_limit = 1000; relative_tolerance = 1e-6; relative_tolerance = 1e-6
   cusp::verbose_monitor<float> monitor(b, 1000, 1e-6, 1e-6);

   std::cout << "\nSolving" << std::endl;

   // timer
   cusp::detail::timer t;

   // setup preconditioner
   cusp::precond::diagonal<float, cusp::device_memory> M(A);

    // solve
    cusp::krylov::cg(A, x, b, monitor, M);

   std::cout << "\nSolving Time: " << t.milliseconds_elapsed() << std::endl;

   // report status
   report_status(monitor);

   return 0;
}
```

Figure III.18 CUSP lance le solveur CG avec un préconditionneur de Jacobi, la matrice est au format CSR

D'autres préconditionneurs comme AINV ou AMG peuvent être combinés avec le solveur CG. Le code est juste modifié à l'étape de génération du préconditionneur :

- pour le préconditionneur AINV :

```
// setup preconditioner
cusp::precond::scaled_bridson_ainv<float, cusp::device_memory> M(A, 0.1);
```

- pour le préconditionneur AMG :

```
// setup preconditioner
cusp::precond::aggregation::smoothed_aggregation<int, float, cusp::device_memory> M(A);
```

CuBlas et Cusparse

Contrairement à CUSP, CuBlas et Cusparse fournissent un ensemble de calculs d'algèbre linéaires pour les matrices et les vecteurs ainsi que les outils pour gérer les données. Donc, le programmeur doit générer son code propre pour les solveurs et les préconditionneurs.

```
/***** CG Code with preconditionner diagonal *****/
/* ASSUMPTIONS:
1. The CUSPARSE and CUBLAS libraries have been initialized.
2. The appropriate memory has been allocated and set to zero.
3. The matrix preconditionner Jacobi (inverse) have been
computed and are present in the device (GPU) memory. */

//1: compute initial residual r = f - A x0 (using initial guess in x)
cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
descrA, valA, csrRowPtrA, csrColIndA, x, 0.0, r);
cublasDscal(n,-1.0, r, 1);
cublasDaxpy(n, 1.0, f, 1, r, 1);
nrmr0 = cublasDnrm2(n, r, 1);
//2: repeat until convergence (based on max. it. and relative residual)
for(i=0; i<maxit; i++){
        //3: Solve M z = r
        precond<<<...>>>(z, M, r, n);

        //4: \rho = r^{T} z
        rhop= rho;
        rho = cublasDdot(n, r, 1, z, 1);
        if(i == 0){
        //6: p = z
        cublasDcopy(n, z, 1, p, 1);
        }
        else{
        //8: \beta = rho_{i} / \rho_{i-1}
        beta= rho/rhop;
        //9: p = z + \beta p
        cublasDaxpy(n, beta, p, 1, z, 1);
        cublasDcopy(n, z, 1, p, 1);
        }
        //11: Compute q = A p (sparse matrix-vector multiplication)
        cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
        descrA, valA, csrRowPtrA, csrColIndA, p, 0.0, q);
        //12: \alpha = \rho_{i} / (p^{T} q)
        temp = cublasDdot(n, p, 1, q, 1);
        alpha= rho/temp;
        //13: x = x + \alpha p
        cublasDaxpy(n, alpha, p, 1, x, 1);
        //14: r = r - \alpha q
        cublasDaxpy(n,-alpha, q, 1, r, 1);
        //check for convergence
        nrmr = cublasDnrm2(n, r, 1);
        if(nrmr/nrmr0 < tol){
        break;
        }
}
```

Figure III.19 Exemple du code pour la résolution en solveur CG avec le préconditionneur de Jacobi par Cublas et Cusparse

Avec le préconditionnement diagonal:

```
/* Calc precondition diagonal y = M^-1 * x */
__global__ void
precond(  float* y,     // output
          float* M,     // diagonal M^-1
          float* x,     // input
          int    n)
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;

   if(i < n){
      y[i] = M[i] * x[i];
   }
}
```

La résolution avec le préconditionneur ILU peut se trouver dans [17].

III.4. Evaluation de performances

Dans cette section, nous effectuons des expériences pour évaluer la performance des kernels CUDA utilisés lors de l'assemblage et la résolution. Le problème considéré est magnétostatique linéaire en 2D dont la formulation (III.43) en potentiel vecteur est présentée dans la section III.1.1.

III.4.1. Géométrie

L'expérience modélise un noyau en acier placé dans un champ magnétique, montré en Figure III.20. Le noyau est en matériau ferromagnétique avec la perméabilité relative μr = 1000. La bobine est parcourue par un courant continu dont la densité vaut 1A/mm2.

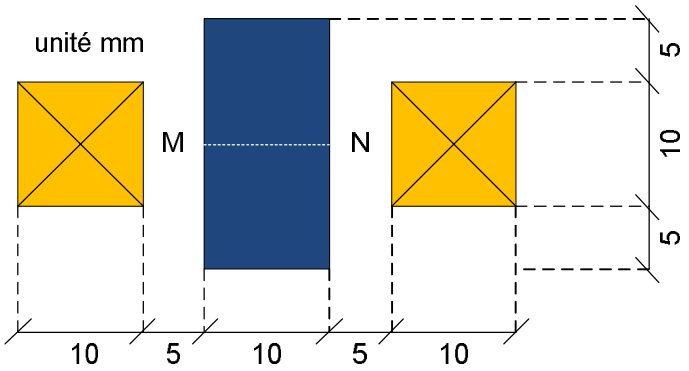


Figure III.20 Géométrie de la simulation en magnétostatique utilisée

III.4.2. Maillage et matrice

La géométrie du problème est maillée en éléments triangulaires, cf. Figure III.21. La densité d'éléments autour du noyau et des bobines est plus importante afin d'assurer la précision de la solution sur cette zone. Pour évaluer la performance de la résolution en fonction du nombre d'éléments, nous considérons le problème selon plusieurs maillages issus d'un maillage grossier. Les maillages plus fins sont obtenus par la subdivision d'un triangle initial en 4 sous-triangles. Le nombre d'éléments, de DOFs et la taille de la matrice sont reportés dans le Tableau III-2. Le « minNZ », « maxNZ », « moyen » sont le nombre minimal, maximal et moyen des termes non nuls par DOFs respectivement de la matrice des coefficients.

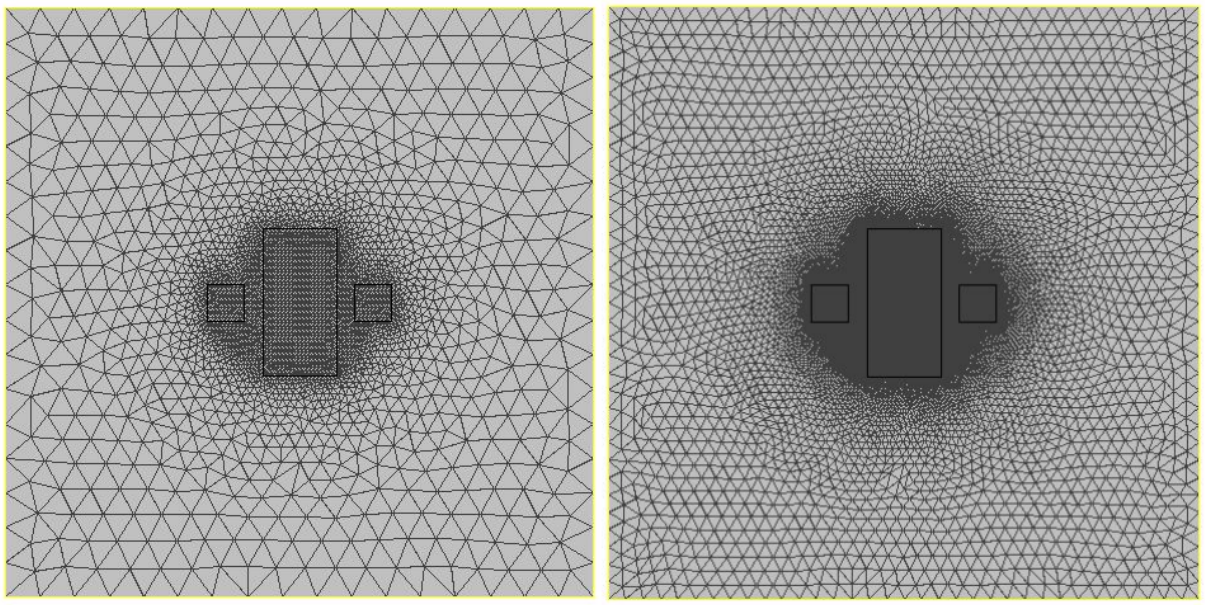


Figure III.21 Maillage du domaine : maillage initial (gauche) et plus fin (droit)

Le problème est étudié pour des éléments du 1er ordre et 2ème ordre. Du point de vue de la MEF, l'intégration est différente et l'élément du 2ème ordre demande un calcul plus lourd. Les maillages 1, 2 et 3 correspondent aux éléments du 1er ordre et les maillages 4, 5 et 6 aux éléments du 2ème ordre.

Tableau III-2 Informations des maillages utilisés selon le nombre d'éléments, le nombre de DOF inconnus et le nombre des termes non nuls de la matrice

Maillage	Nb nœuds	Nb éléments	Nb DOFs	NNZ	minNZ	maxNZ	moyen
1	3027	5984	2959	20563	3	9	7
2	12037	23936	11901	83021	4	9	7
3	48009	95744	47737	333601	4	9	7
4	12037	5984	11901	135971	4	25	11
5	48009	23936	47737	547235	4	25	11
6	189201	95744	188657	2166109	4	25	11

III.4.3. Validation du résultat avec le logiciel Flux2D

Avant d'estimer la performance des calculs parallèle, on doit s'assurer que le résultat est exact. Une solution de référence est obtenue par le logiciel Flux2D.

La Figure III.22 montre notre solution sur une carte du potentiel vecteur A et du champ magnétique B.

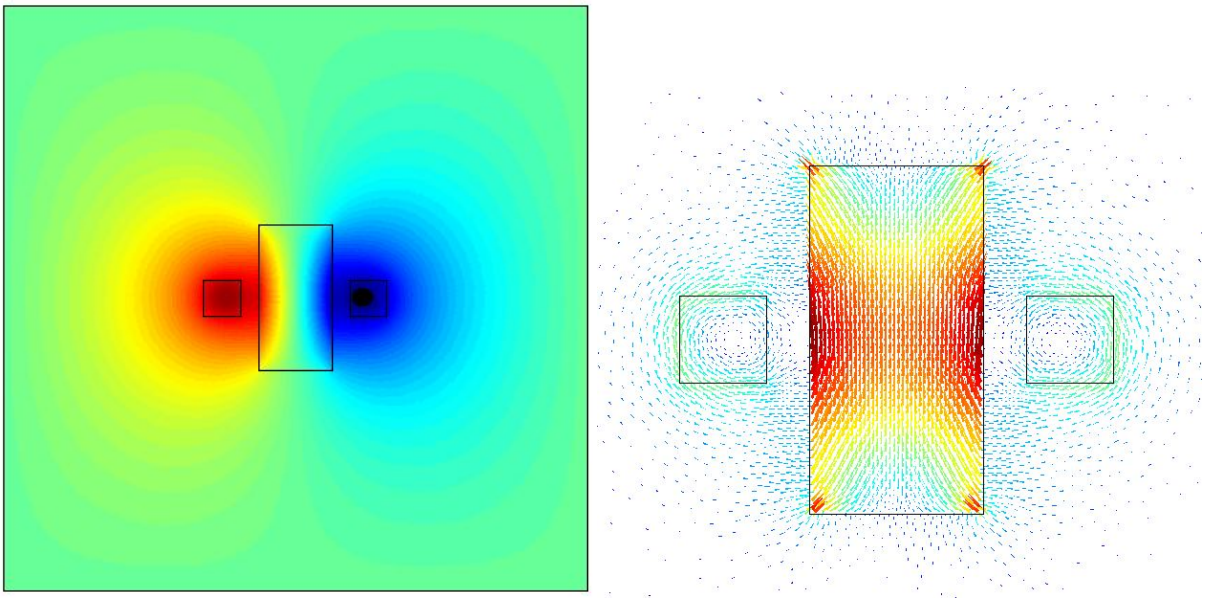


Figure III.22 Distribution du potentiel vecteur A (iso-valeur) et le vecteur du champ magnétique B

La valeur du champ magnétique **B** sur le chemin MN est comparée entre la solution de Flux2D et notre solution. Les écarts de notre calcul par rapport à Flux2D sont calculés par :

$$eps_i = \left| \frac{\left\| B_i^{ref} \right\| - \left\| B_i \right\|}{\left\| B_i^{ref} \right\|} \right| \times 100\%$$

Le résultat montre un écart moins de 0,3%, cf. la Figure III.23.

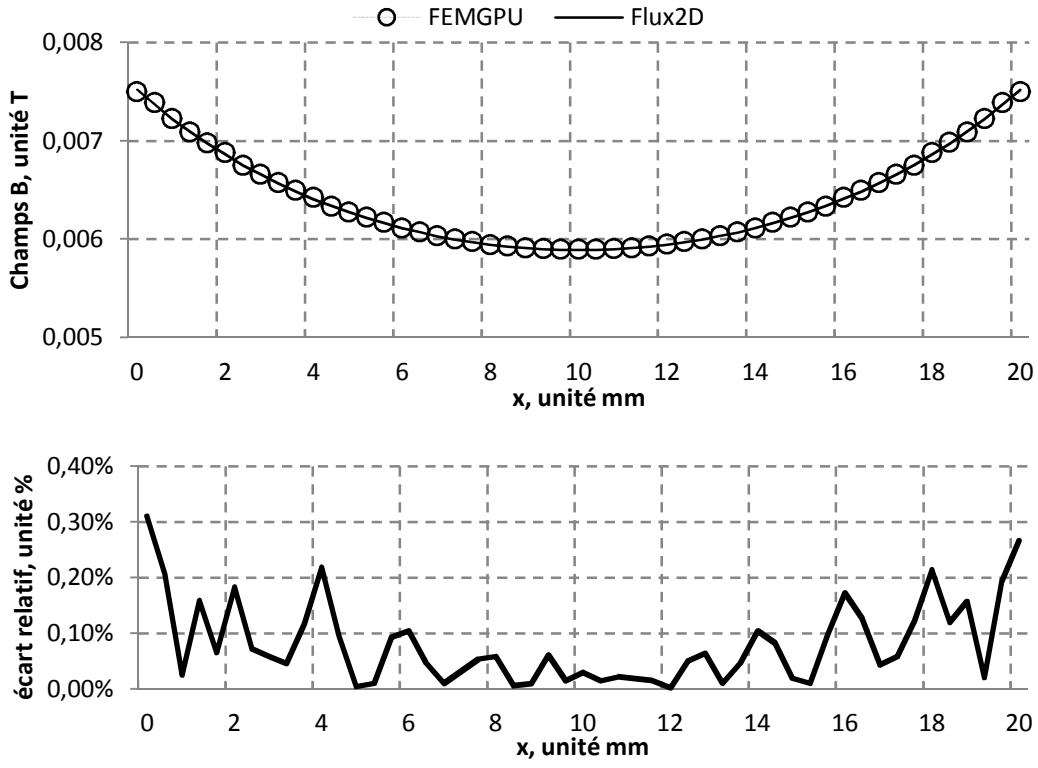


Figure III.23 Valeur du champ magnétique B et l'écart sur le chemin MN

III.4.4. Matériel informatique

Toutes les expériences sont exécutées sur une plateforme équipée d'un processeur Intel Xeon 2,67 GHz, RAM 4 Go. Le calcul sur CPU utilise 4 cœurs. Le GPU est une carte NVIDIA Quadro K5200, 8 GB RAM, de capacité 3.5, avec une puissance de calcul de 3073 GFLOPS et une bande passante mémoire de 192GB/s. Elle dispose de 2304 cœurs. Le code MEF original est développé dans un environnement JAVA v7.4 et exécuté sur CPU en mode processeur unique. Les kernels parallèles sont codés en C/C++. Ils sont exécutés sur GPU mais contrôlés par CPU en invoquant les fonctions en JAVA à travers le package jCuda et JNI. Le calcul du kernel est effectué en simple précision arithmétique pour optimiser les performances du GPU. Nous constatons alors une différence sur le résultat à cause de cette moindre précision numérique. La norme relative de la différence du résultat (sur la matrice et le vecteur) est d'environ 10^{-6}.

III.4.5. Intégration et assemblage parallèle sur GPU

Dans cette section, nous analysons le temps de calcul d'intégration numérique et d'assemblage entre les approches parallèles déjà connues dans la littérature et notre méthode d'assemblage :

- Assemblage séquentiel (CPU) comme la référence.
- Assemblage parallèle par le coloriage des éléments, cf. Section III.2.3
- Assemblage parallèle par la décomposition en pièce, cf. Section III.2.4
- Assemblage parallèle par l'approche de tri, cf. Section III.2.5
- Assemblage parallèle par l'utilisation de fonctions atomiques CUDA. La procédure d'assemblage parallèle se base sur la fonction atomicAdd() de CUDA pour éviter la condition de concurrence. Le calcul numérique et l'assemblage est combiné dans un kernel commun.

Nous utilisons un chronomètre de CPU pour enregistrer les temps de la version CPU et un chronomètre de GPU avec la synchronisation CPU-GPU pour compter le temps des kernels exécutés sur GPU.

Le temps d'intégration et d'assemblage des approches sont rassemblés dans le Tableau III-3. Il est à noter que le temps de coloriage ou de décomposition du maillage par pièce ne sont pas intégrés dans cette synthèse. La durée de ces procédures est reportée dans le Tableau III-4.

Tableau III-3 Temps de calcul numérique et d'assemblage et l'accélération de calcul parallèle sur GPU par rapport le calcul séquentiel sur CPU

Maillage	CPU	Atomique	Coloriage	Pièce	Tri
1	125 (1x)	40 (3,1x)	33 (3,8x)	17 (7,4x)	14 (8,9x)
2	525 (1x)	83 (6,3x)	69 (7,6x)	45 (11,7x)	34 (15,4x)
3	2046 (1x)	305 (6,7x)	256 (8,0x)	173 (11,8x)	117 (17,5x)
4	174 (1x)	131 (1,3x)	114 (1,5x)	51 (3,4x)	28 (6,2)
5	1008 (1x)	295 (3,4x)	219 (4,6x)	131 (7,7x)	76 (13,3x)
6	5282 (1x)	454 (11,6x)	413 (12,8x)	319 (16,6x)	153 (34,5x)

Tableau III-4 Temps de calcul de coloriage et de décomposition de maillage, unité ms

Maillage	coloriage	décomposition
1	30	18
2	115	67
3	591	315
4	33	64
5	143	304
6	700	1260

En synthèse, le temps d'assemblage augmente quasi linéairement en fonction de la taille du problème mesuré au nombre de nœuds et d'éléments. Le résultat du timing montre une nette accélération du calcul parallèle par rapport au calcul séquentiel. L'accélération est différente selon les approches. Parmi les approches parallèles, notre approche de tri donne la meilleure accélération, suivie par l'approche par pièce et puis l'approche par coloriage et enfin par l'approche d'atomique. L'accélération augmente avec la taille du problème. Pour le petit problème qui ne comporte que 6000 éléments, l'accélération atteint un peu moins de 10x. Ce dernier est de 35x pour le cas de 100 000 éléments.

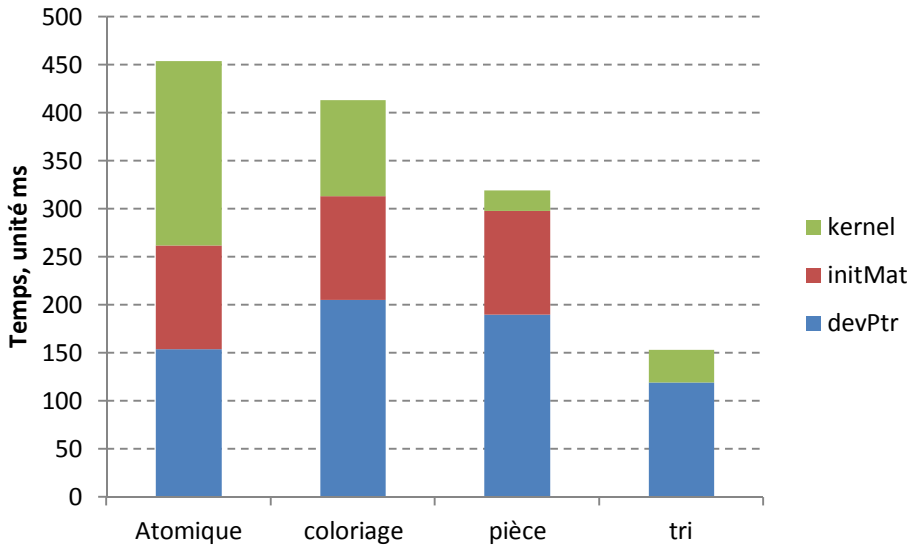


Figure III.24 Comparaison du temps d'exécution du kernel selon les approches

La Figure III.24 montre la répartition du temps entre les trois étapes selon les approches parallèles proposées dans le cas du maillage 6, cf. Tableau III-2. Le « kernel » est le temps consacré à l'intégration et l'assemblage par le kernel CUDA. Le « initMat » est le temps pour initialiser les tables d'indice de la matrice de rigidité. Le « devPtr » est le temps pour initialiser la mémoire GPU, transférer les données entre CPU et GPU et la mise à jour des données. Le ratio de temps varie selon l'approche considérée.

- Pour l'approche « atomique », le temps d'exécution est quasiment aussi long que le temps de préparation des données. Le temps de préparation est dédié à la

construction topologique de la matrice d'assemblage et à la manipulation des données.

- Pour l'approche par « coloriage », le temps « initMat » est similaire mais le temps de chargement augmente. Par contre, le temps d'exécution du kernel CUDA est meilleur que pour l'approche « atomique ».
- L'approche d'assemblage par pièce donne le meilleur temps d'exécution du kernel CUDA. Cela confirme que l'usage de la mémoire partagée procure une efficacité considérable par rapport à la mémoire globale qui elle est utilisée dans l'approche par coloriage et l'approche de tri. Cependant, les temps de préparation deviennent prépondérants et la performance totale de la méthode est restreinte de ce fait.
- Enfin, l'approche d'assemblage par tri donne une meilleure performance en termes de temps total. Le temps « initMat » est nul car la topologie de la matrice est construite dans la phase de la réduction de cette méthode. Le temps « kernel » est la somme de trois calculs : kernel d'intégration numérique, le tri des termes non nuls et la réduction des termes non nuls.

La répartition du temps de calculs selon les maillages est montrée dans la Figure III.25. Nous constatons que proportionnellement le temps « initMat » et «devPtr » augmente en fonction de taille de maillage, tandis que le temps du kernel diminue. En conséquence, plus le maillage est grand, plus le coût pour initialiser la matrice est important. Cela ralentit la performance des approches d'assemblage « atomique », par coloriage et par pièce mais est peu significatif pour notre méthode d'assemblage par tri.

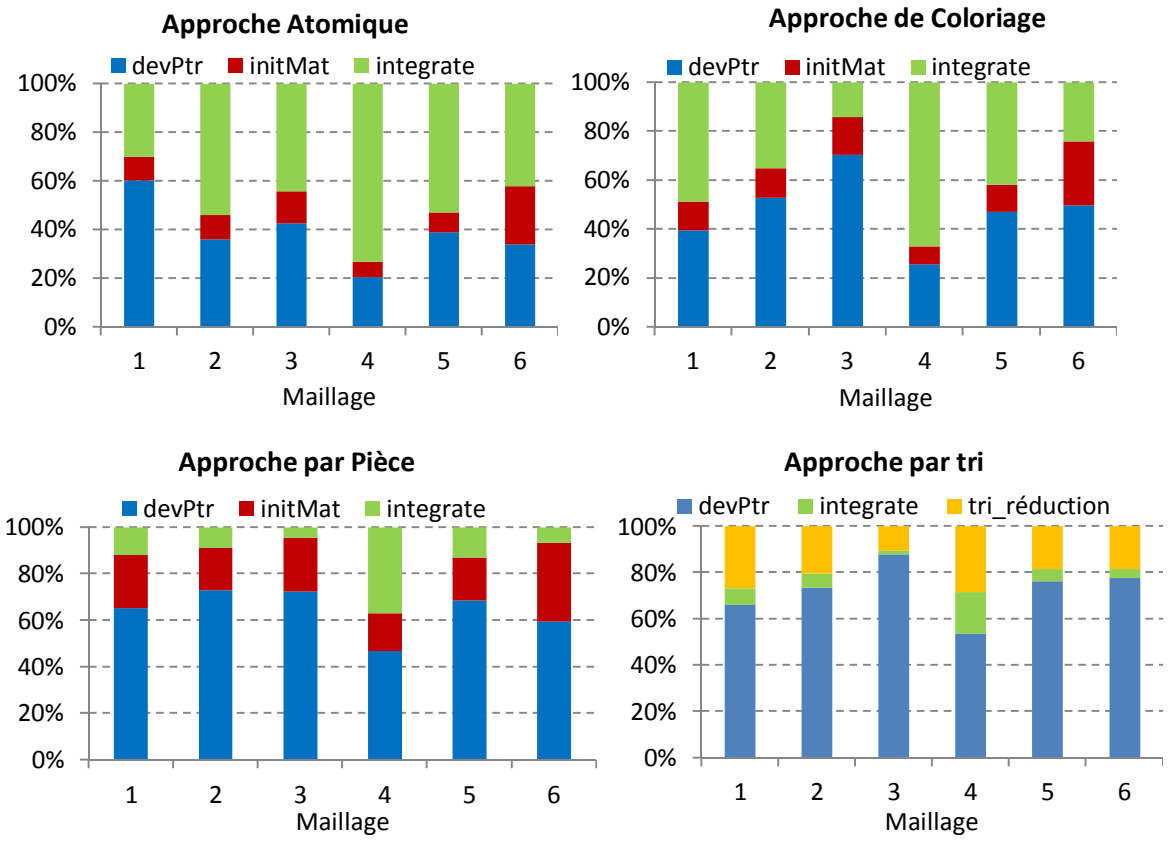


Figure III.25 Répartition de temps des processus différents

III.4.6. Résolution parallèle sur GPU

Dans cette évaluation, nous considérons le problème magnétostatique 2D sur le cas du maillage 6, cf. Tableau III-2 . Nous mettons en place le solveur CG (Conjugate Gradient) avec des préconditionneurs différents et sur deux plateformes différentes: CPU et GPU pour évaluer la performance de la procédure de résolution selon le choix de solveur et de préconditionneur.

La convergence du solveur CG est déterminée en fonction de la norme des résidus de la solution. Le critère d'arrêt est 10^{-8}. Nous évaluons la performance des préconditionneurs ci-dessous :

- CG avec le préconditionneur ILU(0) exécuté sur CPU

- CG avec le préconditionneur Jacobi exécuté sur GPU

- CG avec le préconditionneur AINV du package CUSP, exécuté sur GPU

- CG avec le préconditionneur AMG du package CUSP, exécuté sur GPU

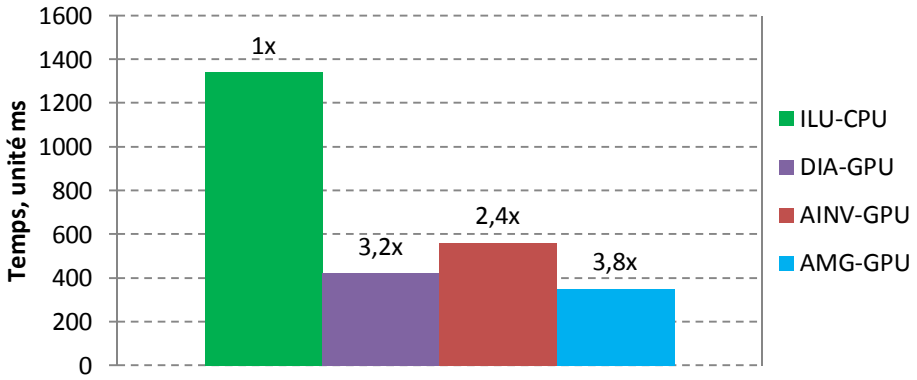


Figure III.26 Comparaison du temps du solveur CG avec les préconditionneurs différents

La Figure III.26 montre les temps de calcul du solveur CG avec des préconditionneurs différents en milliseconde. Le temps de résolution du solveur CG exécuté sur CPU est plus long que les autres solveurs sur GPU, quel que soit le type de préconditionneur. Le solveur CG exécuté sur GPU est en moyenne de 3 à 4 fois plus rapide que sur CPU, ce qui est malgré tout sensiblement moins efficace que les algorithmes développés sur mesure pour le maillage ou l'intégration et l'assemblage dans les parties précédentes où les accélérations avaient un ordre de grandeur de plus. Le solveur CG préconditionné par AMG donne le temps le plus court.

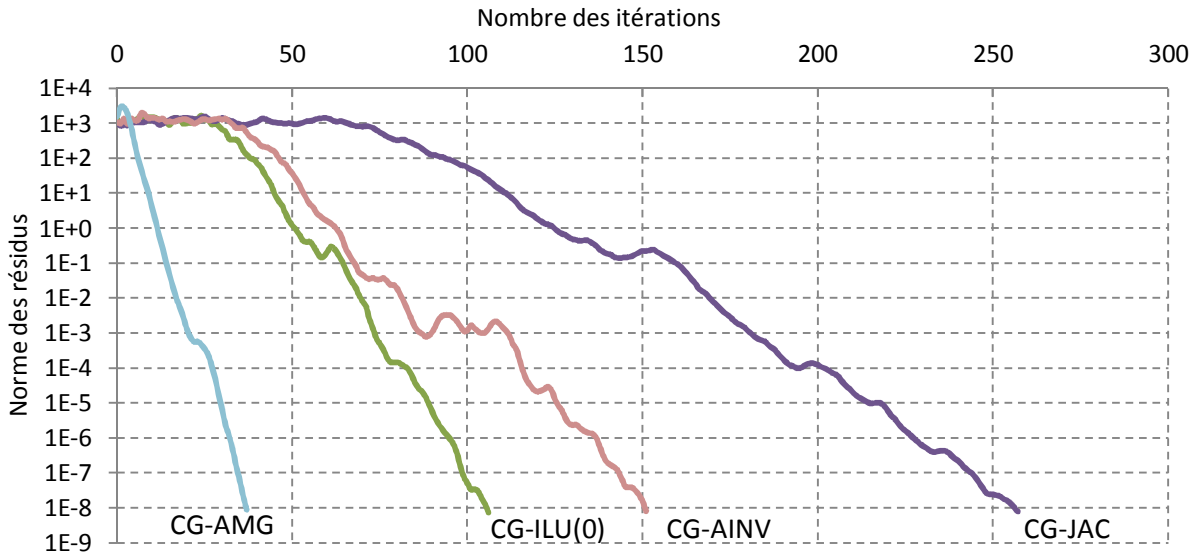


Figure III.27 Evolution du taux de convergence du solveur CG préconditionné en fonction de l'itération

Le taux de convergence du solveur dépend du type de préconditionneur utilisé, comme le montre la Figure III.27. Le nombre d'itérations est environ de 250 avec le préconditionneur diagonal (Jacobi) et environ de 100 à 150 avec le préconditionneur ILU et AINV et à peine quelques dizaines avec le préconditionneur AMG. Donc, du point de vue du taux de convergence, le préconditionneur AMG est le plus robuste. De plus, nous constatons une descente très régulière du résidu avec le préconditionneur AMG tandis que celui d'autres préconditionneurs ont des décroissances qui se réduisent progressivement.

III.5. Conclusion

Dans ce chapitre, nous avons présenté des algorithmes de calcul parallèle pour la méthode des éléments finis appliquée au domaine de la magnétostatique. Nous avons introduit deux formulations bien connues pour analyser le problème magnétostatique, il s'agit du potentiel scalaire et du potentiel vecteur en 2D ainsi qu'en 3D. Dans les expériences et les algorithmes étudiés, nous utilisons la formulation 2D en potentiel vecteur A et une discrétisation par fonctions d'interpolation nodales à titre d'illustration. La transposition des algorithmes aux problèmes 3D est généralement aisée, seules les limites de mémoire peuvent venir compliquer les choses mais celles-ci ont été explicitées.

Nous avons présenté les deux approches les plus utilisées dans la littérature pour l'intégration numérique ainsi que l'assemblage parallèle sur la plateforme CUDA. En général, les difficultés que rencontrent ces démarches se concentrent sur deux aspects : l'optimisation de l'usage de la mémoire, tant en taille qu'en rapidité d'accès et la concurrence entre threads. Aucune de ces deux approches ne parvient totalement à s'affranchir de l'une ou l'autre de ces contraintes sans introduire une lourde pénalité de prétraitement des données. Nous avons également introduit une nouvelle approche, appelée approche globale par tri.

- L'approche par coloriage des éléments est le choix le plus utilisé mais sa performance est limitée par rapport à d'autres méthodes. Les facteurs limitant résident dans l'accès non optimisé aux données, en particulier les données des nœuds ainsi que la mise à jour de la matrice et du

vecteur dans la phase d'assemblage. De plus, le temps de coloriage dépend grandement de la taille du maillage. Le principal avantage de cette approche est que l'algorithme est simple et stable.

- L'approche d'assemblage par pièce se base sur la décomposition du problème en plusieurs « pièces » disjointes dont la taille doit être adaptée à la mémoire partagée du GPU. La mémoire partagée est utilisée pour non seulement optimiser l'accès aux données mais aussi effectuer l'assemblage parallèle. Par contre, l'interface entre les pièces demande un traitement spécifique qui se traduit par un assemblage de ces termes non nuls en mémoire globale. De plus, le nombre des éléments est variable selon la pièce et cela peut ralentir la performance du kernel. L'inconvénient de cette méthode est la taille limitée de la mémoire partagée du GPU qui ne s'adapte qu'aux discrétisations dont le nombre de degrés de liberté par élément est faible. Par ailleurs, l'accès concurrent aux données n'est résolu qu'au prix de l'utilisation de fonctions « atomicAdd » du CUDA. La performance de cette méthode est la meilleure du point de vue de l'exécution des kernels, même si elle demande un prétraitement assez lourd et une décomposition du maillage.

- L'approche globale par tri consiste à utiliser la mémoire globale en séparant les deux étapes, intégration numérique et assemblage. Le calcul l'intégration numérique est exécuté en parallèle avec chaque élément attribué à un thread et le résultat est stocké en mémoire globale. Puis, les termes non nuls sont assemblés dans la matrice en parallèle avec chaque DOF attribué à un thread. La méthode demande un minimum de prétraitement par rapport d'autres approches. Notre méthode donne la meilleure performance. Elle peut s'appliquer à des discrétisations très diverses, tant en nombre d'éléments qu'en ordre.

La parallélisation de la résolution a reposé sur des outils et des packages natifs de CUDA avec un solveur CG et différents préconditionneurs. Le package CuSparse contient des outils pour définir et convertir le stockage de la matrice creuse. Le package Cublas fournit des moyens de calculs pour une matrice et un vecteur dense. Le package CUSP fournit un ensemble de solveur et de préconditionneurs. L'utilisation du solveur CG du package CUSP avec ses propres préconditionneurs (Jacobi, AINV et surtout AMG) s'est avéré être le meilleur choix, même si l'accélération est en deçà des attentes.

CHAPITRE IV

Exploitation – Validation

SOMMAIRE

IV.1. Exploitation Post-traitement de FEM

Dans la phase d'exploitation, il existe plusieurs façons de présenter les résultats et en particulier les distributions de quantité sur le domaine. Les quantités vectorielles comme les champs ou inductions magnétiques (B, H) sont exprimées par des flèches donnant direction et amplitude, tandis que les quantités scalaires, comme les modules de l'induction ou le potentiel scalaire magnétique sont exprimés par les couleurs ou des iso-valeurs. Comme l'illustre la Figure IV.1, la valeur d'une quantité sur le domaine est représentée par un ensemble des courbes (iso-lignes) ou un ensemble des couleurs (iso-valeurs), chaque courbe ou couleur correspond à une gamme de valeurs de cette quantité.

Comme pour l'ensemble de la méthode des éléments finis, le post-traitement repose également sur une interpolation qui se fait à l'échelle de l'élément. Le calcul de post-traitement dépend linéairement de deux paramètres : la subdivision des éléments en plusieurs niveaux et l'échelle des valeurs affichées. La Figure IV.1 montre les lignes iso-valeur correspondant à 21 valeurs du potentiel vecteur A sur le domaine, et la distribution de la couleur sur 64 sous-triangles par élément. Le post-traitement demande un calcul lourd et coûteux dans le cas où le nombre d'éléments est élevé et le besoin d'affichage en iso-valeurs est précis. La performance de post-traitement peut être grandement améliorée par un calcul parallèle sur GPU.

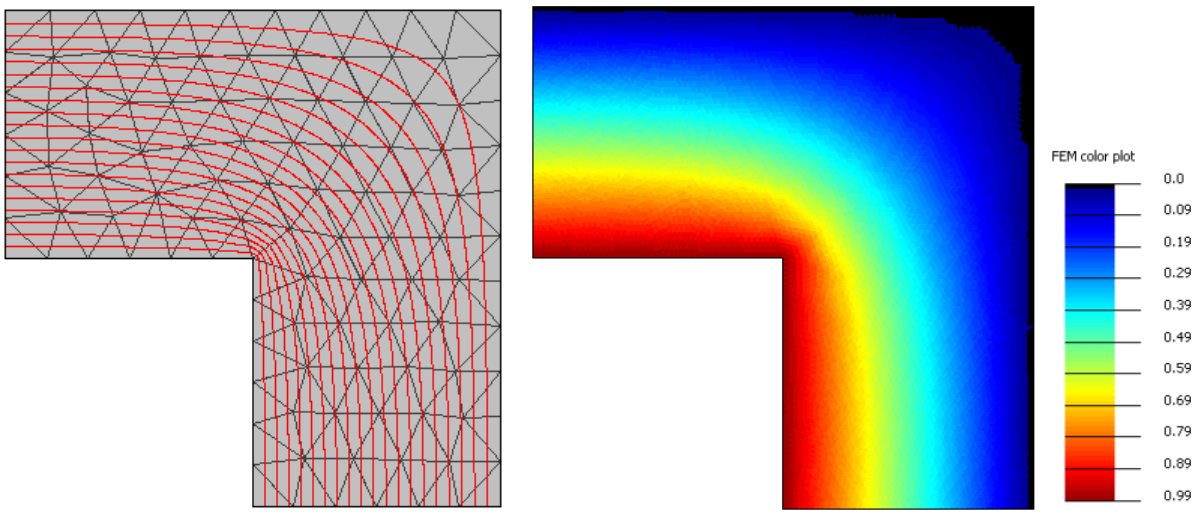


Figure IV.1 Illustration de l'affichage des iso-valeurs du potentiel A en 2D

Clarification du problème

Pour illustrer le calcul des iso-valeurs, nous nous baserons sur un élément triangulaire. La construction de la courbe iso-valeur est un problème inverse : rechercher tous les points pour lesquels la valeur de la quantité a une valeur connue :

$$\begin{cases} \varphi(r) = \sum_{i=1}^{N_i} w_i(r)\,\varphi_i \\ recherche \;\; r \subset \Omega : \varphi(r) = v \end{cases} \qquad \text{(IV.1)}$$

Avec w_i la fonction d'interpolation nodale, ψ_i la valeur de la quantité au nœud correspondant, N_i le nombre des nœuds du domaine.

Chaque iso-valeur se présente sous forme d'une courbe continue qui parcourt les éléments. La détermination exacte de chaque point sur la courbe est couteuse si, comme c'est possible avec des éléments du second ordre par exemple, on souhaite calculer sa forme courbe. En conséquence, la courbe est approchée par des segments de ligne droite au niveau de l'élément comme illustré par la Figure IV.2. Au lieu de devoir déterminer tous les points sur la courbe dans l'élément, il faut seulement déterminer deux points d'intersection de la courbe avec les arêtes de l'élément. Pour lisser la courbe, l'élément est divisé récursivement en plusieurs niveaux et la courbe est constituée d'un ensemble de segments.

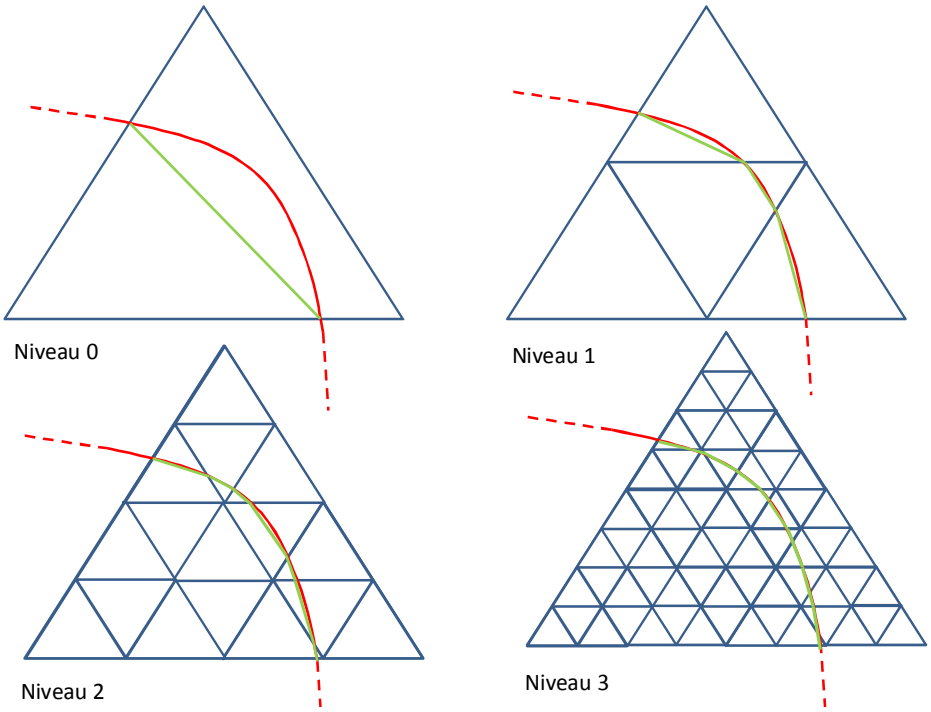


Figure IV.2 Lissage de la courbe d'iso-valeur selon le niveau de décomposition de l'élément

La détermination des deux points d'intersection entre la courbe iso-valeur et l'arête du triangle devient un système non linéaire:

$$\begin{cases} \varphi(x,y) = \sum_{i=1}^{N_i^e} w_i(x,y)\ \varphi_i \quad \text{sur } \Omega^e \\ recherche\ \ r=(x,y)\subset a^e : \varphi(x,y)=v \end{cases} \qquad \text{(IV.2)}$$

avec

a^e l'arête de l'élément triangulaire

N^e_i le nombre des nœuds d'interpolation élémentaire

La résolution de l'équation non-linéaire (IV.2) se base sur un solveur itératif simple, comme la méthode de Newton-Raphson.

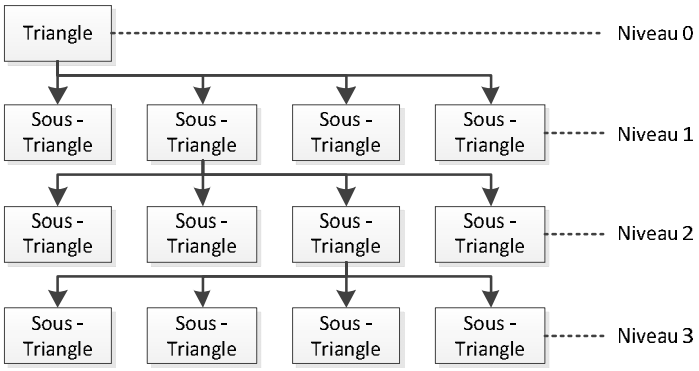


Figure IV.3 Décomposition du triangle assimilée à une structure de quadtree

La construction de la courbe d'iso-valeurs est un calcul récursif car il divise le triangle successivement en 4 sous-triangles. La subdivision est effectuée jusqu'à un niveau prédéfini, noté *k*. Ainsi chaque élément triangulaire est divisé en 4^k sous-triangles. La Figure IV.3 montre une décomposition en 3 niveaux (k=3, $4^3 = 64$ sous-triangles/triangles) dont la structure des sous-triangles est similaire avec celui d'un Quadtree. La courbe est déterminée à partir des sous-triangles de niveau *k*.

Le post-traitement des iso-valeurs suit le pseudocode suivant :

0. définir *k* le niveau de décomposition de triangle, *vp* iso-valeur de la quantité
1. pour chaque élément triangulaire e = 1 à N^e
2. si $\min(\varphi_i) \leq vp \leq \max(\varphi_i)\ \forall i = 1 \dots N_i^e$, iso-lignes est parcouru l'élément.
   si non, passer à autre élément.
3. diviser récursivement e par *k* niveaux,
4. pour chaque sous-triangle de niveau k :
5. si $\min(\varphi_i) \leq vp \leq \max(\varphi_i)\ \forall i = 1 \dots N_i^{es}$, résoudre l'équation (IV.2)
   si non, passer à autre sous-triangle
6. fin de tous sous-triangles
7. fin de tous éléments triangulaires

Parallélisation du calcul des iso-valeurs sur GPU

Pour un calcul parallèle sur GPU, l'usage de fonctions récursives est limité à certains GPU qui acceptent l'invocation dynamique des kernels (ce qui correspond à la capacité de calcul 3.x et plus). Une autre approche permet d'éviter la fonction récursive et s'adapte bien au mode de calcul sur GPU. Au lieu de travailler par élément, on associe chaque sous-triangle de niveau *k* à un thread. Le kernel s'effectue selon les trois étapes suivantes :

Etape 1 : charger les données de l'élément correspondant au sous-triangle dans la mémoire partagée. Les données se composent de la connectivité des nœuds, des coordonnées des nœuds et de la valeur des DOFs nodaux. Parce que chaque élément (triangle) est divisé en 4^k sous-triangles, alors les données de cet élément sont partagées par 4^k threads consécutifs.

Etape 2 : calculer les données géométriques de chaque sous-triangle à partir de son indice, les réserver sur la mémoire partagée

Etape 3 : construire la ligne iso-valeur dans le sous-triangle.

Le code du kernel CUDA est résumé par exemple comme cela:

```
__global__ void
kernel_build_polyline_Device( int nIsoval,        // number of isovalues
                     float* isoval,    // values of isolines
                     int deep,         // level of subdivision
                     ElemConnect* elem,  // table of elements connectivity
                     NodeCoord*  nodes,      // table of nodes coordinates
                     DofData* dofs,    // table of nodal dofs values
                     int nPolyline,    // number of polylines per isovalue
                     Polyline* isolines)  // output
{
   // size = nTriPerBloc * nnpe * (dim + 1) + blockDim * nnpe * dim
   extern __shared__ float sm[];

   // one thread one subtriangle
   int gid = threadIdx.x + blockIdx.x * blockDim.x;

   if(gid >= nPolylines)
       return;

   // number of sub triangles per triangle
   int nbr_sub_per_tri = 1<<2*deep;

   // global index of triangle corresponding
   int tri_gid = gid / nbr_sub_per_tri;

   // local index of sub in the list of sub
   int sub_tri_tid = gid % nbr_sub_per_tri;

   // number of triangles shared by all threads of bloc
   int nTriPerBloc = (blockDim.x - 1)/nbr_sub_per_tri + 1;

   // coordinates table of triangle, size = nTriPerBloc * nnpe * dim
   float* tri_coo = &sm[(threadIdx.x / nbr_sub_per_tri) * nNpE * dim];

   // values of nodal dofs in triangle// size = nTriPerBloc * nnpe
   float* dof_val = &sm[nTriPerBloc * nNpE * dim + (threadIdx.x / nbr_sub_per_tri) * nNpE];

   // load data of triangles (one thread one triangle)
   if(threadIdx.x % nbr_sub_per_tri == 0)
   {
       loadTriangleDevice(tri_gid, tri_coo, dof_val, elem, nodes, dofs);
   }
   __syncthreads();

   // load subtriangle, size = blockDim * nnpe * dim
   float* sub_tri_coo = &sm[nTriPerBloc * nNpE * (dim+1) + threadIdx.x * nNpE * dim];
   loadSubTriangleDevice(deep, sub_tri_tid, sub_tri_coo);

   for (int i = 0; i < nIsoval; i++)
   {
       float vp = isoval[i];
       if(nNpE == 3)
       {
          buildIsolineSubTriangle<3,2>(vp, tri_coo, dof_val, gid, sub_tri_coo, i, polyline);
       }
       else if(nNpE == 6)
       {
          buildIsolineSubTriangle<6,2>(vp, tri_coo, dof_val, gid, sub_tri_coo, i, polyline);
       }
   }
}
```

Evaluation de la performance du kernel

Nous évaluons la performance du kernel sur le problème de magnétostatique TEAM Workshop 25 (voir Figure IV.4), sur un maillage triangulaire initial de 4259 éléments. Le nombre d'éléments du maillage est modifiable par un paramètre de densité. Le potentiel A est présenté par 21 courbes d'iso-valeurs.

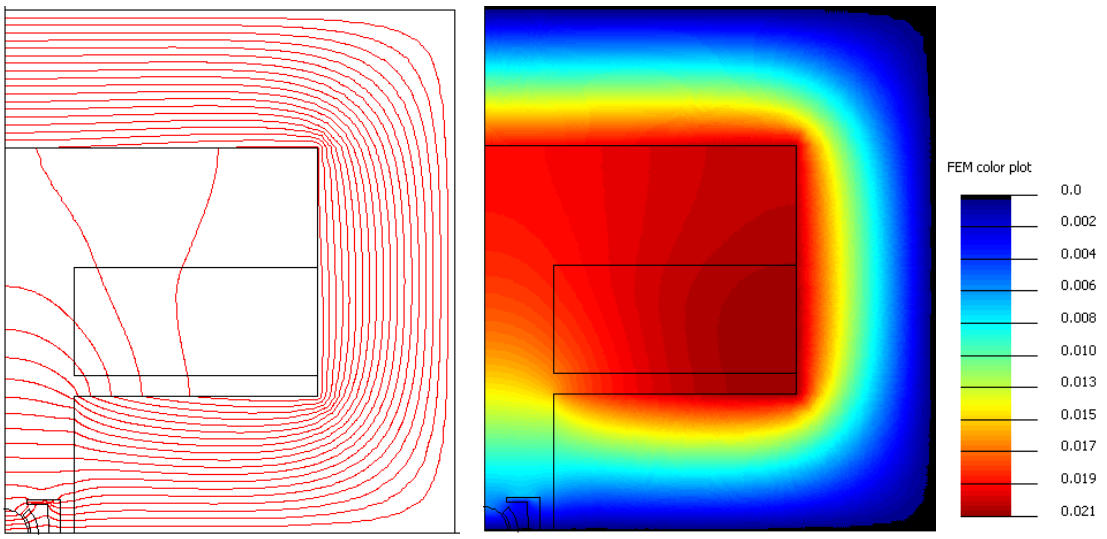


Figure IV.4 Cas-test pour le post-traitement des iso-valeurs du potentiel scalaire (TEAM Workshop 25)

La construction de chaque courbe d'iso-valeurs est exécutée sur CPU et GPU. Le Tableau IV-1 montre le temps d'exécution des deux implantations sur plusieurs maillages différents.

Tableau IV-1 Temps de la construction des iso-valeurs exécuté sur CPU et GPU pour le cas-test

Maillage (nœuds, triangles)	CPU temps d'exécution	GPU temps d'exécution	Accélération GPU/CPU
(2224, 4259)	504 ms	12,9 ms	39,07
(8373, 16374)	1005 ms	30,45 ms	33,00
(22761, 44905)	2335 ms	65,47 ms	35,66
(50870, 100814)	5091 ms	150,75 ms	33,77

Le résultat montre une accélération de 30x de l'exécution parallèle sur GPU par rapport l'exécution d'origine sur CPU.

IV.2. Mise en place générale

Dans les sections précédentes, on a présenté le portage sur GPU de la méthode des éléments finis pour chaque grande phase du calcul : maillage, intégration numérique et procédure d'assemblage, résolution et post-traitement. Le calcul parallèle sur GPU fournit des accélérations remarquables pour chaque étape individuelle mais il existe un point noir, c'est l'incontournable transfert de données entre la mémoire de l'hôte (CPU) et la mémoire de GPU et son coût associé.

De plus, on a souvent souligné que la structure de données sur l'hôte et sur GPU doit être assez différente : pour des raisons d'efficacité de programmation, les données sur l'hôte adoptent généralement une programmation orientée objet tandis que les données sur GPU demandent une approche vectorielle sous forme des vecteurs de données. La transformation entre deux types de données ralentit la performance de calcul parallèle sur GPU.

Un kernel CUDA demande le transfert des données de la mémoire de l'hôte vers la mémoire du GPU et évidemment, le résultat du kernel est à son tour transféré de la mémoire GPU vers la mémoire de l'hôte. Nous savons que chaque kernel CUDA fonctionne selon les cinq étapes ci-dessous :

- allouer la mémoire sur GPU pour les données

- transférer les données existantes sur la mémoire de l'hôte vers la mémoire GPU

- exécuter le kernel CUDA

- transférer le résultat actuel sur la mémoire GPU vers la mémoire de l'hôte

- libérer la mémoire GPU

Les étapes d'allocation et de récupération de la mémoire GPU deviennent inutiles si les kernels successifs utilisent des données communes. Il faut donc proposer un contexte commun et général qui gère toutes les données sur GPU.

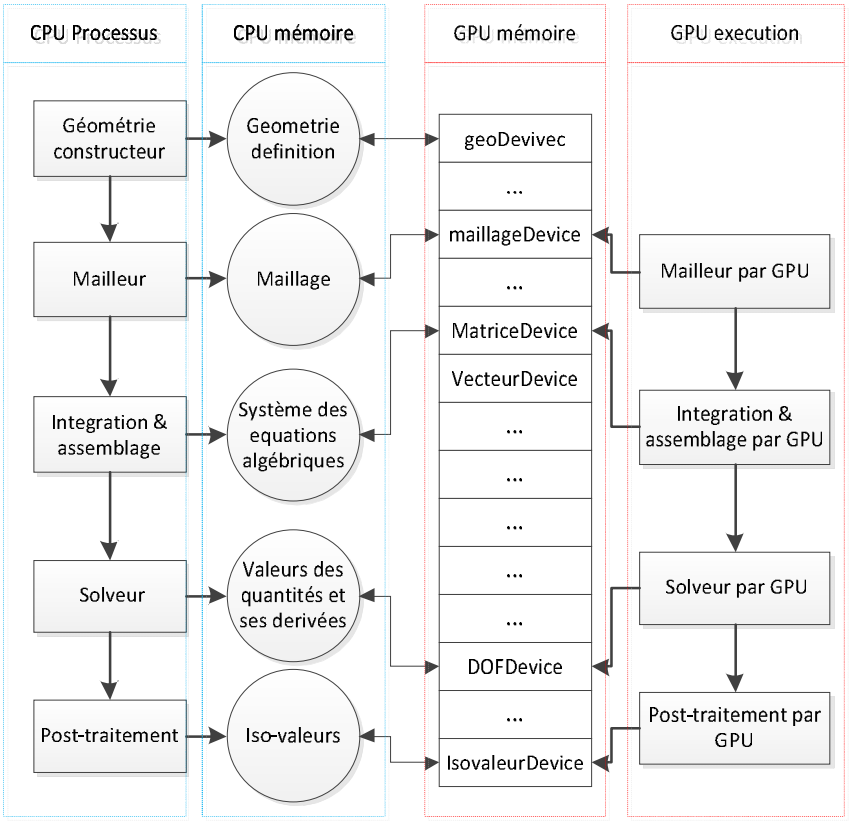


Figure IV.5 Cadre général des données de la MEF pour des exécutions parallèles sur GPU

Dans cette section, nous proposons une approche générale du calcul parallèle sur GPU pour la simulation par éléments finis. L'approche est globale à toutes les procédures de la MEF, sauf la construction de la géométrie et les procédures concernant la définition du problème par l'utilisateur. La Figure IV.5 montre notre approche dans le cas où toutes les procédures exigeantes en calcul sont exécutées sur GPU pour améliorer la performance de la simulation. Il s'agit des procédures de maillage, d'intégration numérique et d'assemblage du système des équations algébriques, de résolution et de post-traitement. Comme principe général, nous considérons le calcul parallèle sur GPU comme une seconde implantation des fonctions d'origine exécutée sur CPU. Cela signifie que l'utilisateur peut choisir l'exécution soit sur CPU, soit sur GPU, soit laisser le programme choisir automatiquement le lieu de l'exécution selon la taille du problème. En outre, les données du problème sont réservées et présentes sur les deux mémoires à la fois, CPU et GPU, sous des formats différents. Les données sur GPU sont gérées

par un contexte CUDA commun existant en parallèle avec celles du CPU. En conséquence, la mise à jour des données s'effectue de manière bidirectionnel CPU-GPU s'il y a des changements de données.

IV.3. Validation

IV.3.1. Description du cas test

Dans cette partie, nous appliquons le calcul parallèle sur GPU sur une simulation par éléments finis en magnétostatique. La simulation se base sur le problème TEAM Workshop 25 [72] dont la géométrie du problème est décrite par la Figure IV.6. S'agissant d'un problème d'optimisation, nous choisissions pour les paramètres géométriques variables les valeurs suivantes :

R1 = 7,2 mm, L2 = 15,3 mm, L3 = 30 mm, L4 = 11,5 mm

Le courant qui parcourt les bobines est choisi à 4253 AT. La propriété du matériau ferromagnétique est non-linéaire et caractérisée par des paires de valeurs B-H. Le comportement B-H du matériau est simplifié par une fonction arc tangente à deux coefficients dont la forme donnée à la Figure IV.7.

$$B(H) = C_1 \arctan(C_2 \mu_0 H) + \mu_0 H \text{ avec } C_1 = \frac{2B_s}{\Pi} \text{ et } C_2 = \frac{\mu - 1}{C_1} \qquad \text{(IV.3)}$$

avec $\mu = 1000$ et $B_s = 2(T)$.

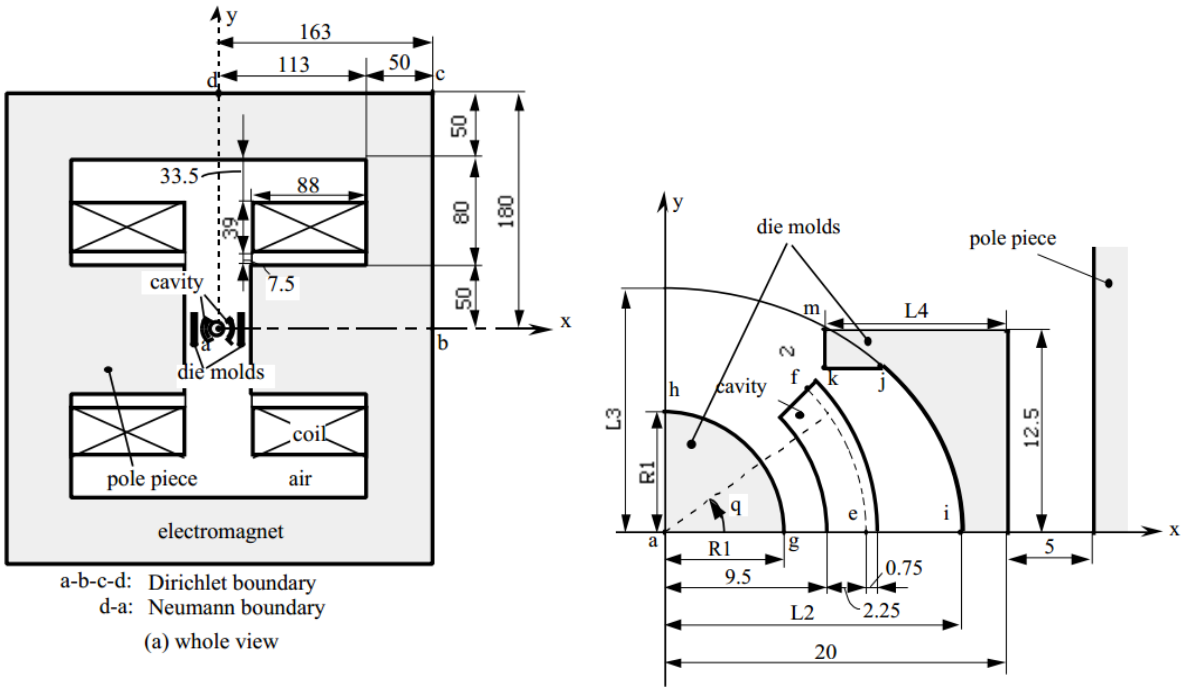


Figure IV.6 Géométrie du problème TEAM Workshop 25

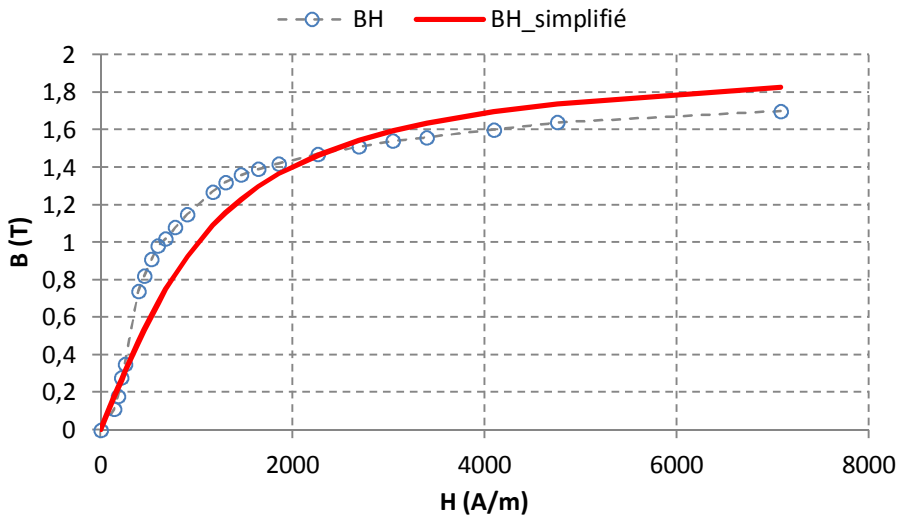


Figure IV.7 Comportement non-linéaire du matériau ferromagnétique de TEAM Workshop 25

La formulation du problème est détaillée dans le chapitre 3, section 3.1. Pour l'exécution sur GPU, nous utilisons le mailleur par bulles parallélisé (voir chapitre 2, section 2.2), le calcul d'intégration numérique et la procédure d'assemblage parallèle par tri (voir chapitre 3, section 3.2), le solveur itératif de type Gradient Conjugué avec un préconditionneur AMG du package CUSP. Le critère d'arrêt du solveur itératif est de 10^{-8}.

Pour résoudre le problème non linéaire lié au comportement du matériau, nous mettons en place la résolution itérative par la méthode Newton-Raphson (NR) avec un suivi de la convergence qui repose sur la norme de l'incrément de la solution entre deux pas consécutif. Le critère d'arrêt du calcul NR est de 10^{-4}. En appliquant ce critère au cas test, 3 itérations de NR sont nécessaires ce qui correspond à 4 boucles de calcul FEM, comprenant l'intégration, l'assemblage et la résolution. Figure IV.8 montre les différentes procédures réalisées dans une simulation par MEF.

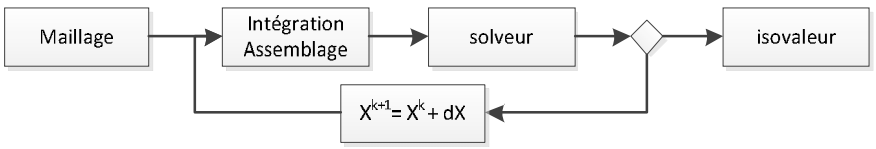


Figure IV.8 Illustration des processus effectués d'une simulation MEF pour le cas test

L'expérience est effectué sur la plateforme équipée d'un processeur Intel Xeon 2,67 GHz, RAM 4 Go et une carte GPU NVIDIA Quadro, K5200, 8 GB RAM, de capacité 3.5, avec une puissance de calcul de 3073 GFLOPS et une bande passante mémoire de 192GB/s. Le calcul sur CPU utilise 4 cœurs.

IV.3.2. Maillage et système d'équations

Nous effectuons les expériences sur plusieurs maillages à différentes densités, reportés dans le Tableau IV-2. Le maillage initial se compose de 4247 éléments triangulaires. Les autres maillages sont construits à partir du maillage initial en divisant chaque triangle par 4 sous-triangles. En conséquence, le nombre d'éléments du maillage résultant est multiple de 4 du maillage initial.

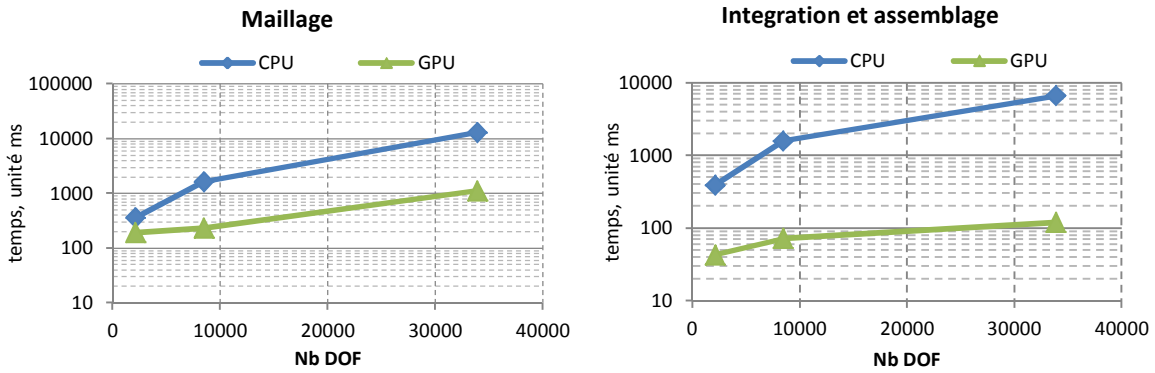
Tableau IV-2 Les maillages utilisés dans l'expérience

Maillage	Nb nœuds	Nb éléments	Nb inconnus	Nb non zéros
	Elément d'ordre 1			
1	2218	4247	2104	14384
2	8682	16988	8455	58467
3	34351	67952	33898	235820
	Elément d'ordre 2			
4	8682	4247	8475	95581
5	34351	16988	33978	386898
6	136653	67952	136068	1556998

L'ordre de l'élément est limité au 2eme ordre qui est un bon compromis entre le temps de calcul et la précision de la solution. La simulation est exécutée sur CPU et GPU et le temps d'exécution des deux implantations est comparé.

IV.3.3. Elément du premier ordre

La Figure IV.9 montre la comparaison du temps pour le cas des éléments du premier ordre. Le résultat du timing nous montre que l'accélération GPU dans la simulation dépend de la taille du problème. En outre, l'effet du calcul parallèle sur GPU est différent selon la partie concernée. La performance des procédures de maillage, d'intégration, d'assemblage et d'iso-valeurs est améliorée considérablement par le GPU. Pour la procédure de résolution, les deux courbes CPU et GPU se croisent. Le point d'intersection correspond à 15 000 DOF simulés, point auquel la performance de résolution sur CPU et GPU sont équivalentes. En conséquence, le choix de meilleur solveur dépend du nombre de DOF du maillage : le solveur exécuté sur CPU est à privilégier pour les cas moins de 15 000 DOF et vice versa.



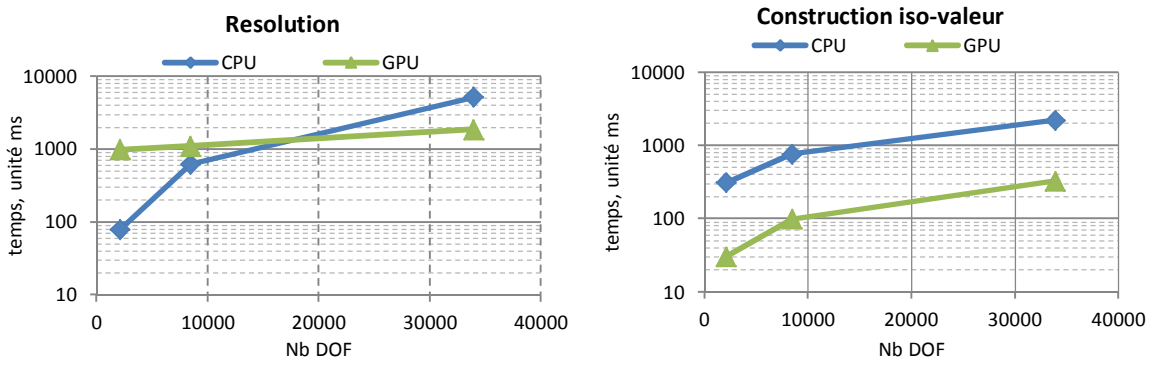


Figure IV.9 Comparaison du temps de CPU et GPU pour le cas des éléments du premier ordre

La dépendance du temps de simulation à la taille du problème mesuré par le nombre de DOF est résumée à la Figure IV.10. Trois implantations sont comparées selon la plateforme où la simulation est effectuée, CPU, GPU ou « GPGPU » cas de calcul hétérogène entre CPU et GPU sur laquelle le choix d'exécution est optimisé soit sur CPU soit sur GPU pour le solveur.

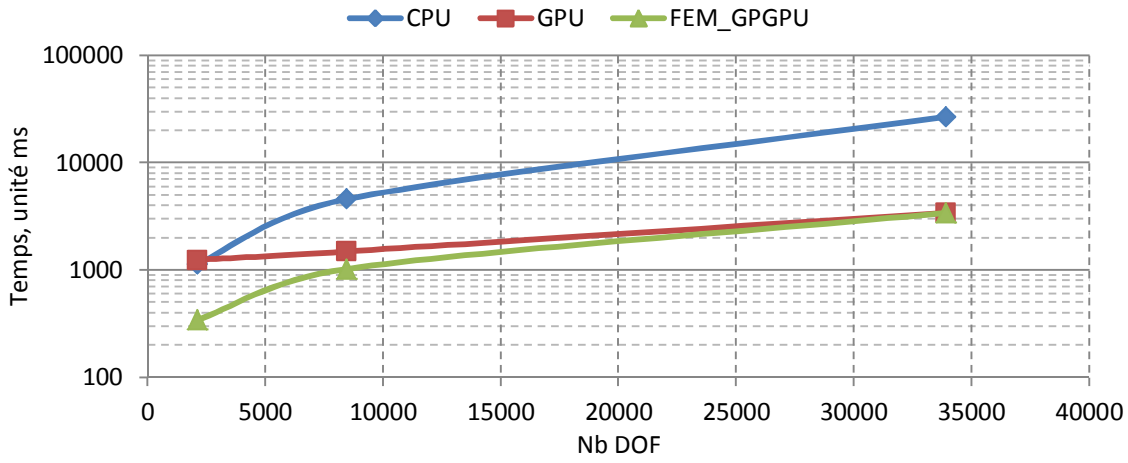


Figure IV.10 Evolution du temps de simulation en fonction du nombre de DOF pour le cas de l'élément du premier ordre

Le résultat montre également une estimation de temps pour terminer une simulation selon le nombre de DOF. Si le maillage contient de 30 000 DOF, ce temps correspond à 3 secondes. Ce dernier est de 20 000 DOF pour 2 secondes et 8 000 DOF pour 1 seconde.

IV.3.4. Elément du deuxième ordre

Le rapport du nombre nœuds/élément du 2ème ordre équivaut environ à quatre fois celui du 1er ordre, comme le montre le Tableau IV-2. La Figure IV.11 montre le résultat du timing des procédures pour le cas de l'élément du deuxième ordre. La performance de calcul sur GPU est bien meilleure que sur CPU pour toutes les parties de calcul MEF. Cependant, nous constatons encore une fois dans la résolution, une limite de DOF pour laquelle le solveur exécuté sur CPU est meilleur que sur GPU. Le solveur exécuté sur GPU est à préférer pour un système de plus de 20 000 DOF dans ce cas test. Il est aussi intéressant de constater que certaines étapes semblent assez insensibles au changement d'ordre des éléments (en particulier maillage et post traitement), alors que d'autres sont plus lourdement affectées (résolution en particulier et intégration / assemblage dans une moindre mesure).

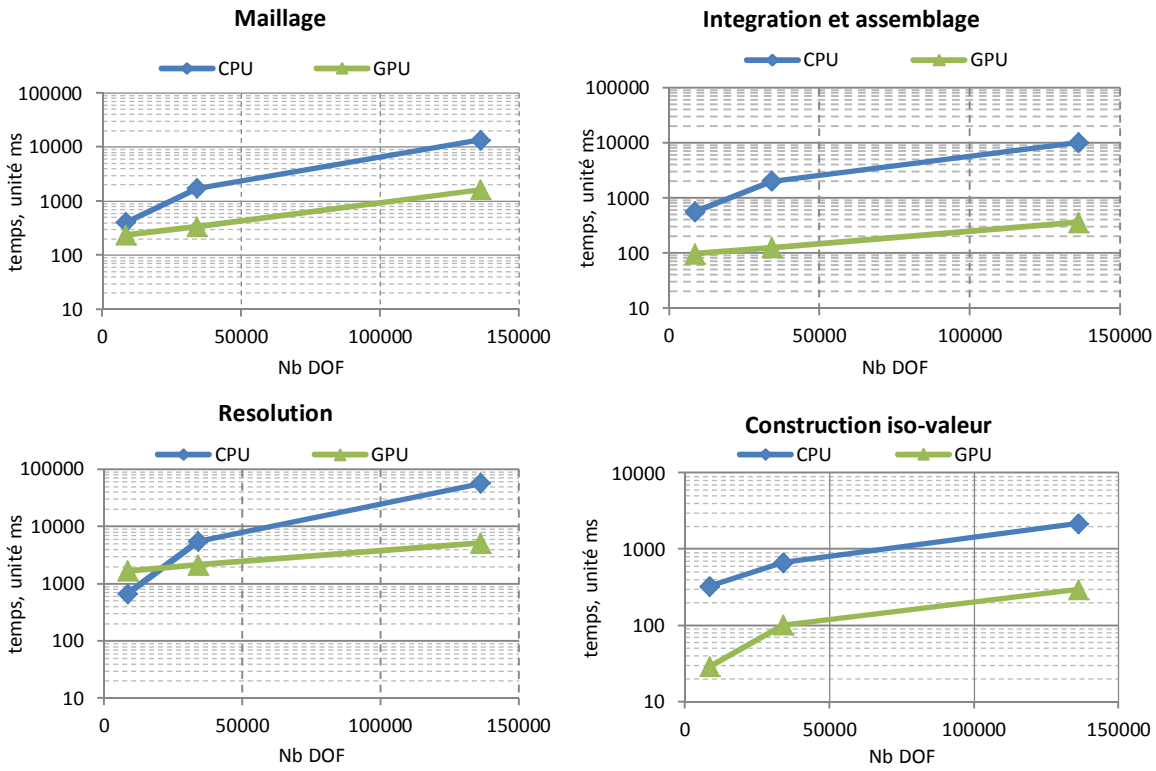


Figure IV.11 Comparaison du temps de CPU et GPU pour le cas des éléments du deuxième ordre

La Figure IV.12 nous donne la relation entre le temps de simulation et le nombre de DOF pour le cas test avec les éléments du deuxième ordre. En comparaison avec le cas du 1er ordre, la taille du problème traité est plus étendue. Pour le cas test, notre méthode peut traiter le problème de 10000 DOF en une seconde, 23 000 DOFs dans deux secondes et environ de 40 000 DOF dans trois secondes… sachant que le temps de résolution compte pour plus des trois quarts de ces valeurs

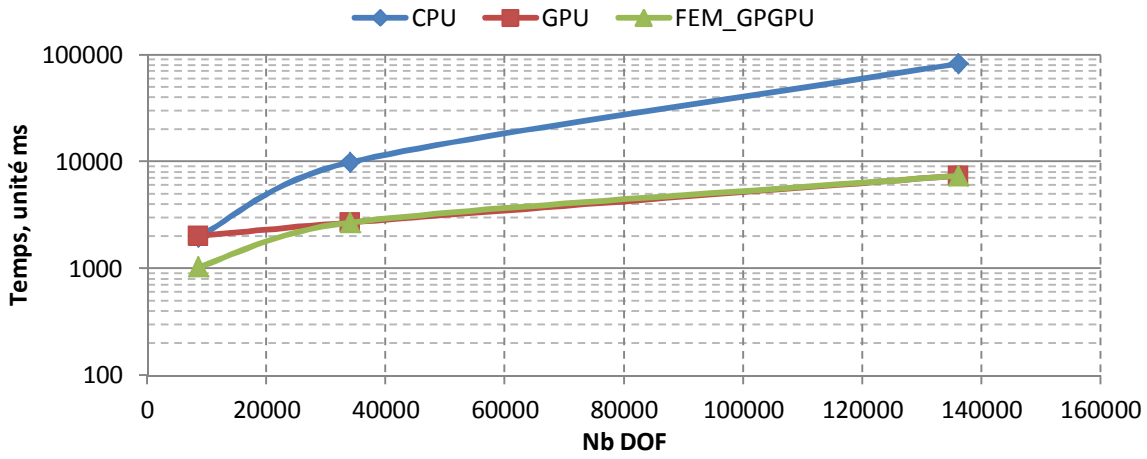


Figure IV.12 Evolution du temps de simulation en fonction du nombre de DOF pour le cas de l'élément du deuxième ordre

La répartition du temps des procédures dans la simulation est différente entre CPU et GPU, comme le montre la Figure IV.13. Nous constatons un changement essentiel de la simulation par éléments finis sur CPU et sur GPU. Pour le cas de CPU, le temps consommé est assez équitablement réparti entre le maillage, l'intégration, l'assemblage et la résolution, tandis que sur

GPU la phase de résolution consomme la majorité du temps. Cela semble confirmer que des gains de performance peuvent être recherchés pour cette étape.

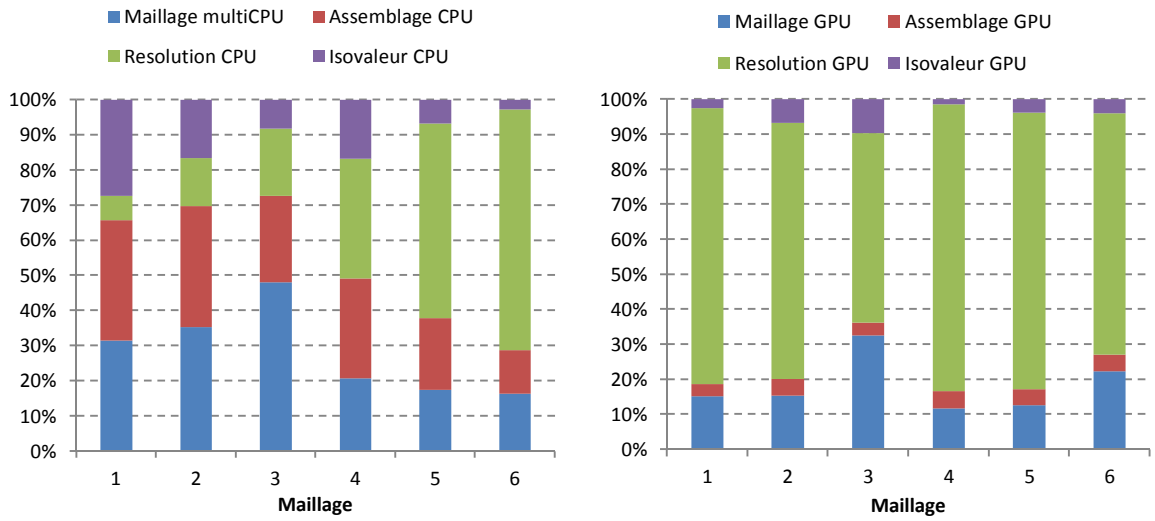


Figure IV.13 Répartition de temps de la simulation par CPU (gauche) et par GPU (droite)

L'intérêt du calcul parallèle sur GPU par rapport sur CPU est différent selon les procédures. L'accélération du code GPU/CPU est reportée par le Tableau IV-3.

Tableau IV-3 Accélération des procédures par GPU contre CPU dans le cas test

Maillage	Mailleur	Intégration assemblage	Résolution	Iso-valeur	Total
1	1,89	9,09	0,08	10,23	3,33
2	7,05	22,08	0,57	7,67	4,48
3	11,67	55,12	2,79	6,80	7,91
4	1,71	5,79	0,41	11,31	0,98
5	5,11	16,10	2,56	6,58	3,66
6	8,24	28,51	11,14	7,33	11,18

IV.3.5. Temps de réponse de la simulation MEF au changement des paramètres

Dans un processus d'optimisation ou d'analyse de sensibilité, la MEF doit fournir la réponse du dispositif vis-à-vis d'un **petit** changement de paramètre géométrique ou/et physique. Le temps nécessaire pour mettre à jour la solution est différent selon le paramètre qui a changé :

- Un changement de densité de courant entraine seulement la mise à jour du vecteur b du système d'équation. Seule la résolution doit être relancée dans ce cas. Le temps de réponse est une fraction du temps de calcul du solveur, car la solution précédente peut servir de point de départ.
- Sur un changement de paramètre physique comme les propriétés des matériaux par exemple, l'intégration, l'assemblage ainsi que la résolution doivent être relancés. Le temps de réponse est la somme des temps de ces deux procédures, avec toujours un gain à espérer sur le solveur en partant de la solution précédente.

- Un petit changement de paramètres géométriques (par exemple R1, L1, L2, L3 dans le problème Team Workshop 25) demande une intervention du mailleur pour reformer le maillage. Pour autant, tout le maillage n'est pas à recalculer ; la méthode de maillage par bulle permet de ne déplacer que les quelques bulles impactées [73]. Et puis la boucle intégration, assemblage et résolution doivent être relancés. Le temps de réponse est égal à une fraction des temps de ces trois procédures (gain sur le maillage en particulier).

En croisant cette analyse avec la taille du problème, caractérisé par le nombre de DOF, il devient possible d'estimer le temps de réponse à une sollicitation, synthétisé dans le Tableau IV-4. Pour cela, nous avons considéré que toutes les modifications demandées sont des petits incréments, ce qui se justifie dans une démarche interactive où l'utilisateur cherche à sentir la sensibilité de son dispositif. Dans ce cas, il convient évidemment de profiter de la solution précédente pour calculer la nouvelle solution. En considérant qu'en moyenne, une recherche d'un incrément de solution est cinq fois moins cher que le calcul complet de la solution en partant de zéro (moins d'itérations de NR et de CG), et en se rapportant à Figure IV.11 par exemple, en 1 seconde de 50 000 à 100 000 degrés de libertés semblent accessibles.

Pour autant, il faut investir sur la partie solveur, qui consomme près de 80% du temps de toutes les étapes de la MEF dans les implantations sur GPU.

Tableau IV-4 Estimation de la taille du problème à l'adaptation du temps de réponse

Paramètre changé	Procédures	Le nombre de DOF traité dans 1 seconde
Densité du courant	Résolution	$5\ 10^4$ à 10^5 DOF
Perméabilité	Intégration Assemblage Résolution	$5\ 10^4$ à 10^5 DOF
Paramètres géométrique R1, L1, L2, L3	Maillage Intégration Assemblage Résolution	2 à $5\ 10^4$ DOF

IV.4. Conclusion

Dans ce chapitre, un calcul parallèle sur GPU est proposé afin d'accélérer le post-traitement de la simulation par éléments finis. Le calcul des iso-valeurs est adapté à la parallélisation sur GPU. La méthode proposée a pu être validée à l'aide d'une expérience présentée dans la section 0. Le temps d'exécution montre une accélération de plusieurs dizaines de fois par rapport au calcul traditionnel sur CPU.

A partir de la méthodologie mise en place dans ce travail, une approche générale a été proposée pour répondre au cas où toutes les étapes du calcul par éléments finis sont conduites sous forme de calcul parallèle sur GPU. Afin d'optimiser cette approche, il est nécessaire de réserver la structure de données à la fois sur la mémoire du CPU et du GPU pour réduire le coût de transfert entre les deux plateformes. L'alliance CPU et GPU rend la simulation plus performante en réduisant le coût de calcul un ordre de grandeur. L'intérêt du calcul parallèle sur GPU pour le calcul d'intégration et d'assemblage est plus important que d'autres procédures comme le maillage ou la résolution. En conséquence, le coût de la simulation dépend essentiellement à la phase de résolution.

Cherchant à atteindre une simulation MEF toujours plus rapide, le travail dans ce chapitre a quantifié la relation entre le temps de la simulation et la taille du problème simulé. Dans notre implantation, le « temps réel » (1 seconde) n'est atteignable que pour un problème dont la taille est limitée à une centaine de milliers d'inconnues.

Les expériences réalisées dans ce chapitre présentent quelques limitations :

- Le stockage des données sur la mémoire de GPU est limité par la taille de la mémoire, habituellement de quelques GB. Donc, il est nécessaire de bien évaluer la capacité de la mémoire GPU avant le chargement des données : le stockage net des données d'un million d'éléments triangulaires d'ordre 2 demande environ 300MB de mémoire en simple précision, et environ de 500MB en double précision.

- l'optimisation du calcul des kernels CUDA nécessite un investissement en temps important, surtout dans un environnement de programmation comme JAVA. Le manque d'outils natif en C/C++ comme le «Visual Profiler » a limité la capacité d'optimisation du kernel CUDA en JAVA. Dans ce travail, l'optimisation en JAVA que nous avons conduite se base essentiellement sur la simple analyse des temps des kernels sans pouvoir décomposer plus finement ces temps.

Les expériences montrent une croissance quasi linéaire de l'accélération du calcul parallèle avec la taille du problème, caractérisé par le nombre d'éléments du maillage ou le nombre d'inconnues du système d'équations.

CONCLUSION GENERALE ET PERSPECTIVES

Les travaux réalisés dans la cadre de cette thèse permettent de proposer une approche basée sur les GPUs comme accélérateurs de performance pour la simulation par éléments finis dans le domaine de l'électromagnétisme. L'introduction du GPU dans un outil de calcul numérique par la MEF s'est appuyée sur l'accélération des étapes de maillage, d'intégration numérique et d'assemblage, de résolution et de post-traitement. Au final, une plateforme de calcul hétérogène mêlant CPU et GPU a été proposée dans laquelle toutes les procédures exigeant des traitements complexes sont traitées sur CPU tandis que les calculs lourds et uniformes sont portés sur GPU.

Le code parallèle a été développé avec des approches dédiées respectant les contraintes et tirant partie des forces de ce type de plateforme afin de profiter des capacités de calcul numérique très forte des GPU. Les calculs sur GPU ont pris en compte les paramètres suivants :

- l'équilibrage des calculs entre les threads parallèles

- la coopération et la communication entre les threads

- l'optimisation de l'accès à la mémoire GPU, le bon usage de la mémoire partagée, des registres, des mémoires constantes et des mémoires de texture

- l'optimisation des opérations algébriques du code et l'optimisation de la configuration du kernel

Les expériences successives ont fait l'objet de validation des algorithmes pour chaque implantation en calcul parallèle sur GPU :

- Le maillage par bulles parallélisé permet de générer automatiquement des maillages de très haute qualité dans un temps inférieur ou comparable à ceux nécessaires pour des maillages de moindre qualité par des techniques de Delaunay.

- Une nouvelle approche de calcul parallèle sur GPU est proposée pour la procédure d'intégration numérique et d'assemblage de la MEF. Le calcul d'intégration par élément est assigné aux threads parallèles pour obtenir une meilleure performance de la parallélisation. Au lieu d'assembler chaque terme non nul, la méthode proposée construit une procédure générale basée sur le tri et la réduction / construction de la matrice globale. Cette méthode réduit fortement le coût d'intégration et d'assemblage. Une comparaison entre notre méthode et d'autres méthodes existantes a été réalisée. L'expérience a confirmé que notre méthode était plus efficace et se comportait bien non seulement pour les problèmes de grandes tailles mais également pour les ordres élevés d'interpolation.

- Des implantations des solveurs itératifs de famille Krylov présentes dans les packages CUDA ont été testées. Le solveur de type Gradient Conjugué avec un préconditionneur approprié permet d'améliorer considérablement le temps de résolution, surtout pour les systèmes de taille au-delà de la dizaine de milliers d'inconnues. Cependant, en comparaison avec la procédure d'assemblage, l'accélération du calcul parallèle sur GPU pour la phase de résolution est moins

nette. Tandis que l'assemblage parallèle sur GPU permet de traiter des centaines de milliers d'éléments en quelques centaines de millisecondes, la résolution du système correspondant demande plutôt des secondes.

En mettant en œuvre le calcul parallèle sur GPU pour toutes les procédures de la simulation MEF, une simulation en temps réel semble à portée de main. Tout mis bout à bout, les expériences réalisées montrent une accélération d'un ordre de magnitude de la simulation sur une plateforme hétérogène CPU et GPU. Le temps de simulation, comprenant maillage, résolution et exploitation, se trouve désormais réduit à quelques secondes pour 10^5 DOF au lieu de quelques dizaines de secondes à l'origine. Le principal verrou réside désormais dans le solveur, qui n'ayant pas fait l'objet de développements spécifiques, peut sans doute être optimisé et permettre de conduire prochainement des études de sensibilité en « temps réel ».

A la suite de ces travaux, quelques perspectives sont envisagées. Premièrement, une étude complète sur les solveurs itératifs parallélisé sur GPU est vraiment nécessaire afin d'améliorer la performance de la phase de résolution. Deuxièmement, les approches de calcul parallèle pourraient être testées avec d'autres formulations et d'autres modèles physiques. Troisièmement, en considérant l'intérêt du calcul parallèle pour les simulations de grande taille, l'étude pourrait être poursuivie en travaillant sur une extension multi-GPU dans le but de proposer un logiciel plus performant pour les grandes simulations éléments finis en 3D.

Bibliographie

[1] CUDA C Programming Guide, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[2] Tesla C1060 Computing Processor Board - Nvidia, www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf

[3] Intel® Core i7-3700 Desktop Processor Series www.intel.com/content/dam/support/us/en/documents/processors/corei7/sb/core_i7-3700_d.pdf

[4] D. Komatitsch, D. Michéa, et G. Erlebacher, « Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA », *J. Parallel Distrib. Comput.*, vol. 69, n° 5, p. 451□ 460, mai 2009.

[5] Z. Fu, T. James Lewis, R. M. Kirby, et R. T. Whitaker, « Architecting the finite element method pipeline for the GPU », *J. Comput. Appl. Math.*, vol. 257, p. 195□ 211, févr. 2014.

[6] J. Zhang et D. Shen, « GPU-Based Implementation of Finite Element Method for Elasticity Using CUDA », 2013, p. 1003□ 1008.

[7] Y. Cai, G. Li, et H. Wang, « A Parallel Node-based Solution Scheme for Implicit Finite Element Method Using GPU », *Procedia Eng.*, vol. 61, p. 318□ 324, 2013.

[8] A. Dziekonski, P. Sypek, A. Lamecki, et M. Mrozowski, « GPU-Accelerated Finite-Element Matrix Generation for Lossless, Lossy, and Tensor Media [EM Programmer's Notebook] », *Antennas Propag. Mag. IEEE*, vol. 56, n° 5, p. 186–197, 2014.

[9] M. G. Knepley et A. R. Terrel, « Finite element integration on GPUs », *ACM Trans. Math. Softw. TOMS*, vol. 39, n° 2, p. 10, 2013.

[10] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon, et J.-M. Jin, « GPU accelerated finite-element computation for electromagnetic analysis », *Antennas Propag. Mag. IEEE*, vol. 56, n° 2, p. 39–62, 2014.

[11] T. Okimura, T. Sasayama, N. Takahashi, et S. Ikuno, « Parallelization of Finite Element Analysis of Nonlinear Magnetic Fields Using GPU », *IEEE Trans. Magn.*, vol. 49, n° 5, p. 1557-1560, mai 2013.

[12] P. Foteinos et N. Chrisochoides, « Dynamic parallel 3D Delaunay triangulation », in *Proceedings of the 20th International Meshing Roundtable*, Springer, 2011, p. 3–20.

[13] G. Rong, T.-S. Tan, T.-T. Cao, et others, « Computing two-dimensional Delaunay triangulation using graphics hardware », in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008, p. 89–97.

[14] T.-T. Cao, A. Nanjappa, M. Gao, et T.-S. Tan, « A GPU accelerated algorithm for 3D Delaunay triangulation », in *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2014, p. 47–54.

[15] C. Cecka, A. J. Lew, et E. Darve, « Assembly of finite element methods on graphics processors », *Int. J. Numer. Methods Eng.*, vol. 85, n° 5, p. 640–669, 2011.

[16] M. Ament, G. Knittel, D. Weiskopf, et W. Strasser, « A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform », in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, p. 583–592.

[17] M. Naumov, « Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS », *Nvidia White Pap.*, 2011.

[18] R. Li et Y. Saad, « GPU-accelerated preconditioned iterative linear solvers », *J. Supercomput.*, vol. 63, n° 2, p. 443–466, 2013.

[19] N. Bell et M. Garland, « Efficient sparse matrix-vector multiplication on CUDA », Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[20] R. Hochberg, *Matrix multiplication with cuda-a basic introduction to the cuda programming model*. Shodor, 2012.
[21] The Kepler GK110 Whitepaper - Nvidia, http://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
[22] J. Sanders et E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*, 3. printing. Upper Saddle River, NJ: Addison-Wesley, 2011.
[23] S. Liang, *The Java Native interface: programmer's guide and specification*. Reading, Mass.: Addison-Wesley, 1999.
[24] J. Docampo, S. Ramos, G. L. Taboada, R. R. Exposito, J. Tourino, et R. Doallo, « Evaluation of Java for general purpose GPU computing », in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, 2013, p. 1398–1404.
[25] CUDA Samples, http://docs.nvidia.com/cuda/pdf/CUDA_Samples.pdf
[26] T. Yokoyama, V. Cingoski, K. Kaneda, et H. Yamashita, « 3-D automatic mesh generation for FEA using dynamic bubble system », *Magn. IEEE Trans. On*, vol. 35, n° 3, p. 1318–1321, 1999.
[27] Z. Tang, Y. Le Menach, E. Creuse, S. Nicaise, F. Piriou, et N. Nemitz, « Residual and equilibrated error estimators for magnetostatic problems solved by finite element method », *IEEE Trans. Magn.*, vol. 49, n° 12, p. 5715‑5723, déc. 2013.
[28] P. Frey et P.-L. George, *Mesh generation*. John Wiley & Sons, 2013.
[29] K. Shimada et D. C. Gossard, « Bubble mesh: automated triangular meshing of non-manifold geometry by sphere packing », in *Proceedings of the third ACM symposium on Solid modeling and applications*, 1995, p. 409–419.
[30] M. Filipiak, « Mesh generation », *Edinb. Parallel Comput. Cent. Univ. Edinb. Edinb.*, 1996.
[31] D. J. Mavriplis, « An advancing front Delaunay triangulation algorithm designed for robustness », *J. Comput. Phys.*, vol. 117, n° 1, p. 90–101, 1995.
[32] Q. Du et D. Wang, « Recent progress in robust and quality Delaunay mesh generation », *J. Comput. Appl. Math.*, vol. 195, n° 1‑2, p. 8‑23, oct. 2006.
[33] B. Delaunay, « Sur la sphère vide », Izvestia Akademii Nauk SSSR, *Otdelenie Matematicheskikh i Estestvennykh Nauk*, 7:793-800, 1934.
[34] K. Ho-Le, « Finite element mesh generation methods: a review and classification », *Comput.-Aided Des.*, vol. 20, n° 1, p. 27–38, 1988.
[35] A. Nanjappa, « Delaunay triangulation in R3 on the GPU », 2012.
[36] C. Arens, « The Bowyer-Watson algorithm; An efficient implementation in a database environment », *Delft University of Technology, Faculty of Civil Engineering and Geosciences, Department of Geodesy, Section GIS Technology*, 2002.
[37] Bowyer, Adrian. "Computing dirichlet tessellations." *The Computer Journal 24.2* (1981): 162-166.
[38] Watson, David F. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes." *The computer journal 24.2* (1981): 167-172.
[39] R. Hemsley, « Interpolation on a magnetic field », *Bristol University, Tech. Rep*, 2009.
[40] J. F. Thompson, B. K. Soni, et N. P. Weatherill, *Handbook of grid generation*. CRC press, 1998.
[41] Ruppert, Jim. "A Delaunay refinement algorithm for quality 2-dimensional mesh generation." *Journal of algorithms 18.3* (1995): 548-585.
[42] L. P. Chew, « Guaranteed-quality mesh generation for curved surfaces », in *Proceedings of the ninth annual symposium on Computational geometry*, 1993, p. 274–280.
[43] J. R. Shewchuk, « Delaunay refinement algorithms for triangular mesh generation », *Comput. Geom.*, vol. 22, n° 1, p. 21–74, 2002.
[44] Chrisochoides, Nikos P. "A survey of parallel mesh generation methods." *Brown University, Providence RI-2005* (2005).

[45] G. E. Blelloch, G. L. Miller, et D. Talmor, « Developing a practical projection-based parallel Delaunay algorithm », in *Proceedings of the twelfth annual symposium on Computational geometry*, 1996, p. 186–195.
[46] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, et P. Foteinos, « Towards exascale parallel delaunay mesh generation », in *Proceedings of the 18th international meshing roundtable*, Springer, 2009, p. 319–336.
[47] M. Qi, T.-T. Cao, et T.-S. Tan, « Computing 2D constrained Delaunay triangulation using the GPU », *IEEE Trans. Vis. Comput. Graph.*, vol. 19, n° 5, p. 736–748, 2013.
[48] C. Navarro, N. Hitschfeld-Kahler, et E. Scheihing, « A parallel gpu-based algorithm for delaunay edge-flips », in *The 27th European Workshop on Computational Geometry, EuroCG*, 2011, vol. 11.
[49] V. Cingoski, R. Murakawa, K. Kaneda, et H. Yamashita, « Automatic mesh generation in finite element analysis using dynamic bubble system », *J. Appl. Phys.*, vol. 81, n° 8, p. 4085, 1997.
[50] J.-H. Kim, H.-G. Kim, B.-C. Lee, et S. Im, « Adaptive mesh generation by bubble packing method », *Struct. Eng. Mech.*, vol. 15, n° 1, p. 135–150, 2003.
[51] M. Ebene-Ebene, Y. Marechal, et D. Ladas, « An Adaptive Remeshing Technique Ensuring High Quality Meshes », *IEEE Trans. Magn.*, vol. 44, n° 6, p. 1222‑1225, juin 2008.
[52] K. J. Aström et R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.
[53] W. H. Press, S. A. Teukolsky, W. T. Vetterling, et B. P. Flannery, *Numerical recipes in C*, vol. 2. Citeseer, 1996.
[54] M. Ebene, " Régularisation de maillage éléments finis par « Bubble » : Impact sur la qualité de la solution dans les simulations numériques en génie électrique", Avril 2009.
[55] F. Nobuyama, S. Noguchi, et H. Igarashi, « The Parallelized Automatic Mesh Generation Using Dynamic Bubble System With GPGPU », *IEEE Trans. Magn.*, vol. 49, n° 5, p. 1677‑1680, mai 2013.
[56] V. Garcia, E. Debreuve, F. Nielsen, et M. Barlaud, « K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching », in *Image Processing (ICIP), 2010 17th IEEE International Conference on*, 2010, p. 3757–3760.
[57] Y. Yan, M. Grossman, et V. Sarkar, « JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA », in *Euro-Par 2009 Parallel Processing*, Springer, 2009, p. 887–899.
[58] W. Abu-Sufah et A. A. Karim, « An Effective Approach for Implementing Sparse Matrix-Vector Multiplication on Graphics Processing Units », 2012, p. 453‑460.
[59] Frenkel, Daan, and Berend Smit. "Understanding molecular simulations: from algorithms to applications." Academic, San Diego (1996).
[60] M. Kelly et A. Breslow, « Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach », *Retrieved June13*, 2011.
[61] S. Green, « Cuda particles », *NVidia Whitepaper*, vol. 2, n° 3.2, p. 1, 2008.
[62] B. H. Heidelberger, « Consistent collision and self-collision handling for deformable objects », ETH Zurich, 2007.
[63] A. Chin, « Locality-preserving hash functions for general purpose parallel computation », *Algorithmica*, vol. 12, n° 2‑3, p. 170–181, 1994.
[64] « Fast construction of k-nearest neighbour graphs for point clouds.pdf ». .
[65] C. Geuzaine et J.-F. Remacle, « Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities », *Int. J. Numer. Methods Eng.*, vol. 79, n° 11, p. 1309–1331, 2009.
[66] G. Dhatt et G. Touzot, *Une présentation de la méthode des éléments finis*. Paris : Québec: Maloine, Presses de l'Université Laval., 1981.
[67] R. Bridson et W.-P. Tang, « Refining an approximate inverse », *J. Comput. Appl. Math.*, vol. 123, n° 1, p. 293–306, 2000.

[68] A. Dziekonski, P. Sypek, A. Lamecki, et M. Mrozowski, « Finite element matrix generation on a GPU », *Prog. Electromagn. Res.*, vol. 128, p. 249–265, 2012.

[69] A. Dziekonski, P. Sypek, A. Lamecki, et M. Mrozowski, « Accuracy, Memory, and Speed Strategies in GPU-Based Finite-Element Matrix-Generation », *IEEE Antennas Wirel. Propag. Lett.*, vol. 11, p. 1346‑1349, 2012.

[70] A. Dziekonski, P. Sypek, A. Lamecki, et M. Mrozowski, « Generation of large finite-element matrices on multiple graphics processors: GENERATION OF LARGE FINITE ELEMENT MATRICES ON MULTIPLE GPUS », *Int. J. Numer. Methods Eng.*, vol. 94, n° 2, p. 204‑220, avr. 2013.

[71] G. Karypis et V. Kumar, « A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices », *Univ. Minn. Dep. Comput. Sci. Eng. Army HPC Res. Cent. Minneap. MN*, 1998.

[72] TEAM Workshop 25
http://www.compumag.org/jsite/images/stories/TEAM/problem25.pdf.

[73] C. Hérault, « Vers une simulation sans maillage des phénomenes électromagnétiques », *Institut National Polytechnique de Grenoble-INPG*, 2000.

[74] A. N. Chernikov et N. P. Chrisochoides, « Parallel 2D graded guaranteed quality Delaunay mesh refinement », *in Proceedings of the 14th International Meshing Roundtable*, 2005, p. 505–517.

Publications

Articles de conférences

- V-Q Dinh, B. Bannwarth, Y. Marechal, « *Parallelization on GPU of Mesh Generation Using Dynamic Bubble System* » CEFC 2014, 25-28 May 2014, Annecy, France.

Abstract :

The meshing method based on a dynamic bubble system is known for its ability to generate high quality meshes. However, the computation time remains a problem since the motion convergence is slow and it increases drastically with the number of nodes. Our goal is to significantly reduce the simulation time of bubbles motion by using a massively parallel computing based on the CUDA NVIDIA architecture. In this paper, we explain in detail the parallel computations and algorithms for the bubble packing meshing method and give the main results obtained.

- V-Q Dinh, Yves Marechal, « Vers une simulation en temps réel par éléments finis sur GPU », NUMELEC 2015, 3 - 5 Juin 2015, St. Nazaire, France.

Abstract :

Dans cet article, nous introduisons une technique d'assemblage parallèle sur GPU pour la méthode des éléments finis (MEF) appliqué dans le calcul de champ magnétique. En fait, chaque thread calcule l'intégration associée à un élément. Pour éviter les conflits de mémoire, nous avons introduit une procédure rapide basée le tri et le réarrangement des non-zéro (NZ) entrées par son indice de ligne. Enfin, un procédé de réduction par ligne est exécutée à assembler NZ dans la matrice de rigidité. Cet algorithme ne nécessite aucun prétraitement sur maille mais aussi profiter de la puissance de calcul parallèle du GPU à travers l'équilibrage de charge. Dans nos tests, l'utilisation de cette méthode a amélioré la vitesse d'assemblage jusqu'à 20x fois plus rapide.

- Q. Dinh, Y. Marechal., G. Meunier, "*Toward Real-Time Finite-Element Simulation on GPU*." Compumag 2015, 28 Juin – 2 July 2015, Montréal, Québec, Canada.

Abstract :

In this paper, we introduce a parallel assembly technique on NVIDIA CUDA GPUs for finite element method applied in the magnetic field. In particular, each thread calculates the integration associated with an element. Then, we use the index of row for sorting and rearranging elementary non-zero entries. Finally, a reducing process by row is executed to assemble NZ in the stiffness matrix. This algorithm does not require any preprocessing on mesh but also take advantage of parallel computing power of GPU through load balancing. In our tests, using this parallel assembly improved the speed assembling up to 20x times faster.

Articles de revues internationales

- Dinh Quang, Yves Marechal. "Toward Real-Time Finite-Element Simulation on GPU." *IEEE Transactions on Magnetics* , vol. 52, n° 3, p. 1-4, 2016.
  doi: 10.1109/TMAG.2015.2477602

Abstract :

In this paper, we introduce some parallel techniques on NVIDIA compute unified device architecture GPU for the finite-element method applied in the magnetic field computation. To ensure the load balance, each parallel thread performs the integration of one element. In the assembly step, we introduced a fast procedure based on the sorting and rearrangement of non-zero entries on the GPU global memory. Then, a reducing process is executed to obtain the resulting coefficient matrix in a sparse format. About the solving step, we use the conjugate gradient iterative solver with a variety of preconditioning techniques. Our implementation does not require any preprocessing on mesh, but takes advantage of the parallel computing power of GPU. In our test, this parallel strategy improved the performance 30 times faster on the assembly process and four times faster on the solving process.

- V-Q Dinh, Yves Marechal, « GPU-based Parallelization for Bubble Mesh Generation », COMPEL, submitted on August 2016

Abstract:

In FEM computations, the mesh quality improves the accuracy of the approximation solution and reduces the computation time. The dynamic bubble system meshing technique can provide high quality meshes but the packing time is slow. This paper aims to improve the running time of the bubble meshing by using the advantages of parallel computing on GPU. Our study is based on the analysis of running time on CPU. A massively parallel computing based CUDA architecture is proposed to improve the bubbles displacement and database updating. Constraints linked to hardware considerations are taken into account. Last, speedup factors are provided on test cases and real scale examples. The numerical experiences show the efficiency of our parallel performance reaches a speedup of 30x compared to the serial implementation. Our contribution is so far limited to 2D geometries although the extension to 3D is straightforward regarding the meshing technique itself and our GPU implementation. Our works are based on a CUDA environment which is widely used by developers. C\C++ and Java were the used programming languages. Other languages may of course lead to slightly different implementations. Our approach makes it possible to use bubble meshing technique for both initial design and optimization, since excellent meshes can be built in a few seconds. Compared to previous works, our contribution points out that the scalability of the bubble meshing technique needs to solve 2 key issues: reach a $\Theta(N)$ global cost of the implementation and reach a very fast size map interpolation strategy

VERS UNE SIMULATION PAR ÉLÉMENTS FINIS EN TEMPS RÉEL POUR LE GÉNIE ÉLECTRIQUE

Résumé:

Les phénomènes physiques dans le domaine de génie électrique sont basés sur les équations de Maxwell qui sont des équations aux dérivés partielles dont les solutions sont des fonctions s'appuyant sur les propriétés des matériaux et vérifiant certaines conditions aux limites du domaine d'étude. La méthode des éléments finis (MEF) est la méthode la plus couramment utilisée pour calculer les solutions de ces équations et en déduire les champs et inductions magnétiques et électriques. De nos jours, le calcul parallèle GPU (Graphic Processor Unit) présente un potentiel important de performance à destination du calcul numérique par rapport au calcul traditionnel par CPU. Le calcul par GPU consiste à utiliser un processeur graphique (Graphic Processor Unit) en complément du CPU pour accélérer les applications en sciences et en ingénierie. Le calcul par GPU permet de paralléliser massivement les tâches et d'offrir ainsi un maximum de performances en accélérant les portions de code les plus lourdes, le reste de l'application restant affectée au CPU. Cette thèse s'inscrit dans le contexte de modélisation dans le domaine de génie électrique utilisant la méthode des éléments finis. L'objectif de la thèse est d'améliorer la performance de la MEF, voire d'en changer les modes d'utilisation en profitant de la grande performance du calcul parallèle sur GPU. En effet, si grâce au GPU, le calcul parvenait à s'effectuer en quasi temps réel, les outils de simulation deviendraient alors des outils de conception intuitifs, qui permettraient par exemple de « sentir » la sensibilité d'un dimensionnement à la modification de paramètres géométriques ou physiques. Un nouveau champ d'utilisation des codes de simulation s'ouvrirait alors. C'est le fil conducteur de ce travail, qui tente, en abordant les différentes phases d'une simulation par la MEF, de les accélérer au maximum, pour rendre l'ensemble quasi instantané. Les phases de maillage, intégration, résolution et exploitation sont abordées successivement. Pour chacune de ces grandes étapes de la simulation d'un dispositif, les méthodes de la littérature sont examinées et de nouvelles approches sont proposées. Les performances atteintes sont analysées et comparées au cout de l'implantation traditionnelle sur CPU. Les détails d'implantation sont décrits assez finement, car la performance globale des approches sur GPU sont très liés à ces choix.

Mots clés: **Méthode des Eléments Finis, Calcul Parallèle, Génie Electrique, CUDA**

TOWARDS A REAL-TIME SIMULATION BY FINITE ELEMENTS FOR ELECTRICAL ENGINEERING

Abstract:

The physical phenomena in the electrical engineering field are based on Maxwell's equations in which solutions are functions verifying the material properties and satisfying certain boundary conditions on the field. The finite element method (FEM) is the most commonly used method to calculate the solutions of these equations and deduce the magnetic and electric fields. Nowadays, the parallel computing on graphics processors offers a very high computing performance over traditional calculation by CPU. The GPU-accelerated computing makes use of a graphics processing unit (GPU) together with a CPU to accelerate many applications in science and engineering. It enables massively parallelized tasks and thus accelerates the performance by offloading the compute-intensive portions of the application to the GPU while the remainder of the application still runs on the CPU. The thesis deals with the modeling in the magnetic field using the finite element method. The aim of the thesis is to improve the performance of the MEF by taking advantage of the high performance parallel computing on the GPU. Thus if the calculation can be performed in near real-time, the simulation tools would become an intuitive design tool which allow for example to "feel" the sensitivity of a design modification of geometric and physical parameters. A new field of use of simulation codes would open. This is the theme of this work, which tries to accelerate the different phases of a simulation to make the whole almost instantaneous. So in this thesis, the meshing, the numerical integration, the assembly, the resolution and the post processing are discussed respectively. For each phase, the methods in the literature are examined and new approaches are proposed. The performances are analyzed and compared. The implementation details are described as the overall performance of GPU approaches are closely linked to these choices.

Keywords: **Finite Element Method, Parallel Computing, Electrical Engineering, CUDA**