Parallèlisme des calculs numériques appliqué aux géosciences

Gauthier Sornet

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE : LIFO

Thèse présentée par :

Gauthier SORNET

soutenue le : **Jeudi 10 Octobre 2019**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Parallélisme des calculs numériques appliqué aux géosciences

THÈSE DIRIGÉE PAR :

Sébastien LIMET	Professeur des Universités, LIFO - Univ. d'Orléans

THÈSE CO-ENCADRÉE PAR :

Sylvain JUBERTIE	Maître de Conférences, LIFO - Univ. d'Orléans

RAPPORTEURS :

Jean-François MÉHAUT	Professeur des Universités, LIG - Univ. de Grenoble Alpes
Denis BARTHOU	Professeur des Universités, LABRI - INRIA Bordeaux

EXAMINATEURS :

Mirian HALFELD FERRARI ALVES, présidente du jury	Professeur des Universités, LIFO - Univ. d'Orléans
Jean-François MÉHAUT	Professeur des Universités, LIG - Univ. de Grenoble Alpes
Denis BARTHOU	Professeur des Universités, LABRI - INRIA Bordeaux
Fabrice DUPROS	Ingénieur de recherche, Etablissement d'affectation : ARM
Faiza BOULAHYA	Ingénieur, Etablissement d'affectation : BRGM
Sébastien LIMET	Professeur des Universités, LIFO - Univ. d'Orléans

Remerciements

Nous remercions tout d'abord la Région Centre-Val de Loire d'avoir cofinancé cette thèse. On remercie également toutes les personnes ayant participé de près ou de loin à ces travaux. De manière non exhaustive, voici une liste de personnes que l'on remercie tout particulièrement : Sébastien LIMET, Sylvain JUBERTIE, Fabrice DURPOS, Faiza BOULAHYA, Philippe THIERRY, Florent de MARTIN ainsi que nos deux rapporteurs et les membres du jury. On remercie aussi l'Université d'Orléans, le LIFO et le BRGM ainsi que les personnes qui y travaillent. Enfin, je voudrais également remercier Stéphanie INGRAND de m'avoir soutenu durant cette période.

Sommaire

Liste des figures

Chapitre 1

Introduction

Durant la dernière décennie, les ruptures technologiques introduites dans le design des supercalculateurs ont progressivement conduit à revisiter l'architecture des grands codes de calcul scientifique. Par exemple, cette situation se vérifie dans le domaine des géosciences qui agrègent des enjeux industriels, scientifiques et sociétaux majeurs tels que l'étude du changement climatique, la protection de la ressource en Eau ou l'exploration pétrolière et gazière. L'une des spécificités des géosciences vient des phénomènes physiques souvent compliqués à modéliser. Ces derniers nécessitent la prise en compte de différentes échelles de temps sur des domaines de simulation tridimensionnelle intégrant différentes sources d'hétérogénéités. Au niveau des architectures parallèles, différentes évolutions impactent significativement l'implémentation des algorithmes classiques du domaine des géosciences. Tout d'abord, la hiérarchisation croissante des niveaux de mémoire constitue un challenge pour l'exploitation efficace des architectures sous-jacentes. En effet les topologies symétriques précédemment implémentées ont progressivement disparues. Ces dernières constituaient de sévères goulots d'étranglement, notamment pour les noyaux de calcul avec une faible intensité arithmétique. Les architectures NUMA (Non-Uniform Memory Access) se sont rapidement généralisées offrant de meilleures performances, notamment dans un contexte d'augmentation continue du nombre de cœurs de calcul. Ces performances sont généralement obtenues au prix de stratégies fines pour le placement des allocations mémoire et des tâches de calcul de l'application cible. Les prochaines architectures devraient encore accroître la profondeur de cette hiérarchie avec par exemple l'implémentation d'approches telles que la « High Bandwidth Memory (HBM) ». Ensuite, le design des processeurs a profondément évolué ces dernières années. En effet, on peut observer l'importance croissante des unités vectorielles de taille et de complexité grandissantes qui fournissent la majeure partie de la puissance de calcul. Typiquement, on retrouve aujourd'hui des unités SIMD (Single Instruction Multiple Data) avec des vecteurs de taille 512 bits. L'exploitation efficace de ces capacités de traitement vectoriel est traditionnellement déléguée au compilateur. En fonction de la complexité du noyau de calcul et des transformations applicables, les performances observées sont souvent décevantes. Dans ce cas, le développeur est contraint de guider le compilateur en réorganisant son algorithme et les structures mémoires associées. Une autre option consiste à s'appuyer sur les extensions de langages (intrinsics) permettant une meilleure prise en compte de l'architecture sous-jacente. Dans ce cas, la portabilité de l'application peut-être significativement impactée. Enfin, des facteurs variés influencent significativement le design des

architectures déployées pour la simulation à haute performance. Par exemple, l'amélioration continue des techniques de gravure permet aujourd'hui de disposer de processeurs comportant plusieurs dizaines de cœurs. De plus, les besoins croissants en ressources de calcul des domaines liés à l'intelligence artificielle accélèrent l'émergence de systèmes hybrides associant processeurs généralistes et accélérateurs. La nécessaire maîtrise de la consommation énergétique de ces grands systèmes constitue également un challenge.
Cette thèse se situe dans le domaine du calcul haute performance et vise à explorer l'impact de différentes caractéristiques des architectures modernes sur les applications géoscientifiques. Pour cette étude, une classe généraliste de noyau de calcul est d'abord considérée. En effet, les stencils constituent d'excellents proxys des codes de calcul issus de la méthode des différences finies. Cette dernière est largement utilisée, notamment dans le domaine du risque sismique, de l'hydrogéologie ou pour la modélisation des écoulements complexes. Dans un deuxième temps, un noyau de calcul extrait de l'application Efispec3D est étudié. Cette application trouve son utilité pour l'évaluation du risque sismique et repose sur la méthode des éléments spectraux. Les contributions de cette thèse se situent donc à plusieurs niveaux. Une analyse approfondie des performances applicatives est menée afin de caractériser les performances. Cette analyse s'appuie en partie sur les modèles théoriques de coût tels que le modèle roofline. De plus, afin d'accroître la généricité des optimisations proposées, différents mécanismes d'abstraction ont été introduits. Enfin, sur le plan algorithmique, une technique de composition de stencils a été introduite. En offrant un cadre théorique robuste, cette dernière permet d'améliorer l'intensité arithmétique de cet opérateur.

Ce manuscrit est organisé en cinq chapitres. Le premier chapitre propose un cadre à l'étude avec une description des noyaux classiques issus des méthodes des différences finies et des éléments finis. Il propose également une revue des stratégies classiquement implémentées et propose un positionnement des travaux décrits dans les chapitres suivants. Le chapitre 2 s'intéresse aux stencils de calcul et propose en introduction une étude détaillée des performances applicatives. L'adéquation entre les résultats expérimentaux et le modèle de coût théorique est notamment discutée. La méthode de composition permettant d'améliorer l'intensité arithmétique est également décrite dans cette section.

Le chapitre 4 aborde les verrous spécifiques à la méthode des éléments spectraux. Il s'agit notamment d'évaluer l'impact de la vectorisation. Un ensemble de techniques spécifiques est également introduit afin de profiter du contexte spécifique des modélisations géoscientifiques notamment en termes d'accès aux données.

Chapitre 2

État de l'art

Cette thèse se situe dans le milieu applicatif des simulations scientifiques pour les géosciences. Ainsi, dans ce chapitre, nous allons passer en revue l'évolution conjointe des simulations numériques et des ordinateurs. Dans un premier temps, nous présenterons ce qu'est un modèle physique et différentes méthodes de résolution classiques en calcul numérique. Les ordinateurs ont beaucoup évolué depuis leurs premiers pas. On peut dire que les différents secteurs applicatifs ont façonné leur évolution. Cette thèse s'intéresse aux moyens de bénéficier implicitement de ces évolutions afin de les rendre accessibles aux utilisateurs géoscientifiques. Ainsi, dans un second temps, nous retracerons les différentes avancées de ces architectures numériques ainsi que les contraintes techniques de programmation associées. Ceci va nous emmener à une problématique d'évaluation croisée entre l'algorithme et la machine. Enfin, nous pourrons positionner ces travaux de thèse en partant de ces constats.

2.1 Modèles de simulation et noyaux de calculs

Nous allons voir dans cette section comment modéliser et implémenter des phénomènes physiques. Pour ce faire, nous commencerons par expliquer la notion de modèle. Nous nous intéresserons aux équations aux dérivées partielles. Alors, nous aborderons 2 noyaux classiques de résolution approchée de ces équations. Nous verrons que ces méthodes impliquent un nombre élevé de calculs pour atteindre des résolutions acceptables sur des domaines de taille suffisante. La quantité de calculs qui découle de ces résolutions nous mène au calcul par ordinateur.

2.1.1 Du modèle continu de la physique aux approximations discrètes

La modélisation d'un phénomène physique consiste à en créer une représentation plus ou moins simplifiée en se basant sur les propriétés du phénomène. Lorsque l'on s'intéresse aux valeurs physiques mesurables d'un phénomène, il s'avère pratique de le décrire par des contraintes mathématiques. Les mathématiques nous permettent d'exprimer des relations et contraintes entre les différentes valeurs physiques considérées. Ainsi, l'espace et le temps peuvent être mis en relation de façon continue avec une température, une pression, une force et toute autre nature de valeurs impliquées dans le modèle. Une relation physique peut ainsi s'exprimer mathématiquement via une Équations aux Dérivées Partielles

(EDP). L'équation 2.1 est un exemple d'EDP qui modélise l'évolution thermique dans l'espace et le temps avec $u(x, y, z, t)$ la température aux coordonnées $(x; y; z)$ à l'instant t.

$$\frac{\partial u(x,y,z,t)}{\partial t} = \frac{\partial^2 u(x,y,z,t)}{\partial x^2} + \frac{\partial^2 u(x,y,z,t)}{\partial y^2} + \frac{\partial^2 u(x,y,z,t)}{\partial z^2} \tag{2.1}$$

Il est alors plus simple d'approcher la solution exacte par une solution discrétisée en espace et en temps. Il existe plusieurs méthodes d'approximation numérique. L'annexe A présente un exemple de discrétisation via la méthode des différences finies. De ces méthodes d'approximation découlent des noyaux de résolution numérique. Dans cette thèse, 2 classes de noyaux de calcul sont abordées. La sous-section suivante présente la classe de noyaux stencils. Elle est suivie par la classe de noyaux élément fini.

2.1.2 La classe de noyaux stencils

Nous venons de voir que la résolution discrète peut se formuler et se calculer à la main. Toutefois, quelle que soit la méthode de discrétisation, les calculs sont démultipliés selon le nombre de dimensions et la finesse de définition du domaine. Bien que, les calculs sont réalisables à la main, ils nécessiteraient une armée de personnes bien organisées afin de les évaluer manuellement. Les processeurs numériques possèdent des armées de transistors capables de réaliser des calculs à haute cadence. Ainsi, les simulations sont tout naturellement devenues numériques pour y être exécutées sur silicium. Nous allons maintenant aborder notre première classe d'algorithme numérique. La résolution numérique par les noyaux Stencils.

Calcul séquentiel d'un noyau stencil explicite

Les stencils sont des schémas numériques résultant essentiellement de la méthode par différences finies. Cependant, il est tout à fait possible qu'un stencil soit issu de la méthode par volumes finis ou éléments finis. Commençons par une implémentation séquentielle simple d'un solver numérique employant un schéma numérique explicite. Pour notre exemple, la résolution se fera via un stencil explicite correspondant à une somme pondérée d'un voisinage. Le domaine de calcul est une grille 3D cartésienne des valeurs physiques. La résolution pour notre domaine est réalisée par applications successives du stencil. La fonction 2.1 détermine les nouvelles valeurs en chaque point du domaine pour chaque itération. Les bords du domaine se caractérisent par un manque de points relatifs nécessaires à l'application du stencil. Aux bords du domaine, une fonction adaptée permet de composer avec les seules informations disponibles en ces points (voisinage restreint, valeur par défaut ...). En dehors des bords, le calcul stencil s'applique par pondération d'un voisinage.

Nous venons de voir une implémentation séquentielle d'un calcul stencil à schéma explicite. Il s'agit d'une implémentation naïve, n'exploitant pas les spécificités de l'architecture cible. Nous avons donc mené des travaux dans ce sens.

```
float *appliquer_stencil(
  float *domaine_t,
  Stencil3D &s)
{
 float *domaine_t1 = allouer_domaine();
 for(int z=0;z<dom_taille_z;++z)
  for(int y=0;y<dom_taille_y;++y)
   for(int x=0;x<dom_taille_x;++x)
    if(est_au_bord(x,y,z))
     domaine_t1[cord(x,y,z)] =
      cond_bords(&domaine_t[cord(x,y,z)],s);
    else
     domaine_t1[cord(x,y,z)] =
      fonc_stencil(&domaine_t[cord(x,y,z)],s);
 return domaine_t1;
}
```

FIGURE 2.1 – Implémentation séquentielle simple du calcul stencil explicite.

Travaux sur l'optimisation des noyaux stencils

L'un des aspects critiques des stencils concerne les mouvements mémoires. Des efforts significatifs ont été menés pour améliorer les performances de ces noyaux selon différentes architectures [38, 22, 55]. Évidemment, ces efforts portent principalement sur l'amélioration de la réutilisation des données chargées en mémoire cache voire en registre. Parmi elles, les techniques de tuilage spatial offrent un premier niveau d'optimisation, mais ces approches souffrent de plusieurs limitations. Tout d'abord, l'efficacité de cette approche est limitée en raison de la très faible possibilité de réutilisation inhérente aux Stencils. Dans [53], l'accélération limitée obtenue via de telles stratégies est décrite dans une boucle de Jacobi tridimensionnelle (17%). De plus, l'impact des mécanismes de bas niveau tels que le préchargement anticipé des données en cache, la vectorisation ou l'utilisation efficace des caches mémoire sont sous-estimées comme décrit dans [17, 4].
Pour surmonter les limites du tuilage spatial, les efforts ont été concentrés sur le tuilage du domaine de calcul dans l'espace et dans le temps. Dans ce cas, les algorithmes exploitent la nature itérative du noyau. Ce sont des algorithmes de parcours par tuilage temporel. Les algorithmes d'asymétrie temporelle ou de cache-oblivious [25] s'appuient sur une idée similaire pour effectuer plusieurs pas de temps dans chaque sous-domaine. La principale différence vient du critère de blocage explicite dans le premier cas alors que l'approche implicite appelée oblivious exploite une coupe récursive dans l'espace-temps afin de tenir au mieux dans les mémoires caches. La recherche de la forme optimale de la tuile spatio-temporelle est également un point critique. La forme appelée tuile diamant présente de bonnes performances [42, 48]. Afin de réaliser un usage optimisé des ressources d'architectures modernes, l'implémentation du noyau de calcul se doit d'être vectorielle. Cependant, la décomposition spatio-temporelle complique l'optimisation automatique par les compilateurs. De plus, les implémentations au niveau applicatif du calcul vectoriel restent délicates, car l'organisation du code doit être profondément revue.

Beaucoup de travail a également été fait sur les techniques d'autoréglage. Plusieurs frameworks ont été introduits pour optimiser les calculs stencil sur les architectures mo-

dernes (par exemple PATUS [14] ou PLUTO[1]). Parmi eux, le compilateur de stencil Pochoir [61] a suscité beaucoup d'intérêt. Il implémente des calculs du stencil dans un langage spécifique au domaine (DSL) et exploite un algorithme de coupe hyperespace afin d'optimiser la réutilisation du cache et le parallélisme. Le compilateur Pochoir s'appuie sur la bibliothèque multithreading Cilk et sur le compilateur Intel pour vectoriser.

Au cours de la réalisation de nos travaux sur les stencils, d'autres solutions ont vu le jour. En effet, l'article [27] présente des outils et techniques plus récents que Pochoir. Parmi les outils, Girih[2] est un framework permettant la mise en oeuvre du calcul stencil optimisé à la compilation selon la plateforme cible. Le framework BOAST traité dans l'article [65] permet l'optimisation portable de plusieurs types de noyaux. Ces travaux soulignent l'intérêt de découpler la description du noyau des combinaisons d'optimisations à mettre en oeuvre selon le schéma de calcul et la plateforme cible.

Ainsi, les stencils pâtissent de la limite des performances mémoires. Nous allons voir que cela est aussi le cas pour les éléments finis.

2.1.3 La classe de noyaux éléments finis

Nous allons tout d'abord présenter la méthode éléments finis.

Le maillage

Un maillage est une représentation discrétisée du domaine physique de calcul. Les éléments unitaires (mailles) ne se recouvrent pas. En 2D ces mailles sont des polygones (triangles, quadrilatères), en 3D il s'agit de polyèdres (tétraèdres, prismes ou hexaèdres). Si tous les polyèdres ont les mêmes voisins indexables sur chaque dimension, on parle de maillage structuré, sinon de maillage non structuré. À chaque maille correspondent des grandeurs physiques (vitesse, accélération, température...) liées au domaine et au phénomène étudié. Enfin, les points du maillage sont appelés nœuds ou points physiques. Ainsi, un nœud peut être commun à plusieurs éléments adjacents (cf. la figure 2.2). Ce sont en ces nœuds que l'on souhaite avoir une solution approchée de notre EDP.

En géosciences, le domaine d'étude peut correspondre au sous-sol. Le raffinement du maillage de chaque couche géologique va dépendre des caractéristiques physiques de chaque couche. Un tel maillage est de fait non structuré de par la gestion des interfaces entre les couches géologiques (cf. l'illustration 2.3) Nous verrons plus loin l'impact de la structure du maillage sur les algorithmes et leurs performances de calcul.

Ordre d'approximation

Résoudre une EDP par la méthode des éléments finis revient à trouver une fonction approchée, combinaison linéaire de fonctions de base. Ces fonctions de base vont dépendre de la forme des éléments finis, de la physique résolue et de l'ordre d'approximation souhaité. Elles sont liées à des points appelés points d'intégration.

1. PLUTO code disponible à l'adresse http://plutocode.ph.unito.it.
2. Framework Girih disponible à l'adresse https://github.com/ecrc/girih.

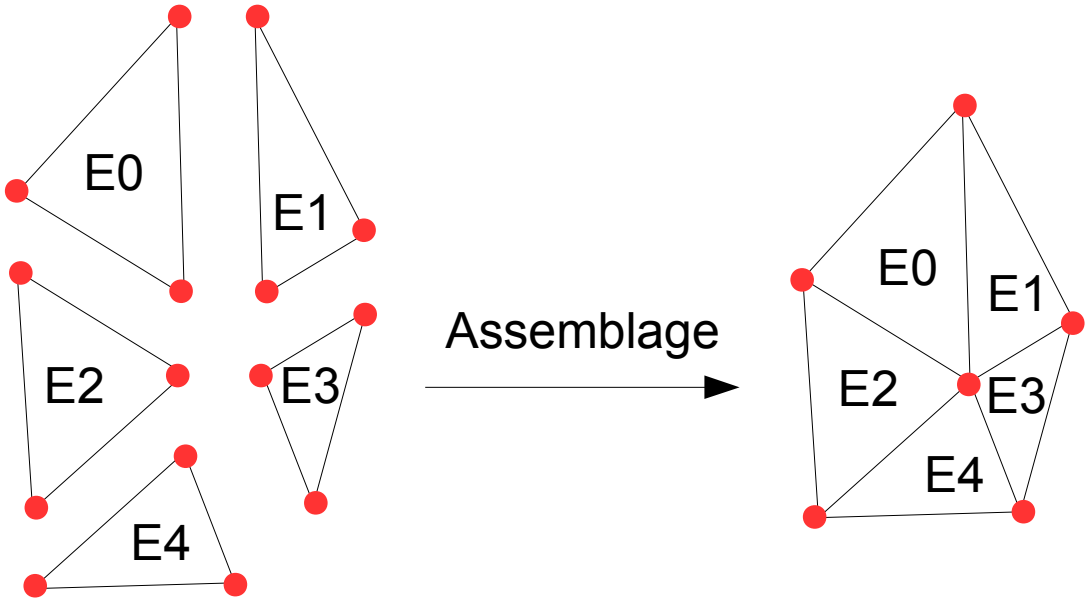

FIGURE 2.2 – Assemblage d'éléments finis.

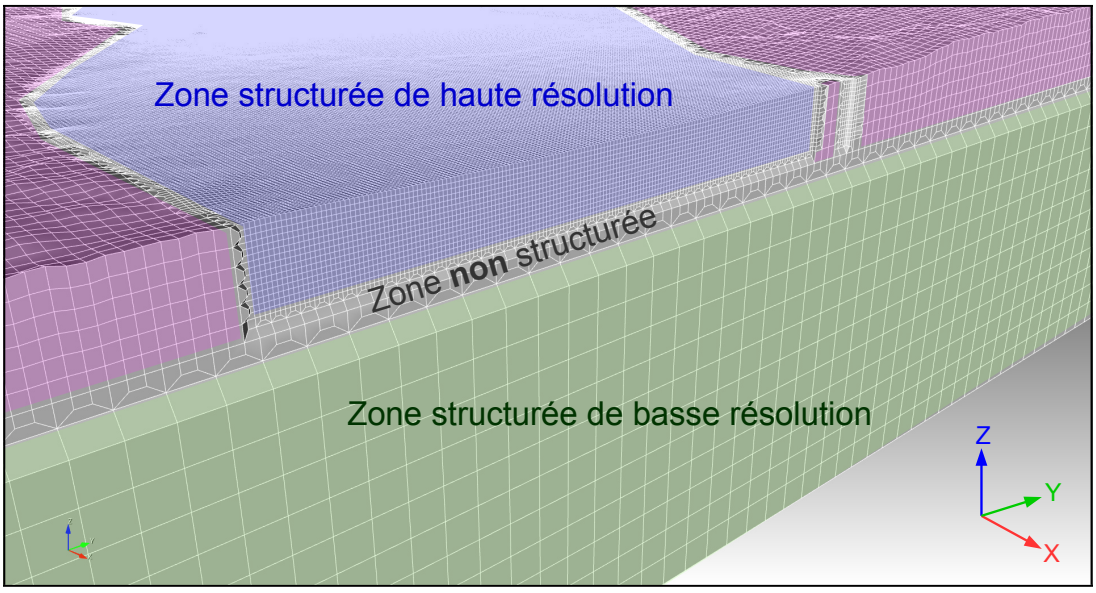

FIGURE 2.3 – Illustration d'un terrain formé par son maillage volumique. Il s'agit du bassin sédimentaire de Volvi en Grèce. Ce rendu 3D est assuré par le logiciel CUBIT disponible à l'adresse https://www.osti.gov/biblio/10176386.

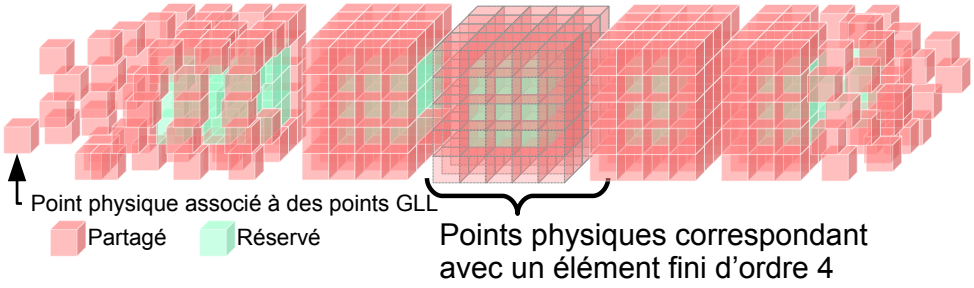

FIGURE 2.4 – Illustration d'une grille régulière de points physiques associés à des éléments finis d'ordre 4 dont l'épaisseur de partage est d'un point physique.

Le nombre de points d'intégration varie selon l'ordre d'approximation. Un ordre n implique $n+1$ points d'intégration ce qui en 3D donne $(n+1)^3$ points par élément fini. Dans le cas des éléments spectraux, les fonctions de base sont des polynômes d'ordre élevé (polynômes de Legendre), les points d'intégration sont les points de Gauss-Lobatto-Legendre (GLL). Ainsi, on peut voir dans l'illustration 2.4 qu'un élément fini spectral d'ordre 4 se compose de $5*5*5 = 125$ points GLL et ces derniers correspondent avec 125 points physiques du maillage du domaine physique dont on réalise la simulation.

L'ordre d'approximation a également un impact sur le temps de calcul. En effet, pour des raisons de stabilité numérique il est nécessaire de réduire le pas de temps quand l'ordre d'approximation augmente. Cependant, nous observons plus loin dans ce chapitre que le calcul d'éléments d'ordre plus élevé réutilise plus de données provenant des caches. Ainsi, l'efficacité de calcul étant accrue, cela réduit le surcoût induit par l'augmentation de l'ordre d'approximation.

Maintenant que nous avons vu la composition GLL des éléments, la sous-section suivante présente ce qu'implique techniquement un maillage structuré ou non structuré.

Maillage structuré et non structuré

Un maillage 3D pour la simulation peut être structuré ou non structuré.

Afin de définir ce qu'est un maillage structuré, nous prenons la définition sur le site mecagom de l'Université de Technologie de Compiègne :
« Un maillage est dit structuré lorsque la localisation des nœuds qui le constituent est définie par des indices, le nombre d'indices étant égal à la dimension géométrique du problème (un en 1D, deux en 2D et trois en 3D). Graphiquement, ces maillages se caractérisent sous la forme de 'grilles' de nœuds. La connaissance d'un nœud par ses indices permet facilement de connaître ses nœuds voisins en incrémentant et décrémentant ses indices. Le maillage structuré est généré en reproduisant plusieurs fois une forme de maille élémentaire. [3] »
Les maillages structurés nous contraignent à avoir une résolution constante. Nous verrons par la suite qu'ils s'intègrent assez naturellement dans nos architectures matérielles.
De leur côté, les maillages non structurés autorisent plus de souplesse notamment en permettant de changer de résolution. Plus généralement, ils permettent une adaptation du

3. Définition de la notion de maillage structuré http://www.utc.fr/~mecagom4/MECAWEB/EXEMPLE/FICHES/MASAF1.htm.

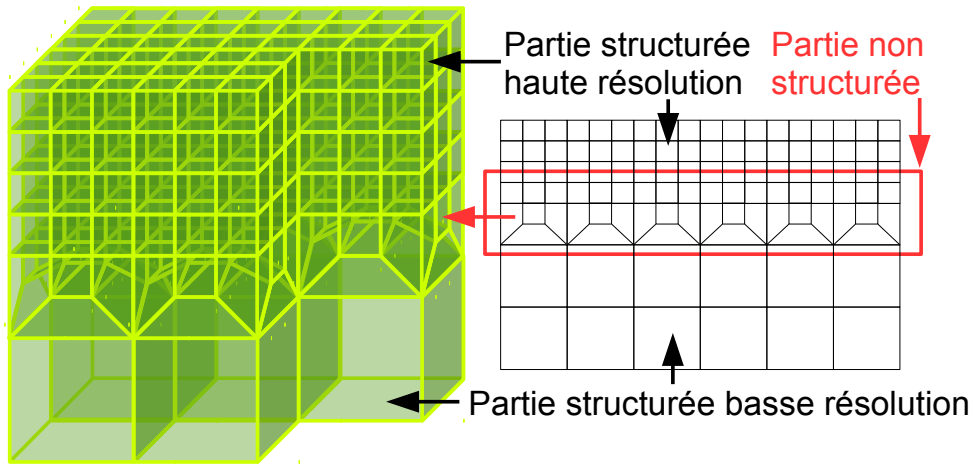

FIGURE 2.5 – Exemple d'interface de maillage bi-dimensionnel entre deux niveaux de résolution.

maillage au domaine. La figure 2.5 présente un exemple de maillage non structuré. Dans cette illustration, un changement de résolution entre 2 parties structurées impose dans ce maillage la présence de polyèdres dont l'interface diffère. Par conséquent, l'ensemble du maillage est non structuré.

La méthode des éléments finis peut être implémentée sur des maillages structurés ou non structurés. Cependant, la performance peut en pâtir.

Structure des données de calculs

Nous venons de voir qu'un maillage 3D se compose de polyèdres ayant des interfaces communes. À chaque polyèdre correspond un élément fini lequel contient des points d'intégration (points liés à l'ordre d'approximation). L'objet de la simulation consiste à calculer les évolutions de valeurs physiques. En effet, l'objet d'une simulation peut être de calculer la température ou toute autre unité mesurables en un point après plusieurs secondes en partant d'un état initial connu. Ces grandeurs physiques évoluent suivant une intégration continue locale de chaque sous-domaine délimité par chaque élément fini. Par conséquent, chaque élément fini permet la mise à jour de grandeurs physiques aux points physiques correspondants avec les points d'intégration. De fait, la structure de données doit identifier les éléments ainsi que leurs compositions en points d'intégration référençant chacun un point physique correspondant. De plus, chaque point d'intégration peut avoir des paramètres qui leur sont propres. En effet, tous les polyèdres ne sont pas nécessairement identiques que ce soit en type ou en taille. En résumé, la structure mémoire pour une simulation d'un terrain est une liste d'éléments à résoudre de pas de temps en pas de temps. Nous prendrons l'exemple de la structure de données définie par le diagramme 2.6. Elle correspond à une structure de données destinée à la simulation par éléments spectraux d'ordre 4.

Différentes implémentations mémoires de cette structure de données sont possibles. Il est possible d'implémenter la structure en SoA (Structure of Array). Par conséquent,

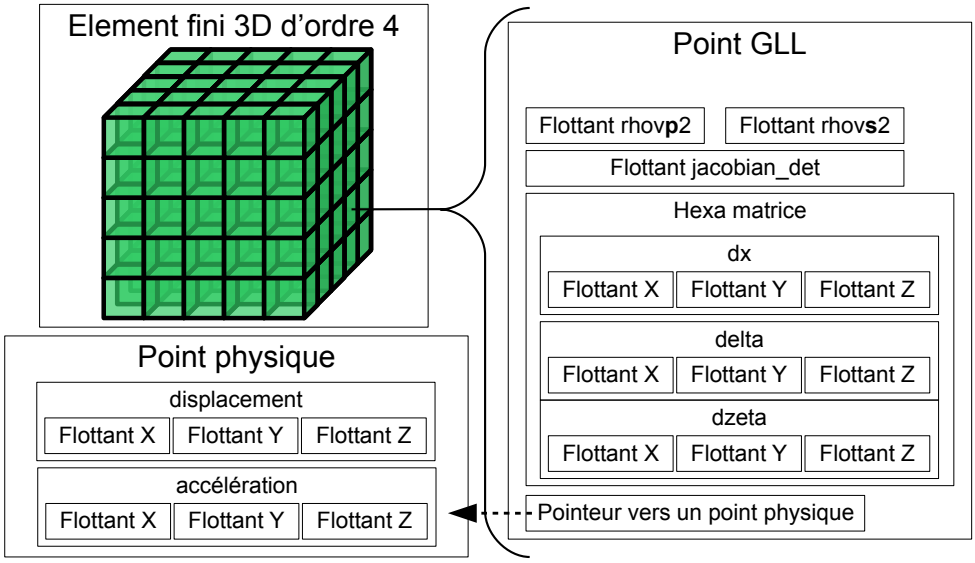

FIGURE 2.6 – Exemple de structure abstraite des données de simulation d'éléments finis spectraux d'ordre 4. Les points physiques représentent les valeurs physiques d'intérêt de la simulation. Ils sont référencés via le paramétrage des points GLL (points d'interpolation) propres à chaque élément fini. La simulation requiert donc une liste d'éléments finis et une liste de points physiques.

l'implémentation possède autant de tableaux de flottants qu'il y a d'attributs comme le montre l'exemple de code 2.7. Toutefois, il est aussi possible de regrouper les données dans des structures et d'en faire des tableaux. Ainsi, contrairement au SoA, un tableau de structure AoS (Array of Structure) contient une suite de données hétérogènes comme dans l'exemple 2.8.

Les travaux de l'article [33] ont évalué ces 2 approches pour le noyau d'une application de simulation par éléments finis. Ils concluent à un faible impact sur les performances.

La structure de données 2.6 précédemment décrite est faite pour simuler des maillages non structurés. Ceci implique des irrégularités dans le partage des points physiques entre éléments finis. Ainsi, le fait de pouvoir référencer les points physiques dans la composition des éléments assure la souplesse nécessaire. Cependant, l'utilisation de références aux points physiques implique un accès indirect à leurs données. D'une part, la référence en elle-même a un coût mémoire et nécessite un accès en 2 temps par points d'interpolations. D'autre part, les points d'interpolations composant un élément peuvent se retrouver physiquement plus ou moins dispersés en mémoire. Or, dans la sous-section 2.2.1 suivante est expliqué que l'accès à des données non contigües peut réduire l'efficacité de la bande passante mémoire.

En un premier temps, on cherche une solution pour rassembler au mieux les données. En effet, on peut profiter de la souplesse du référencement. De fait, il est possible de réorganiser les données afin d'améliorer leurs contiguïtés. Il existe pour cela des approches telles que l'algorithme de Cuthill-McKee [16, 37] cherchant à optimiser la proximité des données selon leurs dépendances. Nous considérons dans nos travaux que le partition-

```
//Données de tous les points d'interpolations de la simulation.
float *displacementX,*displacementY,*displacementZ;
float *accelerationX,*accelerationY,*accelerationZ;

//èParamtres de tous les points d'interpolations
//pour chaque élément fini.
float *rhovp2,*rhovs2,*jacobian_det;
float *dxidx,*dxidy,*dxidz;
float *detdx,*detdy,*detdz;
float *dzedx,*dzedy,*dzedz;
//Références des points physiques de tous les points d'interpolations
//pour chaque élément fini.
unsigned int *glonum;
```

FIGURE 2.7 – Exemple d'implémentation d'une structure par SoA.

```
//Données de tous les points d'interpolations de la simulation.
struct _SV3D{float x,y,z;};
struct _SPInterp{_SV3D disp, accel;};
//èParamtres d'un point d'interpolation.
struct _SParamRefPPhy
{
 float rhovp2, rhovs2;
 float jacobian_det;
 _SV3D dxid, detd, dzed;
 //Référence d'un point physique.
 unsigned int idRefPPhy;
};
struct _SElmtO4 //Un élément d'ordre 4
{
 //Se compose de 125 références paramétrées
 _SParamRefPPhy prmRfPPhy[5*5*5];
};
_SPInterp *glls;
_SElmtO4 *elements;
```

FIGURE 2.8 – Exemple d'implémentation de la structure de données par AoS.

ner METIS se charge déjà de réordonner et d'optimiser les données. Ainsi, un des axes d'optimisation présentés dans cette thèse concerne le caractère hybride des maillages partiellement structurés. La sous-section suivante concerne les dépendances de données entre éléments finis.

Notions de voisinage et accès concurrents

Nous venons de voir l'implantation des données en mémoire. Les points physiques peuvent se retrouver référencés par plusieurs éléments finis. Or, chaque itération d'élément fini implique une mise à jour de ses points associés. Par conséquent, des éléments partageant les mêmes points physiques ne peuvent être mis à jour au même moment. Ainsi, des liens de dépendance de voisinage entre les éléments sont induits par le fait qu'ils partagent au moins un point d'interpolation en commun. Il en découle que les mises à jour en parallèle d'éléments finis liés entre eux se doivent d'être synchronisées. Cependant, comme dans le calcul stencil du chapitre 3, la donnée d'entrée du calcul de notre noyau est différente de la donnée de sortie. Ainsi, il est possible d'employer un parallélisme asynchrone (multi-cœurs) à condition d'assurer la synchronisation des données dépendantes.

Travaux sur l'optimisation des noyaux éléments finis

Des travaux d'évaluation de la performance d'un noyau numérique par éléments finis ont déjà été menés. En effet, des approches assez simples telles que le benchmark Mantevo [43] et l'optimisation de la performance applicative sont issues d'efforts continus du fait de l'émergence de nouvelles architectures (capacités croissantes de vectorisation par exemple).

Outre les problèmes scientifiques, l'un des défis principaux consiste à faire face à des percées majeures, tant du côté matériel que logiciel, qui conduiront la communauté à l'Exaflops. Par exemple, le mur de consommation d'énergie, pour des systèmes construits avec plusieurs millions de noyaux hétérogènes, reste un sujet ouvert. Par conséquent, l'évolutivité et l'efficacité (calcul et mouvements de mémoire) des applications actuelles sont critiques.

En ce qui concerne la modélisation des séismes sur des systèmes à mémoire distribuée, plusieurs références ([54, 64, 63]) soulignent le passage à l'échelle des performances des formulations explicites pour résoudre l'équation élastodynamique. Dans ce cas, nous bénéficions d'un nombre limité de communications point à point entre sous-domaines MPI voisins.

Des travaux importants ont été réalisés pour étendre ce parallélisme aux processeurs ([26, 12]) hétérogènes et de faible puissance, principalement pour la méthode des différences finies. Par exemple, les stratégies ([60, 46]) d'auto-tuning et d'apprentissage machine ont été envisagées pour explorer de larges espaces de paramètres d'optimisation (flags de compilation, tuilage espace-temps, mappage des threads et des allocations mémoire). Ces approches n'ont pas encore fait l'objet d'une étude approfondie pour les méthodes par éléments finis d'ordre élevé. Ceci est probablement dû à la complexité de

l'implémentation de tels noyaux.

De plus, peu d'articles traitent de parallélisme de bas niveau pour les noyaux basés sur les éléments finis ([8, 23, 62, 13, 44]). Parmi ces contributions, les algorithmes de coloration des maillages ont suscité beaucoup d'intérêt ces dernières années en raison de leur capacité à réduire les synchronisations nécessaires au maintien de la cohérence des mémoires partagées. En contrepartie, l'attaque du calcul suivant un ordonnancement issu du coloriage n'optimise pas les caches mémoires. Dans l'article [58], nous avons étendu cette approche à l'assemblage FEM parallélisé sur des architectures multi-cœurs en implémentant des lots d'éléments colorés ainsi que des stratégies de vectorisation avancées.

Enfin, il est possible de modifier l'implémentation de la structure des données comme décrite en détail dans l'article [33]. Dans cette référence, les auteurs ont comparé la présentation des données SOA (Structure of Arrays) à l'AOS (Array of Structures). Cette stratégie montre un impact limité sur les performances qui sont principalement régies par l'efficacité au niveau du parallélisme SIMD. Si nous exploitons les connaissances de la physique en ce qui concerne des géométries spécifiques (par exemple en géosciences), nous pouvons bénéficier d'approches hybrides qui combinent un maillage structuré (avec accès direct aux données pour la partie principale du calcul) et des maillages non structurés (avec accès indirect aux données). Cette approche a été mise en œuvre avec succès dans les travaux présentés par l'article [30].

Les travaux réalisés sur les éléments finis participent aux évolutions des applications utilisant de tels noyaux de calcul. Parmi ces dernières, l'application Efispec3D [70] [4] implémente une résolution de l'équation d'élastodynamique par assemblage d'éléments spectraux. Une partie des travaux de cette thèse porte sur cette application, développée en fortran95. D'autres applications telles que SPECFEM3D [35] implémentent également une résolution par assemblage d'éléments spectraux de cette équation. Le framework BOAST précédemment cité via l'article [65] s'applique également à l'optimisation du noyau de l'application SPECFEM3D. Du fait de ces avancées, des applications peuvent ainsi proposer des solutions de simulations précises à grande échelle [37].

Nous venons de voir les noyaux stencils et éléments finis. On remarque qu'ils ont des points communs : ces implémentations requièrent la connaissance locale d'un certain voisinage. Nous verrons par la suite comment il est possible d'allier les contraintes spécifiques des calculs aux capacités et contraintes matérielles des machines.

2.2 Architecture des ordinateurs

Les noyaux de calcul que nous venons de présenter effectuent quasi exclusivement des opérations arithmétiques. Leurs accès à la mémoire peuvent être réguliers ou irréguliers selon les cas. Afin de les optimiser pour les architectures actuelles, il convient de bien comprendre le fonctionnement des processeurs et de leurs différentes unités de calcul et de gestion mémoire. Les machines ont évolué depuis l'EDVAC [5] dont l'architecture conçue

4. Site web d'Efispec3D à l'adresse http://efispec.free.fr
5. EDVAC :https://fr.wikipedia.org/wiki/Electronic_Discrete_Variable_Automatic_Computer

par John Von Neumann se résumait à une unité de calcul, de traitement et de mémoires. Elles ont été améliorées et miniaturisées afin d'être plus performantes, moins énergivores et moins coûteuses. Cependant, ces améliorations viennent avec une complexification de l'architecture : ajout d'unités de calcul, d'asynchronisme, de parallélisme, de niveaux de mémoire cache, etc, rendant l'exploitation de leur performance de plus en plus difficile. Il est important de noter que les compilateurs ne sont pas capables d'effectuer toutes ces optimisations automatiquement, particulièrement l'exploitation du parallélisme, et que le processus d'optimisation reste un travail de spécialiste.

Les architectures utilisées dans cette thèse se composent de plusieurs processeurs, d'unités vectorielles, et sont représentatives de celles présentes dans les stations de travail et des supercalculateurs actuels. Pour bien comprendre et justifier les optimisations considérées dans les chapitres suivants, nous commençons par présenter le fonctionnement d'un processeur en commençant par son aspect le plus simple et en ajoutant les différents niveaux de parallélisme ensuite. La figure 2.9 présente une vision simplifiée d'un processeur multi-cœurs, nous commençons donc par présenter le fonctionnement d'un seul cœur.

2.2.1 Fonctionnement d'un processeur scalaire

Le fonctionnement de nos processeurs modernes est assuré par un assemblage de nombreuses mémoires et unités de traitement distinctes. Elles permettent entre autres le décodage des instructions, leurs exécutions, l'anticipation des instructions suivantes ainsi que la gestion des accès mémoires. La compréhension du fonctionnement de ces dernières est nécessaire à l'optimisation des codes de calculs présentés dans cette thèse ainsi qu'à leurs analyses.

Dans cette section, nous allons premièrement présenter le fonctionnement de la mémoire cache, puis détailler les parties dites *front-end* (ou *fetch and decode*) et *back-end* qui composent chaque cœur.

La mémoire cache

Une première piste pour accélérer le fonctionnement d'un programme est de rapprocher des blocs de données le plus possible des unités de calcul. La mémoire cache est une mémoire présente dans le processeur, beaucoup plus rapide que la mémoire vive (RAM), mais dont l'intégration limite la quantité. Elle sert de mémoire intermédiaire entre la RAM et les registres. L'accès à une donnée par le processeur ne se fait plus directement à partir de la mémoire, mais à travers cette mémoire cache. Une lecture d'une donnée en RAM déclenche la copie d'un bloc de mémoire plus large contenant cette donnée vers la mémoire cache, la donnée est ensuite extraite de la mémoire cache vers un registre. L'ajout de mémoire cache est justifié par l'observation suivante : il est fréquent qu'un programme accède plusieurs fois aux mêmes données, de plus, ces accès sont souvent contigus. Cela est entre autres dû à des parcours séquentiels de tableaux ou de structures de données, ce qui est le cas pour le code de stencil considéré dans cette thèse. Le cache permet donc d'optimiser les codes présentant une forte localité spatiale (contiguïté) et/ou temporelle (réutilisation) dans leurs accès aux données. Il en va de même pour les instructions des programmes exécutées les unes après les autres, sauf dans le cas de branchements (structures conditionnelles, boucles). En effet, ces dernières s'exécutent le plus souvent à la suite

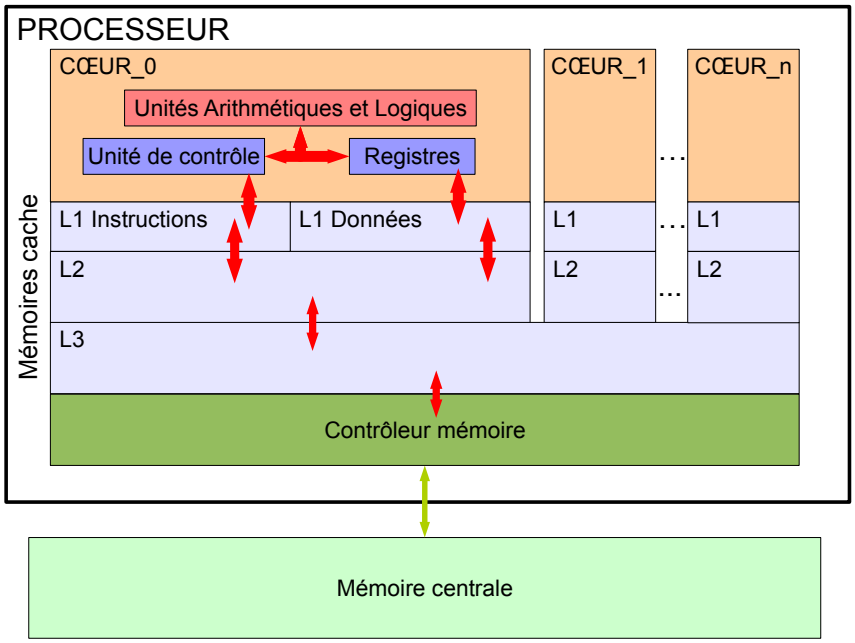

FIGURE 2.9 – Diagramme hiérarchique typique d'un processeur moderne avec la mémoire centrale.

et peuvent être réexécutées plusieurs fois dans des boucles. Afin d'améliorer l'efficacité de la mémoire cache, il faut que sa vitesse soit proche de celle des registres et que sa taille soit la plus importante possible. Cependant, plus la mémoire cache est rapide et plus elle est coûteuse et donc disponible en quantité limitée. Une solution est de la découper en plusieurs parties également appelées niveaux. On intègre généralement une petite quantité très rapide appelée cache L1 au plus près des unités de calcul, puis un cache L2 plus large, mais plus lent, et éventuellement d'autres niveaux. À chaque lecture en mémoire, un bloc de données transite de la RAM au cache L1 en passant par les niveaux intermédiaires.

Nous détaillons maintenant plus techniquement l'architecture et le fonctionnement de la mémoire cache des processeurs actuels. Ceux-ci disposent généralement de trois niveaux de cache : L1, de 32ko à 64ko, L2, de 512ko à 2Mo, L3, de quelques Mo à quelques dizaines de Mo. Notons que les niveaux L1 et L2 sont propres à chaque cœur alors que le niveau L3 est partagé par tous les cœurs. Les données circulent donc par blocs entre la RAM et entre les niveaux de mémoires cache. Ces blocs sont typiquement constitués de 64 octets sur les architectures actuelles et sont appelés *lignes de cache*. Par conséquent, la mémoire centrale est accédée sur la base d'une adresse mémoire alignée sur 64 octets pour ainsi transférer toute une ligne. Lorsque le processeur effectue une lecture ou une écriture de données à une adresse mémoire, la mémoire cache est tout d'abord interrogée. Si la ligne comprenant cette adresse n'est pas chargée en cache L1, un défaut de cache se déclenche afin de la récupérer depuis un autre niveau et le cas échéant depuis la RAM. Lorsque la ligne de cache est disponible en cache L1, les données sollicitées peuvent être extraites et envoyées dans des registres. L'accès à d'autres données présentes dans la même ligne de cache L1 nécessite donc juste l'extraction de celles-ci de la ligne de cache vers le registre. Ce fonctionnement est aussi valable pour les instructions, qui disposent d'un cache L1 dédié, mais partagent les autres niveaux. Dans le cas où un niveau de mémoire cache est plein, et

qu'une nouvelle donnée est demandée, une ligne de cache doit être libérée avant l'insertion de la nouvelle ligne. Ce mécanisme appelé éviction est basé sur des politiques prenant en compte les dates d'accès (politique LRU, Least Recently Used) ou les fréquences d'accès (politique LFU, Least Frequently Used) aux lignes de cache.

Le mécanisme de la mémoire cache est généralement utilisé en combinaison avec un mécanisme de prédiction d'accès aux données, implanté dans une unité appelée *prefetcher*. Cela permet d'améliorer encore les performances en anticipant les données et les instructions qui vont être accédées plus tard par le processeur. Si la prédiction est effectuée correctement, la donnée est déjà disponible en cache lorsqu'une instruction va effectivement y accéder. Cette technique exploite le fait que les communications avec la mémoire peuvent se faire de manière asynchrone avec les unités de calcul lorsque les données sont indépendantes. La prédiction peut être basée sur des heuristiques, suivant des schémas d'accès connus comme le fait d'itérer sur un tableau, ou sur une analyse statistique des accès mémoire. Ce mécanisme est donc plus adapté au calcul de type stencil dont le motif d'accès aux données est très régulier, et par contre beaucoup moins adapté à la méthode des éléments finis ou les éléments sont reconstitués à partir de points accédés par un tableau d'indirections. Dans ce dernier cas, en effet l'adresse pour accéder à un point est calculée en fonction d'une valeur lue dans le tableau d'indirections.

Un exemple fréquemment utilisé pour illustrer l'impact de la mémoire cache et du *prefetcher* sur les performances est la multiplication de matrices. L'algorithme naïf consiste à effectuer le produit scalaire entre une ligne de la première matrice A et une colonne de la seconde matrice B pour obtenir une valeur de la matrice résultante C. Le parcours des matrices A et C se font de manière contiguë en mémoire. Chaque valeur de C est écrite une seule fois. Chaque ligne de A est réutilisée pour chaque élément d'une ligne de C donc si une ligne de A tient en cache, alors chaque ligne de A n'est chargée qu'une fois à partir de la mémoire. Par contre la matrice B est accédée de manière non contiguë. Le produit scalaire implique qu'une valeur de la première ligne est accédée, puis une valeur sur la ligne suivante et ainsi de suite jusqu'à la dernière ligne. Si la matrice B tient entièrement en cache, la matrice n'est copiée qu'une seule fois depuis la mémoire et on exploite bien le cache. Si la matrice B ne tient pas en cache, il faut la recharger autant de fois que de lignes de A si on suppose qu'une colonne de B tient en cache. En inversant les boucles internes, l'ordre des calculs change, mais le calcul effectué est toujours le même. Dans ce cas, la matrice B est accédée de manière contiguë et la ligne de cache suivante peut être préchargée pendant que les calculs sur les données de la ligne de cache courante sont effectués. Cependant chaque élément de la matrice C est accédée plusieurs fois pour accumuler les valeurs. Une autre optimisation consiste à transposer la matrice B avant d'effectuer le calcul de manière à combiner l'accès à la matrice B de la version précédente et une unique écriture pour chaque valeur de C de la première version.

D'autres mécanismes ou caractéristiques doivent également être considérés pour maximiser les performances des caches, par exemple leur associativité ou l'unité TLB (Translation Lookaside Buffer), cependant ces aspects ne sont pas abordés ici, car ils sont beaucoup moins prépondérants pour les types de calculs considérés dans cette thèse.

Maintenant que nous avons présenté le fonctionnement de la mémoire cache, nous allons nous intéresser au fonctionnement d'un cœur et de la manière dont il traite les instructions.

Traitement des instructions

Un programme est composé d'une suite d'instructions encodées suivant le jeu d'instruction du processeur. Ces instructions sont amenées de la mémoire jusqu'au cache L1 d'instructions puis traitées par le processeur. Pour décrire comment se passe le traitement des instructions, nous allons prendre l'exemple de la microarchitecture Skylake dont le diagramme se trouve à figure 2.10. Cette microarchitecture se compose de deux parties appelées *front end* (partie supérieure encadrée dans la figure 2.10) et *back end* (partie inférieure).

Le *front end* est également appelé unité *fetch and decode* car il se charge de récupérer chaque instruction, de la décoder et la transformer en micro-instructions qui seront ensuite traitées par le *back end*. Les instructions peuvent être de taille variable, il faut donc décoder l'instruction, déterminer ses opérandes et leurs tailles pour déterminer où commence l'instruction suivante et ainsi de suite pour chaque instruction. La production de micro-instructions permet de découper les instructions en sous-tâches potentiellement indépendantes qui pourront être réordonnancées plus tard afin de maximiser l'utilisation des unités du processeur. On peut considérer que les instructions complexes x86, de type CISC (Complex Instruction Set Computing) sont décomposées en instructions plus simples de type RISC (Reduced Instruction Set Computing). Par exemple, l'instruction `add` peut effectuer l'addition de deux valeurs stockées dans des registres, ou d'une valeur stockée dans un registre avec une valeur en mémoire. Dans ce dernier cas, une micro-instruction supplémentaire sera ajoutée pour charger la donnée avant d'effectuer l'addition. Les micro-instructions sont finalement placées dans une file d'attente avant d'être transmises au *back end*. On observe également sur la figure 2.10 que le *front end* possède également un cache de micro-instructions en sortie des décodeurs pour accélérer le décodage lorsqu'une instruction est exécutée à nouveau, par exemple si elle se trouve dans une boucle. Une unité de prédiction de branchement, *Branch Prediction Unit* est également présente afin d'anticiper l'instruction suivante à exécuter dans le cas de sauts, par exemple pour les structures conditionnelles ou les boucles.

Le *back end* récupère les micro-instructions dans la file du *front end* et les distribue aux différentes unités de calcul. Les micro-instructions vont passer par l'unité *Allocate/-Rename/Retire* qui va attribuer ou renommer des registres. En effet, les registres nommés en assembleur (`rax, rbx, rcx, rdx, rbp, ...`) ne correspondent pas à un registre physique et peuvent pointer vers différents registres du *register file*. Ainsi, une instruction assembleur `mov` entre deux registres n'induit pas forcément une copie si le registre origine n'est plus utilisé et peut donc être traduite par un simple renommage du registre. Les micro-instructions sont ensuite envoyées (*Retire*) au *scheduler* qui va les répartir sur les différents ports suivant les instructions effectuées et leurs dépendances. L'ordre d'exécution des micro-instructions à ce niveau n'est pas nécessairement conservé si elles sont indépendantes, on parle d'exécution *out of order*. De même, si deux instructions ont des opérandes distincts et peuvent être attribuées à des ports différents alors elles sont exécutées en parallèle. On parle dans ce cas de parallélisme d'instruction (*ILP : Instruction Level Parallelism (ILP)*) et on qualifie l'architecture de super-scalaire. Sur l'architecture considérée, il est possible d'effectuer une instruction de décalage (*shift*) sur le port 0 et la multiplication de deux entiers (*mul*) sur le port 1. Chaque unité va également prendre un certain nombre de cycles pour exécuter une instruction. Un décalage va par exemple nécessiter un seul cycle, alors qu'une multiplication va en nécessiter cinq. On parle alors

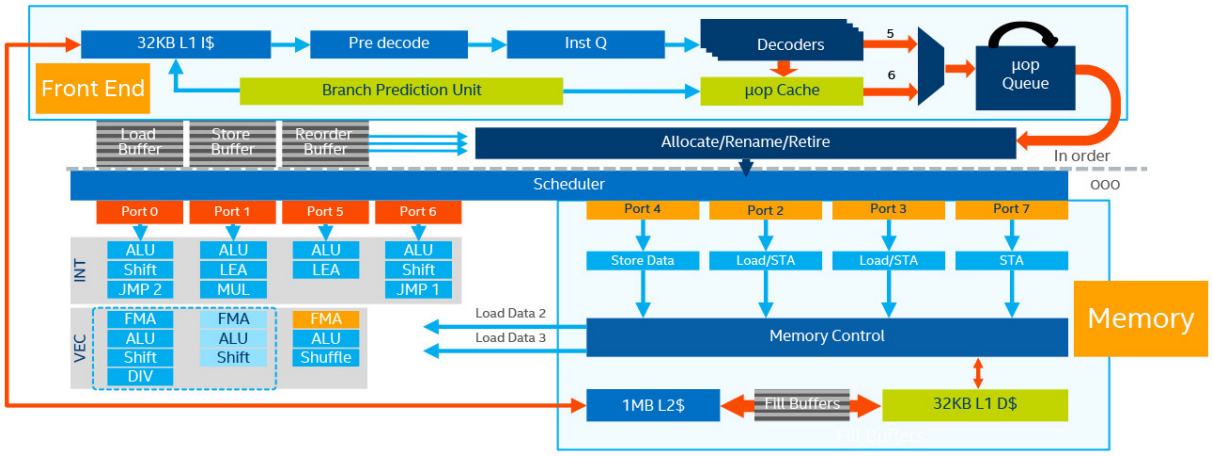

FIGURE 2.10 – Diagramme de la microarchitecture Intel Skylake. Image provenant du site : https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

de la latence de l'instruction qui va correspondre au nombre de cycles nécessaires entre l'entrée et la sortie de l'unité. Si plusieurs unités identiques sont présentes sur plusieurs ports, elles peuvent produire deux résultats par cycle. Dans notre exemple, il y a une unité de décalage sur le port 0 et une sur le port 6. Le *throughput* est défini comme l'inverse du nombre de résultats potentiellement produits par cycle. Dans le cas du décalage il est donc de $0,5$.

La connaissance de l'architecture cible permet d'apporter certaines optimisations au code considéré. Par exemple, il faut éviter de créer des chaînes d'instructions dépendantes, car cela va limiter l'efficacité du mécanisme *out of order*. Chaque instruction devra donc attendre que l'instruction précédente se termine, et donc le nombre de cycles correspondant à sa latence pour commencer. Il faut également éviter de créer de fausses dépendances par l'utilisation d'un même registre temporaire et utiliser plusieurs registres temporaires. De même, le déroulage d'une boucle dont les itérations sont indépendantes est généralement dépendant du nombre de registres disponibles ainsi que du throughput des instructions de manière à masquer la latence des instructions.

Nous venons de présenter le fonctionnement d'un processeur super-scalaire et la manière dont les instructions sont traitées. Les processeurs actuels intègrent également en plus des unités dites vectorielles que nous présentons dans la section suivante.

2.2.2 Les unités vectorielles

Différentes raisons techniques limitent l'augmentation de la cadence de traitement des processeurs. À ces limites, le parallélisme y répond en traitant plus de données en même temps. Les années 70 ont marqué l'arrivée des processeurs vectoriels. L'entreprise Cray annonce en 1975 son premier ordinateur vectoriel le Cray-1. Ces ordinateurs vectoriels étaient les plus puissants de leur génération. Les processeurs x86 sont devenus vectoriels dans les années 90 avec l'arrivée des instructions MMX. De nos jours, l'entreprise ARM qui assure la conception de processeurs annonce l'arrivée prochaine de sa technologie SVE

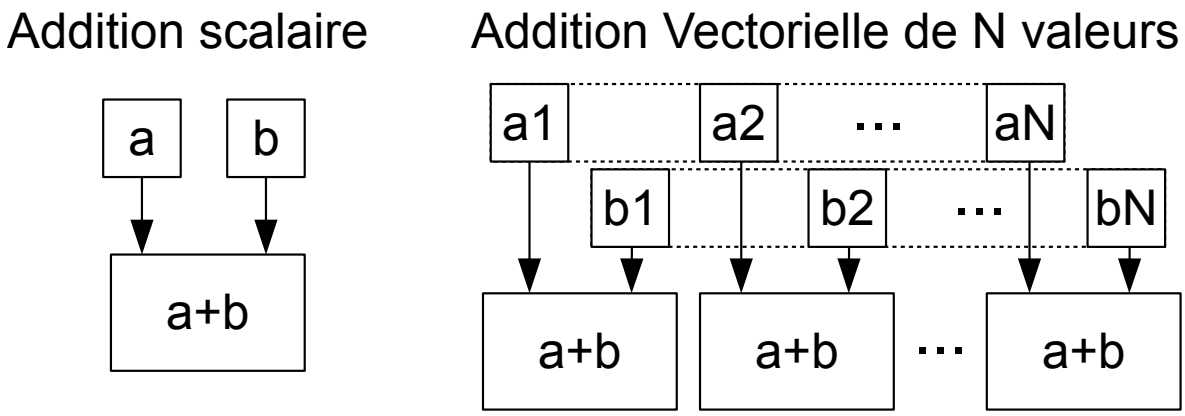

FIGURE 2.11 – D'un module d'addition scalaire à un module d'addition vectorielle.

pour Scalar Vector Extension. Cette technologie est particulièrement attendue en raison de la grande taille de registres vectoriels qu'elle propose (2048 bits). On se retrouve donc avec différentes technologies d'instructions vectorielles selon que l'on soit sous architecture x86 ou ARM. De plus, dans une même famille de processeurs, les jeux d'instructions vectoriels peuvent varier. On citera l'exemple de l'instruction SSE *aligr* pour les vecteurs de 128bits. Elle n'est pas disponible dans le jeu d'instruction AVX pour les vecteurs de 256bits. Toutefois, il est possible d'émuler cette instruction via un assemblage d'autres instructions AVX.

L'idée de base derrière le calcul vectoriel est simple. La partie scalaire des processeurs vus dans la section précédente permet d'exécuter des instructions de calcul scalaire (une valeur calculée par unité arithmétique). Chaque instruction scalaire ne traite qu'une seule donnée à la fois SISD. En revanche, un processeur vectoriel propose des instructions s'appliquant sur plusieurs données à la fois SIMD. Les applications bénéficiant du calcul vectoriel ont la particularité d'avoir leurs données contiguës telles que l'imagerie, le traitement des signaux, l'algèbre linéaire et bien d'autres encore. Ainsi, les applications sont nombreuses et l'effort technique assez faible. En effet, fabriquer une instruction vectorielle consiste à fabriquer plusieurs circuits de calcul scalaire en un comme le montre la figure 2.11. Or, on sait déjà concevoir l'instruction qui additionne 2 valeurs et en retourne la somme. Ainsi, fabriquer une addition vectorielle consiste à démultiplier N fois la circuiterie d'addition. Elle prendre en entrée 2 tableaux de N valeurs pour en retourner un tableau de N valeurs résultant de l'addition valeur à valeur.

Pour qu'un ordinateur scalaire puisse manipuler ses données uniques, il a besoin d'un nombre minimal d'instructions tel que le chargement ou le stockage. Par conséquent, une transformation des données dans les vecteurs n'est pas toujours directe selon le jeu d'instructions vectorielles disponibles. En effet, le décalage d'entier d'un vecteur à un autre peut nécessiter plusieurs instructions vectorielles.

Au cours des évolutions de nos processeurs x86, la taille des vecteurs a plusieurs fois doublé. De ce fait et pour des raisons de rétrocompatibilité, les processeurs les plus récents

cumulent des tailles de vecteurs historiques. Ainsi cohabitent des vecteurs de différentes tailles (64bits, 128bits, 256bits et 512bits). Ceci implique un typage fort sur la taille. En revanche, l'interprétation des données des vecteurs peut facilement être changée. En effet, on parle de vecteur d'un certain nombre de bits. Ces bits sont regroupés en paquets pour former des données typées. Ils peuvent ainsi former des tableaux d'entier 32bits ou alors de flottants simple ou double précision. Les instructions s'appliquent alors suivant un typage faible uniforme de ces groupes de bits (entier 32bits, flottant 32bits, flottant 64bits, etc...).

Regardons maintenant quelques instructions élémentaires utilisées dans les travaux de cette thèse. La partie droite de la figure 2.11 montre un exemple d'additionneur vectoriel $\overrightarrow{c} = \overrightarrow{a} + \overrightarrow{b}$. Cette opération correspond bien à l'addition mathématique. De la même façon que l'addition, mais différemment des mathématiques, la multiplication vectorielle se réalise en produit par composantes à ne pas confondre avec un produit scalaire ou vectoriel. Maintenant que l'on a vu où est le parallélisme du calcul vectoriel, nous allons aborder la manipulation de ces vecteurs.

Pour réaliser certaines manipulations des données contenues dans ces vecteurs, il faut que les instructions adaptées soient disponibles. Dans le cas contraire, il est possible de les d'émuler, mais cela peut être plus lent. En C++, les instructions vectorielles des architectures x86 d'Intel s'appellent des Intrinsics. Ces instructions vectorielles sont regroupées par lot (jeu d'instructions). Les vecteurs de 128bits sont arrivés avec le lot baptisé SSE. Ils ont été enrichis via les lots SSE2, SSE3, SSE3, SSE4.1 et SSE4.2. Le lot AVX apporte les vecteurs de 256bits. Ce lot apporte également de nouvelles instructions pour les vecteurs de 128bits. L'AVX-2 apporte des instructions supplémentaires pour les vecteurs de 256bits, mais aussi pour ceux de 128 et 64 bits. Enfin, l'AVX-512 permet de travailler avec des vecteurs de 512bits. Il ajoute également des instructions pour les vecteurs de plus petite taille. Il existe d'autres lots tels que les Fused Multiply Add (FMA). Ces instructions réalisent en une seule instruction depuis 3 vecteurs la multiplication de 2 des vecteurs additionnés au 3^{ime}.

Plus le processeur est récent et plus il cumule ces lots. Contrairement aux instructions vectorielles Néon des architectures ARM, les Intrinsics ne proposent pas d'instruction de décalage comme dans la figure 2.12. Heureusement, le SSE4 possède une instruction 128bits capable d'émuler directement le décalage. Cependant, si l'on souhaite travailler via l'AVX avec des vecteurs de 256bits, l'émulation d'un décalage est délicat en l'absence du lot d'instructions AVX-2. Nous aurons l'occasion de revenir sur l'intérêt d'avoir une telle fonction lorsque nous aborderons les stencils.

Les processeurs vectoriels doivent réaliser des chargements vectoriels des données. Concernant les processeurs x86 modernes, ils proposent principalement 2 instructions de chargement (load). La première instruction que nous appellerons vLoad exige que l'adresse mémoire soit un multiple de la taille en octet du vecteur. On dit que l'adresse doit être alignée sur la taille du vecteur (l'adresse doit s'aligner sur 16 pour un vecteur 128bits, 32 pour 256bits, etc ...). La seconde instruction que nous appellerons vLoadU (U pour Unaligned) est plus souple et n'exige pas cet alignement, mais elle est plus lente. De façon symétrique, les instructions de stockage que nous appellerons respectivement vS-

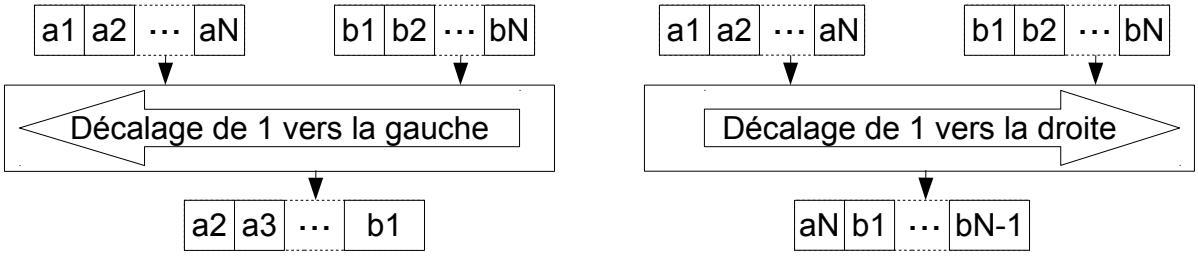

FIGURE 2.12 – Le décalage des données en registre vectoriel.

tore et vStoreU fonctionnent sur ce même principe. Regardons maintenant en quoi vLoad et vStore seraient plus rapides que vLoadU et vStoreU. Tout d'abord, rappelons qu'une ligne de cache représente 64 octets contigus commençant à une adresse mémoire alignée sur 64. On peut exprimer l'adresse de la ligne via la formule $Adresse = 64L + C$ avec L le numéro de la ligne et C le numéro de la colonne. On souhaite maintenant charger un vecteur de 32 octets depuis l'adresse vA. Il faut donc remonter les lignes de cache de l'intervalle $[\frac{vA}{64}; \frac{vA+32-1}{64}]$. On comprend que dans le cas où nos 32 octets sont entre 2 lignes de cache, il faut charger 2 lignes au lieu d'une. Maintenant que nos lignes de cache sont chargées, il reste à récupérer en registre vectoriel ces 32 octets. Les circuits de transfert prévus pour les vecteurs de 256bits sont adaptés à un chargement direct par multiples de 32 octets. Lorsque l'adresse de nos données à mettre en registre 256bits n'est pas alignée sur 32 octets, il faut procéder en plusieurs étapes comme dans la figure 2.13. Ainsi, lorsque nous sommes surs que le chargement de nos vecteurs 256bits se réalise bien depuis des adresses multiples de 32, il est préférable d'utiliser l'instruction de chargement aligné. En effet, cette instruction n'a pas à vérifier si l'adresse est alignée ou non. Alors que l'instruction vLoadU va devoir recomposer si besoin les données comme dans la figure 2.13. On comprend que cette instruction est plus longue à réaliser. À noter que cela est également vrai pour toutes les données sur plusieurs octets (entier, flottant, etc...). En résumé, lorsque l'on travaille avec des vecteurs (et même des données scalaire), il faut rester sur des adresses alignées en mémoire selon la taille du type. Ceci est valable symétriquement pour le stockage. Nous verrons que d'autres instructions permettent de réaliser des transferts entre les registres vectoriels et la mémoire.

Pour résumer, les différentes générations des machines employées dans nos expérimentations proposent des jeux d'instructions vectorielles variables. Il faut parvenir à employer les bonnes instructions en fonction de l'architecture cible.
Différentes approches permettent de réaliser des programmes de calcul via les unités vectorielles. Nous commencerons par la vectorisation automatique proposée par les compilateurs.
Un compilateur a comme objectif de traduire un programme écrit dans un langage de haut niveau vers le langage d'instructions binaire du processeur cible. Dans le cas de nos plateformes x86, le compilateur peut traduire de différentes façons et de ces façons vont dépendre la vitesse d'exécution du programme. Il doit avant tout garantir l'adéquation entre la description faite dans le langage de haut niveau avec le déroulement du programme binaire exécuté. Il ne garantit pas nécessairement l'ordre des opérations arithmétiques ce

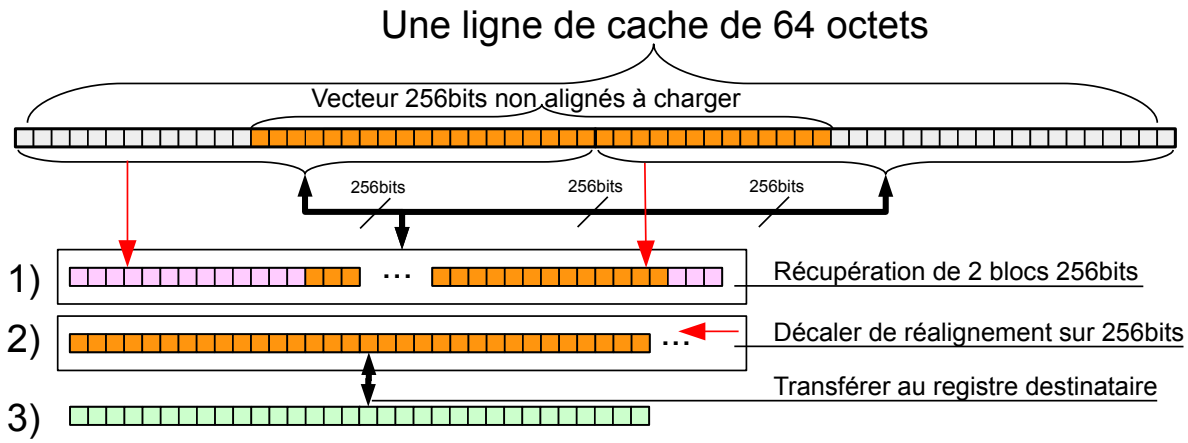

FIGURE 2.13 – Chargement vers un registre 128bits d'un bloc de 16 octets d'adresse **non alignée** sur 16.

qui peut changer les résultats. En effet, suivant l'ordre de ces opérations, les résultats peuvent différés en cas d'arrondis. Une fois que la description est respectée, il peut réaliser différentes optimisations.

Comme nous l'avons vu dans la sous-section 2.2.1 précédente sur le processeur scalaire, le compilateur peut réordonner les instructions de façon à simplifier l'exécution au niveau du pipeline. Les optimisations qui nous intéressent ici sont celles faisant usage d'instructions vectorielles. On appelle cela la vectorisation automatique du compilateur qui met en oeuvre du parallélisme implicite. Le compilateur est capable d'identifier plusieurs situations où il peut remplacer des instructions scalaires par des traitements parallèles via des instructions vectorielles. Ceci à condition de lui demander de le faire suivant des paramètres à la compilation. La difficulté pour lui est qu'il doit respecter la description du programme. De plus, il n'est pas capable d'aller trop loin vers des modifications plus fondamentales du programme tel que modifier une structure de données. En effet, la compilation doit rester assez courte sans quoi l'optimisation serait difficilement rentable. Cependant, on compte parfois sur lui pour vectoriser. Or l'optimisation via le compilateur n'est pas une garantie. Elle dépend des stratégies d'optimisation propres à chaque compilateur. Toutefois, certains s'accord assez bien sur des patrons remarquables ainsi que sur des annotations apportant de la méta information sur le code. Nous allons passer en revue quelques exemples classiques qui rendent le programme vectorisable ou non par un compilateur. Le premier exemple figure 2.14 est un cas simple et aisément vectorisable par un compilateur. Ainsi, le compilateur GCC produit le code assembleur commenté dans l'exemple 2.15.

Le compilateur GCC trouve aisément un autre moyen de réaliser les additions en une addition vectorielle à l'aide d'instructions AVX. Pour cela, il déroule partiellement la boucle par lot de 8 flottants. Cependant, le compilateur ne sait pas si les tableaux v0 et v1 sont alignés. En effet, cela dépend du malloc qui ne garantit pas au compilateur l'alignement mémoire des données. De fait, les chargements mémoires ne sont pas alignés ce qui les rend moins performants. En effet, le compilateur est contraint d'employer des ins-

```
void principale()
{
 i size = 1000;

 float * v0 = (float *)malloc( size * sizeof(float) );
 float * v1 = (float *)malloc( size * sizeof(float) );

 // ... remplissage de v0 et v1 ...

 for( size_t i = 0 ; i < size ; ++i )
 {
  v0[ i ] += v1[ i ];
 }
}
```

FIGURE 2.14 – Exemple de code automatiquement vectorisable en 2 parties par un compilateur.

```
void principale()
{
 // code assembleur pour la boucle obtenue avec
 //la commande objdump -D vecadd -M intel
 // 1098: vmovups ymm1,YMMWORD PTR [rax+rdx*1]
 // 109d: vaddps ymm0,ymm1,YMMWORD PTR [rbx+rdx*1]
 // 10a2: vmovups YMMWORD PTR [rbx+rdx*1],ymm0
 // 10a7: add     rdx,0x20
 // 10ab: cmp     rdx,0xfa0
 // 10b2: jne     1098
 //...
}
```

FIGURE 2.15 – Exemple de code assembleur généré et automatiquement vectorisé par GCC.

```
void principale()
{
 size_t size = 1000;

 float sum = 0.0f;
 float * v0 = (float *)malloc( size * sizeof(float) );

 // ... remplissage de v0 ...

 for( size_t i = 0 ; i < size ; ++i )
 {
  sum += v0[ i ];
 }
}
```

FIGURE 2.16 – Exemple de code automatiquement vectorisable en 2 parties par un compilateur.

tructions vectorielles load et store non aligné. Il en va de même pour nous. Cet allocateur ne nous garantit pas non plus l'alignement de ses allocations mémoires. Cependant, nous pouvons y remédier en faisant appel à un allocateur aligné. Enfin, la version assembleur en commentaire est nécessairement suivie d'une seconde boucle non vectorisée chargée de réaliser le reste modulo 8 des 2 tableaux.

Le second exemple 2.16 consiste à réaliser la somme flottante d'un tableau. Comme pour l'exemple précédent, ce dernier est compilé sous GCC et le code assembleur est présenté en commentaire dans la figure 2.17. On remarque que la boucle est déroulée de manière vectorisée par blocs de 8 flottants (unités vectorielles AVX). Cependant, bien que les chargements soient vectoriels, le cumul ne l'est pas. Ainsi, cette vectorisation est inefficace.

En résumé, le compilateur a besoin d'un peu d'aide pour parvenir à vectoriser. On comprend qu'un développeur cherchant à écrire un code propre via un accès indicé peut être favorisé tout en ignorant les subtilités de la vectorisation automatique. La question qui reste en suspend est de savoir si le compilateur vectorise et qu'il le fait efficacement. Ce qui devient implicite c'est aussi bien la vectorisation que sa qualité. On peut l'aider en précisant des informations tel que l'alignement mémoire des tableaux. On espère ainsi qu'il tiendra compte de ces informations sur l'allégement afin d'utiliser des instructions plus rapides d'accès mémoire alignés. Ceci est sans compter sur la mise à jour des compilateurs changeant les subtilités de cette vectorisation automatique. Dans la pratique, soit on vérifie le code assembleur généré par le compilateur, soit on place explicitement dans le code les appels aux instructions vectorielles.
Cette autre approche consiste à forcer le compilateur dans l'utilisation d'instructions vectorielles. Il existe de nombreux cas où l'implémentation vectorielle explicite ou abstraite est préférée. En effet les travaux [5] réalisent des essais entre CPU et GPU. Via OpenCL, ils essaient de porter automatiquement leur code tout en restant performants. Ils témoignent de la difficulté de profiter automatiquement des instructions vectorielles disponibles. Dans les travaux du livre [39], il y est précisé que la vectorisation ne sera pas explicite. Ceci est présenté tel un risque assumé de leur part et qu'il serait préférable d'expliciter la vectorisation. Nous avons nous-mêmes publié en partie sur ce sujet via l'article [57]. Les

```
void principale()
{
 // code assembleur pour la boucle obtenue avec
 // la commande objdump -D reduction -M intel
 // 1090: vmovups xmm2,XMMWORD PTR [rax]
 // 1094: vmovups ymm3,YMMWORD PTR [rax]
 // 1098: add     rax,0x20
 // 109c: vaddss xmm0,xmm2,xmm1
 // 10a0: vshufps xmm1,xmm2,xmm2,0x55
 // 10a5: vaddss xmm0,xmm0,xmm1
 // 10a9: vunpckhps xmm1,xmm2,xmm2
 // 10ad: vshufps xmm2,xmm2,xmm2,0xff
 // 10b2: vaddss xmm0,xmm0,xmm1
 // 10b6: vextractf128 xmm1,ymm3,0x1
 // 10bc: vaddss xmm0,xmm0,xmm2
 // 10c0: vshufps xmm2,xmm1,xmm1,0x55
 // 10c5: vaddss xmm0,xmm0,xmm1
 // 10c9: vaddss xmm0,xmm0,xmm2
 // 10cd: vunpckhps xmm2,xmm1,xmm1
 // 10d1: vshufps xmm1,xmm1,xmm1,0xff
 // 10d6: vaddss xmm0,xmm0,xmm2
 // 10da: vaddss xmm1,xmm0,xmm1
 // 10de: cmp     rdx,rax
 // 10e1: jne     1090
 // ...
}
```

FIGURE 2.17 – Exemple de code assembleur généré et automatiquement vectorisé par GCC.

essais laissant la vectorisation aux compilateurs sont sans appel. On ne peut pas compter sereinement sur la vectorisation automatique des compilateurs.

Cependant, selon les jeux d'instructions et les tailles de vecteurs disponibles sur l'architecture cible, il n'est pas aisé d'adapter un programme vectoriel pour différentes générations de processeurs. La difficulté est accrue lorsqu'il faut passer des processeurs x86 à des processeurs ARM. Ainsi, la vectorisation explicite n'est pas des plus simples à mettre en oeuvre. En effet, d'une part elle est dépendante de l'architecture, d'autre part il y a plusieurs façons de traiter un même calcul en vectoriel. En effet, dans cette thèse, le calcul d'élément spectral revient à réaliser des opérations matricielles. Il s'offre alors à nous 2 possibilités. Soit, on vectorise le calcul de petites matrices de l'article [1]. Ou bien, on vectorise le calcul de plusieurs éléments à la fois. Selon l'approche, les performances diffèrent, il faut donc les évaluer.
De plus, nous verrons lors du chapitre 3 que l'accès vectoriel au voisinage relatif d'un stencil peut nécessiter l'usage d'une opération de décalage des données dans 2 vecteurs joints. Sous processeur Intel x86, cette opération de shift est aisément réalisable pour les vecteurs de 128bits avec l'instruction alignr du SSE4. En outre, sous AVX, les vecteurs de 256 bits nécessitent l'usage d'une combinaison d'instructions afin de réaliser cette opération. Afin d'aider le développeur, il existe des solutions de vectorisation masquant la dépendance matérielle et le guidant vers les bonnes pratiques Á cette fin, le midleware boost SIMD assure la compatibilité entre des opérations vectorielles de plus haut niveau et les jeux d'instructions disponibles de l'architecture cible. Il est ainsi rendu possible de passer de façon transparente du vectoriel x86 128bits ; 256bits et 512bits au vectoriel ARM. Toutefois, cette librairie n'est pas gratuite pour les vecteurs de 512bits et encore moins libres. Ainsi, pour plus de visibilité, cette thèse a fait l'objet du développement d'un petit groupe de fonctions de substitution découplant les appels d'intrinsics des opérations vectorielles. Dans cette thèse, nous l'appellerons intrinsiX.

Les solutions de calcul vectoriel par abstraction sont un bon compromis. Cela évite de compter sur de la chance via le compilateur ou de se retrouver dans l'incapacité de changer d'architecture. L'article [11] présente un cas d'usage d'instructions SIMD pour la 5G à l'aide de la librairie MIPP [6]. Dans un même objectif que MIPP, on citera la librairie VecCore [7]. Au-delà du calcul vectoriel, il est possible de considérer toute une nuance d'abstraction depuis le bas niveau parallèle jusqu'au niveau métier où il ne reste plus qu'à paramétrer la simulation numérique.
Lorsque l'on remonte dans des niveaux plus abstraits, d'autres ressources processeur peuvent être employées. La sous-section suivante va donc s'employer à présenter l'aspect parallèle multi cœurs des processeurs.

6. MIPP est une librairie portable encapsulant les instructions vectorielles https://github.com/aff3ct/MIPP.

7. VecCore est une librairie découplant le calcul vectoriel du matériel https://github.com/root-project/veccore.

2.2.3 Le multi-cœurs

Dans les années 80, les processeurs étant limités, les ordinateurs étaient mis en réseau pour partager des traitements afin de les réaliser plus rapidement. Les ordinateurs d'architecture SMP (Symétrique shared memory Multy Processors) sont alors apparus. Un ordinateur SMP est constitué de plusieurs processeurs partageant un même module de mémoire. L'ordinateur était encore imposant et plus encore à l'échelle d'une machine qui pouvait en contenir plusieurs. Il y avait alors une forte demande des industriels pour réduire la place occupée par ces machines. La fin du « BigFoot » était ainsi amorcée (surnom dû au caractère imposant de ces machines). La technologie du multi-cœur était née.
Cette technologie consiste à fusionner plusieurs processeurs en un seul. Les processeurs ainsi fusionnés se composent de plusieurs cœurs de traitement. Chaque cœur peut prendre en charge l'exécution d'un thread (fil d'exécution). Certains sont même capables d'en prendre plusieurs en charge à la fois. Ainsi, on différencie les cœurs logiques et les cœurs physiques. En effet, un cœur physique peut exécuter autant de threads en même temps qu'il expose de cœurs logiques. En général, les registres de travail du cœur sont dédoublés pour chaque cœur logique. Ainsi, un cœur peut assurer un suivi de plusieurs threads en même temps. Par conséquent, les threads s'entremêlent dans le pipeline. Cependant, ils restent rattachés à leur contexte d'exécution via leurs registres. Nous verrons dans cette thèse en quoi l'exécution de plusieurs instances d'un programme sur un même cœur physique est contreproductive.

Ces processeurs sont ainsi capables de réaliser du parallélisme de type SPMD. À la différence du SIMD, le SPMD concerne l'exécution multiple d'un même programme sur plusieurs cœurs. Dans ce parallélisme, les chemins logiques peuvent différer suite à des résultats de test conditionnel différents ou à une exécution différée. Ainsi, l'exécution de plusieurs instances du même programme peut s'exécuter en parallèle de façon asynchrone. On appelle ces instances d'exécution des threads (fils d'exécution). Le système d'exploitation est chargé de placer les threads sur les cœurs. Il peut dynamiquement les faire migrer d'un cœur à l'autre, mais ceci est couteux en particulier pour les caches mémoires. Sa politique de placement est configurable afin de répartir au mieux les threads sur les cœurs et d'éviter qu'ils se fassent migrer. En effet, un cœur peut-être capable de suivre l'exécution de plusieurs threads à la fois via l'hyper-threading. Lorsque plusieurs threads d'un même cœur exécutent le même programme, les mêmes ports d'instructions sont fortement sollicités. Ainsi, l'apport de gain n'est pas au rendez-vous. Cependant, elle est plus pertinente lorsque les unités d'exécutions fortement sollicitées sont dupliquées sur plusieurs ports. Ainsi, la technologie d'hyper-threading est plus adaptée à l'exécution de programmes différents (MPMD pour Multy Program Multy Data. On applique différents programmes á plusieurs données (MPMD)).

Maintenant que plusieurs threads partagent la même mémoire, des problématiques déjà rencontrées avec les architectures SMP se posent. Parmi ces problématiques, on souligne celle du maintien de cohérence des lignes de cache. La figure 2.18 présente un cas où 2 cœurs partagent une même ligne de cache. Dans cet exemple, l'un des cœurs écrit dans une ligne partagée ce qui a pour conséquence d'invalider la version de la ligne présente dans l'autre cœur. Dans le cas de nos architectures x86, cette gestion de la cohérence des

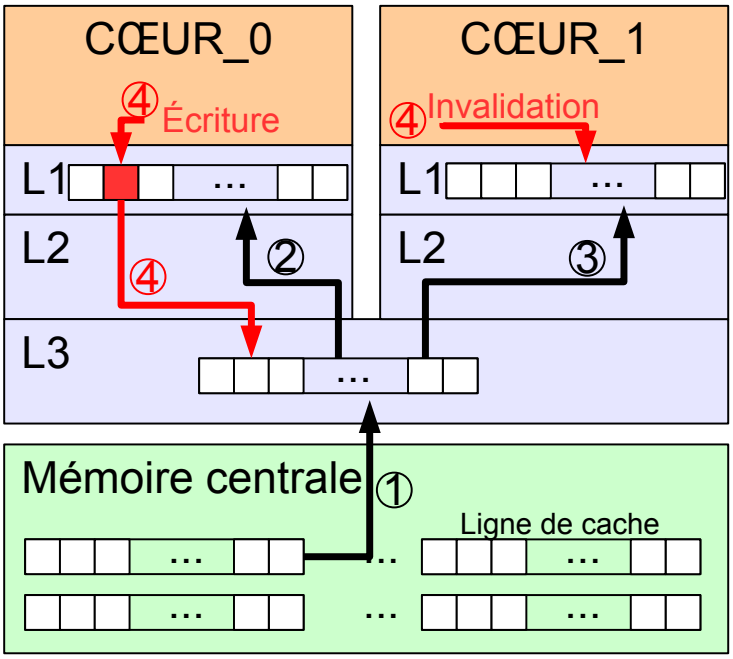

FIGURE 2.18 – Écriture invalidante dans une ligne de cache.

caches est gérée au niveau matériel.

Voici une description précise du scénario de la figure 2.18 :

- Le cœur 0 charge une ligne de cache depuis la RAM.
- Elle se transfère au niveau L3 (1).
- Elle se transfère au niveau L2 puis L1 du cœur 0 (2).
- Suite à une demande d'accès du cœur 1, cette même ligne déjà en L3 se transfère au niveau L2 puis au niveau L1 du cœur 1 (3).
- Le cœur 0 écrit dans cette ligne de cache ce qui implique une invalidation matérielle partout ailleurs. La version de cette ligne de cache en L3 est donc mise à jour depuis le cœur 0.
- Ainsi, lorsque le cœur 1 accèdera à nouveau à cette ligne, il devra la retransférer à jour depuis le L3.

Nous venons de voir qu'il existe des problématiques matérielles au partage multi-cœur d'une mémoire. Les cas d'invalidations de ligne de cache sont toujours source de ralentissement des accès aux données. En effet, dans notre exemple, recharger la ligne de cache depuis le L3 au lieu du L1 reste plus lent. La bonne pratique est donc d'éviter de travailler sur les mêmes lignes de caches aux mêmes moments.

D'autres problèmes se posent également au niveau logiciel. En effet, lorsque 2 cœurs travaillent sur les mêmes données. Ils doivent se synchroniser afin de ne pas y écrire simultanément. Dans le cas contraire, des résultats peuvent être faux. Afin de pallier à cela et suivant les calculs à réaliser, un ordonnancement synchronisé entre les cœurs peut-être nécessaires. Au plus bas niveau, des verrous de type sémaphore ou mutex assurent la synchronisation entre les cœurs via le système d'exploitation. Leur fonctionnement consiste à

ce qu'un thread d'exécution demande au système à passer via un verrou identifié. Dans le cas où ce dernier est verrouillé, le système répond au thread par la négative. Si le verrou est bloquant, le système met le thread demandeur en attente puis le réanime au moment du déverrouillage. Le verrouillage peut dépendre de différentes conditions par exemple d'un certain nombre de passages de thread autorisés à la fois. Des librairies telles que pthread et boost permettent de simplifier la gestion des threads. Elles sont progressivement ajoutées dans les nouveaux standards du C++. Parmi ces nouveaux standards, des données atomiques sont introduites. Le principe est de ne pas lire une donnée alors qu'elle est en train d'être écrite sans quoi les résultats qui en découlent seraient faux. Les données atomiques assurent leur lecture et écriture comme étant transactionnelle. Un cœur ne peut donc lire la donnée alors qu'elle est écrite par un autre. Le caractère threadsafe de ces données atomiques peut paraitre vendeur. Seulement, il n'y a pas de vecteur SIMD de donnée atomique de plus la vitesse d'accès reste à apprécier. Enfin, il reste nécessaire de maitriser ce que l'on fait, car la donnée atomique n'exclut pas tous les mauvais cas d'usage.

La librairie pthread propose également la notion de barrière de threads. Cette barrière permet aux threads de s'attendre afin de repartir tous ensemble d'un même point dans l'exécution du programme. Nous verrons que la librairie OpenMP utilise les barrières de threads notamment dans la parallélisation de boucles itératives. Avant cela, il est nécessaire de parler de la librairie TBB d'Intel. TBB signifies « Threading Building Blocks ». Cette librairie simplifie la gestion des threads et propose d'allouer les threads par pool. Ceci permet d'éviter des appels système intermittents pour la création des threads. Elle est mise en avant par Intel au détriment de la librairie cilk. TBB permet une maitrise plus bas niveau des threads que la librairie OpenMP. Nous remontons petit à petit les niveaux d'abstraction du parallélisme multi-cœur. En effet, OpenMP pour Open Multiprocessing est une librairie de gestion de threads dont la particularité est d'ajouter un métha-langage dans celui de programmation afin d'annoter le parallélisme que l'utilisateur désire. Cette librairie est employée dans cette thèse. Nos implémentations utilisent en particulier la parallélisation des boucles itératives via la directive « parallel for ».
À la différence d'OpenMP, la librairie cilk de chez Intel proposait quant à elle une gestion plus orientée planification de tâche avec de la gestion de dépendance entre elles. Des mécanismes tels que le vol de tâche la rendaient adaptée à du parallélisme dont la dynamique était irrégulière. Les standards et implémentations d'OpenMP ayant bien muri, Cilk est abandonnée au profit de TBB.

Ces librairies de haut niveau abstraient jusqu'à rendre implicite la gestion des threads. Ainsi, il en va de la responsabilité du développeur employant implicitement plusieurs threads de rester vigilant face aux risques d'optimisations contre-performantes. Avec ces environnements de multithreading, il reste toujours possible d'avoir des cas d'accès non synchronisés avec des données partagées. Ceci peut constituer des erreurs de calcul faussant les résultats. Le risque est d'autant plus grand que OpenMP masque le parallélisme ce qui peut devenir une source d'erreurs. De plus, d'autres phénomènes peuvent limiter l'efficacité du parallélisme multithreads. En effet, les threads travaillant en parallèle peuvent pour plusieurs raisons terminer à des moments différents. Parmi ces raisons, il y a les accès mémoires dont la durée n'est pas uniforme. Effectivement, certaines données devront être montées depuis la RAM alors que d'autres se trouvent déjà dans les

caches. De plus, la répartition du travail n'est pas toujours équilibrée. Ainsi un thread qui à moins de travail que les autres terminera plus tôt et devra les attendre. On dit que le thread est en famine (starvation). À raison d'un thread par cœur physique, ceci est problématique lorsque tous les threads vont devoir s'attendre à un moment ou à un autre. Le cumul de ces attentes est d'autant de ressources inexploitées. Il est difficile de garantir une durée homogène dans le traitement des tâches en toute circonstance et d'assurer un bon équilibrage sur l'ensemble des cœurs disponibles. Afin d'y faire face, des techniques ont vu le jour. Le vol de tâche en est une. Celle-ci consiste à décomposer le traitement en plusieurs tâches réparties sur l'ensemble les threads. Lorsqu'un thread termine toutes ses tâches, il vole des tâches en attente chez d'autres threads. Idéalement, les tâches sont assez nombreuses et petites pour que la famine soit minime à l'approche d'une barrière. Il faut également être vigilant concernant la gestion des tâches. En effet, cette gestion des tâches consomme du temps de traitement et de l'espace mémoire. Ainsi, plus il y a de tâches et plus leur gestion prennent des ressources. Il y a donc un juste milieu à trouver entre la taille des tâches et le niveau de déséquilibre cumulé. L'ordonnancement de tâche est un sujet de recherche à part entière comme le montrent les articles [49, 66, 32]. Ce sujet entre en marge de cette thèse. Des environnements d'exécution proposent de gérer l'ordonnancement suivant un graphe de dépendances. On citera ici les travaux [28, 68] sur l'environnement d'exécution C++ HPX optimisé pour l'usage de futures.

Dans la même veine, la librairie StarPU développée à l'Université de Bordeaux permet également d'ordonnancer suivant des dépendances. L'équilibrage dynamique de cette librairie facilite l'exploitation de différentes ressources d'une même machine comme le montrent les travaux [40, 3]. Une machine peut se composer de processeur central, mais aussi de processeurs accélérateurs tels que des GPU. On comprend alors tout l'intérêt d'adapter la charge de travail via l'ordonnancement dynamique des tâches parallélisées sur plusieurs processeurs hétérogènes. Bien que cette thèse se limite à des multi-cœur homogènes de processeurs centraux, ces aspects la concernent. En effet, les cœurs des CPU doivent également être organisés du point de vue logiciel afin de tenir compte de la dépendance des données.

Les architectures SMP bien que miniaturisées en un seul processeur multi cœur ne peuvent augmenter infiniment le nombre de cœurs de traitement sans saturer les accès aux modules partagés de mémoire . Il devient alors clair que les modules mémoires doivent également être dupliqués. Seulement, les accès ne peuvent être directe entre tous les cœurs et tous les modules mémoire de la machine sans recourir à circuiterie bien trop complexe. Ainsi, les architectures à accès de mémoire non uniforme NUMA ont vu le jour. Ces dernières sont présentées dans la sous-section 2.2.4 suivante.

2.2.4 Le NUMA

Les plateformes expérimentales de cette thèse sont des architectures x86 d'Intel. Elles ont la particularité d'être des architectures à accès mémoire non uniforme. L'acronyme NUMA pour « Not Uniform Memory Access (NUMA) » est employé pour identifier ce type d'architecture. Nous venons de voir dans la sous-section 2.2.3 précédente que l'architecture SMP était limitée quant au nombre de processeurs pouvant réaliser des accès concurrents

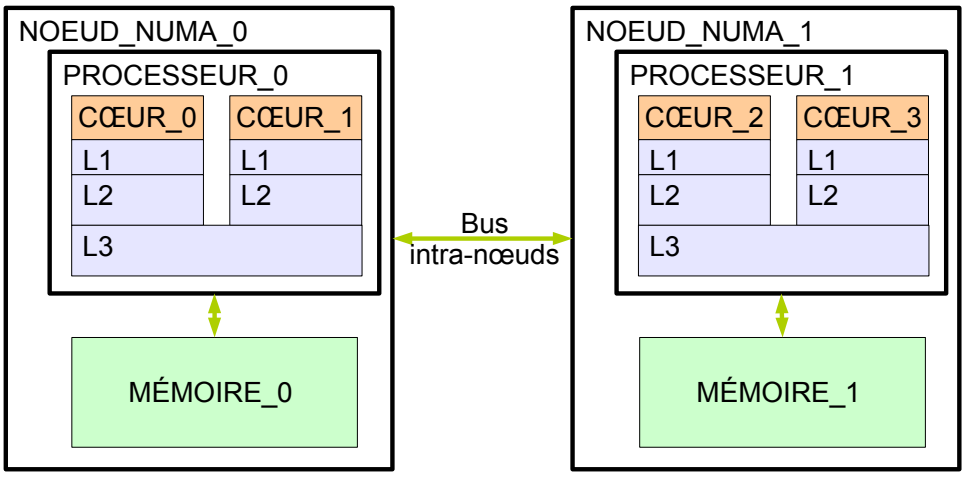

FIGURE 2.19 – Exemple d'architecture NUMA bi-noeuds (bus intra-noeud QPI chez Intel capable de transférer à 25,6 GB/s).

directs. Il est donc difficile d'étendre le partage d'un module mémoire à plusieurs processeurs multi-cœurs. Ainsi, chaque processeur accède directement à leurs modules mémoire via un accès qui lui est réservé. Ce qui nous amène aux architectures NUMA comme dans l'exemple dans la figure 2.19. Nous présenterons dans cette sous-section ces architectures à l'échelle d'un ordinateur. En effet, le NUMA peut exister à l'échelle de plusieurs ordinateurs.

Les architectures NUMA sont composées de plusieurs noeuds NUMA. Ces noeuds se composent d'un processeur ayant un accès privilégié à un espace mémoire. La mémoire RAM restant partagée, les processeurs peuvent toujours accéder aux autres espaces mémoire. Seulement, ils vont devoir passer via un ou plusieurs bus intra-noeuds pour y accéder allongeant ainsi les délais d'accès. En effet plusieurs topologies existent pour lier ces noeuds via des bus. Des noeuds peuvent aussi être liés par 2 bus d'accès réduisant ainsi la saturation des échanges entre eux.

Étant donné que les accès aux différents espaces mémoire ne sont pas les mêmes suivant les cœurs, il est préférable que les données concernant un thread soient autant que possible localisées dans l'espace mémoire privilégié d'accès pour le cœur exécutant le thread. Cependant, en cas de déséquilibre dans la quantité d'accès mémoire d'un threads, ce dernier risque de saturer son accès mémoire. Dans ce cas, soit le déséquilibre peut-être résolu, soit il est préférable de répartir les données sur plusieurs neouds mémoire.

Nous avons réalisé des tests de localité mémoire dans nos architectures NUMA. Les résultats de notre article [59] montrent que la performance est sensible à la localité mémoire. La plateforme expérimentale employée dans ce test dispose de 2 noeuds NUMA. L'attribution de tous les threads sur l'un des noeuds montre que l'allocation strictement distante sur l'autre noeud mémoire est moins performante. En revanche, alterner l'allocution mémoire entre les 2 noeuds est bénéfique. En effet, bien que les cœurs doivent passer via le bus inter-noeuds, cela permet de soulager leur accès direct et ainsi profiter de l'accès du 2^{ime} noeud. C'est pourquoi nos expérimentations se font avec de la pagination alternée

via le paramètre d'interleave de l'environnement d'exécution de la librairie libnuma. Sans quoi toute l'allocation des données se ferait uniquement sur la mémoire d'accès le plus direct pour le thread principale. De plus, cette librairie permet de récupérer des informations sur l'architecture. On peut ainsi connaitre le nombre de noeuds, leurs liaisons et les latences des bus inter-noeuds. Elle permet aussi de connaitre les numéros des cœurs par noeuds. Ces numéros permettent alors de définir une politique d'attache des threads à ces cœurs.

Les expérimentations de cette thèse fixent les threads et alternent les allocations des pages mémoire allouées par les noyaux. Néanmoins, il serait préférable de gérer la localité mémoire directement dans le programme. Cette sous-section termine ce tour d'horizon sur les principaux aspects des architectures impliquées dans cette thèse. Avant de poursuivre, la sous-section suivante résume les points essentiels impliqués dans les travaux cette la thèse.

2.2.5 Conclusion

Les sous-sections précédentes exposent les aspects principaux au croisé des architectures et des applications de cette thèse. En dehors de cette dernière, d'autres architectures sont également employées pour ses mêmes applications. On citera l'exemple des architectures accélératrices de calculs tel que les GPU (Graphics Porcessing Unit (GPU)) ou les FPGA (Field-Programmable Gate Array (FPGA)). De plus, des machines peuvent être le résultat d'assemblages d'ordinateur en architecture distribuée en grappe ou en grille [9, 6]. Chaque architecture a ses points forts et ses points faibles suivant l'application ciblée. L'idéale serait de disposer d'architecture économique et performante pour toutes les applications, mais ce n'est malheureusement pas le cas. En revanche, exploiter autant que possible des ressources d'architectures modernes est ici envisagé pour accélérer l'exécution de nos applications. Nous verrons que les différents points architecturaux évoqués précédemment peuvent être conjointement optimisés. Cependant, il existe des cas où ils rentrent en conflit et peuvent devenir contreproductifs. Après une présentation de nos cas d'applications lors de la section suivante, nous verrons qu'il existe des outils pour identifier et comprendre les phénomènes limitants les performances de nos noyaux de calcul avec les architectures employées.

2.3 Analyse des performances

Dans cette section, nous allons aborder différents modèles, méthodes et outils permettant d'analyser les performances d'exécutions d'un programme. Nous aurons un regard critique sur les bonnes et mauvaises interprétations émergeant de ces prismes plus ou moins déformants.

2.3.1 Modèle d'analyse

Dans cette section, nous allons aborder les modèles d'analyse et d'interprétation des résultats expérimentaux. Ceux présentés ici se paramètrent suivant des indicateurs extraits

des machines et des programmes. Les indicateurs dont nous ferons usage dans cette thèse sont listés ci-dessous :

1. Nombre d'opérations de calcul.
2. Nombre d'octets chargés en registre (estimé en comptabilisant les accès aux variables du programme).
3. Intensité Arithmétique (abrégée en AI) : le ratio entre le nombre d'opérations à effectuer et le nombre d'octets à utiliser pour effectuer un calcul.
4. Intensité Opérationnelle (abrégé en OI) : le rapport entre le nombre d'opérations à effectuer et le nombre d'octets chargés à partir de la mémoire centrale .
5. Temps d'exécution : le nombre de millisecondes passé pour exécuter la zone d'intérêt d'un programme.
6. Performance en GFLOPS : nombre en milliard d'opérations par seconde.
7. Performance en GB/s : nombre en milliard d'octets par seconde.

Dans le cas où les plateformes de calcul sont clairement identifiées, il est possible d'obtenir des indicateurs théoriques. Pour ce faire, il suffit d'interroger la documentation constructeur des composants ou la machine elle-même via son système. Ainsi, il est possible d'estimer la capacité maximale théorique de calcul de la machine en GFLOPS. Si on prend l'exemple d'une machine équipée de 2 processeurs Skylake Intel Xeon Gold 6148, on atteint une capacité maximale de 4096 GFLOPS simple précision (32bits). En effet, ce processeur est équipé de 20 cœurs physiques contenant 2 fois les unités vectorielles 512bits FMA (une opération de multiplication et d'addition en une). De plus, en cas d'utilisation d'instructions vectorielles AVX-512, ce dernier fonctionne à 1.6 Ghz. Il peut donc théoriquement atteindre $4096_{GFLOPS} = 2_{processeurs}(20_{coeurs}(1.6Ghz \times 2_{AVX512}(2_{FMA}(16_{float32}))))$. Ce chiffre est basé sur le cas idéal où tous les cœurs exécutent en parallèle des instructions FMA 16 flottants 32bits 2 par 2 à la suite et donc à chaque cycle. Cette performance théorique permet de connaitre la borne qu'aucun programme ne peut la dépasser. Ainsi, lorsque l'on estime atteindre 2048 GFLOPS sur cette machine, on sait que l'on exploite 50% de la capacité de calcul de la machine. Il reste donc 50% inexploité pour diverses raisons restant à élucider.

La performance maximale peut également être approchée expérimentalement. Pour y parvenir, on exécute sur la machine un programme le plus proche possible du cas idéal cité précédemment avec les FMA. Ce programme est pour nous le noyau de calcul BLAS SGEMM. Il nous a permis d'estimer expérimentalement un maximum de 3826 GFLOPS sur le Skylake double Xeon 6148. Ce programme est instrumenté afin de mesurer un indicateur de temps d'exécution d'une série de calculs. En divisant l'indicateur donnant le nombre de calculs par le temps écoulé, on obtient l'indicateur de performance. Ce chiffre est assez proche de celui théoriquement estimé. L'avantage de cette estimation expérimentale est qu'elle est réalisée sur l'instance de la machine et non via le modèle théorique constructeur. Il faut tout de même garder en tête que la performance atteignable est dépendant du calcul exécuté. En effet, le noyau de calcul BLAS SGEMM est un calcul capable d'approcher la capacité de calcul maximum de la machine. Ce n'est pas le cas de tous les noyaux de calcul. En effet, certains calculs sont plus dépendants de la mémoire, car ils utilisent beaucoup de données pour faire peu de calculs. Pour estimer

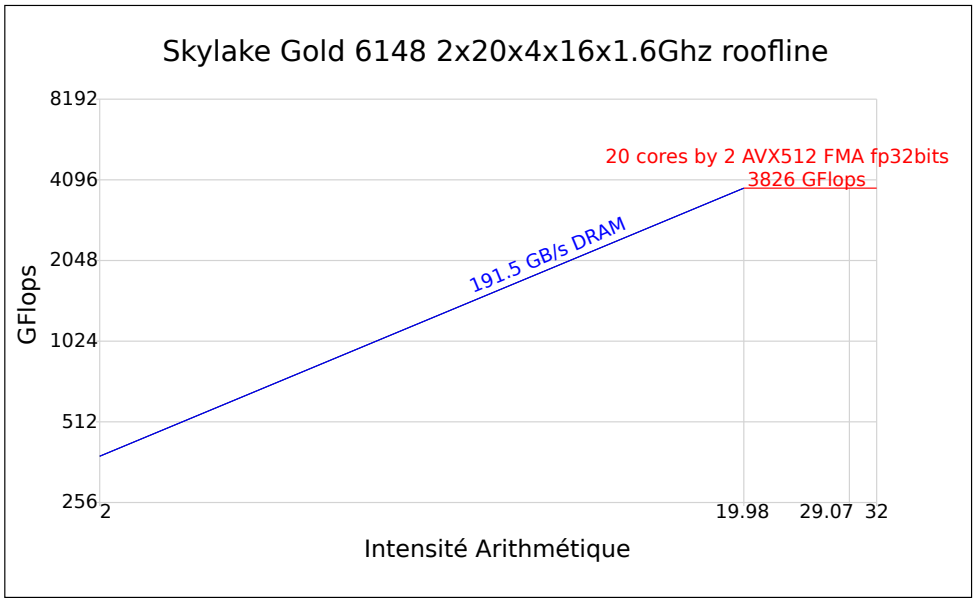

FIGURE 2.20 – ORM d'un Skylake Intel Xeon Gold 6148 double nœud NUMA.

les performances de tels noyaux, il va falloir prendre en compte les performances de la mémoire. À l'instar des performances de calcul, il est possible d'estimer théoriquement et expérimentalement les performances de la mémoire.

Nous verrons dans la prochaine section comment estimer nos premiers indicateurs selon une estimation théorique ou expérimentale. D'autres indicateurs peuvent être utilisés pour essayer de caractériser les programmes de calcul. Par exemple, l'intensité arithmétique d'un programme peut être évaluée en estimant le nombre d'octets transférés pour un nombre d'opérations de calculs réalisés. Plus sa valeur est faible et plus le calcul sera limité par la capacité de transfert des données. Inversement, plus il est fort et plus le calcul sera limité par la capacité de calcul.

Ces indicateurs sont utilisés dans différents modèles. Le modèle roofline employé dans cette thèse est un modèle représentant graphiquement des limites techniques d'une machine. Ces limites sont les indicateurs de performance énoncés précédemment. Afin d'illustrer l'explication de ce modèle, la figure 2.20 représente la machine Skylake précédemment citée.

Comme son nom l'indique, ce modèle délimite les capacités techniques considérées pour une machine donnée via des lignes de plafonnement. Différents niveaux de limites peuvent être considérés suivant le champ d'application du modèle (utilisation ou non d'instructions vectorielles par exemple). Le modèle que nous employons ici est l'ORM pour « Original Roofline Model ». Ce modèle caractérise une machine en considérant au minimum 2 paramètres : une performance de calcul et un performance de transfert mémoire. L'avantage de ce modèle est qu'il s'illustre graphiquement ce qui le rend assez intuitif comme on peut le voir dans la figure 2.20. Ce modèle permet de déduire la performance de calcul maximal que l'on peut espérer atteindre en fonction de l'intensité arithmétique d'un calcul exécuté via la machine caractérisée dans l'ORM. Premièrement, quel que soit le noyau de calcul, ce dernier ne peut dépasser la performance maximale de calcul considérée dans le modèle (limite horizontale rouge sur la figure 2.20). Ensuite, suivant l'intensité arithmétique, la

performance de calcul sera également limitée par la performance des transferts mémoires (ligne bleue sur la figure 2.20). Ainsi, la performance maximale de calcul que l'on peut espérer correspond au facteur le plus limitant des 2.

Dans le modèle roofline Fig. 2.20, le point où se rejoindre la courbes montantes et la ligne horizontale correspond au point d'équilibre caractéristique de la machine suivant ce modèle. Il s'agit du facteur d'intensité arithmétique minimum à partir duquel on peut espérer atteindre la performance de calcul maximale de la machine. Les programmes de calcul allant au-delà de ce point sont limités par la puissance de calcul de la machine. Il peut être calculé simplement en divisant les performances maximales de calcul par celle de transfert mémoire. Ainsi, en reprenant notre plateforme Skylake de la figure 2.20 avec une performance de calcul expérimental maximum de 3826 GFLOPS et une performance de transfert atteignant 191.5 GB/s , notre point d'équilibre se situe au facteur $\frac{3826}{191.5} = 19.979$ calculs par octet transféré. Autrement dit, si un noyau de calcul parvient à réaliser un peu plus de 19 opérations pour chaque octet qu'il transfère, alors ce dernier peut atteindre la performance limite de calcul de la machine. Atteindre la performance de calcul maximal de la machine reste malgré tout délicat.

En effet, cette limite maximale est estimée avec les instructions vectorielles disponibles les plus puissantes à chaque cycle ce qui suppose un pipeline parfait des mêmes instructions pour tous les cœurs à la fois. Lorsque les noyaux de calcul considérés ne peuvent pas bénéficier de telles instructions, des performances maximales plus réalistes doivent être considérées afin d'être plus proches de la réalité. Il est ainsi possible de considérer plusieurs plafonds de performance que ce soit pour le calcul ou les transferts mémoires. On appelle cela un sous-seuillage. Il est donc possible que le graphique d'une roofline soit composé de plusieurs courbes. L'intérêt principal de ce modèle est de situer les noyaux de calculs face aux plateformes d'exécutions. Or, l'usage d'instructions vectorielles FMA de l'architecture Skylake permet d'atteindre le niveau de performances de calcul maximal de la machine. Cependant, cette performance maximale n'a pas de sens pour un programme ne réalisant que des additions. Ainsi, si un programme ne peut bénéficier des instructions les plus puissantes, car il n'en a tout simplement pas le besoin, il faut donc considérer sur le graphique roofline un sous-seuillage adapté à ce dernier.

Dans nos rooflines, les performances maximales considérées sont expérimentalement estimées via des programmes se devant d'approcher les maximums techniques de nos machines dépourvues d'accélérateurs (ex. GPU). Ainsi, nos cas d'usage ont le potentiel de faire appel à toutes les ressources de calculs du CPU telles que les instructions vectorielles et les différents cœurs. Ils peuvent également solliciter les différents canaux mémoires.

D'autres modèles de roofline existent [31, 45, 21] et précisent le modèle que nous employons. Ces autres modèles permettent de prendre en compte des paramètres supplémentaires liés à des caractéristiques de la machine tels que la vitesse de la mémoire cache (CARM) ou des paramètres de la localité des modules mémoire (LARM). Préciser le modèle consiste à prendre en compte des paramètres plus spécifiques afin de couvrir des aspects ayant leur importance dans les expériences menées.

Nous venons de voir comment paramétrer notre modèle suivant des valeurs estimées soit de manière théorique soit de manière expérimentale. Étant donné que les processeurs peuvent disposer de compteurs matériels capables de comptabiliser certains évènements par exemple les défauts de cache (cache miss) impliquant des accès à la mémoire RAM. Ces compteurs peuvent être employés afin de visualiser dynamiquement à l'exécution des

calculs pour obtenir de précieuses informations. Ces informations peuvent être exploitées dans des modèles afin de mieux comprendre les interactions entre le matériel et le logiciel. Ceci nous amène à la section suivante qui présente les différents outils logiciels. Nous verrons ainsi que ces derniers assurent des mesures statiques ou dynamiques des différents indicateurs et qu'ils peuvent présenter les résultats afin d'aider l'analyse.

2.3.2 Outils logiciels d'analyse

La section précédente montre qu'il est possible de paramétrer des modèles afin d'inférer et d'analyser des résultats. Des logiciels assurant la capture d'indicateurs et en assistant l'analyse ont vu le jour. Le choix de ces derniers peut dépendre des architectures cibles, des indicateurs et modèles d'analyse, mais aussi de leur coût. Le coût est aussi bien le montant monétaire des licences, le prix des formations, mais aussi l'investissement en temps afin de maitriser l'outil ou la surcharge induite pour la machine (overhead).

Le tableau 2.1 liste une série d'outils logiciels permettant d'investiguer sur les comportements des programmes. La liste de ces outils que sont PAPI[8], MAQAO[9], SDE[10], Vector Advisor[11] et VTune[12] se trouvent surs internet.

TABLE 2.1 – Liste d'outils logiciels d'analyse d'indicateurs de performance

Nom de l'outil	Licence	Statique	Dynamique	Instrumentation	Type
PAPI	Gratuite	Non	Oui	Oui	Librairie
MAQAO	Gratuite	Oui	Oui	Non	Application
SDE	Gratuite	Oui	Non	Non	Application
Vector Advisor	Payante	Oui	Oui	Non	Application
VTune	Payante	Oui	Oui	Non	Application

Le principe de l'analyse statique consiste à désassembler l'exécutable afin entre autres de comptabiliser les opérations. Des conseils peuvent être apportés notamment lorsqu'un schéma d'instructions mal optimisé est détecté. Par exemple, l'outil SDE réalise un bilan des instructions ce qui permet entre autres d'évaluer le nombre de calculs flottants dans le programme. Le code dans un langage de haut niveau peut également être lu pour proposer des optimisations. L'avantage de l'analyse statique est qu'il n'y a pas de coût de traitement durant l'exécution du programme. Le désavantage est que cette analyse suit des modèles et peut passer à côté de ce qui se passe réellement sur la machine durant l'exécution.

L'analyse dynamique peut quant à elle soit superviser l'exécution du programme et espionner des indicateurs matériels soit faire partie du programme lui-même. Un programme de calcul peut participer à sa propre analyse et on appelle cela l'instrumentation du programme. La librairie PAPI propose plusieurs fonctions de plus ou moins haut niveau pour mesurer des indicateurs depuis le programme lui-même. Ceci laisse à la charge du développeur le paramétrage des fonctions d'introspection de la librairie afin de récupérer les

8. Site de PAPI http://icl.cs.utk.edu/papi/.
9. Site de maqao http://www.maqao.org.
10. software.intel.com/en-us/articles/intel-software-development-emulator.
11. Site de Vector Advisor https://software.intel.com/en-us/advisor.
12. Site de VTune https://software.intel.com/en-us/vtune.

indicateurs désirés. L'avantage de PAPI est qu'il propose de s'abstraire du matériel, mais cette abstraction a ses limites. En effet, lorsque le matériel change, il est possible que les compteurs n'aient pas tout à fait la même signification. Il faut rester vigilant quant aux valeurs que l'on récupère via les compteurs. De plus, les compteurs peuvent ne pas être implémentés sur certains matériels. Afin d'illustrer ce point, nous prendrons l'exemple des opérations flottantes 32bits. Lorsque l'on demande à PAPI de récupérer le compteur du nombre d'opérations flottantes, il se peut que seules les opérations flottantes d'instruction scalaire soient prises en compte. Le fait d'utiliser des instructions vectorielles peut-être compatibilité dans un autre compteur. De plus, il faut connaitre la signification précise des valeurs de ces compteurs. L'unité d'un compteur d'exécution d'instruction vectorielle correspond par exemple au nombre d'exécutions de ces instructions et non au nombre d'opérations flottantes réalisées.

Des difficultés similaires peuvent se retrouver dans l'utilisation d'outils plus sophistiqués. Les applications de profilage Vector Advisor et vTune d'Intel proposent de collecter des indicateurs sous forme de traces d'exécution afin d'en faire une analyse ultérieure. Pour ce faire, une base de données est alimentée durant l'exécution du programme profilé ce qui peut avoir un coût en ressource. Cette base de données est ensuite ouverte par l'outil pour présenter les résultats sous forme de graphiques ou de valeurs. Comme vu précédemment, on peut avoir à faire à des indicateurs dont la définition n'est pas toujours évidente à comprendre et surtout dont l'interprétation n'est pas claire.

Enfin, ces applications font appel à des modèles comme nous avons pu en discuter dans la section 2.3.1. Par exemple , Vector Advisor génère une série de graphiques de type roofline. De plus, cette application peut prodiguer des conseils d'optimisations comme le fait MAQAO en particulier sur l'utilisation des instructions SIMD ou les schémas d'accès mémoire.

Il existe de nombreux outils proposant plus ou moins de fonctionnalités. Nous verrons que certains d'entre eux sont utilisés dans cette thèse afin de comprendre la relation comportementale entre les programmes de calcul et les plateformes d'exécution.

2.4 Positionnement du travail

Cette thèse s'inscrit dans une démarche visant à optimiser des codes de simulations appliquées notamment aux géosciences. L'objectif est de proposer des solutions d'optimisation pour les scientifiques concevant des logiciels de simulation. Ces simulations se fondent sur la résolution d'EDP (EDP). À cette fin, l'état de l'art présente dans la section 2.1 les origines des noyaux de simulation numériques afin d'en préciser 2 catégories. Ainsi, la catégorie des stencils est détaillée dans la section 2.1.2. Elle est suivie des noyaux d'assemblage éléments finis dans la section 2.1.3. Face aux besoins en ressources de calcul, les capacités matérielles ainsi que leurs fonctionnements sont abordés dans la section 2.2.

La thèse s'oriente alors vers une évaluation d'optimisations implicites face à des approches cumulant plusieurs niveaux de parallélisme incluant des approches originales. Les chapitres 3 et 4 détaillent respectivement les travaux de la thèse autour des 2 catégories de noyaux. Elles ont en communs des capacités à bénéficier d'optimisations SIMD et SPMD. L'optimisation via la réutilisation de données déjà chargées dans les processeurs permet également d'optimiser les accès mémoires. L'évaluation de ces optimisations est encadrée

par des modèles d'analyse adaptés pour chacune de ces catégories de noyaux. Ainsi, nous proposons dans cette thèse des modèles d'analyse évalués et mis au point afin d'expliquer les résultats. Ces modèles et outils sont présentés dans l'état de l'art à la section 2.3. Dans le cas où des résultats ne concordent pas avec les modèles d'analyse, des investigations sont menées afin de déterminer leurs limites.

Des mécanismes d'optimisations implicites sont déjà à l'œuvre que ce soit via les compilateurs ou via des environnements d'exécutions de plus haut niveau (OpenMP, Pochoir, Squelettes algorithmiques ...). En recoupant plusieurs implémentations différentes des mêmes noyaux avec nos modèles et outils d'analyse, nous apprécions leur efficacité. Ainsi, l'aspect implicite des optimisations déjà à l'œuvre est évalué dans le but d'envisager de meilleures perspectives pour ces 2 catégories de noyaux. En effet, le cumul des optimisations des parallélismes intra-nœud requiert des efforts constants du fait de l'évolution des architectures. Cependant, les approches mettant en œuvre ces optimisations se doivent d'être au plus près de critères tels que la maintenabilité et l'utilisabilité. Les critères de ces optimisations sont multiples et pas nécessairement compatibles entre eux. Un effort de transparence est également attendu sur ces optimisations qui sont parfois perçues telles des boites noires. Elles doivent également garantir un bon niveau de performance. Par conséquent, la mise en œuvre de ces optimisations doit rester compréhensible. En effet, un utilisateur de telles solutions devrait être en mesure de vérifier le code généré par la solution afin d'en apprécier l'exactitude vis-à-vis de son implémentation initiale. Ainsi, nous verrons qu'il faut savoir positionner le curseur entre performance et transparence notamment lors de la comparaison avec Pochoir dans la section 3.7.

Chapitre 3

Optimisation de noyaux stencils

3.1 Contexte

Pour la résolution des d'EDP associées aux grands problèmes physiques, les stencils de calcul jouent un rôle clé. En effet, ces derniers apparaissent lors de la phase de discrétisation de ces équations traduisant l'approximation des dérivées spatiales du problème.

De nombreux travaux ciblent cette catégorie de noyau numérique [14, 19, 18, 17, 22, 4, 42]. En effet, il s'agit de faire progresser la simulation dans le temps en appliquant ce schéma de calcul plusieurs centaines de milliers de fois. Par conséquent, ce dernier requiert de nombreux calculs pouvant représenter la majeure partie du temps d'exécution d'une simulation.

En ce qui concerne le déroulement du calcul, pour chaque application d'un stencil sur une grille, chaque cellule du domaine est recalculée suivant une pondération de son voisinage (en espace-temps) décrit par le stencil.

La morphologie du stencil (c'est-à-dire le nombre de cellules ainsi que leur répartition espace-temps) a un impact sur la performance globale du noyau. Dans ce chapitre, nous considérons les stencils classiques 7-point et 27-point. Nous verrons que leurs caractéristiques (nombre de cellules, intensité de réutilisation) diffèrent, ce qui en fait des stencils représentatifs. Pour cette raison, ils sont souvent étudiés dans la littérature scientifique [17]. Les principales contributions de ce chapitre sont les suivantes. Tout d'abord, nous y soulignons l'efficacité de chaque niveau d'optimisation (vectorisation, tuilage, composition stencil) en exploitant un ensemble de compilateurs représentatifs (Clang 3.8, Intel 17 et GCC 6.2). De plus, nous montrons la portabilité de notre méthodologie sur deux plates-formes Intel base de bi-processeurs (Ivy Bridge et Broadwell). Enfin, nous positionnons nos résultats par rapport à un modèle théorique roofline [67] et une implémentation de référence réalisée avec le compilateur Pochoir [61].

3.2 Choix des stencils expérimentaux

3.2.1 Les stencils 3D 7-point et 27-point

Nous considérons deux stencils classiques couramment étudiés dans la littérature scientifique [53, 19]. Le premier exemple est un stencil 3D 7-point composé d'une cellule centrale et de six cellules voisines dans chaque direction d'espace (Figure reffig :s7p27pSchema).

FIGURE 3.1 – Représentation visuelle des stencils 7-point et 27-point.

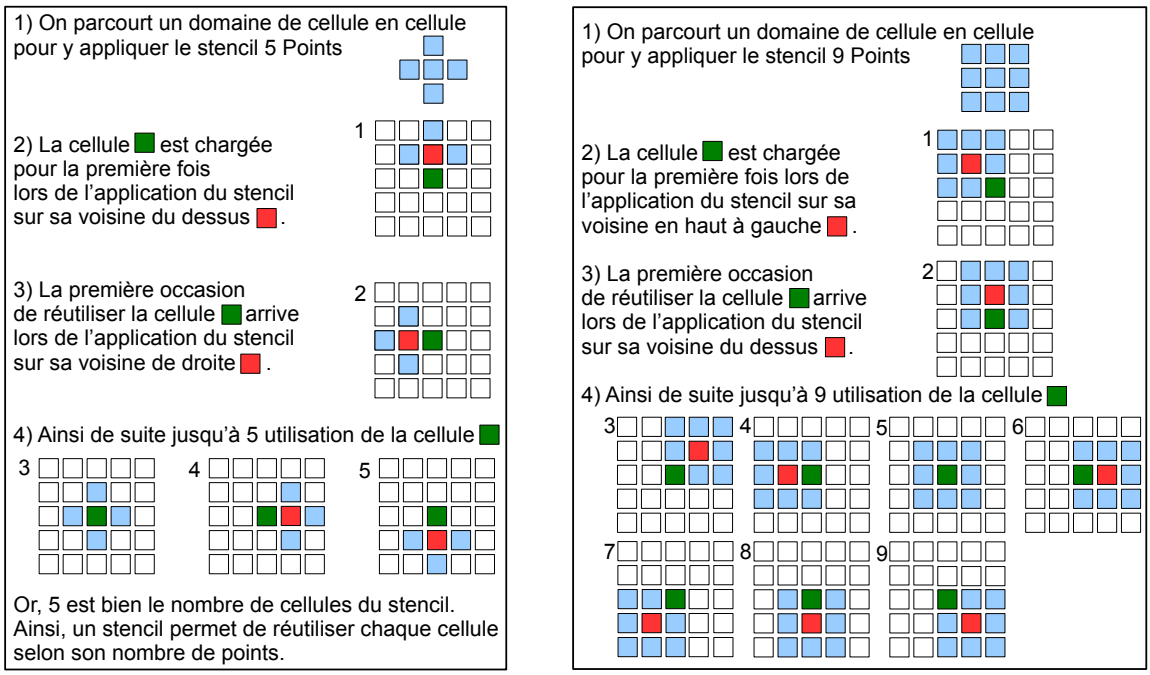

FIGURE 3.2 – Illustration du fait qu'un stencil de N points peut à chaque itération utiliser au maximum N fois une même cellule. En effet, lors de ces 2 exemples, la cellule au centre du domaine est employée à chaque calcul de son voisinage selon le voisinage du stencil appliqué.

Le deuxième exemple est un stencil 3D 27-point composé d'un total de 27 cellules à savoir les 26 cellules voisines comprises dans le cube de $3 \times 3 \times 3 \times 3 \times 3$ autour de la cellule centrale comme décrit dans la figure 3.1.

Ces stencils sont représentatifs en termes de capacité de calcul et de transferts mémoires. Pour une itération indépendante, chaque donnée du domaine de calcul sera chargée au moins une fois (idéalement une seule fois) depuis la mémoire centrale vers les mémoires caches. La figure 3.2 montre que chaque donnée est réutilisée autant de fois que le stencil a de cellules. Nous définissons l'intensité de réutilisation (RI) comme un ratio de données réutilisées par octets chargés pour chaque application du stencil. Il est déterminé par le nombre d'opérations du stencil divisé par le nombre d'octets (quatre octets dans notre cas). De plus, nous réalisons un simple cumul des cellules, ce qui implique que chaque flottant chargé peut au mieux-être réutilisé autant de fois que le nombre de cellules.

Le stencil 7-point effectue l'accumulation de 7 cellules et chaque cellule contient une

```
#define COORD3D(Z,Y,X) (((Z)*n+(Y))*n+(X))

float *domaineA = new float[n*n*n];
float *domaineB = new float[n*n*n];
initialiseDomaine(domaineA,domaineB);

for(int itr=0;itr<nbItr;++itr)
{
 for(int k=1;k<(n-1);++k)
  for(int j=1;j<(n-1);++j)
   for(int i=1;i<(n-1);++i)
    apply7PtsV8(domaineA,k,j,i);

 float *swtch=domaineA;
 domaineA=domaineB;
 domaineB=swtch;
}

delete[] domaineB;
return domaineA;
```

FIGURE 3.3 – Implémentation C++ d'un parcours scalaire appliquant le stencil 7-point.

valeur en virgule flottante de 32 bits (4 octets). Dans ce cas, le facteur d'intensité de réutilisation (RI) est de $7\ /4 = 1,75$. Les plateformes employées ici ont un ratio entre le calcul et la bande passante supérieur à 7. Par conséquent, un octet transféré doit être réutilisé plus de 7 fois en moyenne afin d'atteindre la capacité de calcul maximal de la machine. Ainsi, on dit que le stencil 7-point est limité par la bande passante mémoire. Pour le stencil 27-point, le RI est de $6,75$, ce qui indique que ce stencil est moins limité par la bande passante mémoire.

3.2.2 Implémentation stencil classique

L'application d'un stencil sur une grille de points consiste à parcourir cette grille et à en recalculer chaque point via le schéma de calcul du stencil. Dans le cas d'une grille cartésienne tridimensionnelle, le calcul consiste à utiliser les points voisins dans les directions ascendantes descendantes, gauche droite et avant-arrière pour calculer la nouvelle valeur d'un point. L'algorithme passe ensuite au point suivant en appliquant le même schéma de calcul jusqu'à ce que toute la grille ait été parcourue. Ce dernier correspond à l'exemple du code de calcul 3.3, chaque point est recalculé suivant le schéma stencil 7-point implémenté par la fonction 3.4. On remarque qu'aucun coefficient n'est appliqué aux données voisines. Ce sera le cas des stencils étudié dans ce chapitre afin de mettre l'accent sur le verrou de la bande passante mémoire.

En partant de cette version séquentielle classique, il est assez simple de mettre en place un premier niveau de parallélisme multi-cœurs. En effet, cette imbrication de plusieurs boucles peut-être facilement décomposée. Prenons l'exemple du stencil 3D Jacobi classique à 7 points réalisant une somme pondérée de ces voisins dans un domaine de $n*m*l$ valeurs flottantes simple précision pour chaque itération. Sur les plateformes multi-cœurs, une

```
inline void apply7PtsV8(float *domaineA, int k, int j, int i)
{
 domaineB[COORD3D(k,j,i)] =
  domaineA[COORD3D(k,j,i)]+
  domaineA[COORD3D(k,j,i-1)]+
  domaineA[COORD3D(k,j,i+1)]+
  domaineA[COORD3D(k,j-1,i)]+
  domaineA[COORD3D(k,j+1,i)]+
  domaineA[COORD3D(k-1,j,i)]+
  domaineA[COORD3D(k+1,j,i)];
}
```

FIGURE 3.4 – Implémentation C++ classique d'un stencil 7-point non pondéré.

façon classique d'extraire le parallélisme est d'exploiter les boucles imbriquées provenant des dimensions spatiales du problème. Chaque niveau de boucle représente le parcours de chaque dimension de 0 à l, m et n. L'un des principaux avantages est une utilisation simple des directives OpenMP. Le parallélisme OpenMP appliqué à la boucle externe réalise l tranches de mn cellules. Nous y reviendrons plus en détail dans la section 3.5 sur le multithreading.

La section suivante définit les architectures cibles d'exécution des noyaux de nos stencils de référence dans cette étude.

3.3 Plateformes expérimentales cibles

3.3.1 Les plateformes de calculs, compilateurs et environnements d'exécution

Les deux plateformes bi-processeurs utilisées pour ce chapitre sont décrites dans le tableau 3.1.

TABLE 3.1 – CARACTÉRISTIQUES DES PLATEFORMES

Platforme	**Broadwell**	**Ivy Bridge**
cœurs physiques	2×18	2×12
fréquence de base	2.3GHz	2.7GHz
fréquence avec AVX	2.0GHz	2.2GHz

Afin de réduire les effets venant du au compilateur, nous considérons trois options (Clang 3.8, GCC 6.2 et ICC 17 avec les flags d'optimisation `-O3 -march=native`). Le flag « march » fixé à la valeur native permet implicitement le support AVX et FMA de ces architectures compatibles. OpenMP est utilisé pour l'implémentation du parallélisme à mémoire partagée.

3.3.2 Rooflines des plateformes

La sous-section 2.3.1 présente le modèle d'analyse roofline. L'objectif est d'analyser les graphiques fig. 3.5 correspondants aux plateformes Broadwell et Ivy Bridge et de discuter

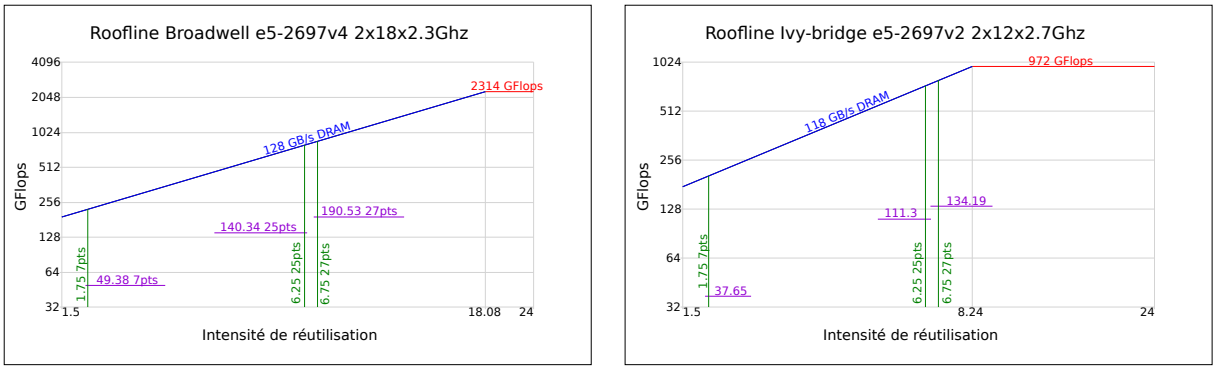

FIGURE 3.5 – Modèles roofline expérimentaux Stream et Linpack des plateformes Broadwell et Ivy Bridge.

des performances expérimentales obtenues de nos implémentations.

Dans le premier cas, le noyau 7-point est fortement memory-bound, sa performance peak est de l'ordre de 100 GFLOPS pour les deux plateformes.

En ce qui concerne le stencil 27-point, son RI est plus élevé (6,25 contre 1,75 pour le 7-point) du fait de son plus grand nombre de cellules. Par conséquent, la performance maximale atteint les 413 GFLOPS sur la plateforme Broadwell et 368 GFLOPS sur la plateforme Ivy Bridge.

3.3.3 Paramètres expérimentaux

Pour tous les noyaux de ce chapitre, le domaine de calcul est composé de valeurs en virgule flottante de simple précision sur une grille régulière cubique de 512^3. Par conséquent, l'empreinte mémoire est d'un ordre de grandeur supérieur à la taille du dernier niveau de mémoire cache L3. Pour chaque exécution, nous réalisons 100 itérations d'application du stencil expérimenté.

La politique de positionnement des threads consiste à interdire leur migration et d'alterner leur placement sur chaque nœud NUMA. En effet, lorsqu'un thread s'exécute sur un cœur, les données de calcul du thread sont remontées dans la mémoire cache de ce cœur. Si le thread est migré, ses données en mémoire cache ne le suivent pas et devront être rechargées depuis le L3 voir la mémoire centrale ce qui est couteux.

Lors de chaque exécution, la commande interleave de la librairie NUMA assure une répartition alternée des plages de données mémoires sur l'ensemble des nœuds. En effet, avant l'exécution en parallèle des calculs, le thread principal alloue les structures de données. Sans cette commande NUMA, le comportement d'allocation physique par défaut se contente d'allouer autant que possible sur le même nœud mémoire le plus proche de ce thread principal. De fait, cette commande permet de s'assurer un équilibrage des accès sur plusieurs canaux mémoires. Cependant, la mémoire étant allouée sans notion de localité avec les threads y accédant, ceci induit une forte sollicitation des bus intra cœur. D'une part les accès via ces bus sont plus couteux et d'autre part ils peuvent tout simplement se retrouver saturés. Une meilleure approche serait de laisser les threads de calcul allouer et charger leurs données. On appelle ce type d'approche, une approche NUMA aware. Des tests de performances avec et sans interleave ainsi que la fixation ou non des threads

ont été réalisés. Il en ressort que l'utilisation de la commande d'interleave permet de doubler les performances. Le fait de fixer les threads permet d'obtenir des performances plus stables d'une exécution à l'autre ainsi que des gains de performances plus élevées. Le chapitre 4 présente plus en détail les avantages de l'entrelacement mémoire (interneave).

Les paramètres expérimentaux étant fixés, abordons la section suivante sur les optimisations des noyaux en commençant au plus bas niveau de notre étude.

3.4 Vectorisation

3.4.1 Passage au calcul vectoriel

Dans notre étude sur les stencils, nous utilisons des unités AVX et AVX-2 supportées sur nos plateformes Intel. Ces unités de traitement sont capables d'opérer avec des vecteurs de 256bits. Ainsi, elles rendent possible le traitement en parallèle de 8 flottants simples précisions. Des compilateurs proposent une vectorisation automatique implicite, mais il n'y a aucune garantie qu'ils parviennent à vectoriser de manière efficace. Ainsi, le flag -o3 demande au compilateur d'optimiser et ce dernier va tenter de vectoriser. Néanmoins, le flag -march=native permet de lui préciser qu'il doit utiliser les spécificités de la machine sur laquelle il compile. En effet, en son absence, sous x86, il emploie les instructions SSE(4 flottants 32bits) alors que les instructions AVX peuvent être disponibles sur la plateforme cible. En résumé, il est nécessaire de préciser la cible au compilateur afin qu'il mette en oeuvre de meilleures optimisations.

Afin de surmonter ces différents verrous, nous proposons une version explicitement vectorisée via des intrinsics. En effet, le développeur est souvent capable de fournir des indications précieuses pour l'optimisation de son application. Néanmoins, afin d'améliorer la lisibilité du code, des fonctions d'abstraction vectorielle inline sont implémentées avec des intrinsics. La figure 3.6 synthétise la stratégie implémentée.

En complément, des opérations supplémentaires de transformation sont nécessaires. En effet, il faut constituer les différents vecteurs contenant les valeurs de la grille. En reprenant la figure 3.6, on remarque qu'un registre vectoriel doit charger les valeurs des cellules 11 à 18. Pour cela, deux options sont disponibles. Comme nous l'avons vu dans la sous-section 2.2.2, il est possible de charger un vecteur quelque soit l'alignement mémoire, mais au détriment de la performance par rapport à un chargement aligné. Dans notre exemple, on considère que les cellules marquées d'une étoile * sont à des adresses mémoires alignées sur 32 octets. Ceci permet de charger de façon alignée les vecteurs A, B, C, D et E. Les couples de vecteurs (B,C) et (C,D) sont shiftés de façon additionner le voisinage gauche et droit du stencil. Cette opération de décalage est plus rapide que de réaliser des chargements non alignés. En effet, on pourrait directement chargé le vecteur depuis la dernière cellule de B ainsi que le vecteur depuis la seconde cellule de C mais ces adresses ne sont pas alignées ce qui est moins performant. De plus le vecteur C est déjà chargé pour les besoins du calcul, on peut ainsi le réutiliser. La difficulté est que l'opération de décalage vectoriel n'existe pas en tant que telle. Il nous faut une instruction de type shift capable de prendre en entrée deux vecteurs et un nombre de décalages de flottants d'un vecteur à l'autre. Cette dernière restitue un vecteur composé des deux entrées suivant le

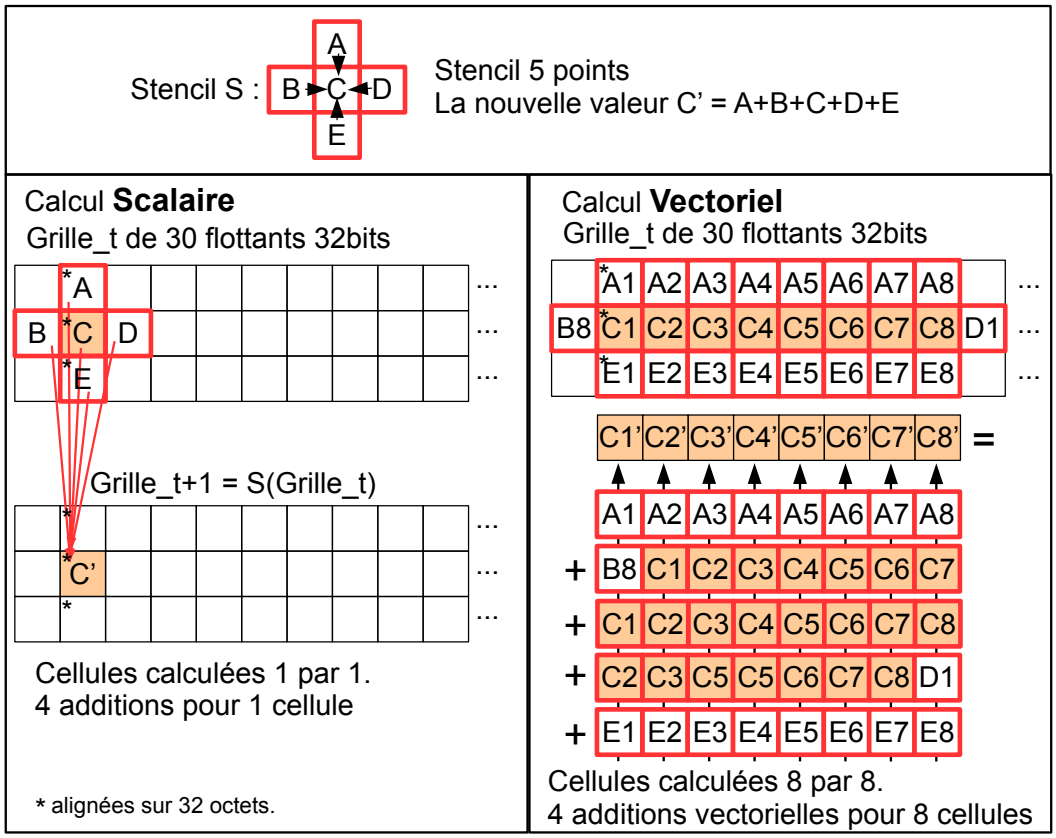

FIGURE 3.6 – Exemple d'un calcul stencil 5-point réalisé en scalaire puis en vectoriel.

nombre de valeurs à décaler.

3.4.2 Uniformisation des familles d'instructions SIMD

Pour les travaux de cette thèse, un fichier d'entête C++ a été défini afin d'uniformiser des opérations de vectorisation, quelques soit la taille du vecteur utilisé. En effet, suivant les différentes générations d'instructions vectorielles, plusieurs tailles de vecteurs existent. Or, certaines instructions opérant sur des registres de 128bits n'ont pas nécessairement leur équivalence dans les instructions opérant sur des registres 256bits. De plus, des processeurs plus récents peuvent proposer des jeux d'instructions additionnelles complétant les instructions d'une taille de registre déjà existant. Par exemple, les processeurs Intel ont commencé par supporter l'AVX apportant les registres vectoriels de 256bits. Puis, sont arrivés des processeurs supportant l'AVX-2. L'AVX-2 ne change pas la taille des registres qui restent au maximum de 256bits, mais l'AVX-2 apporte de nouvelles instructions vectorielles. Ainsi, la fonction de décalage ne s'implémente pas de la même façon selon si on travail avec des vecteurs de 128 bits ou 256bits. Cela dépend également des jeux d'instructions disponibles. Typiquement, les vecteurs de 128bits font partie des jeux d'instructions SSE. Une instruction « alignr » permet de réaliser directement ce décalage vectoriel. Pour les vecteurs de 256bits, le jeu d'instructions AVX ne permet pas d'implémenter directement l'opération de décalage. Il faut donc utiliser plusieurs instructions AVX pour y parvenir (entre 2 et 3 opérations suivant le nombre de décalages à réaliser). Enfin, le jeu d'instruction AVX-2 apporte des instructions supplémentaires pour les vec-

```
#define Alig16(V) ((V&~0b1111)+(V&0b1111?1<<4:0))
#define CORD(Z,Y,X) (((Z)*n+(Y))*Alig16(n)+(X+16))
#define AllocSize sizeof(float)*n*n*(16+Alig16(n)+16)

//Fonction de la librairie boost
float *dmA = (float*)aligned_alloc(64,AllocSize);
float *dmB = (float*)aligned_alloc(64,AllocSize);
initialiseDomaine(dmA);

for(int itr=0;itr<nbItr;++itr)
{
 for(int k=1;k<(n-1);++k)
  for(int j=1;j<(n-1);++j)
   for(int i=0;i<n;i+=8)
    apply7PtsV8(dmA,k,j,i);

 float *swtch=dmA;dmA=dmB;dmB=swtch;
}
aligned_free(dmB);//Fonction de la librairie boost
return dmA;
```

FIGURE 3.7 – Implémentation C++ d'un parcours par vecteurs de 8 flottants appliquant le stencil 7-point.

teurs de 256 bits, dont l'instruction « alignr » pour vecteur 256bits. Ainsi, suivant les jeux d'instructions disponibles, il faut adapter les codes sources. Afin d'illustrer ce qui vient d'être dit, la figure 3.7 présente une implémentation vectorielle de l'implémentation 3.3 vue à la sous-section 3.2.2. La figure 3.8 présente l'implémentation à l'aide des intrinsics de la fonction d'application du stencil 7-point *apply7Pts*.

Cette implémentation vectorielle est réalisée avec des appels d'intrinsics AVX uniquement sans utiliser d'AVX-2. Dans cet exemple nous manipulons des flottants simples précisions (32bits). De fait, on utilise le type « *__m256* » qui correspond à un vecteur de 8 flotants simples précisions. Ceci permet d'opérer 8 flottants à la fois, ainsi ils sont chargés puis cumulés suivant le voisinage du stencil pour enfin sauvegarder leurs résultats. Il est intéressant de noter que les versions récentes des compilateurs surchargent les opérateurs de bases entre ces vecteurs, mais cela n'a pas toujours été le cas. En effet, sans ses surcharges d'opérateurs, il faut manuellement appeler les fonctions d'opérations intrinsics tel que l'instruction d'additions « *_mm256_add_ps(v1,v2)* ». Lorsque la cible est une plateforme disposant des instructions AVX-2, le code peut être légèrement simplifié pour réaliser les décalages entre les vecteurs *« cur »* et *« vgd »*. L'avantage est de pouvoir remplacer 3 instructions par une seule. On peut ainsi espérer gagner en performance de calcul.

Ces variations d'une plateforme à l'autre impliqueraient une maintenance de différentes implémentations. Cela devient vite difficile à gérer, le code devient difficilement lisible et les risques d'erreurs sont élevés. De ce faite, il est nécessaire de proposer des fonctions standardisées. Leurs implémentations peuvent s'adapter selon les ressources disponibles dans la machine cible. Nous avons précédemment abordé dans la section 2.2.2 l'existence de librairies d'abstraction découplant le calcul vectoriel du matériel. Ainsi, dans le cadre

```
inline void apply7PtsV8(float *dmA, int k, int j, int i)
{
 __m256 cur, acc;
 acc = cur = _mm256_load_ps(&dmA[CORD(k,j,i)]);
 acc+=_mm256_load_ps(&dmA[CORD(k+1,j,i)]);
 acc+=_mm256_load_ps(&dmA[CORD(k-1,j,i)]);
 acc+=_mm256_load_ps(&dmA[CORD(k,j+1,i)]);
 acc+=_mm256_load_ps(&dmA[CORD(k,j-1,i)]);
 __m256 vgd = _mm256_load_ps(&dmA[CORD(k,j,i+8)]);
 acc+=_mm256_permute_ps(_mm256_blend_ps
     (cur,
       _mm256_permute2f128_ps(cur,vgd,0x21)
     ,0x11
    ), 0x39);
 vgd = _mm256_load_ps(&dmA[CORD(k,j,i-8)]);
 acc+=_mm256_permute_ps(_mm256_blend_ps
    (cur,
      _mm256_permute2f128_ps(vgd,cur,0x21)
      , 0x88
     ), 0x93 );
 _mm256_store_ps(&dmB[CORD(k,j,i)],acc);
}
```

FIGURE 3.8 – Implémentation C++ intrinsics AVX d'un stencil 7-point non pondéré.

de cette thèse, des fonctions de substitution ont été mises au point afin d'implémenter des opérations vectorielles selon les jeux d'instructions disponibles à la compilation. Du point de vue utilisateur, on fait appel à des fonctions manipulant des vecteurs dont la taille est automatiquement définie. À ce titre, une valeur constante précise la taille des vecteurs utilisés bien qu'il soit possible de forcer une taille de vecteur spécifique.
L'exemple de la figure 3.8 peut ainsi être retravaillé avec nos fonctions de substitution baptisées IntrinsiX 3.9. Ces fonctions de substitution permettent de réécrire l'exemple précédent tel qu'on peut le voir avec la figure 3.9.

3.4.3 Évaluation expérimentale de notre approche vectorielle

L'évaluation des performances repose sur les trois implémentations décrites ci-dessous.

— Version-1 Vectorisation interdite : la vectorisation est désactivée par le flag de compilation `-fno-tree-vectorize`. Cette version correspond à l'implémentation séquentielle classique, dans ce cas nous empêchons toute optimisation de vectorisation automatique du compilateur.
— Version-2 Vectorisation automatique : les compilateurs peuvent vectoriser automatiquement le code. Cette implémentation correspond à la base de référence pour notre comparaison puisque l'ensemble des optimisations automatiques est autorisé.
— Version-3 Vectorisation manuelle : la vectorisation manuelle est effectuée explicitement dans le code comme décrit dans la sous-section 3.4.1.

La première version correspond à l'implémentation classique du calcul stencil. Un flag est précisé à la compilation interdisant l'usage d'optimisations vectorielles. Cette version nous permet de mesurer les gains de performances obtenues avec les versions 2 et 3. La

```
inline void apply7PtsV8(float *dmA, int k, int j, int i)
{
 VFloat cur, acc;
 acc = cur = vLoad<VFloat>(&dmA[CORD(k,j,i)]);
 acc+=vLoad<VFloat>(&dmA[CORD(k+1,j,i)]);
 acc+=vLoad<VFloat>(&dmA[CORD(k-1,j,i)]);
 acc+=vLoad<VFloat>(&dmA[CORD(k,j+1,i)]);
 acc+=vLoad<VFloat>(&dmA[CORD(k,j-1,i)]);

 VFloat vgd = vLoad<VFloat>(&dmA[CORD(k,j,i+VSize)]);
 acc+=vShift<-1>(cur,vgd); //cur<-vgd

 vgd = vLoad<VFloat>(&dmA[CORD(k,j,i-VSize)]);
 acc+=vShift<1>(vgd,cur); //vgd->cur

 vStore(&dmB[CORD(k,j,i)],acc);
}
```

FIGURE 3.9 – Implémentation C++ intrinsiX d'un stencil 7-point non pondéré.

seconde version correspond également à l'implémentation classique, mais la vectorisation est activée dans ce cas. Enfin la troisième version est l'implémentation optimisée manuellement.
Les gains de performance sont calculés par rapport à la première version. Ces gains sont regroupés par stencil dans les tableaux 3.2 et 3.3.

Machine	Compilateur	1-Scalaire	2-Automatique	3-Manuelle
Broadwell	Clang 3.8	0,44	2,94 ($\times 6,70$)	3,97 ($\times 9,05$)
	ICC 17	2,24	2,24 ($\times 1,00$)	8,01 ($\times 3,61$)
	GCC 6.2	0,44	2,73 ($\times 6,22$)	4,14 ($\times 9,44$)
Ivy Bridge	Clang 3.8	0,50	2,98 ($\times 5,90$)	2,19 ($\times 4,35$)
	ICC 17	2,63	2,93 ($\times 1,00$)	2,43 ($\times 0,93$)
	GCC 6.2	0,50	2,91 ($\times 5,78$)	2,20 ($\times 4,36$)

TABLE 3.2 – Performances en GFLOPS et facteurs d'accélération obtenus en mono-cœur pour le stencil 7-point avec l'implémentation vectorisée automatiquement et manuellement par par rapport à la version scalaire.

Tout d'abord, on peut observer que la vectorisation manuelle est globalement plus efficace. La seule exception vient de la plateforme Ivy Bridge avec le compilateur GCC. En complément, plusieurs observations peuvent être faites. On peut noter l'incapacité pour le compilateur ICC à générer du code optimisé. Ce résultat semble surprenant en considérant une exécution sur plateformes Intel. Ce résultat illustre bien les différences de stratégie entre compilateurs. De plus, on observe un écart d'efficacité entre les programmes automatiquement vectorisés et ceux vectorisés à la main. Pour comprendre, les codes assembleurs générés par les compilateurs ont été étudiés. Bien que les stratégies de vectorisations soient proches, la principale différence provient des stratégies automatiques de chargement de vecteurs de façon non alignée. Nos optimisations manuelles présentent l'avantage de forcer l'utilisation des chargements vectoriels alignés. Une comparaison des

Machine	Compilateur	1-Scalaire	2-Automatique	3-Manuelle
Broadwell	Clang 3.8	0,88	5,52 ($\times 6,26$)	6,9 ($\times 7,82$)
	ICC 17	2,24	2,26 ($\times 1,01$)	14,98 ($\times 6,70$)
	GCC 6.2	0,88	5,47 ($\times 6,20$)	5,62 ($\times 6,39$)
Ivy Bridge	Clang 3.8	1,02	6,39 ($\times 6,25$)	6,29 ($\times 6,15$)
	ICC 17	2,42	2,42 ($\times 1,00$)	8,76 ($\times 3,61$)
	GCC 6.2	1,02	6,05 ($\times 5,92$)	5,64 ($\times 5,52$)

TABLE 3.3 – Performances en GFLOPS et facteurs d'accélération obtenus en mono-cœur pour le stencil 27-point avec l'implémentation vectorisée automatiquement et manuellement par par rapport à la version scalaire.

exemples de code 3.4 et 3.9 illustre bien les modifications apportées. En résumé, la vectorisation manuelle apporte un premier niveau de gain de performance et s'avère dans notre cas plus efficace que la vectorisation automatique.

3.5 Multithreading

3.5.1 Pilotage du multithreading

Les processeurs actuels sont pourvus de plusieurs cœurs. Chacun de ces cœurs est capable d'exécuter un ou plusieurs threads à la fois selon leur architecture. Ainsi, ces cœurs permettent d'assurer l'exécution simultanée de plusieurs applications (MPMD). Ils permettent également l'exécution simultanée d'un même programme sur différentes données (SPMD).
À la différence du SIMD vu à la section précédente où une même instruction est appliquée à plusieurs valeurs. Le SPMD correspond à un même traitement lancé en plusieurs instances capables de différer dans l'exécution. En effet, ces instances sont exécutées de manière asynchrone. Elles peuvent suivant les données prendre des chemins logiques différents. Selon l'algorithme à paralléliser, il est parfois nécessaire de synchroniser ces fils d'exécutions. Toute une chaîne de synchronisation est rendue accessible via des appels système. Toutefois, la manipulation des threads avec le système n'est pas triviale. C'est pourquoi nous faisons appel à l'environnement d'exécution OpenMP.
OpenMP n'est pas le seul environnement d'exécution. Néanmoins, il a le mérite d'être supporté sous plusieurs systèmes d'exploitations et compilateurs. Comme son nom l'indique, il est ouvert. Cependant, il faut rester vigilant sur au moins 2 points. Premier point, les implémentations de cet environnement peuvent différer d'une plateforme à l'autre ou d'un compilateur à un autre. Ceci peut avoir des répercussions sur les performances. Deuxième point, certaines optimisations automatiques peuvent être perturbées. En effet, OpenMP est intégré au compilateur et réalise des transformations du code. On a observé des cas où ces transformations perturbent les optimisations automatiques du compilateur.

3.5.2 Mise en oeuvre et découpage du calcul

OpenMP nous propose plusieurs façons de paralléliser notre algorithme. Parmi les différentes approches, OpenMP permet de partir du programme séquentiel et d'en paral-

léliser certaines boucles. L'implémentation séquentielle simple d'un calcul stencil se réalise suivant une imbrication de boucle dont le nombre dépend du nombre de dimensions. Nous travaillons en 3D dimensions ce qui nécessite le parcours sur 3 coordonnées tel que montré dans par la figure 3.10.

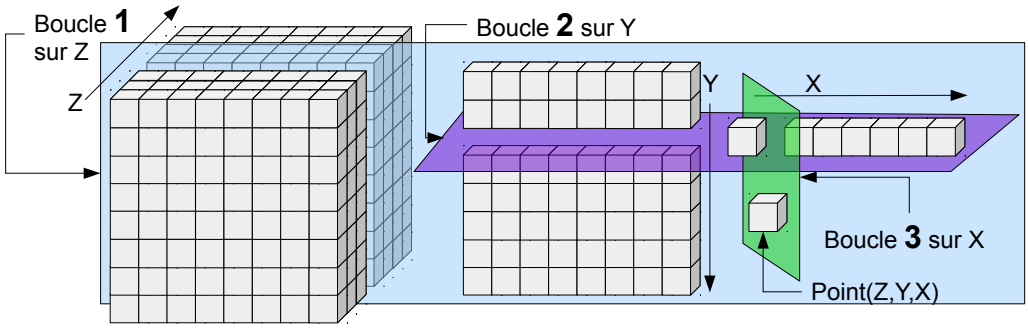

FIGURE 3.10 – L'implémentation séquentielle classique de l'application d'un stencil 3D consiste en 3 boucles imbriquées.

On remarquera que plus la boucle est externe et plus sont corps prend de temps de traitement. En effet, le corps de la première boucle la plus externe cumule récursivement toutes les boucles imbriquées. Ainsi, nous allons paralléliser la première boucle externe à l'aide de l'annotation OpenMP « parallel for ». Cette annotation indique à OpenMP que la boucle est décomposable pour être parallélisée sur plusieurs threads. Le domaine à calculer peut ainsi être réparti par lots de tranches YX sur plusieurs threads tel que montré tout à gauche de la figure 3.15. La difficulté que l'on rencontre ici est l'équilibrage de charge. En effet, le découpage s'avère relativement grossier. Dans le cas où le nombre de tranches disponible n'est pas divisible par le nombre de threads demandé. Alors, tous les threads n'auront pas le même nombre de tranches à calculer. Par exemple, avec trente tranches YX et neuf threads, chaque thread aura trois tranches, mais il restera trois tranches à répartir. Comme, elles ne sont pas davantage décomposables avec cette façon de procéder, $\frac{2}{3}$ des threads seront en famine lors du calcul des 3 dernières tranches. Lors de chaque itération d'une boucle annotée « parallèle for », les threads doivent se synchroniser conduisant à un déséquilibre de charges. L'accroissement de ce déséquilibre réduit le taux d'utilisation des cœurs et l'efficacité du programme à être parallélisé. Pour limiter ce phénomène, une option consiste à paralléliser sur la boucle Z et Y de manière fusionnée. Concrètement, OpenMP recombine les indices Z et Y en un seul qu'il répartit sur les threads. Ceci permet un découpage plus fin comme le montre le cube au centre de la figure 3.15. De fait, il est ainsi possible d'aller jusqu'à la fusion des trois boucles comme le montre le cube de droite fig. 3.15.

Cette dernière décomposition à grain plus fin permet de réduire le taux de famine à chaque fin d'itération. En revanche, dans le cas de nos plateformes, les données écrites doivent être attribuées alignées sur 64 octets pour chaque thread au risque que plusieurs threads partages une même ligne de cache. Le partage par plusieurs threads de même ligne de cache implique des invalidations de ces lignes comme expliquées dans la sous-section 2.2.1. On notera que ce phénomène est assez limité. De plus le calcul vectoriel le réduit assez naturellement à condition de traiter avec des données alignées en mémoire.
Trois autres aspects sont ici plus préoccupants. Le premier concerne le surcoût induit par

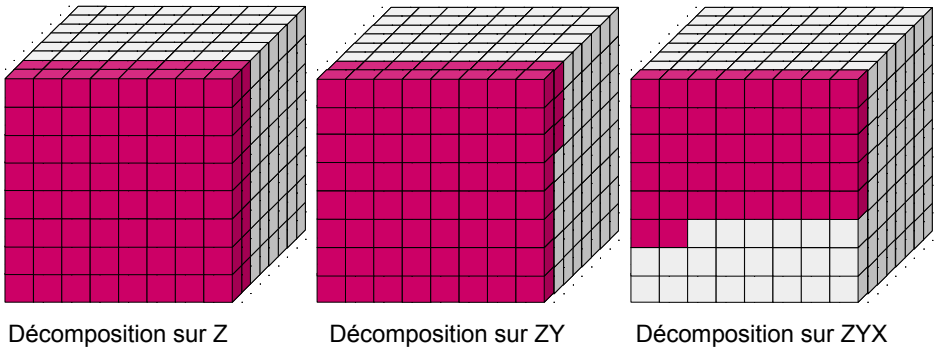

FIGURE 3.11 – Exemple de décomposition suivant les fusions de boucles.

la gestion des threads et de leur synchronisation. En effet, les appels système nécessaires à la gestion des threads sont particulièrement couteux. Cependant, la librairie OpenMP réalise certaines optimisations en allouant en une fois le nombre de threads nécessaires. De plus, la répartition du travail par un découpage des boucles fusionnées réduit dramatiquement nombre de barrières de threads par itération. La deuxième préoccupation est l'équilibrage de charge. En effet, certains algorithmes doivent réaliser des traitements différents en fonction des données. Les exemples considérés ici ne correspondent pas à ce cas. En revanche, toutes les données n'ont pas un temps d'accès identique que ce soit par effet de cache ou par effet NUMA. Dans le cadre de ces travaux, les pages mémoires de nos applications sont circulairement allouées sur les différents noeuds mémoires de façon à homogénéiser les accès. Ceci est rendu possible par l'exécution de nos applications via un binaire issue de la librairie NUMA. Ainsi, nous faisons l'hypothèse que les délais d'accès à la mémoire seront suffisent homogène afin d'obtenir des charges de calcul équilibrées. Enfin le dernier point de vigilance concerne la réutilisation des données au niveau de la mémoire cache. Ce phénomène n'est pas directement lié à la parallélisation multi-threads.

3.5.3 Évaluation des performances

Nous considérons les trois versions décrites précédemment. Pour rappel, la répartition sur différents cœurs est réalisée via le pragma « parallel for » de l'environnement OpenMP. Les résultats pour les stencils 7-point et 27-point sont regroupés dans les tableaux 3.4 et 3.5. On remarque l'impact de la vectorisation. En effet, le tableau 3.2 souligne que la vectorisation automatique avec le compilateur GCC en mono-cœur sur la plateforme Ivy Bridge s'avère meilleure que la vectorisation manuelle. En revanche, les résultats sont inversés pour une exécution parallèle (tableau 3.4). Les directives OpenMP ont un impact sur les optimisations automatiques du compilateur. Par exemple, le tableau 3.4 souligne que le compilateur Clang3.8 n'est plus capable de vectoriser automatiquement.

Ceci peut s'explique par les modifications spécifiques aux directives OpenMP qui change les transformations habituelles effectuées par le compilateur. De plus, la situation où les vecteurs ne sont pas chargés de façon alignée s'aggrave dans un contexte multi-cœurs. Ceci est dû à la répartition du domaine entre les threads. En effet, les threads partagent des lignes de caches. De fait, ils se marchent dessus et doivent vectoriser sans toucher aux données des autres threads.

Machine	Compilateur	1-Scalaire	2-Automatique	3-Manuelle
Broadwell	Clang 3.8	14,79	14,76 ($\times 1,00$)	49,49 ($\times 3,34$)
	ICC 17	24,35	25,03 ($\times 1,03$)	53,02 ($\times 2,17$)
	GCC 6.2	14,75	25,37 ($\times 1,72$)	38,54 ($\times 2,61$)
Ivy Bridge	Clang 3.8	11,60	11,60 ($\times 1,00$)	37,66 ($\times 3,25$)
	ICC 17	26,20	26,39 ($\times 1,01$)	36,91 ($\times 1,41$)
	GCC 6.2	11,60	26,34 ($\times 2,27$)	31,32 ($\times 2,70$)

TABLE 3.4 – Performances en GFLOPS et facteurs d'accélération obtenus en multi-cœurs pour le stencil 7-point avec l'implémentation vectorisée automatiquement et manuellement par par rapport à la version scalaire.

Machine	Compilateur	1-Scalaire	2-Automatique	3-Manuelle
Broadwell	Clang 3.8	29,17	29,08 ($\times 1,00$)	153,86 ($\times 4,14$)
	ICC 17	74,30	107,16 ($\times 1,44$)	190,71 ($\times 1,68$)
	GCC 6.2	29,62	106,12 ($\times 3,58$)	154,73 ($\times 4,14$)
Ivy Bridge	Clang 3.8	23,64	23,63 ($\times 1,00$)	94,20 ($\times 3,99$)
	ICC 17	49,82	111,80 ($\times 2,24$)	122,00 ($\times 2,45$)
	GCC 6.2	23,69	110,62 ($\times 4,67$)	124,76 ($\times 5,27$)

TABLE 3.5 – Performances en GFLOPS et facteurs d'accélération obtenus en multi-cœur pour le stencil 27-point avec l'implémentation vectorisée automatiquement et manuellement par par rapport à la version scalaire.

Les graphiques 3.12 et 3.13 présentent les exécutions de la version trois en faisant varier le nombre de threads. Ce type de graphique est intéressant pour analyser le faible impact du déséquilibre de charge En effet, l'ajout de threads supplémentaires ne génère pas de sauts au niveau des diagrammes. De plus, la forme arrondie dessinée par les diagrammes montre bien que le stencil 7-point est particulièrement sensible à la limite de la performance mémoire. En revanche, on peut observer que le stencil 27-point est beaucoup moins sensible aux aspects mémoire.

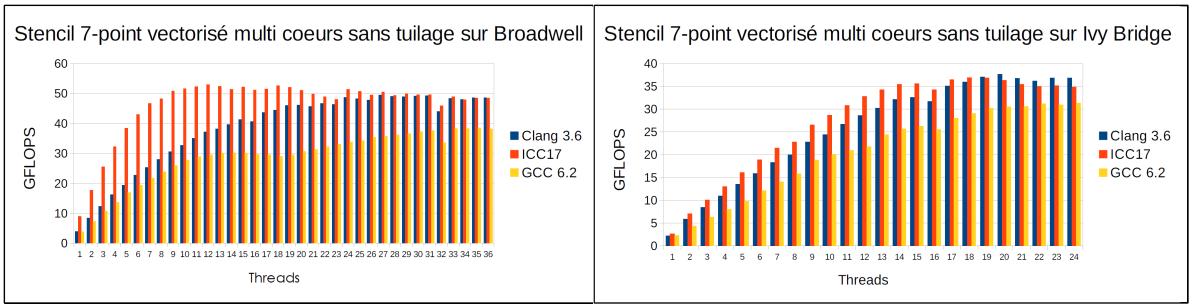

FIGURE 3.12 – Performances de la version multi-cœurs sans tuilage du stencil 7-point.

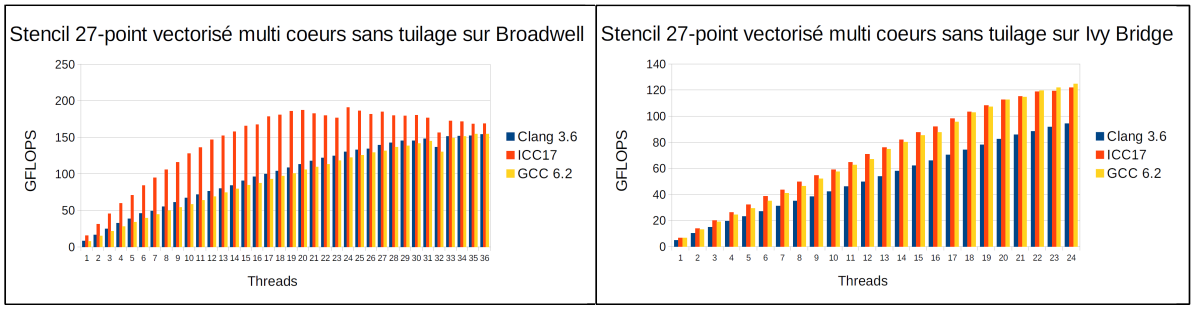

FIGURE 3.13 – Performances de la version multi-cœurs sans tuilage du stencil 27-point.

3.6 Optimisations mémoire par tuilage multi dimensionnel

3.6.1 Intuition et apport théorique du tuilage.

Les architectures impliquées dans cette thèse possèdent trois niveaux de mémoire cache dont les vitesses décroissent à mesure que l'on s'éloigne des cœurs vers la mémoire centrale. Accéder à une donnée depuis la mémoire centrale est plus long que de la réutiliser depuis les mémoires cache 2.2.1. Le verrou est ici lié à la faible intensité arithmétique du calcul stencil. En effet, nos plateformes cibles sont capables de réaliser plus de calcul par seconde que de transfert d'octet depuis la mémoire centrale. Or, l'intensité arithmétique du calcul stencil est environ de 2 opérations pour le chargement d'un flottant simple précision de 4 octets soit un AI de 0,5 (on consomme 2 octets par opérations arithmétiques) Par exemple, en considérant la plateforme bi-processeurs Broadwell 36 cœurs (double port FMA AVX à 2,3Ghz), on atteint 2,6 TFLOPS pour 128 GB/s de bande passante mémoire. De fait, la plateforme est environ 20 fois plus rapide à calculer qu'à transférer 1 octet avec la mémoire centrale. Par conséquent, il faudrait que le calcul réalise 20 opérations par octet transféré pour équilibrer la charge de calcul avec la charge de transfert mémoire. On comprend ici que la bande passante vers la mémoire centrale est un facteur limitant pour ces noyaux stencils. Cependant, la bonne réutilisation des données déjà chargées en mémoire cache peut réduire cette limitation.

En se plaçant dans le cadre de schéma numérique explicite, seuls les pas de temps précédents sont nécessaires. Une approche classique d'optimisation consiste donc à imbriquer les dimensions spatiales et temporelles.

Dans ce cas, les dépendances de données constituent un verrou important. La figure 3.14 montre que la forme du stencil impose le chargement d'un voisinage multi dimensionnel dans plusieurs directions spatiales et temporelles. En effet, le parcours par tuile permet d'améliorer la réutilisation des données chargées dans d'autres directions que celle imposée par le simple parcours des boucles imbriquées précédemment présentées.
De plus, ce tuilage s'applique aussi bien au domaine espace ou temps. En effet, on rappelle que le calcul se réalise à partir de points voisins sur des dimensions spatiales et temporelles.

Notre première approche consiste à parcourir le domaine de calcul par tuiles spatiales de taille ajustée à la capacité de la mémoire cache et à la forme du stencil. Par conséquent, le défi consiste à déterminer la taille $(tx; ty; tz)$ de la tuile qui minimise le nombre de

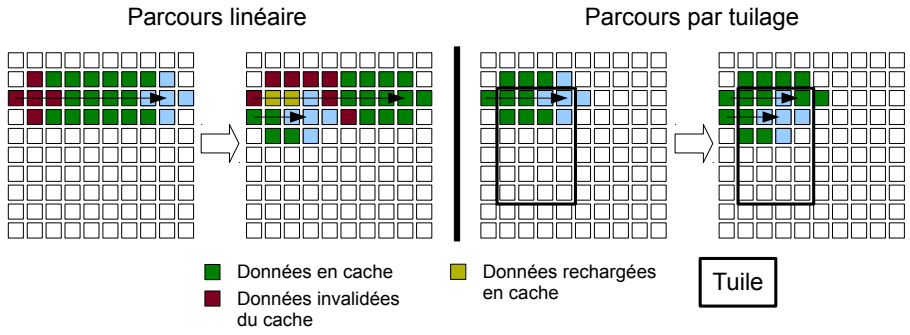

FIGURE 3.14 – Illustration comparative d'un parcours stencil linéaire avec un parcours par tuile d'un domaine de calcul.

défauts de cache et fournit la meilleure performance. Des tailles de tuiles optimisées ont été trouvées expérimentalement. La taille de la tuile optimale est plus grande en tx. De plus, tx est multiple de 64 octets. En effet, les données mémoires sont contigües suivant l'axe de cette composante tx et se chargent par ligne de cache. Enfin ceci concorde bien avec la littérature [17, 48].

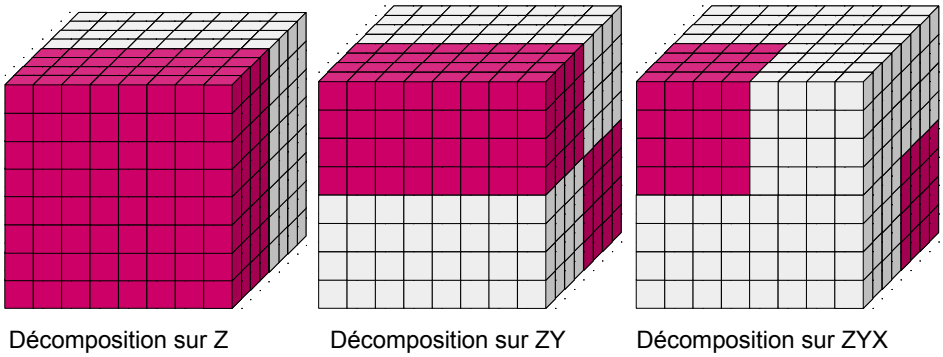

FIGURE 3.15 – Exemple de décomposition en tuile taillée suivant des compositions dimensionnelles croissantes.

Cette stratégie d'attaque du calcul se limite à un pas de temps suivant des composantes spatiales.

3.6.2 Évaluation expérimentale de l'apport du tuilage spatial

Le tuilage est une stratégie visant à améliorer la réutilisation des données en cache. Pour en démontrer l'intérêt, nous allons faire appel à une nouvelle implémentation.

— Version-4 Vectorisation manuelle avec tuilage spatial : cette version ajoute le tuilage spatial à la version précédente. Nous employons la stratégie standard (tuile allongée dans le sens mémoire) associée à une phase de recherche de la taille optimale des tuiles [17, 51].

Les résultats présentés dans les graphiques 3.16 et 3.17 montrent que l'impact du tuilage dépend du stencil. Par exemple, nous observons des gains limités pour le stencil 7-point (10% en moyenne sur Broadwell et 10% en moyenne sur Ivy Bridge) alors que toute

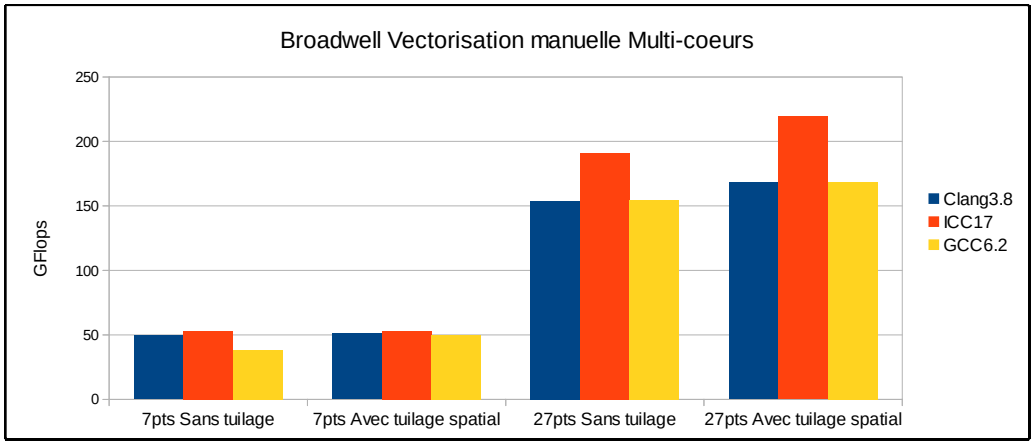

FIGURE 3.16 – Performances de l'expérimentation des versions avec et sans tuilage sur la plateforme Broadwell

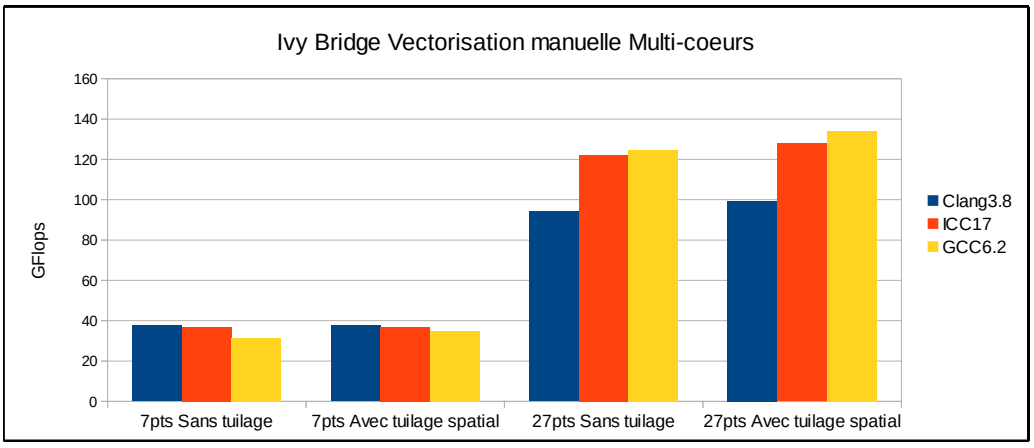

FIGURE 3.17 – Performances de l'expérimentation des versions avec et sans tuilage sur la plateforme Ivy Bridge

proportion gardée le stencil 27-point bénéficie plus de cette stratégie (plus de 14% sur Broadwell et plus de 12% sur Ivy Bridge). En effet, plus un stencil réalise de calculs avec des points et plus ces points sont réutilisables. Ce taux de réutilisation est indiqué par le facteur RI Le stencil 27-point possède un RI de 6,75 alors que le stencil 7-point est de 1,75. Voilà pourquoi, le stencil 27-point est plus sensible à l'optimisation par tuilage spatial.

Le tuilage spatial permet de se rapprocher de ce que le RI promet en termes de réutilisation des données. Ainsi, on peut observer que les résultats concordent assez bien avec ce facteur. Le tableau 3.6 présente les performances maximales obtenues en cumulant toutes les optimisations avec des ratios de performances dus 27-point sur le stencil 7-point. Lorsque l'on calcule le ratio entre les RI des 2 stencils, on obtient $27-point/7-point = 6,75/1,75 = 3,86$. Or, on remarque que les ratios de performances en sont assez proches. Ce qui montre que le facteur RI s'applique assez bien dans notre cas.

	Stencil	27-point	7-point	27-point/ 7-point
	RI	6,75	1,75	3,86
Compilateur	Plateforme	27-point	7-point	27-point/7-point
Clang 3.8	Broadwell	168,11	45,06	3,73
	Ivy Bridge	99,39	37,66	2,64
ICC17	Broadwell	190,53	49,38	3,86
	Ivy Bridge	127,82	36,91	3,46
GCC 6.2	Broadwell	168,89	44,05	3,83
	Ivy Bridge	134,19	34,90	3,84

TABLE 3.6 – Facteur des performances du stencil 27-point sur le stencil 7-point issues de l'expérimentation de la Version-4 vectorisée à la main en multi-cœurs tuilée (avec une tuile de référence issue de la littérature).

La formule de calcul du RI ne prend pas en compte la disposition des points du stencil seulement leur nombre. Un stencil à 3 dimensions peut faire 27 points et être un cube incomplet d'arrêté 3, comme il peut être une grande croix de diamètre 5. En théorie le RI part du constat, mais en pratique la forme du stencil au-delà du simple nombre de points peut avoir un impact sur les capacités réelles à réutiliser les données. Cependant, cela ne veut pas dire que le RI est inadapté puisqu'il s'agit d'un maximum selon une réutilisation spatiale. Des astuces doivent être mises en oeuvre pour s'en rapprocher et compenser le handicap morphologique de certains stencils.

3.6.3 Stratégies du tuilage temporel classique

Nous avons vu l'intérêt de réutiliser les données du cache à cause de la faible intensité arithmétique du calcul stencil. De fait, nous avons implémenté un tuilage de l'espace pour améliorer le taux de réutilisation des données. Ainsi, plus le stencil requière de points voisins et plus les données peuvent être réutilisées. Dans le cas de petits stencils tels que le stencil 7-point, la réutilisation potentielle reste limité. Cependant, un calcul stencil n'est pas uniquement spatial. Le fait qu'il soit appliqué plusieurs fois successivement sur les mêmes points recalculés donne au calcul une dimension temporelle. Ce qui nous intéresse ici est le fait que le stencil s'applique plusieurs fois sur les mêmes points. En effet, lorsque des points sont recalculés, ces données sont en mémoire cache. Elles seront de nouveau nécessaires lors de l'itération suivante. Or, si on attend la prochaine itération pour tout le domaine, les lignes de caches qui nous intéressent seront invalidées sans réutilisation possible.

Prenons le cas simple où une grille doit être recalculée suivant deux applications successives d'un stencil. La grille est alors parcourue par tuilage tenant en cache mémoire. Ainsi, figure 3.18, en appliquant le stencil une seconde fois dans cette même tuile, on réutilisera toutes les données depuis la mémoire cache. Cependant, le nombre d'itérations appliquées à chaque cellule devient plus complexe à suivre. En effet, le parcours linéaire ou par tuilage spatial de l'application d'un stencil permet d'itérer les cellules d'une grille vers une grille d'itération suivante. En cas de tuilage temporel, on itère plusieurs fois dans

la même tuile. Ainsi, on alterne à chaque itération les 2 instances de grille de données. Par conséquent, une instance de grille de données peut avoir des cellules ayant plusieurs itérations de différences.

Dans ce procédé, il y a trois éléments clés.

— Premièrement, il est possible de faire du tuilage temporel avec uniquement deux instances de la grille.
— Ensuite, il s'agit d'être capable d'itérer sur n'importe qu'elle cellule à tout moment. Cet élément clé est assez intuitif au sens où si une cellule ne peut plus être itérée, on n'obtiendra jamais la grille finale. Ceci implique que parmi les deux grilles, les cellules voisines à la bonne itération soient toujours disponibles Il est d'ailleurs préférable que chaque cellule devant itérer ait tous leurs voisins sur la même instance de la grille.
— Enfin, il n'est pas nécessaire de recopier les valeurs d'une grille à l'autre. Ainsi, seules les valeurs calculées sont inscrites d'une grille à l'autre. On remarque d'ailleurs dans l'exemple figure 3.19 que la grille d'instance A ne contient que les itérations paires. Or, cela est aussi valable pour nos précédentes approches (parcours linéaire et tuilage spatial).

Regardons maintenant comment cela fonctionne sur tout un domaine. Dans la figure 3.19, on part d'une grille 2D de cellule à l'itération 0. Le stencil permet de calculer la première tuile de la grille A dont les résultats sont inscrits dans l'instance B. ensuite en 2, on itère temporellement sur cette même tuile pour calculer des cellules à la 2e itération depuis la grille B vers la grille A. Nous en profitons pour remarquer que les cellules au bord de la première tuile ne peuvent passer à la 2e itération. Ceci est dû aux dépendances spatiales et temporelles induites par le stencil 5-point. Ainsi, on réitère dans une tuile réduite de 1 cellule à ces bords. À l'étape 3, nous nous attaquons à la 2e tuile afin de la ramener au même niveau que la première tuile via l'étape 4. Lorsque toutes les tuiles arrivent au même niveau d'itération, il est nécessaire de calculer les nouvelles itérations des cellules bordures que nous avions laissées de côté. Ainsi, tout le domaine atteindra le 2e niveau d'itération. Comme indiqué dans la légende de la figure 3.19, il est possible d'aller plus loin dans l'optimisation par tuilage temporel. L'objectif est ici d'introduire les grands principes du tuilage temporel. Pour plus de détail, il est possible de consulter les travaux [61, 25, 7].

L'exemple de tuilage temporel présenté dans la figure 3.19 est perfectible. D'une part, il faut déterminer la tuile initiale optimale et d'autre part il reste encore beaucoup de données qui ne peuvent bénéficier de la réutilisation temporelle. Afin d'améliorer ces points, l'algorithme de cache-oblivious [25] consiste en un découpage récursif dans l'espace et le temps de la simulation. L'objet de ce découpage est d'identifier des tâches de calcul spatio-temporel qui soit à la fois indépendant des autres et dont les données soient réutilisées depuis les plus petits niveaux de cache possible. Or, la découpe de ces tâches doit répondre à des contraintes pour assurer de couvrir tout le domaine sans oublier des zones à calculer. Ce qui rend délicat d'ajouter d'autres contraintes plus techniques telles que la taille fixe des registres vectorielles. De plus, Pochoir fait appel à l'environnement multithreads Cilk dont l'une des particularités est l'équilibrage de charges. En effet, ce découpage implique de réaliser des tâches irrégulières dont la charge de travail peut grandement différer.

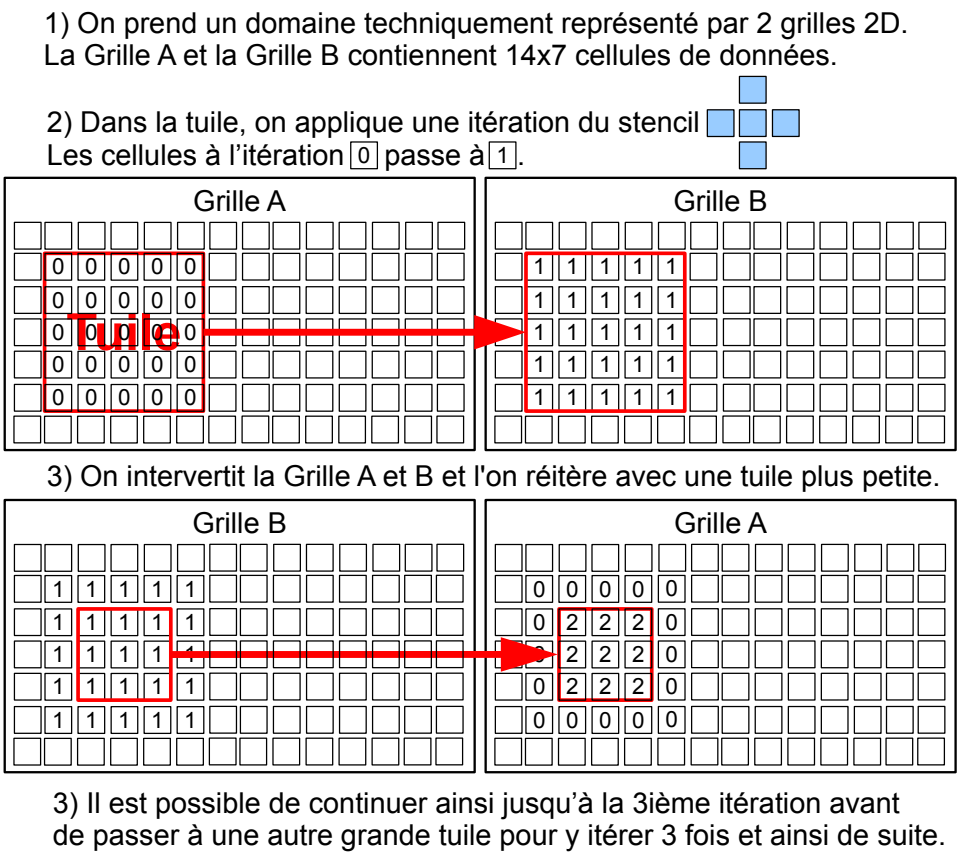

FIGURE 3.18 – Illustration par numéro d'itérations du tuilage temporel du stencil 5-point pour une seule tuile.

Ainsi, cette approche génère de l'irrégularité dans le parcours d'une grille initialement régulière. Par conséquent, il s'avère plus difficile d'exploiter des ressources vectorielles qui s'emploient mieux avec des données et des calculs réguliers. Ce constat nous amène à envisager une autre approche temporelle pour remédier à la faible réutilisabilité spatiale des données en cache lors d'un calcul du stencil 7-point. Nous allons maintenant passer à une façon inhabituelle de mettre en oeuvre l'optimisation par tuilage temporel.

3.6.4 Stratégie du tuilage temporel par composition stencil

Dans ces travaux de thèse, une autre approche du tuilage temporel par composition a été conçue et expérimentée. En effet, il est possible de déterminer le stencil correspondant à la composition de deux stencils. Appliquer ce stencil composé à un domaine revient à appliquer les 2 stencils qui le composent. On fait ici l'hypothèse qu'il sera plus performant d'appliquer cette composition au lieu d'appliquer successivement 2 fois le stencil 7-point. Afin d'illustrer cette approche, la figure 3.20 représente le stencil résultant de cette composition. On remarque de fait que le stencil résultant est un stencil 25-point. Or, nous évoquions précédemment le fait que la réutilisabilité des données chargées en cache dépend du nombre de voisins nécessaires au calcul du stencil. Autrement dit, appliquer le stencil 25-point au lieu de 2 fois le stencil 7-point permet une utilisation plus efficace des caches mémoire. En effet, même en l'absence de tuilage spatial, une attaque linéaire

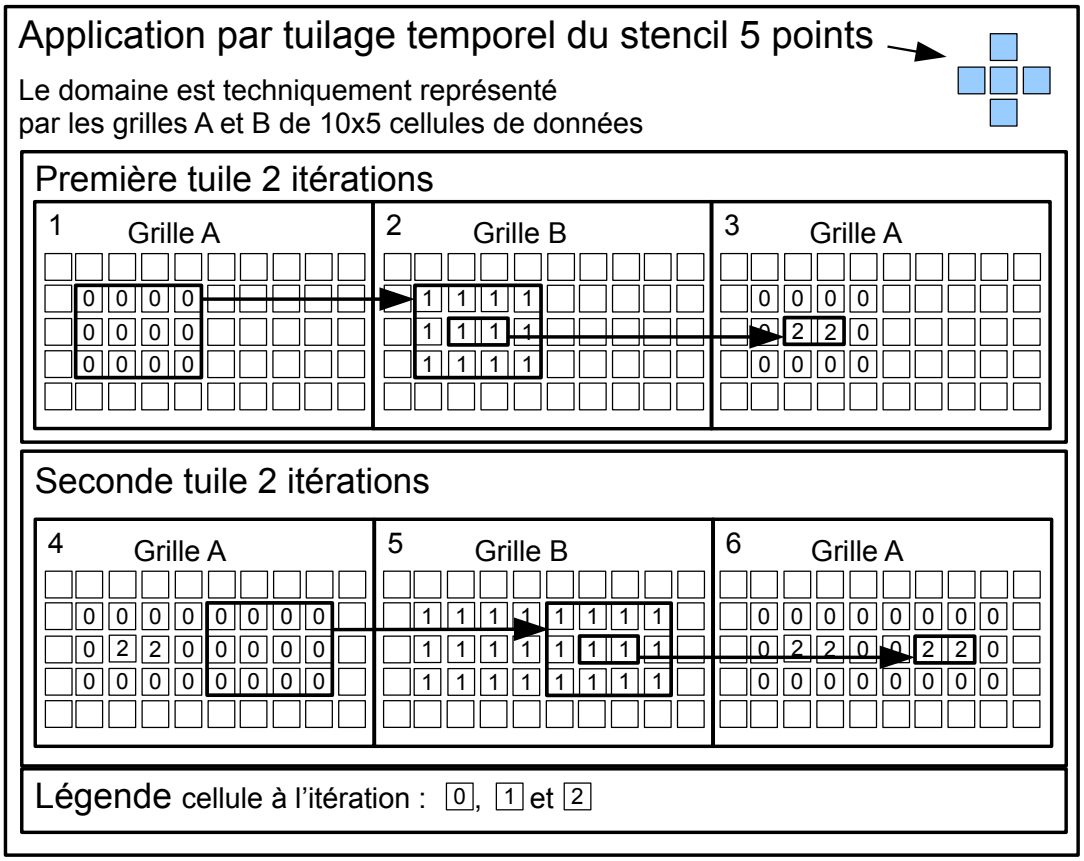

FIGURE 3.19 – Illustration par numéro d'itérations du tuilage temporel du stencil 5-point avec plusieurs tuiles. On réalise la première itération sur neuf cellules. La seconde itération ne se réalise que sur 2 cellules. À la fin, toutes les cellules de la grille B sont à l'itération 1. La grille A est parsemée régulièrement de cellules à l'itération 2. Ainsi, il reste à passer à l'itération de l'ensemble des cellules de la grille A. Il ne s'agit là que d'une première approche qui peut être améliorée. En effet, les secondes itérations peuvent être calculées au fur et à mesure et non à la fin du parcours par tuilage temporel.

ligne par ligne du calcul garantit de réutiliser 17 (5 au centre et 4 fois 4 autour) flottants sur 25. On atteint sans tuilage un taux de 68% contre 3/4=43% avec le stencil 7-point. Le tuilage spatial permet de combler un peu plus l'écart de réutilisation que le stencil permet. Nous rappelons que le facteur RI du stencil 7-point est de 1,75 alors que le RI du stencil composé 25-point est de 6,25. Or, la description standard du stencil souligne l'impact de l'Intensité de réutilisation (RI) sur les performances. En effet, l'intensité de réutilisation, ce nouveau stencil est proche du stencil 27-point (6,75). Dans la littérature, le 27-point est connu pour être plus performant en comparaison du stencil 7-point. Ainsi, cette composition est susceptible d'améliorer les performances du stencil 7-point.

Dans cette partie, nous introduisons une stratégie pour construire un équivalent du stencil 7-point, mais avec une densité plus élevée. Il est important de rappeler que nous évaluons les stencils réalisant uniquement des additions. En effet, les points du stencil 7-point ont un poids de 1. Ainsi, il n'y a pas de multiplication. Cependant, la composition de deux fois ce stencil implique des poids différents de 1. On a donc décidé d'ignorer ces derniers et de ne réaliser que les additions. En effet, l'ajout des multiplications augmente-

rait artificiellement les performances de calculs, mais le temps d'exécution ne diminuerait pas pour autant. Il est donc plus juste de comparer cette nouvelle formulation à armes égales. En pratique, le stencil 7-point impliquerait des multiplications ce qui ne changerait donc rien à le composer.

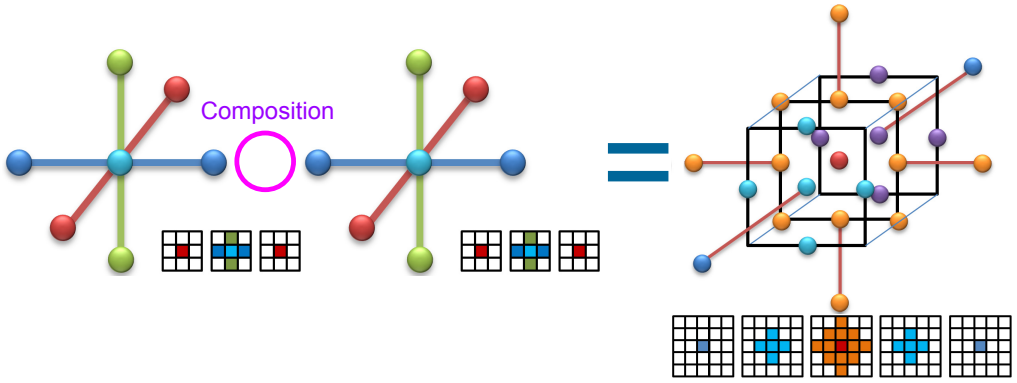

FIGURE 3.20 – Le stencil 25-point (à gauche) correspondant à la composition de deux noyaux 7-point (à droite).

La composition du stencil est définie comme suit. Considérons deux stencils S_A et S_B qui doivent être appliqué consécutivement, alors il est possible de construire un stencil S_C qui est la composition de S_A et S_B. Si nous considérons que S_A et S_B sont de dimension N où la taille de chaque dimension est respectivement $S_A.r_i * 2 + 1$ et $S_B.r_i * 2 + 1$ pour tout $i \in]1, N[$, alors le stencil composé S_C est aussi de dimension N où la taille de chaque dimension est $(S_A.r_i + S_B.r_i) * 2 + 1)$. Il nous est rendu possible d'estimer un ratio de la quantité de calculs entre l'application successive de S_A et S_B par rapport à l'application de S_C. Ce ratio peut être approché par la formule suivante 3.1 :

$$\frac{\prod_{i=1}^{N}(2(S_A.ri + S_B.ri) + 1) - 1}{\prod_{i=1}^{N}(2S_A.ri + 1) + \prod_{i=1}^{N}(2S_B.ri + 1) - 2} \tag{3.1}$$

Si nous appliquons l'équation 3.1 sur le stencil 27-point, sa demi-dimension vaut 1 alors le rapport d'extra calcul est de 2,38. Cela signifie que l'application de la composition par 2 fois du stencil 27-point implique environ 2,38 fois plus d'opérations arithmétiques que deux applications successives du stencil 27-point. Cette formule a le mérite d'approcher ce facteur suivant les demi-dimensions du stencil or, ces paramètres sont assez simples. Cependant, si l'on applique cette même formule sur le stencil 7-point, ce ratio approché sera le même, car la formule ne tient pas compte de la densité de points. Il est toute foi possible de déterminer la valeur exacte de ce ratio d'opérations par la formule 3.2 :

$$\frac{(S_C.nbPoint - 1)Additions}{(S_A.nbPoint + S_B.nbPoint - 2)Additions} \tag{3.2}$$

À la différence de la formule précédente, la composition doit être réalisée pour connaitre le nombre exact de points du stencil résultant. Dans notre cas, pour le stencil 7-point, le

ratio extra computing exacte est de 2 tel qu'on peut le voir en 3.3 :

$$\frac{(25pts-1)Additions}{2time\_steps*(7pts-1)Additions} \tag{3.3}$$

Ceci signifie que l'application du stencil 25-point comptabilise 2 fois plus d'additions que l'application successive de 2 fois le stencil 7-point. Notre plus ancienne plateforme a une puissance maximale de calcul de 2 TFLOPS pour une puissance de transfert maximale de 128 GB/s avec la mémoire centrale. Ainsi, le fait de doubler le nombre d'opérations reste limité par la bande passante de la mémoire centrale. De fait, notre hypothèse est ici bien fondée quant à la possibilité d'amélioration les performances par la composition stencil. Il est temps de vérifier expérimentalement cette hypothèse. La sous-section suivante évalue le gain obtenu avec cette approche.

3.6.5 Évaluation de la composition de stencil

Le tuilage temporel permet d'aller au-delà du RI dans la réutilisation des données en cache. En effet, le RI se contente de considérer une seule itération et donc un tuilage uniquement spatial. En outre, le tuilage temporel tient compte du caractère itératif du calcul stencil pour réutiliser les données déjà sollicitées.

Afin d'en évaluer les gains de performances, une autre implémentation est évaluée.

— Version-5 Composition du stencil 7-point avec lui même : cette version applique le stencil à 25 points résultant de la composition du stencil 7-point avec lui même et n'effectue par conséquent que la moitié des itérations à savoir 50 seulement.

Ainsi, la dernière étape de notre stratégie d'optimisation correspond à une nouvelle formulation du noyau numérique tenant compte de la nécessité d'itérer. Le stencil 7-point est un bon candidat pour évaluer cette approche puisque son RI est le plus faible des RI de nos stencils.

En termes de performance de calcul, le gain moyen entre les stencils 7-point et 25-point est de 3,4. Ainsi, l'intensité de réutilisation du stencil 25-point (nouvelle formulation du stencil 7-point) nous permet d'atteindre de meilleures performances, quel que soit le compilateur ou la plateforme. De plus, nous maintenons la cohérence avec l'analyse théorique. La figure 3.22 montre la performance maximale mesurée avec le 25-point et le stencil 27-point. Sur cette figure, on constate que les mesures sont très similaires, car les deux stencils présentent un facteur RI très proches.

L'interprétation des indicateurs de performances GFLOPS du stencil 25-point peut nous induire en erreur. En effet, ce qui importe vraiment c'est le temps nécessaire pour effectuer le calcul stencil. Afin de bien comprendre, les temps de calcul des stencils 7-point et 25-point sont indiqués dans le tableau 3.7. On constate que le gain de temps entre les stencils 7-point et 25-point est seulement de 1,6. Or, le gain en performance annonce 3,2. En effet, une double application successive du stencil 7-point factorise les calculs en mémorisant des résultats intermédiaires impliquant plus de mouvements mémoires sur ces 2 itérations. Alors que le stencil 25-point implique de refaire ces mêmes calculs intermédiaires ce qui représente 2 fois plus d'additions.

Comme expliqué dans la sous-section 3.6.4, le stencil 25-point est le résultat d'une composition du stencil 7-point avec lui-même, mais il nécessite 4 fois plus d'additions par itération (de 6 ajouts pour le 7-point à 24 pour le 27-point). Cependant, une itération du stencil 25-point est équivalente à 2 itérations du stencil 7-point. Ainsi, pour obtenir les mêmes résultats numériques, nous avons besoin de moitié moins d'itérations. Par conséquent, pour l'ensemble du calcul, ce nouveau stencil 25-point implique de réaliser 2 fois plus d'additions que nécessite la double application successive du stencil 7-point.

En termes de performance équivalente, la vitesse mesurée sure Broadwell est respectivement divisée par 2. De fait, les gains réels obtenus sur Broadwell par rapport à l'application de double itération du stencil 7-point sont d'environs 1,6.

Plateforme	Stencil	itérations	Clang	ICC	GCC
Ivy Bridge	7-point	100	2275	2304	2312
	25-point	50	1588	1259	1352
Broadwell	7-point	100	1819	1762	1863
	25-point	50	1020	891	1096

TABLE 3.7 – Durée en millisecondes du calcul d'application des stencils 7-point (RI=1.75) et 25-point (RI=6.25) composition de 2 fois le stencil 7-point.

Cette approche est originale dans la mesure où elle reformule le stencil.

3.7 Comparaison avec le DSL Pochoir

3.7.1 Principes de fonctionnement du DSL Pochoir

Pochoir est un compilateur de référence dans le domaine de l'optimisation automatique du calcul stencil. Son langage de programmation est un DSL embarqué dans le langage C++. Il suffit donc de compiler via Pochoir un code C++ décrivant le stencil selon le langage du DSL. Pochoir se charge de nous gérer le programme binaire. Afin de réaliser ce fichier binaire, le compilateur Pochoir va générer un nouveau code source à partir de celui d'entrée. Ainsi, ce nouveau code source est un code C++ dont les instructions spécifiques à Pochoir ont été substituées par un algorithme de calcul « cache-oblivious » [25, 61]. Le principe de cet algorithme est qu'il subdivise récursivement l'espace et le temps du calcul ainsi il travaille sur plusieurs itérations avec des tailles de données susceptibles de mieux tenir en mémoire cache. À noter que cet algorithme a fait l'objet d'une démonstration prouvant que le taux de défaut de cache est fortement réduit en attaquant le calcul de cette façon. Cependant, cet algorithme n'est pas trivial et le code généré est plus complexe qu'une imbrication de boucles. Nous allons maintenant regarder ce qu'il donne en pratique avec l'implémentation faite sous Pochoir.

3.7.2 Expérimentation du DSL Pochoir

Le développement de Pochoir privilégie le compilateur d'Intel, ainsi, nous présentons les résultats de nos programmes compilés avec le compilateur ICC17.

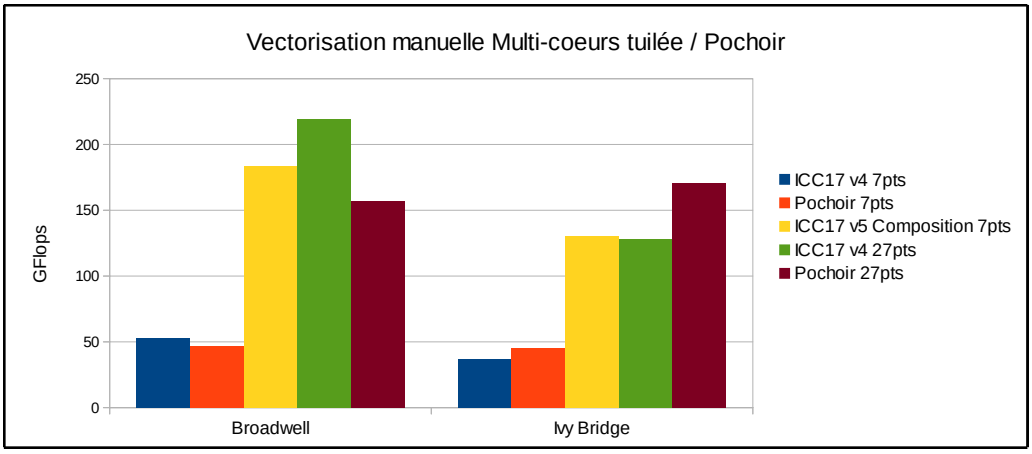

FIGURE 3.21 – Comparaison des versions optimisées 4 et 5 avec le DSL de référence Pochoir pour les stencils 7-point et 27-point sur les plateformes Broadwell et Ivy Bridge

Nous pouvons constater sur le graphique 3.21 que Pochoir parvient difficilement à dépasser nos implémentations cumulant des optimisations. De plus, on remarque qu'il est sensible à la plateforme d'exécution. En effet, le stencil 27-point est plus performant sur Broadwell que sur Ivy Bridge. Alors que le stencil 7-point sur la plateforme Broadwell est aussi performant que sur la plateforme Ivy Bridge. Enfin, la Version-5 de la composition du stencil permet un tuilage temporel sur 2 itérations. Ainsi, les performances obtenues avec une reformulation du stencil 7-point dépassent celles atteintes avec le stencil 7-point compilé sous Pochoir.

À noter que la composition du stencil 7-point donne le stencil 25-point. Or, il peut lui aussi être compilé par Pochoir. Ainsi, ces indicateurs de performance via Pochoir s'élèvent à 102 GFLOPS sur la plateforme Broadwell et à 99 GFLOPS sur la plateforme Ivy Bridge. De plus, le tableau 3.8 présente les durées des calculs entre nos optimisations compilées via ICC et celles assurées via Pochoir. On remarque que les optimisations de Pochoir sont moins efficaces pour le stencil 25-point que pour le stencil 7-point. La raison est simple, le stencil 25-point est déjà une optimisation temporelle du stencil 7-point. Les dépendances liées au voisinage étendu du stencil 25-point imposent à pochoir de réduire davantage sa zone de calcul à mesure qu'il itère dans tuile temporelle. De plus, le découpage du domaine mis en oeuvre par Pochoir rend la vectorisation plus délicate. En effet, il délègue la vectorisation au compilateur ICC17 qui ne parvient pas à la mettre en oeuvre.

Plateforme	Stencil	itérations	ICC	Pochoir
Ivy Bridge	7-point	100	2304	2047
	25-point	50	1259	1653
Broadwell	7-point	100	1762	1978
	25-point	50	891	1607

TABLE 3.8 – Durée en millisecondes du calcul d'application des stencils 7-point (RI=1.75) et 25-point (RI=6.25) composition de 2 fois le stencil 7-point.

En résumé, il est possible d'obtenir de meilleures performances que le DSL de référence

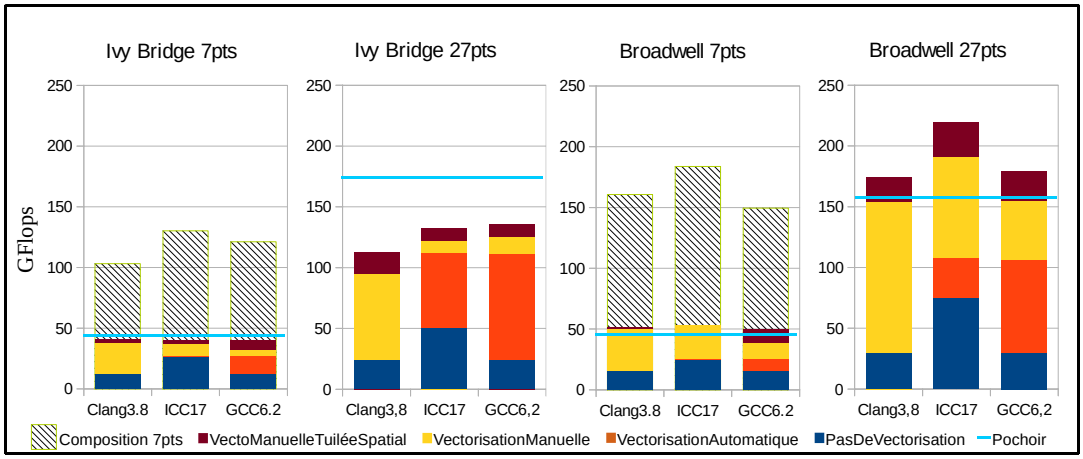

FIGURE 3.22 – Graphiques de synthèse des expérimentations stencil 7-point et 27-point calculés en multi-threads.

Pochoir en considérant une implémentation simplifiée.

3.8 Conclusion

Ce chapitre souligne l'efficacité des approches proposées. Sur la plateforme Broadwell, nos implémentations dépassent les 50 GFLOPS pour le stencil à 7-point et 150 GFLOPS pour le stencil à 27-point. La figure 3.22 synthétise les performances maximales atteintes lors de nos expérimentations. Tout d'abord, la vectorisation explicite via les intrinscis domine (×9 en mono-cœur) celle proposée automatiquement par les compilateurs (×6 en mono-cœur). De plus, le parallélisme multi-threads permet de bénéficier de l'amélioration des performances offertes par l'architecture sous-jacente. En effet, la plateforme Broadwell atteint une performance maximale supérieure à la plateforme Ivy Bridge (roofline fig. 3.5). La situation inverse est observée avec Pochoir. La stratégie de tuilage apporte un gain de performance supplémentaire. Ce dernier est plus significatif avec le stencil 27-point du fait de son RI plus élevée. Enfin, l'approche de tuilage spatial par composition du stencil 7-point permet de surpasser l'implémentation de référence réalisée avec Pochoir.

Comme le montrent les indicateurs de performances, l'optimisation des noyaux stencils peut s'avérer complexe, car les techniques efficaces dépendent de la morphologie du stencil, de l'architecture matérielle sous-jacente et des compilateurs. Prises indépendamment, ces techniques d'optimisation sont simples à mettre en œuvre, cependant elles doivent être combinées afin d'atteindre des performances plus élevées.

En ce qui concerne la bibliothèque Pochoir, elle repose sur une décomposition complexe de l'espace et du temps. Ainsi, pour le stencil 7-point sur la plateforme Ivy Bridge, elle surpasse notre implémentation manuellement vectorisée et tuilée spatiale. Ce qui prouve bien qu'elle peut s'avérer efficace bien qu'elle ne vectorise pas. Néanmoins, notre technique de composition stencil est capable d'atteindre un niveau de performance similaire sur cette plateforme et surpasse l'implémentation de référence sur Broadwell.

Les gains obtenus sont variables selon l'architecture, le stencil et les compilateurs. De plus, pour chaque stencil, nous n'atteignons pas la valeur théorique issue de la Roofline suivant le RI. De fait, on peut raisonnablement supposer que de futurs travaux puissent conduire à de meilleures performances. En effet, au-delà de la limite déduite du RI, le tuilage temporel permet de dépasser celle-ci. Toutefois, l'algorithme et son implémentation proposent de sérieux challenges en termes de portabilité. Une piste serait de séparer l'interface de l'implémentation. Un langage spécifique au domaine (DSL) serait fourni au développeur qui se concentre sur la description des modèles. L'implémentation optimisée serait obtenue en utilisant soit un compilateur spécifique, comme pour Pochoir, soit des techniques de méta programmation.
Enfin, malgré notre volonté à concevoir des approches simples et lisibles, la mise en œuvre de nos contributions dans certaines applications réelles peut s'avérer difficilement exploitable.

Le calcul stencil par tuilage temporel tel qu'il est mis en pratique dans ces travaux correspond à une résolution directement induite d'une EDP sur plusieurs pas de temps. Cependant, les applications industrielles réalisent parfois des étapes de traitement sur le domaine entre chaque application du stencil. Tant que ses étapes se résument à appliquer des conditions limites aux bords, le tuilage temporel s'en sort assez bien. En effet, que ce soit par composition ou par découpage temporel, la zone du calculable se réduit, car elle tient compte du niveau d'itération de chaque point. En outre, dans le cas où d'autres traitements globaux sont à réaliser sur le domaine, l'enchaînement de plusieurs itérations nécessaire au tuilage temporel n'est plus possible en l'état. Pour résoudre ce problème, il devient nécessaire d'intégrer dans la mesure du possible les traitements inter itérations dans le calcul par tuilage temporel.

Chapitre 4

Optimisation de noyaux éléments finis spectraux

L'objectif de ce chapitre est d'analyser un code d'éléments finis spectraux et d'étudier différentes pistes d'optimisation. Notre étude cible l'application EFISPEC [20], une application représentative de la méthode des éléments finis spectraux développée au BRGM. La version standard de ce code est implémentée en Fortran et s'appuie sur la bibliothèque MPI afin d'exploiter la puissance des architectures distribuées. Cette étude sera menée sur le noyau de calcul des forces internes, car il représente un maximum de 90% du temps total de résolution.

4.1 L'application EFISPEC

L'application Efispec3D [1], pour Elements Finis Spectraux, est développée au BRGM et distribuée sous licence open-source. Elle permet d'effectuer des simulations de propagation d'ondes sismiques pour étudier les phénomènes de type tremblements de terre. Le milieu de propagation est modélisé par un maillage irrégulier dont le niveau de détail dépend du milieu considéré. L'application Efispec3D est écrite en Fortran et la parallélisation du calcul est basée sur l'utilisation de la bibliothèque MPI pour pouvoir exploiter les architectures distribuées. La distribution du maillage est effectuée à l'aide de la bibliothèque METIS.

4.1.1 Le modèle numérique d'Efispec3D

L'application Efispec3D s'appuie sur la méthode des éléments finis spectraux (SEM pour Spectral Element Method). Cette méthode est apparue il y a plus de 20 ans dans le domaine de la mécanique des fluides [50, 41, 24]. Le SEM est une formulation spécifique de la méthode des éléments finis pour laquelle les points interpolés et les points en quadrature d'un élément partagent le même emplacement. Ces points sont les points GLL, ce sont les racines $p+1$ de $(1-\xi^2)P'_p(\xi) = 0$, où P'_p désigne la dérivée du polynôme de Legendre du degré p et ξ coordonnée dans l'espace de référence unidimensionnelle $\Lambda = [-1, 1]$.
En d'autres termes, lorsque l'on considère une seule dimension sure $\Lambda = [-1, 1]$, à l'ordre

1. Application Efispec3D disponible à l'adresse http://efispec.free.fr

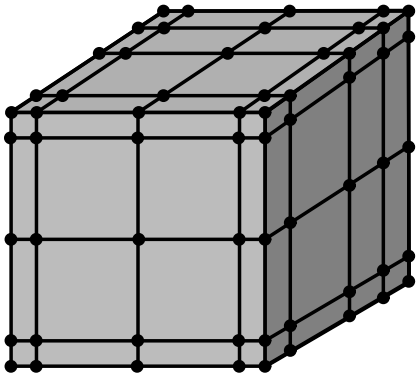

FIGURE 4.1 – Cube de référence composé de $(4+1)^3 = 125$ points GLL.

p, chaque élément fini se compose de $p+1$ racines que sont les points GLL. Nous verrons plus loin que cette particularité des éléments spectraux a un impact sur l'AI.
La généralisation à des dimensions supérieures se fait par la tensorisation de l'espace de référence unidimensionnel. Ceci implique que les $p+1$ points GLL racines sont portés à la puissance du nombre de dimensions. Ainsi, l'impacte sur l'AI en est démultiplié. En trois dimensions, l'espace de référence est le cube $\square = \Lambda \times \Lambda \times \Lambda$ (voir Fig. 4.1).
Le mappage du cube de référence à un élément hexaédrique Ω_e se fait par un difféomorphisme régulier $\mathcal{F}_e : \square \to \Omega_{\Omega_e}$. Dans une méthode par éléments finis, le domaine d'étude est discrétisé en subdivisant son volume Ω en éléments hexaédriques soudés et non chevauchants. Ω_e, $e = 1, \ldots, n_e$ tel que $\Omega = \cup_{e=1}^{n_e} \Omega_e$.
Les éléments Ω_e forment le maillage du domaine. D'une part, chaque élément Ω_e a une numérotation locale des points GLL allant de 1 à $p+1$ le long de chaque dimension de la tensorisation. Cette numérotation a une correspondance avec une numérotation globale des points physiques du maillage. Le mappage de la numérotation locale à la numérotation globale est la phase dite « d'assemblage » de tous les calculs d'éléments finis. Chaque point GLL d'un élément Ω_e est redirigé vers un nombre global unique correspondant à un point physique, $\forall \Omega_e$. Lorsque plusieurs éléments partagent une face, un bord ou un coin commun, la phase d'assemblage additionne la valeur GLL locale dans le système de numérotation global. Ainsi, un point physique global peut dépendre de différents points GLL locaux. Autrement dit, les éléments se composent de points GLL correspondant avec des points physiques du domaine Ω. Certes, les éléments finis ne se chevauchent pas, mais ils sont joints. Ainsi, aux jointures, on voit apparaitre des dépendances entre ces éléments via des correspondances mutuelles entre des points physiques et des points GLL appartenant à différents éléments. Ce qui implique une synchronisation des accès aux points physiques de jointure en cas de calcul en parallèle des éléments finis.

Le calcul de chaque élément fini peut se décomposer en 3 phases :

1. Formation de l'élément à partir des valeurs physiques associées aux points GLL,
2. Calcul des forces internes à l'élément
3. Assemblage par accumulation des forces aux points physiques.

Ainsi, nous allons retrouver ces phases dans les prochaines formulations.

Dans cet article, le problème d'intérêt est l'équation du mouvement dont la formulation faible est donnée par

$$\int_\Omega \rho \mathbf{w}^T \cdot \ddot{\mathbf{u}}\, d\Omega = \int_\Omega \boldsymbol{\nabla}\mathbf{w} : \boldsymbol{\tau}\, d\Omega - \int_\Omega \mathbf{w}^T \cdot \mathbf{f}\, d\Omega - \int_\Gamma \mathbf{w}^T \cdot \mathbf{T}\, d\Gamma$$

où Ω et Γ sont le volume et la superficie du domaine à l'étude, respectivement ; ρ est la densité du matériau ; $\mathbf{w}$ est le vecteur test ; $\ddot{\mathbf{u}}$ est la deuxième dérivée temporelle du déplacement $\mathbf{u}$; $\boldsymbol{\tau}$ est le tenseur de stress ; $\mathbf{f}$ est le vecteur force du volume et $\mathbf{T}$ est le vecteur traction agissant sur Γ. T correspond à la transposition, et les deux points à la contraction du produit tensoriel. Les paramètres locaux à chaque élément fini sont déterminés suivant ces grandeurs physiques dépendantes de la localité de l'élément dans son domaine Ω. Ces paramètres interviennent à la phase 2 dans le calcul des forces internes.

Notre étude se concentre sur les forces internes définies par les éléments suivants (voir [36])

$$\begin{aligned}
\int_{\Omega_e} \boldsymbol{\nabla}\mathbf{w} : \boldsymbol{\tau}\, d\Omega_e \approx \sum_{\alpha=1}^{\mathrm{p+1}} \sum_{\beta=1}^{\mathrm{p+1}} \sum_{\gamma=1}^{\mathrm{p+1}} \sum_{i=1}^{3} \mathrm{w}_i^{\alpha\beta\gamma} \times \Bigg[\\
\omega_\beta \omega_\gamma \sum_{\alpha'=1}^{\mathrm{p+1}} \left[\omega_{\alpha'} \mathcal{J}_e^{\alpha'\beta\gamma} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha'\beta\gamma} \partial_j \xi_{\alpha'} \right] \ell'_\alpha(\xi_{\alpha'}) \right] \\
+ \omega_\alpha \omega_\gamma \sum_{\beta'=1}^{\mathrm{p+1}} \left[\omega_{\beta'} \mathcal{J}_e^{\alpha\beta'\gamma} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha\beta'\gamma} \partial_j \eta_{\beta'} \right] \ell'_\beta(\eta_{\beta'}) \right] \\
+ \omega_\alpha \omega_\beta \sum_{\gamma'=1}^{\mathrm{p+1}} \left[\omega_{\gamma'} \mathcal{J}_e^{\alpha\beta\gamma'} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha\beta\gamma'} \partial_j \zeta_{\gamma'} \right] \ell'_\gamma(\zeta_{\gamma'}) \right] \Bigg]
\end{aligned}$$

avec $\boldsymbol{\tau}$ le tenseur de stress ($= \boldsymbol{c} : \nabla\mathbf{u}$) ; $\mathcal{J}_e^{\alpha'\beta\gamma}$ le jacobien d'un élément Ω_e aux points GLL $\alpha'\beta\gamma$; ω_λ poids d'intégration au point GLL λ ; ξ, η, ζ coordonnées locales le long des trois dimensions du cube de référence ; ℓ'_λ dérivée du polynôme de Lagrange au point GLL λ ; $\boldsymbol{c}$ est le tenseur d'élasticité. Nous pouvons remarquer de la formule précédente que des sommes s'imbriquent et parcourir l'ensemble des points GLL dont le nombre s'élève au cube de l'ordre d'approximation plus un. Ainsi, le calcul des forces internes de chaque élément fini passe par au moins 3 boucles imbriquées. Par conséquent, l'AI s'élève sensiblement en regard de l'ordre d'approximation.
De plus, $\nabla\mathbf{u}$ est le gradient du déplacement défini par

$$\begin{aligned}
\partial_i \mathrm{u}_j\left(\xi_\alpha \eta_\beta \zeta_\gamma\right) = & \left[\sum_{\sigma=1}^{\mathrm{p+1}} u_j^{\sigma\beta\gamma} \ell'_\sigma\left(\xi_\alpha\right) \partial_i \xi_{\alpha\beta\gamma} \right] \\
& + \left[\sum_{\sigma=1}^{\mathrm{p+1}} u_j^{\alpha\sigma\gamma} \ell'_\sigma\left(\eta_\alpha\right) \partial_i \eta_{\alpha\beta\gamma} \right] \\
& + \left[\sum_{\sigma=1}^{\mathrm{p+1}} u_j^{\alpha\beta\sigma} \ell'_\sigma\left(\zeta_\alpha\right) \partial_i \zeta_{\alpha\beta\gamma} \right]
\end{aligned}$$

Ce terme renferme à son tour des sommes liées à ce même ordre d'approximation polynomial.

Cette présentation de la formulation mathématique permet de faire le lien avec le code Fortran des forces internes disponible à l'annexe B. On comprend ainsi l'origine les différentes phases ainsi que celle des boucles imbriquées. De plus, on peut également voir apparaitre les opérations matricielles. Avant d'entrer dans le détail du code source, la sous-section suivante présente les limitations de l'implémentation actuelle du noyau Efispec3D.

4.1.2 Fortran MPI d'Efispec3D

L'application Efispec3D développée au BRGM repose sur une implémentation classique de noyau de résolution par assemblage d'élément fini.

L'implémentation distribuée d'Efispec3D repose sur une stratégie de partitionnement du maillage via METIS [34] et l'échange explicite des frontières entre sous-domaines voisins [35]. Ainsi, des communications non bloquantes permettent la superposition des temps de transfert avec les calculs. Par conséquent, nous observons une accélération presque idéale pour la mise en œuvre de la communication non bloquante alors que la performance de la version MPI bloquante se dégrade pour plusieurs dizaines de milliers de cœurs [10]. Dans ce dernier cas, le ratio entre la communication et le calcul n'est pas équilibré ce qui réduit les performances du parallélisme. Pour illustrer cela, une analyse par

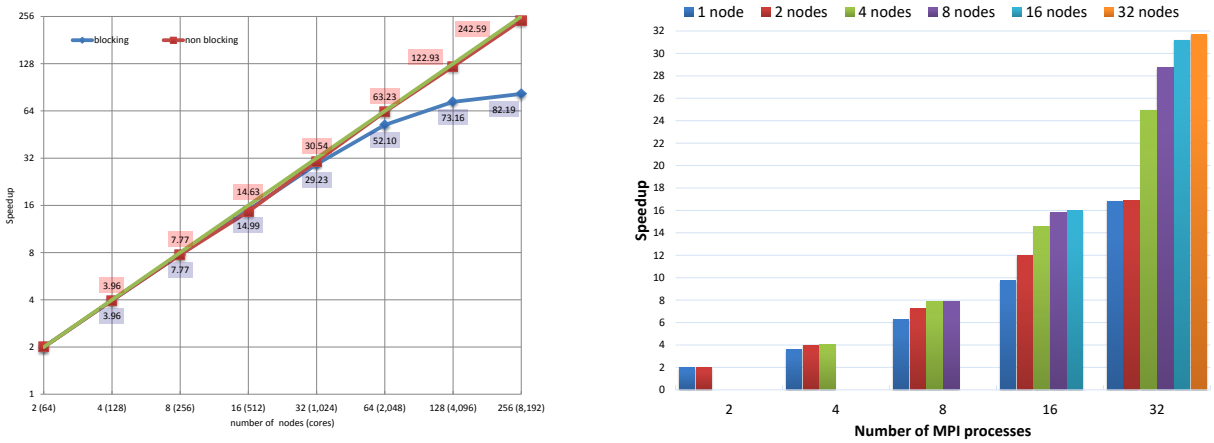

FIGURE 4.2 – Résultats d'un passage à l'échelle fort sur la plate-forme Cray Shaheen II jusqu'à 8 192 cœurs (graphique de gauche). L'accélération à l'aide de 32 processus MPI et de communications MPI non bloquantes sur différents nombres de nœuds de calcul (graphique de droite)

strong scaling est présentée sur la partie gauche de la figure 4.2. Typiquement, la même taille du problème est maintenue pendant que le nombre de cœurs de calcul augmente. Ces expériences ont été réalisées sur le système Shaheen II 2 Petascale. Ce supercalculateur Cray XC40 est situé au laboratoire KAUST (King Abdullah University of Science and Technology) Supercomputing en Arabie Saoudite et se compose de 6 174 nœuds de calcul physique. Chaque nœud de calcul est composé de deux processeurs Intel Xeon E5-2698v3

de 16 cœurs cadencés à 2,3 GHZ. Les nœuds sont connectés via le réseau à grande vitesse Cray Aries avec la topologie Dragonfly. Nous utilisons Cray MPICH Library (7.2.6) pour les communications. Shaheen II 2 était classée #18 sur la liste du Top500 du mois de juin 2017. Nous évaluons les versions MPI bloquantes et non bloquantes en utilisant comme base la performance obtenue sur deux nœuds. La figure 4.2 montre les résultats obtenus avec un maillage de 1,906,624 hexaèdres.

Le langage de programmation Fortran employé pour coder Efispec3D est assez répandu dans le domaine du calcul numérique [56, 2, 69]. Il s'associe avec MPI pour assurer une distribution des calculs. Des tentatives d'optimisations via l'usage de directives, paramètres de compilation ou de transformations du code Fortran ont été menées afin d'aider le compilateur. Ces dernières se sont révélées fortuites. Par ailleurs, l'utilisation d'optimisations plus poussées comme la vectorisation SIMD explicite à l'aide d'intrinsics nous oblige à employer le langage C++. Par conséquent, le noyau de calcul dont il est question dans ce chapitre a tout d'abord été traduit en C++.

Nous venons de voir que l'implémentation Fortran actuelle est encore lacunaire. La sous-section suivante présente les limites du code C++ de cette implémentation ainsi que la solution envisagée pour briser ces limitations.

4.1.3 Noyau de calcul

Dans cette sous-section, nous allons présenter le code traduit en C++ des forces internes du simulateur Efispec3D. Ce code C++ est issu de la traduction depuis le code Fortran. Des tests ont permis de s'assurer que ces 2 codes scalaires produisent des résultats identiques dans la limite des erreurs d'arrondis. Pour des raisons de clarté, le code est ici découpé en sous-fonctions et modifié de façon équivalente à la transcription C++. L'implémentation en Fortran est disponible à l'annexe B suivie de sa traduction en C++ C. L'ensemble des variables partagées par les fonctions de calcul sont regroupées dans la figure 4.3. Ces variables implémentent la structure de données en SoA et en AoS. Le tableau à 3 composantes « displacement » correspond à la valeur physique de displacement 3D pour chaque point GLL. De la même façon, le tableau « accélération » contient leurs données physiques d'accélération 3D. Ainsi, la première dimension du tableau est le numéro de l'élément qui donne accès à une grille 3D de $5 \times 5 \times 5$ paramètres de points GLL. Plus particulièrement, le tableau « elm_refGLL » contiens les indices de chaque point GLL. Ces indices servent à référencer les points physiques se trouvant dans les tableaux « displacement et accélération ».

Avant de détailler le programme, des fonctions courantes en algèbre linéaire sont regroupées dans la figure 4.4. Le code d'origine C++ déroule intégralement ces dernières de par sa traduction depuis le code Fortran. Cependant, notre exemple est factorisé à l'aide de ces fonctions afin de simplifier la lisibilité du code. Tout comme dans le code d'origine, ces fonctions ne traitent qu'un seul élément à la fois.

Nous arrivons au point d'entrée de notre noyau de calcul des forces internes. La fonction décrite à la figure 4.5 prend en paramètre un intervalle des numéros d'éléments à traiter. Ainsi, elle parcourt les numéros dans l'intervalle et applique le calcul pour chacun des éléments correspondants. On peut déjà remarquer que la boucle n'est pas parallélisée

```
//Point 3D des données physiques des GLLs
float displacement[nbPointsGLL][3];
float acceleration[nbPointsGLL}][3];

//èParamtres pour chaque point GLL de chaque élément
float pMatGLL[nbElm][5*5*5][3*3];
float pJacoGLL[nbElm][5*5*5];
float pRovpGLL[nbElm][5*5*5];
float pRovsGLL[nbElm][5*5*5];

//Indice de référence du numéro du point
//physique auquel le point GLL se rattache
int elm_refGLL[nbElm][5*5*5];

//Autres coefficients
float lagrgDrv[5][5],gllWeight[5];
```

FIGURE 4.3 – Tableaux globaux des données de la simulation.

```
//Calcul de l'indice d'un point GLL selon ses coordonnées klm
int crdId(float cg[3]){return (cg[2]*5+cg[1])*5+cg[0];}

//Fonction de multiplication matricielle r=mA*mB
void matriceMultipli(float *r,float *mA,float *mB);
//Fonction d'application de facteur c a la matrice r=c*m
void matriceMultipli(float *r,float c,float *m);
//Fonction d'application de facteur c au point 3D r=c*p
void point3DMultipli(float *r,float c,float *m);

//Calcul la matrice taux selon rovp, rovs et la matrice D
void calculerMatTau(float *matTau, float matD, float vp,float vs);
```

FIGURE 4.4 – Fonctions d'algèbre linéaire.

```
void calcul_force_interne_order4(int elm_deb, int elm_fin)
{
 for(int ie = elm_deb ; ie < elm_fin ; ++ie )
 {
  int cg[3]; //cg[2]=k cg[1]=l cg[0]=m
  float local[5*5*5][3*3];//matrice 3x3 locale par point GLL
  for(int cg[2] = 0 ; cg[2] < 5 ; ++cg[2])
   for(int cg[1] = 0 ; cg[1] < 5 ; ++cg[1])
    for(int cg[0] = 0 ; cg[0] < 5 ; ++cg[0])
  calculMatriceLocale(local[crdId(cg)],ie,cg);

  for(int k = 0 ; k < 5 ; ++k)
   for(int l = 0 ; l < 5 ; ++l)
    for(int m = 0 ; m < 5 ; ++m)
  assembler(local,ie,cg);
 }
}
```

FIGURE 4.5 – Fonction principale noyau de calcul des forces internes

à ce niveau. Dans la figure 4.5, le code la boucle principale se scinde en 2 parties. La première calcule la matrice locale de chaque point GLL de l'élément. Le second cumul les contributions selon la matrice de chaque point GLL et les paramètres y afférent.

La première phase du calcul de chaque élément fini consiste à calculer la matrice locale de chaque point GLL par la fonction détaillée dans le listing 4.6. Ainsi, le tableau « local » est la matrice 3x3 du point GLL aux coordonnées 3D « cg » pour l'élément d'indice « ie ». Ce calcul se décompose à son tour en une succession d'opération, dont des opérations matricielles. On notera que les opérations matricielles se réalisent ici sur des petites matrices. Dans le cas présent, aucune optimisation explicite sur le calcul de ces matrices n'est réalisée dans le code traduit en C++. De plus, l'optimisation du calcul sur de petites matrices n'est pas évidente bien que des travaux existent [1]. Nous allons maintenant nous intéresser à la fonction « calculerMatDud » dont l'objectif est d'établir les valeurs initiales de la matrice locale depuis les valeurs physiques de displacement associés aux points GLL.

La figure 4.7 détaille le calcul des valeurs initiales de la matrice locale à un point GLL. Ce calcul nécessite des données physiques de points GLL référencé dans l'élément traité. De fait, une indirection depuis des indices de référencements est réalisée pour atteindre les données physiques. En effet, le sous tableau « elm_refGLL[ie] » contient $5 \times 5 \times 5 = 125$ entiers dont un pour chaque point GLL de l'élément « ie ». Chacun de ces entiers indique une case à 3 composantes flottantes du tableau « displacement ». Or, le tableau « displacement » contient les données physiques de displacement des points GLL qui sont ainsi accédé via l'indirection.

Le traitement des éléments dans ce code est bien explicitement scalaire et non vectoriel. Comme nous l'avons vu lors de la sous-section 2.2.2, les instructions SIMD forment un premier niveau de parallélisme. Or, le code C++ Efispec3D présenté dans cette sous-section n'explicite pas de calcul vectoriel ou l'usage d'intrinsics. Par conséquent, l'usage des ressources SIMD de la plateforme cible repose sur le compilateur. Cependant, la

```
void calculMatriceLocale(float *local,int ie,int cg[3])
{
 int GLLi = crdId(cg);
 float *mLocGLL = pMatGLL[ie][GLLi];

 float matriceDud[3*3];
 float matD[3*3];
 float matTau[3*3];

 calculerMatDud(matriceDud,ie,cg);
 matriceMultipli(matD,matriceDud,mLocGLL);
 calculerMatTau(matTau,pRovpGLL[ie][GLLi],pRovsGLL[ie][GLLi]);
 matriceMultipli(local,matTau,mLocGLL);
 matriceMultipli(local,pJacoGLL[ie][GLLi],local);
}
```

FIGURE 4.6 – Fonction de calcul de la matrice locale pour un point GLL donné.

```
void calculerMatDud(float* matDud, int ie,int cg[3])
{
 memset((void*)matDud,0x00,sizeof(float)*3*3);
 int GLLi = crdId(cg);
 for( std::size_t i = 0 ; i<5 ; ++i )
 {
  int fd=1;
  for(int p=0;p<3;++p)
  {
   for(int c=0;c<3;++c)
   dud[p][c]+=
    displacement
 [
   elm_refGLL[ie][GLLi+(i-cg[p])*fd]
 ][c]*lagrgDrv[cg[p]][i];
   fd*=5;
  }
 }
}
```

FIGURE 4.7 – Calcul de la matrice des contributions physiques pour un point GLL donné.

```
void assembler(float local[5*5*5][3*3],
               int ie,int cg[3])
{
 float *accel_pg = acceleration[elm_refGLL[ie][crdId(cg)]];
 float matAcu[3*3];
 memset((void*)matAcu,0x00,sizeof(float)*3*3);
 int GLLi = crdId(cg);
 int fd=1;
 for(int p=0;p<3;++p)
 {
  for(int c=0;c<3;++c)
   for(int i=0;i<5;++i)
    matAcu[p*3+c]+=local[GLLi+(i-cg[p])*fd][p*3+c]*
                   lagrgDrv[i][cg[p]]*gllWeight[i];
  fd*=5;
 }
 float f1 = gllWeight[l]*gllWeight[k];
 float f2 = gllWeight[m]*gllWeight[k];
 float f3 = gllWeight[m]*gllWeight[l];
 for(int c=0;c<3;++c)
  accel_pg[c]-=f1*matAcu[0*3+c]+f2*matAcu[1*3+c]+f3*matAcu[2*3+c];
}
```

FIGURE 4.8 – Fonction d'assemblage des contributions de chaque point GLL

vectorisation automatique du compilateur ne garantit aucun résultat. En pratique, la vectorisation automatique par le compilateur peut s'avérer inefficace voir inexistante comme nous l'avons précédemment constaté (Chapitre 3).
La fonction d'assemblage « assembler » accumule les contributions d'accélérations d'un point GLL de coordonnées « cg ». Ainsi, de la même façon, l'accélération 3D du point GLL est accédée par indirection via l'indice « elm_refGLL[ie][0..124] » dans le tableau « accélération ».

Pour résumé, nous avons constaté que ce code n'utilise pas d'intrinsics ni de parallélisme multi threads SPMD. Enfin, les données physiques sont accédés via une indirection. Notre intuition est qu'il est possible d'optimiser ces différents points pour un minimum d'écart avec la version d'origine.
Nous allons vérifier dans la suite de ce chapitre le bien-fondé de ces intuitions.

4.1.4 L'intra nœud MPI

L'implémentation actuelle d'Efispec3D est déjà parallélisée via un calcul découpé par METIS puis distribué par MPI. Or, nos machines d'architectures NUMA ont la particularité d'avoir plusieurs cœurs pour lesquels la mémoire est matériellement partagée. Cette parallélisation distribuée du calcul présente des faiblesses lorsque l'on souhaite distribuer le calcul dans une même machine. Notre intuition est que la mémoire partagée est au cœur de cette faiblesse.

En C++, il existe des solutions pour mettre rapidement en oeuvre un parallélisme

intra-neud. En effet, l'environnement d'exécution OpenMP abordé dans le chapitre précédent est une solution tout indiquée. Ainsi, nous cherchons à déterminer s'il serait plus efficace d'assurer ce parallélisme intra-nœuds en exploitant l'aspect mémoire partagé via OpenMP. Notre intuition repose sur le fait que l'utilisation de MPI en intra-nœuds ne bénéficie pas efficacement de l'aspect matériel de la gestion mémoire. Ceci expliquerait l'inefficacité du parallélisme SPMD assuré par MPI à passer à l'échelle en intra-nœuds. En effet, le graphique de droite de la Figure 4.2 montre la performance obtenue jusqu'à 32 processus MPI avec une répartition naïve des processus. Nous pouvons observer une dégradation des performances lorsque tous les 32 processus MPI s'exécutent sur le même nœud de calcul. L'utilisation d'une distribution alternée des processus sur les 32 nœuds de calcul améliore les performances. Cela peut être dû à une meilleure utilisation de la bande passante mémoire disponible au niveau des nœuds.

La liste d'éléments finis à itérer semble imposer un ordre de traitement de ces derniers. Cependant, il est possible de les traiter dans n'importe quel ordre. Ainsi, une réorganisation avancée de cette liste peut permettre de traiter successivement des éléments contigus en mémoire. L'algorithme de Cuthill-McKee [16] permet notamment d'optimiser la localité des données selon leurs dépendances. En outre, l'exploitation des capacités de vectorisation des processeurs repose sur le compilateur. Malheureusement, l'accès irrégulier à la mémoire inhérent à la méthode des éléments finis empêche toute vectorisation automatique efficace. Ces expériences soulignent que le modèle de programmation fondé sur MPI peut ne pas être suffisant pour extraire la performance optimale de l'architecture sous-jacente. C'est particulièrement vrai au niveau de la mémoire partagée et nous devons donc redéfinir l'algorithme d'assemblage afin de bénéficier à la fois du multithreading et du parallélisme SIMD.

Des axes d'optimisations sont mis en oeuvre et évalués dans la suite de ce chapitre. Cependant, afin de définir le contexte d'évaluation, la section suivante passe en revue les paramètres des expérimentations.

4.2 Plateformes expérimentales cibles

Dans cette section, nous passons en revue le protocole expérimental avec ses paramètres. De ces paramètres, va découler des modèles d'analyse telle que le modèle roofline. Ainsi, nous précisions le contexte des expériences menées afin de pouvoir analyser leurs résultats.

4.2.1 Les plateformes de calculs

Les expérimentations ont été faites avec 2 plates-formes. La plateforme principale employée dans ce papier fonctionne avec un bi-processeur Gold 6148 Intel Skylake. Pour compléter l'expérimentation, nous avons traité avec un bi-processeur Intel Xeon-2697 v4 Broadwell platform.

Leurs caractéristiques techniques théoriques sont précisées dans la table 4.1. On y remarque notamment que la fréquence du processeur peut varier suivant les instructions sollicitées. Ainsi, l'utilisation d'instructions AVX-512 réduit la fréquence processeur à 1,6 GHZ pour la Skylake.

TABLE 4.1 – CARACTÉRISTIQUES DES PLATEFORMES

Platforme	**Skylake**	**Broadwell**
cœurs physiques	2×20	2×18
fréquence de base	2.4GHz	2.3GHz
fréquence avec AVX	1.9GHz	2.0GHz
fréquence avec AVX-512	1.6GHz	-

Les résultats présentés sont issus des programmes expérimentaux compilés avec Clang5. Cependant, afin de confirmer les comportements et les résultats, nous employons également le compilateur ICC17 d'Intel.
L'analyse des impacts de nos optimisations se fait grâce aux expérimentations de différentes implémentations. Ainsi, plusieurs combinaisons de nos optimisations y sont conjointement évaluées. La sous-section suivante présente les modèles et outils d'analyse servant à évaluer ces expérimentations.

4.2.2 Outils et modèles d'analyse

Dans la sous-section 3.3.2 du chapitre précédent, l'idée est de déterminer un facteur de réutilisation par stencil pour se positionner sur la roofline. De la même façon dans ce chapitre, nous constituons un facteur adapté au calcul d'éléments finis pour l'analyse de nos travaux. Ainsi, dans ce chapitre, l'Intensité Opérationelle (OI) nous permet de positionner nos versions de noyaux sur la roofline suivant leur ordre polynomial d'approximation et leur mode d'accès aux points physiques.

Intensité opérationnelle et Rooflines

Chaque élément d'ordre p est composé de $(p+1)^3$ points GLL. Le calcul d'un élément consiste à charger des valeurs de point GLL et à exécuter des opérations en virgule flottante avec cet élément. Tous les octets chargés peuvent être réutilisés par les opérations selon l'ordre des éléments tel que détaillé dans la table 4.2. Pour calculer un élément d'ordre 2, il y a $(2+1)^3 = 27$ points GLL associé à autant de points physiques à charger localement avec d'autres paramètres (pondération et dérivés de Lagrange). Il en coûte 1668 octets à charger, mais 7974 opérations flottantes à calculer. Ensuite, l'intensité opérationnelle d'un élément de calcul de l'ordre 2 est de $7974/1668 = 4,78$. Ce facteur représente un nombre d'opérations flottantes réutilisant chaque octet chargé depuis la mémoire centrale.

Nous comparons nos différentes implémentations de noyau en utilisant leurs Intensités Opérationnelles (OI), comme défini dans [67]. Cet OI est défini comme le rapport entre le nombre d'opérations en virgule flottante et le nombre d'octets chargés à partir de la mémoire centrale.
Chaque élément d'ordre o est composé de $(o+1)^3$ points GLL. Le calcul d'un élément consiste à charger ses valeurs de point GLL et à appliquer des opérations en virgule flottante. La réutilisation des données chargés depuis la mémoire centrale varie selon l'ordre polynomial d'approximation tel que détaillé dans le tableau 4.2. De plus, on remarque dans ce tableau que 2 modes d'accès sont présents. Les spécificités de mise en oeuvre du mode d'accès direct sont abordées plus loin dans ce chapitre. Cependant, on notera que les noyaux d'accès indirect ont un OI plus faible. En effet, ces derniers requièrent

le chargement d'indices d'indirection afin d'accéder aux données physiques associées aux points GLL. Par conséquent, il y a plus de données à charger pour la même quantité de calculs à exécuter. De fait, l'OI est plus faible.

Le calcul d'un élément tridimensionnel d'ordre 4, comme celui de la figure 4.1, nécessite $(4+1)^3 = 125$ valeurs de point GLL à charger à partir de DRAM et de certains paramètres supplémentaires (poids de point GLL et dérivés de Lagrange). Il représente 11120 octets de données sur lesquels 48150 opérations en virgule flottante sont appliquées. Ainsi, l'intensité opérationnelle d'un élément d'ordre 4 est de $48150/11120 = 4,33$. Nous déterminons cet OI pour les différentes implémentations du noyau et nous les comparons entre elles.

TABLE 4.2 – INTENSITÉ OPÉRATIONNELLE (O.I) SELON L'ORDRE D'APPROXIMATION ET LE MODE D'ACCÈS AUX POINTS PHYSIQUES.

Ordre	Flop/Element	Mode d'accès	Octet/Element	OI
2	7974	Direct	2316	3.44
		Indirect	2424	3.29
4	48150	Direct	10620	4.53
		Indirect	11120	4.33
8	411966	Direct	61596	6.69
		Indirect	64512	6.39

Pour comparer l'OI de chacune de nos implémentations, nous proposons d'utiliser le modèle ORM afin d'étudier les performances et les limites de nos implémentations. Cependant, dans [31, 45, 21], les auteurs montrent certaines limites de la ORM lorsque le modèle est utilisé pour piloter le processus d'optimisation. Ces travaux proposent d'autres modèles, tel que le modèle Cache Aware Roofline Model (CARM) et Locality Aware Roofline Model (LARM), qui prennent en compte davantage de détails architecturaux. Dans notre cas, nous n'utilisons l'ORM pour discuter de la performance relative de nos implémentations et non pour piloter le processus d'optimisation. Par ailleurs, nous aurons l'occasion de le mettre en défaut.

Les graphiques rooflines ont été établis pour les deux plates-formes à partir d'un benchmark BLAS SGEMM et du benchmark STREAM [47]. La figure 4.9 montre les rooflines résultantes. Les plates-formes Broadwell et Skylake atteignent une performance de pointe de respectivement 2,314 TFLOPS et 3,826 TFLOPS.

De plus, nous avons introduit un pic de performance applicative pour chaque architecture. Ces valeurs sont calculées avec un petit maillage capable de tenir en mémoire cache. Dans ce cas, nous mesurons une valeur maximale de 948 GFLOPS sur Broadwell et 1236 GFLOPS sur Skylake. Sur la base de ces nouvelles métriques, nous pouvons observer que l'implémentation des versions de l'ordre 8 est limitée par la puissance de calcul des processeurs dans presque tous les cas.

vTune un outil de profilage

Le logiciel applicatif vTune est un outil de profilage statique et dynamique. L'analyse statique d'un exécutable ou d'un code source permet à l'outil d'évaluer les calculs et

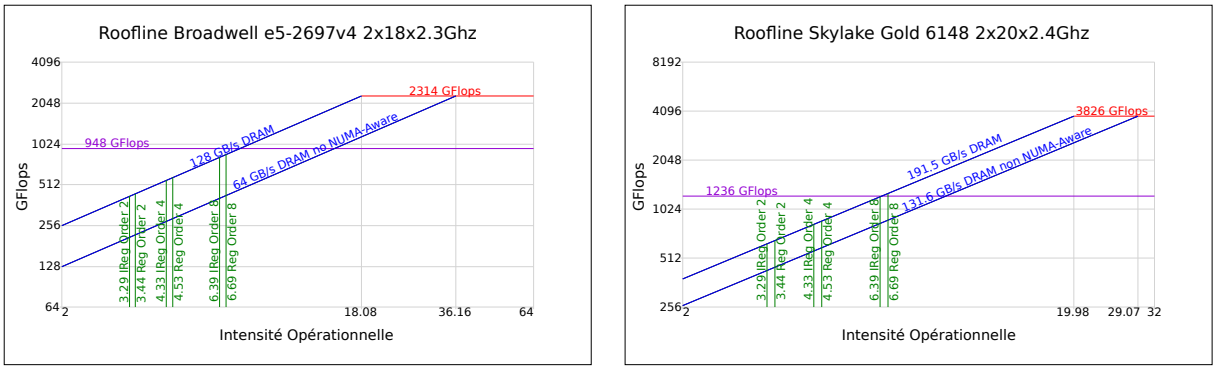

FIGURE 4.9 – Le modèle roofline des plateformes Broadwell et Skylake.

leurs mises en oeuvre. L'objectif est de constituer des indicateurs et produire des conseils d'amélioration.
Cet outil nous sert dans ce chapitre pour relever des compteurs matériels durant l'exécution de nos noyaux. Un des indicateurs que nous allons employer est celui du taux de cycles perdus à attendre les caches mémoires. Plus il est élevé et plus les caches sont saturés. D'autres facteurs similaires ont été observés et nous confortent dans nos interprétations des résultats.
Les expériences sont réalisées suivant ce contexte expérimental. Nous passons lors de la section suivante au premier axe d'optimisation du noyau.

4.3 Vectorisation

Nous allons maintenant aborder une des manières d'exploiter les instructions vectorielles disponibles dans nos architectures cibles. On démarre depuis la version du noyau C++ traduit de l'application Efispec3D. Ainsi, on commence par élaborer une implémentation vectorielle au plus proche de l'existant.

4.3.1 Parallélisation explicite SIMD du noyau séquentiel Efispec3D

Dans le cadre de ces travaux, l'exploitation des instructions vectorielles est basée sur l'utilisation explicite d'intrinsics. En effet, lors du chapitre 3, nous avons observé que les compilateurs ne sont pas toujours capables de vectoriser le code efficacement. Avant d'aborder la particularité de l'accès aux données physiques, nous abordons la vectorisation générale du noyau.

Il existe plusieurs façons de vectoriser le calcul d'éléments finis. Par exemple, une des façons consiste à vectoriser les calculs matriciels du calcul éléments finis. Ainsi, on calcule qu'un élément par cœurs à la foi, mais plus rapidement du fait de la vectorisation de son calcul matriciel.
Dans le cadre de cette thèse, nos noyaux éléments finis vectoriels calculent en chaque cœur plusieurs éléments finis à la fois selon la taille des registres SIMD disponibles. En effet, dans sa version séquentielle, notre noyau de calcul réalise les mêmes opérations pour tous les éléments finis. Ainsi, lorsqu'il est question d'appliquer des opérations entre les points

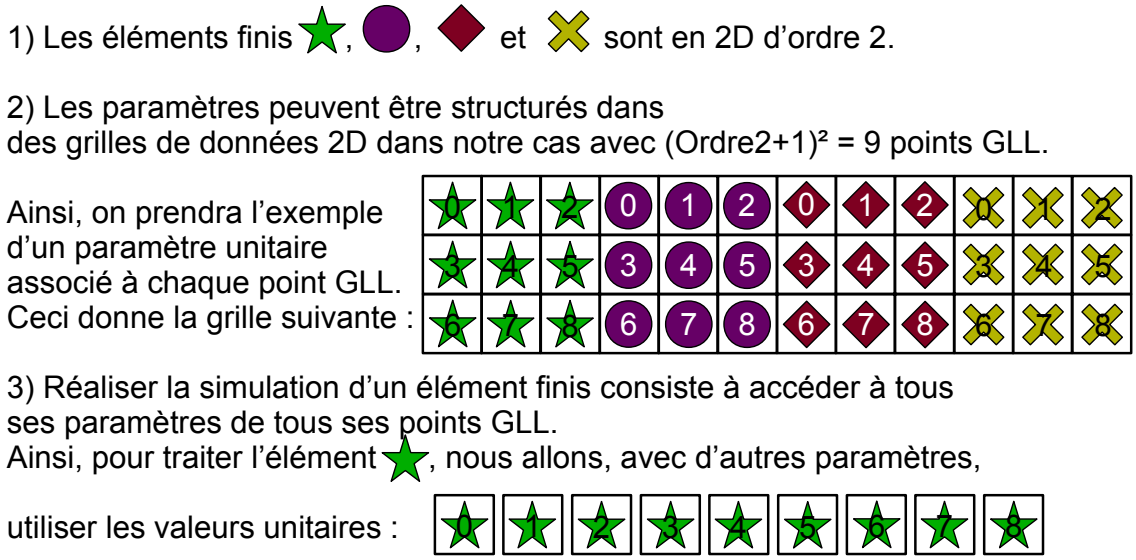

FIGURE 4.10 –

GLL d'un même élément, ceci peut se faire à l'identique pour plusieurs éléments à la fois.

Lors de la résolution d'un seul élément fini, il faut charger localement à l'élément ses paramètres et grandeurs physiques ainsi que d'autres paramètres communs. Ces différentes valeurs sont alors utilisées dans les mêmes opérations réalisées pour chaque élément fini. Or, les paramètres locaux attachés à chaque point GLL peuvent être disposés en grille comme illustrée par la figure 4.10. En effet, ces paramètres sont privés à chaque point GLL comme vu dans la figure 2.6 de la sous-section 2.1.3.

De la même façon, il est possible de charger les valeurs de plusieurs éléments finis à la foi comme montrée dans la figure 4.11. La figure 4.12 présente la disposition en registre vectoriel des paramètres de 4 éléments. Ils peuvent ainsi être traités en même temps en utilisant des instructions vectorielles. En effet, elles permettent d'appliquer les mêmes opérations qu'en séquentiel à chacune des valeurs entre un ou plusieurs registres vectoriels.

On remarque dans la figure 4.11 que la disposition des paramètres nécessite de charger le vecteur en rassemblant des valeurs dispersées dans la grille de données. Or, la figure 4.13 montre qu'il est possible dans cette grille d'entrelacer les paramètres par lot afin de les charger directement dans des registres vectoriels. Reprenons l'illustration 2.6 (sous-section 2.1.3) de la structure abstraite d'un élément fini. Chaque élément 3D d'ordre 4 est décrit par $(4+1)^3 = 125$ instances de la structure de données « Paramétrage local d'un point physique globale ». Ainsi, le calcul scalaire consiste à parcourir un par un des lots de 125 instances de la structure stockés les unes à la suite des autres. Dans l'implémentation de cette structure 2.7 (sous-section 2.1.3), les tableaux de flottants « *dxidx, *dxidy, *dxidz, ..., *dzedz » font partie de la structure « Paramétrage local d'un point physique globale ». Chaque élément possède donc 125 flottants dans chacun de ces tableaux. Prenons l'exemple du flottant dxidx$[5(5(5E+L)+K)+M]$ qui appartient au paramétrage

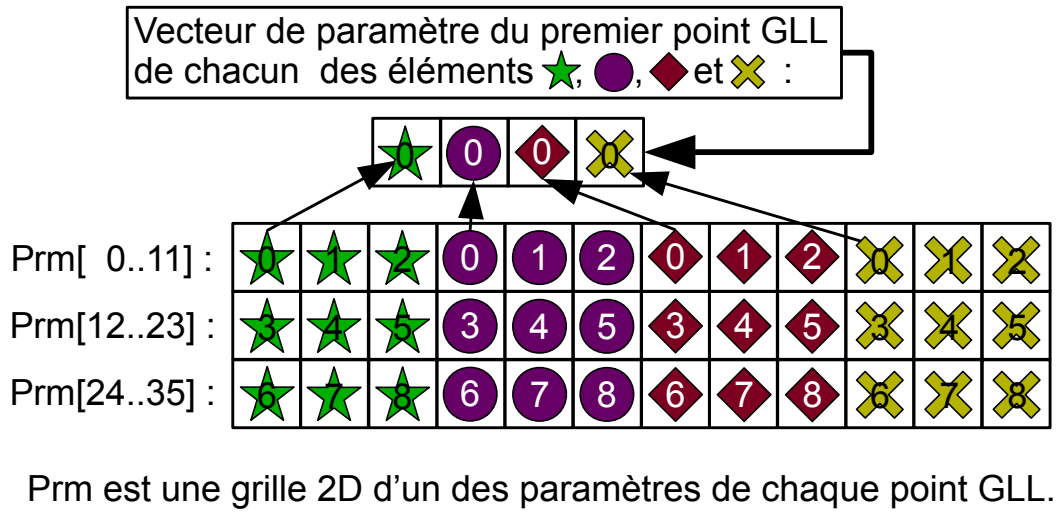

FIGURE 4.11 – Disposition mémoire **sans** alternance du paramétrage des points GLL d'éléments finis 2D à l'ordre 2. L'absence d'alternance nécessite de rassembler les paramètres pour chaque vecteur traitant 4 éléments à la fois. Ceci implique différents accès à des données éloignées et requiert l'usage d'une instruction de restructuration de ces données en vecteur.

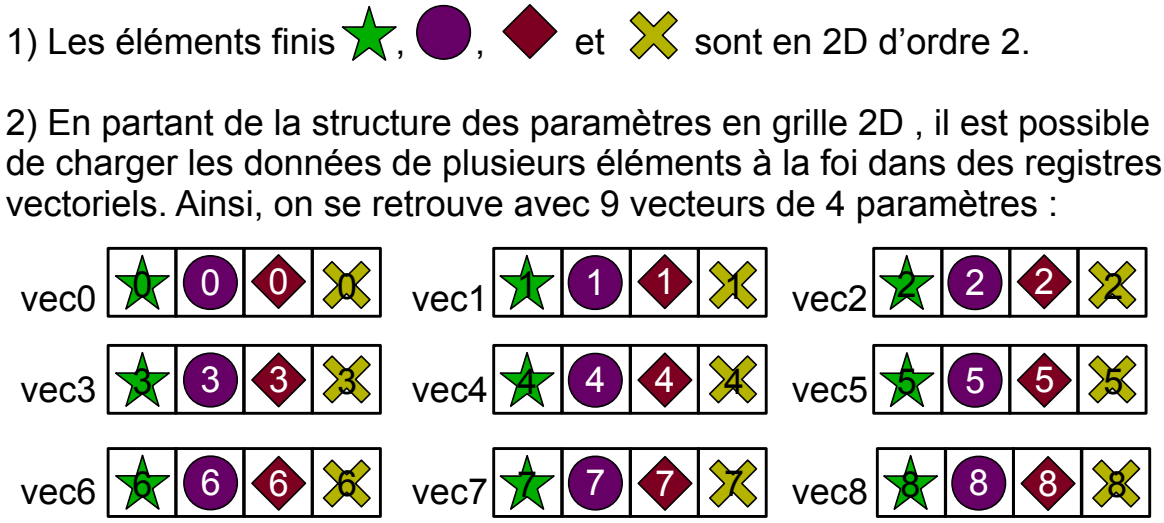

FIGURE 4.12 –

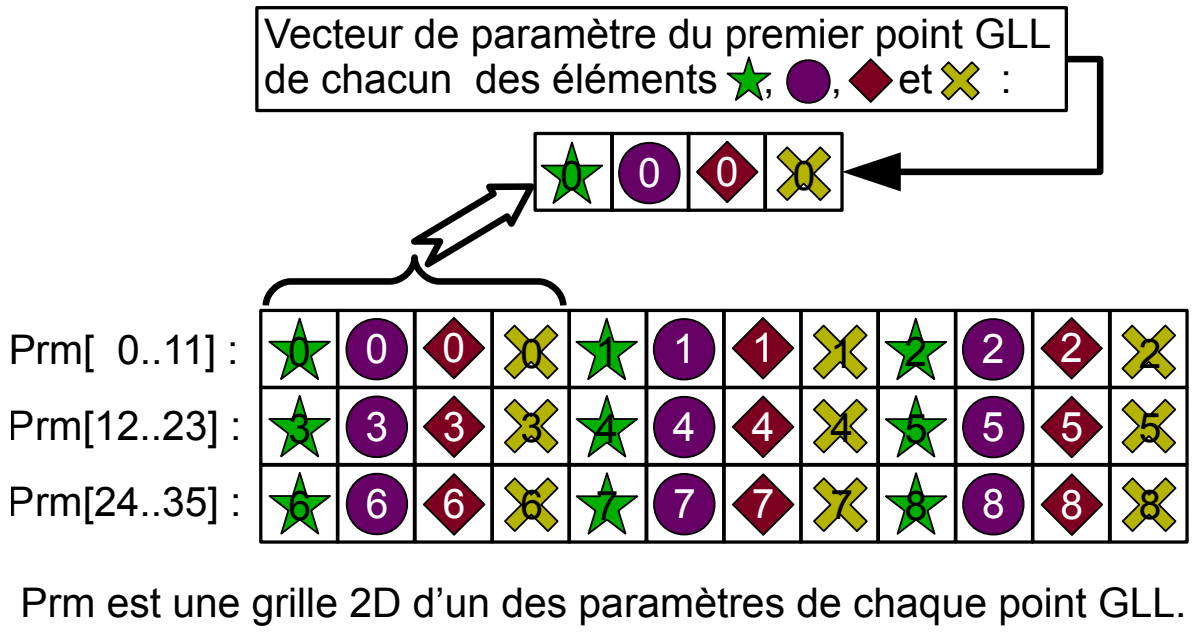

FIGURE 4.13 – Disposition mémoire **avec** alternance du paramétrage des points GLL d'éléments finis 2D à l'ordre 2. L'alternance permet de charger directement les paramètres dans chaque vecteur traitant 4 éléments à la fois. Ainsi, seule l'opération de chargement est nécessaire.

« dxidx » du point GLL $Elemnt_E : GLL_{K,L,M}$. Cette disposition mémoire des paramètres rend particulièrement délicat et peu efficace le chargement de plusieurs éléments à la suite dans des registres vectoriels. En effet, en prenant un vecteur SIMD de 8 flottants, l'idéal serait de charger directement 8 facteurs des 8 éléments en partant du vecteur d'élément E via l'instruction vectorielle vLoad<VFloat>(&dxidx[8(5(5(5$E+L$)$+K$)$+M$)]). Cependant, du fait de la disposition en mémoire élément par élément, le facteur du point GLL $Elemnt_E : GLL_{K,L,M}$ est suivi en mémoire par le facteur du point GLL suivant $Elemnt_E : GLL_{K,L,M+1}$. Or, ce dernier n'est pas le facteur de l'élément suivant $Elemnt_{E+1} : GLL_{K,L,M}$ que l'on cherche à charger à la suite dans un même registre vectoriel. Afin de pallier à ce problème, les instances des 125 structures doivent être entrelacées par vecteurs d'éléments suivant le principe illustré par la figure 4.13. Il faut redisposer les données en mémoire et y accéder en tenant compte de cette transformation. Par conséquent, le facteur flottant dxidx[8(5(5(5$E+L$)$+K$)$+M$)] est suivi de 7 facteurs appartenant aux paramétrages du même point GLL des 7 autres éléments du même vecteur. De fait, il est possible de charger directement en registre vectoriel les données des éléments.

Les architectures cibles que nous employons permettent de traiter de 8(AVX) à 16(AVX-512) flottants simples précisions en même temps. De fait, il nous est possible de traiter de 8 à 16 éléments à la fois sur un seul cœur.

Cas de la structure d'indirection (maillages non structurés)

L'implémentation vectorielle diffère selon le mode d'accès aux données physiques associés aux points GLL des éléments finis. En effet, pour chaque élément fini, il est nécessaire d'identifier les points physiques qui y sont associés selon les coordonnées « K,L et M » d'un des points GLL allant de 1 à 5 à l'ordre 4. Afin de répondre à ce besoin, la simulation de maillages non structurés emploie des références dans les paramètres des points GLL pour chaque élément fini Fig. 2.6 (sous-section 2.1.3).

Dans le cas de l'implémentation du noyau via cette indirection, ces références associant les points GLL aux points physiques sont entrelacées tout comme on entrelace tous les paramètres des points GLL par vecteurs de 8(AVX) ou 16(AVX-512) éléments finis tels que vus précédemment. En outre, en cas d'indirection, les valeurs des points physiques n'ont pas besoin d'être entrelacées, car elles sont référencées. En effet, comme les éléments sont entrelacés avec leurs différents paramètres, les références aux points physiques des points GLL le sont également. Ceci nous permet de charger ces références directement dans un registre vectoriel de 8 entiers (16 en AVX-512). Ce chargement est réalisé comme suite par notre fonction substituable loadGLLref détaillée dans la figure 4.14. Ainsi, les données des points physiques associés aux points GLL se chargent via notre fonction loadGLLval Fig. 4.14.

La fonction loadGLLval permet de récupérer les valeurs flottantes depuis le tableau « displacement » en suivant les indices chargés dans un registre vectoriel depuis le tableau « glonum » des 8*125 références de points GLL par bloc de 8 éléments. Pour sa sauvegarde, on fait appel à la fonction antagoniste storeGLLval Fig. 4.14.

Comme on peut le voir dans la figure 4.14, toutes les instructions vectorielles utilisées dans nos fonctions ne sont pas toujours disponibles. Afin d'absorber ces différences, il suffit d'abstraire les fonctions vectorielles vGather et vScatter. Ainsi, seules leurs implémentations dépendent des jeux d'instructions vectorielles disponibles.

4.3.2 Évaluation de l'approche SIMD d'accès indirect aux points GLL

Le calcul vectoriel via des instructions SIMD est une optimisation calculatoire provenant de chaque cœur processeur. Nous évaluons ici le gain expérimentalement obtenu via ces ressources de calcul selon l'approche présentée lors de la sous-section précédente.

La version programmée sans instruction SIMD explicite ne vectorise pas bien quelque soit les compilateurs malgré le positionnement de flags ou d'annotations pragma. De fait, nous nous intéressons aux gains apportés à l'ordre 4 par la version programmée explicitement avec des instructions SIMD (intrinsics) par rapport à l'implémentation C++ séquentielle initiale.
Le tableau 4.3 présente nos principaux résultats expérimentaux. Ces résultats permettent de constater les écarts de performances entre une optimisation SIMD laissée aux compilateurs et une optimisation SIMD explicite réalisée à la main. Avec un seul cœur et malgré l'accès indirect aux points GLL, nous mesurons un gain de performance supérieur à 3 (de 3,32 GFLOPS à 10,58 GFLOPS sur Broadwell) pour un gain théoriquement maximal

```
//Charge un vecteur 256bits de 8 indices correspondant à
8 \glspl{gloss_pointGLL}
inline __mm256 loadGLLref(int E,int K,int L,int M)
{
 return _mm256_load_si256(&glonum[8*(5*(5*(5*E+K)+L)+M)]);
}

//Charge une des composantes X, Y ou Z
//des points physiques du vecteur d'éléments numéro E
//du vecteur de \glspl{gloss_pointGLL} aux coordonnées K,L,M.
inline __mm256 loadGLLval(int E,int K,int L,int M,int composante)
{
 return _mm256_i32gather_ps(displacement+composante,loadGLLref(E,K,L,M),4);
}

//Cette fonction est l'antagoniste de la précédente.
inline void storeGLLval(int E,int K,int L,int M,int composante,__mm256 vec)
{
 _mm256_i32scatter_ps(acceleration+composante,loadGLLref(E,K,L,M),vec,4);
}
```

FIGURE 4.14 – Fonctions d'accès indirect vectoriel aux points physiques des points GLL. Les éléments finis sont dans cet exemple d'ordre 4 et se composent ainsi de 125 points GLL. Par ailleurs, les registres vectoriels sont de 256bits. Les termes displacement et accélération sont deux tableaux contenant les points physiques de la simulation. Le terme E est un numéro du vecteur de 8 éléments finis. Les termes K,L et M représentent les coordonnées du point GLL cible dans l'élément. Le terme composant fait référence aux composantes X, Y et Z d'un point. Ces composantes s'alternent de façon contiguë en mémoire. Ainsi, les X sont aux indices i*3, les Y aux indices i*3+1 et les Z aux indices i*3+2. Le tableau glonum contient les références pour chaque point GLL selon une succession de petites grilles 3D de 125 entiers (5x5x5) entrelacées par 8.

de 8 à fréquence égale avec l'utilisation d'instructions AVX. La plateforme Skylake permet l'utilisation de registre vectoriel 2 fois plus grand que la plateforme Broadwell via l'AVX-512 pour un gain théorique de 16 à fréquence égale. Ainsi, le gain de l'accès indirect en mono-cœur sur Skylake atteint 4,5 (3,51 GFLOPS à 15,73 GFLOPS). Rappelons que la fréquence baisse avec l'utilisation des instructions vectorielles. Par exemple, sous Skylake la fréquence varie de 2,4Ghz à 1,6Ghz ce qui réduit en réalité le gain potentiellement atteignable d'un facteur de 1,5 (gain de 16 à 10,7).

SIMD manuel ou automatique	Broadwell Clang5/ICC17	Skylake Clang5/ICC17
automatique	3,32/3,29 GFLOPS	3,51/4,18 GFLOPS
manuel	10,58/6,11 GFLOPS	15,73/11,05 GFLOPS

TABLE 4.3 – Performances obtenues à l'ordre 4 avec le maximum de cœurs physiques disponibles pour les implémentations compilées via ICC17 ou Clang5.

Des essais ont également été menés pour vérifier l'apport du passage des registres SIMD 256bits à 512bits. Ainsi, les variantes suivantes ont été évaluées :

- **AVX-2** : Elle fonctionne sur des registres 256 bits et peut fonctionner sur les plateformes Broadwell et Skylake..
- **AVX-512** : Elle fonctionne sur des registres de 512 bits, donc il ne peut fonctionner que sur la plateforme Skylake..
- **AVX-512/256** : Elle est identique à la version AVX-2 mais utilise une instruction de diffusion SIMD 256 bits (accessible par l'intrinsix_mm256_i32scatter_ps) disponible uniquement dans le jeu d'instructions AVX-512.

Le fait de ne calculer qu'avec un seul cœur présente de meilleurs résultats en passant de 256bits(13,31 GFLOPS) à 512bits(15,72). Cependant, on notera que le gain est faible. Nous aurons l'occasion de revenir sur ces essais lorsque l'on distribuera le calcul sur plusieurs threads. De plus, le changement de la structure des données peut également impacter ce gain.

Le calcul réalisé via les instructions SIMD par paquets d'éléments exhibe des résultats encourageants. La prochaine étape consiste à évaluer l'emploi de plusieurs cœurs combinés avec cette vectorisation.

4.4 Parallélisation multithreads

Le calcul du noyau étant vectorisé, il est temps d'utiliser tous les cœurs de la machine. L'exploitation des plateformes multi-cœurs à mémoire partagée n'est pas simple et nous devons faire face à plusieurs difficultés. En effet, pendant la phase d'assemblage (accumulation des résultats de la simulation élément fini dans les points GLL) en parallèle, les éléments finis partagent des points physiques. Il faut donc éviter les conditions de compétition pendant la procédure de calcul en parallèle vue dans la sous-section 2.1.3. Pour ce faire, nous présentons ci-après l'approche multithreads employée dans nos noyaux.

4.4.1 Multithreading pour le noyau EFISPEC

Comme expliqué dans la sous-section 2.2.3 et 3.5, le multithreading peut engendrer plusieurs effets contre-productifs. Un des effets est celui dû au partage entre les threads de mêmes lignes de caches. Dans ce cas, les threads invalident les lignes partagées modifiées par les autres threads. De plus, des effets de compétitions impactent la validité des résultats. Pour surmonter ce point, les threads doivent être synchronisés. Leur synchronisation peut engendrer de la famine. Or, la famine doit être limitée pour ne pas trop impacter les performances. Il en va de même pour les surcoûts liés à la gestion de ces threads notamment dus aux appels système. Enfin, l'accès à la mémoire n'est pas homogène entre les nœuds de traitements et les nœuds mémoires. En effet, la localité en mémoire des données doit se rapprocher au mieux des affinités physiques des cœurs afin de prendre en compte l'effet NUMA.

Pour faire face à toutes ces contraintes, nous devons mettre en œuvre une stratégie à plusieurs niveaux. Du point de vue algorithmique, la stratégie *diviser pour régner* décrite dans [62] semble très élégante. Malheureusement, cela se fait au prix de modifications fondamentales de l'application d'origine.

D'un point de vue bas niveau, l'utilisation d'*atomic* pourrait être une solution, mais ce n'est pas disponible avec les opérations vectorielles SIMD. De plus, les verrous explicites de type mutex sont difficiles à gérer et leurs appels système sont coûteux.
Par conséquent, nous mettons en œuvre une stratégie de coloriage par vecteur de 8 ou 16 éléments comme indiqué dans la figure 4.15 avec un nombre minimal de synchronisations (une par couleur). Autant que possible, nous planifions pour chaque thread le même nombre d'éléments spectraux à traiter. Ainsi, les threads traitent plusieurs vecteurs d'éléments par 8 ou 16 selon la taille des registres vectoriels. L'ordonnancement de la résolution par vecteurs d'éléments finis est présenté par la figure 4.16. Comme notre besoin en parallélisme SPMD est assez régulier, notre stratégie multithreading repose sur la directive *parallel for* de la bibliothèque OpenMP.

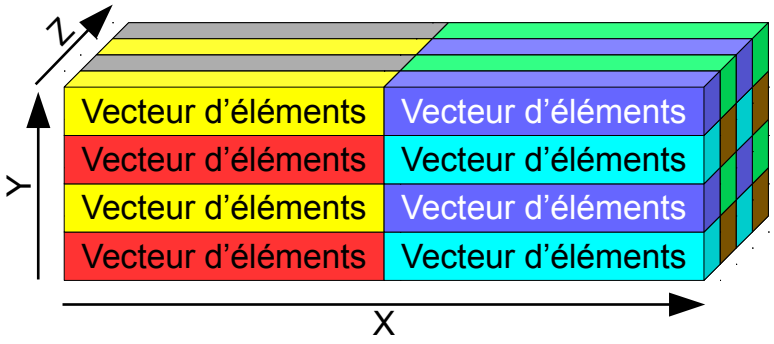

FIGURE 4.15 – Coloriage de vecteurs d'éléments finis d'un maillage structuré.

L'implémentation du calcul en multi-cœurs est impactée par le mode d'accès aux points physiques. Lorsque l'on accède indirectement aux données physiques, le calcul consiste à parcourir les éléments à calculer depuis un ou plusieurs tableaux. Or, lorsque les vecteurs d'éléments sont colorés, ils doivent être séparés suivant leur couleur. Ainsi, nous aurons des listes de plusieurs vecteurs d'éléments de la même couleur (figure 4.15). En effet, les

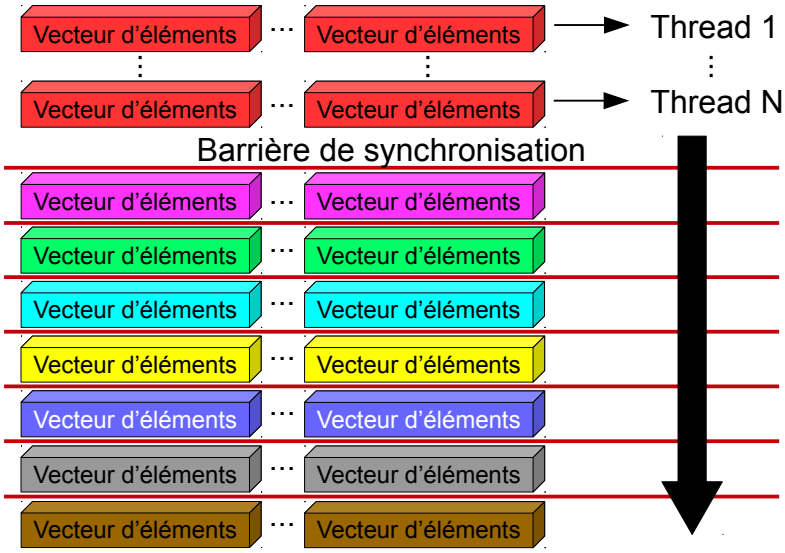

FIGURE 4.16 – Ordonnancement de calcul en parallèle par vecteurs d'éléments finis suivant un coloriage.

vecteurs de même couleur sont indépendants les uns avec les autres. Par conséquent, ils peuvent être traités en parallèle sans aucun risque lié à la compétition des threads.

Nous venons de voir la technique choisie ici assurant l'exécution multi-threads du calcul (SPMD). Le code de cette implémentation est disponible à l'annexe D. Dans une perspective de progression, nous allons tout d'abord évaluer les performances du parallélisme SIMD combiné avec ce parallélisme multi-cœurs.

4.4.2 Évaluation de la vectorisation dans un contexte multi-cœurs

Les courbes de la figure 4.17 présentent les performances obtenues avec un accès indirect des points GLL. Tout d'abord, on observe en bleu que l'implémentation automatiquement vectorisée par le compilateur n'est pas concluante. Sur les deux plateformes, nous atteignons à peine une moyenne de 140 GFLOPS, ce qui représente moins de 5% de la performance théorique maximale. La version explicitement vectorisée à la main montre de bien meilleurs résultats avec un maximum de 316 GFLOPS sur Broadwell et 390 GFLOPS sur Skylake. Cela démontre l'efficacité de notre approche explicite du calcul vectoriel par rapport à la vectorisation automatique de Clang5 ou ICC17 pour nos architectures. On peut aussi remarquer que le gain avec la plateforme Skylake est limité à 23% en comparaison avec l'architecture Broadwell, malgré la disponibilité d'AVX-512 ainsi que d'un plus grand nombre de cœurs de calcul. La tendance est très similaire entre les compilateurs Intel et Clang comme le montre le tableau 4.4.

À l'inverse des essais mono-cœurs, une comparaison entre l'usage des registres 256bits et 512bits présente un faible gain de performance. En effet sous la plateforme Skylake, le parallélisme SIMD 256bits atteint 475,74 GFLOPS là où le parallélisme SIMD 512bits atteint seulement 485,18 GFLOPS. Ces résultats restent cohérents dans le sens où l'optimisation SIMD et SPMD augmente la puissance de calcul. Ces optimisations sont donc concurrentes face aux autres ressources comme la bande passante mémoire. De plus, on remarque qu'en multi-cœurs, le gain entre nos optimisations vectorielles explicites par

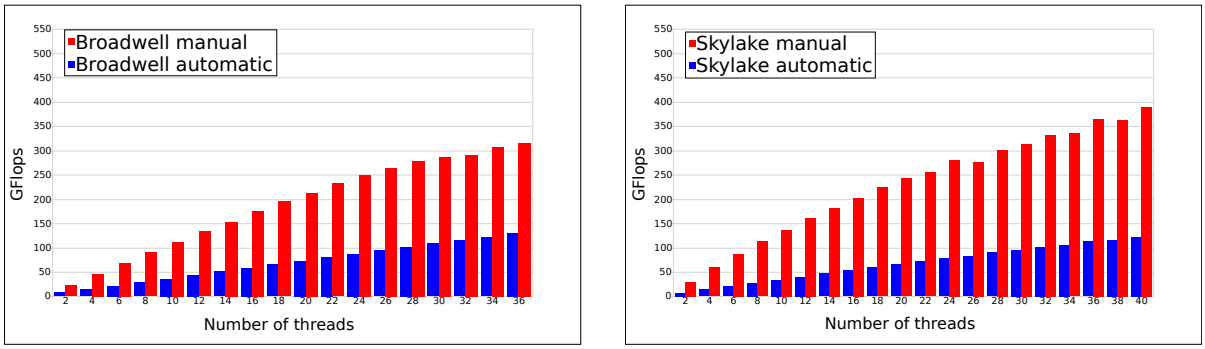

FIGURE 4.17 – Comparaison à l'ordre 4 entre la vectorisation automatique et manuelle sur les plateformes Broadwell et Skylake pour le noyau d'accès indirect en utilisant le compilateur Clang5.

SIMD manuel ou automatique	Broadwell Clang5/ICC17	Skylake Clang5/ICC17
automatique	131/123 GFLOPS	123/143 GFLOPS
manuel	316/218 GFLOPS	390/291 GFLOPS

TABLE 4.4 – Performances obtenues à l'ordre 4 avec le maximum de cœurs physiques disponibles pour les implémentations d'accès indirect compilées via ICC17 ou Clang5.

rapport à celles des compilateurs est moindre qu'en mono-cœur. En effet, avec tous les cœurs de la plateforme Skylake, on atteint un gain d'environs 2,5 là où il est de 4,5 avec un seul cœur. Le parallélisme SPMD compense la vectorisation inefficace des compilateurs du fait de la limitation due à la bande passante mémoire.

Dans cette sous-section, nous constatons de nouveau l'intérêt d'implémenter explicitement le noyau avec des intrinsics. En effet, les compilateurs ne parviennent pas à le faire efficacement en partant du code scalaire. La sous-section suivante va maintenant évaluer la pertinence de notre approche d'optimisation multi-cœurs.

4.4.3 Évaluation du Multithreading pour le noyau vectoriel Efispec3D

Ce second niveau d'optimisation consiste à exploiter plusieurs cœurs de la même machine par un parallélisme asynchrone (SPMD). Par conséquent, ceci accroît la puissance de calcul et sollicite davantage les caches et la bande passante mémoire. Ainsi, il est intéressant d'observer les gains de performance obtenus selon le nombre de cœurs sollicités.

Lorsque l'on regarde les graphiques de la figure 4.17, on constate que passage à l'échelle est satisfaisant. En effet, pour un accès indirect des points GLL, la plateforme Broadwell double sa performance de 22 GFLOPS à 43 GFLOPS en passant de 2 à 4 threads.

La tendance est la même pour les 2 compilateurs ainsi que sur la plateforme Skylake (32 GFLOPS à 63 GFLOPS pour 2 à 4 threads). Ceci indique que l'approche d'ordonnancer par coloriage le calcul des vecteurs d'éléments finis assure un bon parallélisme multi-cœurs avec un faible sûr-coût lié à la gestion des threads. Cependant, la tendance des graphiques est à la baisse à mesure que l'on ajoute des threads supplémentaires. Pourtant, l'apparence assez lisse des courbes n'indique pas un problème d'équilibrage de charge. En

effet, l'atténuation arrondie des performances à mesure que l'on ajoute des threads est caractéristique d'un facteur limitant. Notre analyse statique du noyau via nos rooflines nous indique que ce facteur limitant doit être la performance des accès mémoire.

L'approche par tuilage vue dans le chapitre 3 consiste à réutiliser les données du voisinage chargé en cache mémoire. Or, le coloriage de la figure 4.15 ne va pas dans ce sens puisque le voisinage induit une dépendance qui ne peut être calculée en parallèle. On peut donc supposer que cette approche par coloriage n'optimise pas les caches mémoires. Cependant, des travaux ont été menés sur le sujet [37, 52] concluent que cela impacte les performances à hauteur de 20%. En effet d'une part, l'effet est plus limité que dans le cas des stencils, car le calcul est différent. La différence se situe au niveau des données réutilisables. Par conséquent, seuls les points GLLs partagés seront réutilisés ce qui représente 98 points sur 125 à l'ordre 4 soit 78%. De plus, les calculs réutilisent déjà les points GLL chargés dans chaque élément. De fait, l'effet de cache lié à une mauvaise réutilisation des points inter-éléments est minimisé. D'autre part, notre noyau dont le calcul est vectoriel résout les éléments par paquets. Par conséquent, la réutilisation des points en est améliorée minimisant davantage le risque d'effet de cache. Cependant, il reste possible d'aller encore plus loin et de colorier par lots de plusieurs vecteurs d'éléments finis adjacents. Cette technique permettrait d'induire un effet de tuilage malgré le coloriage. Bien qu'il serait intéressant d'évaluer cette variante, l'impact serait infime. De plus, une attention toute particulière devra être portée sur l'équilibrage des charges de calculs.

Nos premières améliorations du noyau axées puissance de calcul sont encourageant. Comparé à l'implémentation MPI 4.2, le passage à l'échelle intra-nœuds de ces 2 optimisations est meilleur. De plus, à l'ordre 4 par accès indirect sous architecture Skylake, le gain s'élève à 138 (3,51 GFLOPS à 485,18 GFLOPS). Toutefois, des limitations apparaissent sur nos dernières courbes qui ressemblent à un plafonnement. Par conséquent, il reste à analyser d'autres améliorations sur des aspects mémoire. Ainsi, l'analyse de l'impact de l'accès aux données est l'objet de la section suivante.

4.5 Les modes d'accès aux données physiques

Le noyau Efispec3D d'origine calcule les éléments en chargeant les points physiques référencés via un indice comme illustré en haut de la figure 4.18. Ceci engendre une indirection des points GLL locaux vers les points physiques globaux. De plus, les points physiques globaux peuvent avoir une mauvaise contigüité dont la problématique est abordée dans la section 2.2. En l'état, l'accès aux données physiques de chaque point GLL se réalise en 2 temps via le chargement d'un indice supplémentaire pesant sur la bande passante mémoire. Afin de pallier à cette problématique, on propose d'évaluer l'implémentation d'un accès plus direct.

4.5.1 Passage au mode d'accès direct des points physiques

L'accès direct aux points physiques est rendu possible par une disposition régulière en mémoire. Ce mode d'accès est réalisable selon 2 approche.
Il est possible de dédier un point physique par point GLL local de chaque élément fini. L'avantage de cette approche est à la fois de rester sur du maillage non structuré tout

en évitant l'indirection. Cependant, ceci implique de dédoubler les points physiques associés à plusieurs points GLL. Or, ce dédoublage requiert un maintien de la cohérence de ces points physiques ainsi dédoublés. En effet, ils auront beau être dédoublés, ils correspondent à un même point ayant une même valeur. Cela revient à dupliquer une donnée et s'assurer qu'au terme d'un traitement, elles aient toujours la même valeur. Ce qui implique un maintien de cohérence. Or, le maintien en cohérence requiert le chargement d'informations sur quel point est à synchroniser avec quel autre. Cette approche mériterait une validation expérimentale afin de vérifier si elle permet de meilleures performances. D'une part cette approche requiert davantage de mémoire. D'autre part, une étape de mise à jour des points physiques dédoublés est nécessaire. Étant donné la nature des maillages traités dans notre domaine géoscientifique, les maillages présentent de grandes zones structurées comme vues dans la sous-section 2.1.3. Par conséquent, nous allons évaluer une autre approche permettant d'accéder directement aux points physiques.

Lorsque le maillage est structuré, il est possible de disposer les points physiques selon une grille 3D suivant le principe illustré par la figure 4.19. Ainsi, les points physiques partagés sont naturellement induits. En effet à l'ordre N, le premier point physique correspondant au premier point GLL de chaque élément a pour coordonnée les coordonnées de l'élément multiplié part N. De ce fait, il est possible d'accéder directement au point physique correspondant via une adresse calculable selon les coordonnées 3D des éléments. De plus, cette adresse peut-être en partie précalculée au fur et à mesure de l'imbrication des boucles sur les coordonnées des éléments finis. En effet, il n'est plus question de parcourir linéairement un tableau décrivant les éléments, mais une grille 3D régulière d'éléments accédant directement à leurs points physiques.

Ainsi, l'adresse d'un point physique se détermine avec la formulation d'adresse directe en bas de la figure 4.18. Cependant, les maillages non structurés ne permettent pas d'allouer les données physiques sous la forme d'une grille 3D régulière. Afin de comparer l'impact induit par un accès direct ou non des points physiques, les points physiques sont structurés de façon régulière dans les 2 cas tests. Ainsi, on est sûr de n'évaluer que le sur coût lié à l'indirection. De fait, la différence entre les implémentations directes et indirectes vient de la formulation de l'accès aux données des points Fig. 4.18.

En pratique, l'application Efispec3D doit traiter des mailles non structurées. Son implémentation d'origine nécessite l'utilisation d'un accès indirect aux points physiques. Or, les maillages géoscientifiques contiennent des zones étendues dont le maillage est structuré. Ainsi, la majeure partie du maillage est régulier à l'exception des interfaces entre deux régions dont la résolution diffère comme le montre les figures 2.3 et 2.5 (sous-section 2.1.3).

En résumé, pour un même maillage, l'idée est de gérer séparément ses parties structurées de ses parties non structurées.

L'optimisation sur la structure de données vue ci-dessus se cumule avec les optimisations vues précédemment. Cependant, l'implémentation vectorielle et multi-threads du noyau est impactée par ce changement de mode d'accès aux données. Ces modifications sont détaillées dans la sous-section suivante.

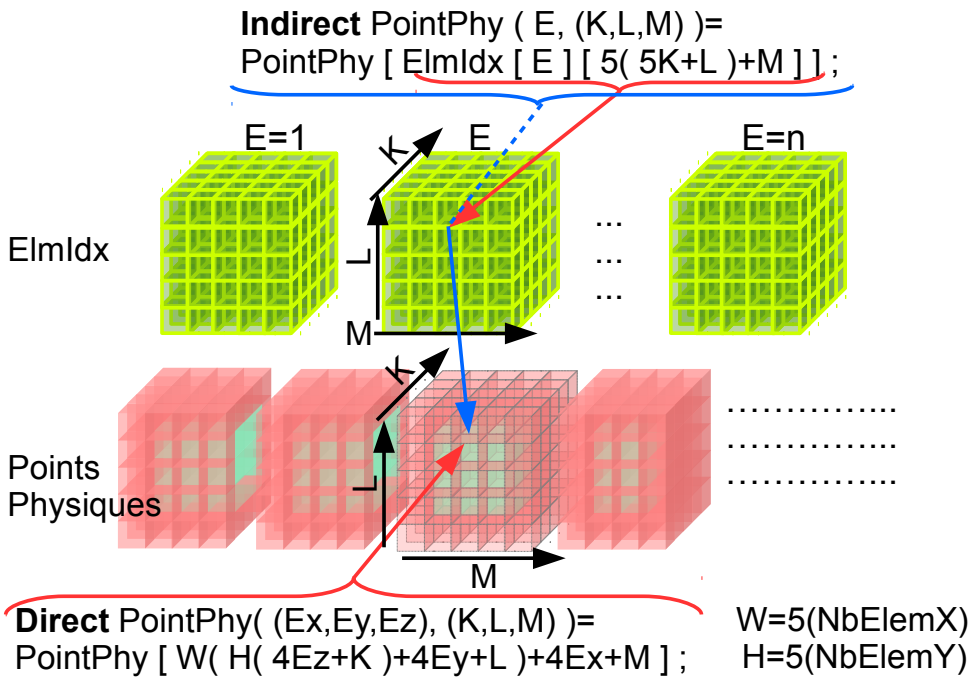

FIGURE 4.18 – Différence technique entre l'accès point physique direct et indirect vers des points physiques disposés de façon régulière issue d'un maillage structuré.

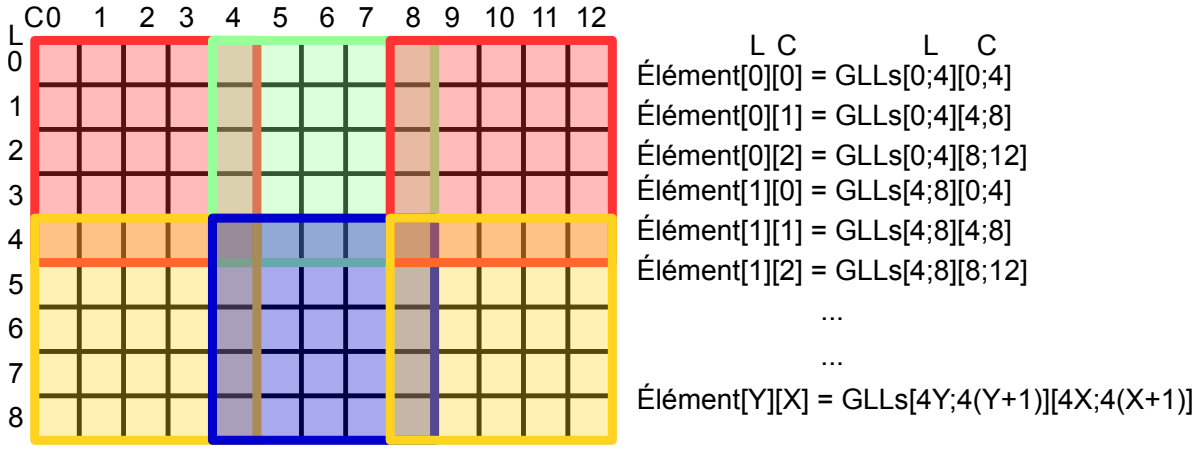

FIGURE 4.19 – Exemple à 2 dimensions d'adressage des points physiques par élément identifié via ses coordonnées 2D.

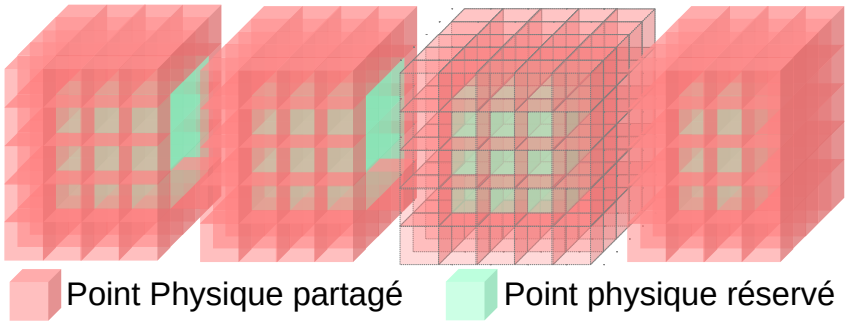

FIGURE 4.20 – Structure régulière permettant l'accès direct aux points physiques

4.5.2 Changement du noyau vers un mode d'accès direct aux points GLL

Les maillages structurés peuvent être traités tels des maillages non structurés. Ainsi, il est plus pratique de traiter tous les maillages comme étant non structurés. Cependant, profiter du caractère structuré d'une majeure partie du maillage peut apporter des gains de performances significatifs. Nous allons détailler les changements opérés dans les optimisations précédentes afin de bénéficier d'un mode d'accès direct aux données physiques.

Évolution du calcul vectoriel lors du passage à l'accès direct des points GLL

Comme vue précédemment, la seule différence liée à l'accès direct réside dans l'accès aux données physiques des points GLL. Dans ce cas présent, les références des points physiques disparaissent de la structure paramétrique des points GLL (Fig. 2.6 sous-section 2.1.3). En effet, celle-ci devient calculable comme on peut le voir avec la figure 4.19. Tout d'abord, les éléments sont disposés en grille régulière Fig. 4.19. Par conséquent, ils doivent être parcourus selon plusieurs boucles imbriquées pour chaque coordonnée. Alors, chaque élément fini 3D est identifié par ses coordonnées « X,Y,Z ». Ainsi, les données des points physiques doivent être structurées en grille tel qu'on peut le voir dans l'illustration 4.21.

En scalaire, l'indice des GLLs est déterminé par la fonction loadGLLref de la figure 4.22. Ainsi, les fonctions loadGLLval et storeGLLval permettent de charger et sauvegarder une valeur physique correspondant à un point GLL d'un élément.

Il est possible de vectoriser sans changer la structure de données des points physiques précédente. Cependant, la vectorisation est simplifiée si l'on entrelace les éléments finis. En effet, le mode d'accès direct vectoriel est plus rapide lorsque les données sont directement chargeables en registre vectoriel. L'illustration 4.23 présente un exemple d'entrelacement des données de points physiques permettant un chargement direct dans des registres vectoriels. Cette technique fait écho à ce qui est présenté dans la sous-section 4.3.1 sur l'entrelacement des paramètres des points GLL.

Les fonctions de calcul d'indice, chargement et sauvegarde des données physiques deviennent vectorielles dans la figure 4.24.

Le reste du code se calcule de la même façon que pour l'accès indirect. Regardons les changements qui opèrent pour le multithreading.

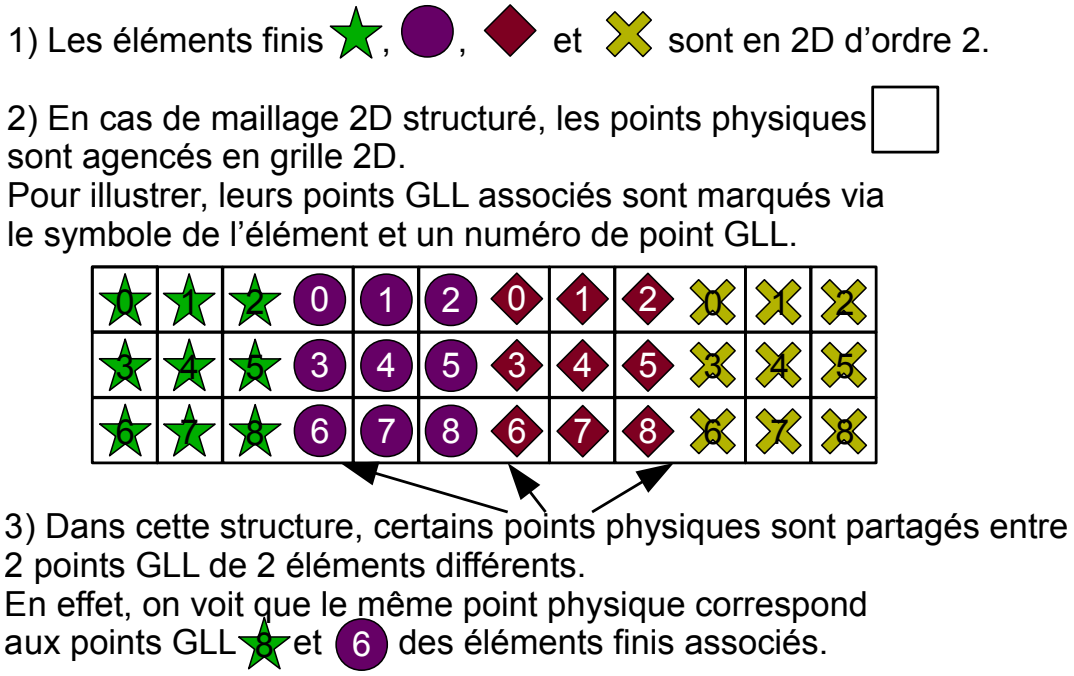

FIGURE 4.21 – Illustration d'une structure en grille 2D de points physiques correspondant avec des points GLL d'éléments finis.

Évolution du multi-threads lors du passage à l'accès direct des points GLL.

Comme la vectorisation, le changement du mode d'accès induit des changements dans le parallélisme multi-threads. À l'instar de la vectorisation, les modifications simplifient le code du coloriage. Tout d'abord, les éléments sont traités en parcourant une grille 3D d'éléments finis. Ainsi, 3 boucles imbriquées parcourent les coordonnées 3D des éléments finis. La formule issue de la figure 4.19 permet de calculer l'adresse mémoire du premier point physique correspondant avec le premier point GLL de chaque élément.

Du fait de la disposition des points physiques en grille 3D, il est directement possible d'accéder à ces derniers depuis l'adresse calculée à partir des coordonnées indicielles du point GLL et de l'élément fini. Ainsi, les éléments sont disposés régulièrement tout en induisant des dépendances avec les points physiques en commun. De fait, le coloriage est immédiat tel qu'on peut le voir dans la figure 4.15. Il en résulte qu'en 3 dimensions, seulement 8 couleurs seront nécessaires. En effet, le schéma du coloriage se répète de 2 vecteurs d'éléments en 2 vecteurs d'éléments dans toutes les directions. Par conséquent, il suffit de parcourir chaque dimension (une par boucle) en avançant de 2 vecteurs pour systématiquement retomber sur des éléments indépendants. Les 2 couleurs par dimension sont induites par le fait de commencer par le premier ou le second vecteur puis d'avancer de 2 en 2.
L'objet de la sous-section qui suit est d'évaluer les gains des modifications induites par ce changement du mode d'accès.

```
//Calcule l'indice des tableaux correspondant à un \gls{gloss_pointGLL}.
inline int loadGLLref(int Ez,int Ey,int Ex,int K,int L,int M)
{
 //W correspond au nombre de points physiques par colonne de la grille 3D.
 //H correspond au nombre de points physiques par ligne de la grille 3D.
 return 3*(W*( H*( 4*Ez+K )+4*Ey+L )+4*Ex+M);
}

//Elle charge la valeur val d'une des composantes des points physiques
//d'un \gls{gloss_pointGLL} aux coordonnées K,L,M
//d'un élément de coordonnées Ez,Ey,Ez.
inline float loadGLLval(int Ez,int Ey,int Ex,int K,int L,int M,int composante)
{
 return displacement[loadGLLref(Ez,Ey,Ex,K,L,M)+composante];
}

//Cette fonction est l'antagoniste de la précédente.
//Elle sauvegarde la valeur val dans une des composantes
//des points physiques d'un \gls{gloss_pointGLL} aux coordonnées K,L,M
//d'un élément de coordonnées Ez,Ey,Ez.
inline void storeGLLval(int Ez,int Ey,int Ex,
                        int K,int L,int M,int composante,float val)
{
 acceleration[loadGLLref(Ez,Ey,Ex,K,L,M)+composante] = val;
}
```

FIGURE 4.22 – Fonctions d'accès direct scalaire aux points physiques des points GLL. Les éléments finis sont dans cet exemple d'ordre 4 et se composent ainsi de 125 points GLL. Les termes displacement et accélération sont deux tableaux contenant les points physiques de la simulation. En accès direct, ces données sont structurées en grille 3D. Les termes Ez,Ey et Ez représentent les coordonnées de l'élément fini. Les termes K,L et M représentent les coordonnées du point GLL cible dans l'élément. Le terme composant fait référence aux composantes X, Y et Z d'un point. Ces composantes s'alternent de façon contiguë en mémoire. Ainsi, les X sont aux indices i*3, les Y aux indices i*3+1 et les Z aux indices i*3+2.

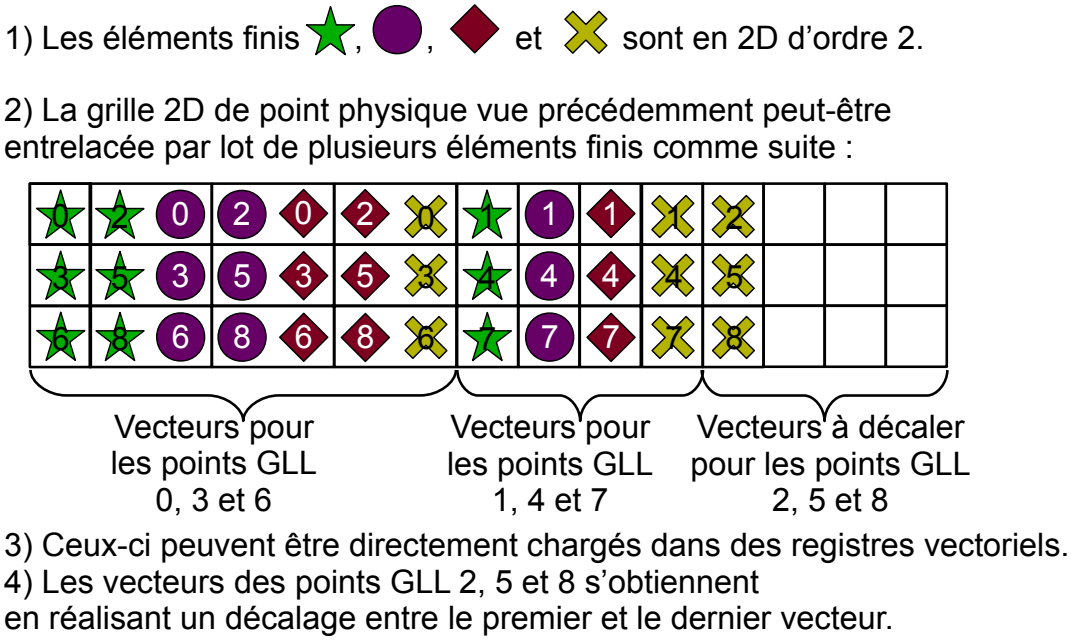

FIGURE 4.23 – Illustration d'une structure en grille 2D de points physiques entrelacés correspondant avec des points GLL d'éléments finis.

4.5.3 Évaluation des performances du mode d'accès direct.

Augmenter la puissance de calcul augmente les besoins en bande passante mémoire. Nos optimisations vues précédemment plafonnent et semblent saturer les accès mémoire. L'objet de cette sous-section est d'analyser les résultats d'expériences afin de mesurer les impacts de ce nouveau mode d'accès aux points physiques.

Impact de la vectorisation

Lors de chaque modification de l'implémentation, nous évaluons l'impact que cela a sur le parallélisme SIMD. Tout d'abord, le tableau 4.5 nous confirme une fois de plus que le SIMD manuel est plus performant que celui laissé aux compilateurs.

SIMD manuel ou automatique	Broadwell Clang5/ICC17	Skylake Clang5/ICC17
automatique	140/122 GFLOPS	135/146 GFLOPS
manuel	344/338 GFLOPS	531/513 GFLOPS

TABLE 4.5 – Performances obtenues à l'ordre 4 avec le maximum de cœurs physiques disponibles pour les implémentations compilées via ICC17 ou Clang5.

De plus, les gains apportés par l'implémentation manuelle du SIMD sont plus élevés avec le mode d'accès direct. En effet, en multithreads sur la plateforme Skylake, le gain moyen de l'accès indirect atteint 2,5. Grâce au mode d'accès direct, le gain moyen est de 3,7 pour cette même plateforme.

```
//Calcul l'indice des tableaux correspondant à un \gls{gloss_pointGLL}.
inline int loadGLLref(int Ez,int Ey,int Ex,int K,int L,int M)
{
 //W correspond au nombre de points physiques par colonne de la grille 3D.
 //H correspond au nombre de points physiques par ligne de la grille 3D.
 //Ils comprennent déàj un multiple de 8.
 return 3*(W*( H*( 4*Ez+K )+4*Ey+L )+4*Ex+8*M);
}

//Elle charge la valeur val d'une des composantes des points physiques
inline __mm256 loadGLLval(int Ez,int Ey,int Ex,
    int K,int L,int M,int composante)
{
 __mm256 vReg = _mm256_load_ps(
    &displacement[loadGLLref(Ez,Ey,Ex,K,L,M)+8*composante]);

 if(M==4)vReg = vShift<-1>(_mm256_load_ps(
    &displacement[loadGLLref(Ez,Ey,Ex,K,L,0)+8*composante]),vReg);
 return vReg;
}

//Cette fonction est l'antagoniste de la précédente.
inline void storeGLLval(int Ez,int Ey,int Ex,
                int K,int L,int M,int composante,float val)
{
 acceleration[loadGLLref(Ez,Ey,Ex,K,L,M)+composante] = val;
}
```

FIGURE 4.24 – Fonctions d'accès direct vectoriel aux points physiques des points GLL. Les éléments finis sont dans cet exemple d'ordre 4 et se composent ainsi de 125 points GLL. Par ailleurs, les registres vectoriels sont de 256bits. Les termes displacement et accélération sont deux tableaux contenant les points physiques de la simulation. En accès direct, ces données sont structurées en grille 3D. Les termes Ez,Ey et Ez représentent les coordonnées de l'élément fini. Les termes K,L et M représentent les coordonnées du point GLL cible dans l'élément. Le terme composant fait référence aux composantes X, Y et Z d'un point. Elles sont contiguës en mémoire par vecteur de 8 valeurs.

Cette approche bénéficie davantage de la puissance de calcul accrue issue du parallélisme SIMD. L'origine de ce bénéfice s'explique ci-dessous avec nos modèles d'analyse.

Analyse théorique de l'impact du mode d'accès sur les performances

Tout d'abord, nous rappelons que l'intensité opérationnelle est plus élevée pour les implémentations dont les accès aux points physiques sont directs par rapport aux accès par indirection. La différence entre ces 2 modes d'accès est d'avantage visible avec la plateforme Skylake.

Comme nous l'avons précédemment observé, le noyau d'origine traite via un accès indirect aux données physiques. La résolution de chaque élément fini implique de charger des points globaux vers les GLLS locales. Bien que les registres vectoriels peuvent être chargés via des indices d'indirection, les indices en questions doivent être chargés depuis la mémoire centrale. L'évaluation du cout des indirections est rendue possible via l'implémentation d'une version de référence accédant directement aux points GLL. Toutefois, cette implémentation dont l'accès est direct peut uniquement traiter des maillages structurés. En revanche, ce mode d'accès du noyau récupère directement les points GLL à partir des coordonnées d'éléments finis sans avoir à passer via une référence indicielle. Par conséquent, son OI est plus élevé. Ainsi, le noyau est moins limité par la mémoire centrale. Il nous reste à le vérifier expérimentalement.

Vérification expérimentale de l'analyse théorique

Les résultats expérimentaux de la figure 4.25 concordent bien avec notre intuition due à un facteur OI plus élevé pour l'accès direct. Ainsi, un accès moins couteux aux données physiques permet à un même calcul d'être réalisé plus vite. Par conséquent, la limitation précédemment observée (plafonnement dans la sous-section précédente) était bien due à la saturation des accès mémoire. Sur la plateforme Skylake, l'écart des performances obtenues entre les 2 modes d'accès est plus marqué. En effet, dans ce cas la version d'accès direct profite des capacités calculatoires accrues par l'AVX-512.
Nous observons à nouveau la courbure du graphique des performances obtenues avec la plateforme Broadwell. Cette courbure témoigne encore une fois de la saturation de la bande passante. Ce comportement pourrait s'expliquer grâce au modèle roofline et souligner que la version de l'ordre 4 est plus susceptible d'être limitée par la bande passante mémoire sur la plateforme Broadwell. En effet, cette plateforme possède une bande passante de 128 GB/s alors que la bande passante de plateforme Skylake est à 192 GB/s. La plateforme Skylake assure un bon passage à l'échelle des deux implémentations. Ceci concorde avec le modèle théorique puisque ces implémentations à l'ordre 4 sont moins limitées par la bande passante mémoire avec la plateforme Skylake.

Nous venons de voir que le mode d'accès aux données joue un rôle essentiel dans l'amélioration des performances. Toujours dans l'optique d'évaluer les comportements avec la mémoire, la sous-section suivante aborde l'impact qu'à l'ordre d'approximation polynomial sur l'OI et ainsi sur les performances.

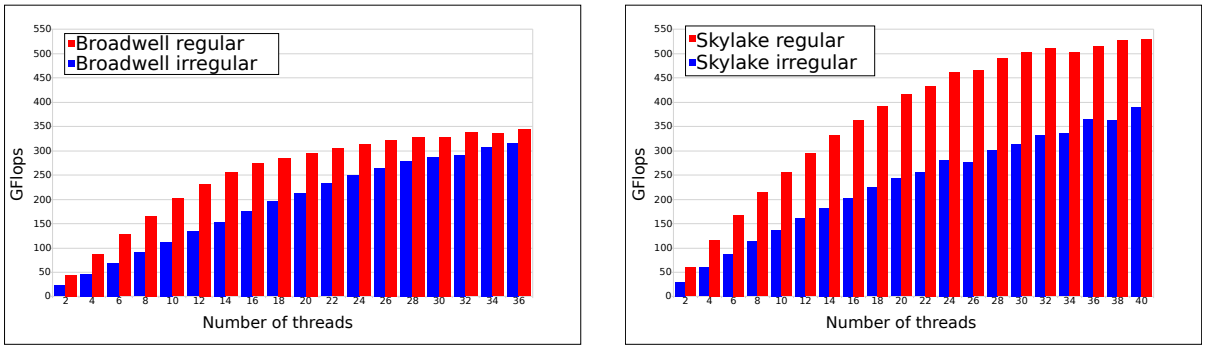

FIGURE 4.25 – Comparaison entre l'accès indirect et direct aux points GLL par le compilateur Clang5 sur les plateformes Broadwell (à gauche) et Skylake (à droite).

4.6 L'ordre d'approximation polynomial

Les optimisations des sous-sections précédentes assurent des gains de performance encourageants. Les expérimentations réalisées montrent que nos axes d'optimisations sont pertinents et concordent avec les modèles d'analyse. L'objet de cette sous-section est d'évaluer le comportement des optimisations précédentes lorsque l'ordre d'approximation des éléments finis évolue.

4.6.1 Analyse théorique de l'impact de l'ordre sur le facteur OI

L'ordre polynomial d'approximation fait varier au cube le nombre de points GLL par élément. Or, la quantité de calcul n'est pas linéaire au nombre de ces points. Par exemple, à l'ordre 2, chaque élément fini se compose de 27 points GLL. Par conséquent, on réalise 7974 opérations flottantes par élément fini alors qu'il y a 48150 opérations à l'ordre 4 pour 125 points GLL. Cela donne environ 295 opérations flottantes pour chaque point point GLL à l'ordre 2 et 385 à l'ordre 4. Nous allons voir en quoi ces changements d'OI impacts les performances et si cela suit nos modèles d'analyse.

4.6.2 Analyse expérimentale de l'impact de l'ordre sur les performances

Les courbes Fig. 4.26 présente les performances obtenues sur la plateforme Skylake par nos implémentations les plus optimisées suivant les 2 modes d'accès aux points GLL pour différents ordres d'approximation. Globalement, on constate bien un impact de l'ordre sur les performances. Cependant, les implémentations de l'accès indirect ne suivent pas notre modèle comme le montre le graphique de gauche Fig. 4.26 contrairement aux implémentations directes sur le graphique de droite Fig. 4.26. En effet, l'ordre 2 est celui dont l'OI est le plus faible et l'ordre 8 celui dont l'OI est le plus élevé. Ainsi, l'ordre 2 devrait être moins performant que l'ordre 8 qui réalise davantage de calculs avec les mêmes données. Afin de clarifier l'origine de ces écarts au modèle, des indicateurs expérimentaux via le logiciel de profilage vTune d'Intel ont été mesurés.

Ces indicateurs sont reportés sous forme de graphiques dans la figure 4.27. Lorsque l'on prêt attention au taux de cycles perdu à attendre une réponse des caches mémoire, on constate pour l'accès indirect que plus l'ordre est élevé et plus on y perd de cycles et donc

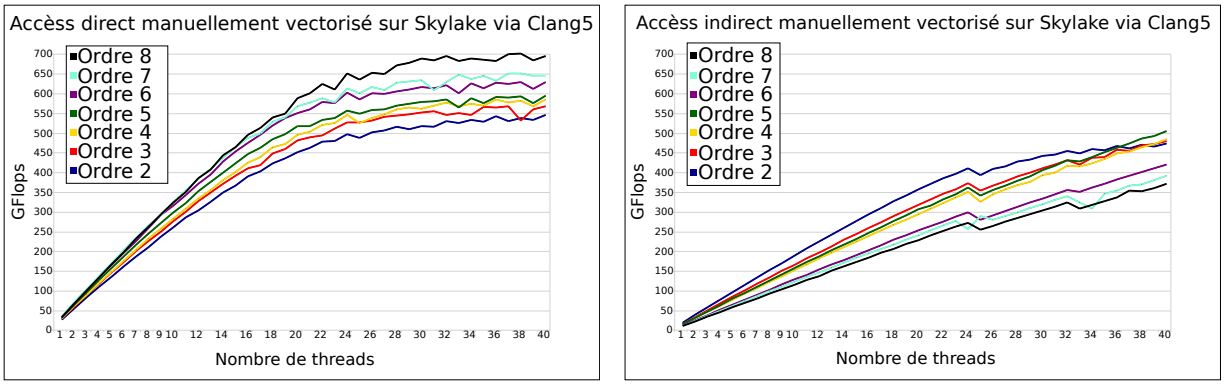

FIGURE 4.26 – Courbes des performances selon différents ordres d'approximation manuellement vectorisé par 16 éléments et compilé avec Clang5 pour la plateforme Skylake.

de temps. En effet, lorsque l'on vectorise par 16 éléments, on se doit de travailler avec 16 éléments sur un même cœur. La situation devient particulièrement plus préoccupante à l'ordre 8 ou 768 points GLL sont nécessaires pour chacun des 16 éléments finis traités en parallèle par un même cœur. On déborde assez rapidement du premier niveau de cache qui ne fait que 32 Kilos octet. En effet, 16 éléments finis représentent environs $64512 * 16$ octets soit 1008 kilo-octets. De plus, le second niveau de cache du cœur Skylake est d'un seul méga-octet. Ainsi, il se retrouve également dépassé. Par conséquent, notre modèle roofline via l'OI ne prend pas en compte ce niveau de détail d'où l'écart.
Les bâtons de la figure 4.27 peuvent paraitre surprenants. En effet, les implémentations de l'accès direct semblent plus limitées par les accès à la mémoire centrale que les implémentations d'accès indirect. Ces bâtons représentent des pourcentages n'atteignant pas les mêmes niveaux de performance. Ces taux de cycles doivent être considérés selon ces niveaux.

Les versions d'accès direct sont limitées par les transferts à la mémoire centrale. En revanche, les implémentations dont l'accès est indirect sont davantage freinées par l'attente des caches mémoires. Ceci explique le fait que les courbes à droite Fig. 4.26 soient peu arrondies suite à l'ajout de cœurs. En effet, le frein dû aux caches mémoires a lieu pour chaque cœur, quel qu'en soit leur nombre. Inversement, les courbes à gauche Fig. 4.26 sont plus arrondies, car le frein de la mémoire centrale s'applique à mesure que l'on ajoute des cœurs.

D'une part, ces derniers résultats corroborent les constats précédemment établis. L'accès direct aux points physiques est plus performant puisqu'il sature moins les accès mémoire. Quelsque soient les modes d'accès, le changement d'ordre a un impact sur les performances. L'évaluation de différents ordres d'approximation laisse apparaitre des comportements expliqués à l'aide du logiciel vTune. Ainsi, on peut dire que la projection d'indicateurs OI sur une roofline reste valable tant que les caches ne sont pas saturées. En effet, l'OI considère que les données du calcul sont immédiatement accédés une fois en cache ce qui en réalité est approximatif. Cependant, tant que les caches ne saturent pas, le modèle reste cohérent avec les observations expérimentales.
D'autres phénomènes peuvent freiner les performances. La sous-section suivante a pour objet d'étude les effets induits par la non-uniformité mémoire (NUMA) de nos architectures.

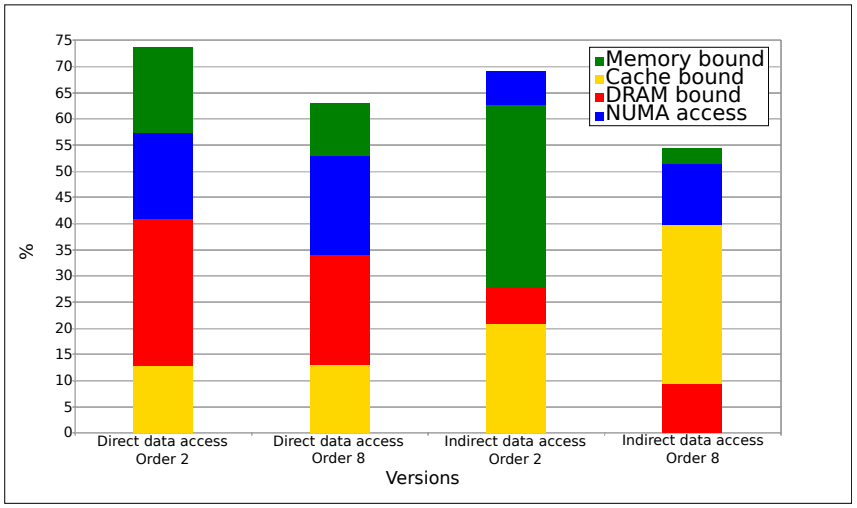

FIGURE 4.27 – Graphique d'indicateurs vTune mesurés depuis la plateforme Skylake des versions d'ordres 2 et 8 manuellement vectorisés par 16 éléments en accès direct et indirect compilés sous Clang5.

4.7 Localité des données en mémoire

La localité des données en mémoire sur les architectures NUMA peut induire des effets impactant les performances. Différents comportements peuvent être observés selon des ressources mémoires limitantes. Ainsi, cette sous-section étudie l'impact de la non-uniformité des accès à la mémoire pour notre noyau d'éléments spectraux.

4.7.1 Le procédé d'évaluation de l'impact NUMA

Dans une architecture NUMA, les nœuds de traitement n'accèdent pas tous de la même façon selon les modules mémoires cible. Les nœuds NUMA forment un couple d'un nœud de traitement et d'un nœud mémoire. Dans un même couple, un nœud de traitement est en accès direct avec le nœud mémoire. Ainsi, selon les nœuds mémoire accédés puis les bus empruntés, les performances de transfère peuvent varier. Dans nos travaux, les threads sont attachés à des cœurs physiques des nœuds NUMA. Lorsque le système alloue de la mémoire à un thread et que ce dernier touche les pages mémoires allouées, ces pages mémoires sont idéalement allouées au nœud mémoire du nœud NUMA où le thread s'exécute. Dans notre noyau, le thread principal se charge d'allouer l'intégralité des données. Par conséquent dans nos expériences, les données sont allouées autant que possible sur le nœud mémoire du nœud NUMA où le thread principal est rattaché. Or, les threads OpenMP du noyau sont réparties entre plusieurs nœuds NUMA disponibles. De fait, le nœud mémoire ayant la totalité des données est alors saturé de demande. De plus, les demandes provenant d'autres nœuds de traitement distant doivent passer via un bus QPI (QPI) ce qui aggrave la saturation. Pour solutionner, il serait préférable d'envisager une localité entre les traitements et les allocations mémoire du domaine de calcul (par nœuds NUMA). Toutefois, dans l'objectif d'avancer pas à pas depuis la version d'origine, nous avons opté pour une autre solution. En effet, lors de nos expériences précédemment

évoquées, nous faisons appel à la commande « numactl -i all "exécutable" » assurant l'allocation alternée des pages mémoire de nos programmes expérimentaux. Ceci permet d'une part de profiter de tous les canaux mémoires disponibles et d'autre part de répartir les accès via les bus QPI.

Une expérience a été menée afin d'évaluer le niveau d'impact suivant la localité des données avec le mode d'accès indirect aux points physiques. Rappelons que la plateforme Skylake a 2 processeurs comprenant chacun 1 nœud NUMA de 20 cœurs (soient au total 2 nœuds NUMA). La plateforme Broadwell se compose de 2 processeurs comprenant chacun 2 nœuds NUMA de 9 cœurs (soient au total 4 nœuds NUMA). Le principe de cette expérience consiste à fixer un thread sur chaque cœur du nœud 0 et d'expérimenter différentes stratégies d'allocation mémoire.

4.7.2 Impact de la non-uniformité des accès mémoire

La figure 4.28 réunit les résultats des plateformes Broadwell et Skylake. La colonne mem0 correspond à l'expérimentation où les threads et les données sont dans le même nœud NUMA. On y atteint les meilleures performances, car la localité est maximale. Les autres colonnes « mem1, mem2, mem3 » sont issues de l'allocation de toutes les données dans les autres nœuds mémoires (allocations distantes). On y constate une chute des performances de près de la moitié. Par ailleurs, la commande numactl -HW nous affiche la matrice des distances entre les nœuds. Elle nous apprend que la distance est double entre un nœud de traitement et un nœud mémoire n'appartenant pas au même nœud NUMA. D'où la chute expérimentalement observée des performances des colonnes « mem1, mem2 et mem3 ». Enfin, la colonne « All » est issue d'une répartition des données sur plusieurs nœuds mémoires (commande numactl -i all). Cette expérience présente de bonnes performances proches de celles obtenues avec une localité optimale (colonne « mem0 »).

Ces résultats soulignent la nécessité de se soucier de la localité des données. Notre approche par répartition des pages mémoire sur les nœuds présente des résultats acceptables. L'amélioration au niveau de l'optimisation noyau de cet aspect NUMA bien que complexe mérite des recherches plus poussées.
Les expériences sur l'effet induit par l'aspect NUMA ont été réalisées pour l'accès indirect aux points physiques pour plusieurs ordres d'approximation polynomiale. Ces expériences corroborent les constats des sous-sections précédentes. En effet, lorsque les threads et les données sont localisés dans le même nœud NUMA, la mémoire centrale n'est pas le facteur le plus limitant (les caches mémoires sont limitants). Par conséquent, le facteur OI n'est pas pertinent dans ce cas. Ainsi, plus l'ordre est faible et plus les performances sont élevées. En revanche, lors que la localité mémoire est distante, la mémoire centrale devient limitante. De fait, les ordres les plus élevés présentent de meilleures performances.
L'analyse des résultats expérimentaux étant terminée, nous passons à la conclusion de ce chapitre.

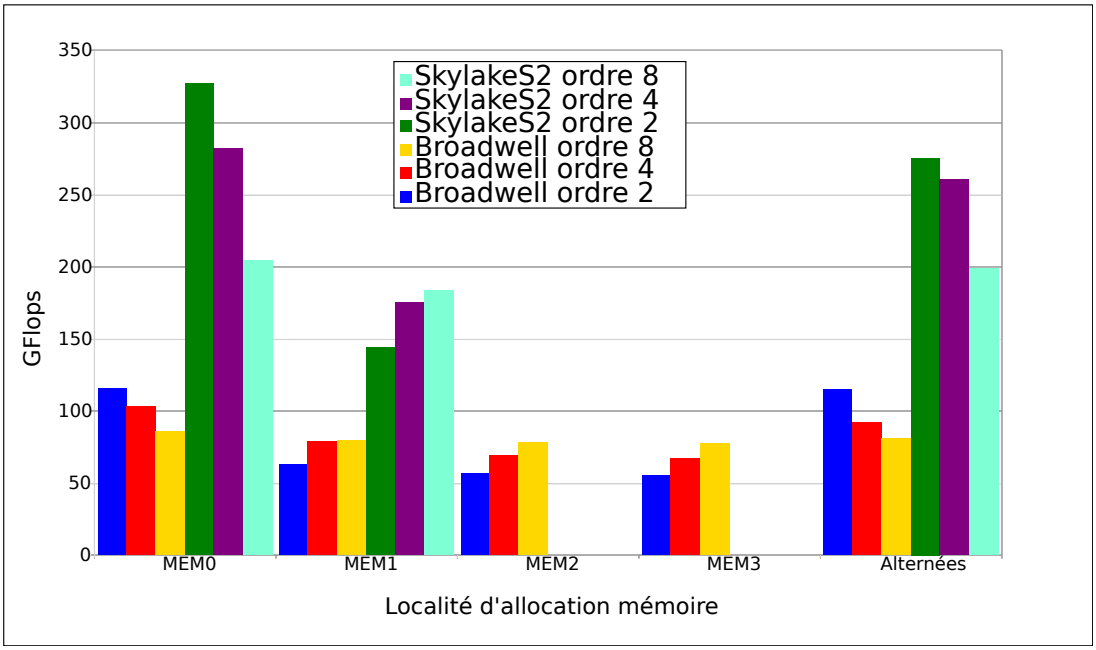

FIGURE 4.28 – Performances des threads attachés au nœud NUMA 0 selon différentes stratégies d'allocation mémoire des noyaux d'accès indirect aux points physiques sur les plateformes Broadwell et Skylake.

4.8 Conclusion

Dans ce chapitre, nous avons identifié des axes d'amélioration du code original d'Efispec3D . En effet, la courbe du passage à l'échelle en intra-nœuds est initialement insatisfaisante. L'optimisation des méthodes numériques les plus populaires pour les architectures parallèles est un effort continu. Bien que les principales caractéristiques de ces noyaux soient déjà bien connues, trouver les meilleurs algorithmes ou implémentations sur chaque architecture peut être considéré comme une quête.

Premièrement, contrairement aux bons résultats sur les architectures distribuées, l'implémentation MPI standard ne permet pas un passage à l'échelle intra-nœuds satisfaisant. En effet, nous avons analysé le passage à l'échelle en distribué des algorithmes parallèles sur la machine petascale Shaheen 2 et nous avons discuté de la performance obtenue avec la parallélisation de la phase d'assemblage. Deuxièmement, l'accès indirect aux points physiques nécessite la collecte de données à partir de différents emplacements de mémoire, ce qui empêche une bonne utilisation de la bande passante. Enfin, l'auto-vectorisation est à peine réalisée par les compilateurs, même en ajoutant des pragmas SIMD dans le code ([57]). Pour chacune de ces limitations, nous avons proposé une solution et une implémentation dans un noyau extrait d'Efispec3D.

Comme expliqué dans la sous-section 4.2.2, nous comparons nos résultats avec l'ORM, mais d'autres investigations peuvent être menées avec le CARM et le LARM pour identifier clairement les facteurs limitants. Nous étudions également la corrélation entre l'intensité opérationnelle et la performance réelle sur deux plates-formes Intel à deux sockets.

Sur la base du modèle théorique, nous avons souligné l'impact des modèles d'accès aux données qui représentent un goulot d'étranglement majeur par rapport à la performance attendue. De plus, à partir des résultats d'expériences, nous avons démontré l'efficacité de nos améliorations ainsi que leurs limites sur les architectures actuelles. L'un des principaux résultats de cette étude est également l'énorme complexité de la prédiction des performances, en particulier sur les puces disponibles qui combinent plusieurs niveaux de parallélisme et de hiérarchies de cache. Nos travaux soulignent certains des aspects clés de la méthode des éléments finis en exploitant le modèle populaire de performance théorique qu'est le modèle roofline.

Des gains significatifs ont été observés sur les plates-formes avec de l'AVX-2 et de l'AVX-512. Nous avons amélioré les performances de calcul de la phase d'assemblage par le parallélisme SIMD d'un facteur de 2,3 sur architecture Broadwell et de 2,9 sur architecture Skylake dans un contexte multi-cœurs. De plus, les résultats obtenus concordent avec les performances théoriques données par les modèles rooflines. D'autres axes d'amélioration pourraient être envisagés puisque nous n'atteignons pas la limite supérieure des roolines. Par exemple, nous prévoyons de mettre en œuvre une meilleure stratégie pour les threads et la localité mémoire sur les architectures NUMA. De plus, la mise en oeuvre de ces optimisations dans l'application Efispec3D complète est une autre étape majeure de notre travail. Cela représente un défi, car la performance observée au niveau de la mini application pourrait être considérablement réduite lorsque l'on considère l'application complète.

De toute évidence, les optimisations proposées bénéficieraient d'une intégration dans un cadre de haut niveau afin de faciliter leur mise en oeuvre dans un grand nombre d'applications scientifiques. Nous pensons également que des efforts significatifs devraient être faits au niveau de la modélisation des performances, probablement avec une utilisation plus large des outils bas niveau, afin de mettre à jour les applications clés sur les processeurs émergents (vtune, PAPI, MAQAO).
Enfin, un Domain Specific Language (DSL) pourrait aider les physiciens à se concentrer uniquement sur la description de leurs calculs, rendant ces optimisations plus accessibles.

Chapitre 5

Conclusion

Les évolutions conjointes des architectures et des modèles de simulation se sont en partie mutuellement façonnées. D'un côté, les architectures ont évolué afin de proposer de plus en plus de ressources améliorées dont certaines sont particulièrement adaptées aux besoins D'un autre côté, les algorithmes des simulations évoluent de façon à mieux correspondre avec le fonctionnement de ces architectures. En effet, ces architectures opèrent sur des domaines discrets. Ainsi, la résolution par discrétisation et en quelque sorte une adaptation des EDP au monde discret des ordinateurs.

Le réseau des acteurs intervenant sur les sujets de la simulation numérique se compose de différents profils. Certains sont concepteurs d'architecture matérielle, logicielle et d'autres encore sont utilisateurs. Parfois, les utilisateurs deviennent aussi des développeurs et architectes improvisés. Il n'est pas simple pour une personne de cumuler plusieurs profils tout en assurant l'expertise nécessaire dans tous ses domaines. Il faut à la fois bien comprendre le fonctionnement des architectures matérielles ainsi que celui des noyaux de calcul afin de les rapprocher d'un optimal. Parvenir à marier les deux est une tâche qui incombe à des personnes expérimentées sur ce sujet. De plus, les personnes expérimentées le sont pour certaines simulations et non pour toutes. De tout temps, chaque acteur porte qu'une partie des connaissances de différents sujets et de tout temps le chalenge est de parvenir à les capitaliser. Ainsi, il est possible d'optimiser l'exécution de calculs pour des architectures matérielles, mais cela prend du temps à chacun de devoir le redécouvrir et le refaire même à l'aide d'articles précis. L'idée est donc de structurer et canaliser ces différentes expertises afin de les rendre utilisables par des non-experts.

Dans cette thèse, nous avons en un premier temps cherché à optimiser des noyaux selon différentes approches visant à mieux exploiter les ressources matérielles d'architectures x86 modernes. Les noyaux étudiés sont largement employés dans des applications (Onde3D, Efispec3D, SPECFM). L'idée était de toucher des noyaux représentatifs d'un large spectre d'applications réelles. Ainsi, leurs optimisations impactent d'autant d'applications y compris la possibilité d'extrapoler ces optimisations à d'autres noyaux.

Nous avons donc commencé notre étude par des noyaux de type stencil. Ainsi, les noyaux explicites non pondérés 7-point 27-point servent à évaluer nos approches performatives. On a tout d'abord montré que la vectorisation automatique des compilateurs

n'est pas efficace. En effet, la vectorisation laissée aux compilateurs permet d'atteindre un gain de 6. Alors que, la vectorisation réalisée explicitement à la main via des intrinsics permet d'atteindre un gain de 9. Un processeur contenant plusieurs cœurs, nous avons abordé l'optimisation multi-cœurs. L'environnement d'exécution multithreads OpenMP a donc été utilisé afin de bénéficier du caractère partagé de la mémoire centrale. On notera que l'ajout de cœurs (SPMD) de calcul peut être concurrent à l'optimisation vectorielle (SIMD) face à la limite de la bande passante mémoire. Ainsi, l'ajout de cœurs peut compenser une mauvaise vectorisation face à la limite de la bande passante mémoire. Toutefois, dans un contexte multithreads, on remarque que les compilateurs vectorisent parfois moins bien étant donné les transformations du code par OpenMP. Par exemple, le compilateur Clang3.8 déclare à la compilation ne pas prendre le risque d'optimiser étant donné les transformations multithreads opérées par OpenMP. Il faut bien comprendre que les implémentations d'OpenMP diffèrent selon le compilateur et sa version.

En cumulant ces 2 optimisations des ressources de calcul, les performances de nos 2 noyaux atteignent une limite due à la bande passante mémoire. Ainsi, nous avons proposé l'indicateur RI qui est caractéristique du potentiel d'utilisation de chaque octet à l'échelle d'une itération pour un stencil donné. Nous avons recoupé le modèle roofline avec ce facteur ce qui nous a donné une performance maximale potentielle en cas de parfaite réutilisation instantanée des octets d'une itération stencil. Afin de nous rapprocher de ce taux maximal de réutilisation à l'échelle de l'itération, nous avons mis en œuvre une approche par tuilage spatial. Les différentes performances ainsi obtenues suivent assez bien le facteur RI lorsque l'on compare le rapport des performances entre le stencil 27-point et 7-point avec le rapport de leur RI de 3,86.

La réutilisation des données à l'échelle de l'itération reste limitée par le facteur RI Ainsi, nous avons exploré des techniques de tuilage temporelles. Notre approche consiste à composer le stencil 7-point avec lui-même pour obtenir le stencil 25-point. Ainsi une itération du stencil 25-point correspond à 2 itérations du stencil 7-point. Or, le facteur RI du stencil 25-point est de 6,25. Par conséquent, les performances doivent se rapprocher de celle du stencil 27-point (RI à 6,75) qui sont 3,8 fois plus élevées que celle du stencil 7-point (RI à 1,75). Or, les performances obtenues avec le stencil 25-point sont bien 3 fois plus élevées que celles du stencil 7-point. Cependant, le gain en temps est de 1,6 du fait que le stencil 25-point requiert 2 fois plus de calculs pour parvenir aux mêmes résultats qu'avec 2 itérations du stencil 7-point.

Nos approches ont été comparées avec le DSL Pochoir. En termes de performances, on arrive au même niveau voir au-delà avec la composition du stencil 7-point. En termes plus techniques, le DSL Pochoir génère un code complexe dont la vectorisation est laissée au compilateur. Or, le compilateur ICC17 ne parvient pas à vectoriser efficacement ce code complexe. Le compilateur Pochoir a le mérite d'être générique pour optimiser les stencils y compris les stencils implicites. Ainsi, l'idée de capitaliser des approches d'optimisations via un DSL y est concrétisée.

Une autre approche de capitalisation a été mise en œuvre avec le framework Girih [1].

1. Framework Girih disponible à l'adresse https://github.com/ecrc/girih.

L'un des avantages de cette approche est de faire appel à OpenMP pour l'intra-nœud et à MPI pour l'extra-nœuds (plusieurs machines en réseau). En effet, OpenMP est plus adaptée à l'intra-nœud via sa gestion par threads bénéficiant de la mémoire matériellement partagée. Alors que MPI est fondé à l'origine sur la synchronisation de processus via des transferts explicites. Enfin, on statuera en un premier temps que pour les stencils, des approches de type DSL, framework, squelette peuvent constituer des solutions à destination de non-experts. Des travaux ont été menés dans ce sens [29, 15].

En partant des premiers résultats précédents réalisés sur des plateformes x86 modernes, nous avons également étudié la sous-catégorie de noyaux issue de la méthode des éléments finis spectraux par assemblage. Nous sommes partis du noyau de calcul des forces internes traduit en C++ depuis l'application Efispec3D initialement développée en Fortran. Le parallélisme MPI de ce noyau était initialement dépourvu de vectorisation ainsi que de synchronisation via de la mémoire partagée. Ce qui explique qu'il était meilleur au passage à l'échelle extra-nœuds (plusieurs machines en réseau) qu'intra-nœuds (plusieurs cœurs d'une même machine partageant de la mémoire).

On a tout d'abord vérifié les gains apportés via notre approche vectorielle face à la vectorisation des compilateurs. Notre stratégie de vectorisation est orientée parallélisme synchronisé de plusieurs éléments. Ainsi, notre vectorisation permet le calcul en parallèle de plusieurs éléments finis pour chaque cœur. Il en résulte un gain de 4,5 sur la plateforme Skylake par rapport à une implémentation séquentielle où la vectorisation est laissée au compilateur Clang5. Ainsi, on constate une fois encore que la vectorisation automatique n'est pas efficace. On s'attaque alors au cumul de cette optimisation avec un parallélisme multithreads. L'environnement d'exécution OpenMP est bien indiqué en raison de la régularité des calculs à partager. On est rapidement confronté à une limitation des performances lors du passage à l'échelle en intra-nœuds. En effet, l'accroissement de la puissance de calcul induit une demande grandissante des transferts mémoires. Or, lorsque cette demande dépasse les capacités de la plateforme, le niveau de performance sature.

La nature des maillages employés en géosciences présentes des zones structurées assez étendues. Ainsi, nous avons étudié les optimisations permises lorsque l'on considère un maillage structuré. Avec cette considération, les données physiques peuvent être disposées en grille 3D. Ainsi, en partant des coordonnées de l'élément fini, on détermine les adresses mémoires des points physiques. De plus, il est aussi possible d'entrelacer les valeurs de ces points physiques afin de les rendre directement chargeables dans des registres vectoriels. D'un côté, on économise le chargement des indices associant les points GLL aux points physiques. De l'autre, le chargement en registre vectoriel est plus direct. Ceci nous permet d'augmenter par 1,9 les performances maximales atteintes. Par conséquent, l'application traitant avec des zones non structurées pourrait traiter séparément ces zones structurées afin de les calculer 1,6 fois plus vite.

La différence majeure des catégories de noyaux que nous étudions se situe au niveau du nombre de calcul et données impliquées par unité à traiter (une cellule pour les stencils et un élément pour les éléments finis). En effet, nos noyaux stencil sont petits et représente entre 6 et 26 additions utilisant entre 7 et 27 valeurs flottantes. Ainsi, l'indice RI est employé pour les stencils, car il est lié au parcours spatial de ces derniers supposé

réutiliser au maximum les données en mémoire cache. Alors que, le calcul d'un élément fini spectral d'ordre 4 représente 48150 opérations arithmétiques pour le chargement de 11120 octets. Par conséquent, nous ne sommes plus à la même échelle. La situation est amplifiée lorsque des vecteurs de 8 ou 16 éléments sont traités en même temps. De fait, on considère qu'indépendamment de leur parcours, chaque élément fini spectral sollicite à son échelle les 2 premiers niveaux de mémoire cache de nos plateformes. Il en découle dans cette thèse que l'indice d'Intensité Opérationnelle (OI) est employé pour exprimer l'intensité des calculs des noyaux éléments finis spectraux. En effet, l'indice OI se définit par l'intensité arithmétique aux données présentes en mémoire cache en faisant abstraction des bandes passantes de ces mémoires caches.

L'estimation de l'Intensité Opérationnelle dépend du noyau considéré. En effet, cet indice change selon l'ordre d'approximation et l'accès aux données physiques. Nous avons vu qu'en traitant avec des maillages structurés, il est possible de rendre les valeurs physiques directement accessibles sans avoir à passer via de l'indirection. Différentes variantes ont donc été placées sur le graphique de la roofline afin d'avoir une idée des performances accessibles pour chaque. Les expériences confirment que la simplification des accès aux données physiques permet d'augmenter les performances atteintes. Ainsi, il est possible d'augmenter de 1,2 la performance déjà élevée et d'atteindre les 580 GFLOPS à l'ordre 4 sur la plateforme Skylake.

Le modèle roofline couplé à l'indice OI prédit une augmentation des performances à chaque augmentation de l'ordre polynomial d'approximation. En effet, le nombre de calculs augmente plus que le nombre de données nécessaires donc l'OI augmente. Par conséquent, la bande passante est moins limitante à mesure que l'ordre est grand. Nous avons expérimentalement vérifié ces prédictions. Toutes les expériences ne suivent pas les prédictions pour les versions vectorisées ou l'accès aux données physiques est indirect. En effet, du fait du besoin accru induit par l'indirection, les mémoires caches sont saturées à l'échelle de l'élément à mesure que l'on augmente l'ordre d'approximation. Ainsi, les performances décroissent ce qui est contradictoire avec les prédictions du modèle roofline. L'artefact a donc été étudié. Il en résulte que le facteur OI n'est plus valable dès lors que les caches sont débordées par le calcul d'un même vecteur éléments. En effet, le facteur OI fait abstraction des mémoires caches sans considérer leurs bandes passantes. Lorsque les caches constituent une source de lenteurs à l'échelle du vecteur d'éléments, alors le modèle décroche de la réalité expérimentale. Les expériences où l'accès aux données physiques est direct réalisent bien une augmentation des performances à mesure que l'ordre augmente.

Le phénomène d'inversion des courbes précédemment évoqué se retrouve lorsque l'on expérimente des changements de localité mémoire. En effet, si les données de la simulation sont allouées sur un nœud NUMA distant du nœud de calcul alors la bande passante mémoire devient le facteur le plus limitant et les effets de caches précédents sont masqués (même avec l'implémentation indirecte). De plus, le logiciel vTune a permis de mettre en évidence les facteurs limitants chaque expérience. Il en résulte que lorsque le calcul local d'un vecteur d'éléments finis ne sature pas les mémoires caches (cas de l'accès direct), alors la principale source de ralentissement devient la bande passante mémoire. Avant d'être limitées par celle-ci, les performances ont augmenté d'au moins 50%.

Les approches d'optimisations de ces noyaux sont encourageantes. Toute foi, une optimisation des localités mémoire mériterait un approfondissement. En l'état, les optimisations permettent de bénéficier davantage des plateformes par rapport à une implémentation séquentielle où les optimisations sont laissées au compilateur. Les espoirs d'aller encore plus loin nécessitent d'envisager des solutions plus complexes. Par conséquent, la mise en œuvre des optimisations évaluées dans cette thèse reste assez simple et ouvre la voie à une intégration plus générale. Il reste à coupler de tels travaux avec ceux sur les DSL, squelette algorithmique ou solution en framework. Ces approches pourraient ainsi être capitalisées à destination de différentes cibles et rendues exploitables par des utilisateurs non experts du HPC.

Bibliographie

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack J Dongarra, Christopher Earl, Joël Falcou, Azzam Haidar, Ian Karlin, Tzanio V Kolev, Ian Masliah, and Stanimire Tomov. High-performance Tensor Contractions for GPUs. In *International Conference on Computational Science 2016 (ICCS 2016)*, volume 80, pages 108–118, 2016.

[2] J.E Akin and M Singh. Object-oriented fortran 90 p-adaptive finite element method. *Advances in Engineering Software*, 33(7) :461 – 468, 2002. Engineering Computational Technology and Computational Structures Technology.

[3] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 291–298, Dec 2010.

[4] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*, pages 772–784, Delft, The Netherlands, August 2009.

[5] Krzysztof Banaś, Filip Krużel, and Jan Bielański. Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering*, 305 :845, 2016.

[6] M. Berekmeri, D. Serrano, S. Bouchenak, N. Marchand, and B. Robu. A control approach for performance of big data systems. *IFAC Proceedings Volumes*, 47(3) :152 – 157, 2014. 19th IFAC World Congress.

[7] Zubair Wadood Bhatti, Roel Wuyts, Pascal Costanza, Davy Preuveneers, and Yolande Berbers. Efficient synchronization for stencil computations using dynamic task graphs. *Procedia Computer Science*, 18 :2428 – 2431, 2013. 2013 International Conference on Computational Science.

[8] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu : Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3) :917–924, July 2003.

[9] Leonardo Brenner, Paulo Fernandes, Jean-Michel Fourneau, and Brigitte Plateau. Modelling grid5000 point availability with san. *Electronic Notes in Theoretical Computer Science*, 232 :165 – 178, 2009. Proceedings of the Third International Workshop on the Practical Application of Stochastic Modelling (PASM 2008).

[10] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa Tikir, David Michéa, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62 thou-

sand processor cores. In *Proceedings of the SC'08 ACM/IEEE conference on Supercomputing*, pages 60 :1–60 :11, Austin, Texas, USA, November 2008. IEEE Press.

[11] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jego. Mipp : a portable c++ simd wrapper and its use for error correction coding in 5g standard. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8. ACM, 02 2018.

[12] Márcio Castro, Emilio Francesquini, Fabrice Dupros, Hideo Aochi, Philippe O. A. Navaux, and Jean-François Méhaut. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54 :108–120, 2016.

[13] Cris Cecka, Adrian J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5) :640–669, 2011.

[14] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Comput. Sci.*, 26(3-4) :205–210, June 2011.

[15] Hélène Coullon, Jose-Maria Fullana, Pierre-Yves Lagrée, Sébastien Limet, and Xiaofei Wang. Blood flow arterial network simulation with the implicit parallelism library skelgis. *Procedia Computer Science*, 29 :102 – 112, 2014. 2014 International Conference on Computational Science.

[16] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[17] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1) :129–159, 2009.

[18] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Austin, Texas, USA, 2008.

[19] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *In In Proc. iWAPT2009 : The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[20] Florent De Martin. Verification of a spectral-element method code for the southern california earthquake center loh.3 viscoelastic case. *Bulletin of the Seismological Society of America*, 101(6) :2855–2865, 2011.

[21] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa. Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model. In *High Performance Computing systems - Performance Modeling, Benchmarking, and Simulation - 8th International Workshop, PMBS 2017*, volume 10724 of *Lecture Notes in Computer Science*, pages 91–113, Denver (CO), United States, November 2017. Springer.

[22] Hikmet Dursun, Ken-Ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A Multilevel Parallelization Framework for High-Order Stencil Computations. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*, pages 642–653, Delft, The Netherlands, August 2009.

[23] Charbel Farhat and Luis Crivelli. A general approach to nonlinear fe computations on shared-memory multiprocessors. *Computer Methods in Applied Mechanics and Engineering*, 72(2) :153 – 171, 1989.

[24] P. F. Fischer and E. M. Rønquist. Spectral-element methods for large scale parallel Navier-Stokes calculations. *Computer Methods in Applied Mechanics and Engineering*, 116 :69–76, 1994.

[25] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[26] Dominik Göddeke, Dimitri Komatitsch, Markus Geveler, Dirk Ribbrock, Nikola Rajovic, Nikola Puzovic, and Alex Ramírez. Energy efficiency vs. performance of the numerical solution of pdes : An application study on a low-power arm-based cluster. *J. Comput. Physics*, 237 :132–150, 2013.

[27] Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. Reproducible stencil compiler benchmarks using prova ! *Future Generation Computer Systems*, 2018.

[28] Arne Hendricks, Thomas Heller, Andreas Schäfer, Max Kasparek, and Dietmar Fey. Evaluating performance and energy-efficiency of a parallel signal correlation algorithm on current multi and manycore architectures. *Procedia Computer Science*, 80 :1566 – 1576, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[29] Coullon Hélène, Le Minh-Hoang, and Limet Sébastien. Parallelization of shallow-water equations with the algorithmic skeleton library skelgis. *Procedia Computer Science*, 18 :591 – 600, 2013. 2013 International Conference on Computational Science.

[30] Tsuyoshi Ichimura, Kohei Fujita, Pher Errol Balde Quinay, Lalith Maddegedara, Muneo Hori, Seizo Tanaka, Yoshihisa Shizawa, Hiroshi Kobayashi, and Kazuo Minami. Implicit nonlinear wave simulation with 1.08t dof and 0.270t unstructured finite elements to enhance comprehensive earthquake simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 4 :1–4 :12, New York, NY, USA, 2015. ACM.

[31] A. Ilic, F. Pratas, and L. Sousa. Cache-aware roofline model : Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1) :21–24, Jan 2014.

[32] Emmanuel Jeannot, Yvan Fournier, and Benjamin Lorendeau. Experimenting task-based runtimes on a legacy computational fluid dynamics code with unstructured meshes. *Computers & Fluids*, 2018.

[33] Sylvain Jubertie, Fabrice Dupros, and Florent De Martin. Vectorization of a spectral finite-element numerical kernel. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, pages 8 :1–8 :7, New York, NY, USA, 2018. ACM.

[34] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1) :359–392, 1998.

[35] D. Komatitsch, J. Ritsema, and J. Tromp. The spectral-element method, Beowulf computing, and global seismology. *Science*, 298(5599) :1737–1742, 2002.

[36] D. Komatitsch and J. Tromp. Spectral-element simulations of global seismic wave propagation-I. Validation. *Geophysical Journal International*, 149(2) :390–412, 2002.

[37] Dimitri Komatitsch, Jesús Labarta, and David Michéa. A simulation of seismic wave propagation at high resolution in the inner core of the earth on 2166 processors of marenostrum. In José M. Laginha M. Palma, Patrick R. Amestoy, Michel Daydé, Marta Mattoso, and João Correia Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2008 : 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers*, pages 364–377, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[38] Marcin Krotkiewski and Marcin Dabrowski. Efficient 3d stencil computations using {CUDA}. *Parallel Computing*, 39(10) :533 – 548, 2013.

[39] Filip Krużel and Krzysztof Banaś. Vectorized opencl implementation of numerical integration for higher order finite elements. *Computers & Mathematics with Applications*, 66(10) :2031, 2013. ICNC-FSKD 2012.

[40] João V.F. Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Maillard. Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures. *Parallel Computing*, 44 :37 – 52, 2015.

[41] Y. Maday and A. T. Patera. Spectral element methods for the incompressible navier-stokes equations. *State of the art survey in computational mechanics*, pages 71–143, 1989.

[42] Tareq M. Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David E. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM J. Scientific Computing*, 37(4), 2015.

[43] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Improving performance via mini-applications. 2009.

[44] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1) :80–97, 2012.

[45] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 898–907, July 2017.

[46] Francisco Martínez-Martínez, María J. Rupérez-Moreno, Marcelino Martínez-Sober, J. A. Solves Llorens, Delia Lorente, Antonio J. Serrano, Sandra Martínez-Sanchis, Carlos Monserrat Aranda, and José David Martín-Guerrero. A finite element-based machine learning approach for modeling the mechanical behavior of the breast tissues under compression in real-time. *Comp. in Bio. and Med.*, 90 :116–124, 2017.

[47] J. McCalpin. Sustainable memory bandwidth in high-performance computers. Technical report, Department of Computer Science School of Engineering and Applied Science University of Virginia, 1995.

[48] Takayuki Muranushi and Junichiro Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51 :1303 – 1312, 2015.

[49] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117 :292 – 302, 2018.

[50] A. T. Patera. A spectral element method for fluid dynamics : laminar flow in a channel expansion. *J. Comput. Phys.*, 54 :468–488, 1984.

[51] J. Reinders and J. Jeffers. *High Performance Parallelism Pearls : Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2014.

[52] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch, Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In Jeffrey K. Hollingsworth, editor, *Proceedings of the ACM / IEEE Supercomputing SC'2012 conference*, page article n 38, Salt Lake City, United States, November 2012. IEEE Computer Society Press. ISBN : 978-1-4673-0804-5.

[53] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *SC'00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 32–38, Dallas, United States, 2000.

[54] D. Roten, Y. Cui, K. B. Olsen, S. M. Day, K. Withers, W. H. Savran, P. Wang, and D. Mu. High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers. In *SC'16 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 957–968, Nov 2016.

[55] Andreas Schafer and Dietmar Fey. High performance stencil code algorithms for gpgpus. *Procedia Computer Science*, 4 :2027 – 2036, 2011.

[56] A. Shterenlikht, L. Margetts, and L. Cebamanos. Modelling fracture in heterogeneous materials on hpc systems using a hybrid mpi/fortran coarray multi-scale cafe framework. *Advances in Engineering Software*, 2018.

[57] Gauthier Sornet, Fabrice Dupros, and Sylvain Jubertie. A multi-level optimization strategy to improve the performance of stencil computation. *Procedia Computer Science*, 108 :1083 – 1092, 2017. International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

[58] Gauthier Sornet, Sylvain Jubertie, Fabrice Dupros, Florent De Martin, Philippe Thierry, and Sébastien Limet. Data-layout reorganization for an efficient intra-node assembly of a Spectral Finite-Element Method. In *PDP2018*, Cambridge UK, United Kingdom, March 2018.

[59] Gauthier Sornet, Sylvain Jubertie, Fabrice Dupros, Florent De Martin, and Sebastien Limet. Performance analysis of simd vectorization of high-order finite-element kernels. *HPCS 2018*, 2018.

[60] Paulo Souza, Leonardo Borges, Cedric Andreolli, and Philippe Thierry. Chapter 24 - portable explicit vectorization intrinsics. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls*, pages 463 – 485. Morgan Kaufmann, Boston, 2015.

[61] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[62] Loïc Thébault, Eric Petit, Marc Tchiboukdjian, Quang Dinh, and William Jalby. Divide and conquer parallelization of finite element method assembly. In *Parallel Computing : Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 753–762, 2013.

[63] Josh Tobin, Alexander Breuer, Alexander Heinecke, Charles Yount, and Yifeng Cui. Accelerating seismic simulations using the intel xeon phi knights landing processor. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, pages 139–157, Cham, 2017. Springer International Publishing.

[64] Seiji Tsuboi, Kazuto Ando, Takayuki Miyoshi, Daniel Peter, Dimitri Komatitsch, and Jeroen Tromp. A 1.8 trillion degrees-of-freedom, 1.24 petaflops global seismic wave simulation on the K computer. *IJHPCA*, 30(4) :411–422, 2016.

[65] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. BOAST : A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *International Journal of High Performance Computing Applications*, 32(1) :28–44, January 2018.

[66] Alexander A. Visheratin, Mikhail Melnik, Denis Nasonov, Nikolay Butakov, and Alexander V. Boukhanovsky. Hybrid scheduling algorithm in early warning systems. *Future Generation Computer Systems*, 79 :630 – 642, 2018.

[67] Samuel Williams, Andrew Waterman, and David Patterson. Roofline : An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4) :65–76, April 2009.

[68] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. An extensive empirical study on c++ concurrency constructs. *Information and Software Technology*, 76 :1 – 18, 2016.

[69] Zifeng Yuan and Jacob Fish. Nonlinear multiphysics finite element code architecture in object oriented fortran environment. *Finite Elem. Anal. Des.*, 99(C) :1–15, July 2015.

[70] Loïc Zuchowski, Michael Brun, and Florent De Martin. Co-simulation coupling spectral/finite elements for 3d soil/structure interaction problems. *Comptes Rendus Mécanique*, 346(5) :408 – 422, 2018.

Glossaire

AI l'Intensité Arithmétique correspond au nombre d'opérations de calcul divisé par le nombre d'octets transférés au travers de la hiérarchie mémoire y compris les niveaux de cache. Il s'exprime en $Flop.Octet^{-1}$ [67]. 33, 53, 68, 69, 119

AVX Les instructions x86 Advanced Vector Extensions (AVX) permettent de réaliser des opérations vectorielles de 256bits. vi, 19, 20, 22, 24, 26, 42, 44–47, 53, 77, 82, 83, 117

AVX-2 Les instructions x86 AVX 2 viennent s'ajouter aux instructions AVX afin de proposer plus d'opérations vectorielles d'une taille maximale de 256bits comprenant également des opérations sur de plus petits vecteurs (128bits et 64bits). 20, 44–46, 85, 102, 117

AVX-512 Les instructions x86 AVX 512 permettent de réaliser des opérations vectorielles de 512bits. 20, 33, 76, 77, 82, 83, 85, 87, 97, 102, 117

AVX-512/256 Les instructions x86 AVX 512/256 concernent toutes les instructions vectorielles 256bits des jeux d'instruction AVX, AVX-2 et AVX-512. 85

Broadwell Broadwell est une architecture. Lorsque l'on parle de la Broadwell, il est question d'une machine d'architecture Broadwell. vi–viii, 39, 42, 43, 48, 49, 52–56, 62–64, 76–79, 83, 85, 87, 88, 96, 97, 100–102

C++ Le C++ est un langage de programmation inventé dans les années 1980. Les 2 plus devant le C marquent le fait que le C++ est une incrémentation du langage C. vi, 20, 29, 30, 41, 42, 45–48, 62, 71, 73, 75, 79, 83, 107

CARM Le CARM est une extension du modèle prenant en compte les performances mémoires des différents niveaux de cache. 35, 78, 102, 117, 118

CPU Central Processing Unit. 24, 30, 35

EDP Équations aux Dérivées Partielles. 3, 4, 6, 37, 39, 65, 105, 125–127

FMA Fused Multiply Add. 20, 33, 35, 42, 53

Fortran Le Fortran est un langage de programmation inventé en 1954 par John Backus. Ce langage a été inventé á l'origine pour le calcul numérique. 67, 70, 71, 107, 135

FPGA Field-Programmable Gate Array. 32

GB/s nombre en milliards d'octets par seconde. v, 31, 33, 35, 53, 60, 97

GFLOPS nombre en milliards d'opérations flottantes par seconde. 33, 35, 43, 48, 49, 52, 61, 63, 78, 83, 85, 87–89, 96, 108

GHZ nombre en milliards de fois par seconde. 71, 76

GLL Gauss Lobatto Legendre. v–viii, 8, 10, 67–69, 71, 73–75, 77, 78, 80–85, 87–90, 92–98, 107, 118

GPU Graphics Porcessing Unit. 24, 30, 32, 35

ILP Instruction Level Parallelism. 17

intrinsics Fonctions représentant les instructions vectorielles des processeurs x86 dans les langages de haut niveau (C, C++, ...). 26

Ivy Bridge Ivy Bridge est une architecture. Lorsque l'on parle de l'Ivy Bridge, il est question d'une machine d'architecture Ivy Bridge. vi, 39, 42, 43, 48, 49, 51, 52, 54–56, 62–64

LARM Le LARM est une extension du modèle CARM. Ce modèle LARM prend en compte l'effet NUMA. 35, 78, 102, 118

MPI Message Passing Interface. vii, 12, 67, 70, 71, 75, 76, 89, 102, 107

MPMD Multy Program Multy Data. On applique différents programmes á plusieurs données. 27, 49

NUMA Le NUMA est une caractéristique d'architecture dont les temps d'accès á la mémoire centrale peuvent varier selon la localité des accès et des données. v, viii, 30, 31, 34, 43, 51, 75, 86, 99–102, 108, 118

OI l'Intensité Opérationnelle correspond au nombre d'opérations de calcul divisé par nombre d'octets transférés depuis la mémoire centrale sans tenir compte des nivaux de cache. Ce dernier est défini dans l'article [67]. Comme l'A.I., il s'exprime en $Flop.Octet^{-1}$. 33, 77, 78, 97–99, 101, 108

ORM Le Original Roofline Model (ORM) est un modèle général de représentation graphique des limites techniques d'une machine en termes de capacité de calcul et de débit mémoire. v, 34, 78, 102, 117, 118

point GLL Chaque élément fini spectral se compose localement de points d'intégration appelés points GLL. v–viii, 8, 10, 68, 69, 71, 73–75, 77–84, 89, 90, 92–98, 107

QPI Quick Path Interconnect. v, 31, 100

quadrature Opération qui consiste á construire un carré de même surface que celle d'une figure curviligne. On prendra l'exemple d'un cercle de surface S. La quadrature de ce cercle est un carré de surface S. 67

RI l'Intensité de Réutilisation est le nombre de réutilisations maximales d'un octet depuis les caches pour une seule itération stencil. 40, 41, 43, 55, 56, 58, 61–64, 106, 107

roofline Une roofline est un modèle graphique liant les performances de calcul aux débits mémoires d'une machine selon les facteurs AI. vi, vii, 2, 34, 35, 37, 39, 42, 43, 64, 76–79, 88, 97, 99, 102, 106, 108, 118

SIMD Single Instruction Multy Data. 13, 19, 26, 27, 29, 37, 49, 71, 73, 76, 79, 82, 83, 85, 86, 88, 96, 102, 106

SISD Single Instruction Single Data. 19

Skylake Skylake est une architecture. Lorsque l'on parle de la Skylake, il est question d'une machine d'architecture Skylake. v, vii, viii, 17, 18, 33–35, 76–79, 83, 85, 87–89, 96–102, 107, 108

SPMD Single Program Multy Data. On applique le même programme á plusieurs données. Dans cette thèse, cela désigne les mm̂es calculs en multithreading. 27, 37, 49, 75, 76, 86, 88, 106

SSE Les instructions x86 Streaming SIMD Extensions (SSE) permettent de réaliser des opérations vectorielles de 128bits. 19, 20, 26, 119

stencil Un stencil est un objet mathématique composé de points relatifs (espace/-temps) pondérés définissant ainsi son voisinage spatio temporel. v, vi, viii, 4–6, 12, 13, 20, 26, 37, 39–65, 77, 89, 105–107, 126, 127

TFLOPS nombre en milliers de milliards d'opérations flottantes par seconde. 53, 60, 78

thread Le terme thread provient de l'anglais et se traduit par fil. Dans le cadre de cette thèse, un thread est un fil d'exécution. vi, viii, 12, 27, 29–32, 43, 44, 49–52, 63, 64, 75, 85, 86, 88, 91, 94, 100–102, 107

tuilage spatial Le tuilage spatial est une stratégie de parcours d'un domaine se basant sur une décomposition spatiale de ce domaine en tuile délimitant des sous-espaces. Ainsi, le parcours du domaine consiste á intégralement parcourir le sous-espace formé par une des tuiles avant de passer aux suivants. Dans un contexte multi-thread, les tuiles á parcourir peuvent être réparties entre les threads.. 5, 119

tuilage temporel Le tuilage temporel fonctionne sur le même principe que le tuilage spatial á la différence prêt qu'il décompose le parcours aussi bien dans l'espace que dans le temps.. 5

Annexes

Annexe A

Exemple de discrétisation

La méthode de résolution par approximation présentée dans cette sous-section est une brève introduction aux principes de résolution par discrétisation d'un modèle continu.

Cette méthode se fonde sur le fait que les valeurs physiques mises en relation dans l'EDP sont représentées par des fonctions. En effet, en ce qui concerne l'équation de la chaleur, la température est fonction du temps et de l'espace. On voit apparaitre une mise en relation entre une variation spatiale et une variation de température dans le temps. La méthode des différences finies se fonde donc sur la différence des développements limités d'une même fonction.

Pour mieux comprendre, il faut partir des définitions mathématiques liant la notion de dérivée à la notion de développement limité. Prenons une fonction f(x) continue infiniment dérivable en tout x.
Sa dérivée $f'(x)$ est définie par l'équation A.1.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{A.1}$$

De la même façon, comme $f(x)$ est dérivable, on peut exprimer la fonction f aux abords de x via l'équation A.2.
On remarquera l'ambivalence entre la définition de la dérivée et le développement limité d'une fonction plusieurs fois dérivable. En effet, $f(x+h) = f(x) + h^1 f^{(1)}(x) + o(h^1)$ alors on peut écrire $f(x+h) - f(x) = h^1 f^{(1)}(x) + o(h^1)$. Parconséquente $f^{(1)}(x) = \frac{f(x+h)-f(x)-o(h)}{h}$. Étant donné que h tend vers 0 alors $o(h)$ tend également vers 0 et donc $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$

$$f(x+h) = f(x) + \frac{h^1 f^{(1)}(x)}{1!} + \frac{h^2 f^{(2)}(x)}{2!} + ... + \frac{h^n f^{(n)}(x)}{n!} + o(h^n) \tag{A.2}$$

Les définitions précédentes font apparaitre des termes que l'on retrouve dans nos EDP. Prenons l'exemple de l'équation de la chaleur 2.1.

Nous avons besoin d'exprimer les variations de température pour chaque pas de temps en chaque point de notre domaine. Ainsi, nous pouvons réaliser des inférences avant comme arrière afin de simuler l'évolution des températures dans notre domaine.

En premier lieu, nous posons l'équation A.3. En second lieu nous posons les 2 équations A.4 et A.5.

$$\frac{u^{(1)}(x,y,z,t)}{\partial t} = \lim_{\tau \to 0} \frac{u(x,y,z,t+\tau) - u(x,y,z,t)}{\tau} \tag{A.3}$$

$$u(x+h,y,z,t) = u(x,y,z,t) + \frac{h^1 u^{(1)}(x,y,z,t)}{\partial x} + \frac{h^2 u^{(2)}(x,y,z,t)}{2!\partial x^2} + o(h^2) \tag{A.4}$$

$$u(x-h,y,z,t) = u(x,y,z,t) - \frac{h^1 u^{(1)}(x,y,z,t)}{\partial x} + \frac{h^2 u^{(2)}(x,y,z,t)}{2!\partial x^2} + o(h^2) \tag{A.5}$$

Ainsi, on en déduit A.6 qui implique le résultat A.7.

$$u(x+h,y,z,t) + u(x-h,y,z,t) = \frac{2u(x,y,z,t) + 2h^2 u^{(2)}(x,y,z,t)}{2\partial x^2} + 2o(h^2) \tag{A.6}$$

$$\frac{u^{(2)}(x,y,z,t)}{\partial x^2} = \frac{u(x+h,y,z,t) + u(x-h,y,z,t) - 2u(x,y,z,t) - 2o(h^2)}{h^2} \tag{A.7}$$

Lorsque h tend vers 0, le terme $o(h^2)$ tend vers 0. Ce qui donne le résultat final A.8.

$$\begin{aligned} \lim_{\tau \to 0} \frac{u(x,y,z,t+\tau) - u(x,y,z,t)}{\tau} = \\ \lim_{h \to 0} \frac{1}{h^2} [&u(x+h,y,z,t) + u(x-h,y,z,t) + \\ &u(x,y+h,z,t) + u(x,y-h,z,t) + \\ &u(x,y,z+h,t) + u(x,y,z-h,t) - \\ &6u(x,y,z,t)] \end{aligned} \tag{A.8}$$

On peut donc en déduire que le point aux coordonnées $(x;y;z)$ sera au pas de temps suivant égal à une somme pondérée de son voisinage lui compris au pas de temps courant.

Ce résultat nous amène au problème suivant. À supposer que nous ayons juste besoin de trouver la valeur d'un point après une seconde de simulation, l'analyse précédente nous indique qu'il y a un nombre infini de pas de temps puisque τ tend vers 0. À noter que l'on néglige, $o(h^2)$ car h tend également vers 0. De fait, nous aurions besoin de considérer un voisinage lui aussi infini. De plus, les résultats intermédiaires seraient arrondis au chiffre significatif près.
En pratique, les résultats attendus tolèrent un niveau d'approximation. Idéalement, l'ordre

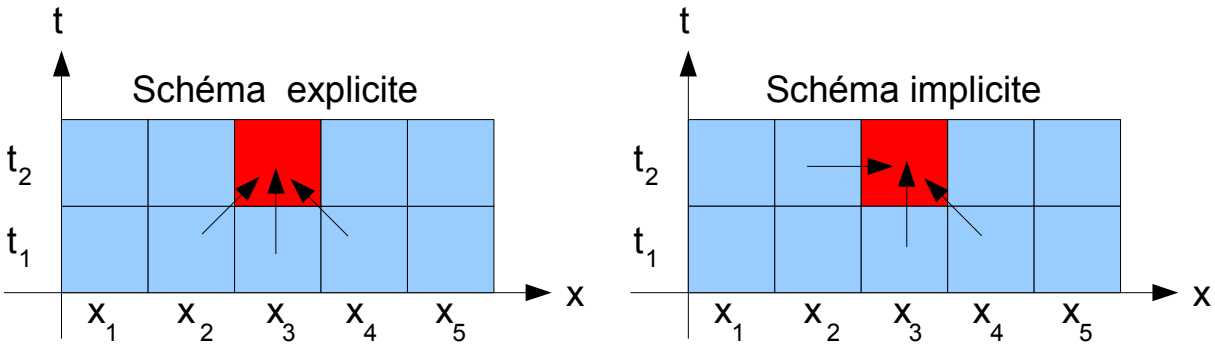

FIGURE A.1 – Exemple sur 1 dimension d'un stencil à schéma explicite à gauche et implicite à droite.

de grandeur du cumul des termes négligés $o(h^2)$ doit être inférieur au seuil d'approximation exigée. Ainsi, les chiffres significatifs seront exploitables. Pour arriver à résoudre notre problème précédent, on va passer du continu au discret. Il y a peu de différences en termes de formulation mathématique. En effet, on considère simplement que τ et h ont une valeur réelle définie et sont des paramètres de la simulation. Ainsi, il ne nous est pas nécessaire d'avoir tous les points continus du domaine, mais simplement les points espacés de h et ce dans toutes les dimensions. Il en va de même pour τ où chaque pas de temps sera de τ secondes. Notre domaine est délimité par un nombre de points suivant les 3 dimensions et forme ce que l'on appelle un maillage. Nous allons maintenant pouvoir simuler 1 seconde en réalisant $\frac{1}{\tau}$ pas de temps.

Pour conclure sur cette méthode, la solution approchée A.8 nous permet d'en déterminer un schéma numérique aussi appelé stencil. En effet, la solution A.8 nous indique le voisinage nécessaire ainsi que leur poids pour calculer chaque point au pas de temps suivant. Elle nous permet d'en déduire le stencil permettant la résolution approchée de notre EDP de la chaleur. Les stencils sont des objets mathématiques courant en simulation numérique. Ils peuvent être appliquer tel un opérateur sur des domaines en réalisant des sommes pondérées pour inférer de pas de temps en pas de temps. À noter que chaque cellule d'un stencil fait référence à un voisinage suivant l'espace et le temps. Dans le cas où le stencil requière des voisins inconnus, on parle de schémas implicites. L'application de tels schémas implique des dépendances dans l'ordonnancement du calcul de façon à calculer le voisinage inconnu. La figure A.1 présente la résolution d'une inconnue x_3 au pas de temps t_2 et résume la différence entre schémas explicites dont le voisinage est connu au temps t_1 et implicite dont le voisinage est inconnu au temps t_1.

La méthode des différences finies nous a permis de comprendre le besoin de passer du continu au discret pour résoudre approximativement des EDP. En effet, cette méthode étant la plus ancienne, elle part du continu brut des EDP en faisant apparaitre la discrétisation afin de parvenir à résoudre et évaluer la grandeur physique en certains points. De cette discrétisation émerge un schéma récurant de sommes pondérées du voisinage correspondant à l'objet mathématique stencil abordé dans cette thèse. D'autres méthodes plus récentes profitent de la notion de maillages discrétisés pour aborder la résolution d'une autre manière telle que les méthodes par volumes et éléments finis.

Annexe B

Sources en Fortran du calcul des forces internes d'EFISPEC

```
module cifo4

  implicit none

  integer :: IG_NGLL = 5
  integer, allocatable, dimension(:,:,:,:) :: ig_hexa_gll_glonum

  real, allocatable, dimension(:,:) :: rg_gll_displacement
  real, allocatable, dimension(:,:) :: rg_gll_lagrange_deriv

  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dxidx
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dxidy
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dxidz

  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_detdx
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_detdy
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_detdz

  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dzedx
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dzedy
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_dzedz

  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_rhovp2
  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_rhovs2

  real, allocatable, dimension(:,:,:,:) :: rg_hexa_gll_jacobian_det
  real, allocatable, dimension(:) :: rg_gll_weight

  real, allocatable, dimension(:,:) :: rg_gll_acceleration
  real, allocatable, dimension(:,:) :: rg_gll_acceleration_ref

contains

  subroutine compute_internal_forces_order4( elt_start, elt_end )

    integer, intent(in) :: elt_start
    integer, intent(in) :: elt_end
```

```
integer iel,igll, k, l , m
real duxdxi, duydxi, duzdxi
real duxdet, duydet, duzdet
real duxdze, duydze, duzdze

real duxdx, duxdy, duxdz
real duydx, duydy, duydz
real duzdx, duzdy, duzdz

real dxidx, dxidy, dxidz
real detdx, detdy, detdz
real dzedx, dzedy, dzedz

real trace_tau
real tauxx, tauxy, tauxz
real tauyy, tauyz
real tauzz

real tmpx1, tmpy1, tmpz1
real tmpx2, tmpy2, tmpz2
real tmpx3, tmpy3, tmpz3

real fac1, fac2, fac3

real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpx1
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpx2
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpx3
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpy1
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpy2
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpy3
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpz1
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpz2
real, dimension(IG_NGLL,IG_NGLL,IG_NGLL)          :: intpz3

real, dimension(3, IG_NGLL,IG_NGLL,IG_NGLL) :: rl_displacement_gll
real, dimension(3,IG_NGLL,IG_NGLL,IG_NGLL) :: rl_acceleration_gll

do iel = elt_start,elt_end

   do k = 1,IG_NGLL          !zeta
      do l = 1,IG_NGLL       !eta
         do m = 1,IG_NGLL    !xi

            igll = ig_hexa_gll_glonum(m,l,k,iel)
            rl_displacement_gll(1,m,l,k) = rg_gll_displacement(1,igll)
            rl_displacement_gll(2,m,l,k) = rg_gll_displacement(2,igll)
            rl_displacement_gll(3,m,l,k) = rg_gll_displacement(3,igll)

         enddo
      enddo
   enddo

   do k = 1,IG_NGLL          !zeta
    do l = 1,IG_NGLL       !eta
     do m = 1,IG_NGLL    !xi
```

```
        duxdxi = rl_displacement_gll(1,1,l,k)*rg_gll_lagrange_deriv(1,m) &
               + rl_displacement_gll(1,2,l,k)*rg_gll_lagrange_deriv(2,m) &
               + rl_displacement_gll(1,3,l,k)*rg_gll_lagrange_deriv(3,m) &
               + rl_displacement_gll(1,4,l,k)*rg_gll_lagrange_deriv(4,m) &
               + rl_displacement_gll(1,5,l,k)*rg_gll_lagrange_deriv(5,m)


        duydxi = rl_displacement_gll(2,1,l,k)*rg_gll_lagrange_deriv(1,m) &
               + rl_displacement_gll(2,2,l,k)*rg_gll_lagrange_deriv(2,m) &
               + rl_displacement_gll(2,3,l,k)*rg_gll_lagrange_deriv(3,m) &
               + rl_displacement_gll(2,4,l,k)*rg_gll_lagrange_deriv(4,m) &
               + rl_displacement_gll(2,5,l,k)*rg_gll_lagrange_deriv(5,m)

        duzdxi = rl_displacement_gll(3,1,l,k)*rg_gll_lagrange_deriv(1,m) &
               + rl_displacement_gll(3,2,l,k)*rg_gll_lagrange_deriv(2,m) &
               + rl_displacement_gll(3,3,l,k)*rg_gll_lagrange_deriv(3,m) &
               + rl_displacement_gll(3,4,l,k)*rg_gll_lagrange_deriv(4,m) &
               + rl_displacement_gll(3,5,l,k)*rg_gll_lagrange_deriv(5,m)

        duxdet = rl_displacement_gll(1,m,1,k)*rg_gll_lagrange_deriv(1,l) &
               + rl_displacement_gll(1,m,2,k)*rg_gll_lagrange_deriv(2,l) &
               + rl_displacement_gll(1,m,3,k)*rg_gll_lagrange_deriv(3,l) &
               + rl_displacement_gll(1,m,4,k)*rg_gll_lagrange_deriv(4,l) &
               + rl_displacement_gll(1,m,5,k)*rg_gll_lagrange_deriv(5,l)

        duydet = rl_displacement_gll(2,m,1,k)*rg_gll_lagrange_deriv(1,l) &
               + rl_displacement_gll(2,m,2,k)*rg_gll_lagrange_deriv(2,l) &
               + rl_displacement_gll(2,m,3,k)*rg_gll_lagrange_deriv(3,l) &
               + rl_displacement_gll(2,m,4,k)*rg_gll_lagrange_deriv(4,l) &
               + rl_displacement_gll(2,m,5,k)*rg_gll_lagrange_deriv(5,l)

        duzdet = rl_displacement_gll(3,m,1,k)*rg_gll_lagrange_deriv(1,l) &
               + rl_displacement_gll(3,m,2,k)*rg_gll_lagrange_deriv(2,l) &
               + rl_displacement_gll(3,m,3,k)*rg_gll_lagrange_deriv(3,l) &
               + rl_displacement_gll(3,m,4,k)*rg_gll_lagrange_deriv(4,l) &
               + rl_displacement_gll(3,m,5,k)*rg_gll_lagrange_deriv(5,l)

        duxdze = rl_displacement_gll(1,m,l,1)*rg_gll_lagrange_deriv(1,k) &
               + rl_displacement_gll(1,m,l,2)*rg_gll_lagrange_deriv(2,k) &
               + rl_displacement_gll(1,m,l,3)*rg_gll_lagrange_deriv(3,k) &
               + rl_displacement_gll(1,m,l,4)*rg_gll_lagrange_deriv(4,k) &
               + rl_displacement_gll(1,m,l,5)*rg_gll_lagrange_deriv(5,k)

        duydze = rl_displacement_gll(2,m,l,1)*rg_gll_lagrange_deriv(1,k) &
               + rl_displacement_gll(2,m,l,2)*rg_gll_lagrange_deriv(2,k) &
               + rl_displacement_gll(2,m,l,3)*rg_gll_lagrange_deriv(3,k) &
               + rl_displacement_gll(2,m,l,4)*rg_gll_lagrange_deriv(4,k) &
               + rl_displacement_gll(2,m,l,5)*rg_gll_lagrange_deriv(5,k)

        duzdze = rl_displacement_gll(3,m,l,1)*rg_gll_lagrange_deriv(1,k) &
               + rl_displacement_gll(3,m,l,2)*rg_gll_lagrange_deriv(2,k) &
               + rl_displacement_gll(3,m,l,3)*rg_gll_lagrange_deriv(3,k) &
               + rl_displacement_gll(3,m,l,4)*rg_gll_lagrange_deriv(4,k) &
               + rl_displacement_gll(3,m,l,5)*rg_gll_lagrange_deriv(5,k)
```

```
      dxidx = rg_hexa_gll_dxidx(m,l,k,iel)
      dxidy = rg_hexa_gll_dxidy(m,l,k,iel)
      dxidz = rg_hexa_gll_dxidz(m,l,k,iel)
      detdx = rg_hexa_gll_detdx(m,l,k,iel)
      detdy = rg_hexa_gll_detdy(m,l,k,iel)
      detdz = rg_hexa_gll_detdz(m,l,k,iel)
      dzedx = rg_hexa_gll_dzedx(m,l,k,iel)
      dzedy = rg_hexa_gll_dzedy(m,l,k,iel)
      dzedz = rg_hexa_gll_dzedz(m,l,k,iel)

      duxdx = duxdxi*dxidx + duxdet*detdx + duxdze*dzedx
      duxdy = duxdxi*dxidy + duxdet*detdy + duxdze*dzedy
      duxdz = duxdxi*dxidz + duxdet*detdz + duxdze*dzedz
      duydx = duydxi*dxidx + duydet*detdx + duydze*dzedx
      duydy = duydxi*dxidy + duydet*detdy + duydze*dzedy
      duydz = duydxi*dxidz + duydet*detdz + duydze*dzedz
      duzdx = duzdxi*dxidx + duzdet*detdx + duzdze*dzedx
      duzdy = duzdxi*dxidy + duzdet*detdy + duzdze*dzedy
      duzdz = duzdxi*dxidz + duzdet*detdz + duzdze*dzedz

      trace_tau = (rg_hexa_gll_rhovp2(m,l,k,iel)
                           -2.0*rg_hexa_gll_rhovs2(m,l,k,iel))
                                 *(duxdx+duydy+duzdz)
      tauxx     = trace_tau
                           +2.0*rg_hexa_gll_rhovs2(m,l,k,iel)*duxdx
      tauyy     = trace_tau
                           +2.0*rg_hexa_gll_rhovs2(m,l,k,iel)*duydy
      tauzz     = trace_tau
                          +2.0*rg_hexa_gll_rhovs2(m,l,k,iel)*duzdz
      tauxy     = rg_hexa_gll_rhovs2(m,l,k,iel)*(duxdy+duydx)
      tauxz     = rg_hexa_gll_rhovs2(m,l,k,iel)*(duxdz+duzdx)
      tauyz     = rg_hexa_gll_rhovs2(m,l,k,iel)*(duydz+duzdy)

      intpx1(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                        *(tauxx*dxidx+tauxy*dxidy+tauxz*dxidz)
      intpx2(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                        *(tauxx*detdx+tauxy*detdy+tauxz*detdz)
      intpx3(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxx*dzedx+tauxy*dzedy+tauxz*dzedz)

      intpy1(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxy*dxidx+tauyy*dxidy+tauyz*dxidz)
      intpy2(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxy*detdx+tauyy*detdy+tauyz*detdz)
      intpy3(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxy*dzedx+tauyy*dzedy+tauyz*dzedz)

      intpz1(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxz*dxidx+tauyz*dxidy+tauzz*dxidz)
      intpz2(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxz*detdx+tauyz*detdy+tauzz*detdz)
      intpz3(m,l,k) = rg_hexa_gll_jacobian_det(m,l,k,iel)
                                 *(tauxz*dzedx+tauyz*dzedy+tauzz*dzedz)

    enddo
  enddo
```

```
      enddo

      do k = 1,IG_NGLL
       do l = 1,IG_NGLL
        do m = 1,IG_NGLL

        tmpx1 = intpx1(1,l,k)*rg_gll_lagrange_deriv(m,1)*rg_gll_weight(1) &
              + intpx1(2,l,k)*rg_gll_lagrange_deriv(m,2)*rg_gll_weight(2) &
              + intpx1(3,l,k)*rg_gll_lagrange_deriv(m,3)*rg_gll_weight(3) &
              + intpx1(4,l,k)*rg_gll_lagrange_deriv(m,4)*rg_gll_weight(4) &
              + intpx1(5,l,k)*rg_gll_lagrange_deriv(m,5)*rg_gll_weight(5)

        tmpy1 = intpy1(1,l,k)*rg_gll_lagrange_deriv(m,1)*rg_gll_weight(1) &
              + intpy1(2,l,k)*rg_gll_lagrange_deriv(m,2)*rg_gll_weight(2) &
              + intpy1(3,l,k)*rg_gll_lagrange_deriv(m,3)*rg_gll_weight(3) &
              + intpy1(4,l,k)*rg_gll_lagrange_deriv(m,4)*rg_gll_weight(4) &
              + intpy1(5,l,k)*rg_gll_lagrange_deriv(m,5)*rg_gll_weight(5)

        tmpz1 = intpz1(1,l,k)*rg_gll_lagrange_deriv(m,1)*rg_gll_weight(1) &
              + intpz1(2,l,k)*rg_gll_lagrange_deriv(m,2)*rg_gll_weight(2) &
              + intpz1(3,l,k)*rg_gll_lagrange_deriv(m,3)*rg_gll_weight(3) &
              + intpz1(4,l,k)*rg_gll_lagrange_deriv(m,4)*rg_gll_weight(4) &
              + intpz1(5,l,k)*rg_gll_lagrange_deriv(m,5)*rg_gll_weight(5)

        tmpx2 = intpx2(m,1,k)*rg_gll_lagrange_deriv(l,1)*rg_gll_weight(1) &
              + intpx2(m,2,k)*rg_gll_lagrange_deriv(l,2)*rg_gll_weight(2) &
              + intpx2(m,3,k)*rg_gll_lagrange_deriv(l,3)*rg_gll_weight(3) &
              + intpx2(m,4,k)*rg_gll_lagrange_deriv(l,4)*rg_gll_weight(4) &
              + intpx2(m,5,k)*rg_gll_lagrange_deriv(l,5)*rg_gll_weight(5)

        tmpy2 = intpy2(m,1,k)*rg_gll_lagrange_deriv(l,1)*rg_gll_weight(1) &
              + intpy2(m,2,k)*rg_gll_lagrange_deriv(l,2)*rg_gll_weight(2) &
              + intpy2(m,3,k)*rg_gll_lagrange_deriv(l,3)*rg_gll_weight(3) &
              + intpy2(m,4,k)*rg_gll_lagrange_deriv(l,4)*rg_gll_weight(4) &
              + intpy2(m,5,k)*rg_gll_lagrange_deriv(l,5)*rg_gll_weight(5)

        tmpz2 = intpz2(m,1,k)*rg_gll_lagrange_deriv(l,1)*rg_gll_weight(1) &
              + intpz2(m,2,k)*rg_gll_lagrange_deriv(l,2)*rg_gll_weight(2) &
              + intpz2(m,3,k)*rg_gll_lagrange_deriv(l,3)*rg_gll_weight(3) &
              + intpz2(m,4,k)*rg_gll_lagrange_deriv(l,4)*rg_gll_weight(4) &
              + intpz2(m,5,k)*rg_gll_lagrange_deriv(l,5)*rg_gll_weight(5)

        tmpx3 = intpx3(m,l,1)*rg_gll_lagrange_deriv(k,1)*rg_gll_weight(1) &
              + intpx3(m,l,2)*rg_gll_lagrange_deriv(k,2)*rg_gll_weight(2) &
              + intpx3(m,l,3)*rg_gll_lagrange_deriv(k,3)*rg_gll_weight(3) &
              + intpx3(m,l,4)*rg_gll_lagrange_deriv(k,4)*rg_gll_weight(4) &
              + intpx3(m,l,5)*rg_gll_lagrange_deriv(k,5)*rg_gll_weight(5)

        tmpy3 = intpy3(m,l,1)*rg_gll_lagrange_deriv(k,1)*rg_gll_weight(1) &
              + intpy3(m,l,2)*rg_gll_lagrange_deriv(k,2)*rg_gll_weight(2) &
              + intpy3(m,l,3)*rg_gll_lagrange_deriv(k,3)*rg_gll_weight(3) &
              + intpy3(m,l,4)*rg_gll_lagrange_deriv(k,4)*rg_gll_weight(4) &
              + intpy3(m,l,5)*rg_gll_lagrange_deriv(k,5)*rg_gll_weight(5)

        tmpz3 = intpz3(m,l,1)*rg_gll_lagrange_deriv(k,1)*rg_gll_weight(1) &
              + intpz3(m,l,2)*rg_gll_lagrange_deriv(k,2)*rg_gll_weight(2) &
```

```
              + intpz3(m,l,3)*rg_gll_lagrange_deriv(k,3)*rg_gll_weight(3) &
              + intpz3(m,l,4)*rg_gll_lagrange_deriv(k,4)*rg_gll_weight(4) &
              + intpz3(m,l,5)*rg_gll_lagrange_deriv(k,5)*rg_gll_weight(5)

        fac1 = rg_gll_weight(l)*rg_gll_weight(k)
        fac2 = rg_gll_weight(m)*rg_gll_weight(k)
        fac3 = rg_gll_weight(m)*rg_gll_weight(l)

        rl_acceleration_gll(1,m,l,k) = (fac1*tmpx1
                                      + fac2*tmpx2 + fac3*tmpx3)
        rl_acceleration_gll(2,m,l,k) = (fac1*tmpy1
                                      + fac2*tmpy2 + fac3*tmpy3)
        rl_acceleration_gll(3,m,l,k) = (fac1*tmpz1
                                      + fac2*tmpz2 + fac3*tmpz3)

        enddo
       enddo
      enddo

      do k = 1,IG_NGLL          !zeta
        do l = 1,IG_NGLL       !eta
          do m = 1,IG_NGLL    !xi

            igll = ig_hexa_gll_glonum(m,l,k,iel)

            rg_gll_acceleration(1,igll) = rg_gll_acceleration(1,igll)
                                - rl_acceleration_gll(1,m,l,k)
            rg_gll_acceleration(2,igll) = rg_gll_acceleration(2,igll)
                                        - rl_acceleration_gll(2,m,l,k)
            rg_gll_acceleration(3,igll) = rg_gll_acceleration(3,igll)
                                        - rl_acceleration_gll(3,m,l,k)
          enddo
        enddo
      enddo
    enddo

  end subroutine compute_internal_forces_order4
end module cifo4
```

Annexe C

Sources traduites en C++ du calcul des forces internes d'EFISPEC

Le code source qui va suivre est directement extrait de l'implémentation en Fortran de l'application Efispec3D.

```
#include <iostream>
#include <vector>
#include <fstream>

#include <chrono>
#include <boost/align/aligned_allocator.hpp>

#include <string.h>

using namespace std;

#define ORD 4 //Nombre de degre de liberte

#define NB_ELEM (NB_ELEM_AX*NB_ELEM_AX*NB_ELEM_AX)

#define AXSZ (ORD+1) //Nombre de sous elements en unidimensionnel
#define ITRF 1 //Taille de l'interface inter elements
#define IELACCS (AXSZ-ITRF)
#define AXSize (IELACCS*NB_ELEM_AX+ITRF) //Nombre de GLL en unidimentionnel
#define NSize (AXSize*AXSize*AXSize)

#define IDX2( m, l ) ( AXSZ * l + m )
#define IDX3( m, l, k ) ( (AXSZ*AXSZ) * k + AXSZ * l + m )
#define IDX4( m, l, k, iel ) ( (AXSZ*AXSZ*AXSZ) * iel \\
                  + (AXSZ*AXSZ) * k + AXSZ * l + m )

using FAlgAloc = boost::alignment::aligned_allocator< float, 64 >;
using FloatVect = std::vector< float,AlgAloc >;

using IAlgAloc = boost::alignment::aligned_allocator< uint32_t, 64 >;
using UIntVect = std::vector< uint32_t, IAlgAloc >;

UIntVect  rg_gll_id;
FloatVect rg_gll_displacement;
FloatVect rg_gll_weight;
```

```
FloatVect rg_gll_lagrange_deriv;
FloatVect rg_gll_acceleration;

std::vector< FloatVect > rg_hexa_gll_did;

FloatVect rg_hexa_gll_rhovp2;
FloatVect rg_hexa_gll_rhovs2;
FloatVect rg_hexa_gll_jacobian_det;

void compute_internal_forces_order4( std::size_t elt_start,
                          std::size_t elt_end )
{
 for(std::size_t nIel=elt_start;nIel<elt_end;++nIel)
 {
  const float *rg_hexa_gll_did_loc = rg_hexa_gll_did[nIel].data();

  std::size_t nIelX = nIel*AXSZ*AXSZ*AXSZ;
  const uint32_t *rg_gll_id_loc = &(rg_gll_id[nIelX]);
  const float *rg_hexa_gll_rhovp2_loc = &(rg_hexa_gll_rhovp2[nIelX]);
  const float *rg_hexa_gll_rhovs2_loc = &(rg_hexa_gll_rhovs2[nIelX]);
  const float *rg_hexa_gll_jacobian_det_loc =
               &(rg_hexa_gll_jacobian_det[nIelX]);

  float local[ AXSZ * AXSZ * AXSZ * 9 ];

  for( std::size_t k = 0 ; k < AXSZ ; ++k )
  {
   for( std::size_t l = 0 ; l < AXSZ ; ++l )
   {
    for( std::size_t m = 0 ; m < AXSZ ; ++m )
    {
     //[dxi det dze][xyz]
     float dud[3][3];memset((void*)dud,0x00,sizeof(dud));

     for( std::size_t i = 0 ; i<AXSZ ; ++i )
     {
      uint32_t igll[3];
      float coeff[3];

      igll[0] = rg_gll_id_loc[IDX3(i, l, k)];
      coeff[0] = rg_gll_lagrange_deriv[ IDX2( i, m ) ];

      igll[1] = rg_gll_id_loc[IDX3(m, i, k)];
      coeff[1] = rg_gll_lagrange_deriv[ IDX2( i, l ) ];

      igll[2] = rg_gll_id_loc[IDX3(m, l, i)];
      coeff[2] = rg_gll_lagrange_deriv[ IDX2( i, k ) ];

      for( std::size_t p = 0 ; p<3 ; ++p )
      {
       for( std::size_t j = 0 ; j<3 ; ++j )
        dud[p][j] += rg_gll_displacement[ j + igll[p] ] * coeff[p];
      }//p
     }//i
```

```
    //[dxi det dze][xyz]
    const float *rg_hexa_gll_did_loc_mat = &(
                        rg_hexa_gll_did_loc[9*IDX3(m,l,k)]);

    //[dxdydz][xyz]
    float du[3][3];
    for(int j=0;j<3;++j)
    for(int i=0;i<3;++i)
    {
     du[j][i] = 0;
     for(int k=0;k<3;++k)//rg_hexa_gll_did_loc_mat[k:dxdydz]
      du[j][i] += dud[k][i]*rg_hexa_gll_did_loc_mat[3*k+j];
    }//i

    auto rhovp2 = rg_hexa_gll_rhovp2_loc[ IDX3( m, l, k ) ];
    auto rhovs2 = rg_hexa_gll_rhovs2_loc[ IDX3( m, l, k ) ];

    auto t_tau = ( rhovp2 - 2.0 * rhovs2 )
                      *(du[0][0]+du[1][1]+du[2][2]);
    auto tauxx     = t_tau + 2.0*rhovs2*du[0][0];
    auto tauyy     = t_tau + 2.0*rhovs2*du[1][1];
    auto tauzz     = t_tau + 2.0*rhovs2*du[2][2];
    auto tauxy     =               rhovs2*(du[1][0]+du[0][1]);
    auto tauxz     =               rhovs2*(du[2][0]+du[0][2]);
    auto tauyz     =               rhovs2*(du[2][1]+du[1][2]);

    float *local_loc = &(local[9*IDX3( m, l, k )]);
    float jaco = rg_hexa_gll_jacobian_det_loc[ IDX3( m, l, k ) ];
    float accu;

    float tau[3][3];
    tau[0][0] = tauxx;tau[0][1] = tauxy;tau[0][2] = tauxz;
    tau[1][0] = tauxy;tau[1][1] = tauyy;tau[1][2] = tauyz;
    tau[2][0] = tauxz;tau[2][1] = tauyz;tau[2][2] = tauzz;

    //local[ [klm][xyz][123] ]
    for(int j=0;j<3;++j)
     for(int i=0;i<3;++i)
     {
      accu = 0;
      for(int k=0;k<3;++k)//accu += tau[j][k]*dXiEtZe[i][k];
       accu += tau[j][k]*du[i][k];//dXiEtZe[i][k];
      local_loc[3*j+i] = accu*jaco;
     }//i

   }//m
  }//l
 }//k

 float fac[3];
 std::size_t bc[3];

 for( bc[2] = 0 ; bc[2] < AXSZ ; ++bc[2] )
 {//k
  for( bc[1] = 0 ; bc[1] < AXSZ ; ++bc[1] )
  {//l
```

```
    fac[0] = rg_gll_weight[ bc[1] ]*rg_gll_weight[ bc[2] ];

    for( bc[0] = 0 ; bc[0] < AXSZ ; ++bc[0] )
    {//m
     fac[1] = rg_gll_weight[ bc[0] ]*rg_gll_weight[ bc[2] ];
     fac[2] = rg_gll_weight[ bc[0] ]*rg_gll_weight[ bc[1] ];

     float *local_loc = &(local[9*IDX3( bc[0], bc[1], bc[2] )]);
     float *rg_gll_acceleration_loc_loc = &(
            rg_gll_acceleration[
                rg_gll_id_loc[IDX3(bc[0], bc[1], bc[2])] ]);

     for(int i=0;i<3;++i)
     {
      float accu = 0;
      int f=9;

      for(int c=0;c<3;++c)
      {
       float *local_loc_mat = &(local_loc[i*3+c]);
       float tmp = 0;

       for(int u=0;u<AXSZ;++u)
        tmp+=local_loc_mat[ f*(u-bc[c]) ]
                  *rg_gll_lagrange_deriv[ IDX2( bc[c], u ) ]
                  *rg_gll_weight[ u ];

       accu += fac[c]*tmp;
       f*=AXSZ;
      }//c

      rg_gll_acceleration_loc_loc[i]-=accu;
     }//i
    }//m
   }//l
  }//k

 }//nIel
}
```

Annexe D

Sources en C++ du calcul vectorisé des forces internes d'EFISPEC

```
#include <iostream>
#include <vector>
#include <fstream>

#if defined(_OPENMP)
#include <omp.h>
#endif

#include <chrono>
#include <boost/align/aligned_allocator.hpp>

#include <string.h>

using namespace std;

#define ORD 4 //Nombre de degre de liberte

#define NB_ELEM (NB_ELEM_AX*NB_ELEM_AX*NB_ELEM_AX)

#define AXSZ (ORD+1) //Nombre de sous elements en unidimensionnel
#define ITRF 1 //Taille de l'interface inter elements
#define IELACCS (AXSZ-ITRF)
#define AXSize (IELACCS*NB_ELEM_AX+ITRF) //Nombre de GLL en unidimentionnel
#define NSize (AXSize*AXSize*AXSize)

#define IDX2( m, l ) ( AXSZ * l + m )
#define IDX3( m, l, k ) ( (AXSZ*AXSZ) * k + AXSZ * l + m )
#define IDX4( m, l, k, iel ) ( (AXSZ*AXSZ*AXSZ) * iel \\
                  + (AXSZ*AXSZ) * k + AXSZ * l + m )


using FAlgAloc = boost::alignment::aligned_allocator< float, 64 >;
using FloatVect = std::vector< float,AlgAloc >;

using IAlgAloc = boost::alignment::aligned_allocator< uint32_t, 64 >;
using UIntVect = std::vector< uint32_t, IAlgAloc >;

//Les 2 vecteurs se structurent par lot de 16 elements finis entrelaces.
```

```
UIntVect  rg_gll_id;
std::vector< FloatVect > rg_hexa_gll_did;

FloatVect rg_hexa_gll_rhovp2;
FloatVect rg_hexa_gll_rhovs2;

FloatVect rg_hexa_gll_jacobian_det;

FloatVect rg_gll_weight;
FloatVect rg_gll_lagrange_deriv;

FloatVect rg_gll_displacement;
FloatVect rg_gll_acceleration;


#define VSize 16
#define VFloat VFloat16

inline void vSftStore(float *rg_gll_acceleration_loc_loc1,
  VFloat16 v1, VFloat16 reg,
  float *rg_gll_acceleration_loc_loc2, VFloat16 v2)
{
 _mm512_store_ps(rg_gll_acceleration_loc_loc2,
  _mm512_mask_blend_ps(0x0001,v2,
   vShift<1>(reg,vZero<VFloat16>()) ));
 _mm512_store_ps(rg_gll_acceleration_loc_loc1,
  _mm512_mask_blend_ps(0x0001,
   vShift<1>(vZero<VFloat16>(),reg),v1 ));
}

inline VFloat vLoadGLLComp(const uint32_t *gllIds,int c)
{
    return vGather<VFloat>(
                           &(rg_gll_displacement[c]),
                           gllIds);
}

inline void vAccuGLLComp(const uint32_t *gllIds,int c, const VFloat v)
{
    vScatter(
    &(rg_gll_acceleration[c]),
    gllIds,
    vSub(vGather<VFloat>(&(rg_gll_acceleration[c]),gllIds),v));
}

inline VFloat16 vAdd(VFloat16 a,VFloat16 b){return _mm512_add_ps(a,b);}
inline VFloat16 vSub(VFloat16 a,VFloat16 b){return _mm512_sub_ps(a,b);}
inline VFloat16 vMul(VFloat16 a,VFloat16 b){return _mm512_mul_ps(a,b);}

void compute_internal_forces_order4( std::vector<UIntVect> &colors )
{
 for(std::vector<UIntVect>::iterator col=
      colors.begin(); col!=colors.end(); ++col)
 {
  #pragma omp parallel for
  for(uint32_t nIel=0;nIel<col->size();nIel+=VSize)
```

```
//Chaque vecteur d'une couleur contient un multiple de 16 elements finis.
{
 uint32_t nIelX = col->at(nIel);

 /*float *rg_gll_displacement_loc = &rg_gll_displacement[nElemX];
 float *rg_gll_acceleration_loc = &rg_gll_acceleration[nElemX];*/
 const float *rg_hexa_gll_did_loc = rg_hexa_gll_did[nIelX/VSize].data();

 nIelX *= AXSZ*AXSZ*AXSZ;
 const uint32_t *rg_gll_id_loc = &(rg_gll_id[nIelX]);
 const float *rg_hexa_gll_rhovp2_loc = &(rg_hexa_gll_rhovp2[nIelX]);
 const float *rg_hexa_gll_rhovs2_loc = &(rg_hexa_gll_rhovs2[nIelX]);
 const float *rg_hexa_gll_jacobian_det_loc =
              &(rg_hexa_gll_jacobian_det[nIelX]);

 //float local[ VSize * 5 * 5 * 5 * 9 ];
 //FloatVect local(VSize * 5 * 5 * 5 * 9);
 __attribute__((aligned(64))) VFloat local[ AXSZ * AXSZ * AXSZ * 9 ];

 for( std::size_t k = 0 ; k < AXSZ ; ++k )
 {
  for( std::size_t l = 0 ; l < AXSZ ; ++l )
  {
   for( std::size_t m = 0 ; m < AXSZ ; ++m )
   {
    //[dxi det dze][xyz]
    __attribute__((aligned(64))) VFloat dud[3][3];
    for(int j=0;j<3;++j)for(int i=0;i<3;++i)dud[j][i] = vZero<VFloat>();

    for( std::size_t i = 0 ; i<AXSZ ; ++i )
    {
     uint32_t igll[3];
     float coeff[3];

     igll[0] = VSize*IDX3(i, l, k);
     coeff[0] = rg_gll_lagrange_deriv[ IDX2( i, m ) ];

     igll[1] = VSize*IDX3(m, i, k);
     coeff[1] = rg_gll_lagrange_deriv[ IDX2( i, l ) ];

     igll[2] = VSize*IDX3(m, l, i);
     coeff[2] = rg_gll_lagrange_deriv[ IDX2( i, k ) ];

     for( std::size_t p = 0 ; p<3 ; ++p )
     {
      for( std::size_t j = 0 ; j<3 ; ++j )
      {
       VFloat vReg = vLoadGLLComp(&(rg_gll_id_loc[igll[p]]),j);

       dud[p][j]=
            vMAdd(vReg,
            vSet1<VFloat>(coeff[p]),
            dud[p][j]);
      }//j
     }//p
    }//i
```

```
//[dxi det dze][xyz]
const float *rg_hexa_gll_did_loc_mat =
       &(rg_hexa_gll_did_loc[VSize*9*IDX3(m,l,k)]);

/[dxdydz][xyz]
//float du[3][3*VSize];
//FloatVect du(3*3*VSize);
__attribute__((aligned(64))) VFloat du[3][3];

for(int j=0;j<3;++j)
 for(int i=0;i<3;++i)
 {
  //du[j][i] = 0;
  du[j][i] = vZero<VFloat>();
  for(int k=0;k<3;++k)
   //du[j][i] += dud[k][i]*rg_hexa_gll_did_loc_mat[VSize*3*k+j];
   du[j][i] =
        vMAdd(dud[k][i],
        vLoad<VFloat>(&(rg_hexa_gll_did_loc_mat[VSize*(3*k+j)])),
        du[j][i]);
 }//i

auto rhovp2 = vLoad<VFloat>(
              &(rg_hexa_gll_rhovp2_loc[ VSize*IDX3( m, l, k ) ]));
auto rhovs2 = vLoad<VFloat>(
              &(rg_hexa_gll_rhovs2_loc[ VSize*IDX3( m, l, k ) ]));

auto t_tau = vMul(
    ( vMAdd(vSet1<VFloat>(-2.0f), rhovs2, rhovp2) ),
    ( vAdd(du[0][0],vAdd(du[1][1],du[2][2])) )
      );

auto tauxx = vMAdd(vSet1<VFloat>(2.0f),vMul(rhovs2,du[0][0]),t_tau);
auto tauyy = vMAdd(vSet1<VFloat>(2.0f),vMul(rhovs2,du[1][1]),t_tau);
auto tauzz = vMAdd(vSet1<VFloat>(2.0f),vMul(rhovs2,du[2][2]),t_tau);
auto tauxy =                  vMul(rhovs2,vAdd(du[1][0],du[0][1]));
auto tauxz =                  vMul(rhovs2,vAdd(du[2][0],du[0][2]));
auto tauyz =                  vMul(rhovs2,vAdd(du[2][1],du[1][2]));

//FloatVect tau(3*3*VSize);
//float tau[3][3*VSize];
VFloat tau[3][3];
tau[0][0] = tauxx; tau[0][1] = tauxy; tau[0][2] = tauxz;
tau[1][0] = tauxy; tau[1][1] = tauyy; tau[1][2] = tauyz;
tau[2][0] = tauxz; tau[2][1] = tauyz; tau[2][2] = tauzz;

VFloat *local_loc = &(local[9*IDX3( m, l, k )]);
auto jaco = vLoad<VFloat>(&(
          rg_hexa_gll_jacobian_det_loc[ VSize*IDX3( m, l, k ) ]));
VFloat accu;

//local[ [klm][xyz][123] ]
for(int j=0;j<3;++j)
for(int i=0;i<3;++i)
{
```

```
        accu = vZero<VFloat>();
        for(int k=0;k<3;++k)//accu += tau[j][k]*dXiEtZe[i][k];
        accu = vMAdd(tau[j][k],du[i][k],accu);
        local_loc[3*j+i]= vMul(accu,jaco);
       }//i

     }//m
    }//l
   }//k

   float fac[3];
   std::size_t bc[3];

   for( bc[2] = 0 ; bc[2] < AXSZ ; ++bc[2] )
   {//k
    for( bc[1] = 0 ; bc[1] < AXSZ ; ++bc[1] )
    {//l
     fac[0] = rg_gll_weight[ bc[1] ]*rg_gll_weight[ bc[2] ];

     for( bc[0] = 0 ; bc[0] < AXSZ ; ++bc[0] )
     {//m
      fac[1] = rg_gll_weight[ bc[0] ]*rg_gll_weight[ bc[2] ];
      fac[2] = rg_gll_weight[ bc[0] ]*rg_gll_weight[ bc[1] ];

      VFloat *local_loc = &(local[9*IDX3( bc[0], bc[1], bc[2] )]);

      const uint32_t *rg_gll_id_loc_loc =
              &(rg_gll_id_loc[VSize*IDX3(bc[0], bc[1], bc[2])]);

      for(int i=0;i<3;++i)
      {
       VFloat accu = vZero<VFloat>();
       int f=9;

       for(int c=0;c<3;++c)
       {
        VFloat *local_loc_mat = &(local_loc[(i*3+c)]);
        VFloat tmp = vZero<VFloat>();

        for(int u=0;u<AXSZ;++u)
         tmp = vMAdd(vSet1<VFloat>(
           rg_gll_lagrange_deriv[ IDX2( bc[c], u ) ]*rg_gll_weight[ u ] ),
           local_loc_mat[ f*(u-bc[c]) ], tmp);

        accu = vMAdd(vSet1<VFloat>(fac[c]),tmp,accu);
        f*=AXSZ;
       }//c

       vAccuGLLComp(rg_gll_id_loc_loc,i, accu);
      }//i
     }//m
    }//l
   }//k

  }//nIel
 }//col
```

```
}
```

Gauthier SORNET

Parallélisme des calculs numériques appliqué aux géosciences

Résumé :

La résolution discrète de modèle par calcul numérique profite à un nombre vertigineux d'applications autant industrielles que d'intérêt général. Aussi, il est nécessaire que les évolutions hétérogènes des machines soient exploitées par les logiciels de calcul. Ainsi, cette thèse vise à explorer l'impact du parallélisme d'architecture moderne à destination des calculs géoscientifiques. Par conséquent, ces travaux de thèse s'intéressent tout particulièrement aux catégories de noyaux de résolution stencil et éléments finis spectraux. Des travaux déjà réalisés sur ces noyaux consistent à structurer, découper et attaquer le calcul de façon à exploiter en parallèle les ressources des machines. De plus, ils apportent également des méthodes, modèles et outils d'analyse expérimentale. Notre approche consiste pour un noyau de calcul donné à y cumuler différentes capacités de parallélisme. En effet, les dernières évolutions de processeur Intel x86 disposent de multiples capacités de parallélisme superscalaire, vectoriel et multi-coeurs. Les résultats de nos travaux concordent avec ceux de la littérature. Tout d'abord, on constate que les optimisations vectorielles automatisées des compilateurs sont inefficaces. Par ailleurs, les travaux de cette thèse parviennent à affiner les modèles d'analyse. Ainsi, on observe une adéquation des modèles affinés avec les observations expérimentales. De plus, les performances atteintes dépassent des implémentations parallèles de références. Ces découvertes confortent la nécessité d'intégrer ces optimisations à des solutions d'abstraction de calcul. En effet, ces solutions intègreraient davantage la finalité des calculs à optimiser et peuvent ainsi les coupler davantage au fonctionnement des architectures cibles.

Mots clés : Vectorisation, Multi-coeurs, Model de performance, Stencil, Element finit

Parallelism of numerical computing applied to geosciences

Abstract :

Discrete model resolution by numerical calculation benefits a dizzying number of industrial and general interest applications. It is also necessary for heterogeneous machine evolutions to be exploited by calculation software. Thus, this thesis aims to explore the impact of parallelism in modern architecture on geoscientific calculations. Consequently, this thesis work is particularly interested in the categories of stencil resolution kernels and spectral finite elements. Work already carried out on these kernels consists in structuring, cutting and attacking the calculation in such a way as to exploit the resources of the machines in parallel. In addition, they also provide methods, models and tools for experimental analysis. Our approach consists in combining different parallelism capacities for a given calculation kernel. Indeed, the latest Intel x86 processor developments have multiple superscalar, vector and multi-core parallelism capabilities. The results of our work are consistent with those in the literature. First of all, we notice that the automated vector optimizations of compilers are inefficient. In addition, the work of this thesis succeeds in refining the analysis models. Thus, we observe an adequacy of the refined models with the experimental observations. In addition, the performance achieved exceeds parallel reference implementations. These discoveries confirm the need to integrate these optimizations into calculation abstraction solutions. Indeed, these solutions would better integrate the purpose of the calculations to be optimized and can thus better couple them to the functioning of the target architectures.

Keywords : Vectorization, Multi-core, Performance model, Stencil, Finite element

Géosciences pour une Terre durable
brgm

BRGM
3 Avenue Claude Guillemin
F-45100 ORLÉANS

Région Centre-Val de Loire

Centre Val de Loire
UNIVERSITÉ
Spirit of the Loire Valley

LIFO
Bâtiment IIIA, Rue Léonard de Vinci
B.P. 6759
F-45067 ORLÉANS Cedex 2

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS
LIFO