Parallélisation d'un code éléments finis spectraux. Application au contrôle non destructif par ultrasons

Carlos Carrascal Manzanares

École doctorale EDITE de Paris
Informatique, télécommunications et électronique

Spécialité: Informatique

THÈSE DE DOCTORAT

En vue de l'obtention du grade de :

Docteur de Sorbonne Université

présentée et soutenue le 8 février 2019 par :

Carlos Carrascal Manzanares

Parallélisation d'un code éléments finis spectraux. Application au contrôle non destructif par ultrasons.

Composition du jury :

Rapporteur	Alain MÉRIGOT	SATIE, Université Paris-Sud
Rapporteur	David DEFOUR	LAMPS, Université de Perpignan Via Domitia
Examinateur	Daniel ETIEMBLE	LRI, Université Paris-Sud
Examinateur	Fabienne JEZEQUEL	LIP6, Sorbonne Université
Directeur	Lionel LACASSAGNE	LIP6, Sorbonne Université
Encadrant	Vincent BERGEAUD	LIST, CEA

THÈSE

REMERCIEMENTS

Tout d'abord, je tiens à remercier Lionel Lacassagne pour la confiance qu'il m'a accordé en acceptant de diriger cette thèse. Je le remercie aussi pour ses conseils qui m'ont permis de mener è bien ces travaux de recherche.

Je remercie également Vincent Bergeaud pour m'avoir proposé cette thèse au sein de son laboratoire du DISC/LDI, ainsi que pour son encadrement et pour son soutien tout au long de cette thèse.

Merci également à Gilles Rougeron et Alexandre Impériale pour leur participation à la réalisation de ce travail. Les éléments finis ne sont pas toujours évidents et les coquilles hispaniques guettent depuis chaque ligne.

J'adresse aussi mes remerciements aux membres du jury qui ont accepté de juger mon travail.

Mes remerciements vont aussi aux collègues que j'ai rencontré pendant ces trois ans, faisant les journées plus agréables : Trini, Ahmad, Célia, Solenne, Roza, Alexis, Romain, Joëlle, Houssam, etc. Malgré les 10 mois sur 12 nuageux, les jours sont moins sombres s'il y a quelqu'un pour les partager. Je ne peux pas vous cacher que Netflix, Steam et le sushi à domicile aident aussi les jours de pluie...

Je n'oublie pas mes amis fidèles qui m'ont accompagné à distance dans cette aventure : Cristo, Sofi, Esther (pas loin !), Lalo, Navas, Leyre, Ramos (bien évidemment je ne peux pas vous mentionner tous !). Une aventure qui a commencé grâce à Marisa, Charo et Guillermo.

Enfin, j'adresse mes plus sincères remerciements à ma famille, ils m'ont soutenu dans mes pires moments moralement, physiquement et, surtout, gastronomiquement. Sans leur trafic des produits espagnols je n'aurais pu finir cette thèse. Mamá, papá, Cris, gracias.

RÉSUMÉ

Le contrôle non destructif (CND) est un ensemble de méthodes fondamentales du processus industriel pour assurer la qualité des objets en détectant les défauts à l'intérieur des pièces. Afin de concevoir et de qualifier ces méthodes, le Département Imagerie et Simulation pour le Contrôle (DISC) du CEA propose la plateforme de simulation CIVA pour modéliser le CND avec différentes techniques, dont les ultrasons. Pour prendre en compte un large éventail de géométries de défaut avec un haut niveau de précision, le logiciel CIVA-SFEM basé sur la méthode des éléments finis spectraux d'ordre élevé (SFEM) a été développé. Le coût de calcul de cette méthode numérique constitue un frein à son utilisation industrielle.

Le sujet de cette thèse consiste à étudier diverses pistes pour optimiser le temps de calcul de la méthode SFEM. L'objectif est d'améliorer la performance en se basant sur des architectures facilement accessibles, à savoir des processeurs multicœurs SIMD et des processeurs graphiques. Les noyaux de calcul étant limités par les accès mémoire (signe d'une faible intensité arithmétique), la plupart des optimisations présentées visent la réduction et l'accélération des accès mémoire. Une indexation améliorée des matrices et vecteurs, une combinaison des transformations de boucles, un parallélisme de tâches (*multithreading*) et un parallélisme de données (instructions SIMD) sont les transformations visant l'utilisation optimale de la mémoire cache, l'utilisation intensive des registres et la parallélisation multicœur SIMD. Les résultats sont concluants : les optimisations proposées augmentent la performance (entre ×6 et ×11) et accélèrent le calcul (entre ×9 et ×16). L'implémentation codée explicitement avec des instructions SIMD est jusqu'à ×4 plus performant que l'implémentation vectorisée. L'implémentation GPU est entre 2 et 3 fois plus rapide qu'en CPU, sachant qu'une connexion haut débit NVLink permettrait un meilleur masquage des transferts de mémoire.

Les transformations proposées par cette thèse composent une méthodologie pour optimiser des codes de calcul intensif sur des architectures courantes et pour tirer parti au maximum des possibilités offertes par le *multithreading* et les instructions SIMD.

ABSTRACT

Non-Destructive Testing (NDT) is a set of fundamental industrial process methods to ensure the objects quality by detecting defects within industrial parts. In order to design and qualify new NDT methods, the Département Imagerie et Simulation pour le Contrôle (DISC) within the CEA develops the CIVA simulation platform to model the CND with diverse techniques, including ultrasounds. In order to take into account a wide range of defect geometries with a high level accuracy, the CIVA-SFEM software based on the high order spectral finite element method (SFEM) has been developed. The cost of using this numerical method constitutes a major hindrance on its industrial use.

The subject of this thesis is to study numerous ways to optimize the SFEM computation time. The goal is to improve performance based on easily accessible architectures, namely SIMD multicore processors and graphics processors. As the computational kernels are limited by memory accesses (indicating a low arithmetic intensity), most of the optimizations presented are aimed at reducing and accelerating memory accesses. Improved matrix and vectors indexing, a combination of loop transformations, task parallelism (multithreading) and data parallelism (SIMD instructions) are transformations aimed at cache memory optimal use, registers intensive use and multicore SIMD parallelization. The results are convincing : the proposed optimizations increase the performance (between ×6 and ×11) and speed up the computation (between ×9 and ×16). The SIMDized implementation is up to ×4 better than the vectorized implementation. The GPU implementation is between two and three times faster than the CPU one, knowing that a NVLink high-speed connection will allow a correct masking of memory transfers.

The proposed transformations form a methodology to optimize intensive computation codes on common architectures and to make the most of the possibilities offered by multithreading and SIMD instructions.

PUBLICATIONS

Ce document s'inspire des publications scientifiques suivantes :

— Carrascal, C., Imperiale, A., Rougeron, G., Bergeaud, V., et Lacassagne, L. (2018). A Fast Implementation of a Spectral Finite Elements Method on CPU and GPU Applied to Ultrasound Propagation. *Advances in Parallel Computing*, pages 339–348

— Carrascal, C., Lacassagne, L., Chouh, H., Jubertie, S., Bergeaud, V., et Imperiale, A. (2019). Data layouts and threading strategies for Spectral Finite Elements for SIMD multi-core processors and GPUs. *The International Journal of High Performance Computing Applications* To be published.

Table des matières

Chapitre 1

Introduction

Ce document présente les travaux effectués dans le cadre de la thèse intitulée *Parallélisation d'un code éléments finis spectraux. Application au contrôle non destructif par ultrasons*. Cette thèse a été supervisée par le professeur Lionel Lacassagne de Sorbonne Université. Elle a été financée par le Département Imagerie et Simulation pour le Contrôle (DISC) du Commissariat à l'Énergie atomique (CEA LIST), les encadrants au CEA étant Vincent Bergeaud, Alexandre Impériale et Gilles Rougeron.

Un des axes majeurs d'activité du DISC est de proposer un ensemble d'outils de simulation et de modélisation pour le contrôle non destructif (CND). Ces outils sont intégrés au sein de la plateforme multi technique CIVA [Imperiale et al., 2018]. Parmi les différentes techniques disponibles figurent les ultrasons utilisés pour détecter les échos sur les défauts. Leur simulation est actuellement modélisée par une méthode semi-analytique [Leymarie et al., 2006], rapide, mais parfois insuffisante pour certaines configurations.

Afin de combler cette lacune, un logiciel basé sur la méthode des éléments finis (CIVA-SFEM) a été développé au sein du département. Du fait de sa souplesse, ce type de méthode permet de prendre en compte un grand nombre de spécificités de la géométrie et des matériaux présents dans la configuration de contrôle. Cependant, par nature, le coût de calcul des méthodes numériques est fatalement lié à la fréquence du signal de contrôle, qui est très élevée dans les cas concrets d'application.

Afin de pallier ces limitations, le logiciel se base sur la méthode des éléments finis spectraux d'ordre élevé combinée avec une stratégie de décomposition de domaine. Grâce à cette combinaison, le logiciel est basé sur une structure de données régulière très propice à la parallélisation (en tâches ou données) d'un grand nombre d'opérations numériques.

Le sujet de cette thèse consiste à étudier diverses pistes pour optimiser le temps de calcul du logiciel CIVA-SFEM. L'enjeu est de profiter de ses particularités afin d'atteindre la vitesse attendue par l'industrie dans un contexte d'utilisation de machines standards (PC ou station de travail) sans utiliser des architectures distribuées basées sur MPI. Les optimisations ont donc comme objectif d'améliorer la performance en se basant sur des architectures facilement accessibles, à savoir des processeurs multicœurs SIMD et des processeurs graphiques.

Après cette introduction, le chapitre 2 propose une présentation du contrôle non destructif, le domaine d'application pour lequel les travaux de cette thèse sont envisagés. La plateforme CIVA est présentée, en particulier le module de simulation sur lequel les travaux de thèse sont menés : CIVA-SFEM.

Afin de saisir le contexte informatique dans lequel ces simulations doivent être exécutées, le chapitre 3 s'écarte du domaine du CND pour aborder les architectures (CPU, MIC, GPU) et les outils informatiques disponibles pour l'obtention de performances élevées. Les choix d'architectures cibles, de compilateurs et de langages y sont présentés.

Le chapitre 4 décrit plus précisément CIVA-SFEM pour présenter la méthode des éléments finis spectraux, en se focalisant sur les propriétés de la méthode du point de vue de l'optimisation matérielle. La description du noyau de calcul et sa décomposition en tâches informatiques, en utilisant les connaissances du chapitre précédent, marquent la fin de ce chapitre.

Le chapitre 5 décrit les transformations réalisées dans l'écriture du noyau de calcul pour améliorer la performance et la vitesse de calcul. À la suite de l'état de l'art des publications traitant de la performance de calcul sur les méthodes éléments finies, on explicite la maquette numérique de référence et les modifications génériques qui peuvent avoir un impact fort sur la performance. La synthèse des combinaisons des modifications appliquées sur la maquette numérique nous permet d'enchaîner avec le chapitre 6. Celui-ci est consacré à la présentation des résultats des transformations du chapitre précédent.

Les conclusions obtenues grâce aux travaux de cette thèse sont présentées dans le chapitre 7, et le manuscrit se termine avec les perspectives des travaux futurs dans le chapitre 8.

Chapitre 2

La simulation en contrôle non destructif

Sommaire

Dans le domaine du contrôle non destructif (CND), l'usage de la simulation tend à se généraliser pour la mise au point de nouveaux contrôles ou pour l'analyse des résultats d'une acquisition. Cependant, le temps de calcul constitue encore un frein à cet usage. Ce premier chapitre présente succinctement le domaine d'application pour lequel les transformations et optimisations visant la réduction du temps de calcul sont envisagées.

Il rappelle le contexte du contrôle non destructif industriel ainsi que les méthodes d'inspection les plus couramment utilisées. L'application au contrôle par ultrasons et de santé intégrée est ensuite détaillée. Enfin, la plateforme CIVA est présentée, en particulier ses fonctionnalités en matière de simulation, pour enfin terminer avec le module de calcul sur lequel les travaux de thèse sont menés : CIVA-SFEM.

2.1 Le contrôle non destructif

S'assurer de la qualité des objets et outils produits a toujours été une préoccupation majeure du processus industriel. Les objets industriels produits, quel que soit le secteur d'activité (transport, énergie, pétrochimie...) sont tributaires du bon fonctionnement des composants qui les constituent, malgré les contraintes mécaniques que ceux-ci subissent. Il est nécessaire, pour des raisons de sécurité et de coût, de s'interroger sur l'état de santé de ces composants au cours de leur vie, afin d'intervenir avant d'atteindre la rupture. De même, dès la production, caractériser ces composants mécaniques permet de s'assurer de l'intégrité des ouvrages réalisés.

Du sommaire constatation de la résistance d'une structure à des méthodes plus poussées, plusieurs méthodes ont été élaborées au fil du temps pour tenter d'analyser une pièce mécanique a priori plutôt que d'atteindre un point de rupture. Le CND est un ensemble de méthodes qui permettent, par ordre de difficulté, de détecter, de localiser, d'identifier et de dimensionner des défauts dans des pièces ou des assemblages mécaniques, qu'ils soient en surface ou internes, sans en dégrader les propriétés d'usage.

2.1.1 Contexte industriel

Dans le cadre industriel, à partir des connaissances sur les contraintes subies par une pièce et un usage donné, des calculs de mécanique de la rupture permettent d'évaluer la nocivité d'un défaut. Toutefois, certains défauts peuvent être tolérés sous certaines conditions, sur une pièce donnée pour un usage spécifique. Il est ainsi possible de dresser un cahier des charges, par exemple en termes de taille critique, pour les défauts présents. Le domaine du CND ne consiste pas à se prononcer sur la nocivité d'un défaut, mais à répondre à un cahier des charges concernant la recherche de défauts dont la nocivité aura été établie a priori par une étude préalable.

Afin d'augmenter la production tout en s'assurant de la qualité des pièces manufacturées et de leur sécurité, le CND s'est développé conjointement à l'essor d'un grand nombre d'industries de pointe. Parmi les secteurs clés où ces méthodes sont utilisées de manière intensive, on trouve :

— L'industrie nucléaire : tout au long du cycle de vie d'une centrale, un ensemble de pièces doit être contrôlé régulièrement pour éviter les fuites de matières dangereuses. Ce contrôle régulier permet, par comparaison, de s'assurer que le matériel ne s'est pas endommagé au cours de son utilisation.

— L'industrie aéronautique : à la fabrication, les pièces complexes, souvent en matériaux composites pour leurs propriétés de rigidité et leur poids léger, sont longuement contrôlées. Le coût du contrôle représente de 10 à 30% de leur prix.

— L'industrie ferroviaire : les rails et les trains sont régulièrement contrôlés. Tous les ans, la SNCF inspecte près de 25,000 km de rails. Ces contrôles sont réalisés par des matériels roulants spécialisés.

— L'industrie sidérurgique : contrôle en fabrication de tôles, de tubes...

— La distribution : inspection de pipelines (eau, pétrole, chauffage urbain...).

2.1.2 Les différentes méthodes de contrôle les plus couramment utilisées

Le secteur industriel a vu croître ses besoins en contrôle, mais aussi se développer différentes méthodes de contrôle, répondant à des problématiques industrielles diverses. Le choix de la méthode dépend des contraintes portant sur le contrôle à effectuer.

Les premiers critères à prendre en compte sont le but du contrôle, le type de défaut à détecter ainsi que le besoin de le caractériser (détection, localisation, identification, dimension). Interviennent également des contraintes physiques : la nature des pièces inspectées (matériau, forme...) et l'environnement dans lequel doit avoir lieu le contrôle (contraintes thermiques, chimiques, de pression, radioactivité...). Enfin, le caractère économique du contrôle est à considérer qu'il s'agisse du coût même du contrôle (matériel et humain), mais aussi des besoins en termes de cadence, de durée et d'automatisation.

2.1.2.1 Le contrôle ultrasonore

Le contrôle ultrasonore consiste à explorer une pièce au moyen d'une onde mécanique ultrasonore (d'une fréquence en général comprise entre 200 kHz et 100 MHz). Les travaux réalisés dans le cadre de la présente thèse portent en particulier sur ce contrôle ultrasonore. L'onde ultrasonore est générée par un transducteur, le plus souvent piézoélectrique. Une impulsion électrique est convertie par le transducteur en vibration mécanique qui se propage dans la pièce jusqu'à se réfléchir sur les faces de celle-ci, comme la Figure 2.1 le montre. En présence d'un défaut, une partie du faisceau d'ondes ultrasonores est réfléchie prématurément et est renvoyée vers le capteur sous la forme d'un écho. Ce dernier reconvertit alors l'onde reçue en un signal électrique. L'opérateur peut ensuite analyser le signal reçu et caractériser le défaut en fonction de l'écho, cependant cette interprétation nécessite une très grande expertise.

Cette méthode permet de réaliser une détection en volume dans une pièce. Elle est particulièrement adaptée à la recherche de défauts de type fissures qui agit comme des réflecteurs. Elle apporte des informations de positionnement, de nature, et de dimension sur les différents défauts détectés. En fonction du positionnement et de l'orientation des

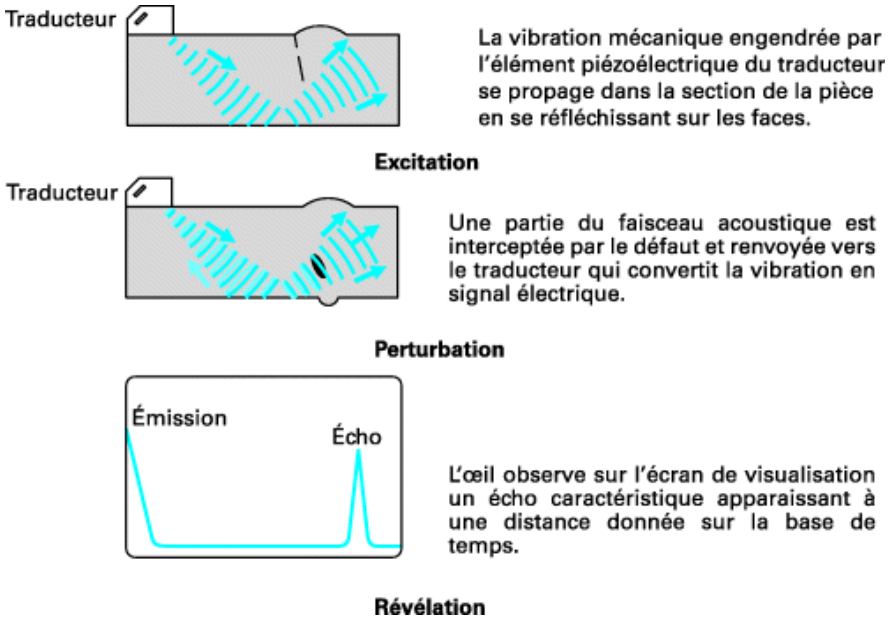


FIGURE 2.1 – Principe du contrôle par ultrasons. (Source : Techniques de l'ingénieur [référence BM6450 v1]).

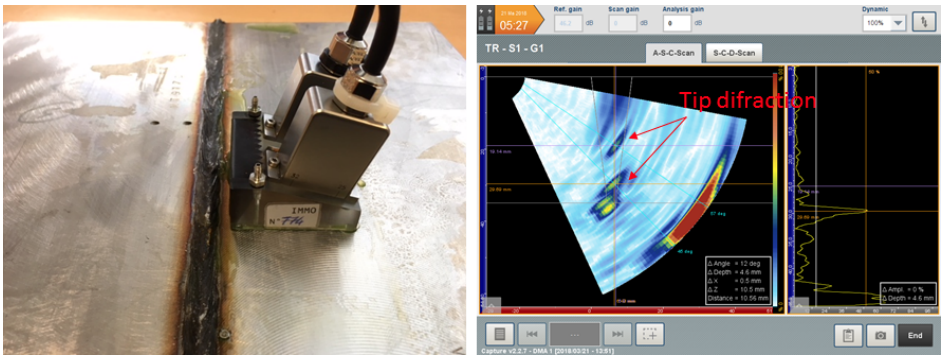


FIGURE 2.2 – Inspection d'acier et analyse avec un GEKKO (système portatif pour le CND). (Source : m2m.)

défauts par rapport au faisceau émis, la sensibilité du contrôle varie et ce dernier doit être optimisé. Les ultrasons se propageant difficilement dans l'air, on préfère utiliser un couplant lorsque le transducteur n'est pas au contact de la pièce : bien souvent, la pièce à inspecter est immergée dans de l'eau.

Le contrôle ultrasonore est très utilisé dans le cadre industriel, car c'est une méthode qui permet de réaliser des examens en profondeur de pièces de grandes dimensions et parce que la propagation des ondes ultrasonores n'est pas limitée par le type de matériau à contrôler (Figure 2.2). Les progrès en électronique ont permis le développement de capteurs complexes de type multi éléments. Plus généralement, l'essor de l'informatique a généralisé l'usage de la simulation pour mieux analyser des résultats d'inspection et faciliter la mise au point de nouveaux contrôles.

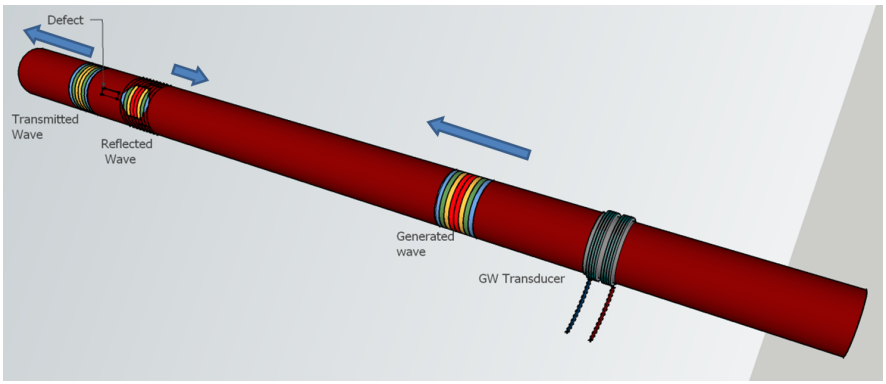


FIGURE 2.3 – Principe du contrôle par ondes guidées. (Source : Guided Wave Testing Europe.)

FIGURE 2.4 – Émetteur inspectant des voies ferrées par ondes guidées. (Source : CEA.)

2.1.2.2 Ondes guidées

Les ondes guidées sont des ondes ultrasonores transmises dans la structure contrôlée, qui se propagent au long de la structure, la propagation étant contrainte et guidée par les limites géométriques de la structure. Comme on peut voir dans Figure 2.3, les ondes guidées dans la conduite se propagent le long de l'axe et elles sont reflétées par les changements subis dans l'aire de la section transversale comme les défauts de corrosion ou les fissures, par exemple. Les tests par ondes guidées représentent une technique de contrôle non destructif pour la localisation des défauts en utilisant le temps d'arrivée, la vitesse de propagation des ondes guidées et le dimensionnement des défauts avec l'aide de l'amplitude des signaux. Les tests par ondes guidées utilisent des ultrasons à basse fréquence, habituellement entre 5 et 250 kHz pour générer des longueurs d'onde comparables ou supérieures à la taille du profil de la conduite dans le but d'acquérir une propagation à longues distances avec une atténuation réduite. Ce contrôle est notamment utilisé pour le contrôle des voies ferrées (Figure 2.4) et le contrôle des longs tuyaux de gaz ou hydrocarbures.

2.1.2.3 Le contrôle par courants de Foucault

Le contrôle par courants de Foucault (*Eddy currents* en anglais) consiste à utiliser les propriétés conductrices de la pièce inspectée pour étudier le comportement électromagnétique de celle-ci lorsqu'elle est soumise à une excitation donnée. Deux courants

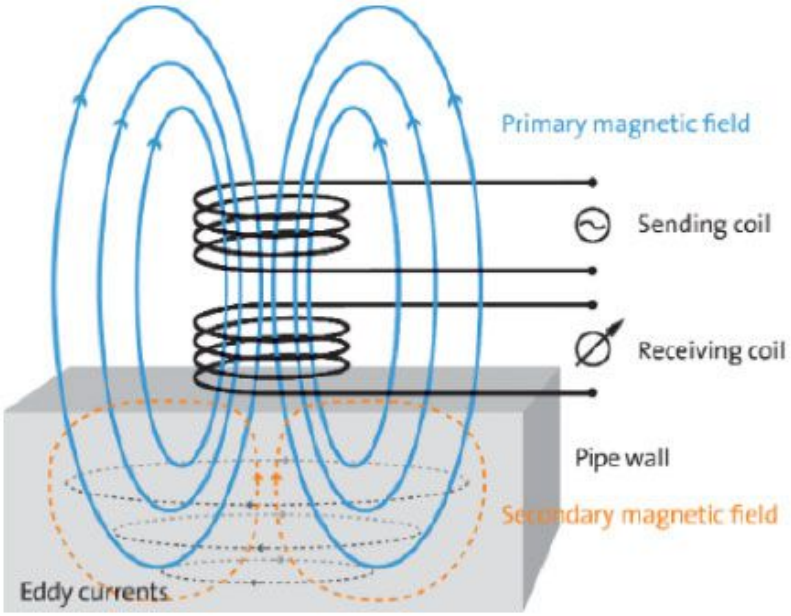


FIGURE 2.5 – Principe du contrôle par courants de Foucault. (Source : cmseddyscan.)

en opposition de phase sont induits dans deux sections voisines de la pièce par deux bobines (Figure 2.5) ; en l'absence de défaut cela forme un système équilibré. En présence d'un défaut, le système est déséquilibré et l'impédance de la bobine ou du capteur varie ; le signal obtenu par la chaîne de mesure peut être observé et analysé. Le champ magnétique utilisé dans le cadre de cette méthode est engendré par une onde sinusoïdale ou une onde à large bande passante dont la fréquence varie typiquement de 10 kHz à 10 MHz.

Le contrôle par courants de Foucault ne s'applique qu'aux matériaux conducteurs. La détection d'un défaut peut se faire en profondeur, mais est limitée par l'effet de peau des courants électriques. Elle est particulièrement adaptée aux géométries simples dans lesquelles le comportement électromagnétique est facilement modélisable. L'interprétation des signaux résultants est délicate et requiert un opérateur expert.

Cette méthode est entre autres utilisée pour le contrôle de produits de géométrie cylindrique (barres et tubes) avec l'aide de sondes (Figure 2.6). L'absence de contact entre la pièce et les bobines permet des cadences élevées (plus de 2m/s) et éventuellement sous l'eau ou à de très hautes températures. Elle est utilisée dans le cadre de l'industrie nucléaire pour le contrôle de tubes des générateurs de vapeur. Dans l'aéronautique, des pièces à fortes contraintes sont aussi inspectées telles que les bords d'attaques de l'aube d'un compresseur.

2.1.2.4 Le contrôle par radiographie

Dans le contrôle par radiographie, la pièce à inspecter est soumise à un rayonnement ionisant électromagnétique de type X. Celui-ci est issu d'une source placée d'un côté de la pièce. Sur le côté opposé est placé un film qui réagit au rayonnement. Lors de l'exposition, une image se forme sur le film après un temps donné ; celui-ci peut être

FIGURE 2.6 – Sondes à bobines pour l'inspection des tubes. (Source : ZETEC.)

observé pour analyse : en présence d'un manque de matière la pièce absorbe moins de rayonnement, ce qui induit une plus forte densité au niveau du film. Le film peut également être remplacé par un convertisseur qui transforme alors le rayonnement reçu en signal électrique, formant ainsi une image numérique. Cette méthode permet la détection des défauts en volume, et la nature de ceux-ci peut être observée sur l'image résultante. En revanche, cette méthode est coûteuse en investissement et l'usage d'un rayonnement ionisant impose des contraintes strictes de sécurité pour les opérateurs. De plus, le type de rayonnement influe sur sa capacité à pénétrer un matériau donné et comme dans le cas de la méthode par ultrasons, la détection d'un défaut dépend de son orientation par rapport au faisceau.

Cette méthode est utilisée dans de nombreux domaines industriels, car tous les matériaux peuvent être traversés par une source d'intensité suffisante. Elle s'applique principalement au contrôle d'un produit fini, dans des domaines aussi variés que la mécanique, l'électronique ou l'alimentaire. Les techniques de type tomographie associées à des méthodes de reconstruction par projection inverse des images numériques permettent d'obtenir des images volumiques d'une pièce.

2.1.2.5 Thermographie par infrarouges

Un apport de chaleur intense, suivi d'une analyse de la propagation de la chaleur, permet de mettre en évidence, à l'aide d'une caméra thermique, des défauts surfaciques ou sous-jacents tels que des fissures, des décollements de revêtements ou des délaminages. Le flux de chaleur peut être généré par méthode optique, acoustique, électrique ou par déformation mécanique. Ce principe est représenté par la Figure 2.9.

Des équipements de thermographie active ont été spécifiquement conçus pour répondre aux besoins des applications CND (Figure 2.10). La méthode d'échauffement choisie, le positionnement de la source et la durée de l'échauffement sont notamment fonction du matériau à inspecter, de son épaisseur ainsi que de la nature, de la position

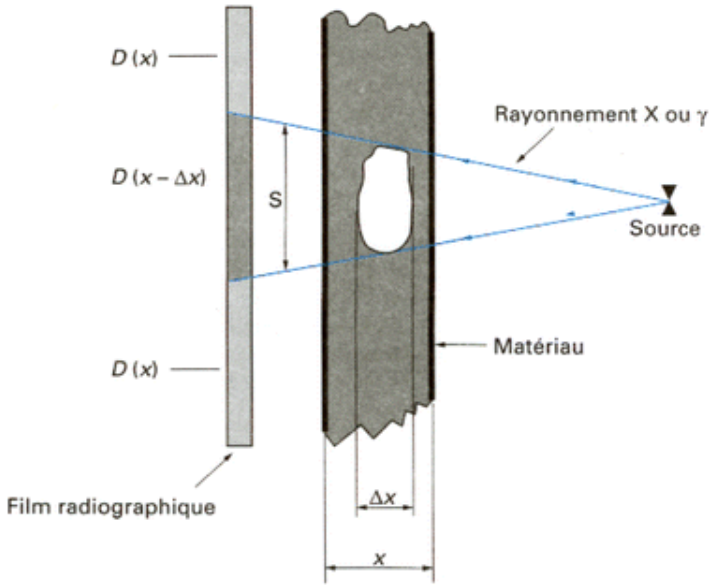


FIGURE 2.7 – Principe du contrôle par radiographie. (Source : Techniques de l'ingénieur [référence R1400 v1].)

FIGURE 2.8 – La plateforme CND par tomographie rayons X robotisée. (Source : CEA.)

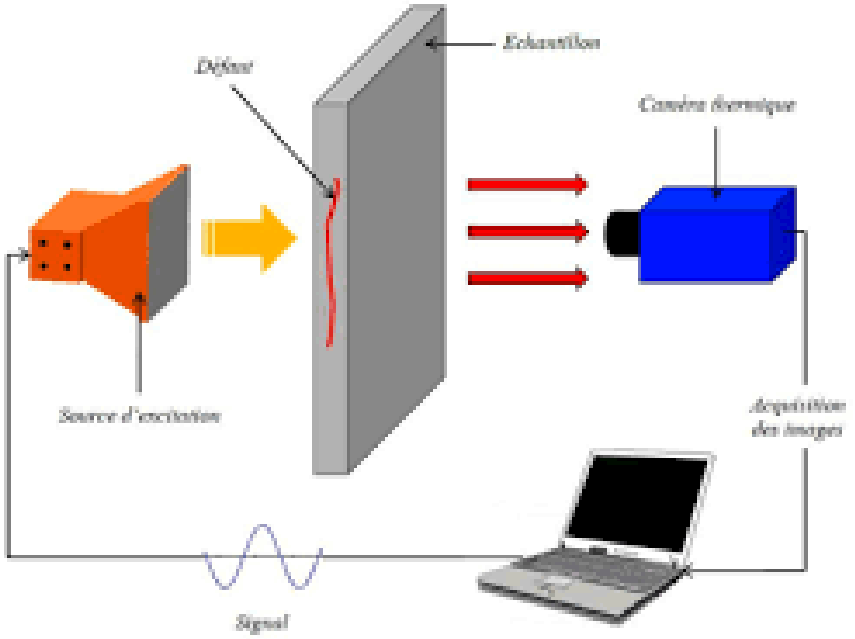


FIGURE 2.9 – Principe du contrôle par thermographie par infrarouges. (Source : Cetim.)

et de l'orientation du défaut attendu.

Il existe trois grands principes pour générer un flux de chaleur dans la pièce inspectée : une méthode optique par lampes halogènes et lampes flash, un procédé acoustique (avec un dispositif à ultrasons), et la génération d'un courant induit.

Bien qu'existant depuis de nombreuses années, la technique du contrôle par radiographie infrarouge a récemment connu un essor considérable du fait de la forte baisse des coûts des caméras infrarouges à haute résolution.

2.1.3 Usages industriels

Le contrôle non destructif est utilisé dans l'industrie afin de s'assurer de la qualité d'un objet, en matière de sécurité, de durabilité et de traçabilité. Il peut aussi permettre de vérifier la conformité des pièces contrôlées à un certain nombre de normes.

Dans certains contextes, il a été nécessaire d'élaborer de nouveaux contrôles sur des pièces en fonctionnement alors qu'ils n'avaient pas été prévus initialement. Par exemple, la production du parc nucléaire français varie en fonction de la demande, ce qui impose des contraintes sur un certain nombre de pièces. Celles-ci n'étaient pas prises en compte dans le plan de contrôle initial. Aujourd'hui, les contrôles tendent à être intégrés directement en ce qui concerne la conception d'une pièce afin d'envisager a priori les méthodes qui pourront être employées au cours de sa vie.

Cette prise en compte du contrôle dès la conception a renforcé le développement et l'usage de la simulation de contrôles non destructifs.

FIGURE 2.10 – Caméra thermique. (Source : Cetim.)

2.1.3.1 La simulation de contrôle

La simulation de contrôle se base sur une description de la pièce mécanique à inspecter, d'un couple émetteur/récepteur (capteurs en ultrasons, bobines en courants de Foucault...) ainsi que de défauts potentiels. Un modèle physique, variant selon la technique employée, sert de base au code de simulation numérique. La simulation est utilisée dans quatre buts principaux :

- Concevoir un contrôle lors de la création d'une pièce. En faisant varier indépendamment les paramètres intervenant lors du contrôle (capteur, défaut ou pièce), il est possible d'analyser en détail leur influence sur le contrôle à des coûts réduits.
- Qualifier un contrôle afin de démontrer les performances d'une méthode, c'est-à-dire sa capacité à répondre à un cahier des charges (imposé par exemple par une autorité de sûreté).
- Aider à l'interprétation en générant des résultats correspondant à une situation parfaitement maîtrisée, l'opérateur peut valider ou invalider son interprétation de résultats expérimentaux.
- Former des opérateurs en permettant au futur expert d'étudier de très nombreux cas.

2.2 Contrôle par ultrasons

L'utilisation d'une onde ultrasonore est l'une des méthodes pour réaliser un contrôle non destructif afin de détecter des défauts à l'intérieur d'une pièce mécanique. Un *capteur*, pouvant être à la fois *émetteur* et *récepteur* de l'onde ultrasonore, est placé à la surface du matériau à contrôler. L'onde émise se propage dans le matériau jusqu'à se réfléchir sur des interfaces délimitant des milieux d'impédance acoustique différente, c'est à dire ne propageant pas les ondes à la même vitesse. Lorsque les ultrasons réfléchis sont captés par le capteur récepteur, on parle de signal d'écho. On définit le temps de vol de l'onde comme le temps écoulé entre l'émission par le capteur de l'onde ultrasonore et la réception de l'écho par le ou les capteurs en réception. La mesure du

temps de vol et de l'amplitude du signal reçu permet de caractériser le défaut inspecté et peut ainsi conduire un expert à conclure sur l'altération des propriétés mécaniques de la pièce.

2.2.1 Principe d'un capteur ultrasonore

Un capteur ultrasonore est un capteur électroacoustique composé le plus souvent d'un ou plusieurs transducteurs émetteurs/récepteurs d'onde ultrasonore. Ces transducteurs sont les éléments actifs du capteur chargés des conversions du *signal électrique* en *signal acoustique* et inversement.

Plusieurs procédés permettent de générer ou de détecter des ultrasons :

— par l'utilisation de **lasers** pour exciter un matériau et détecter les ultrasons, résultats d'une extension thermique ou d'une ablation (de l'ordre du nanomètre) ;
— par **induction** électromagnétique ;
— par effet **électrostatique** (capacitif) ;
— par **magnétostriction**, procédé pour lequel la création d'un champ magnétique (par une bobine par exemple) cause la déformation (élongation ou contraction) du matériau et la création d'une onde ultrasonore ;
— par effet **piézoélectrique**, procédé le plus utilisé, découvert par les frères Curie et Curie [1880].

Un transducteur piézoélectrique, illustré en Figure 2.11, se compose principalement d'une lame piézoélectrique (une lame mince pour les composants dans la gamme du mégahertz et plus). Pour l'émission, un signal électrique met en vibration la lame par un effet piézoélectrique inverse. À la réception, la vibration ultrasonore crée un champ électrique par un effet piézoélectrique direct détecté par les électrodes présentes de part et d'autre de la lame. Ce type de transducteur est aussi bien adapté à l'émission qu'à la réception.

2.2.2 La simulation du CND

L'intérêt des simulations de contrôle est d'éviter de réaliser de multiples contrôles coûteux pour assister l'utilisateur dans la compréhension et l'analyse des phénomènes ultrasonores pendant une inspection. Il existe de nombreux logiciels commerciaux permettant des simulations de contrôle par ondes ultrasonores. Certains gros logiciels multiphysiques, reposant sur des codes d'éléments finis [Sayas, 2008], possèdent des modules de simulations acoustiques et piézoélectriques qui peuvent être utilisés pour le CND. Ces outils génèrent, à une calibration près, des résultats équivalents à ceux obtenus en acquisition. On peut citer parmi ceux-ci : *COMSOL* (COMSOL) et *Abaqus* (Dassault Systèmes). Dans cette même catégorie des codes utilisant les éléments finis pour modéliser la propagation de l'onde, on peut également citer deux outils plus spécifiquement dédiés au domaine du CND : *POGO* et *PZFlex*.

Le logiciel POGO [Huthwaite, 2014], *open source*, intègre des développements faits pour effectuer des simulations des éléments finis sur des problèmes en élastodynamique.

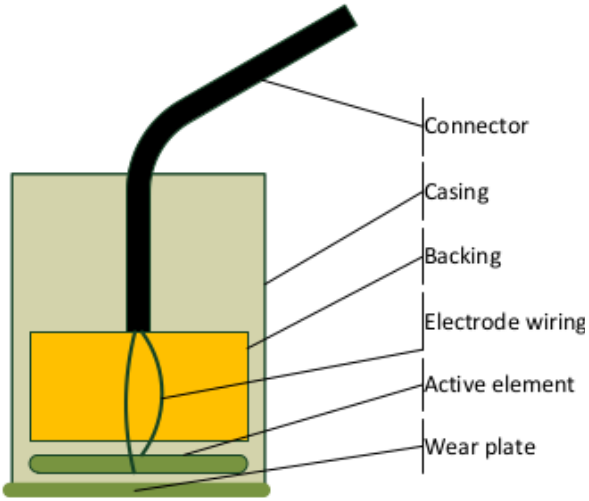


FIGURE 2.11 – Décomposition d'un transducteur piézoélectrique pour l'émission d'ondes ultrasonores. (Source : Lambert [2015].)

Il utilise des cartes graphiques NVIDIA en intégrant des noyaux de calculs codés en CUDA. Ce logiciel emploie une stratégie optimisée pour le chargement des données afin de diminuer la communication entre les multiples blocs de calculs exécutés par le GPU. Pour ce faire, il utilise un partitionneur de maillage aligné efficient et une réorganisation de mémoire à l'intérieur de chaque subdivision.

Basé sur un code d'éléments finis, *PZFlex* [Wojcik et al., 1993] a pour domaines, non seulement le CND, mais aussi l'échographie ultrasonore médicale et la thérapie par ultrasons. Récemment *OnScale* (www.onscale.com) a ajouté nouvelles fonctionnalités multiphysiques et la possibilité d'utiliser ses services dans une plateforme de calcul distribué *Cloud HPC*. Le code *PZFlex* est parallélisé sur processeur multicœur et les calculs peuvent être distribués sur plusieurs nœuds. Néanmoins, les temps de calcul sont souvent assez conséquents. La Figure 2.12 présente le front d'onde de polarisation transversale (dite ■onde T■) et les lobes de réseau visibles sur une capture d'écran de PZFlex, lors de la simulation de la propagation d'une onde T dans une pièce comprenant une soudure.

Des logiciels beaucoup plus légers basés sur des lancers de rayons sont disponibles. Parmi ceux-là, on peut citer *UTMan* (distribué par UTSim) dédié à la mise au point rapide de contrôles sur soudure. Il permet de déplacer interactivement un capteur et de visualiser la propagation d'un faisceau de rayons dans la pièce. La Figure 2.13 présente une inspection réalisée avec le logiciel UTMan.

Dans la même catégorie, on peut citer *ES Beam Tool* [Zhou et Derek Kim, 2016] (distribué par Eclipse Scientifique). Il permet à l'aide de nombreux outils de définir le type de capteur (mono ou multiéléments), la géométrie de la pièce (CAO ou soudures prédéfinies) et de placer des défauts. Puis, à l'aide de faisceaux de rayons, il permet de visualiser les zones insonifiées. Il donne ainsi rapidement une idée du positionnement idéal des capteurs pour un contrôle donné. Il permet aussi de visualiser des données métier comme la limite de champ proche pour un faisceau donné. Un module optionnel Beam-Tool Ascan permet d'obtenir une estimation des signaux d'échos spéculaires. La Figure 2.14 représente l'interface de ce module.

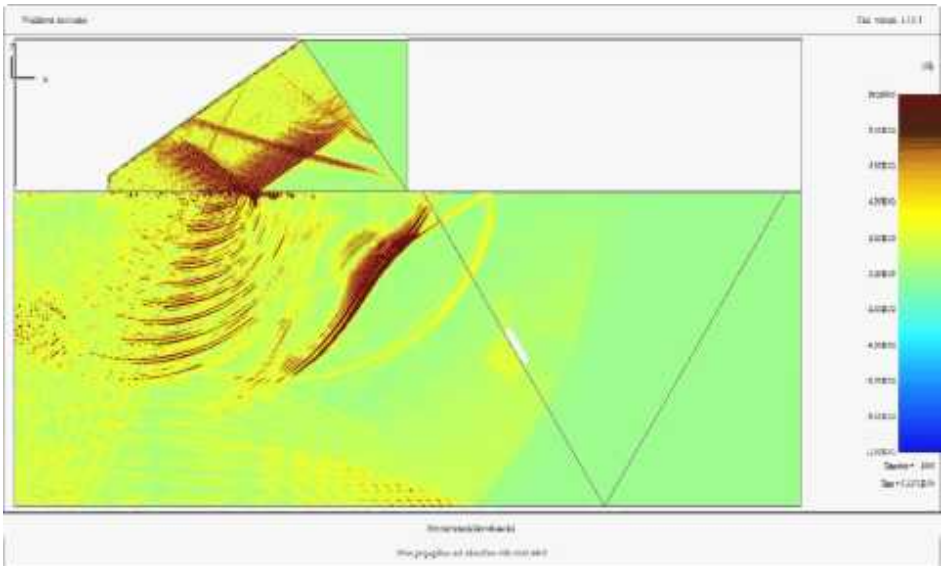


FIGURE 2.12 – PZFlex : Propagation d'une onde transverse ultrasonore émise par un capteur multiélément au contact dans une pièce comprenant une soudure.

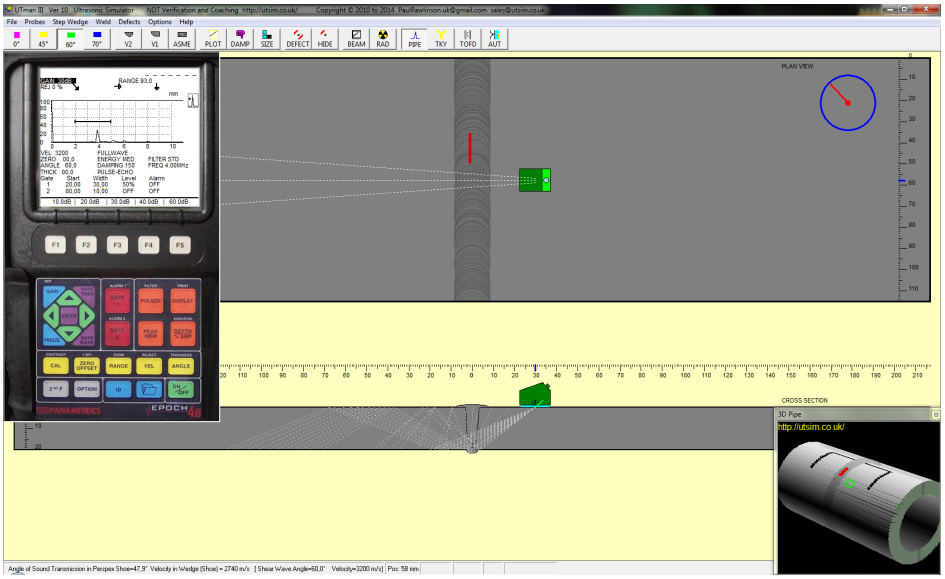


FIGURE 2.13 – UTMan : inspection d'une soudure raccord de deux tuyaux à l'aide d'un capteur mono élément que l'on peut déplacer à la souris.

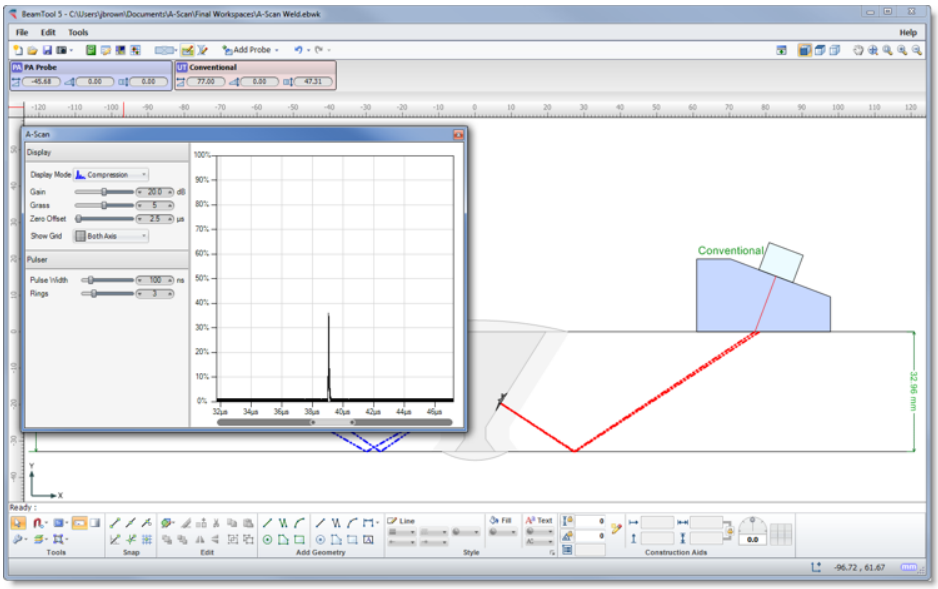


FIGURE 2.14 – Module Beam Tool AScan.

L'approche des noyaux de calcul principaux CIVA est une approche intermédiaire entre ces deux extrêmes (code de type éléments finis spectraux ou calculs rapides par lancer de rayons). Ses codes de simulation semi-analytiques [Darmon et al., 2011] permettent de traiter un grand nombre des cas d'intérêt pour la simulation du contrôle industriel. En revanche, il y a un certain nombre de situations où ces méthodes semi-analytiques ne sont pas suffisantes. Il faudra donc faire un compromis entre la vitesse apportée pour la méthode existante et la polyvalence d'une autre méthode : la méthode des éléments finis spectraux.

2.3 Contrôle santé intégré

Les CND sont coûteux non seulement en termes d'équipement et de personnel, mais également en raison de l'indisponibilité qu'ils entraînent (par exemple due au démontage de la pièce à inspecter, à l'arrêt du réacteur ou du pipeline). De plus, la géométrie parfois complexe de ces structures rend difficile l'inspection de certaines zones, réduisant la fiabilité du contrôle.

Une approche plus récente consiste à intégrer directement des capteurs à la structure étudiée et aux laisser-sur-place, afin de suivre de manière très régulière (possiblement en temps réel) l'évolution de l'état de santé de la structure. On parle alors de contrôle santé intégré (*Structural Health Monitoring* en anglais) (SHM). Le SHM est un ensemble de techniques permettant d'*intégrer des capteurs et des actionneurs* dans une structure dans le but d'enregistrer, analyser et localiser des endommagements de cette structure, selon Balageas et al. [2006]. Dans la Figure 2.15 on observe une pièce carrée d'aluminium avec un défaut et 8 émetteurs / récepteurs intégrés.

À tout instant, le système SHM doit être capable de répondre à cinq niveaux d'informations (basés sur une classification initialement proposée par Rytter [1993]) sur

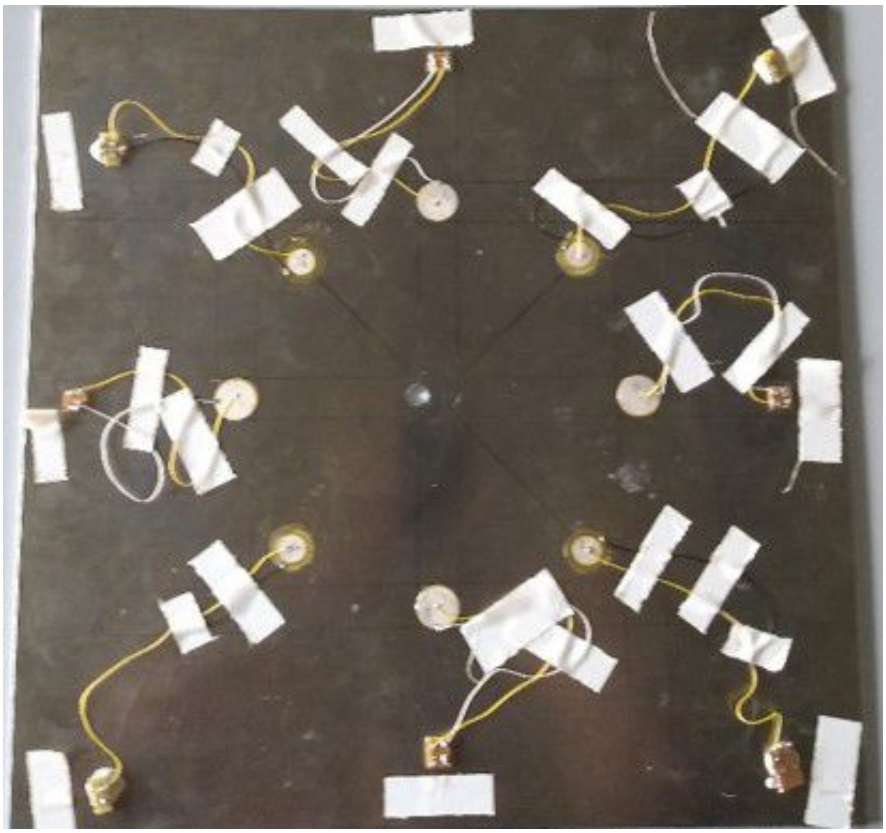


FIGURE 2.15 – Pièce d'aluminium de taille 400×400×3 mm avec un trou et 10mm 8 émetteurs / récepteurs intégrés. (Source : CEA.)

l'état de santé de la structure contrôlée :

1. **Détection** Indication qualitative sur la présence d'un défaut ou non. Cela se manifeste en général par une alarme.
2. **Localisation** Indication de la position probable du défaut.
3. **Classification** Indication sur le type de défaut détecté.
4. **Taille** Estimation de la géométrie du défaut.
5. **Pronostic** Prédiction sur la santé de la structure. Estimation de la durée de vie résiduelle.

Pour les quatre premières fonctions, qu'on pourrait synthétiser comme *diagnostic*, le SHM est très proche du CND puisque ses moyens physiques d'investigation sont similaires. Mais le SHM cherche à aller plus loin que le CND. Une première raison à cela concerne le dernier niveau d'information, c'est-à-dire accéder à la durée de vie résiduelle. Cette notion de pronostic fait partie intégrante de la démarche de déploiement d'un système SHM en lien direct avec la stratégie de maintenance. Ensuite, le caractère intégré du SHM introduit certaines contraintes empêchant d'utiliser à l'identique des techniques de CND. En effet, le CND se base généralement sur des méthodes consistant à déplacer un capteur sur la structure inspectée ou sur des équipements de grandes dimensions généralement déportés de la structure. En SHM, tout cela n'est pas possible puisque les capteurs sont intégrés.

En fournissant des informations en temps réel sur la structure qu'il surveille, un système SHM permet de réduire les temps d'immobilisation et d'apporter une sécu-

rité en diminuant le nombre d'accidents. Le SHM permet aussi de revoir la conception en évitant de surdimensionner la structure, ce qui entraînerait des limitations de performances (notamment en aéronautique où la masse est un facteur clé). De plus, un système SHM peut inspecter des zones qui seraient inaccessibles avec des méthodes de CND classiques. Ce cas se présente pour des pièces nécessitant que la structure soit démontée puis remontée pour y accéder (cuves de réacteurs nucléaires, plateformes pétrolières ou éoliennes en mer).

Avec un système SHM, il est également possible d'effectuer de la maintenance prévisionnelle. En effet, en suivant le vieillissement de la structure en temps réel, ces technologies permettent d'anticiper les défaillances, ce qui présente un réel intérêt en matière logistique (acheminement de pièces de rechange ou de personnel qualifié). Enfin, on peut imaginer déclencher les opérations de maintenance uniquement sur alertes du système SHM, ce qu'on appelle la maintenance conditionnelle.

Il est illusoire de remplacer totalement le CND par le SHM, car il semble impossible de concevoir un ensemble de systèmes contrôlant la totalité de structures pouvant être très complexes. De plus, dans certains cas les performances du SHM ne sont pas satisfaisantes (la finesse du diagnostic peut être insuffisante), alors il pourra servir de prédiagnostic pour un CND ciblé et donc bien plus efficient. SHM et CND sont donc complémentaires.

2.4 La plateforme CIVA

Le logiciel CIVA est une plateforme multitechnique d'expertise pour le contrôle non destructif.

2.4.1 Présentation générale de CIVA

La plateforme CIVA comporte des modules de simulation, d'imagerie et d'analyse pour à la fois concevoir et optimiser des procédés de contrôle ainsi que prédire leurs performances dans des configurations de contrôle réalistes.

CIVA prend en charge des contrôles par ultrasons (UT), ondes guidées (GWT), courants de Foucault (ET), radiographie et tomographie (RT), thermographie par infrarouges (IT) et contrôle santé intégré (SHM). Ces options sont visibles en la page d'accueil de CIVA, dont une capture d'écran est en Figure 2.16.

Les simulations réalisées par le logiciel CIVA permettent de prendre en compte la plupart des paramètres influents sur un contrôle (capteur, géométrie de la configuration, matériaux de la pièce inspectée, types de défauts recherchés). Ainsi il est possible, dès la conception d'une pièce, d'anticiper les contrôles nécessaires en minimisant la réalisation de maquettes coûteuses tout en maximisant la fiabilité de ces contrôles en qualifiant la performance et la pertinence d'une configuration donnée. CIVA offre également le calcul des courbes *Probability Of Detection* (POD), à partir de nombreuses simulations se basant sur la prise en compte d'incertitudes sur des paramètres d'entrée d'une configuration. La simulation entre également en jeu lors de la conception d'un

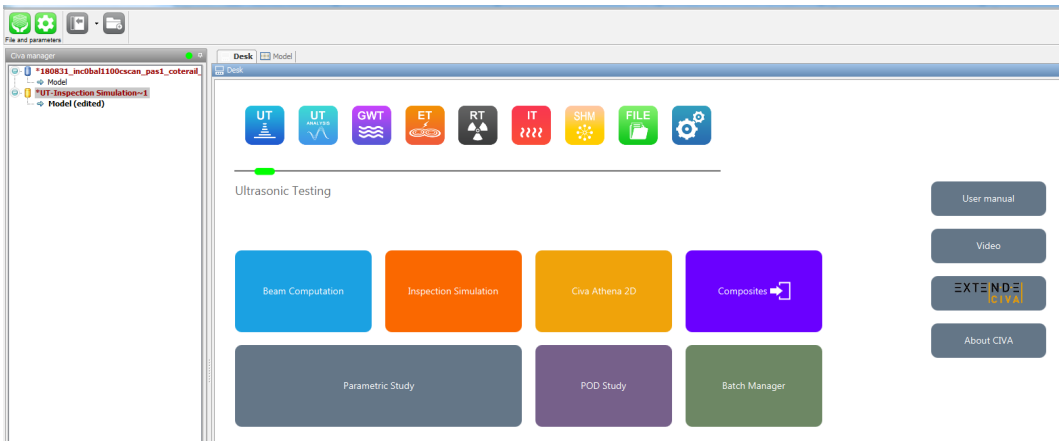


FIGURE 2.16 – Page d'accueil de CIVA. (Source : CEA.)

capteur, en guidant celle-ci, en particulier dans le cadre des technologies multiéléments ou *Electromagnetic Acoustic Transducer* (EMAT), pour générer des ondes ultrasonores au moyen d'une excitation électromagnétique.

La plateforme CIVA est utilisée dans de multiples domaines : du nucléaire à l'aéronautique, des transports à la métallurgie, en passant par l'aérospatiale et la pétrochimie...

2.4.2 Développement et distribution

Le logiciel CIVA est développé par le Département Imagerie et Simulations pour le Contrôle (DISC) du CEA-LIST, composé d'une équipe d'environ 120 personnes. Le département s'organise autour de trois axes :

— la modélisation de contrôles non destructifs par ultrasons, par méthodes électromagnétiques et rayons X ;

— le développement de capteurs innovants pour le contrôle par ultrasons et par méthodes électromagnétiques ;

— l'expertise technique et le support auprès des partenaires industriels.

2.4.3 CIVA, un modèle complet

Le logiciel CIVA, actuellement disponible dans sa version 2017, s'est construit sur une complexification progressive des contrôles proposés à la simulation et à l'analyse. Un des atouts de CIVA est de proposer un modèle semi-analytique des contrôles par ultrasons permettant d'accélérer les simulations.

CIVA possède un ensemble de modèles suffisamment génériques pour simuler de nombreuses configurations. Les simulations, outre leur aspect de mise au point des contrôles, peuvent aussi être utilisées lors de la phase d'analyse. Par exemple, simuler le contrôle théorique permet de mieux distinguer le signal du bruit provenant des conditions expérimentales, ou encore d'utiliser les paramètres de configuration déterminés

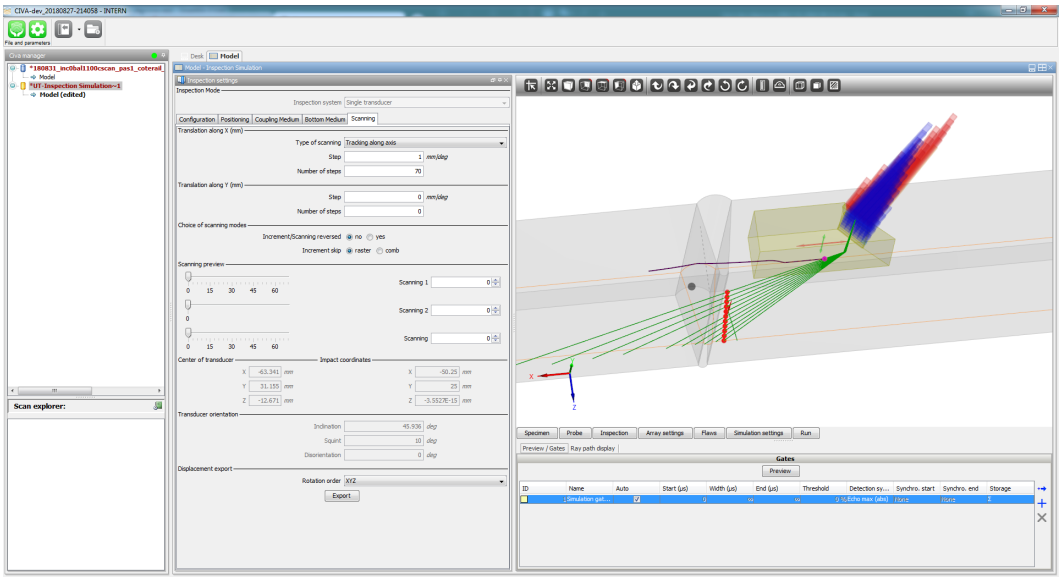


FIGURE 2.17 – Page de simulation d'inspection de CIVA-UT. (Source : CEA.)

lors d'une simulation de mise au point.

CIVA propose des modules utilisateur concernant les techniques suivantes :

2.4.3.1 Contrôles ultrasonores

Le module UT de CIVA simule la propagation de faisceaux ultrasonores et les interactions faisceau/défaut et faisceau/géométrie de la pièce (échos de fond, de surface, de coin, de diffractions, ombrages...) pour des pièces dont les matériaux et la géométrie sont potentiellement complexes. Il dispose d'une large variété de défauts et de capteurs modélisés, permettant de couvrir de très nombreux cas (Figure 2.17). Ce module est chargé notamment du traitement des matériaux composites (un assemblage d'au moins deux composants non miscibles dont les propriétés se complètent, possédant des propriétés que les composants seuls ne possèdent pas).

2.4.3.2 Radiographie et tomographie

Le module RT calcule le rayonnement diffusé ou direct d'une source dans une pièce pouvant contenir un ou plusieurs défauts. Les sources simulées peuvent être une source X ou une source gamma. Le module CT vient compléter le module RT de CIVA en offrant la possibilité de réaliser plusieurs simulations RT pour reconstruire en 3D la configuration.

2.4.3.3 Courants de Foucault

Le module CF sert à prédire le champ électromagnétique induit dans la pièce. Il est possible de calculer le diagramme d'impédance normalisé d'un capteur donné et de simuler la réponse d'un défaut.

2.4.3.4 Thermographie par infrarouges

Le module IT permet simuler un apport de chaleur dans une pièce, fixant la source du flux. La propagation de la chaleur est visible à l'extérieur et à l'intérieur de la pièce.

2.4.3.5 Ondes guidées

CIVA dispose d'un module GWT pour la simulation de la propagation d'ondes dans des guides d'ondes plans et tubulaires ainsi que l'interaction de ces faisceaux avec des défauts présents sur le guide.

2.4.3.6 Couplage avec le code ATHENA

Le module ATHENA2D permet de réaliser un couplage entre les codes d'éléments finis ATHENA (développés par EDF) et les modèles UT analytiques de CIVA. Ainsi, le modèle CIVA permet de décrire rapidement les conditions de la propagation du faisceau d'ondes dans la pièce aux frontières de la zone d'intérêt. Au sein de celle-ci, le modèle d'éléments finis prend le relais et calcule le comportement du faisceau.

2.4.3.7 SHM

Pour accompagner le développement de la thématique SHM au CEA, le département a récemment créé un module appelé CIVA-SHM qui permet de lancer des simulations sur des configurations représentatives du SHM (Figure 2.18). Il s'agit de pièces planes ou cylindriques, avec des matériaux métalliques ou composites. Ces plaques sont instrumentées par des réseaux de capteurs piézoélectriques. CIVA donne la possibilité de créer un certain nombre de défauts (trous de rivet, surépaisseur, délaminage) et de simuler la propagation des ondes sur chaque couple émetteur / récepteur de capteurs.

2.4.4 Vision commerciale

Les différents modèles de CIVA sont régulièrement validés à l'aide de mesures réalisées dans le cadre de contrôles existants. En particulier, avant la mise à disposition commerciale d'un nouveau modèle, ce dernier est testé et une étude de validation est présentée lors de congrès sur le contrôle non destructif.

CIVA propose également à ses clients tout un éventail de personnalisations. Tout d'abord, ceux-ci peuvent utiliser leurs propres modèles spécifiques à travers un système de plug-in afin de bénéficier de la puissance de l'imagerie et de la modélisation CIVA. De plus, en fonction de ses besoins, un utilisateur peut acquérir uniquement les modules qui lui sont nécessaires. Ainsi, il existe CIVA-E, une version simplifiée de CIVA à destination des universités et des centres de formation.

Enfin, CIVA est en cours de déclinaison sur plateforme embarquée pour faciliter l'analyse et la réalisation de contrôles sur site. Cette plateforme permet de bénéficier de toute la puissance d'analyse de CIVA sous la forme d'une interface tactile plus facile

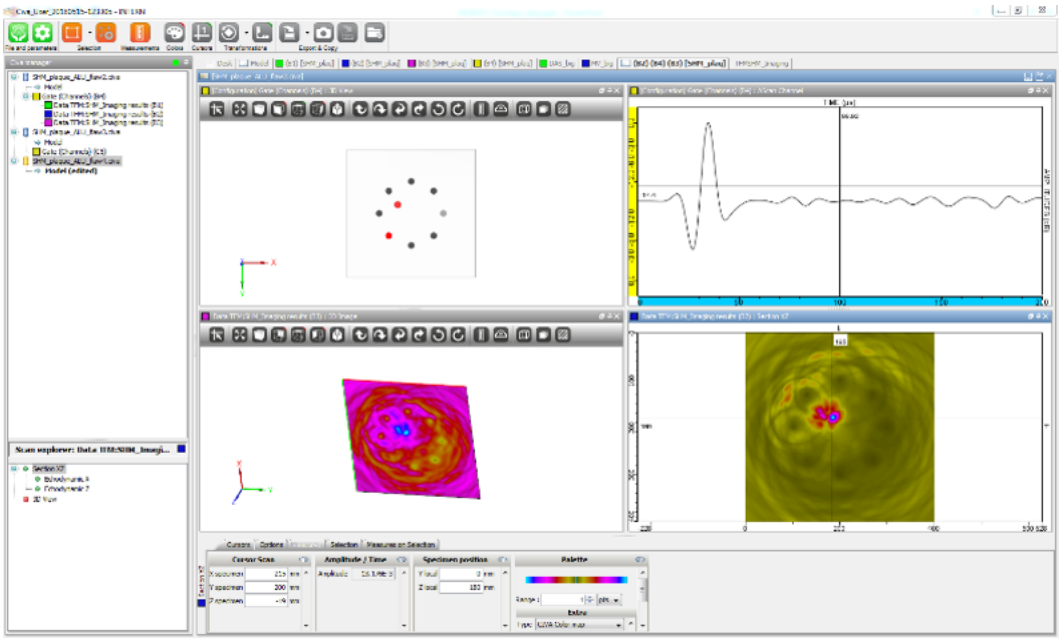


FIGURE 2.18 – Page de simulation d'inspection de CIVA-SHM. (Source : CEA.)

à appréhender.

2.5 CIVA-SFEM

Les solutions semi-analytiques développées dans CIVA permettent de traiter un grand nombre de cas d'intérêt pour la simulation du contrôle industriel. Néanmoins, il existe un certain nombre de situations dans lesquelles ces modèles s'avèrent insuffisants :

— le modèle semi-analytique de CIVA suppose que la taille du défaut est supérieure à la longueur d'onde ;

— les ondes rampantes (ondes se propageant à la surface des pièces) ne sont pas prises en compte par le modèle rayon ;

— les modèles semi-analytiques travaillent par mode de propagation, ce qui fait qu'ils sont mal adaptés dans des situations où de très nombreux modes doivent être pris en compte pour avoir une simulation fidèle à la réalité.

Depuis plusieurs années, le CEA est donc engagé dans le développement d'un code éléments finis, CIVA-SFEM [Imperiale et al., 2018] qui permet de lever ces limitations. Il est notamment utilisé dans deux contextes au sein de CIVA : l'amélioration des modèles de propagation des ondes ultrasonores (CIVA-UT) et les calculs de la thématique SHM (CIVA-SHM).

2.5.1 Calcul UT

Le module de CIVA-UT met en place une stratégie de couplage [Choi et al., 2014; Leymarie et al., 2006] entre un modèle semi-analytique et CIVA-SFEM. Le principe est simple et montré en Figure 2.19 : les lancements de rayons dans la région de la pièce éloignée du défaut sont traités selon le modèle plus rapide semi-analytique, puis CIVA-SFEM se charge du calcul dans la région d'intérêt, entourant le défaut. Entre ces deux étapes de calcul, CIVA-UT va convertir les résultats des lancers de rayons sur l'interface entre les deux régions vers les conditions initiales sur le bord (voir sous-section 4.1.1) requis pour la méthode des éléments finis spectraux. De plus, entourant la région d'intérêt, une couche absorbante artificielle parfaitement adaptée (en anglais *perfectly matched layer* ou PML) est placée pour simuler une frontière ouverte et éviter la réflexion des ondes.

En effet, habituellement, les défauts sont de plusieurs ordres de grandeur plus petits que la pièce. Utiliser la méthode SFEM partout dans la pièce serait inefficiente quand seulement la région proche du défaut est susceptible d'en bénéficier.

Cette stratégie de couplage est assez proche de celle déjà mise en place dans CIVA pour le couplage avec le code ATHENA. Néanmoins, l'introduction de CIVA-SFEM offre de nouvelles perspectives. En effet, CIVA-SFEM est plus rapide grâce à la stratégie des éléments finis spectraux, qui permettent l'utilisation d'éléments finis d'ordre élevé. Cela rend possible des calculs éléments finis 3D qui sont inaccessibles avec ATHENA. CIVA-SFEM est également capable de prendre en compte des géométries plus précises, en adaptant le maillage aux parois de la pièce et aux défauts.

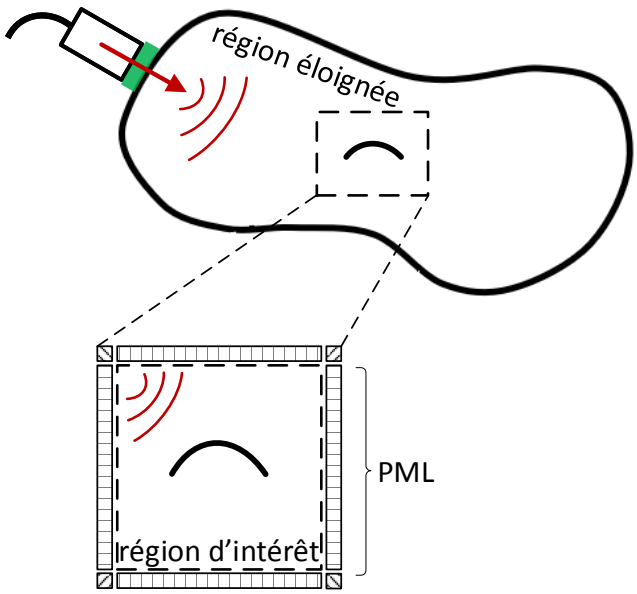


FIGURE 2.19 – Stratégie de couplage de CIVA-UT entre un modèle semi-analytique et la méthode des éléments finis spectraux de CIVA-SFEM.

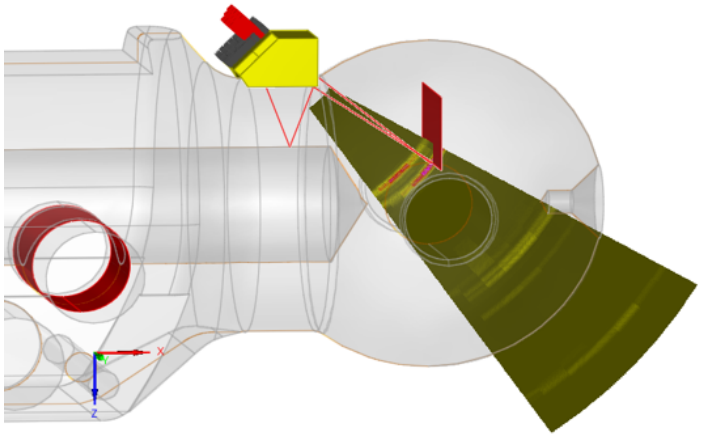


FIGURE 2.20 – Exemple d'une simulation de CIVA-UT en utilisant CIVA-SFEM.

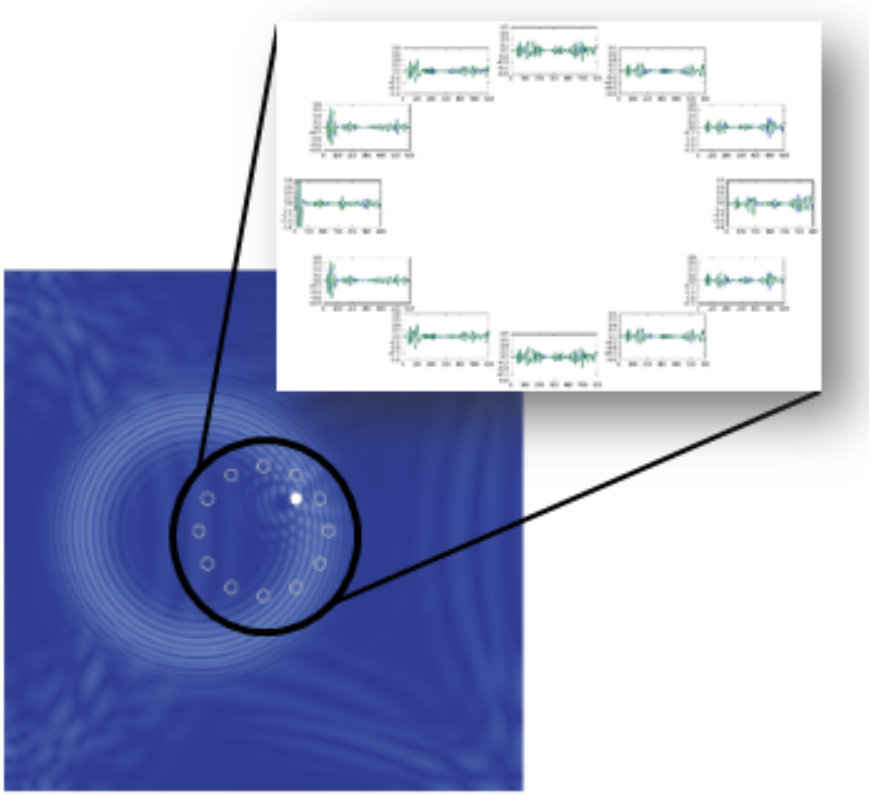


FIGURE 2.21 – Onde émise par l'émetteur de gauche et les ondes reçues ailleurs.

2.5.2 Calcul SHM

Le module CIVA-SHM exploite les capacités de calcul de CIVA-SFEM d'une manière différente. Contrairement au calcul des ultrasons décrit dans le paragraphe précédent, le calcul ne met pas en œuvre de couplage avec les algorithmes basés sur les méthodes rayons de CIVA. Ce module utilise plusieurs calculs de CIVA-SFEM pour obtenir par imagerie des reconstructions tomographiques permettant des comparaisons entre l'expérience et la simulation.

Prenons comme exemple la pièce d'aluminium de la Figure 2.15. S'il y a N émetteurs / récepteurs, CIVA-SFEM va simuler la propagation de l'onde entre chaque paire de capteurs, donc un total de N^2 simulations. Du fait de cette dépendance quadratique au nombre de capteurs, rapidement, le volume des simulations s'accumule. Il est donc crucial que le module CIVA-SFEM soit assez rapide. À partir de l'ensemble des N^2 signaux ainsi simulés, CIVA-SHM va procéder à une reconstruction tomographique de l'image, permettant de localiser les défauts (fissures, trous, surcroît ou perte d'épaisseur) dans la plaque. Dans la Figure 2.21 on observe la simulation de l'onde émise par l'émetteur le plus à gauche et les signaux obtenus au niveau de chaque récepteur (incluant lui-même, ce qui donne un total de 12). Une fois appliquée l'imagerie sur l'ensemble des signaux simulés entre chaque paire, on obtient la reconstruction de la Figure 2.22.

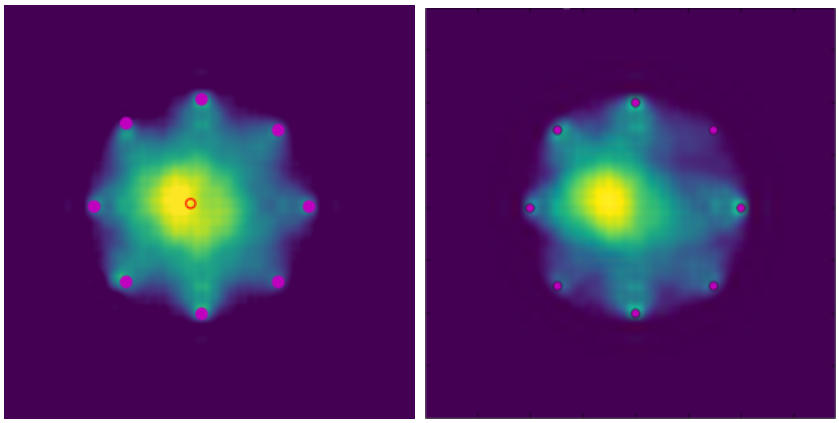


FIGURE 2.22 – À droite, résultat de l'imagerie sur la plaque de la Figure 2.15. À gauche, le résultat de l'expérimentation.

2.6 Synthèse du chapitre

Ce chapitre a mis en évidence les exigences industrielles du CND, en particulier par ultrasons. Un panorama des outils de simulations de contrôles par ultrasons a été présenté, des logiciels les plus simples et plus rapides aux outils disposant de modèles complets, mais bien plus lents. En particulier, la plateforme CIVA a été présentée, tant son module de simulation des ultrasons (CIVA-UT) que son module de contrôle de santé intégré (CIVA-SHM). Ces deux modules utilisent CIVA-SFEM, la méthode des éléments finis spectraux pour obtenir des résultats dans des configurations pour lesquelles le modèle semi-analytique de CIVA est insuffisant.

C'est en vue de rendre compétitifs dans un contexte industriel ces modules de simulation que ces travaux de thèse ont été menés. Afin de saisir le contexte informatique dans lequel ces simulations doivent être obtenues, le chapitre suivant traite des architectures disponibles pour obtenir des performances élevées, ainsi que des contraintes d'implémentation associées.

Chapitre 3

Architectures parallèles et outils de programmation

Sommaire

Ce chapitre s'écarte du domaine du contrôle non destructif pour aborder les outils informatiques disponibles pour l'obtention de simulations rapides. Les travaux réalisés dans cette thèse ont pour objectif de fournir des simulations utilisant des éléments finis spectraux rapides sur les ordinateurs standards (utilisant des CPU et GPU). Dans le cadre industriel, les utilisateurs et experts CND travaillent sur des stations de calculs puissantes, mais disposant de composants généralistes.

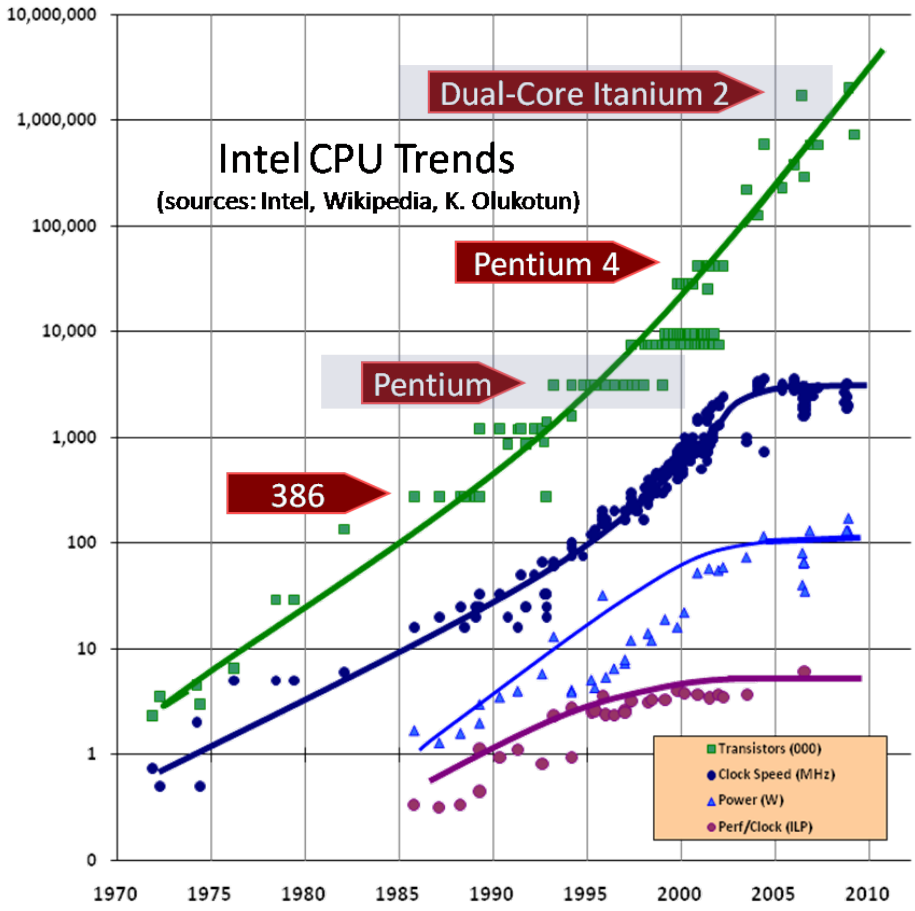


FIGURE 3.1 – Historique des CPU Intel. (Source : Sutter [2005].)

3.1 Panorama des architectures parallèles

Depuis la fin des années 1960, le matériel informatique suit la loi empirique énoncée par Gordon Moore indiquant que la densité de transistors sur une puce de silicium double approximativement tous les deux ans [Moore, 2006]. Même s'il est prévu que des phénomènes physiques risquent d'engendrer des problèmes à l'horizon 2020 (technologie de gravure, effets quantiques, désintégration alpha...), le matériel disponible jusqu'à aujourd'hui a suivi cette tendance.

Cette loi de Moore se traduit, au niveau des ordinateurs, par un doublement de nombre des transistors interprété comme un doublement de la puissance de calcul. Jusqu'à 2004, cette augmentation était obtenue principalement par l'augmentation de la fréquence de calcul. Depuis, les constructeurs ont arrêté d'accroître la fréquence de fonctionnement des processeurs en raison des contraintes thermiques (3.1). Ces architectures s'orientent aujourd'hui vers des matériels proposant des capacités de calcul parallèle permettant l'exécution de plusieurs traitements simultanément à des fréquences devenues quasi constantes au fil des années.

3.1.1 Supercalculateurs

Dans le domaine de la simulation informatique, les machines les plus emblématiques sont les superordinateurs. Depuis longtemps, ils disposent de capacités de calcul parallèle en regroupant plusieurs sous-machines afin de les faire collaborer pour résoudre un problème donné. Il s'agit de machines imposantes, visant à atteindre les plus hautes performances de calcul possible par rapport aux technologies disponibles.

Ces installations regroupent plusieurs nœuds de calcul en une grille de calcul pour leur permettre de collaborer et de communiquer à travers le réseau. Les nœuds sont composés d'un ou plusieurs processeurs partageant une mémoire commune et souvent de coprocesseurs (aujourd'hui, les plus rapides utilisent des GPU ou des MIC). L'ensemble des nœuds est installé dans des bâtiments spécifiques pour répondre aux contraintes d'alimentation électrique, de refroidissement et d'interconnexion. Aujourd'hui, à plein régime, le plus gros d'entre eux consomme près d'une vingtaine de mégawatts, soit l'équivalent d'une petite ville de 15,000 habitants. L'utilisation de ces superordinateurs a donné naissance à la discipline du calcul à haute performance (ou *High Performance Computing*, HPC).

Depuis 1993, deux fois par an, le projet `TOP500` recense les 500 plus puissants superordinateurs mondiaux selon un *benchmark* `LINPACK` (basé sur de l'algèbre linéaire telle qu'une décomposition LU de matrice). La Figure 3.2 présente les performances obtenues sur la première et dernière machine du classement ainsi que la puissance sommée de l'ensemble des 500 machines, à une date donnée. La constante régularité d'accroissement des performances y est visible. Celle-ci suit une évolution exponentielle depuis plus de 20 ans (croissance linéaire selon l'échelle logarithmique). Il est intéressant de noter que les machines ont une obsolescence rapide : il faut environ 8 ans avant que la machine la plus rapide ne devienne la plus lente du classement en termes de performances brutes.

Les composants disponibles pour les stations de calcul individuelles d'aujourd'hui présentent des performances équivalentes aux machines du `TOP500` d'il y a quelques années. Cela ouvre aux stations de travail la possibilité de réaliser des simulations relativement complexes dans des temps raisonnables grâce à leurs capacités de calcul.

Dans le cadre du contrôle non destructif, les ordres de grandeurs des simulations sont bien plus modestes comparés à des simulations en climatologie, météorologie, énergie ou armement. Les stations de travail aujourd'hui présentent un bon compromis puissance/coût et sont déjà couramment utilisées pour réaliser des simulations standards. Ces plateformes vont être ciblées par les présents travaux de thèse.

3.1.2 Unité centrale de traitement CPU

Le premier composant utilisé pour obtenir de la puissance de calcul est l'unité centrale de traitement (connu sous le nom de *Central Processing Unit* ou CPU, par ses sigles en anglais). C'est un processeur pouvant traiter toutes les opérations nécessaires au fonctionnement d'un ordinateur ou d'une station de travail. Processeur central de l'ordinateur, il est souvent utilisé comme argument de vente auprès du grand public pour exprimer la capacité de traitement d'un ordinateur personnel.

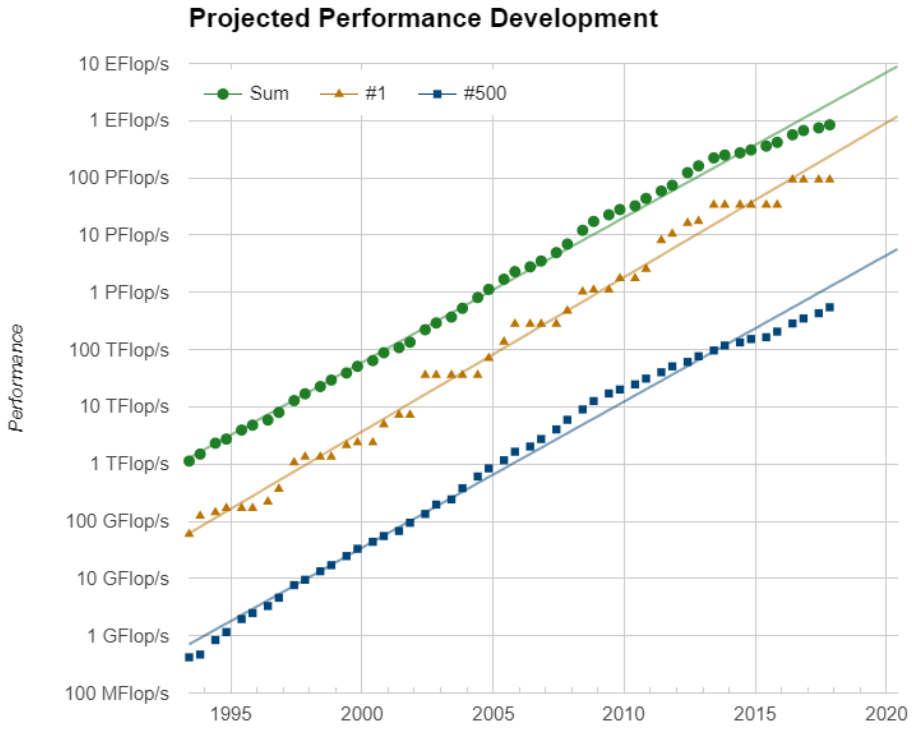


FIGURE 3.2 – Chronologie des performances des supercalculateurs du `TOP500`. (Source : `TOP500`.)

Les traitements qu'il doit réaliser lui sont transmis par les programmes sous la forme d'opérations d'arithmétique de base, d'opérations logiques, ou de gestion d'entrée/sorties. Les CPU possèdent des composants dédiés à chacun de ces types d'opérations : une unité de contrôle chargée de décoder le programme depuis la mémoire en instructions (CU, *control unit*), une unité arithmétique et logique pour les calculs entiers (ALU, *arithmetic logic unit*), une unité à virgule flottante dédiée au calcul sur nombres flottants (FPU, *floating point unit*) ainsi que d'autres composants dédiés à des tâches spécifiques (usage multimédia tel que l'encodage/décodage de flux, chiffrement...).

Les processeurs disposent d'un ensemble de fonctionnalités dédiées pour améliorer leurs performances :

— **Traitement pipeline** pour optimiser l'exécution de plusieurs instructions qui peuvent se superposer (toutes passent par différentes phases : rapatriement du code d'instruction, décodage du binaire, exécution, modification des registres, accès mémoire).

— **Exécution superscalaire** où l'on optimise l'exécution de plusieurs instructions non plus en séquençant leur traitement sur toute la chaîne, mais en utilisant plusieurs unités de calcul adaptées pour traiter ces instructions simultanément.

— **Exécution out-of-order** pour laquelle les instructions sont exécutées dans un ordre différent de celui du programme sans briser les dépendances de données.

— **Prédiction de branchement** qui permet au processeur de prédire l'exécution d'un branchement pour, encore une fois, optimiser son pipeline.

À ces outils propres à l'exécution du code se rajoutent des mémoires caches pour mettre dans un tampon les informations mémoires qui sont potentiellement utiles à la suite du programme. Pour multiplier les capacités des CPU, le parallélisme des données et le parallélisme de tâches sont aussi disponibles sous forme des instructions SIMD et processeurs multicœurs.

3.1.2.1 Mémoires caches

Au fur et à mesure de l'évolution des architectures (de fréquences de plus en plus rapides) des CPU a été introduite une hiérarchie des accès mémoire de plus en plus complexe. Comme l'indique la Figure 3.3, plusieurs niveaux de mémoire existent, les plus proches du processeur sont les plus rapides, mais ont également la moins grande capacité. Il est crucial de prendre en compte cette hiérarchie des accès mémoire pour optimiser un logiciel.

Dans ces mémoires caches, les données sont stockées à proximité des processeurs, c'est-à-dire avec une bande passante plus importante que celle de la mémoire externe. Avant l'existence des instructions SIMD (voir la sous-sous-section 3.2.1.2), la mémoire externe était capable d'alimenter le processeur pour maintenir une exécution stable. De nos jours, surtout après l'apparition du SIMD 512 bits et des instructions capables de consommer 16 données en simple précision en même temps, la bande passante fournie par les mémoires caches est indispensable. La mémoire cache est divisée et organisée en plusieurs niveaux :

— **L1**. Habituellement, il y a deux mémoires caches du type L1. Une est dédiée aux instructions du programme (L1i), l'autre est consacrée aux données (L1d).
— **L2**. Mémoire cache commun des instructions et données.

Sur certaines architectures, comme les processeurs multicœurs possédant plusieurs unités de calcul, d'autres niveaux de mémoire cache peuvent se trouver (voir la Figure 3.3) :

— **L3**. Mémoire cache similaire au L2, mais dans ce cas partagé par tous les cœurs du CPU.
— **L4**. Assez rare, il est du type mémoire vive dynamique au lieu de mémoire vive statique comme les autres mémoires caches. Il est disponible notamment pour les GPU portables. En CPU pour le calcul haute performance il est disponible pour POWER8 [Mewes et al., 2014].

Les mémoires caches les plus petites ayant les niveaux les plus proches des cœurs de calcul offrent une bande passante plus importante. La Figure 3.4 montre les débits mesurés (Go par seconde) des différents niveaux de la mémoire cache avec l'aide de l'outil STREAM [McCalpin, 1995]. Cet outil permet de mesurer la bande passante des problèmes intensifs en mémoire (lire, écrire, copier, mise en échelle, additionner, et les deux derniers combinés). Il a été modifié pour permettre d'obtenir la bande passante maximale pour différentes tailles de problèmes. Lorsque la taille augmente, les données tiennent dans des mémoires caches moins rapides, mais de plus grandes capacités, et la bande passante diminue.

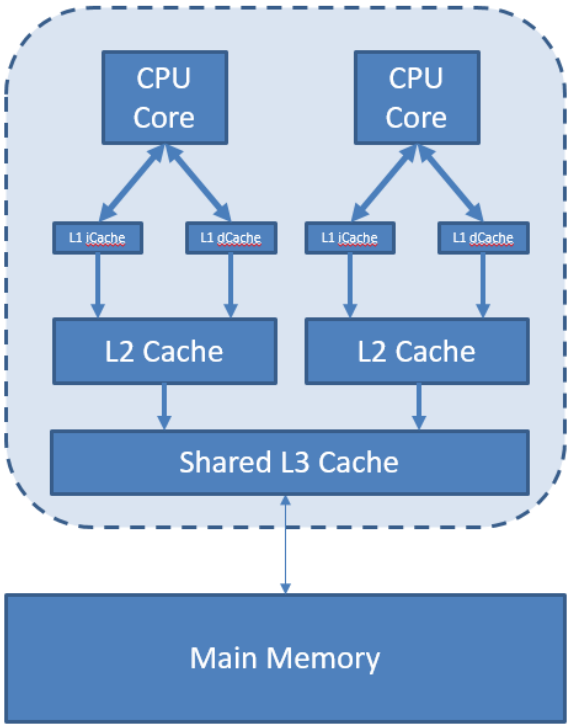


FIGURE 3.3 – Schéma des niveaux de mémoire pour un processeur . (Source : Oxford Protein Informatics Group.)

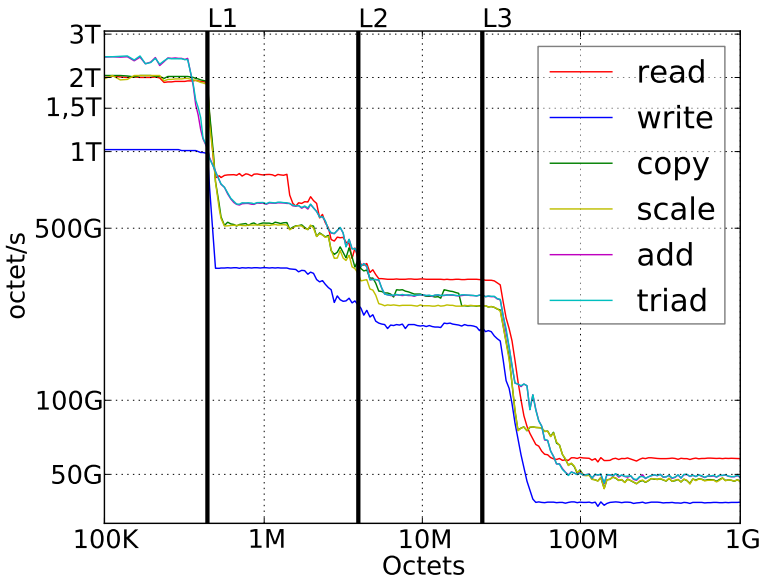


FIGURE 3.4 – Bande passante des différents niveaux de la mémoire cache calculée par l'outil STREAM modifié sur une machine E5-2683 v3 (Haswell, 14 cœurs, 2GHz, L1 32Ko/cœur, L2 256Ko/cœur, L3 20Mo).

3.1.2.2 Parallélisme des données

Une première approche afin de multiplier les capacités des CPU consiste à jouer sur le parallélisme de données : c'est le cas lorsqu'un traitement constitué d'une succession d'opérations séquentielles est appliqué à plusieurs données tout en étant contrôlé par un seul et unique flux de contrôle.

Des instructions SIMD ont été ajoutées aux CPU, afin d'effectuer la même opération sur plusieurs données, en général stockées dans des tableaux linéaires. Pour les applications scientifiques, des processeurs vectoriels ont été développés et exploités à partir de la fin des années 70, notamment sur les plateformes développées par Cray. On peut citer en particulier le Cray-1 en 1976 qui est le premier processeur industriel à disposer de registres et d'une unité de calcul vectoriels.

Aujourd'hui, la majorité des CPU modernes dispose de leurs propres jeux d'instructions en supplément des instructions scalaires existantes. Ces instructions s'apparentent aux instructions vectorielles, au sens où elles permettent d'exécuter sur plusieurs données la même instruction à la fois. Elles sont couramment appelées SIMD (*Single Instruction Multiple Data* selon la taxonomie de Flynn [1972]). Pour le grand public, ces instructions se sont principalement développées pour permettre des traitements multimédias toujours plus gourmands.

Et, par ailleurs, il y a un retour des cartes accélératrices vectorielles. Par exemple, NEC SX-Aurora TSUBASA [NEC-Corporation, 2017], dans le même format qu'une carte graphique.

3.1.2.3 Parallélisme de tâches

Une autre approche développée pour augmenter les capacités de traitement des CPU a été la mise en place des CPU composées de plusieurs cœurs ou unités de calculs, gravés au sein de la même puce. Cette approche joue sur le parallélisme de tâches : lorsque des tâches complexes, chacune distincte et disposant de son propre flux de contrôle, collaborent afin de traiter un problème donné.

Le premier processeur multicœur a été développé par IBM. Il s'agit du POWER4 (2001). En 2003, Sun à la suite d'IBM a commercialisé l'UltraSPARC IV comportant deux cœurs UltraSPARC III. En 2004, HP a produit le PA-8800 composé de deux cœurs PA-8700.

Il faut attendre 2005 pour voir apparaître les premiers exemplaires de CPU grand public Intel et AMD composés, pour chaque constructeur, de deux cœurs identiques.

Aux détails d'implémentation près, les cœurs regroupés sur une même puce partagent les mêmes entrées/sorties, en particulier vers la mémoire de l'ordinateur. Les mémoires caches sont parfois regroupées pour plusieurs cœurs, encourageant un accès cohérent à la mémoire pour les tâches réparties sur les cœurs d'une même puce. Cette proximité est parfois mise à profit pour faciliter les transferts de données entre les cœurs.

3.1.2.4 Architectures matérielles des CPU Intel

Cette section présente les principales caractéristiques architecturales des différents processeurs étudiés afin de mieux appréhender les spécificités de chacun d'entre eux. Intel a l'habitude de présenter une nouvelle architecture et, plus tard, une nouvelle méthode de fabrication lui permettant une réduction de la taille et une augmentation du nombre des cœurs. Quelques améliorations sont aussi présentées, mais l'unité d'exécution reste la même.

Architecture Intel Nehalem. Les processeurs Intel d'architecture Nehalem ont été commercialisés à partir de l'hiver 2008. Il s'agit d'une architecture multicœurs native, c'est à dire, dont les différents cœurs sont sur le même *die* de silicium, gravé en 45 nanomètres. Cette architecture gère la technologie *Hyper-Threading*, permettant à un cœur de travailler simultanément sur deux flux d'instructions différents, sous la forme de deux cœurs logiques, en masquant automatiquement les latences. Cette architecture gère un accès triple canal à la mémoire centrale de l'ordinateur à travers trois niveaux de mémoire cache successifs.

- Une mémoire cache L1 de 64Ko séparés en 32Ko dédiés aux données et 32Ko d'instructions.
- Une mémoire cache L2 de 256Ko par cœur.
- Une mémoire cache L3 de plusieurs Mo partagée par l'ensemble des cœurs d'une même puce. La taille d'une ligne de mémoire cache de L3 est de 64 octets.

Cette architecture gère des instructions x86-64bits, et des instructions SIMD de type SSE, SSE2, SSE3 (et SSSE3), SSE4.1 et SSE4.2. La taille des registres SIMD étant de 128 bits, il est possible d'effectuer ainsi des opérations simultanément sur 4 nombres flottants en simple précision ou sur 2 nombres flottants en double précision. L'unité d'exécution Nehalem est présentée par la Figure 3.5.

Architecture Westmere. En début d'année 2010, Intel a mis à disposition les premiers processeurs d'architecture Westmere, amélioration de Nehalem, cette fois-ci gravé en 32 nanomètres. Le nombre de cœurs maximum par CPU passe alors de 4 à 6. Cette nouvelle architecture améliore également les performances énergétiques des processeurs ainsi que leurs capacités de virtualisation matérielle et de chiffrement (instructions AES).

Architecture Sandy Bridge. En janvier 2011, Intel a présenté l'architecture Sandy Bridge, gravée en 32 nanomètres comme les processeurs Westmere. Intel a procédé à des améliorations sur la gestion interne des micro-opérations et des prédictions de branchement. La gamme Sandy Bridge gère désormais les instructions AVX (de longueur 256 bits). Les instructions spécifiques au chiffrement (AES/SHA-1) ont été améliorées. Avec des registres SIMD de 256 bits, il est désormais possible de manipuler jusqu'à 8 nombres flottants en simple précision ou 4 nombres flottants en double précision simultanément. L'unité d'exécution Sandy Bridge est détaillée par la Figure 3.6. Les CPU Sandy Bridge peuvent disposer de jusqu'à 8 cœurs physiques chacun.

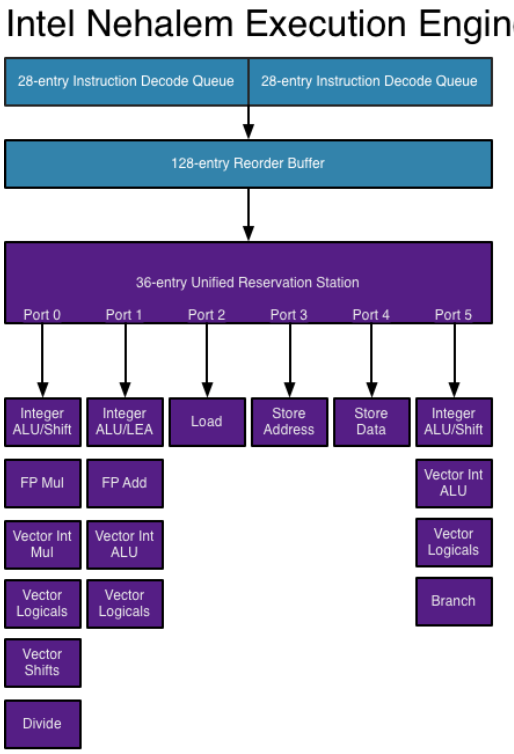


FIGURE 3.5 – Unités d'exécution Nehalem. (Source : Anandtech.)

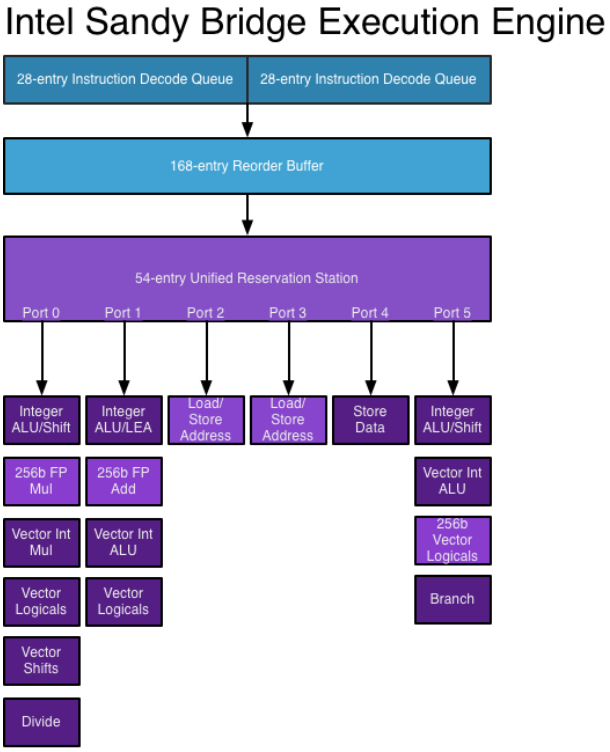


FIGURE 3.6 – Unités d'exécution Sandy Bridge. (Source : Anandtech.)

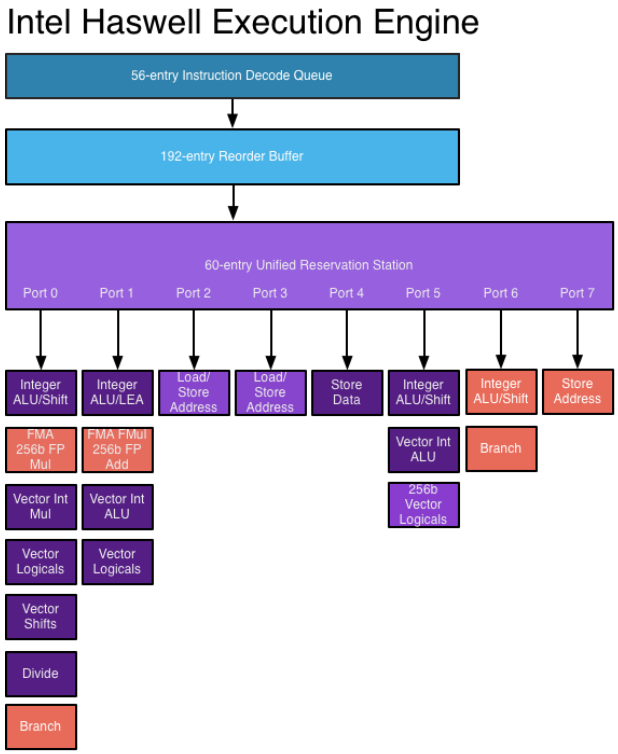


FIGURE 3.7 – Unités d'exécution Haswell. (Source : Anandtech.)

Architecture Ivy Bridge. La gamme suivante se nomme Ivy Bridge et a été commercialisée au printemps 2012. Cette fois-ci, la finesse de gravure est de 22 nanomètres. Le nombre maximum de cœurs est augmenté, en fonction du segment visé par chaque gamme Ivy Bridge (gamme Ivy Bridge -E, mono socket, 6 cœurs ; -EN, mono et bi socket, 10 cœurs ; -EP, quadri socket, 12 cœurs et -EX, octo socket, 15 cœurs). En matière architecturale, les améliorations touchent principalement aux fonctionnalités grand public, avec une meilleure gestion du GPU intégrée et de meilleures fonctionnalités multimédias.

Architecture Haswell. L'architecture suivante d'Intel, Haswell, a été mise à disposition en juin 2013, elle permet l'utilisation des instructions AVX2 offrant des fonctionnalités SIMD sur les nombres entiers, la récupération des données en mémoire non contiguë et la diffusion ou permutation des données entre plusieurs mots de 128 bits. Gravée en 22 nanomètres, l'unité d'exécution Haswell est présentée par la Figure 3.7. On y observe la présence de deux unités réalisant les FMA (Fused Multiply-Add) SIMD sur les ports 0 et 1 pour des vecteurs de largeur 256 bits. Les CPU Haswell peuvent disposer de jusqu'à 28 cœurs physiques.

Architecture Broadwell. L'amélioration de Haswell, nommé Broadwell, est disponible depuis septembre 2014 et tombe à 14 nanomètres. Le nombre des cœurs physiques disponibles augmente à 22. Broadwell introduit les instructions Intel ADX pour les opérations des entiers et Intel Quick Sync Video pour améliorer le décodage de vidéo du GPU intégré.

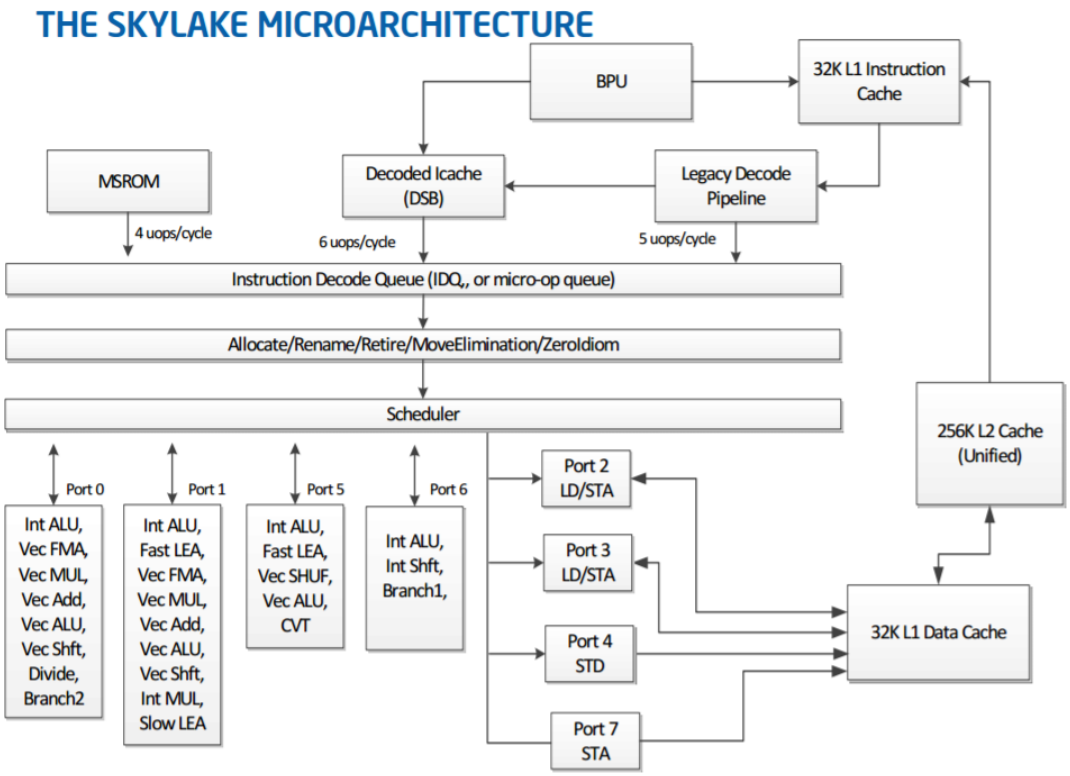


FIGURE 3.8 – Microarchitecture Skylake. (Source : Anandtech.)

Architecture Skylake. La dernière architecture étudiée est sortie commercialement en août 2015. Gravée elle aussi à 14 nanomètres, Skylake supporte jusqu'à 18 cœurs. Elle améliore les unités d'exécution et ajoute la capacité d'utiliser DDR4 SDRAM. La microarchitecture est présentée par la Figure 3.8. Skylake gère Intel MPX (*Memory Protection Extensions*) et Intel SGX (*Software Guard Extensions*), liés à la sécurité de la gestion de mémoire. En plus, il gère aussi les instructions SIMD AVX512 de longueur 512 bits, permettant d'effectuer des opérations simultanément sur 16 nombres flottants en simple précision ou 8 nombres flottants en double précision. Pour la première fois, tous les processeurs de la gamme Skylake ne gèrent pas les mêmes types d'extension. Les processeurs Skylake qui nous ont intéressés dans le cadre de cette thèse ont cinq types d'extension disponibles parmi les dix-huit extensions existantes :

— **F**, *Foundation*. Développe les instructions SSE-SSE4.2, AVX et AVX3 à 512 bits.
— **CD**, *Conflict Detection Instructions*. Améliore la détection des conflits afin de mieux vectoriser les boucles.
— **VL**, *Vector Length Extensions*. Permets à la plupart des instructions d'utiliser les registres en 128 et 256 bits.
— **BW**, *Byte and Word Instructions*. Ajoute les opérations pour 8 et 16 bits entiers.
— **DQ**, *Doubleword and Quadword Instructions*. Améliore les opérations d'entiers de 32 et 64 bits.

À partir de Skylake, Intel a rencontré des difficultés pour continuer avec sa politique de réduction de la taille de gravure (prévue pour être en 10 nanomètres) donc il a présenté nouvelles architectures (Kaby Lake, Coffee Lake) qui ajoutent des extensions liées au *deep learning*, aux calculs de Galois, au chiffrement ou à la manipulation des bits, mais encore en 14 nanomètres. En mai 2018 Intel a commercialisé des processeurs

Architecture	nm	# cœurs	Fréquence GHz	Extension SIMD	Largueur SIMD bits	# FMA	π F32	Π F32
Nehalem	45	4	3.3	SSE4.2	128	0	4	16
Westmere	32	6	3.5	SSE4.2	128	0	4	24
Sandy Bridge	32	8	3.1	AVX	128	0	4	32
Ivy Bridge	22	15	2.8	AVX	256	0	8	120
Haswell	22	18	2.5	AVX2	256	2	32	576
Broadwell	14	22	2.2	AVX2	256	2	32	704
Skylake	14	28	2.5	AVX512	512	2	64	1,792

TABLEAU 3.1 – Récapitulatif des architectures CPU Intel. Nombre des cœurs maximal et fréquence basse. FMA par cœur. π est le parallélisme intrinsèque par cœur (opération / cycle / cœur) et Π est le parallélisme intrinsèque total (opération / cycle).

avec l'architecture Cannon Lake en 10 nanomètres, mais de manière très limitée et sans version Xeon.

3.1.2.5 Synthèse d'architectures matérielles des CPU Intel

Le Tableau 3.1 présente un récapitulatif des architectures précédentes. Au fur et à mesure que la quantité des cœurs dans les architectures augmente, la fréquence de base (la fréquence minimale assurée par le fabricant) diminue. Et cette particularité se retrouve lorsque l'on utilise conjointement les différents cœurs d'un processeur. Mettons comme exemple le processeur Intel Xeon Platinum 8180, architecture Skylake, de 28 cœurs (correspondant à la dernière ligne du Tableau 3.1). Si on fait monter la quantité des cœurs utilisés, la fréquence turbo (la plus haute possible) tombe jusqu'à la fréquence basse (voir la Figure 3.9). Mais la fréquence tombe aussi lorsque le processeur exécute des jeux d'instruction plus élevés : la fréquence basse est 2.5GHz pour SSE (ou non AVX), 2.1GHz pour AVX2 et seulement 1.7 pour AVX512. Pour comparer correctement la performance d'un cœur contre plusieurs cœurs ou de l'utilisation des vecteurs des tailles différentes (128, 256 ou 512 bits), il faut soit fixer la fréquence, soit inclure sa variation dans les calculs.

Le fabricant va jouer avec cette variation de fréquence pour offrir des processeurs de différent prix et consommation électrique. Par exemple, comparons la fréquence des processeurs Intel Xeon Platinum 8180 et Intel Xeon Platinum 8176 (voir la Figure 3.10). La version supérieure consomme en moyenne 205W au lieu de 165W, et coûte 10,000$ au lieu de 8,720$. La fréquence du processeur supérieure tombe plus lentement que l'autre processeur lorsque les cœurs actifs augmentent.

Finalement, la Figure 3.11 montre la fréquence de toute la gamme Intel Xeon architecture Skylake (Platinum, Gold, Silver, Bronze) par rapport à la consommation moyenne.

3.1.3 Intel Xeon Phi MIC

Xeon Phi est une série de processeurs *many-integrated-core* (MIC) conçus par Intel. Les utilisateurs ciblés sont les mêmes que les utilisateurs des processeurs graphiques.

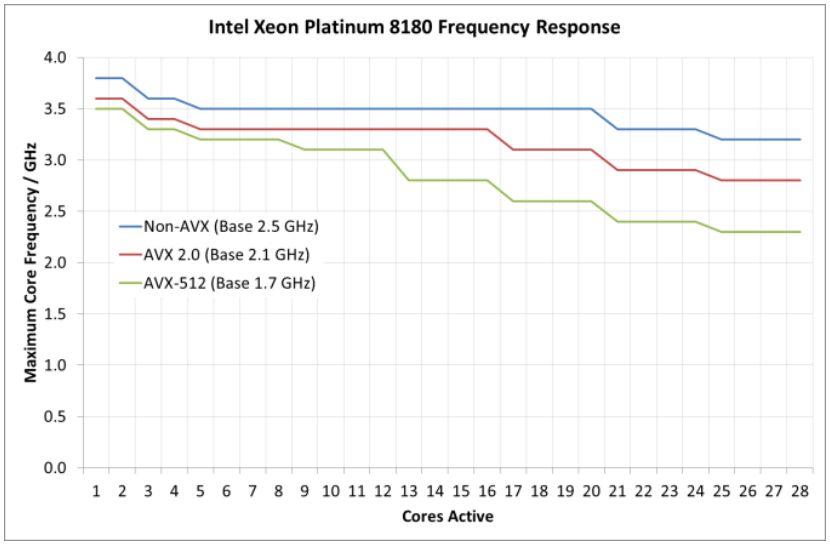


FIGURE 3.9 – Fréquence du processeur Intel Xeon Platinum 8180 selon le nombre de cœurs actifs et le jeu d'instructions utilisé. (Source : Anandtech.)

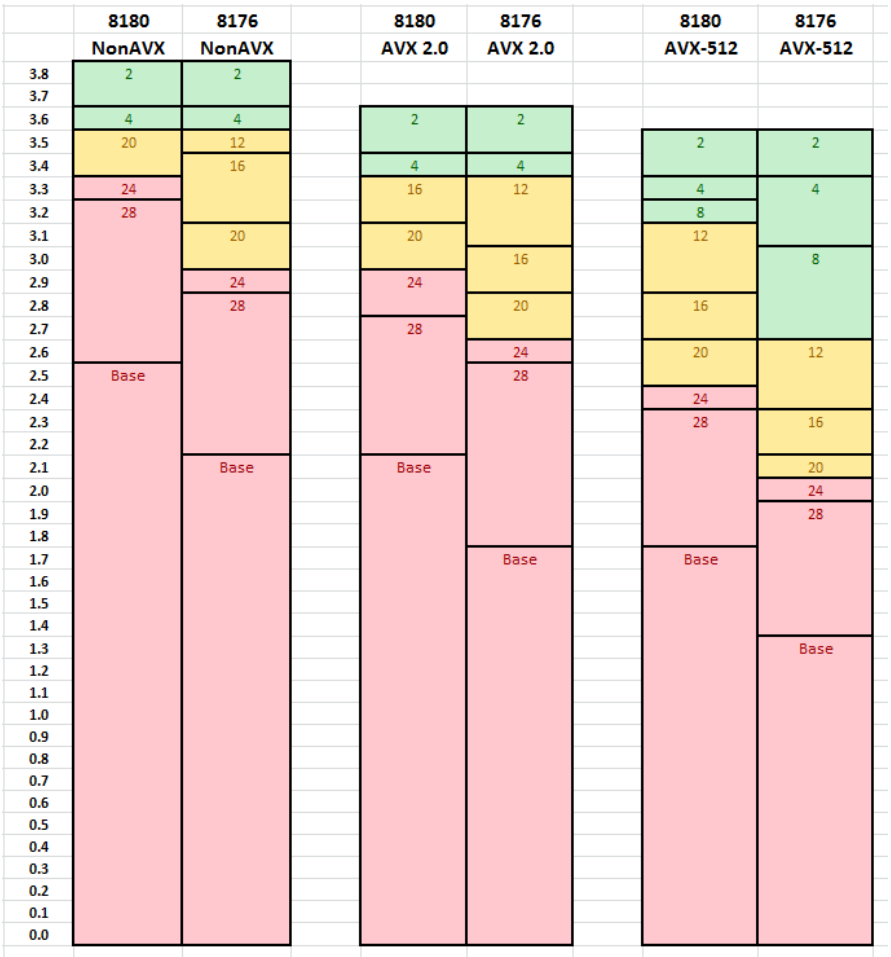


FIGURE 3.10 – Fréquence des processeurs Intel Xeon Platinum 8180 et Intel Xeon Platinum 8176 selon le nombre de cœurs actifs et le jeu d'instructions utilisé. (Source : Anandtech.)

Intel Skylake-SP AVX-512 Turbo Frequencies

AnandTech	Cores	LLC	TDP	Base	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Platinum 8180	28	38.50	205	1.7	3.5	3.5	3.3	3.3	3.2	3.2	3.2	3.2	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8	2.6	2.6	2.6	2.6	2.4	2.4	2.4	2.4	2.3	2.3	2.3	2.3
Platinum 8176	28	38.50	165	1.3	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0	1.9	1.9	1.9	1.9
Platinum 8170	26	35.75	165	1.3	3.5	3.5	3.3	3.3	2.9	2.9	2.9	2.9	2.5	2.5	2.5	2.5	2.2	2.2	2.2	2.2	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9	1.9	1.9		
Platinum 8168	24	33.00	205	1.9	3.5	3.5	3.3	3.3	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.5	2.5	2.5	2.5				
Platinum 8164	26	35.75	150	1.2	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9	1.8	1.8	1.8	1.8	1.8	1.8		
Platinum 8160	24	33.00	150	1.4	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0				
Platinum 8153	16	22.00	125	1.2	2.6	2.6	2.4	2.4	2.0	2.0	2.0	2.0	1.7	1.7	1.7	1.7	1.6	1.6	1.6	1.6												
Gold 6154	18	24.75	205	2.1	3.5	3.5	3.3	3.3	3.2	3.2	3.2	3.2	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8	2.7	2.7										
Gold 6152	22	30.25	140	1.4	3.5	3.5	3.3	3.3	2.9	2.9	2.9	2.9	2.5	2.5	2.5	2.5	2.2	2.2	2.2	2.2	2.0	2.0	2.0	2.0	2.0	2.0						
Gold 6150	18	24.75	165	1.9	3.5	3.5	3.3	3.3	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.5	2.5										
Gold 6148	20	27.50	150	1.6	3.5	3.5	3.3	3.3	3.1	3.1	3.1	3.1	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.2	2.2	2.2	2.2								
Gold 6142	16	22.00	150	1.6	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.2	2.2	2.2	2.2												
Gold 6140	18	24.75	140	1.5	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	2.1	2.1										
Gold 6138	20	27.50	125	1.3	3.5	3.5	3.3	3.3	2.7	2.7	2.7	2.7	2.3	2.3	2.3	2.3	2.0	2.0	2.0	2.0	1.9	1.9	1.9	1.9								
Gold 6136	12	24.75	150	2.1	3.5	3.5	3.3	3.3	3.1	3.1	3.1	3.1	2.7	2.7	2.7	2.7																
Gold 6134	8	24.75	130	2.1	3.5	3.5	3.3	3.3	2.7	2.7	2.7	2.7																				
Gold 6132	14	19.25	140	1.7	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.3	2.3														
Gold 6130	16	22.00	125	1.3	3.5	3.5	3.1	3.1	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9												
Gold 6128	6	19.25	115	2.3	3.5	3.5	3.3	3.3	2.9	2.9																						
Gold 6126	12	19.25	125	1.7	3.5	3.5	3.3	3.3	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3																
Gold 5122	4	16.50	105	2.7	3.5	3.5	3.3	3.3																								
Gold 5120	14	19.25	105	1.2	2.9	2.9	2.5	2.5	1.9	1.9	1.9	1.9	1.6	1.6	1.6	1.6	1.6	1.6														
Gold 5118	12	16.50	105	1.2	2.9	2.9	2.4	2.4	1.8	1.8	1.8	1.8	1.6	1.6	1.6	1.6																
Gold 5115	10	13.75	85	1.2	2.9	2.9	2.2	2.2	1.7	1.7	1.7	1.7	1.6	1.6																		
Silver 4116	12	16.50	85	1.1	1.8	-	-	-	-	-	-	-	-	-	-	1.4																
Silver 4114	10	13.75	85	1.1	1.8	1.8	1.6	1.6	1.5	1.5	1.5	1.5	1.4	1.4																		
Silver 4112	4	5.50	85	1.1	1.8	1.8	1.4	1.4																								
Silver 4110	8	11.00	85	1.0	1.8	1.8	1.6	1.6	1.3	1.3	1.3	1.3																				
Silver 4108	8	11.00	85	0.9	1.8	1.8	1.5	1.5	1.2	1.2	1.2	1.2																				
Bronze 3106	8	11.00	85	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8																				
Bronze 3104	6	8.25	85	0.8	0.8	0.8	0.8	0.8	0.8	1																						

FIGURE 3.11 – Fréquence de la gamme Intel Xeon architecture Skylake selon le nombre de cœurs actifs, la taille de la mémoire cache de plus bas niveau (*Last Level Cache* ou LLC), la consommation moyenne électrique (TDP) et la fréquence de base. (Source : Anandtech.)

En revanche, les logiciels originalement conçus pour des CPU nécessitent des modifications minimales pour s'en servir. Dans notre cas, à la suite d'une première série de tests négatifs sur un modèle Knights Corner (KNC, première génération), puis à l'annonce de discontinuité des modèles Knights Landing (KNL, deuxième génération) en 2015 et l'annulation de la troisième génération Knights Hill (KNH), les MIC ont été écartés de notre étude en faveur des cartes graphiques.

3.1.4 Processeurs graphiques GPU

Très vite, la tâche de représenter une information sur un écran a été déportée sur un composant matériel spécifique. Dans les années 1950 sont apparus les afficheurs vectoriels, pour lesquels le faisceau parcourait des vecteurs délimités par leurs extrémités de manière similaire à un écran d'oscilloscope par exemple. Dans les années 1970 sont apparus les *frame-buffers* (tampons de trame), mémoires suffisamment puissantes pour stocker la représentation de l'écran comme une succession de pixels codés en binaire. La tâche de remplir ces *frame-buffers* a été rapidement déportée vers des circuits spécialisés pour répondre, entre autres, aux besoins des applications vidéo ludiques.

Ces processeurs graphiques, ou GPU (pour *Graphical Processing Unit*) ont évolué pour acquérir de plus en plus de fonctionnalités. Ils ont au début, dans les années 80, disposé de capacités de traitement de formes géométriques planes simples jusqu'à être capables, au milieu des années 1990, de faire de la rendue 3D. Enfin, pour offrir plus de possibilités aux moteurs graphiques, les GPU sont devenus programmables, cela a permis de contrôler de plus en plus finement les traitements apportés sur l'ensemble des pixels. De plus, les GPU ont développé des capacités de type *stream processors*

(processeurs de flux), appliquant sur des flux de données des chaînes de traitements, exploitant des capacités de traitement parallèle limitées.

Dès 2001, Larsen et McAllister [2001] ont été parmi les premiers pionniers à utiliser les capacités intrinsèques des GPU pour les calculs de produits matriciels afin de les appliquer à des flux de données quelconques. Les capacités de programmation des GPU se sont rapidement mises au niveau des CPU fournissant entre autres des opérations mathématiques sur nombres flottants, des boucles...

À partir de 2006, les fabricants de GPU ont exploré le domaine du calcul mathématique sur GPU (*General-Purpose Computing on GPU* ou GPGPU) en fournissant des interfaces de programmation (abrégées API, de l'anglais *Application Programming Interface*) spécifiques pour réaliser des traitements génériques et se passer des astuces utilisant les API de rendues graphique souvent limité et peu adapté.

En 2006, NVIDIA a introduit la gamme GeForce 8 qui reposait entièrement sur un *stream processor* générique, appelé *Compute Unified Device Architecture* (CUDA). Cette architecture s'accompagnait d'un langage de programmation proche du C permettant un accès direct aux composants du GPU.

Son concurrent ATI a répondu la même année en proposant l'architecture *Close To Metal* (CTM), qui allait s'avérer plus complexe à utiliser. À la suite du rachat d'ATI par AMD, fabricant de CPU x86, CTM a été abandonné en 2011 et la programmation des GPU de la marque s'est reportée sur le langage OpenCL, unifiant la programmation GPU et CPU, langage présenté plus avant. En raison de la moindre maturité apparente des capacités de programmation en OpenCL et d'un environnement de développement plus complet côté NVIDIA, il a été décidé de travailler avec les GPU de ce fabricant.

3.1.4.1 GPU NVIDIA

CUDA ou *Compute Unified Device Architecture* est à la fois le nom de la technologie de calcul sur les GPU de la marque NVIDIA et celui du langage de programmation associé. Les différentes versions matérielles sont référencées sous le nom de *Compute Capability* (CC). La CC conditionne les capacités du GPU et son efficience dans certains types d'opérations, avec une rétrocompatibilité par rapport aux versions antérieures.

Les différentes unités de traitement, également appelées cœurs CUDA, sont regroupées en multiprocesseurs. Chacun de ces multiprocesseurs ne possède qu'une seule unité de contrôle en vue de réaliser des traitements parallèles. Cela diffère de la structure d'un CPU, comme montre la Figure 3.12. La composition d'un multiprocesseur ainsi que la taille de sa mémoire partagée dépendent de la CC du GPU. Le nombre de multiprocesseurs est variable au sein d'une gamme.

Notons les trois types de mémoire disponibles sur le GPU :

— **La mémoire globale** (*global*) : lente d'accès (bien que plus rapide que celle de CPU), mais de grande capacité (aux alentours d'un à deux Go sur les GPU de loisir haut de gamme, plusieurs Go pour la gamme professionnelle). Sa durée de

FIGURE 3.12 – Différence schématique entre un CPU et un GPU CUDA. (Source : Cuda Toolkit Documentation, NVIDIA.)

vie est celle de l'application dans son ensemble.

  - **Accès simple** : il y a une mémoire cache L2 pour y accéder. Accès lents hors mémoire cache. Lecture et écriture depuis la CPU et le GPU.
  - **Mémoire constant** : accès optimisé pour la lecture des constants. Lecture depuis le GPU, écriture depuis la CPU. Taille limitée par multiprocesseur.
  - **Textures** : accès optimisés pour les calculs d'interpolation en 2D. Lecture et écriture depuis le GPU. Taille limitée par multiprocesseur.
- **La mémoire partagée** (*shared memory*) : local à un multiprocesseur, cette mémoire est accessible à tous ses cœurs CUDA. La quantité de mémoire partagée utilisée par les *threads* sur chaque multiprocesseur est définie à l'exécution d'un noyau (code de calcul CUDA unitaire). Cette mémoire joue aussi le rôle d'une mémoire cache L1 pour accéder au mémoire cache L2.
- **Les registres** (*registers*) : qui sont propres à chaque *thread*. Leur usage est déterminé par le compilateur lors de la compilation d'un noyau et il est également possible d'imposer une limite maximum lors de la compilation (en maximum de registre par *thread*).

Les multiprocesseurs utilisent le modèle de programmation *Single Instruction Multiple Thread* (SIMT) : à chaque cycle, tous les cœurs d'un multiprocesseur exécutent simultanément la même instruction, faisant avancer un ensemble de *threads* d'autant. Les multiprocesseurs ordonnancent ces *threads* par groupe de 32 *threads* appelé *warp*. Tous les *threads* de ce *warp* sont traités simultanément par le multiprocesseur. Les points d'ordonnancement se trouvent à la fin de chaque *warp*.

L'efficience maximale de ce modèle est atteinte lorsque chaque *thread* du *warp* exécute la même instruction. Lorsqu'il y a une divergence due aux données, les deux branches divergentes sont alors exécutées consécutivement par le *warp*. Les performances sont dégradées par les exécutions successives des différentes branches pour les *threads* actifs.

Les contextes de chaque *warp* exécutés sur un multiprocesseur sont conservés sur la puce pour toute la durée d'exécution du *warp*. L'ordonnanceur peut donc changer de *warp* sans surcoût, ce qui permet, à chaque cycle, de sélectionner un nouveau *warp* prêt à exécuter sa prochaine instruction si le *warp* en cours subit une synchronisation ou la latence d'une instruction. Ceci permet d'optimiser l'exploitation des cœurs CUDA.

3.1.4.2 Architectures matérielles des GPU NVIDIA

Cette section présente les principales caractéristiques architecturales des différents processeurs graphiques étudiés afin de mieux appréhender les spécificités de chacun d'entre eux. Au cours des années NVIDIA a présenté plusieurs architectures avec des différences notables entre elles, surtout les dernières années. Dans le monde des fabricants des cartes graphiques s'impose l'idée de mettre à disposition du client des cartes graphiques spécifiques selon leurs besoins : HPC, *gaming*, *deep learning*, *big data*, *crypto coin mining*, etc.

Architecture Fermi. Les GPU NVIDIA reposants sur la microarchitecture Fermi sont gravés en 40nm pour les processeurs de bureau et comprennent environ 3.0 milliards de transistors [NVIDIA-Corporation, 2009]. La mémoire associée au coprocesseur graphique est de type GDDR5. Ces cartes sont connectées à l'hôte par une interface PCI Express 2.0 x16 permettant d'atteindre un débit de transfert maximal de 8 Go/s.

Les *Streaming Multiprocessors* (SM) associés sont composés de :

- **32 cœurs CUDA** capables de réaliser des calculs flottants en simple précision, répartis en 2 tableaux de 16 cœurs CUDA en collaboration.
- **16 unités gérant la mémoire** capables de transformer à la volée les adresses pour faciliter des accès bidimensionnels.
- **4 unités de fonctions spéciales** (ou SFU pour *Special Functions Units*) pour réaliser les instructions transcendantales telles que les fonctions trigonométriques et les racines carrées.
- **64Ko de mémoire ultra rapide** répartis à l'exécution entre la mémoire cache L1 et la mémoire partagée.
- **1 interface vers la mémoire cache L2**.
- **32K registres 32 bits** afin de permettre à chaque *thread* d'avoir ses propres registres, indépendants des autres *threads*, jusqu'à un maximum de 63 registres par *thread* pour un noyau de calcul.

Un cœur de calcul CUDA réalise des calculs flottants (selon le standard IEEE 754-2008) et entiers en 32 bits à l'aide d'ALU et FPU adaptées. Chaque cœur peut réaliser une instruction FMA en un cycle d'horloge. Ces cœurs sont capables de réaliser des calculs en double précision en deux cycles d'horloge.

Les SM, représentés par la Figure 3.13, arrangent les *threads* en *warps* de 32. Chaque SM dispose de deux ordonnanceurs et de deux unités de distribution d'instructions afin d'exécuter de manière concurrente deux instructions à la fois. Les ordonnanceurs sélectionnent deux *warps* et distribuent une instruction de chaque *warp* à un groupe de 16 cœurs, 16 unités de gestion mémoire, ou 4 FSU. La plupart des instructions peuvent être distribuées en double : deux instructions entières, deux instructions flottantes, ou des mélanges d'instructions.

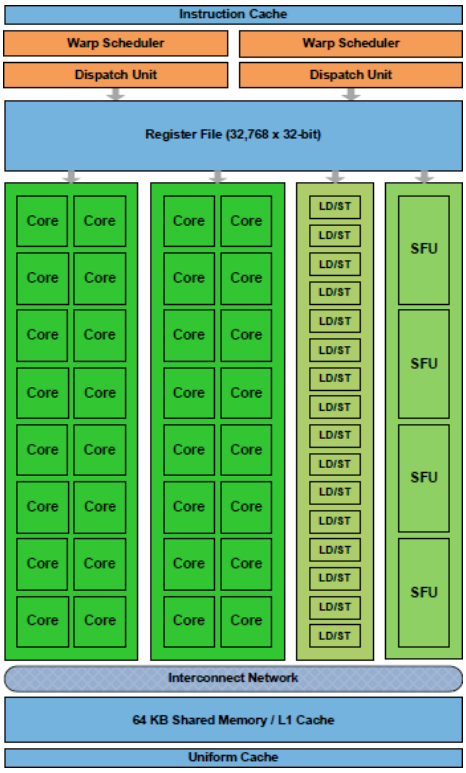


FIGURE 3.13 – Fermi Streaming Multiprocessor. (Source : Fermi Whitepaper, NVIDIA.)

Architecture Kepler. En 2012, trois ans après Fermi, NVIDIA a mis sur le marché une nouvelle microarchitecture GPU, du nom de Kepler [NVIDIA-Corporation, 2012]. Cette architecture, désormais gravée en 28nm, comporte plus de 7.1 milliards de transistors, communique avec l'hôte en PCI Express 3.0 ×16 (un débit de 16 Go/s).

La nouvelle génération de *Streaming Multiprocessors* est dénommée SMX, représenté par la Figure 3.14. Ils consomment beaucoup moins d'énergie que la génération précédente : deux SMX Kepler consomment environ l'équivalent de 90% de la consommation d'un SM Fermi. Ils disposent de 192 cœurs CUDA chacun, répartis par tableaux de 32 éléments. Le nombre d'unités responsables des accès mémoires est doublé à 64 éléments. Le nombre de FSU passe à 16. Les capacités de calcul en nombres flottantes en double précision sont améliorées par l'ajout de FPU dédiées, au nombre de 64. La quantité de registres est doublée, avec 64K registres disponibles et une possibilité d'utiliser jusqu'à 255 registres par *thread*.

Comme son prédécesseur, l'unité d'exécution reste le *warp* de 32 *threads*. Cependant, les ordonnanceurs des SMX ont connu des améliorations : 4 ordonnanceurs peuvent distribuer jusqu'à 8 instructions par cycle (2 par *warp*). De plus, les instructions en double précision peuvent s'exécuter conjointement à d'autres instructions.

Architecture Maxwell. Deux ans plus tard, NVIDIA a présenté l'architecture Maxwell [NVIDIA-Corporation, 2014]. Encore gravé en 28nm et comprenant autour 8 milliards des transistors, ses *Streaming Multiprocessors* (SMM) sont capables de doubler la performance sans augmenter la consomption d'énergie. Chaque SMM (représenté par la Figure 3.15) a quatre ordonnanceurs de *warp* séparés, avec un total de 128 cœurs

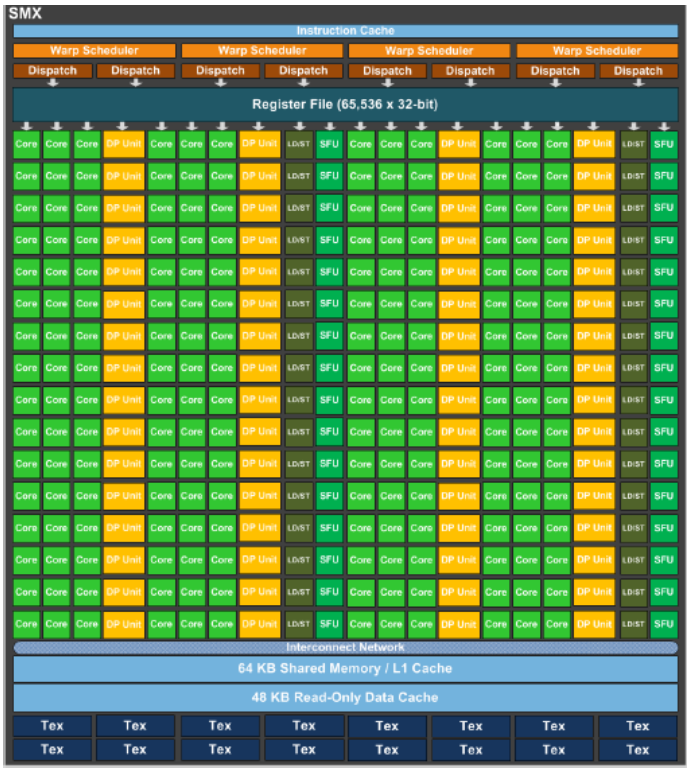


FIGURE 3.14 – Kepler Streaming Multiprocessor. (Source : Kepler-GK110 Whitepaper, NVIDIA)

CUDA, chacun avec ses propres ressources des instructions. Ces ordonnanceurs sont améliorés pour réduire les opérations redondantes. La hiérarchie de mémoire a été modifiée pour laisser une mémoire partagée de 96Ko pas combiné avec la mémoire cache L1. En ce cas, la mémoire cache L1 et la fonction de cache de textures ont été fusionnées. Grâce aux changements, chaque cœur CUDA est ×1.4 plus performant qu'un cœur Kepler, et ×2 plus performant par rapport à la consommation d'énergie. Comme un SMM de Maxwell occupe un espace bien inférieur à un SMX Kepler, mais délivre la même performance, la quantité des SMM est normalement doublée, arrivant jusqu'à 24 SMM.

Architecture Pascal. En 2016, NVIDIA a proposé l'architecture Pascal [NVIDIA-Corporation, 2016]. Plus destiné aux calculs haute performance que son prédécesseur, la gravure a été divisé par deux (14nm) et le nombre de transistors a grimpé jusqu'à 12 milliards. Visant à être utilisés par des supercalculateurs pour des calculs d'intelligence artificiel, *big data*, *deep learning*, etc., l'architecture Pascal a été conçue pour permettre la scalabilité. Le système NVLink 1.0 permet un débit de 160Go entre différents GPU, ce qui est ×5 fois plus rapide que les transferts avec l'hôte. L'idée est de permettre la construction des schémas où un seul CPU contrôle plusieurs GPU.

En plus, NVIDIA met en valeur spécialement les performances mesurées en opérations flottantes par seconde (flop/s), supposés d'être au moins ×3 par rapport à l'architecture Maxwell. Une augmentation des opérations est normalement inutile sans une augmentation de la bande passante. Pascal est la première architecture utilisant la mémoire HBM2, ×3 plus rapide que la mémoire de type GDDR5.

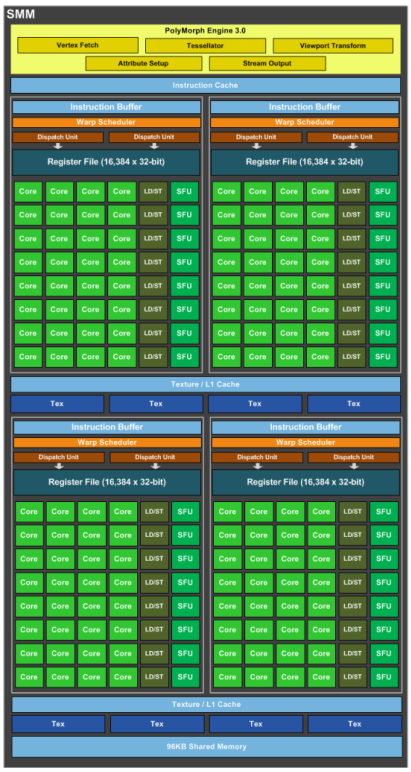


FIGURE 3.15 – Maxwell Streaming Multiprocessor. (Source : Maxwell-GTX980 Whitepaper, NVIDIA)

Le *Streaming Multiprocessors* (à nouveau SM) de Pascal, représenté par la Figure 3.16, diffère assez de celui de Maxwell. Avec deux ordonnanceurs de *warp* séparés, un SM a seulement 64 cœurs CUDA de simple précision, au lieu des 128 de Maxwell. Ces cœurs sont capables aussi d'opérer sur des nombres flottants de demi-précision (16 bits), à une cadence ×2 par rapport aux nombres flottants en simple précision. En plus de ces cœurs, il ajoute 32 cœurs CUDA de double précision. La taille des registres reste la même, c'est-à-dire qu'il y a plus des ressources pour chaque cœur CUDA. Enfin, la quantité totale des SM peut arriver à 50, permettant doubler la bande passante effective, d'augmenter les registres et les *warps* disponibles pour exécution.

Architecture Volta. La dernière architecture proposée par NVIDIA est nommée Volta [NVIDIA-Corporation, 2017], communique avec l'hôte en PCI Express 4.0 ×16 (un débit de 32 Go/s). Cette architecture est trop récente pour être bien traitée dans ce document, mais il faut dire qu'elle est encore plus dédiée au *deep learning* que Pascal. Le système NVLink 2.0 a été amélioré et la mémoire HBM2 aussi. L'efficience énergétique, comme habituellement entre les architectures de NVIDIA, est doublée. Le changement le plus important au niveau de *Streaming Multiprocessors*, voir la Figure 3.17, est l'apparition de 64 cœurs CUDA dédiés aux opérations en entier et 8 cœurs CUDA Tensor, chacun d'entre eux capable d'opérer 64 opérations FMA par cycle.

3.1.4.3 Synthèse d'architectures matérielles des GPU NVIDIA

Le Tableau 3.2 présent un récapitulatif des architectures précédentes.

FIGURE 3.16 – Pascal Streaming Multiprocessor. (Source : Pascal-GP100 Whitepaper, NVIDIA.)

FIGURE 3.17 – Volta Streaming Multiprocessor. (Source : Volta-V100 Whitepaper, NVIDIA.)

Architecture	nm	# transistors	# SM	# cœurs	Fréquence GHz	Π F32	PCIe Go/s	NVLink Go/s
Fermi	40	3.0	14	448	0.7	896	8	-
Kepler	28	7.1	14	2,688	0.8	5,376	16	-
Maxwell	28	8.0	24	3,072	1.0	6,144	16	-
Pascal	14	12.0	28	3,584	1.4	7,168	16	160
Volta	12	21.1	80	5,120	1.2	10,240	32	300

TABLEAU 3.2 – Récapitulatif des architectures GPU NVIDIA. Nombre des transistors en milliards. Fréquence basse. Un seul FMA par cœur. Π est le parallélisme intrinsèque total (op / cycle).

3.2 Les langages de programmation et les outils associés

Le nombre d'outils pour la programmation parallèle des architectures précédemment présentées (CPU et GPU) est très important. Cette section décrit certains d'entre eux parmi les plus utilisés et accessibles depuis un code C. Ils sont regroupés par famille, en allant de ceux, bas niveau, au plus proche du matériel, vers ceux qui exposent le plus grand niveau d'abstraction.

3.2.1 Les outils natifs

Une première approche consiste à utiliser directement les outils bas niveau disponibles pour programmer les composants sur lesquels les noyaux de calcul doivent s'exécuter. Cette approche permet une gestion très fine du matériel, au coût d'une spécialisation des programmes développés.

3.2.1.1 Multithreading

Cette approche consiste à utiliser les outils disponibles sur tout système d'exploitation pour programmer des ensembles de tâches et ordonnancer leur exécution. Originellement, chaque système d'exploitation dispose de sa propre implémentation de *threading* spécifique (dont l'exposition, à travers les différents langages de programmation varie). Plusieurs alternatives sont apparues pour unifier l'usage des *threads* et fournir des solutions normalisées et portables aux développeurs. Pour les langages *C* et *C++* on peut citer `pThread` sur les systèmes d'exploitation compatibles avec la norme `POSIX` (Linux, BSD, OS X ...) et les initiatives d'unification au moyen d'encapsulations de plus haut niveau (`Boost.Thread` par exemple). Les deux plus souvent utilisés sont les directives OpenMP et la bibliothèque Intel Threading Building Blocks (TBB).

OpenMP. Solution très utilisée pour obtenir des codes de calcul *multithread*, OpenMP prend la forme de directives de préprocesseurs (`pragma`) intégrées aux codes *C*, *C++* et Fortran. OpenMP est accompagnée d'une bibliothèque de fonctions de contrôle. OpenMP a été pour la première fois spécifiée pour le langage Fortran en octobre 1997, puis en octobre 1998 dans le langage *C* et *C++*, par l'*OpenMP Architecture Review*

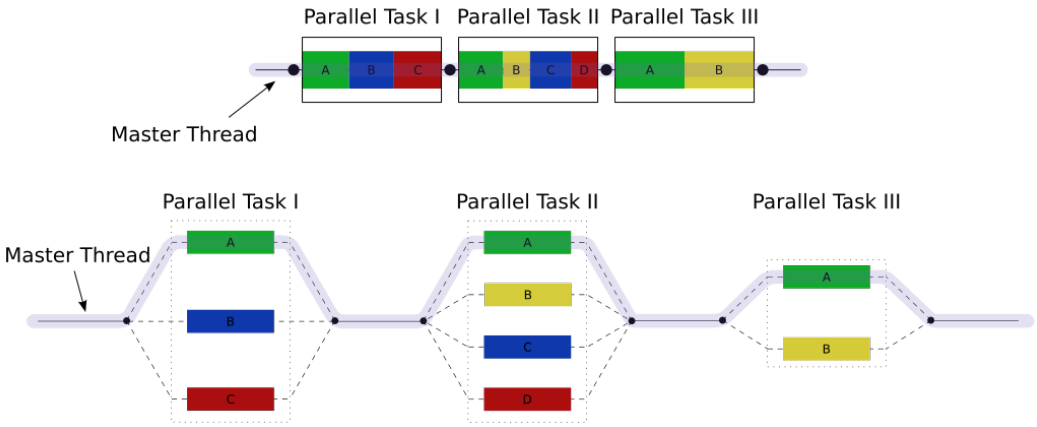


FIGURE 3.18 – Schéma de `fork/join` OpenMP. En haut, la version séquentielle avec 3 tâches parallèles. En bas, l'exécution parallèle. (Source : Wikipédia.)

Board, consortium regroupant les acteurs principaux de l'informatique matérielle et logicielle. On peut citer entre autres AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Red Hat, Texas Instruments, Oracle Corporation, ...

OpenMP fournit une solution portable de *multithreading* pour différents systèmes d'exploitation (Linux, MacOS, Windows) et différents CPU. Son support est géré directement au niveau du compilateur, qui prend en compte les directives afin de procéder aux transformations de code nécessaires pour permettre aux programmes de s'exécuter de manière *multithread*. Grâce à cette utilisation de directives de programmation, la programmation OpenMP est très peu intrusive au sein d'un code existant, mais le nombre de décorations peut diminuer la lisibilité. Lorsque le compilateur ne dispose pas du support OpenMP, les directives sont simplement ignorées. OpenMP permet facilement de réaliser un calcul régulier en répartissant les itérations d'une boucle sur différents *threads*.

De manière générale, le modèle OpenMP repose sur une succession de sections parallèles et de sections séquentielles. Les tâches de calcul intensif, décorées par les directives spécifiques, constituent les tâches parallèles. Chaque tâche est subdivisée sur des *threads* secondaires contrôlés par le *thread* principal, sur lesquels l'ensemble du domaine de calcul est réparti. La Figure 3.18 présente cette stratégie, avec un *thread* principal qui répartit plusieurs tâches différentes sur des *threads* secondaires. La bibliothèque OpenMP permet à chaque *thread* de se renseigner sur l'état dans lequel le programme se trouve et de connaître son indice dans l'ensemble des *threads* associés à une tâche.

Les primitives fournies permettent, notamment, de manipuler la manière dont les tâches sont découpées. L'utilisation canonique de cet outil consiste à découper le domaine d'itérations d'une boucle `for`. Il est possible à la fois de configurer le partage des données entre les *threads*, la gestion du placement du domaine parmi les *threads* pour optimiser les accès mémoire, la manière dont les *threads* seront ordonnancés, ou encore la gestion du nombre de *threads* dédiés à cette tâche. En particulier, il est possible d'indiquer à OpenMP si les variables sont privées (`private`, propres à chaque *thread*) ou partagées entre les threads (`shared`). OpenMP permet également de procéder à des réductions, en précisant sur quelle variable la réduction a lieu et quelle opération

appliquer.

Intel Threading Building Blocks (TBB). TBB est une bibliothèque de *templates* *C++* pour faciliter l'usage des capacités multicœurs sur des processeurs généralistes. Cette bibliothèque combine des structures de données et des algorithmes permettant au développeur de s'abstraire des complications de synchronisation liées à l'utilisation de *threads* classiques. Plutôt que de fournir un accès direct aux *threads* bas niveau, Intel TBB permet de créer des tâches associées à des opérations. Ces tâches sont déléguées aux différents cœurs du processeur dynamiquement par les mécanismes de la bibliothèque qui optimisent l'usage des caches du CPU.

Dans TBB, les tâches s'imbriquent les unes à la suite des autres dans des graphes de tâches que la bibliothèque TBB consomme au fil des algorithmes de calcul. La manière dont ces tâches vont s'enchaîner peut-être définie à l'aide d'un certain nombre de motifs classiques (pipeline, graph parallélisme...). Afin de faciliter le portage d'applications existantes, TBB s'accompagne de conteneurs similaires à ceux de la STL (*Standard Template Library*) du *C++*. Enfin, TBB fournit des fonctions de gestion mémoire optimisées pour travailler dans un environnement *multithread*.

Pour équilibrer la charge de travail entre les cœurs de calcul TBB implémente un système dit de *task stealing* : dès qu'un *thread* de calcul termine sa tâche, l'ordonnanceur lui assigne des tâches de calcul prévues préalablement pour d'autres *threads* qui eux n'ont pas encore terminé leur travail. TBB fournit aussi des primitives adaptées à la parallélisation de boucles de la même manière qu'OpenMP, afin de faire ressortir du parallélisme de données. Cependant, cette approche est plus intrusive que les directives de compilation utilisées par OpenMP.

3.2.1.2 Extensions SIMD

La gestion des cœurs d'un CPU n'est pas le seul point qu'il est possible de manipuler directement. Les instructions SIMD (ou *intrinsics* en anglais) sont utilisables à très bas niveau, directement en assembleur, mais cela s'avère rapidement fastidieux. Une alternative efficiente consiste à utiliser des fonctions *C* appelant directement les instructions sur leurs paramètres directement depuis un code natif en C ou *C++*. Cela nécessite de spécialiser le code pour une extension SIMD particulière, et de coder manuellement. L'utilisation de ces instructions nécessite d'avoir fait ressortir le parallélisme de données directement dans le code et de gérer manuellement les registres SIMD (et les problématiques d'alignement mémoire). Cette approche est appelée SIMDisation par opposition à la vectorisation lorsque c'est le compilateur.

Le codage avec des instructions SIMD en *C* ou en *C++* nécessite une connaissance approfondie de l'algorithme et de l'architecture, tant du côté des données que du côté des opérations. Cette technique permet un contrôle très fin, mais nécessite beaucoup de modifications du code, rendant sa maintenance plus difficile. Le Tableau 3.3 présent un récapitulatif de quelques architectures SIMD.

Les instructions SIMD existent sur l'ensemble des processeurs généralistes depuis

Architecture	Extension	Année	Largueur SIMD	I8	I16	I32	I64	F32	F64
PA-RISC	MAX-1	1994	32	✓	✓				
PA-RISC	MAX-2	1995	64	✓	✓	✓			
SPARC	VIS	1995	64	✓	✓	✓			
MIPS-V	MDMX	1996	64	✓	✓	✓			
MIPS-V	paired-single	1996	64					✓	
Alpha	MVI	1996	64	✓	✓	✓			
x86	MMX	1997	64	✓	✓	✓			
x86	3DNow !	1998	64					✓	
PowerPC	Altivec	1998	128	✓	✓	✓		✓	
x86	SSE	1999	128					✓	
x86	SSE2	2001	128	✓	✓	✓	~	✓	✓
ARM	media extension	2002	32	✓	✓				
x86	SSE3	2004	128	✓	✓	✓	~	✓	✓
x86	SSSE3	2006	128	✓	✓	✓	~	✓	✓
x86	SSE4.1	2007	128	✓	✓	✓	~	✓	✓
x86	SSE4.2	2008	128	✓	✓	✓	✓	✓	✓
ARM	Neon	2008	128	✓	✓	✓	~	✓	
PowerPC	VSX	2009	128	✓	✓	✓	✓	✓	✓
x86	AVX	2011	256	✓	✓	✓		✓	✓
Power	QPX	2012	256						✓
ARMv8	Neon	2013	256	✓	✓	✓	✓	✓	✓
x86	AVX2	2013	256	✓	✓	✓	✓	✓	✓
x86	AVX512	2017	512	✓	✓	✓	✓	✓	✓
ARM	SVE	2017	128-2048	✓	✓	✓	✓	✓	✓

TABLEAU 3.3 – Récapitulatif de quelques architectures SIMD. ~est un support partiel.

plusieurs décades, mais chaque fabricant et chaque génération de CPU disposent de sa ou ses normes et d'instructions spécifiques. Afin d'obtenir des accélérations conséquentes, l'usage de ces instructions nécessite d'une solution unique et peu portable au-delà d'une rétrocompatibilité ascendante au sein des processeurs d'un même fabricant. Ainsi, sur les CPU Intel gérant les instructions AVX (256 bits), il est possible d'utiliser des instructions SSE (128 bits). Le CPU utilise alors la FPU AVX sur des registres SSE.

Certains *wrappers* pour éviter l'écriture manuelle des instructions existent : Boost.SIMD [Estérie et al., 2014], libsimdpp [Kanapickas, 2018], MIPP [Cassagne et al., 2018], UME : :SIMD [Karpiński et McDonald, 2017] et vcl [Fog, 2017]. D'autres bibliothèques et outils pour SIMD sont également disponibles, tels que CilkPlus, Cyme, VecImp [Leisa et al., 2012], ispc [Brodman et al., 2014], Sierra [Leisa et al., 2014] et VC [Kretz et Lindenstruth, 2012]. Cyme et Sierra semblent ne plus être maintenues.

3.2.1.3 CUDA

Le langage de programmation CUDA est le langage associé à la programmation pour le calcul des GPU de marque NVIDIA utilisant l'architecture du même nom. CUDA permet de programmer directement le GPU (ou *device*) comme un coprocesseur du CPU hôte (ou *host*), en gérant manuellement toute la hiérarchie des mémoires disponibles au moyen d'une API conséquente. Il s'agit d'utiliser ces GPU pour réaliser des calculs génériques (GPGPU) par opposition à la programmation classique des GPU pour le rendu vidéo.

Les noyaux de calcul (*kernel* en anglais) et la manière dont les *threads* de calcul vont se répartir la tâche et ses sous-divisions sont gérés au lancement de *kernel* de manière dynamique. Ces *kernels* sont écrits en un sous-ensemble de *C* et *C++* agrémenté d'informations spécifiques aux GPU, en particulier des qualificatifs pour indiquer la localité des mémoires et des accesseurs aux informations de position dans l'espace des calculs.

L'API de CUDA fournit également un support pour le développement de codes massivement parallèles avec un ensemble de routines de contrôles telles que des opérations de calcul atomiques ou des fonctions de synchronisation entre des *threads* collaborants. Les fonctionnalités de l'API supportées par un matériel donné sont définies par la caractéristique *Compute Capability*, qu'il ne faut pas confondre avec la version de l'API qui définit la version logicielle de la bibliothèque CUDA associée.

Modèle d'exécution. Le *kernel* de calcul est exécuté par tous les *threads* légers demandés à l'exécution. Leur répartition se fait à deux niveaux : sur une grille de blocs (*grid* en anglais), chaque *bloc* contenant un ensemble de *threads*, représenté par la Figure 3.19.

Lors de l'exécution d'un *kernel*, les différents *threads* de chaque bloc sont alors divisés en *warp* de 32 *threads* contigus. Un bloc est traité dans son ensemble par un seul multiprocesseur, ses *warps* sont ordonnancés afin de maximiser les performances (minimiser les latences dues aux accès mémoire, aux opérations fondamentales, aux

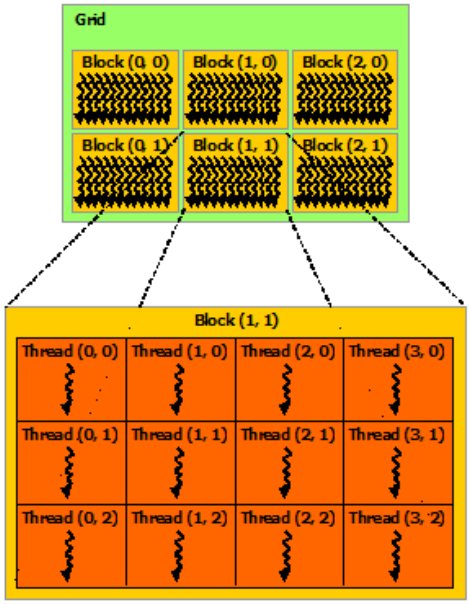


FIGURE 3.19 – Grille des blocs, composés par *threads*. (Source : Cuda Toolkit Documentation, NVIDIA.)

branchements...).

Modèle de mémoire. C'est à l'exécution d'un bloc que la répartition de la mémoire partagée du multiprocesseur est réalisée : un multiprocesseur peut traiter plusieurs blocs si ses ressources physiques lui permettent. La Figure 3.20 montre que la communication de l'hôte est faite directement sur la mémoire globale (soit simple, constante ou texture, voir la sous-sous-section 3.1.4.1 pour plus de détails). Cette mémoire est partagée par la grille de blocs. Tous les *threads* d'un bloc ont accès à une mémoire partagée (ou *shared memory)*. En plus chaque *thread* utilise sa propre cache L1 (ou *local memory*) et ses registres.

Pour permettre la communication entre les différents *threads* et éviter les problèmes d'accès concurrents de mémoire, CUDA fournit trois niveaux de synchronisation :

— Une synchronisation de tous les *threads* d'un bloc en un point de rendez-vous via des barrières.

— Une synchronisation *a posteriori* de l'exécution d'un *kernel* : CUDA garantit que tous les blocs se sont exécutés ; cela permet de synchroniser entre eux plusieurs *kernels* successifs exécutés à la chaîne sur le même GPU.

— Une synchronisation entre le GPU et le CPU (le plus souvent pour récupérer le résultat de traitements) une fois un ensemble de *kernels* exécuté.

3.2.2 Les outils hybrides

Afin d'essayer d'unifier la programmation des architectures hétérogènes, des outils ont été créés. Pour atteindre ce but, ces outils ont fait des choix différents en termes d'API et s'intègrent à des niveaux différents dans la chaîne de développement. Trois

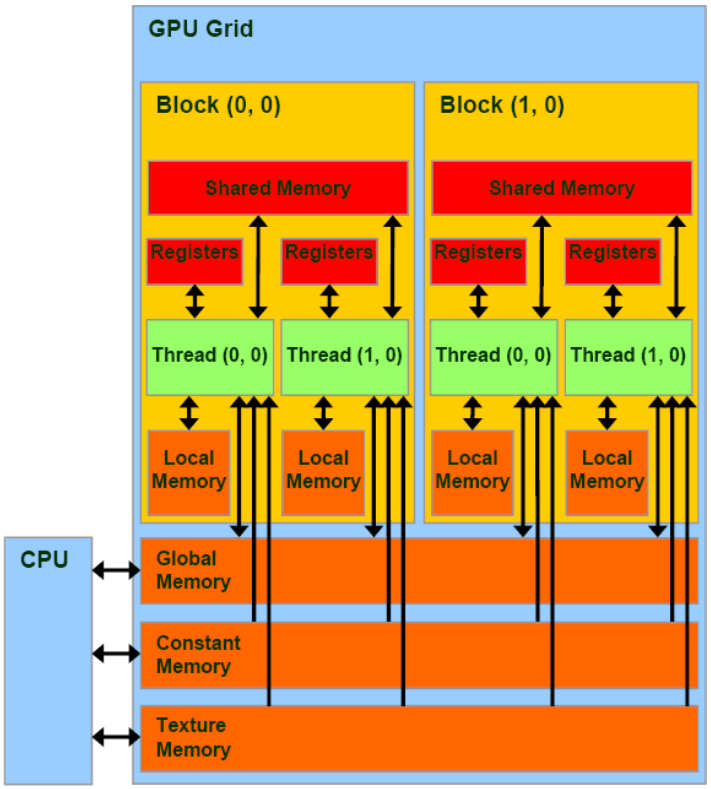


FIGURE 3.20 – Grille de blocs, composés par *threads*. (Source : Cuda Toolkit Documentation, NVIDIA.)

outils sont présentés : OpenCL, OpenACC, Kokkos et StarPU.

OpenCL. Le standard OpenCL [Stone et al., 2010] est un des outils hybrides les plus étendus. Ce standard ouvert est défini par le consortium *Khronos Group* (AMD, Apple, Google, Intel, NVIDIA, etc.). OpenCL offre aux développeurs un environnement uniformisé pour programmer différents types de machines : les CPU multicœurs SIMD, les GPU, les architectures de type Cell ou encore des processeurs spécialisés tels que les processeurs de traitement du signal et certains FPGA (*Field-Programmable Gate Array*). La compatibilité avec OpenCL est assurée par l'industriel qui fournit un pilote spécifique transformant les noyaux de calculs OpenCL en instructions pour son matériel. Ce pilote correspond à une version spécifique du standard qui permet à l'utilisateur de déterminer les fonctionnalités supportées par son matériel. Cependant, l'aspect performance nécessite très souvent des adaptations très spécifiques aux catégories de matériels, en termes de gestion mémoire et de répartition des sous-tâches sur le domaine de calcul. Il est souvent coûteux lorsque c'est possible et en général difficile (dans des conditions de qualité de code industriel) d'obtenir un seul et même code optimal pour de multiples plateformes.

OpenACC. Le standard OpenACC [Levesque et al., 2012] est un standard développé par Cray, CAPS, NVIDIA et PGI et présenté en 2012. Ce standard a comme objectif la simplification des codes parallèles sur des systèmes comprenant des CPU multicœurs et GPU. Dans ce sens, les membres de OpenACC font partie aussi du standard OpenMP et cherchent à faire que OpenACC soit une extension de OpenMP pour les utilisateurs

des cartes graphiques non NVIDIA. Les spécifications sont donc très similaires que pour OpenMP : les utilisateurs utilisent directives de préprocesseurs (voir section 3.2.1.1) sur le code *C*, *C++* ou Fortran afin d'identifier les régions à paralléliser. OpenACC dispose aussi d'une librairie incluant des fonctions telles que `acc_get_num_devices()`, `acc_async_test()`, `acc_init()`, `acc_malloc()`, etc., pour gérer l'utilisation des cartes graphiques de manière directe.

Kokkos. Kokkos [Carter Edwards et al., 2014] est une bibliothèque de *templates* *C++*. Elle est développée par *Sandia National Laboratories* pour permettre la portabilité du code sur différentes architectures, notamment CPU et GPU, dont le but est de conserver la même performance qu'avec le code écrit spécifiquement pour ces architectures. En fait, Kokkos cache à l'utilisateur les modèles de programmation spécialisés en utilisant de manière opaque CUDA pour NVIDIA GPU et `pThread` et OpenMP pour CPU et Intel Xeon Phi. Kokkos met l'accent sur l'impact des accès mémoire pour les différentes architectures. Afin de permettre l'exécution du code parallèle à granularité fine, les données sont stockées dans des vecteurs multidimensionnels flexibles.

StarPU. La bibliothèque StarPU [Augonnet et al., 2011] offre aux programmateurs la gestion haut niveau des dépendances, des transferts des données et un équilibrage des tâches sur des systèmes CPU/GPU/MIC. Cette bibliothèque peut s'utiliser via des `pragma` haut niveau pour le compilateur GCC, des `pragma` OpenMP ou une API *C/C++*. Elle peut aussi intégrer MPI, et elle est compatible avec OpenCL et les systèmes Linux, Windows et MacOS.

3.2.3 Les compilateurs

Il est à noter que les compilateurs proposent tous des optimisations automatiques de code afin de tirer parti des spécificités d'une architecture cible pour un code donné. Ils appliquent un certain nombre de techniques de haut niveau, dont les plus courantes sont :

- **auto parallélisation** qui détecte les sections parallélisables des données - d'un code pour les déporter sur plusieurs *threads* de calculs en parallèle ;
- **vectorisation** qui détecte les traitements réguliers sur des données pour utiliser automatiquement des instructions SIMD ;
- **déroulage de boucle** pour optimiser l'ordre des opérations et tirer parti du pipeline ;
- **extension inline** pour réduire le temps d'exécution ainsi que la consommation mémoire remplaçant un appel de fonction par le code de cette fonction ;
- **optimisation inter procédurale** qui vise à prendre en compte l'imbrication des fonctions pour appliquer des optimisations supplémentaires aux instructions.

La vectorisation et l'auto parallélisation montrent des résultats encourageants. Mais les compilateurs modernes tels que Intel C++ Compiler (ICC), GNU Compiler Collection (GCC) ou Microsoft Visual Studio Compiler (MSVC), ne détectent pas tous

les cas où ces optimisations seraient applicables. Ces compilateurs peinent à optimiser des algorithmes complexes lorsque les structures algorithmiques ne rentrent plus dans les schémas classiques (par exemple, avec de nombreuses boucles imbriquées...). En conséquence, lorsque les noyaux de calculs sont trop complexes, il faut avoir recours à une parallélisation et une écriture SIMD manuelles. Le programmeur peut réaliser des modifications algorithmiques de haut niveau telles que changer l'ordre des boucles afin d'exprimer le parallélisme de son application, optimiser les accès mémoire (cohérence spatiale et temporelle), choisir le grain du parallélisme souhaité, actions que le compilateur ne peut réaliser seul. Une grande partie des travaux réalisés pendant cette thèse utilisent ces idées.

3.3 Architectures ciblées

La maquette développée dans le cadre de cette thèse est destinée à fonctionner sur des plateformes matérielles similaires à celles sur lesquelles CIVA est déjà utilisé. Ces machines sont des stations de calcul disposant de CPU souvent plus puissants que les ordinateurs personnels grand public et disposant potentiellement d'un GPU destiné au calcul. En particulier, ces travaux s'intéressent aux architectures suivantes :

— **CPU Intel x86** déjà supportés par les versions courantes de CIVA et qui représentent une part majoritaire des CPU dans le monde des stations de calcul.
— **GPU NVIDIA** compatible CUDA disposant d'un support du constructeur pour le renouvellement et la mise à jour des outils de développement. À noter que des codes spécifiques à ces GPU sont déjà intégrés pour certaines fonctionnalités de CIVA.

Afin de pouvoir mesurer les performances sur un plus grand panel de machines disponibles (multiples générations de CPU et de GPU), la maquette a été développée de manière portable entre systèmes Windows et systèmes Linux.

3.3.1 Outils natifs et compilateurs

Ces travaux prennent place dans le cadre d'une étude de l'adéquation algorithme/architecture des simulations par éléments finis d'un contrôle non destructif par ultrasons. Les outils hybrides (sous-section 3.2.2) n'ont pas été retenus, afin de ne pas dépendre de composants fonctionnant de manière opaque et parfois peu paramétrables.

Ainsi, en utilisant des outils natifs, on espère l'obtention d'implémentations au plus proche d'une plateforme donnée (gestion mémoire explicite, utilisation d'instructions SIMD sur CPU...). De plus, dans l'objectif d'une intégration de ces développements dans une version future de CIVA, l'utilisation d'outils natifs permet de réduire les dépendances à des composants extérieurs, tout en maintenant la portabilité des performances par rapport aux outils.

Pour le calcul sur CPU Intel, le choix s'est porté sur l'utilisation de l'outil OpenMP pour la programmation *multithread* en raison de sa simplicité d'utilisation et de son

universalité : les calculs de simulation d' éléments finis nécessitent un grand parallélisme de données avec des calculs réguliers, cette bibliothèque est la plus adaptée et la moins intrusive en plus d'être un standard massivement disponible. La bibliothèque Intel TBB n'a pas été choisie pour paralléliser le code afin de réduire les dépendances du code développé. Le compilateur ICC est choisi pour sa bonne capacité de vectorisation. Enfin, dans les études portant sur la SIMDisation (sous-sous-section 3.2.1.2), l'utilisation directe des instructions SIMD est privilégiée au lieu de s'en servir via des outils externes.

Pour les GPU de marque NVIDIA, uniquement CUDA natif est utilisé pour le développement des noyaux de calcul, permettant une maîtrise très fine des traitements et pour pouvoir utiliser tout l'environnement logiciel fourni par NVIDIA (profiler, débogueur...). Sur GPU, NVIDIA se place en leader du calcul à haute performance depuis bientôt quinze ans et met à jour continuellement son offre matérielle ainsi que les outils de développement associés. Cette technologie présente, a priori, une stabilité suffisante pour justifier son utilisation sur des codes à usage industriel. Le compilateur NVIDIA CUDA Compiler (NVCC) a été choisie pour les codes visant les GPU NVIDIA.

3.4 Synthèse du chapitre

Ce chapitre a présenté les principales architectures parallèles (CPU, MIC, GPU), en faisant spécialement attention aux détails et à l'évolution des CPU Intel et GPU NVIDIA. Ces deux matériels sont ceux retenus pour les travaux de cette thèse, puisqu'ils sont communément utilisés pour les simulations traitées. Plusieurs outils natifs, hybrides et compilateurs ont été présentés pour le traitement des parallélismes de tâches et des données. Seuls les outils natifs et les compilateurs d'Intel et NVIDIA sont retenus.

Chapitre 4

La méthode des éléments finis spectraux

Sommaire

Le code CIVA-SFEM, présenté à la section 2.4, permet de résoudre les équations de propagation des ondes ultrasonores dans des milieux fluides (ondes acoustiques) ou dans des solides (ondes élastodynamiques). Dans le cadre de ce travail de thèse, l'équation acoustique a été choisie pour étudier les optimisations informatiques permettant d'accélérer les temps de calcul sur des architectures multicœurs SIMD. À la suite de la présentation de la méthode de discrétisation spatiale et temporelle utilisée en CIVA-SFEM, une solution numérique à cette équation va être proposée. Enfin, une analyse des noyaux de calcul qui constituent l'essentiel de la charge pour les processeurs fermera le chapitre.

Les noyaux de calcul des équations élastodynamiques sont de nature similaire à ceux des ondes acoustiques, mais impliquent des ratios entre opérations flottantes et volumes

de données différents. La façon d'extrapoler les résultats obtenus pour les ondes acoustiques aux ondes élastodynamiques est présentée dans le chapitre 8.

4.1 Éléments Finis pour la propagation d'ondes en régime temporel

Dans le cas des ondes acoustiques, l'inconnue est une fonction scalaire dépendant de l'espace et du temps qui représente la pression p dans un fluide. On cherche à résoudre l'équation des ondes sur un domaine spatial fini Ω. L'inconnue appartenant à l'espace des solutions admissibles $\mathbb{V}(\Omega)$ vérifie l'équation des ondes, sur laquelle on applique la Méthode des Éléments Finis Spectraux (SFEM pour *Spectral Finite Element Method* en anglais). Cette section détaille le déroulement de la méthode.

4.1.1 Formulation forte et faible

On considère le modèle de propagation des ondes acoustiques linéaires. Ce modèle est composé d'une équation aux dérivées partielles (EDP) faisant intervenir la dérivée seconde en espace et temps décrivant la propagation acoustique dans un domaine de calcul Ω, une condition limite (CL) de Neumann vérifiée sur le bord du domaine $\partial\Omega$ et des conditions initiales (CI). Dans l'équation apparaissent la vitesse des ondes acoustiques dans le fluide c, un champ source f et la normale extérieure au domaine $\underline{n}$.

$$\begin{cases} \dfrac{\partial^2}{\partial t^2} p - c^2 \Delta p = f, \text{ dans } \Omega, \\ \underline{\nabla} p \cdot \underline{n} = 0, \text{ sur } \partial\Omega, \\ p|_{t=0} = p_0, \ \dfrac{\partial}{\partial t} p|_{t=0} = p_1. \end{cases}$$

En utilisant la première identité de Green et la condition limite, on obtient la formulation faible

$$\begin{cases} \text{Trouver } p(t) \in \mathbb{V}, \forall v \in \mathbb{V}, \\ \dfrac{\mathrm{d}^2}{\mathrm{d}t^2}(p, v)_{\mathcal{L}^2(\Omega)} + a(p, v) = b(v). \end{cases} \tag{4.1}$$

où

$$\begin{cases} a(p, v) = \displaystyle\int_\Omega c^2 \underline{\nabla} p \underline{\nabla} v \, \mathrm{d}\Omega, \\ b(v) = \displaystyle\int_\Omega f v \, \mathrm{d}\Omega. \end{cases}$$

Cette formulation faible est équivalente à la formulation forte. C'est un problème bien posé : la solution existe, elle est unique et elle dépend de façon continue des données. En pratique, $\mathbb{V}$ est l'ensemble des fonctions de carré intégrable, et dont leurs dérivées sont de carré intégrable.

4.1.2 Espace d'approximation et discrétisation spatiale

La méthode qui nous intéresse est un cas particulier de l'approximation de Galerkin. Notre espace de discrétisation $\mathbb{V}_h(\Omega)$ respecte les trois conditions suivantes :

1. Dimension finie : $\dim(\mathbb{V}_h(\Omega)) < +\infty$,
2. Consistance de la discrétisation : $\mathbb{V}_h(\Omega) \subset \mathbb{V}(\Omega)$,
3. Convergence vers la solution exacte : $\mathrm{dist}(\mathbb{V}_h(\Omega), \mathbb{V}(\Omega)) \underset{h\to 0}{\longrightarrow} 0$.

Si on considère que $\dim(\mathbb{V}_h(\Omega)) < +\infty$ et $\mathbb{V}_h(\Omega) \subset \mathbb{V}(\Omega)$, la formulation faible (4.1) peut alors s'écrire de manière discrète

$$\begin{cases} \text{Trouver } p_h(t) \in \mathbb{V}_h, \forall v_h \in \mathbb{V}_h, \\ \dfrac{\mathrm{d}^2}{\mathrm{dt}^2}(p_h, v_h)_{\mathcal{L}^2(\Omega)} + a(p_h, v_h) = b(v_h). \end{cases} \tag{4.2}$$

En supposant que la dimension de $\mathbb{V}_h$ est N et qu'il est engendré par les fonctions de base lagrangiennes (voir sous-section 4.1.4 pour plus de détails), $\mathbb{V}_h = \mathrm{vect}\{\varphi_i\}_{i=1}^N$, la solution du problème discret s'exprime dans cette base

$$p_h(\underline{x}, t) = \sum_{i=1}^{N} \xi_i(t)\varphi_i(\underline{x}). \tag{4.3}$$

Pour obtenir une formulation matricielle, un système équivalent des équations différentielles ordinaires (EDO), on utilise $v_h = \varphi_1, .., \varphi_N$ dans (4.2)

$$\begin{cases} \mathbb{M}\dfrac{\mathrm{d}^2}{\mathrm{dt}^2}\overrightarrow{P} + \mathbb{K}\overrightarrow{P} = \overrightarrow{B}, \\ \overrightarrow{P}(0) = \overrightarrow{P}_0, \ \dfrac{\mathrm{d}}{\mathrm{dt}}\overrightarrow{P}(0) = \overrightarrow{P}_1, \end{cases} \tag{4.4}$$

où les vecteurs $\overrightarrow{P}$ et $\overrightarrow{B}$ sont

$$\overrightarrow{P}(t) = \begin{pmatrix} \xi_1(t) \\ \vdots \\ \xi_N(t) \end{pmatrix}, \ \overrightarrow{B} = \begin{pmatrix} b(\varphi_1) \\ \vdots \\ b(\varphi_N) \end{pmatrix}. \tag{4.5}$$

et les matrices discrètes $\mathbb{M}$ et $\mathbb{K}$ sont de la forme

$$\mathbb{M} = ((\varphi_i, \varphi_j)_{\mathcal{L}^2(\Omega)})_{i,j=1}^N, \ \mathbb{K} = (a(\varphi_i, \varphi_j))_{i,j=1}^N. \tag{4.6}$$

4.1.3 Discrétisation temporelle

La discrétisation temporelle sur le système EDO a pour objectif d'approcher l'opérateur différentiel d'ordre 2 en temps au moyen du développement de Taylor. En considérant un pas de temps $\Delta\mathrm{t}$, avec une hypothèse de régularité de la solution, le développement de Taylor à l'ordre 2 est

$$\frac{\mathrm{d}^2}{\mathrm{dt}^2}\overrightarrow{P}(t) = \frac{\overrightarrow{P}(t+\Delta\mathrm{t}) - 2\overrightarrow{P}(t) + \overrightarrow{P}(t-\Delta\mathrm{t})}{\Delta\mathrm{t}^2} + O(\Delta\mathrm{t}^2).$$

En notant $\overrightarrow{P}^n$ la solution approchée de $\overrightarrow{P}(n\Delta\mathrm{t})$ et en utilisant un schéma centré d'ordre deux (schéma *saute-mouton*), (4.4) s'exprime comme

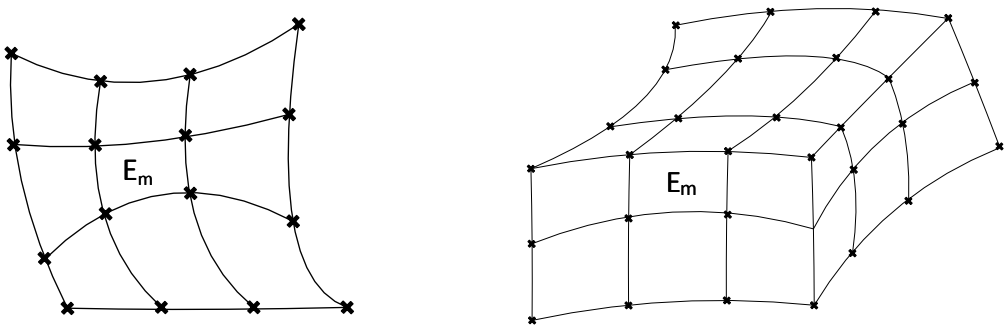


FIGURE 4.1 – À gauche, exemple de maillage 2D formé par 3×3 éléments quadrilatères. À droite, un cas 3D formé par 3×2×3 éléments parallélépipèdes.

$$\mathbb{M}\frac{\vec{P}^{n+1} - 2\vec{P}^{n} + \vec{P}^{n-1}}{\Delta t^2} + \mathbb{K}\vec{P}^{n} = \vec{B}^{n}.$$

Sachant que l'inconnue est $\vec{P}^{n+1}$, il faut inverser la matrice $\mathbb{M}$. Dans la méthode SFEM, la discrétisation spatiale permet d'obtenir une matrice de masse diagonale (voir sous-section 4.1.6). Dans ce contexte, l'inverse est un calcul direct, et le schéma temporel est dit explicite. Cependant, ce schéma doit satisfaire la condition de stabilité de Courant-Friedrich-Lewy (CFL), liée à la conservation de l'énergie discrète

$$\Delta t \leq \frac{2}{\sqrt{\rho(\mathbb{M}^{-1}\mathbb{K})}},$$

où ρ es le rayon spectral, la valeur absolue de la plus grande valeur propre de la matrice d'entrée. Finalement, l'équation complètement discrète est

$$\vec{P}^{n+1} = \mathbb{M}^{-1}(\vec{B}^{n} - \Delta t^2 \mathbb{K}\vec{P}^{n}) + 2\vec{P}^{n} - \vec{P}^{n-1}. \tag{4.7}$$

4.1.4 Discrétisation spatiale par des éléments finis

On considère que le domaine Ω est un maillage quadrangulaire (2D) ou hexaédrique (3D) conforme composé de M éléments, $\overline{\Omega} = \cup_{m=1}^{M} E_m$ (voir Figure 4.1). Le maillage est conforme s'il vérifie trois propriétés :

1. les éléments E_m ne sont pas vides,
2. les éléments E_m ne se recouvrent pas,
3. les frontières d'éléments (arêtes en 2D, faces en 3D) soit appartiennent à deux éléments, soit appartiennent au bord du domaine.

On suppose que le maillage est constitué de mailles non dégénérées, c'est-à-dire, qu'il existe pour chaque maille E_m une transformation bijective différentiable transformant la maille de référence $\hat{E}$ vers cette maille (Figure 4.2).

L'espace de discrétisation $\mathbb{V}_h$ s'exprime en faisant apparaître les transformations $F_m : \hat{E} \longrightarrow E_m$,

$$\mathbb{V}_h = \{p_h \in \mathcal{C}^0(\overline{\Omega}), \forall m = 1, \ldots, M \; p_h|_{E_m} = \hat{p}_h \circ F_m^{-1}, \hat{p}_h \in \mathbb{Q}^q(\hat{E})\}.$$

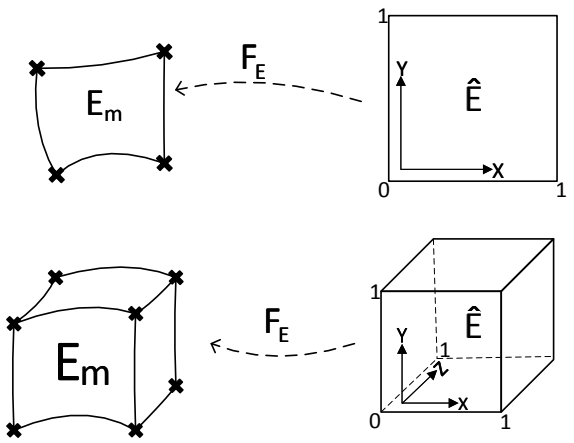


FIGURE 4.2 – Schéma de transformation d'une maille vers la maille de référence unitaire, carré en 2D et cube en 3D.

Ainsi, chaque fonction de l'espace d'approximation a une représentation locale sur la maille de référence. Par la suite, les fonctions de bases de l'espace d'approximation sont définies à partir de fonction de bases locales.

Soit $\hat{E}$ une maille de référence quelconque (non nécessairement le carré unité) et un espace de polynôme $P(\hat{E})$ sur cette maille. En considérant $\hat{\Xi} = \{\underline{\hat{\xi}}_{\hat{i}}\}_{\hat{i}=1}^{\hat{N}}$ un ensemble des points, appelé degrés de liberté (DoF pour *Degrees of Freedom* en anglais), disposé sur la maille de référence, on définit un élément fini unisolvant comme étant le triplet $(\hat{E}, \hat{\Xi}, P(\hat{E}))$, vérifiant la propriété suivante

$$\forall \alpha_1, \cdots, \alpha_{\hat{N}} \in \mathbb{R}^{\hat{N}}, \quad \exists! p \in P(\hat{E}) \ : \ \forall \hat{i} = 1, \cdots, \hat{N} \quad p(\underline{\hat{\xi}}_{\hat{i}}) = \alpha_{\hat{i}}.$$

Si la propriété (fondamentale) d'unisolvance est vérifiée, elle signifie que tout polynôme est défini uniquement par les valeurs qu'il prend aux DoF. La principale conséquence de cette propriété est l'existence de fonctions de base Lagrangienne locales, $\{\hat{\tau}_{\hat{i}}\}_{\hat{i}=1}^{\hat{N}}$ définies par

$$\forall \hat{i}, \hat{j} = 1, \cdots, \hat{N} \quad \hat{\tau}_{\hat{i}}(\underline{\hat{\xi}}_{\hat{i}}) = \delta_{\hat{i}\hat{j}},$$

où

$$\delta_{\hat{i}\hat{j}} = \begin{cases} 1 \text{ si } \hat{i} = \hat{j} \\ 0 \text{ si } \hat{i} \neq \hat{j} \end{cases}$$

Les conditions suffisantes pour assurer l'unisolvance sont :

1. le nombre de DoF est égal à la dimension de l'espace de polynôme ;
2. les positions de DoF sont toutes distinctes.

À partir de l'élément fini de référence, on peut définir un élément fini équivalent sur chaque maille en utilisant la transformation F_m pour définir les degrés de liberté dans le plan physique, que l'on peut noter $\Xi_E = \{\underline{\xi}_{\hat{i}}^E\}_{\hat{i}=1}^{\hat{N}}$. Aussi, on en déduit des fonctions de bases locales définies dans le plan physique par

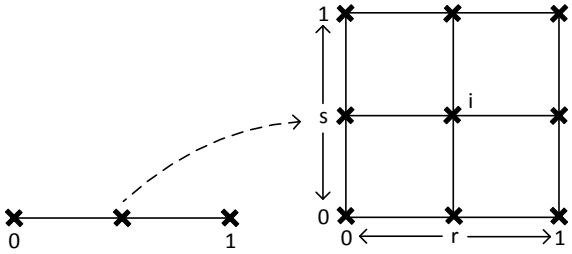


FIGURE 4.3 – Segment unitaire utilisé pour créer l'élément carré $[0;1]^2$.

$$\forall \hat{i}, \hat{j} = 1, \cdots, \hat{N} \quad \tau_{\hat{i}}^{E}(\underline{\xi}_{\hat{j}}^{E}) = \delta_{\hat{i}\hat{j}}.$$

Les degrés de liberté dans le plan physique sur le maillage sont $\Xi = \{\underline{\xi}_i\}_{i=1}^{N}$, l'union des Ξ_E en supprimant les coordonnées redondantes. Les fonctions de base globale sont définies par

$$\forall i, j = 1, \cdots, N \quad \phi_i(\underline{\xi}_j) = \delta_{ij},$$

où

$$\forall m = 1, \ldots, M \ \phi_i|_E = \tau_{\hat{i}}^{E} \circ F_m^{-1}.$$

La correspondance entre un DoF local $\hat{i}$ d'un élément m et le DoF global et assuré par la fonction l_g ou *local to global* définie comme

$$\begin{array}{rccc} l_g \ : & [1, \hat{N}] \times [1, M] & \longrightarrow & [1, N] \\ & (\hat{i}, m) & \longrightarrow & i \end{array}$$

Attention, les $\underline{\xi}$ sont partagés sur les frontières d' éléments et cela entraîne des problèmes de concurrence si les éléments sont utilisés de manière parallèle (cet aspect est traité en section 5.6).

4.1.5 Les éléments finis spectraux

Les éléments finis spectraux se construisent exclusivement sur des quadrilatères en 2D ou des hexaèdres en 3D. C'est-à-dire que l'élément fini de référence, carré ou cube unitaire, est construit par produit tensoriel d'un élément fini unidimensionnel.

Définissons $([0;1], \hat{\Xi}_{1D}, \mathbb{P}_{1D}^{q}([0;1]))$ un élément fini 1D d'ordre q sur le segment $[0;1]$ avec une distribution de $q+1$ points. L'élément fini 2D ou 3D est obtenu par produit de cet élément fini 1D, c'est-à-dire un total de $\hat{N} = (q+1)^d$, où d est la dimension de l'élément. Voir Figure 4.3.

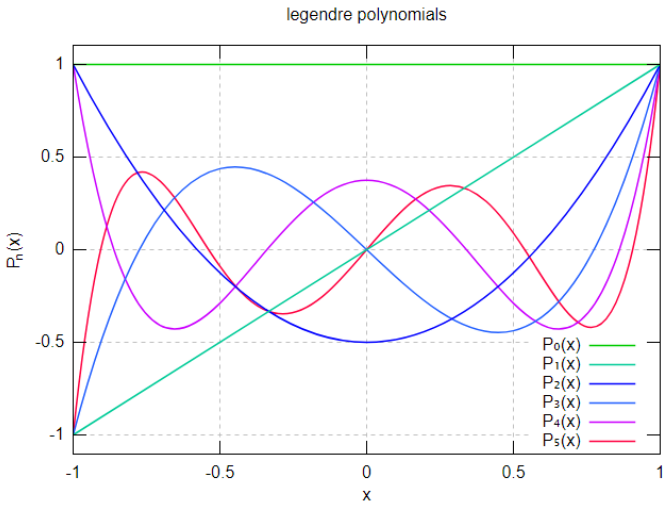


FIGURE 4.4 – Polynômes de Legendre.

Plus précisément, on définit (par exemple, en 2D) $\mathbb{Q}^q_{2D}(\hat{E}) = \mathbb{P}^q_{1D}([0;1])\times\mathbb{P}^q_{1D}([0;1])$ l'espace des polynômes produit d'ordre q ($\mathbb{Q}^q$ est dit tensorielle). Puis on se donne une numérotation locale des DoFs telle que

$$\underline{\hat{\xi}}_{\hat{i}} = (\hat{\xi}_{\hat{r}}, \hat{\xi}_{\hat{s}}), \forall \hat{i} = 1, \dots, \hat{N}, \hat{i} = \hat{s}(q+1) + \hat{r}.$$

Alors la position des DoFs s'exprime simplement par concaténation :

$$\underline{\hat{\xi}}_{\hat{i}} = (\hat{\xi}^{1D}_{\hat{r}}, \hat{\xi}^{1D}_{\hat{s}})^T, \forall \hat{i} = 1, \dots, \hat{N}$$

De plus, les fonctions de bases locales s'expriment comme le produit de fonctions de bases unidimensionnelles. Si $\underline{\hat{x}} = (\hat{x}_1, \hat{x}_2) \in \hat{E}$,

$$\hat{\tau}_{\hat{i}}(\underline{\hat{x}}) = \hat{\tau}^{1D}_r(\hat{x}_1)\hat{\tau}^{1D}_s(\hat{x}_2), \forall \hat{i} = 1, \dots, \hat{N} \tag{4.8}$$

La définition des fonctions de bases locales et de leurs DoFs associés revient donc à se donner une distribution de points 1D. La méthode des éléments finis spectraux d'ordre q repose sur les points de Gauss-Lobatto [Durufle et al., 2009], qui sont les racines de la dérivée du polynôme de Legendre (voir Figure 4.4) d'ordre q (soit $q-1$ points), auxquelles on ajoute les deux extrémités du segment $[0;1]$. Voir Figure 4.5.

L'unisolvance de l'élément fini 1D est assurée par construction, car le nombre de points ainsi construits est égal à $q+1$ et ils sont tous distincts. On verra ultérieurement que l'utilisation de ces points conduit à des propriétés numériques capitales permettant la technique de condensation de masse.

4.1.6 Condensation de la matrice de masse

Pour résoudre (4.7), il faut inverser la matrice de masse $\mathbb{M}$. On applique la technique de la condensation de masse [Cohen et al., 1994] pour obtenir une matrice de masse diagonale, explicitement inversible. En décomposant l'intégration des éléments de la matrice de masse (4.6) sur les fonctions de bases, on a

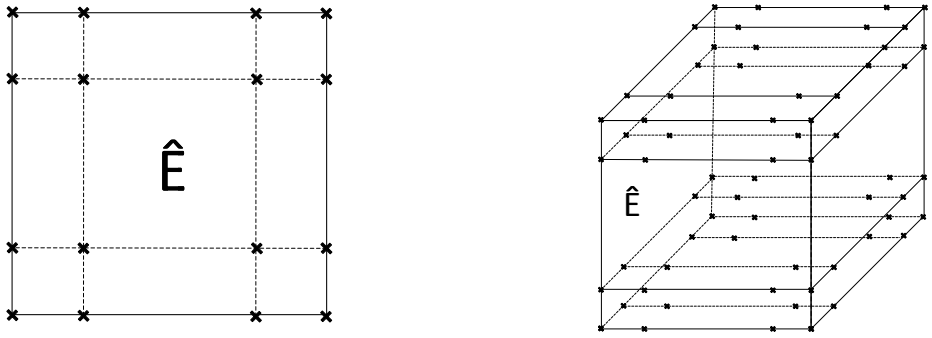


FIGURE 4.5 – Élément fini de référence avec les points de quadrature de Gauss-Lobatto d'ordre 3 en chaque dimension. À gauche, bidimensionnel avec un total de 16 points ; à droite tridimensionnel et 64 points.

$$\mathbb{M}_{ij} = (\varphi_i, \varphi_j)_{\mathcal{L}^2(\Omega)} = \int_{\mathrm{supp}(\varphi_i) \cap supp(\varphi_j)} \varphi_i \varphi_j \, \mathrm{d}\Omega = \sum_{m=1}^{M^{ij}} \int_{E_m} \tau_{\hat{i}} \tau_{\hat{j}} \, \mathrm{d}\Omega.$$

où le support de φ_i, noté $\mathrm{supp}(\varphi_i)$, sont les éléments qui possèdent le DoF associé à φ_i et M^{ij} est donc le nombre des éléments de l'intersection des supports de φ_i et φ_j. On rappelle que $i = l_g(\hat{i}, m)$ et $j = l_g(\hat{j}, m)$. Après changement de variable vers l'élément fini de référence, un nouveau terme apparaît correspondant au déterminant de la matrice jacobienne de la transformation F_m.

$$\sum_{m=1}^{M^{ij}} \int_{E_m} \tau_{\hat{i}} \tau_{\hat{j}} \, \mathrm{d}\Omega = \sum_{m=1}^{M^{ij}} \int_{\hat{E}} \hat{\tau}_{\hat{i}} \hat{\tau}_{\hat{j}} J_{F_m} \, \mathrm{d}\Omega. \tag{4.9}$$

En pratique, le calcul de l'intégrale d'une fonction f continue définie sur $\hat{E}$ est obtenu à partir d'une formule de quadrature d'ordre q. Cette formule est définie par un ensemble de N_q points et poids $\{(\omega_p, \underline{x}_p)\}_{p=1}^{N_q}$ tel que

$$\int_{\Omega} f(\underline{x}) \, \mathrm{d}\Omega \approx \sum_{p=1}^{N_q} \omega_p f(\underline{x}_p).$$

Si on applique ce type d'approximation sur (4.9), on obtient

$$\sum_{m=1}^{M^{ij}} \sum_{\hat{p}=1}^{N_q} \omega_{\hat{p}} \hat{\tau}_{\hat{i}}(\underline{x}_{\hat{p}}) \hat{\tau}_{\hat{j}}(\underline{x}_{\hat{p}}) J_{F_m}.$$

On remarque alors que si les points de quadrature sont identiques aux DoF définissant les fonctions de bases lagrangiennes, alors

$$\sum_{m=1}^{M^{ij}} \sum_{\hat{p}=1}^{\hat{N}} \omega_{\hat{p}} \hat{\tau}_{\hat{i}}(\hat{\underline{\xi}}_{\hat{p}}) \hat{\tau}_{\hat{j}}(\hat{\underline{\xi}}_{\hat{p}}) J_{F_m} = \sum_{m=1}^{M^{ij}} \sum_{\hat{p}=1}^{\hat{N}} \omega_{\hat{p}} \delta_{\hat{i}\hat{p}} \delta_{\hat{j}\hat{p}} J_{F_m} = \sum_{m=1}^{M^{ij}} \omega_{\hat{i}} \delta_{\hat{i}\hat{j}} J_{F_m}.$$

La matrice de masse est diagonale puisque $\mathbb{M}_{ij} = 0$ si $i \neq j$. Les valeurs non nulles sont de la forme

$$\mathbb{M}_{ii} = \sum_{m=1}^{M^{ii}} \omega_{\hat{i}} J_{F_m}.$$

La technique de la condensation de masse permet d'obtenir une matrice de masse diagonale, au prix d'une approximation sur le calcul des intégrales. Trois conditions à

imposer à la quadrature sont nécessaires : stabilité, unisolvance et précision. La **stabilité** se rattache à la matrice de masse étant strictement positive, obligeant aux poids de quadrature la même propriété. L'**unisolvance** assure l'existence de fonctions de base lagrangienne locales et est atteinte avec des points de quadrature distincts comprenant les extrémités du segment. Finalement, la condition de **précision** se dérive de la possible différence des résultats entre le schéma numérique obtenu en utilisant la formule de quadrature et en utilisant l'intégral exact. On peut montrer [Cohen et al., 2001] que la précision nécessaire est atteinte par une quadrature exacte pour des polynômes d'ordre $2q-2$. **La quadrature de Gauss-Lobatto est l'unique qui amasse ces trois conditions** sur des éléments finis quadrangulaires ou hexaédriques. Il faut noter que pour les éléments tétraédriques, une telle formule de quadrature n'existe pas, et que la condensation de masse et plus complexe à mettre en œuvre.

4.1.7 Matrice de rigidité

L'autre matrice présente en (4.7) est la matrice de rigidité $\mathbb{K}$. Cette matrice est plus complexe puisque la méthode de condensation suivie par la matrice de masse n'est pas applicable. Cependant les étapes initiales sont les mêmes. En décomposant aussi la matrice de rigidité (4.6) sur les fonctions de bases, on a

$$\mathbb{K}_{ij} = (a(\varphi_i,\varphi_j))_{i,j=1}^{N} = \int_{\mathrm{supp}(\varphi_i)\cap supp(\varphi_j)} c^2 \underline{\nabla}\varphi_i \underline{\nabla}\varphi_j \,\mathrm{d}\Omega = \sum_{m=1}^{M^{ij}} \int_{E_m} c^2 \underline{\nabla}\tau_{\hat{i}} \underline{\nabla}\tau_{\hat{j}} \,\mathrm{d}\Omega.$$

On applique un changement de variable vers $\hat{E}$ et dans ce cas, à part le déterminant de la matrice jacobienne de F_m, suivant le théorème de dérivation des fonctions composées (ou règle de la chaîne) le gradient fait apparaître la dérivée de la transformation. La vitesse de l'onde acoustique dans le fluide c elle est aussi transformée en $\hat{c}$.

$$\sum_{m=1}^{M^{ij}} \int_{E_m} c^2 \underline{\nabla}\tau_{\hat{i}} \underline{\nabla}\tau_{\hat{j}} \,\mathrm{d}\Omega = \sum_{m=1}^{M^{ij}} \int_{\hat{E}} \hat{c} (DF^{-T} \underline{\nabla}\hat{\tau}_{\hat{i}})^T (DF^{-T} \underline{\nabla}\hat{\tau}_{\hat{j}}) J_{F_m} \,\mathrm{d}\Omega.$$

Si on substitue $DF^{-1} = \frac{co(DF)^T}{J_{F_m}}$, $co(DF)$ étant la matrice des cofacteurs de DF,

$$\sum_{m=1}^{M^{ij}} \int_{\hat{E}} (\underline{\nabla}\hat{\tau}_{\hat{i}})^T \left(\hat{c} \frac{co(DF)^T co(DF)}{J_{F_m}} \underline{\nabla}\hat{\tau}_{\hat{j}} \right) \mathrm{d}\Omega. \tag{4.10}$$

Si

$$\underline{\nabla}_{\underline{\hat{x}}} \hat{\tau}_{\hat{i}}(\underline{\hat{x}}) = \begin{pmatrix} \dfrac{\partial}{\partial \hat{x}_1} \hat{\tau}_{\hat{i}}(\underline{\hat{x}}) \\ \dfrac{\partial}{\partial \hat{x}_2} \hat{\tau}_{\hat{i}}(\underline{\hat{x}}) \end{pmatrix} \begin{matrix} \in \mathbb{P}_{1D}^{q-1}(\hat{E}) \times \mathbb{P}_{1D}^{q}(\hat{E}) \subset \mathbb{Q}_{2D}^{q}(\hat{E}) \\ \in \mathbb{P}_{1D}^{q}(\hat{E}) \times \mathbb{P}_{1D}^{q-1}(\hat{E}) \subset \mathbb{Q}_{2D}^{q}(\hat{E}) \end{matrix},$$

l'opérateur du gradient peut être projeté sur la base $\{\hat{\tau}_{\hat{i}}\}$ de forme

$$\underline{\nabla}_{\underline{\hat{x}}} \hat{\tau}_{\hat{i}}(\underline{\hat{x}}) = \sum_{\hat{j}=1}^{\hat{N}} \underline{\alpha}_{\hat{i}\hat{j}} \hat{\tau}_{\hat{j}}(\underline{\hat{x}}).$$

Les valeurs de $\underline{\alpha}_{\hat{i}\hat{j}}$ sont connues et ne dépendent que de $\{\hat{\tau}_{\hat{i}}\}_{\hat{N}}^{\hat{i}}$ sur $\hat{E}$ puisque

$$\underline{\nabla}_{\underline{\hat{x}}}\hat{\tau}_{\hat{i}}(\underline{\hat{\xi}}_{\hat{k}}) = \sum_{\hat{j}=1}^{\hat{N}} \underline{\alpha}_{\hat{i}\hat{j}}\hat{\tau}_{\hat{j}}(\underline{\hat{\xi}}_{\hat{k}}) = \underline{\alpha}_{\hat{i}\hat{k}}.$$

Si on introduit ce gradient dans (4.10)

$$\sum_{m=1}^{M^{ij}} \int_{\hat{E}} \left(\sum_{\hat{k}=1}^{\hat{N}} \underline{\alpha}_{\hat{i}\hat{k}}\hat{\tau}_{\hat{k}}(\underline{\hat{x}}) \right)^T \left(\hat{c}\frac{co(DF)^T co(DF)}{J_{F_m}} \left(\sum_{\hat{k}=1}^{\hat{N}} \underline{\alpha}_{\hat{j}\hat{k}}\hat{\tau}_{\hat{k}}(\underline{\hat{x}}) \right) \right) \mathrm{d}\Omega.$$

En appliquant la quadrature de Gauss-Lobatto de la sous-section précédente,

$$\sum_{m=1}^{M^{ij}} \sum_{\hat{p}=1}^{N_q} \omega_{\hat{p}} \left(\sum_{\hat{k}=1}^{\hat{N}} \underline{\alpha}_{\hat{i}\hat{k}}\hat{\tau}_{\hat{k}}(\underline{\hat{\xi}}_{\hat{p}}) \right)^T \left(\hat{c}\frac{co(DF)^T co(DF)}{J_{F_m}} \left(\sum_{\hat{k}=1}^{\hat{N}} \underline{\alpha}_{\hat{j}\hat{k}}\hat{\tau}_{\hat{k}}(\underline{\hat{\xi}}_{\hat{p}}) \right) \right),$$

et comme les points de quadrature sont les DoF définissant les fonctions de base lagrangienne, alors

$$\sum_{m=1}^{M^{ij}} \sum_{\hat{p}=1}^{\hat{N}} \omega_{\hat{p}} \underline{\alpha}_{\hat{i}\hat{p}}^T \left(\hat{c}\frac{co(DF)^T co(DF)}{J_{F_m}} \underline{\alpha}_{\hat{j}\hat{p}} \right).$$

Finalement,

$$\mathbb{K}_{ij} = \sum_{m=1}^{M^{ij}} A^T D A. \tag{4.11}$$

où A est définie comme la structure de taille $d \times \hat{N} \times \hat{N}$ correspondant à $\underline{\alpha}_{\hat{j}\hat{p}}$ où d = dimension, et

$$D = \left(\omega_{\hat{k}}\hat{c}\frac{co(DF)^T co(DF)}{J_{F_m}}\Big|_{\underline{\hat{\xi}}_{\hat{k}}} \right)_{\hat{k}=1}^{\hat{N}}. \tag{4.12}$$

4.2 Propriétés remarquables de la méthode SFEM du point de vue de l'optimisation informatique

4.2.1 Schéma explicite

Comme a été dit dans la section précédente, la méthode SFEM est basée sur les propriétés des quadratures de Gauss-Lobatto afin d'appliquer la méthode de condensation de masse et d'obtenir une matrice de masse diagonale. Grâce à ce point essentiel, la discrétisation temporelle peut se faire de manière explicite. Alors que dans nombre de solveurs éléments finis, l'essentiel du temps de calcul est mobilisé dans la résolution du système linéaire permettant de résoudre le schéma implicite, dans notre cas, la plus grande partie des opérations sont celles permettant de multiplier localement la matrice de rigidité (4.11) et le vecteur représentant la solution d'un élément.

Les résolutions de systèmes linéaires des problèmes éléments finis sont typiquement effectuées grâce à des méthodes itératives (de type Cholesky, BiCGStab, GMRES), méthodes qui nécessitent de nombreux produits matrice-vecteur sur matrices creuses. Ces opérations ne sont pas très propices à l'emploi efficient du parallélisme des machines (et de leurs contraintes associées), car elles sont de faible intensité arithmétique et sont essentiellement limitées par les accès mémoire. Dans notre cas, la situation est

beaucoup plus favorable.

4.2.2 Maillage structuré par blocs

Il faut également remarquer que la formulation SFEM s'applique à des éléments 2D de type quadrangulaire et 3D de type hexaédrique. En effet, l'obtention des points de quadrature en plusieurs dimensions se fait par produit tensoriel des polynômes 1D de Gauss-Lobatto. Cette caractéristique de la méthode SFEM est à double tranchant. Le point positif de cette propriété est bien évidemment la condensation de masse qui permet d'obtenir des schémas explicites très efficients. Le point négatif est que l'obtention de maillages quadrangulaires ou hexaédriques peut être une tâche ardue sur des géométries complexes. Alors que les algorithmes de maillage triangulaires et tétraédriques sont bien éprouvés et que de très nombreux outils commerciaux (ADINA [Bathe, 1986], Femap [Software, 2014], CENTAUR [CentaurSoft, 2018]) et open source (GMSH [Geuzaine et Remacle, 2009], Afront [Scheidegger et al., 2005], libMesh [Kirk et al., 2006]) existent, le maillage quadrangulaire et hexaédrique reste un sujet de recherche ouverte [Shepherd et Johnson, 2008].

Ce qui est une limitation du point de vue de la généricité des géométries traitables est plutôt un atout du point de vue de l'optimisation informatique. Partant d'une région d'intérêt (voir la Figure 4.6), CIVA-SFEM applique une méthode de décomposition du domaine pour établir les sous-domaines en positionnant les défauts comme des interfaces. Les défauts sont donc paramétrés comme interfaces qui bloquent la communication. Chacun des sous-domaines est maillé avec un maillage irrégulier différent (en certains cas très simples, comme la Figure 4.6, quelques sous-domaines peuvent être similaires, normalement ce n'est pas le cas). En effet, les maillages gérés par CIVA-SFEM sont des maillages irréguliers obtenus par transformations par polynômes des maillages réguliers. C'est-à-dire que les maillages sont géométriquement irréguliers, mais topologiquement réguliers. Sur le maillage régulier, l'indiçage (fonction l_g) des degrés de liberté se fait de manière implicite. Ce n'est pas nécessaire de rechercher les informations dans une table de connectivité pour connaître les relations entre mailles et degrés de liberté.

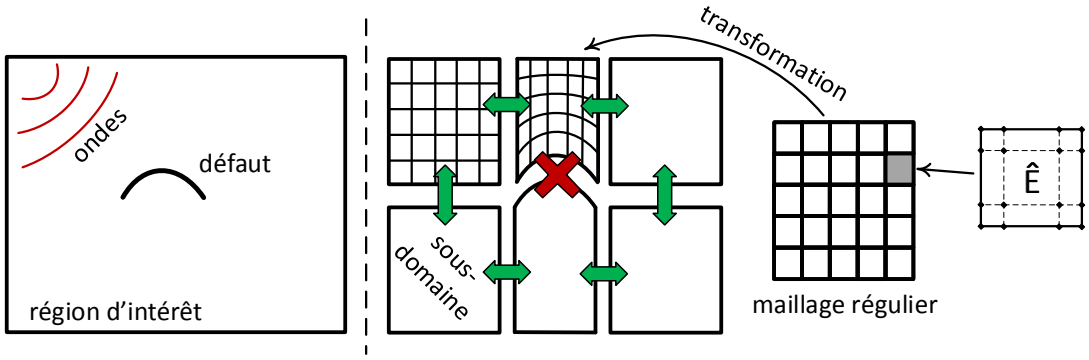


FIGURE 4.6 – De gauche à droite : région d'intérêt surfacique avec un défaut ; même région divisée en six sous-domaines ; maillage régulier après transformation ; élément fini de référence d'ordre 3.

4.3 Description du noyau de calcul PRL

Pour appliquer le schéma numérique (4.7), il faut effectuer une inversion de la matrice de masse diagonale et calculer son produit avec un vecteur ainsi que quelques additions et soustractions simples. En revanche, le produit concernant la matrice de rigidité est l'opération la plus complexe à faire.

La matrice de rigidité n'est pas assemblée ni stockée, sa taille est simplement trop importante. Faire les opérations ***à la volée*** a été choisi. L'optimisation de ce calcul s'avère, donc, fondamentale pour la performance globale du code.

Le noyau de calcul intéressant est le produit local de la matrice de rigidité $\mathbb{K}_{ij}$ avec le vecteur élément fini $\overrightarrow{P}_m^n$ représentant la solution d'un élément, où le produit va être accumulé sur $\overrightarrow{P}_m^{n+1}$. On définit donc la fonction PRL (Produit Rigidité Locale) comme

$$\texttt{PRL}(\overrightarrow{P}_m^n, \overrightarrow{P}_m^{n+1}) \text{ exécute } \overrightarrow{P}_m^{n+1} \mathrel{+}= \mathbb{K}_m \overrightarrow{P}_m^n = A^T D A \overrightarrow{P}_m^n. \tag{4.13}$$

Cette section présente comment découper la fonction PRL en tâches de calcul, décrire leurs formes et analyser l'intensité arithmétique pour décider des meilleures voies pour optimiser le temps de calcul. Afin de rendre ces équations plus lisibles, on simplifie les notations en écartant les références à la discrétisation temporelle. Pour cette étude, on présente le cas acoustique des ondes dans un matériau homogène, ce qui permet de simplifier les opérateurs. Cela signifie que chaque DoF ou point de l'élément représente une variable une simple ou double précision stockée en mémoire. Par ailleurs, l'étude a été aussi faite pour le cas élastique (chaque point étant deux ou trois variables en simple ou double précision) et les matériaux inhomogènes (voir le chapitre 8).

4.3.1 Argument et résultat de la fonction

La fonction PRL a l'argument $\overrightarrow{P}_m^n$, qu' on nomme U. Ce vecteur local est la solution dans le pas de temps n de l'élément m. Ce vecteur est récupéré du vecteur global suivant le schéma de la Figure 4.7. L'autre argument $\overrightarrow{P}_m^{n+1}$ est donc son correspondant

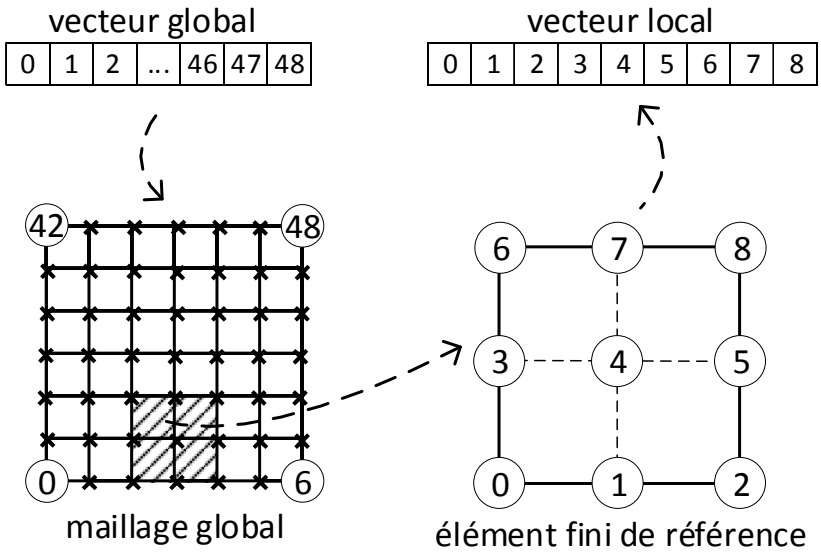


FIGURE 4.7 – Schéma du passage entre le vecteur global et local. La maille globale, en ce cas de taille 7×7, est située en mémoire en forme de vecteur. L'élément d'ordre 2 à lire est situé parmi ce vecteur. Une fois lu, il est traité comme un vecteur local de taille, en ce cas, 9.

Ordre	2	3	4	5	6	7	8	9
2D	9	16	25	36	49	64	81	100
3D	27	64	125	216	343	512	729	1000

TABLEAU 4.1 – Taille du vecteur d'entrée U et du vecteur de sortie V selon l'ordre d'approximation.

dans le pas de temps suivant, où le produit va être accumulé.

La taille de ce vecteur dépend exclusivement de l'élément fini de référence, c'est-à-dire de l'ordre d'approximation (voir Tableau 4.1 pour plusieurs exemples).

La fonction `PRL` est donc de la forme

$$\texttt{PRL}(U, V) \text{ exécute } V \mathrel{+}= A^T D A U. \tag{4.14}$$

4.3.2 Matrice du gradient

À la suite de la lecture du vecteur d'entrée U, la première opération est le produit de la matrice du gradient A par U. La dernière opération consiste à multiplier A^T par le résultat des autres opérations et accumuler sur V. La forme spécifique de cette matrice du gradient va être étudiée.

La matrice A de taille $d \times \hat{N} \times \hat{N}$, où d est la dimension de la géométrie, représente en chaque position (i, j) le vecteur gradient de la fonction de base i appliquée sur le point $\hat{\underline{\xi}}_{\hat{j}}$. On va supposer un espace bidimensionnel et un ordre d'approximation 2 afin de simplifier les idées qui suivent. Si $\hat{\underline{\xi}}_{\hat{j}} = (\hat{\xi}_1, \hat{\xi}_2)$, alors

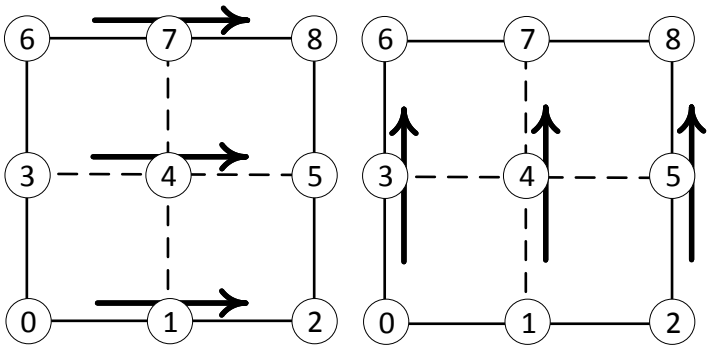


FIGURE 4.8 – À gauche, direction du gradient de la première dimension dans l'élément bidimensionnel fini de référence de taille 3×3. À droite, idem, mais pour la seconde dimension.

$$A_{\hat{i}\hat{j}} = \underline{\alpha_{\hat{i}\hat{j}}} = \left(\frac{\partial \hat{\tau}_{\hat{i}}}{\partial \hat{\xi}_1}(\underline{\hat{\xi}}_{\hat{j}}), \frac{\partial \hat{\tau}_{\hat{i}}}{\partial \hat{\xi}_2}(\underline{\hat{\xi}}_{\hat{j}}) \right)^T = \left(A^1_{\hat{i}\hat{j}}, A^2_{\hat{i}\hat{j}} \right)^T .$$

Il y a une matrice du gradient pour chaque dimension, et chacune correspond au gradient de la fonction de base dans la direction de cette dimension. Si on rappelle la caractéristique tensorielle de $\hat{\tau}_{\hat{i}}$, cette fonction de base locale s'exprime comme le produit de bases unidimensionnelles (voir (4.8)). Si on détaille A^1_{ij} (première dimension), où $\hat{i} = (\hat{r}, \hat{s})$ et $\hat{j} = (\hat{t}, \hat{u})$, alors

$$\frac{\partial \hat{\tau}_{\hat{i}}}{\partial \hat{\xi}_1}(\underline{\hat{\xi}}_{\hat{j}}) = \frac{\partial (\hat{\tau}^{1D}_{\hat{r}}(\underline{\hat{\xi}}_{\hat{t}}) \hat{\tau}^{1D}_{\hat{s}}(\underline{\hat{\xi}}_{\hat{u}}))}{\partial \hat{\xi}_1} = \frac{\partial \hat{\tau}^{1D}_{\hat{r}}(\underline{\hat{\xi}}_{\hat{t}})}{\partial \hat{\xi}_1} \hat{\tau}^{1D}_{\hat{s}}(\underline{\hat{\xi}}_{\hat{u}}).$$

En tenant en compte la base lagrangienne,

$$A^1_{\hat{i}\hat{j}} = \begin{cases} 0 & \text{si } \hat{s} \neq \hat{u} \\ \frac{\partial \hat{\tau}^{1D}_{\hat{r}}(\underline{\hat{\xi}}_{\hat{t}})}{\partial \hat{\xi}_1} & \text{si } \hat{s} = \hat{u} \end{cases}$$

Ainsi, seuls les gradients de la première dimension de fonctions de base qui sont dans la même ligne sont non nuls. Les ensembles des indices non nuls sont $\{0, 1, 2\}$, $\{3, 4, 5\}$ et $\{6, 7, 8\}$. Les gradients de la deuxième dimension sont non nuls dans la même colonne : $\{0, 3, 6\}$, $\{1, 4, 7\}$ et $\{2, 5, 8\}$. La Figure 4.8 montre ces ensembles dans les deux dimensions.

Grâce à la nature tensorielle des coordonnées des DoF sur l'élément fini de référence, les coefficients non nuls des ensembles sont les mêmes dans toutes les dimensions. Il y a 9 coefficients $\{a_k\}$ dans les deux matrices. La disposition des coefficients et représentée par la Figure 4.9.

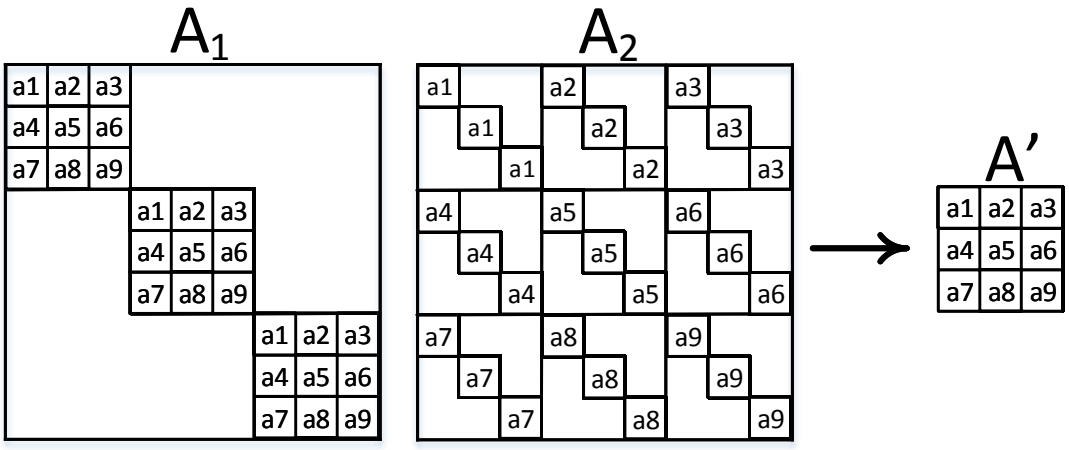


FIGURE 4.9 – À gauche, matrices des gradients des fonctions de base correspondant à la direction de la première et deuxième dimension, pour l'EFR de Figure 4.8. Ces matrices de taille 9×9 ont un total de 9 coefficients différents qui peuvent être stockés dans une matrice réduite, comme celle de droite.

Ces matrices peuvent être de taille assez importante. Vu que les matrices initiales A_1 et A_2 sont creuses et les coefficients sont répétés et fixes pour l'ensemble du maillage, ces coefficients ne sont calculés et stockés qu'une seule fois dans une matrice unique réduite et dense A'.

4.3.3 Transformation géométrique

Entre les produits des matrices du gradient A et A^T, il y a plusieurs opérateurs liés aux matériaux (la vitesse des ondes) et à la transformation F de chaque élément vers l'élément fini de référence. Ces opérateurs, nommées D en (4.12), sont appliquées sur chaque point de l'élément. Le produit du gradient est en réalité composé de deux ou trois produits différents (un pour chaque dimension). Donc, il faut savoir que $A^1 \times U$ génère la première coordonnée des points à traiter pour ces opérateurs, $A^2 \times U$ la deuxième coordonnée et $A^3 \times U$ la troisième, si elle existe. De même, les deux ou trois coordonnées résultant de ce produit, seront utilisées respectivement pour A^{1^T}, A^{2^T} et A^{3^T}.

Pour un cas acoustique homogène, la vitesse $\hat{c}$ est une constante. Le cofacteur de la matrice dérivée de la transformation, $co(DF)$, est une matrice de taille 2×2 ou 3×3 selon la dimension du problème. On suppose que la transformation est constante pour chaque élément, et donc le cofacteur l'est aussi. Finalement, le déterminant du jacobien, J_{F_m}, est constant aussi. De manière générale, ces coefficients varient entre les éléments et, suivant notre approche non assemblée, sont aussi calculés à la volée.

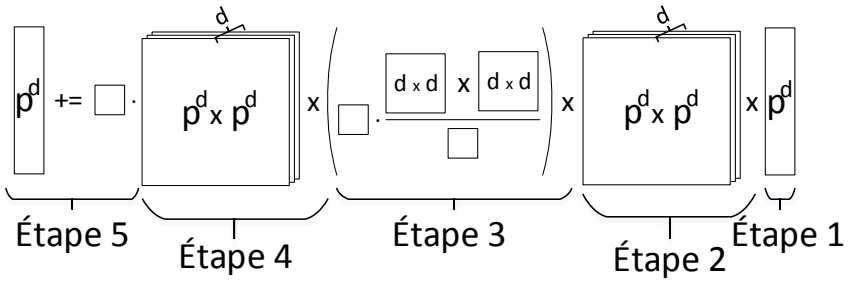


FIGURE 4.10 – Les étapes de l'équation visualisées sous forme de matrices, de vecteurs et de produits scalaires. La taille des objets est spécifiée, à l'exception des scalaires, en utilisant $p = ordre + 1$ et la dimension d des éléments.

4.4 Étapes du noyau de calcul

La fonction `PRL` est divisible en cinq tâches ou étapes fondamentales (représentées par la Figure 4.10) :

1. lecture du vecteur local U,
2. produit des matrices A par U,
3. produit de la matrice D par la combinaison du résultat de l'étape précédente,
4. produit des matrices A^T par le résultat de l'étape précédente et accumulation,
5. écriture des résultats en V.

Markall et al. [2013], qui étudient l'implémentation des méthodes des éléments finis en CPU et GPU, utilisent aussi une approche pour l'utilisation des matrices locales en trois étapes très similaires aux étapes de calcul précédentes. On s'intéresse ici à une fonction, donc on ajoute une étape de lecture et une d'écriture.

On utilise l'intensité arithmétique (IA), le ratio entre le nombre des calculs et les accès en mémoire, pour positionner le code. Une IA petite montre que l'algorithme est probablement limité par les accès en mémoire (*memory bound*). Les actions à mettre en œuvre dans ce cas doivent viser la diminution du nombre de lectures et une meilleure utilisation des mémoires caches. Une IA grande montre un algorithme probablement limité par les opérations (*compute bound*). Dans ce cas, la meilleure utilisation du matériel disponible (parallélisation et vectorisation) permet, généralement, des améliorations importantes en temps de calcul. La Figure 4.11 montre différents exemples d'algorithmes et leurs IA associées.

On a calculé l'IA des étapes du noyau de calcul. Il faut noter que ce noyau a quelques particularités pour comprendre l'IA obtenue : il y a une lecture multiple des données initiales afin de préparer le double/triple produit du gradient, les opérateurs de l'étape deux ne sont pas fusionnés et l'accumulation des résultats est faite pendant l'écriture des résultats. Selon le Tableau 4.2, l'intensité arithmétique est 0.5 pour un ordre 2, cohérente avec la complexité $O(1)$ de la Figure 4.11, puisque notre algorithme est composé des produits de matrices creuses avec des vecteurs. La formule générale est donnée

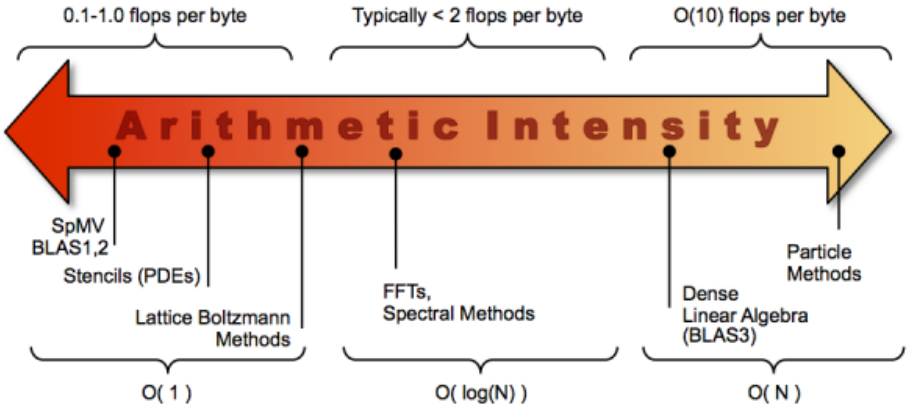


FIGURE 4.11 – IA des différents algorithmes. (Source : Performance and Algorithm Research : Roofline Performance Model, Berkeley Lab's Computational Research Division.)

Étape	MUL+ADD	LOAD+STORE	AI
1	0	9+9	0.0
2	54+54	162+54	0.5
3	162+36	225+108	0.6
4	54+54	162+54	0.5
5	9+18	27+9	0.7
Total	441	819	**0.5**

TABLEAU 4.2 – Initial IA_{2D} pour chaque étape, ordre 2.

par (4.15). Le noyau de calcul est donc limité pas les accès mémoire.

$$\begin{aligned} IA_{2D}^{\text{initial}}(p = ordre + 1) &= \frac{8p^3 + 25p^2}{16p^3 + 43p^2} \sim 0.5 \\ IA_{3D}^{\text{initial}}(p = ordre + 1) &= \frac{12p^4 + 51p^3}{24p^4 + 84p^3} \sim 0.5 \end{aligned} \tag{4.15}$$

4.5 Synthèse du chapitre

Ce chapitre a présenté la méthode d'éléments finis spectraux, depuis la formulation forte jusqu'à las matrices d'éléments finis, en passant par la discrétisation temporelle et spatiale. Le schéma explicite et le maillage structuré par blocs sont deux propriétés qui permettent une implémentation plus performante de la méthode. La décomposition en étapes du noyau de calcul et le calcul de l'intensité arithmétique constate que le noyau est limité par les accès mémoire.

Les optimisations doivent viser la réduction des accès mémoire, la réutilisation des variables et un meilleur remplissage des mémoires caches pour accélérer la lecture des données. Les travaux de cette thèse visant ces objectifs sont décrits dans le chapitre 5.

Chapitre 5

Optimisation du code SFEM

Sommaire

Ce chapitre décrit les transformations réalisées dans l'écriture du noyau de calcul `PRL` décrit dans la section 4.3 pour améliorer l'efficience du *multithreading*, réduire le nombre d'accès mémoire et améliorer la persistance des données dans les mémoires caches pour alimenter le mieux possible les cœurs SIMD de calcul des processeurs ainsi que les unités de calcul des GPU.

Premièrement, on trouve une synthèse des travaux publiés pour l'amélioration de la performance sur des noyaux de calcul similaires.

À la suite de la mise en place d'une maquette qui sert de version de référence, on discute de la précision numérique. Ensuite, quatre modifications de nature différente sont présentées, mais applicables sur n'importe quel algorithme, qui peuvent avoir un grand impact sur la performance : les transformations mathématiques, sur les boucles, le balayage des données et la disposition des données en mémoire. Parfois, il y aura des points spécifiques à la méthode SFEM, mais les transformations sont décrites pour être généralisables à d'autres algorithmes d'éléments finis ou de différences finies.

La dernière section 5.8 présente une synthèse des modifications et les implémentations des codes à évaluer. Notamment, la différence entre une vectorisation (utilisation des instructions SIMD par le compilateur), une SIMDisation (utilisation explicite des instructions SIMD) et l'effet de l'application de ces modifications sur les cartes graphiques sont abordées. Il en résulte deux versions particulières pour les CPU multi-cœurs SIMD (vectorisé et SIMDisé) et deux pour les GPU (utilisant les registres ou la *shared memory*).

5.1 État de l'art

La littérature pour l'amélioration de la performance est très vaste, mais les méthodes numériques utilisées en codes de calcul pour résoudre des équations sont notamment plus importantes. Bien que chaque algorithme puisse avoir une réponse différente aux travaux de cette thèse, ils sont susceptibles d'améliorer leur performance.

D'autres auteurs ont fait des travaux similaires pour des algorithmes différents, dont des études similaires qui montrent la diversité des champs d'application. Déjà en 1994 on peut trouver des travaux pour implémenter de manière parallèle les éléments finis spectraux [Ben Belgacem et Maday, 1994]. En 1998 les éléments finis spectraux sont présentés pour leur capacité à simuler de manière efficiente la propagation des ondes sismiques acoustiques [Komatitsch et Vilotte, 1998] ou élastiques [Komatitsch et al., 1999] en géophysique. L'implémentation sur cartes graphiques a dû attendre une dizaine d'années. L'étude des implémentations spécifiques (POGO [Huthwaite, 2014] et SPECFEM [Komatitsch et al., 2010, 2009]) se trouve dans la sous-sous-section 5.8.3.2.

Pour les méthodes des éléments finis, on trouve largement plus de travaux. Dans la suite, quelques-uns par ordre chronologique sont discutés. Markall et al. [2010] proposent un Unified Form Language pour générer des codes optimisés en GPU depuis des spécificités de haut niveau. Zumbusch [2012] s'intéressent aux différences finies en mul-

tiprocesseurs vectoriels. Ils étudient une SIMDisation pour différents *memory layouts*, l'utilisation d'un ou plusieurs GPU et une comparaison entre simple et double précision. Ils proposent, à la différence des travaux de cette thèse, une parallélisation au niveau temporel. Schöberl et al. [2014] présentent l'outil public NGSolve pour implémenter des éléments finis de manière plus simple et optimisée. Zhang et Shen [2013] présentent une implémentation en CUDA des éléments finis en faisant varier le maillage entre mailles triangulaire, quadrilatères, tétraédriques ou hexaédriques. Dziekonski et al. [2014] font pour leur part une comparaison entre une implémentation CPU (Sandy Bridge 32 cœurs) et un GPU (Tesla K40) pour en obtenir une accélération de ×32.

En 2014, Hoole et Sivasuthan [2014] font une révision des méthodologies des éléments finis en GPU. La même année, Huan-Ting Meng et al. [2014] proposent différents schémas pour l'utilisation des *threads* en éléments finis pour l'analyse des ondes électromagnétiques. Plus tard, Dziekonski et al. [2016] comparent encore des implémentations entre GPU et CPU, et cette fois avec une accélération de ×4.7. Banaś et al. [2016] comparent différentes stratégies d'implémentation entre CPU et GPU, mais pour des éléments finis de premier ordre au lieu d'ordre élevé.

Finalement, la thèse de Ljungkvist [2017] est très enrichissante. Il se concentre sur la phase d'assemblage de la matrice pour établir le système linéaire qui va être résolu, suivant la méthode des éléments finis. Il considère l'exemple de l'équation de Poisson pour illustrer la méthode. Il travaille à la fois avec des multiprocesseurs et des GPU et analyse, comme dans ce manuscrit, la différence d'intensité arithmétique et de bande passante pour différents ordres. Il compare les temps de calcul sur CPU entre différentes stratégies pour assembler les matrices (au lieu d'*à la volée*, voir section 4.3), l'utilisation de la stratégie de coloriage contre l'utilisation des opérations atomiques et finalement effectue une comparaison entre CPU-GPU.

Il existe de nombreux autres travaux pour d'autres méthodes. La méthode des différences finies dans le domaine temporel (FDTD, Finite Difference Time-Domain) a été rapidement implémentée sur GPU [Balevic et al., 2008]. Rossi et al. [2008] ont comparé les implémentations sérialisées et les implémentations parallèles avec OpenCL et CUDA pour la méthode Transmission-Line Method pour la simulation des champs magnétiques (avec une stratégie de coloriage pour traiter les interfaces des éléments). Kun [2009] obtiennent une accélération de ×8 en utilisant un GPU au lieu d'un CPU pour une méthode Finite Element Time-Domain (FETD). Peng et al. [2009] ont étudié la SIMDisation d'un code de simulation séismique pour des serveurs de calcul multicœur.

En 2011, Yurtesen et al. [2011] proposent une comparaison entre une SIMDisation SSE, une vectorisation AVX et une implémentation GPU pour la méthode d'Arakawa (différences finies). Toujours pour la méthode des FDTD, Livesey et al. [2012] comparent l'utilisation des extensions SSE pour les processeurs Intel et AMD. Cette même année, Dursun et al. [2012] proposent le parallélisme externe entre nœuds d'un serveur de calcul et le parallélisme interne en chaque nœud avec une SIMDisation d'extension SSE pour une méthode FDTD d'ordre élevé. La librairie CFD Builder pour la simulation de mécanique des fluides ou *computational fluid dynamics* (CFD) a été présentée par Jayaraj et al. [2014]. Ils comparent le stockage par briquettes (les éléments en *AoS*, voir sous-section 5.7.2) contre le *tiling* (la disposition géométrique, voir sous-section 5.7.1). Hadade et di Mare [2016] optimisent pour des processeurs *multi-core*

et *many-core* un code CFD. Ils se servent de l'intensité arithmétique pour décider des optimisations. Ensuite, ils testent la vectorisation, SIMDisation ou l'utilisation des *wrappers* avec des données alignées ou non alignées et les *memory layout AoS*, *SoA* et *AoSoA*.

Il y a des auteurs qui ont varié leur approche pour présenter des optimisations sur plusieurs algorithmes en même temps. Ryoo et al. [2008] proposent des optimisations en CUDA pour des algorithmes d'éléments finis, cryptographie, corrélation statistique, etc. Klöckner et al. [2012] présentent PyCUDA, un outil pour générer de code haute performance CUDA pour n'importe quelle méthode. Asahi et al. [2017] présentent la fusion des *kernels* dans le cas des accès mémoire indirects ou non contigus pour étudier l'utilisation des mémoires caches en GPU et *many-core-processors* pour un schéma des différences finies et une méthode semi-lagrangienne.

Une littérature importante traite des opérations sur les matrices (denses ou creuses) pour n'importe quelle méthode. Par exemple, Williams et al. [2007] font une analyse similaire à l'analyse faite dans les sections postérieures (alignement mémoire, SIMDisation, déroulage des boucles, utilisation des registres ou mémoire partagée en GPU, *auto-tuning*, fusion des *kernels*) dans le contexte des produits des matrices disperses-vecteurs. Bell et Garland [2009] étudient le stockage des matrices creuses sur GPU selon qu'elles sont structurées ou non. De leur côté, Reguly et Giles [2012] ont étudié l'impact des niveaux mémoire d'un GPU, en vue de la coopération et la granularité des *threads*. Masliah et al. [2016] ont concentré leurs efforts sur le produit des matrices denses de taille moins de 32×32 en *multi-core* (CPU), *many-core* (GPU) et *many-integrated-cores* (Xeon Phi).

Les travaux présentés en cette thèse ne sont pas spécifiques pour la méthode des éléments finis spectraux. Ils peuvent être appliqués à d'autres schémas des éléments finis, des schémas numériques ou des algorithmes bien différents. Évidemment, dans chaque cas il faudra appliquer les modifications et valider leur application.

5.2 Version de référence

Afin d'explorer diverses pistes pour l'optimisation du noyau de calcul `PRL` de CIVA-SFEM, a été mis en place un environnement de travail donnant plus de flexibilité que le code complet de CIVA. Une maquette numérique (complètement en double précision) a été donc créée reproduisant fidèlement les opérations décrites dans la section 4.4, mais qui ne contient pas l'ensemble des routines d'initialisation et de post-traitement du code complet (appelé *Ref*).

Afin de pouvoir avoir accès à de nombreuses architectures, le code est portable sous Linux et Windows. Les compilateurs Intel Compiler (ICC) et Microsoft Visual Studio Compiler (MSVC) permettent de le compiler. Les calculs ont été codés en *C99* plutôt qu'en *C++* afin d'éviter des structures de données avancées entraînant des gestions complexes de la mémoire ainsi que des héritages de classe inefficiente (fonctions virtuelles).

Dans la suite de ce chapitre qui décrit différentes optimisations apportées à cette implémentation, la maquette numérique est nommée version de référence. C'est à cette

version qu'on compare les accélérations obtenues grâce aux optimisations successives.

5.3 Modification de la précision numérique

Dans la perspective de bénéficier au maximum du parallélisme des instructions SIMD, il est très important de savoir si les noyaux de calcul doivent être exécutés en double précision, ou s'ils sont suffisamment stables du point de vue des erreurs d'arrondi pour être exécutés en simple précision.

Dans le cas de ces travaux, la propagation d'erreurs dans un code éléments finis, la sensibilité aux erreurs d'arrondi doit être vérifiée empiriquement. Cette section étudie la problématique de la précision de calcul pour les problèmes itératifs et montrer que dans cette application, à savoir la simulation des ondes ultrasonores par la méthode des éléments finis spectraux, il est possible de faire un calcul en simple précision qui s'avère suffisamment précis et que le recours à la double précision (et la perte de performance SIMD associée) n'est pas utile.

5.3.1 Généralités sur les erreurs numériques

Afin de mieux caractériser les erreurs numériques dont on parle, il est utile de présenter le modèle de vérification et validation (V&V). Ce modèle [Oberkampf et Trucano, 2002] permet de déterminer la confiance dans la modélisation et la simulation d'une manière critique. Dans la suite de ce paragraphe, ce modèle schématisé par la Figure 5.1 est décrit.

Le schéma fait apparaître la distinction entre la validation, qui consiste à analyser la différence entre la réalité et un modèle mathématique, et la vérification, qui consiste à vérifier la cohérence entre ce modèle mathématique conceptuel et son implémentation informatique. Entre la réalité physique et la valeur résultant de l'exécution de la machine il existe un écart de précision qui doit être validé au moyen de l'obtention de résultats expérimentaux ou d'un modèle de référence. C'est l'objet de la procédure de

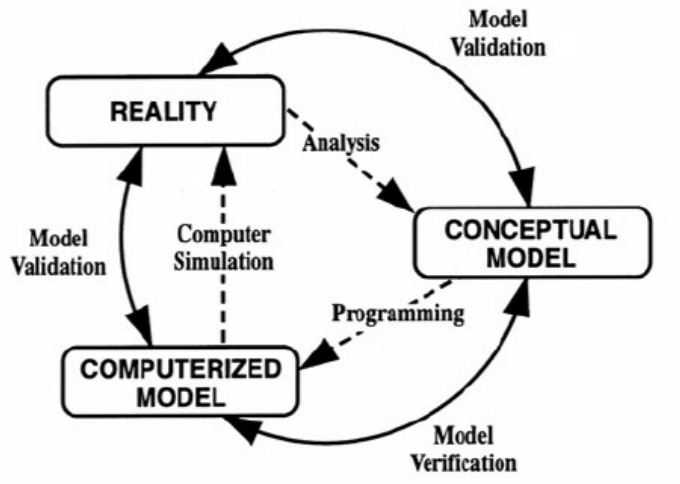


FIGURE 5.1 – Schéma du modèle de vérification et validation. (Source : Schlesinger [1979].)

validation. Dans la suite de ce paragraphe, les différentes erreurs qui doivent être prises en compte lors de l'étape de vérification sont décrites.

La première erreur qui apparaît est l'*erreur de modèle*, à savoir l'écart entre la propagation des ondes dans un milieu réel et l'équation associée (4.1). La solution de cette équation est appelée *vraie valeur* mathématique v. Puisque cette équation n'est pas résoluble explicitement, il s'avère nécessaire d'utiliser un schéma numérique d'approximation.

L'*erreur d'approximation* correspond à l'écart entre la solution mathématique exacte v et celle obtenue à partir du schéma numérique. Dans ce cas, l'erreur d'approximation est bornée par la consistance théorique de la discrétisation temporelle et de l'espace d'approximation. Cette consistance, ou précision, est d'ordre 2 en temps (sous-section 4.1.3) et d'ordre q en espace (sous-section 4.1.2).

La solution de (4.7) se nomme *valeur approximée* $v(\Delta t, h)$. Cette valeur va être obtenue par l'exécution du code associé sur une machine de calcul. Il faut donc vérifier que le code n'a pas d'*erreurs d'implémentation* (tests unitaires).

Ensuite, l'arithmétique flottante sur laquelle repose l'exécution du code va produire une *erreur d'arrondi*. Cette erreur d'arrondi s'accumule au fur et à mesure de l'avancée en temps de la simulation.

La version originale du code utilise des variables en double précision. Ces variables en virgule flottante sont encodées en 64 bits, au lieu de 32 bits pour la simple précision. En termes absolus, la double précision permet une erreur d'arrondi plus basse que la simple précision. En revanche, l'espace mémoire dédié est exactement le double. Si les problèmes dépassent, de manière générale, les centaines de Mo, voire des Go, l'utilisation de simple ou double précision devient un enjeu important. En plus de cet enjeu de capacité mémoire de la machine, le passage en simple précision donne la possibilité d'avoir accès à deux fois plus de données sur la mémoire cache. Enfin, le passage de double à simple précision d'un code permet d'utiliser des instructions SIMD avec deux fois plus d'opérations simultanées à l'intérieur d'un registre. L'écart entre les valeurs obtenues en utilisant d'une part la double précision, d'autre part la simple précision, a été nommé *erreur flottante*.

5.3.2 Évaluation de l'erreur flottante

Afin de discerner si le passage des variables de double à simple précision est possible, il faut fixer une méthode d'évaluation de l'erreur flottante et les seuils à partir desquels le passage en simple précision ne peut plus être considéré comme acceptable. Ces deux décisions sont en rapport avec le type du calcul à modifier et les contraintes d'utilisation qui sont dictées par la précision attendue sur les observables physiques d'intérêt pour l'utilisateur. D'une manière générale, il est difficile de prédire l'impact du passage en simple précision des différentes parties du calcul. Un calcul intermédiaire qui contribue faiblement au calcul principal peut se permettre une erreur plus importante. À l'inverse, un calcul récursif ou itératif risque de convertir une erreur insignifiante en une perte totale de la signification du résultat.

Parmi les mesures des erreurs les plus souvent utilisées, on trouve l'erreur absolue ϵ_a (5.1) et l'erreur relative ϵ_r (5.2), où v_d correspond à la valeur des variables en double précision et v_s en simple précision.

$$\epsilon_a = |v_d - v_s| \tag{5.1}$$

$$\epsilon_r = \frac{\epsilon_a}{|v_d|} \tag{5.2}$$

Dans ce cas, l'effet de l'utilisation des variables en simple précision plutôt que des variables en double précision doit être évalué sur l'ensemble du calcul itératif. Il faut donc suivre l'erreur dans le temps $\epsilon_r(t)$.

Ici, la variable considérée est la variation de pression au passage de l'onde acoustique. Les valeurs ne doivent pas être comparées directement, mais leur écart doit être rapporté à la surpression maximale dans le domaine de calcul. L'erreur relative brute n'est pas un indicateur pertinent, car elle sera très sensible à d'infimes variations pour des valeurs proches de 0.

On utilise l'erreur absolue la plus grande (5.3) et la valeur la plus grande dans le domaine d'approximation spatiale (5.4). Puis l'évolution de l'erreur en utilisant au moins 5,000 pas de temps est étudiée. Le seuil fixé est donc $\epsilon_r(t) < 0.01$ pour $0 < t < 5,000$ ou, dit d'une autre manière, moins de 1% d'erreur sur la valeur la plus grande de la variable. Cette valeur est largement suffisante pour l'interprétation physique des résultats.

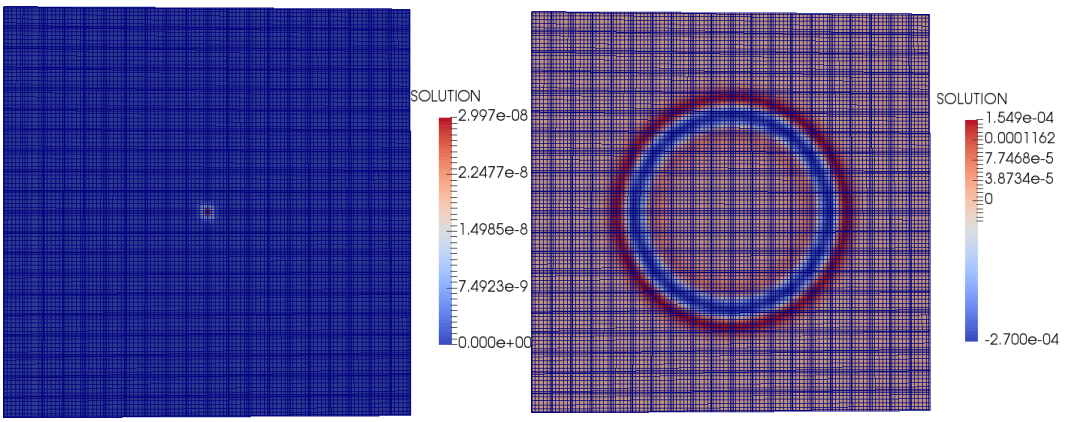
$$\epsilon_a^{max}(t) = max_t(|v_d - v_s|),\ \epsilon_r^{max}(t)\frac{\epsilon_a}{max_t(|v_d|)} \tag{5.3}$$

$$\epsilon_a^{max}(t) = max_t(|v_d - v_s|),\ \epsilon_r^{max}(t)\frac{\epsilon_a}{max_t(|v_d|)} \tag{5.4}$$

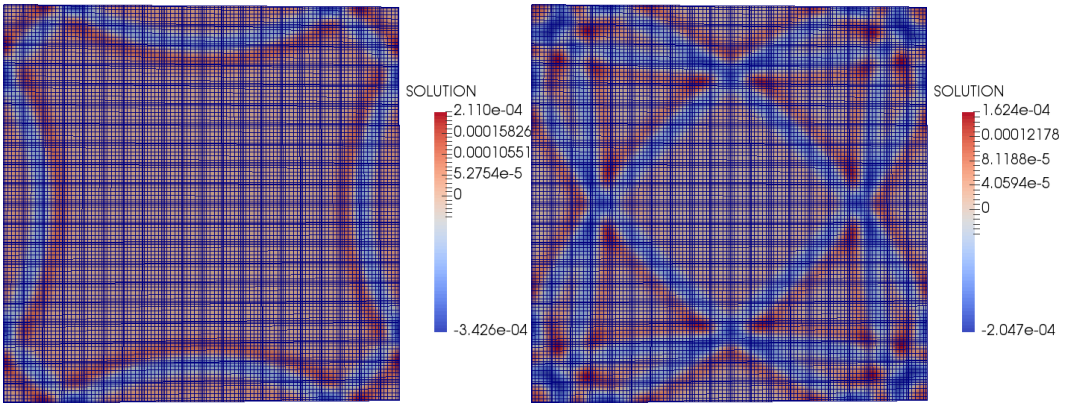
5.3.3 Analyse des résultats

Le cas étudié a pour conditions aux limites un bord libre et des conditions initiales nulles. Le point source est représenté par une gaussienne en espace et une ondelette de Ricker en temps. Le maillage est composé de 20×20 éléments 2D pour les ordres d'approximation trois, cinq et sept, correspondant à 3,721, 10,201 et 19,881 points ou variables stockées en mémoire. La Figure 5.2 présente la propagation de l'onde pour l'ordre trois.

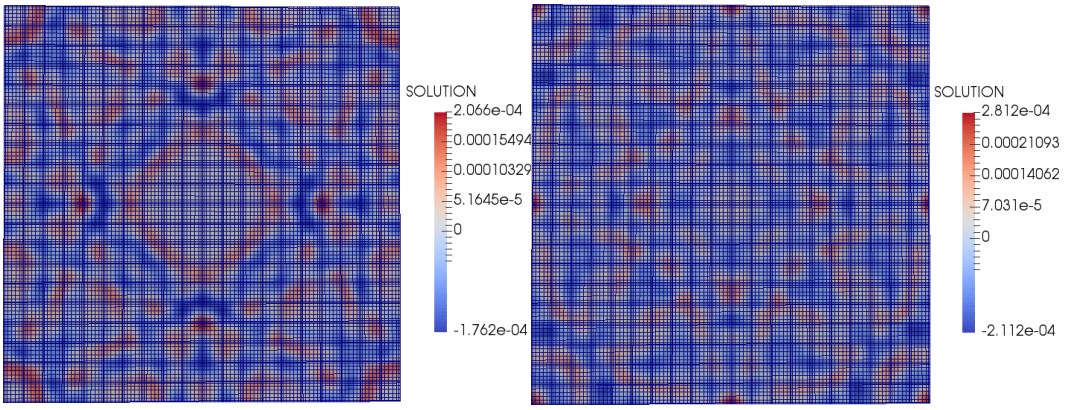
Afin de mieux comprendre les résultats, l'outil ParaView a été utilisé pour visualiser l'erreur absolue, l'erreur relative et l'erreur relative logarithmique. La Figure 5.3a montre les erreurs pour 500 pas de temps. L'erreur absolue augmente lentement (toujours en dessous de 10^{-10}). L'erreur relative varie par rapport aux différents ordres d'approximation, mais il n'y a là rien de surprenant puisque la quantité des variables à traiter et les opérations varient entre eux. L'erreur relative logarithmique montre que l'erreur reste limitée à moins de 0.01%. Il n'y a pas de différence appréciable entre les résultats obtenus pour différents ordres d'approximation. On a donc choisi de ne montrer que l'ordre d'approximation 3 dans la Figure 5.3b comprenant jusqu'à 5,000 pas de temps.



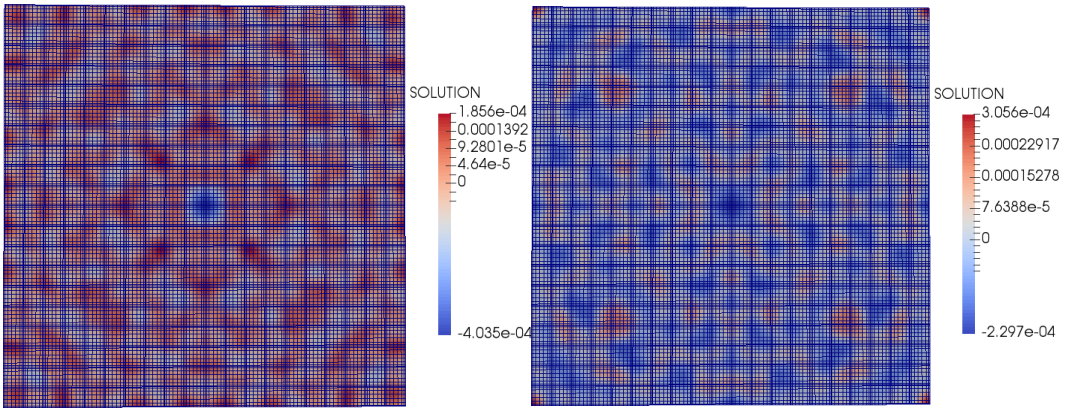
(a) $t = 0$ (b) $t = 200$



(c) $t = 400$ (d) $t = 1,000$



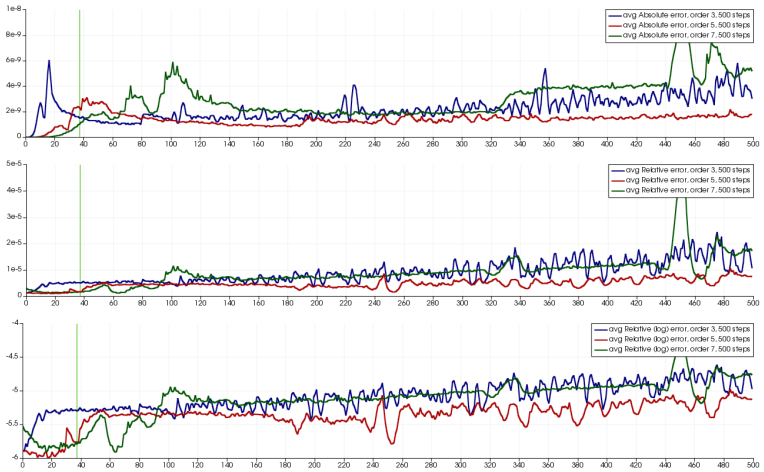
(e) $t = 2,000$ (f) $t = 5,000$



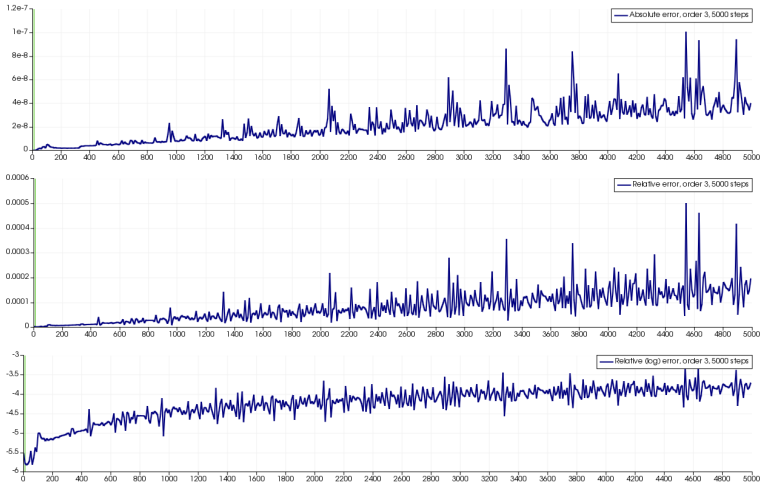
(g) $t = 10,000$ (h) $t = 50,000$

FIGURE 5.2 – Propagation de l'onde pour différents pas de temps t.

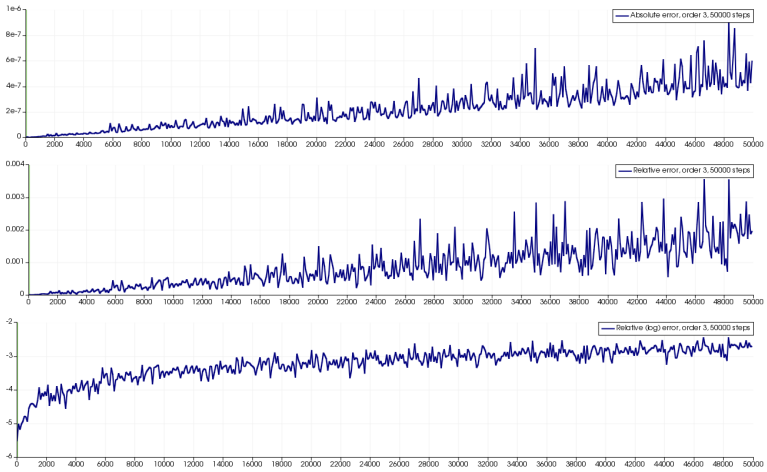
L'erreur absolue montre une croissance linéaire, et l'erreur relative reste inférieure à 0.1%, ce qui est inférieur au seuil demandé. Afin de vérifier la croissance linéaire de l'accumulation de l'erreur, on a testé jusqu'à 50,000 pas de temps (Figure 5.3c) dépassant largement la quantité de pas de temps habituel pour ce type de calcul. Effectivement, l'erreur relative représente encore moins d'un 1%, mais s'en approche suffisamment pour présumer que ce seuil va être franchi avant les 100,000 pas de temps.



(a) Ordre 3, 5 et 7. 500 pas de temps.



(b) Ordre 3 et 5,000 pas de temps.



(c) Ordre 3 et 50,000 pas de temps.

FIGURE 5.3 – Erreur absolue, erreur relative et erreur relative logarithmique.

5.4 Transformations mathématiques et indexation des matrices

Les transformations présentées dans cette section consistent à étudier l'algorithme pour comprendre toute la structure interne : relation entre variables, direction de lecture des vecteurs/tableaux, taille des données, série numérique des indices... Ensuite, des parties du code ont été réécrites pour rendre l'ensemble plus performant, notamment vis-à-vis d'une meilleure utilisation de la mémoire cache.
On rappelle ici les cinq étapes de la fonction `PRL`, ainsi présentées au chapitre 4 :

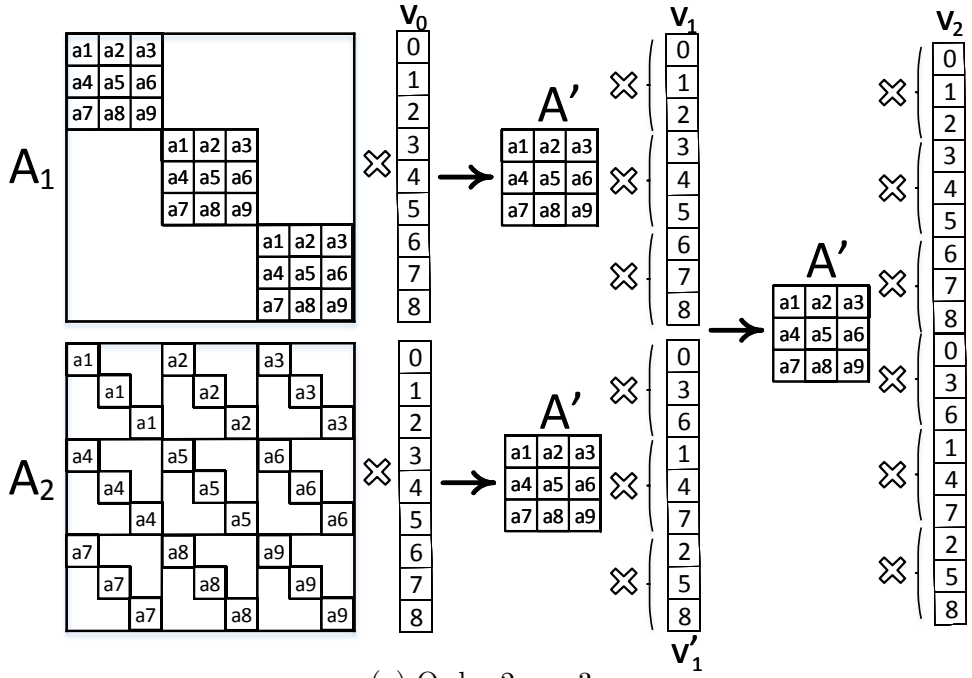
1. lecture du vecteur local U,
2. produit des matrices A par U,
3. produit de la matrice D par la combinaison du résultat de l'étape précédente,
4. produit des matrices A^T par le résultat de l'étape précédente et accumulation,
5. écriture des résultats en V.

Comme cela a été décrit dans la sous-section 4.3.2, la matrice de gradient A' est une petite matrice dense qui contient l'ensemble des coefficients nécessaires pour le calcul de gradient. La méthode SFEM permet alors de travailler avec une matrice dense au lieu d'une matrice creuse. Comme l'illustre la Figure 5.4, l'implémentation du produit matrice-vecteur réduit les deux produits matrices-vecteurs de taille p^2 en dp produits matrices-vecteurs de taille p.

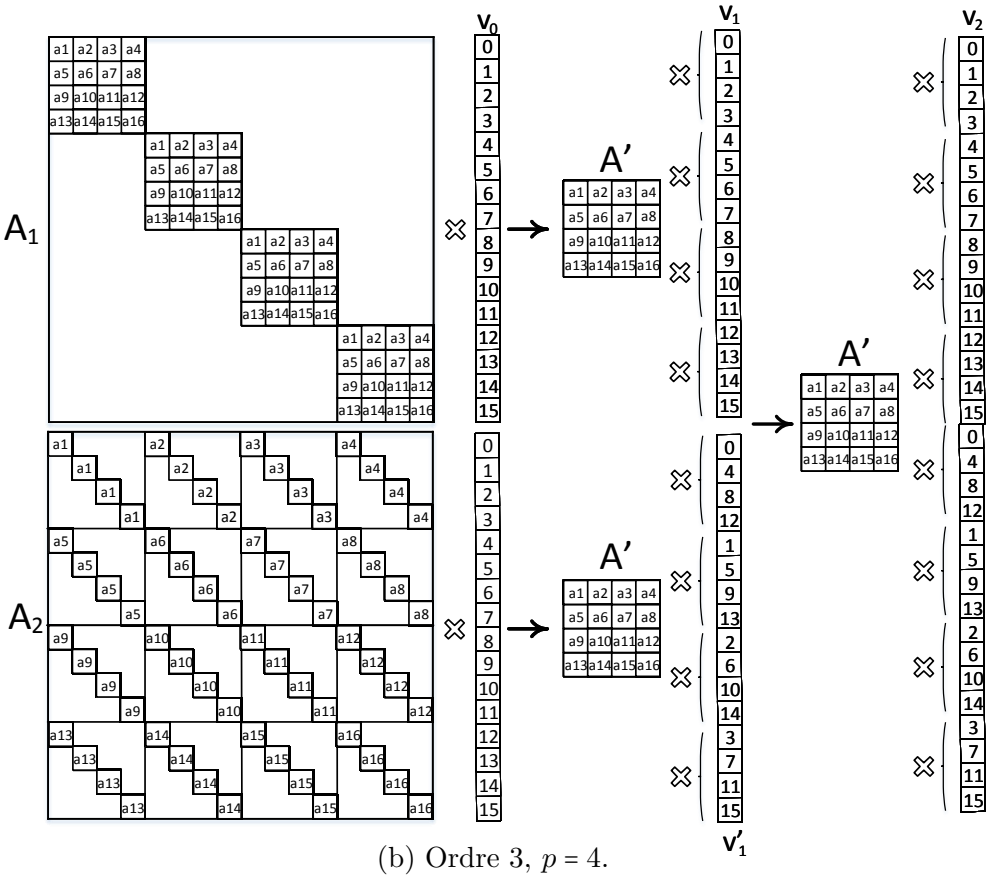
Le fait de ne stocker qu'une fois chaque coefficient non nul plutôt que de stocker la matrice complète a été présenté en sous-section 4.3.2. Il ne s'agit pas d'une nouveauté, car ce système de stockage était déjà présent dans la maquette de référence. La nouveauté réside dans le stockage d'un second vecteur V_1' local réorganisant les données (voir Figure 5.4). Comme on peut le constater, les indices de 0 à 8 dans le second vecteur ont un ordre différent. Cela double l'espace requis pour le stockage, mais le coût est négligeable compte tenu de la faible empreinte mémoire de ce vecteur (mémoire cache L1). Ce deuxième vecteur V_1' ajoute une charge de travail à l'étape 1, puisqu'il nécessite des chargements supplémentaires. Cette charge supplémentaire est plus que compensée par le gain dans les étapes 2 et 4, puisqu'elle permet une lecture concurrente en mémoire. Par simplicité, ces deux vecteurs ont été fusionnés dans un vecteur $v2$ (voir la Figure 5.5).

Le code est modifié par rapport à la version d'origine. Auparavant, il y avait trois boucles `for loops` entraînant des calculs d'indices et un balayage non contigu du vecteur pendant le deuxième produit (Algorithme 5.1). Dans le cas 3D, un troisième produit $A_3 \times v_0$ était également non contigu et avec des pas de mémoire encore plus grands.

Une fois optimisé, l'Algorithme 5.2 ne contient qu'une boucle `for loop` exécutant des petits produits matrice-vecteur $A' \times v_2$. Ces petits produits ne changent que selon l'ordre de l'élément. L'implémentation est donc faite avec un `switch case` basé sur l'ordre composé des deux boucles de taille fixe. Grâce aux tailles fixes des boucles, le compilateur peut, lui-même, dérouler les boucles ou faire les optimisations qu'il estime nécessaires (vectoriser le code). La série des indices est linéaire et le balayage de la mémoire est contigu.



(a) Ordre 2, $p = 3$.



(b) Ordre 3, $p = 4$.

FIGURE 5.4 – Schéma de l'étape 2 en 2D. La réduction des deux grandes matrices A_1 et A_2 creuses structurées vers une matrice réduite dense A' provoque la duplication du vecteur v_0 en v_1 et v'_1 pour éviter des accès non contigus en mémoire. Finalement, ces deux instances sont fusionnées en v_2.

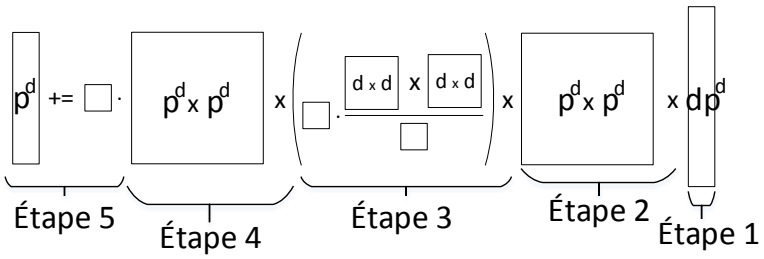


FIGURE 5.5 – Les étapes de l'équation avec une petite matrice dense visualisées sous forme de matrices, de vecteurs et de produits scalaires. Les tailles des objets sont spécifiées, à l'exception des scalaires, en utilisant $p = ordre + 1$ et dimension d. Voir la Figure 4.10.

Algorithme 5.1: Version originale de l'étape 2 : un nid de boucles par gradient.

```
p ← ordre + 1;
// Gradient axe X
for k ← 0 to p do
    for i ← 0 to p do
        for j ← 0 to p do
            GradientX[k × p + i] += A′[i][j] × v0[k × p + j];
// Gradient axe Y
for k ← 0 to p do
    for i ← 0 to p do
        for j ← 0 to p do
            GradientY[i × p + k] += A′[i][j] × v0[j × p + k];
```

Algorithme 5.2: Version modifiée de l'étape 2 : fusion de la boucle externe + spécialisation des boucles internes.

```
p ← ordre + 1;
// Gradient de deux axes combinés
switch p do
    case 3 do
        // dimension × p = 6
        for k ← 0 to 6 do
            for i ← 0 to 3 do
                for j ← 0 to 3 do
                    GradientX[k × 3 + i] += A′[i][j] × v2[k × 3 + j];
    case 4 do
        // dimension × p = 8
        for k ← 0 to 8 do
            for i ← 0 to 4 do
                for j ← 0 to 4 do
                    GradientX[k × 4 + i] += A′[i][j] × v2[k × 4 + j];
    case ... do
        ...
```

L'étape 4 est un miroir de l'étape 2 : il s'agit de la même opération où la matrice A' est transposée.

5.5 Transformation des boucles

Les transformations de cette section modifient beaucoup le code. Il y a deux objectifs : permettre aux processeurs d'appliquer toutes les optimisations de façon optimale et profiter des accès de mémoire plus rapides (mémoires caches et registres).

5.5.1 Loop unwinding

Le *loop unwinding* est un cas particulier du *loop unrolling*, où la boucle est complètement déroulée. Le déroulage plus habituel, le *loop unrolling*, consiste à dupliquer le corps de boucle d'un certain ordre, tandis que dans le cas du *loop unwinding* la boucle va entièrement disparaître du code. Cette technique est restreinte à des boucles dont l'intervalle d'itération est petit et connu au moment de la compilation. Elle présente plusieurs avantages : elle évite les branchements, permet de conserver tous les résultats temporaires dans des registres (scalarisation) et aide les processeurs *out-of-order* à schéduler de manière efficiente les instructions [Lacassagne et al., 2014].

5.5.2 Scalarisation

Conserver les résultats temporaires dans des registres et une transformation très importante, car l'algorithme est limité par les accès mémoire. La pression des registres est donc plus élevée et le compilateur peut générer un *spill code* pour stocker temporairement les variables dans la mémoire. Cependant, même si cela se produit, les registres écrits en mémoire seront stockés dans la mémoire cache L1. Il ne sera donc pas moins efficient que de stocker une petite partie des données dans un tableau local qui sera réutilisé dans un futur proche et lui aussi stocké dans la mémoire cache L1 (ou L2).

5.5.3 Fusion des opérations

Comme il y a plusieurs produits de matrice-vecteur et matrice-matrice, il est possible d'aller plus loin et de fusionner les opérations de toutes les étapes. Une telle fusion est possible, car tout est connu à la compilation.

5.5.4 Factorisation

Tous les calculs sont fusionnés et séparés des instructions de chargement et de stockage. Il est facile de trouver et d'éliminer les sous-expressions communes et de réaliser une factorisation du code.

5.5.5 Restructuration des calculs et des accès mémoire dans le code

Le code est divisé en trois parties pour simplifier la démarche. Ces parties correspondent avec l'idée d'une fonction : entrée des arguments, exécution et sortie des résultats. Dans ce cas, les trois parties sont :

1. Chargements de la mémoire - pendant cette première partie, toutes les données nécessaires pour effectuer les calculs dans des variables scalaires sont lues. Il n'y a qu'une seule lecture par donnée. Le compilateur décidera quel scalaire sera chargé dans les registres ou dans la mémoire cache (*spill code*). Cela correspond parfaitement à l'étape 1 du noyau de calcul `PRL`.
2. Calculs - dans la deuxième partie, les calculs sont effectués en utilisant les variables scalaires fixées auparavant. Chaque ligne du code enregistre son résultat dans une nouvelle variable scalaire. Le compilateur décidera dans quel ordre il effectuera le calcul et il comprendra facilement quelles variables sont réutilisées et lesquelles peuvent être des variables *inlined*, c'est-à-dire non déclarées, mais remplacées par leur valeur. Les étapes 2, 3 et 4 sont fusionnées dans cette partie.
3. Écriture de mémoire - enfin, les résultats des variables scalaires sont accumulés et stockés dans le vecteur de sortie. Le compilateur décidera dans quel ordre il écrira ces résultats dans la mémoire globale. Cela correspond à l'étape 5 du noyau de calcul `PRL`.

Ces transformations sont illustrées à travers d'un exemple plus simple que le code réel : l'Algorithme 5.3 décrit deux multiplications successives de type matrice-vecteur-scalaire qui sont fusionnées en un seul ensemble de calcul (Algorithme 5.4). L'intensité arithmétique (si $N = 1$) qui valait initialement (16 MUL + 8 ADD) / (24 LOAD + 8 STORE) = 0,75 passe à (12 MUL + 8 ADD) / (8 LOAD + 4 STORE) = 1,67.

Algorithme 5.3: Double produit matrice-vecteur (et un scalaire) avec boucles. IA = 0.75.

```
// Premier produit: A += M × B/k
for j ← 0 to N do
    for i ← 0 to N do
        A[j] += M[j][i]B[i]/k;
// Deuxième produit: B += M^T × A · k
for j ← 0 to N do
    for i ← 0 to N do
        B[j] += M[i][j]A[i]k;
```

Algorithme 5.4: Double produit matrice-vecteur avec déroulage total des boucles, scalarisation, fusion des opérations et factorisation. Pour $N = 1$, IA = 1.67.

```
// MEM:LOAD
A_0 ← A[0], A_1 ← A[1];
B_0 ← B[0], B_1 ← B[1];
M_00 ← M[0][0], M_01 ← M[0][1];
M_10 ← M[1][0], M_11 ← M[1][1];
// CALC
A_0 ← A_0 + (M_00 B_0 + M_01 B_1)/k;
A_1 ← A_1 + (M_10 B_0 + M_11 B_1)/k;
B_0 ← B_0 + (M_00 A_0 + M_10 A_1) * k;
B_1 ← B_1 + (M_01 A_0 + M_11 A_1) * k;
// MEM:STORE
A[0] ← A_0, A[1] ← A_1;
B[0] ← B_0, B[1] ← B_1;
```

5.5.6 Noyau opérationnel minimum

En raison de la structure du calcul - une combinaison de multiplications matricielles et scalaires - plus d'optimisations peuvent être faites, notamment la factorisation de sous-expressions communes ce qui réduit encore plus le nombre des accès mémoire. La Figure 5.6 illustre la façon dont l'étape 2 est complètement factorisée en une seule matrice, au contraire de la situation de la Figure 5.5. Cela réduit le nombre de calculs de $17p^2$ en 2D et de $24p^3$ en 3D.

La fusion et la factorisation suppriment les séquences multiples d'écriture et de lecture des données qui se produisent entre deux étapes consécutives, conduisant à

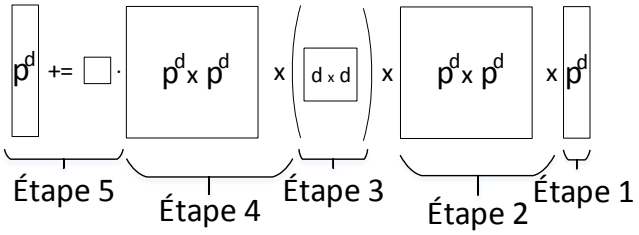


FIGURE 5.6 – Les étapes de l'équation factorisée visualisées sous forme de matrices, de vecteurs et de produits scalaires. Les tailles des objets sont spécifiées, à l'exception des scalaires, en utilisant $p = ordre + 1$ et dimension d. Voir la Figure 5.5.

Étape	MUL+ADD	LOAD+STORE	IA
1	0	9+0	-
2	72+36	0	-
3	36+18	0	-
4	54+45	0	-
5	9+9	9+9	-
Total	279	27	10.3

TABLEAU 5.1 – IA_{2D} après modifications pour chaque étape si ordre = 2.

une réduction considérable du nombre d'accès à la mémoire. Cela se traduit par de nouvelles expressions pour les intensités arithmétiques (5.5). Dans l'expression de l'intensité arithmétique, on peut noter qu'au numérateur, les coefficients de degré 3 pour le cas 2D et de degré 4 pour le cas 3D ont disparu par rapport à (4.15). Il en résulte des IA augmentant avec p contrairement aux IA initiales qui étaient presque constantes.

$$\begin{aligned} AI_{2D}^{\text{transf}}(p = ordre + 1) &= \frac{8p^3 + 7p^2}{3p^2} = 2.67p + 2.33 \\ AI_{3D}^{\text{transf}}(p = ordre + 1) &= \frac{12p^4 + 16p^3}{3p^3} = 4p + 5.33 \end{aligned} \tag{5.5}$$

Quel que soit l'ordre, l'intensité arithmétique est largement supérieure à 1. On peut espérer obtenir un code qui soit *compute bound* tant que les données tiennent et persistent dans les mémoires caches. Le Tableau 5.2 fournit ces valeurs pour les petits ordres et le Tableau 5.1 détaille la complexité des cinq étapes séparément pour l'ordre 2 en 2D, à comparer avec le Tableau 4.2.

Ordre	2	3	4	5	6	7	8
IA_{2D}	10.3	13.0	15.7	18.3	21.0	23.7	26.3
IA_{3D}	17.3	21.3	25.3	29.3	33.3	37.3	41.3

TABLEAU 5.2 – IA après modifications pour différents ordres.

5.5.7 Génération du code pour les différents noyaux

Comme dit précédemment, on a travaillé avec un cas particulier de l'équation d'onde (voir chapitre 4). Pour l'intégration de ce travail d'optimisation dans le code CIVA-SFEM, il faut appliquer le travail effectué à un plus large éventail de cas. De plus, utiliser les transformations décrites dans cette section nécessite d'avoir un fichier source pour chaque ordre d'approximation, dont la taille peut être considérable (entre 100 et 2,000 lignes de code par valeur de p). L'écriture de ces fichiers à la main peut devenir extrêmement fastidieuse.

Pour la maintenabilité de l'ensemble, il est donc nécessaire de passer par un outil de génération du code. Un tel générateur a été développé en $C++$. Cet outil génère les codes correspondants aux domaines 2D et 3D, pour l'équation des ondes acoustiques et pour les différents ordres étudiés. Ce générateur de code va être utilisé comme base pour l'intégration des travaux dans CIVA. La section 8.3 donne plus de détails sur l'intégration et le générateur de code.

5.6 Stratégie de parcours des données pour le multithreading

Dans cette section, sont décrites les différentes stratégies étudiées pour optimiser le parallélisme de tâches. Chaque tâche est chargée d'exécuter le calcul d'un élément du maillage. Les tâches doivent se combiner en évitant les problèmes d'accès simultanés sur les interfaces communes du maillage (la raison des points partagés entre les mailles est traitée en sous-section 4.1.4). Deux stratégies pertinentes sont analysées : la coloration et la duplication des frontières, ainsi que certaines stratégies dérivées.

5.6.1 Stratégie de coloration

Cette stratégie consiste à attribuer des couleurs différentes aux éléments en s'assurant que chaque paire d'éléments ayant des points ou variables communs n'a pas la même couleur pour éviter que les points des interfaces appartenant à deux éléments soient mis à jour simultanément (problème de concurrence). Tous les éléments d'une couleur peuvent alors être traités en parallèle sans problème de concurrence. Dans le cas d'un maillage quadrangulaire/hexaédrique, il y a quatre couleurs en 2D et huit couleurs en 3D. La Figure 5.7 représente la façon dont la stratégie de coloration, nommée *Color*, transforme un maillage 2D en un maillage avec quatre ensembles de couleurs différentes. Cette stratégie est communément utilisée dans le cadre des méthodes des éléments finis, notamment pour la combinaison hybride MPI et OpenMP [Crivellini et Franciolini, 2018].

Bien que cette stratégie soit simple à paralléliser (une boucle `omp parallel for` est suffisante), elle présente plusieurs inconvénients. Tout d'abord, toutes les données chargées en mémoire cache ne sont pas utiles à cause des chargements avec sauts, ce qui entraîne une utilisation inefficiente de la mémoire cache. À titre d'exemple, dans la Figure 5.7, lors du traitement de la couleur rouge, on charge en mémoire cache l'ensemble des données de la couleur verte. Deuxièmement, l'ensemble des données des interfaces

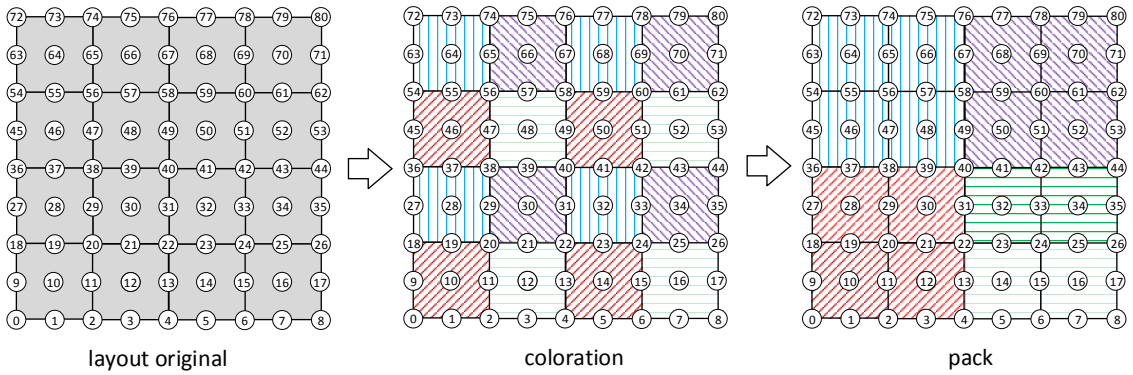


FIGURE 5.7 – La stratégie de coloration expose 4 ensembles d'éléments 2D sans interface commune entre les éléments du même ensemble. La variante pack regroupe 2×2 éléments contigus.

doit être chargé en mémoire cache autant de fois qu'il y a de couleurs communes (jusqu'à 4 fois en 2D et 8 fois en 3D). Troisièmement, en fonction de l'alignement des données et de la stratégie de répartition des tâches sur les *threads*, un partage d'une même ligne de mémoire cache (*false sharing*) entre les *threads* peut se produire.

Une variante de la stratégie de coloration est la stratégie nommée *Pack*. Cela consiste à regrouper des éléments contigus (typiquement 2×2, mais la taille pourrait être plus grande) pour réduire la quantité de *false sharing*. Chaque tâche est chargée d'un pack, les éléments d'un pack étant traités séquentiellement par le même *thread*, il n'y a donc pas de problème de concurrence. Par rapport à la stratégie de coloration, cette stratégie permet notamment de réduire la part relative des interfaces dans le calcul total.

5.6.2 Stratégie de duplication des frontières

Cette stratégie consiste à séparer les éléments en dupliquant les points de l'interface. Deux tâches peuvent alors traiter deux éléments contigus simultanément sans problème de concurrence, parce que chacune a une copie de l'interface commune.

Dans une géométrie 2D, il existe plusieurs manières de dupliquer les frontières (Figure 5.8) : selon la direction verticale, nommée *V-dup* ; sur la direction horizontale, nommée *H-dup* ; et les deux combinés, nommé *HV-dup*. En 3D, il y a plus de combinaisons possibles parmi les trois duplications possibles : vertical, horizontal ou en profondeur (*depth* en anglais). La version avec les trois duplications s'appelle *HVD-dup*.

Une fois tous les éléments traités, un post-traitement est inclus pour additionner la contribution de chaque point dupliqué sur la variable commune et propager son apport à chaque copie.

En 2D, lorsqu'on travaille avec la duplication horizontale *H-dup*, une tâche rassemble le traitement d'une ligne horizontale d'éléments. Pour les éléments appartenant à une même ligne, il n'est pas nécessaire de dupliquer les points des interfaces verticales entre eux, puisque le traitement est fait par un même *thread* de manière séquentielle.

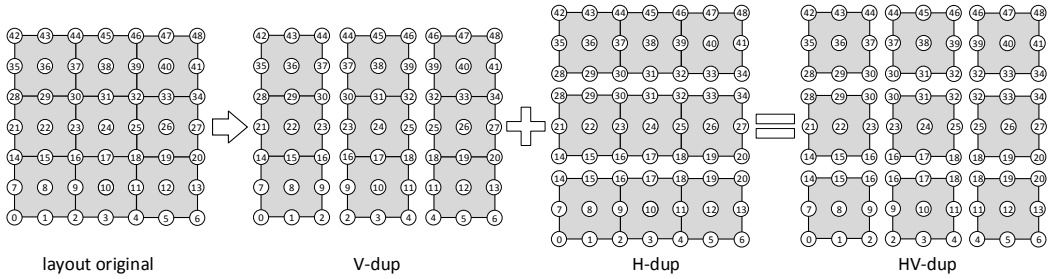


FIGURE 5.8 – Duplication des frontières : *V-dup* duplique les interfaces verticales, *H-dup* duplique les interfaces horizontales et *HV-dup* duplique les deux.

La problématique des accès mémoire concurrents ne se pose pas. Le post-traitement est simple (Algorithme 5.5) : il faut parcourir tous les points des frontières entre éléments verticaux (la ligne frontière et la ligne inférieure) et additionner l'apport. En *C*, parcourir les données d'un tableau dans le sens des lignes est performante d'un point de vue des mémoires caches. L'algorithme a une IA égale à (0 MUL + 1 ADD) / (2 LOAD + 2 STORE) = 0.25.

Algorithme 5.5: Post-traitement pour *H-dup*. IA = 0.25.

```
// Additionner l'apport en horizontal
e p ← ordre + 1;
for j ← 1 to ElementsAxeY do
    for i ← 0 to PointsAxeX do
        const apport = Output[j × p − 1][i] + Output[j × p][i];
        Output[j × p − 1][i] = apport;
        Output[j × p][i] = apport;
```

Si on travaille avec la duplication verticale *V-dup*, la performance est dégradée. Une tâche rassemble une colonne verticale d'éléments, donc les interfaces dupliquées sont verticales. Le post-traitement doit parcourir les points des frontières entre éléments horizontaux, par exemple, comme on le voit sur l'Algorithme 5.6, une colonne de données et la colonne à sa gauche. Parcourir un tableau dans le sens des colonnes engendre une forte inefficiente dans l'utilisation des mémoires cache. L'intensité algorithmique reste de 0.25.

Algorithme 5.6: Post-traitement pour *V-dup*. IA = 0.25.

```
// Additionner l'apport en verticale
p ← ordre + 1;
for j ← 0 to PointsAxeY do
    for i ← 1 to ElementsAxeX do
        const apport = Output[j][i × p − 1] + Output[j][i × p + 1];
        Output[j][i × p − 1] = apport;
        Output[j][i × p + 1] = apport;
```

Ordre	2	3	4	o
2D géométrie				
Color	4	9	16	o^2
dup sur 1 dimension	6	12	20	$o(o+1)$
dup sur 2 dimensions	9	16	25	$(o+1)^2$
3D géométrie				
Color	8	27	64	o^3
dup sur 1 dimension	12	36	80	$o^2(o+1)$
dup sur 2 dimensions	18	48	100	$o(o+1)^2$
dup sur 3 dimensions	27	64	125	$(o+1)^3$

TABLEAU 5.3 – Quantité des points stockés par élément de géométrie 2D et 3D et différents ordres pour chaque variation de la stratégie de duplication des frontières. La dernière colonne contient l'équation de complexité pour un ordre o.

Pour celle raison, *V-dup* n'est plus considérée comme une stratégie valable. En revanche, elle aide à comprendre les aspects négatifs de la stratégie de duplication double des frontières *HV-dup*. L'algorithme associé à son post-traitement est une combinaison simple des algorithmes présentés pour *H-dup* et *V-dup*, dont la partie verticale est contre-performante. En revanche, cette stratégie a l'intérêt de dupliquer toutes les frontières de manière que chaque élément soit indépendant des autres, ce qui permet de maximiser le nombre des tâches parallèles. C'est une stratégie fondamentale pour la modification du *memory layout* (voir section 5.7).

En 3D, la duplication horizontale, en profondeur ou largeur convient pour le parallélisme des tâches si les éléments dans la direction non dupliquée sont chargés dans la même tâche. Les conclusions sont les mêmes qu'en 2D : le post-traitement de *H-dup* est plus performant que n'importe quelle autre combinaison et l'intérêt de *HVD-dup* est d'obtenir des éléments indépendants.

La duplication des frontières présente des avantages significatifs par rapport à la stratégie de coloration : l'ensemble des éléments n'est scanné qu'une seule fois et il n'y a plus d'accès avec sauts (en *H-dup*). Toutes les données dans les mémoires caches sont utiles. Le seul inconvénient est la quantité de points dupliqués. Le Tableau 5.3 donne les formules de surcoût des duplications, ainsi que les valeurs numériques pour les premiers ordres. Les points dupliqués sont les points qui doivent être parcourus par le post-traitement.

Afin de réduire la quantité de calcul dans l'étape de post-traitement, une spécialisation de la stratégie *H-dup* a été développée pour le parallélisme des tâches : la stratégie *T-dup* (*thread duplication*). Elle consiste à aller au bout de l'idée selon laquelle chaque tâche se charge d'une ligne d'éléments. Dans ce cas, chaque tâche va se charger de la plus grande quantité de lignes d'éléments possible, de manière que tout le maillage soit divisé en blocs de lignes (Figure 5.9).

Par exemple, si le code est exécuté dans une machine capable de soutenir 12 *threads* en même temps, 12 blocs de lignes vont être créés, avec seulement 11 interfaces dupliquées. Chaque *thread* va exécuter une seule tâche chargée d'un bloc d'éléments. De cette manière, les points dupliqués sont réduits au minimum : leur poids en mémoire

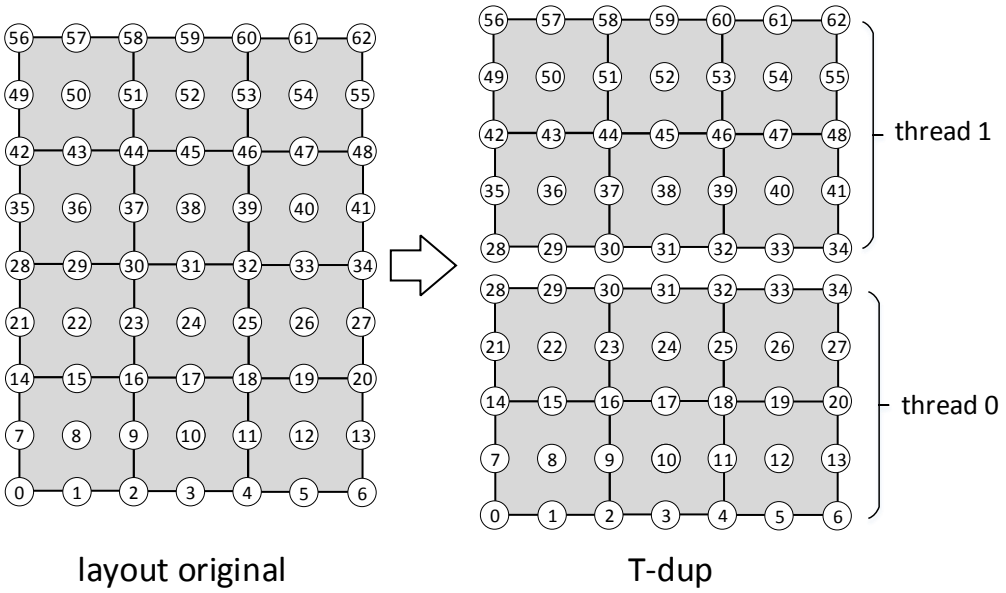


FIGURE 5.9 – Duplication des frontières *T-dup*.

est négligeable et le post-traitement est aussi réduit. De plus, vu que chaque *thread* travaille avec un ensemble de lignes, les données sont complètement coalescentes pour chaque *thread*, ce qui est idéal pour les mémoires caches.

5.7 Modification du memory layout

Cette section présente les différents *memory layout* étudiés pour améliorer la persistance des données dans les mémoires caches et alimenter correctement les cœurs de calcul. Il faut maximiser le parallélisme des données. Plusieurs stratégies sont possibles, et cinq sont présentées : Array of Struct en une et en deux dimensions, Struct of Array, la combinaison Array of Struct of Array et, finalement, la courbe du parcours en Z.

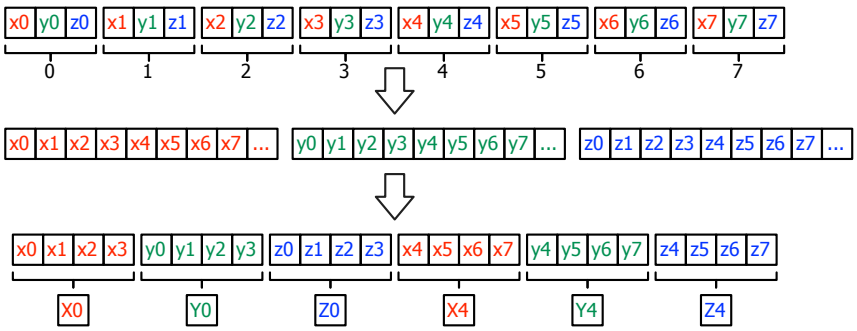


FIGURE 5.10 – Stockages en mémoire : *AoS* (haut), *SoA* (milieu) et *AoSoA* (bas) où `cardinal(SIMD)` = 4. Les lettres minuscules représentent les registres scalaires et les lettres majuscules représentent les registres SIMD.

5.7.1 Disposition géométrique initiale

La disposition en mémoire initiale, la *disposition géométrique* prend soin de maintenir la géométrie du maillage : les points proches géométriquement sont aussi proches en mémoire. Cette disposition peut être interprétée comme la disposition *AoS* en deux ou trois dimensions. Au lieu de stocker des objets complets, il faut stocker des lignes d'objets.

5.7.2 Array of Struct

Le *memory layout AoS* est la manière naturelle de stocker des tableaux des structures en *C*. Cela consiste à aligner des objets complets les uns après les autres. Le code pour accéder au membre `x` de l'élément i^{th} d'un tableau `A` est `A[i].x`. Ce *memory layout* utilise un seul pointeur actif et réduit l'éviction systématique de la mémoire cache qui apparaît lorsque plusieurs pointeurs partagent les mêmes derniers bits et que l'associativité de la mémoire cache n'est pas suffisante. Mais ce *memory layout* est difficile à vectoriser, car les membres `x` des différents membres du tableau ne sont pas contigus en mémoire.

5.7.3 Struct of Array

Le *memory layout SoA* est beaucoup plus propice à la vectorisation. L'idée est d'avoir un tableau par membre et de les regrouper dans une structure. L'accès s'écrit `A.x[i]`. Ce *memory layout* est par défaut dans *Fortran 77*. Cela aide la vectorisation du code. Mais cette écriture utilise autant de pointeurs actifs que le nombre de membres des objets et peut augmenter le nombre d'évictions systématiques de la mémoire cache lorsque le nombre de pointeurs actifs est supérieur à l'associativité de la mémoire cache. Cela devient critique en *multithreading* : la mémoire cache L3 (ou la mémoire cache du dernier niveau) a généralement une associativité inférieure au produit du nombre de *threads* par le nombre d'objets de la structure.

5.7.4 Array of Struct of Array

Le *memory layout* en mémoire *AoSoA*, également connu sous le nom *Hybrid* SoA, essaye de combiner les avantages de *AoS* et *SoA* pour le SIMD. L'idée fondamentale est d'avoir un *memory layout SoA* de taille fixe et de regrouper ces structures dans un tableau. Ainsi, cet arrangement donne la même opportunité de vectorisation qu'avec *SoA*, mais minimise le nombre de pointeurs actifs comme dans *AoS*. Le membre `x` est accessible de la manière suivante : `A[i / soa_size].x[i % soa_size]`. Une valeur typique pour la taille de *SoA* est le cardinal du registre SIMD (ou un petit multiple de celui-ci). Ce schéma d'accès peut être simplifié lors d'une itération sur de tels objets. La boucle sur les éléments est divisée en deux boucles imbriquées : une itération sur la partie *AoS* et une itération sur la partie *SoA*. Un tel code est plus difficile à écrire, notamment parce qu'il nécessite un traitement spécifique des limites des boucles.

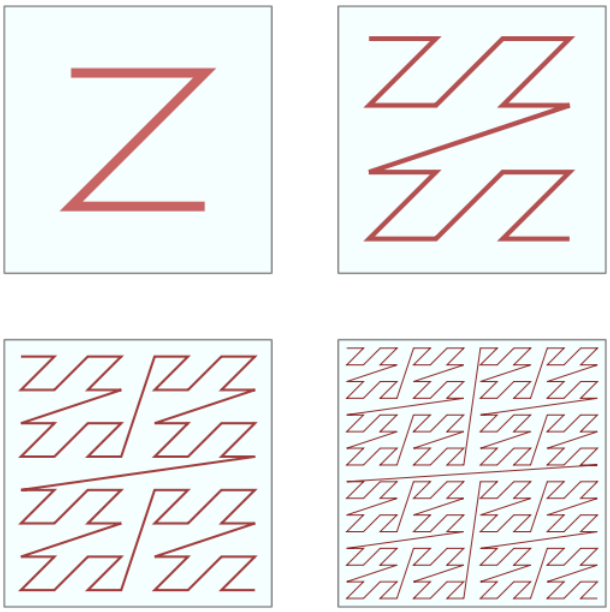


FIGURE 5.11 – Quatre itérations de la courbe d'ordre Z en géométrie 2D. (Source : Wikipédia.)

5.7.5 Courbe au parcours en Z

Le parcours en Z (*Z-order* aussi appelée ordre de Morton ou courbe de Lebesgue), permet mapper des données multidimensionnelles vers une dimension préservant la localité des données. Cette disposition est utilisée souvent pour le stockage des données des arbres binaires de recherche. Il est similaire à la disposition *SoA*, car chaque membre est dans un tableau. Le stockage suit le parcours Z, comme on peut voir dans la Figure 5.11. La valeur Z, de la nouvelle position de chaque donnée dans le tableau unidimensionnel, est le résultat de l'entrelacement des représentations binaires des coordonnées de cette donnée. La Figure 5.12 montre l'obtention des valeurs Z par l'entrelacement des coordonnées binaires. Ce *memory layout* n'est valide que pour les membres qui sont des carrés ou cubes et dont la dimension est une puissance de deux. *Z-order* est connu par préserver la localité des données en géométries 2D et 3D et par sa facilité à trouver la nouvelle position d'une donnée spécifique.

5.7.5.1 Application au SFEM

On montre un exemple de l'amélioration de localité du parcours en Z via un élément fini tridimensionnel d'ordre 3 composé par 64 points ou variables. Chaque point correspond à une variable stockée en simple précision dans un vecteur. Une mémoire cache charge dans une ligne (typiquement 64 octets) 16 valeurs de simple précision.

Le Tableau 5.4 présente comment ses points sont stockés par lignes de mémoire cache en utilisant un simple ordre incrémental des indices et le Tableau 5.5 stocke ce même élément en utilisant le parcours Z.

Le post-algorithme chargera trois interfaces de cet élément pour additionner l'apport avec les interfaces correspondantes des autres éléments. Le Tableau 5.6 montre quels points doivent être chargés pour chaque interface et la quantité des lectures d'une ligne

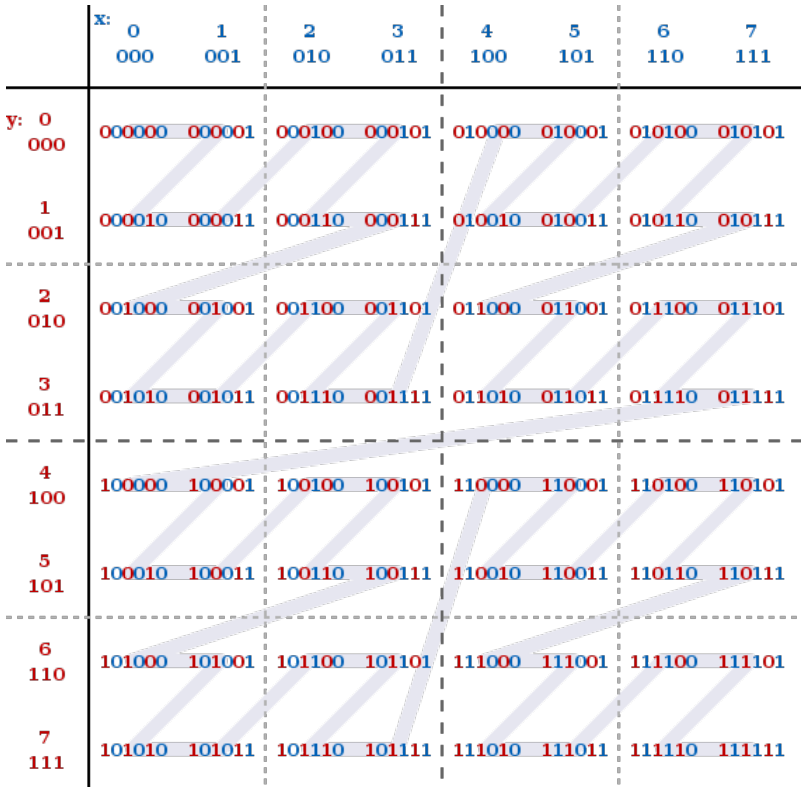


FIGURE 5.12 – Valeurs Z pour la géométrie 2D de taille 8×8. (Source : Wikipédia.)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

TABLEAU 5.4 – Élément tridimensionnel composé de 64 points dans un tableau ordonné par positionnement géométrique.

de mémoire cache nécessaires selon les deux dispositions en mémoire. L'ordre incrémental nécessite 9 charges et *Z-order* 8. Cela semble une très petite différence, mais lorsque ce post-algorithme sera exécuté une fois par élément et par itération temporelle dans des cas comportant des millions d'éléments et des centaines d'itérations, il peut s'avérer fondamental.

5.7.6 GPU

Le *memory layout* pour l'utilisation efficiente d'un GPU est *AoSoA*, où la taille de *SoA* est un multiple de la taille d'un *warp* (typiquement 32 ou 64 éléments). Un *warp*, étant le module d'exécution le plus petit d'une carte graphique, se comporte de

0	1	4	5	16	17	20	21	2	3	6	7	18	19	20	23
8	9	12	13	24	25	28	29	10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53	34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61	42	43	46	47	58	59	62	63

TABLEAU 5.5 – Stockage en Z d'un élément tridimensionnel composé de 64 points.

Interface	Points à charger	Ordre géométrique	Ordre Z
1	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1	2
2	0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51	4	2
3	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60	4	4

TABLEAU 5.6 – Pour chacune des trois interfaces : les points à charger, le nombre de chargements de lignes de mémoire cache nécessaires lorsque les données sont stockées selon l'ordre géométrique et selon le *Z-order*.

la même manière qu'un processeur exécutant une instruction SIMD.

5.7.7 Alignement des données

L'alignement des données est également très important. La configuration matérielle impose des contraintes sur les adresses des éléments. Il est plus facile (sinon obligatoire) pour le CPU de charger un registre de la mémoire lorsque l'adresse est un multiple de la taille du registre. En code scalaire, les chargements des `float` doivent être alignés sur 4 octets. Ceci est fait automatiquement par le compilateur. Cependant, les registres de vecteurs sont plus grands. L'adresse de chargement doit être un multiple de la taille du registre SIMD : 16 pour SSE, 32 pour AVX et AVX2 et 64 pour AVX512.

L'allocation de mémoire alignée doit être assurée par des fonctions spécifiques telles que `posix_memalign`, `_mm_malloc` (compilateur Intel) ou `aligned_alloc` (en *C11*). On pourrait aussi vouloir aligner les données avec la taille du cache (généralement 64 octets). Cela peut améliorer le taux d'accès réussis (*cache hits*) en évitant que les données soient divisées en plusieurs lignes de cache lorsqu'elles entrent dans une ligne de cache. De plus cela évite le *false sharing* entre les *threads*.

5.8 Synthèse des modifications

Jusqu'ici, quatre modifications dont le but est d'accélérer les calculs ont été présentées. Les changements proposés sont de nature très différente. Il y a deux grands groupes : d'une part, les modifications inhérentes au domaine applicatif du code et au code lui-même, d'autre part les modifications possibles selon l'architecture cible.

Les deux premières modifications (section 5.4 et section 5.5) sont liées aux aspects logiciels. Les modifications mathématiques et d'indices peuvent se mettre en place après une étude de la méthode et de son implémentation. Finalement, les techniques présentées pour les modifications des boucles sont indépendantes de la machine où le code va être exécuté. L'application de cette modification peut être étudiée sur n'importe quel code de calcul.

Les deux dernières modifications (section 5.6 et section 5.7) sont liées à l'architecture matérielle. Le balayage des données aura plus ou moins d'efficience selon le nombre des *threads* chargés du calcul et la taille de la mémoire cache. Le *memory layout* et l'utilisation des instructions SIMD pour obtenir une vectorisation du code ou sa SIMDisation va dépendre de l'architecture visée.

Implémentation	Transformation logicielle	Transformation matérielle		Version
		Balayage des données	Disposition mémoire	
Référence	Sans	*Color*	Géométrique	*Ref*
Vectorisation	Indexation	*Color*	Géométrique	*Index*
	Indexation + Boucles	*Color*	Géométrique	*Color*
		Pack	Géométrique	*Pack*
		H-dup	Géométrique	*H-dup*
		T-dup	Géométrique	*T-dup*
		HV-dup	Géométrique	*HV-dup*
		HV-dup	*AoS*	*AoS*
		HV-dup	*SoA*	*SoA*
		HV-dup	*Z-order*	*Z-order*
SIMDisation		*T-dup*	*AoSoA*	SIMD
GPU		*HV-dup*	*AoSoA*	gpu1
		HV-dup	*SoA*	gpu2

TABLEAU 5.7 – Récapitulatif des transformations applicables aux types d'implémentations étudiées dans la thèse.

Dans le cadre de ce travail de thèse, des types d'implémentation ont été étudiés. Sur CPU, une implémentation consiste à utiliser le *multithreading* et la vectorisation en organisant le code pour que le compilateur génère automatiquement des instructions SIMD. Aussi sur CPU, une autre implémentation consiste à créer un code *multithreading* et explicitement SIMDisé. Sur GPU, l'implémentation est réalisée à partir de l'environnement de développement CUDA.

Les sous-sections qui suivent présentent dans le détail les stratégies matérielles utilisées pour chacun de ces types d'implémentations : pour chacune, on décrira quelle combinaison de balayage des données et de *memory layout* ont été étudiées et utilisées. La version de référence et une version pour mesurer l'impact de la transformation mathématique et d'indices uniquement sont aussi présentées. Le Tableau 5.7 récapitule les grandes lignes de ces implémentations, qui vont être étudiées en détail dans les sous-sections qui suivent.

5.8.1 Vectorisation des calculs sur CPU

Afin d'aider le compilateur à vectoriser le code et utiliser les instructions SIMD automatiquement, les transformations *loop unwinding*, scalarisation et fusion des opérations ont été appliquées. La scalarisation et le déroulement des boucles sont faits pour aider le compilateur à comprendre quelles sont les différentes données indépendantes qui doivent être chargées et calculer en parallèle des sous-expressions.

Lorsque le compilateur est confronté à des calculs d'index complexes et à des nids de boucles, le compilateur trouve généralement des dépendances de données (même si elles n'existent pas) et n'est pas capable de vectoriser.

En ce qui concerne le balayage des données, la méthode initiale *Color*, la version *Pack* et les stratégies de duplication des frontières sont combinées avec les dispositions

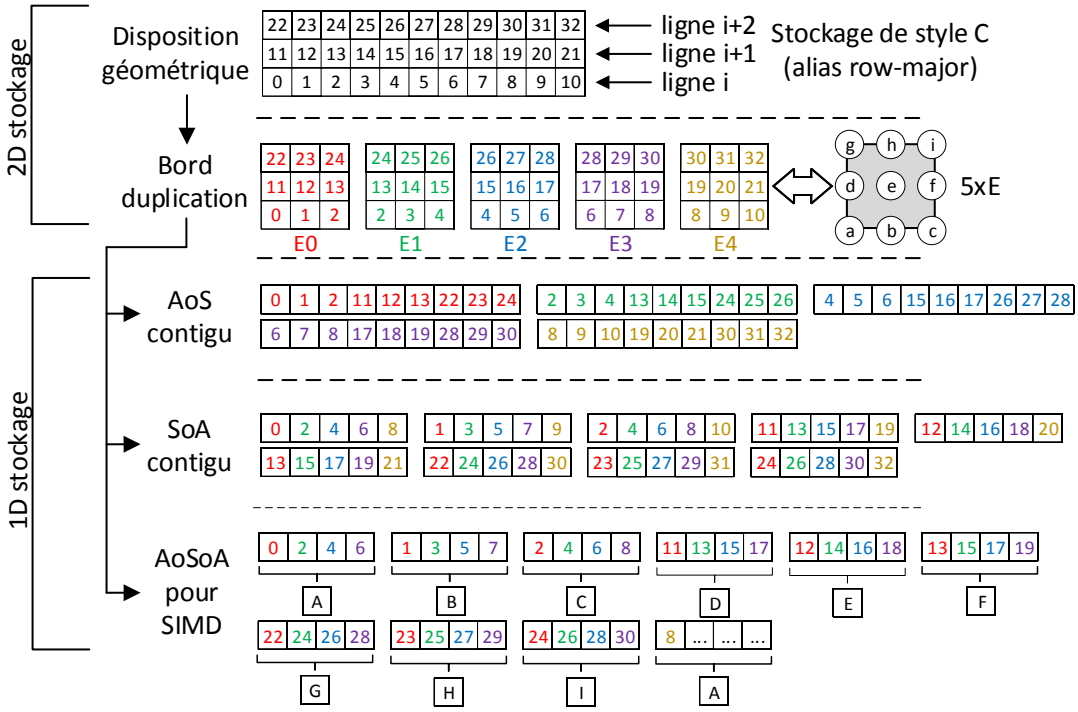


FIGURE 5.13 – Transformation du *memory layout* et duplication des frontières appliquée aux éléments finis. On utilise un schéma 2D pour sa simplicité : tout le stockage 1D est contigu dans la mémoire. Les lettres minuscules représentent les registres scalaires et les lettres majuscules représentent les registres SIMD.

de mémoire présentées. *Color* et *Pack* n'acceptent que la disposition géométrique des données. *H-dup*, *T-dup* et *HV-dup* peuvent aussi être utilisées avec la disposition géométrique. C'est utile pour comparer l'effet de la duplication des points frontières par la disposition géométrique et les *memory layout AoS*, *SoA* et *Z-order*. La conversion de la disposition géométrique en *AoS*, puis *SoA* et *AoSoA* est exposée dans la Figure 5.13).

Le *memory layout Z-order* a aussi été évalué pour une autre raison : on s'attend à ce que le post-traitement soit plus performant que pour *AoS* , car le parcours en Z possède une meilleure localité des données. En revanche, le parcours en Z étant particulièrement complexe, le compilateur risque de ne pas pouvoir vectoriser le code (ou que la vectorisation soit inefficiente). De plus, seuls les éléments avec des dimensions égales à une puissance de deux sont susceptibles d'utiliser ce *memory layout*. On n'inclut pas l'ordre 1 (taille 2) parce que le *memory layout* est la même que pour *SoA*.

5.8.2 SIMDisation des calculs sur CPU

Le codage explicite avec des instructions SIMD en *C* (ou en *C++*) nécessite une connaissance approfondie de l'algorithme, tant du côté des données que du côté des opérations. Cette technique permet un contrôle très fin du codage, mais nécessite beaucoup de modifications du code, rendant sa maintenance plus difficile.

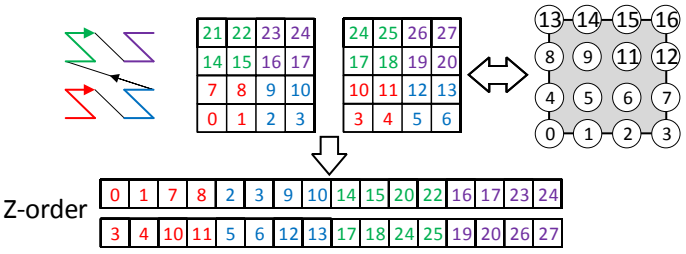


FIGURE 5.14 – Deux éléments de taille 4×4 en mémoire respectant *Z-order*.

Technologie	SSE	AVX	AVX2	AVX512
32-bit `float`	4	8	8	16
64-bit `double`	2	4	4	8

TABLEAU 5.8 – Cardinal SIMD des extensions Intel.

La scalarisation et le *memory layout AoSoA* (pour l'entrelacement des données) sont obligatoires, car les instructions SIMD manipulent des variables SIMD et non des cases mémoire de tableaux. L'utilisation des instructions SIMD est combinée avec au moins une duplication verticale. Une duplication horizontale est également requise pour le *multithreading*, et *T-dup*a été choisie. Les autres modifications logicielles sont optionnelles, mais sont aussi appliquées. La Figure 5.13 montre les différents *memory layout* et comment les points des éléments contigus sont dupliqués et entrelacés (ici le cardinal du SIMD est supposé être égal à 4). En fonction du jeu d'instructions SIMD (SSE, AVX, AVX512) et du type de nombres flottants utilisés, le cardinal varie. Le Tableau 5.8 répertorie les possibilités.

Suivant l'exemple du double produit matrice-vecteur déroulé de l'Algorithme 5.4, on peut montrer le fonctionnement de l'instruction `FMA` (fusion d'un produit et une somme ou *fuse multiply-add* en anglais). Cela permet enchaîner un produit et une somme. L'IA de l'Algorithme 5.7 est (12 FMA) / (8 LOAD + 4 STORE), soit 1. Une IA plus petite signifiant que le processeur consomme rapidement les données, il faut donc faire attention à la vitesse avec laquelle les données sont amenées.

5.8.3 Parallélisation des calculs sur GPU

L'algorithme et les modifications détaillées précédemment sont susceptibles d'être implémentés sur des GPU. Cette voie a d'ailleurs été choisie par plusieurs équipes ayant travaillé sur l'accélération des méthodes SFEM, comme les équipes de Komatitsch et al. [2010] et de Huthwaite [2014]. Ensuite plusieurs caractéristiques sont présentées. Quelques-unes peuvent être favorables à une augmentation de la performance de cet algorithme en GPU, et d'autres peuvent empêcher de pleinement tirer parti de la capacité de parallélisation des GPU.

Du côté positif, il y a trois grands axes qui peuvent aider à améliorer la performance. Premièrement, la méthode des éléments finis spectraux est conçue pour permettre la création de tâches identiques : l'algorithme applique les mêmes opérations sur des mil-

Algorithme 5.7: Double produit matrice-vecteur avec instruction `FMA` si $N = 1$. IA = 1.33.

```
// FMA(a, b, c) = a × b + c
// MEM:LOAD
A_0 ← A[0], A_1 ← A[1];
B_0 ← B[0], B_1 ← B[1];
M_00 ← M[0][0], M_01 ← M[0][1];
M_10 ← M[1][0], M_11 ← M[1][1];
k^-1 ← 1/k;
// CALC
A_0 ← FMA(FMA(M_01, B_1, FMA(M_00, B_0, 0)), k, A_0));
A_1 ← FMA(FMA(M_11, B_1, FMA(M_10, B_0, 0)), k, A_1));
B_0 ← FMA(FMA(M_10, A_1, FMA(M_00, A_0, 0)), k^-1, B_0));
B_1 ← FMA(FMA(M_11, A_1, FMA(M_01, A_0, 0)), k^-1, B_1));
// MEM:STORE
A[0] ← A_0, A[1] ← A_1;
B[0] ← B_0, B[1] ← B_1;
```

liers, voire des millions, d'éléments identiques. Cette caractéristique est notamment liée au caractère explicite du schéma SFEM et au maillage structuré par blocs. L'algorithme peut profiter de la grande quantité des processeurs CUDA parallèles. Sans un maillage structuré, la méthode des éléments finis devient plus complexe [Reguly et Giles, 2015].

Deuxièmement, les modifications logicielles permettent d'atteindre une intensité arithmétique élevée (voir le Tableau 5.2). Les cartes graphiques sont particulièrement adaptées aux codes dans lesquelles la quantité de calculs est importante par rapport aux nombres d'accès mémoire.

Enfin, troisièmement, comme a été dit précédemment, *AoSoA* est une disposition en mémoire idéale pour l'utilisation des unités d'exécution (*warps*) des GPU (en utilisant la duplication des frontières *HV-dup*).

Du côté négatif, dans le contexte d'un matériel spécifique tel que les cartes graphiques, les avantages des transformations de boucles et de scalarisation ne sont pas garantis : le nombre limité de registres par *thread* peut représenter un problème. Les différents niveaux de mémoire des cartes graphiques sont aussi à prendre en compte. L'utilisation de mémoire constante et de la mémoire partagée est probablement indispensable, plutôt que de n'utiliser que des registres. Finalement, la mémoire globale externe d'une carte graphique est limitée et, dans un contexte de problèmes de contrôle non destructif où les maillages nécessitent des dizaines de giga-octets, il faudra diviser le domaine complet en sous-domaines qui tiennent dans la mémoire.

Les deux paragraphes suivants présentent les deux sujets principaux étudiés concernant l'utilisation des cartes graphiques : les temps de transfert des données et les temps de calcul. Dans le cadre de l'étude du temps de calcul, deux implémentations de l'algorithme sont discutées : une privilégiant l'utilisation des registres, l'autre avec une utilisation prioritaire de la mémoire partagée. Pour finir, deux logiciels de simulation SFEM sont comparés.

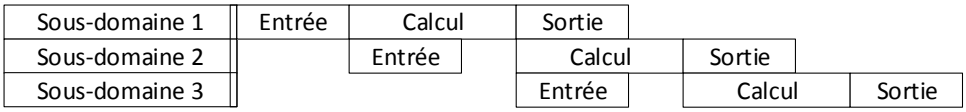


FIGURE 5.15 – Masquage des temps de transfert et de calcul.

5.8.3.1 Temps de transfert

Avant le lancement d'un *kernel*, il faut transférer tous les coefficients constants utilisés dans la mémoire constante, une mémoire paginée spécialisée qui ne permet que la lecture, et qui est plus rapide que la mémoire globale. Ces coefficients représentent environ 1 Ko de valeurs et leur temps de transfert est négligeable. D'autre part, il faut également transférer les données d'entrée (qui seront lue pendant l'étape 1) et les données de sortie (qui seront écrites pendant l'étape 5). La stratégie consiste à charger (potentiellement plusieurs) sous domaines afin de remplir la mémoire GPU et de calculer une itération temporelle avant de les remplacer par un nouveau sous-domaine (la décomposition par sous-domaines est expliquée en sous-section 4.2.2).

Au temps de calcul de chaque sous-domaine, il faut ajouter un temps de transfert pour les données d'entrée vers la mémoire globale, et un certain temps de transfert pour les données de sortie vers le *host* pour récupérer les résultats. Ces transferts de mémoire peuvent être asynchrones : ils peuvent être exécutés en même temps que la carte graphique effectue les calculs des autres sous-domaines. Presque tout le temps de transfert peut être masqué s'il est inférieur au temps de calcul (Figure 5.15).

5.8.3.2 Temps de calcul

Une fois les données chargées dans le GPU, il faut les traiter le plus rapidement possible pour tirer parti de la puissance de calcul des GPU. Deux implémentations sont étudiées : la première, proche de la structure du code pour le CPU, est basée sur une utilisation intensive des registres ; la seconde est basée sur l'utilisation de la mémoire privée des GPU (que NVIDIA appelle *shared memory* car partagé par les cœurs d'un *warp*).

La première implémentation de l'algorithme utilise les *warps* comme unités de traitement SIMT. Le *memory layout* utilisé est *AoSoA* dont le cardinal *SoA* est la taille d'un *warp* (usuellement 32, mais la taille peut également être 16 ou 64). Cette implémentation n'utilise pas de mémoire partagée et repose largement sur des registres, comme le montre l'Algorithme 5.8. Lorsqu'un *thread* du *warp* exécute la tâche associée à un élément, il charge et stocke l'élément et gère tous les résultats intermédiaires dans des variables scalaires (qui doivent être mappés sur des registres physiques). Plus le nombre de registres utilisés par *thread* est élevé, moins il est possible d'utiliser de *threads*, ce qui diminue le taux d'occupation. Néanmoins, avoir un faible taux d'occupation ne signifie pas nécessairement avoir de mauvaises performances [Volkov, 2010].

Algorithme 5.8: Implémentation du noyau de calcul privilégiant l'utilisation des registres.

```
// Load (stage 1)
register A[32] ← input;
register B[32] ← input;
...
register P[32] ← input;
// Calcul
stage1();
stage2();
stage3();
// Store (stage 5)
output ← A[32];
output ← B[32];
...
output ← P[32];
```

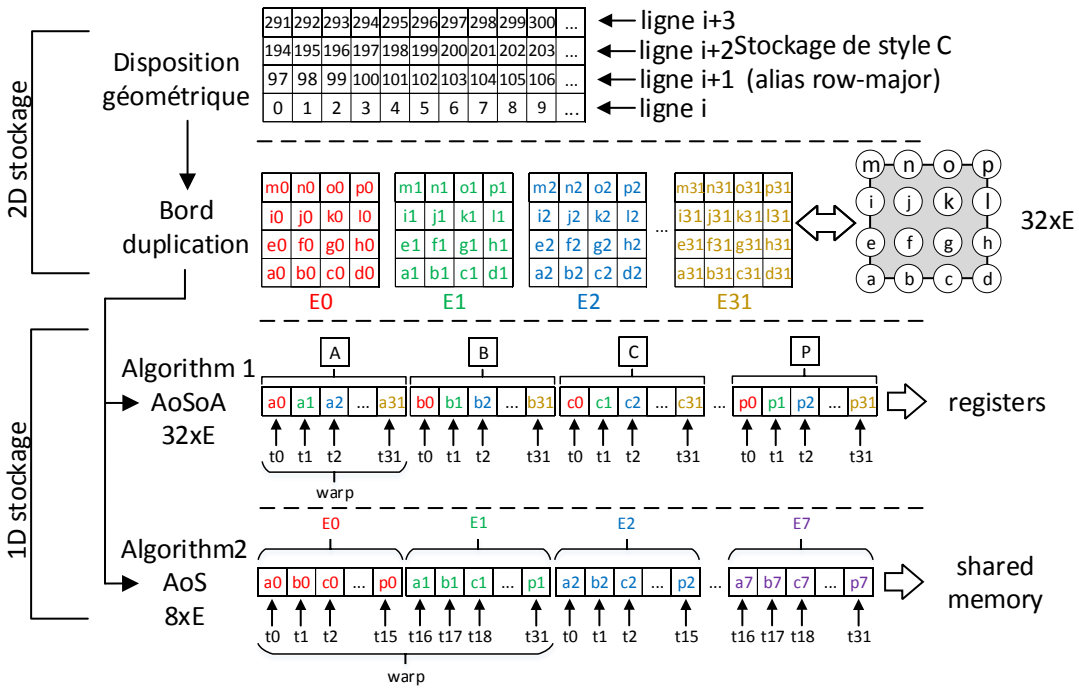


FIGURE 5.16 – La première implémentation de l'algorithme est *AoSoA* consommant 32 éléments par bloc et en utilisant des registres. La seconde implémentation est *AoS* consommant 8 éléments par bloc et en utilisant la mémoire partagée.

La seconde implémentation utilise autant de mémoire partagée que possible pour limiter l'utilisation des registres au compteur de boucle. Le *memory layout* correspond à *AoS*. Un *warp* de 32 *threads* charge deux éléments de 16 points ou variables chacun. Cette implémentation, contrairement à la première, ne peut gérer que des éléments bidimensionnels d'ordre 3. Si la taille des éléments n'est pas un diviseur de 32, certains *threads* des *warps* utilisés seront inactifs. De plus, pour remplir la mémoire partagée, un déroulage de la boucle d'un facteur 4 a été effectué. La boucle peut être déroulée d'une autre taille si la mémoire partagée est capable d'accueillir les données.

Comme les données sont partagées entre les *threads*, une synchronisation est nécessaire entre chaque étape, ce qui ralentira le calcul. Ceci est montré dans l'Algorithme 5.9. L'utilisation de moins de registres permet d'augmenter le taux d'occupation (diminue ainsi le *spill* de registres dans la mémoire cache L1 du cœur CUDA), mais la latence de la mémoire partagée est supérieure à la latence du registre. De plus, la mémoire partagée maximale (l'agrégation des différentes mémoires *shared*) du GPU limite le nombre de *warps* actifs et donc le taux d'occupation.

Algorithme 5.9: Implémentation du noyau de calcul privilégiant l'utilisation de la mémoire partagée.

```
// Load (stage 1)
shared E0[16] ← input;
shared E1[16] ← input;
...
shared E7[16] ← input;
syncthreads();
// Calcul
stage1();
syncthreads();
stage2();
syncthreads();
stage3();
syncthreads();
// Store (stage 5)
output ← E0[16];
output ← E1[16];
...
output ← E7[16];
```

Les deux implémentations utilisent une duplication des frontières pour utiliser *AoS* ou *SoA*, une utilise un *thread* par élément et l'autre un *thread* par point. Cecka et al. [2011] propose trois implémentations différentes. La première applique un *thread* par élément et la stratégie de coloration. La deuxième stocke les résultats dans un *buffer* en mémoire globale avant d'accumuler sur le vecteur résultat pour éviter la concurrence. La troisième utilise ce *buffer*, mais en mémoire partagée. À cause de sa taille limitée, il faut appliquer une décomposition de domaine pour ne pas avoir besoin de plus mémoire partagée que disponible. Finalement, les auteurs proposent d'utiliser un *thread* par point ainsi que des registres, mais ils concluent que cette méthode n'est pas compétitive à cause de la réutilisation des données.

Le logiciel POGO [Huthwaite, 2014], qui est *open source*, propose des simulations par des éléments finis pour des problèmes élastodynamiques. Il utilise des cartes graphiques NVIDIA et intègre des noyaux de calculs codés en CUDA. Ce logiciel contient une stratégie optimisée pour le chargement des données afin de diminuer la communication entre blocs. Pour ce faire, il utilise un partitionnement de maillage aligné efficient et une réorganisation de mémoire à l'intérieur de chaque subdivision.

D'autre part, le logiciel SPECFEM [Tromp et al., 2008] est un code de calcul de référence pour la propagation des ondes sismiques acoustiques et élastiques pour des calculs géophysiques (disponible dans www.geodynamics.org). Le schéma numérique repose sur les éléments finis spectraux. Le code offre le support des cartes graphiques en utilisant OpenCL et CUDA. Ce code [Komatitsch et al., 2009] est implémenté de manière similaire à la deuxième implémentation précédente. Les éléments bidimensionnels utilisés par la deuxième implémentation sont d'ordre 3 pour remplir un *warp* avec 2 éléments de 16 chacun, et leurs éléments tridimensionnels sont d'ordre 4 pour remplir trois *warps* avec 1 élément de 125 points. Ainsi, trois *threads* vont travailler sur des valeurs vides créées artificiellement. Cette technique s'appelle *padding* [Karakasis et al., 2009]. La problématique des accès contigus est résolue avec une stratégie de coloriage au lieu d'une duplication des frontières. Ils vont aussi diviser leurs maillages en paquets qui tiennent dans la mémoire du GPU et faire des transferts asynchrones pour recouvrir le temps de transfert. D'autres similarités sont l'utilisation de la mémoire constante pour les coefficients constants et la réduction des registres en faveur de la *shared memory*. Ultérieurement, le code a été préparé pour être utilisé sur un système distribué de GPU en utilisant communication MPI non bloquant [Komatitsch et al., 2010].

5.9 Synthèse du chapitre

Ce chapitre a présenté un ensemble de transformations. Les premières visent à modifier l'indexation des matrices et des vecteurs locaux. Les deuxièmes sont des transformations logicielles valables pour CPU et GPU consistant à fusionner les opérateurs afin d'augmenter l'intensité arithmétique. Les troisièmes concernent les stratégies de parcours. En plus de la stratégie classique de *Color* (et son extension *Pack*), des stratégies de duplications des frontières sont présentées pour plus de concurrence des calculs, en vue d'une parallélisation par *thread* sur CPU et GPU. Les dernières transformations sont intrinsèquement liées aux architectures cibles. Concernant les CPU SIMD il s'agit du *memory layout AoSoA* appliqué à des données 2D. Concernant les GPU, il s'agit d'adapter les *memory layout AoSoA* et *AoS* aux GPU en ayant recours à la *shared memory* ou aux registres de chaque cœur CUDA. Ces transformations sont évaluées dans le prochain chapitre.

Chapitre 6

Analyse des résultats

Sommaire

Ce chapitre est consacré à la présentation des résultats et à l'analyse d'impact des transformations du chapitre précédent. Nous allons mesurer et analyser les performances obtenues lors de l'exécution des types des implémentations présentées dans le 5.7. Premièrement est présentée la méthodologie d'évaluation de la performance suivie pour obtenir les résultats sur CPU et GPU. Ensuite, les résultats et leur analyse seront présentés pour chacun des types des implémentations : vectorisation sur CPU, SIMDisation sur CPU et parallélisation sur GPU.

6.1 Méthodologie d'évaluation de la performance

Afin d'évaluer l'impact des transformations sur les performances, il y a en place une méthodologie. Plusieurs processeurs multicœurs ont été utilisés pour évaluer l'impact des transformations sur des architectures et le nombre des cœurs. Différentes architectures de GPU ont également été testées. Dans cette section, la méthodologie de la collecte des résultats est présentée, avant de discuter de leur interprétation dans la section suivante.

Famille	Westmere	Broadwell	Skylake
Xeon	X5690	E5-2640 v4	Gold 6126
Fréquence (GHz)	3.47	2.4	2.6
# cœurs	2×6	2×10	2×12
SIMD	SSE4.2	AVX2	AVX512
FMA par cœur	0	2	2
Largeur SIMD	128	256	512
π F32	4	32	64
Π F32	48	640	1,536
L1 Cache (Ko, par cœur)	32	32	32
L2 Cache (Ko, par cœur)	256	256	1,024
L3 Cache (Mo, par CPU)	12	25	19.25

TABLEAU 6.1 – Détails de trois processeurs utilisés pour évaluer les performances. SIMD fait référence à la dernière extension SIMD prise en charge par le CPU. Tous utilisent 2 *sockets* connectés avec un accès mémoire non uniforme (NUMA). La taille de la mémoire cache correspond à la mémoire cache de données, pas à la mémoire cache d'instructions. π est le parallélisme intrinsèque par cœur (opération / cycle / cœur) et Π est le parallélisme intrinsèque total (opération / cycle).

6.1.1 Processeurs multicœurs (CPU)

Les trois familles des processeurs multicœurs sont détaillées dans le Tableau 6.1. Il y a trois jeux d'instructions SIMD différents : SSE (128 bits), AVX (256 bits) et AVX512 (512 bits). Les trois machines sont biprocesseurs. Le nombre des cœurs, la taille des mémoires caches ainsi que le parallélisme intrinsèque par cœur ($\pi F32$) et le parallélisme intrinsèque total ($\Pi F32$) sont indiqués.

Environnement.

Le code a été compilé avec le compilateur Intel C++ Compiler (ICC) version 18.0.1. Les optimisations ont été activées avec `-O3`. La parallélisation des boucles est effectuée avec un `# pragma omp parallel for` standard et l'option de compilation correspondante `-qopenmp`. La vectorisation est forcée à l'aide de `# pragma ivdep` (qui n'est pas nécessaire pour la SIMDisation). L'utilisation des jeux d'instructions est effectuée à l'aide des options de compilation suivantes : `-no-simd` et `-no-vec` pour le code scalaire ; `-xSSE4.2`, `-xAVX` et `-xCORE-AVX512` pour les codes vectorisés ou SIMDisés.

La numérotation des cœurs varie selon la distribution Linux, donc une numérotation complète des cœurs est fournie pour forcer une affinité *compacte*.

Mesures de temps d'exécution.

La mesure de temps est effectuée avec la fonction `rdtsc()`, qui retourne le nombre de cycles. Les technologies *Speed-Step*, *Turbo-Boost* et *Hyper-Threading* ont été désactivées pour garantir une fréquence d'horloge fixe pendant tout le *benchmark*. Cela assure des expériences stables et reproductibles. Chaque mesure est effectuée cinq fois et la valeur minimale est retenue. On peut ainsi observer que si le nombre de cœurs double (en passant de 12 à 24) entre Westmere et Skylake le parallélisme est multiplié par 32. Soit un ×16 à nombre des cœurs égaux. C'est cette augmentation de parallélisme qu'il

	ordre					
version	2	3	4	5	6	7
géométrie 2D						
Color	144	324	576	900	1,296	1,764
H-dup	216	432	720	1,080	1,512	2,016
T-dup	145	326	578	903	1,299	1,768
HV-dup	324	576	900	1,296	1,764	2,304
géométrie 2D SIMD						
petit	-	7	-	-	-	-
grand	-	326	-	-	-	-
géométrie 3D						
Color	62	207	490	956	1,650	2,618
H-dup	92	275	610	1,142	1,918	2,983
T-dup	76	240	549	1,047	1,718	2,796
HV-dup	137	365	761	1,369	2,235	3,405
HVD-dup	205	486	949	1,640	2,605	3,888

TABLEAU 6.2 – Tailles des jeux de données (Mo) pour les différentes stratégies du balayage des données. Le tableau prend en compte à la fois les vecteurs d'entrée et de sortie, où chaque variable est stockée sur un nombre flottant simple précision 32 bits. *Pack* a exactement la même taille que Coloration. *T-dup* suppose le cas où il y a 24 *threads* disponibles, c'est-à-dire 23 frontières dupliquées. Lorsque l'on utilise moins de *threads* la quantité des interfaces dupliquées se réduise, jusqu'au cas d'un seul *thread* correspondant à *Color*.

faut réussir à utiliser de manière efficiente.

Jeux de données.

Les tests sont effectués sur des géométries 2D et 3D. Les ordres utilisés vont de 2 à 7, c'est-à-dire avec des éléments de 3 à 8 points par dimension. La taille du maillage 2D est de $3,072 \times 1,536 = 4,718,592$ éléments et le maillage 3D de $144 \times 144 \times 48 = 995,328$ éléments. Ce sont des tailles typiques dans les calculs SFEM pour le CND (au moins plusieurs centaines de mégaoctets). Le type de donnée est la virgule flottante de simple précision (32 bits).

Les jeux de données utilisés pour la SIMDisation sont un peu différents. Étant donné que les unités SIMD traitent les données à un taux très élevé (jusqu'à 16 fois plus rapidement que les unités scalaires) et sollicitent fortement la mémoire, il est nécessaire de prendre en compte deux tailles de jeux de données pour évaluer leur impact sur les performances maximales. Les deux jeux de données sont 2D et d'ordre 3. Le plus petit jeu de données est composé de $256 \times 384 = 98,304$ éléments, et est entièrement contenu dans les mémoires caches. Le plus grand est composé de $3,072 \times 1,536 = 4,718,592$, et ne peut tenir en mémoire cache.

Plus d'informations sur les tailles de jeux de données sont disponibles dans le Tableau 6.2.

GPU	GeForce GTX TITAN	NVIDIA Quadro P5000	NVIDIA Tesla V100
Architecture	Kepler	Pascal	Volta
Fréquence (GHz)	0.837	1.607	1.246
# cœurs	2,688	2,560	5,120
# SM	15	20	80
Shared memory (Ko)	48	96	96
Π F32	5,376	5,120	10,240
Performance F32 (GFlop/s)	4,495	8,228	14,131
BP externe (Go/s)	288	288	897
Ratio	15.6	28.5	15.8
PCI Express (Go/s)	16 (3.0 x16)	16 (3.0 x16)	32 (4.0 x16)
NVLink (Go/s)	-	160 (1.0)	300 (2.0)

TABLEAU 6.3 – Propriétés de trois différents GPU NVIDIA. La fréquence, la performance et la bande passante (BP) externe correspondent aux valeurs de base. Si l'architecture permet de configurer la mémoire entre la *shared memory* et la mémoire cache L1, la plus grande mémoire partagée possible est privilégiée. Π est le parallélisme intrinsèque total (opération / cycle).

6.1.2 GPU

La description des cartes graphiques utilisées se trouve dans le Tableau 6.3. Si l'intensité arithmétique caractérise un algorithme, le même rapport peut être calculé pour un processeur, c'est le rapport entre la puissance de calcul (en GFlop/s) et la bande passante (BP) externe (en Go/s). Ce rapport permet de voir si une architecture est bien équilibrée ou non. Le tableau montre aussi le débit par PCI Express, par NVLink et la version correspondante.

On peut remarquer que les cartes Titan et V100 bien que ne possèdent pas le même nombre de cœurs ni la même bande passante externe ont un ratio très proche. La carte Quadro P500 semble, elle, privilégier la puissance de calcul au détriment des transferts.

Environnement.

Les performances et la bande passante théorique sont calculées en utilisant l'intensité arithmétique. Ces valeurs ont été vérifiées à l'aide de l'outil *Nvidia Nsight*. Le compilateur est NVIDIA CUDA Compiler (NVCC), version 9.1. L'optimisation complète (`-Ox`) est activée.

Mesures de temps d'exécution.

La mesure de temps est effectuée avec la gestion des événements CUDA : `cudaEventCreate()`, `cudaEventRecord()`, `cudaEventSynchronize()`, `cudaEventElapsedTime()` et `cudaEventDestroy()`.

Jeu de données.

En ce qui concerne le GPU, seule une géométrie 2D composée de $2,048 \times 2,048 = 4,194,304$ éléments d'ordre 3 est utilisée, ce qui donne un jeu de données de 512 Mo.

6.2 Vectorisation des calculs sur CPU

Le Tableau 6.4 rappelle les versions qui vont être comparées pour les codes vectorisés sur CPU.

Implémentation	Transformation logicielle	Transformation matérielle		Version
		Balayage des données	Disposition mémoire	
Référence	Sans	*Color*	Géométrique	*Ref*
Vectorisation	Indexation	*Color*	Géométrique	*Index*
	Indexation + Boucles	*Color*	Géométrique	*Color*
		Pack	Géométrique	*Pack*
		H-dup	Géométrique	*H-dup*
		T-dup	Géométrique	*T-dup*
		HV-dup	Géométrique	*HV-dup*
		HV-dup	*AoS*	*AoS*
		HV-dup	*SoA*	*SoA*
		HV-dup	*Z-order*	*Z-order*

TABLEAU 6.4 – Récapitulatif des implémentations de type vectorisation sur CPU.

6.2.1 Méthodologie d'analyse

Afin d'évaluer l'impact des optimisations, des transformations et les combinaisons des balayages des données et des *memory layout* – pour les géométries 2D et 3D – deux graphiques sont fournis pour chaque architecture du Tableau 6.1. La première évalue la scalabilité de chaque version (GFlop/s par rapport au nombre de *threads*) et va se situer dans la colonne de gauche. La seconde se concentre sur l'évolution du temps d'exécution (tous les cœurs sont actifs) pour traiter 1 million de points par ordre de 2 à 7 (soit de 3 à 8 points par dimension). Ce graphique se situe dans la colonne de droite. La partie blanche des barres représente le temps de l'étape de post-traitement en raison de la duplication des frontières.

Ces graphiques incluent la performance de la version optimisée par Intel du *benchmark* LINPACK. Il permet de mesurer le temps pour résoudre un système dense et aléatoire d'équations linéaires en double précision ce qui permet d'obtenir la performance en GFlop/s. Cette mesure est considérée comme la performance maximale réelle (et non théorique) de la machine.

Comme toutes les architectures héritent des versions précédentes de SIMD et peuvent associer des instructions (par exemple, deux instructions SSE dans une instruction AVX sur un ordinateur AVX ou deux instructions AVX dans une instruction AVX-512, sur un ordinateur AVX512), on fournit de résultats que pour la dernière et la plus grande taille de registre SIMD disponible par machine. Plusieurs tableaux sont ajoutés afin de compléter les analyses présentées dans les paragraphes suivants.

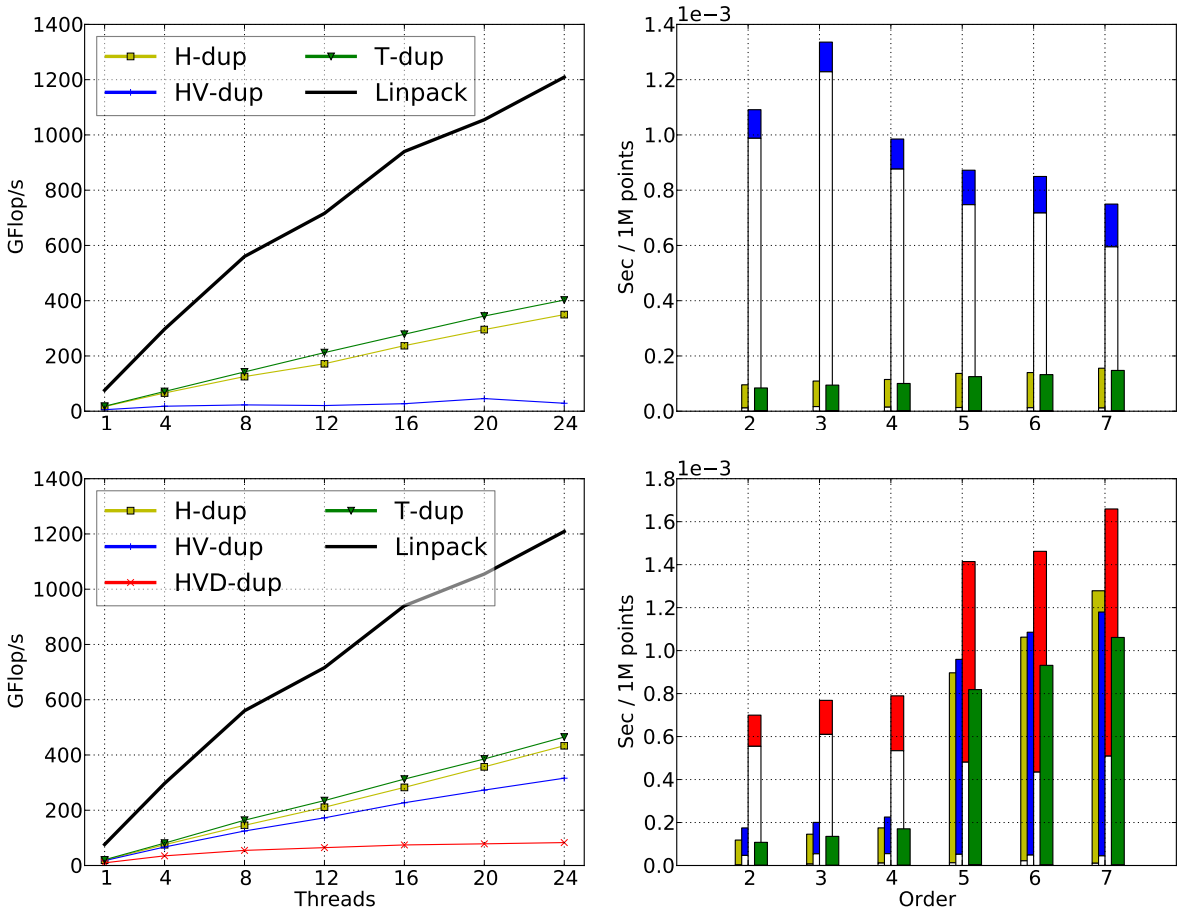


FIGURE 6.1 – Surcoût de la duplication des frontières (partie inférieure blanche de chaque barre) pour les jeux de données 2D et 3D sur Skylake.

6.2.2 Analyse des résultats

Toutes les versions du Tableau 6.4 ont été évaluées sur toutes les machines. Toutefois, afin de limiter le nombre de graphiques et leur complexité, les stratégies de duplication inefficientes ne sont pas présentées après la Figure 6.1.

Pour le jeu de données 2D, l'étape de post-traitement (partie blanche de la barre) de la stratégie *HV-dup* rend cette stratégie inefficiente par rapport à *H-dup* et *T-dup*. Parcourir les frontières dans sens vertical est fortement inefficient vis-à-vis de la mémoire cache. Il est aussi observable que le temps pour l'étape de post-traitement se réduit au fur et à mesure que l'ordre augmente : il y a moins de *false sharing* lorsque la distance des frontières augmente. Pour le jeu de données 3D, *HVD-dup* est inefficiente pour la même raison (la différence de temps d'exécution entre les ordres faibles (2, 3, 4) et les ordres élevés (5, 6, 7) sera analysée plus tard). Dans ce cas, le temps de post-traitement reste similaire puisque la distance en profondeur entre frontières de n'importe quel ordre est assez éloignée pour éviter l'effet de *false sharing*.

La Figure 6.2 regroupe les graphes des codes scalaires et vectorisés sur Westmere, Broadwell et Skylake, pour les ensembles de données 2D (première ligne) et 3D (deuxième ligne). La colonne de gauche montre la scalabilité et la performance, et la colonne de droite indique le temps d'exécution nécessaire pour traiter un million de

# *threads*	Performance GFlop/s	Scalabilité	Efficience	Performance GFlop/s	Scalabilité	Efficience
	Skylake scalaire			Westmere SSE4.2		
1	4	1.0	100%	5	1.0	100%
4	16	4.0	100%	14	2.7	68%
8	31	8.0	100%	26	5.2	65%
12	47	11.9	99%	39	7.8	65%
16	62	15.7	98%			
20	77	19.5	97%			
24	92	23.4	98%			
	Broadwell AVX2			Skylake AVX512		
1	22	1.0	100%	21	1.00	100%
4	81	3.6	91%	81	3.9	98%
8	143	6.4	80%	164	7.9	99%
12	190	8.5	71%	235	11.3	94%
16	256	11.5	72%	313	15.1	94%
20	195	8.7	44%	386	18.6	93%
24				465	22.4	93%

TABLEAU 6.5 – Performance de la stratégie *T-dup* pour la géométrie 3D et l'ordre 3, la scalabilité (performance / performance pour 1 *thread*) et l'efficience (scalabilité / # *threads*).

points (temps par rapport à l'ordre).

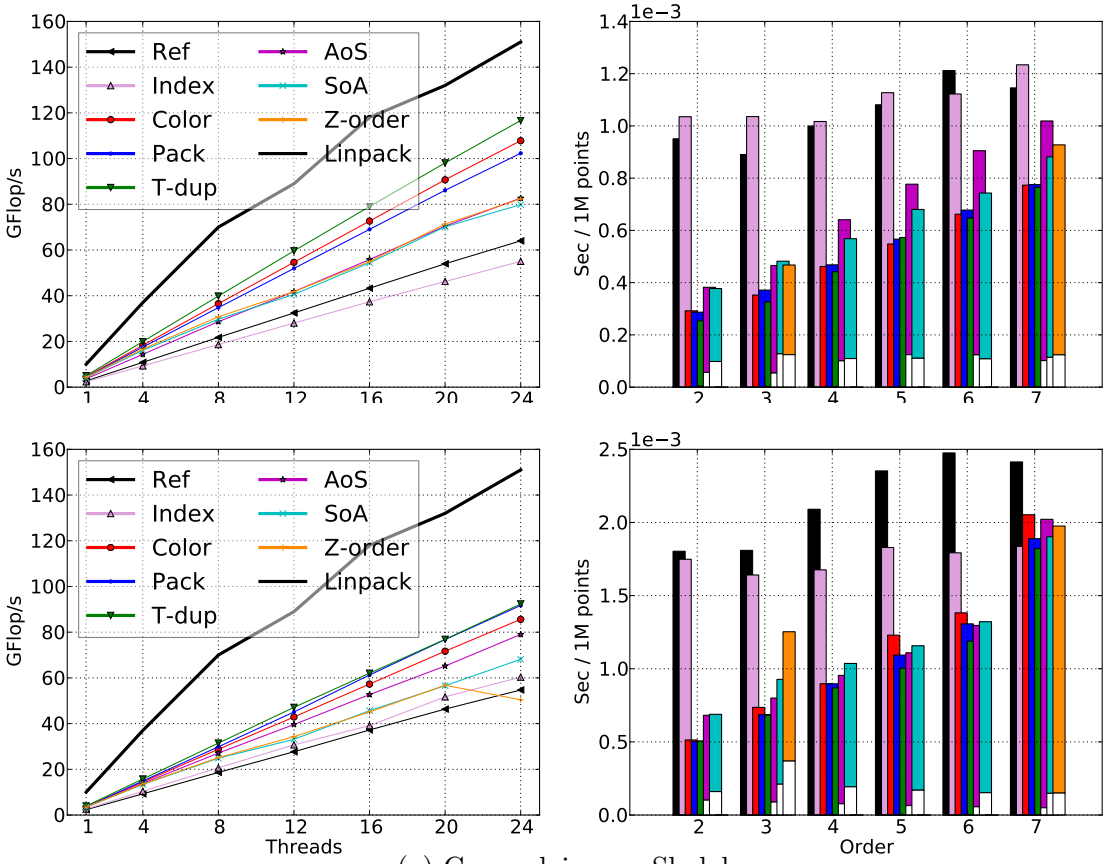
Scalabilité.

Le Tableau 6.5 montre la scalabilité pour la stratégie *T-dup*, mais les autres stratégies se comportent d'une manière similaire. La chute d'efficience sur la machine Westmere est due au changement de l'affinité. Le serveur de calcul utilisé ne permet pas fournir une numérotation des cœurs pour forcer une affinité compacte. La machine Broadwell montre une chute quand tous les cœurs sont actifs, probablement à cause d'un processus non supprimé correctement. La machine Skylake offre des efficiences presque parfaites.

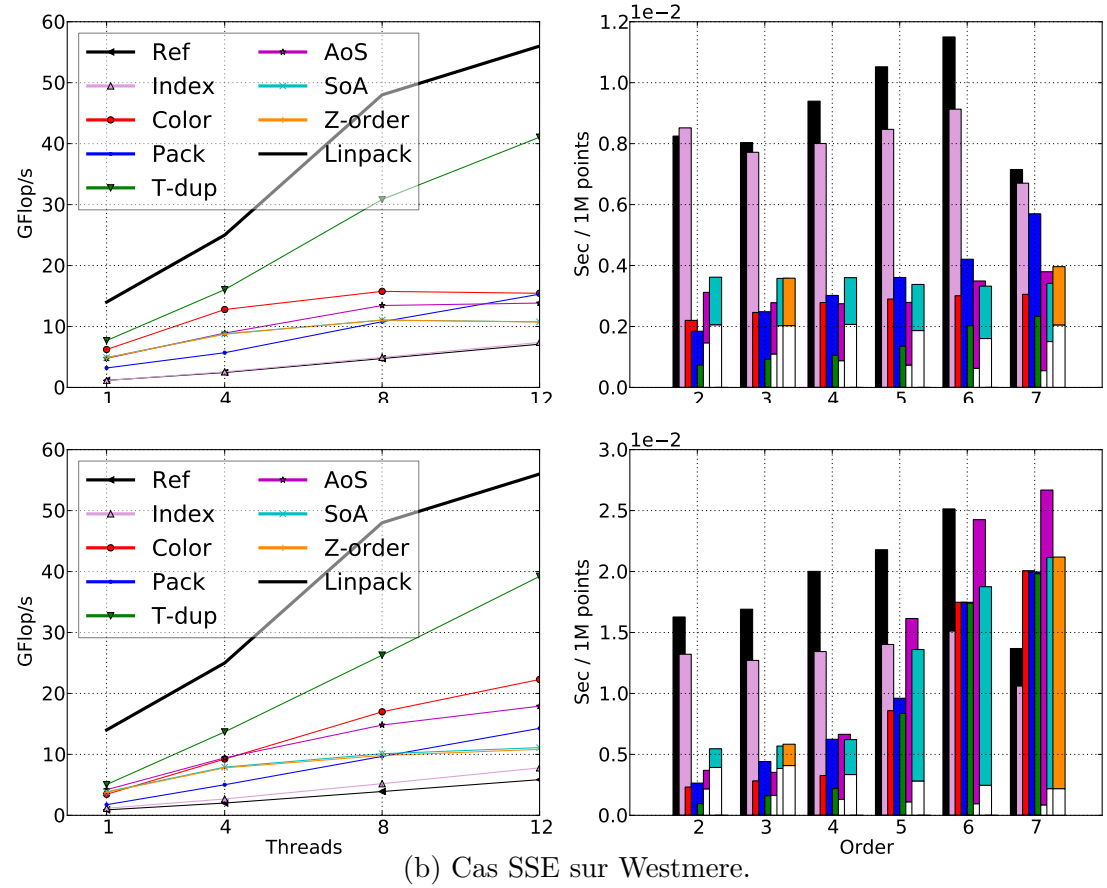
Scalaire.

Les stratégies *H-dup* et *Color* sont plus performantes que la stratégie de référence (voir le Tableau 6.6, 84% et 68% respectivement en 2D, 69% et 56% en 3D), mais *H-dup* se montre 10% supérieure grâce à la localité des données qui lui permet une meilleure utilisation des mémoires caches. La performance de la stratégie *Pack* est 5% inférieure à *Color* en 2D, mais très similaire à *T-dup* en 3D. Cette stratégie aide à éviter le *false sharing* sur des éléments plus éloignés dans la mémoire, mais la complexité ajoutée pour les boucles internes n'est pas compensée en 2D.

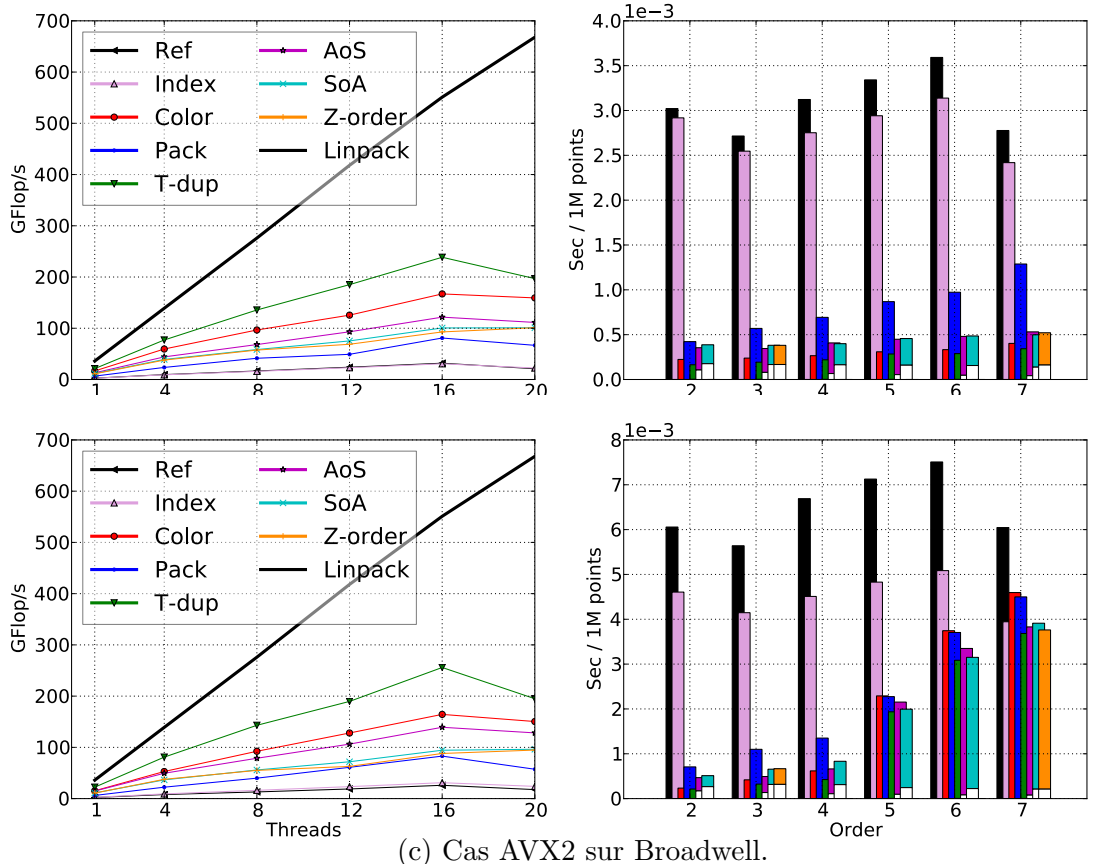
Toutes les stratégies ont une forte intensité arithmétique (voir le Tableau 5.2). Dans ce contexte, la modification du modèle d'accès à la mémoire (*AoS*, *SoA*, *Z-order*) représente au plus une différence de performances de 25%-30% par rapport à *Ref.* Il faut noter que *SoA* et *Z-order* ont des résultats très proches en 2D parce que le *memory layout* est similaire (chaque membre est dans un tableau, voir la sous-section 5.7.5).



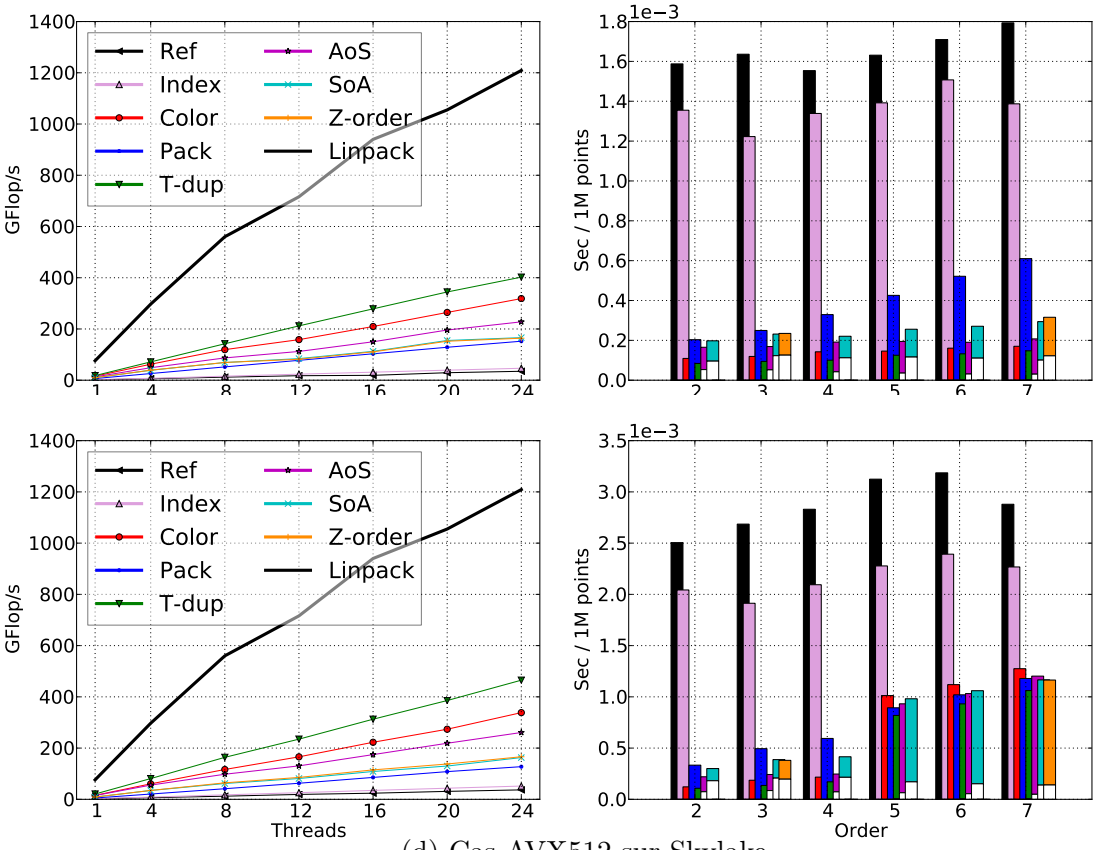
(a) Cas scalaire sur Skylake.



(b) Cas SSE sur Westmere.



(c) Cas AVX2 sur Broadwell.



(d) Cas AVX512 sur Skylake.

FIGURE 6.2 – Performances des codes scalaires et vectorisés CPU (GFlop/s en fonction du nombre de *threads*, temps d'exécution en fonction du nombre maximal de *threads*) pour le jeu de données 2D (la première ligne de chaque machine) et 3D (la deuxième ligne). La partie inférieure blanche de chaque barre représente le surcoût de la duplication des frontières.

Version	Performance GFlop/s	Accélération	Performance GFlop/s	Accélération
	géométrie 2D		géométrie 3D	
Ref	64	×1.0	55	×1.0
Color	108	×1.7	86	×1.3
Pack	102	×1.6	92	×1.7
T-dup	117	×1.8	92	×1.7
AoS	80	×1.2	68	×1.2
SoA	83	×1.3	79	×1.4
Z-order	82	×1.3	50	×0.9

TABLEAU 6.6 – Performance scalaire en 24 *threads* et ordre 3 de la machine Skylake des stratégies *Ref*, *Color*, *T-dup*, *AoS*, *SoA* et *Z-order*. L'accélération est exprimée par rapport aux performances de *Ref*.

En revanche, en 3D la disposition des données n'est pas du tout similaire, et *SoA* est 56% plus performante que *Z-order*.

Vectorisation.

Par rapport à la performance maximale réelle, les codes scalaires fonctionnent bien et se situent entre 70% et 90% de la performance de LINPACK pour les jeux de données 2D et 3D, comme cela est présenté dans le Tableau 6.7. Les codes vectorisés sont plus rapides que les codes scalaires, mais la vectorisation n'est pas parfaite : l'écart entre LINPACK et les codes vectorisés augmente avec la largeur SIMD (rapport 1 3 pour AVX512 sur Skylake). Une inspection du code assembleur montre que les instructions SIMD sont générées, mais ces instructions sont des instructions SIMD scalaires (avec le suffixe `_ss` au lieu de `_ps`) qui manipulent une seule valeur (comme `mulss` ou `addss`). De plus, les codes AVX2 et AVX512 vectorisés ne profitent pas des instructions FMA, contrairement à LINPACK.

Les rapports des vectorisations d'Intel montrent une autre problématique : les étapes de lecture et écriture (étapes 1 et 5 de la section 4.4) ne sont pas performantes. Le message généré est le suivant : `remark #15415: vectorization support: irregularly indexed load was generated for the variable <U[offset]>, 64-bit indexed, part of index is linear but may overflow`. Ce message apparaît pour chaque lecture et écriture des données des étapes 1 et 5. Le code assembleur utilise des instructions `gather`, qui sont moins performantes.

Variation de taille de problème.

Afin de donner plus d'informations sur la performance des codes et leur relation avec la mémoire cache, la taille d'un problème de géométrie 3D a été variée sur la machine Skylake AVX512 (voir la Figure 6.3), depuis ~ 768Ko (remplissage de la mémoire cache L1) jusqu'à ~ 90Go (remplissage de la RAM). En général, la performance augmente au fur et au mesure que la mémoire cache L2 se remplit. Il faut avoir suffisamment de données pour qu'un calcul de ce type profite de l'utilisation des cœurs et de la vectorisation. Une fois que les données sortent de la mémoire cache L2 et commencent à remplir la mémoire cache L3, la performance diminue. C'est le point d'inflexion où le

	LINPACK	*T-dup* 2D		*T-dup* 3D	
	Performance GFlop/s	Performance GFlop/s	Écart	Performance GFlop/s	Écart
scalaire	151	117	77%	92	61%
SSE4.2	56	41	73%	39	70%
AVX2	668	197	29%	195	29%
AVX512	1,209	402	33%	465	38%

TABLEAU 6.7 – Performance de la stratégie *T-dup* et du *benchmark* LINPACK pour les différentes extensions, et leur écart (*T-dup*/ LINPACK). Les machines utilisent tous les *threads* disponibles et l'ordre 3.

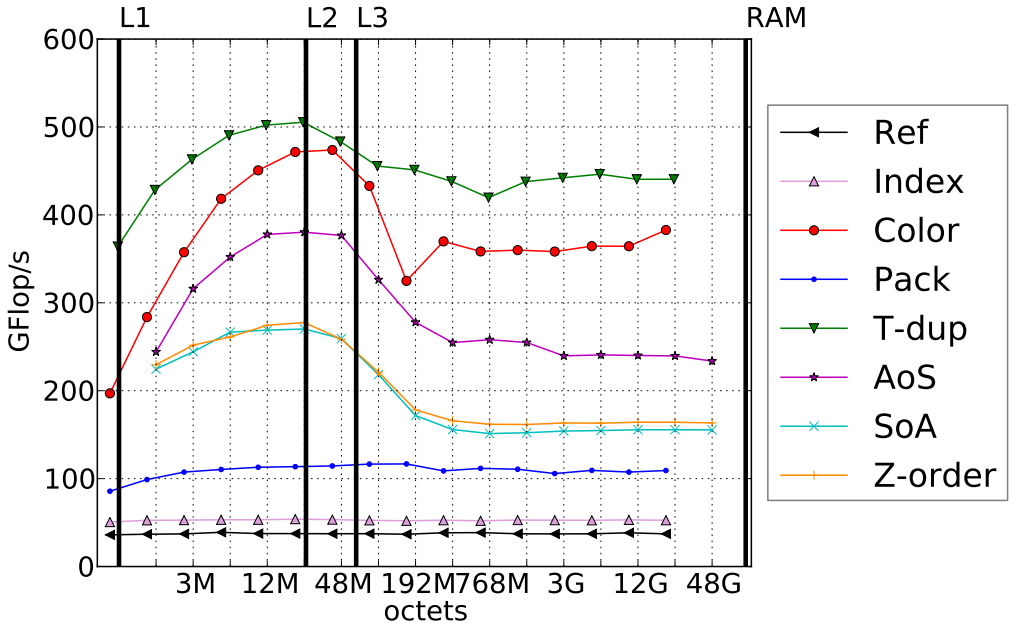


FIGURE 6.3 – Performance des différentes stratégies pour un éventail de tailles de jeux de données. L'abscisse est en log_2, la géométrie est 3D, l'ordre est 3 et la machine est Skylake.

délai d'accès mémoire devient un problème par rapport aux données. Cette mémoire cache est aussi partagée entre les cœurs, donc le risque de *false sharing* est plus élevé. Autour d'une taille de problème de 768Mo, déjà dans la RAM, la performance est stabilisée. Notons que selon le Tableau 6.2, la taille des problèmes va de 62Mo en L3 (*Color* ordre 2) jusqu'à 3.8Go en RAM (*HVD-dup* ordre 7).

Temps d'exécution.

Si on se concentre maintenant sur le temps d'exécution, le premier point significatif est que le jeu de données 2D les codes vectorisés *Ref*, *Index* et *Pack* ont de très mauvaises performances. La raison est simple : les codes ne sont pas vectorisés. L'existence des boucles à l'intérieur des calculs d'un élément empêche la vectorisation. Sur la colonne à droite de la Figure 6.2 on observe l'effet de l'étape de post-traitement. Prenons comme exemple les résultats des stratégies ayant le post-traitement des interfaces sur la machine Skylake AVX512 (voir le Tableau 6.8). Le temps de post-traitement de *T-dup* est négligeable. Le temps de calcul de *T-dup* n'est pas très loin des autres, voire même dans certains cas (*AoS* 3D) est plus lent. Les trois autres *memory layouts* sont proches

Ordre	*T-dup*		*AoS*		*SoA*		*Z-order*	
	Calcul	Post	Calcul	Post	Calcul	Post	Calcul	Post
géométrie 2D								
2	3.5	0.04	4.3	4.1	4.8	2.3	-	-
3	7.1	0.08	8.2	9.3	8.8	3.9	8.2	9.5
4	11.7	0.10	12.8	13.3	17.6	5.0	-	-
5	21.1	0.13	23.7	19.8	27.0	6.1	-	-
6	30.5	0.16	37.0	25.6	36.8	7.2	-	-
7	44.4	0.19	58.3	30.5	53.4	9.2	58.5	37.0
géométrie 3D								
2	2.9	0.02	3.2	4.9	3.9	2.0	-	-
3	8.6	0.03	11.5	13.2	10.0	5.4	11.7	12.5
4	21.1	0.09	24.9	26.7	21.6	9.1	-	-
5	175.2	0.80	174.3	36.7	186.6	13.7	-	-
6	316.8	1.15	310.2	51.7	333.6	18.7	-	-
7	539.1	1.63	523.2	70.8	587.6	25.0	521.8	71.3

TABLEAU 6.8 – Temps (ms) de calcul et de post-traitement des stratégies *T-dup*, *AoS*, *SoA* et *Z-order* pour le calcul et le post-traitement utilisant la machine Skylake (24 *threads*) et l'extension AVX512.

entre eux, mais le temps de post-traitement engendre une différence. *AoS* et *Z-order* ont des temps de post-traitement proches, donc la localité des données de *Z-order* n'est pas meilleure que dans *AoS*. Le temps de post-traitement de *SoA* sont plus faibles. En effet, *SoA* permet de parcourir tous les points des frontières de manière contiguë. En tout cas, tous sont plus inefficientes que *T-dup*.

Géométrie 3D.

Pour le jeu de données 3D, les mêmes observations et analyses peuvent être effectuées. Mais on peut aussi observer que pour les ordres élevés (5, 6, 7), il y a une baisse supplémentaire des performances. Ceci est dû à la non-vectorisation des ordres élevés. Le message du rapport de vectorisation d'Intel est le suivant : `remark #25460: No loop optimizations reported`. Il est impossible de forcer le compilateur à vectoriser. Les codes étant très longs, il le refuse systématiquement.

Variation en fonction de l'ordre.

Lorsque l'ordre augmente, le temps d'exécution par point le fait également. La Figure 6.4 présente le temps d'exécution par un million de points de *T-dup* et le nombre des calculs théorique par ordre pour la même quantité des points, en utilisant (5.5). En 2D, on constate que le temps de calcul augmente plus lentement que le nombre de calculs théorique. Le temps de calcul de cette géométrie est moins limité par les calculs que la géométrie 3D aux ordres 2, 3 et 4. En 3D, comme on l'a observé précédemment, à partir de l'ordre 5, la performance est dégradée du fait de l'absence de vectorisation. Ces comportements sont à peu près les mêmes pour toutes les extensions (Skylake scalaire, Westmere SSE4.2 et Broadwell AVX2).

Accumulation des optimisations.

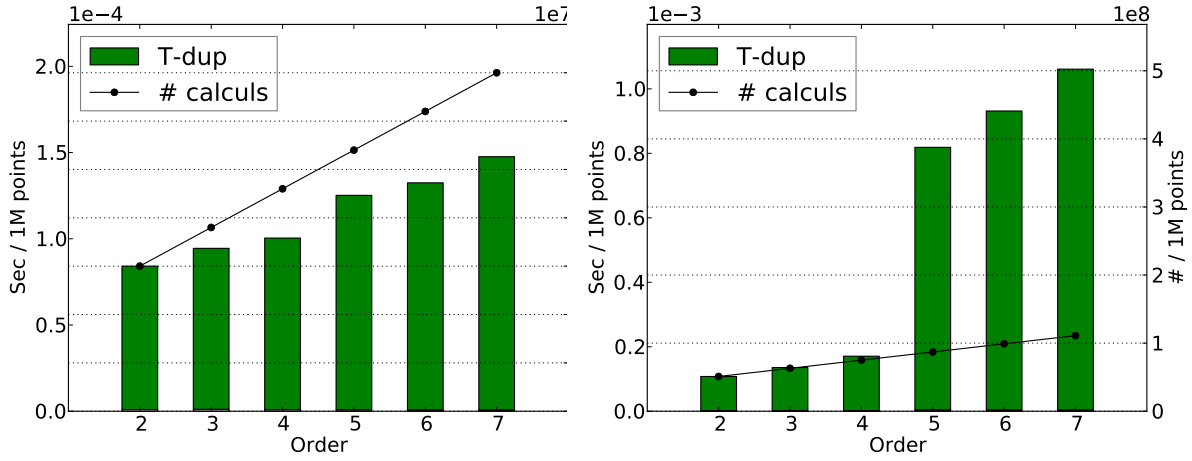


FIGURE 6.4 – Temps d'exécution (bars) de *T-dup* pour traiter un million des points en géométries 2D (gauche) et 3D (droite) sur la machine Skylake AVX512. Nombre des calculs (ligne) pour ces points pour chaque ordre.

La présentation des résultats par accumulation des optimisations aide à comprendre l'effet des optimisations introduites en chapitre 5 (voir le Tableau 6.9). Partons du code de référence. En 2D, la transformation des indices semble négative, mais en 3D elle est bénéfique. Cela est dû à la faible distance en mémoire des variables de l'élément bidimensionnel. Dans ce cas, les accès mémoire sont contigus, mais cela ne compense pas l'obligation de stocker deux fois le même vecteur pendant l'étape 1. En 3D, même s'il faut stocker trois copies, un accès contigu postérieur est avantageux. Les transformations des boucles offrent en 2D une accélération de 3 en temps. Le nombre d'opérations étant réduit, le gain en performance est alors d'un facteur 2 (changement dans l'intensité arithmétique de (4.15) à (5.5)). En 3D, un phénomène semblable se produit. La scalabilité est proche de l'efficience optimale. Le balayage des données permet d'augmenter la performance de 8% et la vectorisation de 345% et 504% respectivement en 2D et 3D.

En résumé, toutes les optimisations ont permis une accélération allant de ×221 à ×314 par rapport au code initial scalaire et séquentiel. Si l'on se compare à la version parallélisée du code de référence sans optimisations, on obtient un code 9.4 en 2D et 13.4 en 3D plus rapide. À noter qu'avec le code de référence, forcer la vectorisation a un effet négatif sur la performance.

Transformation	Version	# *threads*	Extension	Temps ms	Accél.	Performance GFlop/s	Accél.
Code référence géométrie 2D							
sans	*Ref*	1	scalaire	1,579	-	3	-
parallélisation	*Ref*	24	scalaire	67	×23.5	64	×23.5
vectorisation	*Ref*	24	AVX512	124	× 0.5	35	× 0.5
				total	×**12.8**	total	×**12.8**
Nouveaux codes géométrie 2D							
sans	*Ref*	1	scalaire	1,579	-	3	-
indexation	*Index*	1	scalaire	1,833	× 0.9	2	× 0.9
boucles	*Color*	1	scalaire	627	× 2.9	5	× 1.9
parallélisation	*Color*	24	scalaire	27	×23.6	108	×23.6
balayage données	*T-dup*	24	scalaire	25	× 1.1	117	× 1.1
vectorisation	*T-dup*	24	AVX512	7	× 3.4	402	× 3.4
				total	×**221.4**	total	×**147.6**
Code référence géométrie 3D							
sans	*Ref*	1	scalaire	2,711	-	2	-
parallélisation	*Ref*	24	scalaire	115	×23.5	55	×23.5
vectorisation	*Ref*	24	AVX512	171	× 0.7	37	× 0.7
				total	×**15.8**	total	×**15.8**
Nouveaux codes géométrie 3D							
sans	*Ref*	1	scalaire	2,711	-	2	-
indexation	*Index*	1	scalaire	2,413	× 1.1	3	× 1.1
boucles	*Color*	1	scalaire	1,101	× 2.2	4	× 1.4
parallélisation	*Color*	24	scalaire	47	×23.5	86	×23.5
balayage données	*T-dup*	24	scalaire	43	× 1.1	92	× 1.1
vectorisation	*T-dup*	24	AVX512	9	× 5.0	465	× 5.0
				total	×**314.1**	total	×**199.9**

TABLEAU 6.9 – Temps de calcul et performance obtenus avec les différentes transformations cumulatives en géométrie 2D et 3D et l'ordre 3.

6.3 SIMDisation des calculs sur CPU

Le Tableau 6.10 rappelle les versions qui vont être comparées pour les codes SIMDisés sur CPU.

Implémentation	Transformation logicielle	Transformation matérielle		Version
		Balayage des données	Disposition mémoire	
SIMDisation	Ind. + Boucles	*T-dup*	*AoSoA*	SIMD

TABLEAU 6.10 – Récapitulatif des implémentations de type SIMDisation sur CPU.

6.3.1 Méthodologie d'analyse

Les algorithmes du code SIMDisé sont profondément liés à la taille du registre SIMD. L'algorithme gère des éléments 2D dont la quantité des points du premier axe est multiple de 4, c'est-à-dire d'ordre 3 ou 7 (ce dernier si la taille de registre est au moins 8, l'extension AVX ou supérieure). La complexité des algorithmes quand cette condition n'est pas remplie est très haute et n'a pas été étudiée pendant cette thèse. Afin de pouvoir tester sur la machine Westmere, seulement l'ordre 3 est testé.

On évalue la scalabilité (GFlop/s par rapport au nombre de *threads*), mais pas le temps d'exécution par ordre. Ces graphiques incluent aussi la performance maximale réelle fournie par le *benchmark* LINPACK. La différence par rapport à la méthodologie suivie pour les codes vectorisés est la variation de la taille des jeux de données utilisés (petit à gauche, grand à droite, voir le Tableau 6.2), afin d'évaluer correctement leur impact sur les performances.

6.3.2 Analyse des résultats

La Figure 6.5 (à gauche) montre que, pour le petit jeu de données, les performances des codes SIMDisé, sur les trois machines, correspondent aux performances de LINPACK (et surpassent même celles de LINPACK pour AVX512). Mais pour les grands ensembles de données (à droite), les performances chutent jusqu'à 50% de la performance crête de LINPACK (voir le Tableau 6.11). Le petit jeu de données tient dans la mémoire cache L2 et l'unité SIMD est alimentée des données à grande vitesse. L'exécution est *compute bound*. En revanche, le grand jeu de données est dans la mémoire cache L3, plus lente et partagée entre plusieurs cœurs. La chute de la performance en AVX2 et de manière encore plus marquée en AVX512 est dû à une alimentation insuffisante des unités SIMD. L'exécution est *memory bound*.

Afin d'analyser en détail les baisses, toutes les versions du code SIMD (SSE, AVX, AVX2 = AVX + FMA et AVX512) ont été comparées sur la machine Skylake, capable de toutes les exécuter (la Figure 6.6). Pour le petit jeu de données, on peut observer une accélération linéaire presque parfaite (voir le Tableau 6.12), sans chute de performance. L'extension SSE4.2 permet dans les deux cas d'être assez proche du ratio réel par rapport à une exécution scalaire, mais lorsque l'unité SIMD demande plus de données, la performance se dégrade (voir le Tableau 6.13). Les instructions FMA 256 bits

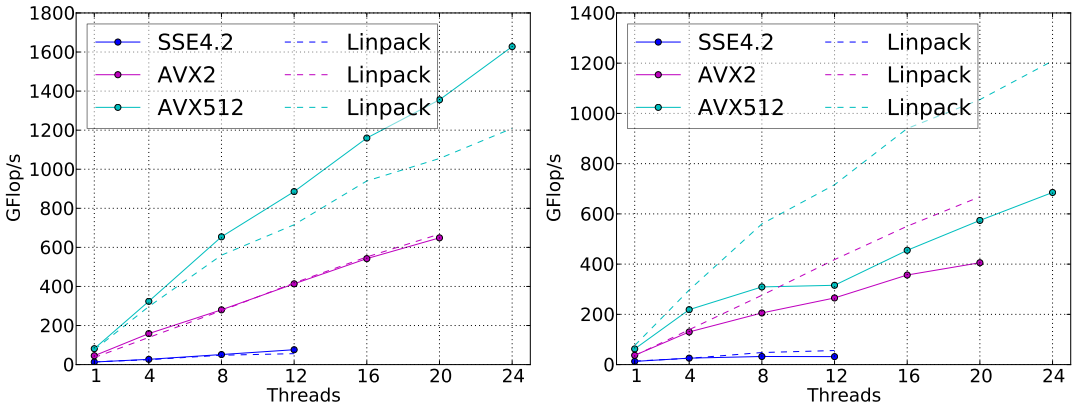


FIGURE 6.5 – Performance de la version SIMD (GFlop/s) contre LINPACK pour le petit (à gauche) et le grand (à droite) jeux de données, sur Westmere (bleu), Broadwell (rose) et Skylake (cyan).

# *threads*	LINPACK	Petit jeux		Grand jeux	
	Performance GFlop/s	Performance GFlop/s	Écart	Performance GFlop/s	Écart
Westmere SSE4.2					
1	14	13	94%	13	91%
4	25	27	108%	26	103%
8	48	51	107%	32	67%
12	56	76	136%	32	57%
Broadwell AVX2					
1	36	45	126%	37	103%
4	139	159	114%	130	94%
8	276	280	102%	206	74%
12	418	414	99%	265	63%
16	551	542	98%	357	65%
20	668	649	97%	406	61%
Skylake AVX512					
1	76	81	107%	62	82%
4	297	324	109%	219	75%
8	560	654	117%	309	55%
12	716	886	124%	316	44%
16	940	1,160	123%	455	48%
20	1,055	1,355	128%	574	54%
24	1,209	1,628	135%	685	57%

TABLEAU 6.11 – Performance de la stratégie SIMD et LINPACK pour les différentes extensions, et leur écart (SIMD / LINPACK).

# *threads*	Performance GFlop/s	Scalabilité	Efficience	Performance GFlop/s	Scalabilité	Efficience
	scalaire			SSE4.2		
1	5	1.0	100%	19	1.0	100%
8	43	8.0	100%	148	7.9	99%
16	85	15.7	99%	292	15.6	98%
24	127	23.7	99%	429	22.9	96%
	AVX			AVX2		
1	35	1.0	100%	45	1.0	100%
8	280	7.9	99%	355	7.9	99%
16	556	15.7	98%	698	15.6	97%
24	805	22.7	95%	998	22.2	93%
	AVX512					
1	81	1.0	100%			
8	654	8.0	100%			
16	1160	14.2	89%			
24	1628	20.0	83%			

TABLEAU 6.12 – Performance de la stratégie SIMD pour la taille petite de problème, la scalabilité (performance / performance pour 1 *thread*) et l'efficience (scalabilité / # *threads*), sur la machine Skylake.

(AVX2) offrent un gain de temps supplémentaire par rapport à AVX qui permettent d'atteindre presque le ratio idéal de 8. Pour les grands ensembles de données, les performances d'AVX, AVX2 et AVX512 sont identiques : les instructions FMA 256 bits et 512 bits n'offrent pas d'accélération supplémentaire par rapport à AVX. Ces codes sont *memory bound* car la quantité d'accès mémoire reste trop élevée par rapport à la quantité de calculs.

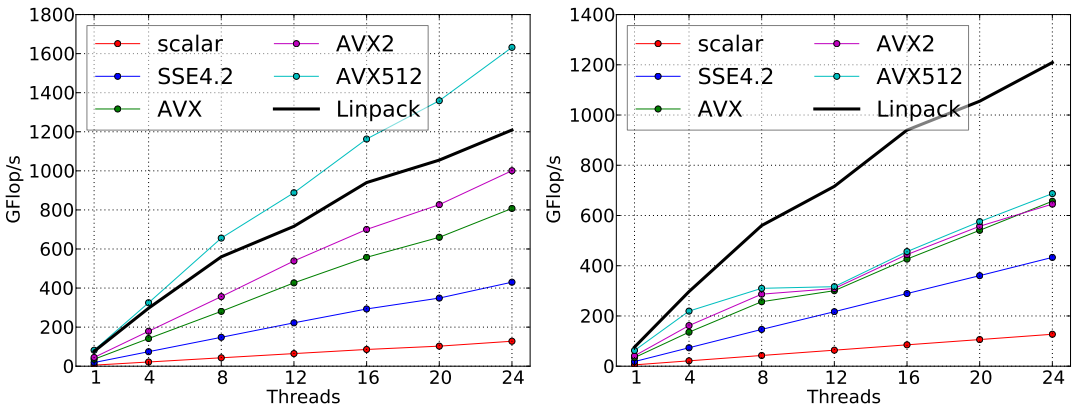


FIGURE 6.6 – Performance de toutes les versions SIMD (GFlop/s) contre LINPACK pour le petit (à gauche) et grand (à droite) jeu de données sur Skylake. La taille du jeu de données limite la performance SIMD.

Comparaison avec la vectorisation.

La Figure 6.7 montre la performance en variant la taille du jeu de données (correspondant avec la Figure 6.3 de la section précédente). Dans ce cas, la version SIMD et la

	Petit jeux		Grand jeux	
	Performance GFlop/s	Ratio	Performance GFlop/s	Ratio
scalaire ($\pi = 1$)	127	1.0	127	1.0
SSE4.2 ($\pi = 4$)	429	3.4	432	3.4
AVX ($\pi = 8$)	805	6.3	655	5.2
AVX2 ($\pi = 8$)	998	7.8	644	5.1
AVX512 ($\pi = 16$)	1,628	12.8	685	5.4

TABLEAU 6.13 – Performance du gain SIMD par rapport au parallélisme SIMD associé (cardinal du SIMD, no FMA).

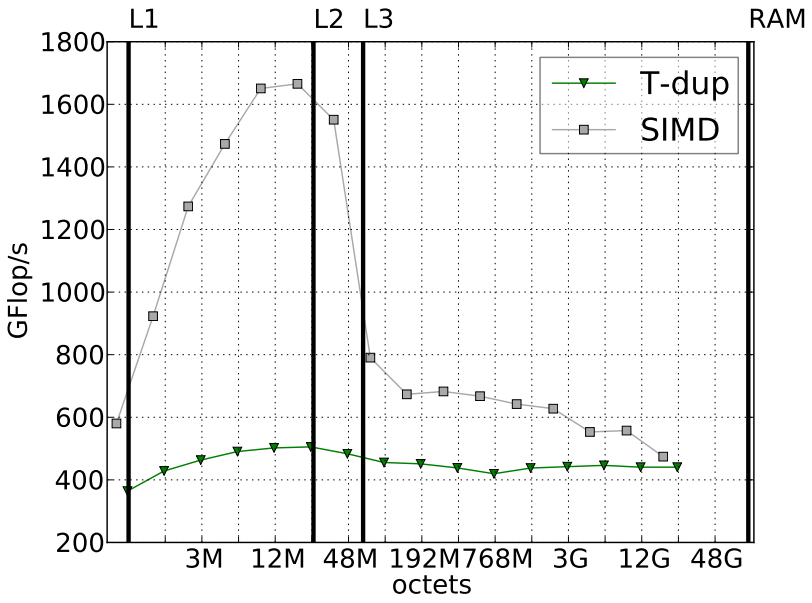


FIGURE 6.7 – Performance de la stratégie *T-dup* vectorisée et du code SIMD pour un éventail de tailles de problème. L'axe des abscisses est donné en log_2. La machine est Skylake.

stratégie *T-dup* vectorisée sont comparées. La performance du SIMD chute fortement quand les données sortent de la mémoire cache L2, les unités SIMD étant incapables d'obtenir les données à la vitesse nécessaire. Une fois dans la RAM, la performance est stabilisée, mais avec une pente descendante vers la performance du code vectorisé. Vers le point de performance maximale permis pour une exécution *memory bound*.

La vectorisation fournit une accélération supérieure à ×1 pour Westmere et supérieure à ×3 par rapport au code scalaire, sauf sur Broadwell pour un grand jeu de données. La SIMDisation fournit × 2 sur Broadwell par rapport au code vectorisé, puisque la taille de données affecte de la même manière le code vectorisé que le code SIMDisé. En revanche, contre, une plus grande différence existe sur Skylake. Le ×4 pour le petit jeu de données certifie que les unités SIMD profitent largement des données dans les mémoires caches plus rapides. En revanche, pour le grand jeu de données, le ratio chute à × 1.7, où la saturation de la mémoire ne permet pas de profiter des instructions 512 bits.

Accumulation des optimisations.

Taille	Performance GFlop/s			Gain			π
	sca	vec	SIMD	vec/sca	SIMD/vec	Total	
Westmere SSE4.2							
petit	39	47	76	×1.2	×1.6	×2.0	×4
grand	33	41	32	×1.3	×0.8	×1.0	×4
Broadwell AVX2							
petit	91	327	649	×3.6	×2.0	×7.1	×8
grand	84	197	406	×2.3	×2.1	×4.8	×8
Skylake AVX512							
petit	117	410	1,628	×3.5	×4.0	×13.9	×16
grand	117	402	685	×3.4	×1.7	×5.9	×16

TABLEAU 6.14 – Performances et gains de la stratégie *T-dup* scalaire (sca), vectorisée (vec) et SIMDisée (SIMD), pour les jeux de données petit et grand et les machines Westmere, Broadwell et Skylake. Finalement, parallélisme SIMD associé (cardinal du SIMD, no FMA).

Transformation	Version	# *threads*	Extension	Temps ms	Accél.	Performance GFlop/s	Accél.
Nouveaux codes géométrie 2D							
none	*Ref*	1	scalaire	1,579	-	3	-
indexation	*Index*	1	scalaire	1,833	× 0.9	2	× 0.9
boucles	*Color*	1	scalaire	627	× 2.9	5	× 1.9
parallélisation	*Color*	24	scalaire	27	×23.6	108	×23.6
balayage données	*T-dup*	24	scalaire	25	× 1.1	117	× 1.1
SIMDisation	SIMD	24	AVX512	4	× 5.7	685	× 5.9
				total	×**367.5**	total	×**251.4**

TABLEAU 6.15 – Temps de calcul et performance obtenus avec les différentes transformations cumulatives pour le petit jeu de données.

Afin d'évaluer l'impact de toutes les transformations qui rendent la version SIMD assez complexe (déroulement de la boucle, entrelacement des données pour *AoSoA*, post-traitement pour la duplication des bords *T-dup*), on peut utiliser le Tableau 6.15, très similaire au Tableau 6.9 de la section précédente. Cette fois, la dernière transformation est la SIMDisation. Au lieu d'une accélération de ×3.45, la SIMDisation a une accélération de ×5.82 en temps et ×5.87 en performance (différence due au changement du nombre des opérations) pour le grand jeu de données.

En résumé, faire la SIMDisation permet une accélération totale de ×368 par rapport au code de référence scalaire séquentiel. Si l'on se compare à la version parallélisée du code de référence sans optimisations (Tableau 6.9), on obtient un code ×16 plus rapide.

Comparaison avec SPECFEM2D.

SPECFEM2D [Tromp et al., 2008] est un logiciel de calcul de référence pour la propagation des ondes sismiques dans des milieux acoustiques et élastiques en 2D. La version générale est discutée dans la sous-sous-section 5.8.3.2. SPECFEM2D et CIVA-SFEM partagent le même schéma numérique, reposant tous deux sur les éléments finis spectraux, mais travaillent à des échelles physiques différentes : CIVA-SFEM travaille à l'échelle millimétrique pour les applications du contrôle non destructif par ultrasons,

	SPECFEM2D	SIMD
Éléments	32,480	33,124
Points	130,561	133,225
Temps ms total	25,559	579
Temps ms / pas de temps	12.8	0.3
Temps ms / 1M de points	97.9	2.2
Performance GFlop/s	-	71

TABLEAU 6.16 – Temps d'exécution séquentiel de SPECFEM2D et du code SIMDisé.

alors que SPECFEM permet des calculs à l'échelle kilométrique pour des calculs de géophysique.

La machine utilisée pour la comparaison est Skylake. On utilise l'ordre 2, SPECFEM2D n'autorisant que des calculs à l'ordre 1 ou 2. SPECFEM2D utilise un maillage carré généré par Gmsh [Geuzaine et Remacle, 2009]. Alors les tailles de problème ne se correspondent pas exactement : SPECFEM2D utilise 32,480 éléments et la maquette utilisons 33,124 éléments (182×182, avec une taille de problème de 1Mo). Le logiciel semble incapable de gérer des maillages de taille plus grande alors que la mémoire RAM du serveur de calcul et l'espace disque étaient suffisants.

La même source pour les ondes a été définie. Par ailleurs, SPECFEM2D a été modifié pour supprimer toutes les écritures des données et les contrôles de flux afin de pouvoir comparer uniquement le temps de calcul. Le nombre de pas de temps a été fixé à 2,000. La maquette est compilée avec le compilateur ICC, tandis que SPCEFEM2D utilise IFC. Les options d' optimisations, OpenMP et AVX512 ont été activés.

Pour obtenir une comparaison valide, on ne peut pas mesurer l'exécution *multithreading*, car SPECFEM2D est conçu pour utiliser des architectures à mémoire distribuée et MPI, tandis que la maquette est conçue pour utiliser OpenMP sur des multiprocesseurs à mémoire partagée. La maquette bénéficie d'une bonne scalabilité, mais SPECFEM2D souffre d'un surcoût de communication MPI. On compare donc les exécutions sur un seul *thread*.

Le code SIMDisé a un facteur d'accélération substantiel de ×44 par rapport à SPECFEM2D (voir le Tableau 6.16). Il y a plusieurs raisons. Premièrement, SPECFEM2D vise à traiter des géométries plus générales en utilisant des maillages non structurés, alors que notre code n'utilise que des maillages réguliers. Cette structure de données plus générique peut générer des accès indirects et des configurations de mémoire plus complexes. Deuxièmement, les dépendances des données entre calculs de différentes itérations empêchent la vectorisation du code (vérifié avec le rapport d'optimisation d'Intel).

6.4 Parallélisation des calculs sur GPU

Le Tableau 6.17 rappelle les versions qui vont être comparées pour les codes parallélisés sur GPU.

Implémentation	Transformation logicielle	Transformation matérielle		Version
		Balayage des données	Disposition mémoire	
GPU	Ind. + Boucles	*HV-dup*	*AoSoA*	gpu1
		HV-dup	*SoA*	gpu2

TABLEAU 6.17 – Récapitulatif des implémentations de type parallélisation sur GPU.

Fonction	Débit Go/s	Efficience
CUDA Bandwidth Test	6.40	100%
`malloc`	3.44	54%
`cudaMallocHost`	6.29	98%
`cudaHostAllocWriteCombined`	6.28	98%

TABLEAU 6.18 – Débit de transfert et l'efficience par rapport au débit réel.

6.4.1 Méthodologie d'analyse

Le code GPU est spécialisé et n'est valable – comme le code SIMDisé – que pour la géométrie 2D et l'ordre 3. Les résultats vont être présentés en tableaux : le débit (Go/s) pour le temps de transfert et la performance (GFlop/s) pour le temps de calcul.

6.4.2 Analyse des résultats pour le temps de transfert

Afin de mesurer le temps de transfert des données depuis le *host* vers le *device*, la carte graphique Titan est utilisée. Cette carte graphique est connectée via un PCI Express x16 2.0, mais elle accepte une connexion PCI Express 3.0 x16.

On considère un problème de taille 2.4Go, un total de 4.8Go entre les données d'entrée et de sortie, qui tient dans la mémoire globale du GPU. Différents types de transfert sont testés pour les comparer au débit de transfert réel fourni par l'outil CUDA Bandwidth Test. Il convient de souligner que ce débit pourrait être plus élevé en utilisant une connexion de carte mère avec une architecture PCI Express 3.0 x16, allant jusqu'à 16 Go/s selon les spécifications techniques de la technologie.

La fonction basique d'allocation de mémoire `malloc` est testée, puis `cudaMallocHost`, qui est cachée par pages, et enfin `cudaHostAllocWriteCombined`, qui devrait être lisible plus rapidement par la carte graphique et plus lentement par le *host*. Le Tableau 6.18 montre que `cudaMallocHost` et `cudaHostAllocWriteCombined` ont une efficience similaire de 98%. Le choix d'utilisation est `cudaMallocHost`, car il est plus facile à mettre en œuvre.

Le débit est facilement atteignable. Soit en Maxwell et Pascal (jusqu'à 16Go/s via le PCI Express 3.0 x16), ou Volta (jusqu'à 32 Go/s via le PCI Express 4.0 x16), le débit de transfert des données peut être supposé comme maximal par rapport au débit réel.

Algorithme	# registres	Occupation	Performance GFlop/s	Bande passante externe Go/s
Titan				
v1 (registres)	63	50%	706	223
v2 (shared)	32	38%	383	121
P5000				
v1 (registres)	72	44%	836	264
v2 (shared)	32	75%	847	267
V100				
v1 (registres)	64	50%	1945	614
v2 (shared)	32	100%	1409	445

TABLEAU 6.19 – Performance et bande passante externe pour les GPU.

6.4.3 Analyse des résultats pour le temps de calcul

L'Algorithme 5.8 (v1) est plus performant que l'Algorithme 5.9 (v2) sur Titan et V100, alors que l'algorithme v2 est proche de v1 sur P5000 (voir le Tableau 6.19). Ces résultats peuvent être reliés au ratio calcul / bande passante : lorsque le ratio est faible, les données peuvent être rapidement chargées dans des registres sans utiliser la mémoire partagée. En revanche, lorsque ce ratio est élevé, les données doivent être chargées dans la mémoire partagée. En d'autres termes, lorsque la bande passante est suffisante, l'algorithme v1 peut utiliser beaucoup de ressources et donc être plus rapide que l'algorithme v2.

Grâce à cette quantité de bande passante (mémoire HBM2), le V100 est proche (ou plus rapide) à une machine Skylake biprocesseur. Ces résultats peuvent être différents pour une machine Skylake comprenant plus de cœurs (et plus de mémoire cache) ou un processeur avec plus de bande passante comme les AMD ryZen.

6.4.4 Entrelacement du calcul et de transfert et masquage

Les problèmes CND 3D habituels sont trop importants pour être entièrement exécutés dans une carte graphique. Chaque sous-domaine doit être transféré vers la carte graphique, calculé, et récupéré vers le *host* (voire la Figure 5.15). Afin de masquer les temps de transfert et optimiser la démarche, il faut que le temps de transfert soit égal ou inférieur au temps de calcul. Le Tableau 6.20 présent le temps de transfert estimé et le temps de calcul pour la taille de jeux de données $2048 \times 2048 = 4,164,304$, correspondant à 512Mo.

Pour les trois GPU, la connexion PCI Express s'avère insuffisante (par un ordre de magnitude). La technologie NVLink sert pour connecter plusieurs GPU, mais les Power8 et Power9 disposent eux aussi des connexions NVLink (1.0 et 2.0 respectivement). Cette connexion permet de transférer assez rapidement les données depuis la mémoire principale vers le GPU. En utilisant ces CPU, la connexion est suffisante entre le GPU P5000 et le CPU Power8 et assez proche entre le GPU V100 et le CPU Power9 (le temps de transfert de l'implémentation v1 est 25% plus lent que le temps de calcul).

Algorithme	Temps ms transfert		Temps ms calcul
	PCI Express	NVLink	
Titan			
	3.0 x16	-	
v1 (registres)	31.3	-	3.6
v2 (shared)	31.3	-	6.7
P5000			
	3.0 x16	1.0	
v1 (registres)	31.3	3.1	3.1
v2 (shared)	31.3	3.1	3.0
V100			
	4.0 x16	2.0	
v1 (registres)	15.6	1.7	1.3
v2 (shared)	15.6	1.7	1.8

TABLEAU 6.20 – Temps de transfert estimé de 512Mo et temps de calcul pour les implémentations sur GPU.

6.5 Conclusion du chapitre

Ce chapitre analyse les résultats des optimisations présentées dans le chapitre précédent. Différents types des implémentations ont été étudiés : la vectorisation sur CPU, la SIMDisation sur CPU et la parallélisation sur GPU. Les résultats de chacune de ces implémentations ont été analysés et disséqués. Les bénéfices d'un code optimisé puis vectorisé (performance ×200) ou SIMDisé (performance ×250) parlent d'eux même. Les avantages de l'utilisation des GPU sont eux aussi évidents : la capacité d'utiliser des milliers des cœurs CUDA en parallèle sur ce type d'algorithme de calcul permet d'atteindre des performances plus élevées que sur des CPU.

Pour finir ce chapitre, il faut obtenir une comparaison entre différents matériels. Il est intéressant de se placer d'un autre point de vue, à savoir celui du coût. Le Tableau 6.21 montre le temps et la performance de l'implémentation plus performante sur CPU (le code SIMDisé) et sur GPU (v1 ou v2 selon l'architecture). La taille de jeux de données 2D est $2048 \times 2048 = 4,164,304$ éléments d'ordre 3. La performance de la meilleure version du code (v1, utilisant des registres) sur le GPU V100 est 3 fois supérieure à celle du meilleur code (SIMD) sur la machine Skylake, 4 fois supérieure à celle de la machine Broadwell et 2 fois supérieure à la version du code v2 sur la P5000. Mais, bien évidemment, le prix est aussi supérieur pour une carte graphique qui est une des plus évoluées de la gamme et possède 5120 cœurs. Le meilleur ratio prix / performance est la SIMDisation en Broadwell et le code GPU sur la machine P5000.

Il est néanmoins très important de noter que cette comparaison a été faite sur un cas particulier à savoir l'ordre 3 en 2D. Les codes vectorisés, qui sont de manière systématique moins performante que les codes SIMDisés, possèdent un atout considérable par rapport aux deux autres : elle est beaucoup plus facile à généraliser à différents ordres et au 3D. Cette question de la généralisation et de la maintenabilité du code est présentée dans le chapitre 8.

Version	Architecture	Temps ms	Prix $	Performance GFlop/s	$ / GFlop/s
SIMD	Skylake	3.9	1,776	678	2.6
SIMD	Broadwell	6.0	939	439	2.1
v1	V100	1.3	5,995	1,945	3.1
v2	P5000	3.0	1,759	847	2.1

TABLEAU 6.21 – Temps et performance pour les versions les plus performantes sur CPU (SIMDisé sur les machines Skylake et Broadwell) et sur GPU (v1 sur le GPU V100 et v2 sur le GPU P5000). Le prix de Skylake et Broadwell est le prix de vente conseillé officiel d'Intel, qui a arrêté la production de Westmere. Le prix des GPU est le prix constaté sur le marché. Ce tableau ne tient pas en compte le prix du *host*.

Chapitre 7

Conclusion

Le but de cette thèse étant d'optimiser le temps de calcul du logiciel CIVA-SFEM, un code basé sur la méthode des éléments finis spectraux. Cette plateforme multitechnique propose un ensemble d'outils de simulation et de modélisation pour le contrôle non destructif. CIVA-SFEM traite spécifiquement de la technique de contrôle non destructif par ultrasons pour détecter de potentiels défauts à l'intérieur de pièces industrielles. Contrairement aux méthodes semi-analytiques de CIVA qui sont basées sur des modèles de tracés de rayon, la méthode des éléments finis spectraux d'ordre élevé permet de simuler avec un haut niveau de précision un grand éventail de géométries de défaut, et permet de prendre en compte des ondes de surface. Le prix à payer est celui d'un coût de calcul plus élevé que les méthodes semi-analytiques. L'objectif était donc de réduire le temps de calcul afin de rendre l'outil viable pour une utilisation industrielle sur des architectures standard PC et stations de travail. La thèse s'est focalisée sur des implémentations optimisées pour matériels standard, à savoir des processeurs multicœurs SIMD et des cartes graphiques NVIDIA.

Il a été démontré que les noyaux initiaux de calculs de la méthode des éléments finis spectraux ont une efficience qui est limitée par les accès mémoire. Cela est dû à leur faible intensité arithmétique. L'objectif a donc été de réduire la quantité des accès mémoire, et d'accélérer les accès indispensables via une utilisation efficiente des mémoires caches.

Cette thèse a présenté quatre transformations de différentes ampleurs pour optimiser les accès mémoire du code CIVA-SFEM.

La première optimisation proposée est une indexation améliorée des matrices et vecteurs parcourus afin d'obtenir des accès contigus en mémoire et une meilleure réutilisation des données en mémoire cache .

La deuxième optimisation est une combinaison de transformations de boucles (*loop unwinding*, scalarisation) et de fusion des opérations (factorisation des calculs). Ces transformations visent à profiter des accès mémoire plus rapides grâce à l'utilisation intensive des registres.

Grâce à la combinaison de la méthode des éléments finis spectraux avec la stratégie de décomposition de domaine, la structure de données est régulière et propice aux optimisations. Cela va permettre d'appliquer plusieurs stratégies pour le parallé-

lisme des tâches et des données. La troisième optimisation explorée est la mise en place de stratégies de parallélisme de tâches. Ont notamment été étudiées une stratégie de coloration des éléments pour permettre un parallélisme des *threads* sans encourir des problèmes d'accès concurrents, ainsi que la possibilité d'une duplication des frontières entre éléments.

La quatrième optimisation est relative au parallélisme des données. Ces transformations sont liées aux architectures cibles : des codes vectorisés sur CPU (disposition géométrique, *array of struct* et *struct of array*), des codes SIMDisés sur CPU (*struct of array* hybride) et des codes spécifiques sur GPU (*struct of array* hybride et registres combinés à une scalarisation ou à de la *shared memory*).

La combinaison des ces transformations a produit 13 codes différents qui ont été testés sur plusieurs architectures CPU et GPU.

Les codes vectorisés montrent en général une bonne scalabilité en géométries 2D et 3D. La performance de la vectorisation souffre d'une utilisation insuffisante des instructions SIMD (comme le FMA). Les codes obtenus sont plus performants que le code scalaire, mais s'éloignent des performances optimales au fur et à mesure que la largueur des registres SIMD augmente. En faisant varier la taille du problème résolu, il a été montré que les performances des codes sont sensibles à la taille des mémoires caches. La variation du temps de calcul en fonction de l'ordre montre que les codes sont aussi sensibles à la quantité des calculs, ce qui prouve que les optimisations nous ont fait sortir d'un régime purement limité par les accès mémoire. Il y a un certain équilibre entre l'impact des accès mémoire et le nombre des calculs. Finalement, la stratégie de duplication des frontières *T-dup* limitant la duplication au nombre des *threads* avec le *memory layout* géométrique permet un code 13 fois plus rapide en 3D que la version parallélisée du code de référence. En 2D, le code est 9 fois plus rapide.

Les codes SIMDisés permettent une utilisation des instructions SIMD encore plus performante que les codes vectorisés. Quand la taille de problème permet aux données de rester à l'intérieur des caches, la performance est 4 fois plus élevée que le code vectorisé. Si ce n'est pas le cas, on observe une tendance à obtenir des performances diminuées (mais plus élevées que les codes vectorisés). Le code SIMDisé est 15.6 fois plus rapide que le code de référence.

Les codes GPU montrent le besoin d'avoir des cartes graphiques avec un bon équilibre entre la performance et la bande passante interne. La performance maximale n'est pas atteignable si la bande passante interne n'est pas suffisante. Dans le cas de la carte P5000 comparée à la Titan, on observe ainsi que les performances sont les mêmes, alors que la puissance de calcul de la P5000 est deux fois supérieure. La bande passante interne a empêché l'exploitation de cette puissance de calcul. Par ailleurs, sur une architecture de pointe (V100), le nombre des registres par cœur et la bande passante interne sont élevés. La performance obtenue peut ainsi être multipliée en utilisant des implémentations très intensives en registres.

Les deux premières transformations présentées sont applicables sur n'importe quel code de calcul intensif. Ce sont des principes d'écriture génériques et des orientations à suivre pour optimiser la performance. Les deux dernières transformations, portant sur

la parallélisation des tâches et des données, s'appuient sur un maillage structuré (calcul des indices explicite). Néanmoins, elles sont aussi applicables sur des maillages non structurés faisant attention aux indices indirects qui peuvent dégrader la performance.

Les travaux présentés dans cette thèse peuvent être comparé aux travaux réalisés sur deux codes implémentant la méthode des éléments finis spectraux ayant fait l'objet de nombreuses publications : SPECFEM et POGO.

Dans le cas de SPECFEM, l'accent est mis sur la parallélisation multi domaines, la parallélisation étant réalisée avec MPI. Une implémentation GPU a également été réalisée afin de pouvoir adresser des architectures de type supercalculateur avec GPU. Le code fonctionne sur des maillages hexaédriques non structurés, avec des éléments d'ordre élevés.

Dans le cas de POGO, les maillages utilisés sont des maillages non structurés triangulaires ou tétraédriques. Ceci permet notamment de travailler avec des mailleurs tétraédriques et triangulaires, ce qui offre la possibilité d'exploiter de nombreux outils de maillage disponibles, et de traiter des géométries complexes, alors que le maillage hexaédrique reste un sujet de recherche et nécessite des outils ad hoc pour chaque type de géométrie rencontré. L'implémentation de POGO permet d'exploiter les capacités des GPU. Néanmoins, l'implémentation est limitée aux éléments finis spectraux d'ordre un.

Les travaux présentés ici ont mis notamment l'accent sur la possibilité d'utiliser les architectures courantes et de tirer parti au maximum des possibilités offertes par le *multithreading* et la SIMD.

Chapitre 8

Perspectives

Sommaire

Ce chapitre aborde les perspectives ouvertes par les travaux de thèse. Dans un premier temps, la possibilité d'utiliser le calcul hybride, c'est-à-dire les noyaux de calcul sur CPU et GPU en même temps, est évoquée. Ensuite, les options pour l'optimisation automatique sont discutées. Finalement, la question de la valorisation de ces travaux se pose, concernant l'intégration des travaux dans la plateforme CIVA et les problématiques associées.

8.1 Calcul hybride GPU/CPU

Quand une ou plusieurs cartes graphiques sont utilisées, le modèle maître-esclave est le plus utilisé pour organiser le séquençage des calculs. Dans ce paradigme, le CPU est chargé d'initier la copie des données vers les GPU, le lancement des *kernels*, et la récupération des résultats. Les tâches de calcul ne sont faites que par les cartes graphiques, tandis que le CPU se voit confier un rôle de simple planificateur. Dans ce cas, qui vise essentiellement des configurations facilement accessibles aux utilisateurs de CIVA, l'utilisation d'une seule carte graphique est la plus fréquente, la gestion est plus légère et le CPU peut se charger lui-même de quelques tâches de calcul. On parle alors d'un modèle hybride. Un code qui applique ce modèle est plus complexe et plus compliqué à maintenir, puisque plusieurs paramètres doivent être pris en compte pour fixer les charges du travail respectif de la carte GPU et du CPU.

Le premier point à traiter est le transfert des données vers la carte GPU. Il a été déjà mentionné que la taille des problèmes traités ne tient pas nécessairement dans la mémoire interne de la carte graphique. En revanche, CIVA-SFEM découpe le maillage en sous-domaines pour des contraintes géométriques (sous-section 4.2.2), et ces sous-domaines sont d'une taille qui tient dans la mémoire. Il semble donc pertinent d'associer

une tâche à l'exécution d'un seul sous-domaine.

Papadrakakis et al. [2011] propose précisément une décomposition de domaine pour un calcul hybride pour une méthode FETI (*Finite Element Tearing and Interconnect* en anglais). Le transfert d'un sous-domaine doit être asynchrone, ainsi que le lancement du *kernel* associé. L'utilisation de *cuda streams* est nécessaire pour lier chaque transfert, exécution et récupération des données de manière asynchrone vis-à-vis de la CPU.

Le deuxième point à prendre en considération est l'équilibrage de la charge du travail entre GPU et CPU. Si le GPU est plus rapide que le CPU, il faut lui donner plus de tâches. Lang et Rünger [2013] proposent des fonctions pour évaluer l'équilibrage des tâches pour une méthode d'éléments finis. Les arguments des fonctions sont les temps de transfert et d'exécution du GPU et CPU, les éléments traités par chacun et le nombre des processeurs. Dollinger et Loechner [2015] proposent pour leur part un équilibrage des données guidé par la prédiction du temps. C'est un système automatique qui génère un code parallèle OpenMP pour CPU et un autre CUDA pour GPU, et il les utilise pour alimenter un algorithme de planification. Finalement, la bibliothèque StarPU, présentée en sous-section 3.2.2, offre un équilibrage des tâches sur des systèmes CPU/GPU/MIC.

8.2 Optimisation automatique et précalcul

La plupart des noyaux de calcul peuvent profiter d'optimisations automatiques, ou *auto-tuning* en anglais, pour améliorer la performance. Ces optimisations automatiques peuvent être au niveau des options de compilation ou programmées à l'intérieur des codes pour être déclenchées lors de l'exécution. Datta et al. [2008] discutent les deux approches : utiliser des options de compilation spécifiques à chaque architecture et faire varier les paramètres du code pour analyser les résultats. Cette section étudie les perspectives de ces travaux dans le cadre de la deuxième approche.

Plusieurs paramètres influençant l'efficience de l'implémentation de la méthode des éléments finis spectraux sont à prendre en compte, mais il y en a deux qui ont une importance particulière : la taille de chaque sous-domaine et l'ordre. Pendant le chapitre 6, il a été discuté de l'influence de ces deux variables sur différentes architectures et leur impact. Dans certains cas, comme les ordres plus élevés, leur utilisation est déconseillée. Mais dans d'autres, il manque encore une étude plus approfondie. Une combinaison des ordres plus élevés et de maillages composés de moins d'éléments, ou le contraire pourrait être plus performant.

Un tel *auto tuning* pendant l'exécution est compatible avec un équilibrage des données comme celui qui a été présenté pour un calcul hybride. Un modèle prédictif pourrait tester un maillage prédéfini avec différents ordres en CPU et GPU, afin de trouver un équilibrage des sous-domaines sur les différentes architectures, où chaque sous-domaine peut être d'une taille et un ordre spécifique. Ces travaux pourraient augmenter la performance totale du noyau de calcul. En revanche, modifier l'ordre entraîne un changement de la précision du calcul (erreur d'approximation, voir sous-section 5.3.1). Il faudrait donc ajouter des mécanismes de contrôle pour borner l'erreur au moment de changer l'ordre.

8.3 Intégration des travaux dans CIVA

Les travaux réalisés pendant cette thèse s'inscrivent dans le développement de CIVA-SFEM. La création d'une maquette a permis d'étudier des voies d'optimisation parfois inexplorables dans le cadre d'un développement industriel d'un outil de la taille de CIVA. Mais comme cela est rappelé dans les choix techniques de la section 3.3, le but est bien d'enrichir CIVA avec les apports de cette thèse.

Cette section discute les enjeux propres à l'intégration de ces travaux. Premièrement, les modifications algorithmiques nécessaires pour appliquer les travaux à un plus grand éventail de cas : l'équation élastodynamique, les matériaux inhomogènes et les matériaux anisotropes. Cela amène au deuxième enjeu, la mise en œuvre d'un générateur de code pour alléger la création et la maintenabilité des codes. La troisième problématique est liée à la relation entre les sous-domaines qui forment un partitionnement de la région d'intérêt et la relation entre cette région et la région au-delà de la PML (Perfectly Matching Layer).

8.3.1 Généralisation des équations disponibles

Les travaux de cette thèse ont porté sur le cas acoustique des ondes dans un matériau homogène. L'intégration dans CIVA nécessite le traitement de deux cas, acoustique et élastodynamique. De plus, ces deux équations doivent être combinées avec des matériaux homogènes, inhomogènes et anisotropes. Heureusement, il est possible d'utiliser un seul algorithme plus général pour traiter les trois types de matériaux. Par la suite sont décrits, de manière simplifiée, les changements nécessaires pour traiter ces cas.

La différence entre une onde acoustique et une onde élastique est la caractérisation vectorielle de la deuxième, par opposition au caractère scalaire de la première. En effet, les variables d'une onde élastique sont des vecteurs, possédant donc deux ou trois valeurs au lieu d'une, en fonction de la dimension. Si en acoustique on a une variable par point de l'élément, en élastique à chaque point correspondent plusieurs variables. Le volume de données lu et écrit est donc doublé ou triplé. Les opérations matricielles doivent être appliquées sur chacune des valeurs des variables. Cela signifie un problème doublé/triplé en taille, en opération et en longueur du code.

Afin de prendre en compte les différentes géométries utilisées par CIVA-SFEM, une généralisation du problème est nécessaire. Dans le cas d'éléments non parallélépipédiques, les coefficients supposés constants (voir sous-section 4.3.3) ne le sont plus : ils varient selon la maille. On parle notamment des coefficients de l'étape trois du noyau de calcul, concernant précisément la transformation géométrique de chaque maille depuis l'élément fini de référence. De plus, il faut accéder aux positions réelles des points du maillage original. Cela signifie la lecture d'un vecteur de deux/trois valeurs pour chaque point de la maille, et donc en conséquence plus de lectures de données et plus d'opérations. Traiter ces maillages est plus lent. Il existe la possibilité de mailler une partie des sous-domaines avec des éléments parallélépipèdes afin d'utiliser le noyau de

calcul spécifique plus rapide, et les sous-domaines plus complexes avec un noyau généralisé plus lent. En tout cas, beaucoup de maillages représentatifs du CND peuvent appliquer le noyau spécifique et profiter de plus des accélérations.

8.3.2 Choix de la stratégie pour intégration dans CIVA

Cette thèse a présenté plusieurs transformations différentes classées en types d'implémentations : vectorisation sur CPU, SIMDisation sur CPU et parallélisation sur GPU. Le choix d'implémenter l'une ou l'autre va dépendre de la généricité et la maintenabilité, au-delà de la performance.

Les implémentations n'ont pas le même niveau de généralité quand on parle des différents ordres. Par exemple, les algorithmes du code SIMDisé sont profondément liés à la taille du registre SIMD. L'algorithme gère des éléments 2D et 3D dont la quantité des points du premier axe est multiple de 4, c'est-à-dire d'ordre 3 ou 7 (si la taille de registre est au moins 8, l'extension AVX ou supérieure). La complexité des algorithmes quand cette condition n'est pas remplie est très haute et n'a pas été étudiée pendant cette thèse. Les codes en GPU souffrent d'une manière similaire. La version utilisant la mémoire partagée s'appuie dans une lecture des éléments par un *warp*. Le cas présenté va lire 2 éléments de taille 16. Toutes les tailles des éléments qui ne sont pas diviseurs ou multiples de 32 vont avoir des *threads* sous-utilisés. Cela peut diminuer la performance.

La maintenabilité est un facteur à considérer, puisqu'au fur et à mesure que les architectures avancent le code doit être adapté. Le code vectorisé a besoin d'un simple changement des options de compilation. C'est le compilateur qui effectue les changements. Le code SIMDisé doit s'adapter aux nouvelles extensions : le cardinal SIMD augmente (voir le Tableau 5.8) et de nouvelles instructions apparaissent (FMA en AVX2, *gather* et *scatter* en AVX512). Les algorithmes doivent évoluer avec l'extension SIMD. De la même manière, les algorithmes GPU évoluent lorsque l'architecture évolue. Par exemple, l'architecture Volta inclut les nouveaux cœurs CUDA Tensor dont l'algorithme pourrait bénéficier.

8.3.3 Générateur de code

Deux aspects se révèlent problématiques concernant le noyau de calcul quand on veut appliquer les résultats des travaux de cette thèse : la grande quantité des fichiers à gérer et leur longueur. En effet, le code est spécifique pour chaque taille d'un élément et selon le type d'onde (acoustique ou élastique) traité, donc il faut des fonctions différentes pour chaque cas. Un fichier a été créé pour chaque fonction, pour éviter lors de la compilation la création d'objets monstrueux. En plus, l'écriture de centaines, voire milliers, de lignes de code par fichier est fastidieux, et sa maintenance est presque impossible. Les changements qui pourraient se présenter doivent être apportés à tous les fichiers : un générateur de code est indispensable.

La quantité des fichiers qui doivent être générés est assez importante et montre la quantité de travail. Traiter une onde acoustique ou élastique génère 2 fichiers. Mais l'enjeu principal réside dans les variations d'ordre. Les cas dont on a discuté pendant ces travaux présument que l'ordre pour chaque axe est invariable. Cela n'est toujours

pas forcément le cas. Générer le code pour un éventail d'ordres entre 2 et 7 en 2D où chaque axe peut avoir un ordre différent génère 36 options. En 3D, 216 cas sont possibles. Étant donné qu'il y avait déjà 2 options possibles pour chaque dimension, un total de $36 * 2 + 216 * 2 = 504$ cas, c'est-à-dire, fichiers, sont nécessaires.

L'inclusion de cette énorme quantité des fichiers dans CIVA-SFEM pose un dernier problème. Ils ne peuvent pas être ajoutés simplement au projet au risque de le rendre inutilisable à cause de sa taille. Ils vont être utilisés comme une bibliothèque externe, avec une API détaillée. Cette bibliothèque ne sera compilée que quand un changement propre se produira, plutôt qu'à chaque fois que CIVA-SFEM, en développement continu, sera compilé. L'inclusion de ces travaux nécessite donc de modifier la chaîne de compilation de CIVA.

Bibliographie

Asahi, Y., Latu, G., Ina, T., Idomura, Y., Grandgirard, V., et Garbet, X. (2017). Optimization of Fusion Kernels on Accelerators with Indirect or Strided Memory Access Patterns. *IEEE Transactions on Parallel and Distributed Systems*, 28(7) :1974–1988. 94

Augonnet, C., Thibault, S., Namyst, R., et Wacrenier, P.-A. (2011). StarPU - a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198. 69

Balageas, D., Fritzen, C.-P., et Güemes, A., editors (2006). *Structural health monitoring*. ISTE, London ; Newport Beach, CA. 30

Balevic, A., Rockstroh, L., Tausendfreund, A., Patzelt, S., Goch, G., et Simon, S. (2008). Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs. pages 327–334. IEEE. 93

Banaś, K., Krużel, F., et Bielański, J. (2016). Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering*, 305 :827–848. 93

Bathe, K.-J. (1986). Finite elements in CAD and ADINA. *Nuclear Engineering and Design*, 98(1) :57–67. 83

Bell, N. et Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. page 1. ACM Press. 94

Ben Belgacem, F. et Maday, Y. (1994). A spectral element methodology tuned to parallel implementations. *Computer Methods in Applied Mechanics and Engineering*, 116(1) :59–67. 92

Brodman, J., Babokin, D., Filippov, I., et Tu, P. (2014). Writing scalable SIMD programs with ISPC. pages 25–32. ACM Press. 66

Carrascal, C., Imperiale, A., Rougeron, G., Bergeaud, V., et Lacassagne, L. (2018). A Fast Implementation of a Spectral Finite Elements Method on CPU and GPU Applied to Ultrasound Propagation. *Advances in Parallel Computing*, pages 339–348.

Carrascal, C., Lacassagne, L., Chouh, H., Jubertie, S., Bergeaud, V., et Imperiale, A. (2019). Data layouts and threading strategies for Spectral Finite Elements for SIMD multi-core processors and GPUs. *The International Journal of High Performance Computing Applications*.

Carter Edwards, H., Trott, C. R., et Sunderland, D. (2014). Kokkos - Enabling many-core performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12) :3202–3216. 69

Cassagne, A., Aumage, O., Barthou, D., Leroux, C., et Jégo, C. (2018). MIPP - a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5g Standard. pages 1–8. ACM Press. 66

Cecka, C., Lew, A. J., et Darve, E. (2011). Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5) :640–669. 123

CentaurSoft (2018). CENTAUR Software. 83

Choi, W., Skelton, E., Lowe, M. J. S., et Craster, R. V. (2014). Development of efficient hybrid finite element modelling for simulation of ultrasonic Non-Destructive Evaluation. 37

Cohen, G., Joly, P., Roberts, J., et Tordjman, N. (2001). Higher Order Triangular Finite Elements with Mass Lumping for the Wave Equation. *SIAM Journal on Numerical Analysis*, 38(6) :2047–2078. 81

Cohen, G., Joly, P., et Tordjman, N. (1994). Higher-order finite elements with mass-lumping for the 1d wave equation. *Finite Elements in Analysis and Design*, 16(3) :329–336. 79

Crivellini, A. et Franciolini, M. (2018). On the Implementation of OpenMP and Hybrid MPI/OpenMP Parallelization Strategies for an Explicit DG Solver. *Advances in Parallel Computing*, pages 527–536. 108

Curie, P. et Curie, J. (1880). Développement, par pression, de l'électricité polaire dans les cristaux hémièdres à faces inclinées. *Comptes Rendus de l'Académie des Sciences*, pages 294–295. 27

Darmon, M., Chatillon, S., Mahaut, S., Calmon, P., Fradkin, L. J., et Zernov, V. (2011). Recent advances in semi-analytical scattering models for NDT simulation. *Journal of Physics : Conference Series*, 269(1) :012013. 30

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., et Yelick, K. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. pages 1–12. IEEE. 154

Dollinger, J. et Loechner, V. (2015). CPU+GPU Load Balance Guided by Execution Time Prediction. 154

Dursun, H., Kunaseth, M., Nomura, K.-i., Chame, J., Lucas, R. F., Chen, C., Hall, M., Kalia, R. K., Nakano, A., et Vashishta, P. (2012). Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters. *The Journal of Supercomputing*, 62(2) :946–966. 93

Durufle, M., Grob, P., et Joly, P. (2009). Influence of Gauss and Gauss-Lobatto quadrature rules on the accuracy of a quadrilateral finite element method in the time domain. *Numerical Methods for Partial Differential Equations*, 25(3) :526–551. 79

Dziekonski, A., Lamecki, A., et Mrozowski, M. (2016). GPU-accelerated finite element method. pages 1–2. IEEE. 93

Dziekonski, A., Sypek, P., Lamecki, A., et Mrozowski, M. (2014). GPU-Accelerated Finite-Element Matrix Generation for Lossless, Lossy, and Tensor Media [EM Programmer's Notebook]. *IEEE Antennas and Propagation Magazine*, 56(5) :186–197. 93

Estérie, P., Falcou, J., Gaunard, M., et Lapresté, J.-T. (2014). Boost.SIMD - generic programming for portable SIMDization. pages 1–8. ACM Press. 66

Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960. 47

Fog, A. (2017). C++ vector class library. 66

Geuzaine, C. et Remacle, J.-F. (2009). Gmsh - A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11) :1309–1331. 83, 144

Hadade, I. et di Mare, L. (2016). Modern multicore and manycore architectures : Modelling, optimisation and benchmarking a multiblock CFD code. *Computer Physics Communications*, 205 :32–47. 93

Hoole, S. R. H. et Sivasuthan, S. (2014). GPU computations for finite element optimization : A review of the methodology and problems for study. pages 1–6. IEEE. 93

Huan-Ting Meng, Bao-Lin Nie, Wong, S., Macon, C., et Jian-Ming Jin (2014). GPU accelerated finite-element computation for electromagnetic analysis. *IEEE Antennas and Propagation Magazine*, 56(2) :39–62. 93

Huthwaite, P. (2014). Accelerated finite element elastodynamic simulations using the GPU. *Journal of Computational Physics*, 257, Part A :687–707. 27, 92, 119, 124

Imperiale, A., Chatillon, S., Darmon, M., Leymarie, N., et Demaldent, E. (2018). UT simulation using a fully automated 3d hybrid model : Application to planar backwall breaking defects inspection. page 050004. 15, 37

Jayaraj, J., Lin, P.-H., Woodward, P. R., et Yew, P.-C. (2014). CFD Builder : A Library Builder for Computational Fluid Dynamics. pages 1029–1038. IEEE. 93

Kanapickas, P. (2018). Portable header-only zero-overhead C++ low level SIMD library. 66

Karakasis, V., Goumas, G., et Koziris, N. (2009). Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. pages 356–364. IEEE. 124

Karpiński, P. et McDonald, J. (2017). A high-performance portable abstract interface for explicit SIMD vectorization. pages 21–28. ACM Press. 66

Kirk, B. S., Peterson, J. W., Stogner, R. H., et Carey, G. F. (2006). libMesh - a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4) :237–254. 83

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., et Fasih, A. (2012). PyCUDA and PyOpenCL - A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3) :157–174. 94

Komatitsch, D., Göddeke, D., Erlebacher, G., et Michéa, D. (2010). Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science - Research and Development*, 25(1-2) :75–82. 92, 119, 124

Komatitsch, D., Michéa, D., et Erlebacher, G. (2009). Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5) :451–460. 92, 124

Komatitsch, D. et Vilotte, J.-P. (1998). The Spectral Element method : an efficient tool to simulate the seismic response of 2d and 3d geological structures. *ResearchGate*, 88(2) :368–392. 92

Komatitsch, D., Vilotte, J.-P., Vai, R., Castillo-Covarrubias, J. M., et Sánchez-Sesma, F. J. (1999). The spectral element method for elastic wave equations—application to 2-D and 3-D seismic problems. *International Journal for Numerical Methods in Engineering*, 45(9) :1139–1164. 92

Kretz, M. et Lindenstruth, V. (2012). Vc - A C++ library for explicit vectorization. *Software : Practice and Experience*, 42(11) :1409–1430. 66

Kun, L. (2009). Graphics Processor Unit (GPU) acceleration of Time-Domain Finite Element Method (TD-FEM) algorithm. pages 928–931. IEEE. 93

Lacassagne, L., Etiemble, D., Hassan-Zahraee, A., Dominguez, A., et Vezolle, P. (2014). High Level Transforms for SIMD and Low-level Computer Vision Algorithms. In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 49–56, New York, NY, USA. ACM. 104

Lambert, J. (2015). *Parallélisation de simulations interactives de champs ultrasonores pour le contrôle non destructif*. PhD thesis, Paris 11. 28, I

Lang, J. et Rünger, G. (2013). Dynamic Distribution of Workload between CPU and GPU for a Parallel Conjugate Gradient Method in an Adaptive FEM. *Procedia Computer Science*, 18 :299–308. 154

Larsen, E. S. et McAllister, D. (2001). Fast matrix multiplies using graphics hardware. pages 55–55. ACM Press. 55

Leisa, R., Hack, S., et Wald, I. (2012). Extending a C-like language for portable SIMD programming. page 65. ACM Press. 66

Leisa, R., Haffner, I., et Hack, S. (2014). Sierra - a SIMD extension for C++. In *Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing - WPMVP '14*, pages 17–24, Orlando, Florida, USA. ACM Press. 66

Levesque, J. M., Sankaran, R., et Grout, R. (2012). Hybridizing S3d into an Exascale application using OpenACC : An approach for moving to multi-petaflops and beyond. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, UT. IEEE. 68

Leymarie, N., Calmon, P., Fouquet, T., et Schumm, A. (2006). *Semi-Analytical-Fem Hybrid Modeling of Ultrasonic Defect Responses*. 15, 37

Livesey, M., Costen, F., et Xiaoling Yang (2012). Performance of Streaming SIMD Extensions Instructions for the FDTD Computation. *IEEE Antennas and Propagation Magazine*, 54(3) :160–168. 93

Ljungkvist, K. (2017). *Finite Element Computations on Multicore and Graphics Processors*. PhD Thesis, Uppsala University, Division of Scientific Computing, Computational Science. 93

Markall, G. R., Ham, D. A., et Kelly, P. H. (2010). Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1) :1815–1823. 92

Markall, G. R., Slemmer, A., Ham, D. A., Kelly, P. H. J., Cantwell, C. D., et Sherwin, S. J. (2013). Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1) :80–97. 88

Masliah, I., Abdelfattah, A., Haidar, A., Tomov, S., Baboulin, M., Falcou, J., et Dongarra, J. (2016). High-Performance Matrix-Matrix Multiplications of Very Small Matrices. In Dutot, P.-F. et Trystram, D., editors, *Euro-Par 2016 : Parallel Processing*, volume 9833, pages 659–671. Springer International Publishing, Cham. 94

McCalpin, J. (1995). STREAM - Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25. 45

Mewes, K., Piedmonte, C., Frishberg, M., et Cohen, B. (2014). Whitepaper, IBM POWER8 - Performance and Cost Advantages in Business Intelligence Systems. Technical report, The Edison Group, New York. 45

Moore, G. E. (2006). Lithography and the future of Moore's Law. *IEEE Solid-State Circuits Society Newsletter*, 11(3) :37–42. 42

NEC-Corporation (2017). NEC SX-Aurora TSUBASA Brochure. Technical report. 47

NVIDIA-Corporation (2009). Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture : Fermi. Technical report. 57

NVIDIA-Corporation (2012). Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture : Kepler GK110. The Fastest, Most Efficient HPC Architecture Ever Built. Technical report. 58

NVIDIA-Corporation (2014). Whitepaper, NVIDIA GeForce GTX 980. Featuring Maxwell, The Most Advanced GPU Ever Made. Technical report. 58

NVIDIA-Corporation (2016). Whitepaper, NVIDIA Tesla P100, The Most Advanced Datacenter Accelerator Ever Built. Featuring Pascal GP100, the World's Fastest GPU. Technical report. 59

NVIDIA-Corporation (2017). Whitepaper, NVIDIA Tesla V100 GPU Architecture. The World's Most Advanced Data Center GPU. Technical report. 60

Oberkampf, W. L. et Trucano, T. G. (2002). Verification and validation in computational fluid dynamics. *Progress in Aerospace Sciences*, 38(3) :209–272. 95

Papadrakakis, M., Stavroulakis, G., et Karatarakis, A. (2011). A new era in scientific computing : Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13–16) :1490–1508. 154

Peng, L., Seymour, R., Ken-ichi Nomura, Kalia, R. K., Nakano, A., Vashishta, P., Loddoch, A., Netzband, M., Volz, W. R., et Wong, C. C. (2009). High-order stencil computations on multicore clusters. pages 1–11. IEEE. 93

Reguly, I. R. et Giles, M. (2012). Efficient sparse matrix-vector multiplication on cache-based GPUs. pages 1–12. IEEE. 94

Reguly, I. Z. et Giles, M. B. (2015). Finite Element Algorithms and Data Structures on Graphical Processing Units. *International Journal of Parallel Programming*, 43(2) :203–239. 120

Rossi, F. V., So, P. P., Fichtner, N., et Russer, P. (2008). Massively parallel two-dimensional TLM algorithm on graphics processing units. pages 153–156. IEEE. 93

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., et Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. page 73. ACM Press. 94

Rytter, A. (1993). *Vibrational Based Inspection of Civil Engineering Structures*. PhD Thesis, Aalborg University, Denmark. 30

Sayas, F. (2008). *A gentle introduction to the Finite Element Method*. 27

Scheidegger, C. E., Fleishman, S., et Silva, C. T. (2005). Triangulating Point Set Surfaces with Bounded Error. In *Symposium on Geometry Processing*. 83

Schlesinger, S. (1979). Terminology for model credibility. *SIMULATION*, 32(3) :103–104. 95, III

Schöberl, J., Arnold, A., Erb, J., Melenk, J. M., et Wihler, T. P. (2014). C++11 implementation of finite elements in NGSolve. Technical report. 93

Shepherd, J. F. et Johnson, C. R. (2008). Hexahedral mesh generation constraints. *Engineering with Computers*, 24(3) :195–213. 83

Software, S. P. (2014). Whitepaper - Femap assembly modeling. Enabling product teams to quickly and easily model the interaction between components in a complex assembly. 83

Stone, J. E., Gohara, D., et Shi, G. (2010). OpenCL - A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3) :66–73. 68

Sutter, H. (2005). The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3) :202–210. 42, I

Tromp, J., Komattisch, D., et Liu, Q. (2008). Spectral-element and adjoint methods in seismology. *Communications in Computational Physics*, 3(1) :1–32. 124, 143

Volkov, V. (2010). Better Performance at Lower Occupancy. *NVIDIA*. 121

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., et Demmel, J. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. page 1. ACM Press. 94

Wojcik, G., Vaughan, D., Abboud, N., et Mould, J. (1993). Electromechanical modeling using explicit time-domain finite elements. In *Proceedings of IEEE Ultrasonics Symposium*, pages 1107–1112, Baltimore, MD, USA. IEEE. 28

Yurtesen, E., Ropo, M., Aspnas, M., et Westerholm, J. (2011). SSE Vectorized and GPU Implementations of Arakawa's Formula for Numerical Integration of Equations of Fluid Motion. pages 341–348. IEEE. 93

Zhang, J. et Shen, D. (2013). GPU-Based Implementation of Finite Element Method for Elasticity Using CUDA. pages 1003–1008. IEEE. 93

Zhou, X. J. et Derek Kim, H. Y. (2016). E-beam tool-to-tool matching methodology : DI : Defect inspection and reduction. In *2016 27th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*, pages 289–293, Saratoga Springs, NY, USA. IEEE. 28

Zumbusch, G. (2012). Tuning a Finite Difference Computation for Parallel Vector Processors. pages 63–70. IEEE. 92

Table des figures

Liste des tableaux

Liste des algorithmes