Parallélisation de simulations interactives de champs ultrasonores pour le contrôle non destructif

Jason Lambert

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE 442
Sciences et Technologies de l'Information, des Télécommunications et des Systèmes

Laboratoire d'Informatique (CEA-LIST/DISC)
Laboratoire de Recherche en Informatique (Université Paris-Sud)

THÈSE

Physique
Adéquation Algorithme Architecture

par

Jason Lambert

Parallélisation de simulations interactives de champs ultrasonores pour le contrôle non destructif

Date de soutenance : 03/07/2015

Composition du jury :

Directeur de thèse :	Lionel LACASSAGNE	Maître de Conférences (LRI, Université Paris Sud)
Rapporteurs :	Didier CASSEREAU	Maître de conférences (ESA, ESPCI ParisTech)
	Dominique HOUZET	Professeur (GIPSA-LAB, Genoble INP)
Examinateurs :	Alain MERIGOT	Professeur (IEF, Université Paris Sud)
	Michel PAINDAVOINE	Professeur (LEAD, Université de Bourgogne)
Encadrant CEA :	Gilles ROUGERON	Ingénieur chercheur (LDI, CEA-LIST/DISC)

Remerciements

Une thèse de doctorat, c'est l'accomplissement de trois ans de travail d'un doctorant, qui viennent après déjà un certain nombre d'années d'études, pour ma part, un diplôme d'ingénieur de l'ENSIIE. Mais ce travail n'aurait pas été rendu possible sans l'aide de nombreuses personnes qui ont participé, de près où de loin, à la réussite de cette aventure.

Je tiens à exprimer ici toute ma reconnaissance à Gilles Rougeron et à Lionel Lacassagne pour m'avoir fait confiance pour ces travaux. Leur bienveillance m'a accompagné et leurs conseils et leur soutien ont été plus que précieux pour progresser significativement tout au long de l'élaboration de cette thèse, pour développer mon esprit critique et aiguiser mon sens scientifique.

Mes remerciements vont également à Messieurs Didier Cassereau et Dominique Houzet qui ont accepté de rapporter cette thèse malgré le temps fort court qui leur a été donné. Je remercie pareillement Messieurs Alain Merigot et Michel Paindavoine d'avoir accepté de faire partie du jury et de l'attention portée à mes travaux.

Je remercie tout particulièrement Stéphane Le Berre de son intérêt pour mes travaux et pour avoir accepté de lancer un tel projet dans son laboratoire.

Je remercie également Sylvain Chatillon et Vincent Bergeaud pour avoir suivi de près ces travaux et pour avoir pris le temps de me conseiller sur leur réalisation.

Je n'oublie pas toute l'équipe du DISC qui m'a accueilli.

Je salue chaleureusement mes *co-bureaux* qui m'ont supporté (et réciproquement) : Clémence, Serena, Longhui, Hamza et Loïc ! Je remercie également toute la bande, l'ancienne équipe "passerelle", pour les moments passés ensemble à plaisanter/râler/décompresser : Thomas, Nicolas, Bastien, Blandine, Tom, Marouane, Adrien, Eudrey, Matthieu, Audrey, Simon, Léonard, Eduardo, Antoine, Arnaud, Mathilde.

Je salue également les amis qui ont subi parfois ma mauvaise humeur et pour qui je n'ai pas été toujours disponible (surtout pendant la rédaction) mais qui ont poussé l'abnégation jusqu'à assister à ma soutenance quand bien même, leur appétence pour le sujet n'était pas très grande !

Enfin, je remercie mes parents de m'avoir soutenu et encouragé dans cette voie, qui s'est peu à peu imposée à moi, depuis mes jeunes débuts sur le Spectravideo SV318 familial (son processeur Z80 à 3.6 MHz accompagné de 32 Ko de RAM !) jusqu'à aujourd'hui.

Merci à toutes celles et ceux qui m'ont encouragé tout au long de ce projet.

Sommaire

Table des figures

Liste des tableaux

Tables des programmes

Chapitre

1 Introduction

- TODO : tableau 5.32 : mettre CIVA11 en secondes, ajouter un séparateur de 3 chiffres, retirer les chiffres apres les virgules inutiles
- TODO : CUDA / nb reg par bloc fermi / 512 ?
- TODO : CUDA : 7.12 diviser la freq de la mémoire pour rester coherent avec les doc cuda
- TODO : finir les annexes
- TODO : page de garde

Les travaux présentés dans le cadre de cette thèse s'inscrivent dans le domaine général du contrôle non destructif (CND). Il s'agit de l'ensemble des méthodes d'inspection et de contrôle d'une pièce mécanique sans dégradation des propriétés. Le CND est utilisé dans un large champ applicatif, de la fabrication à la maintenance, dans un ensemble de domaines industriels variés : énergie, transports, construction... Les techniques de CND sont très nombreuses, et parmi les plus courantes on retrouve le contrôle par ultrasons, l'usage de rayons X ou des méthodes électromagnétiques (courants de Foucault). Le CND est utilisé pour détecter les défauts et les caractériser (par ordre de difficulté : localisation, identification, dimensionnement). Il est possible de définir trois pans du CND industriel : l'inspection en elle même (acquisition d'un contrôle sur une pièce), la reconstruction (réutilisation de données acquises en entrées d'un modèle physique afin d'en faciliter l'analyse) et la simulation (obtention de données de contrôle purement modélisées). Plus spécifiquement, la simulation est utilisée à plusieurs niveaux : en amont de l'acquisition pour la mise au point du contrôle (choix du capteur, positionnement...), en aval de l'acquisition (incorporée aux algorithmes de reconstruction ou utilisée pour aider à la compréhension des phénomènes physiques complexes). Elle peut notamment être utilisée lors de la formation des opérateurs.

La simulation et la reconstruction sont réalisées au moyen d'ordinateurs, le plus souvent généralistes. Dans ce contexte, le Département d'Imagerie et Simulation pour le Contrôle (DISC) du CEA-LIST développe la plateforme logicielle CIVA pour l'analyse, la conception et la définition de configurations CND. Cette plateforme dispose de modèles multi-techniques complets. Dans le cadre du CND ultrasonore, les temps de calcul des simulations de contrôle peuvent atteindre typiquement quelques dizaines de secondes à plusieurs heures.

Depuis une dizaine d'années, on observe dans l'informatique généraliste un changement de paradigme afin de permettre aux matériels de continuer à suivre, voire d'excéder, les prédictions de la loi de Moore[Moo95]. La puissance des matériels informatiques continue d'augmenter de manière exponentielle mais leur fréquence de fonctionnement n'augmente plus et suit désormais un plateau (en raisons de phénomènes physiques). Aussi, la tendance n'est plus à l'augmentation de la fréquence de calcul d'une seule unité de calcul mais plutôt à la démultiplication des unités de calcul et des niveaux de parallélisme disponibles[1].

Le contexte d'utilisation standard de CIVA consiste pour l'expert en CND à travailler sur des machines allant d'un PC de bureau à la station de calcul. Trois grandes classes d'architectures parallèles sont ainsi disponibles pour optimiser les performances de calcul des modèles développés :

- le processeur généraliste multicœurs (GPP, *general purpose processor*) ;
- les coprocesseurs généralistes *manycore* (MIC, *many integrated core*) ;
- les cartes graphiques (GPU, *graphical processing units*).

La simulation de champ ultrasonore CIVA, c'est-à-dire la simulation du faisceau d'ondes ultrasonores rayonnées dans la pièce à la manière d'une échographie médicale, utilise un modèle dont le domaine d'application est très étendu (géométrie de la pièce, matériaux et capteurs complexes). Ce code de calcul est développé depuis de nombreuses années et continuellement optimisé. Jusqu'ici, il est parallélisé sur un des niveaux disponibles du GPP (répartition des tâches sur les différentes unités de calcul).

Afin de tirer parti du nouveau paradigme et exploiter la puissance théorique impressionnante (de l'ordre du Téraflops, 10^{12} opérations par seconde) de ces architectures parallèles, les algorithmes existants doivent être portés. Pour autant l'exploitation réelle de leur puissance de calcul impose des contraintes qui conduisent à les repenser et à les adapter. De plus, ces architectures étant assez spécifiques, elles nécessitent, pour en tirer parti entièrement, une spécialisation des codes de calculs pour un même algorithme.

Les travaux réalisés au cours de cette thèse ont visé le développement d'une simulation de champ ultrasonore très rapide sur ces trois classes d'architectures. Outre l'accélération des temps de calcul dans l'absolu, un usage interactif des simulations de champ a pu être atteint. Ainsi lorsque l'utilisateur prépare sa configuration, la modification d'un des paramètres provoque une nouvelle simulation de champ dont le résultat est affiché quasi-instantanément. L'obtention de simulations interactives et valides physiquement est, à ce jour, inédite dans le cadre du CND. Cette performance n'a pu être atteinte toutefois qu'au prix d'une restriction du périmètre d'application de la simulation de champ par rapport aux modèles complets de CIVA. D'une part, élargir le champ d'application nécessite de modéliser toujours plus de cas par des ensembles d'équations spécifiques conduisant à optimiser de nouveaux codes. D'autre part, les configurations de contrôles les plus complexes ne peuvent pas être obtenues par des modèles purement analytiques.

La démarche qui a permis d'atteindre ce résultat est ensuite développée :

- Dans le chapitre 2 sont présentés, le domaine du CND, la plateforme CIVA et l'utilisation des simulations de champ dans un contexte industriel.
- Le chapitre 3 passe en revue les architectures parallèles disponibles sur une machine généraliste et présente, pour chacune, ses spécificités matérielles ainsi que les outils logiciels existants pour en tirer parti (langages de programmation et bibliothèques optimisées). Un

[1] Il se dit désormais "*The free lunch is over*"[Sut05] concernant le développement logiciel.

état de l'art de l'usage de ces architectures dans le cadre de la simulation ultrasonore en CND est également présenté.

- Le chapitre 4 expose le modèle restreint mais régulier dérivé de celui de CIVA permettant d'obtenir une simulation rapide de champ. Une implémentation de référence de ce modèle est détaillée afin permettre de le valider par rapport aux résultats CIVA. A partir de cette implémentation, une première analyse du comportement est effectuée et ses limitations sont discutées.
- Ensuite, trois chapitres, chacun correspondant à une des trois classes d'architectures présentées, détaillent les optimisations effectuées avant de revenir sur l'adéquation de ces optimisations au moyen d'une analyse fine des performances obtenues. Les chapitres 5, 6 et 7 présentent respectivement les optimisations sur GPP, MIC et GPU.
- Enfin, des conclusions sur l'adéquation de ces architectures avec la simulation de champ rapide et éventuellement interactive en contexte industriel sont dressées. A partir de ces premiers résultats, des perspectives sur ces travaux et leur industrialisation sont également présentées.

Chapitre

2 La simulation en contrôle non destructif

Dans le domaine du contrôle non destructif, l'usage de la simulation tend à se généraliser pour la mise au point de nouveaux contrôles ou 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 simulations interactives sont envisagées.

Il rappelle le contexte du contrôle non destructif industriel ainsi que les différentes méthodes d'inspection utilisées. Dans la mesure où les travaux réalisés ne concernent que la technique de contrôle par ultrasons, seul le modèle de propagation d'une onde ultrasonore est traité. Son usage dans le cadre du contrôle est ensuite abordé à l'aide de quelques exemples. Enfin, la plateforme CIVA est présentée ; en particulier ses fonctionnalités en matière de contrôle ultrasonore.

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 avant d'atteindre la rupture. De même, dès la production, caractériser ces composants mécaniques permet de s'assurer de l'intégrité des ouvrages réalisés.

De la 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 contrôle non destructif[1] 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 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 terme de taille critique, pour les défauts présents. Le domaine du contrôle non destructif 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 contrôle non destructif 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 :

[1] D'autres procédés industriels existent, tels que le contrôle destructif. Il s'agit alors, en se basant sur des critères de validité statistique, d'inspecter quelques échantillons pour décider de la validité d'un lot de pièces. Son usage a lieu dans des industries qui ne sont pas dans un contexte de "zéro défaut" telles que les utilisatrices du CND.

- 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.
- 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 essieux sont régulièrement contrôlés. Tous les ans, la SNCF inspecte près de 25000 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...).

En 2005, le marché du contrôle non destructif représentait un volume global de l'ordre du milliard d'euros[Lhe05]. Les analystes de *Lucintel* ont estimé une croissance du marché d'ici 2018 permettant d'atteindre 2,058 milliards de dollars US [Luc13].

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 terme de cadence, de durée et d'automatisation.

2.1.2.1 Méthodes "directes"

La méthode la plus simple pour inspecter une pièce consiste à l'**examiner visuellement** (avec une source de lumière adaptée) afin de constater ou non la présence de défaut. La lumière utilisée et observée peut être dans le domaine du visible ou dans le domaine invisible, captée avec des outils adaptés. Cette méthode permet uniquement d'identifier la présence de défaut débouchant sur une pièce.

L'un des premiers procédés assimilable à du contrôle non destructif dans le cadre de la production peut être ce que l'on désigne comme **le geste du potier**[Lhe05]. Il s'agit de frapper doucement une pièce de poterie pour la faire résonner afin d'écouter les vibrations produites pour vérifier si celles-ci sont conformes à celles attendues par le potier. Cette méthode manuelle s'apparente à l'ensemble des méthodes dites vibratoires : une vibration est envoyée dans la pièce à contrôler puis un accéléromètre mesure la réponse vibratoire obtenue à travers la pièce. En analysant le spectre du signal reçu, il est possible par comparaison avec le spectre d'une pièce saine de référence, de se prononcer sur la salubrité du sujet de l'inspection. Ces méthodes vibratoires sont globales et peu coûteuses, mais ne peuvent s'appliquer qu'à des pièces de géométrie très simple, elles ne permettent que la détection de la présence d'un défaut suffisamment grand par rapport à la longueur d'onde de la perturbation. Industriellement, cette méthode est facilement automatisable et permet des cadences élevées pour un coût relativement faible.

2.1.2.2 Premiers procédés

Une méthode, similaire au geste du potier, dite **émission acoustique** consiste simplement à écouter une pièce pour détecter l'apparition d'un défaut. Dans une pièce sous contrainte, le début de la fissuration produit une onde qui se propage dans toute la pièce, un traducteur détecte cette onde et signale l'apparition du défaut. Plusieurs traducteurs, mis en réseau, peuvent permettre de localiser l'endroit où apparait le défaut. Cette méthode est applicable à de grandes structures sous contraintes, notamment les ouvrages d'art de type ponts, barrages... La calibration des traducteurs est essentielle pour détecter les ondes résultantes de la fissuration car celles-ci sont d'amplitude très faible.

Dans le cadre de pièces en matériaux ferromagnétiques, en aimantant la pièce à l'aide d'un électro-aimant, la **magnétoscopie** permet d'orienter un révélateur le long de la direction d'induction imposée. De fines particules, piégées par le champ magnétique perturbé, sont observables à l'œil ou au moyen d'outils spécialisés (loupe, endoscope, lumière ultraviolette...). Cette méthode est cependant limitée aux matériaux ferromagnétiques, et ne permet pas de déterminer la profondeur d'un défaut. Par contre, elle est globale et permet d'inspecter en une opération l'ensemble d'une pièce de manière relativement rapide et peu coûteuse. Elle est utilisable dans un grand nombre de domaines, pour le contrôle de pièces métalliques de géométrie variée. Les progrès récents concernant cette méthode se concentrent sur les têtes d'aimantation afin d'améliorer la détection par rapport à l'orientation du défaut. Cependant, cette méthode n'est pas pour autant aisément automatisable en raison du caractère spécifique de l'inspection et de la nécessité de nettoyer la pièce.

Une autre technique simple à mettre en œuvre est le **ressuage** d'une pièce, pour obtenir une localisation précise de défauts débouchants[Lhe05]. Elle consiste à enduire la pièce bien nettoyée d'un produit pénétrant, souvent un produits pétrolier coloré ou fluorescent, qui pénètre par capillarité à l'intérieur des défauts débouchants. La surface de la pièce est ensuite lavée pour en éliminer l'excès de produit et séchée. Enfin, un révélateur est appliqué pour permettre l'observation d'une tache caractéristique en présence d'un défaut. Cependant, cette méthode ne permet pas d'apprécier la profondeur d'un défaut, et son interprétation nécessite une certaine expérience. Les premières utilisations de cette méthode remontent aux années 1880 dans le domaine des chemins de fer. Elle est souvent utilisée pour le contrôle de produits finis ou en cours d'utilisation, de composition métallique ou minérale. Les principaux utilisateurs de cette technique sont l'industrie aéronautique pour le contrôle des moteurs et l'industrie nucléaire pour le contrôle des soudures sur le circuit primaire des centrales nucléaires.

2.1.2.3 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 le contrôle ultrasonore. Il est abordé ici dans le cadre de la revue des méthodes existantes de contrôle non destructif, mais sera détaillé plus avant.

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. 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 défaut de type fissures qui agissent 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 défauts par rapport au faisceau émis, la sensibilité du contrôle varie et ce dernier doit être optimisé. Les ultrasons se propagent 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. Le premier brevet d'une méthode de contrôle non destructif par ultrasons date de 1931[Mü31]. 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.

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, γ ou neutronique[Lhe05]. 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 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. Par contre, cette méthode est coûteuse en investissement et l'usage d'un rayonnement ionisant impose des contraintes strictes de sécurité pour les personnes. 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 Le contrôle par courants de Foucault

Le **contrôle par courants de Foucault** 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 en opposition de phase sont induits dans deux sections voisines de la pièce par deux bobines ; 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[2]. Elle est particulièrement adaptée aux géométries simples dans lesquelles le comportement électromagnétique est facilement modélisable. L'interprétation des signaux résultats est délicate et requiert un opérateur expert.

Cette méthode est entre autre utilisée pour le contrôle de produits de géométrie cylindrique (barres et tubes). 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. L'automatisation de ces contrôles est rendue possible grâce aux capacités des systèmes de traitement du flux de données[Lhe05].

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 termes de sécurité, de durabilité et de traçabilité. Il peut aussi permettre de vérifier la satisfaction d'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 au niveau de la conception d'une pièce afin d'envisager *a priori* les différentes manières dont ils pourront être effectués 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.

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 sur le contrôle (capteur, défaut ou pièce), il est possible d'analyser en détails 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 sureté).
- **aider à l'interprétation** en générant des résultats correspondant à une situation parfaitement maitrisée, 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.

2.1.3.2 L'apport de l'interactivité

L'interactivité peut se définir d'un grand nombre de manières. Dans le cadre de ce travail, l'interactivité a pour objectif de permettre à l'opérateur de faire varier un paramètre et voir

[2]L'effet de peau est un phénomène électromagnétique pour lequel, à une fréquence élevée, le courant tend à se propager en surface de ses conducteurs

se transformer les résultats de la simulation de manière fluide. On peut ainsi fixer un objectif de simulation réalisée en un temps de calcul de l'ordre de la persistance rétinienne (au sens cinématographique), c'est à dire 1/25 de seconde = $40ms$.

La nature des résultats sera dans cette thèse limitée à des images en 2 dimensions (champ ultrasonores ou B-Scan/Scan sectoriels en écho - *c.f.* la définition de ces notions plus loin au paragraphe 2.3.2). Les performances mesurées seront indiquées en images par seconde, abrégé *fps*. Produire des résultats plus complexes, en N dimensions, nécessiterait de la part de l'utilisateur, dans la majorité des cas, un temps d'analyse qui ne serait pas cohérent avec les besoins d'interactivité.

Dans le cadre d'une simulation, la première utilisation de l'interactivité est de fournir à l'opérateur les moyens de ressentir au plus proche le comportement de son système d'acquisition. Il doit pouvoir corriger rapidement son contrôle pour se focaliser sur la région d'intérêt comme s'il utilisait un capteur portatif. Le code de simulation rapide, permettant l'interactivité, peut, par ailleurs, être utilisé pour de nombreuses applications requérant une étude paramétrique portant sur un ou plusieurs paramètres d'entrée du modèle (analyse d'incertitude, optimisation d'un contrôle, probabilité de détection...).

2.2 Propagation d'une onde ultrasonore

Avant d'exposer en détail la technique du contrôle non destructif ultrasonore, quelques informations sur la physique de la propagation des ondes ultrasonores dans un milieu solide isotrope ainsi que son interaction avec les surfaces de la pièce sont données.

Une onde mécanique est un phénomène de propagation d'une perturbation locale de la matière dans un milieu. Elle est issue d'une source qui impose au milieu une excitation. On parle d'ultrasons pour les ondes mécaniques de fréquences supérieures à 20 kHz ; dans l'air elles produisent un son dont la fréquence est trop élevée pour être audible par l'oreille humaine. Dans le cadre du contrôle non destructif, les signaux employés sont de types impulsionnels avec une oscillation amortie de fréquence centrale fc.

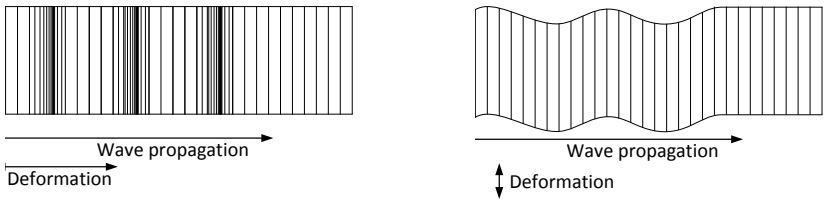
2.2.1 Modèle de propagation dans un milieu isotrope

Ce type d'onde n'entraîne pas, dans un milieu, de déplacement global de celui-ci mais un déplacement temporaire de la matière.

Les caractéristiques du milieu de propagation influent sur la propagation de l'onde. Dans un milieu isotrope sans atténuation, sa fréquence reste constante : la vitesse de l'onde est définie par $v = d/t$ où v est la célérité de l'onde dans le milieu (en $m \cdot s^{-1}$), d la distance parcourue (en m) et t le temps de parcours (en s). A partir de la vitesse de propagation de l'onde et de sa fréquence, on peut définir la longueur d'onde λ comme étant la distance parcourue au cours d'une période T ($T = 1/f$) : $\lambda = v \times T = v/f$.

Les ondes mécaniques transportent de l'énergie qui se propage dans la même direction que l'onde : la perturbation du milieu se retrouve identique, un peu plus loin un peu plus tard, à la condition que la dimension du milieu soit grande devant la longueur d'onde.

En fonction de la direction dans laquelle a lieu l'oscillation de matière par rapport à la direction de propagation de l'onde, il est possible de définir plusieurs types d'onde. La figure 2.1 illustre les deux types d'onde plane rencontrés (ondes longitudinales et transverses). On parle respectivement d'ondes de mode L et T. Dans un milieu, ces ondes n'ont pas la même vitesse de déplacement.



(a) Onde Longitudinale (b) Onde Transversale

FIGURE 2.1 – Propagation d'une onde mécanique dans un milieu isotrope

2.2.2 Interaction d'une onde avec une interface

Par analogie avec l'optique, il est possible de décrire la propagation d'une onde mécanique comme un "rayon sonore" obéissant aux mêmes lois que la lumière, selon le principe d'optique géométrique, à la condition que la longueur d'onde soit faible face aux dimensions des surfaces rencontrées. Au niveau d'une interface, l'onde se comporte de la même manière qu'un rayon optique, par réflexion ou réfraction selon les lois de Snell-Descartes :

$$\frac{v_1}{v_2} = \frac{sin(\theta_1)}{sin(\theta_2)} \tag{2.1}$$

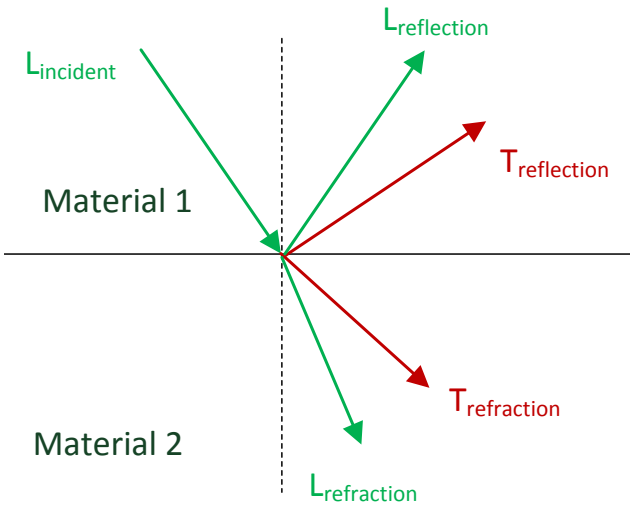


FIGURE 2.2 – Réfraction et Réflexion potentielles pour une onde incidente L (en vert les ondes L, en rouge les ondes T)

Lorsqu'un rayon incident rencontre une surface plane, interface entre deux milieux, les interactions ont lieu dans le plan d'incidence. Ce plan est le plan contenant le rayon incident et la normale locale à la surface. A l'incidence, des phénomènes de réflexion et de réfraction

ont généralement lieu. En fonction de l'angle d'incidence, certains sont plus ou moins atténués. L'amplitude en déplacement de ces phénomènes est donnée par les coefficients de Fresnel (dont le calcul est donné en Annexe A). De plus, des phénomènes de conversion de mode ont lieu : l'onde incidente produit des ondes de mode L et T suivant l'angle d'incidence et les vitesses de propagation des milieux. La figure 2.2 illustre ce phénomène dans le cas général : une onde L incidente crée des réflexions en modes L et T et des ondes réfractées de modes L et T.

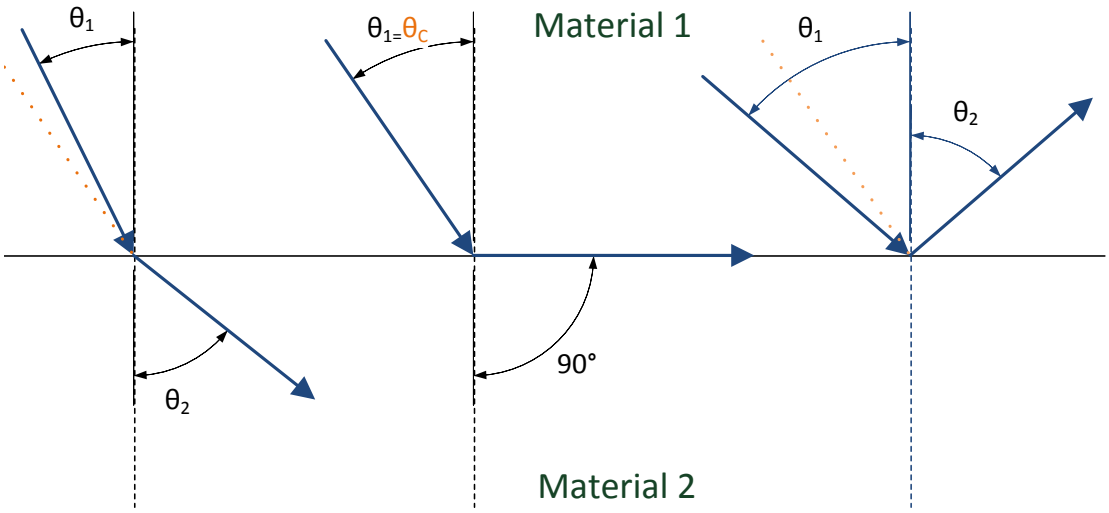


FIGURE 2.3 – Comportements possibles de l'onde à l'interface entre deux milieux (loi de Snell-Descartes) : de gauche à droite, transmission, angle critique et réflexion

Pour un mode d'incidence donné (L ou T), à l'interface entre deux milieux, plusieurs comportements sont possibles conformément à la loi de Snell-Descartes, en fonction de la vitesse de propagation de l'onde étudiée dans le second milieu.

- Dans le cas général, la **transmission** a lieu avec ou sans conversion de mode. La loi de Snell-Descartes, donnée à l'équation 2.1, exprime la réfraction du faisceau à travers cette interface, en fonction de la vitesse de l'onde dans chacun des milieux, comme illustré par la figure 2.3 (schéma de gauche).

- Lorsque l'onde incidente atteint la surface avec un certain angle, le rayon réfracté se propage le long de la surface. Lorsque $sin(\theta_2)$ atteint 1, le rayon réfracté se propage le long de l'interface : l'onde est rasante (*c.f.* le schéma central de la figure 2.3). On peut déterminer l'angle θ_C, **angle critique**, tel que

$$\theta_C = arcsin\left(\frac{v_1}{v_2}\right) \tag{2.2}$$

- L'onde incidente peut être réfléchie avec ou sans conversion de mode. La réflexion est illustrée par le schéma de droite sur la figure 2.3. Avec conversion, la **réflexion** se comporte de la même manière que dans le cas général de la transmission. Lorsqu'il n'y a pas de conversion de mode, la réflexion est totale. Au delà de θ_C, le rayon incident n'est pas transmis et la réflexion est totale. Le rayon réfléchi avec un angle $\theta_{Refl} = -\theta_I$.

- En dehors des phénomènes de réflexion et transmission sur les interfaces, on peut citer l'existence de diffractions apparaissant le long des arêtes vives de la pièce ou des défauts.

En fonction de l'angle de l'onde incidente et des vitesses de propagation, les différents comportement décrits ici peuvent se combiner.

2.3 Présentation du 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 conduire ainsi un expert à conclure sur l'altération des propriétés mécaniques de la pièce.

2.3.1 Principe d'un capteur ultrasonore

Un capteur ultrasonore est un capteur électro-acoustique 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 et/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 (via 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 [CC80].

Un transducteur piézoélectrique 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 via un effet piézoélectrique inverse. A la réception, la vibration ultrasonore crée un champ électrique via 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.

2.3.1.1 Les différents types de transducteurs ultrasonores

Dans le cas classique d'étude de pièces industrielles en acier, dans la mesure où l'air n'est pas un bon milieu d'incidence en raison des fortes réflexions air-acier (vitesse des ondes L dans l'air $340m/s$, vitesse des ondes L et T dans l'acier respectivement $5900m/s$ et $3230m/s$), on utilise un

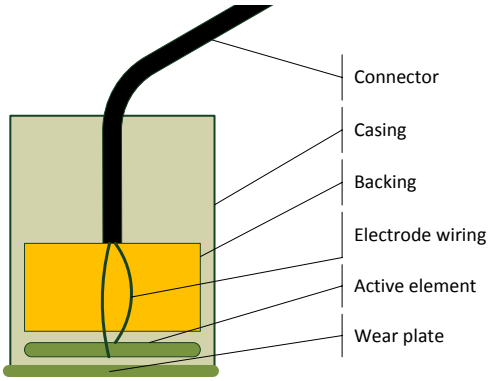


FIGURE 2.4 – Décomposition d'un transducteur piézoélectrique pour l'émission d'ondes ultrasonores

matériau qualifié de **milieu couplant** entre le capteur et la pièce. Les conditions dans lesquelles les contrôles sont ainsi réalisés peuvent résulter de deux principales méthodes d'inspection :

- Par immersion de la pièce inspectée, bien souvent dans l'eau ; l'**inspection en immersion** permet une plus grande souplesse, par exemple, pour orienter le capteur afin de focaliser l'onde émise dans la zone inspectée.
- Lorsque l'immersion est impossible, par exemple lors d'un contrôle sur site, il est nécessaire de travailler directement sur des pièces sans démontage ; un sabot dont le fond est adapté à la pièce inspectée est utilisé afin de procéder à une **inspection au contact**. On utilise en général un gel pour améliorer le contact. Ce type de contrôle s'apparente à une échographie médicale.

A ces types d'inspections sont associés des capteurs spécifiques : des **capteurs en immersion** qui sont placés directement dans le milieu couplant et des **capteurs au contact** qui sont placés sur le sabot. Ces types de capteurs peuvent alors être décomposés en fonction de leurs manière d'émettre et de recevoir des ultrasons.

Les traducteurs mono-élément sont composés d'un seul capteur piézoélectrique global ;

Les capteurs à émission/réception séparées disposent de deux transducteurs, l'un dédié à l'émission et l'autre à la réception. Ces capteurs permettent de ne pas subir les "échos de sabot" qui peuvent venir parasiter les résultats lorsque les fonctions d'émission et de réception sont confondues en un même élément. De plus, leur positionnement permet d'inspecter des zones difficiles d'accès.

Les traducteurs multi-éléments sont composés de multiples éléments indépendants contrôlés électroniquement. Il est possible de contrôler finement quels éléments reçoivent ou émettent, ainsi que de leur appliquer des lois de retards afin d'obtenir le faisceau désiré. En envoyant des impulsions synchrones sur plusieurs petits éléments proches, il est possible de synchroniser les ondes pour obtenir une onde résultante équivalente à un front d'onde plan, comme illustré à la figure 2.5. Ce type de découpe peut s'appliquer à toutes les formes de transducteur. Le CEA développe une évolution de ces traducteurs : des traducteurs flexibles, au contact et sans sabot, qui épousent la forme de la pièce à inspecter

malgré des déformations complexes. Ils mesurent en temps réel les déformations du profil afin de les compenser en appliquant les lois de retards correspondantes.

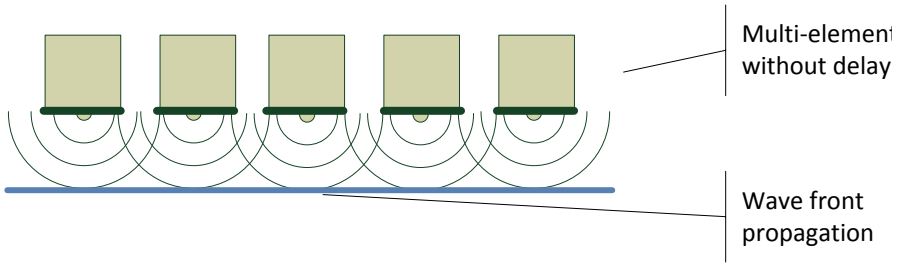


FIGURE 2.5 – Capteur multi-éléments générant un front d'onde plan

2.3.1.2 Les différents principes de focalisation

Afin d'orienter le faisceau émis par le transducteur et de concentrer l'énergie en une zone donnée, on cherche à focaliser les ondes sonores émises pour améliorer la détection de défaut. Plusieurs moyens permettent de focaliser une onde :

Lentille acoustique : de la même manière qu'en optique, on peut placer une lentille directement sur la face du transducteur. Les caractéristiques de la focalisation dépendront des propriétés de la lentille (figure 2.6a).

Lame piézoélectrique mise en forme : sur les capteurs récents, en particulier piézocomposites, il est possible de mettre en forme directement le transducteur afin de s'affranchir des inconvénients liés à l'utilisation d'une lentille (figure 2.6b).

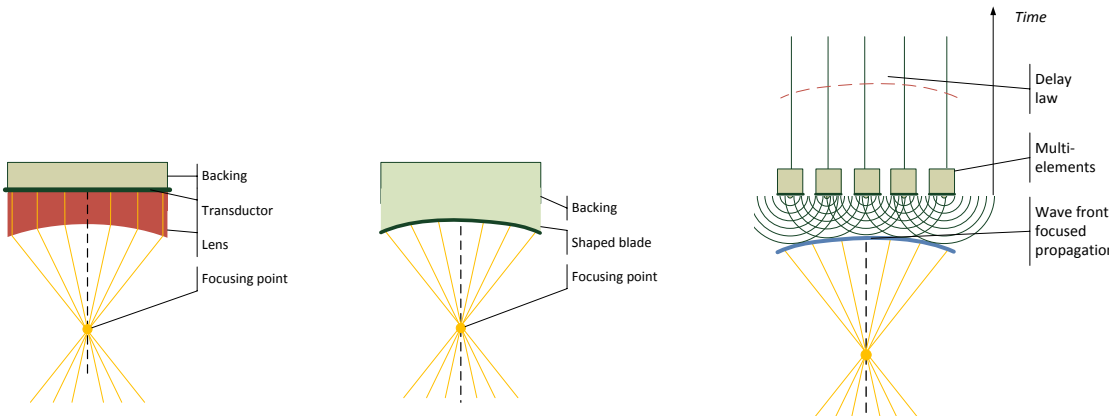
Focalisation électronique par loi de retards : sur un capteur multi-éléments, il est possible de simuler électroniquement la présence d'une lentille. L'application de retards sur l'ensemble des éléments permet de les faire contribuer, en phase et collectivement, à la focalisation. Ainsi, en prenant en compte, comme retards, les temps de vol correspondants aux trajets en un point, il est possible de focaliser le capteur multi-éléments en ce point. Inversement en décalant temporellement les signaux reçus, avant sommation, il est possible d'améliorer la détection (figure 2.6c).

2.3.2 Quelques techniques de visualisation

Un contrôle non destructif par ultrasons génère une multitude de données. Ces données doivent être mises en forme afin de permettre leur exploitation par l'expert réalisant l'inspection de la pièce. La figure 2.7 présente les visualisations de résultat d'un contrôle ultrasonore couramment utilisées en fonction du type d'information recueilli.

Ainsi, on visualise, en général, comme résultat d'une inspection, soit :

Un A-Scan correspondant au signal reçu par le capteur, c'est à dire la transcription électrique du déplacement engendré par l'onde émise, représenté par une courbe de l'amplitude du déplacement en fonction du temps (*c.f.* la figure 2.7a).



(a) Montage d'un capteur avec lentille

(b) Capteur mis en forme

(c) Focalisation électronique d'un capteur multi-éléments

FIGURE 2.6 – Différents procédés de focalisation

Un B-Scan représentant la juxtaposition d'un ensemble de signaux, obtenus en des positions contiguës le long d'un axe. L'amplitude est codée en niveaux de couleurs pour obtenir une image représentant dans un axe la position et dans l'autre le temps. Le plus classiquement, il s'agit des signaux mesurés aux différentes positions lors d'un déplacement du capteur. A l'aide d'un séquençage électronique, il est aussi possible, en jouant sur les lois de retards, de simuler un déplacement en allumant successivement une partie des éléments du capteur. Sa visualisation est la même que celle d'un déplacement. Il est possible de présenter un **B-Scan vrai** en modifiant l'axe des signaux pour que l'image suive le chemin de l'onde ultrasonore dans la pièce et pour mieux recaler l'image sur la pièce visualisée en 3D afin de faciliter la compréhension des résultats par l'expert (la figure 2.7b présente une telle visualisation).

Un S-Scan ou balayage angulaire s'inspire de la représentation B-Scan des signaux obtenus par un capteur multi-éléments. L'image correspond aux signaux obtenus pour plusieurs lois de retards faisant varier l'angle de l'onde. L'image résultat s'approche de ce qui est classiquement présenté en échographie médicale, de forme circulaire (*c.f.* la figure 2.7c).

Il est possible de coupler d'autres découpes de l'espace de données pour obtenir des simulations plus étendues, par exemple en couplant un balayage électronique selon un premier axe avec un déplacement mécanique dans un second axe. Cependant, dans le cadre d'une simulation rapide, l'objectif est d'afficher de manière interactive une image du contrôle sur l'écran de l'opérateur, afin que celui-ci puisse la faire évoluer selon ses manipulations. Des données volumiques sont à la fois plus complexes à appréhender pour un opérateur et plus longues à simuler. Dans ce cadre, la visualisation des résultats se concentre sur des **B-Scan** et des **S-Scan**.

2.3.3 Exemple de contrôle non destructif par ultrasons

Afin d'élaborer des modèles de simulation de contrôle, il est nécessaire de les valider expérimentalement. Lors de la conférence *Quantitative NonDestructive Evaluation* (QNDE) 2013, les performances de simulations d'écho ont été comparées dans le cadre du *benchmark* à des me-

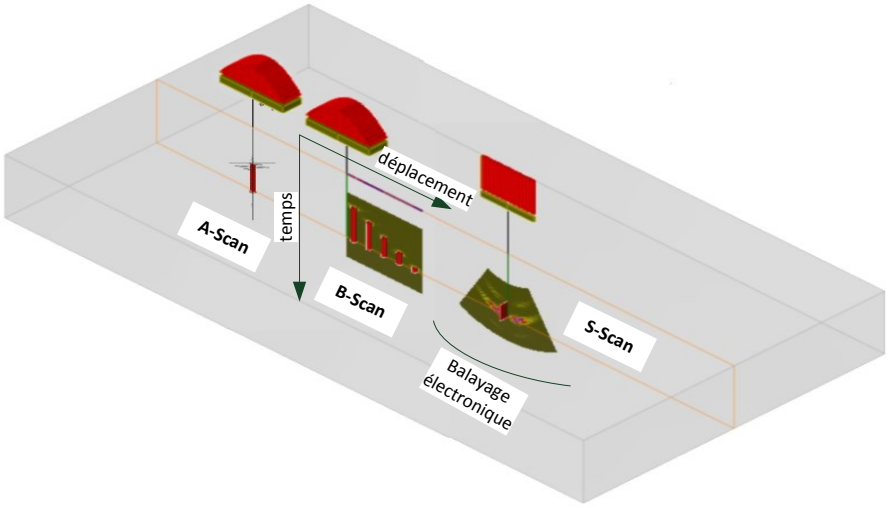
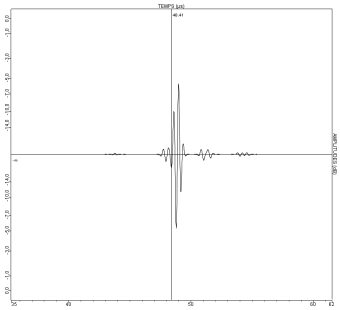
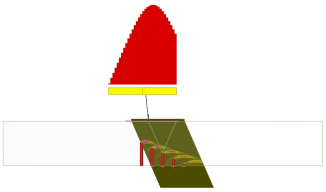


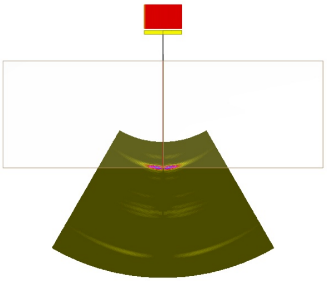
FIGURE 2.7 – Montage représentant les différentes visualisations dans la pièce



(a) A-Scan



(b) B-Scran vrai (rayons redressés le long des trajets)



(c) S-Scan d'un défaut plan

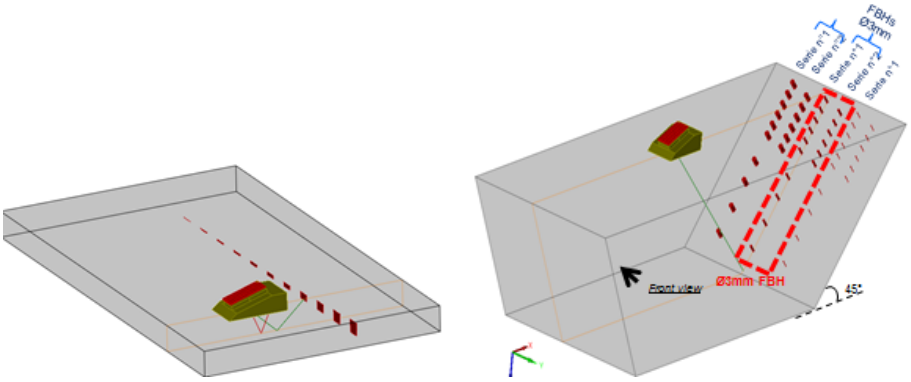
FIGURE 2.7 – Visualisation d'un contrôle non destructif ultrasonore

sures réalisées sur des cales usinées d'acier homogène, à l'aide d'un traducteur multi-éléments au contact. Un exemple de ces signaux obtenus en acquisition est présenté à la figure 2.9.

Le transducteur émet un faisceau d'ondes ultrasonores dans le sabot en plexiglas. A la rencontre de la première interface (plexiglas/acier) avec la pièce, une partie de ce faisceau est réfléchie et le reste pénètre dans la pièce. L'onde réfléchie, lorsqu'elle est reçue par le capteur fonctionnant également en mode réception, est appelée "onde d'interface" (ou onde de face avant). Dans le cadre de ce contrôle, il est à noter que les échos de sabot sont négligés.

Les phénomènes mis en jeu lors des contrôles réalisés pour le *benchmark* QNDE 2013 sont les suivants. En l'absence de défaut, le faisceau se propage à travers la pièce jusqu'au fond de la pièce (acier/air) ; l'écho reçu correspondant à la réflexion est nommé "écho de fond". Dans le cas du *benchmark*, dans les pièces de référence, deux types de défauts usinés sont présents : des entailles pour l'une des pièces, des trous à fond plat dans la seconde, respectivement illustrés par les figures 2.8a et 2.8b. Le fond des trous fond plat et les parois des entailles font office de réflecteur, renvoyant une partie de l'onde au récepteur et produisant des échos de défaut. L'analyse de ces échos en amplitude (après étalonnage des signaux simulés par rapport aux

signaux des acquisitions) et en temps de vol permet de juger de la qualité d'une simulation d'écho.



(a) Pièce présentant des entailles sur sa face arrière - vue du modèle CIVA

(b) Pièce présentant des séries de trous fond plat - vue du modèle CIVA

FIGURE 2.8 – Modélisation des pièces utilisée lors du *benchmark* de la conférence QNDE 2013

2.3.4 La simulation de contrôle non destructif

L'intérêt des simulations de contrôle est d'éviter de réaliser de multiples contrôles coûteux pour assister l'utilisateur dans la compréhension et/ou 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 et reposant sur des codes d'éléments finis possèdent des modules de simulations acoustiques et piézoélectriques qui peuvent être utilisés pour le contrôle non destructif. Ces outils génèrent, à une calibration près,

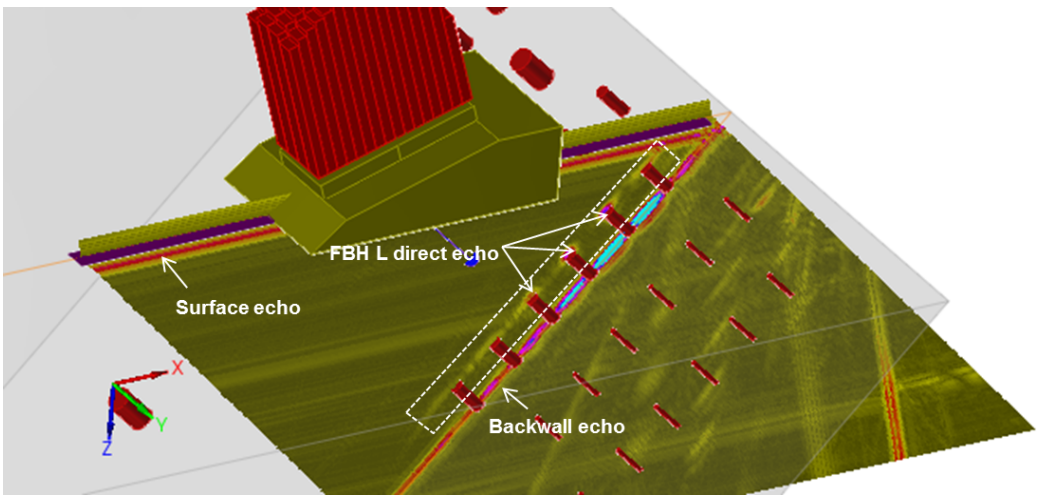


FIGURE 2.9 – Image des signaux d'acquisition recalés sur le modèle de la cale dans CIVA

des résulats équivalents à ceux obtenus en acquistion. On peut citer parmi ceux-ci : COMSOL (COMSOL) et Abaqus (Dassault Systèmes).

Concernant les outils plus spécifiques au domaine du contrôle non destructif, on peut citer PZFlex (Weidlinger Associates). Basé sur un code d'éléments finis, il a pour domaines, non seulement le contrôle non destructif, mais aussi l'échographie ultrasonore médicale et la thérapie par ultrasons. Ce code est parallélisé sur processeur multi-cœurs et les calculs peuvent être distribués sur plusieurs nœuds. Néanmoins les temps de calcul sont souvent assez conséquents. La figure 2.10 présente le front d'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.

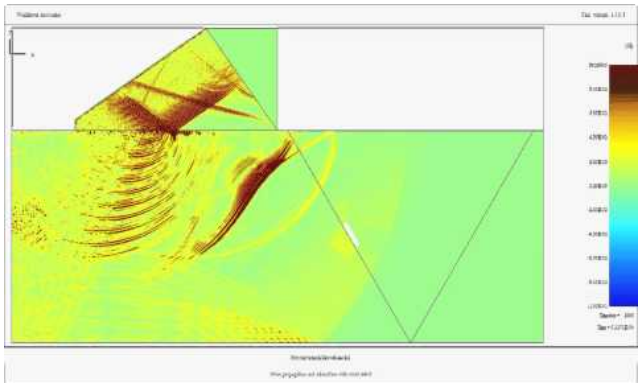


FIGURE 2.10 – PZFlex : Propagation d'une onde transverse ultrasonore émise par un capteur multi-éléments au contact dans une pièce comprenant une soudure

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. Une estimation du A-Scan résultant est donnée. La figure 2.11 présente une inspection réalisée avec le logiciel UTMan.

Dans la même catégorie, on peut citer ES Beam Tool (distribué par Eclipse Scientifique). Il permet à l'aide de nombreux outils de définir le type de capteur (mono ou multi-éléments, TOFD[3]), 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 2.12 représente l'interface de ce module.

L'approche des noyaux de calcul principaux CIVA est une approche intermédiaire entre ces deux extrêmes (code de type éléments finis ou calculs rapides par lancer de rayons). Ses codes de simulation semi-analytiques permettent d'obtenir des résultats complexes (champs ultrasonores ou signaux d'échos) et valides dans les limites de ces modèles semi-analytiques bien plus rapidement que des simulations par éléments finis. Dans la continuité de cette approche, la présente thèse propose des simulations en temps interactif.

[3]TOFD : *time of flight diffraction*, deux capteurs sabot sont placés "face-à-face" de manière à ce que le faisceau émis et le faisceau reçu "éclairent" la zone où se situe défaut

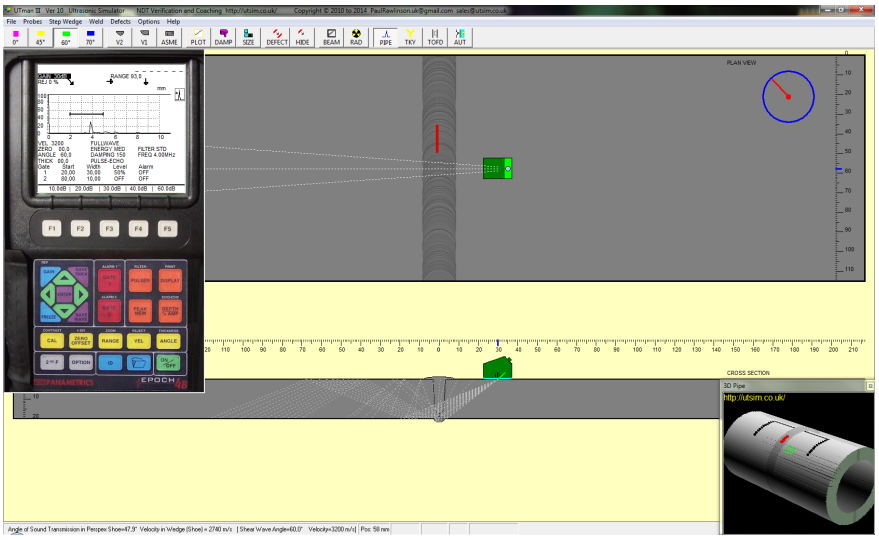


FIGURE 2.11 – UTMan :inspection d'une soudure raccord de deux tuyaux à l'aide d'un capteur mono-élément que l'on peut déplacer à la souris

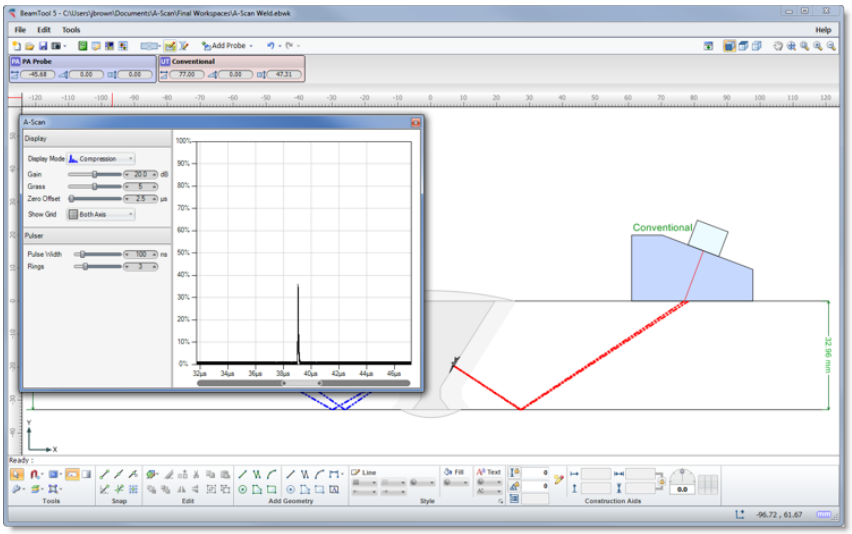


FIGURE 2.12 – Module Beam Tool AScan

La section suivante détaille les caractéristiques de la plateforme CIVA autour de laquelle ce travail de thèse s'articule.

2.4 La plateforme CIVA

Le logiciel CIVA est une plateforme multi-techniques d'expertise pour le contrôle non destructif.

2.4.1 Présentation générale

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

CIVA prend en charge des contrôles par Ultrasons (UT), Ondes Guidées (GWT), Radiographie (RT), Tomographie Calculée (CT) et Courants de Foucault (CF).

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 capteur, en guidant celle-ci, en particulier dans le cadre des technologies multi-éléments ou *Electromagnetic Acoustic Transducer* (EMAT, pour générer des ondes ultrasonores via 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...

2.4.1.1 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.

2.4.1.2 CIVA, un modèle complet

CIVA, actuellement disponible dans sa version 11, s'est construit sur un 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 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 lors d'une simulation de mise au point.

CIVA propose des modules concernant les techniques suivantes :

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.

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'un large panel de défauts et de capteurs modélisés, permettant de couvrir de très nombreux cas.

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.

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.

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 relai et calcule le comportement du faisceau, en particulier les interactions faisceau/défaut et faisceau/pièce. Cette approche hybride permet un gain de temps et de précision en calculant l'ensemble du comportement du faisceau dans la région d'intérêt.

Les différents modèles de CIVA sont régulièrement validés, au cours des manipulations 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. De plus, le CEA participe chaque année aux bancs d'essai de la conférence QNDE.

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 plugin 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. 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 à appréhender.

2.4.2 CIVA UT

Le module CIVA UT comprend un panel d'outils destinés à la simulation de l'intégralité d'une procédure de contrôle.

2.4.2.1 La simulation avec CIVA UT

Plusieurs capteurs sont disponibles à la simulation :

- Capteur au contact, en immersion ;

- Pastille d'émission de forme rectangulaire ou circulaire ;
- Pastille à focalisation, mise en forme ou avec une lentille ;
- Surface d'émission plane, cylindrique focalisée, sphérique, bifocale, trifocale, ou Fermat ;
- Traducteur en Émission/Réception séparées, en configuration symétrique ou non symétrique ;
- Capteurs mono ou multi-éléments ;
- Barrettes encerclantes ou encerclées pour l'inspection de tubes ;
- Capteurs multi-éléments flexibles de découpe 2D ou 3D.

Il est possible de modéliser dans CIVA une pièce de trois manières différentes. Tout d'abord de nombreux modèles paramétriques existent, en géométrie 2D extrudée : plan, cylindrique, conique et sphériques ; ou en 3D : coude, piquage ou alésage. Par ailleurs, CIVA permet de modéliser des géométries plus complexes, soit sous forme de CAO 2.5D, d'extrusion plane ou cylindrique, soit sous forme de CAO 3D par import de fichiers.

CIVA gère les pièces homogènes et hétérogènes. Chacun des matériaux constituant la pièce peut être isotrope ou anisotrope quelconque. Il est également possible de simuler plusieurs types de matériaux composites en fonction de leur structure. Enfin, un bruit de structure (distribution aléatoire de points de densité et réflectivité ajustables pour chaque milieu constitutif de la pièce) peut être ajouté pour refléter les conditions réelles du contrôle.

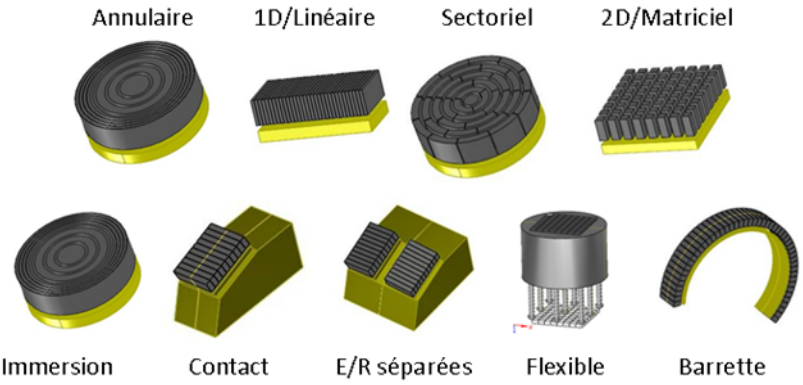


FIGURE 2.13 – Différents découpages de capteurs UT multi-éléments gérés par CIVA

Source : Image fournie par EXTENDE

CIVA permet de piloter un ensemble de capteurs multi-éléments, avec des configurations plus ou moins complexes de séquences de lois de retards. Les capteurs gérés peuvent être de formes très diverses, avec, pour les plus classiques : des capteurs annulaires, linéaires, sectoriels, matriciels ou encore flexibles... La figure 2.13 présente l'ensemble des découpages des capteurs gérés par CIVA[RID12]. Les lois de retards peuvent être calculées par CIVA pour l'ensemble des capteurs/découpages présentés, sur les différentes pièces modélisées.

En une position donnée, on peut procéder à un séquençage (c'est à dire une succession de sélections d'éléments actifs, en émission et en réception). Pour une séquence donnée, on peut effectuer pour chaque tir en émission un ou plusieurs tirs en réception. Les types de lois de retards pour un tir donné sont :

- focalisation en un point,

- focalisation en plusieurs points,
- focalisation en une direction,
- focalisation en plusieurs directions successives (balayage),
- focalisation en profondeur et direction...
- focalisation en direct ou en rebond pour un mode donné.

Ces lois peuvent être identiques en chaque position du capteur, ou au contraire différentes pour s'adapter à la géométrie ou à un changement de matériaux. Les retards peuvent être pré-calculés ou calculés dynamiquement (cas d'un capteur flexible) en chaque position.

CIVA peut simuler une configuration de contrôle avec un nombre quelconque de défauts présents dans le volume d'une pièce. Les types de défauts considérés sont les suivants :

Réflecteurs étalons : trou génératrice[4], trou à fond plat et trou à fond hémisphérique ;

Défauts plans de taille et d'orientation quelconques, rectangulaires, semi-elliptiques ou de contour CAO 2D, simulant des fentes ;

Défauts multi-facettes : combinaison de défauts plans permettant de simuler des fentes plus complexes ;

Inclusions (de forme cylindrique, sphérique ou ellipsoïdale) : défauts constitués d'un matériaux solide.

La figure 2.14 présente une représentation de ces différents modèles de défauts dans l'interface de CIVA.

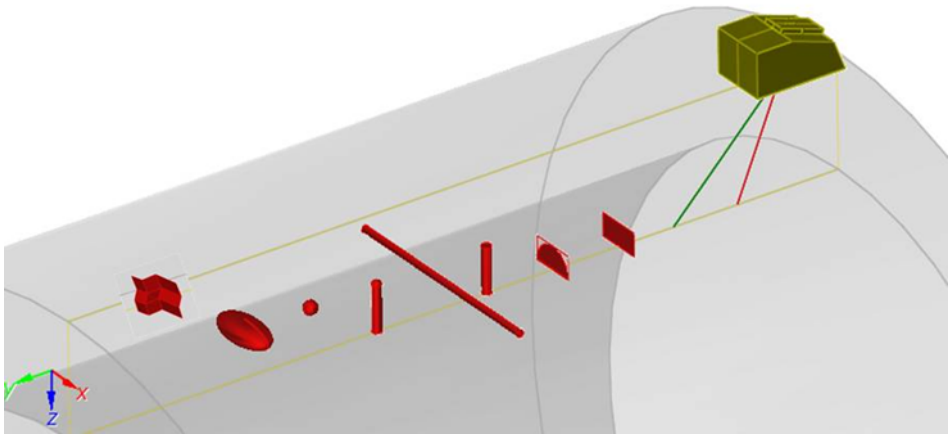


FIGURE 2.14 – Différents défauts compatibles avec le mode CIVA UT

Source : Image fournie par EXTENDE

[4]Un trou génératrice modélise une perforation de la pièce de forme cylindrique sur la surface (latérale) de laquelle les ultrasons se réfléchissent.

2.4.2.2 Résultat des simulations

La première forme des résultats de simulation UT est la simulation d'un champ ultrasonore, c'est à dire le faisceau ultrasonore rayonné dans la pièce. L'amplitude de celui-ci peut être visualisée dans la matière, de même que les directions locales et les fronts d'onde. Il est également possible d'en extraire une information sous la forme d'animation de propagation. CIVA peut également simuler l'interaction faisceau/défaut et prédire l'amplitude et le temps de vol des échos dans la pièce (échos de défaut ou de géométrie, échos directs spéculaires ou de diffraction).

Un certain nombre d'outils d'analyse permettent de compléter les simulations dans CIVA. Ainsi, sont fournis des outils de traitement du signal (classiques ou évolués), un outil de segmentation pour le regroupement 3D des signaux et la création de rapports d'examen. Un outil de tracé de rayons permet des analyses rapides. Sont également intégrés des outils de reconstruction, en particulier la méthode Focalisation En Tous Points (FTP), illustrée par la figure 2.15.

A l'issue d'une acquisition ou d'une simulation multi-éléments (figure 2.16a), cette méthode permet de reconstruire en chaque point d'une zone d'intérêt l'amplitude obtenue en combinant les signaux par une focalisation *a posteriori* (par manipulation des lois de retards par rapport au temps de vol, voir figure 2.16b).

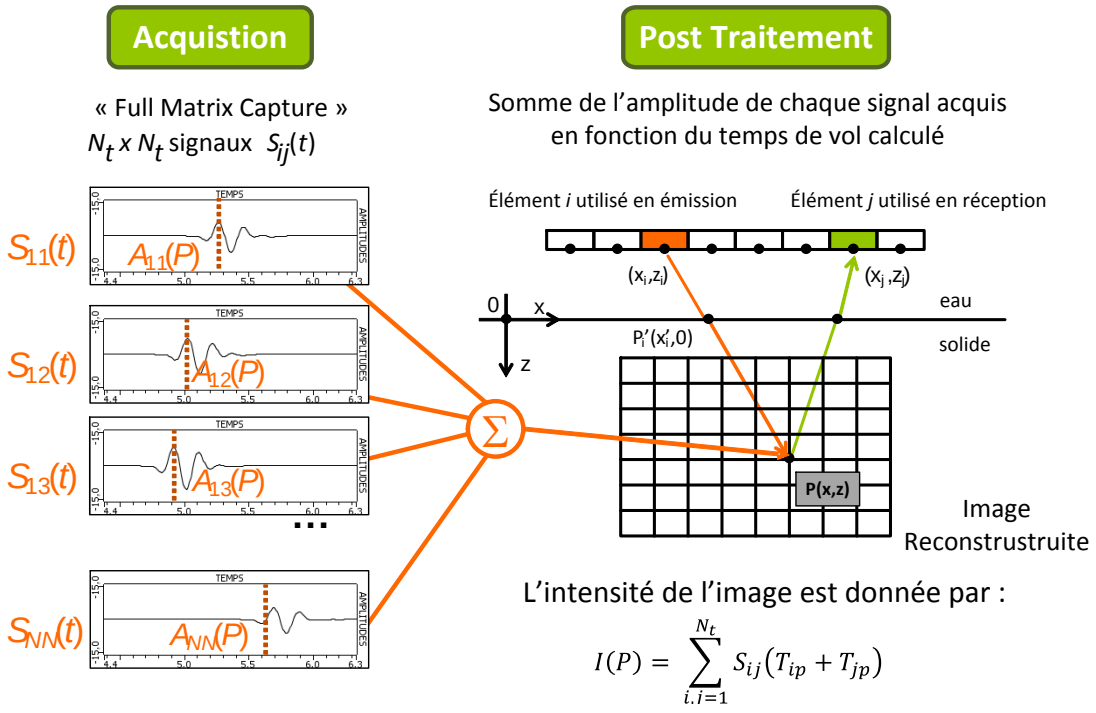
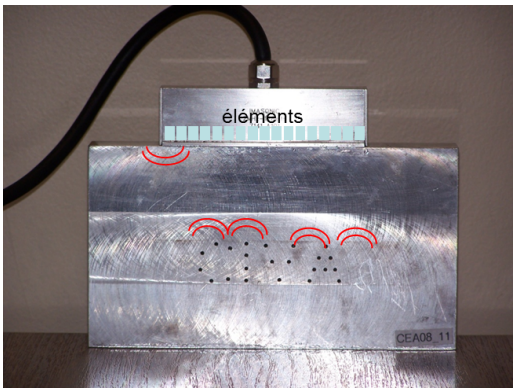
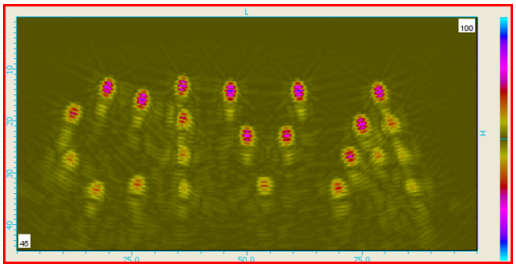


FIGURE 2.15 – Principe de la méthode Focalisation En Tous Points



(a) Acquisition de la matrice de signaux (tous les couples émetteurs/récepteurs)



(b) Reconstruction à partir des signaux

FIGURE 2.16 – Illustration de la méthode FTP afin de reconstruire une image à partir des signaux acquis sur une pièce contrôlée

2.5 Conclusion

Ce chapitre a mis en évidence les exigences industrielles du contrôle non destructif, 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 avec ses modèles semi-analytiques complets qui lui permettent d'accélérer les temps de calcul sur un large ensemble de configurations de contrôle.

C'est en vue de doter CIVA d'une simulation de champ très rapide 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 et des contraintes d'implémentation associées.

Chapitre

3 Architectures parallèles pour la simulation

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 de champ ultrasonore rapide sur la même catégorie d'ordinateurs que la plateforme CIVA. Dans le cadre industriel, les utilisateurs et experts CND travaillent sur des stations de calculs puissantes mais disposant de composants généralistes.

3.1 Panorama des architectures parallèles et des outils de programmation associés

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 [Moo95]. Même s'il est prévu que des phénomènes physiques risquent d'engendrer des bruits parasites 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 la puissance de calcul. Jusqu'en 2004 cette augmentation était obtenue principalement par l'augmentation de la fréquence de calcul. Depuis, les constructeurs ont arrêté d'accroitre la fréquence de fonctionnement des processeurs en raison des contraintes thermiques (l'énergie libérée par un processeur est proportionnelle à sa fréquence et au carré de sa tension, qui dépend elle-même de la fréquence :

l'énergie libérée dépend donc approximativement au cube de la fréquence de fonctionnement). 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 Du supercalculateur à la station de travail

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 possibles 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 et/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 15000 habitants. L'utilisation de ces superordinateurs a donné naissance à la discipline du calcul à haute performance (ou *High Performance Computing*, HPC).

Depuis 1993, et deux fois par an, le projet **TOP500**[1] recense les 500 plus puissants superordinateurs mondiaux selon un *benchmark* LINPACK (basé sur de l'algèbre linéaire tel qu'une décomposition LU de matrice). La figure 3.1 présente les performances obtenues de ces ordinateurs, respectivement pour les machines classées première et dernière 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 (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.

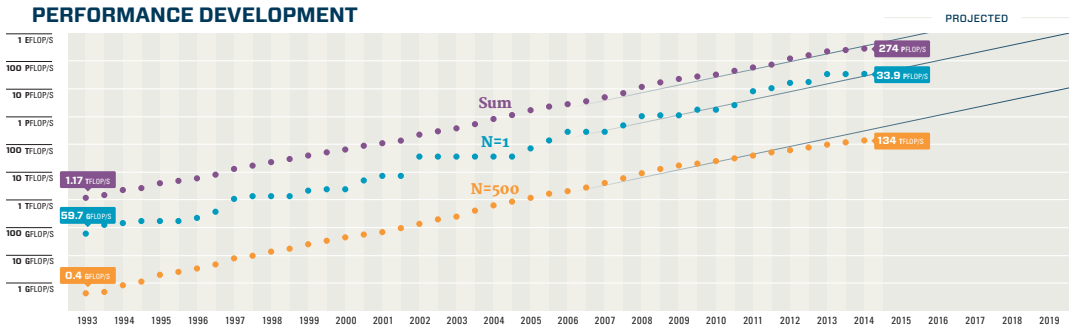


FIGURE 3.1 – Chronologie des performances des supercalculateurs du TOP500

Source : TOP500 - www.top500.org

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, ouvrant aux

[1] www.top500.org

stations de travail la possibilité de réaliser des simulations relativement complexes dans des temps raisonnables grâce à leurs capacités de calcul. Par exemple, un coprocesseur Intel Xeon Phi 7120P (commercialisé en novembre 2012), est annoncé comme fournissant une puissance de calcul de 1.2 TFlops, soit l'équivalent d'une machine présente au **TOP500** de 1998 à 2005.

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 d'aujourd'hui présentent un bon compromis puissance/coût et sont déjà couramment utilisées pour réaliser des simulations standardes. Ces plateformes vont être ciblées par les présents travaux de thèse avec pour objectif de franchir un cap en obtenant des temps de calcul interactifs.

3.1.2 Les capacités des processeurs généralistes

Le premier composant utilisé pour obtenir de la puissance de calcul est le processeur généraliste, (ou *general purpose processor* GPP, aussi connu sous le nom de *Central Processing Unit* ou CPU). 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.

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 GPP possèdent des composants dédiés à chacun de ces types d'opération : 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 :

Pipelining 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, accès mémoire, modification des registres).

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.

A ces outils propres à l'exécution du code, se rajoutent des caches pour mettre dans un tampon les informations mémoire qui sont potentiellement utiles à la suite du programme. Ces caches sont organisés en plusieurs niveaux, avec très souvent un cache L1 dédié aux instructions du programme, un second cache L1 consacré à l'accès mémoire, lui même dépendant d'un cache L2. Sur certaines architectures, un cache L3 est présent entre la mémoire globale et le GPP.

3.1.2.1 Les différents types de parallélisme algorithmique

Algorithmiquement, il est possible de distinguer deux grandes classes de parallélisme. La première est le **parallélisme de données** : lorsqu'un traitement constitué d'une succession d'opérations

séquentielles est appliqué à un ensemble de données contrôlé par un seul et unique flux de contrôle. La seconde est le **parallélisme de tâches** : lorsque des tâches complexes, constituant un ensemble, collaborent afin de traiter un problème donné, chacune distincte et disposant de son propre flot de contrôle.

3.1.2.2 Les instructions vectorielles

Une première approche afin de multiplier les capacités des GPP consiste à jouer sur le parallélisme de données. En effet, très souvent, il est demandé au GPP d'exécuter la même opération sur un ensemble de données (par exemple de l'algèbre linéaire sur de très grosses données).

Des instructions vectorielles ont été ajoutées aux GPP, 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 GPP 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 [Fly72]). Cependant, il faut garder à l'esprit qu'il ne s'agit pas d'instructions vectorielles en tant que telles : une instruction vectorielle s'exécute sur un registre de données en fonction d'un masque, ces instructions SIMD ne permettent pas toujours de s'appliquer conditionnellement[2]. Pour le grand public, ces instructions se sont principalement développées pour permettre des traitements multimédia toujours plus gourmands.

3.1.2.3 Origines des multicœurs

Une autre approche développée pour augmenter les capacités de traitement des GPP a été la mise en place de GPP composés de plusieurs cœurs ou unités de calculs, gravés au sein de la même puce. Le premier processeur multicœur a été développé par IBM. Il s'agit du POWER4, sorti en 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 GPP 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 caches sont parfois regroupés 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 processeurs étudiés

Cette section présente les principales caractéristiques architecturales des différents processeurs étudiés afin de mieux appréhender les spécificités de chacun d'entre eux.

[2]Parmi les GPP grand public, les processeurs de technologie ARM peuvent disposer d'un jeu d'instructions NEON permettant une exécution conditionnelle. Les instructions AVX-512 bientôt disponibles permettront également d'utiliser des masques conditionnels.

3.1.2.4.1 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 *HyperThreading*, 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 cache successifs.

- Un **cache L1** de 64 kilo-octets séparés en 32 kilo-octets dédiés aux données et 32 kilo-octets d'instructions ;
- Un **cache L2** de 256 kilo-octets par cœur ;
- Un **cache L3** de plusieurs méga-octets partagé par l'ensemble des cœurs d'une même puce. La taille d'une ligne de 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 3.2.

En début d'année 2010, Intel a mis à disposition les premiers processeurs d'architecture **Westmere**, amélioration de *Nehalem*, cette fois-ci gravés en 32 nanomètres. Le nombre de cœurs maximum par GPP 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).

3.1.2.4.2 Architecture Sandy Bridge En janvier 2011, Intel a poursuivi sa stratégie dénommée "tick-tock", alternant une évolution majeure "tock" avec une évolution de la précédente génération, "tick", souvent en augmentant la finesse de gravure du silicium. L'architecture *Sandy Bridge* constitue un "tock", 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 simple précision ou 4 nombres flottants double précision simultanément. L'unité d'exécution Sandy Bridge est détaillée par la figure 3.3. Les GPP *Sandy Bridge* peuvent disposer de jusqu'à 8 cœurs physiques chacun.

La gamme suivante, "tick", se nomme **Ivy Bridge** et a été commercialisée au printemps 2012. Cette fois-ci, la finesse de gravure est de 22 nanomètres. Comme lors du "tick" précédent, 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). Au niveau architectural, les améliorations touchent principalement aux fonctionnalités grand public, avec une meilleure gestion du GPU intégré et de meilleures fonctionnalités multimédia.

3.1.2.4.3 Architecture Haswell La dernière architecture Intel commercialisée fait partie de la catégorie "tock" selon la dénomination de l'entreprise. Mise à disposition en juin 2013, elle permet l'utilisation des instructions AVX2 offrant des fonctionnalités SIMD sur les nombres entiers. L'unité d'exécution Haswell est présentée par la figure 3.4. On y observe la présence de deux unités réalisant les FMA vectorielles sur les ports 0 et 1 pour des vecteurs de largeur 256 bits.

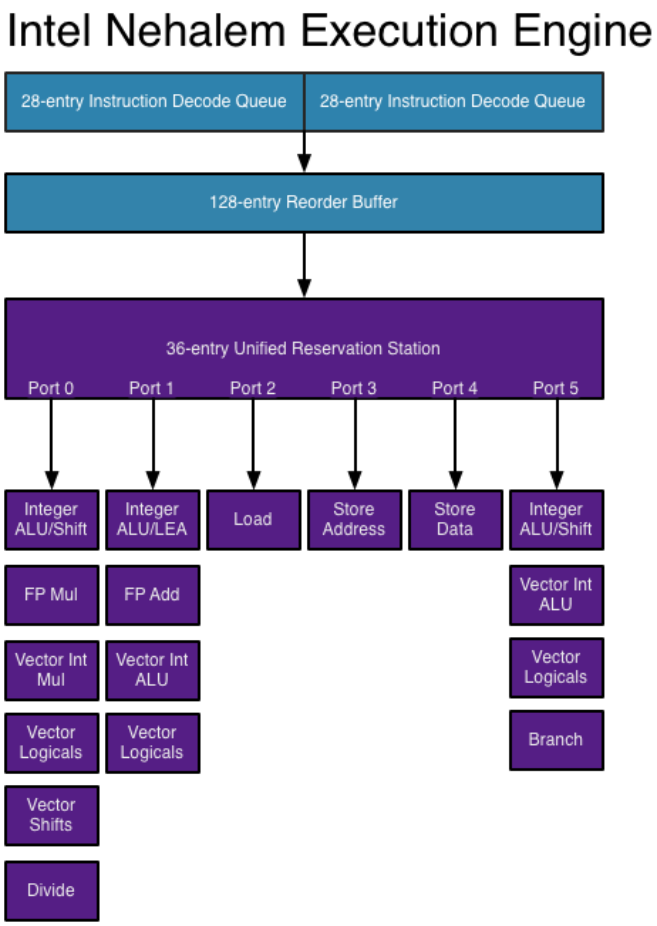


FIGURE 3.2 – Unités d'exécution Nehalem

Source : Anandtech

3.1.3 Le coprocesseur Many Integrated Cores

Entre 2005 et 2010, Intel a vu sa position de fabricant leader du TOP500 s'effriter avec l'arrivée d'un composant, offrant des performances importantes, utilisé comme coprocesseur de calcul : le processeur graphique. Afin de répondre, Intel a misé sur une architecture agrégeant un grand nombre de cœurs GPP moyennement puissants sur un même coprocesseur afin d'obtenir des performances importantes : le coprocesseur **Many Integrated Cores** (MIC). Ces cœurs reposent sur une architecture x86 compatible avec les GPP précédents et disposent d'une unité de traitement d'instructions SIMD d'une largeur de 512 bits.

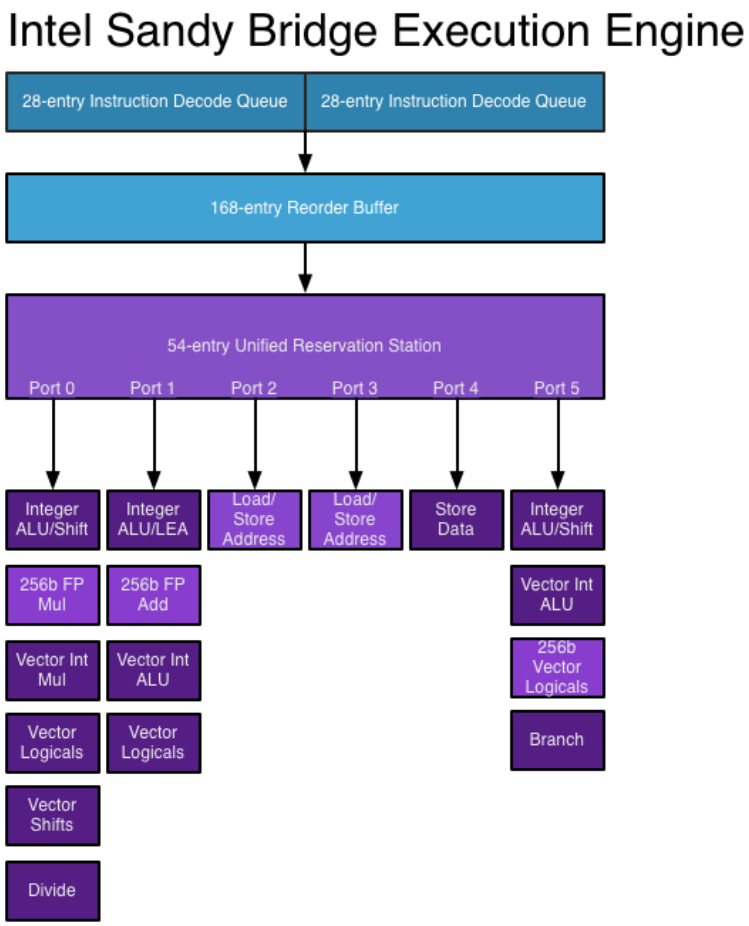


FIGURE 3.3 – Unités d'exécution Sandy Bridge

Source : Anandtech

3.1.3.1 Historique

Intel a mis au point en mai 2010 un prototype, au nom de code *Knight Ferry*, produit à destination de quelques partenaires particuliers (CERN, Korea Institute of Science and Technology Information (KISTI) et Leibniz Supercomputing Centre). Ce prototype présentait 32 cœurs compatibles x86, cadencés à 1.2 GHz accompagnés de 2 giga-octets de mémoire GDDR5 et pouvait offrir des performances dépassant 750 GFlops.

A la suite de cette première génération prototype, Intel a dévoilé en juin 2012, une nouvelle génération reposant sur la micro-architecture *Knight Corner* et destiné à la commercialisation sous le nom de *Xeon Phi*. Les expérimentations réalisées dans le cadre de ce travail de thèse ont été effectuées sur une architecture de ce type dont les détails sont présentés dans la section suivante.

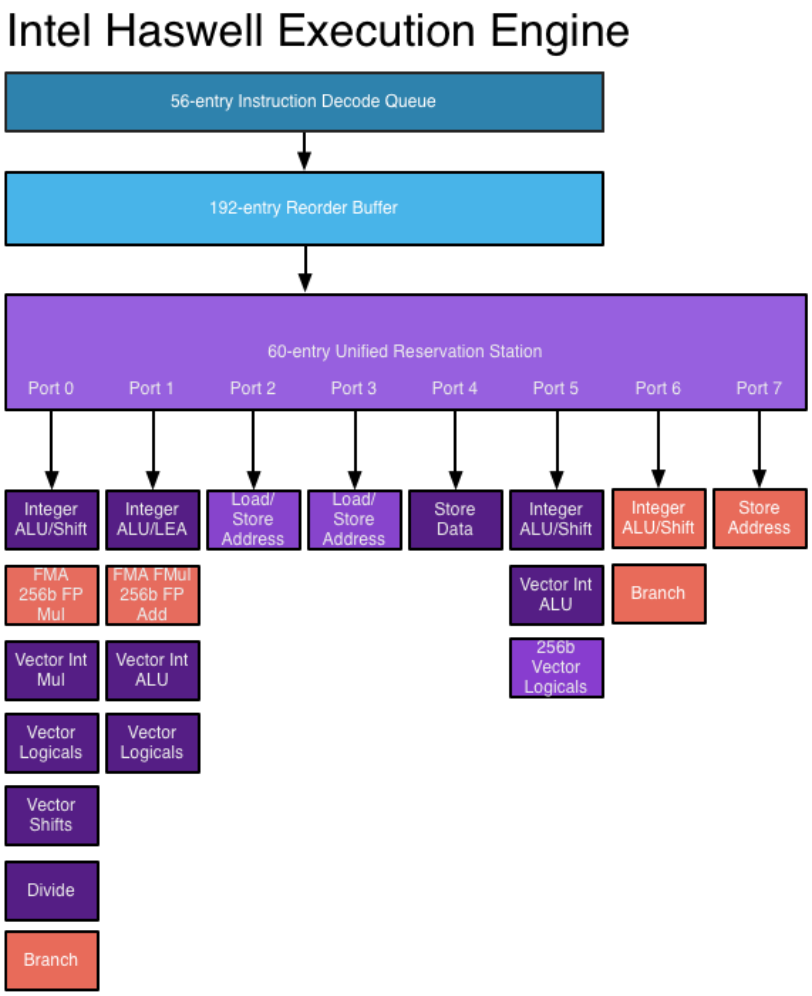


FIGURE 3.4 – Unités d'exécution Haswell

Source : Anandtech

3.1.3.2 Xeon Phi - Knight Corner

Le design du Knight Corner se base sur une version modifiée du design des cœurs Pentium des générations précédentes, gravé en 22nm. Ce coprocesseur dispose de plus de 50 cœurs *in-order*, compatibles avec l'architecture x86-64bits, capables d'exécuter simultanément quatre *threads* matériels à la fois par cœur (à la manière de l'*HyperThreading*) afin de masquer les latences. Les cœurs sont cadencés à plus de 1 GHz. Chacun dispose d'un cache L2 de 512 kilo-octets, cohérent avec le reste des caches L2 des autres cœurs permettant une bande passante à plus de 25 méga-octets par seconde. Cette cohérence est assurée par l'architecture globale en anneau bidirectionnel (*c.f.* figure 3.5), permettant d'interroger tous les caches L2 avant de se reporter sur la mémoire globale lors d'un accès mémoire.

Par rapport à l'architecture originelle Pentium, les cœurs disposent d'une unité SIMD spé-

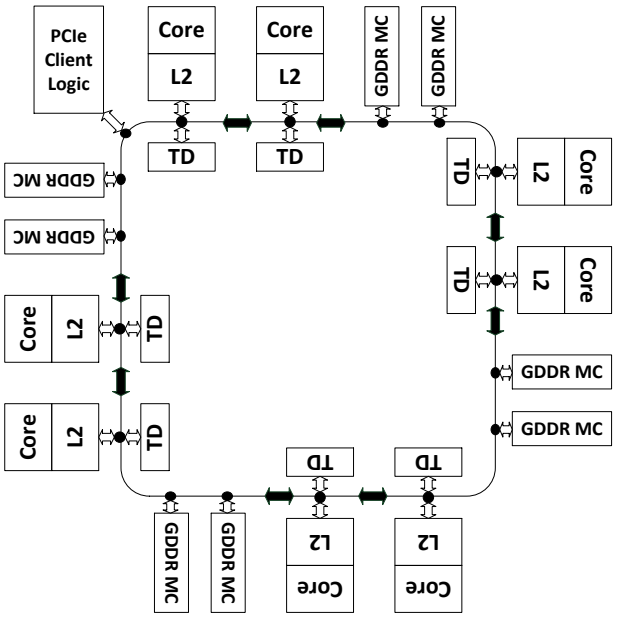


FIGURE 3.5 – Architecture en anneau bidirectionnel d'un Xeon Phi

cifique permettant l'usage d'instructions de 512 bits de large. Enfin, de nouvelles instructions SIMD de type *scatter/gather* permettent d'augmenter les capacités à gérer les accès mémoire dans des algorithmes SIMD et à rapprocher les machines SIMD modernes des anciennes machines vectorielles.

Intel a déjà annoncé une seconde génération de coprocesseurs *Xeon Phi*, dont l'architecture porte le nom de *Knight Landing*. Ils devraient supporter, comme la future micro architecture GPP *Skylake*, des instructions vectorielles AVX3.1 de largeur 512 bits, unifiant les Xeon généralistes (GPP) et *Xeon Phi*. Les premières informations indiquent également que les *Xeon Phi Knight Landing* pourraient se décliner non seulement comme coprocesseurs mais également comme processeurs GPP principaux. Cependant, ces processeurs n'étaient pas encore disponibles au moment où ce document a été rédigé.

3.1.4 Récapitulatif des instructions vectorielles supportées par architecture GPP et Many Core

Le tableau 3.1 récapitule les instructions supportées par les différentes architectures proposées.

On observe que Intel tend à élargir les capacités de calcul SIMD de ses GPP tout en visant une certaine convergence avec les futures architectures MIC (en particulier, grâce au jeux d'instructions AVX3.x). Cependant, un frein à l'usage de ces instructions de largeur toujours plus importante est l'expression d'une régularité suffisante des algorithmes.

3.1.5 Du processeur graphique au GPGPU

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

Architecture	SSE 1 à SSE 4.2	AVX	AVX 2	KNC MIC	AVX 3.x
Largeur (bits)	128	256	256	512	512
Nb. simple précision	4	8	8	16	16
Nb. double précision	2	4	4	8	8
GPP Nehalem	X				
GPP Sandy Bridge	X	X			
GPP Ivy Bridge	X	X			
GPP Haswell	X	X	X		
GPP Skylake (futur)	X	X	X		3.2
MIC Knight Corner				X	
MIC Knight Landing (futur)	?	?	X	?	3.1

TABLE 3.1 – Instructions SIMD supportées par architecture

d'oscilloscope par exemple. Dans les années 1970 sont apparus les *framebuffers* (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 *framebuffers* a été rapidement déportée vers des circuits spécialisés pour répondre, entre autre, 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 du rendu 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 ont été parmi les premiers pionniers [LM01] 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 GPP fournissant entre autre des opérations mathématiques sur nombres flottants, des boucles...

3.1.5.1 Description générique et contexte d'utilisation

A 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 rendu graphique souvent limitées et peu adaptées.

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. Suite au rachat d'ATI par AMD, fabriquant de GGP 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 GGP, langage présenté plus avant. En raison de la moindre maturité apparente des capacités de programmation en OpenCL

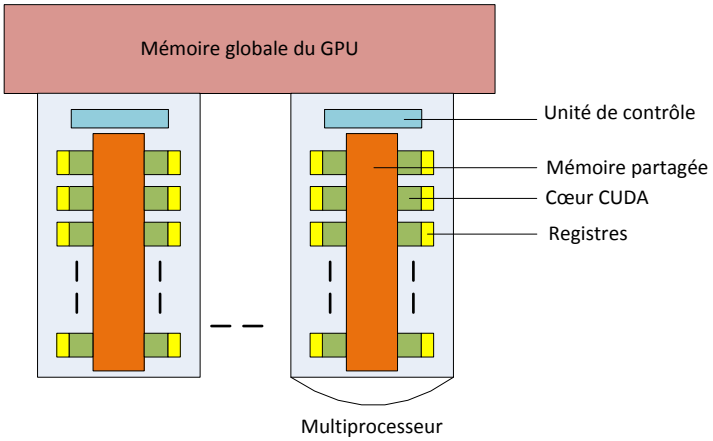


FIGURE 3.6 – Présentation globale d'un GPU CUDA

et d'un environnement de développement plus complet côté nVidia, il a été décidé de travailler avec les GPU de ce fabricant. Ces choix techniques sont explicités à la section 3.3.3.

3.1.5.2 Présentation générique des 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. Physiquement, un GPU CUDA peut se modéliser selon le schéma 3.6 : une mémoire centrale et des multiprocesseurs. Les différentes versions matérielles sont référencées sous le nom de *Compute Capability* (CC). La CC conditionne les capacités du GPU et son efficacité 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. La composition d'un multiprocesseur ainsi que la taille de sa mémoire partagée dépend 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 : lente d'accès mais de grande capacité (aux alentours d'un à deux giga-octets sur les GPU de loisir haut de gamme, plusieurs giga-octets pour la gamme professionnelle). Sa durée de vie est celle de l'application dans son ensemble. Les GPU de CC 2.x et supérieurs disposent d'un cache L2 commun à tous les multiprocesseurs pour l'accès à la mémoire globale.
- La mémoire partagée : locale 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). Sur les GPU de CC 2.x, elle assure les rôles de cache L1 et de mémoire partagée.
- Les registres, 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*).

La mémoire globale du GPU peut être vue, par l'application CUDA, de plusieurs manières, en fonction des types de caches utilisés (optimisés pour certains types d'accès), et des capacités qu'a

l'application à pouvoir ou non y écrire des informations à l'exécution. Le tableau 3.2 présente les différents accès possibles à la mémoire globale du GPU.

Type d'accès à la mémoire globale	Localité / Durée de vie	Accès Noyau	Cache	Accès mémoire
Accès direct	GPU / application	Lecture / Écriture	L1 et L2 sur les GPU de CC 2.x	Accès lents hors cache (centaines de cycles)
« Mémoire constante » (64Ko maximum pour l'application)	GPU / application	Lecture	8Ko par multiprocesseur	Accès optimisés pour les données constantes – les arguments du noyau par exemple. Écriture par l'hôte.
Textures	GPU / application	Lecture	Entre 6 et 8Ko par multiprocesseur	Arrangement et cache optimisés pour les calculs de texture (calculs d'interpolation 2D). Accès par des primitives spécifiques.
Surfaces	GPU / application	Lecture / Écriture	Non	Arrangement optimisé pour les calculs de textures (calculs d'interpolation). Accès par des primitives spécifiques.
« Mémoire locale » (16 - 512Ko sur GPU récents / *thread*)[3]	Par *thread* / *thread*	Lecture / Écriture	L1 et L2 sur GPU de CC 2.x	Accès lent. Le compilateur y place les variables ne pouvant entrer dans des registres (limite imposée par l'utilisateur et/ou le matériel).

TABLE 3.2 – Récapitulatif des méthodes d'utilisation de la mémoire globale d'un GPU CUDA

Les multiprocesseurs utilisent l'architecture *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

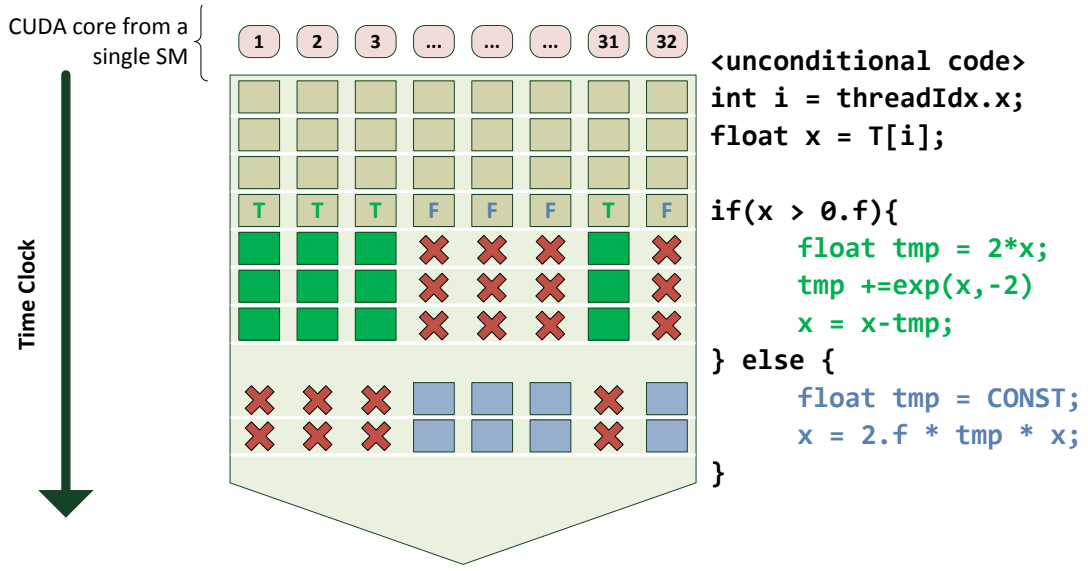


FIGURE 3.7 – Gestion de la divergence au sein d'un *warp* en CUDA lié à un bloc `if then else`

par le multiprocesseur. Les points d'ordonnancement se trouvent à la fin de chaque *warp*.

L'efficacité 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* comme illustré à la figure 3.7. Les performances sont dégradées par les exécutions successives des différentes branches pour les *threads* actifs.

L'architecture SIMT se rapproche du modèle vectoriel SIMD à la différence que les données restent propres à chaque *thread* et ne sont pas exposées à l'ensemble de l'application. Chaque *thread* commence à la même adresse programme mais possède son propre compteur ordinal et ses propres registres.

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.

3.1.5.3 L'architecture Fermi

Les GPU nVidia CUDA reposant sur la micro-architecture Fermi sont gravés en 40nm pour les processeurs de bureau et comprennent environ 3.0 milliards de transistors. Le livre blanc de nVidia présente cette architecture en détail [nVi09]. La mémoire associée au coprocesseur graphique est de type GDDR5. Ces cartes sont connectées à l'hôte par une interface *PCI-Express v2* 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 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 "2D".

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.

64 kilo-octets de mémoire ultra rapide répartis à l'exécution entre cache L1 et mémoire partagée.

1 interface vers le 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* (*fused multiply-add*), pour effectuer $A \times B + C$ en un cycle d'horloge. Ces cœurs sont capables de réaliser des calculs double précision en 2 cycles d'horloge.

Les SM arrangent les *threads* en *warps* de 32 [nVi15] 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. Cependant les instructions en double précision ne peuvent en bénéficier.

3.1.5.4 L'architecture Kepler

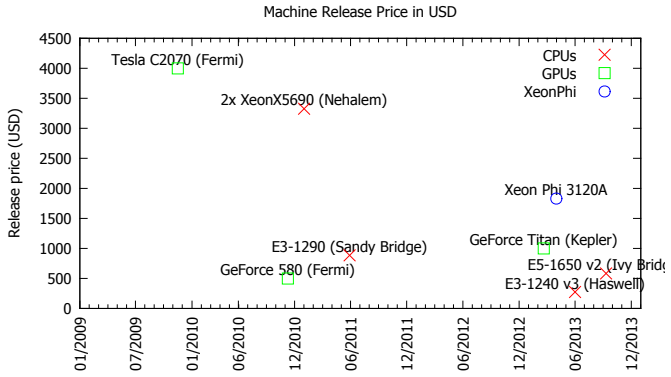
En 2012, trois ans apres Fermi, nVidia a mis sur le marché une nouvelle microarchitecture GPU, du nom de Kepler, décrite dans le livre blanc [nVi12]. Cette architecture, désormais gravée en 28nm, comporte plus de 7.1 milliards de transistors, communique avec l'hôte en *PCI-Express 3.0* $\times$16 (une bande passante de 15.75 Go/s).

La nouvelle génération de *Streaming Multiprocessors* est dénommée *SMX*. 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 flottants 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*.

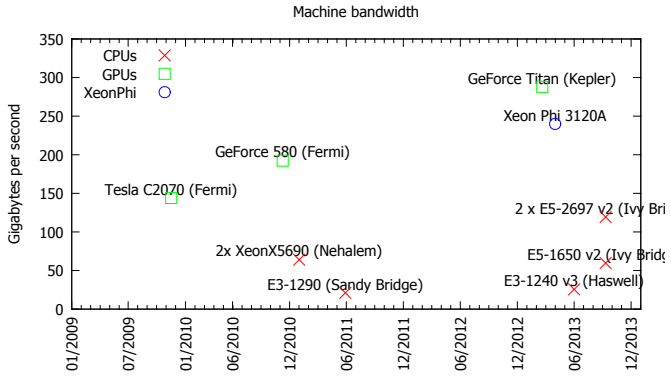
Comme son prédécesseur, l'unité d'exécution reste le *warp* de 32 *threads*. Cependant, les ordonnanceurs des SMX ont connu des améliorations : 4 ordonnanceurs peuvent distribuer jusqu'à 8 instructions par cycle (2 par warp). De plus, les instructions double précision peuvent s'exécuter conjointement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

3.1.6 Résumé des architectures étudiées

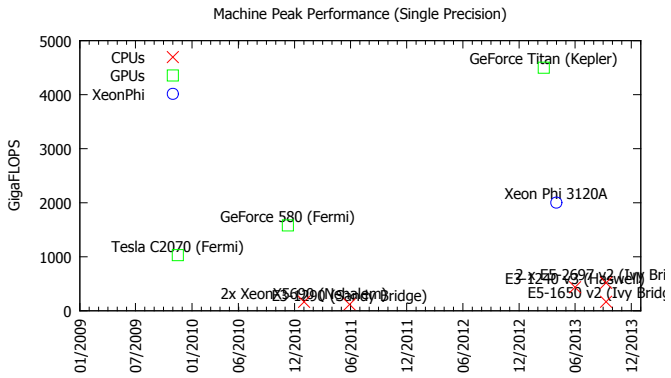
Afin de récapituler l'ensemble des machines étudiées, les graphiques suivants 3.8 présentent les performances en termes de bande passante, d'opérations flottantes à la seconde et de prix.



(a) Prix à la commercialisation



(b) Bande passante



(c) Opérations flottantes à la seconde

FIGURE 3.8 – Performances des architectures étudiées

3.2 Les langages de programmation et les outils associés

Le nombre d'outils pour la programmation parallèle des architectures précédemment présentées (GPP, *manycore* et GPU) est très important. Cette section décrit certains d'entre eux parmi les

plus usités et accessibles depuis un code C et/ou 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.

Afin d'illustrer les techniques utilisées, un code d'exemple calculant la fractale de Mandelbrot va être utilisé. La fractale de Mandelbrot est l'ensemble des points complexes C pour lesquels la suite de nombres complexes, définie par récurrence, est bornée (en module). La récurrence est définie, dans $\mathbb{C}$ par les formules :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + C \end{cases} \tag{3.1}$$

Dans $\mathbb{R}$, elle correspond à :

$$\begin{cases} (x_0, y_0) = (0, 0) \\ x_{n+1} = x_n^2 - y_n^2 + \mathrm{Re}(C) \\ y_{n+1} = 2 \times x_n * y_n + \mathrm{Im}(C) \end{cases} \tag{3.2}$$

La représentation usuelle de cette fractale est illustrée à la figure 3.9 pour chacun des points du plan complexe. Le programme de référence 1 présenté ci-après stocke le nombre d'itérations réalisées pour un point lors du calcul de la fractale.

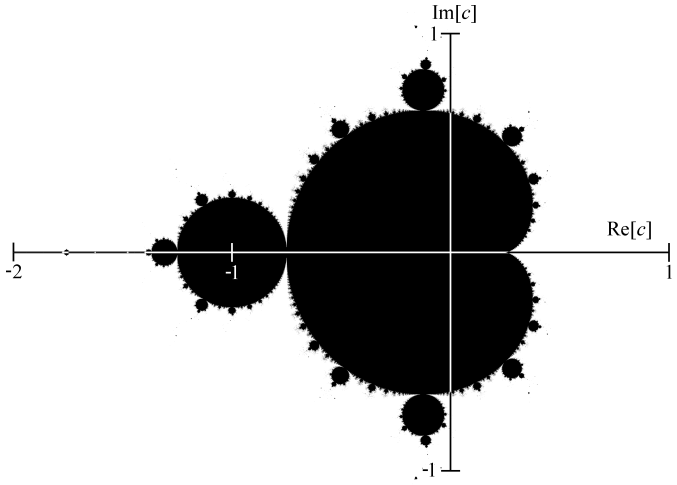


FIGURE 3.9 – Représentation de la fractale de Mandelbrot

Source : Wikipedia

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 moderne pour programmer des ensembles de tâches et ordonnancer leur exécution. Originellement, chaque

Programme 1 Mandelbrot : Boucle Extérieure (scalaire)

```
1          int i, j;
2          float dx, dy;
3          float x, y;
4          uint iter;
5
6          //h et w sont respectivement la hauteur et la largeur de l'image
7          //(x0,y0) et (x1,y1) sont les extrémités de l'image
8          dx = (x1-x0)/w;
9          dy = (y1-y0)/h;
10
11         for(i=0; i<h; i++)
12         {
13                 for{(j=0; j<w; j++)
14                 {
15                         //conversion (i,j) -> (x,y)
16                         x = x0 + j * dx;
17                         y = y0 + i * dy;
18
19                         iter = mandelbrot(x, y, max);
20                         M[i][j] = iter;
21                 }
22         }
23
```

Programme 2 Mandelbrot : Boucle Intérieure (scalaire)

```
1  int mandelbrot(float x0, float y0, int max)
2  {
3          int iter = 0;
4          float xi=x0;
5          float yi=y0;
6          float xi1,yi1;
7          float mod;
8
9          for(iter=0; iter<max; iter++)
10         {
11                 mod=xi*xi+yi*yi;
12                 if(mod>4) break;
13                 xi1=xi*xi-yi*yi+x0;
14                 yi1=2*xi*yi+y0;
15                 xi=xi1;
16                 yi=yi1;
17         }
18         return iter;
19 }
```

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, Unix, BSD, OS X...), les initiatives d'unification au moyen d'encapsulations de plus haut niveau (`Boost.Thread` par exemple) et plus récemment l'unification à travers le langage C++11.

3.2.1.2 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 Board*, consortium regroupant les acteurs principaux de l'informatique matérielle et logicielle. On peut citer entre autre 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 (Windows, Unix, OS X) et différents GPP. 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. 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 3.10 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 iconique de cet outil consiste à découper le domaine d'itérations d'une boucle `for`. Dans ce cas de figure, 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.

Pour illustration dans le cadre du calcul de la fractale de Mandelbrot, avec l'usage d'OpenMP sur la boucle `for` sur les pixels utilisés, une directive `#pragma omp for` vient précéder cette boucle, comme présenté à la ligne 13 du Programme 3.

Le standard OpenMP 2.0 qui existe depuis 2002 (C et C++) est particulièrement adapté au parallélisme de données. Entre 2008 et 2011, sa version 3.0 puis 3.1 a été développée pour s'attaquer aux problématiques de parallélisation de tâches *data-dependant* : il est désormais possible de définir des tâches irrégulières (par exemple, des algorithmes récursifs, ou des schémas producteurs/consommateurs...) que l'ordonnanceur d'OpenMP va répartir selon les différents *threads*. Des directives de compilation et des fonctions de la bibliothèque associée permettent de définir la manière dont ces tâches sont crées et sont organisées leurs barrières de synchronisation. Les spécifications d'OpenMP 4.0 ont été rendues publiques à partir de juillet 2013. Elles permettent de déporter le code à exécuter à partir d'un certain niveau de boucle vers un coprocesseur de calcul spécifique, tel qu'un GPU ou un *Xeon Phi*. Chacun dispose de ses propres *threads* et de sa propre mémoire partagée pour réaliser les calculs qui lui sont assignés. De plus, OpenMP ajoute également des directives destinées à autoriser le compilateur à procéder à des vectorisations sur

Programme 3 Mandelbrot : Boucle Extérieure (scalaire+OpenMP)

```
1  int i, j;
2  float dx, dy;
3  float x, y;
4  uint iter;
5
6  //h et w sont respectivement la hauteur et la largeur de l'image
7  //(x0,y0) et (x1,y1) sont les extrémités de l'image
8  dx = (x1-x0)/w;
9  dy = (y1-y0)/h;
10
11 //Les calculs des itérations de cette boucle sont parallélisées.
12 //On calcule ainsi plusieurs lignes de l'image simultanément.
13 #pragma omp parallel for shared(M,dx,dy,x0,y0) private(x,y,iter)
14 for(i=0; i<h; i++)
15 {
16         for{(j=0; j<w; j++)
17         {
18                 //conversion (i,j) -> (x,y)
19                 x = x0 + j * dx;
20                 y = y0 + i * dy;
21
22                 iter = mandelbrot(x, y, max);
23                 M[i][j] = iter;
24         }
25 }
```

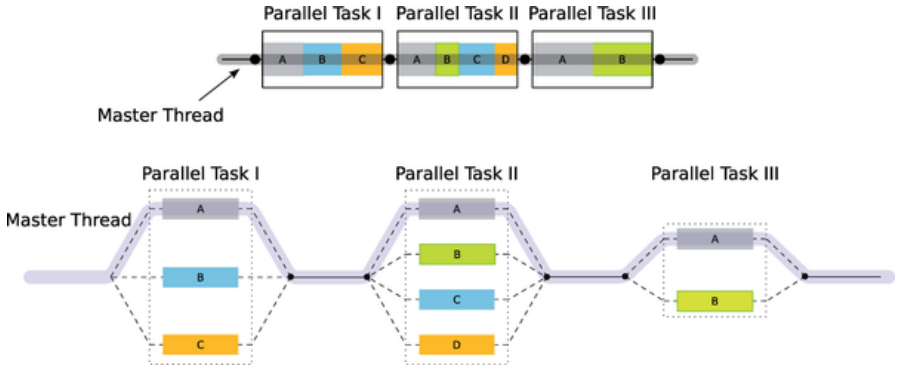


FIGURE 3.10 – Principe de fork/join OpenMP afin de répartir les sections parallèles

Source : Wolfgang Dautermann, Linux Magazine Issue 94/2008

des schémas simples.

3.2.1.3 Intel Cilk+

Intel Cilk+ est un autre outil *multithread*, une extension des langages C et C++, destinée à la programmation parallèle. Il permet l'expression du parallélisme d'un niveau fin, sous la forme de tâches, au moyen d'extensions légères du langage. Cilk+ supporte les architectures x86 (GPP et *Xeon Phi*) et fonctionne sous les systèmes Windows, Linux et OS X.

Cilk+ peut s'adapter aux algorithmes présentant un parallélisme de données comme à ceux présentant un parallélisme de tâches. Il permet la découpe fine d'un problème en un ensemble

de sous-processus qui fonctionnent indépendamment. Son API est très basique pour permettre cette découpe et repose sur les mots clés suivants :

- `cilk_for` crée un ensemble de sous tâches pour les itérations d'une boucle, à exécuter en parallèle ;
- `cilk_spawn` indique qu'une fonction sera exécutée de manière asynchrone ; Cilk+ décide à l'exécution si cette fonction est exécutée en parallèle suivant les emplacements de tâche disponibles ;
- `cilk_sync` indique un point de rendez-vous dans une fonction pour laquelle toutes les sous tâches doivent avoir fini de s'exécuter avant que l'exécution poursuive ; cette synchronisation est implicite lors des fonctions crées avec `cilk_spawn`.

En complément de ces mots clés, Cilk fournit des outils pour faciliter la mise en place de réductions et éviter les inter-blocages, courants dans ce genre de schémas.

Depuis 2010, Intel a développé les capacités de son produit afin d'obtenir des performances encore plus importantes sur ses propres machines. Il est possible de citer :

- Une notation tableau, permettant de travailler facilement sur des vecteurs de taille arbitraire et de réaliser des opérations simples d'algèbre linéaire.
- Une directive de compilation `#pragma simd` qui permet au compilateur de passer outre ses restrictions habituelles pour réaliser l'auto-vectorisation d'une boucle donnée (lorsque par exemple le programmeur est sûr que les accès mémoire réalisés sont réguliers et sans *aliasing*).
- Un SDK pour développer plus facilement des applications Cilk+, en détectant entre autres des erreurs de type concurrence critique ou encore pour analyser le potentiel de passage à l'échelle d'un code.

3.2.1.4 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 GPP.

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 parallelism...). Afin de faciliter le portage d'applications existantes, TBB s'accompagne de containeurs similaires à ceux de la STL (*Standard Template Library*) du C++ pour fournir des solutions réentrantes[4] aux algorithmes usuels. 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.

[4]Un algorithme ou une fonction est réentrant s'il peut être interrompu pendant son exécution et appelé simultanément par plusieurs tâches sans risques pour le bon fonctionnement du logiciel.

3.2.1.5 Instructions intrinsèques SIMD

La gestion des cœurs d'un GPP n'est pas le seul point qu'il est possible de manipuler directement. Les instructions SIMD sont utilisables à très bas niveau, directement en assembleur mais cela s'avère rapidement fastidieux. Une alternative efficace consiste à utiliser des fonctions C appelant directement les instructions intrinsèques qui effectuent une instruction SIMD sur son ou ses 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 la gestion de l'absence de ces instructions. 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 *SIMDization* par opposition à la vectorisation qui consiste à faire ressortir les calculs sous la forme de vecteurs et de matrices. La *SIMDization* est réalisée manuellement, là où la vectorisation est automatisée par le compilateur.

Pour illustrer le codage manuel des instructions SIMD, les algorithmes 4 et 5 présentent, respectivement, les boucles extérieures et intérieures du calcul de la fractale de Mandelbrot pour des instructions SIMD SSE4.2. Les instructions intrinsèques existent sur l'ensemble des processeurs généralistes depuis plusieurs décades, mais chaque fabriquant et chaque génération de GPP disposent de sa ou ses normes et d'instructions spécifiques. Afin d'obtenir des accélérations conséquentes, l'usage de ces instructions nécessite une expertise spécifique sur chaque architecture visée et le développement d'une solution unique et peu portable au delà d'une rétrocompatibilité ascendante au sein des processeurs d'un même fabriquant. Ainsi, sur les GPP Intel gérant les instructions AVX (256 bits), il est possible d'utiliser des instructions SSE (128 bits). Le GPP utilise alors la FPU AVX sur des registres SSE.

Programme 4 Mandelbrot : Boucle Extérieure (SSE4.2)

```
1  typedef __m128i vuint32; // vecteur 128bits d'entiers
2  typedef __m128 vfloat32; // vecteur 128bits de flottants
3
4  int i, j;
5  float dx, dy;
6  float sx, sy;
7  vfloat32 x, y;
8  vuint32 iter;
9
10 dx = (x1-x0)/w;
11 dy = (y1-y0)/h;
12 for(i=0; i<h; i++) {
13         for(j=0; j<w/4; j++)     {
14                 sx = x0 + j * dx * 4;    //conversion (i,j) -> (x,y)
15                 sy = y0 + i * dy ;               //calcul 4 par 4 des pixels
16                 x = _mm_setr_ps(sx, sx+dx, sx+2*dx, sx+3*dx); //vecteur x
17                 y = _mm_setr_ps(sy, sy, sy, sy); //vecteur y
18                 iter = mandelbrot_sse42(x, y, max);
19                 M[i][j] = iter; // suppose un alignement mémoire
20         }
21 }
```

3.2.1.6 Boost.SIMD

Boost.SIMDse présente sous la forme d'une bibliothèque de *templates C++* open-source qui offre au développeur une approche simple d'utilisation mais puissante pour utiliser les instructions SIMD au moyen d'abstractions typées objet [EGF+12].

Programme 5 Mandelbrot : Boucle Intérieure (SSE4.2)

```
1  typedef __m128i vuint32; // vecteur 128bits d'entiers
2  typedef __m128 vfloat32; // vecteur 128bits de flottants
3
4  int mandelbrot_sse42(vfloat32 x0, vfloat32 y0, int max){
5          vuint32 iter = _mm_set1_epi32(0);
6          const vfloat32 mod_max = _mm_set1_ps(4);
7
8          vfloat32 xi=x0;
9          vfloat32 yi=y0;
10         //precalcul pour la multiplication complexe
11         vfloat32 x2=_mm_mul_ps(xi,xi);
12         vfloat32 y2=_mm_mul_ps(yi,yi);
13         vfloat32 xy=_mm_mul_ps(xi,yi);
14         int mask=0xF;
15
16         for(iter=0; iter<max; iter++)
17         {
18                 //multiplication complexe
19                 vfloat32 yi1 = _mm_add_ps(xy,xy); // yi1 = 2*xi*yi + y0
20                 vfloat32 mod = _mm_add_ps(x2,y2);
21                 vfloat32 xi1 = _mm_sub_ps(x2,y2); // xi1 = xi*xi-yi*yi+ x0
22                 vfloat32 var_cmp=_mm_cmplt_ps(mod,mod_max);
23
24                 yi=_mm_add_ps(yi1,y0);//addition complexe
25                 xi=_mm_add_ps(xi1,x0);//pour obtenir la nouvelle iteration (x
       ,y)
26                 //multiplication complexe
27                 xy = _mm_mul_ps(xi,yi);
28                 y2 = _mm_mul_ps(yi,yi);
29                 x2 = _mm_mul_ps(xi,xi);
30
31                 //evolution des variables
32                 vuint32 var_cmpi=_mm_castps_si128(var_cmp);
33                 iter=_mm_sub_epi32(iter,var_cmpi);
34
35                 mask&=_mm_movemask_ps(var_cmp); //mask si modules <4
36                 if(!mask) break; //sortie prematurée si tous les pixels
       depassent 4
37         }
38         return iter;
39 }
```

Celle-ci fournit au développeur des méthodes encapsulant les fonctionnalités classiques disponibles dans les jeux d'instructions SIMD les plus répandus. A la compilation, au moyen de *templates*, le code généré fait appel à l'instruction SIMD correspondante si elle existe sur la plateforme cible, ou à un code équivalent, utilisant les instructions disponibles, le plus efficacement possible. Boost.SIMD permet d'utiliser ainsi des méthodes de haut niveau, faisant apparaitre l'algorithmie plutôt que de la noyer sous un ensemble d'instructions spécifiques à l'architecture. De plus, le code écrit est portable sur l'ensemble des plateformes supportées par Boost.SIMD. Les jeux d'instructions supportés sont nombreux :

processeurs Intel/AMD x86 SSE, SSE2, SSE3, SSSE3, SSE4.1 et SSE4.2, AVX, AVX2, XOP, FMA3, FMA4
processeurs Intel Xeon Phi MIC
processeurs IBM PowerPC VMX, VSX (Altivec)

processeurs IBM BlueGene-Q QPX
processeurs de type ARM NEON
processeurs TI C6x

Boost.SIMD offre également toutes les routines nécessaires à l'utilisation des registres SIMD dans les algorithmes standards de la *Standard Template Library (STL)* du C++. Les Programmes 6 et 7 présentent le code SIMD réalisé par l'intermédiaire de Boost.SIMD. A la lecture, ceux-ci ont beau respecter les contraintes de régularité SIMD (il faut itérer tant que tous les pixels n'ont pas atteint la condition de sortie), le code est beaucoup plus lisible que le code équivalent utilisant les fonctions intrinsèques. De plus, à la compilation, il s'adapte automatiquement aux instructions supportées sur la machine.

Programme 6 Mandelbrot : Boucle Extérieure (Boost.SIMD)

```
1  int i, j;
2  float dx, dy;
3  float sx, sy;
4  sidm::pack<float> x, y;
5  sidm::pack<int> iter;
6  int size = simd::pack<float>::static_size;
7  //Permet de traiter "size" pixel a la fois, en fonction du GPP visé
8  dx = (x1-x0)/w;
9  dy = (y1-y0)/h;
10 for(i=0; i<h; i++) {
11         for(j=0; j<w/size; j++)          {
12                 sx = x0 + j * dx * size;                //conversion (i,j) ->
       (x,y)
13                 sy = y0 + i * dy ;
14
15                 x = simd::enumerate<float>(sx, dx); //vecteur x
16                 y = sidm::pack<float>(sy, sy, sy, sy); //vecteur y
17                 iter = mandelbrot_boost(x, y, max);
18                 M[i][j] = iter;
19         }
20 }
```

3.2.1.7 Intel SPMD Program Compiler (ISPC)

Les laboratoires d'Intel ont développé un compilateur spécifique, **ISPC**, pour faciliter l'accès aux performances des instructions SIMD. Cependant il ne fait pas partie de la suite d'outils de compilation standard de Intel Compiler.

Cet outil prend en entrée des codes dans une variante du langage C dans lesquels l'algorithme est décrit selon le modèle SPMD (*Single Program, Multiple Data*). Le code de cet algorithme ressemble à une version scalaire, mais est décoré de manière à faire ressortir les informations nécessaires à sa transformation automatique. Il y est précisé quelle variable est constante pour toutes les itérations, comment les boucles sont à répartir sur la largeur des instructions SIMD... Le code compilé par ISPC est créé dans sa propre unité de compilation et doit être déclaré pour l'établissement des liens. Lors de la compilation, il faut préciser à l'outil le jeu d'instructions SIMD ciblé. Le code alors généré est interopérable avec une application classique et peut être inclus lors de l'édition des liens pour obtenir l'application accélérée finale.

Selon Intel, son usage permet d'atteindre aisément une accélération de l'ordre de $\times 3$ sur une architecture SIMD de 4 données de large (SSE...) et une accélération avoisinant $\times 5$ à $\times 6$ sur architecture de largeur 8 (AVX, AVX2...). Il est à noter que *ISPC* est une technologie ouverte,

Programme 7 Mandelbrot : Boucle Intérieure (Boost.SIMD)

```
1  simd::pack<int> mandelbrot_boost(simd::pack<float>x0, simd::pack<float>y0,
       int max)
2  {
3          simd::pack<int> iter(0);
4          const simd::pack<float> mmax(4.);
5
6          simd::pack<float> xi = x0;
7          simd::pack<float> yi = y0;
8          simd::pack<float> x2 = xi*xi;
9          simd::pack<float> y2 = yi*yi;
10         simd::pack<float> xy = xi*yi;
11         int mask=0xF;
12         for(int iter_scal=0; iter_scal<max; iter_scal++) {
13                 simd::pack<float> yi1 = xy+xy; // yi1 = 2*xi*yi + y0
14                 simd::pack<float> mod = x2+y2;
15                 simd::pack<float> xi1 = x2-y2; // xi1 = xi*xi - yi*yi + x0
16                 simd::pack<simd::logical<float>> var_cmp = simd::is_less(mod,
       mmax);
17
18                 yi = yi1+y0;
19                 xi = xi1+x0;
20
21                 xy = xi*yi;
22                 y2 = yi*yi;
23                 x2 = xi*xi;
24
25                 iter = simd::seladd(var_cmp, iter, c1); //iter+1 si modules <
        4
26                 int a = simd::hmsb(simd::genmask(var_cmp));
27                 mask &= a;
28                 if(!mask) break;
29         }
30         return iter;
31 }
```

ainsi il a été possible de l'utiliser pour générer des instructions Altivec de cardinal 4 sur ses GPP Power/Power PC [LEHZ+14][BBFT14].

Pour illustrer l'usage de *ISPC*, le Programme 8 présente la routine spécifique de calcul Mandelbrot qu'il faut fournir à ISPC afin qu'il génère le fichier binaire optimisé. L'usage du mot clé `uniform` permet de qualifier des variables dont la valeur sera identique tout au long de la vie de la parallélisation. Le mot clé `foreach` permet de répartir une ligne de pixels sur les instructions SIMD utilisées de manière automatique (cette méthode définit alors elle même le type de la variable `i`).

3.2.1.8 CUDA (Compute Unified Device Architecture)

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 comme un coprocesseur du CPU hôte, 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 et la manière dont les *threads* de calcul vont se répartir la tâche et

Programme 8 Mandelbrot : Boucle Extérieure (ISPC)

```
1  export void mandelbrot_ispc(uniform float x0, uniform float y0, uniform float
       x1, uniform float y1, uniform int width, uniform int height, uniform int
       maxIterations, uniform int output[])
2  {
3          float dx = (x1 - x0) / width;
4          float dy = (y1 - y0) / height;
5          for (uniform int j = 0; j < height; j++) {
6                  foreach (i = 0 ... width) {
7                  float x = x0 + i * dx;
8                  float y = y0 + j * dy;
9
10                 int index = j * width + i;
11                 output[index] = mandelbrot_ispc(x, y, maxIterations);
12                 }
13         }
14 }
```

ses sous-divisions sont gérés au lancement de noyaux de manière dynamique. Ces noyaux sont écrits en un sous-ensemble de C et/ou 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 (*c.f.* le modèle CUDA sur les répartitions possibles de l'espace de calcul en différents maillages, section 3.2.1.9).

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 atomique[5] ou des fonctions de synchronisation entre des *threads* collaborant. 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.

3.2.1.9 Le modèle de calcul CUDA

Le noyau 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, chaque bloc contenant un ensemble de *threads*. La figure 3.11 présente différentes configurations possibles pour les blocs et les grilles CUDA d'un noyau.

Lors de l'exécution d'un noyau, les différents *threads* de chaque bloc sont alors divisés en *warp* de 32 *threads* contigus (le dernier *warp* est complété si nécessaire par des *threads* « vides » : *padding*) comme présenté par la figure 3.12. 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 branchements...). 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 le lui permettent.

Pour permettre la communication entre les différents *threads*, CUDA fournit trois niveaux de synchronisation :

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

[5] Une opération atomique est une opération qui s'effectue sans pouvoir être interrompue par les autres *threads*, il s'agit le plus souvent d'opérations de calcul sur des données en mémoire

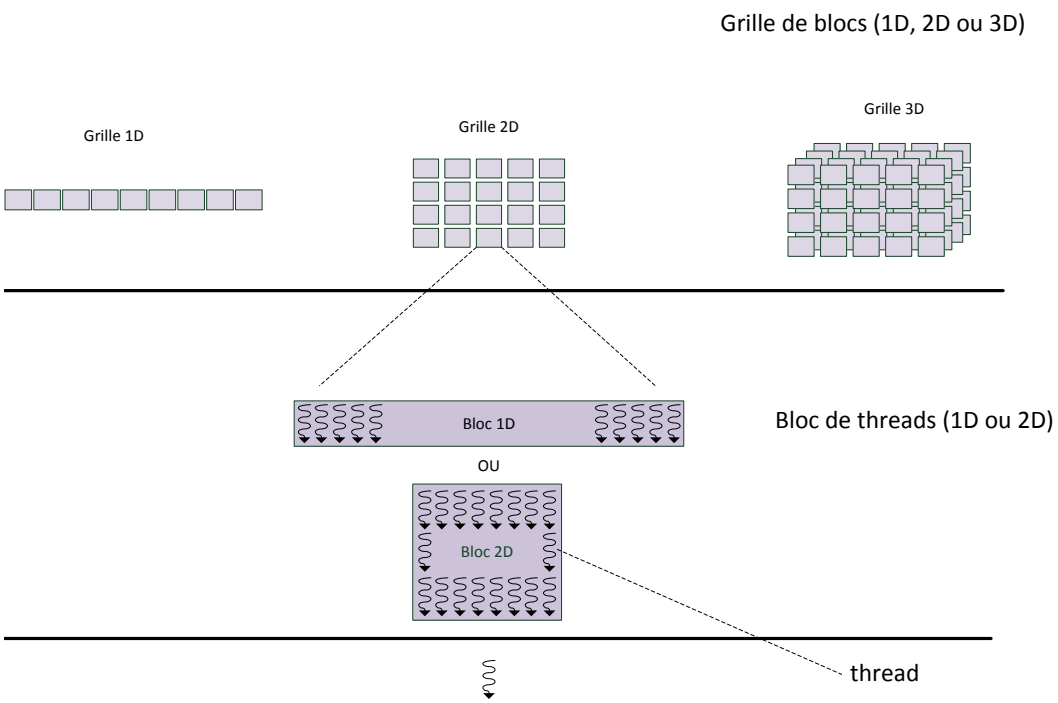


FIGURE 3.11 – Découpe de l'espace de calcul selon le modèle CUDA : Grilles et blocs

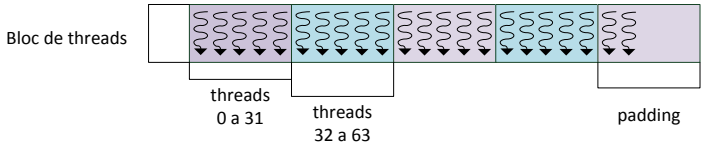


FIGURE 3.12 – Division d'un bloc en *warps*

- Une synchronisation *a posteriori* de l'exécution d'un noyau : CUDA garantit que tous les blocs se sont exécutés ; cela permet de synchroniser entre eux plusieurs noyaux successifs exécutés à la chaîne sur le même GPU.
- Enfin il est possible de synchroniser le GPU et le GPP (le plus souvent pour récupérer le résultat de traitements) une fois un ensemble de noyaux exécuté.

Les Programmes 9 et 10 illustrent l'usage de CUDA pour le calcul de la fractale de Mandelbrot. On observe sur le premier la similarité entre les fonctions CUDA et les fonctions écrites en C, où seul diffère la décoration `__device__`. Le noyau, quant à lui décoré d'un `__global__`, est appelé par une séquence spécifiant les paramètres de la grille de calcul, côté hôte, comme présenté par le Programme 11, afin d'attribuer à chaque bloc CUDA une ligne de pixels de l'image. Le passage de paramètres de la grille de threads au noyau est indiqué au moyen d'un chevron triple `<<< tailleGrille, tailleBloc >>> Noyau(...)`.

Avec les versions antérieures de CUDA, il était impossible de lancer des noyaux de calcul imbriqués. Depuis la Compute Capability 3.5 (introduit par la microarchitecture Kepler), nVidia a introduit plusieurs possibilités : une permet de créer des noyaux imbriqués et/ou récursifs

Programme 9 Mandelbrot : boucle interne CUDA

```
1  template<class T> __device__ inline int CalcMandelbrot(const T x0, const T y0
       , const int max)
2  {       //Type T variable par template
3          T x, y, xx, yy, xC, yC ;
4          //Initialisation
5          xC = x0 ;
6          yC = y0 ;
7          y = 0 ;
8          x = 0 ;
9          yy = 0 ;
10         xx = 0 ;
11
12         int iter;
13         for(iter=0; iter<max && (xx + yy < T(4.f); iter++)
14         {       //Multiplication complexe et ajout de C
15                 y = x * y * T(2.f) + yC ;
16                 x = xx - yy + xC ;
17                 yy = y * y;
18                 xx = x * x;
19         }
20
21         return iter;
22 } // CalcMandelbrot
```

Programme 10 Noyau CUDA de calcul Mandelbrot (inspiré des *CUDA Samples* fournis par nVidia)

```
1  template<class T>
2  __global__ void Mandelbrot(int *dst, const int imageW, const int imageH,
       const int max, const T xOff, const T yOff, const T dx, const T dy)
3  {
4          // position dans l'image (calcul par tuile)
5          const int ix = blockDim.x * blockIdx.x + threadIdx.x;
6          const int iy = blockDim.y * blockIdx.y + threadIdx.y;
7          //Chaque thread du bloc correspond a un pixel different
8          if ((ix < imageW) & (iy < imageH))
9          {
10                 // Position en float
11                 const T xPos = (T)ix * dx+ xOff;
12                 const T yPos = (T)iy * dy+ yOff;
13
14                 // Mandelbrot de la position courante
15                 int m = CalcMandelbrot<T>(xPos, yPos, max);
16
17                 int pixel = imageW * iy + ix;
18                 dst[pixel] = m;
19         }
20 } // Mandelbrot
```

(*Dynamic Parallelism*), afin de faciliter l'exploration d'espaces de calculs non réguliers en densité. Une autre est destinée à faire s'exécuter plusieurs noyaux sur un même GPU (*Hyper-Q*) afin de profiter de tous les SM disponibles sans devoir attendre la fin de l'exécution d'un noyau précédent. Cette possibilité, renforcée par les capacités d'ordonnancement entre plusieurs grilles distinctes, est développée par la technologie *Grid Management Unit*, afin de maximiser toujours

Programme 11 Code CUDA hôte d'appel au noyau Mandelbrot

```
1 dim3 threads(BLOCKDIM_X, BLOCKDIM_Y);
2 dim3 grid(iDivUp(imageW, BLOCKDIM_X), iDivUp(imageH, BLOCKDIM_Y));
3
4 Mandelbrot<float><<< grid, threads >>>(dst, imageW, imageH, max, (float)xOff,
      (float)yOff, dx, dy);
```

plus l'utilisation d'un GPU.

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. Deux principaux outils disponibles pour cibler les architectures GPP et GPU sont détaillés ici.

3.2.2.1 OpenCL

Le standard OpenCL est un standard ouvert défini par le consortium *Khronos Group* qui regroupe un ensemble d'industriels pour la création de standards ouverts et gratuits pour la lecture de contenus multimédia sur de nombreuses plateformes. Parmi les plus connus, il est possible de citer : AMD, Apple Inc., ARM Holdings, Creative Labs, id Software, Ericsson, Google, Intel, Mozilla Foundation, nVidia, Samsung Electronics, Sony Computer Entertainment, Oracle, Texas Instruments. Ce consortium est à l'origine des standards OpenGL et WebGL.

OpenCL offre aux développeurs un environnement uniformisé pour programmer différents types de machines : les GPP 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.

La programmation OpenCL repose sur le modèle d'abstraction présenté à la figure 3.13. Il est à noter une grande similarité avec le modèle CUDA, dont il est visiblement inspiré (nVidia faisant partie du *Khronos Group*).

- Les programmes OpenCL sont, d'une part exécutés par l'hôte (*host*) qui distribue les noyaux de calculs aux différents composants (matériels compatibles OpenCL). Il s'agit de l'ordinateur sur lequel sont connectés les différents composants compatibles OpenCL (*device* dans le jargon OpenCL). D'autre part, par les composants eux même, qui exécutent des noyaux de façon massivement parallèle. L'API OpenCL fournit côté hôte un ensemble de bibliothèques pour sélectionner et gérer les tâches de calculs réparties sur les multiples composants.
- Les noyaux sont les fonctions de calculs traitant le parallélisme de données ou de tâches. Des appels aux directives spécifiques d'OpenCL permettent de déterminer la manière dont ces noyaux sont exécutés sur les composants (nombre de tâches de calcul, mémoire requise...).
- Les tâches de calcul sont décomposées hiérarchiquement sur l'ensemble du domaine de calcul. Ces décompositions, le long d'une grille à N dimensions, forment un ensemble de *work groups*. Ces *work groups* sont ensuite à nouveau décomposés en tâches unitaires appelées *work items*. Cette découpe suit le schéma présenté à la figure 3.14. Une grille de tâches

est répartie sur le *device* matériel OpenCL sélectionné, chaque *work group* à l'intérieur de celle-ci est assigné à un seul *compute unit*. Les *work items* sont exécutés par les *processing elements* jusqu'à ce que l'intégralité du *work group* soit traité.

- Afin de modéliser les accès mémoire au niveau du composant pendant l'exécution du noyau, OpenCL fournit un modèle de mémoire accompagnant le domaine. Cette structure mémoire, fortement inspirée de la hiérarchie mémoire des GPU, fournit cependant une abstraction suffisamment généralisable pour s'adapter aux différentes architectures supportant OpenCL.

  La figure 3.15 présente la hiérarchie mémoire correspondant à un Compute Device OpenCL.

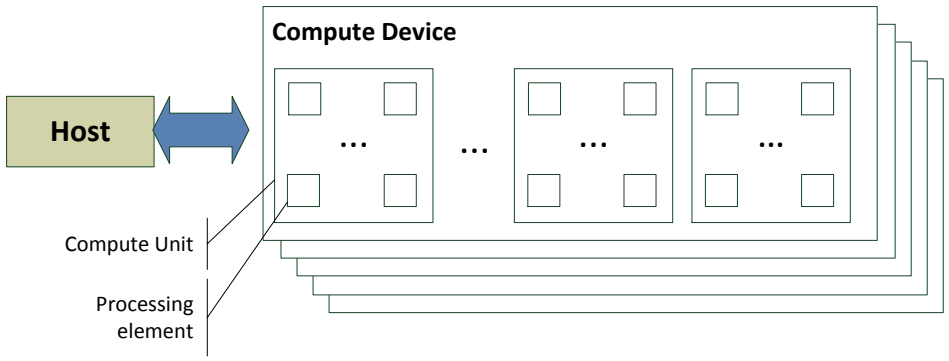


FIGURE 3.13 – Modèle matériel général d'OpenCL

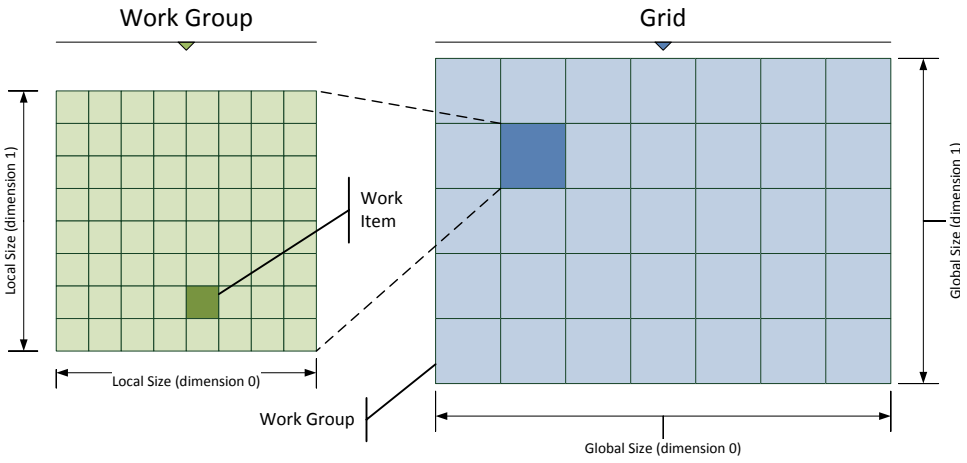


FIGURE 3.14 – Modèle de répartition des tâches de calcul d'OpenCL

Les Programmes 12 et 13 présentent le code générique du noyau de calcul OpenCL pour réaliser le calcul d'une fractale de Mandelbrot. Les codes s'apparentent, comme en CUDA, à du C et/ou C++ décoré. Il est à noter la fonction `get_global_id(n)` qui permet d'obtenir l'indice total d'un *thread* au sein de la grille, selon la dimension `n`.

Cette volonté d'homogénéisation permet ainsi d'exécuter sur des matériels hétérogènes le même noyau de calcul. Chaque matériel compatible OpenCL fournit un compilateur et un driver

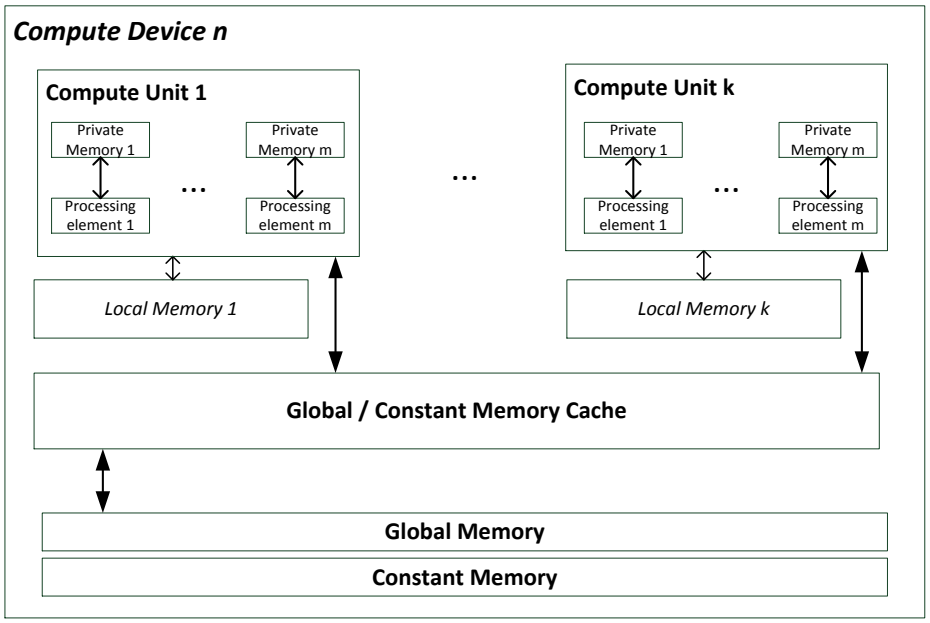


FIGURE 3.15 – Modèle matériel d'un périphérique compatible OpenCL

Programme 12 Mandelbrot : boucle interne OpenCL

```
1  int CalcMandelbrot(float x0, float y0, const int max)
2  {
3          int iter = 0;
4          float xn = x0;
5          float yn = y0;
6          for(iter=0; iter<max; iter++)
7          {
8                  if(xn*xn+yn*yn > 4.f)
9                          break;
10                 float xn1 = xn*xn -yn*yn;
11                 float yn1 = 2*xn*yn;
12                 xn = xn1 + xn;
13                 yn = yn1 + yn;
14         }
15         return iter;
16 } // CalcMandelbrot
```

à la machine hôte, s'occupant respectivement de compiler dans un format binaire spécifique au matériel puis d'exécuter/gérer les noyaux sur le *device*. Ainsi, une même version du noyau en termes de code/fichier source, peut être répartie sur plusieurs matériels compatibles OpenCL différents.

Cependant, l'aspect performance nécessite très souvent des adaptations très spécifiques aux catégories de matériels, en termes de gestion mémoire (stratégie d'accès mémoire pour les GPP) et de répartition des sous tâches sur le domaine de calcul. Il est souvent coûteux lorsque c'est possible et en général impossible (dans des conditions de qualité de code industriel) d'obtenir un seul et même code optimal pour de multiples plateformes.

Programme 13 Noyau OpenCL de calcul Mandelbrot

```
1  void Mandelbrot(__global int image*, int imageW, int imageH, float x0, float
       y0, float dx, float dy)
2  {
3          // process this block
4          const int ix = get_global_id(0);
5          const int iy = get_global_id(1);
6
7          if ((ix < imageW) && (iy < imageH))
8          {
9                  float cx = x0 + ix * dx;
10                 float cy = y0 + iy * dy;
11                 // Calculate the Mandelbrot index for the current location
12                 int m = CalcMandelbrot<T>(cx, cy, max);
13
14                 // Convert the Mandelbrot index into a color
15
16                 // Output the pixel
17                 int pixel = imageW * iy + ix;
18
19                 image[pixel] = m;
20         }
21 } // Mandelbrot
22
```

3.2.2.2 C++ Accelerated Massive Parallelism

(**C++Amp**) est un modèle de programmation native basé sur du C++ permettant de cibler différents composants adaptés au parallélisme de données tels que les GPU. L'objectif est de proposer une solution simple à utiliser, en fournissant une auto-optimisation lorsque l'utilisateur ne précise aucun paramètre, tout en laissant la possibilité à un expert de paramétrer finement son code. C++Amp est développé par Microsoft mais ses spécifications sont libres pour encourager d'autres acteurs à implémenter ce standard sur leurs propres architectures [Mic13].

La solution proposée par Microsoft prend la forme d'une bibliothèque basée sur l'API DirectX 11 permettant de viser l'ensemble des GPU grand public supportés par son système d'exploitation Windows. Dans cette solution, lorsque les codes ne peuvent pas s'exécuter sur le ou les GPU de la machine (par exemple, par manque de mémoire), un repli est effectué sur le GPP de la machine en exécutant une version SIMD (en SSE) du code.

D'autres implémentations commencent à apparaitre pour offrir plus de modularité aux solutions utilisant C++Amp. Par exemple, fin 2013, HSA Foundation a mis à disposition une implémentation générant des instructions OpenCL pour garantir une portabilité plus importante. Fin août 2014 AMD a annoncé la mise à disposition d'un compilateur Open Source C++Amp 1.2 pour les plateformes Linux et Windows visant les architectures GPP comme GPU.

A l'usage, il est possible d'inclure dans du code C++ toutes les instructions pour gérer les données sur les différents accélérateurs et de déporter les traitements et les fonctions qui doivent y être appliqués.

3.2.3 Les bibliothèques natives

Dans le cadre du calcul scientifique, il existe plusieurs bibliothèques regroupant des calculs ayant trait aux calculs mathématiques sur de grands ensembles de données.

3.2.3.1 FFTW

L'implémentation **Fastest Fourier Transform in the West** est l'implémentation de la Transformée de Fourier Rapide (FFT de *Fast Fourier Transform*), considérée comme étant l'implémentation libre la plus performante. Cette bibliothèque de calcul, développée par Matteo Frigo et Steven G. Johnson du Massachusetts Institute of Technology (MIT) [FJ05], est distribuée sous la license libre GNU-GPL mais des licences commerciales peuvent être obtenues auprès du MIT.

Le noyau de calcul se base sur l'algorithme classique de Cooley-Tukey [CT65]. La bibliothèque est écrite en C et fait appel aux instructions SIMD disponibles sur les GPP. Son fonctionnement repose sur une analyse statistique des performances de la machine hôte réalisée lors de l'installation de la bibliothèque. En fonction de cette analyse, des opérations souhaitées et des paramètres numériques, le calcul est alors dirigé vers la routine la plus optimale au regard de ces paramètres.

En pratique, l'utilisation de FFTW se fait en deux étapes, avec une première phase consistant à construire un "plan" de FFTW, c'est à dire un pré-calcul de coefficients pour une taille de signal donnée. Ce plan est ensuite utilisé pour calculer la transformée en elle même. Pour des signaux ayant les mêmes caractéristiques, il est possible de réutiliser ce plan pour gagner en vitesse.

Dans le cas d'une application parallélisée, FFTW permet de réutiliser un même plan pour des signaux ayant les mêmes caractéristiques et de le partager entre plusieurs *threads* pour économiser en temps d'initialisation et en mémoire. Les appels aux fonctions de calcul de FFT de cette bibliothèque sont réentrants et peuvent être répartis sur des *threads* différents.

3.2.3.2 Intel MKL

(*Intel Math Kernel Library*) est une bibliothèque regroupant un ensemble de fonctions optimisées, destinée à accélérer le développement d'applications parallèles performantes. Les routines disponibles sont optimisées manuellement pour obtenir les meilleurs performances sur les différents processeurs Intel. Selon le fabricant, sur ses processeurs, des *benchmarks* indiquent que cette bibliothèque est la plus performante (par rapport à FFTW).

Elle offre entre autres des versions optimisées des opérations couramment utilisées dans les applications scientifiques. Sont notamment optimisées les manipulations de larges jeux de données matricielles et/ou vectorielles (primitives BLAS de niveau 1,2 et 3), les solveurs LAPACK, des Transformées de Fourier Rapide... Cette bibliothèque supporte les environnements Linux, Windows et OS X. Il est nécessaire d'obtenir une licence commerciale pour chaque machine de développement, mais les composants binaires sur lesquels repose l'application développée peuvent être redistribués sans restrictions particulières (usage commercial compris).

Concernant son algorithme de calcul de FFT, MKL fonctionne de manière analogue à FFTW avec une construction de plan puis le calcul à proprement dit. En termes de parallélisation *multithread*, la MKL propose deux approches : soit un appel réentrant dans le contexte d'un code parallélisé, soit une fonction appelée une seule fois réalisant en un appel l'exécution parallélisée. Les plans de calcul, quant à eux, peuvent être partagés entre plusieurs *threads* ou dupliqués en fonction des besoins. Il existe une encapsulation des fonctions MKL qui permet d'utiliser les conventions d'appel FFTW pour faciliter le portage d'un code vers la bibliothèque MKL.

3.2.3.3 cuFFT

Il s'agit de l'implémentation nVidia CUDA de la Transformée de Fourier Rapide, à destination de ses GPU. Cette bibliothèque, basée sur l'algorithme classique de Cooley-Tukey et Blustein, permet le calcul collaboratif d'une FFT par les différents cœurs du GPU à chaque exécution.

Le fonctionnement de celle-ci s'apparente aux fonctionnements de MKL et de FFTW : création d'un plan pour un signal d'une taille donnée puis exécution de la transformation. Sur GPU, ce plan réside sur la mémoire globale du GPU. Pour optimiser les accès mémoire à l'exécution sur le GPU (et donc le temps de calcul), cuFFT optimise la création du ou des plans qui lui seront nécessaires en fonction du nombre et de la taille des signaux à traiter.

Cependant, cuFFT est adaptée au calcul de blocs composés de plusieurs signaux identiques sur lesquels la même opération est appliquée au moyen d'un seul appel côté hôte. Cela permet de tirer parti de toute les performances d'un GPU au prix d'un certaine rigidité. Ainsi, il est à noter que cuFFT se prête bien aux calculs sur des données résidentes sur le GPU, mais que l'éventuel coût des transferts mémoire peut être un frein à son utilité s'il faut transférer beaucoup de signaux avant de leur appliquer une opération. Enfin, un autre facteur de rigidité est l'impossibilité d'appeler une opération de cuFFT depuis l'intérieur d'un noyau.

Dans le cadre du contrôle non destructif, on peut remarquer que cuFFT est particulièrement capable de réaliser les opérations de calcul de FFT sur de multiple signaux de mêmes caractéristiques (format, taille).

3.2.4 Les compilateurs

Il est a 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 (*multithread*) qui détecte les sections data-parallélisables d'un code pour les déporter sur plusieurs *threads* de calculs en parallèle ;

auto 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 ;

optimisation inter procédurale qui vise à prendre en compte l'imbrication des fonctions pour appliquer des optimisations supplémentaires aux instructions.

L'auto vectorisation et l'auto-parallélisation montrent des résultats encourageants. Mais les compilateurs modernes tels que le Intel C++ Compiler, GCC ou encore CLang, 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.

Il est à noter que seul ICC (*Intel C++ Compiler*) permet de compiler des codes natifs à destination des architectures *Xeon Phi*.

3.3 Industrialisation des codes parallèles en vue de l'intégration dans un logiciel commercial

Les travaux réalisés dans le cadre de cette thèse visent à développer une maquette de simulation de contrôle ultrasonore interactif en vue de la réutilisation de ces travaux et algorithmes pour une industrialisation après la fin de la thèse. Dans cette section, un rappel sur l'existant du

logiciel CIVA est réalisé, puis des considérations de génie logiciel seront évoquées pour appuyer les choix concernant les bibliothèques et les outils retenus.

3.3.1 L'existant du logiciel CIVA

Le logiciel CIVA est codé dans différents langages interfacés entre eux. La plateforme (interface homme-machine, gestion de fichiers, visualisation des données résultats, gestion du modèle de données...) est réalisée en Java. Cette plateforme fait appel à des noyaux de calcul natifs développés pour la plupart en C++, Matlab et parfois Fortran, se présentant physiquement sous forme de bibliothèques dynamiques. CIVA s'installe et s'exécute sur des ordinateurs compatibles x86 possédant une version récente du système d'exploitation Windows (32 et 64 bits).

Plusieurs algorithmes faisant appels aux GPU de marque nVidia ont été intégrés dans CIVA. Des codes de calcul en tomographie utilisent des algorithmes écrits en CUDA pour accélérer certaines simulations. Antoine Pedron a, dans le cadre de sa thèse [Ped13], développé des versions GPU/OpenCL d'algorithmes rapides de reconstruction de Vue Vraies Cumulées et de Focalisation en Tous Points, tirant parti des architectures GPU. Ces codes sont intégrés dans CIVA 11, en vue de leur utilisation dans CIVA Acquire.

3.3.2 Qualité du logiciel

La **Qualité du logiciel** est une appréciation globale d'un produit logiciel. La norme *ISO/CEI 9126* définit six groupes d'indicateurs de qualité des logiciels :

- **la capacité fonctionnelle**, réponse aux besoins de fonctionnalités de l'utilisateur final.
- **la facilité d'utilisation**, niveau de l'effort à fournir par un nouvel utilisateur du logiciel.
- **la fiabilité**, capacité du logiciel à fournir des résultats corrects y compris lorsque les conditions d'exécution ne sont pas optimales (par exemple lorsque la mémoire est trop faible).
- **la maintenabilité**, mesure de l'effort nécessaire à fournir pour modifier, transformer ou corriger le logiciel.
- **la performance**, rapport entre les ressources mises à disposition pour l'exécution de l'application et le temps de calcul.
- **la portabilité**, capacité du logiciel à s'exécuter sur des systèmes d'exploitations et des matériels variés.

Bien évidement, un logiciel doit tendre à un niveau de qualité optimal pour chacun de ces critères. Cependant, en pratique, il faut souvent réaliser des choix et faire des compromis sur le niveau de qualité de ces critères en fonction des priorités.

3.3.3 Choix techniques - Maquettage hors CIVA

Il est décidé, pour avoir toute la latitude nécessaire à l'optimisation, de réaliser une maquette en dehors de la plateforme CIVA. Cette autonomie permet également une plus grande marge de manœuvre dans le choix des outils utilisés. Cependant, ces travaux sont réalisés en gardant à l'esprit l'objectif d'industrialisation post-thèse au sein de CIVA. Les solutions soit non pérennes (*proof of concept*, utilisation de technologies exotiques), soit trop éloignées de l'existant sont donc écartées.

Dans le cadre de ces travaux, les critères de **performance** et de **maintenabilité** (lisibilité du code) sont privilégiés pour la phase de développement, afin de trouver un juste milieu entre

performance et code bas niveau. Lors du développement, la fiabilité des résultats est constamment vérifiée à l'aide de critères métier[6].

Les problématiques de **capacité fonctionnelle** et de **facilité d'utilisation** seront développés *a posteriori*, lors de l'intégration dans la plateforme CIVA. La première version de la maquette, mise au point au cours de cette thèse, fonctionne en ligne de commande à partir d'un fichier décrivant la configuration du contrôle.

3.3.3.1 Architectures ciblées

Cette maquette 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 GPP souvent plus puissants que les ordinateurs personnels grand public et disposant potentiellement de GPU destinés au calcul. En particulier, ces travaux s'intéressent aux architectures suivantes :

GPP Intel compatible x86 déjà supportés par les versions courantes de CIVA et qui représentent une part majoritaire des GPP dans le monde des stations de calcul.

GPU de marque nVidia compatible CUDA présentant une architecture stable depuis 2008 et disposant d'un support du constructeur pour le renouvellement et la mise à jour des outils de développement. A noter que des codes spécifiques à ces GPU sont déjà intégrés pour certaines fonctionnalités de CIVA.

Coprocesseurs Intel Xeon Phi dont le développement semble activement soutenu par Intel. Ces coprocesseurs se placent sur un segment similaire à celui des GPU (des coprocesseurs généralistes financièrement abordables).

Afin de pouvoir mesurer les performances sur un plus grand panel de machines disponibles (multiples générations de GPP, de GPU et de *Xeon Phi*), la maquette a été développée de manière portable (multi systèmes d'exploitation) La majorité des outils présentés plus haut permettent de tirer parti des performances de ces architectures mais sont aussi compatibles avec d'autres architectures processeur. Ces capacités de portabilité permettraient également le développement d'une maquette ciblant d'autres architectures, par exemple des processeurs Power/PowerPC et/ou ARM, à moindre coût.

3.3.3.2 Utilisations d'outils natifs

Ces travaux prennent place dans le cadre d'une étude de l'adéquation algorithme/architecture des simulations de contrôle non destructif par ultrasons. Les outils hybrides n'ont pas été retenus, afin de ne pas dépendre de composants fonctionnant de manière *automagique* et parfois peu paramétrables :

- OpenCL dépend largement des compilateurs disponibles pour une architecture donnée. nVidia ne supporte pour l'instant officiellement que OpenCL 1.1 et les outils n'ont pas été mis à jour contrairement aux outils CUDA qui évoluent régulièrement. Dans la mesure où OpenCL ne permet pas une performance optimale avec le même code, autant réaliser la spécialisation des codes dans les langages les plus adaptés.
- L'usage de C++Amp et OpenMP 4.0 pour utiliser les GPU et les MIC, car ces outils ne semblent pas laisser assez de marge de manœuvre pour le développement des codes optimisés.

[6] Les travaux de fiabilité numérique n'ont matériellement pas pu être réalisés au cours de cette thèse (voir les travaux de A. Pedron et l'utilisation de CADNA, *c.f.* la section 3.4.3 de [Ped13])

Ainsi, en utilisant des outils natifs, il est espéré l'obtention d'implémentations au plus proche d'une plateforme donnée (gestion mémoire explicite, utilisation d'instructions SIMD sur GPP...) sans subir les aléas d'outils fonctionnant tels des boites noires. 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.

3.3.3.3 Choix d'outils de programmation

Pour le calcul sur GPP Intel, le choix s'est porté sur l'utilisation des outils suivants :

- **OpenMP** pour la programmation *multithread* en raison de sa simplicité d'utilisation et de son universalité : les calculs de simulation 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 spécification 2.0 est disponible pour la majorité des compilateurs existants) ;
- **Boost.SIMD** pour l'usage des instructions SIMD dans les algorithmes, tant pour sa facilité d'utilisation, sa portabilité pour gérer un grand nombre de jeux d'instructions, que pour son côté bibliothèque de *templates* uniquement afin de ne rajouter ni dépendances ni intermédiaires dans la chaîne de compilation ;
- **Intel MKL** en raison de ses performances afin de réaliser les calculs liés au traitement du signal (opérations FFT).
- **Compilateur Intel** pour la compilation du code hôte C++ et du code *Xeon Phi*.

Sur GPP, le choix d'OpenMP et de la bibliothèque MKL paraît à l'heure actuel pérenne : l'un est un standard ouvert développé depuis des années, l'autre est une bibliothèque développée et maintenue activement par des équipes Intel. Les bibliothèques Intel TBB et Cilk+ n'ont pas été choisies pour paralléliser le code afin de réduire les dépendances du code développé. OpenMP répond aux besoins et la richesse de ces solutions n'apparait pas nécessaire. L'algorithme étudié s'est avéré très régulier et présente un parallélisme de boucle de haut niveau, ce qui le rend facile à équilibrer à l'aide des moyens de contrôle d'OpenMP. A l'avenir, il sera aisé de changer d'outil de parallélisation si ces outils sont nécessaires.

De plus sa compatibilité en terme d'API permet une éventuelle bascule vers FFTW. Le choix de Boost.SIMD permet quant à lui de privilégier le développement rapide d'algorithmes utilisant les instructions SIMD avec un niveau d'abstraction suffisant vis à vis des jeux d'instructions utilisés et d'obtenir une meilleure lisibilité du code. Bien sûr, cette bibliothèque est récente et sa pérennité est encore inconnue. Cependant, une fois le parallélisme donné exprimé dans un algorithme, le travail de réécriture du code si le choix de son remplacement était fait, requerrait un effort moindre : le parallélisme et la régularité des calculs auront déjà été exprimés.

Pour les GPU de marque nVidia, le choix est le suivant :

- **CUDA natif** 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 (profileur, debuggueur...) ;
- **cuFFT** pour les transformations de signaux.

Sur GPU, nVidia se place en leader du calcul à haute performance depuis bientôt dix 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.

3.4 État de l'art : calcul de champ sur architectures parallèles

Dans le domaine du contrôle non destructif, plusieurs techniques sont utilisées pour calculer la propagation d'ondes ultrasonores rayonnées à l'intérieur de pièces à contrôler. Cette section présente une étude bibliographique sur les travaux existants en calcul de champ sur architectures parallèles. Sont tout d'abord présentés les calculs réalisés sur des méthodes "globales" de type éléments ou différences finis, avant de décrire des méthodes hybrides plus rapides basées sur des modèles semi-analytiques. L'étude bibliographique se poursuit sur les travaux de parallélisation d'algorithmes de reconstruction d'images à partir de signaux d'acquisition. Pour finir, elle traite des composants utilisables pour des simulations interactives, particulièrement les méthodes de calcul de trajets.

3.4.1 Méthodes basées sur les éléments ou les différences finis

Ces méthodes sont utilisées pour résoudre numériquement des équations aux dérivées partielles. Elles reposent sur des discrétisations fines de l'espace en "cellules" et du temps de propagation en "pas de temps". Des équations différentielles décrivent le comportement au sein des cellules, ainsi que les conditions aux limites (valeurs imposées, gradients imposés...). Les solutions approchées de ces équations sont déterminées itérativement pour chaque cellule, pour chaque pas de temps. Le schéma numérique temporel propage les résultats d'un pas de temps au suivant.

Dans la méthode différences finies (DF), les dérivées spatiales et temporelles sont obtenues numériquement à l'aide de combinaisons de la fonction en un nombre fini de points du maillage. Cela restreint le maillage spatial à des grilles cartésiennes sur lesquelles le calcul des dérivées spatiales est possible ce qui empêche de définir des géométries complexes. Cependant, la régularité spatiale des données est particulièrement bien adaptée au calcul sur GPU.

Dans le cadre des éléments finis (EF), on résout un problème dérivé de l'équation aux dérivées partielles (EDP) de départ. On représente la fonction solution de l'EDP sur une série de fonctions de forme dont le support est défini par une cellule élémentaire (triangle, quadrangle, tétraèdre, hexaèdre...). La solution du problème s'obtient par résolution d'une formulation intégrale de l'EDP. Comme pour les différences finies, l'évolution dans le temps se fait par propagation d'un pas de temps au suivant. De par sa capacité à être utilisée sur divers types de cellules, cette méthode permet de gérer des géométries complexes via une étape de maillage de la géométrie.

En revanche, comme ces deux méthodes modélisent l'ensemble des modes de propagation acoustique, elles ne permettent pas de simuler indépendamment les phénomènes se déroulant dans la pièce inspectée (découplage de mode par exemple) rendant l'analyse complexe.

Par ailleurs, le temps de calcul dépend essentiellement des granularités temporelle et spatiale choisies. Celles-ci sont nécessairement fines pour des simulations précises. Le temps de calcul nécessaire pour obtenir ces simulations est donc en général très important ce qui les rend souvent incompatibles avec des objectifs d'exploitation industrielle.

Ces méthodes EF et DF peuvent calculer le champ rayonné dans une pièce mais également prendre en compte la présence d'un défaut, celui-ci étant modélisé comme une composante géométrique de la pièce avec un milieu à l'intérieur (en général de l'air) et avec un raffinement adaptatif du maillage pour mieux gérer l'interface ainsi introduite.

Les méthodes de ce type sont à gros grain, adaptées à la parallélisation pour un pas de temps donné mais nécessitent d'optimiser les transferts de données aux interfaces des différents échantillons spatiaux de la grille, voire de résoudre des systèmes linéaires en parallèle. Plusieurs développements de ces méthodes ont été réalisées sur GPU et ont permis des accélérations substantielles, en particulier dans le cadre du contrôle non destructif.

On peut citer les travaux de D. Romero du *Instituto de Automática Industrial* (Madrid, Espagne) qui ont visé, dès 2009, à utiliser les capacités de calcul massivement parallèles des GPU pour réaliser des calculs de champ dans le domaine du contrôle non destructif en s'appuyant sur la méthode de Piwakowski. Leurs résultats ont été initialement communiqués lors de la conférence QNDE 2009 ([RMGMU10]) puis publiés dans [RLMGM$^+$11]. Bien qu'affichant des accélérations entre 1 et 2 ordres de grandeur sur GPU par rapport à un code sur GPP, la simulation présentée pour 12900 points de champ nécessite près de 2,8s de temps de calcul pour la résolution la plus large (GTX 295). Une seconde publication vient compléter ces résultats [RLMGMA$^+$12]. Pour des résolutions que l'auteur décrit comme présentant un bon accord, le temps de calcul est de l'ordre de 15,32s pour une zone de champ de 19300 points (48,25x101 mm dans l'eau, GPU de type nVidia Quadro 4000, environ 0,79ms/point). Pour une même finesse, les travaux précédents présentent, sur un GPU de même ordre de puissance, des temps comparables de l'ordre de 0,73ms/pt (12900 points soit 32x100mm).

Dans le domaine de la santé, F. Varray et al. ont présenté des travaux réalisés au *laboratoire CREATIS* (Université Lyon I, France) qui apportent des simulations rapides et qui traitent des problématiques au delà de la propagation linéaire en permettant le calcul du fondamental et de la seconde harmonique du champ ultrasonore[VCR$^+$11]. Avec des simulations réalisées en des temps inférieurs à la seconde, cette méthode est très rapide cependant les caractéristiques du corps humain et le fonctionnement des transducteurs médicaux (au contact) ne sont pas transposables immédiatement au contrôle non destructif.

Le logiciel de simulation de contrôle *ComWave* développé par *ITOCHU Techno-Solutions Corporation* (Japon) a intégré une nouvelle fonctionnalité de calcul de champ pour répondre aux besoins de simulations massives telles que l'étude de la variabilité d'un paramètre dans des temps raisonnables. L'utilisation de multiples GPU (2×Tesla C2075) a permis de gagner environ un facteur 10 par rapport aux temps de calcul multi-CPU à travers une grille (5 nœuds de calcul 2×X5675). De plus, l'utilisation d'un modèle adaptatif définissant, en fonction des vitesses dans les milieux, différentes zones dans les pièces inspectées, a permis de séparer l'espace en plusieurs régions de pas temporels différents permettant de gagner en performances intrinsèques sur la méthode ([HNS$^+$12] et [SNHI]).

C.A. Nahas, du *Centre for NDE, Indian Institute of Technology* (Madras, Inde) a, quant à lui, traité la génération de A-Scan en quelques dizaines de microsecondes sur des GPU grand public[NRBK12].

On peut noter également les travaux de M. Cherry de l'*University of Dayon Research Institute* (Dayton, OH, USA) qui propose l'usage de GPU pour traiter des grilles spatiales 3D sur des matériaux complexes [CABB13]. Là encore, le GPU se prête aux simulations de contrôles par éléments finis même si l'auteur précise que le facteur 100 est atteint par rapport à un code GPP de référence qui n'était ni parallélisé ni vectorisé.

Pour conclure dans le cadre du contrôle non destructif sur deux résultats "récents" (par rapport à l'obsolescence fulgurante des matériels informatiques), on peut citer les travaux de M. Molero et al., du *Centro de Acústica Aplicada y Evaluación No Destructiva* (Madrid, Espagne) qui ont présenté en 2013 un portage sur GPU de simulations 2D de la propagation d'une onde à travers des matériaux anisotropes en immersion[MIV13] développé en OpenCL. Malgré des accélérations annoncées de ×19 en passant sur architecture GPU par rapport à un GPP, les temps de calcul restent encore importants. Sur le GPU le plus performant présenté (AMD Radeon 7970) la simulation d'un échantillon de 35×50 mm sur la plus fine résolution nécessite 32,71 secondes. (A titre de comparaison avec les GPU étudiés dans le cadre de ces travaux de thèse, sur une GeForce GTX 570 proche de la GTX 580 utilisée ici, cette même simulation nécessite 38,00 secondes). En se basant sur la connaissance des cartes graphiques plus récentes, les performances de ces dernières se sont accrues, mais pas d'un facteur suffisant (de l'ordre d'au moins ×50) pour

atteindre des performances compatibles avec l'interactivité.

De son côté, P. Huthwaite de l'*Imperial College* (Londres, Royaume-Uni) a proposé une généralisation du calcul par éléments finis pour l'élastodynamique sur GPU. Cette généralisation se base sur une découpe des cellules spatiales représentant la configuration, adaptée aux contraintes mémoire des GPU nVidia. Le tout a été rassemblé dans un programme Open Source nommé POGO [Hut14]. L'auteur annonce des gains de 1 à 2 ordres de grandeur par rapport à des simulations commerciales sur "GPP équivalents". P. Huthwaite remet cependant en cause l'optimalité du logiciel de référence considéré dans la mesure où il n'a obtenu qu'un facteur 16 entre GPU et GPP sur un code autonome optimisé pour GPP.

L'ensemble de ces travaux publiés montre que la simulation du champ rayonné dans une configuration de contrôle, assistée d'architecture massivement parallèle de type GPU, est un question déjà largement traité dans la littérature. Cependant, il est à noter que la complexité de la physique de l'acoustique du solide mise en jeu dans un contrôle industriel ne permet pas, contrairement au domaine médical, d'obtenir par méthodes de différences ou d'éléments finis des simulations interactives.

3.4.2 Méthodes basées sur un modèle hybride

Pour accélérer les simulations, il a été proposé de coupler les méthodes numériques (éléments ou différences finis) avec des méthodes semi-analytiques.

On peut notamment citer les travaux de W. Choi de l'*Imperial College* (Londres, Royaume-Uni) portant sur un couplage de deux méthodes pour obtenir des simulations d'écho complexes, réalisés dans le cadre du projet européen SIMPOSIUM [CSLC14]. Dans le cadre de ces simulations d'échos, le champ servant de données d'entrée au calcul de la réponse défaut est défini en découpant l'espace en deux zones. Autour du capteur et jusque dans la pièce, la propagation de l'onde est obtenue par le modèle semi-analytique de CIVA, bien adapté pour modéliser une région relativement régulière. Une fois dans la région d'intérêt, autour du défaut, la méthode d'éléments finis 3D prend le relai pour modéliser l'ensemble des comportements complexes engendrés par l'imperfection. Le couplage entre les interfaces des deux domaines de calcul se réalise par le calcul des fonctions de Green aux interfaces. Enfin, pour encore gagner en performances, l'outil POGO est utilisé pour réaliser les calculs d'éléments finis sur un GPU. Ainsi, des performances substantielles sont obtenues par rapport aux logiciels de simulation concurrents sans perdre en complexité sur les configurations simulées.

En dehors de ces méthodes numériques (ou hybrides) il n'y a pas, à la connaissance de l'auteur, de publications concernant des méthodes semi-analytiques développées sur architectures parallèles. Aussi, avant d'exposer le modèle de calcul de champ développé, l'étude bibliographique est complétée par un exposé des résultats dans le domaine de la reconstruction, connexe à celui de la simulation.

3.4.3 Méthodes de reconstruction sur GPU

La reconstruction nécessite de modéliser certains comportements physiques de l'onde qui, selon les méthodes, peuvent être également utilisés dans le cadre de la simulation. En particulier, certains de ces modèles seront utilisés par la suite dans la simulation rapide de champ (trajets, traitement du signal rapide...). Dans ce domaine, il existe un certain nombre d'articles visant à atteindre des performances de reconstruction temps réel par rapport à des acquisitions.

On peut citer les travaux de D. Romero et al. qui ont proposé une première implémentation temps réel de reconstruction par la méthode SAFT (*Synthetic Aperture Focalisation Technique*) sur GPU en 2009 [RMGM$^+$09]. Cette méthode a par la suite été améliorée par O. Martìnez-

Graullera et al., du *Centro de Acústica Aplicada y Evaluación No Destructiva* (Madrid, Espagne), pour réaliser réalisé un filtrage par apodisation des images obtenues par méthode SAFT [MGHM+11] en temps réel sur GPU.

Une autre méthode reconstruction a bénéficié de plusieurs accélérations sur GPU. Il s'agit de la méthode de Focalisation en Tous Points (FTP), basée sur une acquisition *Full Matrix Capture* (FMC) dans laquelle sur un capteur multi-éléments, successivement, chaque élément est émetteur pendant que tous les éléments reçoivent, générant une matrice de signaux. La reconstruction FTP consiste à focaliser le champ en chaque point de la zone reconstruite. Pour ce faire, le temps de vol entre l'émetteur et le point considéré est calculé ainsi que celui entre le point et le récepteur : le temps de vol total permet de récupérer l'information utile dans le signal correspondant au couple émetteur/récepteur. La contribution de chaque couple est sommée en chaque point, focalisant ainsi le capteur. La figure 2.16a présente cette méthode.

D. Romero a proposé une accélération de la méthode FTP au moyen de GPU [RLMGMA+11]. Dans ces travaux qui ne concernaient que des calculs au contact sans réfraction/réflexion, le temps de vol nécessaire à la focalisation des signaux d'acquisition est obtenu par un calcul géométrique de distance.

Pour sa part, M. Sutcliffe de la *Swansea Metropolitan University* (Swansea, Royaume Uni) a présenté des optimisations par post-traitement par la méthode FTP sur GPU [SWD+12] lui permettant d'obtenir sur GPU des performances temps réels de reconstruction sur des configurations au contact.

Au sein du CEA-LIST, des travaux importants ont été entrepris afin de développer une méthode FTP généraliste et de la porter sur GPU. Les travaux ont tout d'abord été présentés lors de la conférence DASIP 2012 (*Design & Architectures for Signal & Image Processing*) [LPG+12] avant d'être étendus et intégrés dans CIVA au cours des travaux de thèse d'A. Pedron [Ped13] puis complétés [RLI+14]. D'autres méthodes de reconstruction ont également été étudiées pour intégration dans CIVA. On peut citer la méthode TDTE (*Time Domain Topological Energy*) proposée par EADS [DRLP13]. Elle se base sur le calcul de l'image de l'énergie topologique obtenue par la combinaison du champ émis dans une pièce saine et du champ adjoint portant un signal d'écho. Ces champs étaient obtenus originellement par une méthode de différences finies. R. Pautel, lors d'un stage au CEA-LIST, a adapté cette méthode en se basant sur un calcul de champ par les modèles semi-analytiques de CIVA et étudié sa portabilité sur GPU. Une fois les réponses impulsionnelles par mode et par élément calculées et transférées sur GPU, il a été possible de déterminer les deux champs par sommation des réponses élémentaires et convolutions avec le signal de référence adéquat [Pau13].

3.4.4 Calculs des trajets

Le modèle présenté dans le cadre de ces travaux de thèse repose sur la simulation de pinceaux représentant la propagation élémentaire de l'onde ultrasonore rayonnée dans la pièce inspectée. Pour déterminer ces pinceaux, il est nécessaire de calculer le trajet de l'onde entre le capteur et le point de champ à travers l'ensemble des interfaces. Il s'agit d'un problème connu et largement abordé dans d'autres domaines tels que la sismologie (*two-point ray-tracing*). Il est intéressant de revenir brièvement sur les différentes approches analytiques développées dans le cadre du contrôle non destructif pour le calcul des trajets de l'onde dans la pièce inspectée.

D. Romero [RLMGMA+11] utilise dans le cas d'un capteur direct au contact de la surface d'entrée de la pièce inspectée un trajet géométrique direct linéaire entre le point source et le point de champ. Bien qu'efficace, cette approche simple est limitée sur le nombre de configurations traitées.

Dans le cadre de la méthode FTP sur GPU développée évoquée ci-dessus [LPG+12], le choix a été fait d'utiliser une résolution numérique de l'équation de Snell-Descartes. Ainsi, les trajets en immersion dans des milieux homogènes isotropes à travers des interfaces canoniques sont obtenus au moyen de la résolution de polynômes de degrés 4 à 16. Cette résolution est effectuée numériquement par les méthodes de Newton et de Laguerre.

J. Dziewierz du *Center for Ultrasonic Engineering, University of Strathclyde* (Glasgow, Royaume-Uni) propose un algorithme GPU dédié au calcul des trajets sur GPU en résolvant analytiquement l'équation de Snell-Descartes au passage d'interfaces surfaces planes [DG13]. Il présente les performances supérieures de son modèle direct en le comparant à des résolutions numériques de type Simplexe et de type méthode de Newton. Cependant, il fait également part de problèmes d'instabilité numérique sur plusieurs zones de l'espace en raison du mauvais conditionnement des paramètres d'entrée de ses formules pour des calculs en nombres flottants simple précision.

Dans le cadre des travaux de thèse présentés le choix a été fait de retenir une résolution numérique d'équations plutôt qu'une résolution analytique, de la même manière que pour le calcul de FTP déjà implémentés sur CIVA, afin de ne pas risquer une instabilité numérique. Les solutions directes permettant d'obtenir analytiquement les trajets ne concernent qu'un petit nombre de surfaces, une recherche itérative permettra une plus grande généricité à la simulation en traitant une plus grande variété de géométries et de trajets.

Les travaux précédents réalisés sur la méthode FTP [LPG+12] et complétés dans la thèse d'A. Pedron[Ped13] ont montré qu'un grand nombre de trajets à travers des surfaces pouvaient être déterminés au moyen de la résolution d'équations pouvant prendre la forme de polynômes aux coefficients réels de degrés variés. La forme polynômiale est particulièrement adaptée à une évaluation rapide au moyen du schéma de Hörner, ce qui permet d'itérer rapidement et d'approximer la solution. Ces travaux ont montré une stabilité numérique des méthodes de résolution, en particulier la méthode de Laguerre qui permet de déterminer chacune des racines d'un polynôme séquentiellement : la méthode converge sur une (racine simple) ou deux racines (racines complexes conjuguées) et le degré du polynôme est réduit d'autant par division polynomiale. Des problèmes de stabilité ont été soulevés pour un très faible nombre de polynômes de haut degré (12 voire 16), permettant de réaliser des calculs en flottants simples précisions. La résolution numérique d'équations de forme polynomiale permet d'envisager une plus grande souplesse dans l'extension future des travaux à des géométries plus complexes (surfaces canoniques non planes, rebonds avec conversion...).

Dans le cas d'une interface plane, la recherche de trajets est encore plus simplifiée : avec une bonne approximation de la racine recherchée, la méthode de Newton itère directement vers la racine réelle. Il faut noter cependant que ces approches ne sont plus disponibles dans le cadre de trajets en mode rebond avec conversion de mode : leur calcul revient à la résolution d'un système d'équations. Le modèle choisi et ses limitations seront détaillés dans la prochaine section de ce document.

3.5 Conclusion

Dans le contrôle non-destructif par ultrasons, le calcul de champ est utilisé pour déterminer les caractéristiques du faisceau d'ondes émises par le traducteur au sein de la pièce contrôlée, le plus souvent dans une sous région d'intérêt déterminée par l'opérateur. Jusqu'à maintenant, l'opérateur détermine la ou les configurations de contrôle pertinentes par rapport à ses besoins par une succession de simulations, souvent chronophages, ajustées par approximation successives des paramètres grâce à son expertise. Fournir un résultat interactif lui permettrait de déterminer visuellement et très rapidement la ou les configurations pertinentes.

Pour atteindre cette interactivité, le choix s'est porté sur l'utilisation d'architectures GPP, GPU et MIC, compatibles avec les cibles matérielles actuellement visées par la plateforme CIVA (station de calcul). L'usage d'outils natifs a été retenu afin de tirer parti au maximum des capacités de calcul de ces architectures matérielles, sources de parallélisme : *multithreading* et instructions SIMD sur GPP, mais également calcul sur coprocesseurs déportés (GPU et Xeon Phi).

Le prochain chapitre présentera le modèle de simulation de champ ultrasonore qui a été mis au point en s'inspirant des modèles complets de CIVA. A partir de ce modèle est obtenue une implémentation de référence dont les caractéristiques seront analysées.

Chapitre

4 Simulation de calcul de champ

Les architectures informatiques permettent d'obtenir beaucoup de puissance de calcul à la condition d'y exécuter un algorithme exprimant suffisamment de parallélisme. Dans l'objectif d'atteindre des performances très rapide, voire interactives, de simulation de champ ultrasonore, il convient d'analyser le modèle semi-analytique au moyen d'une implémentation de référence en vue de sa parallélisation.

4.1 Présentation du modèle

Cette section présente le modèle utilisé pour la simulation du champ ultrasonore rayonné dans une pièce par un transducteur au contact ou en immersion.

4.1.1 Qu'est ce qu'un calcul de champ ?

La simulation du champ, c'est-à-dire le faisceau rayonné par un capteur, est un outil devenu indispensable pour la conception, la validation et l'interprétation de contrôles. L'objectif du calcul de champ est de modéliser en chaque point de la zone simulée le faisceau émis par le capteur. Il est à noter que les simulations de faisceaux sont aussi les données d'entrée pour les simulations d'interaction faisceau/défaut.

Mathématiquement, le rayonnement d'un transducteur est décrit par une intégrale de diffraction (intégrale de Rayleigh-Sommerfeld). Dans certaines configurations simples (par exemple pièce plane, capteur circulaire en mode direct...), il est possible de calculer analytiquement le champ ultrasonore en tous points. Cependant, puisqu'aucune hypothèse n'est faite ici, ni sur la forme de la pièce, ni sur la forme du transducteur, cette intégrale est évaluée numériquement. La surface émettrice du transducteur est discrétisée en plusieurs points sources et leurs contributions sont sommées aux différents points de champ. La contribution de chaque point source est un champ élémentaire qui est obtenu en résolvant localement une équation différentielle par une fonction de Green. La méthode des pinceaux permet d'évaluer cette contribution. Il est alors possible d'obtenir le rayonnement du transducteur par intégration des contributions élémentaires le long du transducteur.

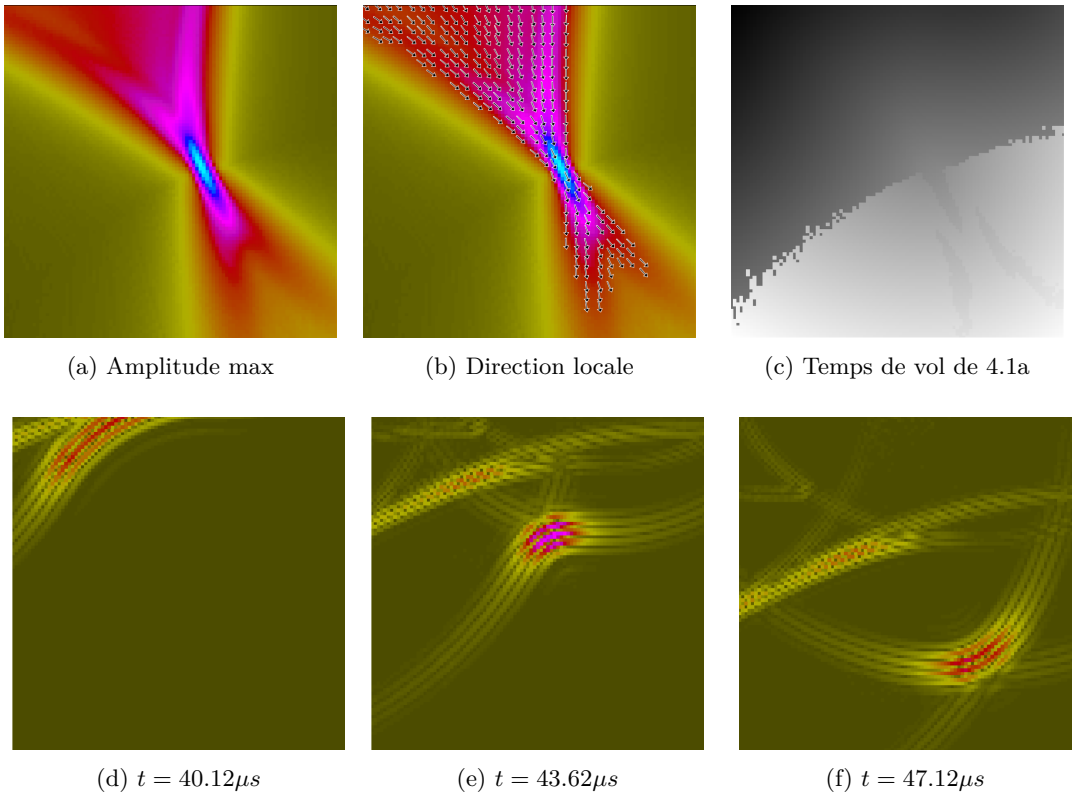
La méthode des pinceaux permet de simuler l'ensemble des phénomènes électrodynamiques résultant de la propagation d'un faisceau (déplacement, vitesse, contraintes). Dans le cadre de la simulation interactive, l'opérateur veut généralement estimer les caractéristiques du faisceau propagé pour la configuration choisie et s'intéresse à un champ de déplacement. En chaque point, le champ ainsi simulé prend la forme d'un signal numérique représentant le déplacement de l'onde propagée. En fonction des besoins de la simulation, il est possible d'obtenir des signaux de déplacement vectoriel ou d'amplitude du module du déplacement. La nature du résultat peut aussi être une image, sur la zone de champ, de l'amplitude maximum de ces signaux (*c.f.* les figures 4.1a et 4.1b). Il est aussi possible de s'intéresser au temps correspondant au maximum du module de déplacement (*c.f.* la figure 4.1c) ou d'obtenir une animation de l'évolution du déplacement en fonction du temps (*c.f.* les figures 4.1d, 4.1e et 4.1f).

4.1.2 Modèle des pinceaux

Le modèle des pinceaux permet d'estimer les contributions élémentaires des différents points sources en approximant le front d'onde par des pinceaux se propageant dans les différents matériaux. Cette méthode est détaillée dans le chapitre "La méthode des pinceaux" [GLC06]. Dans la présente section son principe est brièvement rappelé. Les spécificités du calcul dans le cadre de la simulation de champ interactive sont soulignés.

4.1.2.1 Principe

L'onde émise en un point source n'est pas une onde plane : le front d'onde sphérique de cette onde élémentaire se déforme au passage des interfaces traversées en fonction de leurs géométries et des milieux rencontrés suivant le plan d'incidence et le plan normal. Or, une onde sphérique se propageant dans un milieu homogène et isotrope peut être approximée localement par une onde plane dont l'amplitude décroit en fonction de l'inverse de la distance de propagation (son intensité décroit alors en fonction de l'inverse de la distance au carré). Cette approximation est valide dès lors que la distance entre le point source et le point d'observation est supérieure à quelques fois la longueur d'onde. Le modèle des pinceaux est une généralisation de cette observation à des matériaux complexes.



(a) Amplitude max (b) Direction locale (c) Temps de vol de 4.1a

(d) $t = 40.12\mu s$ (e) $t = 43.62\mu s$ (f) $t = 47.12\mu s$

FIGURE 4.1 – Plusieurs résultats de la simulation de champ dont l'évolution du front d'onde (d, e et f)

En un point d'observation, le principe d'acoustique géométrique indique que l'énergie émise par une source se propage le long du **chemin d'acoustique géométrique**, ou **chemin de Fermat** : l'onde se propage de la source au point d'observation le long du trajet minimisant le temps de parcours. Ce chemin va de la source au point d'observation en suivant la loi de Snell-Descartes au passage de chaque interface (*c.f.* figure 4.2). Il dépend des vitesses de propagation dans les milieux de part et d'autre de l'interface, vitesses qui dépendent elles-mêmes du type d'onde considérée.

A partir de ce chemin, on peut déterminer géométriquement un pinceau décrivant localement cette onde sous la forme d'un cône de sommet le point source se déformant à travers les surfaces rencontrées. L'intensité acoustique du pinceau évolue à l'image de la déformation locale du front d'onde (*c.f.* la figure 4.3) : la grandeur appelée divergence du pinceau décrit cette intensité.

4.1.2.2 Modélisation

Un pinceau représente l'onde élémentaire propagée par une source ponctuelle sous la forme d'un cône de sommet la source. Il décrit les variations locales des quantités électrodynamiques. Dans le plan perpendiculaire au rayon axial de direction z, on définit dx et dy, les dimensions spatiales du tube formé par les rayons paraxiaux. Les deux grandeurs (dS_x et dS_y) donnent la projection des vecteurs lenteur paraxiaux dans ce même plan (*c.f.* figure 4.4). Le pinceau est donc défini

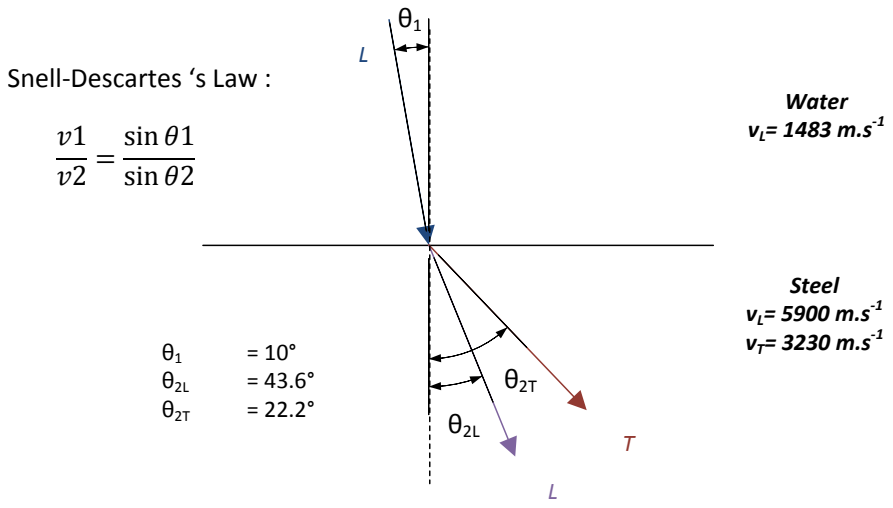


FIGURE 4.2 – Loi de Snell-Descartes - Illustration de réfraction Eau-Acier

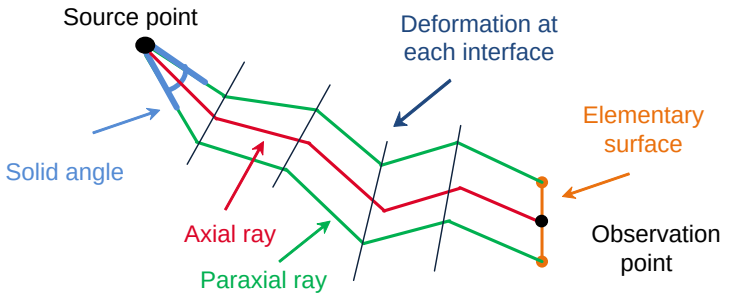


FIGURE 4.3 – Déformation du pinceau au passage des interfaces le long du rayon axial/trajet optique

localement dans le repère "rayon" par le vecteur :

$$\Psi = \begin{pmatrix} dx \\ dy \\ dS_x \\ dS_y \end{pmatrix}$$

L'évolution du pinceau peut être décrite au moyen d'une matrice (4×4) L, appelée matrice de propagation, qui définit une relation linéaire entre deux vecteurs Ψ et Ψ' décrivant le même pinceau à deux instants différents.

$$\Psi' = L.\Psi \tag{4.1}$$

Cette matrice L peut être subdivisée en quatre sous matrices (2×2) A, B, C et D :

$$L = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{4.2}$$

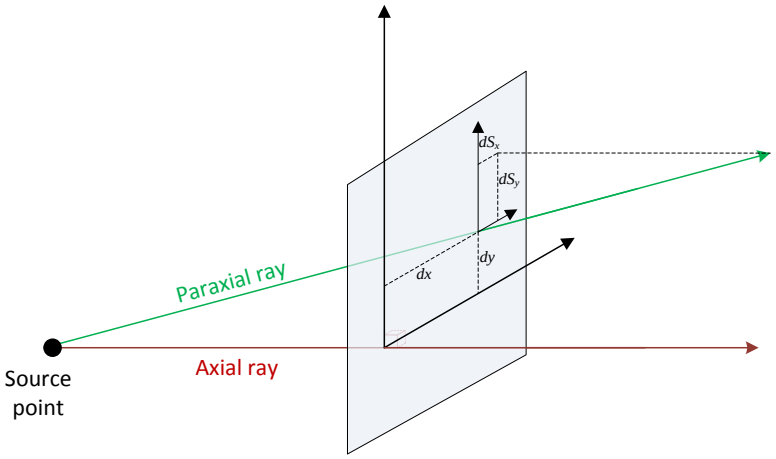


FIGURE 4.4 – Définition des grandeurs d'un pinceau

Par analogie avec l'intensité acoustique produite par une sphère pulsante, il est possible de définir l'intensité acoustique I_Ω correspondant à un angle solide Ω (*c.f.* [GLC06], eq. 1.103). On peut écrire l'énergie à la source du pinceau $I_\Omega.d\Omega$ où I_Ω est l'intensité de la source acoustique par unité d'angle solide, et $d\Omega$ l'angle solide au niveau source. On suppose que l'énergie du pinceau ne se disperse pas par ses côtés mais uniquement lors de sa propagation : à partir de l'intensité acoustique $I(N)$ au point de calcul, on obtient l'énergie par $I(N).dS$.

4.1.2.3 Propagation d'un pinceau

Afin de modéliser complètement un pinceau, il est nécessaire de décrire son évolution tout au long de sa propagation à travers les milieux au moyen d'une succession de propagations élémentaires. La matrice de propagation du pinceau est le produit des matrices de propagation élémentaires de chacun des différents médias rencontrés : les matrices de propagation à travers les matériaux L_{prop_k} et les matrices de propagation à travers les interfaces rencontrées L_{interf_m}.

$$\Psi' = L_{prop_n} \cdot L_{interf_{n-1}} \cdot L_{prop_{n-1}} \cdots L_{interf_2} \cdot L_{prop_2} \cdot L_{interf_1} \cdot L_{prop_1} \cdot \Psi \tag{4.3}$$

Ce modèle permet d'exprimer le pinceau pour une multitude de trajets. Cependant, la présente étude se limite aux trajets qu'il est possible d'obtenir par résolution numérique d'expressions analytiques. C'est pourquoi cette section ne traite que des trajets suivants : propagation en milieu homogène isotrope et franchissement d'interfaces planes, pour des modes directs et demi-bond sans conversion de mode[1].

4.1.2.3.1 Propagation d'un pinceau dans un milieu homogène isotrope Au cours de sa propagation, le segment (dx, dy) du pinceau Ψ subit une homothétie de facteur $\frac{r}{s}$, comme illustré par la figure 4.5, et devient Ψ'.

[1]Les trajets directs à travers des surfaces de géométrie courbes ou complexes peuvent être mis en équation (polynômes de degrés supérieurs à 4 et résolus par la méthode de Laguerre...), de même les trajets directs à conversion de mode peuvent être obtenus par des systèmes à deux inconnues qui peuvent être résolus par la méthode de Newton 2D

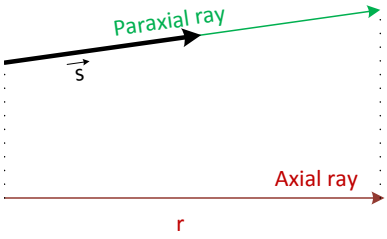


FIGURE 4.5 – Évolution d'un pinceau dans un milieu isotrope homogène

La matrice de propagation correspondante est L_{iso} :

$$\Psi' = L_{iso}.\Psi = \begin{bmatrix} 1 & \frac{r}{s} \times 1 \\ 0 & 1 \end{bmatrix}.\Psi \tag{4.4}$$

où 0 est la matrice (2×2) nulle et où 1 est la matrice (2×2) identité.

4.1.2.3.2 Franchissement d'interface Le franchissement, par réfraction ou réflexion, d'une interface par un pinceau entraîne sa déformation.

Interface plane Pour une interface plane, la littérature donne la matrice de propagation suivante :

$$\Psi' = L_{inter}.\Psi = \begin{bmatrix} \Theta_2\Theta_1^{-1} & 0 \\ 0 & \Theta_2^{-T}\Theta_1^{T} \end{bmatrix}.\Psi \tag{4.5}$$

où les matrices (2×2) Θ_1 et Θ_2 sont la projection du pinceau sur l'interface dans la direction du rayon axial, respectivement *avant* et *après* l'interaction avec l'interface plane [Gen03].

Dans le cas d'une interface plane, les matrices Θ_i s'écrivent :

$$\Theta_i = \begin{bmatrix} cos(\theta_i) & 0 \\ 0 & 1 \end{bmatrix} \tag{4.6}$$

où θ_i est l'angle formé par la normale à la surface et le rayon considéré.

Coefficients d'interaction Pour chaque réfraction ou réflexion sur une interface, le pinceau est également affecté d'un coefficient de transmission ou de réflexion, dit coefficient de Fresnel, traduisant la répartition de l'énergie entre les différentes ondes issues de l'interaction. Ces coefficients sont issus des relations de continuité aux interfaces (continuité des déplacements et des contraintes normales à l'interface). Le détails des calculs est donnés en annexe A.

Il est à noter que les coefficients obtenus sont des coefficients en amplitude.

$$Amplitude_2 = T_{1\to2}.Amplitude_1 \tag{4.7}$$

Afin de tenir compte du coefficient de transmission **en amplitude**, il faut multiplier le rapport des intensités par le coefficient de transmission **en énergie**. Ce dernier se rapporte au ratio des surfaces traversées, de part et d'autre de l'interface (*c.f.* la figure 4.6).

L'intensité étant égale au carré de l'amplitude, il faut prendre en compte la racine du ratio des surfaces, ce qui se ramène au ratio des projections.

$$T_E = \sqrt{|\frac{\cos\theta 2}{\cos\theta_1}|} \tag{4.8}$$

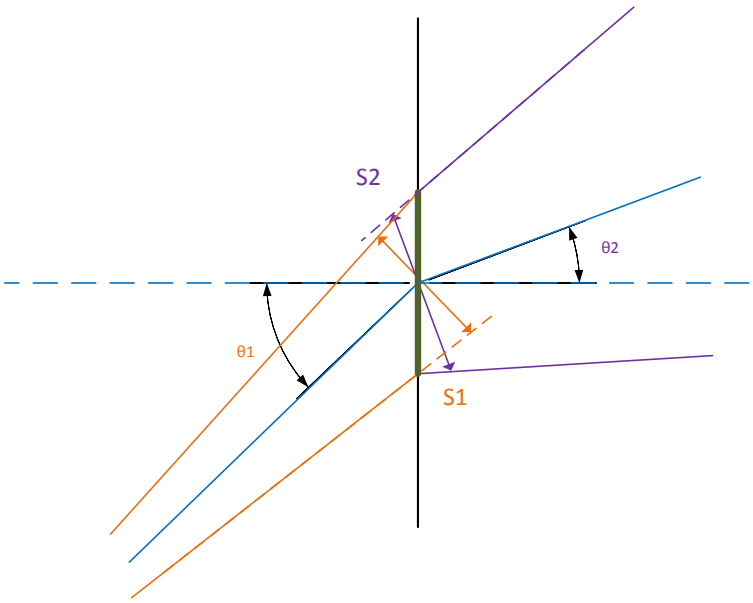


FIGURE 4.6 – Rapport des surfaces traversées

Ainsi on obtient l'expression finale de la transmission de l'amplitude à l'interface :

$$Amplitude_2 = T_{1\to 2}.T_E.Amplitude_2 = T_{1\to 2}.\sqrt{|\frac{\cos\theta 2}{\cos\theta_1}|}.Amplitude_1 \tag{4.9}$$

4.1.2.4 Divergence du pinceau

A partir du principe de conservation de l'énergie, on définit le **facteur de divergence** en intensité R_I par

$$\frac{I(N)}{I_\Omega} = \frac{1}{R_I^2} = \frac{d\Omega}{dS} \tag{4.10}$$

Puisque dS et $d\Omega$ peuvent être exprimés à l'aide de Ψ et Ψ', à partir de la matrice de divergence on obtient $dS = dx' \cdot dy'$ et $d\Omega = \frac{dS_x \cdot dS_y}{s_1^2}$ où s_1 est la lenteur dans le milieu source. On obtient l'expression du facteur de divergence (*c.f.* [Gen03], équation 3 ; [GLC06] eq. 1.81) :

$$R_I^{-2} = {s_1}^2 detB \tag{4.11}$$

Le chapitre 1.2.2.5 de [GLC06] donne les expressions détaillées des intensités. L'intensité à l'origine du pinceau I_Ω est obtenue à partir de la formulation de l'onde sphérique induite par une sphère pulsante ; dans le cadre d'un contrôle, cette onde est hémisphérique d'intensité moitié moindre. L'intensité au point de calcul $I(N)$ est obtenue à l'aide de la norme du vecteur de Poynting exprimée à l'aide du déplacement particulaire élémentaire induit par la source ponctuelle.

On en déduit l'expression du **coefficient de divergence** en amplitudes (*c.f.* équations 2.22 , 2.23 et 2.24 de [Gen99]) :

$$DF = \frac{1}{2\pi\sqrt{detB}} \tag{4.12}$$

Cette expression repose sur la conservation de l'énergie au niveau du pinceau, pour tenir compte des effets de transmission aux différentes interfaces il faut alors prendre en compte les coefficients de transmission associés aux différentes interfaces. Elle prend la forme d'un coefficient global résultat de la multiplication de l'ensemble des coefficients élémentaires.

L'amplitude du déplacement du pinceau considérée est alors :

$$Amplitude = DF * T_{1\to 2} * T_E \tag{4.13}$$

avec $T_{1\to 2}$ le coefficient de Fresnel en transmission et T_E le coefficient d'énergie (en amplitude).

4.1.3 Formulation complète - cas plan

Dans ce paragraphe est détaillé l'ensemble des calculs permettant d'obtenir l'amplitude d'un pinceau, en mode direct, à travers deux milieux homogènes isotropes comme illustré à la figure 4.7.

A l'aide des matrices de propagation élémentaire vues précédemment, on obtient les matrices de propagations suivantes, avec respectivement L_{Prop_1} et L_{Prop_2} les matrices de propagation dans les milieux 1 et 2, et L_{surf} la matrice de courbure de la surface plane.

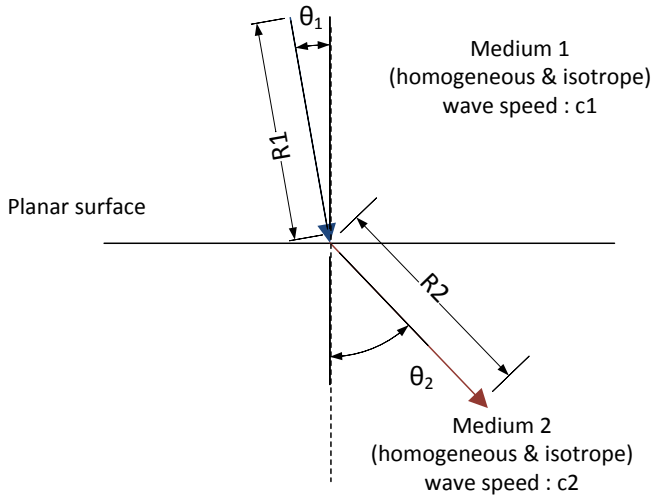


FIGURE 4.7 – Trajet à travers une interface plane Eau-Acier

$$L_{Prop_1} = \begin{pmatrix} 1 & 0 & R_1c_1 & 0 \\ 0 & 1 & 0 & R_1c_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$L_{Prop_2} = \begin{pmatrix} 1 & 0 & R_2c_2 & 0 \\ 0 & 1 & 0 & R_2c_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$L_{surf} = \begin{pmatrix} \frac{\cos\theta_2}{\cos\theta_1} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{\cos\theta_1}{\cos\theta_2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

On obtient ainsi la matrice correspondant au trajet du pinceau par multiplication des matrices élémentaires, soit :

$$\Psi' = L \cdot \Psi$$

avec

$$L = L_{Prop_2} \cdot L_{surf} \cdot L_{Prop_1} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

En particulier, la sous-matrice B obtenue, de dimension 2×2 , est :

$$B = \begin{pmatrix} \frac{R_1c_1\ \cos\theta_2}{\cos\theta_1} + \frac{R_2c_2\ \cos\theta_1}{\cos\theta_2} & 0 \\ 0 & R_2c_2 + R_1c_1 \end{pmatrix}$$

Son déterminant $detB$ est alors :

$$detB = (R_2c_2 + R_1c_1)\left(\frac{R_1c_1\ \cos\theta_2}{\cos\theta_1} + \frac{R_2c_2\ \cos\theta_1}{\cos\theta_2}\right) \tag{4.14}$$

La section du pinceau considéré avec le traducteur étant égale à dS, l'amplitude du déplacement associé au pinceau est :

$$Amplitude = dS * DF * T_{1\to2} * T_E \tag{4.15}$$

avec $T_{1\to2}$ le coefficient de Fresnel en transmission et T_E le coefficient d'énergie (en amplitude).

Pour une onde L, le déplacement du pinceau est défini à partir de la direction du trajet au niveau du point de calcul. Pour une onde T, celui-ci est donné par la polarisation de l'onde, calculée par le produit vectoriel de la direction du pinceau et de la normale au plan de calcul. Le temps de vol du pinceau correspond au temps nécessaire à l'onde ultrasonore pour aller du capteur au point de calcul.

Ainsi, les grandeurs suivantes caractérisent un pinceau :

$$\begin{array}{ll} \text{amplitude} & \to\ \mathbb{C} \\ \text{temps de vol} & \to\ \mathbb{R} \\ \overrightarrow{\text{direction du déplacement}} & \to\ \mathbb{R}^3 \end{array}$$

4.1.4 Réponse impulsionnelle

Un intérêt de la méthode des pinceaux est de permettre l'approximation de la réponse impulsionnelle d'un traducteur au moyen de l'intégration d'un ensemble de sources ponctuelles. La réponse élémentaire de ces sources est d'abord présentée avant de s'intéresser à la formation de la réponse globale.

4.1.4.1 Réponse élémentaire

Discrétiser de manière infinitésimale le transducteur permettrait de prendre en compte les pinceaux comme des Diracs $\delta(t-\tau)$ (où τ est le temps de vol du pinceau considéré), cependant cela nécessiterait un temps de calcul qui tendrait vers l'infini (et donc trop important). On utilise par conséquent un échantillonnage surfacique du transducteur, correspondant à des pinceaux d'étalement temporel Δ_t, centré sur le temps de vol τ du rayon initial. La réponse impulsionnelle résultante est une fonction rectangle de hauteur $\frac{A_\delta}{\Delta_t}$ où A_δ est l'amplitude du Dirac équivalent.

L'étalement temporel Δ_t correspond à la différence de marche du pinceau sur la surface considérée ΔS autour du point capteur : c'est l'écart maximum en temps entre deux rayons parallèles au chemin géométrique considéré issus de deux points extrêmes de la surface (*c.f.* figure 4.8).

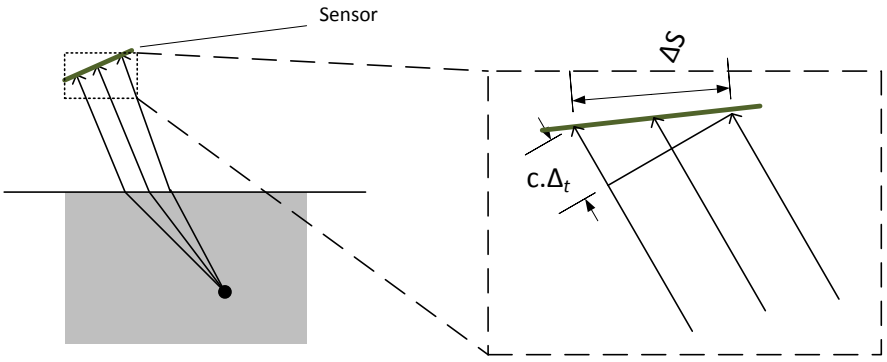


FIGURE 4.8 – Différence de marche d'un pinceau sur une surface élémentaire Δ_S (avec c la célérité et Δ_t l'étalement temporel)

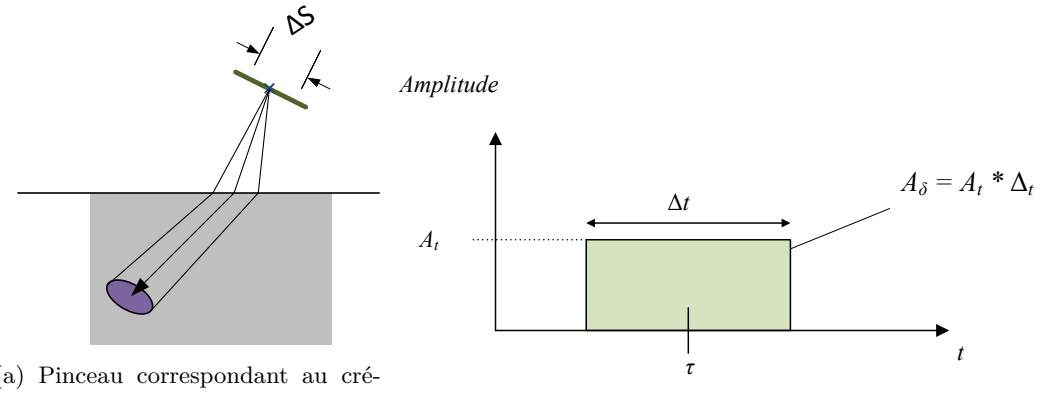
La figure 4.9 représente la réponse impulsionnelle obtenue ; son intégrale (figure 4.9b) est identique à celle du Dirac équivalent : elle représente la même énergie. Pour rappel, l'amplitude du Dirac $\delta(t-\tau)$ correspondant est

$$A_\delta = DF \times T_{1\to 2} \times T_E \times \Delta S \tag{4.16}$$

4.1.4.2 Sommation en une réponse globale

La réponse impulsionnelle du traducteur, au point de calcul, est obtenue en intégrant la contribution de l'ensemble des points sources, c'est à dire en sommant l'intégralité des contributions des pinceaux émis par le traducteur. On superpose ainsi les réponses impulsionnelles élémentaires de chacune des sources ponctuelles afin d'obtenir la réponse impulsionnelle complète du traducteur (*c.f.* figure 4.10). C'est lors de cette sommation qu'il est possible de prendre en compte, dans le cas d'un transducteur multi-éléments, les lois de retards appliquées aux différents éléments. Pour rappel, les réponses impulsionnelles correspondent au déplacement complexe du pinceau : chacune de leurs composantes est sommée sur un signal correspondant (soit 6 signaux au total pour 3 composantes de déplacement complexes).

Cette réponse impulsionnelle correspond au champ associé à l'émission d'une impulsion (un Dirac temporel) par le transducteur. Pour prendre en compte le signal de l'excitation appliquée



(a) Pinceau correspondant au créneau et ΔS capteur associé

(b) Formation d'un créneau

FIGURE 4.9 – Étalement temporel d'un pinceau

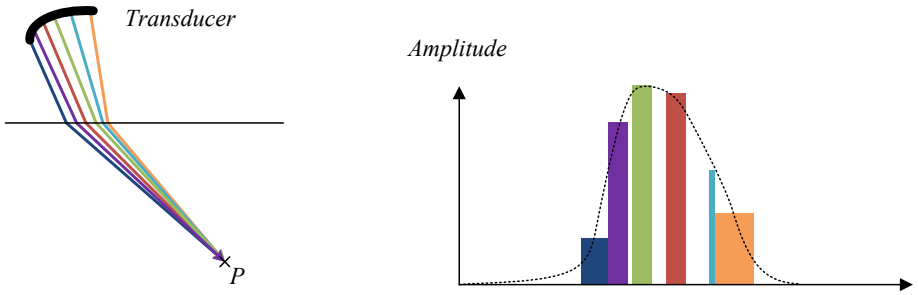


FIGURE 4.10 – Sommation des réponses impulsionnelles élémentaires

au transducteur, dit signal de référence, il faut convoluer la réponse impulsionnelle avec ce dernier. Or, comme cette réponse peut être complexe, sa partie réelle est convoluée avec le signal de référence alors que sa partie complexe est convoluée avec la Transformée de Hilbert du signal de référence. L'équation 4.17 présente le calcul de la convolution d'un signal de réponse impulsionnelle selon une de ses composantes scalaires $s(t)$.

$$\begin{gathered} \text{avec } \mathcal{F} \text{ et } \mathcal{H} \text{ les Transformées, respectivement, de Fourier et Hilbert} \\ s\left(t\right) = s_{RI}\left(t\right) \bigotimes \left(sigref\right)\left(t\right) \\ s\left(t\right) = \mathcal{F}^{-1}\left(\mathcal{F}\left(\mathrm{Re}\left(s_{RI}\right)\left(t\right)\right) \times \mathcal{F}\left(sigref\left(t\right)\right) + \mathcal{H}\left(\mathrm{Im}\left(s_{RI}\right)\left(t\right)\right) \times \mathcal{F}\left(sigref\left(t\right)\right)\right) \end{gathered} \tag{4.17}$$

4.1.4.3 Calcul du module et extraction des amplitudes, temps et directions locales

Le modèle de champ développé permet d'obtenir pour chaque pinceau les différentes contributions électrodynamiques en un point (déplacement, vitesses, contraintes). Dans le cadre du calcul de champ interactif, le champ que l'on souhaite simuler est un champ de déplacement. Afin d'obtenir des simulations interactives facilement interprétables par l'utilisateur, la quantité

de données affichées (et donc d'intérêt) est restreinte : un module du déplacement plutôt qu'un signal à trois dimensions, une cartographie 2D ou 3D du maximum d'amplitude plutôt que des signaux. Le module du déplacement est obtenu comme il suit :

$$|\overrightarrow{déplacement}(t)| = \sqrt{\overrightarrow{déplacement}_x(t)^2 + \overrightarrow{déplacement}_y(t)^2 + \overrightarrow{déplacement}_z(t)^2} \quad (4.18)$$

Pour obtenir les cartographies d'amplitude, de temps de vol et de directions locales, on recherche le maximum d'amplitude A_{max} du signal du module de déplacement. A ce A_{max} correspond le temps de vol T_{max} et la direction locale du déplacement max $\overrightarrow{D_{max}}$. Pour ne pas risquer de problème de déphasage du signal, ce maximum est obtenu au moyen de l'enveloppe du signal de déplacement.

$$\{A_{max}, T_{max} \text{ avec } A_{max} = |\overrightarrow{déplacement}(T_{max})| \text{ tel que } \forall t\ A_{max} \geq |\overrightarrow{déplacement}(t)|\} \quad (4.19)$$

$$\{\overrightarrow{D_{max}}, T_{max} \text{ avec } \overrightarrow{D_{max}} = \overrightarrow{déplacement}(T_{max})\} \quad (4.20)$$

On peut ainsi visualiser, comme indiqué au paragraphe 4.1.1 (figure 4.1) les différents types de résultats associés à un champ.

4.1.5 Principe algorithmique

A partir du modèle décrit dans les sections précédentes, il est possible de formuler un algorithme généraliste du calcul de champ, présent à l'algorithme 1.

Algorithme 1 : Principe algorithmique du calcul de champ

entrées :
Description de la structure inspectée
Description du capteur subdivisé en $N_{capteur}$, au signal de référence $sigref(t)$
Zone de champ simulée (discrétisée en points de champ)

sorties :
A_{max} l'amplitude du maximum de déplacement
T_{max} le temps de vol correspondant à A_{max}

```
1  foreach P, point de champ do
2      RI_déplacement(t) = 0 ;
3      foreach E, point source sur le capteur do
4          foreach Pinceau, pinceau potentiel do
               // L'espace des pinceaux potentiels dépend des modes simulés et de
                  la géométrie de la pièce
5              Rayon = RésolutionPolynomialeTrajet(P,E,Pinceau);
6              Pinceau = CaractéristiquesPinceaux(Rayon);
7              RI_déplacement(t)+=Pinceau
8      déplacement(t) = RI_déplacement(t) ⊗ sigref(t);
9      Amplitude(t) = Enveloppe( Module( déplacement(t) ) );
10     {(A_max, T_max)|A_max = Amplitude(T_max), ∀t A_max ≥ Amplitude(t)};
```

4.2 Implémentation de référence

Dans cette section, les choix algorithmiques qui ont été faits afin de mettre en place une implémentation de référence vont être discutés. Cette première version constitue un code séquentiel en C++.

A partir de l'algorithme 1, il est possible de définir les grandes étapes du calcul de champ :

- Calcul des pinceaux reliant les points de champ à l'échantillonnage capteur suivant les différents modes souhaités ;
- Sommation des contributions des pinceaux afin d'obtenir la réponse impulsionnelle du capteur ;
- Extraction de l'information de champ pour restituer le champ à l'utilisateur.

En terme d'implémentation, une étape supplémentaire doit être prise en compte : l'analyse de la taille du signal de champ sur lequel il faut sommer les contributions des pinceaux (en prenant en compte les retards éventuels), afin de pouvoir allouer la mémoire avant de réaliser l'opération pour éviter de coûteuses ré-allocations dynamiques.

On peut numéroter les différentes phases de calcul ainsi :

Étape 1 Calcul des pinceaux

Étape 1.1 Calcul d'un trajet entre le capteur et le point de champ

Étape 1.2 Calcul du pinceau (amplitude, temps de vol et étalement temporel)

Étape 2 Analyse de la taille mémoire des signaux

Étape 3 Traitement du signal

Étape 3.1 Sommation des contributions des pinceaux et application des retards

Étape 3.2 Convolution, module, enveloppe et recherche du maximum

L'algorithme 2 présente l'enchaînement de ces étapes. Les sections suivantes détaillent chacune des étapes d'un point de vue théorique.

4.2.1 Calcul d'un pinceau (Étape 1)

Le calcul d'un pinceau permet de déterminer la réponse impulsionnelle élémentaire associée à un couple (point de champ, point source). Il s'agit de l'opération de base du calcul de champ.

4.2.1.1 Calcul du trajet (Étape 1.1)

La première étape du calcul d'un pinceau consiste à en déterminer le trajet reliant le point source et le point d'observation pour la configuration donnée.

C'est un problème connu et largement abordé dans d'autres domaines tels que la sismologie (*two-point ray-tracing*). Il s'agit de résoudre le problème d'optimisation du temps de parcours entre les deux points (chemin de Fermat).

En restreignant le cadre des trajets recherchés à des cas plus simples, hors des cas complexes (anisotropie, hétérogénéité, continument variable...) parfois rencontrés en sismologie comme en CND, il est possible de résoudre directement ce problème d'optimisation en trouvant le chemin suivant les lois de Snell-Descartes aux interfaces. Ce chemin peut s'écrire de manière analytique sous la forme d'un système d'équations composé d'une ou plusieurs équations.

Algorithme 2 : Principe d'implémentation du calcul de champ

entrées :
Description de la structure inspectée
Description du capteur subdivisé en $N_{capteur}$, au signal de référence $sigref(t)$
Zone de champ simulée (discrétisée en points de champ)

sorties :
A_{max} l'amplitude du maximum de déplacement
T_{max} le temps de vol correspondant à A_{max}

1 **foreach** *P, point de champ* **do**
  `// ETAPE 1 : Calcul d'un pinceau`
2 **foreach** *E, point source sur le capteur* **do**
3 **foreach** *Pinceau, pinceau potentiel* **do**
  `// l'ensemble des pinceaux potentiels reflète la complexité géométrique de la simulation (modes simulés et nombre de surfaces)`
  `// ETAPE1.1 : Détermination du trajet du pinceau`
4 Rayon = RésolutionPolynomialeTrajet(P,E,Pinceau);
  `// ETAPE1.2 : Caractéristiques du pinceau`
5 TdV = TempsDeVol(Rayon);
6 ΔT = EtalementTemporel(Rayon);
7 $\overrightarrow{d}$ = Déplacement(Rayon);
  `// ETAPE 2 : Recherche de la taille des signaux`
8 RechercheTailleSignaux(...);
9 $\overrightarrow{RI_{déplacement}}(t) = \vec{0}$; `// Allocation mémoire des signaux`
  `// ETAPE 3 : Traitement du signal`
  `// ETAPE 3.1 : Sommation des contributions de chaque pinceau`
10 **foreach** *E, point source sur le capteur* **do**
11 **foreach** *Pinceau, pinceau potentiel* **do**
12 **foreach** $i \in [\![TdV - 0.5\Delta T; TdV + 0.5\Delta T]\!]$ **do**
13 $\overrightarrow{RI_{déplacement}}[i] += \overrightarrow{d}$;
  `// ETAPE 3.2 : Extraction du maximum d'amplitude du signal de déplacement`
14 $\overrightarrow{déplacement}(t) = \overrightarrow{RI_{déplacement}}(t) \bigotimes sigref(t)$;
15 Amplitude(t) := Enveloppe(Module($\overrightarrow{déplacement}(t)$));
16 $\{(A_{max}, T_{max}) | A_{max} = Amplitude(T_{max}), \forall t\ A_{max} \geq Amplitude(t)\}$;

Le choix a été fait dans ces travaux, à partir des connaissances antérieures concernant le calcul de trajets, de pratiquer une résolution numérique de ces solutions analytiques. Cela restreint la recherche de trajets aux configurations dans lesquelles la formulation analytique d'un trajet est connue : dans des pièces composées de matériau homogène et isotrope, pour des capteurs en immersion ou au contact (avec ou sans sabot), on cherche des trajets en mode direct ou en mode demi-bond sur le fond de la pièce. La résolution numérique itérative approche le trajet optimal, cela revient en quelque sorte à calculer des trajets par lancer de rayons en procédant par essais

et erreurs pour converger vers une solution à un critère près.

Les différents types de trajets supportés dans le cadre des travaux de simulation interactive sont présentés ci-dessous.

4.2.1.1.1 Mode direct Le mode direct correspond à un contrôle en immersion ou au contact dans lequel le traducteur est séparé de la pièce à contrôler par un milieu couplant. L'onde va alors se propager dans deux milieux aux vitesses de propagation différentes résultant en un phénomène de réfraction à l'interface couplant-pièce.

Cas d'une interface plane La figure 4.11 illustre le calcul d'un trajet entre le point source S et le point de calcul P à travers une interface plane. La recherche du trajet vise à déterminer les coordonnées de I le point d'impact, c'est à dire le point d'intersection du trajet avec la surface.

A partir du schéma 4.11b :

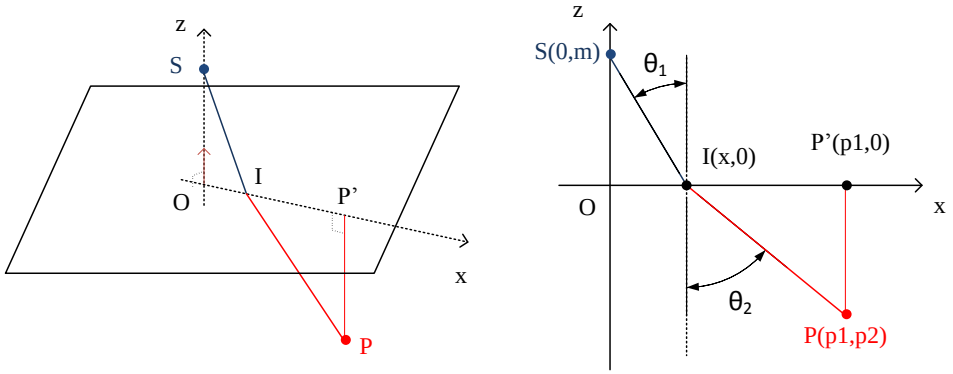
$$sin(\theta 1) = \frac{x}{\sqrt{x^2+m^2}} \quad \text{et} \quad sin(\theta 2) = \frac{p_1 - x}{\sqrt{(p_1-x)^2+(p_2)^2}} \quad .$$

D'après la relation de Snell-Descartes, $\frac{v1}{v2} = \frac{sin(\theta 1)}{sin(\theta 2)}$, on obtient l'équation :

$$v\frac{x}{\sqrt{x^2+m^2}} = \frac{p_1 - x}{\sqrt{(p_1-x)^2+(p_2)^2}} \quad \text{avec} \quad v = \frac{v_2}{v_1} \quad .$$

Afin de faciliter l'usage de méthodes itératives d'évaluation et de résolution, élever cette équation au carré permet d'obtenir un polynôme de degré 4, $P[X]$ qu'il faut résoudre :

$$P[X] = X^4 - 2p_1X^3 + \left(p_1^2 + \frac{v^2p_2^2 - m^2}{v^2-1}\right)X^2 + \frac{2p_1m^2}{v^2-1}X - \frac{p_1^2m^2}{v^2-1} = 0 \qquad (4.21)$$



(a) Trajet recherché (b) Changement de repère

FIGURE 4.11 – Recherche du trajet direct à travers une interface plane

Recherche numérique polynomiale De nombreuses méthodes de résolution polynomiale sont disponibles afin de trouver les racines d'un polynôme. Dans le cadre de la recherche de trajets, les polynômes adressés sont des polynômes à coefficients réels. Dans le cas général des trajets à travers des interfaces canoniques, on choisit la méthode de Laguerre, validée sur cette

catégorie de polynômes (*c.f.* le chapitre 3 des travaux de thèse d' A. Pédron [Ped13]), en raison de sa bonne convergence et de sa stabilité numérique[2]

Dans le cas d'une interface plane, la recherche de trajet consiste à obtenir une des racines réelles, la figure 4.11b montre que celle-ci vérifie $0 < x < p_1$ pour $p_1 \neq 0$. Dans les cas où $m = 0$ ou $p_1 = 0$, le trajet est déterminé immédiatement, avec pour racine $x = 0$. Dans le cas général, il est possible d'initialiser la méthode de Newton avec une approximation de la racine cherchée pour l'obtenir en un faible nombre d'itérations. Cette méthode est rappelée par l'algorithme 7. L'approximation initiale de la racine proposée est : $x_0 = 0.1 \times p_1$.

Algorithme 3 : Méthode de Newton pour la recherche de racine de $P[X]$

```
1 x_1 = x_0 − P[x_0]/P'[x_0];
2 for iter=1 ;iter<MAX ;iter++ do
3     if P'[x_n] < ε then
4         break
5     if P[x_n]/P'[x_n] < ε then
6         break
7     x_{n+1} = x_n − P[x_n]/P'[x_n] ;
```

4.2.1.1.2 Mode rebond sans conversion de mode

Afin de traiter le rebond de l'onde sur les surfaces de fond, il convient de séparer les rebonds sans conversion de mode, dont le trajet peut être obtenu simplement dans le cas plan, des rebonds avec conversion de mode.

Rebond sur une interface plane Dans le cas d'un rebond sans conversion sur une interface plane, l'obtention du trajet consiste à se ramener, par symétrie sur le fond plan, à un trajet direct sans conversion de mode jusqu'au point $P_{symetry}$. La figure 4.12 illustre la recherche d'un trajet vers P par rebond sans conversion de mode sur le plan de fond.

4.2.1.1.3 Géométries CAO 2D d'extension plane à interfaces multiples

Les pièces inspectées sont parfois plus complexes qu'une surface d'entrée et une surface de fond : il est possible de les définir au moyen de CAO assemblant plusieurs surfaces canoniques.

Dans ce cas, pour un mode donné, le champ résultant correspond au champ traversant chaque surface d'entrée (mode direct) ou au champ traversant chaque couple { surface d'entrée, surface de fond } (mode rebond). Cela revient à se rapporter aux cas précédents et à parcourir exhaustivement toutes les surfaces d'entrées ou tous les couples de surfaces (entrée/fond) possibles.

Pour les configurations très complexes, il est possible de mettre en place des stratégies pour réduire algorithmiquement le nombre de couples de surfaces parcourues. Celles-ci n'ont pas été ajoutées aux implémentations réalisées dans ces travaux.

[2]Ces résultats valident la méthode de Laguerre pour la recherche de trajets dans le cadre de la méthode FTP, c'est à dire principalement la recherche du temps de vol associé. Une contre-validation sur la précision des trajets en calculs simple précision dans le domaine géométrique pourrait compléter l'étude afin d'étudier l'erreur en nombres significatifs perdus. Cependant, les ordres de grandeurs des configurations étudiées ici et dans [Ped13] étant les mêmes, les artefacts doivent être du même ordre : il est possible en l'état des travaux de ne pas prioritiser cette contre-validation pour l'instant.

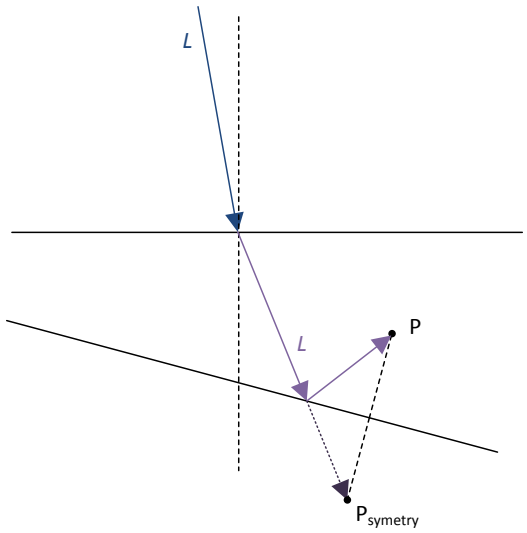


FIGURE 4.12 – Calcul de trajet par symétrie sur le fond (rebond sans conversion de mode)

4.2.1.1.4 Validité du trajet La détermination du trajet est purement mathématique à partir d'une surface, mais elle ne prend pas en compte les contraintes du contrôle. Une fois le trajet déterminé, quelques vérifications géométriques s'imposent afin de s'assurer, dans le cas des configurations de contrôle, de la validité de ce dernier. Ainsi, il faut vérifier entre autres :

1. que le trajet trouvé ne sort pas des surfaces qui ont servi à son calcul analytique ;
2. que le trajet trouvé, dans le cadre d'un capteur au contact, passe par le fond du sabot du transducteur ;
3. que le trajet trouvé se passe bien dans la pièce ou ne rencontre pas d'obstacles (autres surfaces, défaut...) : on parle d'ombrage ou de vérification de la présence d'occlusions[3].

Pour le calcul de champ, tous les trajets valides trouvés correspondent à des pinceaux qui contribuent à former la réponse impulsionnelle. Dans le cas d'un trajet non valide, le pinceau obtenu est écarté et ne contribue pas.

La figure 4.13 présente quelques trajets invalides qu'il est mathématiquement possible d'obtenir lors de la simulation de champ dans une pièce. De gauche à droite, un trajet en dehors de la surface réelle (mais sur la surface analytique infinie, cas 1), deux trajets ne passant pas par le fond du sabot du transducteur (cas 2) et enfin un trajet sortant de la pièce (cas 3).

4.2.2 Caractéristiques du pinceau (Étape 1.2)

Une fois le trajet calculé pour un pinceau, il faut en déterminer les caractéristiques principales. Celles-ci sont obtenues depuis les informations géométriques du trajet.

[3]Ce test de validation n'a pas été implémenté au cours de cette thèse. Cela n'invalide pas les résultats des simulations pour les configurations présentées ci-après (4.3.5). La conclusion offre des perspectives pour la résolution de ce problème (*c.f.* chapitre 8)

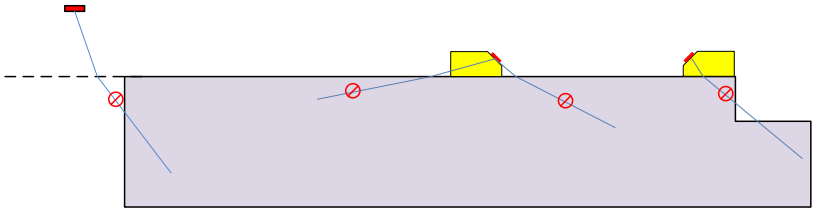


FIGURE 4.13 – Exemples de trajets non valides que la résolution numérique peut obtenir

4.2.2.1 Amplitude du pinceau

calcul de DF Le calcul du coefficient de divergence, DF, passe théoriquement par le calcul de la matrice de propagation de chaque pinceau. Dans la mesure où l'implémentation de référence travaille sur un nombre restreint de configurations de champ, il est possible de se passer des calculs coûteux des produits des matrices élémentaires. Les formulations analytiques directes sont utilisées : DF est donné par l'équation 4.12 à partir de la matrice de propagation calculée à l'équation 4.14. Ainsi, pour ces trajets, à partir des formulations connues, on obtient une formulation analytique du coefficient DF.

calcul des coefficients de Fresnel Ces coefficients sont obtenus analytiquement à partir de l'angle d'incidence du rayon axial et des caractéristiques des milieux de part et d'autre de l'interface. Ces formules analytiques sont spécifiques aux types de matériaux (interface solide-liquide ou solide-solide), au type d'interaction (réflexion/réfraction) et au type de modes du rayon (L-L, L-T, T-L, T-T).

4.2.2.2 Temps de vol

Le temps de vol associé au pinceau dépend de son trajet. A partir des distances parcourues et des vitesses de l'onde dans les milieux traversés, on détermine le temps de vol.

4.2.2.3 Mesure de l'étalement temporel du pinceau

Les pinceaux sont calculés de manière analytique, leur répartition est définie *a priori*, en fonction des besoins de l'utilisateur.

4.2.2.3.1 Principe de l'étalement temporel La surface du capteur est subdivisée en $N_{capteur}$ points sources auxquels sont associés des surfaces élémentaires dS (avec dS=Surface active du capteur / $N_{capteur}$).

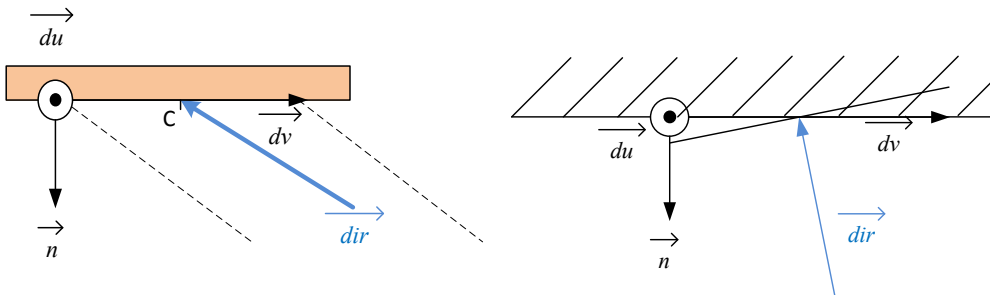
Lors du calcul de champ, le coefficient de divergence DF d'un pinceau est calculé "à l'envers" en inversant le rôle des points sources et des points de champ. On peut fait l'exercice de pensée du calcul d'un pinceau du point de champ vers le point source. Dans ce cas la surface élémentaire ΔS est celle du pinceau projeté sur la surface capteur. La contribution d'un pinceau à la réponse impulsionnelle ne se traduit pas par un Dirac calculé point à point à l'aide d'un rayon, mais par une fonction ayant un certain étalement temporel. Dans la modélisation utilisée (*c.f.* équation 4.16), elle est approchée par une fonction rectangle sur un intervalle ΔT et d'amplitude A telle que :

$$\frac{A}{dS} = DF \times T_{1\to 2} \times T_E \tag{4.22}$$

4.2.2.3.2 Échantillonnage du capteur L'échantillonnage spatial du capteur détermine les grandeurs suivantes, permettant le calcul de l'étalement temporel des pinceaux sur le capteur, pour chaque échantillon surfacique :

- son point centre C ;
- la normale à l'échantillon $\vec{n}$;
- les vecteurs $\vec{du}$ et $\vec{dv}$ décrivant les bords du pinceau projeté sur l'échantillon.

A partir de ces grandeurs, illustrées sur la figure 4.14, on projette la direction du trajet sur $\vec{du}$ et $\vec{dv}$ sur le capteur. En prenant en compte la lenteur du milieu, il est possible de calculer le décalage temporel associé au pinceau.



(a) Repère pinceau appliqué à l'échantillon capteur

(b) Calcul de l'étalement projeté correspondant au pinceau sur le capteur

FIGURE 4.14 – Calcul de l'étalement projeté correspondant au pinceau sur le capteur

4.2.2.3.3 Recalage temporel des contributions Les pinceaux et les créneaux associés ne sont que rarement "recalés" sur l'échantillonnage temporel discret utilisé pour les signaux. Ainsi il faut s'assurer que l'ensemble de l'énergie d'un créneau contribue au signal en prenant en compte les extrémités de celui-ci. Comme présenté sur la figure 4.15, l'amplitude ajoutée au signal sur les extrémités d'un créneau correspond à la même "aire" que celle de son "débordement".

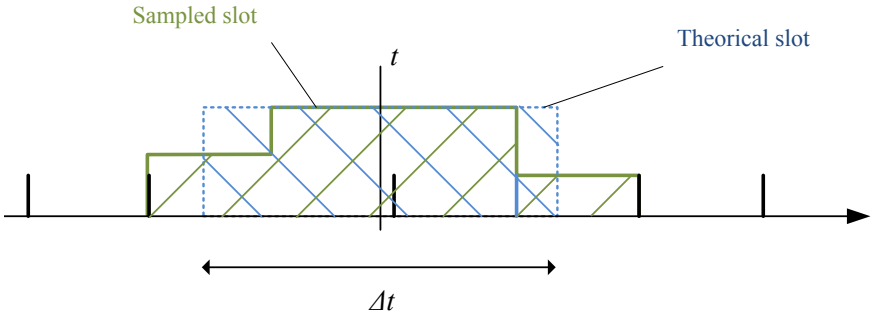


FIGURE 4.15 – Étalement temporel

4.2.3 Recherche de la taille des signaux (Étape 2)

Afin d'allouer une fois pour toute l'emplacement mémoire correspondant à l'ensemble des signaux utilisés (signaux résultats et signaux de réponses impulsionnelles) et de les aligner sur une même taille pour réutiliser les coefficients des Transformées de Fourier Rapides, il convient de déterminer les tailles de ces signaux. Avant de pouvoir effectuer la sommation des signaux de déplacements, une première étape consiste à parcourir l'ensemble des pinceaux élémentaires pour déterminer l'étalement temporel de chacun. L'algorithme 4 présente cette recherche de taille, en nombre d'échantillons, en prenant en compte l'échantillonnage du signal de référence émis par le capteur ($fréquence$) en parcourant l'ensemble des pinceaux, prenant en compte la loi de retards appliquée au capteur et l'ensemble des étalements temporels.

Algorithme 4 : Recherche de l'étalement temporel des signaux

entrées :
$fréquence$, la fréquence d'échantillonnage du capteur
L'ensemble des pinceaux calculés
La loi de retards appliquée
sorties : size, le nombre d'échantillons temporels requis par signal

```
1  size :=0 ;
2  foreach P, point de champ do
3      foreach E, point source sur le capteur do
4          min_point :=INT_MAX ;
5          max_point :=0 ;
6          foreach Pinceau, pinceau potentiel do
               // l'ensemble des pinceaux potentiels reflète la complexité
                  géométrique de la simulation (modes simulés et nombre de
                  surfaces)
7              t :=DataPinceaux(Pinceau)→TdV;
8              Δt :=DataPinceaux(Pinceau)→ ΔT;
9              R :=LoiRetards(E);
10             min_point :=min(min_point, ((t + R) − 0.5dt) × 1/fréquence);
11             max_point :=max(max_point, ((t + R) + 0.5dt) × 1/fréquence);
12     size :=max(size, max_point − min_point) ;
```

Une fois le nombre d'échantillons déterminé, la taille des signaux est la puissance de 2 strictement supérieure ou égale au nombre minimum d'échantillons nécessaires, afin d'obtenir une taille de signal optimale pour les calculs de FFT. Pour prendre en compte la convolution du signal de référence avec les réponses impulsionnelles, en mémoire, la taille en échantillons des signaux est la puissance de 2 juste supérieure à la taille obtenue à l'étape 2 augmentée de la taille du signal de référence. Ainsi, dans la mesure où un signal de référence usuel est composé de plusieurs centaines d'échantillons, les signaux en mémoire sont très souvent sur échantillonnés et complétés par des zéros[4].

[4]Cette complétion a également comme effet secondaire d'augmenter la résolution fréquentielle des résultats obtenus par FFT, cependant le traitement du signal réalisé n'est pas affecté (mis à part une quantité de calcul un peu plus importante).

4.2.4 Traitement du signal (Étape 3)

Une fois la taille des signaux de champ déterminée, et ceux-ci alloués, il est possible de procéder au calcul de champ proprement dit. Pour ce faire, les contributions des pinceaux sont sommées sur les réponses impulsionnelles, composante par composante, en tenant compte des lois de retards. Ensuite ces réponses complexes sont convoluées avec le signal de référence pour obtenir des signaux de champ par composante. Enfin, on calcule pour les différents points de champ l'enveloppe du module de ces signaux afin d'en extraire le maximum d'amplitude ainsi que le temps de vol correspondant à ce maximum.

4.2.4.1 Sommation des contributions des pinceaux (Étape 3.1)

Dans le cadre des transducteurs multi-éléments, il faut de tenir compte de la loi de retards appliquée au capteur. Lors de la sommation, connaissant l'élément correspondant au pinceau, le temps de vol des pinceaux élémentaires est retardé en fonction du retard correspondant à l'élément.

4.2.4.2 Traitement du signal (Étape 3.2)

Une fois obtenues toutes les composantes de la réponse impulsionnelle par sommation des réponses élémentaires, des traitements leurs sont appliqués en chaque point de champ pour former le signal de champ.

4.2.4.2.1 Obtention du champ Pour passer de la réponse impulsionnelle au signal émis par le transducteur, il faut tenir compte de la pulsation source (signal de référence). Les réponses impulsionnelles de déplacement, formées par sommation des contributions des pinceaux, selon les trois coordonnées $\overrightarrow{déplacement}(t) = \{x(t), y(t), z(t)\}$ doivent être convoluées avec le signal de référence. Cette convolution est effectuée, composante par composante, dans le domaine fréquentiel, requérant l'usage de Transformées de Fourier Rapides (FFT). Pour rappel, la convolution peut être obtenue par multiplication dans le domaine fréquentiel :

$$sref(t) \bigotimes s(t) = \mathcal{F}^{-1}(\mathcal{F}(sref)(f) \times \mathcal{F}(s)(f)) = \mathcal{F}^{-1}(SREF(f) \times S(f))$$

La contribution en déplacement des pinceaux peut être complexe : la convolution est réalisée entre le signal réel et le signal de référence alors que le signal imaginaire est quant à lui convolué avec la Transformée de Hilbert du signal de référence.

Les deux paragraphes qui suivent reprennent les notations de l'équation 4.17 pour présenter les opérations appliquées sur chaque composante de la réponse impulsionnelle. Il convient de rappeler ici que les signaux de champ, les composantes des réponses impulsionnelles et le signal de référence sont de même taille en nombres d'échantillons. Chaque étape nécessite la Transformée de Fourier du signal de référence $\mathcal{F}(sigref(t))$. Il est a noter que les signaux de réponse impulsionnelle sont réels, par composante, tout comme le signal de référence, ce qui permet d'utiliser les opérations de FFT de type *Real to Complex* (R2C) et *Complex To Real* (C2R) qui tirent parti de la redondance Hermitienne des résultats de la FFT (data[i] est le conjugué de data[n-i]). Bien que ces opérations provoquent quelques désagréments d'usage (disparité de taille sur les signaux d'entrée de N éléments réels et de sortie (N/2)+1 éléments complexes), ce format permet de limiter l'impact mémoire et d'obtenir des opérations de FFT algorithmiquement plus rapides. Il faut noter que la FFT R2C est toujours dans le sens direct et que la FFT C2R est toujours dans le sens inverse.

Implémentation de la convolution La convolution de la partie réelle utilise le résultat des Transformées de Fourier Rapide de la partie réelle de la réponse impulsionnelle

$$SRE_{RI}(f) = \mathcal{F}\left(\mathrm{Re}\left(s_{RI}\right)(t)\right)$$

La convolution de la partie imaginaire utilise le résultat de la FFT de la partie imaginaire de la réponse impulsionnelle

$$SIM_{RI}(f) = \mathcal{F}\left(\mathrm{Im}\left(s_{RI}\right)(t)\right)$$

Pour obtenir la Transformée de Hilbert, on utilise ses propriétés dans le signal fréquentiel :

$$\mathcal{H}\left(\mathrm{Im}\left(s_{RI}\right)(t)\right) = -i \times \mathcal{F}\left(\mathrm{Im}\left(s_{RI}\right)(t)\right) = (-i) \times SIM_{RI}(f)$$

Ainsi, à partir des FFT des composantes réelles et imaginaires de la réponse impulsionnelle considérée, on procède à la convolution dans le domaine fréquentiel :

$$\mathcal{F}^{-1}((SRE_{RI}(f) + (-i) \times SIM_{RI}(f)) \times (SREF(f)))$$

L'algorithme 5 reprend ce principe et présente le noyau de calcul de la convolution.

Algorithme 5 : Implémentation de la convolution

entrées :
$\mathrm{Re}\left(s_{RI}\right)(t)$ la partie réelle de la réponse impulsionnelle (tableau de réels)
$\mathrm{Im}\left(s_{RI}\right)(t)$ la partie imaginaire de la réponse impulsionnelle (tableau de réels)
$sref(t)$ le signal de référence
N la taille des signaux en nombre d'échantillons temporels

sorties : $output(t)$ le signal de sortie réel, de taille N échantillons temporels

`// Calcul des FFT des composantes réelles et imaginaires`

1 $SRE_{RI}(f)$ = FFT_R2C($\mathrm{Re}\left(s_{RI}\right)(t)$);
2 $SIM_{RI}(f)$ = FFT_R2C($\mathrm{Im}\left(s_{RI}\right)(t)$);
3 $SREF(f)$ = FFT_R2C($sref(t)$);

```
// Les signaux contiennent des complexes entrelacés :
// S[2*i] = partie réelle , S[2*i+1] = partie imaginaire
// Il n'y a que N/2 complexes dans les signaux post FFT
```

4 **foreach** $k \in [[1, N/2]]$ **do**
    `// RI=`$SRE_{RI}[k] - i * SIM_{RI}[k]$
5 RI=$(SRE_{RI}[2*k] + SIM_{RI}[2*k+1]) + i(SRE_{RI}[2*k+1] - SRE_{RI}[2*k])$;
    `// REF =` $SREF[k]$
6 REF=$SREF[2*k] + i.SREF[2*k+1]$;
    `// OUTPUT[k]=RI*REF - la convolution revient à faire une multiplication dans le domaine fréquentiel`
7 PROD = RI * REF;
8 OUTPUT[2*k]=Re($PROD$);
9 OUTPUT[2*k+1]=Im($PROD$);
10 $output(t)$ = FFT_C2R(OUTPUT(f));

`// Retour au domaine temporel`

4.2.4.2.2 Calcul de l'enveloppe Une fois les signaux de déplacement convolués, on obtient le module de déplacement en calculant la norme 2 des trois composantes du déplacement (qui sont réelles après convolution) :

$$module(t) = \sqrt{x(t)^2 + y(t)^2 + z(t)^2}$$

Afin de préparer l'extraction du maximum d'amplitude du signal de déplacement (et ne pas être dépendant des problèmes de déphasage), l'enveloppe du signal de module est calculée (voir figure 4.16).

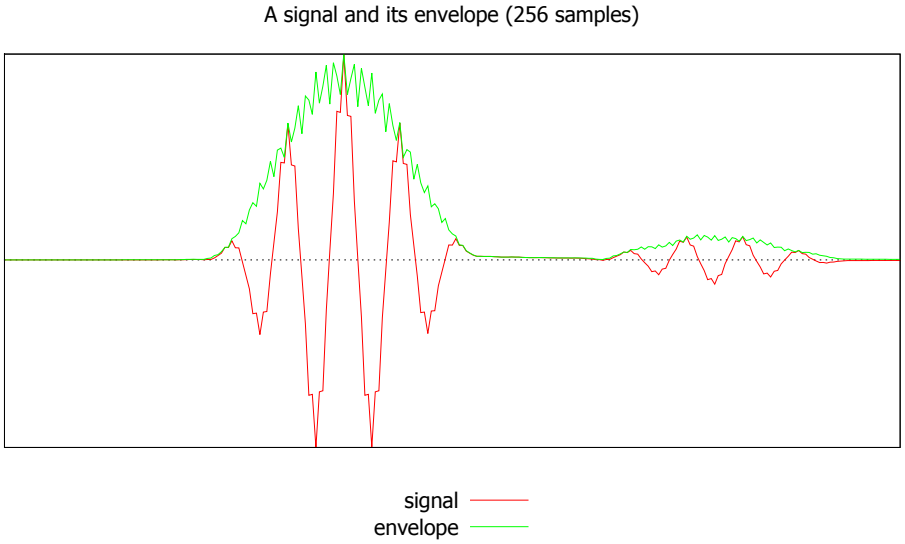


FIGURE 4.16 – Un signal et son enveloppe

Dans le domaine fréquentiel, l'enveloppe est obtenue par :

1. Calcul de la FFT *Complex To Complex* (C2C) directe du signal de module de déplacement ;
2. Doublement des fréquences positives des deux spectres obtenus ;
3. Mise à zéro des fréquences négatives des deux spectres ;
4. Calcul de la FFT C2C inverse du spectre de fréquences ainsi modifié ;
5. Calcul du signal réel résultant de la norme 2 du signal complexe (partie réelle, partie imaginaire).

L'algorithme 6 schématise l'implémentation du calcul de l'enveloppe.

4.2.4.2.3 Extraction du maximum Le maximum du module d'amplitude du signal de champ est recherché en parcourant, en chaque point de champ, l'amplitude du signal du module de déplacement. Cette enveloppe est un signal à valeurs positives réelles. A partir d'un parcours simple et d'une recherche de maximum, on obtient l'échantillon correspondant à l'indice i_{max}. Afin d'être précis et de ne pas sous évaluer l'amplitude obtenue, on utilise une interpolation

Algorithme 6 : Implémentation du calcul de l'enveloppe

entrées :
$s\_re(t)$ la partie réelle du signal de module (tableau de réels)
$s\_im(t)$ la partie imaginaire du signal de module (tableau de réels, tous nuls)
N la taille des signaux en nombre d'échantillons temporels

sorties : $enveloppe(t)$ le signal d'amplitude, réel, de taille N échantillons temporels

```
// Calcul de la FFT directe du signal de module
(SRE(f),SIM(f))=FFT_C2C(s_re(t), s_im(t));
// Les deux composantes sont obtenues par la même opération
// Doublement des fréquences positives des deux spectres obtenus
foreach k ∈ [[1, N/2]] do
    SRE[k]*=2;
    SIM[k]*=2;
// Mise à zero des fréquences négatives des deux spectres
foreach k ∈ ]]N/2, N]] do
    SRE[k]=0;
    SIM[k]=0;
// Calcul de la FFT inverse du signal de module
(s_re(t), s_im(t))=FFT_C2C(SRE(f),SIM(f));
// Signal résultant
foreach k ∈ [[1, N]] do
    enveloppe[k]=√(s_re[k]² + s_im[k]²);
```

quadratique à une dimension sur un voisinage autour de i_{max} pour obtenir une valeur précise de A_{max} le maximum d'amplitude [Syk09].

La figure 4.17 présente cette interpolation dans le cas général ; dans le cas du calcul de champ, l'interpolation est facilitée par la répartition régulière des échantillons temporels et peut être alors réalisée à partir des valeurs aux échantillons $i_{max} - 1$ et $i_{max} + 1$. Ainsi on approxime la valeur du signal et le temps de vol correspondant au maximum d'amplitude du champ en chaque point.

4.2.5 Récapitulatif

4.2.5.1 Définition des paramètres

Afin de décrire la complexité d'une configuration de calcul de champ, il convient de définir les paramètres caractéristiques :

N_{point} le nombre de points de calcul sur lesquels la simulation de champ est réalisée ;

$N_{capteur}$ le nombre de points sources sur le capteur ;

N_{surfin} **et** $N_{surfback}$ respectivement, les nombres de surfaces d'entrée et de fond de la pièce ;

N_{modes} le nombre de modes de champ demandés (peut se décomposer en $N_{modes\_directs}$ et $N_{modes\_indirects}$ pour les modes directs et indirects respectivement).

Un autre paramètre essentiel afin de qualifier une configuration, mais qui ne peut être obtenu avant l'exécution, est *size* : la taille des signaux en nombre d'échantillons. *size* dépend de

l'étalement temporel des différents pinceaux et de leurs temps de vol (convertis en échantillons à l'aide de la fréquence d'échantillonnage). Ce paramètre n'est pas une donnée d'entrée de la simulation, mais est déterminé à l'exécution afin de réserver l'espace mémoire nécessaire aux signaux utilisés (Étape 2).

4.2.5.2 Algorithme de référence

Le calcul de champ consiste à enchaîner l'ensemble des étapes présentées précédemment. Le choix a été fait de rassembler ces étapes en "blocs" de calcul afin de faire ressortir la régularité des calculs de ces sous-tâches : l'algorithme de référence est représenté en deux parties successives, par les algorithmes 7 et 8. Il s'agit d'une formulation complète du calcul de champ dans le cadre d'une CAO 2D de surfaces planes en entrée et au fond de la pièce inspectée. Dans le cas d'une simulation en mode direct uniquement, la surface de fond n'influe pas sur les simulations de champ : $N_{surfback} = 1$ et il n'y a qu'un seul niveau de boucle, celui sur les surfaces d'entrée.

Il est à noter que cette implémentation de référence diffère, par son organisation, du principe général d'implémentation du calcul de champ présenté par l'algorithme 2 : ici, le traitement est séparé par grande étape (1, 2 et 3). Cette approche permet une plus grande souplesse dans la mesure de l'optimalité des accélérations proposées permettant de discerner les performances de chacune des étapes principales.

4.2.6 Validation métier de l'implémentation de référence

Bien que proche de celui de CIVA, le modèle utilisé dans ces simulations présente des différences en ce qui concerne les formulations utilisées pour les calculs des caractéristiques des pinceaux et les contraintes appliquées aux données. La principale différence porte sur l'obtention des trajets des pinceaux. Dans CIVA les trajets reliant les points de champ à la surface du capteur sont obtenus par une méthode itérative basée sur un lancer de rayons, alors qu'ici on utilise une résolution numérique d'équation analytique. D'autres différences minimes existent comme l'utilisation ici de formules analytiques (spécialisées selon les modes de trajets) pour le calcul des facteurs de divergence en lieu et place de matrices dans CIVA. Les calculs sont réalisés en

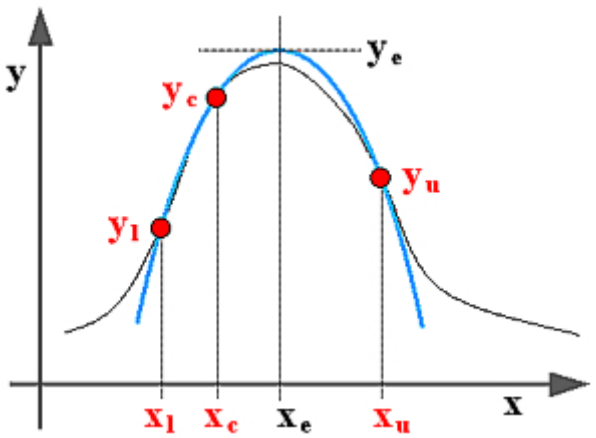


FIGURE 4.17 – Interpolation tique en 3 points

Source : Three-point interpolation of a real function by Stanislav Sýkora

Algorithme 7 : Calcul de champ de référence - Partie 1/2

entrées :
Description de la structure inspectée
Description du capteur subdivisé en $N_{capteur}$ éléments
Taille utile des signaux *size*

sorties :
A_{max}, l'amplitude du maximum de déplacement
T_{max}, le temps de vol correspondant à A_{max}

```
// ETAPE 1 : Calcul d'un pinceau
```

1 **foreach** $i_{point} \in [\![0; N_{point}[\![$ **do**
2 **foreach** $i_{surfback} \in [\![0; N_{surfback}[\![$ **do**
3 **foreach** $i_{surfin} \in [\![0; N_{surfin}[\![$ **do**
4 **foreach** $i_{mode} \in [\![0; N_{mode}[\![$ **do**
5 **foreach** $i_{capteur} \in [\![0; N_{capteur}[\![$ **do**
6 **if** i_{mode} *direct* **then**
 `// ETAPE1.1 : Détermination du trajet du pinceau`
7 Rayon = RésolutionPolynomialeTrajet(i_{point},$i_{capteur}$ i_{surfin}, i_{mode});
 `// ETAPE1.2 : Caractéristiques du pinceau`
8 TdV = TempsDeVol(Rayon);
9 ΔT = EtalementTemporel(Rayon);
10 DF :=calculDF(Rayon, i_{mode}) ;
11 CoeffTransmission :=Transmission(Rayon ,i_{surfin},i_{mode}) ;
12 $\overrightarrow{d}$ = dS($i_{capteur}$) * DF * CoeffTransmission * Direction(Rayon);
13 **else**
 `// ETAPE1.1 : Détermination du trajet du pinceau`
14 Rayon = RésolutionPolynomialeTrajet(i_{point},$i_{capteur}$ i_{surfin}, $i_{surfback}$, i_{mode});
 `// ETAPE1.2 : Caractéristiques du pinceau`
15 TdV = TempsDeVol(Rayon);
16 ΔT = EtalementTemporel(Rayon);
17 DF :=calculDF(Rayon, i_{mode}) ;
18 CoeffTransmission :=Transmission(Rayon ,i_{surfin},i_{mode}) ;
19 CoeffReflexion :=Reflexion(Rayon ,$i_{surfback}$,i_{mode}) ;
20 CoeffTotal := CoeffTransmission $\times$ CoeffReflexion ;
21 DF :=calculDF(Rayon, i_{mode}) ;
22 $\overrightarrow{d}$ = dS($i_{capteur}$) * DF * CoeffTotal * Direction(Rayon);

nombres flottants simple précision pour l'implémentation de référence alors que CIVA utilise une précision double. Ou encore, une fenêtre temporelle globale est utilisée pour les signaux alors que ce n'est pas le cas pour CIVA.

L'objet de la présente section est de vérifier à l'aide de critères métier que le code de référence obtient des résultats proches de ceux de CIVA pour les configurations qu'il sait traiter. Par ailleurs cela validera ce code de façon plus générale dans la mesure où les simulations CIVA

Algorithme 8 : Calcul de champ de référence - Partie 2/2

```
   // ETAPE 2 : Recherche de la taille des signaux
23 RechercheTailleSignaux(...);
24 RI_déplacement[i_point](t) = 0⃗;                    // Allocation mémoire des signaux
   // ETAPE 3 : Traitement du signal
25 foreach i_point ∈ [0; N_point[ do
       // ETAPE 3.1 : Sommation des contributions de chaque pinceau
26     foreach i_surfback ∈ [0; N_surfback[ do
27         foreach i_surfin ∈ [0; N_surfin[ do
28             foreach i_mode ∈ [0; N_mode[ do
29                 foreach i_capteur ∈ [0; N_capteur[ do
30                     foreach i ∈ [TdV − 0.5ΔT; TdV + 0.5ΔT] do
31                         RI_déplacement[N_point](t) += d⃗;
       // ETAPE 3.2 : Extraction du maximum d'amplitude du signal de
          déplacement
32     déplacement(t) = RI_déplacement[i_point](t) ⊗ sigref(t);
33     Amplitude(t) := Enveloppe( Module( déplacement(t) ) );
34     {(A_max, T_max)|A_max = Amplitude(T_max), ∀t A_max ≥ Amplitude(t)};
```

(d'échos utilisant ce calcul de champ) sont elles-mêmes validées par rapport à des acquisitions. Enfin, les versions optimisées qui sont présentées dans la suite du document ont été constamment vérifiées pour s'assurer que les différentes optimisations appliquées n'introduisent pas d'erreurs et que les résultats restent identiques à ceux de l'implémentation de référence.

Dans le cadre du contrôle non destructif par ultrasons, le champ peut désigner deux notions proches : soit l'image des signaux de déplacement, soit simplement les deux images, celle des amplitudes maximum du déplacement et celle des temps de vol associés. La première validation que réalise l'expert est simple en ce qu'il procède à une validation visuelle de l'image mais aussi complexe dans la mesure où d'une part il n'est pas possible de mesurer le champ directement (les seules références pour ce faire sont d'autres simulations ou éventuellement la mesure indirecte d'un champ transmis par un autre capteur en réception seule et situé à l'opposé de la pièce) et d'autre part la qualification de cette image est difficile. Dans le cas d'une simulation d'écho, les mesures existent et l'expert cherche à borner l'incertitude de la mesure par rapport à la simulation. L'usage est de tolérer des seuils de l'ordre de 2 à 3 décibels maximum. Le décibel ou *dB* est l'unité logarithmique exprimant le ratio entre deux valeurs : pour obtenir une mesure en dB de la valeur A par rapport à la mesure de référence A_{ref}, il faut calculer $20 \log \frac{A}{A_{ref}}$

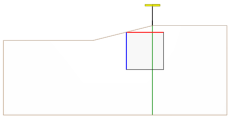
Suivant ces mêmes principes, les simulations de champs obtenues par les implémentations développées sont vérifiées sur un ensemble de configurations "de référence". En particulier, les critères étudiés sont :

- l'allure générale de l'image d'amplitude max du champ sur l'ensemble de la zone afin d'avoir une estimation qualitative sur la présence des phénomènes simulés. Par exemple, sur la figure 4.18b on observe deux faisceaux qui correspondent aux deux modes de champs simulés ;
- les échodynamiques décrivant l'allure du maximum de champ le long d'un axe, générale-

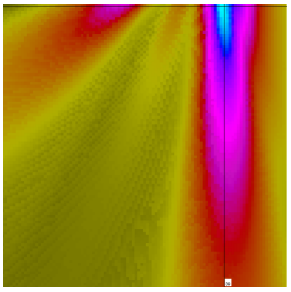
ment pour les zones 2D en profondeur et le long du plan d'inspection (figures 4.18d et 4.18e). Ceux-ci sont mesurés au niveau du point d'amplitude maximum et sur quelques autres points comparés manuellement (écart en amplitude et aspect qualitatif global des échodynamiques) ;

- le temps de vol associé avec le point d'amplitude maximum (écart en micro-secondes) ;
- la mesure de la tache focale du faisceau (hauteur et largeur, en millimètres, à -3dB du maximum d'amplitude) (figure 4.18c), à privilégier en calcul mode par mode sur des zones de champ plus larges que la tache focale ;
- l'étude des signaux de champ dans leur ensemble, en quelques points de la zone en commençant par le point de maximum d'amplitude et sur quelques autres mesures manuelles (figure 4.18f). En raison des bons résultats métier des validations, cette mesure n'a pas été déployée sur l'ensemble des configurations étudiées.

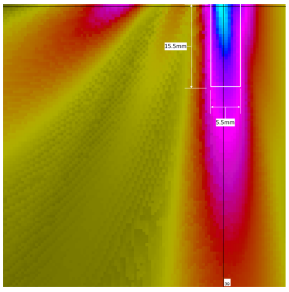
Les mesures réalisées, comme présentées ci-dessous, n'ont pas fait apparaître de différence majeure : le bon accord métier obtenu permet de se satisfaire de cette validation.



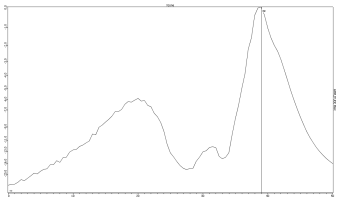
(a) Configuration de champ testée



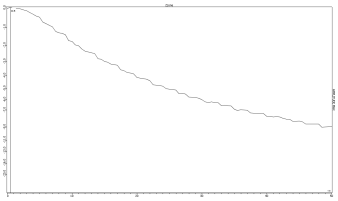
(b) Champ total en amplitude du module de déplacement



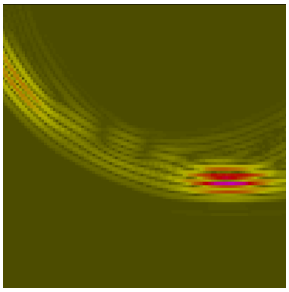
(c) Dimensions de la tache focale



(d) Échodynamique au point A_{max} le long de l'axe X



(e) Échodynamique au point A_{max} le long de l'axe Z



(f) Signaux temporels - $A(t)$ pour un t donné

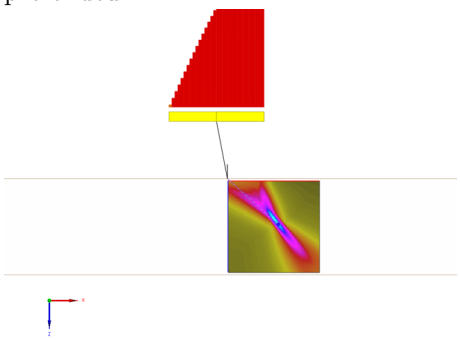
FIGURE 4.18 – Points de validation pour une configuration donnée

4.2.6.1 Configurations de validation

Afin de valider le calcul de champ, plusieurs configurations ont été mises en place. Dans une première série de configuration, le champ est vérifié de manière "unitaire", afin de qualifier mode par mode les simulations.

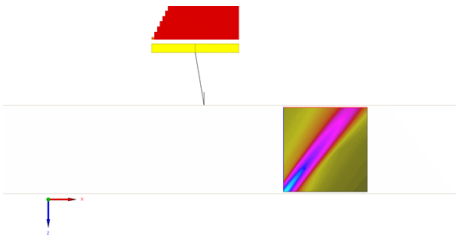
Ces configurations concernent toutes un capteur de 32 éléments en immersion, avec un signal émis de fréquence centrale à 2,5 MHz et échantillonné temporellement à 90 MHz, simulant une zone de 101×101 avec un pas spatial de $\lambda_{onde}/6$ (soit environ 0,4mm en mode L soit 40mm×40mm et respectivement 0,2mm et 20mm×20mm pour une onde T). La pièce considérée est une pièce plane en acier isotrope, aux surfaces d'entrée et de fond planes. La figure 4.19 présente, pour chaque mode considéré, la loi de retards sélectionnée.

Mode L direct : focalisation à 45 ° , à 30mm de profondeur



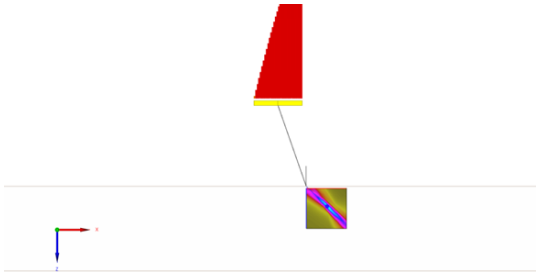
(a) Configuration mode L direct

Mode L avec rebond sans conversion : focalisation à 45 ° en un point après rebond



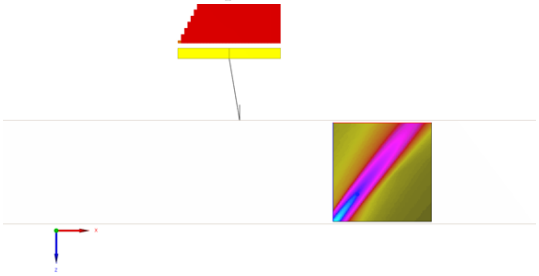
(b) Configuration mode L rebond L

Mode T direct : focalisation à 45 ° , à 10mm de profondeurs



(c) Configuration mode T

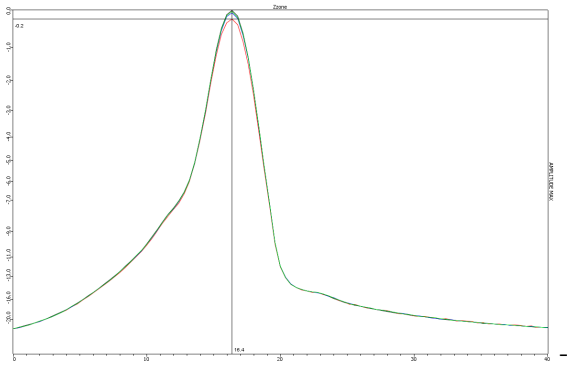
Mode T avec rebond sans conversion : focalisation à 45 ° en un point



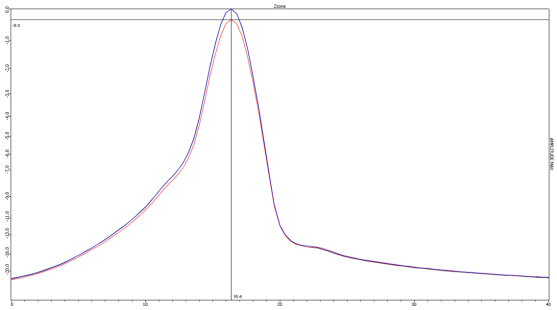
(d) Configuration mode T rebond T

FIGURE 4.19 – Illustrations des configurations de validation

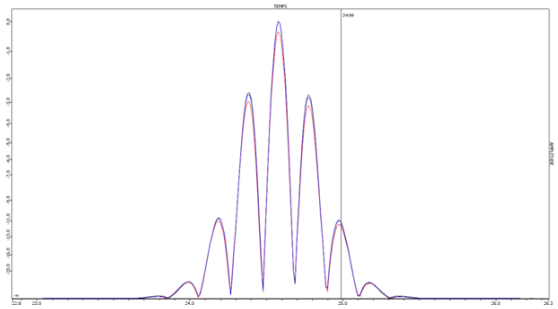
CIVA 11.0a a été utilisé comme référence sur ces configurations. Dans ce logiciel, un paramètre de précision permet de faire varier la précision des simulations de champ réalisées, ce qui influe sur le nombre de pinceaux utilisés pour réaliser les calculs, c'est à dire la finesse de l'échantillonnage de la surface du capteur. La table 4.1 donne ; pour chacune des configurations étudiées, les différences obtenues en dB pour différentes valeurs de précision des calculs de champs dans CIVA par rapport à un calcul CIVA en précision 10. On observe que la qualité des simulations semble converger progressivement. Dès que la précision est de 3, les écarts sont très faibles par rapport à la précision 10 (*c.f.* la figure 4.20a pour la configuration de référence, en L˚45). Ces mêmes calculs de champs CIVA en précision 10 servent de base de comparaison pour valider les calculs de l'implémentation de référence de l'algorithme rapide.



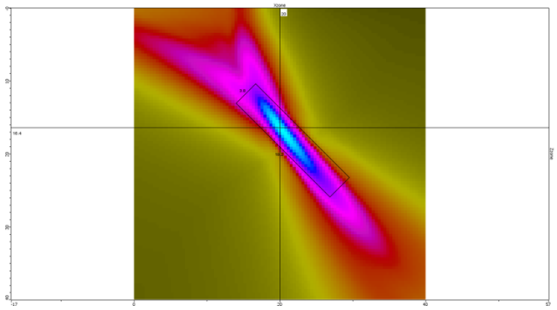
(a) Configuration de CIVA pour différentes précisions vs CIVA précision 10
Rouge : 1 ; Bleu : 3 ; Vert : 5 ; Noir : 10



(b) Comparaison de l'échodynamique de la zone de champ selon un axe
Rouge échantillonnage $\lambda/2$; Bleu : $\lambda/4$; Noir : CIVA précision 10



(c) Comparaison au point maximum des signaux de champ
Rouge échantillonnage $\lambda/2$; Bleu : $\lambda/4$; Noir : CIVA précision 10



(d) Mesure de la tache focale depuis l'interface CIVA
Échantillonnage $\lambda/4$

FIGURE 4.20 – Illustrations des mesures de validation sur la configuration L direct(figure 4.19a)

Configuration	Précision 1	Précision 3	Précision 5
L direct	0.2 dB	0.07 dB	0.04 dB
L rebond sans conversion	0.07 dB	0.01 dB	0.01 dB
T direct	0.5 dB	0.2 dB	0.07 dB
T rebond sans conversion	0.4 dB	0.08 dB	0.05 dB

TABLE 4.1 – Comparaisons des écarts CIVA11.0a précision 10 VS CIVA11.0a pour différentes valeurs de précision

A partir de cette version de référence, les configurations ont alors été simulées par l'implémentation du calcul de champ ultrasonore rapide de référence. Pour illustrer les mesures réalisées sur les différentes versions des calculs, la figure 4.20 récapitule ces mesures.

Les dimensions des taches focales (T.F. mesurée en mm à -3 dB, dans l'axe à 45 ° ou 135 ° correspondant au faisceau) (figure 4.20d), les écarts d'amplitude au point maximum et de temps de vol associé ont été mesurés (figure 4.20c).

Des comparaisons qualitatives des signaux au point de plus forte amplitude et des écho-dynamiques associés ont également été réalisées afin de déceler d'éventuels problèmes (figure 4.20b).

Deux finesses d'échantillonnage du capteur ont été utilisées pour qualifier les résultats de simulation rapide : en $\lambda/2$ et en $\lambda/4$ (λ longueur d'onde, selon le mode considéré). La table 4.2 récapitule l'ensemble de ces mesures, pour plusieurs configurations "basiques" ainsi que pour les deux finesses d'échantillonnage du capteur (écart d'amplitude en dB au point maximum, écart en temps de vol, écarts sur les dimensions de tache focale).

Config.	Échant. nb pinceaux/pt champ	Écart Amax	Écart Tmax	Écart Tache Focale	
				Largeur	Hauteur
L direct	$\lambda/2$=1.2 mm, 320 p.	0.3 dB	0 μs	3.8 vs 3.8 = 0.0 mm	18.3 vs 18.2 = 0.1 mm
	$\lambda/4$=0.6 mm, 1216 p.	0.01 dB	0 μs	3.8 vs 3.8 = 0.0 mm	18.2 vs 18.2 = 0.0 mm
L rebond	$\lambda/2$=1.2 mm, 320 p.	0.2 dB	0 μs	5.7 vs 5.8 = 0.1 mm	29.2 vs 28.8 = 0.4 mm
	$\lambda/4$=0.6 mm, 1216 p.	0.02 dB	0 μs	6.3 vs 5.8 = 0.5 mm	29.1 vs 28.8 = 0.3 mm
T direct	$\lambda/2$=0.65 mm, 544 p.	0.2 dB	0 μs	2.2 vs 2.1 = 0.1 mm	22.8 vs 22.7 = 0.1 mm
	$\lambda/4$=0.325 mm, 2176 p.	0.07 dB	0 μs	2.1 vs 2.1 = 0.0 mm	22.8 vs 22.7 = 0.1 mm
T rebond	$\lambda/2$=0.65 mm, 544 p.	0.2 dB	0 μs	3.6 vs 3.5 = 0.1 mm	25.3 vs 25.4 = 0.1 mm
	$\lambda/4$=0.325 mm, 2176 p.	0.04 dB	0 μs	3.6 vs 3.5 = 0.1 mm	25.6 vs 25.4 = 0.2 mm

TABLE 4.2 – Implémentation de référence VS CIVA11.0a précision 10

Les résultats présentés par le tableau 4.2 sont conformes aux observations réalisées à l'aide de CIVA 11.0a. Lorsque l'échantillonnage au niveau du capteur s'affine, le champ simulé devient de plus en plus précis et converge vers celui de CIVA. Les erreurs mesurées au niveau de la tache focale sont minimes.

La faiblesse des écarts constatés et cette convergence valident la simulation numérique du champ par l'implémentation de référence du point de vue métier sur ces configurations "unitaires". Les résultats obtenus à l'aide d'un échantillonnage de l'ordre de $\lambda/2$ sont déjà très satisfaisants. Lors des simulations interactives, la résolution proposée par défaut sera de cet ordre de grandeur tout en laissant une possibilité à l'utilisateur de modifier ce paramètre pour gagner en précision.

L'ensemble des configurations étudiées et leur validité sont présentées en annexe B.

4.3 Analyse de l'implémentation de référence

Après avoir décrit et validé l'implémentation de référence, cette section tente de qualifier son comportement à l'aide de diverses métriques et mesures. L'objectif de cette analyse consiste à déterminer l'optimalité d'une implémentation, à partir des capacités connues d'une machine, à qualifier cette optimalité et à modéliser les limites de la machine.

4.3.1 Analyse de haut niveau

La présente section réalise un tour d'horizon complet de la simulation de champ.

4.3.1.1 Format et stockage des données en mémoire

Les données de champ sont stockées en mémoire en utilisant le schéma *structure de tableaux*[5]. Par exemple, l'ensemble des pinceaux calculés est stocké comme une structure composée d'un

[5]Connue également en anglais comme *Structure of Arrays* ou *SoA*

tableau de déplacement (composante par composante), un tableau de temps de vol, un tableau d'étalement temporel et un tableau d'amplitude (parties réelle et imaginaire sous forme de tableaux distincts). Ainsi les composantes d'un pinceau sont stockées en mémoire au même index, dans les différents tableaux.

Le détail des différentes structures mémoires utilisées par la maquette est précisé dans l'annexe C.

4.3.1.2 Complexité algorithmique

A partir des algorithmes 7 et 8, on peut calculer les complexités algorithmiques de chacune des étapes de la simulation de champ, rappelées dans le tableau 4.3. On pose la valeur $N\_trajets$ le nombre de trajets (par point de champ, par élément de surface du capteur) calculée comme étant :

$$N\_trajets = N_{surfin} \times N_{modes\_direct} + N_{surfin} \times N_{surfback} \times N_{modes\_rebond}$$

L'étape 3.2 et ses sous-étapes font appel à un ensemble de sous fonctions aux complexités algorithmiques variées (FFT, convolutions et autres traitements du signal...) dépendant de la taille du signal.

Étapes de calcul	Complexité Algorithmique
Étape 1 : Calcul des pinceaux	$O\left(N_{point} \times N_{capteur} \times N_{trajets} \right)$
Étape 2 : Analyse de la taille des signaux	$O\left(N_{point} \times N_{capteur} \times N_{trajets} \right)$
Étape 3.1 : Sommation des pinceaux	$O\left(N_{point} \times N_{capteur} \times N_{trajets} \right)$
Étape 3.2 : Traitement du signal	$O\left(N_{point} \times size \times log(size) \right)$

TABLE 4.3 – Complexités algorithmiques

4.3.2 Intensité arithmétique

L'intensité arithmétique est une métrique théorique qu'il est possible de déterminer sur des algorithmes. Il s'agit de mesurer le ratio entre le nombre total d'opérations réalisées par rapport à la quantité de données déplacées (en octets). Par exemple, pour l'opérateur BLAS-1 *axpy* réalisant sur toute la longueur de deux vecteurs x et y l'opération suivante :

$$y[i] \leftarrow \alpha \times x[i] + y[i]$$

- deux données sont lues ;
- une donnée est écrite ;
- deux opérations sont réalisées (une addition et une multiplication).

L'intensité arithmétique de *axpy* sur des vecteurs de taille N flottants double précision, est de

$$\frac{\text{Nombre d'opérations flottantes}}{\text{Nombre d'accès mémoire}} = \frac{2N}{3*8N} = \frac{1}{12} \approx 0.0833 FLOP/Octet$$

.

Ainsi, une intensité arithmétique élevée indique un algorithme nécessitant beaucoup de calculs pour peu d'accès mémoire et inversement une intensité arithmétique faible indique un algorithme gourmand en accès mémoire. La figure 4.21 présente différents algorithmes usuels en termes d'intensité arithmétique.

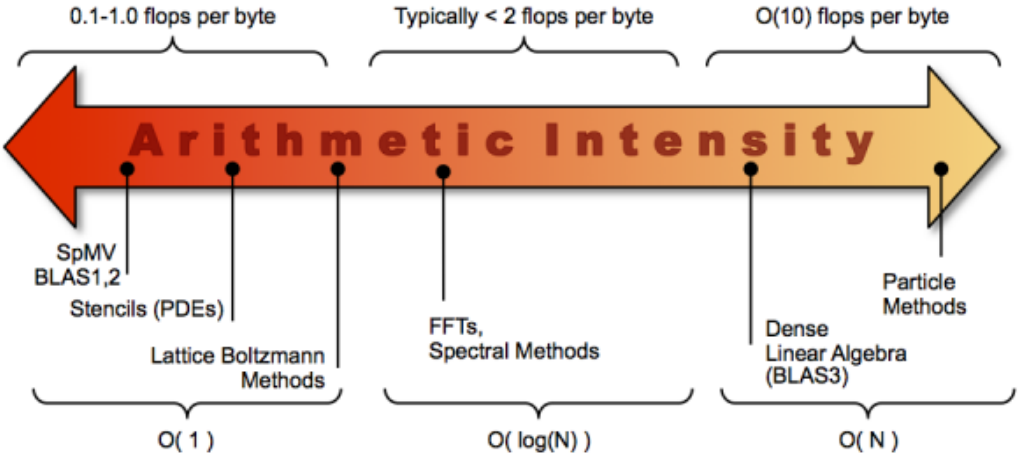


FIGURE 4.21 – Intensité arithmétique de différents algorithmes classiques, de la plus faible à gauche à la plus forte à droite

Source : Lawrence Berkeley National Laboratory, Département Compute Science, Californie, États-Unis
http://www.crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/roofline/

Il est à noter, d'un point de vue plus proche de l'implémentation, que les accès mémoire ne sont pas aussi facilement modélisables qu'un simple décompte utilisé classiquement avec le calcul d'une intensité arithmétique. Dans les GPP modernes, des caches entrent en jeu afin d'optimiser les accès. En particulier les caches récupèrent la mémoire par ligne de cache, d'une largeur supérieure à la taille d'un flottant (sur les GPP Intel étudiés, ces lignes sont larges de 64 octets, soit 16 flottants simple précision). Ainsi plusieurs données proches sont accédées en même temps pour un même coût en cycle. Une intensité arithmétique prenant en compte ces accès en cache permettrait de qualifier plus finement le comportement d'une implémentation donnée sur un GPP donné.

4.3.2.1 Roofline model

Le modèle *Roofline* est un modèle permettant de qualifier les performances d'un ou plusieurs algorithmes pour un GPP donné, sous la forme d'un graphique facilement interprétable. Il est composé de deux axes : l'axe des abscisses présente l'intensité arithmétique en flops/octet et l'axe des ordonnées correspond aux performances d'une implémentation en flops/cycle. Sur ce graphique sont présentées deux données d'entrée, servant de limite et de repères pour l'analyse : la performance crête du GPP en flops/cycles et la bande passante maximum de la mémoire en octets/cycles. Le schéma 4.22 présente le graphique associé pour un GPP donné, pour deux algorithmes. On y observe très clairement l'enveloppe des performances dans laquelle les algorithmes vont se trouver et quel est le facteur limitant d'une implémentation donnée.

Les données théoriques de performance crête et mémoire d'un GPP peuvent être obtenues à partir des documentations constructeurs. Par ailleurs l'utilisation de *benchmarks* usuels permet de déterminer les performances maximales atteignables par des algorithmes réalistes (ie : pas spécialement étudiés pour atteindre l'optimal d'un GPP donné mais des algorithmes réels utilisés

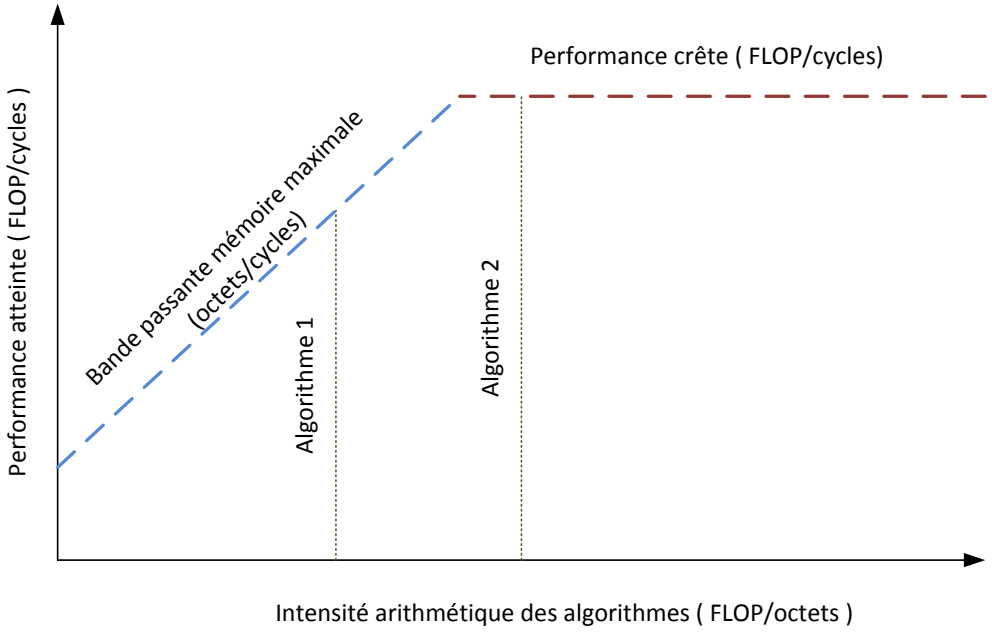


FIGURE 4.22 – Schéma du modèle *Roofline*

pour résoudre des problèmes concrets).

- Pour mesurer les instructions par cycle et autres FLOPs, l'algorithme LINPACK est l'un des plus classiques ; il s'agit du même genre d'algorithmes d'algèbre linéaire que ceux utilisés à beaucoup plus grande échelle sur les super-ordinateurs du top500.
- La mémoire peut être mise à l'épreuve par les logiciels STREAM ou encore PAPI afin d'en déterminer la bande passante maximale atteignable.

4.3.3 Un programme complexe

Dans sa globalité, l'algorithme de calcul de champ est complexe et son comportement est grandement lié aux données d'entrée. Ainsi la géométrie et les modes souhaités influent le comportement du calcul des pinceaux, les retards influent sur l'étape de sommation... Les différentes sous-étapes du calcul de champ ont des comportements variés.

Afin de qualifier l'algorithme dans son ensemble, chacune de ses étapes et sous-étapes de calcul sera identifiée qualitativement.

4.3.3.1 Mesure automatique de l'intensité arithmétique

Sur des algorithmes simples dont le noyau de calcul est petit, le calcul de l'intensité arithmétique est réalisable manuellement. Mais ici, dans le cadre du calcul de champ, la présence d'étapes complètement dépendantes des données d'entrées et imprévisibles (par exemple la sommation des contributions des pinceaux (Étape 3.1) qui dépend des caractéristiques de la configuration), rend une approche manuelle non fiable et non vérifiable.

Une première approche a été d'utiliser les capacités du langage C++ pour surcharger le type `float` par une classe chargée de compter les différentes opérations et en ajoutant des compteurs

aux accesseurs de données en mémoire (dans des tableaux). Cette première approche a montré plusieurs limites. Tout d'abord, le décompte exhaustif ne tient pas compte des éventuelles optimisations que sait réaliser le compilateur sur les opérations comme, par exemple, les factorisations ou les précalculs/optimisations de certaines constantes... Ensuite, les accès mémoire décomptés tiennent compte de l'implémentation réelle mais pas de l'algorithme théorique (fonction décomposée en sous fonction pour gain de maintenabilité...). Pour ces raisons, les résultats obtenus par cette méthode sur un code complexe ne permettent pas de caractériser de façon théorique l'algorithme car ils sont trop dépendants d'une implémentation donnée.

L'autre approche étudiée a été l'usage d'un outil spécialisé, l'*Intel Software Development Emulator* (Intel SDE), afin d'obtenir directement une mesure depuis un code compilé binaire : le GPP considéré est simulé et les différentes micro-opérations utilisées sont décomptées. Bien qu'il dispose de plusieurs fonctions de paramétrages afin d'obtenir un résultat précis, l'objet de l'analyse est un exécutable dans son ensemble. De plus, le logiciel remonte les informations concernant les opérations utilisées, mais pas la manière dont celles-ci sont utilisées. Par exemple les opérations flottantes scalaires peuvent parfois être déportées automatiquement par le compilateur sur l'unité de calcul vectoriel en occupant un seul emplacement du vecteur. Intel SDE affiche alors de la même manière une telle opération (un flottant) qu'un usage SIMD classique (plusieurs flottants). Cet outil n'a pas permis d'effectuer une mesure de l'intensité arithmétique des algorithmes développés de façon satisfaisante.

Face à ce constat, le choix a été fait de procéder par une étude des performances à la fois relatives et absolues, afin d'en tirer les conclusions d'adéquation algorithme/architecture.

4.3.4 Les différentes mesures de performances

4.3.4.1 Mesure de passage à l'échelle

Mesurer le passage à l'échelle d'une implémentation sur un matériel donné consiste à mesurer, sur un processeur avec une capacité de N calculs en parallèle l'accélération entre un algorithme séquentiel sur une unité de calcul et une implémentation de cet algorithme réparti sur les N unités de calcul. Ainsi sur un GPP, une première mesure peut être réalisée quant au passage à l'échelle d'un code sur les différents cœurs du processeur. Une seconde mesure peut étudier l'impact des instructions vectorielles, en fonction du type flottant choisi. Enfin, une dernière information peut comparer une version de référence séquentielle scalaire avec une version parallélisée et vectorisée d'un algorithme. On peut calculer l'efficacité de l'optimisation entre le gain mesuré et le gain théorique maximal et l'exprimer en pourcentage.

Dans l'analyse, il faut également procéder avec précaution pour éviter les cas de mauvaise foi dans lesquelles des études présentent des valeurs d'accélération extraordinaires en comparant un code optimisé sur une machine puissante avec une implémentation non optimale sur une machine modeste. Ces pratiques ont été dénoncées dès 1991 par D. H. Bailey dans *Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers* [Bai91]. Elles sont depuis régulièrement remises à jour et adaptées à des environnements d'optimisations différentes, dans différentes publications et articles. On peut citer [Hag10] [Pak11] et [Bai11] par exemple.

Cette précaution est d'autant plus nécessaire lorsque l'on compare deux matériels différents : tous n'ont pas les mêmes capacités d'accélération ni les mêmes performances crêtes. Ainsi, un GPU n'est pas fait pour fonctionner en mode *monothread* et comparer les performances d'une implémentation "monothread GPU" avec des performances utilisant les centaines/milliers de cœurs du GPU n'a pas grand sens. De même, il n'est pas possible de restreindre l'exécution d'un noyau CUDA sur un seul *streaming multiprocessor* à moins de limiter l'exécution à un seul bloc, ce qui altère le comportement d'un l'algorithme. Enfin, comparer les performances d'un GPU

haut de gamme avec un GPP médiocre permet d'obtenir des accélérations spectaculaires au prix d'un faible effort.

4.3.4.2 Mesures absolues

Les mesures absolues sont des informations brutes sur le temps d'exécution des différentes implémentations pour les différentes architectures. Ces mesures ne permettent pas facilement de qualifier une implémentation par rapport aux capacités d'un matériel pour juger de son optimalité. Il faut réaliser des études comparatives sur l'évolution individuelle des différents paramètres.

La mesure de performance dans l'absolu permet d'obtenir une indication définitive pour qualifier l'interactivité d'un algorithme de simulation de champ : une mesure du temps d'exécution se transforme en un nombre d'images par seconde par simple division, ce qui permet de vérifier si la simulation est interactive ou non (fournissant plus de 25 images de champ par seconde selon le critère défini au paragraphe 2.1.3.2) qui permet de savoir si le niveau de performance atteint est satisfaisant.

4.3.5 Configurations de référence

Le paragraphe 4.2.5.1 présente les différents paramètres d'entrée de l'algorithme de calcul de champ. A partir d'une première configuration d'origine, un jeu de configurations est défini en faisant évoluer un à un les paramètres métier indépendamment les uns des autres.

- En augmentant le nombre de points de champs, N_{points}, le temps de calcul de la simulation doit augmenter linéairement.
- En augmentant la finesse de l'échantillonnage du capteur, $N_{capteurs}$, la quantité de pinceaux calculés augmente. Par contre, leur temps de vol étant sensiblement le même que celui de pinceaux d'une finesse plus petite, les signaux sur lesquels leurs contributions vont venir se sommer restent de taille identique.
- En augmentant l'échantillonnage temporel des signaux, leur taille $size$ en nombre d'échantillons augmente linéairement, impactant les étapes de traitement du signal.
- En utilisant différentes lois de retards, il est possible de focaliser plus ou moins le faisceau du capteur dans la zone simulée : d'une part les signaux sont regroupés sur un plus petit étalement temporel, d'autre part cela impacte les besoins de synchronisation venant sommer plusieurs contributions au même endroit temporel (et donc au même emplacement mémoire).
- En augmentant le nombre de modes simulés, l'étape de calcul des pinceaux est influencé quasi linéairement (des divergences peuvent apparaitre entre les modes directs et rebonds). Cependant, le mode influe sur le temps de vol des pinceaux, et donc sur la taille des signaux.
- En complexifiant la géométrie de la pièce simulée (N_{surfin} et $N_{surfback}$) le nombre de pinceaux possibles pour un point de champ augmente d'autant :

$$N_{capteur} * (N_{surfin}(N_{modes\_directs} + N_{surfback} * N_{modes\_rebonds}))$$

Les différentes configurations sont présentées en détail en annexe B.

4.3.6 Conclusions sur l'analyse de l'implémentation de référence

Afin de faire un premier état des lieux sur l'implémentation de référence sur l'ensemble des configurations étudiées, ses performances ont été mesurées sur les différentes étapes du calcul de champ. La figure 4.23 présente cette répartition sur le GPP Xeon Westmere pour un code *mono-thread* (les caractéristiques de ce GPP sont présentées dans le chapitre traitant des optimisations GPP, à la section 5.1). Les performances des différentes étapes de calcul y sont rassemblées par configuration en un histogramme de somme totale 100% du temps de calcul. On y observe le côté prépondérant des étapes 1 et 3 par rapport à l'étape 2 du calcul de champ.

Les performances, à la fois en temps et en images par seconde, de cette implémentation de référence *monothread* sur le GPP Xeon Westmere, sont données par la table 4.4. Les performances de cette implémentation sont bien plus lentes que l'objectif que l'on s'est fixé de simulations de champ interactives. Suivant la configuration étudiée, la cadence se simulation est comprise entre une image et un demi centième d'image par seconde.

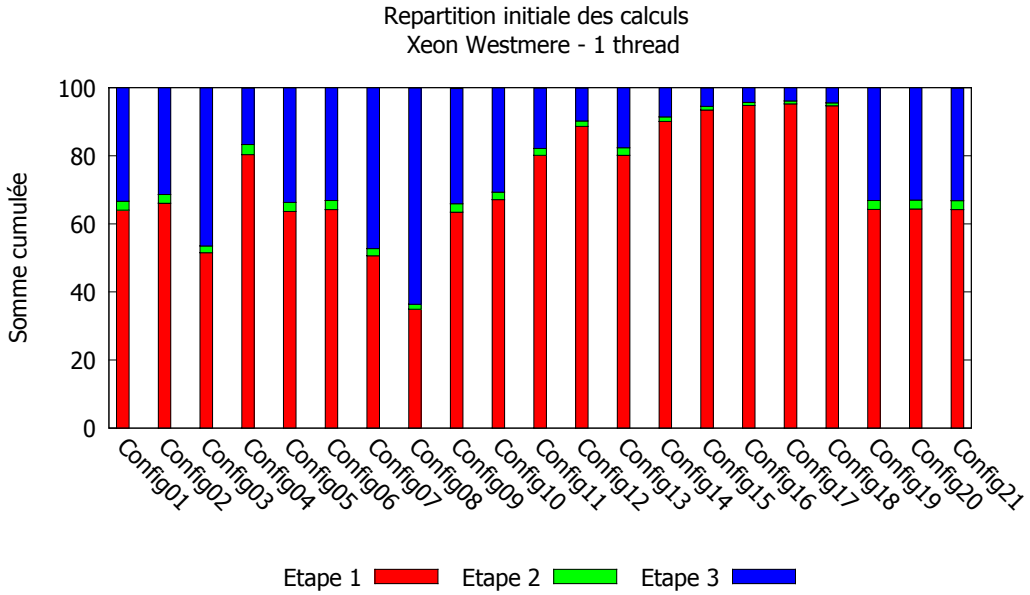


FIGURE 4.23 – Répartition des calculs par étape du calcul de champ sur l'implémentation de référence (GPP Xeon Westmere)

Ces mesures montrent qu'il est nécessaire d'optimiser les performances de la simulation de champ afin d'obtenir sur GPP des simulations interactives. En particulier, le rapport de temps écoulé dans les étapes 1 et 3 par rapport à l'étape 2 indiquent que ces deux étapes du calcul devront être traitées en priorité.

	2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz Westmere	
Config01	1850 ms	0.54 fps
Config02	2006 ms	0.50 fps
Config03	4986 ms	0.20 fps
Config04	6611 ms	0.15 fps
Config05	7294 ms	0.14 fps
Config06	28978 ms	0.03 fps
Config07	2343 ms	0.43 fps
Config08	3402 ms	0.29 fps
Config09	3884 ms	0.26 fps
Config10	8849 ms	0.11 fps
Config11	4349 ms	0.23 fps
Config12	12940 ms	0.08 fps
Config13	2192 ms	0.46 fps
Config14	5834 ms	0.17 fps
Config15	18478 ms	0.05 fps
Config16	54973 ms	0.02 fps
Config17	182315 ms	0.005 fps
Config18	199036 ms	0.005 fps
Config19	1863 ms	0.54 fps
Config20	1869 ms	0.54 fps
Config21	1868 ms	0.54 fps

TABLE 4.4 – Performance de l'implémentation de référence du calcul de champ (GPP Xeon Westmere - exécution *monothread*)

Chapitre

5 Optimisations sur architectures généralistes

Cette section présente tout d'abord l'index des machines sur lesquelles les optimisations sur les processeurs généralistes (GPP, *General Purpose Processor*) vont être réalisées et mesurées. Celles-ci sont présentées en distinguant d'une part les optimisations de haut niveau et d'autre part celles, plus bas niveau, reposant sur l'utilisation d'instructions SIMD. Les impacts de ces différentes optimisations sont ensuite analysés, étape par étape, avant d'en réaliser une synthèse. Enfin, des conclusions quant à l'utilisation de ce type d'architecture seront tirées.

5.1 Index des GPP étudiés

Dans cette section du chapitre traitant du calcul de champ, l'architecture GPP-x86 est abordée au moyen d'un échantillon représentatif des GPP du moment. Celui-ci est décrit dans le tableau 5.1.

Famille	Gamme Xeon/Core	Microarchitecture	Modèle (et nb. GPP)	Fréquence GPP	N nb cœurs	Canaux mémoire par GPP
Xeon Westmere	Xeon	Westmere	2×X5690	3.47 GHz	2×6	3
Xeon Sandy Bridge	Xeon	Sandy Bridge	E3-1290	3.60 GHz	4	2
Xeon Ivy Bridge	Xeon	Ivy Bridge	E5-1650v2	3.50 GHz	6	4
Xeon Ivy Bridge 2x12	Xeon	Ivy Bridge	2×E5-2697v2	2.70 GHz	2×12	4
Xeon Haswell	Xeon	Haswell	E3-1240v3	3.40 GHz	4	2

TABLE 5.1 – Liste des GPP étudiés et leurs caractéristiques

Sur ces GPP, les performances ont été mesurées sous Linux, au moyen d'exécutables compilés pour chacune de ces architectures. Le compilateur employé est Intel C et C++ Compiler version 15.0.0.

5.2 Optimisations de haut niveau

5.2.1 Parallélisation multithread via OpenMP

La première optimisation consiste à répartir les traitements sur les différents cœurs du GPP en faisant appel à la bibliothèque OpenMP.

Le code de l'implémentation de référence est naturellement réentrant et les traitements sont répartis sur les points de champ au moyen de boucles. OpenMP permet de paralléliser aisément ces boucles `for()`. Sur l'algorithme de référence 7 et 8 cette parallélisation est appliquée sur les boucles parcourant les différents points de champ :

Étape 1 calcul des réponses impulsionnelles élémentaires à la ligne 1 de l'algorithme 7 ;

Étape 2 recherche de la taille des signaux à la ligne 2 de l'algorithme 4 ;

Étape 3 sommation des réponses impulsionnelles à la ligne 25 de l'algorithme 8 ;

Concernant le paramétrage de cette parallélisation, deux choix ont été faits :

- L'utilisation d'un ordonnancement dynamique, car la charge de travail sur chaque point n'est pas la même (divergence au niveau de Newton, sommations des contributions).
- La répartition des données par *chunk* de plusieurs points de champ pour ne pas que les *threads* OpenMP se retrouvent affamés de calculs et que les coûts de changement de contexte deviennent trop importants (même si l'*HyperThreading* l'allège en tenant deux *threads* en vie sur le cœur).

5.2.2 Fusion des boucles - Algorithme vertical

En fusionnant les boucles sur les points de champ de l'algorithme de référence, on peut regrouper les traitements par point de champ pour obtenir l'algorithme 9. Il présente une algorithmie verticale. Il est à noter que la version de la fonction $RechercheTailleSignal(\ldots)$ travaille désormais sur les réponses impulsionnelles locales au point de champ considéré. Cette version présente de prime abord un avantage théorique relatif à l'empreinte mémoire par rapport à la version de référence. Il n'est plus nécessaire d'allouer la mémoire pour l'ensemble des opérations, chaque itération de la boucle principale travaillant sur un espace local à un seul point de champ. L'optimisation des zones mémoire attribuées n'a pas été mise en place dans la maquette développée afin d'utiliser les mêmes structures mémoires pour toutes les implémentations présentées.

La boucle principale travaillant sur les points de champ, à la ligne 1 de l'algorithme 9, peut être parallélisée à l'aide d'OpenMP de la même manière qu'évoqué lors du paragraphe précédent. Le second avantage de cette fusion est que dans ce contexte d'exécution parallèle, chaque *thread* a plus de traitements à réaliser. Par contre, cette version du calcul de champ utilise des signaux temporels de taille variable (en nombre d'échantillons) et doit donc créer autant de plans pour les FFT que de tailles différentes.

Algorithme 9 : Calcul de champ vertical

entrées :
Description de la structure inspectée
Description du capteur subdivisé en $N_{capteur}$ éléments
Taille utile des signaux $size$

sorties :
A_{max}, l'amplitude du maximum de déplacement
T_{max}, le temps de vol correspondant à A_{max}

foreach $i_{point} \in [\![0; N_{point}[\![$ **do**
  `// ETAPE 1 : Calcul d'un pinceau`
  foreach $i_{surfback} \in [\![0; N_{surfback}[\![$ **do**
    foreach $i_{surfin} \in [\![0; N_{surfin}[\![$ **do**
      foreach $i_{mode} \in [\![0; N_{mode}[\![$ **do**
        foreach $i_{capteur} \in [\![0; N_{capteur}[\![$ **do**
          `// ETAPE1.1 : Détermination du trajet du pinceau`
          `// ETAPE1.2 : Caractéristiques du pinceau`
          `...;`
          `// Même principe que l'algorithme "horizontal" présenté en 7 et 8`
  `// ETAPE 2 : Recherche de la taille des signaux`
  RechercheTailleSignaux(...);
  $\overrightarrow{RI_{déplacement}}(t) = \vec{0}$; `// Allocation mémoire des signaux`
  `// ETAPE 3 : Traitement du signal`
  `// ETAPE 3.1 : Sommation des contributions de chaque pinceau`
  foreach $i_{surfback} \in [\![0; N_{surfback}[\![$ **do**
    foreach $i_{surfin} \in [\![0; N_{surfin}[\![$ **do**
      foreach $i_{mode} \in [\![0; N_{mode}[\![$ **do**
        foreach $i_{capteur} \in [\![0; N_{capteur}[\![$ **do**
          foreach $i \in [\![TdV - 0.5\Delta T; TdV + 0.5\Delta T]\!]$ **do**
            $\overrightarrow{RI_{déplacement}}(t) += \overrightarrow{d}$;
  `// ETAPE 3.2 : Extraction du maximum d'amplitude du signal de déplacement`
  $\overrightarrow{displacement}(t) = \overrightarrow{RI_{déplacement}}(t) \bigotimes impulse(t)$;
  Amplitude(t) := Enveloppe(Module($\overrightarrow{displacement}(t)$));
  $\{(A_{max}, T_{max}) | A_{max} = Amplitude(T_{max}), \forall t\ A_{max} \geq Amplitude(t)\}$;

5.2.3 Utilisation de la bibliothèque Intel MKL

Le calcul de champ nécessite, en chaque point de champ, plusieurs opérations de type Transformée de Fourier Rapide, comme présenté par le paragraphe 4.2.4.2.1 :

- Afin de convoluer les 3 coordonnées des réponses impulsionnelles, on utilise 6 appels à FFT-R2C et 3 appels à FFT-C2R.

- Afin d'obtenir l'enveloppe du module du signal de déplacement, on fait appel 2 fois à FFT-C2C, une fois dans le sens direct et une fois dans le sens inverse.

L'implémentation de référence, qui reprend des codes du traitement du signal issus de CIVA, utilise une implémentation des opérations de FFT inspirée des codes proposés dans *Numerical Recipes* (NRC) [PTVF07]. Elle est simple d'utilisation mais ne tire pas parti des spécificités matérielles des processeurs modernes, ni des caches, ni des instructions SIMD.

Afin d'atteindre le maximum de performances, les implémentations de la bibliothèque Intel MKL sont utilisées pour effectuer ces opérations de FFT. Cette bibliothèque a été présentée à la sous-section 3.2.3.2. Sur l'algorithme de référence 8, la taille des signaux est obtenue à la ligne 23. Après celle-ci, une phase d'initialisation des plans FFT est ajoutée. Sur l'algorithme vertical 9, les tailles de signaux sont locales à un point de champ et peuvent varier d'un point à l'autre.

Il convient de remarquer que les signaux utilisés sur différents points de champ sont souvent de tailles similaires : en effet, la FFT de la MKL comme celle de NRC fonctionnent sur des signaux de taille "puissance de 2"[1]. Les FFT de la MKL peuvent donc réutiliser les plans d'un point de champ à l'autre. L'implémentation a été réalisée de façon à permettre une initialisation des plans de FFT à la volée : lorsqu'un point de champ nécessite une taille, si le plan associé n'existe pas encore, un verrou est placé pour le créer, sinon le plan existant est utilisé.

Afin d'évaluer son efficacité, un *minibenchmark* spécifique à été mis en place. Il permet d'appliquer les opérations de FFT sur des signaux de longueur variable avec des contenus aléatoires au moyen d'un code séquentiel. Dans la mesure où les plans ne seront pas recalculés dans le contexte de la simulation de champ, ce *minibenchmark* est réalisé, pour la MKL, sur le calcul uniquement. On compare ainsi l'efficacité du code originel issu des FFT CIVA avec celle des FFT optimisées de la bibliothèque MKL.

Les performances de ce *minibenchmark* sont présentées par la figure 5.1 sur plusieurs GPP de type Xeon. On y observe, pour une taille de signaux réaliste dans le cadre du calcul de champ, les performances de l'implémentation FFT CIVA et de la bibliothèque MKL. Ces performances sont présentées en cycles, **normalisés** par la complexité des calculs d'une FFT C2C : $O(N \cdot logN)$. On y observe, sur les différentes architectures, une très nette accélération en passant de la FFT CIVA à la FFT MKL.

De plus, ces performances normalisées devraient rester stables quelle que soit la taille des signaux. Pour les signaux très petits, la FFT MKL présente un pic en temps de calcul. Cette bibliothèque utilise un mécanisme de plans pour les FFT, ces plans sont des coefficients précalculés qui permettent d'accélérer les calculs au prix d'un stockage en mémoire. On peut, en conséquence, supposer que ce décrochage est lié à la quantité de mémoire nécessaire pour les FFT MKL dont l'accès n'est pas sans coût.

Lorsque l'on observe l'autre côté du spectre, les performances normalisées de la FFT MKL restent stables alors que celles de la FFT CIVA subissent une augmentation importante pour des signaux plus grands (qui semble linéaire avec la taille des signaux pour l'unité utilisée). Celle-ci se fait "en place" et n'utilise pas un plan afin de pré-calculer et réutiliser des valeurs intermédiaires. L'absence de précalcul des données du plan sur la FFT CIVA entraîne une explosion des temps d'accès.

Sur la plage correspondant aux tailles de signaux qui sont utilisés pour le calcul de champ, les FFT MKL présentent des gains allant de $\times 5$ à $\times 10$ par rapport aux FFT CIVA. Il faut aussi rappeler que le code des FFT CIVA est un code scalaire, alors que la bibliothèque MKL propose des routines vectorisées suivant les jeux d'instructions disponibles de chaque machine. L'objectif de la nouvelle implémentation étant la performance en temps d'exécution, dans le contexte du

[1] La FFT fournie par la MKL peut fonctionner sur des signaux de taille arbitraire, mais, d'après la documentation sur certains multiples dont les "puissances de 2" ses performances sont optimales.

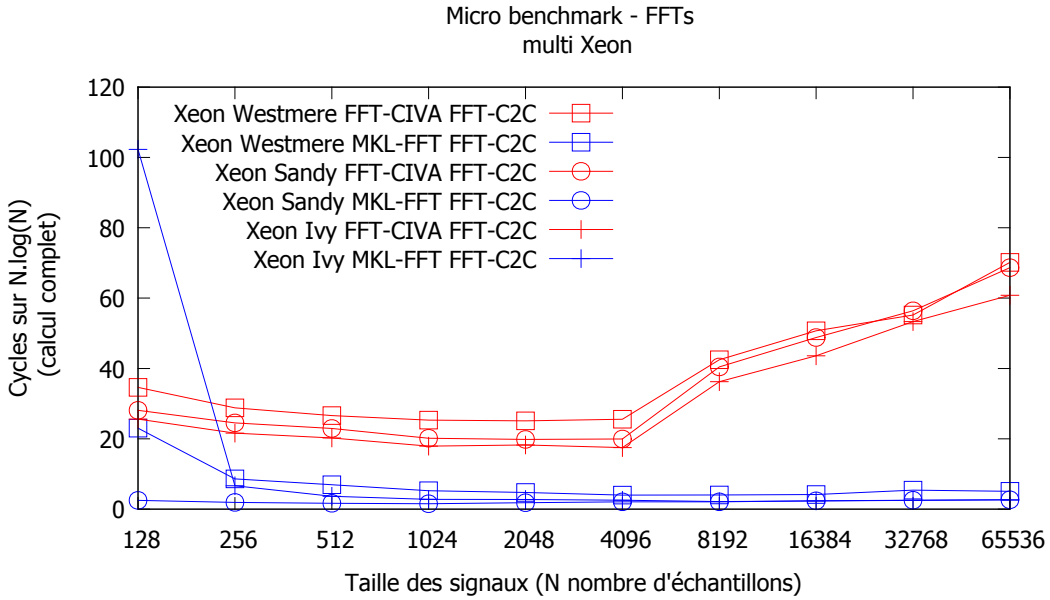


FIGURE 5.1 – Performances de la FFT C2C : implémentation FFT MKL vs FFT CIVA sur plusieurs GPP Xeon - en bleu la FFT MKL, en rouge la FFT CIVA ; les points sont associés par GPP

calcul de champ. Les signaux seront de taille "moyenne" sur les configurations de champ classique (au moins 256 éléments). Ces observations justifient le remplacement des opérations de FFT par des appels à la bibliothèque MKL FFT.

5.3 Instructions SIMD et optimisations de bas niveau

Afin de tirer parti des instructions SIMD disponible sur GPP, les différentes étapes du calcul de champ ont été vectorisées sur GPP au moyen de Boost.SIMD.

5.3.1 Étape 1 : Calcul des pinceaux

Le calcul des pinceaux est une étape qui offre un calcul *a priori* régulier sous certaines conditions. Pour déterminer les caractéristiques des pinceaux passant par le même ensemble d'interfaces pour un même mode donné, on effectue des calculs très similaires. Les pinceaux calculés ont donc été regroupés naturellement, par mode, pour calculer les trajets entre un point de champ et plusieurs échantillons capteur suivant vec_len le cardinal des registres SIMD utilisés.

$$pinceau\_SIMD = \{pinceau[i+0], pinceau[i+1], \ldots, pinceau[i+vec\_len-1]\}$$

Lorsque le nombre d'échantillons n'est pas un multiple de la largeur du vecteur, le calcul des quelques derniers pinceaux est réalisé en scalaire. La validation des simulations de champ obtenues a permis d'activer les opérations FMA sur l'architecture l'autorisant (Haswell) : la modification de l'arrondi du calcul n'affecte pas la qualité des trajets trouvés.

En entrée, le stockage en mémoire des informations de capteur sous forme de structure de tableau permet un chargement direct dans des registres SIMD. De même, en sortie, le stockage

des informations pinceaux en mémoire sous cette même forme permet un transfert direct des informations depuis les registres SIMD vers la mémoire.

La majorité des calculs réalisés au niveau du calcul des pinceaux reste identique au niveau algorithmique, réalisant uniquement la même opération sur plusieurs données à la fois.

5.3.1.1 Étape 1.1 : Calcul des trajets analytiques

Le calcul des trajets analytiques se déroule en deux phases. La première consiste à résoudre le polynôme de degré 4 correspondant au trajet recherché afin d'obtenir le point d'impact du rayon sur la surface d'entrée considérée. La seconde vérifie la validité géométrique des trajets recherchés.

5.3.1.1.1 Polynômes et méthode de Newton La méthode de Newton est une méthode itérative dont la convergence est quadratique si la solution initiale choisie est proche de la solution recherchée (et linéaire sinon), ce qui est en général le cas pour l'application. En revanche, le nombre d'itérations peut varier d'un polynôme à un autre (lorsque la racine trouvée est suffisamment proche du zéro machine). Dans l'implémentation actuelle, pour ne pas itérer un trop grand nombre de fois, ce nombre est borné à 10 itérations. Dans son implémentation vectorielle, il est nécessaire de prendre en compte cette variation : il ne faut pas continuer à raffiner les solutions qui ont atteint le critère d'arrêt pour éviter des instabilités numériques. L'algorithme 10 présente les évolutions adoptées pour gérer cette divergence au moyen des vecteurs logiques contenant *vec_len*, le cardinal des registres SIMD, booléens. A l'exception de la fonction `simd::all()`, les fonctions utilisées ici sont tout à fait équivalentes à leurs homologues scalaires. `simd::all()` quant à elle assure le lien avec le contrôle de la boucle `for` de la ligne 8 à l'aide d'un booléen scalaire obtenu par réduction des différents éléments de son vecteur argument (opérateur `ET` logique).

5.3.1.1.2 Construction et validation des trajets Le calcul de trajet est vectorisé entre un point de champ et plusieurs éléments pour un mode et une géométrie (surface d'entrée ou couple de surface entrée/fond) donnés. Le calcul du trajet d'un pinceau consiste à se ramener au cas canonique présenté au paragraphe 4.2.1.1 par des transformations géométriques, afin de former le polynôme de degré 4 correspondant. La recherche du zéro de ce polynôme permet d'obtenir le point d'intersection du trajet avec la surface plane considérée. Ce point étant obtenu par rapport au plan infini contenant la surface, des validations géométriques ont lieu pour s'assurer qu'il se trouve effectivement dans le rectangle correspondant à la surface plane. La validation géométrique d'un paquet de trajets est ainsi complètement régulière sur le plan algorithmique. Les données de géométrie sont dupliquées *vec_len* fois dans les registres SIMD pour permettre de traiter autant de trajets à la fois.

5.3.1.2 Etape 1.2 : Caractéristiques des pinceaux

Les caractéristiques des pinceaux sont leur temps de vol, leur étalement temporel, leur direction de déplacement et l'amplitude de ce dernier. Les pinceaux sont groupés, pour le calcul SIMD, par type de trajet et couple de surfaces. Pour tous les trajets d'un même registre SIMD, le temps de vol est obtenu simplement à partir des distances parcourues dans les différents matériaux et des vitesses de l'onde dans ceux-ci. L'étalement temporel est obtenu à partir des informations sur les échantillons capteur (forme et normale) et de la direction des trajets. De même, la direction du déplacement est obtenue, au point de calcul, à partir de la direction du trajet. Les pinceaux partageant un même mode ont tous la même polarité.

Algorithme 10 : Algorithme de Newton SIMD

entrées :
X_0 vecteur des valeurs initiales proposées
poly_simd les coefficients des *vec_len* polynômes étudiés
vec_len le cardinal des registres SIMD

sorties :
X_n vecteur des racines trouvées

```
// simd_fp_t est le type vecteur flottant de largeur vec_len
typedef boost::simd::pack<float> simd_fp_t ;
// simd_logical_fp_t est le type vecteur logique de largeur vec_len
typedef boost::simd::pack<boost::simd::logical<float» simd_logical_fp_t ;
simd_fp_t xn = x0;
simd_fp_t fxn, fpxn;
values2real(poly_simd, xn, fxn, fpxn);
// Calcul les P[xn] et P'[xn] à coefficients réels
simd_fp_t abs_fpxn = simd::abs(fpxn);
simd_fp_t abs_fxn = simd::abs(fxn);
simd_logical_fp_t is_fxn_lt_epsilon = abs_fxn < SIMD_FP_EPSILON;
simd_logical_fp_t is_finished = is_fxn_lt_epsilon && (abs_fpxn <
SIMD_FP_EPSILON);
for (it = 0; it<MAXITE AND (simd::all(is_finished) == false); it++ ) do
    simd_fp_t descente = fxn / fpxn;
    simd_fp_t xn_plus_1 = xn - descente;
    simd_fp_t abs_descente = simd::abs(descente);
    simd_logical_fp_t is_descente_lt_epsilon = abs_descente <
    SIMD_FP_EPSILON;
    // dernière étape
    xn = simd::if_else(is_finished, xn, xn_plus_1);
    // simd::if_else équivalent de l'opérateur ternaire, élément par élément
    is_finished = is_finished OR is_descente_lt_epsilon ;
    values2real(poly_simd, xn, fxn, fpxn);
    abs_fxn = simd::abs(fxn) ;
    is_fxn_lt_epsilon = abs_fxn < SIMD_FP_EPSILON;
    // P(xn)==0 !
    abs_fpxn = simd::abs(fpxn);
    const simd_logical_fp_t is_fpxn_lt_epsilon = abs_fpxn < SIMD_FP_EPSILON;
    // Division par P'(xn)==0 !
    is_finished = is_finished OR is_fpxn_lt_epsilon OR is_fxn_lt_epsilon;
```

Les calculs des caractéristiques de pinceaux contigus sont très réguliers. Algorithmiquement, des divergences peuvent survenir lors des calculs des coefficients de Fresnel pour un ensemble de trajets donné, en fonction de l'angle d'incidence par rapport à l'angle critique. Les pinceaux sont regroupés dans un registre SIMD en des pinceaux physiquement proches, ils sont en général tous sur- ou sous-critiques. Afin de systématiser le calcul, toutes les branches (angle d'incidence en dessous, égal à et au dessus de l'angle critique) sont calculées puis un masque sélectionne la valeur correspondant à la valeur juste à travers les trois résidus. Une simple addition des registres composés de 0 et de valeurs permet de reconstruire, sur toute la largeur des instructions

SIMD utilisées, les caractéristiques correspondant aux différents pinceaux évalués dans le paquet courant. L'algorithme 11 présente une vision globale, pour trois valeurs possibles de l'angle d'incidence (avant, égal et après l'angle critique).

Algorithme 11 : Algorithme de Newton SIMD

entrées :
θ vecteur des angles d'incidence
θ_{crit} l'angle critique pour le mode considéré
sorties :
F le coefficient calculé

```
// Les masques sont des registres SIMD
1 mask_under_crit = θ < θ_crit;
2 mask_equal_crit = θ == θ_crit;
3 mask_over_crit = θ > θ_crit;
4 ...
// Les coefficients contiennent beaucoup de données inutiles/erronées qui
   ne seront pas utilisées
5 F_under_crit = ...;
6 F_equal_crit = ...;
7 F_over_crit = ...;
// Les valeurs des coefficients sont calculées de manière systématique
8 ...
9 F= ( mask_under_crit & F_under_crit ) OR ( mask_equal_crit & F_equal_crit)
  | ( mask_over_crit & F_over_crit);
```

5.3.2 Étape 2 : Recherche de la taille des signaux

La recherche de la taille maximum des signaux en nombre d'échantillons étant une opération qui à la fois s'exécute rapidement et qui est limitée par les accès mémoire, son optimisation n'a pas été prioritaire : elle n'a pas été vectorisée.

5.3.3 Étape 3 : Traitement du signal

La dernière étape, permettant de passer d'un ensemble de pinceaux en un point à une information de champ, consiste à réaliser plusieurs traitements sur les signaux de réponses impulsionnelles et de déplacement. Chaque contribution de pinceau constitue une réponse impulsionnelle de déplacement élémentaire. Dans une première sous-étape, ces contributions sont sommées en prenant en compte les lois de retards de la configuration simulée afin de former les réponses impulsionnelles globales. Dans un second temps, la réponse impulsionnelle globale de déplacement est convoluée avant d'en extraire l'information de champ : un signal de déplacement, un maximum d'amplitude et le temps de vol associé.

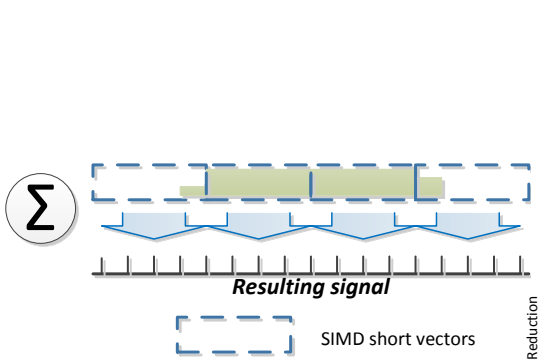
5.3.3.1 Étape 3.1 : Sommation des contributions de chaque pinceau

La première phase réalise la sommation des contributions de chaque pinceau sur les réponses impulsionnelles. Lors de cette étape, le temps de vol du pinceau, recalé par la loi de retards, est

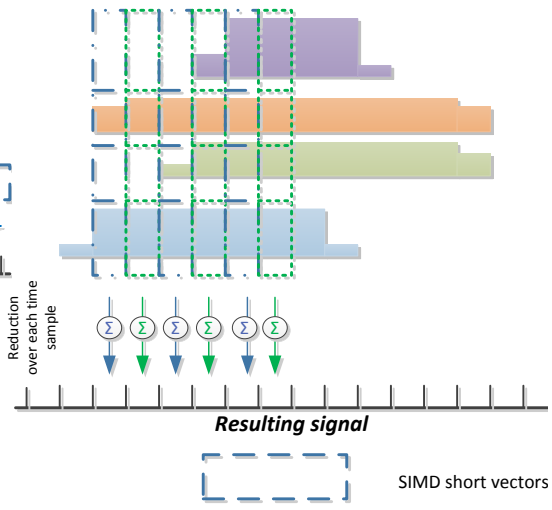
converti en un index parmi les échantillons temporels de la réponse impulsionnelle. L'étalement temporel du pinceau donne, autour de ce temps de vol, les bornes supérieures et inférieures des échantillons sur lesquels s'ajoute la contribution.

D'un pinceau à l'autre, même très proches géométriquement (même point source, points capteurs voisins), les échantillons concernés pour cette étape de sommation sont à la fois proches (temporellement) mais aussi différents (décalages sur les bornes). Deux approches régulières peuvent être étudiées afin de réaliser cette sommation au moyen d'instructions SIMD, cependant chacune comporte des inconvénients :

- Une sommation "horizontale" dans laquelle les registres SIMD sont "parallèles" aux signaux de réponse impulsionnelle. La contribution d'un pinceau est répartie dans plusieurs registres SIMD qui sont alors sommés sur la réponse impulsionnelle. Cette découpe est représentée par la figure 5.2a. Elle nécessite, pour être optimale, que les contributions des pinceaux soient alignées en mémoire (une fois converties en échantillons temporels). De plus, le traitement nécessaire des registres SIMD aux bornes est complexe (branchements pour placer les bornes).
- Une sommation "verticale" dans laquelle les contributions de plusieurs pinceaux sont calculées simultanément, remplissant les différents éléments de registres SIMD. Un registre SIMD de contribution correspond à un échantillon temporel donné et contient des zéros ou la valeur de la contribution à sommer. Une réduction permet d'accumuler, sur le registre SIMD, la valeur de la contribution à ajouter à la réponse impulsionnelle. Cette approche est illustrée par la figure 5.2b. Les pinceaux étant regroupés par proximité "d'ordre" (le plus souvent, d'éléments du capteur se suivant), cette sommation nécessite des contributions très focalisées en temps, le risque étant de devoir sommer beaucoup de zéros sur les signaux.



(a) Approche "horizontale"



(b) Approche "verticale"

FIGURE 5.2 – Deux *SIMDization* potentielles de la phase de sommation des contributions des pinceaux

Cependant, cette phase de sommation est grandement déséquilibrée en termes de calculs par rapport aux besoins en accès mémoire. Pour un pinceau, il faut lire en mémoire les informations nécessaires au calcul de sa contribution puis, sur un certain nombre d'échantillons temporels en fonction de son étalement. Sa contribution est ensuite ajoutée aux **six composantes** de la réponse impulsionnelle en déplacement (trois coordonnées, à valeurs complexes d'où 6 écritures mémoire).

L'étape de sommation des contributions des pinceaux est grandement dépendante des données. La manière dont les pinceaux vont ou non contribuer de manière cohérente aux signaux de déplacement : l'alignement des contributions, leur recouvrement entre-elles. Afin d'évaluer la pertinence de ces approches, il convient d'étudier la phase de sommation des contributions des pinceaux en détail.

5.3.3.1.1 Analyse topographique des pinceaux Dans un cadre métier, l'opérateur n'a pas une connaissance *a priori* du faisceau et souhaite déterminer ses caractéristiques. De manière générale, il est raisonnable de considérer la région d'intérêt, c'est à dire la zone de champ, plus large que le faisceau ultrasonore. En effet, si la zone se trouve exactement sur le faisceau, l'opérateur ne verra pas l'insonification par différence avec le reste de la pièce et élargira alors sa zone d'intérêt.

Afin de donner un aperçu du comportement des contributions des pinceaux, trois configurations parmi les configurations du *benchmark* de performance sont utilisées. Celles-ci comportent des caractéristiques communes : capteur multi-éléments en immersion, pièce en acier isotrope à surface plane, mode L direct, contrôle en L0 (à la verticale du capteur) et même zone de champ plus large que le faisceau ultrasonore émis. Leurs variations concernent les lois de retards utilisées et la focalisation appliquée dans la pièce :

- La configuration 19 présente une absence de focalisation.
- La configuration 20 focalise le faisceau en un point, sous le fond de la pièce.
- La configuration 21 focalise le faisceau en un point au sein de la zone de champ simulée.

La variation de la tache focale est représentée par la figure 5.3.

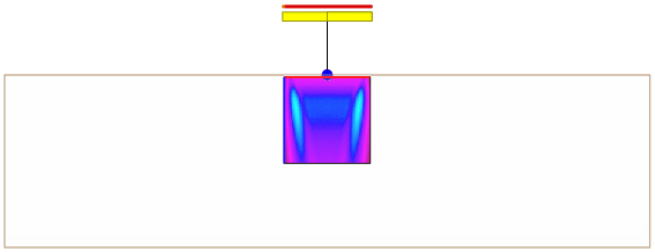
Sur ces configurations, deux mesures sont réalisées. L'une, concerne l'étalement moyen des pinceaux et est présentée à la figure 5.4. L'autre mesure représentée à la figure 5.5 montre le recouvrement temporel des contributions des pinceaux.

Sont également présentées les informations concernant la simulation globale d'une part, et la "tache focale" d'autre part. **Ici, pour une aisance d'implémentation, la "tache focale" n'est pas obtenue géométriquement mais concerne tous les points de champ d'amplitude à -3 décibels du maximum.**

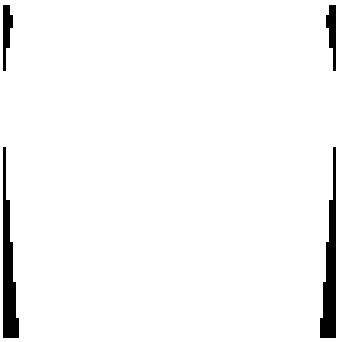
Sur la figure 5.4a d'une configuration à l'autre, seuls les retards changent (la géométrie de la pièce, le capteur et la zone du capteur restent inchangés). Les pinceaux restent donc identiques. Leur largeur varie ici de 2 à environ 28 échantillons temporels identiques.

La figure 5.4b montre que les pinceaux de plus forte amplitude ont un étalement temporel de plus en plus petit, plus la focalisation se rapproche du capteur. En focalisant de plus en plus près du capteur, la tache focale est plus petite et se concentre dans la zone d'intérêt.

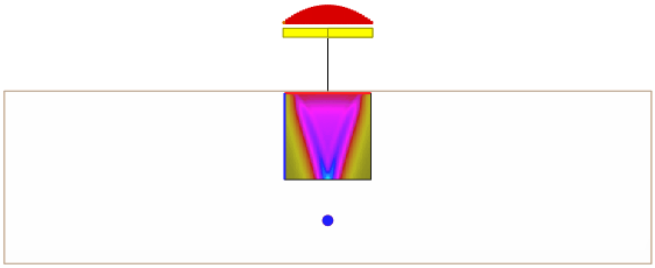
Les pixels constituant la tache focale sont dans l'axe du capteur, les pinceaux associés à la tache focale arrivent normalement sur la surface du capteur : l'étalement temporel de ceux qui contribuent à la tache focale est plus resserré au fur et à mesure que la tache focale se rapproche du capteur (mais l'ensemble des pinceaux est toujours le même sur les trois configurations).



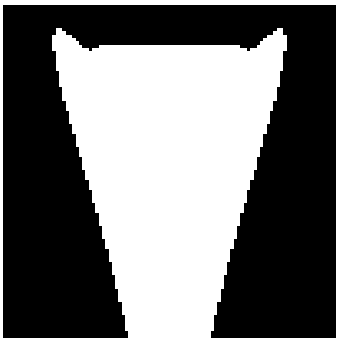
(a) Configuration 19 - Champ simulé (Image CIVA)



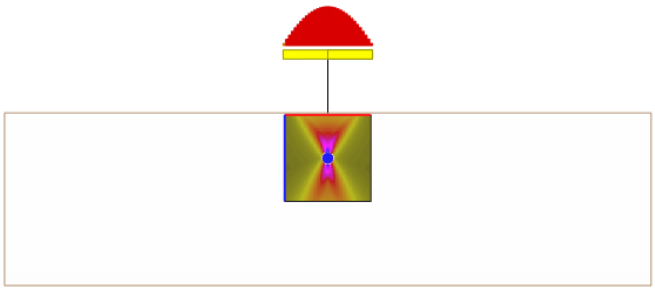
(b) Tache focale - configuration 19



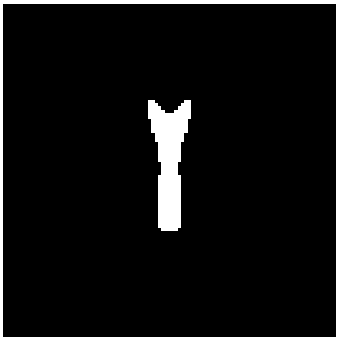
(c) Configuration 20 - Champ simulé (Image CIVA)



(d) Tache focale - configuration 20



(e) Configuration 21 - Champ simulé (Image CIVA)



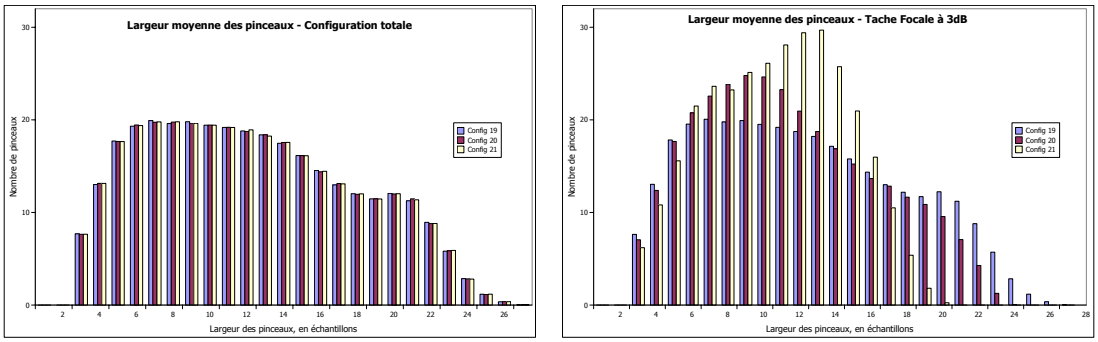
(f) Tache focale - configuration 21

FIGURE 5.3 – Évolution de la tache focale sur les configurations étudiées (en blanc)

Ce comportement, de raccourcissement de l'étalement temporels des pinceaux, ne serait pas aussi marqué avec une focalisation inclinée (les pinceaux se concentreraient sur la même zone du signal, mais seraient naturellement plus larges).

Cependant, ce raccourcissement n'est pas régulier : les pinceaux sont, dans leur ensemble, de taille très variable et peu régulière.

Concernant la sommation en elle même, les figures 5.5a et 5.5b présentent l'emplacement,



(a) Configuration totale (b) Tache focale à -3dB

FIGURE 5.4 – Largeur des pinceaux en nombre d'échantillons temporels

en termes d'échantillons temporels, sur lequel la sommation a lieu pour former les réponses impulsionnelles en déplacement. Elles permettent d'observer le recouvrement qui s'opère au fur et à mesure que la focalisation croit.

Sur les trois configurations étudiées, les signaux sont de taille 1024 échantillons. La figure 5.5a montre un recouvrement épars des contributions sur une moitié de signal (la taille du signal nécessaire est probablement inférieure à 1024 et a été étendue pour des facilités de calcul de FFT). La figure 5.5b précise qu'au niveau de la tache focale, plus la focalisation augmente et plus les sommations se concentrent en un faible nombre d'échantillons temporels. La focalisation permet aux différentes contributions des pinceaux d'arriver en phase. Le recouvrement temporel de ces pinceaux augmente donc notablement pour les pixels de la tache focale, mais ce phénomène est localisé : il n'a lieu que sur une partie de l'image de champ, et est beaucoup moins notable sur la globalité de l'image.

Lors de la sommation sur une même géométrie, en fonction de la focalisation, la phase de sommation parcourt toute une plage de possibilités.

- Lorsqu'il n'y a pas de focalisation la quantité de contributions par échantillons temporels est, pour toute l'image, à peu près similaire, mais la largeur des contributions des pinceaux est très variable. Une telle approche rend complexe la mise en œuvre d'une sommation "horizontale" car il faut prévoir les cas aux limites concernant les pinceaux non centrés sur la largeur d'un registre SIMD ainsi que les contributions débordant d'un multiple de la largeur d'un registre SIMD (alignement mémoire).
- Lorsqu'il y a focalisation, la répartition des contributions lors de la sommation reste globalement la même en moyenne (les pinceaux sont les mêmes). Par contre, la tache focale nécessite une attention particulière. L'étalement des contributions des pinceaux qui y contribuent est globalement plus court, mais elles se recouvrent beaucoup plus. La vectorisation, par une approche "verticale", est rendue difficile par la variabilité du nombre de contributions par échantillon temporel à un instant donné.

Ce qui précède confirme ainsi que le comportement de la phase de sommation est grandement dépendant des données d'entrée, c'est à dire de la configuration de contrôle que l'on souhaite étudier.

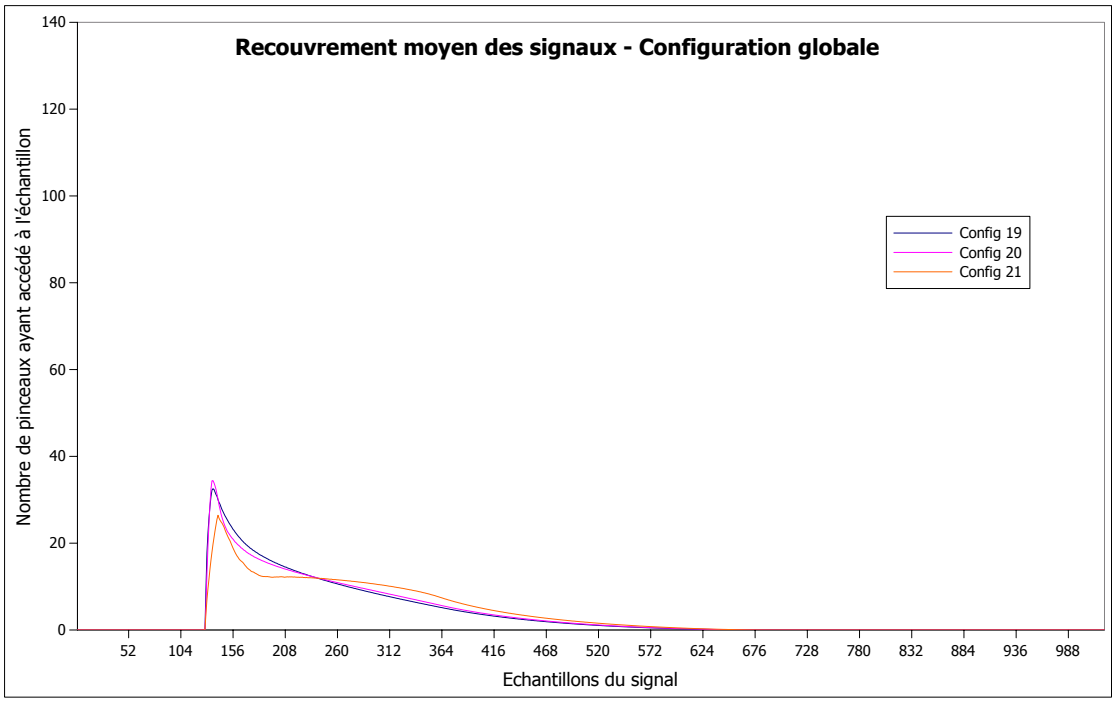
5.3.3.1.2 Version scalaire différentielle Afin de réduire le déséquilibre vis à vis des accès mémoire, une approche différentielle de l'étape de sommation a été étudiée. Dans cette approche,

le créneau de la contribution est transformé en un information "différentielle" : les bornes du créneau correspondent ainsi à une augmentation ou une diminution de l'amplitude du déplacement. Ces différences sont sommées non pas sur les échantillons temporels mais sur les bornes de ces derniers. Une fois toutes les contributions traitées, un parcours de ces bornes permet de reconstruire la réponse impulsionnelle de déplacement sommée. Elle est illustrée par la figure 5.6.

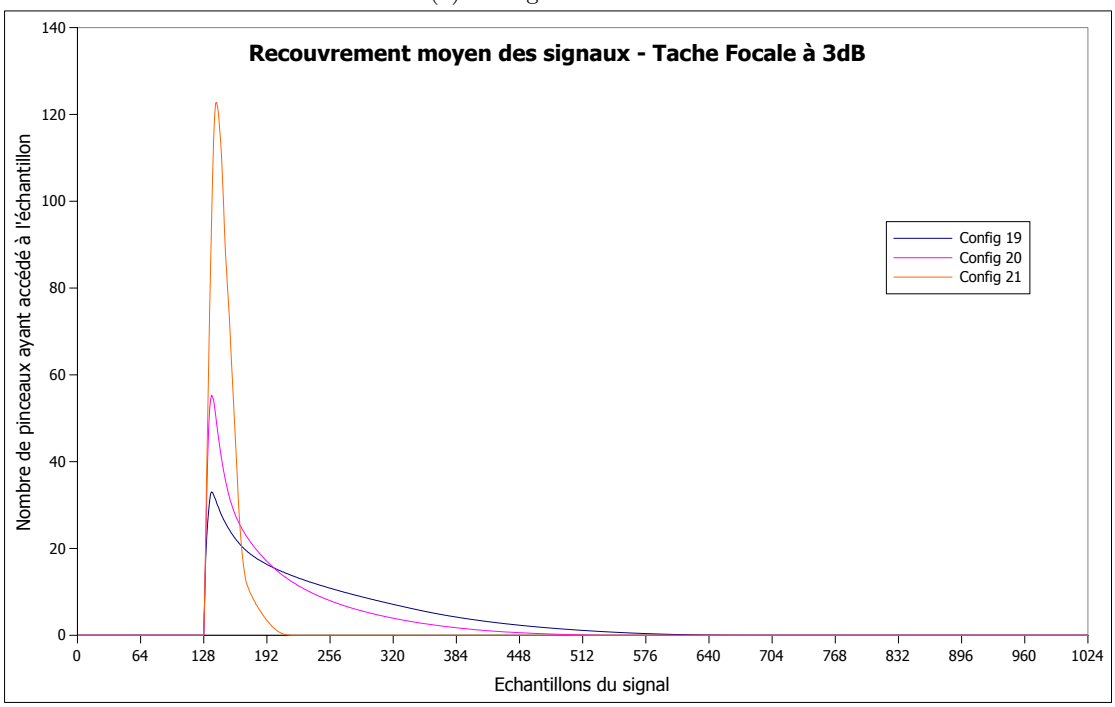
Cette méthode évite de réaliser un accès mémoire sur l'ensemble des échantillons concernés par un pinceau et ne nécessite que quatre accès, pour les bornes du créneau de contribution. Elle a été implémentée de manière scalaire.

5.3.3.2 Étape 3.2 : Extraction du maximum d'amplitude du signal de déplacement

Afin d'extraire le maximum d'amplitude du signal de déplacement, il faut transformer les six composantes de la réponse impulsionnelle en déplacement en un signal d'amplitude de champ résultat. Les signaux sur lesquels les réponses impulsionnelles sommées sont d'une taille puissance de 2 ; les registres SIMD permettent de traiter simultanément également un nombre de données flottantes multiple de 2. La taille des signaux étant un multiple de la largeur des registres SIMD, la vectorisation des opérations régulières sur les signaux, en dehors des calculs de FFT déjà vectorisés à l'aide de la bibliothèque MKL, se fait de manière évidente. Ainsi, le calcul de la norme de trois signaux est parfaitement régulier et vectorisé. De même le calcul de l'enveloppe et la convolution avec le signal de référence sont bien adaptés à la vectorisation en répétant régulièrement des opérations sur l'ensemble des signaux (respectivement, l'une réalisant un produit de deux nombres complexes sur toute la longueur du signal et l'autre réalisant deux opérations sur deux demi-signaux).



(a) Configuration totale



(b) Tache focale à 3dB

FIGURE 5.5 – Recouvrement moyen des signaux en un point, en échantillons. En abscisse le nombre de pinceaux venus contribuer à l'échantillon. En ordonnée, l'indice des échantillons du signal (pour ces configuration, tous les points de champ ont des réponses impulsionnelles de même taille, 1024 échantillons, seul varie le t0 permettant de décaler le premier échantillon dans le temps)

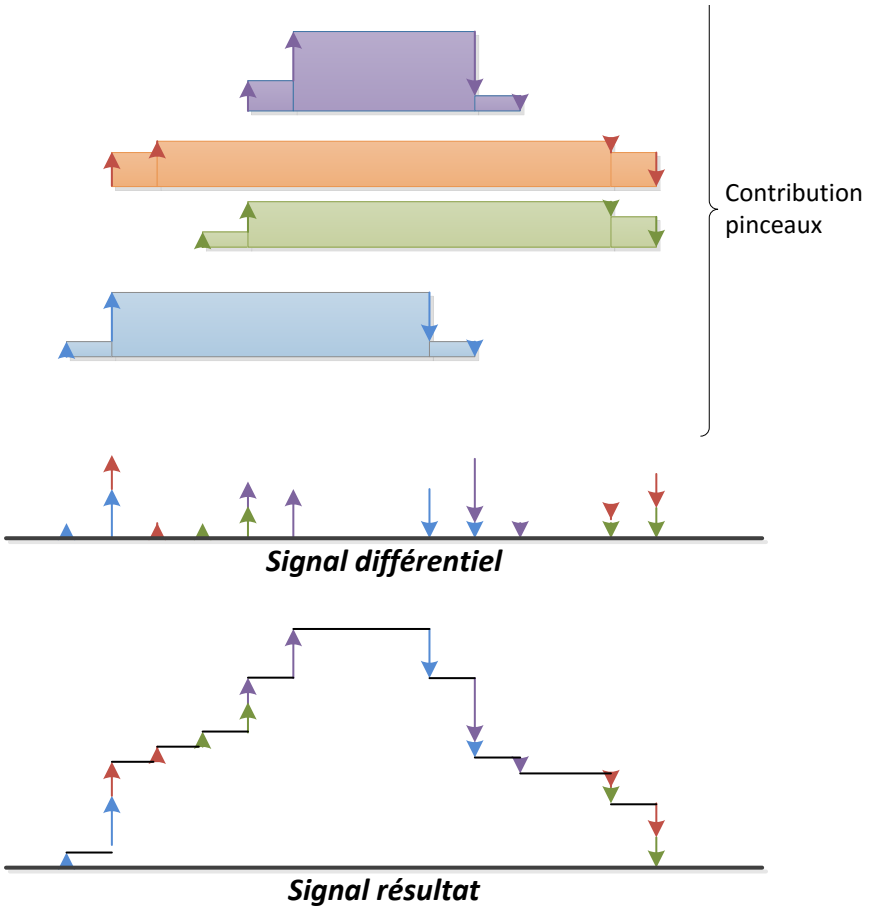


FIGURE 5.6 – Principe de la sommation différentielle de la contribution des pinceaux

5.4 Analyse du comportement des algorithmes optimisés sur GPP

L'analyse du comportement des algorithmes présentés pour le calcul de champ sur les différentes architectures matérielles testées permet de déduire l'adéquation des algorithmes par rapport aux architectures des différents GPP étudiés. Afin de pouvoir analyser les résultats globaux obtenus, les performances de chacune des sous-étapes du calcul de champ sont présentées en partant du noyau de calcul le plus unitaire possible pour remonter aux codes de calculs complets.

5.4.1 Étape 1 : Calcul des pinceaux

Le calcul des pinceaux émis par le capteur bénéficie d'une accélération par utilisation d'instructions SIMD, ses performances sont présentées sous-étape par sous-étape avant d'en faire la synthèse.

5.4.1.1 Étape 1.1 : Calcul de trajet

Le calcul du trajet se compose d'un noyau d'évaluation numérique, réalisant l'approximation de la racine polynômiale par la méthode de Newton et d'un ensemble de phases de validation du trajet (allant de la construction du polynôme aux validations géométriques).

5.4.1.1.1 Calcul du polynôme et méthode de Newton - Minibenchmark Le calcul du trajet fait appel à la résolution numérique par la méthode de Newton d'un polynôme de degré quatre. La méthode de Newton est rappelée par l'algorithme 7.

Le polynôme $P[X]$ est évalué, pour les différentes valeurs de x_n obtenues à chaque itération, par la méthode de Horner ce qui permet d'obtenir en un nombre réduit de calculs la valeur de $P[x_n]$ et $P'[x_n]$. Ce principe est illustré par l'équation 5.1.

$$\begin{aligned} P[X] &= a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4 \\ P[X] &= a_0 + X(a_1 + X(a_2 + X(a_3 + Xa_4))) \end{aligned} \tag{5.1}$$

Algorithme 12 : Évaluation de $P[X]$ et de $P'[X]$ par la méthode de Horner - P polynôme de degré 4

```
1 p = a_4;
2 p' = 0;
3 for i=3; i>=0; i– do
4 |   p' ← X × p' + p;
5 |   p ← X × p + a_i;
```

La probabilité que le nombre d'itérations de la méthode de Newton pour converger sur une racine soit le même pour deux polynômes est faible et impossible à déterminer *a priori*.

Cette étape du calcul de champ est très rapide par rapport au calcul des pinceaux, cependant, son périmètre est connu et restreint ce qui permet d'en faire une analyse fine. Le nombre d'instructions exécutées lors de la phase de résolution de polynôme de degré 4 dépend de deux quantités : le nombre d'itérations effectuées et le nombre de polynômes traités. On peut ainsi obtenir, par régression linéaire, les performances d'une itération de la méthode pour un polynôme. Afin de réaliser une mesure équitable de l'implémentation SIMD du calcul polynomial, une version dédiée de cette méthode de Newton a été produite, réalisant un nombre déterministe d'itérations qui ne prend pas en compte d'autres critères d'arrêt.

De manière relative, cette mesure permet de juger de l'accélération obtenue en comparant les performances scalaires avec les performances SIMD sur une machine donnée. De même elle permet de comparer entre elles plusieurs architectures GPP. En outre, cela permettra de valider l'usage de Boost.SIMD par rapport à une *SIMDization* manuelle. Le code obtenu est suffisamment concis pour permettre une étude absolue de ses performances.

Les GPP modernes sont très complexes et utilisent beaucoup d'astuces matérielles afin d'accélérer les performances d'une application (*c.f.* 3.1.2 pour un bref aperçu de ces capacités). Le fondeur des GPP étudiés, Intel, met les informations concernant les performances par architecture et par instruction en termes de latence et de débit à disposition des utilisateurs[2]. A partir de ces informations, on peut modéliser très grossièrement les GPP sur lesquels le code de Newton va s'exécuter. Cette modélisation ne prend bien évidement pas en compte les capacités les plus avancées des GPP. D'une part, en sommant les latences des instructions, il est possible d'obtenir une borne supérieure du temps d'exécution du code, comme si par exemple aucune instruction ne pouvait correctement bénéficier du *pipeline*. D'autre part, en sommant les débits (nombre de cycles nécessaires pour exécuter l'instruction), il est possible d'obtenir un temps minimum pour le calcul. Ainsi il est possible d'encadrer les performances mesurées par ces bornes théoriques et constater ou non de la bonne accélération.

Sur l'ensemble des architectures étudiées, les mesures réalisées sont présentées par la figure 5.7[3]. Pour chaque machine, pour chaque version exécutée les performances sont affichées en cycles dans la légende. Ces données sont obtenues pour 1, 10, 100 et 1000 itérations, mesurées sur des jeux de polynômes composés de 1000, 10000 et 100000 polynômes de degré 4 aux coefficients aléatoires[4]. Ces grandes valeurs permettent d'obtenir une valeur asymptotique du temps de calcul de l'itération et de ne pas risquer d'erreur de mesure sur les petites valeurs. Ces valeurs asymptotiques par architecture et par implémentation sont données en cycles par itérations par polynôme sur les courbes (CPIT dans la légende), ce qui correspond à la pente de la courbe. Pour illustrer l'absence de problème à utiliser des valeurs très importantes, pour Xeon Westmere, on présente les graphiques 5.7a et 5.7b, réalisés à partir du même jeu de données, le premier en utilisant toutes les mesures et le second n'utilisant que les mesures dans la plage $[0; 10]$ itérations. On y constate que les performances mesurées sont du même ordre, avec un écart inférieur à 5%, justifiant ainsi l'usage de grandes valeurs.

En plus de ces mesures, le décompte des instructions utilisées a aussi été effectué depuis le code assembleur généré. Ce décompte a ainsi permis de constater que sur un code aussi concis, Boost.SIMD et le code manuel génèrent les mêmes ensembles d'instructions. Les graphiques de la figure 5.7 montrent que, pour chaque architecture, les performances de Boost.SIMD et du code manuel sont du même ordre de grandeur, validant l'usage de Boost.SIMD. Le nombre d'instructions nécessaire par le calcul de Newton est présenté par le tableau 5.2. Pour obtenir les instructions d'une version il convient de regarder les chiffres d'une colonne.

Le compilateur utilisé ici est Intel C++ Compiler (ICC). Afin de réaliser les opérations sur des nombres flottants (sous forme de scalaires), ICC fait appel à des registres SIMD 128 bits, en n'utilisant qu'un élément du registre (1 sur 4 en simple précision, 1 sur 2 en double précision) plutôt que d'utiliser la FPU x87. Ainsi, une opération réalisée sur des scalaires et la même opéra-

[2]Dans ce chapitre, le terme débit est utilisé comme traduction du *Throughput* dans les données de *l'Intel Intrinsic Guide* présentant un nombre de cycles par instruction (unité de temps par quantité) plutôt que sa définition scientifique (quantité par unité de temps). Cela permet d'obtenir des informations cohérentes entre les cycles mesurés et les valeurs théoriques.

[3] Il convient de noter, eu égard aux options de compilation de Boost.SIMD, que le paramétrage de la compilation n'a pas permis de configurer la génération de code pour obtenir des instructions FMA 128 bits sur Xeon Haswell via Boost.SIMD.

[4]Toutes les mesures utilisent la même graine pour générer les coefficients à l'aide d'un algorithme de Mersenne Twister

tion réalisée sur des registres 128 bits ont les mêmes performances. De même le décompte scalaire est regroupé avec le décompte SSE pour ne pas alourdir le tableau : les instructions scalaires générées sont alors suffixées d'un `ss` pour *single simple (precision)* au lieu de `ps` pour *packed simple (precision)*. Les instructions préfixées d'un `v` correspondent à une instruction 256 bits. L'instruction `cmp` est utilisée pour la comparaison de l'itérateur avec le nombre maximum d'itérations dans la boucle.

Newton	128 bits - Scalaire/SSE 128 bits	256 bits / AVX	AVX2 et Utilisation FMA
divps	1		
vdivps		1	1
subps	1		
vsubps	1	1	
mulps	7		
vmulps		7	
addps	7		
vaddps		7	
fmaddps			8
cmp	1	1	1

TABLE 5.2 – Minibenchmark Newton - Décompte des instructions sur différentes architectures

Les tableaux 5.3 et 5.4 rappellent les performances de chaque instruction, pour les différentes architectures de GPP, sur les registres SIMD de taille respectivement 128 et 256bits.

Architecture	Instruction	Latence	Débit		Latence	Débit
Haswell	divps	13	5	subps	3	1
Ivy Bridge	divps	13	6	subps	3	1
Sandy Bridge	divps	14	14	subps	3	1
Westmere	divps	14	12	subps	3	1
Nehalem	divps	14	12	subps	3	1
Architecture	Instruction	Latence	Débit		Latence	Débit
Haswell	mulps	5	0.5	addps	3	1
Ivy Bridge	mulps	5	1	addps	3	1
Sandy Bridge	mulps	5	1	addps	3	1
Westmere	mulps	4	1	addps	3	1
Nehalem	mulps	4	1	addps	3	1
Architecture	Latence	Débit				
Haswell	1	0				
Ivy Bridge	1	0				
Sandy Bridge	1	0				
Westmere	1	0				
Nehalem	1	0				

TABLE 5.3 – Performances des instructions SIMD 128bits

Source : Intel Intrinsics Guide

Usage des instructions SIMD 128bits A partir du décompte d'instructions et de leurs performances par architecture, il a été possible de borner les performances mesurées par des

Architecture	Instruction	Latence	Débit		Latence	Débit
Haswell	vdivps	21	13	vsubps	3	1
Ivy Bridge	vdivps	21	14	vsubps	3	1
Sandy Bridge	vdivps	29	28	vsubps	3	1
Architecture	Instruction	Latence	Débit		Latence	Débit
Haswell	vmulps	5	0.5	vaddps	3	1
Ivy Bridge	vmulps	5	1	vaddps	3	1
Sandy Bridge	vmulps	5	1	vaddps	3	1

TABLE 5.4 – Caractéristiques des instructions SIMD 256 bits

Source : Intel Intrinsics Guide

limites théoriques définies en début de paragraphe. Les valeurs indiquées par Intel correspondent à un calcul SIMD sur tout la largeur du registre ; afin d'être cohérent avec ces données, il convient de multiplier la pente des courbes (ou cycles par itération par polynôme, CPIT) de la figure 5.7 par le cardinal du vecteur SIMD correspondant à l'implémentation (en nombre de flottants simples précision). Les performances obtenues sur instructions scalaires et instructions SIMD 128bits sont présentées par les tables 5.5 et 5.6 respectivement. On observe sur la table 5.6 de très bonnes accélérations par rapport au scalaire.

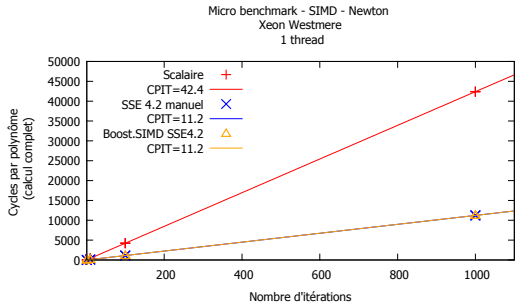
Scalaire	Temps max théorique $\sum$ latences	cycles mesurés par itération par polynôme (CPIT, code scalaire)	Temps min théorique $\sum$ débits
Haswell	75	41.9	17.5
Ivy Bridge	75	47.1	22
Sandy Bridge	76	47.5	30
Westmere	69	42.4	28

TABLE 5.5 – Performances mesurées en cycles de Newton sur instructions scalaires (code manuel)

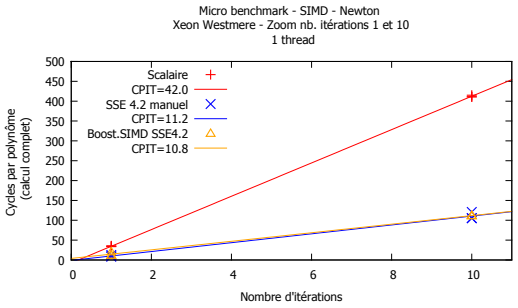
128 bits	Temps max théorique		Mesure		Temps min théorique	
	$\sum$ latences	Gain	CPIT pour un registre SIMD	Gain	$\sum$ débits	Gain
Haswell	75	×4.0	42.9	×3.9	17.5	×4.0
Ivy Bridge	75	×4.0	48.1	×3.9	22	×4.0
Sandy Bridge	76	×4.0	49.0	×3.9	30	×4.0
Westmere	69	×4.0	44.9	×3.8	28	×4.0

TABLE 5.6 – Performances mesurées en cycles de Newton sur instructions 128 bits (version SIMD manuelle) et gains par rapport aux codes scalaires.. Pour rester cohérent avec les chiffres théoriques (par registre SIMD), les temps en CPIT mesurés sont ramenés au cardinal du SIMD.

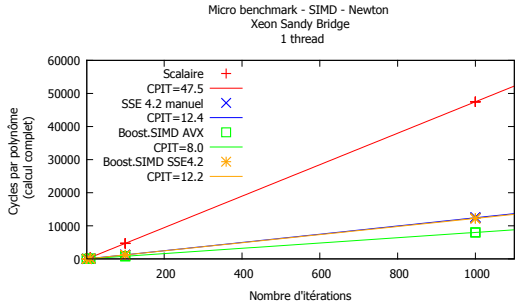
Usage des instructions SIMD 256bits Cette même approche est reprise à la table 5.7 afin d'observer le comportement sur des instructions SIMD de largeur 256 bits. Dans un premier temps, les codes mesurés ne font pas appel aux instructions FMA. Tout d'abord, on constate que les accélérations théoriques ne sont plus aussi lisses qu'entre scalaire et SSE (ce n'est plus le même registre occupé par 1 ou plusieurs éléments). Même au niveau des latences et débits, le gain n'est plus de ×8. En particulier concernant les débits, les gains maximums ne sont plus que de l'ordre de ×5. En comparant les performances des instructions de division `div` et `vdiv`, on



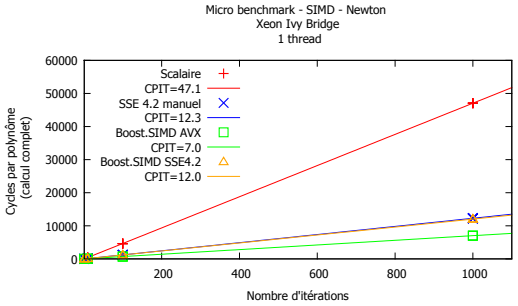
(a) Xeon Westmere



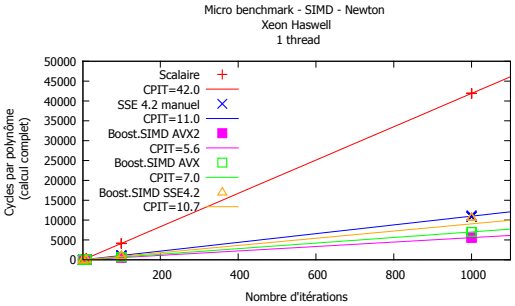
(b) Xeon Westmere - Zoom pour nb. iter 1 et 10



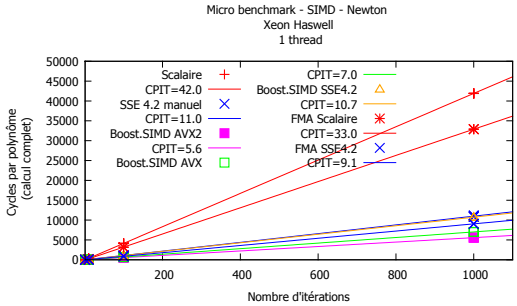
(c) Xeon Sandy Bridge



(d) Xeon Ivy Bridge



(e) Xeon Haswell



(f) Xeon Haswell

FIGURE 5.7 – Minibenchmark - Approximation de la racine d'un polynôme par la méthode de Newton

observe une chute du débit par élément de l'ordre de 50% sur l'ensemble des architectures (une division sur un vecteur deux fois plus long est presque deux fois plus lente en termes de débits). Les performances mesurées présentent ici une accélération supérieure au gain obtenu au niveau des débits (borne supérieure des performances).

Instructions FMA - architecture Haswell Il reste à analyser les bénéfices sur l'architecture Haswell qui propose désormais des instructions FMA. Tous les jeux d'instructions (scalaire, SIMD 128 bits et SIMD 256 bits) peuvent tirer parti des instructions FMA disponibles. Les versions scalaires et SIMD ont toutes les mêmes performances en termes de débit/latence par

256 bits	Temps max théorique		Mesure		Temps min théorique	
	$\sum$ latences (cycles)	Gain	CPIT pour un registre SIMD	Gain	$\sum$ débits (cycles)	Gain
Haswell	83	×7.2	56.2	×6.0	25.5	×5.5
Ivy Bridge	83	×7.2	56.2	×6.7	30	×5.9
Sandy Bridge	91	×6.7	64.0	×5.9	44	×5.5

TABLE 5.7 – Performances mesurées en cycles (Boost.SIMD sans FMA) de Newton sur instructions 256 bits et gains par rapport aux codes scalaires. Pour rester cohérent avec les chiffres théoriques (par registre SIMD), les temps en CPIT mesurés sont ramenés au cardinal du SIMD.

vecteur, rappelés par la table 5.8. L'usage des instructions FMA peut être décidé automatiquement par le compilateur lorsqu'il détecte la bonne succession d'opérations, à condition que les options de compilation le lui permettent (une opération FMA fait subir aux opérandes seul un arrondi, contrairement à la succession multiplication addition qui leur fait subir deux arrondis).

Architecture	Instruction(s)	Latence	Débit
Haswell	(v)fmaddps	5	0.5
Correspondance	(v)add + (v)mul	8	1.5

TABLE 5.8 – Caractéristiques des instructions FMA et la correspondance avec les instructions SIMD classiques. Les performances sont les mêmes pour les instructions SIMD 128 bits et 256 bits.

Source : Intel Intrinsics Guide

Le tableau 5.9 montre les performances avec et sans FMA sur chaque jeu d'instructions. A côté des mesures, en ligne, est présentée l'accélération obtenue par rapport à la version scalaire. Sous les versions avec et sans FMA se trouve le gain dû aux instructions FMA pour le jeu d'instructions considéré. On constate tout d'abord que les instructions FMA suivent la même tendance que les versions sans FMA : les gains obtenus entre FMA SIMD et FMA scalaire sont très proches. Sur l'ensemble des tailles de registres SIMD testés, les instructions FMA offrent des gains entre ×1.2 à ×1.3, indiquant un gain certain à leur usage (à un arrondi près sur les opérandes).

Haswell	Cardinal	Latence	Gain	Pente×Cardinal	Gain	Débit	Gain
Scalaire	1	75		41.9		17.5	
Scalaire FMA	1	59		32.9		11	
Gain FMA		×1.3		×1.3		×1.3	
128 bits	4	75	×4.0	42.9	×3.9	17.5	×4.0
128 bits FMA	4	59	×4.0	36.3	×3.6	11	×4.0
Gain FMA		×1.3		×1.2		×1.6	
256 bits	8	83	×7.2	56.2	×6.0	25.5	×5.5
256 bits FMA	8	67	×7.0	44.7	×5.9	19	×4.6
Gain FMA		×1.2		×1.3		×1.3	

TABLE 5.9 – Impact des instructions FMA sur architecture Haswell (codes scalaires manuels, code 256 bits Boost.SIMD), gains mesurés par rapport au code scalaire sans FMA

Conclusions sur le minibenchark Ce *benchmark* a été possible sur le code concis de la recherche de racine par la méthode de Newton sur un polynôme réel de degré 4. Il a été possible de montrer à la fois la bonne adéquation des implémentations et la bonne qualité du code obtenu au moyen de Boost.SIMD. Ce type de comparaison avec les caractéristiques théoriques des instructions est pertinent sur un *minibenchmark* aux dimensions réduites, cependant il n'est pas applicable à un code complet. D'une part, plus le code est long en termes d'instructions et plus à la fois le compilateur et le GPP ont la place pour optimiser l'ordre des instructions exécutées rendant encore plus approximatif le décompte. A titre d'illustration, la figure 3.4 présente la configuration des unités d'exécution sur architecture Haswell : suivant le type instructions à exécuter, il y aura plus ou moins de contention au niveau des ports du répartiteur du GPP. De plus, le décompte des instructions ainsi que l'ordre dans lesquelles elles sont exécutées restent très difficiles à obtenir, les GPP mettant en place des stratégies pour masquer les latences. Ces conditions rendent hasardeuse la prédiction des performances théoriques. Enfin, cette approche est extrêmement chronophage à mettre en place. Il n'a ainsi pas été possible d'obtenir une mesure fiable et automatique des instructions générées malgré des tentatives utilisant des outils tels que *Intel Software Development Emulator*. Par ailleurs, cette approche ne prend pas en compte un élément crucial des performances d'un code final : la performance des accès mémoire (externe) ne dépendant pas que du code mais également du comportement des caches.

5.4.1.1.2 Construction et validation des trajets

Les transformations et validations géométriques dépendent grandement des configurations des contrôles simulés. Il n'a pas été possible d'analyser de manière autonome cette phase du calcul des trajets.

5.4.1.2 Étape 1.2 : Caractéristiques des pinceaux

De même, cette étape est dépendante des résultats du calcul de trajet et de la validité d'un pinceau et dépend totalement des données d'entrée. Il n'a pas été possible d'analyser de manière individuelle cette phase du calcul des trajets, un analyse globale de l'étape 1 doit être réalisée.

5.4.1.3 Remarque d'implémentation

Il convient de noter que par la suite de ce chapitre, les mesures de certaines configurations de champ ne figurent pas dans les résultats présentés. Il ne s'agit pas d'omission, mais d'une absence de résultat en raison de l'implémentation retenue pour le maquettage du calcul de champ. Le code maquette mesure les performances de toutes les variantes algorithmiques du calcul de champ en vue d'obtenir les performances maximales sur chaque architecture, en particulier l'algorithme horizontal. Cet algorithme travaille par étape, et nécessite à un instant du calcul de tenir en mémoire toutes les données temporaires, en particulier à l'étape 3 où doivent résider en mémoire l'ensemble des pinceaux et les signaux de réponse impulsionnelle. Sur les GPP les plus modestes en mémoire et les configurations de champ les plus complexes, il n'est pas possible de conserver autant d'informations à la fois. Il s'agit des configurations 17 et 18, qui ne s'exécutent pas sur les GPP modestes en termes mémoire (Xeon Sandy Bridge, Xeon Ivy Bridge et Xeon Haswell).

L'absence de ces résultats n'a pas eu d'impact négatif sur les analyses qui vont être proposées dans la suite de ce chapitre : les autres configurations sont suffisantes pour obtenir une analyse algorithmique et les performances attendues sur ces configurations très complexes sont très loin de l'objectif d'interactivité. Il s'agit d'une problématique purement d'implémentation de ces travaux scientifiques et des solutions pour la résoudre sont proposées en conclusion de ce document, en vue d'une industrialisation du code de simulation de champ rapide (*c.f.* 8.1.1).

5.4.1.4 Étape 1 : résultats globaux

Sur les configurations de champ étudiées, plusieurs mesures ont été réalisées. Sur différentes machines, à l'aide d'une **algorithmie horizontale**, les performances de l'étape 1 du calcul de champ ont été mesurées en version scalaire et en version *SIMDizée*.

A aussi été mesurées, pour l'ensemble des configurations de champ, le nombre d'itérations nécessaires au calcul de trajet. A partir de la largeur des registres SIMD, les divergences dues aux données ont été calculées.

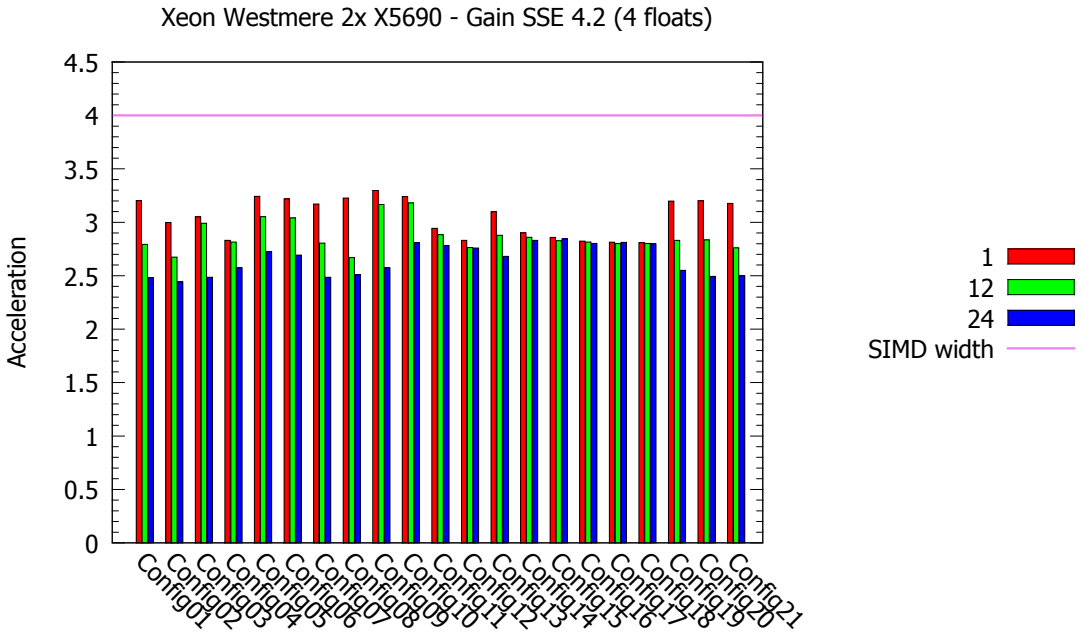


FIGURE 5.8 – Etape 1 : SIMD SSE 4.2 sur Xeon Westmere. Les histogrammes représentent les accélérations mesurées, par configuration par nombre de *threads*. En violet est l'accélération théorique maximale obtenue par le cardinal du vecteur SIMD. La courbe noire est le nombre de surface par configuration, et la courbe orange, la régularité théorique du calcul de Newton, ramenée au cardinal du vecteur SIMD considéré.

5.4.1.4.1 Instructions SIMD SSE 4.2 - 128 bits simple précision On peut observer à la lecture du graphique 5.8 les variations du gain obtenu en utilisant les instructions SSE 4.2 (4 flottants simple précision par registre) sur l'ensemble des configurations de champ, sur l'architecture Xeon Westmere. L'accélération obtenue entre la version scalaire et SIMD est significative. Ainsi, l'accélération moyenne obtenue sur un *thread* est de $\times 3.1 \pm 0.2$ et l'efficacité de cette *SIMDization* est décroissante plus le nombre de *threads* augmente : $\times 2.9 \pm 0.1$ sur 12 *threads* et $\times 2.6 \pm 0.1$ sur 24 *threads* (avec de l'*hyperthreading* activé). Les gains les plus faibles de l'*hyperthreading* sont sur les configurations 15 à 18, faisant appel à un grand nombre de surfaces.

Afin d'étudier plus en détail ces variations, la figure 5.9 montre les accélérations sur différentes architectures, sur un seul *thread* de calcul. Elles ont la même allure sur toutes les architectures. Sur cette figure apparait également le niveau de régularité théorique (au niveau de la divergence de Newton sur le nombre d'itérations nécessaires pour obtenir une racine, en orange), pour

	Nb cœurs	1	N	2×N (HT)
Xeon Westmere (2 GPP)	N=2×6	×3.1±0.2	×2.9±0.1	×2.6±0.1
Xeon Sandy Bridge	N=4	×3.2±0.2	×3.2±0.1	×3.2±0.1
Xeon Ivy Bridge	N=6	×3.1±0.2	×3.0±0.2	×3.0±0.1
Xeon Haswell	N=4	×3.1±0.2	×3.0±0.2	×3.0±0.1

TABLE 5.10 – Étape 1 : Moyenne et écart type de l'accélération sur le calcul des pinceaux au moyen des instructions SSE4.2

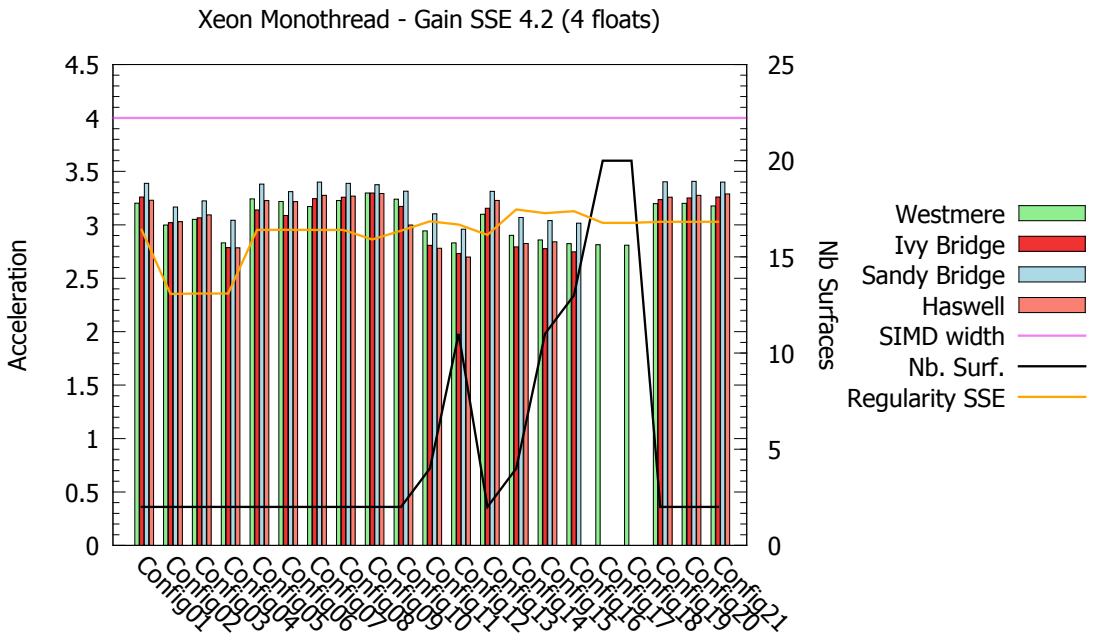


FIGURE 5.9 – Étape 1 : SIMD SSE 4.2 sur 1 *thread* pour différentes architectures

chaque configuration, par rapport à la largeur des registres SSE 4.2. Le nombre de surfaces utiles au calcul de champ figure également sur le graphique, en noir.

De légers décrochements en performance peuvent être observés pour quelques configurations. Les configurations 02, 03 et 04 voient leur performances chuter alors qu'elles correspondent à une augmentation, à échantillonnage surfacique et à taille du capteur constants, d'éléments actifs du capteur. L'augmentation de ces pinceaux à tracer entre les éléments du capteur et les points de champ perturbe la régularité du calcul de Newton, cause probable du décrochement des performances.

Les configurations 11, 12 et de 14 à 18 constituent un autre ensemble de configurations dont les performances sont moins bonnes. Ces baisses de performance n'ont pas lieu sur des configurations qui, à partir d'une même géométrie, simulent plusieurs modes de champ (entre les configurations 1 et 13, ou entre les configurations 1, 9 et 10). Les pinceaux de ces configurations 11, 12 et 14 à 18 sont réguliers (par rapport à la largeur des vecteurs SSE) : les pinceaux sont calculés par surface et par mode. Les configurations incriminées ont la particularité d'être composées de plusieurs surfaces d'entrée/fond par rapport aux modes simulés. Ainsi, il est possible de conjecturer que les transformations géométriques permettant de valider le trajet du pinceau obtenu nécessitent à la

fois des transformations requérant des opérations coûteuses en latence et des accès mémoire assez importants (chargement de la structure "surface", matrices de changement de repères, latence telle racine carrée...).

La table 5.10 présente une synthèse de ces performances sous la forme d'une moyenne et de l'écart type associé, y figurent également les performances pour un plus grand nombre de *threads*. L'étape de calcul des pinceaux est très complexe en faisant intervenir plusieurs types d'algorithmes les uns à la suite des autres afin d'obtenir les contributions de chaque pinceau, il n'a pas été possible d'aller au delà de cette corrélation pour montrer un lien de cause à effet.

5.4.1.4.2 Instructions SIMD AVX/AVX2 - 256 bits simple précision Sur les architectures Sandy Bridge, Ivy Bridge et Haswell, des instructions SIMD de largeur 256 bits (8 flottants simple précision par registre) sont supportées. Il s'agit des instructions AVX et des instructions AVX2 sur architecture Haswell. Cibler l'architecture Haswell permet de tirer parti des instructions FMA disponibles.

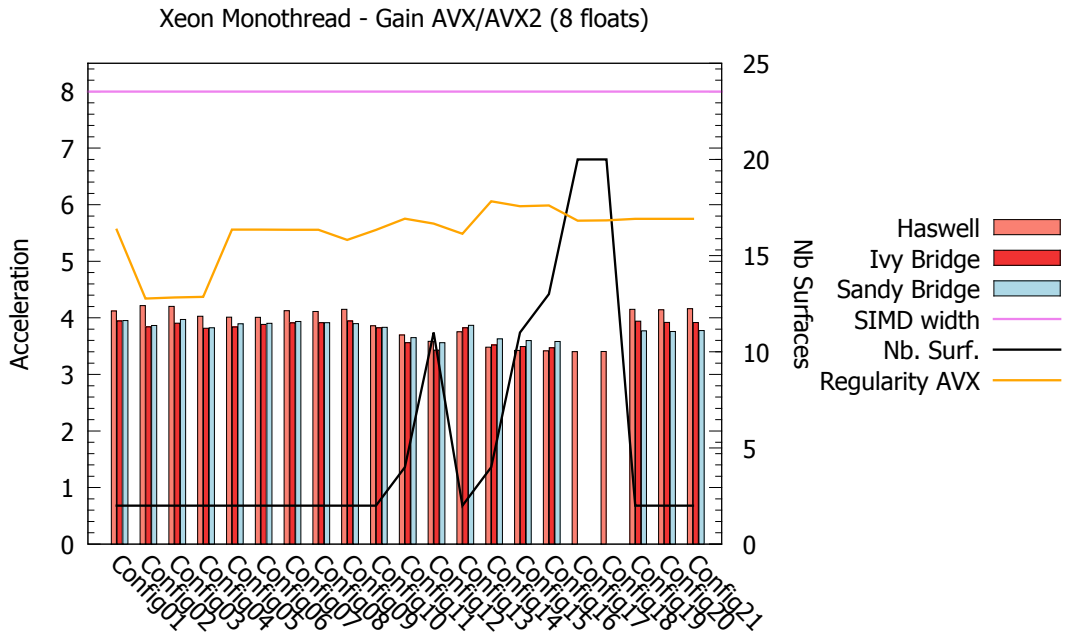


FIGURE 5.10 – Étape 1 : SIMD AVX (256 bits) sur 1 *thread* pour différentes architectures

La figure 5.10 reprend les configurations étudiées dans le cas de l'utilisation d'instructions SIMD 256 bits. On y observe toujours une courbe représentant la régularité des pinceaux calculés par rapport au cardinal du registre SIMD (en orange) et une courbe affichant le nombre de surfaces (en noir). Elle permet d'obtenir des accélérations (autour d'un facteur $\times 4$), supérieures à celles obtenues avec des instructions SIMD 128 bits. En termes d'allure générale, on observe des performances moins chahutées que sur les résultats 128 bits (*c.f.* figure 5.9).

	Nb cœurs	1	N	2×N (HT)
Xeon Sandy Bridge	N=4	×3.8±0.1	×3.8±0.1	×3.6±0.1
Xeon 2×Ivy Bridge	N=24	×3.7±0.2	×3.4±0.1	×3.2±0.1
Xeon Ivy Bridge	N=6	×3.8±0.2	×3.7±0.2	×3.6±0.1
Xeon Haswell	N=4	×3.9±0.3	×3.8±0.3	×4.0±0.1

TABLE 5.11 – Étape 1 : Moyenne et variance de l'accélération AVX (et FMA sur Haswell)

Les performances globales de l'étape 1 sur instructions SIMD 256 bits sont récapitulées par le tableau 5.11. On y constate une bonne accélération SIMD, entre ×3.7 et ×3.9 en *mono-thread*. Celle-ci est, de prime abord, surprenante lorsque l'accélération théorique due à la largeur du registre SIMD est de ×8. En examinant plus avant les instructions utilisées par le calcul des pinceaux, on constate l'utilisation de divisions et de racines carrées en plus des opérations arithmétiques "classiques" (sommer, soustraire, multiplier).

On observe sur le tableau 5.13 les performances 256 bits de la racine carrée, telles que fournies par le fondeur Intel. Ce tableau est à comparer avec le tableau 5.12 qui présente cette même instruction sur registre 128 bits. Les instructions 256 bits nécessitent des latences du même ordre de grandeur là où le débit de ces instructions est doublé. Ainsi, il est possible de supposer que le fondeur réutilise les mêmes unités de calcul entre 256 et 128 bits, les registres deux fois plus larges ayant besoin de deux fois plus de temps pour le traverser. On retrouve ici, de manière encore plus marquée, ce qu'on peut observer sur les opérations de division (tables 5.3 et 5.4) Ainsi, le potentiel d'accélération maximum est réduit et dépend de l'enchaînement des opérations nécessaires au calcul de champ. Sur cette phase de calcul de pinceaux, il y a à la fois des opérations géométriques (validation des trajets, normes de vecteurs...) et des calcul des coefficients nombreux. La performance obtenue parait acceptable.

Architecture	Instruction	Latence	Débit
Haswell	sqrt	18	7
Ivy Bridge	sqrt	20	7
Sandy Bridge	sqrt	22	14
Westmere	sqrt	25	16
Nehalem	sqrt	25	16

TABLE 5.12 – Caractéristiques des instructions SIMD 128 bits `sqrt`

Source : Intel Intrinsics Guide

Architecture	Instruction	Latence	Débit
Haswell	sqrt	21	14
Ivy Bridge	sqrt	21	14
Sandy Bridge	sqrt	29	28

TABLE 5.13 – Caractéristiques des instructions SIMD 256 bits `sqrt`

Source : Intel Intrinsics Guide

Par contre, une fois le code *multithread* activé, les performances décroissent significativement sur le Xeon 2×Ivy Bridge. Dans la mesure où ce phénomène n'est pas aussi marqué avec le mono-GPP Xeon Ivy Bridge, une explication possible est une contention mémoire pour le calcul

des pinceaux. Le 2×Ivy Bridge double le nombre de cœurs et donc de threads alors que les deux Xeon Ivy Bridge étudiés disposent d'autant de canaux mémoire par GPP (et donc de la même bande passante).

5.4.2 Étape 2 : Recherche de la taille des signaux

Cette étape comme indiqué au du paragraphe 5.3.2, du fait de son faible coût relatif n'a pas été *SIMDizée*.

5.4.3 Étape 3 : Traitement du signal

La dernière phase du calcul de champ se compose de deux familles d'algorithmes qui se succèdent : la sommation des contributions des pinceaux d'une part, et d'autre part, une phase de recherche du maximum d'amplitude, obtenu au moyen d'opérations de traitement du signal à partir d'outils standards (FFT, convolution, recherche de maximum, algèbre linéaire...).

5.4.3.1 Étape 3.1 : Sommation des contributions des pinceaux

Ce paragraphe va traiter de la mesure des performances des différentes implémentations des différentes stratégies de sommation proposées à la section 5.3.3.1. L'étape de sommation des contributions des pinceaux est grandement dépendante des données, au sens où elle dépend de la manière dont les pinceaux se font concurrence ou non lorsqu'ils accèdent en mémoire au même échantillon temporel. L'impact théorique de ces différentes focalisations des pinceaux a été mis en lumière 5.3.3.1.1 sur un jeu de trois configurations dont la géométrie reste constante, seule la focalisation électronique des données évoluant. Elle met en lumière le besoin de considérer l'impact des données sur la stratégie adoptée. Pour rappel, sur cette étape 3 de sommation des contributions des pinceaux sur la réponse impulsionnelle de déplacement, quatre versions du calcul sont mesurées.

- Sommation scalaire : code de référence ;
- Version scalaire différentielle : qui tente de minimiser pour chaque pinceau les accès mémoire ;
- Version SIMD horizontale : qui utilise des registres SIMD pour ajouter plus rapidement les contributions d'un seul pinceau à la réponse impulsionnelle ;
- Version SIMD verticale : qui regroupe les pinceaux par paquets correspondant (d'un nombre de pinceaux correspondant au cardinal d'un registre SIMD), afin d'évaluer plusieurs contributions pour un même pas de temps de la réponse impulsionnelle.

Lorsque les informations du pinceau sont chargées en mémoire, avant de calculer les bornes de sa contribution à la réponse impulsionnelle de déplacement un test est réalisé afin de déterminer si le pinceau contribue, ou s'il doit être éliminé (la boucle scalaire passe alors au pinceau suivant, la boucle vectorielle ne saute le paquet de pinceaux que s'ils doivent être tous éliminés). Il peut y avoir plusieurs raisons à cela : lors de l'étape 1.2, le pinceau correspondant au quadruplet (trajet, mode, point de champ, point capteur) a été marqué comme étant invalide (souvent pour des raisons géométriques), ou bien lors de l'évaluation des lois de retards le pinceau se voit associer un retard en amplitude électronique nul (dépendant du séquençage du capteur multi-éléments)[5]. La table 5.14 présente le pourcentage de pinceaux qui contribuent effectivement

[5]Ne se produit que sur les configurations 2 à 5 de l'étude. Un simple test permettra de réduire ces calculs surnuméraires lors de l'industrialisation

Configuration	Pourcentage	Nombre total de pinceaux
Config01	100%	3.3×10^6
Config02	100%	3.3×10^6
Config03	100%	3.3×10^6
Config04	100%	1.3×10^7
Config05	100%	1.3×10^7
Config06	100%	5.1×10^7
Config07	100%	3.3×10^6
Config08	100%	3.3×10^6
Config09	100%	6.5×10^6
Config10	100%	1.3×10^7
Config11	55%	9.8×10^6
Config12	12%	3.3×10^7
Config13	100%	3.3×10^6
Config14	37%	9.8×10^6
Config15	10%	3.3×10^7
Config16	3%	9.8×10^7
Config17	1%	3.3×10^8
Config18	2%	3.6×10^8
Config19	100%	3.3×10^6
Config20	100%	3.3×10^6
Config21	100%	3.3×10^6

TABLE 5.14 – Étape 3 : Pourcentages des pinceaux valides

au champ pour les configurations étudiées. On observe que les configurations avec une forte complexité géométrique, 11 et 12, ainsi que 14 à 18, présentent un fort taux de pinceaux invalides.

Les quatre stratégies de sommation ont été mesurées sur plusieurs architectures de GPP présentant des jeux d'instructions différents. Leurs performances ont été mesurées sur l'ensemble des configurations étudiées. Le corps de texte commente ces résultats dans leur globalité. L'ensemble des résultats est présenté à l'annexe D par les tables suivantes :

- sur Xeon Westmere 2×12 cores (SSE4.2) à la table 3,
- sur Xeon Sandy Bridge 4 cores (AVX) à la table 4,
- sur Xeon Ivy Bridge 6 core (AVX) à la table 5,
- sur Xeon Haswell 4 core (AVX2) à la table 6.

Une première constatation générale est que, quelle que soit la méthode de sommation étudiée, pour l'ensemble des GPP la variation du nombre de *threads* (1, N cœurs et 2×N *threads* pour activer l'*hyperthreading*) n'influe pas l'accélération mesuré par rapport à la sommation scalaire de référence de manière significative. On peut noter, pour les configurations avec beaucoup de pinceaux rejetés, une légère décroissance de rendement de la sommation SIMD verticale (configurations 15 à 18), mais les ordres de grandeur ne sont pas affectés.

5.4.3.1.1 Version différentielle La version utilisant une approche différentielle de la sommation des contributions des pinceaux est très souvent la plus rapide, sur l'ensemble des architectures de GPP étudiées. En revanche, sur des configurations ne présentant pas de pinceaux à éliminer, ses performances sont bonnes avec des gains obtenus supérieurs à ×2 et avoisinant

×2.4 sur l'ensemble des GPP. Sur les configurations présentant beaucoup de pinceaux invalides, elle n'améliore la sommation des contributions que des pinceaux valides : sa capacité à sauter les pinceaux invalides est la même que celle de la version de référence scalaire. Le gain mesuré n'est que de l'ordre de ×2.0 à ×2.1 sur les configurations 16 à 18 (pour l'ensemble des GPP) contre ×2.4 à ×2.8 sur les configurations bien conditionnées.

5.4.3.1.2 Approches SIMD horizontale et verticale Les techniques à base de *SIMDization* sont optimales dans des cas ultra-réguliers :

- La sommation horizontale est optimale lorsque tous les pinceaux ont une taille multiple de la largeur du vecteur SIMD et que leur temps de vol permet de les aligner en mémoire avec le signal ;
- La sommation verticale est optimale lorsque tous les pinceaux ont la même taille et contribuent au même échantillon (pour minimiser les emplacements vides dans les registres SIMD).

La version horizontale, traitant les pinceaux un par un, se comporte de la même façon que la version scalaire pour éliminer les pinceaux rejetés. Ses performances restent supérieures à la sommation scalaire.

Sur des configurations avec peu de pinceaux valides, la version verticale s'en tire beaucoup mieux et ses performances ressortent. Comme les pinceaux sont traités en même temps, lorsqu'ils sont tous rejetés alors le gain est maximal. Sur les configurations 2 à 4 dont la géométrie de la pièce est identique mais pour laquelle le nombre de pinceaux contribuant par point de champ augmente, on aurait pu s'attendre à un gain substantiel des versions SIMD du fait que les éléments du capteur sont activés au "milieu" des éléments déjà actifs. Ainsi la probabilité de collision pour l'accès aux réponses impulsionnelles est plus forte et les registres SIMD peuvent être mieux utilisés en étant plus "remplis" de contributions. Par contre, sur les configurations 7 et 8 pour lesquelles l'échantillonnage temporel est plus élevé, il n'y a pas d'effet significatif dû à la collision des contributions des pinceaux.

De même, sur les configurations 19 à 21, pour lesquelles la focalisation électronique des éléments augmente le taux de collisions en mémoire des pinceaux de la tache focale, il n'y a pas d'effet remarquable. Ce qui devait être un cas favorable pour la version SIMD verticale ne lui permet pas *in fine* d'accélérer les traitements en profitant de la grande quantité de pinceaux qui se recouvrent comme l'on pouvait l'espérer après l'analyse topographique des pinceaux (5.3.3.1.1). Ainsi, la version de sommation SIMD verticale offre de bonnes performances sur l'ensemble des configurations, mais reste sur les configurations "bien formées" moins performante que la version scalaire différentielle.

5.4.3.1.3 Conclusion et stratégie de sommation retenue L'étape de sommation est dépendante des caractéristiques de la configuration simulée. D'une part le nombre de pinceaux qui contribuent au champ dépend de la complexité géométrique de la pièce et du choix d'implémentation des étapes de calcul précédentes. D'autre part les performances de la sommation sont liées à la manière dont le faisceau simulé se focalise dans la pièce. Les configurations sur lesquelles la sommation SIMD est performante sont celles pour lesquelles trop de pinceaux sont calculés alors qu'il ne contribuent pas au champ (trop de surfaces traitées). Sur les configurations bien conditionnées, les performances de la sommation différentielle sont telles sur les différentes architectures GPP que cette version de la sommation est retenue pour la version optimisée du code de calcul de champ rapide.

5.4.3.2 Étape 3.2 : Extraction du maximum d'amplitude

Afin de transformer la réponse impulsionnelle de déplacement en maximum d'amplitude, cette étape fait appel aux sous opérations suivantes :

- **FFT_R2C** ;
- **convolution** dans le domaine fréquentiel ;
- **FFT_C2R** ;
- **module** d'un signal de vecteur (signal de déplacement) ;
- **enveloppe** d'un signal réel ;
- **FFT_C2C** nécessaire à la routine précédente ;
- **maximum** du signal d'enveloppe, pour obtenir le maximum de déplacement et le temps de vol associé.

Les routines réalisant des FFT ont été accélérées par la bibliothèque Intel MKL, comme présenté précédemment à la section 5.2.3. Les fonctions de convolution, de calcul du module et de calcul de l'enveloppe[6] ont été optimisées à l'aide de Boost.SIMD afin de tirer parti des jeux d'instruction SIMD disponibles sur les différents GPP.

La table 5.15 reprend les performances mesurées en cycles GPP écoulés, sur des signaux de taille arbitraire (10000 signaux de taille 2048 échantillons temporels). Les performances affichées sont mesurées sur des codes *monothread*. Puisque ces opérations sont quasi-régulières sur l'ensemble des échantillons des signaux (hormis aux bornes), le calcul de l'intensité arithmétique de celles-ci est aisé. Cependant, ce calcul fait ressortir la faible intensité arithmétique des opérations. Celle-ci indique des fonctions plutôt *memory bound*.

	version	convolution (hors FFT)	module	enveloppe
Intensité Arithmétique opérations/octet	(Lecture/Écriture), en flottants	3N/1N	3N/1N	6N
	Opérations (flop)	8N	6N	5N
	Ratio	0.5	0.37	0.21
Xeon Westmere	Scalaire	1.5×10^8	3.8×10^8	1.1×10^9
	SSE 4.2	1.3×10^8	1.4×10^8	7.6×10^8
	Gain	$\times 1.2$	$\times 2.7$	$\times 1.4$
Xeon Sandy Bridge	Scalaire	0.9×10^8	3.0×10^8	7.2×10^8
	AVX	0.6×10^8	0.8×10^8	4.8×10^8
	Gain	$\times 1.7$	$\times 3.8$	$\times 1.5$
Xeon Ivy Bridge	Scalaire	0.7×10^8	1.1×10^8	4.4×10^8
	AVX	0.4×10^8	0.5×10^8	3.3×10^8
	Gain	$\times 1.7$	$\times 2.1$	$\times 1.3$
Xeon Haswell	Scalaire	0.8×10^8	1.4×10^8	5.3×10^8
	AVX2	0.5×10^8	0.6×10^8	3.8×10^8
	Gain	$\times 1.6$	$\times 2.3$	$\times 1.4$

TABLE 5.15 – Performances des traitements optimisées (10000 signaux de taille 2048 échantillons, performances en cycles)

Ainsi, les traitements mesurés montrent une accélération notable des différentes phases du traitement du signal nécessaire au calcul de champ, mais ce gain reste en raison de sa faible intensité arithmétique, inférieur aux largeurs des registres SIMD disponibles.

[6] La fonction de calcul d'enveloppe contient deux appels à la FFT_C2C MKL qui sont mesurés pour les deux implémentations

5.4.4 Passage à l'échelle de la parallélisation

Les sections précédentes ont traité des performances, principalement *monothread*, en optimisant l'algorithme pour tirer parti des capacités de calcul SIMD d'un cœur de GPP. La présente section détaille les gains obtenus en parallélisant le code de calcul de champ, en opposant les performances de l'algorithme horizontal de référence avec celles de l'algorithme vertical de simulation de champ (présenté à la sous-section 5.2).

L'API OpenMP permet de configurer la distribution des *threads* sur les différents cœurs d'un GPP. A un premier niveau, directement dans le code, à côté du `pragma parallel omp for` il est possible de définir la stratégie d'ordonnancement avec laquelle la boucle `for` est répartie sur les cœurs et ainsi de prendre en compte la régularité du calcul. Dans le cadre du calcul de champ, en particulier pour la version verticale, les données générées par un point de champ ne sont pas réutilisées par ses voisins (contributions des pinceaux, réponses impulsionnelles et signaux de champ). Aussi la répartition des indices des boucles parallélisées se fait dynamiquement.

Afin d'illustrer les résultats des différentes stratégies de parallélisation, la table 5.16 présente quelques résultats sur Xeon Westmere et Xeon 2×Ivy Bridge, les GPP présentant à la fois le plus de cœurs et gérant l'*hyperthreading*. Globalement on y observe que l'*hyperthreading* est presque toujours avantageux en termes de performances sur le calcul de champ complet.

On remarque, sur Xeon Westmere, que la parallélisation se déroule efficacement lorsque l'*hyperthreading* n'est pas activé (autant de *threads* que les N=12 cœurs). Lorsque l'*hyperthreading* est activé, l'efficacité dépasse les 100% (par rapport au nombre de cœurs physiques) sur le Xeon Westmere pour les étapes de calcul les plus intensives (étape 1 SIMD). Sur l'étape 3, la performance est moins marquée et ne bénéficie pas de l'*hyperthreading*. Cela indique un déséquilibre entre les accès mémoire qui sont saturés et le calcul. Une fois ramené à l'algorithme vertical, puisque l'étape 1 est prépondérante en temps d'exécution et qu'elle subit les meilleures accélérations, le gain total est de 118%.

Sur le Xeon 2×Ivy Bridge, ces observations sont aussi valables. Cependant, par rapport au Xeon Westmere, le ratio de son nombre de cœurs par canal mémoire est moins bon (12/4 contre 6/3). Cela explique la performance moindre du 2×Ivy Bridge, en proportion du nombre maximal de *threads*. OpenMP ne permet pas d'obtenir une efficacité supérieure à 90% sur le code vertical, par rapport à 24 cœurs (sur 48 *threads*). Cependant, une accélération de ×21.8 sur une machine disposant de 24 cœurs physiques / 48 cœurs logiques est satisfaisante par rapport au coûts de développement engagés dans l'obtention de cette accélération.

		Scalaire				SIMD				
		Step 1	Step 2	Step 3	Total Horizontal	Step 1	Step 2	Step 3	Total Horizontal	Vertical
Xeon Westmere	N=12	×11.6 96%	×10.9 91%	×10.7 90%	×11.3 94%	×10.4 87%	×11.0 91%	×10.6 88%	×10.5 88%	×11.1 93%
	2N=24	×16.2 134%	×15.6 130%	×10.6 88%	×13.8 114%	×13.3 110%	×15.5 130%	×10.7 88%	×11.8 98%	×14.2 118%
Xeon 2×Ivy Bridge	N=24	×20.0 83%	×16.7 70%	×15.7 65%	×18.4 77%	×17.5 73%	×16.8 70%	×15.2 63%	×16.1 67%	×17.1 71%
	2N=48	×27.2 114%	×18.6 78%	×15.0 62%	×21.7 90%	×22.6 94%	×18.7 78%	×15.0 62%	×17.5 74%	×21.8 90%

TABLE 5.16 – Performances du *multithreading* sur Xeon Westmere et Xeon 2×Ivy Bridge : accélération et efficacité sur la configuration 01 (par rapport au nombre de cœurs physiques)

5.5 Synthèse des accélérations obtenues

Cette sous-section récapitule les gains obtenus à l'aide des différentes optimisations apportées au code de calcul de champ sur GPP en vue d'obtenir des performances interactives.

5.5.1 Répartition des traitements avant/après optimisation

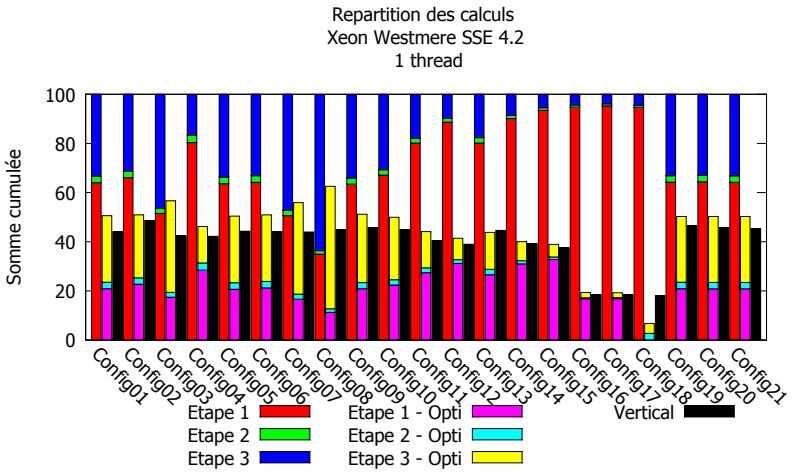
Afin d'observer la totalité des accélérations obtenues au moyen des différentes optimisations apportées au calcul de champ, cette section présente la répartition avant et après des différentes étapes du calcul. Pour rappel, la version de référence utilise déjà la bibliothèque MKL pour les opérations de traitement du signal. L'implémentation optimisée reprend les optimisations individuelles suivantes :

- Étape 1 : Calcul des pinceaux *SIMDizé* ;
- Étape 2 : pas d'optimisation spécifique ;
- Étape 3 : sommation scalaire différentielle, traitement du signal *SIMDizé* ;
- Version verticale : regroupe touts ces optimisations en une seule boucle de haut niveau.

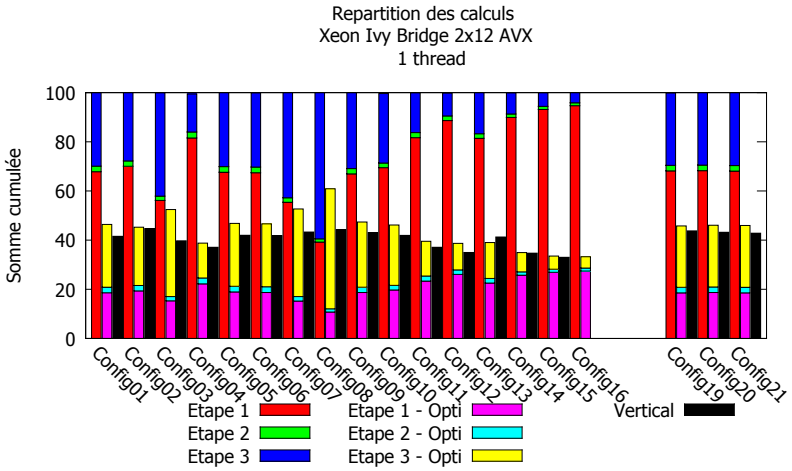
Ainsi la figure 5.11 représente la répartition de ces trois grandes étapes du calcul de champ, sur l'ensemble des configurations de champ et sur l'ensemble des GPP étudiés en exécution *monothread*. Chaque sous figure indique les mêmes informations, chacune pour un GPP donné sous forme d'histogrammes par configuration. Tout d'abord y apparait cumulé et ramené à 100% le temps d'exécution en cycles des trois étapes sur la version de référence, sur la barre de gauche d'une configuration. La barre centrale d'une configuration présente les temps des versions optimisées, exprimés en pourcentage du temps total de la version de référence. Enfin, la barre de droite présente les performances, en cycles, de la version verticale, regroupant les accélérations. Ainsi, il y est possible de lire l'accélération globale apportée et les accélérations particulières de chaque étape du calcul de champ.

Globalement, on observe sur la figure 5.11 que l'étape 2, qui n'a pas subi d'optimisation spécifique, son temps d'exécution est toujours très faible par rapport au temps d'exécution du code optimisé.

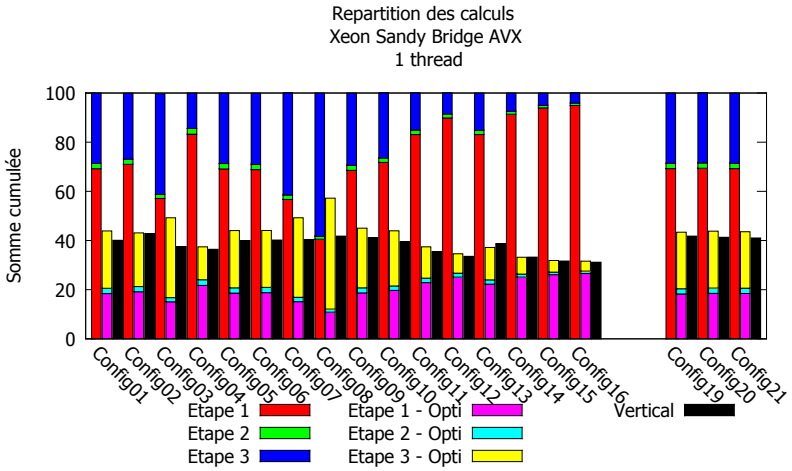
La figure 5.12 présente, pour sa part, les performances avant/après optimisation, cette fois ci entre un code de référence et des codes optimisés tous les deux *multithreads*. La parallélisation ne change pas l'ordre d'importance en temps de calcul des différentes étapes du champ. Les optimisations bas niveau gardent un impact important sur l'ensemble des configurations. Ainsi l'implémentation regroupant ces optimisations avec un code "vertical" est la plus rapide.



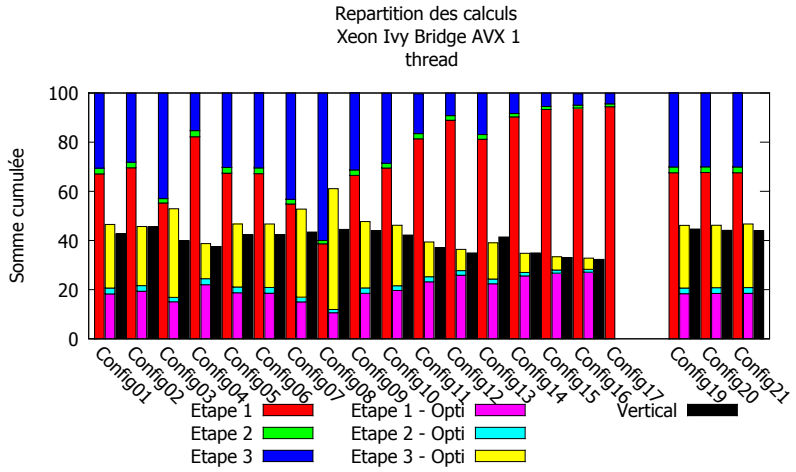
(a) Xeon Westmere



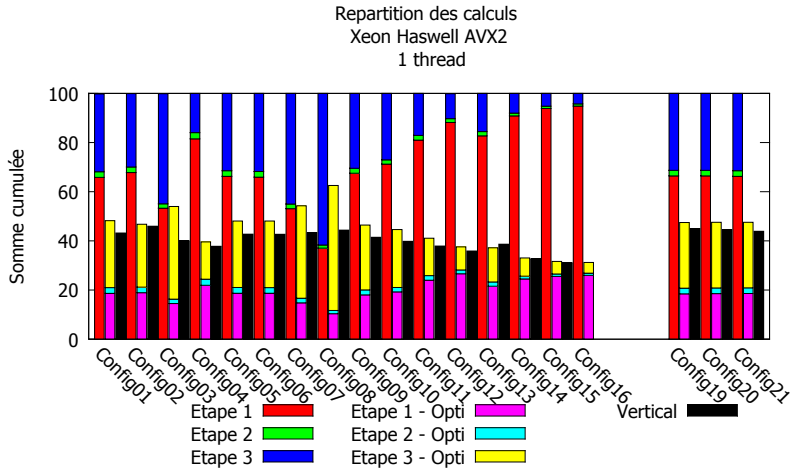
(b) Xeon Ivy 2x12



(c) Xeon Sandy Bridge

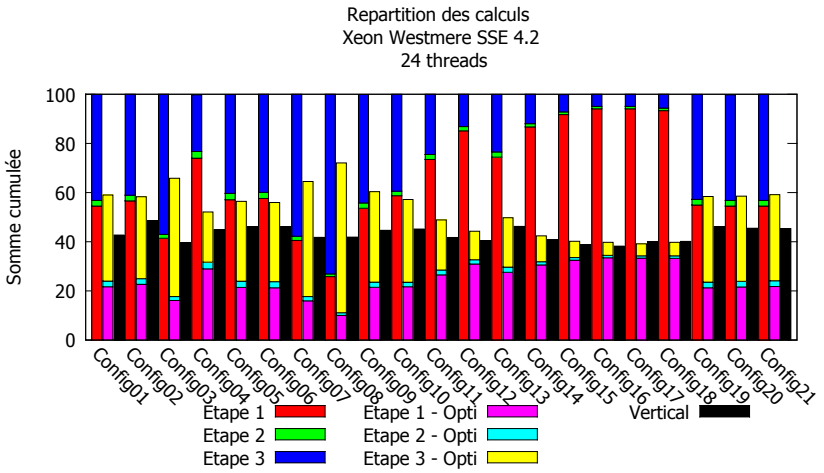


(d) Xeon Ivy Bridge

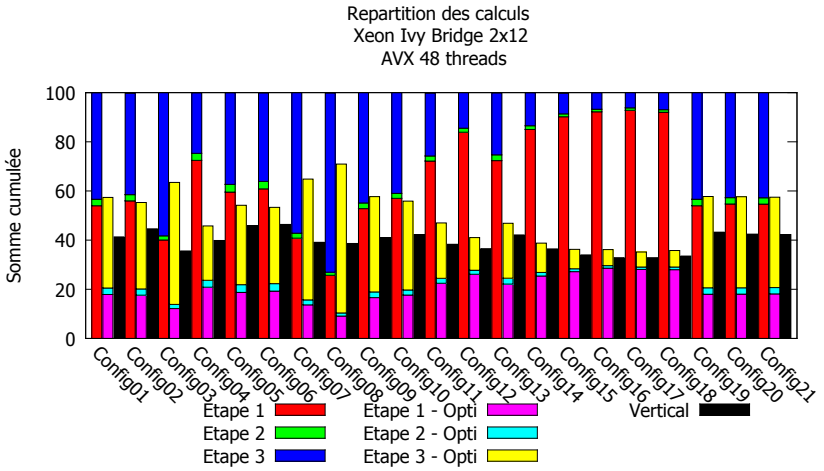


(e) Xeon Haswell

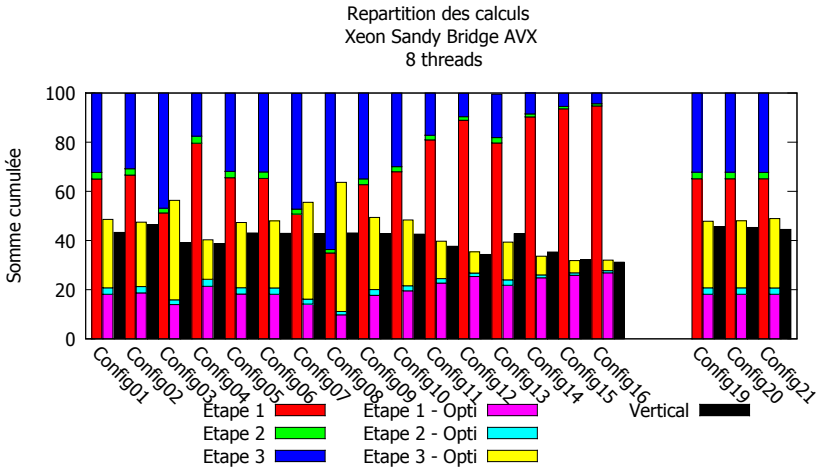
FIGURE 5.11 – Répartition des étapes du calcul de champ avant/après optimisation *monothread*



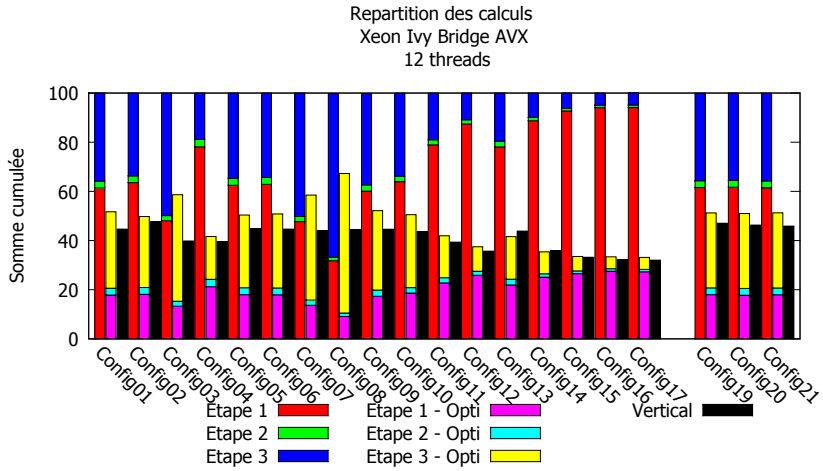
(a) Xeon Westmere



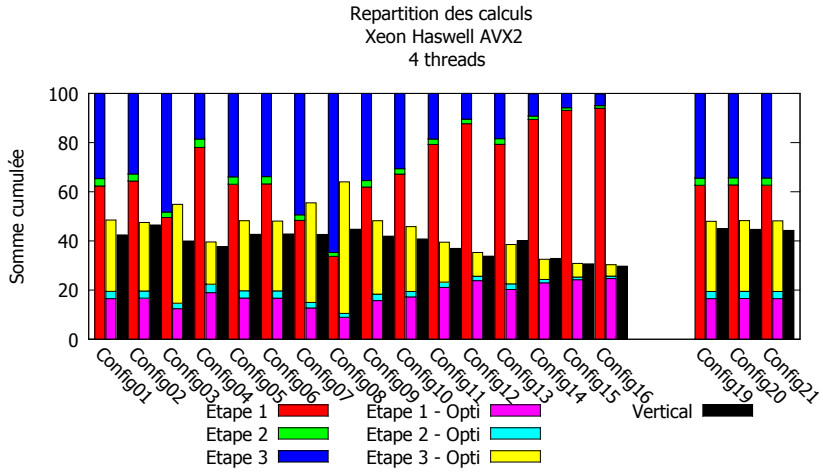
(b) Xeon Ivy 2x12



(c) Xeon Sandy Bridge



(d) Xeon Ivy Bridge



(e) Xeon Haswell

FIGURE 5.12 – Répartition des étapes du calcul de champ avant/après optimisation (tout code **multithread**)

5.5.2 Sur une configuration de référence

Le présent paragraphe rassemble les résultats obtenus, en millisecondes, sur les différents GPP étudiés, pour la configuration 1 du jeu de *benchmarks*. La table 5.17 présente l'ensemble des GPP étudiés et référence les tables où sont détaillés ces résultats.

En particulier le tableau 5.18 présente les performances sur architecture Westmere, au moyen de l'utilisation des instructions SSE (128bits) et en utilisant les 12 cœurs du processeur Xeon X5690. Ligne par ligne, les performances des trois étapes du calcul de champ sont présentées indépendamment. Elles sont obtenues à l'aide d'une version horizontale du calcul de champ avec et sans optimisations. Il faut noter que le code "sans optimisations" utilise, à l'étape 3, un algorithme une sommation différentielle scalaire et la bibliothèque Intel MKL pour le calcul des FFT. La version optimisée qui lui est comparée utilise les instructions SIMD. Sont également comparées les performances de la version horizontale (obtenues par sommation des temps par étapes) et celles d'une version verticale (déclinée en implémentation scalaire d'une part et SIMD d'autre part). Chacune des accélérations dues aux registres SIMD est rappelée, ainsi que les performances des versions *multithread*ées : l'une sans *hyperthreading* et l'autre avec. La dernière colonne du tableau rappelle l'accélération totale obtenue, en combinant la version optimisée et l'*hyperthreading*. Les performances de la version la plus rapide sont situées en bas à droite du tableau (version verticale, optimisée et *hyperthread*ée). Enfin, la dernière ligne récapitule les images par seconde obtenues pour chacune des versions.

Pour une étape de calcul donnée, l'ensemble des architectures GPP étudiées obtient, malgré des jeux d'instructions différents, des gains du même ordre. Le point crucial pour obtenir des performances sur GPP reste le nombre de cœurs disponible sur une machine, permettant une accélération quasi linéaire grâce à l'*hyperthreading*. Cela est très visible sur les architectures Xeon Westmere 2×6 cœurs (table 5.18) et Xeon Ivy Bridge 2×12 cœurs (table 5.21) pour lesquelles la parallélisation fournit des gains d'un ordre de grandeur supérieur à la *SIMDization* (×11.12 par rapport à ×2.0 au total sur Westmere, ×17.1 par rapport à ×2.18 sur Ivy Bridge). On observe que sur ce même Xeon Ivy Bridge 2×12 cœurs, l'objectif d'interactivité est atteint pour la configuration 1 avec 38.0 fps. De plus, le Xeon Westmere 2×12 cœurs obtient un taux de rafraichissement notable de 17.4 fps alors qu'il s'agit de la machine la plus ancienne du *benchmark*.

Sur une machine modeste, disposant de 4 cœurs et d'instructions SIMD AVX2/FMA telles que le Xeon Haswell, l'accélération finale n'est que de ×11.5 sur la configuration de référence. A l'inverse, sur une machine disposant de 24 cœurs, le 2×Xeon IvyBridge, le calcul est accéléré d'un facteur ×47.6.

Sur des configurations où les traitements entre le calcul des pinceaux (étape 1) et la sommation des contributions (étape 3) sont déséquilibrés de par la complexité géométrique mise en jeu (beaucoup de pinceaux à calculer pour parcourir les surfaces, mais peu qui contribuent) telles que la configuration 16, dans la mesure où l'étape 1 est la plus accélérée, l'accélération sur le 2×Xeon Ivy Bridge atteint un facteur ×83.3. Alors que les étapes 2 et 3 "ne sont accélérées" que de facteurs ×30.5 et ×18.4 respectivement. Cependant, des performances interactives ne sont pas atteintes pour ce genre de configuration. Les performances sont de 1.94fps sur le puissant 2×Xeon Ivy Bridge.

Processeur	Architecture	SIMD	Nb cœurs/*Hyperthreading*	Table
2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz	Westmere	SSE	12/24	5.18
1 x Intel(R) Xeon(R) CPU E3-1290 @ 3.60 GHz	Sandy Bridge	AVX	4/8	5.19
1 x Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50 GHz	Ivy Bridge	AVX	6/12	5.20
2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz	Ivy Bridge	AVX	24/48	5.21
1 x Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz	Haswell	AVX2	4/8	5.22

TABLE 5.17 – Récapitulatif des performances globales des GPP présentés (configuration 01)

2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz									
SSE 128 bits		Scalaire 1 *thread*	SIMD 1 *thread*	Gain SIMD (scalaire/SIMD)	12 cores + SIMD	Gain // (SIMD 1th/12th)	24 *threads* + SIMD	Gain HT (SIMD 1th/24th)	Gain Total (par étape) (scalaire 1th / SIMD 24 th)
Config01 Horizontal	Étape 1	1185.3 ms	386.2 ms	x3.1	37.2 ms	x10.4	29.1 ms	x13.3	x40.7
	Étape 2	48.8 ms	48.8 ms	x1.0	4.4 ms	11.0	3.1 ms	x15.6	x15.6
	Étape 3	616.3 ms	500.7 ms	x1.2	57.5 ms	x8.8	57.9 ms	x8.7	x10.7
	Total	1850.5 ms	935.7 ms	x2.0	99.1 ms	x9.4	90.2 ms	x10.4	x20.5
Config01	Vertical	1648.4 ms	817.0 ms	x2.0	73.4 ms	x11.2	57.5 ms	x14.2	x28.7
		0.6 fps	1.2 fps		13.6 fps		17.4 fps		

TABLE 5.18 – Performances de la Config01 sur architecture Westmere (2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz - 12 cores)

1 x Intel(R) Xeon(R) CPU E3-1290 @ 3.60 GHz									
AVX 256 bits		Scalaire	SIMD	Gain SIMD	4 cores	Gain //	8 *threads*	Gain HT	Gain Total (par étape)
Config01	Étape 1	1057.4 ms	280.9 ms	x3.8	71.3 ms	x4.0	53.5 ms	x5.3	x19.8
	Étape 2	33.1 ms	33.2 ms	x1.0	8.6 ms	x3.9	7.8 ms	x4.2	x4.2
	Étape 3	439.2 ms	356.1 ms	x1.2	113.7 ms	x3.1	96.9 ms	x3.7	x4.5
	Total	1529.7 ms	670.2 ms	x2.3	193.6 ms	x3.5	158.1 ms	x4.2	x9.7
	Vertical	1419.0 ms	611.3 ms	x2.3	154.2 ms	x4.0	127.9 ms	x4.78	x11.10
		0.7 fps	1.6 fps		6.5 fps		7.8 fps		

TABLE 5.19 – Performances de la Config01 sur architecture Sandy Bridge (1 x Intel(R) Xeon(R) CPU E3-1290 @ 3.60 GHz - 4 cores)

1 x Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50 GHz									
AVX 256 bits		Scalaire	SIMD	Gain SIMD	6 cores	Gain //	12 *threads*	Gain HT	Gain Total (par étape)
Config01	Étape 1	906.3 ms	246.2 ms	x3.7	42.2 ms	x5.9	32.3 ms	x7.7	x28.1
	Étape 2	31.8 ms	32.0 ms	x1.0	5.7 ms	x5.6	5.1 ms	x6.3	x6.2
	Étape 3	411.7 ms	348.0 ms	x1.2	70.2 ms	x5.0	65.2 ms	x5.3	x6.3
	Total	1349.8 ms	626.3 ms	x2.2	118.1 ms	x5.3	102.6 ms	x6.10	x13.2
	Vertical	1258.4 ms	575.2 ms	x2.2	98.2 ms	x5.9	81.0 ms	x7.1	x15.6
		0.8 fps	1.8 fps		10.2 fps		12.3 fps		

TABLE 5.20 – Performances de la Config01 sur architecture Ivy Bridge (1 x Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50 GHz - 6 cores)

2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz									
AVX 256 bits		Scalaire	SIMD	Gain SIMD	24 cores	Gain //	48 *threads*	Gain HT	Gain Total (par étape)
Config01	Étape 1	935.2 ms	257.0 ms	x3.6	14.7 ms	x17.5	11.4 ms	x22.6	x82.1
	Étape 2	31.2 ms	31.2 ms	x1.0	1.9 ms	x16.8	1.7 ms	x18.7	x18.7
	Étape 3	414.2 ms	352.1 ms	x1.2	26.4 ms	x13.2	27.6 ms	x12.8	x15.0
	Total	1380.7 ms	640.3 ms	x2.2	43.0 ms	x14.9	40.7 ms	x15.8	x34.0
	Vertical	1253.7 ms	573.9 ms	x2.2	33.5 ms	x17.1	26.3 ms	x21.8	x47.6
		0.8 fps	1.7 fps		29.8 fps		38.0 fps		

TABLE 5.21 – Performances de la Config01 sur architecture Ivy Bridge (2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz - 24 cores)

1 x Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz									
AVX2 256 bits		Scalaire	SIMD	Gain SIMD	4 cores	Gain //	8 *threads*	Gain HT	Gain Total (par étape)
Config01	Étape 1	755.4 ms	213.7 ms	x3.5	54.6 ms	x3.9	38.9 ms	x5.5	x19.4
	Étape 2	26.6 ms	26.8 ms	x1.0	7.5 ms	x3.6	7.0 ms	x3.9	x3.8
	Étape 3	364.0 ms	311.6 ms	x1.2	94.5 ms	x3.3	81.6 ms	x3.8	x4.5
	Total	1146.0 ms	552.0 ms	x2.1	156.6 ms	x3.6	127.4 ms	x4.3	x9.0
	Vertical	1150.2 ms	494.5 ms	x2.3	124.9 ms	x4.0	100.1 ms	x5.0	x11.5
		0.9 fps	2.0 fps		8.0 fps		10.0 fps		

TABLE 5.22 – Performances de la Config01 sur architecture Haswell (1 x Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz - 4 cores)

5.5.3 Synthèse sur l'ensemble des configurations

Le précédent paragraphe a montré les performances sur la configuration de champ 1 pour tous les GPP. Cette configuration est la base du jeu de configurations étudiées, à partir de laquelle on effectue des variations (géométrie, mode, nombre de pinceaux...). Par construction, c'est la moins coûteuse en temps de calcul. Afin de conclure sur l'interactivité, les performances sur l'ensemble des configurations de champ sont mesurées sur les architectures GPP ayant les capacités à atteindre un nombre d'images de champ par seconde de l'ordre de l'interactivité (25 images/seconde). Les performances du 2×Xeon X5690 d'architecture Westmere sont présentées dans le tableau 5.23 et celles du 2×Xeon E5-2697 v2 d'architecture Ivy Bridge sont regroupées dans la table 5.24.

Les performances du Xeon Westmere sur l'implémentation de référence (scalaire *monothread*) présentées dans le tableau 5.23 mettent en lumière la totalité de l'optimisation réalisée. A titre d'illustration, les performances du modèle de calcul complet de CIVA 11.0 obtenues sur le même 2×Xeon X5690 Westmere y sont également présentées. On observe une accélération significative par rapport à l'implémentation de référence, d'un facteur allant de ×30 à ×40 suivant les configurations et les répartitions des étapes du calcul. Par rapport à CIVA 11, on observe sur le Xeon Westmere une accélération significative sur les différentes configurations. Cependant, on peut remarquer que le code de CIVA disposant d'heuristiques plus complexes, l'impact de l'augmentation de la complexité géométrique de la configuration se fait moins sentir. Ainsi les performances des configurations 14 à 17 restent quasi constantes car le modèle élimine les couples de surface n'ayant pas d'impact sur le calcul de champ.

On obtient de bonnes performances sur le Xeon Westmere, sans atteindre toutefois l'interactivité puisque pour les configurations les plus simples on obtient environ de 17 images/seconde. La table 5.24 qui indique les performances de la simulation de champ sur le double-GPP ×2 Xeon Ivy Bridge E5-2697 v2 montre que pour plusieurs configurations, l'objectif d'interactivité, c'est à dire un nombre supérieur à 25 images/seconde, est atteint !

	2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz				CIVA 11.0
	Architecture	Westmere	2×6 cœurs	SSE 128bits	
	Impl. Ref	Optimisé	(gain)	Images par seconde	
Config01	1850 ms	57.5 ms	×32.2	17.4 fps	40 s
Config02	2006 ms	69.9 ms	×28.7	14.3 fps	2 min 30 s
Config03	4986 ms	152.9 ms	×32.6	6.5 fps	2 min 32 s
Config04	6611 ms	188.9 ms	×35.0	5.3 fps	2 min 34 s
Config05	7294 ms	228.7 ms	×31.9	4.4 fps	2 min 53 s
Config06	28978 ms	899.2 ms	×32.2	1.1 fps	11 min 38 s
Config07	2343 ms	76.2 ms	×30.7	13.1 fps	1 min 40 s
Config08	3402 ms	120.3 ms	×28.3	8.3 fps	1 min 56 s
Config09	3884 ms	126.1 ms	×30.8	7.9 fps	1 min 55 s
Config10	8849 ms	284.1 ms	×31.1	3.5 fps	4 min 54 s
Config11	4349 ms	119.5 ms	×36.4	8.4 fps	1 min 00 s
Config12	12940 ms	329.4 ms	×39.3	3.0 fps	57 s
Config13	2192 ms	67.7 ms	×32.4	14.8 fps	57 s
Config14	5834 ms	150.4 ms	×38.8	6.7 fps	1 min 13 s
Config15	18478 ms	447.2 ms	×41.3	2.2 fps	1 min 20 s
Config16	54973 ms	1299.8 ms	×42.3	0.8 fps	1 min 21 s
Config17	182315 ms	4488.7 ms	×40.6	0.2 fps	1 min 21 s
Config18	199036 ms	4865.2 ms	×40.9	0.2 fps	2 min 50 s
Config19	1863 ms	62.4 ms	×29.9	16.0 fps	39 s
Config20	1869 ms	61.4 ms	×30.4	16.3 fps	39 s
Config21	1868 ms	61.1 ms	×30.6	16.4 fps	39 s

TABLE 5.23 – Performances du Xeon Westmere 2×X5690 sur l'ensemble des configurations

	2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz Ivy Bridge AVX 256bits	
	Temps de calcul	Images par seconde
Config01	26.3 ms	38.0 fps
Config02	30.4 ms	32.9 fps
Config03	64.7 ms	15.5 fps
Config04	81.3 ms	12.3 fps
Config05	102.7 ms	9.7 fps
Config06	404.3 ms	2.5 fps
Config07	33.0 ms	30.3 fps
Config08	52.2 ms	19.2 fps
Config09	54.2 ms	18.5 fps
Config10	123.4 ms	8.1 fps
Config11	51.7 ms	19.4 fps
Config12	139.2 ms	7.2 fps
Config13	29.3 ms	34.1 fps
Config14	62.2 ms	16.1 fps
Config15	180.0 ms	5.6 fps
Config16	514.4 ms	1.9 fps
Config17	1694.3 ms	0.6 fps
Config18	1863.2 ms	0.5 fps
Config19	27.7 ms	36.1 fps
Config20	27.0 ms	37.0 fps
Config21	26.9 ms	37.1 fps

TABLE 5.24 – Performances du Xeon Ivy Bridge 2×E5-2697v2 sur l'ensemble des configurations

5.5.4 Remarque sur l'impact du compilateur

Avant de conclure, il convient de revenir sur les performances présentées lors de la conférence COFREND 2014 [LRL14]. Les mesures présentées avaient été réalisées sur le même GPP Xeon Ivy Bridge 2×E5-2697v2, mais sous le système d'exploitation Windows, au moyen d'exécutables compilés avec **MSVC2013**. En outre, il faut noter les différences de mesures réalisées à cette occasion :

- seul le jeu d'instruction SSE4.2 a été utilisé, pas les instructions AVX[7] ;
- la phase de sommation (étape 3.1) est la version scalaire de référence, les accélérations concernant surtout la partie traitement du signal ;
- l'*Hyperthreading* n'était pas activé sur le GPP (il n'y avait que 24 *threads* actifs).

Ainsi, les performances qui avaient été obtenues, rappelées dans la table 5.25, sont à mettre en relation avec la table 5.21 quant aux performances brutes sur la machine 2×E5-2697v2 avec des instructions AVX. Il y a presque un facteur ×2 entre les deux versions (code vertical optimisé passant de 20.4 fps à 38.0 fps), alors que ni les instructions SIMD AVX ni l'*hyperthreading* ne peuvent être les seuls responsables d'un tel ordre de grandeur. On peut donc penser que l'utilisation du compilateur Intel est responsable de cet état.

2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz							
SSE 128 bits		Scalaire	SIMD	Gain SIMD	24 cores	Gain //	Gain Total (par étape)
Config01	Étape 1	2776.3ms	644.1ms	x4.31	31.8ms	x20.3	x87.3
	Étape 2	35.8ms	35.8ms	x1.0	2.0ms	x17.9	x17.9
	Étape 3 (Scalaire)	808.2ms	508.3ms	x1.6	29.5ms	x17.2	x27.4
	Total	3620.2ms	1240.7ms	x2.9	63.3ms	x19.6	x57.2
	Vertical	2969.2ms	899.8ms	x3.3	48.9ms	x18.4	x60.7
		0.3 fps	1.1 fps		20.4 fps		

TABLE 5.25 – Performances **MSVC2013 - Windows** de la Config01 sur architecture Ivy Bridge (2 x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz - 24 cores)

5.5.5 Conclusion sur l'implémentation GPP

5.5.5.1 Performances

L'utilisation de GPP permet d'obtenir des simulations de champ interactives sur plusieurs configurations de champ significatives. L'utilisation des instructions SIMD de ces GPP permet de bénéficier d'un premier facteur d'accélération par rapport à l'implémentation séquentielle scalaire de référence. A plus haut niveau, l'utilisation d'OpenMP permet de tirer parti des cœurs des GPP et de bénéficier des capacités de l'*hyperthreading*. Pour plusieurs configurations, l'objectif d'interactivité, c'est à dire un taux supérieur à 25 images/seconde, est atteint ! En particulier sur les machines performantes disposant d'un grand nombre de cœurs de calcul.

Le nombre de cœurs d'un GPP reste le facteur prépondérant dans les performances que ce GPP sera capable d'atteindre. Les mesures effectuées ont montré que la bande passante est un critère limitant secondaire, surtout concernant la phase de sommation des pinceaux. Par contre, l'intérêt de passer d'un jeu d'instructions SIMD de 128 bits à 256 bits est pour l'instant discutable en termes de performances. Les gains effectifs dans le cadre du calcul de champ sont similaires (évolution de ×3 à ×4 environ pour un doublement de la largeur). Côté fondeur, les

[7] Un problème de compatibilité sur la version courante de Boost.SIMD de l'époque et MSVC2013.

performances des instructions "de calcul généraliste" 256 bits ne sont pas toutes au niveau de performance des instructions de la génération précédente (comme par exemple les instructions de division et de racine carrée...). On pourrait remettre en question l'efficience d'un portage SSE vers AVX d'un tel code (bien que les instructions soient proches), cependant l'outil Boost.SIMD permet de les utiliser sans effort d'implémentation supplémentaire : l'accélération reste utile et efficiente.

A partir de la configuration de référence, on peut établir le tableau 5.26 qui rappelle les performances de chaque GPP étudié et les performances atteintes. On y constate que le prix de l'interactivité est de 138 USD/fps. Le GPP le plus "rentable" est le modeste mais récent (2013) Xeon Haswell avec un rapport coût/image de 27.3 USD/fps mais la cadence atteinte n'est que de 10 fps. Cette mesure n'a de valeur qu'indicative, la puissance de calcul d'un GPP n'est pas une fonction linéaire de son prix. Cet indicateur ne permettra que d'obtenir une estimation grossière des performances que pourra atteindre une machine pour un budget donné.

	2×Xeon Westmere	Xeon Sandy Bridge	Xeon Ivy Bridge	2×Xeon Ivy Brigde	Xeon Haswell
Prix à la sortie (USD)	3326	885	583	5228	273
Date de sortie	01/2011	05/2011	09/2013	09/2013	06/2013
Type de machine	bi-GPP	mono-GPP	mono-GPP	bi-GPP	mono-GPP
FPS	17.4	7.8	12.3	38.0	10.0
Interactivité	non	non	non	oui	non
USD / fps	191	113	47	138	27

TABLE 5.26 – Performances économiques des GPP - configuration 01

L'étape qui bénéficie au mieux de ces accélération est l'étape 1, celle du calcul des pinceaux. Sa régularité globale, à part la recherche de racines à l'aide de la méthode de Newton qui heureusement n'est pas prépondérante, permet de tirer parti des instructions SIMD et ses traitements se parallélisent de manière satisfaisante. Le reste du code bénéficie principalement de la parallélisation et de l'usage d'une bibliothèque optimisée. En particulier pour l'étape 3, le choix a été fait de ne pas retenir la sommation favorable à une grande quantité de pinceaux (sommation SIMD) mais de garder une version scalaire différentielle qui présente de meilleures performances dans les cas d'utilisation visés : un faisceau focalisé dont les contributions sont denses parmi les pinceaux. Le traitement du signal, sur cette étape, a bénéficié des instructions SIMD et de la parallélisation.

5.5.5.2 Limitations

La simulation du champ sur GPP permet d'atteindre des performances proches de l'interactivité sur des géométries de taille réaliste. Cependant, dès que la complexité augmente les performances chutent, notamment :

- lorsqu'il est nécessaire de tracer énormément de pinceaux pour au final en éliminer un grand nombre pour des raisons physiques ;
- lorsque peu de pinceaux contribuent effectivement au champ, la sommation différentielle scalaire qui vise à réduire les accès mémoire ne présentant pas d'accélération significative.

Enfin, une limitation concerne le coût mémoire lié à la gestion des données temporaires. Elle sera aisément résolue avec l'utilisation de l'algorithmie verticale. Pour permettre l'usage des différentes versions du calcul de champ testées, les données temporaires (pinceaux à maintenir en mémoire pour appliquer les retards, signaux en tout genre...) sont relativement lourdes à maintenir en vie et ne permettent pas la gestion de configurations très complexes sur des machines ne disposant pas de beaucoup de mémoire.

Chapitre

6 Optimisations sur architecture MIC

6.1 Xeon Phi : un accélérateur déporté

Le Xeon Phi est un accélérateur déporté, destiné à augmenter les performances de serveurs et de stations de calcul. Il est développé par Intel pour exécuter du code compatible x86 spécifique. La méthode la plus simple (en termes de développement) pour mesurer les performances de cette architecture consiste à utiliser un code de calcul GPP standard et de le compiler à l'aide du compilateur ICC en activant le support pour le Xeon Phi. Il peut y être copié puis directement exécuté depuis le sous-système d'exploitation résidant sur le Xeon Phi, proche d'un système d'exploitation Linux. C'est ce mode de fonctionnement qui est retenu pour la simulation du calcul de champ, plutôt que le comportement "accélérateur déporté" d'une application hybride, utilisant un Xeon Phi piloté depuis le PC hôte (à la manière d'un GPU).

Quelques contraintes sont cependant à prendre en compte pour obtenir un code compatible depuis une application existante. Bien que le Xeon Phi soit compatible x86, le code d'origine ne doit pas faire appel à des instructions SIMD "usuellement" disponibles sur les architectures x86 Intel telles que les jeux d'instructions SSE, AVX... car elles ne sont pas supportées par le Xeon Phi.

6.1.1 Spécificités d'implémentation

Afin de tirer parti des spécificités du Xeon Phi, l'implémentation du calcul de champ repose sur des versions scalaires et des versions *SIMDizées* à l'aide de Boost.SIMD. Pour le calcul de champ, l'algorithmie utilisée est la même que sur les architectures des GPP. En raison de la similitude des algorithmes entre les architectures GPP et ceux exploitant le Xeon Phi, l'analyse est également faite en gardant le même plan.

Les mesures présentées dans ce chapitre ont été réalisées ici sur un Xeon Phi 3120, de microarchitecture Knights Corner. Il s'agit d'un coprocesseur cadencé à 1.1 GHz, composé de 57 cœurs de calcul et disposant de 6 giga-octets de mémoire vive. Il convient de noter, en terme de compatibilité avec les codes GPP, que les Xeon Phi de cette génération ne peuvent exécuter des instructions SIMD des jeux d'instructions "classiques" des GPP. Seules les instructions SIMD KNC (SIMD 512 bits) peuvent s'y exécuter.

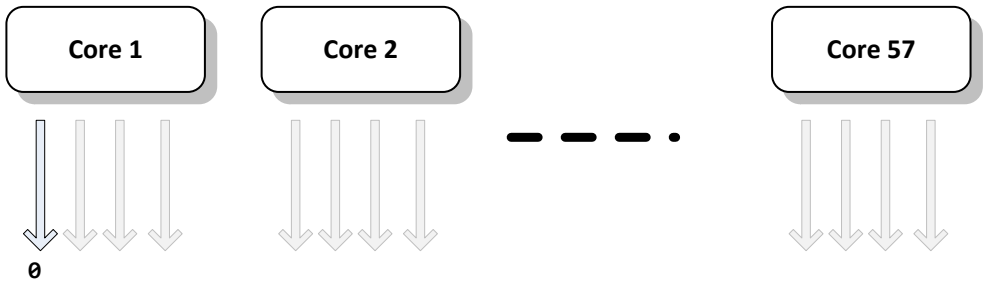
6.1.2 Étape 1 - Calcul des pinceaux

6.1.2.1 Minibenchmark - Calcul du polynôme et méthode de Newton

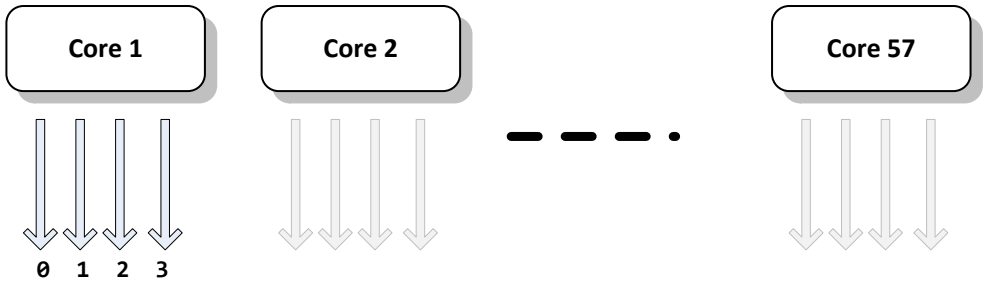
En pratique, dans le cadre du calcul de champ réalisé sur GPP au chapitre précédent (*c.f.* paragraphe 5.4.1.1.1), l'analyse réalise une comparaison des performances de Boost.SIMD avec une version SSE codée manuellement. Elle ne pourra pas être exécutée sur Xeon Phi, contrairement au GPP, en raison de l'absence d'instructions SSE sur cette architecture. On se contente donc d'exécuter le code réalisant le *minibenchmark* de la résolution de polynômes de degrés 4 par la méthode de Newton en scalaire et en vectoriel Boost.SIMD. Pour rappel, ce *minibenchmark* se déroule dans les mêmes conditions que sur GPP. Le nombre d'itérations nécessaires pour obtenir un zéro d'un polynôme par la méthode de Newton dépend des données (coefficients et approximation de la racine initiale). Pour éviter ces variations et réaliser une mesure équitable de l'implémentation SIMD du calcul polynomial, une version *minibenchmark* de la méthode de Newton réalise un nombre d'itérations fixé (sans critère d'arrêt). Ce code n'utilise aucune donnée d'entrée du calcul de champ (les coefficients des polynômes étant générés aléatoirement), le modèle d'exécution *on device* ne limite pas ni n'impacte les performances mesurées.

Le Xeon Phi est une architecture nativement *hyperthread*ée, pour laquelle il est conseiller d'utiliser pour 4 *threads* par cœur de calcul pour obtenir des performances correctes. L'implémentation du *minibenchmark* fait donc appel à OpenMP afin de paralléliser le calcul des différents polynômes sur différents threads. Afin de forcer l'exécution de plusieurs *threads* sur un seul cœur, l'affinité *thread*/cœur a été configurée en mode *compact* (les *threads* se regroupent afin de remplir en priorité les cœurs). Les mesures sont ainsi réalisées pour 1, 4 et 228=57×4 *threads* afin d'évaluer les performances réelles d'un cœur (4 *threads*) de Xeon Phi et d'évaluer également le passage à l'échelle du *minibenchmark* sur les 57 cœurs qui composent l'accélérateur étudié (comparaison 4 contre 228 *threads*). La figure 6.1 présente la manière dont les *threads* sont répartis sur les différents cœurs du Xeon Phi, pour 1, 4 et 228 *threads*.

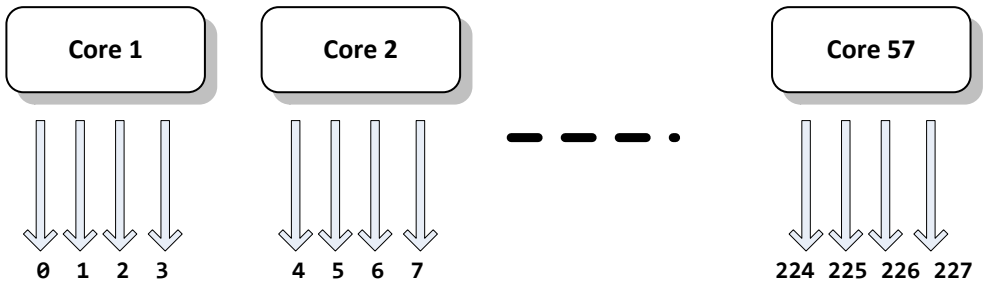
Il n'est pas possible de réaliser une analyse aussi poussée que sur GPP des performances des implémentations en étudiant le comportement très bas niveau des instructions. Les latences et les débits des instructions ne sont pas communiqués par le fondeur. Cependant, la méthode de



(a) Xeon Phi : calcul *monothread*



(b) Xeon Phi : calcul 4 *threads* (Affinité des *threads* : `compact`)



(c) Xeon Phi : calcul 228 *threads* (Affinité des *threads* : `compact`)

FIGURE 6.1 – Illustration des capacités d'un Xeon Phi 57 cœurs pour l'hyperthreading - Affinité des *threads* : `compact`

mesure des performances de l'itération de Newton au moyen d'un modèle linéaire reste valable et appliquée dans ce qui suit, en particulier en analysant les accélérations obtenues.

Cycles par itération	Scalaire	Boost.SIMD	Gain SIMD
1 *threads* (1 cœur)	68.1	4.0	× 17.1 / 106%
4 *threads* (1 cœur)	30.1	1.9	× 15.8 / 99%
1 vs 4 *threads*	× 2.3	× 2.1	-
228 *threads* (57 cœur)	0.537	0.035	× 15.3 / 96%
4 vs 228 threads	× 56.1 / 99%	× 54.3 / 95%	-
Efficacité (57 cœurs)	99%	93%	-

TABLE 6.1 – Minibenchmark Newton - Performances sur Xeon Phi - Boost.SIMD : SIMD KNC 512 bits (16 flottants sp)

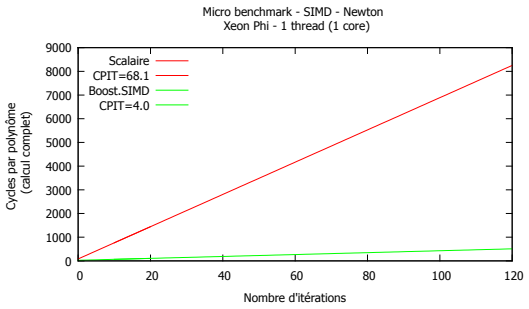


FIGURE 6.2 – Minibenchmark Newton - Xeon Phi 1 *thread*/1 cœur

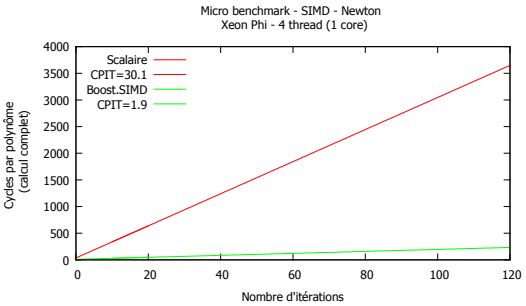


FIGURE 6.3 – Minibenchmark Newton - Xeon Phi 4 *threads*/1 cœur

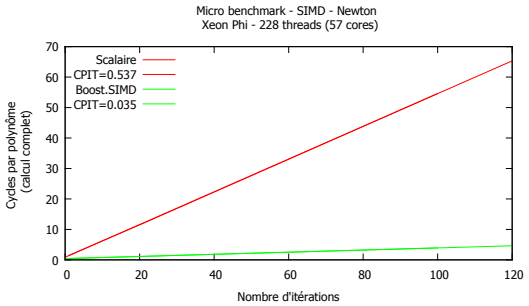


FIGURE 6.4 – Minibenchmark Newton - Xeon Phi 228 *threads*/57 cœurs

Les performances de ces différents cas d'utilisation du Xeon Phi sont présentées sur les graphiques 6.2, 6.3 et 6.4 pour un nombre de *threads* de 1, 4 et 228. La table 6.1 rappelle ces performances et fait apparaître les gains obtenus entre les des différentes versions de l'algorithme.

On observe tout d'abord que le gain des instructions SIMD est, sur un cœur (version 1 *thread*), sur-linéaire et présente une accélération supérieure à ×16 (×17.1). Il est possible que le Xeon Phi utilise des instructions KNC 512 bits en masquant tous les éléments sauf 1 pour obtenir une version équivalente à une instruction *single simple (precision)*. Ainsi, si les instructions sont les mêmes à un masquage près, les performances devraient être linéaires en passant de scalaire à SIMD KNC. Le Xeon Phi est conçu pour tirer parti de plusieurs *threads* par cœur afin de cacher les latences. Avec un usage *monothread*, il est probable que les performances soient influencées par ces latences de l'architecture Xeon Phi. En particulier au niveau des instructions de contrôle qui s'expriment différemment en scalaire et en SIMD.

Lorsque le nombre de *threads* exécutés sur 1 cœur est passé à 4, le gain des instructions SIMD devient ×15.8. On constate également que le passage de 1 à 4 *threads* permet d'obtenir des gains globaux de respectivement ×2.3 et ×2.1. Sur la version tirant parti des 228 *threads* / 57 cœurs du Xeon Phi, on obtient une performance de × 15.3 correspondant à une efficacité de 96% grâce à l'utilisation d'instructions SIMD KNC. En comparant les performances des versions 4 et 228 *threads*, que ce soit les versions scalaires entre-elles ou Boost.SIMD entre-elles, on observe la bonne utilisation des 57 cœurs du Xeon Phi. La version scalaire présente un gain de × 56.1 soit une efficacité de 99% au regard du nombre de cœurs. La version Boost.SIMD présente une accélération de × 54.3 soit une efficacité de 93%.

On peut donc en conclure que sur un algorithme avec une bonne intensité arithmétique, les différents niveaux de parallélisation sont bien exploités (passage à l'échelle sur les cœurs et les registres SIMD).

6.1.2.2 Étape 1.2 : Caractéristiques des pinceaux

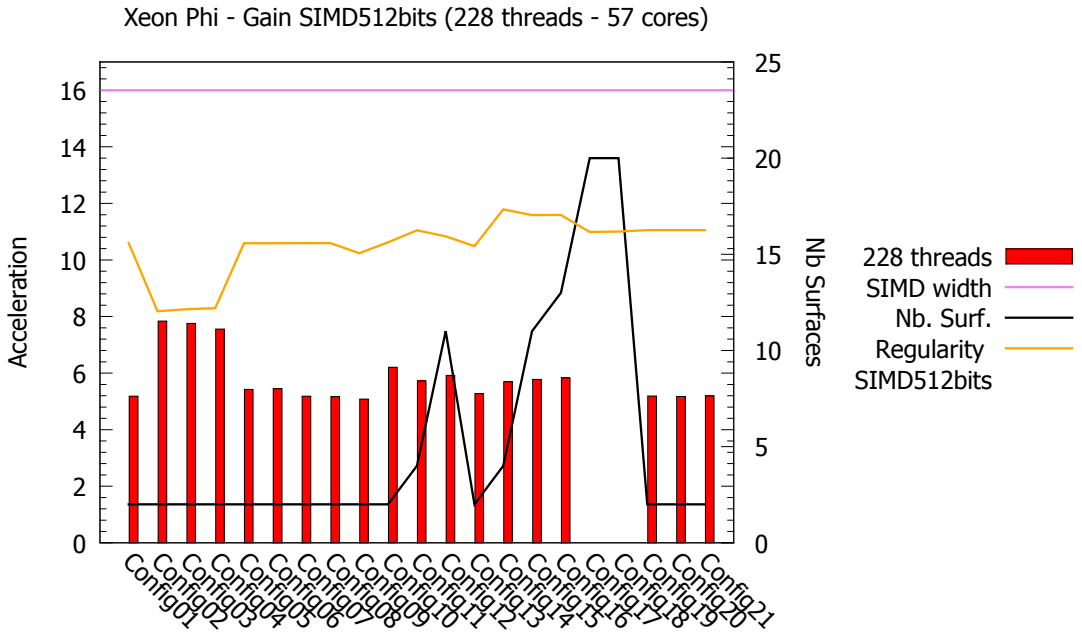


FIGURE 6.5 – Etape 1 : SIMD KNC 512 bits sur Xeon Phi

Puisqu'il n'est pas possible de découpler suffisamment le calcul des trajets avec le calcul des caractéristiques des pinceaux, il convient d'étudier l'impact de la *SIMDization* sur le calcul des pinceaux dans leur intégralité.

L'usage des instructions SIMD KNC 512 bits (16 flottants simple précision) sur l'architecture Xeon Phi permet d'obtenir une accélération conséquente des performances, de l'ordre de $\times 5.8$ en moyenne, comme observé sur la figure 6.5. En rouge figure l'accélération obtenue à l'aide des instructions SIMD KNC 512 bits, par rapport au gain théorique en violet (cardinal du registre SIMD). Cette accélération est plus proche des accélérations obtenues sur des architectures supportant les instructions AVX et/ou AVX2-FMA telles que les GPP Haswell et Ivy Bridge, que du gain théorique des instructions SIMD.

L'hypothèse suivante peut-être avancée pour expliquer cette baisse d'efficacité, par rapport au quasi $\times 16$ observé sur le *minibenchmark*. Comme pour les instructions AVX sur GPP, certaines instructions SIMD KNC 512 bits les plus complexes ne passent pas à l'échelle par rapport à leur version scalaire, y compris sur des GPP récents. Toutefois, Intel ne communique pas encore les débits et latences des instructions sur Xeon Phi. Un *microbenchmark* de chacune des instructions mises en jeu sur le code de calcul des pinceaux permettrait d'estimer ces valeurs. Dans le cadre de ces travaux de thèse, il n'a pas été réalisé pour des raisons de "rentabilité". Le gain obtenu est déjà appréciable sur les performances des simulations de champ. De plus un tel *microbenchmark*

n'a que de très faibles chances d'apporter des pistes pour l'évolution du code en vue d'obtenir des accélérations sensibles.

6.1.3 Étape 2 - Mesure de la taille des signaux

Pour les mêmes raisons que sur GPP, puisque cette étape n'est pas prépondérante dans le calcul de champ, elle n'a pas bénéficié d'une *SIMDization*.

6.1.4 Étape 3 - Traitement du signal

L'étape 3 du calcul de champ, comportant les sous-étapes 3.1 et 3.2, a bénéficié des accélérations apportées sur GPP. L'étape 3.1 de sommation des contributions des pinceaux sur les signaux de déplacement a été optimisée afin d'utiliser une sommation scalaire différentielle plutôt qu'une implémentation *SIMDizée*. Cette optimisation semble encore plus se justifier dans le cadre du Xeon Phi : les registres sont plus larges, alors que le jeu de données est toujours aussi peu régulier.

Concernant l'étape 3.2, qui réalise les traitements du signal, toutes les versions du code implémentées se basent sur la bibliothèque Intel MKL pour les FFT. Les autres opérations sur les signaux de déplacement (convolution, recherche de maximum...) ont quant à elles bénéficié des instructions SIMD KNC (à travers Boost.SIMD).

Les performances de ces différentes implémentations sur l'ensemble des configurations de champ sont rappelées en annexe à la table 7. Il en ressort que sur architecture Xeon Phi, les gains obtenus par l'implémentation de la sommation différentielle sont moins spectaculaires que sur GPP. Ainsi l'accélération observée se trouve entre $\times 1.0$ (pas d'accélération) et $\times 1.6$ sur l'ensemble des configurations. Le Xeon Phi disposant d'une bande passante supérieure au GPP, l'implémentation "naïve" de la sommation scalaire s'en trouve moins influencée. On peut remarquer que sur les configurations où peu de pinceaux contribuent aux signaux (élimination géométrique...), les performances de la sommation différentielle sont égales à celles de la sommation scalaire. Comme sur GPP, lorsqu'il n'y a pas d'accès mémoire pour ajouter une contribution, une version SIMD régulière afficherait une plus grosse accélération mais l'élimination des pinceaux n'est pas ce que l'on souhaite optimiser !

Enfin, l'usage des instructions vectorielles sur la partie 3.2 du calcul de champ permet d'obtenir un facteur d'accélération compris entre $\times 1.0$ et $\times 1.3$ (pour rappel, la référence mesurée utilise déjà les instructions SIMD via la MKL). Pour conclure, l'optimisation de cette étape 3 du calcul de champ (sommation différentielle et *SIMDization* du traitement du signal) permet d'obtenir un facteur d'accélération se trouvant dans une fourchette de $\times 1.0$ à $\times 2.2$.

6.2 Synthèse des accélérations obtenues

Afin de mesurer l'impact global des accélérations obtenues, cette section présente les performances mesurées sur l'ensemble du calcul de champ pour l'architecture Xeon Phi avant et après optimisation. Les performances sur la configuration de référence (configuration 01) sont ensuite détaillées avant de conclure sur les performances absolues du Xeon Phi pour la simulation de champ.

6.2.1 Répartition des traitements

La figure 6.6 présente la répartition des étapes de calcul pour chacune des configurations. Une première colonne (ramenée à 100%) présente le temps de chacune des étapes de calcul 1 à 3.

Une seconde colonne présente en temps cumulé le coût des différentes étapes de calcul une fois celles-ci optimisées. Enfin, une dernière colonne (en noir) présente les performances de la version verticale optimisée, ramenées à la même échelle (100% représentant le temps de la version "naïve" du champ sur Xeon Phi).

Sur l'ensemble des configurations, on observe une diminution du temps de calcul dédié à l'étape 1, cependant les optimisations sur l'étape 3 sont proportionnellement moins efficaces. L'algorithme vertical est souvent le plus rapide. Mais de façon surprenante, pour certaine configurations (2, 3 et 10) ce n'est pas le cas.

Le fondeur ne fourni pas la latence des instructions du Xeon Phi. Dans la mesure où les cœurs accèdent à une mémoire partagée à travers un bus en anneau, la latence mémoire peut varier en fonction des cœurs et de la distance avec les données (leur position dans le banc mémoire). Ainsi il est possible que l'algorithmie verticale se retrouve en des cas défavorables sur ces configurations (accès aux données de configuration et de géométrie...).

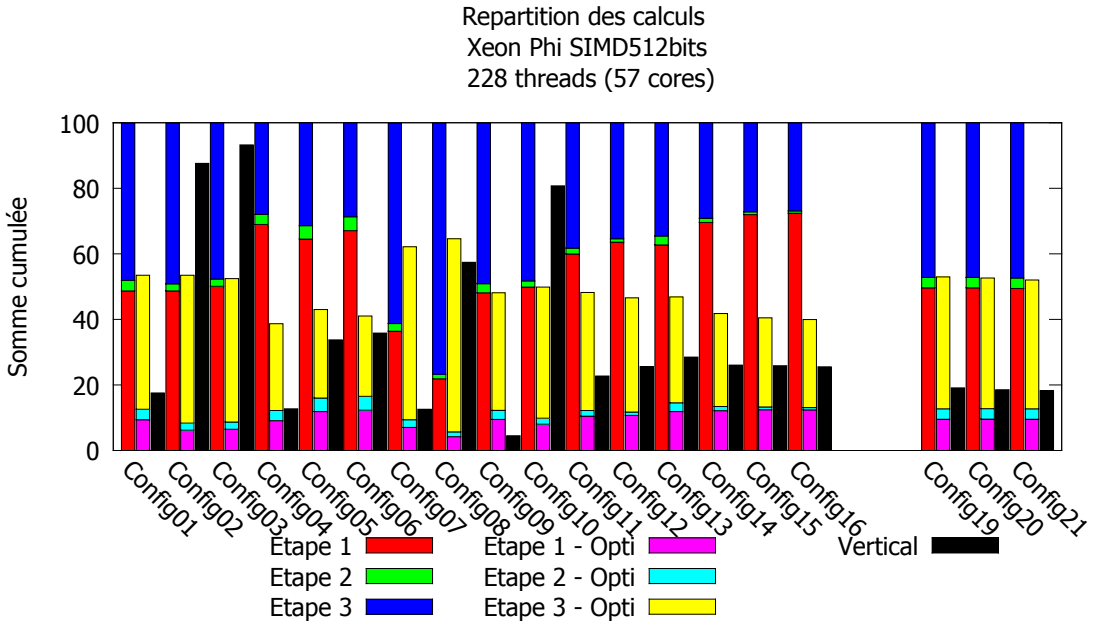


FIGURE 6.6 – Répartition des étapes de calcul de champ avant et après optimisation (sur Xeon Phi)

6.2.2 Analyse sur une configuration

La table 6.2 présente les performances optimisées des différentes étapes de calcul de champ sur architecture Xeon Phi, pour la configuration 01. Les différentes phases sont présentées avant et après optimisation, ainsi que les résultats obtenus pour la version verticale de l'algorithme regroupant les calculs en une seule boucle de haut niveau.

Horizontalement, on constate un bon passage à l'échelle de 1×4 *threads* à 57×4 *threads* pour l'étape 1 du calcul de champ. Par contre, l'*hyperthreading* n'apporte pas autant de performances sur les étapes 2 et 3, ce qui explique le faible passage à l'échelle de l'algorithme horizontal total. De manière non évidente, l'algorithme vertical permet un meilleur équilibrage de charge des

différents calculs pour différents points de champ (les opérations ne sont pas regroupées par type), l'accélération mesurée est, ×82.2, sur-linéaire par rapport au nombre de cœurs.

De leur côté, les optimisations bas niveau ont permis des accélérations substantielles telles que évoqués dans les précédentes sections de ce chapitre. Sur cette configuration, le calcul de champ atteint, en combinant les différentes optimisations, une performance de 23.2 images par seconde (soit un gain de ×242.5 par rapport à la version scalaire horizontale totale[1]).

1x XeonPhi							
SIMD 512 bits		Scalaire	SIMD	Gain SIMD	228 threads	Gain OpenMP+HT	Gain Total (par étape)
Config01	Étape 1	6738.8 ms	1189.0 ms	x5.7	23.1 ms	x51.4	x291.3
	Étape 2	373.9 ms	373.5 ms	x1.0	7.8 ms	x47.8	x47.9
	Étape 3	3338.8 ms	2515.3 ms	x1.3	147.1 ms	x17.1	x22.7
	Total	10451.4 ms	4077.8 ms	x2.6	178.1 ms	x22.9	x58.7
	Vertical	9637.8 ms	3547.5 ms	x2.7	43.1 ms	x82.2	x223.4
		0.1 fps	0.3 fps		23.2 fps		

TABLE 6.2 – Performances de la Config01 sur architecture XeonPhi

6.2.3 Synthèse sur l'ensemble des configurations

Les performances des versions horizontales et verticales optimisées pour Xeon Phi sont présentées par la table 6.3, pour l'ensemble des configurations de champ étudiées.

[1] utilisant la bibliothèque Intel MKL

	1x XeonPhi		
	SIMD	512bits	
	Horizontale	Vertical	Best FPS
Config01	131 ms	**43.1 ms**	23.2 fps
Config02	**670 ms**	1094.7 ms	1.49 fps
Config03	**637 ms**	1136.3 ms	1.57 fps
Config04	342 ms	**112.4 ms**	8.9 fps
Config05	315 ms	**247.5 ms**	4.0 fps
Config06	1151 ms	**1008.5 ms**	1.0 fps
Config07	204 ms	**41.3 ms**	24.2 fps
Config08	353 ms	**313.9 ms**	3.2 fps
Config09	244 ms	**22.9 ms**	43.6 fps
Config10	**734 ms**	1184.0 ms	1.36 fps
Config11	318 ms	**149.7 ms**	6.7 fps
Config12	1005 ms	**551.7 ms**	1.8 fps
Config13	138 ms	**83.8 ms**	11.9 fps
Config14	358 ms	**222.9 ms**	4.5 fps
Config15	1135 ms	**725.3 ms**	1.4 fps
Config16	3359 ms	**2148.0 ms**	0.5 fps
Config17	-	-	-
Config18	-	-	-
Config19	129 ms	**46.4 ms**	21.5 fps
Config20	128 ms	**45.0 ms**	22.2 fps
Config21	127 ms	**44.5 ms**	22.5 fps

TABLE 6.3 – Performances des algorithmes optimisés (approche horizontale et verticale) sur Xeon Phi sur l'ensemble des configurations (exécution sur 228 *threads*). En gras l'algorithme le plus rapide.

6.3 Conclusion

L'utilisation des accélérateurs MIC permet d'obtenir au prix d'un faible effort d'ingénierie informatique un code de calcul performant de calcul de champ à partir d'une implémentation GPP grâce à l'usage d'outils portables (OpenMP, Boost.SIMD et Intel MKL). Les performances obtenues sont à la limite de l'interactivité sur plusieurs configurations de calcul de champ avec un maximum de l'ordre de 24 images par seconde. Ainsi, sur un Xeon Phi 3120A, le prix par images/seconde de la simulation sur la configuration 01 revient à 78.9 USD/fps.

Cependant, l'usage du Xeon Phi présente encore quelques limitations pour la simulation de champ rapide :

- Sur certaines configurations, les performances de l'algorithme vertical chutent par rapport à celles de l'algorithme horizontal. Il est nécessaire, en vue de l'industrialisation, de repérer ces cas afin de sélectionner le bon algorithme (à moins d'accepter un comportement dégradé).
- Les implémentations courantes des sommations des pinceaux sont basées sur les codes SIMD du GPP. Elles ne tirent pas parti des instructions spécifiques disponibles sur Xeon Phi. Leur usage pourrait accroitre les performances de la sommation.
- Les performances des instructions SIMD KNC, pourtant sur des registres de 512 bits de large, ne sont pas satisfaisantes en l'état : les accélérations théoriques de $\times 16$ en simple précision semblent encore difficiles à atteindre pour un code de calcul "généraliste" tel que le calcul des pinceaux. Ce non passage à l'échelle des performances des instructions SIMD par rapport aux registres moins large a déjà été constaté sur processeur généraliste, *c.f.* 5.4.1.1.1, lorsque la largeur des registres SIMD est doublée au passage des jeux d'instructions SSE 4.2 vers AVX, certaines instructions voient leurs performances par élément se dégrader (par exemple une opération de division sur un registre deux fois plus large met deux fois plus de cycles à s'exécuter). Ainsi, exprimer un parallélisme de données de $\times 16$ n'est pas automatiquement gage d'une accélération d'autant, cependant par rapport aux codes scalaires il y a tout de même une accélération.

Chapitre

7 Optimisations sur architecture GPU

La dernière architecture abordée pour accélérer le calcul de champ est l'architecture GPU, plus particulièrement celles des GPU de marque nVidia. Afin de tirer parti des performances disponibles, il est nécessaire d'adapter les algorithmes du calcul de champ en une version spécifique. Cette adaptation est faite à la fois au niveau de la programmation du fait de la nécessité de coder les algorithmes qui sont déportés sur GPU à l'aide du langage CUDA, mais également au niveau algorithmique afin de répartir les calculs de la simulation de champ sur les différents multiprocesseurs du GPU

7.1 Structure de l'algorithmie GPU

Le modèle de programmation CUDA, présenté dans le chapitre 3.2.1.8, nécessite de regrouper les traitements en des noyaux de calcul exécutant massivement un algorithme. Chacun d'entre eux répartit l'ensemble des calculs qu'il réalise, par bloc, sur les différents multiprocesseurs du GPU (*c.f.* 3.1.5.2). Ainsi, l'algorithmie de la version CUDA du calcul de champ se rapproche de la version "horizontale", par étape, proposée sur GPP, correspondant à l'algorithme de référence (présenté dans les listings 7 et 8). Le calcul du champ s'effectue alors sur plusieurs points de champs en même temps. Toutes les données temporaires nécessaires au calcul, en particulier des informations de signaux de réponses impulsionnelles de déplacement, sont tenues en mémoire en même temps.

Afin d'obtenir une implémentation du calcul de champ efficace et efficiente à développer, celle-ci se base sur deux bibliothèques fournies et optimisées par nVidia.

- D'une part la bibliothèque *CUDA Fast Fourier Transform library* (cuFFT) qui permet le calcul de Transformées de Fourier Rapide sur des ensembles de signaux présentant tous les mêmes caractéristiques.
- D'autre part la bibliothèque *CUDA Basic Linear Algebra Subroutines* (cuBLAS) qui propose des routines réalisant des opérations d'algèbre linéaires classiques (BLAS 1, 2 et 3) sur des ensembles de vecteurs.

Les routines cuBLAS utilisées dans le calcul de champ sont des routines de niveau 1 qui contribuent à la construction du signal de champ.

Dans le cadre du calcul de champ, ces bibliothèques opèrent sur des tableaux contenant plusieurs signaux. Tous les signaux d'un même point de champ ont les mêmes caractéristiques de taille (en nombre d'échantillons). De plus, l'algorithme de référence travaillant par grandes étapes sur plusieurs points (algorithmie "horizontale"), les signaux de deux points présentent le même état (données complexes ou réelles, dans le domaine fréquentiel ou temporel). Les signaux partagent ainsi le même "format" pour le calcul des plans de cuFFT et cette régularité permet d'appliquer les routines BLAS nécessaires à la formation de la réponse impulsionnelle globale (addition de deux vecteurs ou mise à l'échelle d'un signal).

Pour obtenir une plus grande homogénéité du développement, les appels aux bibliothèques cuFFT et cuBLAS ont été masqués au moyen d'une classe spécifique gérant des "paquets" de

plusieurs signaux d'un nombre d'échantillons temporels donné. Elle permet, entre autre, l'utilisation d'un ensemble de plans de FFT uniques en mémoire pour les signaux de même type. Cette classe fait elle-même appel à une sous-classe personnalisée associant les pointeurs d'accès aux données en mémoire hôte avec leurs correspondants vers la mémoire GPU (et les routines d'allocation/de libération mémoire).

Le fonctionnement de CUDA impose à l'hôte de gérer l'exécution des noyaux de calcul, les appels aux bibliothèques et la gestion de la mémoire globale du GPU. Il est également à noter que les données correspondant à une configuration donnée sont bien trop volumineuses pour résider dans la mémoire constante du GPU, large de 64 kilo-octets. Celle-ci est utilisée au maximum en comptant les informations de géométrie du contrôle (surfaces et milieux). Le reste des données est stocké en mémoire globale.

Par contre, comme sur architecture MIC, les tailles des données résultant du calcul de champ sont très faibles devant la bande passante du GPU vers l'hôte. La problématique du temps des transferts mémoire ne sera pas abordée. Ceux-ci pourront être masqués par le calcul de l'image suivante dans le cadre d'une simulation interactive. Les données intermédiaires du calcul de champ résident quant-à-elles dans la mémoire globale du *device*. Mais comme leur durée de vie est restreinte au temps du calcul, elles sont désallouées au fur et à mesure que le calcul progresse.

7.2 Noyaux de calcul

Afin de profiter de la régularité des calculs entre points de champs voisins, les noyaux créés correspondent aux étapes du calcul de champ précédemment définies. Cette section les présente brièvement et détaille leurs particularités.

7.2.1 Étape 1 : Calcul de pinceaux

Lors de cette étape, les noyaux regroupent les étapes 1.1 (calcul du trajet) et 1.2 (caractéristiques du pinceau) dans un seul et même noyau de calcul. Cependant, chacun des modes possibles du trajet de l'onde entre le capteur et le point de champ est traité par un noyau de calcul différent. Cette découpe permet un travail par mode, rendant les calculs pour tous les points de champ systématiques par mode (par opposition au GPP où pour chaque point de champ, tous les modes sont calculés au sein de la même boucle). Un premier noyau est dédié aux trajets en mode direct et un second est dédié aux trajets avec rebond sans conversion de mode. Ainsi, pour une simulation demandant les modes d'onde L direct et L rebond, chacun des noyaux sera appelé pour l'ensemble des points de champ.

Les *threads* CUDA travaillent chacun sur un pinceau à la fois, au moyen d'une linéarisation de l'espace des pinceaux possibles. Afin de bénéficier du rangement des données capteur sous la forme de structure de tableau, les tâches de calcul sont divisées en blocs CUDA selon les points de champ et les *threads* CUDA sont disposés de manière à accéder aux échantillons du capteur de manière contiguë et cohérente. Les *warps* CUDA accèdent ainsi à la mémoire globale de manière cohérente pour lire les informations capteur et également lors de la sauvegarde des caractéristiques du pinceau avec un même schéma.

La seule cause réelle de divergence qui peut survenir lors du calcul est liée à des différences sur le nombre d'itérations nécessaires pour la résolution de polynômes par la méthode de Newton. Le pire cas étant un *thread* nécessitant plus d'itérations que les 31 autres de son *warp*. Comme avec les instructions SIMD, tous doivent alors attendre.

Les calculs des coefficients de Fresnel sont source de divergence en fonction de l'angle d'incidence, cependant les blocs travaillent pour un même point de champ : les pinceaux sont globalement, la plupart du temps, incidents avec un angle de même sorte (sur- ou sous-critique).

Par contre, si les tests de validité du trajet indiquent qu'un pinceau doit être défaussé, le *thread* entre dans une branche divergente n'exécutant aucune opération. Cela ne présente pas de surcoût pour le reste du *warp*, il y aura juste une perte d'efficacité.

7.2.2 Étape 2 : Recherche de la taille des signaux

Cette étape consiste à parcourir l'ensemble des pinceaux retenus pour contribuer au champ et l'étalement temporel des contributions de ces pinceaux en tenant compte des lois de retards appliquées. Elle se traduit en deux noyaux CUDA : le premier réalise la mesure de cet étalement temporel point de champ par point de champ, le second procède à une réduction de ces informations pour obtenir la valeur globale du nombre d'échantillons nécessaire. L'usage de deux noyaux spécifiques par mode permet d'une part d'obtenir un calcul systématique et sans distinction de mode pour le premier noyau et d'autre part de faciliter la synchronisation. Mais les deux noyaux faisant partie du même CUDA *stream*, il n'y a pas de recouvrement de l'un par l'autre même sur les GPU le supportant. La fin du calcul du premier est la barrière de synchronisation qui déclenche le second.

Le premier noyau fonctionne de manière analogue aux noyaux de calcul des pinceaux de l'étape 1, mais procède de manière plus globale et sans distinction des modes. Les blocs CUDA sont distribués sur les points de champ et les *threads* parcourent linéairement l'espace des pinceaux à mesurer. Chaque bloc est dédié aux pinceaux d'un seul point de champ. Enfin, au moyen d'opérations atomiques minimum et maximum, les *threads* du bloc collaborent pour obtenir les bornes supérieures et inférieures de l'étalement temporel total des pinceaux d'un point de champ donné. Afin de pouvoir communiquer ces informations au noyau suivant, celles-ci sont enregistrées en mémoire globale à la fin de l'exécution de chaque bloc.

Le second noyau ne requiert qu'un seul bloc CUDA. Ses *threads* parcourent les informations d'étalement temporel de tous les points de champs et obtiennent une information partielle, propre à chaque *thread*. Une opération atomique de recherche de maximum permet de récupérer par réduction la valeur du nombre d'échantillons nécessaires pour l'ensemble des points de champs de la configuration.

Une fois cette valeur obtenue sur le GPU, la donnée peut être copiée vers l'hôte pour lui permettre d'effectuer l'allocation et l'initialisation à zéro des signaux utiles au calcul de champ.

7.2.3 Étape 3 : Traitement du signal

L'étape 3 est décomposée en une multitude de noyaux de calcul différents, afin de permettre à l'hôte d'insérer les appels aux bibliothèques externes nécessaires. C'est, en particulier, le cas de l'étape 3.2 qui nécessite beaucoup de Transformées de Fourier Rapides.

7.2.3.1 Étape 3.1 : Sommation des contributions de chaque pinceau

En termes de découpage de l'espace de calcul, le noyau réalisant la sommation des contributions des pinceaux se comporte de la même façon que le noyau de calcul de l'étalement temporel des pinceaux. A chaque point de champ correspond un bloc CUDA et au sein de ce bloc les *threads* travaillent sur l'ensemble des pinceaux disponibles. Ces *threads* calculent, à partir des pinceaux et de la loi de retards en vigueur, la contribution du pinceau suivant les trois composantes complexes du déplacement et ajoutent aux signaux de réponse impulsionnelle le créneau d'amplitude correspondant. Pour éviter les conflits d'accès entre les *threads*, ces sommations ont lieu au moyen d'opérations atomiques.

7.2.3.2 Étape 3.2 : Extraction du maximum d'amplitude du signal de déplacement

Cette étape du calcul est une succession d'appels à des noyaux spécifiques et d'appels à des bibliothèques externes.

7.2.3.2.1 Convolution avec le signal de référence Les contributions des pinceaux sont sommées en 6 signaux (3 composantes réelles et 3 composantes imaginaires), formant la réponse impulsionnelle en déplacement. Afin de calculer la convolution dans le domaine fréquentiel entre la réponse impulsionnelle de déplacement et le signal de référence, cette dernière est tout d'abord transformée par la bibliothèque cuFFT (sur chacune de ses composantes).

Le noyau spécifique à la convolution est construit de manière à ce que les blocs de calcul CUDA se répartissent des signaux différents, un signal par bloc. Le bloc réalise à la fois la convolution de la partie réelle de la réponse impulsionnelle et la Transformée de Hilbert de la partie imaginaire. Ainsi, ses *threads* peuvent calculer en tous points du signal la valeur correspondante du déplacement, en travaillant au niveau d'un emplacement mémoire.

La bibliothèque cuFFT est alors à nouveau utilisée pour repasser dans le domaine temporel. Cependant, le calcul nécessite l'utilisation de cuBLAS pour remettre à l'échelle les échantillons temporels. En effet, cuFFT ne remultiplie pas automatiquement les signaux calculés par l'inverse de la longueur des signaux lorsqu'une FFT inverse est demandée.[1]

7.2.3.2.2 Calcul du Module de déplacement Le calcul du module de déplacement est obtenu de manière systématique, en appliquant la norme 2 sur l'ensemble des signaux.

7.2.3.2.3 Calcul de l'enveloppe du module de déplacement Le calcul de l'enveloppe fait à nouveau appel à la bibliothèque cuFFT pour passer du domaine temporel au domaine fréquentiel et inversement.

Le noyau de calcul spécifique au calcul de l'enveloppe associe un bloc à chaque point de champ. Les *threads* CUDA travaillent en parallèle pour calculer la nouvelle valeur de chaque échantillon du signal (doublement de l'amplitude des fréquences positives, mise à zéro des fréquences négatives).

7.2.3.2.4 Extraction du maximum d'amplitude et du temps de vol correspondant Enfin, le dernier noyau de calcul parcourt les informations de l'enveloppe du déplacement pour en obtenir le maximum. Un bloc CUDA traite un seul point de champ ; les *threads* CUDA associés parcourent les échantillons du signal de déplacement correspondant. Par réduction en mémoire partagée, le maximum de déplacement est obtenu ainsi que l'indice de l'échantillon correspondant. Le *thread* 0 écrit alors en mémoire globale, dans les images de champ, l'amplitude maximum du déplacement et le temps de vol associé.

7.3 Algorithmie générale

Cette section récapitule brièvement l'ensemble des noyaux développés et leur enchaînement.

[1] Avec cuFFT, `FFT_C2R(FFT_R2C(signal))` a une amplitude de `1/N` fois le signal initial, avec `N` la longueur du signal.

7.3.1 Récapitulatif des noyaux développés

- **Étape 1**
  - Calcul des pinceaux - mode direct
  - Calcul des pinceaux - mode rebond sans conversion
- **Étape 2**
  - Étalement temporel des signaux, point par point
  - Calcul de la taille des signaux, en nombre d'échantillons par réduction
- **Étape 3.1**
  - Sommation des contributions des pinceaux, point par point
- **Étape 3.2**
  - Convolution (et Transformée de Hilbert) dans le domaine fréquentiel
  - Calcul du module de déplacement (3 composantes)
  - Calcul de l'enveloppe dans le domaine fréquentiel
  - Extraction du maximum

7.3.2 Algorithme général et synchronisations

La figure 7.1 présente l'enchaînement algorithmique des différents noyaux CUDA lors du calcul de champ sur GPU. Bien que l'hôte soit celui qui dirige et ordonne les noyaux de calcul, toute la simulation de champ se déroule sur le GPU : les noyaux sont empilés dans une file par l'hôte de bout en bout. Il n'y a qu'entre les étapes 1 et 2 qu'il est nécessaire à l'hôte de reprendre la main. Une fois la taille des signaux déterminée, le GPP est utilisé pour allouer les signaux des réponses impulsionnelles en déplacement qui vont être nécessaires au calcul de champ et à initialiser les plans des FFT à l'aide de la bibliothèque cuFFT.

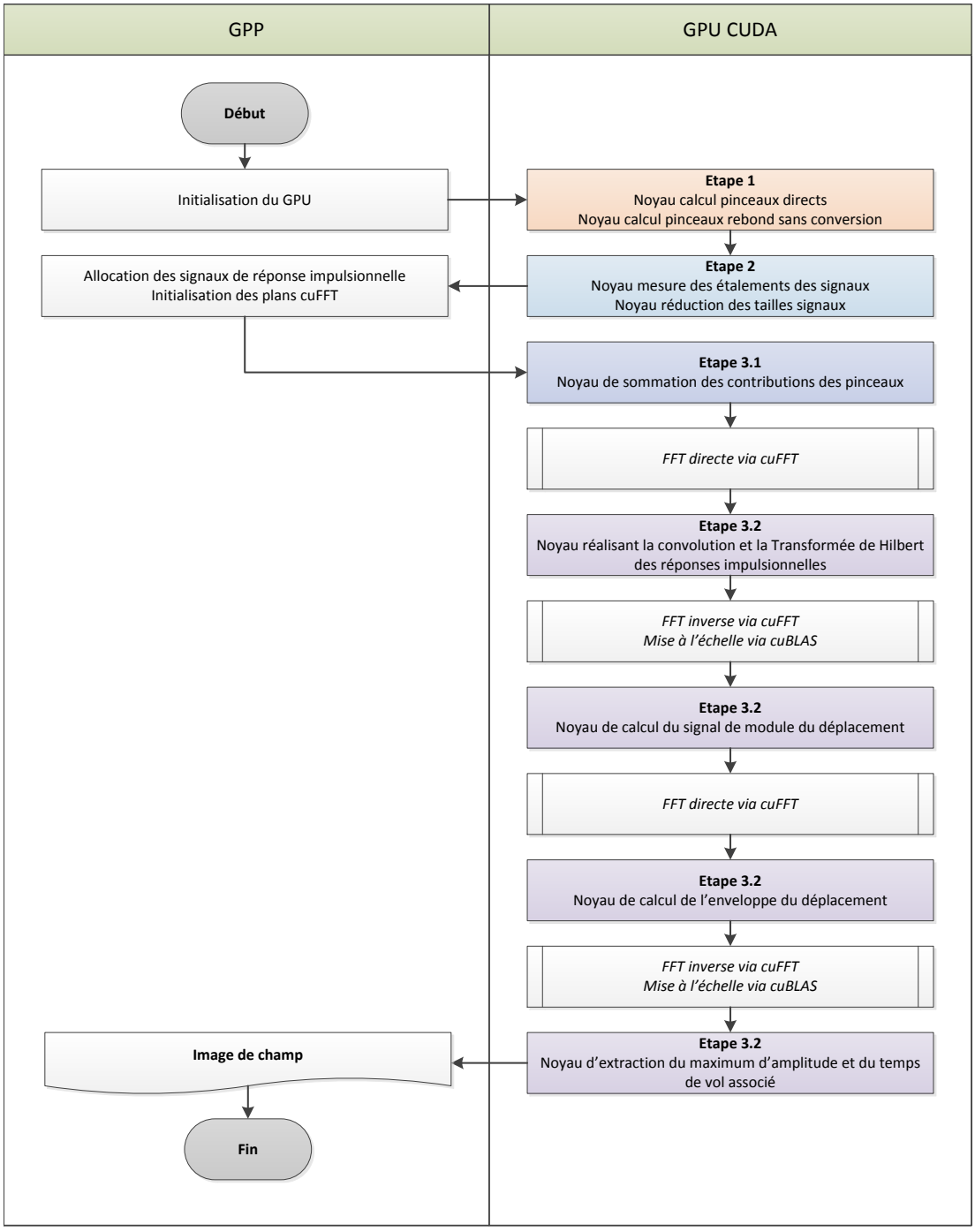


FIGURE 7.1 – Principe algorithmique du calcul de champ sur GPU et enchaînement des noyaux de calcul CUDA

7.4 Paramétrage des noyaux de calcul

Le langage CUDA permet d'influer sur les performances d'un même noyau de calcul à plusieurs niveaux en fonction des paramètres de sa compilation et de son exécution (découpe de l'espace en grille de blocs, registres alloués par *thread*, besoins en mémoire déclarés). Dans un premier temps, cette section revient brièvement sur des considérations théoriques afin de sélectionner ces paramètres, en particulier pour ne pas limiter les performances. Dans un second temps, des considérations pratiques sont abordées notamment, l'exploration exhaustive des combinaisons de ces paramètres afin d'en sélectionner les valeurs les plus pertinentes pour la performance.

7.4.1 Théorie

Le premier paramètre sur lequel il est possible d'influer se décide à la compilation. Il s'agit du nombre maximum de registres dont chaque *thread* CUDA dispose pour s'exécuter. L'une des méthodes pour fixer cette limite consiste à la spécifier par unité de compilation (par un simple paramètre de ligne de commande). C'est ce qui a été choisi ici afin de faciliter en pratique le parcours systématique présenté à la section 7.4.2.

Les données techniques des GPU en fonction de leur génération (*Compute Capability*) sont présentées dans le tableau 7.1, en particulier la quantité de registres que doivent se partager tous les blocs résidents sur un même multiprocesseur. Le taux d'occupation théorique du GPU est appelé, dans le jargon CUDA, par sa désignation anglaise *Occupancy*. Des outils existent qui permettent d'obtenir une information théorique de l'*Occupancy* pour une combinaison de ressources demandées. Ces outils utilisent le ratio du nombre de *warps* qu'il sera possible de faire résider sur le GPU par rapport à sa capacité maximale tout en respectant les diverses limites présentées par la table 7.1. Cependant, cette valeur n'est pas non plus un critère de bonne performance, elle ne prend en compte ni les performances effectives des calculs ni les problèmes liés aux accès mémoire. Elle sert couramment à mettre en lumière les limites d'un paramétrage donné. Inversement, une valeur d'*occupancy* trop faible indique que le noyau n'aura pas d'alternative pour masquer les problèmes de latence s'ils surviennent, ce qui en fait cependant un bon indicateur de non performance [Vol10].

Minimiser le nombre de registres nécessaire par *thread* permet de loger plus de *warps* sur un même multiprocesseur du GPU. Inversement, en autorisant un maximum de registres par *thread*, le multiprocesseur risque d'être sous utilisé. Toutefois, réduire le nombre de registres n'est pas non plus gage de performance. Plus l'algorithme est complexe et plus le compilateur est tenté, pour réduire la pression sur les registres, de stocker des informations en mémoire locale[2]. Ce phénomène s'appelle *register spilling*. Il est bien souvent nécessaire de trouver un point d'équilibre pour le paramétrage du nombre de registres par *thread* autorisés.

Le second paramètre est fixé au moment de l'exécution, en sélectionnant le nombre de blocs CUDA et le nombre de *threads* les composant[3]. Ainsi, si un noyau est mal paramétré, un multiprocesseur peut être saturé et ne pas être exploité au maximum. Réciproquement, si le nombre de blocs est trop faible, c'est le GPU dans son ensemble qui risque d'être sous utilisé.

Ne pouvant pas baser le paramétrage des noyaux de calcul du champ sur une valeur théorique formelle, il est réalisé empiriquement. Afin d'explorer l'ensemble des paramétrages des noyaux, le paragraphe suivant effectue une mesure des performances pour chacune des combinaisons.

[2] La mémoire locale a un coût d'accès beaucoup plus lent qu'un registre, puisqu'il s'agit d'un accès particulier à la mémoire globale du GPU, comme précisé au paragraphe 3.1.5.2

[3] La mémoire partagée est également paramétrable à l'exécution, mais elle n'intervient pas dans le paramétrage des noyaux du calcul de champ

7.4.2 Pratique

Puisque la majorité des noyaux n'utilise pas la mémoire partagée du GPU, les noyaux sont exécutés pour un ensemble de couples de valeurs (nombre de registres par *thread*, nombre de *threads* par bloc). Ces noyaux sont ensuite exécutés sur des GPU de deux familles différentes, Fermi et Kepler, pour l'ensemble de ces couples de paramètres.

GPU Architecture *Compute Capability*	GeForce 580 Fermi 2.0	Tesla C2070 Fermi 2.0	GeForce Titan Kepler 3.5
Nombre de registres (total)	32K	32K	64K
Nombre de registres maximum par *thread*	63	63	255
Nombre maximum de *threads* par bloc	1024	1024	1024
Nombre maximum de *threads* par multiprocesseur	1536	1536	2048
Nombre maximum de *warps* par multiprocesseur	48	48	64
Nombre maximum de blocs par multiprocesseur	8	8	16
Mémoire partagée totale par multiprocesseur	48 Ko	48 Ko	48 Ko
Mémoire partagée totale par bloc	48 Ko	48 Ko	48 Ko

TABLE 7.1 – Spécifications techniques - Limites pour le calcul de l'*Occupancy*

Source : nVidia - CUDA Programming Guide

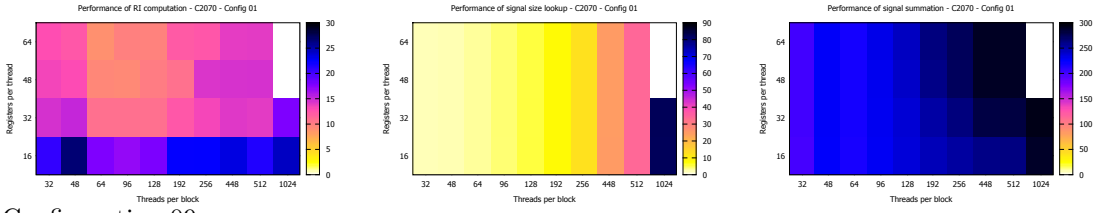
A partir des données techniques, rappelées dans le tableau 7.1, il est possible de définir les bornes dans lesquelles faire varier le nombre de registres par *thread* et le nombre de *threads* par bloc. Ces valeurs sont présentées dans le tableau 7.2. Puisque les cartes GTX 580 et C2070 sont de même génération (*Compute Capability*), elles disposent du même nombre de registres et de *threads* par multiprocesseurs, les résultats et conclusions sont interchangeables. Il faut noter que le nombre de registres par *thread* est une valeur maximum que le compilateur ne peut pas dépasser. Suivant ses algorithmes d'optimisation il peut tout à fait décider d'en utiliser moins. Cela explique pourquoi, sur les noyaux les plus simples, la performance n'évolue plus lorsque seule la limite de registre augmente.

Compute Capability	2.0	3.5
Nombre de registres par *thread*	[16 ;32 ;48 ;63]	[16 ;32 ;48 ;64 ;78 ;80 ;112 ;128 ;196 ;255]
Nombre de *threads* par bloc	[32 ;48 ;64 ;98 ;128 ;192 ;256 ;448 ;512 ;1024]	

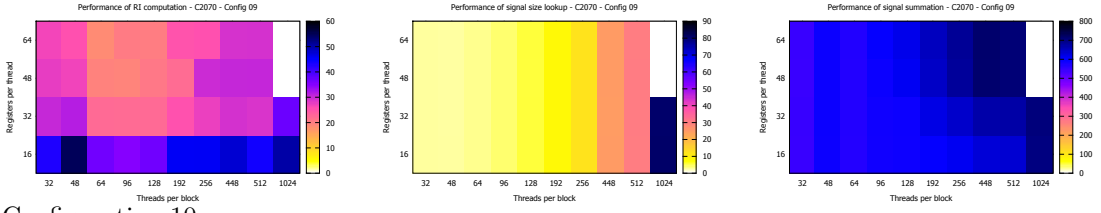
TABLE 7.2 – Paramètres explorés

Ces explorations ont été réalisées sur les configurations 1, 9 et 10 pour lesquelles le nombre de modes de champ simulés varie. Cela implique notamment qu'il y a plus de pinceaux à calculer et que les signaux de déplacement sont plus longs à cause des différences de vitesses des ondes L et T . . . Les performances associées sont réunies dans des cartes de chaleur où la couleur représente le temps d'exécution, en ms.

Performances des noyaux des étapes 1, 2 et 3.1 (par colonne) sur Tesla C2070 (Fermi)
Configuration 01



Configuration 09



Configuration 10

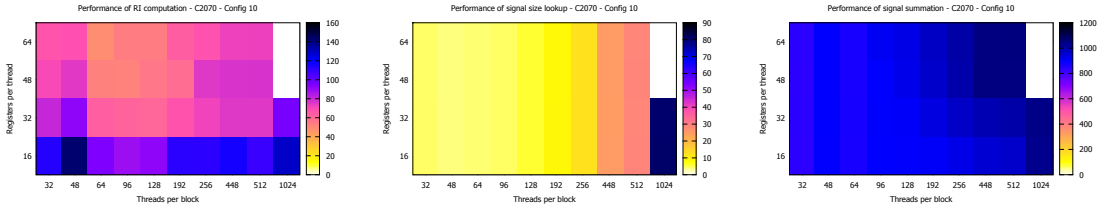
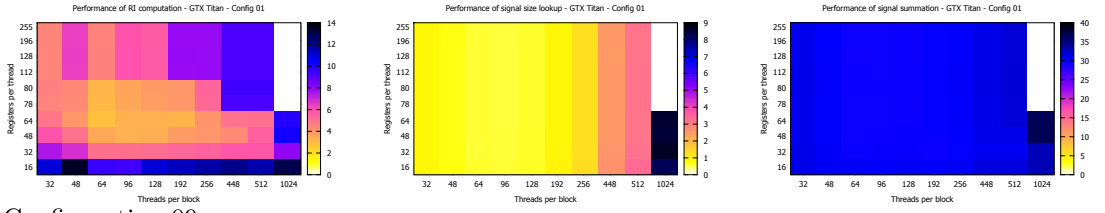
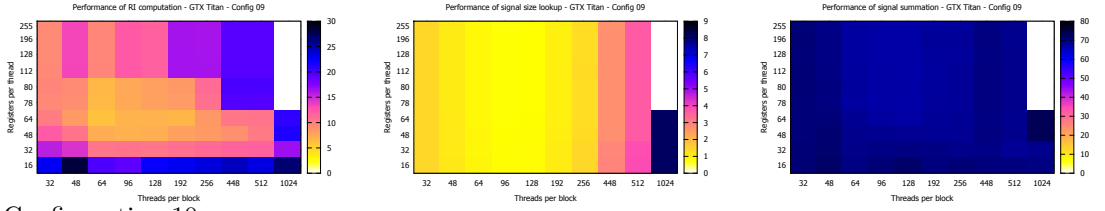


FIGURE 7.2 – Cartes de chaleur en fonction du nombre de registres par *thread* CUDA et du nombre de *threads* par bloc. Tesla C2070 (Fermi). La gradation de chaleur indique les performances obtenues pour chaque couple de paramètres (taille des blocs / nombre de registres maximum) utilisé comme coordonnées. Les performances vont du plus rapide (jaune) au plus lent (bleu foncé).

Performances des noyaux des étapes 1, 2 et 3.1 (par colonne) sur GTX Titan (Kepler)
Configuration 01



Configuration 09



Configuration 10

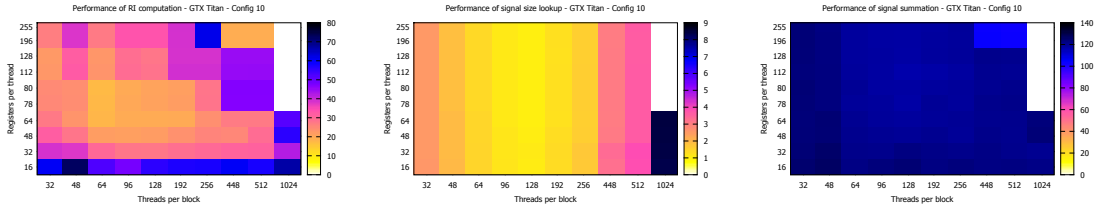


FIGURE 7.3 – Cartes de chaleur en fonction du nombre de registres par *thread* CUDA et du nombre de *threads* par bloc. GTX Titan (Kepler). La gradation de chaleur indique les performances obtenues pour chaque couple de paramètres (taille des blocs / nombre de registres maximum) utilisé comme coordonnées. Les performances vont du plus rapide (jaune) au plus lent (bleu foncé).

A partir des figures 7.2 et 7.3 qui présentent les cartes de chaleur pour différentes configurations de champ et différents noyaux CUDA, on peut déterminer les valeurs optimales des paramètres. Les temps d'exécution vont du jaune (rapide) au bleu foncé (lent). On y cherche les coordonnées correspondant aux paramètres permettant aux noyaux de s'exécuter le plus rapidement possible. L'allure de ces cartes est stable par famille de GPU. On observe que sur l'ensemble des configurations étudiées, les maximum locaux sont, par noyau et par GPU, dans les mêmes régions des cartes. Un paramétrage maximal obtenu à partir d'une configuration est bon, voire maximal, sur les autres configurations de champ. Cela permet de s'abstraire des erreurs relatives de mesure dues aux proximités en temps de calcul de paramètres proches.

Il apparait ainsi, à l'examen des figures de la première colonne que les performances du calcul des pinceaux sont mauvaises lorsque le nombre de registres est limité à 16 par *thread*. Il y a *register spilling*. Le calcul est complètement noyé dans les accès à la mémoire locale utilisés pour désengorger les registres et respecter la contrainte. Inversement, lorsque le paramétrage est de 1024 *threads* par bloc, les performances ne sont pas optimales non plus, le nombre de *warps* résidents sur un multiprocesseur est limité. De plus, avec 1024 *threads* par bloc et un grand nombre de registres, le code ne peut pas être exécuté car il aurait besoin de trop de registres par multiprocesseur au total ; ces configurations apparaissent en blanc sur les cartes de chaleur indiquant une absence de données. Les optimums locaux dépendent du noyau considéré et de la

génération de GPU visée.

A partir de ces cartes de chaleur, on fixe pour la version "performances" du calcul de champ décrite ci-après, les paramètres de nombre de registres par *threads* et le nombre de *threads* par bloc CUDA. Le nombre maximal de registres par *threads* est une consigne donnée au compilateur, il peut décider d'en utiliser moins sur un noyau donné. Par contre, la découpe en grille de blocs et la quantité de *threads* par bloc est une contrainte "dure". Le découpage de l'espace de calcul s'effectue selon cette géométrie.

Le tableau 7.3 présente les valeurs des paramètres retenues par noyau. Il s'agit du nombre effectif de registres utilisés tel qu'indiqué par le compilateur NVCC (le maximum imposé se trouve entre parenthèses).

Certains chiffres du tableau peuvent apparaître surprenants, ainsi, pour un GPU de *Compute Capacity* 3.5, l'optimum du noyau 3.1 est atteint pour un total de 128 registres par *thread* maximum. Cela s'explique par une imprécision de la mesure. Dès que la limite du nombre de registres par *thread* est suffisamment grande, le compilateur n'assigne pas plus de registres par *thread* que nécessaire et il génère toujours le même code. Par exemple, 54 pour le noyau 3.1 est la quantité nécessaire au compilateur. Puisque les exécutables sont les mêmes, les différences en temps de calcul entre ces jeux de paramètres proches sont minimes, ce qui explique la présence d'une valeur théorique plutôt importante par rapport à la quantité effective de registres par *thread* allouée (128 théoriques pour 54 alloués, alors que les valeurs 64 et 96 ont été testées).

Le tableau 7.3 présente à part les valeurs correspondantes pour deux noyaux, l'un réalisant le calcul d'une convolution et Transformée de Hilbert, l'autre réalisant le calcul de l'enveloppe. Comme ces noyaux n'effectuent aucun branchement conditionnel en fonction des données, hormis les tailles de signaux, les entrées n'influent pas sur les performances. Ainsi les valeurs ont été obtenues par profilage spécifique sur des ensembles de signaux aléatoires mesurés hors du calcul de champ.

Compute Capability	2.0		3.5	
Noyau	registres/*thread*	*threads*/bloc	registres/*thread*	*threads*/bloc
Paramètres obtenus par exploration				
1. Calcul des pinceaux (2 noyaux)	63 (64)	64	64	512
2. Mesure des signaux temporel	11 (32)	32	8 (196)	96
2. Réduction de la taille	19 (32)	32	13 (198)	96
3.1 Sommation des contrib.	15 (16)	32	54 (128)	512
3.2 Extraction du maximum	13 (16)	32	17 (128)	512
3.2 Module du déplacement	15 (16)	32	18 (128)	512
Noyaux optimisés par profilage spécifique				
3.2 Convolution + Hilbert	18	512	21	512
3.2 Enveloppe	11	512	11	512

TABLE 7.3 – Paramètres retenus pour les noyaux de calcul de champ

7.5 Analyse des performances

Les expériences précédentes ont permis d'obtenir l'implémentation du calcul de champ ultrasonore sur GPU la plus adaptée à un matériel de *Compute Capability* donné.

7.5.1 Performances globales

Le tableau 7.4 récapitule les performances obtenues sur trois GPU différents pour les différentes configurations de calcul de champ étudiées dans le cadre de ces travaux (leur description est

Performances	GTX 580 Fermi		Tesla C2070 Fermi		GTX Titan Kepler	
Configuration	Temps de calcul	FPS	Temps de calcul	FPS	Temps de calcul	FPS
Config01	153.4 ms	6.5 fps	210.4 ms	4.8 fps	34.8 ms	**28.7** fps
Config02	301.6 ms	3.3 fps	411.2 ms	2.4 fps	53.1 ms	18.8 fps
Config03	476.3 ms	2.1 fps	650.9 ms	1.5 fps	80.9 ms	12.4 fps
Config04	711.6 ms	1.4 fps	955.2 ms	1.1 fps	95.4 ms	10.5 fps
Config05	-	-	825.7 ms	1.2 fps	134.6 ms	7.4 fps
Config06	-	-	-	-	376.3 ms	2.7 fps
Config07	302.6 ms	3.3 fps	417.2 ms	2.4 fps	63.7 ms	15.7 fps
Config08	-	-	918.8 ms	1.1 fps	129.7 ms	7.7 fps
Config09	413.1 ms	2.4 fps	564.3 ms	1.8 fps	78.5 ms	12.7 fps
Config10	-	-	886.9 ms	1.1 fps	144.3 ms	6.9 fps
Config11	234.0 ms	4.3 fps	326.1 ms	3.1 fps	62.1 ms	16.1 fps
Config12	-	-	479.4 ms	2.1 fps	120.7 ms	8.3 fps
Config13	187.4 ms	5.3 fps	252.7 ms	4.0 fps	30.7 ms	**32.6** fps
Config14	199.2 ms	5.0 fps	274.6 ms	3.6 fps	48.8 ms	20.5 fps
Config15	-	-	438.0 ms	2.3 fps	110.8 ms	9.0 fps
Config16	-	-	827.4 ms	1.2 fps	251.5 ms	4.0 fps
Config17	-	-	-	-	-	-
Config18	-	-	-	-	-	-
Config19	135.8 ms	7.4 fps	187.6 ms	5.3 fps	31.9 ms	**31.4** fps
Config20	130.6 ms	7.7 fps	180.5 ms	5.5 fps	31.9 ms	**31.3** fps
Config21	128.2 ms	7.8 fps	177.1 ms	5.7 fps	32.0 ms	**31.4** fps

TABLE 7.4 – Résultat des optimisations sur GPU - Performances en ms et en fps

donnée au paragraphe 4.3.5). Tout d'abord, certaines configurations trop gourmandes en mémoire ne peuvent s'exécuter sur tous les GPU en raison de l'approche globale des accès mémoire de l'implémentation. En effet, pour rentabiliser les échanges hôte/GPU, les données temporaires nécessaires à la réalisation de l'étape 3 de sommation sont allouées sur l'ensemble de la zone de champ.

Le GPU Tesla C2070 (Fermi) dispose de plus de mémoire que le GTX 580 (Fermi) ce qui permet au calcul de champ de s'exécuter avec succès sur un plus grand nombre de configurations. Concernant la configuration 06, zone de champ très définie avec 400×400 points de champ, seule le GTX Titan permet d'effectuer le calcul avec succès alors que le Tesla C2070 possède autant de mémoire. L'allocation de gros blocs mémoire semble avoir été optimisée sur l'architecture Kepler. Enfin, on peut noter que, sur le GTX Titan, plusieurs configurations atteignent et dépassent l'interactivité, générant plus de 25 images par seconde. Ces configurations sont présentées en gras dans le tableau de résultats 7.4.

7.5.2 Répartition des temps de calcul par étape

L'implémentation CUDA de la simulation de champ rapide comporte plusieurs points de mesure permettant d'obtenir les temps d'exécution de chacune des sous-étapes du calcul.

Il s'agit des points suivants :

Initialisation du GPU afin de mesurer le temps nécessaire à l'allocation mémoire des pinceaux et au transfert des données de configuration depuis l'hôte.

Étape 1 qui mesure le temps du calcul des pinceaux.

Étape 2 qui réalise la mesure du temps de calcul de la largeur de signaux nécessaires au calcul de champ.

Étape 3 qui mesure les temps de calcul du traitement du signal, de la sommation des contributions des pinceaux (3.1) jusqu'à celui de l'extraction du maximum (3.2).

Copie GPU vers Hôte qui mesure le temps de transfert de l'image de champ, en amplitude et en temps de vol, depuis le GPU vers la machine hôte.

La figure 7.4 représente la proportion du temps de calcul, sur plusieurs GPU, pour toutes les configurations de champ, dans les différentes étapes du calcul de champ à l'aide d'histogrammes cumulés de 0 à 100%. Les étapes d'initialisation et de copie du résultat du calcul de champs du GPU vers l'hôte sont négligeables face aux temps d'exécution des étapes 1, 2 et 3.

Ensuite, on observe la prépondérance de l'étape 3 sur les deux autres dans la quasi totalité des configurations. Après profilage, le noyau étant le temps le plus long à s'exécuter est le noyau 3.1, celui qui réalise la sommation des contributions des pinceaux. Il repose sur l'usage d'opérations atomiques pour ajouter les contributions aux réponses impulsionnelles de déplacement. La sous-section suivante étudie en détail l'impact des opérations atomiques sur ce noyau.

7.5.3 Impact des opérations d'additions atomiques

L'implémentation GPU fait appel à des opérations atomiques dans deux des étapes du calcul de champ : les étapes 2 et 3.1. Comme cette dernière étape est de loin la plus coûteuse, cette sous-section se focalise dans un premier temps sur son optimisation[4].

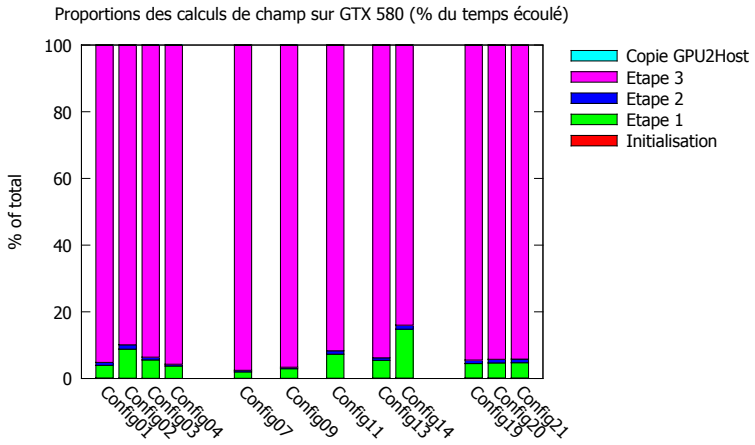
Ce comportement de concurrence au sein d'un bloc est l'un des points améliorés par les nouvelles architectures GPU et dont nVidia vante les bénéfices sur les architectures de GPU nouvelle génération Kepler. Afin de valider ce comportement, les opérations atomiques d'addition de l'étape 3.1 sont désactivées au moyen d'une macro remplaçant l'opération atomique par une opération standard de lecture/écriture mémoire. Ce remplacement est illustré par le listing 14. Le résultat est bien entendu erroné, mais le nombre d'accès mémoire réalisés reste le même, ce qui permet d'évaluer s'il y a ou non un impact des opérations atomiques sur les performances du calcul de champ.

Programme 14 CUDA : AtomicAdd et contournement

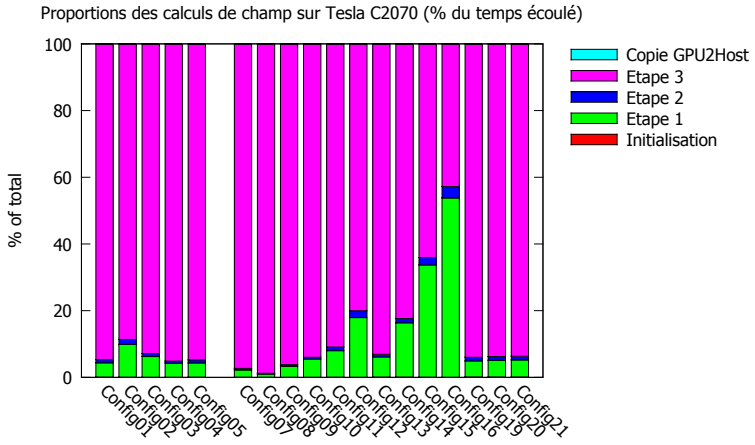
```
1 //Prototype standard de la fonction d'addition atopique
2 float atomicAdd(float* address, float val);
3 //Macro de remplacement (résultat faux si concurence)
4 #define atomicAdd(address,val) ( (*(address))+=val );
```

Le tableau 7.5 présente les performances, sur la configuration de référence avec et sans opérations atomiques, sur le calcul total du champ et sur la sous-étape 3.1. Il met en évidence l'importance de l'étape 3.1, qui représente la plus grosse partie du temps de calcul total sur toutes les cartes. En comparant les performances avec ou sans opérations atomiques (Atom. ON/Atom. OFF), sur les GPU GTX 580 et Tesla C2070 tous deux de génération Fermi, l'impact de ces opérations est important et représente respectivement 62.1% et 57.9% du total, soit 65.3% et 61.1% de l'étape 3.1. Par contre, ce même tableau montre que sur l'architecture Kepler, représentée par le GPU GTX Titan, ces opérations atomiques sont quasiment gratuites en termes de temps d'exécution, elles ont un impact très faible ! Ces opérations représentent moins de 4% du temps de calcul total de champ.

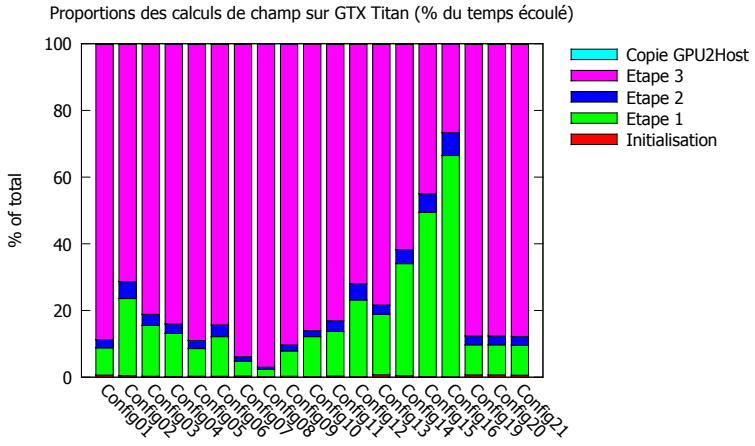
[4] Un retour sur l'étape 2 sera effectué en conclusion de cette sous-section, au paragraphe 7.5.3.2.



(a) Répartition par étape du temps de calcul de champ sur GTX 580



(b) Répartition par étape du temps de calcul de champ sur Tesla C2070



(c) Répartition par étape du temps de calcul de champ sur GTX Titan

FIGURE 7.4 – Répartition par étape du temps de calcul de champ sur plusieurs GPU

	Étape 3.1			
Configuration 1	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	145.9	50.7	95.2	65.3 %
Tesla C2070	199.2	77.5	121.7	61.1 %
GTX Titan	30.8	29.5	1.4	4.4 %
	Calcul Total			
Configuration 1	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	153.4	58.2	95.2	62.1 %
Tesla C2070	210.4	88.7	121.7	57.9 %
GTX Titan	34.8	33.4	1.4	3.9 %

TABLE 7.5 – Évaluation des opérations atomiques - Performances en ms

Sur l'ensemble des configurations, les opérations atomiques sont très coûteuses sur architecture Fermi ce qui n'est pas le cas sur Kepler. En revanche sur cette architecture Kepler, un phénomène surprenant apparaît pour certaines configurations, en particulier sur les configurations 19, 20 et 21 qui ne diffèrent que par la loi de retards appliquée.

Sur ces configurations, la focalisation électronique fait varier le schéma de sommation des contributions. Les tableaux 7.6, 7.7 et 7.8 indiquent que l'usage des instructions atomiques (colonne *Atom. ON* par rapport à *Atom. OFF*) **accélère l'exécution du code de calcul**. On remarque également que plus la focalisation se fait dans la zone de champ et moins les opérations atomiques sont avantageuses. Lorsqu'il y a focalisation, les pinceaux contribuent avec un temps de vol proche dans la zone de champ, afin de maximiser l'énergie. La concurrence pour la sommation sur le signal augmente. Inversement, lorsque le capteur n'est pas focalisé, la concurrence est moins présente. On peut supposer en conséquence que le motif des accès mémoire est à la base des différences de performances des opérations atomiques.

Ce phénomène est aussi présent sur les configurations 2, 3 et 4, qui, en allumant des éléments supplémentaires du capteur, augmentent la proportion de mémoire sur laquelle les pinceaux contribuent à un même signal (plus d'éléments du capteur actifs entraîne plus de pinceaux pour des temps de vol assez proches).

Ce comportement n'est pas mentionné dans le *White Paper* de l'architecture Kepler [nVi12] ni sur l'article du weblog nVidia dédié à CUDA traitant des opérations atomiques [Lui14]. Des investigations supplémentaires sont nécessaires afin de vérifier si ces variations des performances sont liées à des erreurs de mesure ou s'il s'agit des capacités de l'architecture Kepler.

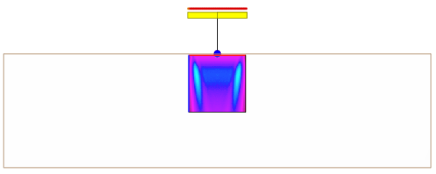


FIGURE 7.5 – Configuration 19 - Champ simulé (Image CIVA)

	Étape 3.1			
Configuration 19	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	128.3	49.6	78.7	61.4 %
Tesla C2070	176.3	75.6	100.6	57.1 %
GTX Titan	27.9	29.0	-1.1	-4.1 %
	Calcul Total			
Configuration 19	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	135.8	57.1	78.7	58.0 %
Tesla C2070	187.6	86.9	100.6	53.7 %
GTX Titan	31.9	33.0	-1.1	-3.6 %

TABLE 7.6 – Configuration 19 - Évaluation des opérations atomiques - Performances en ms

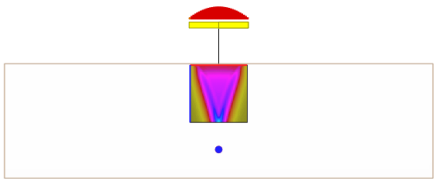


FIGURE 7.6 – Configuration 20 - Champ simulé (Image CIVA)

	Étape 3.1			
Configuration 20	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	123.1	48.9	74.2	60.3 %
Tesla C2070	169.2	74.7	94.5	55.9 %
GTX Titan	27.9	28.8	-0.9	-3.1 %
	Calcul Total			
Configuration 20	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	130.6	56.4	74.2	56.8 %
Tesla C2070	180.5	86.0	94.5	52.4 %
GTX Titan	31.9	32.8	-0.9	-2.7 %

TABLE 7.7 – Configuration 20 - Évaluation des opérations atomiques - Performances en ms

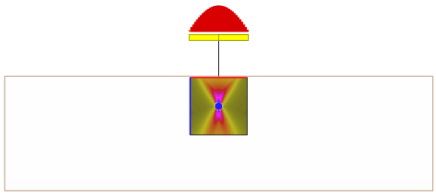


FIGURE 7.7 – Configuration 21 - Champ simulé (Image CIVA)

	Étape 3.1			
Configuration 21	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	120.6	48.5	72.2	59.8 %
Tesla C2070	165.8	73.9	91.9	55.4 %
GTX Titan	28.0	28.4	-0.4	-1.5 %
	Calcul Total			
Configuration 21	Atom. ON	Atom. OFF	Surcoût op. atom.	% des op. atom. sur total
GTX 580	128.2	56.0	72.2	56.3 %
Tesla C2070	177.1	85.2	91.9	51.9 %
GTX Titan	32.0	32.4	-0.5	-1.4 %

TABLE 7.8 – Configuration 21 - Évaluation des opérations atomiques - Performances en ms

7.5.3.1 Minibenchmark des opérations atomiques

Afin de mesurer précisément l'impact des opérations atomiques en fonction du motif avec lequel les accès mémoire vont être effectués, un *minibenchmark* a été mis en place, inspiré des mesures proposées par *Farzad* sur le forum *stackoverflow* [Far14]. Il consiste à mesurer les performances d'une opération atomique d'addition sur un entier 32 bits sur un tableau de taille fixe (dans le cas où la valeur de retour de l'opération atomique n'est pas conservée par le noyau comme pour les algorithmes du calcul de champ[5]).

En définissant quatre motifs réguliers, il va être possible de caractériser les performances des opérations atomiques afin de pouvoir rapprocher les comportements des différentes configurations de champ (étape 3). Dans le cas du calcul de champ, l'opération atomique considérée est l'addition de flottant simple précision. Différents noyaux ont été réalisés afin de mesurer les performances de l'opération atomique en mémoire globale et en mémoire partagée, chacun utilisant un motif d'accès mémoire particulier. Le tableau 7.9 présente les stratégies de calcul de l'adresse mémoire sur laquelle l'opération atomique est réalisée, pour les différentes valeurs du `tid`, indice linéaire du *thread*. Les algorithmes en résultant sont :

- L'algorithme **coalescent** réalise une écriture cohérente en mémoire, de manière linéaire et contiguë : accès sans concurrence entre les *threads*.
- L'algorithme **warp restricted** comprime les adresses mémoire en assignant la même adresse à tout un *warp* : concurrence des *threads* d'un même *warp*.
- L'algorithme **address restricted** masse les différents *warps* sur les mêmes 32 éléments mémoire (par un accès cohérent) : concurrence *warp* à *warp* par bloc.
- L'algorithme **same address** regroupe toutes les opérations en une seule adresse mémoire : concurrence de tous les *threads* d'un noyau.

[5]Dans certains cas, le compilateur peut être amené à optimiser l'opération atomique.

Les mesures sont tout d'abord réalisées sur la **mémoire globale du GPU**, dont les résultats sont rassemblés au tableau 7.10. Les écarts entre Fermi et Kepler indiquent des améliorations constantes sur les différentes stratégies de sommation. Pour chaque stratégie proposée, il est possible de déduire des résultats la méthode avec laquelle le GPU CUDA réalise son traitement atomique. Des temps rapides indiquent des raccourcis spécifiques aux traitements d'un bloc, alors que des temps d'accès longs sont témoins d'un accès à des niveaux de mémoire supérieure (mémoire partagée, cache L2 et/ou mémoire globale).

- Pour un algorithme `coalescent`, l'opération atomique est peu coûteuse sur un GPU Fermi (un surcoût de l'ordre de 20%) et sur GPU Kepler elle devient même plus rapide qu'un accès classique.
- Pour un algorithme `warp restricted`, indépendamment du type de GPU, les opérations atomiques sont coûteuses par rapport à un accès standard, signe d'une sérialisation des différents accès des *warps* d'un même bloc. On remarque également une nette accélération relative (en termes de cycles) avec la nouvelle architecture (presque deux ordres de grandeur).
- Pour un algorithme `address restricted`, les versions avec opérations atomiques sont les plus rapides pour les deux générations de GPU. Cela met en lumière la capacité de ces GPU à traiter, au niveau du bloc, l'accès atomique sans repasser par la mémoire globale. Par contre la lenteur des opérations atomiques par rapport à l'accès `coalescent` montre tout de même des besoins de synchronisation entre *warps*.
- Un algorithme `same address` présente les performances les moins bonnes en accès atomique en raison du besoin de synchronisation maximal au sein et à l'extérieur du *warp*. Cette stratégie présente dans le même temps l'accès direct le plus rapide à la mémoire globale car le GPU n'a besoin d'accéder qu'à une très petite plage de mémoire globale.

Les résultats obtenus pour les mêmes stratégies sur l'accès à **la mémoire partagée** via des opérations atomiques, sont rassemblés dans le tableau 7.11. D'une manière globale, les performances Fermi et Kepler ont la même allure. Dans chaque cas, les opérations atomiques sont plus lentes que des accès directs équivalents. Par contre, les performances dépendent des accès concurrents aux différentes banques mémoire :

- Pour une stratégie `coalescent`, les accès atomiques sont optimaux : chaque *thread* accède à une banque mémoire différente et il n'y a pas de synchronisation entre deux *warps*.
- Pour une stratégie `warp restricted`, les collisions d'accès ont lieu pour tous les *threads* d'un *warp* vers une même banque mémoire.
- Pour des accès mémoire `address restricted`, la concurrence n'a lieu qu'entre deux *threads* de la même position dans deux *warps* différents : le GPU réussit à ordonnancer les *warps* afin de minimiser les latences des accès à la mémoire partagée.
- Pour un accès `same address`, tous les *threads* du bloc sont en concurrence pour accéder à la même banque mémoire : le temps d'accès atomique est le plus long. Par contre, comme pour un accès à la mémoire globale, avec cette stratégie, l'accès direct est également le plus rapide.

atomicAdd(ptr + stride , value)						
code version	**fomula**	**Memory stride value (for each tid)**				
coalescent	tid	[0;1; ... ; 31]	[32; ... ; 63]	[64; ... ; 95]	[96; ... ; 127]	...
warp restricted	tidi>>5	[0; 0; .... ; 0]	[... ; 1 ; ...]	[... ; 2 ; ...]	[... ; 3 ; ...]	...
address restricted	tid&(32-1)	[0;1; ... ; 31]	[0;1; ... ; 31]	[0;1; ... ; 31]	[0;1; ... ; 31]	...
same address	0	[0; 0; .... ; 0]	[0; 0; .... ; 0]	[0; 0; .... ; 0]	[0; 0; .... ; 0]	...

TABLE 7.9 – Différentes stratégies d'utilisation des opérations atomiques (`warpsize = 32`)

Operation		stride	GPU			
			GTX 580 Fermi		GTX TITAN Kepler	
			μs	cycles	μs	cycles
`coalescent`	Atomic	`tid`	2057	3.18×10^6	**1198**	1.05×10^6
`coalescent`	Normal		1723	2.66×10^6	1231	1.08×10^6
`warp restricted`	Atomic	`tid >> 5`	66682	1.03×10^8	3124	2.73×10^6
`warp restricted`	Normal		901	1.39×10^6	575	5.03×10^5
`address restricted`	Atomic	`tid & (32-1)`	**11654**	1.80×10^7	**4461**	3.90×10^6
`address restricted`	Normal		17660	2.73×10^7	27822	2.43×10^7
`same address`	Atomic	`0`	372829	5.76×10^8	33762	2.95×10^7
`same address`	Normal		257	3.97×10^5	121	1.06×10^5

TABLE 7.10 – Mesures des opérations atomiques en mémoire globale (temps en μs et en cycles)

Operation		stride	GPU			
			GTX 580 Fermi		GTX TITAN Kepler	
			μs	cycles	μs	cycles
`coalescent`	Atomic	`tid`	1072	1.66×10^6	1117	9.77×10^5
`coalescent`	Normal		856	1.32×10^6	734	6.42×10^5
`warp restricted`	Atomic	`tid >> 5`	13376	2.07×10^7	18467	1.62×10^7
`warp restricted`	Normal		695	1.07×10^6	596	5.22×10^5
`address restricted`	Atomic	`tid & (32-1)`	1958	3.02×10^6	2376	2.08×10^6
`address restricted`	Normal		399	6.16×10^5	247	2.17×10^5
`same address`	Atomic	`0`	54509	8.42×10^7	55927	4.89×10^7
`same address`	Normal		341	5.27×10^5	211	1.85×10^5

TABLE 7.11 – Mesures des opérations atomiques en mémoire partagée (temps en μs et en cycles)

7.5.3.2 Conclusion sur l'étape 3.1

Pendant l'étape 3.1, étape de sommation des contributions des pinceaux sur un signal de réponse impulsionnelle, les échantillons, sur lesquels une contribution vient s'ajouter, sont soumis à une

opération atomique additive. Cette étape du calcul étant prépondérante en temps de calcul, la présente sous-section va étudier plus avant l'impact des opérations d'addition atomique. Pour rappel, dans la mesure où chaque bloc de ce noyau correspond à un seul point de champ, chaque bloc dispose de ses propres signaux : la concurrence d'accès aux signaux provient des *threads* d'un même bloc.

Les résultats présentés au paragraphe précédent permettent de comprendre les performances mesurées des simulations de champ. Les sommations sans opérations atomiques sont généralement plus rapides que les versions algorithmiquement justes utilisant les opérations atomiques. Cependant, sur certains motifs d'accès mémoire, les opérations atomiques sont plus rapides, en mémoire globale, que les accès classiques.

Pour certaines configurations, les paramètres du calcul font passer la phase 3.1 de sommation des contributions des pinceaux dans un cas de figure défavorable aux opérations non-atomiques. Par exemple, sur la configuration 19, sans focalisation, les pinceaux d'un point de champ sont biens répartis (sans recouvrement) et les *threads* chargés de la sommation sont faiblement en concurrence avec les *warps* voisins. Le motif qui s'en approche le plus est le cas `coalescent` (les pinceaux d'un *warp* ont des temps de vol proches et se somment dans un faible nombre de lignes mémoire). Inversement, sur la configuration 21 focalisée, les pinceaux de la tâche focale voient leurs contributions se sommer "au même endroit" sur le signal. Les contributions des pinceaux traitées par un *warp* débordent sur celles du *warp* voisin, de façon similaire à un motif `warp restricted`. Ceci explique les variations de performances, et le gain obtenu par l'utilisation des opérations atomiques.

Ces résultats montrent que dans beaucoup de cas, les progrès réalisés par les GPU nVidia ont rendu les opérations atomiques, de coûteuses sur architecture Fermi à quasiment gratuites sur GPU Kepler. Cela permet d'éviter de devoir effectuer des transformations algorithmiquement lourdes telles que la transposition des accès mémoire. Cette évolution permet d'envisager, sur les GPU de générations actuelle et future, le développement de noyaux de calculs algorithmiquement simples sans dégrader les performances du calcul.

Le noyau de sommation n'est pas le seul à utiliser des opérations atomiques, les noyaux de l'étape 2 utilisent des recherches de maximum atomiques. Pendant l'étape 2, les deux noyaux de parcours des étalements temporels font appel à des opérations de type minimum/maximum atomiques (l'un sur tous les pinceaux, l'autre pour effectuer une réduction sur l'ensemble du champ)[6]. Dans ces deux cas, ces opérations sont utilisées pour éviter la concurrence entre *threads* d'un même bloc pour l'accès mémoire. Il y a concurrence entre les *threads* du premier noyau pour déterminer pour chaque point de champ un intervalle temporel à partir des contributions des pinceaux. Le second noyau détermine par réduction en un seul bloc de *threads* l'intervalle temporel global par réduction. Les accès sont *coalescents* car les pinceaux sont stockés sous forme de structure de tableaux. Par contre, comme cette étape 2 n'est pas prépondérante en temps de calcul, un tel *minibenchmark* sur les opérations de maximum atomique n'a pas été réalisé. On supposera, par analogie avec les additions atomiques, qu'il s'agit d'un accès performant.

7.5.4 Passage à l'échelle

Cette section traite du passage à l'échelle, sur des GPUs d'une même architecture, des différentes étapes de calcul à partir de quelques mesures. En rapportant un temps de calcul à un nombre de cycles total passés sur GPU, il est possible d'estimer la quantité de calcul (en se rapportant aux cycles du multiprocesseur) ou la quantité de mémoire (en se rapportant aux cycles mémoire).

[6] Afin de procéder à la réduction des informations temporelles, les noyaux de l'étape 2 utilisent un peu de mémoire partagée. La quantité utilisée est très faible devant les limites d'un multiprocesseur CUDA. Aussi son impact est nul.

En observant ces cycles sur différents GPU d'une même famille, s'il y a une corrélation pour un type de cycles, il sera possible de mettre en évidence que les performances du noyau considéré sont limitées soit par la mémoire soit par la puissance de calcul. Par contre, lorsqu'il n'y a pas de corrélation, il n'est pas possible de qualifier le noyau.

Dans cette sous-section, on va étudier les différents noyaux du calcul de champ à partir de la famille Fermi. Les deux études ont été réalisées sur l'ensemble des noyaux, cependant seuls les résultats pertinents sont présentés, étape par étape. Les GPU Fermi et Kepler utilisant des schémas de fonctionnement proches en termes d'équilibre calcul/mémoire, des conclusions quant à l'exécution future du champ seront tirées.

7.5.4.1 Étape 1 : passage à l'échelle des calculs - cycles multiprocesseurs

GPU	GeForce GTX 580	Tesla C2070	GeForce GTX Titan
Architecture	Fermi	Fermi	Kepler
Compute Capability	2.0	2.0	3.5
Nombre de multiprocesseurs	16 SM	14 SM	14 SMx
Fréquence des cœurs CUDA	1544 MHz	1150 MHz	875 MHz
Fréquence de la mémoire globale	2004 MHz	1500 MHz	1502 MHz
Type de la mémoire globale	GDDR5	GDDR5	GDDR5

TABLE 7.12 – Spécifications techniques des GPU

Source : nVidia

A partir des temps d'exécution des différentes étapes du calcul, il est possible d'en déduire le nombre de cycles passés dans chaque étape en multipliant le temps d'exécution par la fréquence du GPU rappelée au tableau 7.12. Les *warps* s'exécutent de manière synchrone sur les différents multiprocesseurs du GPU. En prenant en compte le nombre de multiprocesseurs de chaque GPU, on obtient la mesure des cycles exécutés en mode "calculs"[7].

La figure 7.8 présente ces résultats concernant l'étape 1 du calcul de champ. Sur les GPU de même architecture Fermi (Tesla C2070 et GTX 580), les cycles écoulés sont identiques pour une configuration donnée (sur les configurations supportées par les deux GPU). Cela confirme que cette étape de calcul de pinceaux est *compute bound*. Par application de deux règles de trois sur les fréquences et le nombre de multiprocesseurs disponibles sur un GPU d'une architecture Fermi ou Kepler donné, à partir de ces mesures, il est possible d'anticiper le coût de cette étape de la simulation pour un nouveau GPU (pour ces configurations de champ). La juxtaposition des performances Fermi/Kepler permet aussi de mettre en lumière la meilleure performance des GPU Kepler, qui exécutent le même code de simulation en un nombre moindre de cycles.

7.5.4.2 Étape 3 : passage à l'échelle - cycles mémoire

Cette mesure de passage à l'échelle peut également être réalisée pour l'étape 3 du calcul de champ, en estimant le nombre de cycles mémoire écoulés. Puisqu'un GPU ne dispose que d'une seule mémoire globale, il n'est pas nécessaire de prendre en compte les multiprocesseurs. La figure 7.9 représente les résultats correspondant à l'étape 3 du calcul de champ. La constance

[7]Les codes de simulation de champ s'exécutent en simple précision. Sur des codes en double précision il faudrait tenir compte de la disparité de performances simple/double précisions entre les cartes Fermi de gamme GeForce et les GPU de gamme Tesla. Ces disparités entre gammes existent également pour les architectures plus récentes entre GPU à destination de l'affichage/jeu vidéo et GPU plus orienté GPGPU.

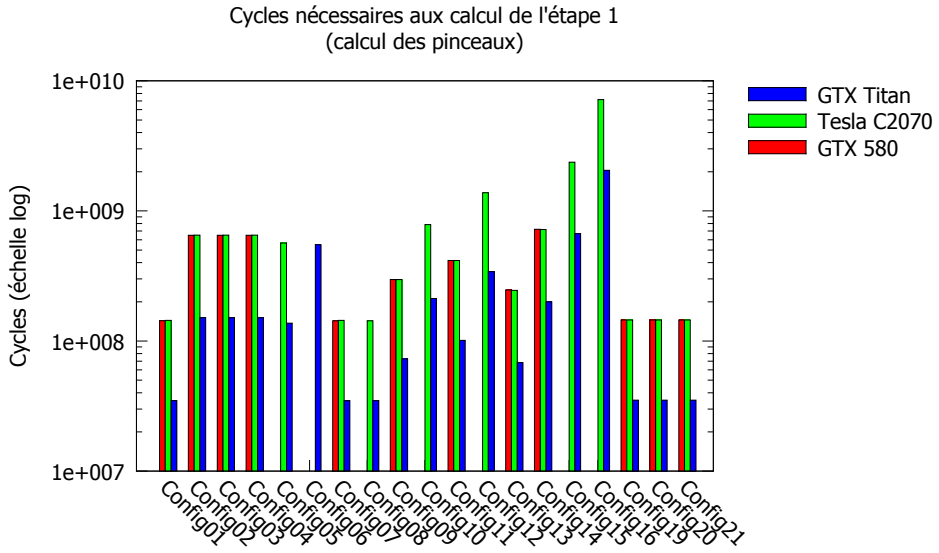


FIGURE 7.8 – Cycles GPU écoulés pour le calcul de l'étape 1 sur différents GPU

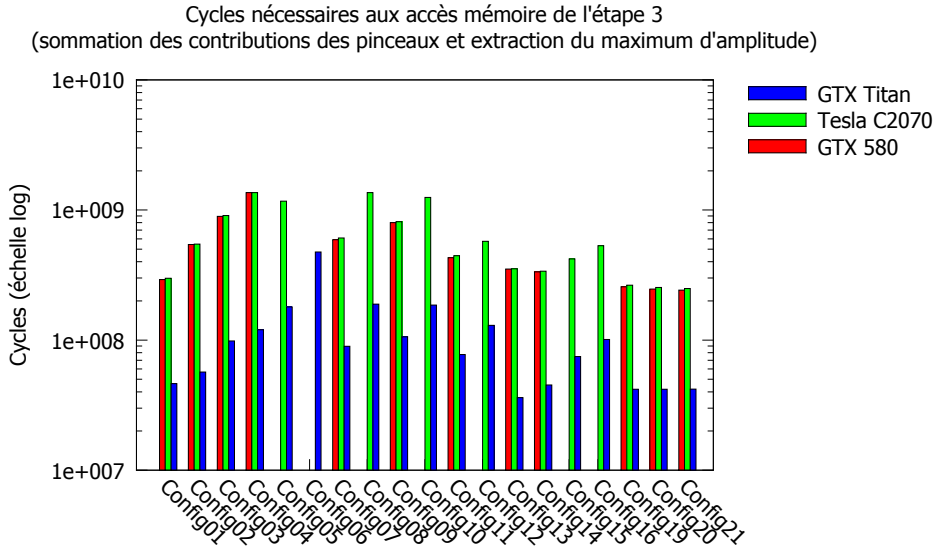


FIGURE 7.9 – Cycles mémoire écoulés pour le calcul de l'étape 3 sur différents GPU

des cycles écoulés pour l'accès mémoire sur cette étape du calcul, pour deux GPU de la même famille, révèle la nature *memory bound* de l'étape 3. De la même manière que sur les cycles de calcul écoulés, une règle de trois permet d'anticiper, sur un nouveau GPU d'une des architectures étudiées, les performances attendues.

7.5.4.3 Étape 2 : pas de constance du nombre de cycles

Les deux types de transformations présentées ci-dessus sont appliqués aux mesures réalisées sur l'étape 2 : extraction du nombre de cycles de calcul d'une part, et obtention du nombre de cycles passés à faire des accès mémoire d'autre part. Il n'apparait pas, sur GPU Fermi, de corrélation forte entre les deux GPU considérés pour l'un ou l'autre des chiffres à travers les mesures. La normalisation des cycles de calcul donne un rapport GTX 580/Tesla C2070 moyen de 1.05 ± 0.023 , alors que la normalisation des accès mémoire mesure un ratio de 0.91 ± 0.020. Ainsi il n'y a pas de facteur discriminant sur les performances de ce noyau de calcul.

7.6 Conclusion

7.6.1 Performances

Un portage sur GPU de l'algorithme de simulation de champ rapide a été réalisé. Afin de paramétrer les noyaux de calcul développés, une étude empirique a été mise en place. Les valeurs du nombre de *threads* par bloc et du nombre maximum de registres autorisés par *threads* (à la compilation) ont ainsi pu être sélectionnées.

Dans cette implémentation, le facteur prépondérant en temps de calcul est l'étape 3, réalisant la sommation des contributions des pinceaux et l'extraction du maximum d'amplitude. Sur la sommation, une étude spécifique au moyen d'un *minibenchmark* des opérations atomiques a montré, sur architecture Kepler, les très bonnes performances des opérations atomiques.

L'utilisation de GPU GTX Titan de dernière génération permet d'obtenir, sur un ensemble important de configurations de champ, des performances atteignant l'interactivité (au sens de 25 images par seconde). En réduisant par 2 selon chaque axe les dimensions de la zone de champ par exemple, la zone de champ de la configuration de référence passant de 101×101 à 51×51 points de champs, ces performances interactives semblent également atteignables sur des GPU d'ancienne génération Fermi.

7.6.2 Limitations

L'ensemble des mesures réalisées dans ce chapitre a montré les limitations des matériels en termes de quantités de calcul/quantités d'accès mémoire. Cependant une autre limitation est présente sur le GPU. Il s'agit des contraintes mémoire. Certaines configurations ne peuvent s'exécuter en une seule passe sur des GPU actuels et excèdent la mémoire disponible. Le pic maximal d'occupation mémoire est atteint après la phase d'estimation de la taille des pinceaux (étape 2) : il faut garder en mémoire les pinceaux et il faut tenir les 6 composantes de la réponse impulsionnelle par point de champ.

7.6.3 Perspectives

Le code de calcul de champ actuel ne peut s'exécuter sur les GPU modestes du fait d'une quantité de mémoire embarquée trop restrictive.

Afin de résoudre le problème, puisque le calcul d'un point de champ est autonome vis à vis de ses voisins, un traitement par passe sur un sous-ensemble des points de champ semble être la voie à suivre pour contourner les limites mémoire des GPU. Pour dimensionner le sous-ensemble de pixels traités par un tel algorithme, il sera nécessaire d'estimer à l'avance la taille mémoire requise pour les signaux de réponse impulsionnelle. Dans le cadre d'une utilisation industrielle de la simulation de champ, l'expert CND peut être amené à spécifier une fenêtre temporelle qui peut servir de base à l'estimation. Cependant, cette donnée n'est pas infaillible, elle peut, par

exemple, être trop grande ce qui ne donnera pas de résultats. Il faudra probablement compléter ce guidage utilisateur par une heuristique automatique de la simulation.

L'extension des capacités du calcul de champ en termes fonctionnels (gestion du mode rebond avec conversion, gestion de géométries plus complexes...) prendra la forme de nouveaux noyaux venant compléter l'étape 1. Ainsi, les développements réalisés pour les étapes 2 et 3 restent pertinents.

L'usage de futurs GPU, grâce aux avancées matérielles, permet d'envisager des simulations de champ encore plus rapides. Une mesure du calcul de champ sur la nouvelle architecture *Maxwell* permettra, par analogie, d'estimer les performances pour une configuration donnée sur l'ensemble de la gamme de GPU de génération *Maxwell* par linéarité des performances des étapes 1 et 3.[8].

[8]L'étape 2 des calculs est déjà très rapide aujourd'hui et ses performances devraient s'adapter aux nouvelles générations de GPU

Chapitre

8 Conclusions et perspectives

8.1 Conclusions

Les travaux effectués dans le cadre de cette thèse ont contribué à l'élaboration d'un modèle de calcul de champ dérivé de celui de CIVA. Il présente l'avantage d'être régulier et permet

ainsi l'exploitation des capacités des architectures parallèles visées. Cette régularité a nécessité la restriction de son champ d'application par rapport à CIVA.

Une première implémentation de référence a permis :

- de valider les résultats de calcul par rapport à ceux obtenus dans CIVA et de confirmer la validité du modèle ;
- de caractériser le comportement de l'algorithme en termes de complexité et de temps d'exécution de chacune de ses étapes ;
- de servir de référence pour les algorithmes optimisés sur chacune des architectures matérielles visées.

L'algorithme a été porté et optimisé sur les architectures parallèles visées, offrant des gains en temps de calcul appréciables.

8.1.1 Calcul de champ rapide sur architectures parallèles

L'algorithme réalisant un calcul de champ est un algorithme complexe à paralléliser. Ses différentes étapes ont chacune des comportements divers et des contraintes variées (quantité de calculs, régularité, accès mémoire...). Sur chaque architecture ciblée, une stratégie spécifique a dû être employée afin d'exploiter ses particularités.

Sur GPP, plusieurs optimisations ont été mises en place pour exploiter les deux niveaux de parallélisation disponibles. Dans un premier temps, une fusion des boucles de plus haut niveau sur les points de champ pour chaque étape de calcul en une seule boucle parallélisée (algorithme vertical) a été effectuée. Ensuite, pour bénéficier des instructions vectorielles, les étapes de calcul ont été vectorisées : manuellement d'une part à l'aide de la bibliothèque Boost.SIMD et d'autre part grâce à l'utilisation de la bibliothèque Intel MKL pour les calculs de Transformées de Fourier Rapide. L'étape de sommation des contributions des pinceaux en tenant compte des retards a fait, quant-à-elle, l'objet d'une optimisation spécifique à l'aide d'un algorithme différentiel permettant de limiter les accès mémoire.

Sur MIC, grâce aux grandes similarités avec les GPP et la portabilité des outils utilisés (Boost.SIMD et Intel MKL), le code exécuté est le code GPP recompilé afin d'utiliser le parallélisme *manycore* et les unités de calcul vectoriel de 512 bits.

Sur GPU, l'algorithme a été implémenté en CUDA pour des GPU de marque nVidia. Cette implémentation prend la forme d'un ensemble de noyaux qui s'exécutent successivement sur le GPU et dont le flux de données est piloté depuis la machine hôte. Concernant les opérations de Transformées de Fourier Rapides, la bibliothèque cuFFT a été utilisée.

Pour chacune de ces architectures, une analyse fine de l'efficacité des optimisations apportées a été réalisée.

8.1.2 Bilan général des performances

La table 8.1 reprend les performances de toutes les configurations de champ mesurées, sur chaque classe d'architecture, sur la machine la plus rapide, pour l'implémentation la plus rapide[1]. Ces simulations permettent d'obtenir un calcul de champ complet (amplitude, temps de vol et signaux de déplacement) en des temps très brefs. Chacune des architectures étudiées a atteint ou s'est approchée de très près de l'interactivité sur plusieurs configurations de contrôle réalistes (au sens de 25 images par seconde).

Ces architectures appartenant au même ensemble de matériels en termes de cible (stations de calculs puissantes) et ayant été commercialisées durant la même période (février à septembre

[1] Sur MIC, cela dépend de la configuration adressée.

Matériel Prix à la sortie (USD)	GPP 2×Xeon Ivy Bridge 5228	MIC Xeon Phi E3120 1830	GPU GTX Titan (Kepler) 999
Configurations	Performances en images/seconde		
Configuration 01	38.0	23.2	28.8
Configuration 02	32.9	1.5	18.8
Configuration 03	15.5	1.6	12.4
Configuration 04	12.3	8.9	10.5
Configuration 05	9.74	4.0	7.4
Configuration 06	2.5	1.0	2.7
Configuration 07	30.3	24.2	15.7
Configuration 08	19.2	3.2	7.7
Configuration 09	18.5	43.6	12.7
Configuration 10	8.1	1.36	7.0
Configuration 11	19.4	6.7	16.1
Configuration 12	7.2	1.8	8.3
Configuration 13	34.1	11.9	32.6
Configuration 14	16.1	4.5	20.5
Configuration 15	5.6	1.4	9.0
Configuration 16	1.9	0.5	4.0
Configuration 17	0.6	-	-
Configuration 18	0.5	-	-
Configuration 19	36.1	21.5	31.4
Configuration 20	37.0	22.2	31.3
Configuration 21	37.1	22.5	31.3

TABLE 8.1 – Récapitulatif des performances atteintes sur les différentes configurations de champ, pour le représentant le plus rapide de chaque architecture étudiée, pour l'algorithme le plus rapide (vert : interactivité atteinte $>$ 25 fps, orange : interactivité presque atteinte $>$ 20 fps , rouge : $<$ 1 fps)

2009), leur comparaison est donc pertinente. Des performances en dollars par images/seconde (USD/fps) peuvent être obtenues sur l'ensemble des configurations. En particulier, sur la configuration 01, les scores atteints sont : sur GPP 137 USD/fps, sur MIC 79 USD/fps et sur GPU 35 USD/fps. Ainsi, il est possible de donner le GPU et en particulier la GTX Titan Kepler comme l'architecture la plus économique pour le calcul de champ interactif aujourd'hui. Cependant, cette valeur est bien trop simpliste pour donner des prédictions sur le coût d'une simulation *a priori*. Le prix d'un matériel ne dépend pas que de sa puissance, de nombreux paramètres interviennent également (nombre de cœurs, fréquence, caches, mémoire, stratégie commerciale...).

8.2 Perspectives

Avec des optimisations supplémentaires rendues nécessaires par les exigences du déploiement industriel, les résultats obtenus par les présents travaux pourront contribuer au développement de nouvelles fonctionnalités du logiciel CIVA autour de la simulation interactive.

8.1.1 Optimisations supplémentaires

Le code développé dans le cadre de cette thèse est un code "académique" dédié à la mesure de performances sur les différentes architectures. Outre ces mesures, il permet d'exécuter toutes les versions algorithmiques du code développées et testées. Dans le cadre d'une industrialisation, seule la version la plus efficace sur chaque architecture devra être retenue et remaniée afin de fournir un code de qualité industrielle.

Pour rappel, l'ensemble de tous les pinceaux conservés en mémoire est de :

$$(N_{surfin} \times N_{modes\_direct} + N_{surfin} \times N_{surfback} \times N_{modes\_rebond}) \times N_{point} \times N_{capteur}$$

Ainsi, sur les architectures modestes ne disposant pas d'assez de mémoire pour stocker tout au long du calcul l'ensemble des pinceaux pour tous les points de champ (configurations 16 à 18), l'implémentation actuelle ne peut résider en mémoire et s'exécuter. Les calculs pour chaque point de champ étant indépendants, il sera aisé de réaliser une version bouclant sur les points de champ afin d'obtenir une image complète.

Pour les configurations les plus complexes, les performances des implémentations actuelles ne tiennent pas la cadence interactive. Lorsque la complexité géométrique augmente, le coût du calcul des pinceaux explose car tous sont calculés avant de rejeter ceux n'ayant pas de signification physique. De plus, tenir en mémoire tous les pinceaux, y compris les pinceaux invalides, pour ne pas compter leur contribution au moment de la sommation monopolise des ressources matérielles qui peuvent être nécessaires au reste de l'algorithme. Une heuristique réduisant ces pinceaux (n-uplets surfaces/modes) permettra d'améliorer les performances du calcul de champ sur des configurations complexes en éliminant le besoin de calculer l'intégralité des pinceaux potentiels. Des critères métier, tels que l'insonification des surfaces d'entrée par le capteur peuvent être utilisés. Par ailleurs une approche basée sur les angles critiques peut également éliminer des n-uplets de pinceaux. Ainsi dans le domaine de la vision, des heuristiques ont permis par exemple d'obtenir rapidement des images d'objets de type CAO 3D complexes réfractants (*c.f.* travaux de Walter et al. parus dans SIGGRAPH 2009 [WZHB09]).

Utiliser une heuristique réduisant le nombre de pinceaux calculés permettra également de lever en partie la problématique mémoire avec un couplage à un algorithme "vertical" (qui ne traite qu'un sous ensemble de points de champ à la fois). Il n'y aura plus besoin, à un instant `t`, que des informations des pinceaux utiles pour les points de champ en cours de traitement.

8.1.2 Ombrage et validité du trajet

Afin de prendre en compte des configurations de contrôle réalistes, la recherche de trajet ne doit pas se contenter d'analyser de manière indépendante chacune des surfaces/chacun des couples de surfaces, mais elle doit prendre en compte la configuration dans sa globalité. En particulier lorsque la pièce est composée de multiples surfaces, il faut s'assurer de la validité des trajets trouvés en garantissant qu'il n'y a pas d'occlusion du trajet trouvé par un autre élément de la pièce (surface ou défaut dans le cadre de l'écho). Cette validation n'est pas encore traitée dans l'algorithme rapide, *c.f.* paragraphe 4.2.1.1.4.

Une stratégie possible serait, de la même façon que pour un lancer de rayons, d'utiliser un intersecteur reposant sur une structure accélératrice permettant de partitionner l'espace et ainsi d'éviter de tester exhaustivement toutes les combinaisons.

8.1.3 Extensions fonctionnelles directes du calcul de trajet

Le modèle retenu dans ces travaux de thèse est un modèle semi-analytique qui se base sur la résolution numérique d'équations modélisant le trajet pour des surfaces planes et en direct et

rebond. Son extension possible aux cas de surfaces et modes plus complexes est discutée ci-après.

8.1.3.1 Des surfaces plus complexes

Les extensions directes du modèle du calcul de champ présentées ici n'ont pas été appliquées dans ces travaux de thèse. Celles-ci se placent dans le prolongement des travaux d'A. Pédron [Ped13] concernant le calcul des trajets par des résolutions numériques sur des configurations aux géométries complexes dans des matériaux homogènes isotropes.

8.1.3.1.1 Cas des interfaces canoniques Pour ces interfaces, l'équation prend la forme d'un polynôme dont le degré varie en fonction de la complexité de la géométrie et du type de contrôle :

- cylindre : un polynôme d'un degré inférieur ou égal à 10.
- cône : un polynôme d'un degré inférieur ou égal à 14.
- tore : un polynôme d'un degré inférieur ou égal à 16.

Dans ces configurations, plusieurs racines sont parfois trouvées indiquant la présence de plusieurs trajets/pinceaux correspondants. Celles-ci peuvent être à la fois réelles ou complexes. Si les trajets correspondants sont valides, toutes les racines réelles contribuent au champ. La méthode de Newton utilisée jusqu'ici devra être remplacée par une méthode permettant de trouver ces racines qui sont parfois complexes (mathématiquement), par exemple par la méthode de Laguerre accompagnée d'une déflation du degré du polynôme. Il sera nécessaire d'ajouter une étape de tri de ces racines afin de ne conserver que les racines réelles.

8.1.3.1.2 Cas d'un trajet passant par une surface d'entrée canonique courbe et rebond sur une surface plane Ce cas se rapporte à une symétrie sur une surface plane et une recherche de trajet direct. Ces trajets seront naturellement traités quand les cas des surfaces d'entrée canoniques "non planes" (cylindres, tores...) seront ajoutés.

8.1.3.2 Des modes plus longs

Des modes de trajets plus longs existent mais n'ont pas été traités dans ces travaux.

8.1.3.2.1 Mode rebond plus complexe Que ce soit dans le cas d'un rebond sur une interface courbe ou sur une interface plane avec conversion de mode, la recherche de trajets devient plus complexe. Elle s'apparente à la résolution d'un système d'équations à plusieurs inconnues dépendant de la configuration. Ainsi dans le cadre d'un trajet en rebond avec conversion de mode, entre deux interfaces planes parallèles, on écrit un système d'équations à deux inconnues permettant de s'assurer de suivre Snell-Descartes à la fois en réfraction à la surface d'entrée, et en réflexion avec conversion de mode sur la surface de fond. Ce système peut être résolu par exemple à l'aide d'une méthode de type Newton-2D.

8.1.3.2.2 Limites de l'approche par calculs de trajets analytiques Bien qu'il soit possible d'obtenir des trajets analytiques pour des géométries très complexes (maillage CAO 3D, nombre important de surfaces nécessitant des heuristiques), des types de modes complexes et des matériaux présentant une hétérogénéité minimale (par exemple plusieurs couches isotropes successives), plusieurs limites se posent à l'approche analytique pour l'obtention des trajets :

- les équations à résoudre sont très spécialisées (une équation, voire un système d'équations par type de calcul à modéliser) ;

- les équations sont complexes et coûteuses à résoudre ;
- les trajets restent limités à des pièces isotropes ;
- les heuristiques permettant d'éliminer des pinceaux peuvent être coûteuses à appliquer et parfois peu efficaces pour réduire le nombre de trajets ;
- les occlusions sont coûteuses à déterminer (intersections avec beaucoup de surfaces sur des pièces complexes).

Des travaux sont en cours au CEA-LIST afin d'utiliser un outil de lancer de rayons rapide pour accélérer les temps de calcul dans le cadre de la thèse d'H. Chouh [LCR+14]. En particulier, pour réduire la quantité de rayons calculée, une recherche progressive du capteur est couplée à une approche adaptative de sa finesse de couverture par les pinceaux. Cette approche permet de gérer la complexité géométrique, des modes complexes, ainsi que des matériaux anisotropes et des pièces hétérogènes. Par contre, les coûts en calcul ne permettront probablement pas d'atteindre des performances similaires au calcul direct développé ici pour les configurations les plus simples. Les deux approches sont donc complémentaires.

8.1.4 Vers une intégration dans CIVA

Les développements réalisés ont permis d'atteindre une simulation de champ ultrasonore précise aux performances interactives, ce qui représente une innovation majeure dans le cadre du contrôle non destructif par rapport aux autres outils disponibles sur le marché (*c.f.* chapitre 3). Cette simulation rapide va être intégrée dans la plateforme logicielle CIVA. Il convient à cette occasion de revenir sur les problématiques d'industrialisation du code, en particulier sa qualité, les contraintes de l'intégration et sur les usages possibles d'une simulation interactive.

8.1.4.1 Qualité de code

Les accélérations constatées ont été obtenues grâce à l'adaptation du code aux différentes architectures : analyse, réorganisation algorithmique et codage au moyen d'outils spécifiques. En termes d'effort de développement, à partir d'un algorithme de référence, le temps pour développer un code visant le GPP et le MIC et utiliser les instructions SIMD est du même ordre de grandeur que celui nécessaire à la mise au point d'un algorithme sous la forme de noyaux CUDA. Le développement d'un algorithme optimisé a été grandement facilité par la possibilité de se reporter régulièrement aux résultats valides de l'implémentation de référence.

La mise au point des différentes versions a été facilitée par la maturité des solutions retenues : sur GPP et MIC, les outils de *debugging* permettent d'observer le comportement des différents éléments d'un registre SIMD. Quant au GPU les outils NSight permettent d'observer finement le comportement des *threads* sur l'accélérateur. Par contre, l'usage massif de *templates* C++ dans Boost.SIMD rend la compilation des projets assez longue. De plus les erreurs relevées par le compilateur sont peu lisibles.

Concernant la pérennité du code développé, deux aspects sont à prendre en compte. Les outils utilisés sont, à une échelle de quelques années au moins, compatibles avec les futures architectures qui ont été annoncées. nVidia continue à proposer une rétro-compatibilité des codes dans leur format binaire. Ainsi les développements réalisés seront fonctionnels sur les nouvelles architectures matérielles GPU. De même, concernant les GPP, le code OpenMP et les instructions SIMD restent compatibles avec les nouvelles architectures. Afin de tirer parti des nouveaux jeux d'instructions SIMD, tels que l'AVX-512, une recompilation du code sera nécessaire lorsque le support de ces instructions sera disponible dans Boost.SIMD. L'étude des nouvelles fonctionnalités (*scatter/gather*, instructions masquées) sera également à réaliser afin d'optimiser ou non spécifiquement le code (par exemple pour obtenir un calcul régulier des

pinceaux après élimination des trajets invalides). Cela aura l'avantage de faciliter le portage sur les nouvelles architectures Xeon Phi qui doivent, elles aussi, bénéficier des instructions AVX-512.

Les travaux présentés ont étudié les performances du calcul de champ sur différentes architectures matérielles. Lors de l'industrialisation, pour tirer parti de ces architectures, il sera nécessaire de maintenir en parallèle plusieurs versions des algorithmes codées chacune avec les outils adaptés au matériel. Les extensions et autres évolutions apportées à l'une des versions devront être reportées vers les autres versions afin de garder le même niveau de fonctionnalités. Cela nécessitera de maintenir des compétences spécifiques à chaque architecture tant qu'une solution d'unification des codes telle qu'OpenCL ne sera pas efficiente.

Les nouvelles architectures matérielles, MIC et GPU, visent à rapprocher les accélérateurs déportés vers la carte mère et/ou le GPP afin de réduire encore plus le coût des transferts mémoire. Cela permettra de lever les contraintes liées aux transferts mémoire des données vers les accélérateurs : la gestion manuelle de la localisation mémoire, la mémoire limitée de l'accélérateur et le temps de transfert. Cependant, ces nouveaux paradigmes posent des questions qui restent en suspens sur l'impact qu'auront ces nouvelles architectures sur le code existant, par exemple : les transferts sont aujourd'hui gérés de manière explicites mais seront-ils encore nécessaires/corrects ? Les notions d'hôte et de *device* seront-elles encore significatives ? La manière d'équilibrer la charge de calcul entre les cœurs sera-t-elle encore la même ?

Dans cette optique, les choix concernant l'usage de Boost.SIMD, CUDA et des bibliothèques spécialisées semblent pertinents. A défaut de profiter immédiatement des nouvelles architectures, les performances ne devraient pas être dégradées et l'évolution des implémentations sera aisée.

8.1.4.2 Contraintes de l'intégration

Les développements réalisés permettent d'accélérer les calculs de champ dans le contexte d'un usage classique de CIVA, c'est à dire sur une station de calcul. Les utilisateurs n'étant pas des spécialistes en informatique mais des experts en CND, il est nécessaire de ne pas les importuner avec les considérations techniques à l'usage du logiciel. Il faudra ainsi cibler les différentes architectures disponibles sur la machine et choisir de manière autonome pour l'exécution le matériel à utiliser pour obtenir de meilleures performances. Cette détection sera réalisée à l'exécution et de manière dynamique suivant la configuration de contrôle (choix du matériel, choix de l'algorithme).

Cela implique entre autres de détecter les paramètres de puissance et de mémoire disponibles, et plus précisément :

- sur GPP et MIC, détecter les jeux d'instructions vectorielles supportées ainsi que le nombre de cœurs de calcul disponible ;
- pour les GPU, détecter la présence ou non d'un GPU de calcul et de la compatibilité matérielle de ce dernier avec les binaires de calcul (*Compute Capability*, *c.f.* section 3.1.5.2), vérifier si la mémoire disponible est suffisante pour la configuration traitée.

Il s'agit d'effectuer l'auto adaptabilité du code à la machine sur laquelle il est exécuté. De plus, si la puissance d'un seul matériel n'est pas suffisante pour obtenir un calcul interactif, des questions de calcul hybride et de calcul multi-GPU se posent (répartition de charge entre les accélérateurs).

8.1.4.3 Simulation rapide

La rapidité des calculs pourra être exploitée dans plusieurs procédés de contrôle :

- en lieu et place du code existant CIVA sur les configurations compatibles avec le modèle rapide ;

- afin de déterminer la couverture de zone, c'est à dire étudier l'insonification d'une zone au moyen de plusieurs calculs de champ (déplacement et/ou balayage électronique) ;
- pour réaliser l'étalonnage et/ou la mise au point d'un contrôle (mesure des caractéristiques du champ : orientation, largeur de la tache focale, distance du champ proche...) ;
- en donnée d'entrée pour des algorithmes de reconstruction de type FTP afin d'enrichir les informations et obtenir des images reconstruites de meilleure qualité ;
- en tant qu'outils pour le calcul des lois de retards d'un capteur multi-éléments.

Des travaux ont déjà été réalisés à la marge de cette thèse pour coupler le modèle de champ rapide avec un modèle d'interaction faisceau/défaut de type Kirchoff présentant lui aussi une bonne régularité. Les résultats préliminaires de ces travaux ont été présentés à la conférence QNDE 2013 [LRLC13].

8.1.4.4 Simulation interactive

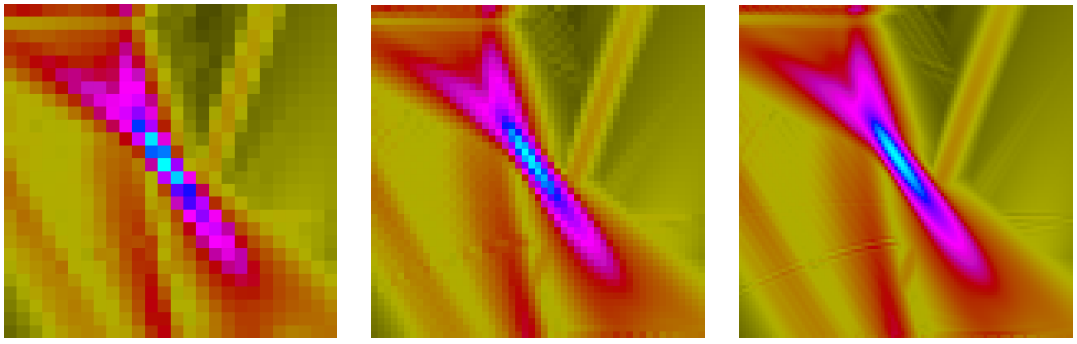
Les performances obtenues permettent de réaliser des simulations de champ ultrasonore en des temps interactifs (25 images par seconde), ce qui représente une avancée majeure dans le cadre du contrôle non destructif. Cela va permettre une plus grande facilité d'utilisation des simulations dans un grand nombre de contextes, par exemple :

- la formation d'opérateurs ;
- l'accélération du temps de mise au point de nouveaux contrôles (en particulier le choix des caractéristiques d'un capteur) ;
- le calcul interactif de lois de retards en permettant la mise en place d'une boucle de rétroaction très rapide entre le champ simulé et la position théorique du point de focalisation (par exemple, pour observer des effets de comportements de déformation du faisceau non intuitifs)[2].

Sur les configurations les plus complexes, un calcul progressif pourra permettre de conserver un temps de réponse satisfaisant tout en offrant une simulation dont la précision se raffinera en quelques passes. Des points de champ constituant un sous ensemble de la zone simulée seront calculés à chaque passe et l'affichage présentera un résultat interpolé sur l'ensemble de la zone de champ à partir des valeurs calculées. La figure 8.1 illustre le calcul progressif d'une zone de champ et des possibilités de lissage des résultats par interpolation bicubique. Idéalement, il sera possible d'activer ou non cette progressivité automatiquement par rétroaction suite à une mesure du temps de calcul sur la configuration en cours.

Enfin, il convient de remarquer que les premières présentations de cet outil ont créé un intérêt auprès des utilisateurs. De nouveaux usages de la simulation interactive vont naître de la mise à disposition de ces outils dans la plateforme CIVA.

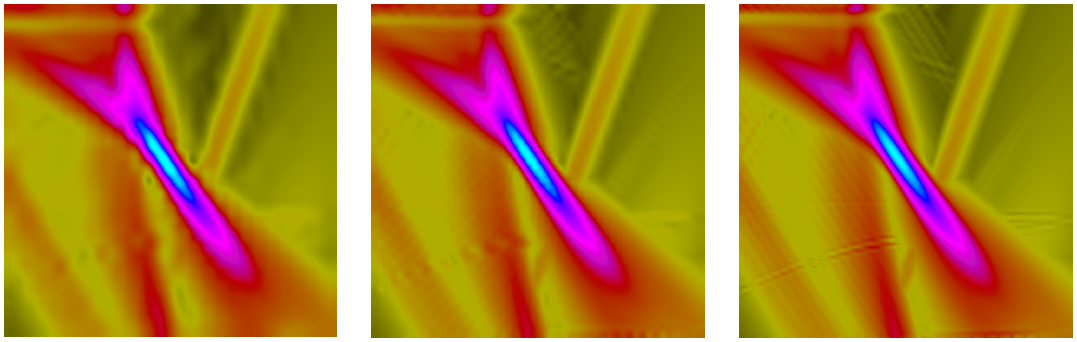
[2] Un démonstrateur GPP du calcul de champ rapide a été mis au point afin de présenter les résultats d'optimisation obtenus.



(a) 26×26 points = 6.6%

(b) 51×51 points = 25.5% (+ 18.9% calculés)

(c) 101×101 points = 100% (+ 74.5% calculés)



(d) 26×26 points = 6.6%

(e) 51×51 points = 25.5% (+ 18.9% calculés)

(f) 101×101 points = 100% (+ 74.5% calculés)

FIGURE 8.1 – Calcul progressif du champ 101×101 points - Pièce plane, focalisation électronique à 45°, modes L et T directs et demi-bonds (sans interpolation *a, b et c*, avec interpolation bicubique *d, e et f*)

Références bibliographiques

[Bai91] David H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, 08/1991 1991.

[Bai11] David H. Bailey. 12 ways to fool the masses : Fast forward to 2011. Manycore and Accelerator-based High-performance Scientific Computing Workshop, University of California, Berkeley, January 2011. http ://www.davidhbailey.com/dhbtalks/dhb-12ways.pdf.

[BBFT14] James Brodman, Dmitry Babokin, Ilia Filippov, and Peng Tu. Writing scalable simd programs with ispc. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 25–32, New York, NY, USA, 2014. ACM.

[CABB13] M. R. Cherry, J. C. Aldrin, T. Boehnlein, and J. L. Blackshire. A parallel mixed 3D grid/explicit FEM scheme for computing elastic wave propagation on a GPU using an irregular mesh. In D. O. Thompson and D. E. Chimenti, editors, *American Institute of Physics Conference Series*, volume 1511 of *American Institute of Physics Conference Series*, pages 75–82, January 2013.

[CC80] Pierre Curie and Jacques Curie. Développement, par pression, de l'électricité polaire dans les cristaux hémièdres à faces inclinées. *Comptes Rendus de l'Académie des Sciences*, 91 :294–295, 1880.

[CSLC14] W. Choi, E. Skelton, M.J.S. Lowe, and R.V. Craster. Development of efficient hybrid finite element modelling for simulation of ultrasonic non-destructive evaluation. In *Proceedings of the 11th European Conference on Non-Destructive Testing (ECNDT 2014)*, 2014.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19 :297–301, 1965.

[DG13] J. Dziewierz and A. Gachagan. Correspondence : Computationally efficient solution of snell's law of refraction. *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on*, 60(6) :1256–1259, June 2013.

[DRLP13] N. Dominguez, G. Rougeron, S. Leberre, and R. Pautel. Phased array ultrasonic processing for enhanced and affordable diagnosis. In D. O. Thompson and D. E. Chimenti, editors, *American Institute of Physics Conference Series*, volume 1511 of *American Institute of Physics Conference Series*, pages 833–840, January 2013.

[EGF+12] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost.SIMD : Generic programming for portable SIMDization. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 431–432, New York, NY, USA, 2012. ACM.

[Far14] Farzad. Cuda atomic operation performance in different scenarios, March 2014.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, Sept 1972.

[Gen99] Nicolas Gengembre. *Modélisation du champ ultrasonore rayonné dans un solide anisotrope et hétérogène par un traducteur immergé, Thèse.* PhD thesis, Université de Paris 07, Paris, FRANCE, 1999.

[Gen03] Nicolas Gengembre. Pencil method for ultrasonic beam computation. In *World Congress on Ultrasonics*, pages 1533–1536, Paris, France, 2003.

[GLC06] Nicolas Gengembre, Alain Lhemery, and Pierre Calmon. *Matériaux et acoustiques 2 - propagation des ondes acoustiques 2*, chapter La méthode des pinceaux, pages 45–68. Lavoissier, 2006.

[Hag10] Georg Hager. Fooling the masses with performance results on parallel computers. Hager's blog, April 2010.

[HNS+12] Yoshiyasu Hirose, Miki Nagano, Yukihiro Sakai, Yoshikazu Iriya, and Yasushi Ikegami. Accelerating calculation for nde simulator of fem by using multi-gpgpus and its procedures for evaluation. *AIP Conference Proceedings*, 1430(1) :1990–1997, 2012.

[Hut14] Peter Huthwaite. Accelerated finite element elastodynamic simulations using the GPU. *Journal of Computational Physics*, 257, Part A :687 – 707, 2014.

[LCR+14] Jason Lambert, Hamza Chouh, Gilles Rougeron, Vincent Bergeaud, Sylvain Chatillon, Lionel Lacassagne, Jean-Claude Iehl, Jean-Philippe Farrugia, and Victor Ostromoukhov. Interactive ultrasonic field simulation for non-destructive testing. In *EGSR*, 2014.

[LEHZ+14] Lionel Lacassagne, Daniel Etiemble, Ali Hassan Zahraee, Alain Dominguez, and Pascal Vezolle. High level transforms for simd and low-level computer vision algorithms. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, WPMVP '14, pages 49–56, New York, NY, USA, 2014. ACM.

[Lhe05] Alain Lhemery. "contrôle non destructif des matériaux, polycopié de cours p01209/11". Supélec, 2005.

[LLLB+13] Jason Lambert, Lionel Lacassagne, Stéphane Le Berre, Gilles Rougeron, and Sylvain Chatillon. High performance simulation of ultrasonic fields for non destructive testing. In *Proceedings of the Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, 2013.

[LM01] E.S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 43–43, Nov 2001.

[LPG+12] Jason Lambert, Antoine Pedron, Guillaume Gens, Franck Bimbard, Lionel Lacassagne, and Ekaterina Iakovleva. Performance evaluation of total focusing method on GPP and GPU. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP), Karlsruhe, Germany, October 23-25, 2012*, pages 1–8. IEEE, 2012.

[LRC15] Jason Lambert, Gilles Rougeron, and Lionel Chatillon, Sylvain Lacassagne. Interactive ultrasonic field simulation for non destructive testing. In *Proceedings of the 2015 Conference on Quality Control by Artificial Vision (QCAV), Le Creusot, France, June 3-5, 2015*, 2015.

[LRL14] Jason Lambert, Gilles Rougeron, and Lionel Lacassagne. Modèles et méthodes de simulation de contrôle non destructif par ultrason massivement parallèles. In *COFREND - PROCEEDINGS POSTERS DOCTORANTS*, 2014.

[LRLC13] Jason Lambert, Gilles Rougeron, Lionel Lacassagne, and Sylvain Chatillon. A fast ultrasonic simulation tool based on massively parallel implementations. In *40th ANNUAL REVIEW OF PROGRESS IN QUANTITATIVE NONDESTRUCTIVE EVALUATION*, 2013.

[Luc13] Lucintel. *Growth Opportunities in Global Non-Destructive Testing Equipment Market 2013-2018 : Trend, Forecasts and Opportunity Analysis.* Lucintel, 2013.

[Lui14] Justin Luitjens. Faster parallel reductions on kepler, Febuary 2014.

[Mü31] O Mülhäuser. Verfahren zur zustundsbistimmung von werkstoffen, besonders zur ermittlung von fehlern darin, January 1931.

[MGHM+11] Oscar Martínez-Graullera, Ricardo T Higuti, Carlos J Martín, Luis G Ullate, David Romero, and Montserrat Parrilla. Improving synthetic aperture image by image compounding in beamforming process. *AIP Conference Proceedings*, 1335(1) :728–735, 2011.

[Mic13] Microsoft Corporation. C++ amp : Language and programming model. Technical Report Version 1.2, Microsoft Corporation, December 2013.

[MIV13] Miguel Molero and Ursula Iturrarán-Viveros. Accelerating numerical modeling of wave propagation through 2-d anisotropic materials using opencl. *Ultrasonics*, 53(3) :815 – 822, 2013.

[Moo95] Gordon E. Moore. Lithography and the future of moore's law. Technical report, Intel, 1995.

[NRBK12] C. A. Nahas, P. Rajagopal, K. Balasubramaniam, and C. V. Krishnamurthy. Graphics processing unit based computation for NDE applications. In *American Institute of Physics Conference Series*, volume 1430 of *American Institute of Physics Conference Series*, pages 1998–2005, May 2012.

[nVi09] nVidia Corporation. Whitepaper, nvidia's next generation cuda compute architecture : Fermi. Technical report, nVidia Corporation, 2009.

[nVi12] nVidia Corporation. Whitepaper, nvidia's next generation cuda compute architecture : Kepler tm gk110. the fastest, most efficient hpc architecture ever built. Technical report, nVidia Corporation, 2012.

[nVi15] nVidia Corporation. NVIDIA CUDA compute unified device architecture programming guide. Technical report, nVidia Corporation, 2015.

[Pak11] Scott Pakin. Ten ways to fool the masses when giving performance results on gpus, December 2011.

[Pau13] Romain Pautel. Reconstruction d'image de données ultrasonores sur gpu - stage m2. Technical report, CEA - UTBM, 2013.

[Ped13] Antoine Pedron. *Développement d'algorithmes d'imagerie et de reconstruction sur architectures à unités de traitements parallèles pour des applications en contrôle non destructif.* PhD thesis, Université de Paris-Sud ; Ecole doctorale Sciences et Technologies de l'Information, des Télécommunications et des Systèmes (STITS - ED 422), may 2013.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition : The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[RID12] Frédéric Reverdy, G Ithurralde, and N Dominguez. Advanced ultrasonic 2d phased-array probes. In *proceedings of the World Conference of NDT 2012*, 2012.

[RLI+14] G. Rougeron, J. Lambert, E. Iakovleva, L. Lacassagne, and N. Dominguez. Implementation of a GPU accelerated total focusing reconstruction method within CIVA software. In *American Institute of Physics Conference Series*, volume 1581 of *American Institute of Physics Conference Series*, pages 1983–1990, February 2014.

[RLMGM+11] D. Romero-Laorden, Oscar Martinez-Graullera, C. J. Martín, M. Pérez, and Luis G. Ullate. Field modelling acceleration on ultrasonic systems using graphic hardware. *Computer Physics Communications*, 182(3) :590–599, 2011.

[RLMGMA+11] David Romero-Laorden, Oscar Martínez-Graullera, Carlos J Martín-Arguedas, Manuel Pérez-López, and Luis Gómez-Ullate. Paralelización de los procesos de conformación de haz para la implementación del total focusing method. In *12 Congreso Español de END*, 2011.

[RLMGMA+12] D. Romero-Laorden, O. Martínez-Graullera, C.J. Martín-Arguedas, M. Pérez, and L.G. Ullate. Técnicas {GPGPU} para acelerar el modelado de sistemas ultrasónicos. *Revista Iberoamericana de Automática e Informática Industrial {RIAI}*, 9(3) :282 – 289, 2012.

[RMGM+09] D. Romero, O. Martinez-Graullera, C.J. Martin, R.T. Higuti, and A. Octavio. Using gpus for beamforming acceleration on saft imaging. In *Ultrasonics Symposium (IUS), 2009 IEEE International*, pages 1334–1337, Sept 2009.

[RMGMU10] David Romero, Oscar Martinez-Graullera, Carlos J. Martin, and Luis G. Ullate. GPGPU techniques to accelerate modelling in NDE. *AIP Conference Proceedings*, 1211 :1991–1998, 2010.

[SNHI] Yukihiro Sakai, Miki Nagano, Yoshiyasu Hirose, and Yasushi Ikegami. An investigation of the optimal simulation condition for NDE with the help of high-speed and high-accuracy FEM simulator. In *JRC-NDE 2012 - Modelling*, pages 308–313.

[Sut05] Herb Sutter. The free lunch is over : A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005.

[SWD+12] Mark Sutcliffe, Miles Weston, Ben Dutton, Peter Charlton, and Kelvin Donne. Real-time full matrix capture for ultrasonic non-destructive testing with acceleration of post-processing through graphic hardware. *NDT & E International*, 51(6) :16 – 23, 2012.

[Syk09] Stanislav Sykora. Three-point interpolation of a real function, Sep 2009.

[VCR+11] Francois Varray, Christian Cachard, Alessandro Ramalli, Piero Tortoli, and Olivier Basset. Simulation of ultrasound nonlinear propagation on gpu using a generalized angular spectrum method. *EURASIP Journal on Image and Video Processing*, 2011(1) :17, 2011.

[Vol10] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010.

[WZHB09] Bruce Walter, Shuang Zhao, Nicolas Holzschuch, and Kavita Bala. Single scattering in refractive media with triangle mesh boundaeries. *ACM Transactions on Graphics*, 28(3), aug 2009.

Publications

Communications et conférences

10/2012, DASIP Jason Lambert, Antoine Pedron, Guillaume Gens, Franck Bimbard, Lionel Lacassagne, and Ekaterina Iakovleva. Performance evaluation of total focusing method on GPP and GPU. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP), Karlsruhe, Germany, October 23-25, 2012*, pages 1–8. IEEE, 2012

07/2013, QNDE Jason Lambert, Gilles Rougeron, Lionel Lacassagne, and Sylvain Chatillon. A fast ultrasonic simulation tool based on massively parallel implementations. In *40th ANNUAL REVIEW OF PROGRESS IN QUANTITATIVE NONDESTRUCTIVE EVALUATION*, 2013

10/2013, SNA+MC Jason Lambert, Lionel Lacassagne, Stéphane Le Berre, Gilles Rougeron, and Sylvain Chatillon. High performance simulation of ultrasonic fields for non destructive

testing. In *Proceedings of the Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, 2013

05/2014, COFREND Jason Lambert, Gilles Rougeron, and Lionel Lacassagne. Modèles et méthodes de simulation de contrôle non destructif par ultrason massivement parallèles. In *COFREND - PROCEEDINGS POSTERS DOCTORANTS*, 2014

06/2014, Poster présenté à EGSR Jason Lambert, Hamza Chouh, Gilles Rougeron, Vincent Bergeaud, Sylvain Chatillon, Lionel Lacassagne, Jean-Claude Iehl, Jean-Philippe Farrugia, and Victor Ostromoukhov. Interactive ultrasonic field simulation for non-destructive testing. In *EGSR*, 2014

06/2015, QCAV Jason Lambert, Gilles Rougeron, and Lionel Chatillon, Sylvain Lacassagne. Interactive ultrasonic field simulation for non destructive testing. In *Proceedings of the 2015 Conference on Quality Control by Artificial Vision (QCAV), Le Creusot, France, June 3-5, 2015*, 2015

Annexes

A Calcul des coefficients de Fresnel entre deux milieux isotropes

A.1 Généralités

L'onde considérée est une onde plane de déplacement se propageant dans la direction définie par le vecteur unitaire $\vec{n}$:

$$\vec{u}(\vec{x},t) = A.\vec{p}.\exp(j\omega(\frac{\vec{n}}{v}.\vec{x} - t))$$

où A est l'amplitude, $\vec{p}$ la polarisation et v la vitesse de phase de l'onde.

Dans le plan d'incidence Oxz, en notant $\vec{S} = \frac{\vec{n}}{v}$ le vecteur lenteurs, les déplacements du mode incident et des modes générés à l'interface plane $z = 0$ peuvent être écrites sous la forme :

pour les modes L : $\begin{cases} u_{Lx} = A_L.v_L.S_x.\exp^{j\omega.S_{zL}z}.\exp^{j\omega.(S_x.x-t)} \\ u_{Lz} = A_L.v_L.S_{zL}.\exp^{j\omega.S_{zL}z}.\exp^{j\omega.(S_x.x-t)} \end{cases}$

pour les modes T : $\begin{cases} u_{Tx} = A_T.v_T.S_{zT}.\exp^{j\omega.S_{zT}z}.\exp^{j\omega.(S_x.x-t)} \\ u_{Tz} = -A_T.v_T.S_x.\exp^{j\omega.S_{zT}z}.\exp^{j\omega.(S_x.x-t)} \end{cases}$ (de type SV i.e. dans le plan d'incidence)

La relation de Snell-Descarte indique que S_x sera commun à tous les modes. Puisque $S = \sqrt{S_x^2 + S_z^2}$, on en déduit les relations :

si $S_x \leq S$ alors :

$S_z^+ = \sqrt{S^2 - S_x^2}$ pour les modes réfractés et $S_z^- = -\sqrt{S^2 - S_x^2}$ pour les modes réfléchis

si $S_x > S$ alors :

$S_z^+ = j\sqrt{S_x^2 - S^2}$ pour les modes réfractés et $S_z^- = -j\sqrt{S_x^2 - S^2}$ pour les modes réfléchis.

En milieu isotrope, les contraintes à la surface sont données par :

$$\begin{cases} \sigma_{zz} = \rho[v_L^2\frac{\partial u_z}{\partial z} + (v_L^2 - 2v_T^2)\frac{\partial u_x}{x}] \\ \sigma_z x = \rho v_T^2[\frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z}] \end{cases}$$

On peut donc exprimer les contraintes de la manière suivante :

pour les modes L : $\begin{cases} \sigma_{zz} = \rho.A_L.v_L[1 - 2.v_T^2.S_x^2] \\ \sigma_{zx} = \rho.A_L.v_L[2.v_T^2.S_x.S_{zL}] \end{cases}$

pour les modes T : $\begin{cases} \sigma_{zz} = -\rho.A_T.v_T[2.v_T^2.S_x.S_{zT}] \\ \sigma_{zx} = \rho.A_T.v_T[1 - 2.v_T^2.S_x^2] \end{cases}$ (de type SV i.e. dans le plan d'incidence)

A partir des conditions aux limites de l'interface séparant deux milieux, on obtient un système d'équations linéaires $Mx = b$ où M est la matrice des continuités et le vecteur x donné par :

$$x_L = \frac{1}{A_L}\begin{pmatrix} A_{L1} \\ A_{T1} \\ A_{L2} \\ A_{T2} \end{pmatrix} = \begin{pmatrix} R_{LL} \\ R_{LT} \\ T_{LL} \\ T_{LT} \end{pmatrix}$$

ou

$$x_T = \frac{1}{A_T}\begin{pmatrix} A_{L1} \\ A_{T1} \\ A_{L2} \\ A_{T2} \end{pmatrix} = \begin{pmatrix} R_{TL} \\ R_{TT} \\ T_{TL} \\ T_{TT} \end{pmatrix}$$

où R et T sont respectivement les coefficients Fresnel de réflexion et transmission.

Ce système peut être résolu par la méthode de Cramer : la $i^{ème}$ coordonnée x_i de la solution est donnée par :

$$x_i = \frac{\det(M_i)}{\det(M)}$$

où M_i est la matrice carrée formée en remplaçant la colonne i de M par le vecteur colonne b.

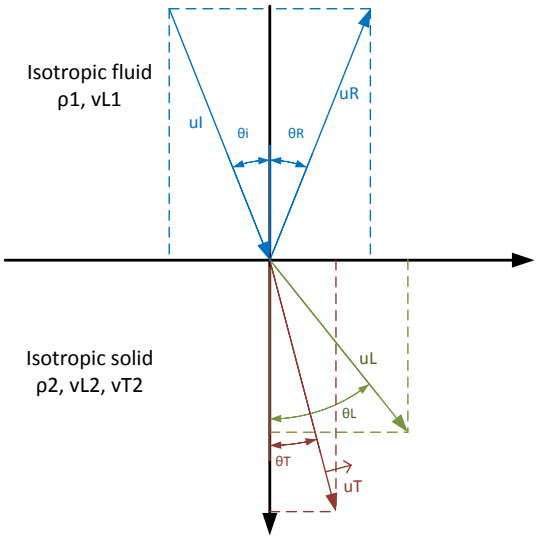


FIGURE 1 – Interface liquide/solide isotrope avec mode incident L

A.2 Cas d'une interface liquide/solide

Dans le cas d'une interface liquide/solide isotrope, les conditions aux limites imposent

- la continuité de la composante normale du déplacement ($u_z^{(2)} = u_z^{(1)}$) :

$$-v_{L1}.S^{-}_{zL_1}R_{LL} + v_{L2}.S^{+}_{zL2}.T_{LL} - v_{T2}.S_x.T_{LT} = v_{L1}.S^{+}_{zL1}$$

- la continuité de la contrainte normale ($\sigma_{zz}^{(2)} = \sigma_{zz}^{(1)}$) :

$$-\rho_1.v_{L1}.R_{LL} + \rho_2.v_{L2}(1 - 2.v^2_{T2}.S^2_x)T_{LL} - \rho_2.v_{T2}(2.v^2_{T2}.S_x.S^{+}_{zT2})T_{LL} = \rho_1.v_{L1}$$

- la nullité de la contrainte tangentielle ($\sigma_{xz}^{(2)} = 0$) :

$$\rho_2.v_{L2}(2.v^2_{T2}.S_x.S^{+}_{zL2}).T_{LL} + \rho_2.v_{T2}(1 - 2.v^2_{T2}.S^2_x)T_{LL} = 0$$

Ce qui amène au système d'équations suivant (écrit sous forme matricielle $Mx = b$) :

$$M = \begin{pmatrix} -v_{L1}.S^1_{zL1} & v_{L2}.S^{+}_{zL2} & -v_{T2}.S_x \\ -\rho_1.v_{L1} & \rho_2.v_{L2}(1 - 2.v^2_{T2}.S^2_x) & -\rho_2.v_{T2}(2.v^2_{T2}.S_x.S^{+}_{zT2}) \\ 0 & \rho_2.v_{L2}(2.v^2_{T2}.S_x.S^{+}_{zL2}) & \rho_2.v_{T2}(1 - 2.v^2_{T2}.S^2_x) \end{pmatrix}, x = \begin{pmatrix} R_{LL} \\ T_{LL} \\ T_{LT} \end{pmatrix}$$

et

$$b = \begin{pmatrix} v_{L1}.S^{+}_{zL1} \\ \rho_1.v_{L1} \\ 0 \end{pmatrix}$$

Ce système peur s'écrire sous la forme $M'x' = b'$ avec

$$M' = \begin{pmatrix} S_{z1} & S_{zL2} & -S_x \\ -\rho_1 & \Phi_2\beta_2 & -2\Phi_2.S_x.S_{zT2} \\ 0 & 2\Phi_2.S_x.S_{zL2} & \Phi_2.\beta_2 \end{pmatrix}, x' = \begin{pmatrix} R_{LL} \\ T_{LL}\frac{S_{L1}}{S_{L2}} \\ T_{LT}\frac{S_{L1}}{S_{T2}} \end{pmatrix}, \text{ et } b' = \begin{pmatrix} S_{z1} \\ \rho_1 \\ 0 \end{pmatrix}$$

en posant $S_{z1} = -S^{-}_{zL1}$, $S_{T1} = \frac{1}{v_{T1}}$, $S_{T2} = \frac{1}{v_{T2}}$,$S_{L1} = \frac{1}{v_{L1}}$, $S_{L2} = \frac{1}{v_{L2}}$, $\Phi_2 = \frac{\rho_2}{S^2_{T2}}$, $\beta_1 = (S^2_{T1} - 2S^2_x)$ et $\beta_2 = (S^2_{T2} - 2S^2_x)$. La solution au système est alors donné par :

$$
\begin{aligned}
R_{LL} &= \frac{\beta^2 + 4.S^2_x.S_{zL2}.S_{zT2} - \rho_1.S_{zL2}.S^2_{T2}/(S_{z1}.\Phi_2)}{\beta^2 + 4.S^2_x.S_{zL2}.S_{zT2} + \rho_1.S_{zL2}.S^2_{T2}/(S_{z1}.\Phi_2)} \\
T_{LL} &= \frac{S_{L2}}{S_{L1}} \frac{2.\rho_1.\beta_2}{\beta_2^2 + 4.S^2_x.S_{zL2}.S_{zT2} + \rho_1.S_{zL2}.S^2_{T2}/(S_{z1}.\Phi_2)} \\
T_{LT} &= \frac{S_{T2}}{S_{L1}} \frac{-4.\rho_1.S_x.S_{zL2}}{\beta_2^2 + 4.S^2_x.S_{zL2}.S_{zT2} + \rho_1.S_{zL2}.S^2_{T2}/(S_{z1}.\Phi_2)}
\end{aligned}
$$

A.3 Autres types d'interfaces

Pour les autres types d'interfaces, les coefficients de Fresnel sont obtenus par le même raisonnement. En particulier, dans ces travaux, les interfaces solide/liquide, solide/solide, et solide/vide ont été considérées.

B Configurations du benchmark

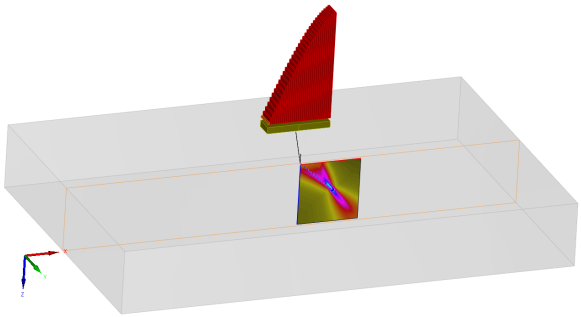
B.1 Présentation des configurations

Cette sous-section présente l'ensemble des 21 configurations utilisées afin d'évaluer les performances du calcul de champ rapide. Ces configurations ont été conçues afin de faire varier les paramètres **algorithmiques** indépendamment les uns des autres (*c.f.* table 1). Les 18 premières configurations sont dérivées d'une première configuration de référence (configuration 1), et font varier la complexité de la simulation suivant différents axes. Les configurations 19 à 21 permettent d'étudier l'effet des lois de retards (focalisation ou non à différentes profondeurs).

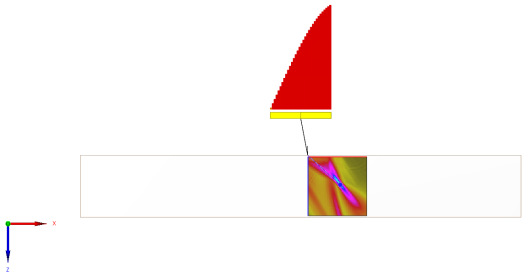
Ces configurations ont été mises au point pour qu'elles conservent autant que possible un sens d'un point de vue métier. Il est à noter que pour les configurations pour lesquelles la complexé géométrique augmente (surfaces d'entrée et surfaces de fond multiples), les problèmes d'occlusions ont été évités pour ne pas sortir du domaine d'application du modèle développé. *c.f.* 2 et 3.

ID	N_{points}	$N_{capteurs}$	$size$	N_{surfin}	$N_{modes\_directs}$	$N_{surfback}$	$N_{modes\_rebonds}$	$\frac{\text{Total Pinceaux}}{N_{capteurs}}$
Configuration de Base								
1	101×101	320	1024	1	$\{L\} = 1$	1	0	1
Finesse de l'échantillonnage capteur								
2	101×101	**320**	1024	1	$\{L\} = 1$	1	0	1
3	101×101	**320**	1024	1	$\{L\} = 1$	1	0	1
4	101×101	**1280**	1024	1	$\{L\} = 1$	1	0	1
Finesse de l'échantillonnage spatial de la zone simulée								
5	201×201	320	1024	1	$\{L\} = 1$	1	0	1
6	401×401	320	1024	1	$\{L\} = 1$	1	0	1
Variation de l'échantillonnage temporel des signaux								
7	101×101	320	**2048**	1	$\{L\} = 1$	1	0	1
8	101×101	320	**4096**	1	$\{L\} = 1$	1	0	1
Augmentation du nombre de modes								
9	101×101	320	1024	1	$\{L, T\} = 2$	1	0	**2**
10	101×101	320	1024	1	$\{L, T\} = 2$	1	$\{LrbL, TrbT\} = 2$	**4**
Augmentation du nombre de surfaces d'entrées								
11	101×101	320	1024	**3**	$\{L\} = 1$	1	0	**3**
12	101×101	320	1024	**10**	$\{L\} = 1$	1	0	**10**
Augmentation du nombre de surfaces de fond								
13	101×101	320	1024	1	0	**1**	$\{LrbL\} = 1$	**1**
14	101×101	320	1024	1	0	**3**	$\{LrbL\} = 1$	**3**
15	101×101	320	1024	1	0	**10**	$\{LrbL\} = 1$	**10**
Complexification de la géométrie								
16	101×101	320	1024	**3**	0	10	$\{LrbL\} = 1$	**30**
17	101×101	320	1024	**10**	0	10	$\{LrbL\} = 1$	**100**
18	101×101	320	1024	**10**	$\{L\} = 1$	10	$\{LrbL\} = 1$	**110**
Configuration de Base 2 - Mêmes échantillonnages, focalisation L 0°								
19	101×101	320	1024	1	$\{L\} = 1$	1	0	1
Influence des retards (collisions de la focalisation a différentes profondeurs)								
20	101×101	320	1024	1	$\{L\} = 1$	1	0	1
21	101×101	320	1024	1	$\{L\} = 1$	1	0	1

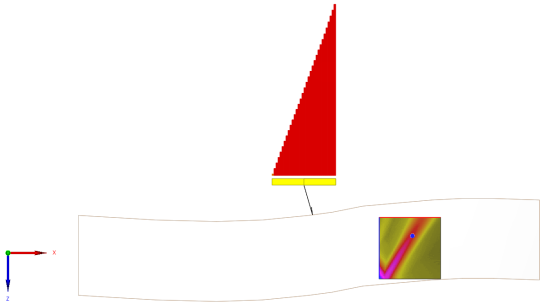
TABLE 1 – Présentation des paramètres numériques des configurations de champ étudies



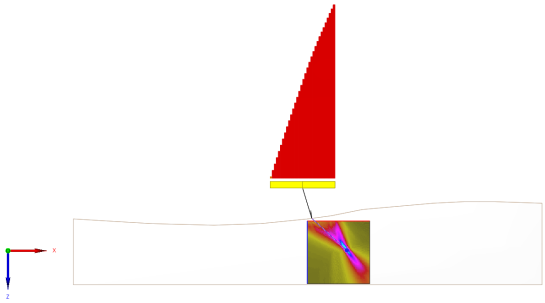
(a) Configuration 01 - référence



(b) Configuration 09 - modes directs L et T

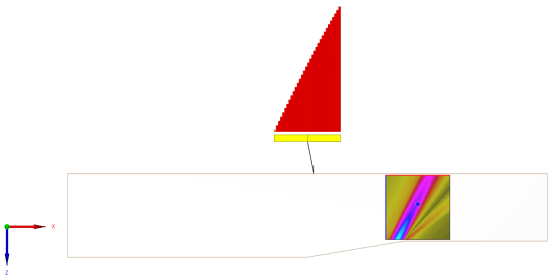


(c) Configuration 11 - 3 entrées, 1 fond, mode direct L

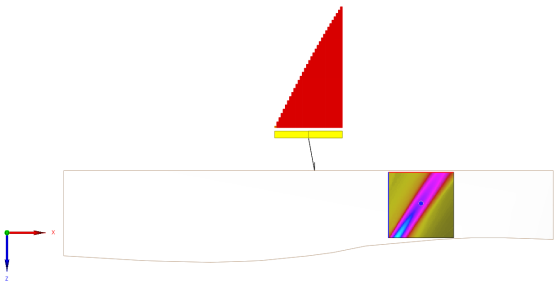


(d) Configuration 12 - 10 entrées, 1 fonds, mode direct L

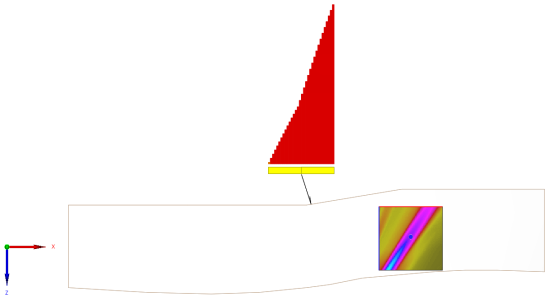
FIGURE 2 – Configurations de champ de complexité géométrique croissante



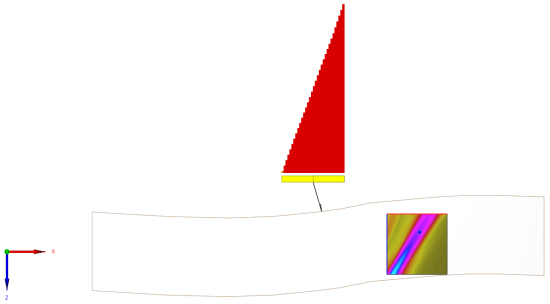
(a) Configuration 14 - 1 entrée, 3 fonds, LrbL



(b) Configuration 15 - 1 entrée, 10 fonds, LrbL



(c) Configuration 16 - 3 entrées, 10 fonds, LrbL



(d) Configuration 17 - 10 entrées, 10 fonds, LrbL

FIGURE 3 – Configurations de champ de complexité géométrique croissante

B.2 Validation des configurations

Le tableau 2 montre que la simulation des configurations du benchmark de test par les codes de simulations de champ rapide sont valides d'un point de vue métier par comparaison avec les résultats de simulation CIVA 11.0.

Configuration	Amax (écart en dB)	Tmax (écart en μs)	Largeur tache focale (erreur relative en %)	Longueur tache focale (erreur relative en %)
1	0.3	0	0.0	0.0
2	0.09	0	0.0	0.0
3	0.09	0	0.0	0.0
4	0.09	0	0.0	0.0
5	0.3	0	0.0	0.0
6	-	-	-	-
7	0.3	0	0.0	0.5
8	0.3	0	0.0	1.1
9	0.3	0	0.0	0.0
10	0.3	0	0.0	0.0
11	0.6	0	3.0	3.0
12	0.7	0	4.9	5.8
13	0.1	0	7.7	2.9
14	0.1	0	2.1	0.0
15	0.1	0	0.9	6.2
16	0.3	0	1.7	2.0
17	0.3	0	0.8	6.1
18	0.2	0	8.0	7.1
19	0.0	0	Capteur non focalisé	Capteur non focalisé
20	0.02	0	4.0	3.6
21	0.1	0	0.0	1.0

TABLE 2 – Mesures de validité des simulations de calcul rapide versus CIVA 11.0 précision 3

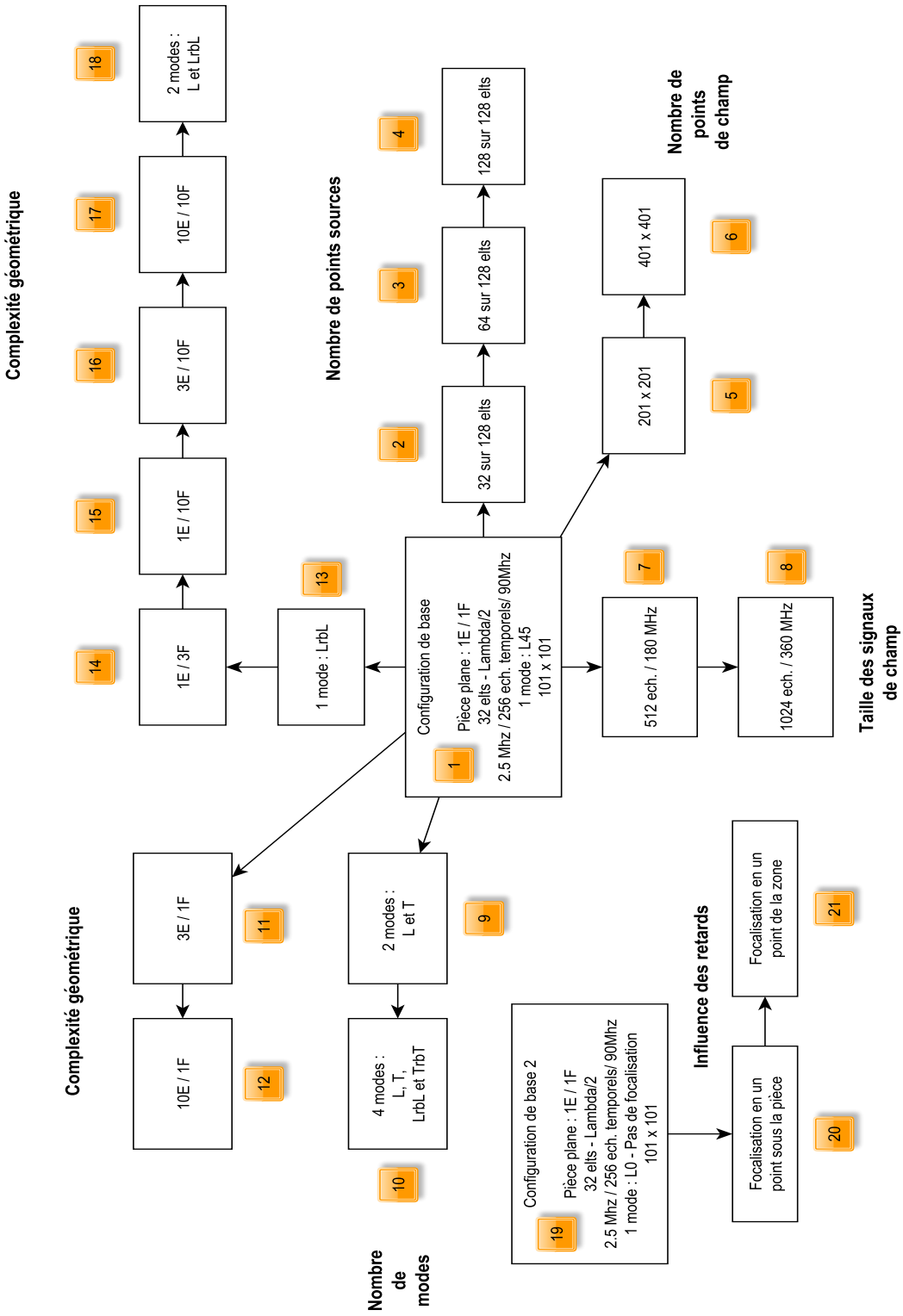


FIGURE 4 – Configurations de champ - arbre de complexité

C Stockage mémoire des données

Cette section présente la manière dont sont stockées en mémoire les différentes informations utiles au calcul de champ. Pour rappel, la notation "tableau" utilisée s'apparente à la notation tableau du $C/C++$:

```
T array[N][M] définit un tableau de N lignes de M colonnes.
array[a][b] = *(array+ (b + a*M))
```

L'intégralité des structures présentées ici, nécessaires au calcul de champ, est stockée en mémoire sous la forme de structure de tableaux. Les présentes sous-sections utilisent la convention suivante :

$$\text{donnée stockée} \rightarrow \begin{cases} \text{sous donnée \#1 datatype} \rightarrow [N]\,[M] \\ \text{sous donnée \#2 datatype} \rightarrow [N]\,[M] \\ \quad ... \\ \text{sous donnée \#i datatype} \rightarrow [N]\,[M] \end{cases}$$

C.1 Points de champ

Les points de champ sont simplement stockés sous la forme d'une structure de tableaux de coordonnées. Leur description est fournie en entrée dans un fichier texte, puis interprétée par le simulateur.

$$\text{centre} \rightarrow \begin{cases} x \text{ flottant} \rightarrow [N_{point}] \\ y \text{ flottant} \rightarrow [N_{point}] \\ z \text{ flottant} \rightarrow [N_{point}] \end{cases}$$

C.2 Informations capteur

Les données concernant les échantillons de la surface du capteur sont stockées sous forme de structure de tableaux. Leur description est fournie en entrée dans un fichier texte, puis interprétée par le simulateur.

$$\begin{array}{rcl} \text{centre} & \rightarrow & \begin{cases} x \text{ flottant} \rightarrow [N_{capteur}] \\ y \text{ flottant} \rightarrow [N_{capteur}] \\ z \text{ flottant} \rightarrow [N_{capteur}] \end{cases} \\ \overrightarrow{\text{normale au capteur}} & \rightarrow & \begin{cases} x \text{ flottant} \rightarrow [N_{capteur}] \\ y \text{ flottant} \rightarrow [N_{capteur}] \\ z \text{ flottant} \rightarrow [N_{capteur}] \end{cases} \\ \overrightarrow{du} & \rightarrow & \begin{cases} x \text{ flottant} \rightarrow [N_{capteur}] \\ y \text{ flottant} \rightarrow [N_{capteur}] \\ z \text{ flottant} \rightarrow [N_{capteur}] \end{cases} \\ \overrightarrow{dv} & \rightarrow & \begin{cases} x \text{ flottant} \rightarrow [N_{capteur}] \\ y \text{ flottant} \rightarrow [N_{capteur}] \\ z \text{ flottant} \rightarrow [N_{capteur}] \end{cases} \\ \text{dS} & \rightarrow & \text{flottant} \rightarrow [N_{capteur}] \\ \text{loi de retards} & \rightarrow & \text{flottant} \rightarrow [N_{capteur}] \\ \text{amplitude loi de retards} & \rightarrow & \text{flottant} \rightarrow [N_{capteur}] \end{array}$$

Dans le cas d'un capteur au contact, une surface de contact sera également fournie afin de vérifier les trajets.

C.3 Réponses impulsionnelles élémentaires

Les réponses impulsionnelles élémentaires sont calculées dans un premier temps et stockée en mémoire avant l'étape de sommation. Une réponse impulsionnelle élémentaire est calculée par point de champ, par élément du capteur, par trajet possible. Le nombre de trajets possible est donné par la formule :

$$N_{trajets} = N_{surfin} * (N_{modes\_directs} + N_{surfback} * N_{modes\_indirects})$$

$$\begin{array}{rcl}
\text{amplitude de déplacement} & \rightarrow & \left\{ \begin{array}{l} \text{Re flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \\ \text{Im flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \end{array} \right. \\
\text{temps de vol} & \rightarrow & \text{flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \\
\text{étalement temporel } \Delta t & \rightarrow & \text{flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \\
\overrightarrow{\text{direction}} & \rightarrow & \left\{ \begin{array}{l} x \text{ flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \\ y \text{ flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \\ z \text{ flottant} \rightarrow [N_{trajets}]\,[N_{point}]\,[N_{capteur}] \end{array} \right.
\end{array}$$

C.4 Signaux des Réponses impulsionnelles

Les signaux des réponses impulsionnelles sont des signaux décrivant l'évolution du déplacement du faisceau au cours du temps. Le déplacement est un signal de vecteur complexe, décrit sous la forme de trois signaux complexes pour chacune des trois coordonnées du plan $\{x, y, z\}$.

$$\begin{array}{rcl}
\overrightarrow{sig_{RI}}(t) \cdot x & \rightarrow & \left\{ \begin{array}{l} \text{Re flottant} \rightarrow [N_{point}]\,[size] \\ \text{Im flottant} \rightarrow [N_{point}]\,[size] \end{array} \right. \\
\overrightarrow{sig_{RI}}(t) \cdot y & \rightarrow & \left\{ \begin{array}{l} \text{Re flottant} \rightarrow [N_{point}]\,[size] \\ \text{Im flottant} \rightarrow [N_{point}]\,[size] \end{array} \right. \\
\overrightarrow{sig_{RI}}(t) \cdot z & \rightarrow & \left\{ \begin{array}{l} \text{Re flottant} \rightarrow [N_{point}]\,[size] \\ \text{Im flottant} \rightarrow [N_{point}]\,[size] \end{array} \right.
\end{array}$$

C.5 Signaux de module

Les signaux de modules sont des signaux dans $\mathbb{R}$ et de taille $size$:

$$\text{signaux} \rightarrow \text{flottant}\ [N_{point}]\,[size]$$

C.6 Cartographies d'amplitude et de temps de vol

Ces deux cartographies sont stockées en mémoire sous la forme de deux tableaux de flottants :

$$\text{flottant}\ [N_{point}]$$

D Performances des différentes implémentations des sommations

Sur l'étape 3 du calcul de champ, c'est à dire la sommation des contributions des pinceaux sur la réponse impulsionnelle de déplacement, quatre versions du calcul sont mesurées, il s'agit des versions présentées au paragraphe 5.3.3.1.

- Sommation scalaire : code de référence ;
- Version scalaire différentielle : qui tente de minimiser pour chaque pinceau les accès mémoire ;
- Version SIMD horizontale : qui utilise des registres SIMD pour ajouter plus rapidement les contributions d'un seul pinceau à la réponse impulsionnelle ;
- Version SIMD verticale : qui regroupe les pinceaux par paquet correspondant à la largeur d'un registre SIMD, afin de faire contribuer plusieurs pinceau à un même pas de temps de la réponse impulsionnelle.

Ces quatre versions ont été mesurées sur plusieurs architectures de GPP présentant des jeux d'instructions différents. Cette section d'annexe présente les mesures réalisées.

D.1 Simulations sur Xeon Westmere - SIMD SSE4.2

Configuration	Algo	N=1	N= 12	2× 12 = 24
Config01	Scalaire	1.00	1.00	1.00
	Différentielle	**2.45**	**2.42**	**2.40**
Ratio KO/OK	SIMD Horizontale	2.19	2.18	2.18
-	SIMD Verticale	2.15	2.12	2.19
Config02	Scalaire	1.00	1.00	1.00
	Différentielle	**2.46**	**2.44**	**2.43**
Ratio KO/OK	SIMD Horizontale	2.19	2.17	2.19
-	SIMD Verticale	2.14	2.13	2.19
Config03	Scalaire	1.00	1.00	1.00
	Différentielle	**2.31**	**2.29**	**2.19**
Ratio KO/OK	SIMD Horizontale	2.22	2.21	2.19
-	SIMD Verticale	2.21	2.21	2.18
Config04	Scalaire	1.00	1.00	1.00
	Différentielle	**2.74**	**2.71**	**2.76**
Ratio KO/OK	SIMD Horizontale	2.08	2.08	2.08
-	SIMD Verticale	2.09	2.11	2.19
Config05	Scalaire	1.00	1.00	1.00
	Différentielle	**2.46**	**2.43**	**2.35**
Ratio KO/OK	SIMD Horizontale	2.20	2.19	2.17
-	SIMD Verticale	2.15	2.14	2.15
Config06	Scalaire	1.00	1.00	1.00
	Différentielle	**2.46**	**2.43**	**2.34**
Ratio KO/OK	SIMD Horizontale	2.20	2.19	2.17
-	SIMD Verticale	2.15	2.16	2.14
Config07	Scalaire	1.00	1.00	1.00
	Différentielle	**2.55**	**2.52**	**2.44**
Ratio KO/OK	SIMD Horizontale	2.21	2.20	2.16
-	SIMD Verticale	1.99	2.00	2.04
Config08	Scalaire	1.00	1.00	1.00
	Différentielle	**2.59**	**2.57**	**2.36**
Ratio KO/OK	SIMD Horizontale	2.22	2.20	2.19
-	SIMD Verticale	1.93	1.93	1.99
Config09	Scalaire	1.00	1.00	1.00
	Différentielle	**2.72**	**2.68**	**2.62**
Ratio KO/OK	SIMD Horizontale	2.15	2.15	2.14
-	SIMD Verticale	2.25	2.24	2.26
Config10	Scalaire	1.00	1.00	1.00
	Différentielle	**2.51**	**2.49**	**2.45**
Ratio KO/OK	SIMD Horizontale	2.16	2.15	2.17
-	SIMD Verticale	2.24	2.23	2.29
Config11	Scalaire	1.00	1.00	1.00
	Différentielle	**2.47**	**2.43**	**2.46**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.17
1.83	SIMD Verticale	2.44	2.40	2.44

Configuration	Algo	N=1	N= 12	2× 12 = 24
Config12	Scalaire	1.00	1.00	1.00
	Différentielle	2.29	2.28	2.36
Ratio KO/OK	SIMD Horizontale	2.10	2.10	2.11
8.67	**SIMD Verticale**	**3.53**	**3.35**	**3.34**
Config13	Scalaire	1.00	1.00	1.00
	Différentielle	**2.58**	**2.54**	**2.56**
Ratio KO/OK	SIMD Horizontale	2.14	2.15	2.13
-	SIMD Verticale	2.22	2.22	2.24
Config14	Scalaire	1.00	1.00	1.00
	Différentielle	2.38	2.39	2.42
Ratio KO/OK	SIMD Horizontale	2.12	2.12	2.10
2.69	**SIMD Verticale**	**2.95**	**2.83**	**2.88**
Config15	Scalaire	1.00	1.00	1.00
	Différentielle	2.20	2.20	2.29
Ratio KO/OK	SIMD Horizontale	2.06	2.06	2.04
9.56	**SIMD Verticale**	**4.63**	**4.20**	**4.22**
Config16	Scalaire	1.00	1.00	1.00
	Différentielle	-	2.12	2.27
Ratio KO/OK	SIMD Horizontale	-	2.03	2.03
29.48	**SIMD Verticale**	-	**5.86**	**5.88**
Config17	Scalaire	-	1.00	1.00
	Différentielle	-	2.04	2.17
Ratio KO/OK	SIMD Horizontale	-	2.01	2.01
102.84	**SIMD Verticale**	-	**7.55**	**7.39**
Config18	Scalaire	-	1.00	1.00
	Différentielle	-	2.09	2.18
Ratio KO/OK	SIMD Horizontale	-	2.03	2.03
53.84	**SIMD Verticale**	-	**6.07**	**6.05**
Config19	Scalaire	1.00	1.00	1.00
	Différentielle	**2.37**	**2.36**	**2.31**
Ratio KO/OK	SIMD Horizontale	2.19	2.19	2.21
-	SIMD Verticale	2.12	2.13	2.17
Config20	Scalaire	1.00	1.00	1.00
	Différentielle	**2.37**	**2.34**	**2.33**
Ratio KO/OK	SIMD Horizontale	2.19	2.19	2.21
-	SIMD Verticale	2.14	2.10	2.18
Config21	Scalaire	1.00	1.00	1.00
	Différentielle	**2.37**	**2.33**	**2.31**
Ratio KO/OK	SIMD Horizontale	2.21	2.18	2.19
-	SIMD Verticale	2.16	2.12	2.19

TABLE 3 – Étape 3 : Performances des variantes de sommation sur Westmere / SSE4.2
2 x Intel(R) Xeon(R) CPU X5690 @ 3.47 GHz

D.2 Simulations sur Xeon Sandy Bridge - SIMD AVX

Configuration	Algo	N=1	N= 4	2× 4 = 8
Config01	Scalaire	1.00	1.00	1.00
	Différentielle	2.48	2.47	**2.43**
Ratio KO/OK	SIMD Horizontale	2.21	2.20	2.19
-	SIMD Verticale	**2.50**	**2.47**	2.40
Config02	Scalaire	1.00	1.00	1.00
	Différentielle	**2.51**	**2.48**	**3.96**
Ratio KO/OK	SIMD Horizontale	2.20	2.19	3.51
-	SIMD Verticale	2.44	2.44	3.79
Config03	Scalaire	1.00	1.00	1.00
	Différentielle	2.35	2.33	3.45
Ratio KO/OK	SIMD Horizontale	2.26	2.25	3.44
-	SIMD Verticale	**2.46**	**2.45**	**3.68**
Config04	Scalaire	1.00	1.00	1.00
	Différentielle	2.75	2.76	**2.82**
Ratio KO/OK	SIMD Horizontale	2.09	2.09	2.06
-	SIMD Verticale	**2.82**	**2.81**	2.81
Config05	Scalaire	1.00	1.00	1.00
	Différentielle	**2.48**	2.49	**2.47**
Ratio KO/OK	SIMD Horizontale	2.21	2.21	2.19
-	SIMD Verticale	2.46	**2.50**	2.45
Config06	Scalaire	1.00	1.00	1.00
	Différentielle	**2.49**	2.48	2.40
Ratio KO/OK	SIMD Horizontale	2.20	2.21	2.16
-	SIMD Verticale	**2.49**	**2.49**	**2.45**
Config07	Scalaire	1.00	1.00	1.00
	Différentielle	**2.61**	**2.62**	**2.44**
Ratio KO/OK	SIMD Horizontale	2.24	2.24	2.13
-	SIMD Verticale	2.23	2.26	2.17
Config08	Scalaire	1.00	1.00	1.00
	Différentielle	**2.66**	**2.66**	**2.44**
Ratio KO/OK	SIMD Horizontale	2.25	2.25	2.20
-	SIMD Verticale	2.14	2.14	2.10
Config09	Scalaire	1.00	1.00	1.00
	Différentielle	**2.81**	**2.80**	**2.68**
Ratio KO/OK	SIMD Horizontale	2.17	2.17	2.12
-	SIMD Verticale	2.64	2.66	2.57
Config10	Scalaire	1.00	1.00	1.00
	Différentielle	2.52	2.57	2.44
Ratio KO/OK	SIMD Horizontale	2.15	2.18	2.16
-	SIMD Verticale	**2.62**	**2.59**	**2.50**
Config11	Scalaire	1.00	1.00	1.00
	Différentielle	2.47	2.47	2.45
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.16
1.83	SIMD Verticale	**2.92**	**2.92**	**2.75**

Configuration	Algo	N=1	N= 4	2× 4 = 8
Config12	Scalaire	1.00	1.00	1.00
	Différentielle	2.27	2.27	2.33
Ratio KO/OK	SIMD Horizontale	2.09	2.10	2.11
8.67	SIMD Verticale	**4.46**	**4.42**	**3.99**
Config13	Scalaire	1.00	1.00	1.00
	Différentielle	2.58	2.58	2.61
Ratio KO/OK	SIMD Horizontale	2.13	2.14	2.11
-	SIMD Verticale	**2.84**	**2.84**	**2.79**
Config14	Scalaire	1.00	1.00	1.00
	Différentielle	2.37	2.37	2.46
Ratio KO/OK	SIMD Horizontale	2.12	2.12	2.12
2.69	SIMD Verticale	**3.81**	**3.79**	**3.56**
Config15	Scalaire	1.00	1.00	1.00
	Différentielle	2.18	2.17	2.24
Ratio KO/OK	SIMD Horizontale	2.03	2.05	2.06
9.56	SIMD Verticale	**6.16**	**6.07**	**5.37**
Config16	Scalaire	1.00	1.00	1.00
	Différentielle	2.10	2.09	2.14
Ratio KO/OK	SIMD Horizontale	2.02	2.02	2.02
29.48	SIMD Verticale	**9.44**	**8.98**	**8.08**
Config17	Scalaire	-	-	-
	Différentielle	-	-	-
Ratio KO/OK	SIMD Horizontale	-	-	-
102.84	SIMD Verticale	-	-	-
Config18	Scalaire	-	-	-
	Différentielle	-	-	-
Ratio KO/OK	SIMD Horizontale	-	-	-
53.84	SIMD Verticale	-	-	-
Config19	Scalaire	1.00	1.00	1.00
	Différentielle	**2.39**	**2.37**	**2.36**
Ratio KO/OK	SIMD Horizontale	2.23	2.21	2.16
-	SIMD Verticale	2.34	2.35	2.25
Config20	Scalaire	1.00	1.00	1.00
	Différentielle	2.37	2.37	**2.33**
Ratio KO/OK	SIMD Horizontale	2.20	2.22	2.17
-	SIMD Verticale	**2.38**	**2.37**	2.29
Config21	Scalaire	1.00	1.00	1.00
	Différentielle	2.41	2.37	2.29
Ratio KO/OK	SIMD Horizontale	2.22	2.21	2.19
-	SIMD Verticale	**2.41**	**2.39**	**2.32**

TABLE 4 – Étape 3 : Performances des variantes de sommation sur Sandy Bridge / AVX
1 x Intel(R) Xeon(R) CPU E3-1290 @ 3.60 GHz

D.3 Simulations sur Xeon Ivy Bridge - SIMD AVX

Configuration	Algo	N=1	N= 6	2× 6 = 12
Config01	Scalaire	1.00	1.00	1.00
	Différentielle	**2.43**	**2.42**	**2.42**
Ratio KO/OK	SIMD Horizontale	2.16	2.17	2.15
-	SIMD Verticale	2.40	2.39	2.39
Config02	Scalaire	1.00	1.00	1.00
	Différentielle	**2.44**	**2.45**	**2.47**
Ratio KO/OK	SIMD Horizontale	2.14	2.15	2.14
-	SIMD Verticale	2.37	2.38	2.36
Config03	Scalaire	1.00	1.00	1.00
	Différentielle	2.29	2.28	2.17
Ratio KO/OK	SIMD Horizontale	2.21	2.19	2.17
-	**SIMD Verticale**	**2.40**	**2.37**	**2.29**
Config04	Scalaire	1.00	1.00	1.00
	Différentielle	**2.75**	2.72	**2.82**
Ratio KO/OK	SIMD Horizontale	2.07	2.07	2.06
-	SIMD Verticale	2.74	**2.74**	2.75
Config05	Scalaire	1.00	1.00	1.00
	Différentielle	**2.45**	**2.44**	**2.44**
Ratio KO/OK	SIMD Horizontale	2.17	2.17	2.16
-	SIMD Verticale	2.43	2.43	2.39
Config06	Scalaire	1.00	1.00	1.00
	Différentielle	**2.42**	**2.44**	2.39
Ratio KO/OK	SIMD Horizontale	2.15	2.17	2.14
-	SIMD Verticale	2.41	2.42	**2.41**
Config07	Scalaire	1.00	1.00	1.00
	Différentielle	**2.57**	**2.55**	**2.49**
Ratio KO/OK	SIMD Horizontale	2.18	2.18	2.14
-	SIMD Verticale	2.19	2.19	2.20
Config08	Scalaire	1.00	1.00	1.00
	Différentielle	**2.61**	**2.61**	**2.45**
Ratio KO/OK	SIMD Horizontale	2.18	2.19	2.19
-	SIMD Verticale	2.07	2.07	2.10
Config09	Scalaire	1.00	1.00	1.00
	Différentielle	**2.79**	**2.79**	**2.73**
Ratio KO/OK	SIMD Horizontale	2.12	2.13	2.08
-	SIMD Verticale	2.60	2.61	2.59
Config10	Scalaire	1.00	1.00	1.00
	Différentielle	2.54	2.54	2.46
Ratio KO/OK	SIMD Horizontale	2.13	2.13	2.16
-	**SIMD Verticale**	**2.54**	**2.54**	**2.50**
Config11	Scalaire	1.00	1.00	1.00
	Différentielle	2.44	2.45	2.48
Ratio KO/OK	SIMD Horizontale	2.13	2.14	2.13
1.83	**SIMD Verticale**	**2.85**	**2.83**	**2.74**

Configuration	Algo	N=1	N= 6	2× 6 = 12
Config12	Scalaire	1.00	1.00	1.00
	Différentielle	2.27	2.26	2.30
Ratio KO/OK	SIMD Horizontale	2.08	2.08	2.10
8.67	**SIMD Verticale**	**4.31**	**4.23**	**3.91**
Config13	Scalaire	1.00	1.00	1.00
	Différentielle	2.59	2.58	2.65
Ratio KO/OK	SIMD Horizontale	2.11	2.13	2.11
-	**SIMD Verticale**	**2.76**	**2.78**	**2.75**
Config14	Scalaire	1.00	1.00	1.00
	Différentielle	2.36	2.36	2.43
Ratio KO/OK	SIMD Horizontale	2.09	2.10	2.11
2.69	**SIMD Verticale**	**3.71**	**3.64**	**3.50**
Config15	Scalaire	1.00	1.00	1.00
	Différentielle	2.17	2.18	2.23
Ratio KO/OK	SIMD Horizontale	2.03	2.04	2.04
9.56	**SIMD Verticale**	**5.92**	**5.82**	**5.29**
Config16	Scalaire	1.00	1.00	1.00
	Différentielle	2.09	2.08	2.15
Ratio KO/OK	SIMD Horizontale	2.02	2.01	2.02
29.48	**SIMD Verticale**	**9.03**	**8.47**	**7.74**
Config17	Scalaire	1.00	1.00	1.00
	Différentielle	-	2.03	2.05
Ratio KO/OK	SIMD Horizontale	-	2.01	2.01
102.84	**SIMD Verticale**	-	**11.91**	**10.57**
Config18	Scalaire	-	-	-
	Différentielle	-	-	-
Ratio KO/OK	SIMD Horizontale	-	-	-
53.84	SIMD Verticale	-	-	-
Config19	Scalaire	1.00	1.00	1.00
	Différentielle	**2.33**	**2.31**	**2.33**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.17
-	SIMD Verticale	2.28	2.26	2.26
Config20	Scalaire	1.00	1.00	1.00
	Différentielle	**2.33**	**2.31**	**2.34**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.14
-	SIMD Verticale	2.29	2.30	2.29
Config21	Scalaire	1.00	1.00	1.00
	Différentielle	2.31	**2.32**	**2.33**
Ratio KO/OK	SIMD Horizontale	2.15	2.16	2.16
-	SIMD Verticale	**2.33**	2.31	2.32

TABLE 5 – Étape 3 : Performances des variantes de sommation sur Ivy Bridge / AVX
1 x Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50 GHz

D.4 Simulations sur Xeon Haswell - SIMD AVX2

Configuration	Algo	N=1	N= 4	2× 4 = 8
Config01	Scalaire	1.00	1.00	1.00
	Différentielle	**2.39**	**2.39**	**2.40**
Ratio KO/OK	SIMD Horizontale	2.16	2.16	2.18
-	SIMD Verticale	2.17	2.23	2.27
Config02	Scalaire	1.00	1.00	1.00
	Différentielle	**2.43**	**2.40**	**2.39**
Ratio KO/OK	SIMD Horizontale	2.18	2.15	2.16
-	SIMD Verticale	2.20	2.18	2.24
Config03	Scalaire	1.00	1.00	1.00
	Différentielle	2.23	2.23	**2.19**
Ratio KO/OK	SIMD Horizontale	2.19	2.19	2.20
-	SIMD Verticale	**2.28**	**2.26**	2.17
Config04	Scalaire	1.00	1.00	1.00
	Différentielle	**2.67**	**2.67**	**2.68**
Ratio KO/OK	SIMD Horizontale	2.06	2.06	2.06
-	SIMD Verticale	2.39	2.40	2.54
Config05	Scalaire	1.00	1.00	1.00
	Différentielle	**2.39**	**2.38**	**2.40**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.19
-	SIMD Verticale	2.22	2.22	2.29
Config06	Scalaire	1.00	1.00	1.00
	Différentielle	**2.39**	**2.38**	**2.39**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.18
-	SIMD Verticale	2.23	2.22	2.27
Config07	Scalaire	1.00	1.00	1.00
	Différentielle	**2.47**	**2.48**	**2.40**
Ratio KO/OK	SIMD Horizontale	2.18	2.18	2.17
-	SIMD Verticale	2.00	2.00	2.01
Config08	Scalaire	1.00	1.00	1.00
	Différentielle	**2.51**	**2.50**	**2.45**
Ratio KO/OK	SIMD Horizontale	2.18	2.19	2.20
-	SIMD Verticale	1.86	1.90	1.89
Config09	Scalaire	1.00	1.00	1.00
	Différentielle	**2.67**	**2.66**	**2.63**
Ratio KO/OK	SIMD Horizontale	2.12	2.13	2.12
-	SIMD Verticale	2.33	2.34	2.37
Config10	Scalaire	1.00	1.00	1.00
	Différentielle	**2.46**	**2.45**	**2.43**
Ratio KO/OK	SIMD Horizontale	2.14	2.14	2.15
-	SIMD Verticale	2.28	2.33	2.31
Config11	Scalaire	1.00	1.00	1.00
	Différentielle	2.39	2.39	2.43
Ratio KO/OK	SIMD Horizontale	2.12	2.13	2.14
1.83	**SIMD Verticale**	**2.48**	**2.54**	**2.58**

Configuration	Algo	N=1	N= 4	2× 4 = 8
Config12	Scalaire	1.00	1.00	1.00
	Différentielle	2.23	2.23	2.27
Ratio KO/OK	SIMD Horizontale	2.08	2.08	2.08
8.67	**SIMD Verticale**	**3.59**	**3.66**	**3.74**
Config13	Scalaire	1.00	1.00	1.00
	Différentielle	**2.51**	**2.51**	2.54
Ratio KO/OK	SIMD Horizontale	2.11	2.11	2.10
-	SIMD Verticale	2.49	**2.51**	**2.58**
Config14	Scalaire	1.00	1.00	1.00
	Différentielle	2.31	2.32	2.36
Ratio KO/OK	SIMD Horizontale	2.09	2.10	2.11
2.69	**SIMD Verticale**	**3.23**	**3.30**	**3.37**
Config15	Scalaire	1.00	1.00	1.00
	Différentielle	2.15	2.16	2.17
Ratio KO/OK	SIMD Horizontale	2.04	2.04	2.04
9.56	**SIMD Verticale**	**4.89**	**4.86**	**5.04**
Config16	Scalaire	1.00	1.00	1.00
	Différentielle	2.08	2.09	2.10
Ratio KO/OK	SIMD Horizontale	-	2.01	2.01
29.48	**SIMD Verticale**	-	**6.60**	**7.15**
Config17	Scalaire	-	-	-
	Différentielle	-	-	-
Ratio KO/OK	SIMD Horizontale	-	-	-
102.84	SIMD Verticale	-	-	-
Config18	Scalaire	-	-	-
	Différentielle	-	-	-
Ratio KO/OK	SIMD Horizontale	-	-	-
53.84	SIMD Verticale	-	-	-
Config19	Scalaire	1.00	1.00	1.00
	Différentielle	**2.31**	**2.30**	**2.31**
Ratio KO/OK	SIMD Horizontale	2.17	2.16	2.19
-	SIMD Verticale	2.09	2.10	2.14
Config20	Scalaire	1.00	1.00	1.00
	Différentielle	**2.31**	**2.30**	**2.30**
Ratio KO/OK	SIMD Horizontale	2.17	2.17	2.19
-	SIMD Verticale	2.12	2.14	2.17
Config21	Scalaire	1.00	1.00	1.00
	Différentielle	**2.30**	**2.31**	**2.32**
Ratio KO/OK	SIMD Horizontale	2.17	2.17	2.20
-	SIMD Verticale	2.10	2.15	2.19

TABLE 6 – Étape 3 : Performances des variantes de sommation sur Haswell / AVX2
1 x Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz

D.5 Simulations sur Xeon Phi - SIMD 512bits

Configuration	Algo	N=1
Config01	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.25
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.17
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.47
Config02	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.09
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.09
Config03	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.05
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.09
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.15
Config04	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.51
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.06
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.59
Config05	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.16
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.17
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.35
Config06	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.17
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.18
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.38
Config07	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.29
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.16
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.49
Config08	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.26
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.30
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.64
Config09	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.61
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.37
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	2.20
Config10	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.21
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.21
Config11	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.18
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.06
1.83	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.25
Config12	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.08
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.02
8.67	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.10
Config13	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.08
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.08
Config14	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.03
2.69	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.03
Config15	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.01
9.56	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.01
Config16	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.00
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.00
29.48	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.00
Config17	Etape 3 - Scalaire	-
	Etape 3 dont 3.1 - Différentielle	-
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	-
102.84	Etape 3 : 3.1 Différentielle et 3.2 SIMD	-
Config18	Etape 3 - Scalaire	-
	Etape 3 dont 3.1 - Différentielle	-
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	-
53.84	Etape 3 : 3.1 Différentielle et 3.2 SIMD	-
Config19	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.24
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.17
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.45
Config20	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.25
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.19
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.48
Config21	Etape 3 - Scalaire	1.00
	Etape 3 dont 3.1 - Différentielle	1.24
Ratio KO/OK	Etape 3 : 3.1 Scalaire et 3.2 SIMD	1.21
-	Etape 3 : 3.1 Différentielle et 3.2 SIMD	1.50

TABLE 7 – Étape 3 : Performances des variantes de la sommation différentielle
1 x Xeon Phi 3120A

Jason LAMBERT
Laboratoire d'Informatique (CEA-LIST/DISC)
Laboratoire de Recherche en Informatique (Université Paris-Sud)

Parallélisation de simulations interactives de champs ultrasonores pour le contrôle non destructif

Résumé

La simulation est de plus en plus utilisée dans le domaine industriel du Contrôle Non Destructif. Elle est employée tout au long du processus de contrôle, que ce soit pour en accélérer la mise au point ou en comprendre les résultats. Les travaux menés au cours de cette thèse présentent une méthode de calcul rapide de champ ultrasonore rayonné par un capteur multi-éléments dans une pièce isotrope, permettant un usage interactif des simulations. Afin de tirer parti des architectures parallèles communément disponibles, un modèle régulier (qui limite au maximum les branchements divergents) dérivé du modèle générique présent dans la plateforme logicielle CIVA a été mis au point. Une première implémentation de référence a permis de le valider par rapport aux résultats CIVA et d'analyser son comportement en termes de performances. Le code a ensuite été porté et optimisé sur trois classes d'architectures parallèles aujourd'hui disponibles dans les stations de calcul : le processeur généraliste central (GPP), le coprocesseur *manycore* (Intel MIC) et la carte graphique (nVidia GPU). Concernant le processeur généraliste et le coprocesseur *manycore*, l'algorithme a été réorganisé et le code implémenté afin de tirer parti des deux niveaux de parallélisme disponibles, le *multithreading* et les instructions vectorielles. Sur la carte graphique, les différentes étapes de simulation de champ ont été découpées en une série de noyaux CUDA. Enfin, des bibliothèques de calculs spécifiques à ces architectures, Intel MKL et nVidia cuFFT, ont été utilisées pour effectuer les opérations de Transformées de Fourier Rapides. Les performances et la bonne adéquation des codes produits ont été analysées en détail pour chaque architecture. Dans plusieurs cas, sur des configurations de contrôle réalistes, des performances autorisant l'interactivité ont été atteintes. Des perspectives pour traiter des configurations plus complexes sont dressées. Enfin la problématique de l'industrialisation de ce type de code dans la plateforme logicielle CIVA est étudiée.

Mots clés : Contrôle non destructif, Programmation parallèle, Simulation de champ ultrasonore, Processeurs généralistes multicoeurs SIMD, Processeurs graphiques, GPGPU, SIMD, Parallélisme (informatique), Xeon Phi, CUDA, Manycore

Parallelization of ultrasonic field simulations for non destructive testing

Abstract

The Non Destructive Testing field increasingly uses simulation. It is used at every step of the whole control process of an industrial part, from speeding up control development to helping experts understand results. *During this thesis, a fast ultrasonic field simulation tool dedicated to the computation of an ultrasonic field radiated by a phase array probe in an isotropic specimen has been developped.* During this thesis, a simulation tool dedicated to the fast computation of an ultrasonic field radiated by a phase array probe in an isotropic specimen has been developped. Its performance enables an interactive usage. To benefit from the commonly available parallel architectures, a regular model (aimed at removing divergent branching) derived from the generic CIVA model has been developped. First, a reference implementation was developped to validate this model against CIVA results, and to analyze its performance behaviour before optimization. The resulting code has been optimized for three kinds of parallel architectures commonly available in workstations: general purpose processors (GPP), manycore coprocessors (Intel MIC) and graphics processing units (nVidia GPU). On the GPP and the MIC, the algorithm was reorganized and implemented to benefit from both parallelism levels, multhreading and vector instructions. On the GPU, the multiple steps of field computing have been divided in multiple successive CUDA kernels. Moreover, libraries dedicated to each architecture were used to speedup Fast Fourier Transforms, Intel MKL on GPP and MIC and nVidia cuFFT on GPU. Performance and *hardware adequation* of the produced algorithms were thoroughly studied for each architecture. On multiple realistic control configurations, interactive performance was reached. Perspectives to adress more complex configurations were drawn. Finally, the integration and the industrialization of this code in the commercial NDT plateform CIVA is discussed.

Keywords : Non destructive testing, Parallel programming, Ultrasonic field simulation, Multicore general purpose processors, Graphic processing units, GPGPU, SIMD, Parallelism, Xeon Phi, CUDA, Manycore