École Privée des Sciences Informatiques

Mémoire

Étude et implémentation d'une méthode de calcul pour la simulation numérique sur des architectures modernes

---

Adrien Cassagne

Sous l'encadrement d'Arnaud Cueille

Montpellier, le 17 août 2014

Remerciements

Il est temps pour moi de finaliser ce mémoire et d'écrire mes derniers mots dans un format un peu moins conventionnel. Après plusieurs mois de travail autour d'un sujet passionnant je suis fier de pouvoir commencer ma petite liste.

Je tiens à remercier dans un premier temps le CINES pour m'avoir permis de consacrer une partie de mon temps à de la veille et à l'étude de cette méthode de calcul. Et plus particulièrement je voudrais remercier Tyra Van Olmen pour sa relecture d'une grande partie de ce mémoire ainsi que Gabriel Hautreux et Bertrand Cirou avec qui nous avons travaillé sur certaines optimisations présentées dans ce mémoire. Je n'oublie pas non plus Tristan Cabel pour ses conseils en anglais.

Un grand merci à Isabelle d'Ast pour m'avoir permis d'utiliser sa machine de travail sur laquelle j'ai eu accès à de grosses ressources de calcul. C'est grâce à cela que j'ai pu mener à bien une bonne partie des expérimentations de ce mémoire.

Je remercie chaleureusement mes parents, Thérèse et Jean-François Cassagne pour avoir exclusivement financé mes études et pour m'avoir soutenu tout au long de ces cinq années. Mention spéciale pour mon père : son aide dans la réalisation de ce mémoire a été très précieuse grâce à ses relectures et ses conseils avisés en tant qu'expert dans le domaine.

Je pense aussi à Chloé Martet pour avoir participé à la relecture approfondie de ce mémoire. Ses conseils ont permis d'éclaircir et de simplifier certaines explications afin que ce document soit des plus accessible, merci beaucoup.

Enfin je remercie toute l'équipe pédagogique de l'EPSI Bordeaux et les professeurs responsables du cycle d'ingénierie informatique, pour avoir assuré une partie de ma formation et pour avoir suivi la réalisation de ce mémoire.

Attestation de non-plagiat

Je soussigné Adrien Cassagne, auteur du mémoire professionnel "Étude et implémentation d'une méthode de calcul pour la simulation numérique sur des architectures modernes" pour l'obtention du Titre 1 RNCP - Expert informatique et Système d'information, déclare :

- que ce mémoire est un document original,
- avoir obtenu les autorisations nécessaires pour la reproduction d'images, d'extraits, de tableaux, figures ou graphiques,
- ne pas avoir contrefait, falsifié, copié tout ou partie de l'œuvre d'autrui afin de la faire passer pour mienne,

et atteste :

- que toutes les sources d'information utilisées pour ce travail de réflexion et de rédaction sont référencées de manière exhaustive et claire dans la bibliographie/webographie de mon mémoire professionnel,
- être informé et conscient que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, et que le plagiat est considéré comme une faute grave et sanctionné par la Loi.

Fait le 17 août 2014 à Montpellier.

Résumé

Ce mémoire propose d'étudier plusieurs architectures matérielles modernes au travers d'une méthode représentative des codes de simulation numérique existants : les algorithmes de type *stencil*. Le contexte HPC est propice à l'apparition de nouvelles technologies avec une course vers l'exascale plus complexe que prévue.

L'analyse comparative dans ce mémoire est essentiellement basée sur la performance en nombre d'opérations par seconde ainsi que sur l'efficacité énergétique des machines. Dans un premier temps, la méthode de calcul *stencil* est formalisée et explicitée puis trois architectures différentes sont analysées : un CPU traditionnel x86, un CPU ARM basse consommation et un GPU spécialisé dans le calcul.

S'en suit toute une partie dédiée à l'optimisation des *stencils* afin de proposer des versions efficaces sur chacune des architectures. Les biens connus techniques de *Cache Blocking* et de *Register Blocking* sont considérées ainsi que d'autres approches moins courantes comme la méthode *Dimension Lifted and Transposed* et le *Temporal Blocking*.

Enfin les différentes implémentations sont mises à l'épreuve par l'intermédiaire d'un *stencil* de faible ordre engendré par la discrétisation de l'équation de la chaleur. L'analyse est structurée en trois parties différentes suivant les trois architectures. Le modèle *Roofline* est utilisé pour borner les performances maximales atteignables. Sur CPU et GPU, le comportement du code est observé compte tenu de la variation de la taille du problème. La scalabilité faible dans les caches en fonction du nombre de cœurs est aussi étudiée mais seulement pour les CPUs. Pour terminer, une analyse comparative de la consommation énergétique est proposée.

Abstract

In this thesis we will study some modern hardware architectures using a well-known method in digital simulation : the stencil codes. The current HPC context is quite suitable for the arrival of new technologies leaded by the race to exascale computation.

The comparative analysis in this thesis is mainly based on performance (number of floating point operations per second) and on hardware energy efficiency.

We will start by formalising and clarifing the stencil method, then we will take a deeper look at three architectures : one standard x86 CPU, one low consumption ARM CPU and one GPU specialized in computations.

Thereafter, we will describe some stencil optimisations in order to implement efficient versions of code for each of the architectures. We will explore both well-known methods like Cache Blocking and Register Blocking as well as less known ones such as Dimension Lifted and Transposed and Temporal Blocking.

To finish, all these implementations will be tested on a low order stencil using the heat equation discretisation. The analysis will contain three different parts following the three architectures. We will use the Roofline model in order to bound the maximal reachable performance. Then we will study the code internal behavior on CPU and GPU by modifying the problem size. We will also take a look on the weak scalability in caches but only for the CPUs. Lastly, we will present a comparative analysis of energy consumption (also called energy to solution analysis).

Table des matières

Chapitre 1

Introduction

I Introduction au mémoire

Le monde du calcul haute performance (*High Performance Computing*, dans la littérature anglaise) est un secteur informatique et scientifique qui connaît une course effrénée pour l'amélioration de la performance des ordinateurs. Les plus puissants d'entre eux, les supercalculateurs, sont bâtis autour de l'interconnexion de centaines voire de milliers d'ordinateurs quasi traditionnels. Les supercalculateurs répondent bien souvent à des besoins scientifiques pour la simulation de modèles divers. Ainsi, le choix de la complexité du modèle dépend en partie de la capacité du supercalculateur utilisé et c'est pour cette raison que l'amélioration des performances est un véritable enjeu.

La confortable loi de G. Moore (1965) avait prédit que le nombre de transistors allait doubler tous les ans pour un prix constant. Il a donc été assez facile de faire le parallèle avec les performances : ces dernières allaient approximativement doubler tous les ans aussi. Étonnement cette loi s'est avérée juste jusqu'à présent et cela grâce à deux évolutions majeures :

- l'augmentation de la finesse de gravure des processeurs qui a permis de physiquement contenir un nombre toujours plus élevé de transistors dans un espace inchangé,
- l'architecture multi-cœurs qui a répondu à la problématique de chauffe liée à l'augmentation des fréquences (la température augmente suivant le carré de la fréquence).

Aujourd'hui, nous avançons vers une véritable rupture technologique puisque la finesse de gravure des processeurs va se heurter à des limites physiques au delà desquelles le signal électrique ne pourra plus être correctement transmis. De plus, la consommation des supercalculateurs a atteint des sommets et il n'est plus possible de continuer ainsi : l'énergie n'est malheureusement pas infinie. Pour adresser cette problématique, les fondeurs et constructeurs de composants informatiques concentrent maintenant leurs efforts sur l'efficacité énergétique plutôt que sur la performance brute. Cela donne lieu à l'apparition de nouvelles architectures matérielles et par conséquent à de nouveaux modèles de programmation.

Ce mémoire, au travers d'une méthode de calcul bien définie, traite de l'utilisation des architectures modernes dans une optique de performance et de rentabilité énergétique. Le premier chapitre introduit la simulation numérique et ses besoins puis se focalise sur un type d'algorithme très répandu : les *stencils*. Le second chapitre décrit les différentes

architectures matérielles qui ont été utilisées et le troisième expose les implémentations et optimisations mises en œuvre. Dans le quatrième chapitre nous commenterons les expérimentations mises en place et les résultats obtenus. Enfin la dernière partie dresse un bilan des travaux effectués et des évolutions envisagées.

II La simulation numérique et ses besoins

La simulation numérique est une discipline dans laquelle l'utilisation d'un programme informatique est requise afin de simuler un phénomène (physique, chimique, biologique, etc.). Cette approche est très largement utilisée dans les sciences pour essayer de prédire l'évolution d'un phénomène dans le temps et dans l'espace. La plupart des simulations numériques reposent sur des équations dites gouvernantes. Ces dernières sont une approximation d'un phénomène réel et elles sont définies de manière continue. Dans la simulation numérique, ces équations ne peuvent pas être utilisées directement et doivent être discrétisées : les machines de calcul actuelles peuvent uniquement traiter un nombre fini d'opérations. La discrétisation est faite de manière à approximer au mieux les équations d'origines. De ce fait, la simulation numérique ne permet pas de prédire l'évolution d'un système avec une certitude absolue mais elle essaye de s'en rapprocher au maximum. C'est dans ce cadre que sont utilisées les très célèbres méthodes des différences finies et des volumes finis (voir le cours de L. Risser [Ris06] pour plus de précisions).

La simulation numérique a d'abord été très utilisée dans le milieu de la recherche et notamment en physique (physique nucléaire, mécanique des fluides, physique des matériaux, mécanique quantique, sismologie, etc.) mais depuis, elle s'est répandue dans de nombreux domaines comme la chimie (dynamique moléculaire, etc.), la biologie (génomique, etc.), la finance, etc. Aujourd'hui, elle est aussi très prisée des industriels et notamment par :

- l'aéronautique avec la CFD (*Computational Fluid Dynamics*) qui permet, par exemple, de valider le bon écoulement de l'air sur une aile d'avion de manière à maximiser sa portance (cf. Fig 1.1),
- l'industrie pharmaceutique pour la mise au point de nouvelles molécules,
- l'industrie pétrolière avec la simulation sismologique (reconstruction numérique de la topologie des sous-sols afin de déterminer les positions les plus probables de poches d'hydrocarbure ou de roches mères),
- la météorologie pour prédire le temps à quelques jours ou pour prédire des modifications climatiques sur le long terme.

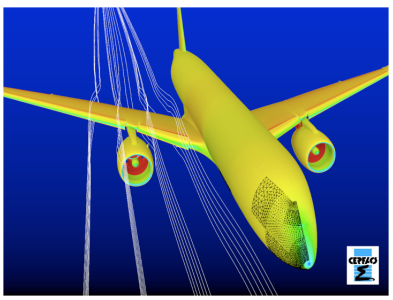

Fig. 1.1 – Simulation de l'écoulement de l'air sur une aile d'avion

III Les algorithmes de type *stencil*

Les algorithmes de type *stencil* sont au cœur de très nombreux codes de simulation numérique et notamment ceux basés sur la méthode des différences finies (voir ci-dessus Sec. II). Il est cependant assez courant de les retrouver dans des domaines très différents comme le traitement de l'image par exemple (filtre d'antialiasing, filtre de netteté, identification de motifs, etc).

III.1 Définition

Un *stencil* (pochoir en français) est un motif (*pattern* dans la littérature anglaise) que l'on applique à une cellule. Les algorithmes de type *stencil* ont la particularité de faire intervenir ce *stencil* autant de fois qu'il y a de cellules à traiter. Le *stencil* caractérise donc l'opération à effectuer alors que l'algorithme a recours à plusieurs utilisations du *stencil*. La Fig. 1.2 illustre un *stencil* particulier (*stencil* au voisinage de VON NEUMANN) sur une grille 2D de $6 \times 6 = 36$ cellules. Ici, pour calculer la valeur d'une cellule, il faut connaître les valeurs des cellules voisines de gauche, de droite, du haut et du bas.

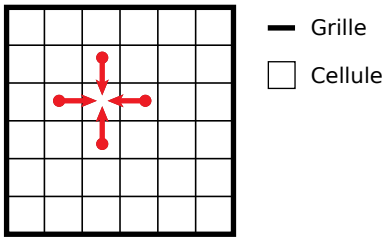

FIG. 1.2 – *Stencil* au voisinage de VON NEUMANN

Dans ce mémoire nous traiterons uniquement les *stencils* pour des maillages structurés que nous appellerons la grille. Cette dernière est la donnée d'entrée de notre problème et c'est elle que nous allons faire évoluer en appliquant le *stencil* sur chacune de ses cellules.

Plus formellement (notations basées sur l'article Wikipedia [1]), un algorithme de type *stencil* peut être caractérisé par un 5-uplet (I, S, S_0, s, T) :

- $I = \prod_{i=1}^{k} [0, ..., n_i]$ représente la topologie de la grille avec k la dimension et n_i le nombre de cellules par direction,
- S est l'ensemble des valeurs que peut prendre une cellule de la grille, en physique on a très souvent $S = \mathbb{R}$,
- $S_0 : I \to S$ définit l'état initial de la grille pour chaque cellule qui la compose,
- $s \in \prod_{i=1}^{l} I$ est le *stencil* à proprement parler, s est un l-uplet de positions (relatives à la cellule calculée) qui définit la forme de voisinage du *stencil*,
- $T : S^l \to S$ est une fonction de transition du l-uplet de valeurs dans S vers un 1-uplet dans S.

1. Stencil code : http://en.wikipedia.org/wiki/Stencil_code

A Itération de Jacobi

Il y a plusieurs méthodes qui permettent de construire un algorithme de type *stencil*. Dans ce mémoire nous allons nous focaliser sur une méthode spécifique : l'itération de Jacobi. Cet algorithme s'articule autour d'un nombre fini d'itérations qui représentent souvent l'avancement en temps de la simulation. À chaque itération, une nouvelle grille (ensemble de cellules) est calculée à partir de la précédente. Cela implique la conservation de la grille précédente jusqu'à la fin du calcul de la nouvelle grille. La Fig. 1.3 illustre le calcul de la nouvelle grille `G2` à partir de la grille initiale `G1` suivant un *stencil* au voisinage de Von Neumann d'ordre 1 (en 2D).

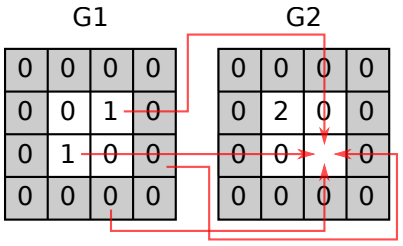

Fig. 1.3 – Itération de Jacobi pour un *stencil* de von Neumann

En reprenant le formalisme présenté en sous section III.1, l'algorithme *stencil* utilisé dans la Fig. 1.3 peut être décrit comme suit :

- $I = [0, 1, 2, 3]^2$,
- $S = \mathbb{N}$,
- $S_0 : I \to S,\ S_0(x, y) = \begin{cases} 0, \forall (x, y) \\ 1, (x, y) = (2, 1) \vee (1, 2), \end{cases}$
- $s = ((-1, 0), (1, 0), (0, -1), (0, 1))$,
- $T : S^4 \to S,\ T(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4$.

B Voisinage : ordre et nombre de points

Le voisinage est une caractéristique très souvent utilisée pour définir un *stencil*. Il représente l'ensemble des cellules voisines nécessaires au calcul d'une cellule dans la grille. Il existe un très grand nombre de voisinages différents utilisés dans la simulation numérique (voir la thèse de M-M. Christen pour plus d'exemples [Chr11]) mais certains reviennent très souvent :

- voisinage de Von Neumann (cf. Fig. 1.4),
- voisinage de Moore (cf. Fig. 1.5).

Un voisinage peut être qualifié par son ordre et son nombre de points. L'ordre correspond à la distance entre la cellule à calculer et sa voisine la plus éloignée et le nombre de points représente le nombre de cellules nécessaires pour calculer la cellule. Les voisinages de Von Neumann (4-point) et de Moore (8-point) présentés dans les Fig. 1.4 et Fig. 1.5 sont d'ordre 1 alors que les voisinages de Von Neumann (8-point) et de Moore (24-point) présentés dans les Fig. 1.6 et Fig. 1.7 sont d'ordre 2. Par abus de langage on parle souvent de l'ordre d'un *stencil* et/ou d'un *stencil* n-point.

FIG. 1.4 – Voisinage de VON NEUMANN en 2D et d'ordre 1

FIG. 1.5 – Voisinage de MOORE en 2D et d'ordre 1

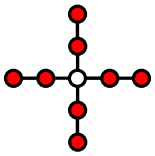

FIG. 1.6 – Voisinage de VON NEUMANN en 2D et d'ordre 2

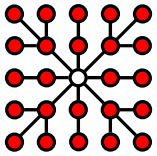

FIG. 1.7 – Voisinage de MOORE en 2D et d'ordre 2

Remarque : il arrive très souvent de considérer la cellule au centre dans le voisinage. Cette cellule occupe la même position spatiale que la cellule résultante. Dans ce cas, le nombre de points est incrémenté de 1. La Fig. 1.8 illustre un voisinage de VON NEUMANN en 2D d'ordre 1 considérant le point centré.

FIG. 1.8 – Voisinage de VON NEUMANN en 2D et d'ordre 1 (5-point)

III.2 Conditions aux limites du domaine

Toutes les cellules de la grille ne peuvent pas être traitées de la même manière. En effet, si l'on prend comme exemple la Fig. 1.3, les cellules grisées au bord ne possèdent pas toutes les voisines nécessaires pour appliquer le *stencil* défini par $s = ((-1, 0), (1, 0), (0, -1), (0, 1))$. Il faut donc faire une distinction entre les cellules au bord (aussi appelées cellules aux frontières, cellules aux limites, cellules externes) et les cellules internes : on parle alors des conditions aux limites du domaine. Il existe plusieurs types de conditions aux limites, les plus connus sont :

- les conditions aux limites de DIRICHLET : les valeurs au bord restent constantes (cf. Fig. 1.3 et Fig. 1.9),
- les conditions aux limites de NEUMANN : les dérivées des valeurs au bord restent constantes (cf. Fig. 1.10),
- les conditions périodiques aux limites : le *stencil* s'applique aussi sur les bords et il utilise les frontières opposées quand une cellule voisine est manquante (cf. Fig. 1.11).

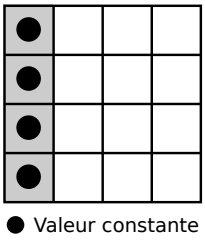

FIG. 1.9 – Condition aux limites de DIRICHLET (frontière gauche)

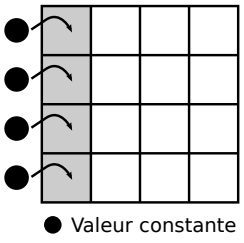

FIG. 1.10 – Condition aux limites de NEUMANN (frontière gauche)

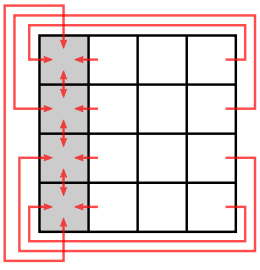

FIG. 1.11 – Condition périodique aux limites (frontière gauche)

Attention un domaine peut être composé de différentes conditions aux limites (périodiques à droite et à gauche puis de DIRICHET en haut et en bas par exemple). Dans

ce mémoire nous utiliserons les conditions aux limites de DIRICHLET et les conditions périodiques aux limites. Les conditions aux limites de NEUMANN ne présentent pas un grand intérêt pour l'étude de la méthode de calcul.

III.3 Complexité et intensité arithmétique

Définition de la complexité d'un algorithme : l'analyse de la complexité des algorithmes permet de quantifier formellement le temps que nécessite l'exécution d'un algorithme en fonction de la taille des données d'entrée. Comme il n'est pas toujours facile de connaître avec exactitude le temps nécessaire à la terminaison d'un algorithme, l'évaluation du temps est souvent proposée pour le pire des cas ou pour le cas moyen. Pour une quantité n de données, la complexité dans le pire des cas est notée : $O(f(n))$ avec f une fonction quelconque.

Définition de l'intensité arithmétique : l'intensité arithmétique d'un algorithme représente le nombre d'opérations (sur des nombres flottants) divisé par le nombre de données utilisées. Plus l'intensité arithmétique est élevée plus l'algorithme est limité par les unités de calcul. Plus l'intensité arithmétique est faible plus l'algorithme est limité par la bande passante mémoire. Pour de plus amples informations, la thèse de M-M. CHRISTEN [Chr11] propose un grand nombre d'exemples et d'applications sur l'intensité arithmétique.

La complexité d'un algorithme de type *stencil* noté A_s dépend du nombre de cellules n présentes dans la grille et de la fonction de transition T (cf. sous section III.1). Soit C la complexité de l'algorithme :

$$C(A_s) = n \times Flops(T) = O(n),$$

avec $Flops(T)$ le nombre d'opérations flottantes effectuées par la fonction de transition T. La complexité d'un algorithme de type *stencil* est donc linéaire par rapport au nombre de cellules.

L'intensité arithmétique I d'un algorithme de type *stencil* dépend directement de la fonction de transition T :

$$I(A_s) = \frac{Flops(T) \times n}{2 \times n} \Leftrightarrow I_s = \frac{Flops(T)}{2},$$

avec $2n$ le nombre total des données nécessaires au calcul d'une nouvelle grille (n cellules dans `G1` et n cellules dans `G2`). On remarque que l'intensité arithmétique est constante par rapport à de la taille du problème (car $Flops(T)$ est un nombre constant). Il est donc nécessaire de bien évaluer la fonction de transition pour le calcul de l'intensité arithmétique. De manière générale, la fonction de transition d'un *stencil* est définie comme suit :

$$T(x_1, x_2, ..., x_l) = \sum_{i=1}^{l} (\alpha_i \times x_i),$$

avec l le nombre de points du *stencil* et $(\alpha_1, \alpha_2, ..., \alpha_l)$ des coefficients constants pour l'itération. Chaque membre $\alpha_i \times x_i$ est appelé une contribution au *stencil*. Le nombre d'opérations flottantes dans T dépend uniquement du nombre de points :

$$Flops(T) = (2 \times l) - 1,$$

en effet, il y a l multiplications et $l-1$ additions.

Le Tab. 1.1 expose l'intensité arithmétique de *stencils* 2D aux voisinages de Von Neumann et de Moore en fonction de l'ordre, du nombre de points et d'une fonction de transition de S^l dans S. Pour chaque voisinage on considère en premier le voisinage sans le point centré puis en second en incluant le point centré. On remarque que plus le nombre de points est élevé plus l'intensité arithmétique est élevée. Sans rentrer dans les détails, sur les architectures modernes seuls les algorithmes avec une intensité arithmétique élevée peuvent espérer se rapprocher des performances crêtes (performances maximales théoriques d'une architecture).

Voisinage	**Ordre**	**Nb. de points**	**Flops**	**Nb. de données**	**Int. arith.**
Von Neumann	1	4	7	2	3,5
Von Neumann	1	5	9	2	4,5
Von Neumann	2	8	15	2	7,5
Von Neumann	2	9	17	2	8,5
Von Neumann	3	12	23	2	11,5
Von Neumann	3	13	25	2	12,5
Moore	1	8	15	2	7,5
Moore	1	9	17	2	8,5
Moore	2	24	47	2	23,5
Moore	2	25	49	2	24,5
Moore	3	48	95	2	47,5
Moore	3	49	97	2	48,5

Tab. 1.1 – Intensité arithmétique de *stencils* 2D en fonction du type de voisinage, de l'ordre et du nombre de points

Chapitre 2

Architectures matérielles

I Les processeurs traditionnels (architecture x86)

Le premier processeur d'architecture x86 a été mis au point par *Intel* en 1978 (l'*Intel* 8086). Tous les processeurs avec l'architecture x86 suivent le modèle CISC (*Complex Instruction Set Computer*) par opposition au modèle RISC (*Reduced Instruction Set Computer*). En d'autres termes, ils possèdent un très grand nombre d'instructions qui peuvent parfois nécessiter plusieurs cycles d'horloge pour s'exécuter. Le principal avantage du modèle CISC est de proposer une rétro compatibilité avec les processeurs x86 précédents. Aucune instruction n'est supprimée, il n'y a que des ajouts. Cela amène cependant une certaine complexité dans les instructions et ne facilite pas la compréhension et la maîtrise des codes.

En 1989, avec l'*Intel 80486*, l'architecture x86 est augmentée d'un *pipeline*. Cette chaîne de traitement consiste en un découpage des instructions classiques en plusieurs instructions élémentaires. Cela permet de ne pas avoir à attendre qu'une instruction classique soit complètement terminée pour en exécuter une autre. Avec un *pipeline* le processeur est capable de commencer à exécuter une partie élémentaire d'une nouvelle instruction sans que la précédente instruction ne soit complètement terminée.

Depuis l'arrivée de l'*Intel Pentium Pro* en 1995, tous les processeurs x86 sont *out-of-order* (sauf les *Atom*) : les instructions ne sont pas nécessairement exécutées dans l'ordre donné par l'exécutable. Cela permet de rentabiliser l'utilisation des unités de traitement du processeur en privilégiant les instructions qui peuvent véritablement être effectuées au lieu de bloquer toute la chaîne d'instructions.

Un autre point très important de l'architecture x86 est la hiérarchie mémoire nécessaire au bon fonctionnement des processeurs. En effet, le principal problème vient de la bande passante de la mémoire vive (RAM, *Random Access Memory*) qui ne permet pas une utilisation maximale du processeur. Pour palier à cette limite matérielle, les processeurs modernes sont équipés de différentes mémoires caches plus rapides que la mémoire RAM. Le principe est assez simple, quand une donnée est accédée pour la première fois dans la mémoire RAM, alors cette donnée est recopiée dans les caches du processeur (plus le cache est physiquement proche du processeur plus il est performant et plus il est petit, on parle alors des différents niveaux de cache L1, L2 et L3). De cette façon, les prochains accès à cette même donnée seront bien plus rapides pour le processeur : l'architecture x86 est optimisée de façon à minimiser la latence des accès aux données.

Enfin, c'est en 2006 que sont apparus les premiers processeurs à deux cœurs avec les *Pentium Dual Core* T20xx. Depuis, le nombre de cœurs n'a pas cessé d'augmenter.

Il existe bien sûr une multitude d'autres mécanismes intégrés au matériel mais ceux décrits ci-dessus définissent les grands principes de l'architecture x86. De plus *Intel* n'est pas le seul fondeur de processeurs x86, il y a notamment AMD, son principal concurrent, qui a joué un rôle essentiel dans l'évolution de cette architecture (instructions 64 bits, multi-cœur, etc.).

I.1 Dernières architectures d'*Intel*

Cette sous section détaille plus finement les dernières évolutions matérielles d'*Intel* suivant les trois grandes dernières architectures du fondeur : *Nehalem*, *Sandy Bridge* et *Haswell*.

A *Nehalem*

L'architecture *Nehalem* a été introduite par *Intel* en 2008. C'est la première architecture multi-cœur à intégrer le contrôleur mémoire dans le processeur (anciennement dans le chipset *North Bridge*). Elle ajoute aussi un nouveau bus de données inter-processeur point à point : le QPI (*QuickPath Interconnect*). Ce bus permet d'améliorer le fonctionnement des nœuds NUMA (*Non Uniform Memory Access*, utilisation de plusieurs processeurs sur une même carte mère) avec l'ajout d'un lien rapide entre les processeurs (bien plus rapide que l'ancien FSB, *Front Side Bus*). L'architecture *Nehalem* marque aussi le retour de l'*Hyper-threading* permettant à deux *threads* de partager les ressources d'un cœur. Dans certains cas, cette technologie permet d'améliorer l'occupation des ressources d'un cœur. De plus, chaque cœur possède le jeu d'instructions vectorielles SSE4.2 (*Streaming SIMD Extensions*) 128 bits : chaque unité vectorielle est capable de calculer 4 nombres flottants simple précision ou 2 nombres flottants double précision en un cycle d'horloge. Comme il y a deux unités vectorielles par cœur (une pour les additions et une pour le multiplications), un cœur est capable, à chaque cycle, de calculer huit flottants simple précision ou quatre flottants double précision. Enfin, chaque cœur possède un cache L1 pour les données et les instructions et un cache L2 pour les données, alors que tous les cœurs partagent le cache L3.

La gamme des processeurs Xeon *Nehalem* d'*Intel* est majoritairement constituée de *quad cores*.

B *Sandy Bridge*

L'architecture *Sandy Bridge* (2011) ressemble beaucoup à l'architecture *Nehalem*. Le lien QPI a été doublé afin d'augmenter sa bande passante mais la véritable amélioration vient du nouveau jeu d'instructions vectorielles AVX (*Advanced Vector Extensions*) 256 bits. Ces nouvelles instructions permettent de traiter 8 nombres flottants simple précision ou 4 nombres flottants double précision en un cycle d'horloge. Cela permet de doubler la performance crête par rapport à *Nehalem* (pour une fréquence égale). Cette évolution confirme aussi un nouveau virage pour le matériel avec le développement de deux niveaux de parallélisme : le nombre de cœurs et la taille des instructions vectorielles. En réalité, les unités vectorielles sont beaucoup plus rentables d'un point de vue énergétique que

l'ajout de cœurs. Elles sont relativement simples : elles effectuent la même opération sur tous les éléments d'un vecteur et cela de manière parfaitement synchrone.

La gamme des processeurs Xeon *Sandy Bridge* d'*Intel* est majoritairement constituée d'*octo cores*.

C *Haswell*

L'architecture *Haswell* (2013) améliore une nouvelle fois les unités vectorielles en supportant le jeu d'instructions AVX2 256 bits et les FMA (*Fused Multiply and Add*). AVX2 apporte de nouvelles instructions vectorielles sur les entiers alors que les FMA permettent de calculer une multiplication et une addition en un cycle d'horloge (sur un vecteur de 256 bits). Cela porte la performance crête à deux fois celle de *Sandy Bridge* (à fréquence égale). Cependant il n'est pas toujours possible de se retrouver dans une situation où le code calcule simultanément des additions et des multiplications et il est donc parfois impossible d'atteindre la performance crête.

La gamme des processeurs Xeon *Haswell* d'*Intel* est majoritairement constituée de processeurs entre 12 et 16 cœurs.

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	**64 Bytes/cycle**
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	**32 Bytes/cycle**
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	**64 Bytes/cycle**
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	**4K+2M shared: 1024, 8-way**

All caches use 64-byte lines

15 Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

FIG. 2.1 – Bande passante des caches L1 et L2 des dernières architectures *Intel*

La Fig. 2.1 montre l'évolution des bandes passantes des caches L1 et L2 pour les dernières architectures d'*Intel*. On remarque que de *Nehalem* à *Sandy Bridge*, seule la bande passante en lecture du cache L1 a été doublée. Cette évolution est normale puisque la taille des registres vectoriels est passé de 128 à 256 bits : une bande passante de 32 octets permet de charger un registre vectoriel par cycle (c'est le strict minimum). Par contre la différence entre *Haswell* et *Sandy Bridge* est bien plus marquée, presque toutes les bandes passantes ont été doublées. Une nouvelle fois cela s'explique par la capacité du

processeur en terme de calcul vectoriel : les opérations FMA doublant la crête théorique du processeur, il faut être capable de charger plus de registres vectoriels en un cycle pour pouvoir atteindre de bonnes performances. Une opération FMA peut s'exprimer de la sorte : $C = A \times B + C$ avec A, B, C des vecteurs. On remarque bien que le calcul nécessite la lecture de 3 registres alors que les précédentes instructions vectorielles (AVX1) n'avaient besoin que de deux vecteurs en lecture. Avec *Haswell*, il est maintenant possible de charger 2 registres de 256 bits en un cycle d'horloge.

II Les processeurs ARMs

Les ARMs (*Advanced Risc Machine*) sont d'architecture RISC (*Reduced Instruction Set Computer*) contrairement aux CPUs x86 et ils sont développés par la société ARM Ltd puis vendus sous forme de licence à des fabricants tiers. De ce fait, la société ARM Ltd ne propose pas directement de processeurs mais seulement des plans. La majorité des ARMs sont encore 32 bit même si les nouvelles architectures tendent à adopter le modèle 64 bit. Les jeux d'instructions des ARMs sont relativement simplistes comparé à ceux des processeurs x86. Très souvent, les cœurs ARMs sont *in-order* et possèdent peu de niveaux de pipeline (entre 5 et 15). Ils adoptent cependant la même stratégie de hiérarchie mémoire que x86 avec l'utilisation de différents niveaux de cache : ils sont donc aussi optimisés de façon à minimiser la latence des accès aux données. Les ARMs sont réputés pour être économes en énergie et on les retrouve très souvent dans l'embarqué et dans les appareils mobiles : la plupart de nos appareils électroniques tels que les téléphones portables et les tablettes sont équipés d'ARMs. De nos jours, leur capacité à consommer peu ainsi que leur modularité intéresse de plus en plus la communauté du HPC et certains projets étudient même la construction de véritables supercalculateurs à base d'ARMs (cf. projet Mont-Blanc [1]).

II.1 *Qualcomm Snapdragon* S4 Pro

Ce mémoire propose d'étudier un ARM *quad core* modifié et produit par *Qualcomm* : le *Snapdragon* S4 Pro (*System on Chip*, SoC haut de gamme des téléphones portables et tablettes en 2013). Ce dernier est basé sur la modification d'un ARM Cortex-A9 : les cœurs sont d'architecture *Krait 200*. Comme le Cortex-A9, le S4 Pro est *out-of-order* (alors que beaucoup d'ARMs sont encore *in-order*) et repose sur le jeu d'instructions ARM v7 32 bit. L'architecture *Krait 200* ajoute cependant quelques améliorations :

- la modification dynamique et désynchronisée de la fréquence des cœurs pour l'économie d'énergie (à la différence de la technologie big.LITTLE [2] proposée par ARM),
- l'augmentation de la profondeur du pipeline (11 niveaux dans les cœurs *Krait 200* contre 8 niveaux dans les cœurs Cortex-A9),
- la gravure en 28 nm (contre 45 nm pour le Cortex-A9).

Chaque cœur *Krait 200* possède un cache L1 de 32 Ko (16 Ko pour les instructions et 16 Ko pour les données) et les cœurs partagent deux par deux un cache L2 de 1 Mo. Le S4 Pro

1. Projet Mont-Blanc : http://www.montblanc-project.eu/
2. Technologie big.LITTLE : http://fr.wikipedia.org/wiki/Big.LITTLE

permet aussi de calculer sur des vecteurs 128 bit avec les instructions NEON [3] : à chaque cycle un cœur est capable de traiter 4 nombres flottants simple précision (attention ces instructions ne sont pas compatibles avec la norme IEEE 754).

III L'alternative : les accélérateurs de calcul

III.1 Les GPUs (*Graphics Processing Unit*)

A Fonctionnement et architecture générale des GPUs

Sur GPU, les traitements sont déportés vers ce que l'on appelle des "noyaux de calcul" (*kernels*). Ces derniers sont exécutés simultanément par plusieurs *threads*, le code est nativement SIMT (*Single Instruction Multiple Threads*). Ce modèle est différent de celui des CPUs : un CPU maximise l'utilisation des unités de calcul en diminuant la latence des accès à la mémoire (hiérarchie de caches) alors qu'un GPU possède une relativement grande latence mais la masque en exécutant des *threads* pendant qu'il effectue les accès à la mémoire. Sur le GPU comme sur le CPU il y a plusieurs types de mémoires :

- **la mémoire globale** : cette mémoire est lente mais de grande capacité (> 2 Go),
- **la mémoire partagée** : cette mémoire est partagée par un groupe de *threads*, elle est plus rapide que la mémoire globale mais sa capacité est inférieure à 64 Ko,
- **les registres** : ils sont propres à un *thread* et sont très rapides d'accès, leur nombre est limité à un maximum de 255 registres par *thread* (attention cette limite peut paraître énorme mais en réalité, une utilisation aussi intensive des registres ne permet pas la création d'un grand nombre de *threads*).

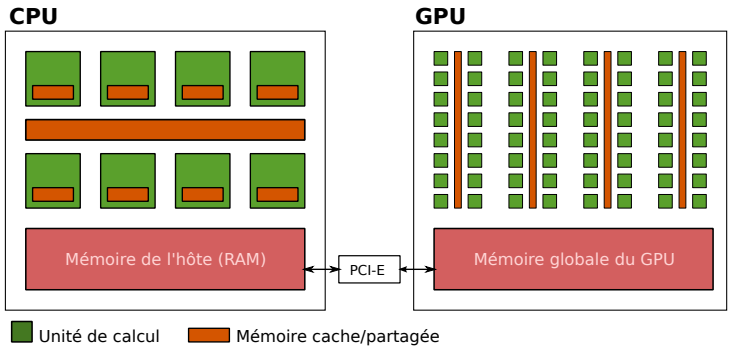

FIG. 2.2 – Architectures simplifiées d'un CPU et d'un GPU

La Fig. 2.2 présente les architectures simplifiées d'un CPU et d'un GPU. Sur GPU, il y a beaucoup plus d'unités de calcul que sur CPU. Cela a un impact sur la parallélisation du code : la granularité est différente. De plus, les transferts de données de la RAM vers la mémoire du GPU passent par le bus PCI-Express et ont un coût important, ce qui amène à optimiser ces transferts. Enfin, et contrairement aux CPUs, les GPUs sont *in-order* : les instructions sont exécutées dans l'ordre défini par l'exécutable.

3. Instructions NEON : http://fr.wikipedia.org/wiki/ARM_NEON

Logique d'exécution Les *threads* d'un noyau de calcul GPU sont organisés en grille de plusieurs blocs.

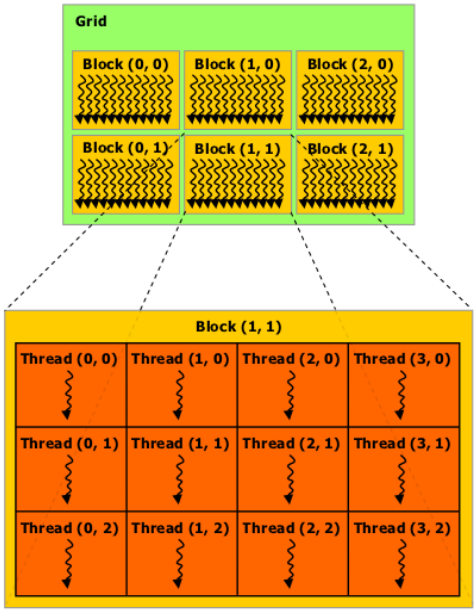

FIG. 2.3 – Exemple de grille 2D (2x3) avec blocs 2D (3x4)

La Fig. 2.3 illustre cette organisation : quand on exécute un *kernel*, il faut spécifier le nombre de blocs ainsi que la taille des blocs (nombre de *threads* par bloc). La grille représente la totalité des *threads* d'un noyau de calcul. Le bloc est un groupe de *threads* où les instructions vont être exécutées en même temps (parallélisme SIMD, *Single Instruction Multiple Data*). Sur les GPUs actuels, il est important de choisir des tailles de blocs qui sont multiples de 32 : cette spécificité vient du matériel qui ordonnance les blocs par groupe de 32 (appelé *warp*).

B La technologie CUDA

Dans le monde des accélérateurs GPU, il y a principalement deux constructeurs : *AMD* (anciennement *ATI*) et *Nvidia*. Afin d'exploiter le potentiel des GPUs en matière de calcul, les constructeurs ont mis différents *frameworks* à disposition des développeurs : OpenCL (*Open Computing Language*) et CUDA (*Compute Unified Device Architecture*) sont les plus répandus. OpenCL est un standard libre et utilisé par différents types de matériels (CPUs, GPUs, Xeon Phi, etc.). Les GPUs *AMD* et *Nvidia* sont compatibles avec cette norme. Cependant *Nvidia* développe son propre *framework* propriétaire depuis 2007 : CUDA. Ce dernier est uniquement disponible sur les GPUs *Nvidia* : il est très réputé et précurseur dans le monde du HPC. CUDA est très souvent en avance sur son concurrent OpenCL, il adopte très rapidement les nouvelles technologies proposées par les derniers GPUs. Le code étudié dans les chapitres suivant a été développé en CUDA `C` avec l'aide du guide de programmation de *Nvidia* [NVI14].

C Historique des architectures GPUs *Nvidia*

Avant de rentrer dans les détails de l'architecture *Kepler*, la table 2.1 rappelle l'historique des architectures GPU *Nvidia* dédiées au calcul. Le terme *compute capability* est employé par *Nvidia* pour désigner les possibilités matérielles de calcul des GPUs. Sans rentrer dans les détails de chaque architecture, voici les évolutions principales :

- de *Tesla* à *Fermi* : augmentation du nombre d'unités de calcul, possibilité d'exécuter plusieurs *kernels* en concurrence, apparition des unités de calcul double précision, ajout des caches L1 et L2,
- de *Fermi* à *Kepler* : augmentation du nombre d'unités de calcul, partage d'un GPU entre plusieurs processus, création de *kernels* au sein d'un *kernel* (calcul adaptatif).

Année de sortie	Architecture	Compute capability	Chipset GPU
2006	*Tesla*	1.0, 1.1, 1.2	G80, G86, GT218, ...
2008	*Tesla*	1.3	GT200, GT200b
2010	*Fermi*	2.0, 2.1	GF100, GF110, GF104, ...
2012	*Kepler*	3.0	GK104, GK106, GK107, ...
2013	*Kepler*	3.5	GK110

Tab. 2.1 – Historique des architectures GPU *Nvidia*

D L'architecture *Kepler*

Fig. 2.4 – Répartition des unités de calcul dans les multiprocesseurs de flux (SMX) d'un GPU de type GK110

Kepler est la dernière architecture de *Nvidia* (les GPUs *Maxwell* ne sont pas encore disponibles pour le calcul). La Fig. 2.4 illustre la répartition des unités de calcul dans les multiprocesseurs de flux (SMX, *Streaming Multiprocessor neXt generation*). Les multiprocesseurs de flux peuvent être apparentés à un coeur CPU alors que les unités de calcul (carrés verts et jaunes) sont assez proches des unités vectorielles d'un CPU. La couleur verte représente les unités de calcul simples précisions et la couleur jaune représente les unités de calcul double précision. Les SMX partagent un cache L2 et, de manière générale, chaque bloc d'une grille de calcul est envoyé à un SMX. S'il y a plus de blocs que de SMX, alors les SMX prennent plusieurs blocs en charge.

FIG. 2.5 – Détails d'un SMX

La Fig. 2.5 détaille l'architecture matérielle d'un multiprocesseur de flux. En plus des unités de calcul traditionnelles, il y a des unités SFUs (*Special Function Units*) dédiées aux fonctions spéciales (sinus, cosinus, racine carré, exponentielle, etc.). Pour résumer, dans un multiprocesseur de flux il y a 192 unités de calcul simple précision, 64 unités de calcul double précision et 32 unités de calcul spéciales. Le placement des *threads* sur les unités de calcul est assuré par 4 *Warp Scheduler* ce qui signifie qu'il peut potentiellement y avoir 4 noyaux de calculs différents qui s'exécutent en simultané sur un SMX. Enfin, pour ce qui est de la mémoire, il y a :

- 256 Ko dédié aux registres (*Register File*), soit 65 536 variables de 32bit,
- 64 Ko de mémoire partagée (*Shared Memory*, dont une partie dédiée au cache L1),
- 48 Ko de mémoire cache en lecture seule (*Read-Only Data Cache*).

Chapitre 3

Implémentation

I Le code `stenCINES`

I.1 Présentation

Le code `stenCINES` est l'implémentation de la méthode de calcul (*stencil*) présentée dans le chapitre 1 section III. Il est écrit en `C++` en essayant de se rapprocher au maximum de la norme 2011 (`C++11`) et en s'appuyant sur le livre de B. Stroustrup [Str13], créateur du langage. Le code a été pensé de manière à rester le plus générique possible et ainsi pouvoir être utilisé pour des *stencils* quelconques. Des compromis sur le paradigme objet ont été faits au profit de la performance. L'objectif principal du code étant d'atteindre les meilleures performances possibles, il a parfois fallu contourner certains aspects problématiques de la programmation orienté objet (cf. Sec. I.2 Sous-sec. A). Les fonctionnalités proposées par `stenCINES` sont les suivantes :

- itération de Jacobi avec deux *buffers*/grilles (cf. Chap. 1 Sec. III.1 et Sous-sec. A),
- choix entre deux types de conditions limites : de Dirichlet ou périodique (cf. Chap. 1 Sec. III.2),
- détection de la convergence pour une précision donnée,
- support de l'implémentation d'un *stencil* quelconque,
- gestion des entrées/sorties et de la grille.

Afin de pouvoir effectuer des comparaisons sur une même architecture ou sur des architectures différentes, `stenCINES` comporte 3 implémentations différentes : une version séquentielle CPU, une version *multi-threads* CPU (utilisation de la bibliothèque OpenMP) et une version GPU (développée avec CUDA). Chacune de ces implémentations sera détaillée dans les sections suivantes ainsi que les optimisations apportées.

I.2 Architecture du code

Le diagramme de classe Fig. 3.1 représente l'organisation du code `stenCINES`. `Simulation` est une des classes les plus importantes du modèle. C'est elle qui gère la boucle des itérations, le choix des conditions limites et les tests de convergence. Elle n'implémente cependant pas le parcours des cellules, ce dernier est laissé aux classes filles `SimulationSTD`, `SimulationCB`, `SimulationRB` et `SimulationDLT`. Chacune de ces classes représente une

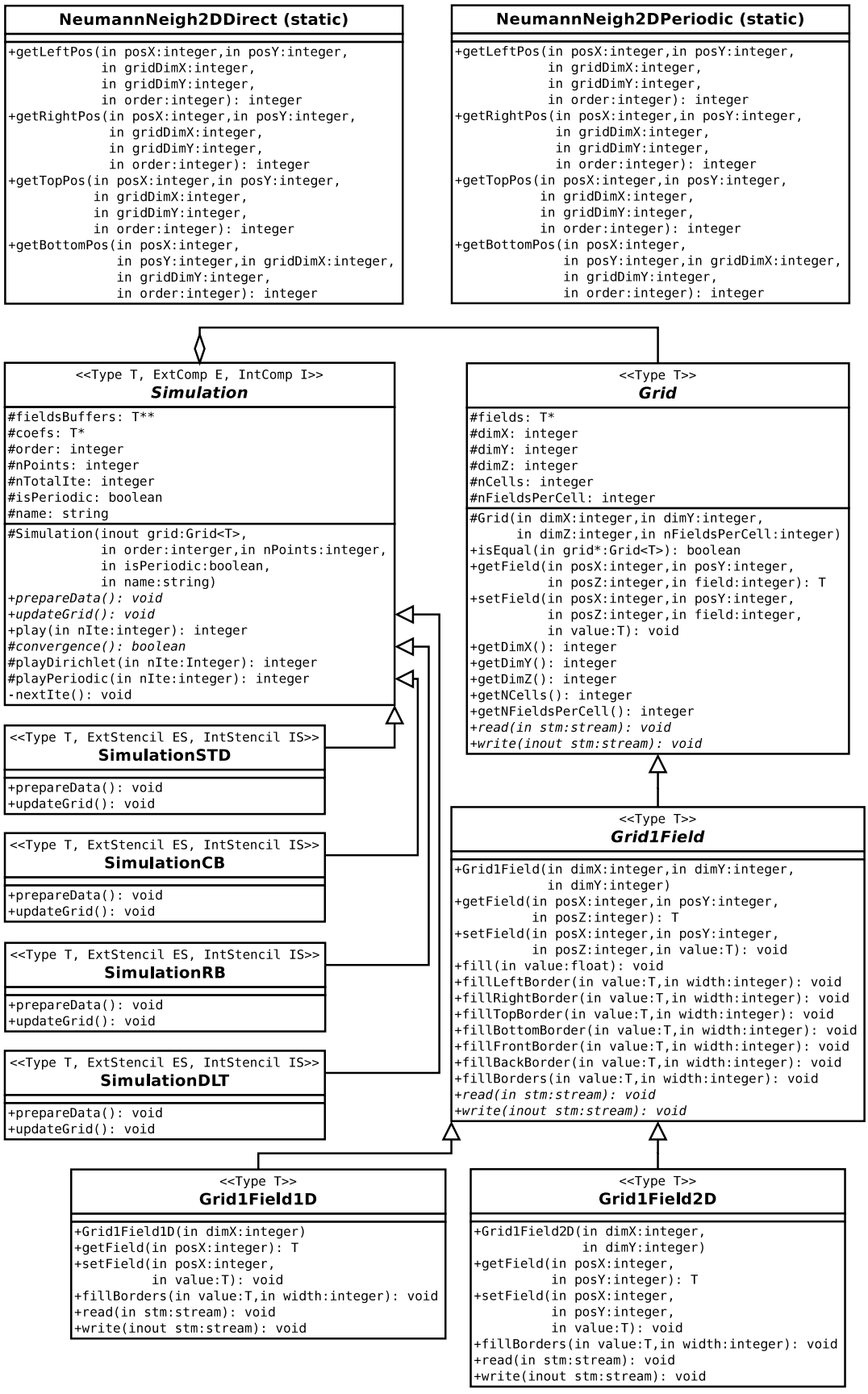

FIG. 3.1 – Diagramme de classe simplifié de `stenCINES` (UML)

implémentation différente du parcours des cellules dans la grille suivant des optimisations différentes. Ce sont aussi ces classes qui sont en charge de modifier (ou non) l'organisation des données de la grille lors de leurs recopies dans des *buffers* spécifiques au traitement (méthode `prepareData()`). En tant qu'utilisateur, l'implémentation d'un *stencil* passe par la construction d'une classe qui hérite directement de `SimulationSTD`, `SimulationCB`, `SimulationRB` ou `SimulationDLT`.

La classe `Grid` propose une représentation naturelle et simple des données de la grille (cf. Sous-sec. B) ainsi que des traitements courants effectués lors de la construction ou la sauvegarde de la grille (initialisation des cellules, sauvegarde des cellules dans un fichier, etc.). La classe *Grid* est assez générique et c'est pour cette raison qu'il existe plusieurs classes héritées :

- `Grid1Field` : une grille contenant un champ par cellule (en physique, un champ correspond souvent à une grandeur comme la vitesse, la température, etc.),
- `Grid1Field1D` : une grille contenant un champ par cellule en une dimension,
- `Grid1Field2D` : une grille contenant un champ par cellulle en deux dimensions.

En résumé, la classe `Grid` correspond à l'entrée/sortie de la classe `Simulation`.

Enfin, les classes `NeumannNeigh2DDirect` et `NeumannNeigh2DPeriodic` sont des classes statiques (pas d'attributs, pas d'instanciation) mise à disposition de l'utilisateur pour calculer directement les voisins dans le cas d'un *stencil* de Von Neumann.

A Généricité avec les *templates*

La principale difficulté liée à la généricité réside dans le fait que le code qui caractérise l'algorithme, le *stencil*, est une routine appelée pour chaque élément de la grille. Ce code est au plus bas niveau : à l'intérieur de la boucle itérative et de la boucle de parcours des cellules. Il apparaît donc clairement que l'appel au *stencil* est un enjeu crucial pour les performances. L'idée la plus élégante du point de vue génie logiciel serait de définir une méthode abstraite dans la classe `Simulation` et de redéfinir cette méthode dans les classes filles. Cependant, en `C++`, cela voudrait dire déclarer une méthode virtuelle pure et donc l'utilisation d'une table virtuelle [1] pour résoudre les appels à la méthode (lors de l'exécution du code). Cette solution a été testée et est apparue inefficace à cause du surcoût trop important lié à l'utilisation de la table virtuelle. Il a donc fallu trouver une alternative viable, en d'autres termes : comment passer un comportement (le *stencil*) à une classe sans que cela n'impacte les performances à l'exécution ? La solution qui a été retenue est l'utilisation des *templates*, ces derniers permettent de spécialiser le code d'une classe au moment de la compilation. Le plus souvent, les *templates* sont utilisés pour gérer des types génériques. Par exemple, dans `stenCINES`, le type des éléments `T` est générique. Cela permet, lors de l'instanciation de la classe, de choisir sur quel type de données le *stencil* effectue des opérations (le plus souvent sur des `double` ou sur des `float`). Mais il est aussi possible de passer du traitement par *template* et c'est ce qui est mis en place dans `stenCINES` pour le passage de la routine en charge de calculer le *stencil*. Comme expliqué précédemment, l'avantage de cette méthodologie est que l'appel au code *stencil* est résolu à la compilation (voire même traité en mode *in line*) à la place d'une résolution dynamique à l'exécution du programme avec une méthode

1. table virtuelle : http://en.wikipedia.org/wiki/Virtual_method_table

virtuelle. Sur le diagramme de classe Fig. 3.1, le *stencil* est passé par *template* dans les classes `SimulationSTD`, `SimulationCB`, `SimulationRB` et `SimulationDLT` (paramètres `ExtStencil ES` pour le *stencil* à l'extérieur du domaine et `IntStencil IS` pour le *stencil* à l'intérieur du domaine). Malheureusement cette méthode avec les *templates* ne permet pas de définir tout le comportement dans les classes et il est nécessaire de déclarer le code *stencil* dans une fonction indépendante (hors des classes). Du point de vue de la programmation orienté objet cela ne respecte pas le principe d'encapsulation.

B Organisation des données dans la grille

L'organisation générique des données dans la classe `Grid` est présentée dans la Fig. 3.2. Pour une meilleure lisibilité, la Fig. 3.2 ne prend pas en compte la troisième dimension mais le principe est inchangé.

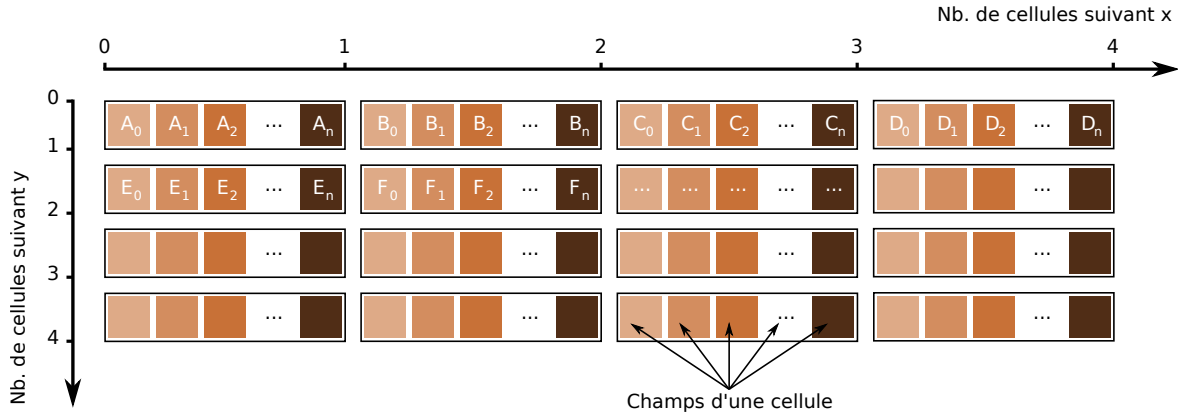

FIG. 3.2 – Organisation des données en mémoire pour une grille 2D de 4×4 cellules

Les cellules sont stockées selon l'axe des abscisses (modèle *row major* standard en `C++`) et pour chaque cellule ses champs sont stockés à la suite. Dans la Fig. 3.2, les lettres donnent l'ordre des cellules en mémoire alors que les numéros correspondent aux différents champs à l'intérieur d'une cellule. Dans la suite de ce mémoire, nous utiliserons un cas simplifié de grille avec uniquement un champ par cellule (cf. Fig. 3.3). Les classes `Grid1Field1D` et `Grid1Field2D` permettent de gérer facilement ce type de grille.

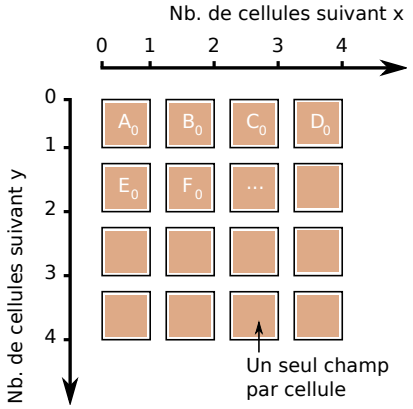

FIG. 3.3 – Organisation des données en mémoire pour une grille 2D (1 champ par cellule)

II Implémentation CPU *in-core*

II.1 Implémentation basique

L'implémentation naturelle d'un *stencil* au voisinage de Von Neumann en 1D est proposée dans le listing 3.1. Cet algorithme suit l'itération de Jacobi et les accès aux grilles `G1` (grille précédente) et `G2` (grille en cours de calcul) sont schématisés dans la Fig. 3.4.

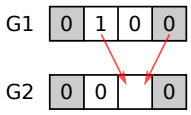

Fig. 3.4 – Accès mémoires pour un Jacobi 1D 2-point

```
1  // loop over iterations
2  for(int curIte = 0; curIte < nIte; curIte++)
3  {
4    // loop over cells
5    for(int i = 1; i < nCells -1; i++)
6      // stencil computation
7      G2[i] = a*G1[i-1] + b*G1[i+1];
8
9    // swap G1 and G2 pointers
10   TmpG = G1;
11   G1   = G2;
12   G2   = TmpG;
13 }
```

Listing 3.1 – Implémentation classique d'un Jacobi 1D 2-point

Dans le listing 3.1, on remarque la présence de deux niveaux de boucle : le premier autour des itérations (ligne 2) et le second autour des cellules (ligne 5). Lors du parcours des cellules, on commence volontairement par la seconde cellule et on termine par l'avant dernière : la première cellule ainsi que la dernière sont au bord du domaine et font l'objet de conditions aux limites de Dirichlet (cf. Chap. 1 Sec. III.2). Le calcul des cellules est effectué dans la boucle la plus profonde (ligne 7) : `G2[i]` est le résultat de l'addition de deux contributions (`a*G1[i-1]` et `b*G1[i+1]`). Chaque contribution se compose d'un élément dans la grille `G1` multiplié par le coefficient `a` pour le voisin de gauche (`G1[i-1]`) ou par le coefficient `b` pour le voisin de droite (`G1[i+1]`) (cf. Chap. 1 Sec. III.3). Enfin, la dernière opération avant le début d'une nouvelle itération est d'échanger les valeurs des pointeurs `G1` et `G2`. La nouvelle grille calculée `G2` devient la prochaine grille `G1`.

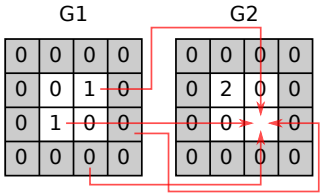

Fig. 3.5 – Accès mémoires pour un Jacobi 2D 4-point

La Fig. 3.5 et le listing 3.2 illustrent l'implémentation d'un algorithme *stencil* au voisinage de Von Neumann (itération de Jacobi) en 2D. Cette dernière ressemble beaucoup à l'implémentation en 1D modulo l'apparition d'une nouvelle boucle pour la deuxième dimension (ligne 5) et l'augmentation du nombre de contribution avec la prise en considération des voisins du haut (`c*G1[i,j-1]`) et du bas (`d*G1[i,j+1]`). Ces deux implémentations (1D et 2D) forment le socle de base pour les commentaires et optimisations abordés dans les sous sections suivantes.

```
1  // loop over iterations
2  for(int curIte = 0; curIte < nIte; curIte++)
3  {
4    // loop over cells
5    for(int j = 1; j < nCellsY -1; j++)
6      for(int i = 1; i < nCellsX -1; i++)
7        // stencil computation
8        G2[i,j] = a*G1[i-1,j  ] + b*G1[i+1,j  ] +
9                  c*G1[i  ,j-1] + d*G1[i  ,j+1];
10
11   // swap G1 and G2 pointers
12   TmpG = G1;
13   G1   = G2;
14   G2   = TmpG;
15 }
```

Listing 3.2 – Implémentation classique d'un Jacobi 2D 4-point

II.2 *Register Blocking*

Cette sous-section est dédiée à l'étude et l'optimisation des accès mémoires dans les caches du processeur pour un *stencil* 2D 4-point (valable en 3D mais pas en 1D).

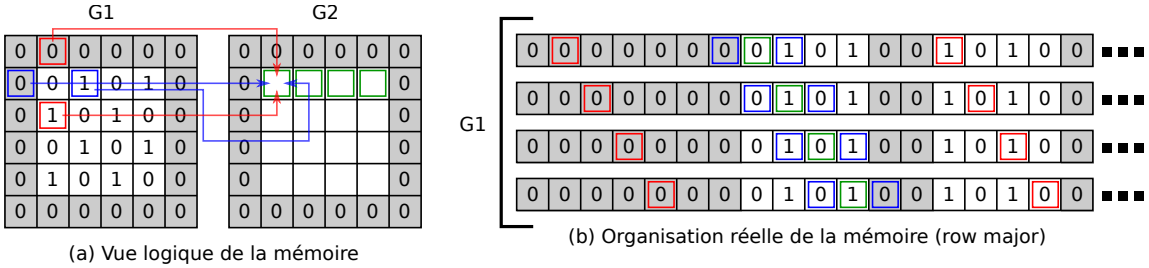

Fig. 3.6 – Accès originel aux données pour un *stencil* 2D

La Fig. 3.6 présente (a) une vision logique des accès mémoires effectués dans `G1` et dans `G2` lors du calcul du *stencil* (b) l'organisation réelle des données en mémoire avec les accès dans `G1`. Les accès dans `G1` sont en lecture alors que les accès dans `G2` sont en écriture. Dans le cadre d'un algorithme *stencil*, l'écriture dans `G2` n'est pas le plus gros problème car elle est contiguë en mémoire (cellules entourées de vert). Nous allons donc nous attarder sur les accès en lecture dans `G1` : les cellules entourées en bleu symbolisent les accès suivant l'axe des abscisses alors que celles entourées en rouge symbolisent les accès suivant l'axe des ordonnées. La Fig. 3.6 (b) montre les accès réels dans la mémoire pour 4 cellules traitées consécutives (une ligne représente les accès dans `G1` pour le calcul

d'une cellule). Attention les cellules entourées en vert ne sont pas des accès dans `G1` mais juste les témoins de la position de la cellule en cours de calcul dans `G2`. On remarque que les accès pour le calcul d'une cellule (cf. ligne 1 de la Fig. 3.6 (b)) ne sont absolument pas contigus en mémoire. Or, sur CPU, les données ne sont pas accédées une par une mais ligne par ligne sachant qu'une ligne contient plusieurs données contiguës. Pour atteindre les performances maximales (en terme de bande passante) il faut donc s'assurer que les accès soient les uns à la suite des autres sinon certaines données sont perdues/inutilisées.

L'article de H. Dursun et al. [DiNW+09] propose une méthode intéressante pour limiter les accès non contigus dans le cache L1 et la réutilisation des registres : le *Register Blocking*.

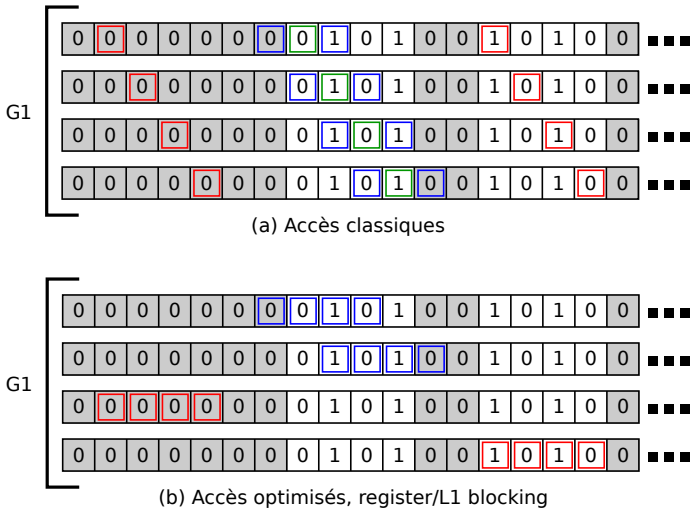

Fig. 3.7 – Maximisation de l'utilisation du *prefetcher* avec du *Register Blocking*

La Fig. 3.7 présente (a) les accès mémoires classiques (b) les accès mémoires avec l'utilisation du *Register Blocking*. Il apparaît clairement que les accès dans (b) sont bien plus contigus mais il n'est plus possible de calculer une cellule par ligne. Pour être capable d'effectuer de tels accès il faut donc traiter les cellules 4 par 4 et c'est là l'astuce de cette méthode. Au lieu de charger toutes les données nécessaires au calcul d'une cellule puis de calculer cette cellule, le *Register Blocking* propose de charger les données nécessaires pour n cellules puis ensuite de calculer les n cellules. Cela augmente le nombre consécutif d'accès mémoires contigus et permet une meilleure réutilisation des registres. Sur les processeurs modernes (x86 et ARM) il y a 16 registres dédiés au stockage des scalaires : un chargement de 4 cellules par 4 permet d'utiliser la totalité de ces registres voire un peu moins car il y a des accès redondants suivant l'axe des abscisses (cf. Fig. 3.7 (b) carrés bleus).

Le listing 3.3 montre l'implémentation de cette technique pour le *stencil* 2D présenté dans la section précédente. On remarque que pour traiter les cellules 4 par 4 la boucle de parcours selon x (ligne 6) a un pas de 4 et pour chaque itération dans cette boucle on remplit des tableaux de taille 4 pour les différents types de voisins (ligne 9 à 13). C'est à la suite de ces chargements que les 4 cellules sont réellement calculées (ligne 16 et 17). Enfin, les nouvelles valeurs sont recopiées dans la mémoire globale : la RAM (ligne 20). Cette technique a été implémentée dans `stenCINES` (classe `SimulationRB`) et les résultats seront commentés plus en détail dans le chapitre 4.

```
1  // loop over iterations
2  for(int curIte = 0; curIte < nIte; curIte++)
3  {
4    // loop over cells
5    for(int j = 1; j < nCellsY -1; j++)
6      for(int i = 1; i < nCellsX -1; i += 4) // use of register blocking
7      {
8        // contiguous accesses into global memory (good usage of prefetching)
9        double next[4], left[4], right[4], top[4], bottom[4];
10       for(int ii = 0; ii < 4; ii++) left  [ii] = G1[i-1+ii,j  ];
11       for(int ii = 0; ii < 4; ii++) right [ii] = G1[i+1+ii,j  ];
12       for(int ii = 0; ii < 4; ii++) top   [ii] = G1[i  +ii,j-1];
13       for(int ii = 0; ii < 4; ii++) bottom[ii] = G1[i  +ii,j+1];
14
15       // stencil computation in registers or L1 cache
16       for(int ii = 0; ii < 4; ii++)
17         next[ii] = a*left[ii] + b*right[ii] + c*top[ii] + d*bottom[ii];
18
19       // copying results (next) into global array (G2)
20       for(int ii = 0; ii < 4; ii++) G2[i+ii,j] = next[ii];
21     }
22
23   // swap G1 and G2 pointers
24   TmpG = G1;
25   G1   = G2;
26   G2   = TmpG;
27 }
```

Listing 3.3 – Implémentation d'un Jacobi 2D 4-point avec du *Register Blocking*

II.3 Vectorisation avec la méthode *Dimension Lifted and Transposed*

Pour se rapprocher des performances crêtes des processeurs récents il est indispensable d'utiliser les instructions vectorielles car elles permettent de calculer plusieurs éléments pendant un cycle d'horloge. Cependant ce type d'instruction est soumis à certaines conditions pour être efficace :

- les chargements des registres vectoriels doivent être alignés en mémoire,
- il faut un assez grand nombre d'instructions à exécuter (les instructions vectorielles possèdent une grande latence).

Dans le cadre d'un *stencil* et plus particulièrement suivant l'axe des abscisses, il n'est pas évident de charger les registres vectoriels en respectant l'alignement des données.

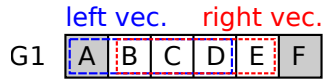

FIG. 3.8 – Conflit lors du chargement de deux vecteurs (*stream conflict*)

La Fig. 3.8 illustre le chargement de deux vecteurs de taille 4 pour le calcul d'un *stencil* en une dimension. Le premier vecteur (en bleu) est constitué des éléments (A, B, C, D) alors que le second vecteur (en rouge) est constitué des éléments (B, C, D, E). Il est donc

possible de calculer de façon vectorielle les nouvelles valeurs de B, C et D. Par contre, il n'est pas possible de calculer la nouvelle valeur de `A` car c'est une cellule au bord et de E car F n'est pas chargé dans les vecteurs. On remarque aussi que si le premier chargement (vecteur bleu) est aligné en mémoire alors le second (vecteur rouge) ne l'est pas puisqu'il est décalé de un : cela met en évidence un problème de conflit de flux (*stream conflict* dans la littérature anglaise).

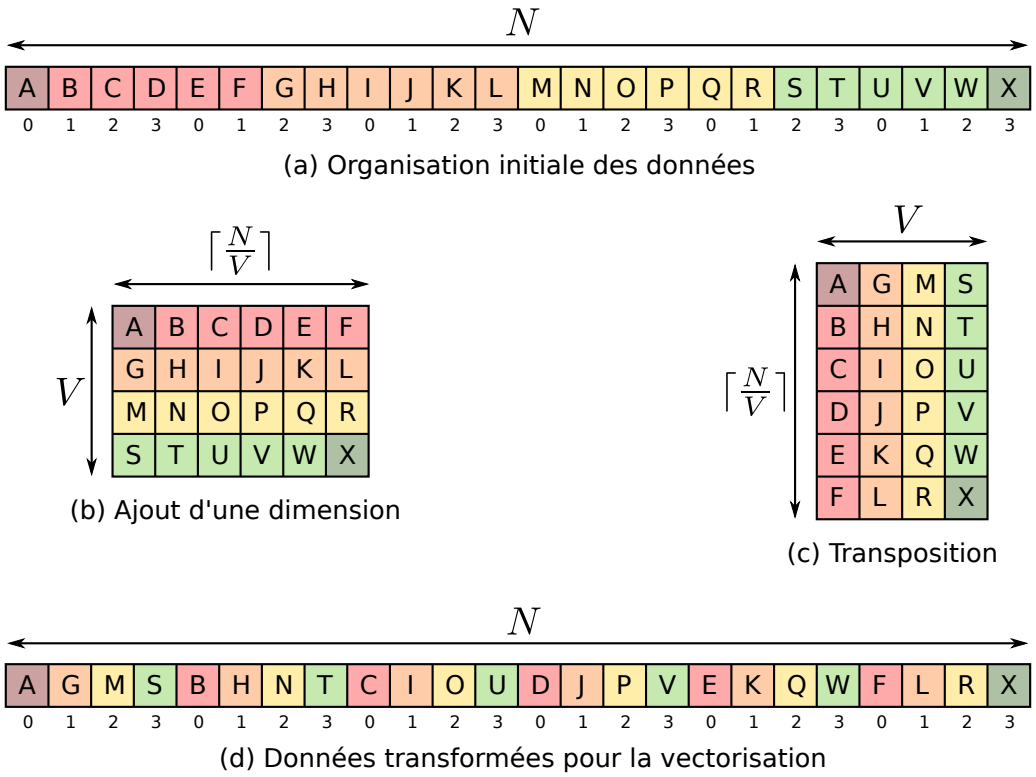

FIG. 3.9 – Méthode *Dimension Lifted and Transposed* pour résoudre les conflits

T. HENRETTY et al. proposent la méthode DLT [HSP+11] (*Dimension Lifted and Transposed*) pour éviter ces conflits. Cette dernière consiste en une modification de la disposition des données en mémoire de manière à limiter les accès non alignés. La Fig. 3.9 illustre la méthode en une dimension (les autres dimensions ne posent pas de problèmes pour la vectorisation). La réorganisation des données passe par la construction d'une deuxième dimension de la taille des registres vectoriels (ici $V = 4$). La première dimension (*Dimension Lifted*) est déduite de la seconde : $\lceil \frac{N}{V} \rceil$ avec N le nombre de cellules suivant la dimension x (cf. Fig. 3.9 (b)). Enfin, cette nouvelle matrice de données est transposée (*Transposed*) : V devient la dimension suivant x et $\lceil \frac{N}{V} \rceil$ devient la dimension suivant y (cf. Fig. 3.9 (c)). La représentation des données en mémoire après l'application de DLT est présentée par la Fig. 3.9 (d) : on remarque que la structure a bien changé par rapport à la Fig. 3.9 (a).

Après cette modification, le chargement des vecteurs est effectué de manière parfaitement alignée dans la mémoire à l'exception des vecteurs aux limites ((A, G, M, S) et (F, L, R, X) dans la Fig. 3.9 (c)). Ces derniers font l'objet d'un traitement scalaire ou d'un traitement vectoriel particulier avec des décalages dans les registres. Plus la dimension x est grande plus le traitement de ces vecteurs aux limites devient négligeable.

Cette méthode a été implémentée dans `stenCINES` (classe `SimulationDLT`) et elle permet d'atteindre des performances nettement plus élevées qu'une version classique tant que la taille des données de la grille n'excède pas la taille des caches.

II.4 *Cache Blocking*

Dans cette sous-section nous considérons que quand la taille des données dépasse la taille des caches, les codes *stencils* sont limités par la bande passante mémoire (cela n'est pas vrai dans le cas où l'ordre du *stencil* est très élevé mais nous ne traiterons pas ce type de *stencils* dans ce mémoire). Dans ce cadre, il est nécessaire de minimiser le nombre d'accès à la mémoire globale. Pour un *stencil*, le nombre minimal d'accès à la mémoire est deux fois le nombre de cellules dans la grille : n cellules en lecture dans `G1` et n cellules en écriture dans `G2` soit $2n$. Comme expliqué dans la section II.2, les accès en écriture dans `G2` ne sont pas les plus problématiques car ils sont parfaitement contigus : pour le calcul d'une cellule on stocke sa valeur dans la case mémoire qui lui correspond directement. La lecture dans `G1` est plus problématique : dans le cas d'un *stencil* 2D 4-point il faut accéder 4 cases mémoires pour le calcul de chaque cellule. Si les 4 lectures sont faites dans la mémoire globale alors il y a $4n$ accès au lieu des n optimaux.

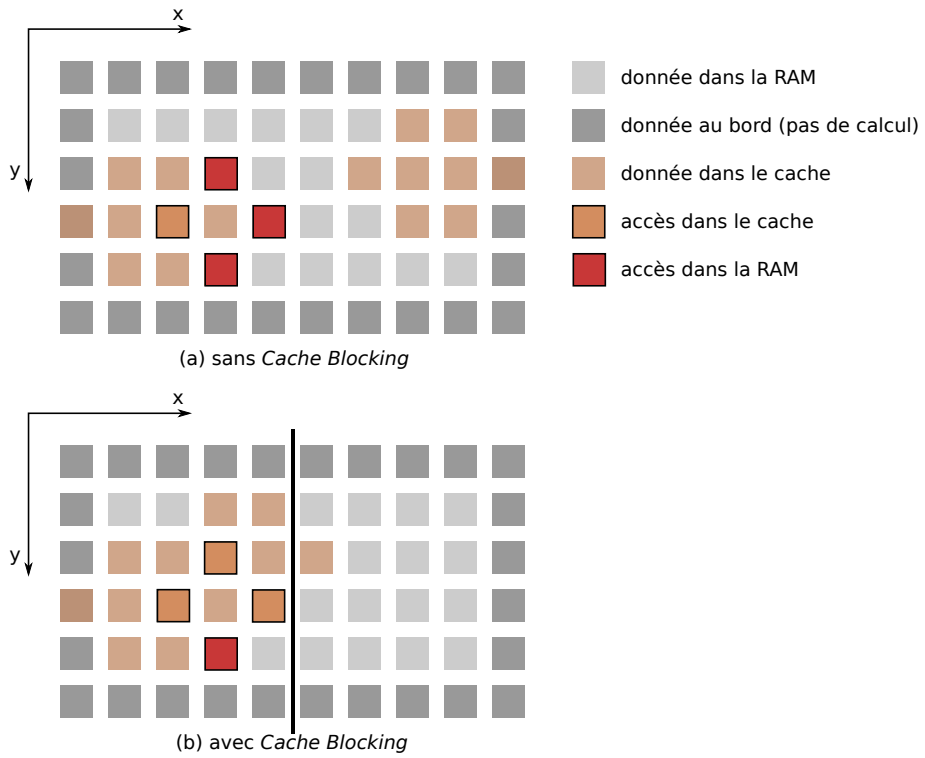

FIG. 3.10 – Accès mémoires dans `G1` avec et sans *Cache Blocking*

Heureusement les caches sont là pour éviter ce problème, les données qui viennent d'être lues sont gardées dans les caches du processeur afin d'être relues beaucoup plus rapidement lors des prochains accès. Cependant la taille du cache est par définition inférieure à la taille de la mémoire globale (RAM) : il n'est donc pas possible de mettre toutes les données en cache. La Fig. 3.10 (a) met en évidence les données qui sont dans

le cache (couleur orange) et les données qui sont dans la mémoire globale (couleur grise) pour un cache qui peut contenir au maximum 16 cellules. On remarque ici que le cache permet d'éviter un chargement dans la mémoire globale : celui de la cellule de gauche. Il y a donc 3 accès à la mémoire globale par cellule alors que l'optimum serait d'en avoir 1 seul.

La résolution de ce problème passe par l'implémentation de la célèbre méthode de *Cache Blocking*. L'idée est de parcourir les données par bloc de manière à maximiser la réutilisation des caches. La Fig. 3.10 (b) illustre cette méthode, ici la grille est découpée en deux parties (gauche et droite) selon l'axe des ordonnées. Le parcours des cellules est d'abord effectué dans la partie gauche puis ensuite dans la partie droite. La dimension des blocs selon x est de 5 ce qui permet au cache de retenir les valeurs des cellules du haut et de droite : le nombre d'accès à la mémoire globale est réduit à 1 (seule la lecture de l'élément du bas est obligatoirement faite dans la mémoire globale). La condition pour que le *Cache Blocking* soit efficace est la suivante (en deux dimensions) :

$$3 \times blockDim * dataSize \leq \frac{cacheSize}{2 \times nThreads}, \tag{3.1}$$

avec $blockDim$ la dimension du bloc selon x, $dataSize$ le nombre d'octets de chaque donnée, $cacheSize$ la taille du cache le plus grand en octet et $nThreads$ le nombre de *threads* lancés en parallèle pour calculer le *stencil* 2D. Dans l'Eq. 3.1 la taille du cache est divisée par deux : la performance des caches est moindre pour une utilisation qui dépasse plus de la moitié de la capacité maximale.

Attention cependant, si $blockDim$ est trop petit alors il y a de fortes chances pour que le surcoût du *Cache Blocking* soit plus important que le gain apporté. Pour minimiser ce surcoût, la plus grande valeur de $blockDim$ est souvent choisie ainsi :

$$blockDim = \frac{cacheSize}{6 \times nThreads \times dataSize}.$$

Remarque : si $blockDim$ est plus grand que la taille de la grille selon x alors le *Cache Blocking* est inutile. De nos jours la taille du cache L3 devient très grande (> 20 Mo) ce qui réduit drastiquement le nombre de configurations de la grille où le *Cache Blocking* apporte un gain.

II.5 *Non-temporal stores*

Dans les précédentes sections l'écriture dans `G2` a été négligée car jugée moins critique que la lecture dans `G1`. Cependant grâce au *Cache Blocking*, la plupart des accès irréguliers en lecture dans la mémoire globale ont été résolus et cette section s'attache à la problématique de sauvegarde des données dans `G2`.

Au premier abord il est difficile d'imaginer qu'il soit possible d'améliorer les performances lors de l'écriture pour un code de type *stencil* : les accès sont parfaitement contigus dans la mémoire globale (RAM). En réalité, le frein vient du mécanisme de cohérence de cache : avant d'écrire une données en mémoire vive (RAM) une ligne de cache est d'abord remontée dans le L3 (mécanisme *write allocate* dans la littérature anglaise). Ce mécanisme permet un gain de performance si la donnée nouvellement écrite et ses

voisines sont rapidement ré-accédée en lecture ensuite. Dans le cadre d'un code de type *stencil* limité par la bande passante mémoire le *write allocate* n'apporte pas de gain de performance puisque les données écrites dans `G2` ne sont par relues avant la prochaine itération. Au contraire, cela pénalise grandement le code quand il est limité par les accès à la RAM : pour chaque calcul de cellule il y a 2 accès en lecture et 1 accès en écriture soit 3 accès au total (au lieu des 2 nécessaires).

Certains compilateurs permettent de désactiver ce mécanisme de mise en cache lors de l'écriture dans la RAM avec l'utilisation des *non-temporal stores* (sauvegardes non-temporelles). Cette option améliore grandement les performances dans le cas d'un code *stencil* limité par la mémoire puis qu'il y a plus que les 2 accès nécessaires à la RAM au lieu de 3 : le gain théorique est de 33%.

III Implémentation CPU *multi-threads*

Cette section décrit l'implémentation parallèle *multi-threads* mise en œuvre dans `stenCINES` pour un *stencil* en deux dimensions. L'approche choisi n'impacte pas l'organisation des données en mémoire même si le *multi-padding* présenté dans l'article de J. Jaeger et D. Barthou [JB12] semble apporter des gains de performances non négligeables.

III.1 Stratégie de parallélisation

Dans `stenCINES` le parallélisme est effectué au niveau du parcours des cellules selon l'axe des ordonnées (comme pour le code `Pochoir` présenté dans l'article de Y. Tang et al. [TCK+11]) : chaque *thread* s'occupe d'une ou de plusieurs lignes de cellules (décomposition par ligne cf. Fig. 3.11). Ce choix a été motivé par deux raisons :

- la boucle de parcours des cellules selon y est effectuée moins de fois que la boucle de parcours des éléments selon x : le surcoût lié à l'attribution des indices pour chaque *thread* est moindre,
- la parallélisation de cette boucle est indépendante des implémentations mises en œuvre (basique, *Cache Blocking*, *Register Blocking* et DLT).

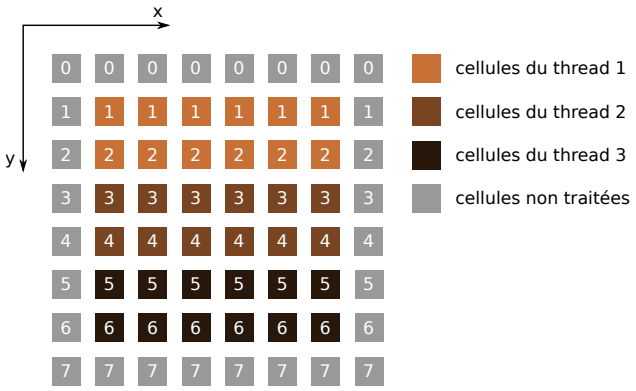

Fig. 3.11 – Exemple de décomposition de la grille par ligne pour le *multi-threading* en fonction des indices de la boucle selon l'axe des ordonnées

Le listing 3.4 montre l'implémentation *multi-threads* avec OpenMP dans `stenCINES`. On remarque que la création des *threads* est effectuée une seule fois avant la boucle des itérations (ligne 4) de façon à minimiser le surcoût lié à la création des *threads*. Un maximum d'indépendance est donné aux *threads* avec la recopie locale de `G1`, `G2`, `TmpG` et `nIte` (ligne 1). Le partage des lignes entre les *threads* est réalisé ligne 6 avec la directive `#pragma omp for`. La clause `schedule(runtime)` spécifie que le distribution est laissée à l'utilisateur lors de l'exécution du code. Cela permet de facilement effectuer des tests sans avoir à recompiler le code et parfois la meilleure distribution dépend de la grille : il ne serait pas raisonnable de compiler un code différent pour chaque grille. Enfin il y a une barrière implicite à la fin de la boucle de parcours des cellules selon l'axe des ordonnées ainsi qu'à la fin du parcours des itérations.

```
1  #pragma omp parallel firstprivate(G1, G2, TmpG, nIte)
2  {
3  // loop over iterations
4  for(int curIte = 0; curIte < nIte; curIte++)
5  {
6  #pragma omp for schedule(runtime)
7    // loop over cells
8    for(int j = 1; j < nCellsY -1; j++)
9      for(int i = 1; i < nCellsX -1; i++)
10       // stencil computation
11       G2[i,j] = a*G1[i-1,j  ] + b*G1[i+1,j  ] +
12                 c*G1[i  ,j-1] + d*G1[i  ,j+1];
13
14   // swap G1 and G2 pointers
15   TmpG = G1;
16   G1   = G2;
17   G2   = TmpG;
18 }
19 }
```

Listing 3.4 – Implémentation parallèle d'un Jacobi 2D 4-point avec OpenMP

III.2 Initialisation des données en parallèle

Les architectures NUMA (*Non Uniform Memory Access*) sont de plus en plus répandues dans les supercalculateurs. Cela se traduit par plusieurs sockets sur une même carte mère. Chaque socket possède son banc mémoire privilégié et communique par les liens QPI pour accéder aux bancs des autres sockets. Ce passage par les liens QPI diminue fortement les performances. Il faut donc minimiser les transits entres sockets et allouer les données nécessaires aux traitements d'un socket sur son banc mémoire privilégié.

On pourrait intuitivement penser que les données sont placées en mémoire lors de l'allocation mais ce n'est pas le cas. En réalité, le système n'alloue pas les pages mémoires dans la RAM avant qu'elles ne soient affectées une première fois. Pour que chaque socket travaille sur son propre banc mémoire il faut donc s'assurer que ce sont les *threads* de ce socket qui initialisent les données et qui travaillent dessus. Cela implique deux choses lors de la programmation OpenMP :

- il est indispensable d'initialiser les données en parallèle,
- la répartition dynamique des indices n'est pas optimale dans un contexte NUMA.

IV Portage sur accélérateur de calcul

Plusieurs travaux ont montrés que les codes de type *stencil* pouvaient tirer avantageusement parti des accélérateurs ([HPS12], [JB12], [MK10], [CU10], [JB12]) et cette section s'attache à l'implémentation efficace de ce type de code sur des GPUs. Les optimisations liées à la mémoire sont parfois spécifiques à l'architecture *Kepler* mais le principe reste le même sur les précédentes architectures.

IV.1 Implémentation basique

Sur GPU, le code est nativement parallèle et l'implémentation passe par un découpage du travail en grille de *threads* puis en blocs. Il faut donc déterminer quelle est la quantité de travail à effectuer pour chaque *threads* : cette démarche est qualifiée de choix du grain de calcul. Dans `stenCINES` c'est un parallélisme à grain fin qui a été retenu : un *thread* calcule une cellule de la grille.

```
1  __global__ void kerStencil( const float *G1, float *G2,
2                              float a, float b, float c, float d,
3                              int nCellsX, int nCellsY )
4  {
5    // get thread position in the grid
6    int i = blockIdx.x * blockDim.x + threadIdx.x;
7    int j = blockIdx.y * blockDim.y + threadIdx.y;
8
9    // compute stencil if this thread cell position is not a border cell
10   if((i > 0 && i < nCellsX -1) && (j > 0 && j < nCellsX -1))
11     G2[i,j] = a*G1[i-1,j  ] + b*G1[i+1,j  ] +
12               c*G1[i  ,j-1] + d*G1[i  ,j+1];
13 }
14
15 // set block and grid sizes for GPU execution
16 dim3 blockSize = {16, 16};
17 dim3 gridDim   = {nCellsX / blockSizeX, nCellsY / blockSizeY};
18
19 // loop over iterations (running on CPU)
20 for(int curIte = 0; curIte < nIte; ++curIte)
21 {
22   // launch kernel (running on GPU)
23   kerStencil<<<gridDim,blockSize>>>( G1, G2, a, b, c, d );
24
25   // swap G1 and G2 pointers
26   TmpG = G1;
27   G1   = G2;
28   G2   = TmpG;
29 }
```

Listing 3.5 – Implémentation basique d'un Jacobi 2D 4-point sur GPU

Le listing 3.5 expose le code simplifié de l'implémentation GPU CUDA du *stencil* 2D 4-point présenté précédemment en sous-section II.1. La fonction `kerStencil` est exécutée sur GPU (ligne 1 à 11) alors que le reste du code est exécuté sur CPU (ligne 13 à 27). La taille des blocs CUDA (ligne 14) est fixée à 256 *threads* ($16 \times 16 = 256$) et la dimension de la grille CUDA est calculée à partir du nombre de cellules et de la taille des blocs (ligne

15) (ici on suppose que les nombres de cellules selon x et y sont des multiples de 16). La boucle itérative est effectuée sur CPU (ligne 18) car le seul moyen de synchroniser les *threads* entre les différents blocs est de terminer le *kernel* (et donc de revenir sur l'hôte CPU). De base, les *threads* GPU sont synchronisés après chaque appel à un *kernel* et c'est très important de s'assurer que tous les *threads* aient terminés leurs calculs et écritures dans la mémoire globale avant d'échanger les pointeurs `G1` et `G2` (ligne 24 à 26) sinon il n'est pas possible de garantir l'exactitude des résultats dans `G2`. Le *kernel* est appelé ligne 21, la syntaxe CUDA est un peu particulière : les dimensions de la grille et la taille des blocs sont spécifiées entre les triples chevrons avant le passage des arguments.

IV.2 Optimisation des accès mémoires

Sur GPU, comme sur CPU, l'optimisation des accès mémoires pour un *stencil* de faible ordre est un enjeu crucial puisque la mémoire globale est le facteur limitant de l'algorithme. Dans un premier temps nous avons travaillé sur l'amélioration de la bande passante entre la mémoire globale et les *threads*. L'article de J. Luitjens [Lui13] explique que les accès par vecteur à la mémoire globale permettent de diminuer le nombre d'instructions à exécuter et ainsi augmenter la performance des noyaux de calcul. Nous avons donc opté pour des accès par vecteur de taille 2. En d'autres termes le grain de calcul a été augmenté d'une cellule par *threads* soit deux cellules par *threads* au total (cf. Fig. 3.12).

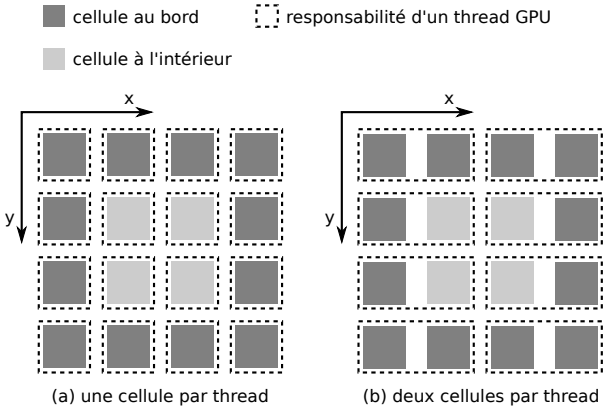

Fig. 3.12 – Nombre de cellules par *thread* en fonction du grain de calcul

Cependant, cette technique pose un problème dans le cadre d'un *stencil* : il faut maintenant gérer le cas où un *thread* est en charge d'une cellule normale et d'une cellule au bord. Cela augmente le nombre de branchements (conditions *if*). Malheureusement, sur GPU, tous les *threads* d'un *warp* exécutent exactement les mêmes instructions en même temps et les branchements sont souvent synonymes d'une diminution du nombre de *threads* actifs. Par exemple, si seulement la moitié des *threads* d'un même *warp* vérifient une condition, alors l'autre moitié est en attente. Enfin les branchements rajoutent une pression supplémentaire sur les noyaux de calcul en augmentant leur nombre d'instructions ce qui peut diminuer leur performance si ils sont limités par la latence ou les instructions.

Une stratégie de *padding* (remplissage en français) a été utilisée afin de diminuer le nombre de conditions à l'intérieur du *kernel* : la Fig. 3.13 illustre cette méthode.

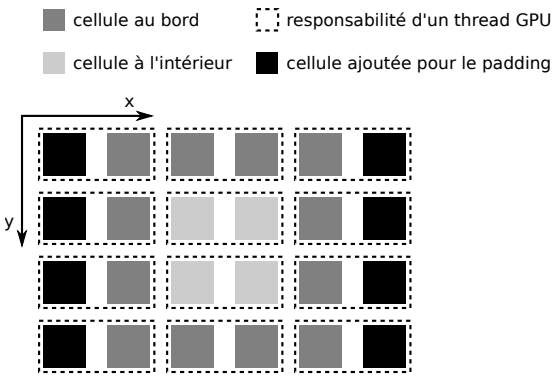

FIG. 3.13 – Nombre de cellules par *thread* avec du *padding*

Pour un *stencil* d'ordre 1, il suffit d'ajouter une donnée en mémoire avant et après chaque ligne (cellules noires). Ces données ne seront pas utilisées pour le calcul mais elle permettent, lors de la répartition des cellules entre les *threads*, de ne plus avoir le cas particulier où un *thread* possède deux types différents de cellule (une cellule à calculer et une cellule au bord). Cette technique augmente la taille de la grille de 2 cellules suivant l'axe des abscisses ce qui est négligeable quand la grille est assez grande.

Nous avons ensuite travaillé sur la minimisation du nombre d'accès en lecture dans la mémoire globale. Pour y parvenir nous avons utilisé la mémoire cache en lecture seule (uniquement disponible depuis l'architecture *Kepler*). Cette mémoire est bien plus rapide que la mémoire globale et elle est commune à chaque *threads* d'un SMX (elle ressemble beaucoup à la mémoire partagée). L'implémentation est relativement facile puisqu'il suffit de décorer le pointeur sur la grille `G1` avec les mots clefs suivant :

- `const` : spécifie au compilateur que les accès à la mémoire sont en lecture seule,
- `__restrict` : spécifie au compilateur qu'il n'y a pas de risque d'*aliasing* (aucun autre pointeur ne pointe sur la même portion mémoire).

Les accès à la mémoire globale sont ensuite automatiquement mis dans le cache en lecture seule ce qui améliore très nettement les performances du *stencil*. Il est aussi possible de faire appel au cache en lecture seule en utilisant un appel intrinsèque (cf. `__ldg()`) mais cela spécialise le code pour l'architecture *Kepler*. Il est bon de noter que cette optimisation ne fonctionne qu'à partir de *Kepler* et il faudrait utiliser la mémoire partagée pour avoir le même comportement sur les architectures plus anciennes (*Fermi* et *Tesla*).

L'activation du cache en lecture seule sur GPU a à peu près le même effet que le *Cache Blocking* sur CPU : il y a un accès en lecture dans la mémoire globale par cellule, les autres accès sont effectués dans le cache.

IV.3 *Temporal Blocking*

Après les précédentes optimisations, le code *stencil* est toujours principalement limité par la bande passante mémoire et cela malgré la réduction du nombre d'accès. De plus, sur GPU, le déséquilibre entre la puissance de calcul et la bande passante mémoire est encore plus important que sur CPU : la performance atteignable est très loin de la performance crête quand le code est fortement limité par la bande passante mémoire. Afin de diminuer cette contrainte sur la mémoire globale nous avons opté pour une stratégie de *Temporal*

Blocking. Cette méthode permet de calculer plusieurs itérations de l'algorithme *stencil* sans repasser par la mémoire globale. Cela se traduit par l'implémentation d'un nouveau *kernel* qui calcule plusieurs itérations à la fois (au lieu d'une seule). Pour pouvoir calculer n itérations sans faire n accès en lecture dans la mémoire globale il faut nécéssairement recalculer certaines valeurs de la grille. Avec le *Temporal Blocking* chaque bloc de *threads* recopie une plus grande partie de la grille `G1` dans la mémoire *on-chip* (mémoire cache, mémoire partagée) que dans l'implémentation basique alors que chaque bloc de *threads* continue de calculer la même partie de la grille `G2`. Le nombre de *threads* utilisés dans chaque bloc est augmenté pour pouvoir calculer les valeurs des cellules lues dans `G1`. Les *threads* qui ne servent qu'à calculer des valeurs de cellules intermédiaires font parti de ce que l'on appellera le halo. Les *threads* du halo n'écrivent pas dans `G2` à la fin du *kernel*.

Cette méthode a aussi ses limites puisque plus le nombre d'itérations calculées est grand plus le nombre de *threads* qui recalculent des valeurs est élevé. Il faut donc trouver un juste milieu et limiter l'impact des accès à la mémoire globale tout en s'assurant que le nombre de *threads* effectifs (les *threads* qui ne font pas parti du halo) permette un gain de performance.

Chapitre 4

Expérimentations

I Un cas concret : l'équation de la chaleur

Nous étudions ici l'équation de la chaleur dans un milieu immobile isotrope homogène, avec des coefficients thermodynamiques constants (α). L'objectif est de discrétiser le problème afin de pouvoir simuler la diffusion de la chaleur (T) avec les machines de calcul. Notons que nous utiliserons toujours un pas de temps (Δt) constant.

I.1 Équation en 1D

Soit T la température en degré Celsius, t le temps en seconde, α le coefficient matériel de transfert de chaleur, x une distance en mètre, l'équation de la chaleur en 1D est exprimée comme suit :

$$\frac{\delta T}{\delta t} = \alpha \frac{\delta^2 T}{\delta x^2}. \tag{4.1}$$

La partie gauche de l'équation 4.1 peut être discrétisée et approximée en utilisant la méthode des différences finies (approximation de la dérivée première en t) :

$$\frac{\delta T(x,t)}{\delta t} \approx \frac{T(x,t+\Delta t) - T(x,t)}{\Delta t}. \tag{4.2}$$

Plus Δt est petit plus l'approximation est proche de la réalité.

De même, la partie droite de l'équation peut être discrétisée et approximée en utilisant la méthode des différences finies (approximation de la dérivée seconde en x) :

$$\alpha \frac{\delta^2 T(x,t)}{\delta x^2} \approx \alpha \frac{T(x+\Delta x,t) - 2T(x,t) + T(x-\Delta x,t)}{(\Delta x)^2}. \tag{4.3}$$

Plus Δx est petit plus l'approximation est proche de la réalité.

Au final, il est possible de calculer une approximation de la température T à $t+\Delta t$ en tout point x avec l'éq. 4.2 et l'éq. 4.3 :

$$\frac{T(x,t+\Delta t) - T(x,t)}{\Delta t} \sim \alpha \frac{T(x+\Delta x,t) - 2T(x,t) + T(x-\Delta x,t)}{(\Delta x)^2}$$

$$\Leftrightarrow$$

$$T(x,t+\Delta t) \sim T(x,t) + \alpha \Delta t \frac{T(x+\Delta x,t) - 2T(x,t) + T(x-\Delta x,t)}{(\Delta x)^2}. \tag{4.4}$$

A Condition de stabilité

Pour que l'équation de la chaleur reste stable il faut respecter la condition suivante :

$$\lambda = \alpha \frac{\Delta t}{(\Delta x)^2} < 0,5, \tag{4.5}$$

avec λ le coefficient de stabilité. Cette condition permet de choisir α, Δt et Δx de façon à rester proche du modèle physique.

I.2 Équation en 2D

L'équation de la chaleur en 2D est exprimée comme suit :

$$\frac{\delta T}{\delta t} = \alpha \left[\frac{\delta^2 T}{\delta x^2} + \frac{\delta^2 T}{\delta y^2} \right]. \tag{4.6}$$

La partie gauche de l'équation 4.6 peut être discrétisée et approximée (dérivée première en t) :

$$\frac{\delta T(x,y,t)}{\delta t} \approx \frac{T(x,y,t+\Delta t) - T(x,y,t)}{\Delta t}. \tag{4.7}$$

De même, la partie droite de l'équation peut être discrétisée et approximée (dérivée seconde en x et en y) :

$$\begin{aligned} \alpha \left[\frac{\delta^2 T(x,y,t)}{\delta x^2} + \frac{\delta^2 T(x,y,t)}{\delta y^2} \right] \approx \alpha \Bigg[& \frac{T(x+\Delta x,y,t) - 2T(x,y,t) + T(x-\Delta x,y,t)}{(\Delta x)^2} \\ & + \frac{T(x,y+\Delta y,t) - 2T(x,y,t) + T(x,y-\Delta y,t)}{(\Delta y)^2} \Bigg]. \end{aligned} \tag{4.8}$$

Au final, il est possible de calculer une approximation de la température T à $t + \Delta t$ en tout point $[x, y]$ avec l'éq. 4.7 et l'éq. 4.8 :

$$\begin{aligned} \frac{T(x,y,t+\Delta t) - T(x,y,t)}{\Delta t} \sim \alpha \Bigg[& \frac{T(x+\Delta x,y,t) - 2T(x,y,t) + T(x-\Delta x,y,t)}{(\Delta x)^2} \\ & + \frac{T(x,y+\Delta y,t) - 2T(x,y,t) + T(x,y-\Delta y,t)}{(\Delta y)^2} \Bigg] \end{aligned}$$

$$\Leftrightarrow$$

$$\begin{aligned} T(x,y,t+\Delta t) \sim T(x,y,t) + \alpha \Delta t \Bigg[& \frac{T(x+\Delta x,y,t) - 2T(x,y,t) + T(x-\Delta x,y,t)}{(\Delta x)^2} \\ & + \frac{T(x,y+\Delta y,t) - 2T(x,y,t) + T(x,y-\Delta y,t)}{(\Delta y)^2} \Bigg] \end{aligned} \tag{4.9}$$

La Fig. 4.1 illustre la méthode de discrétisation utilisée pour le cas 2D.

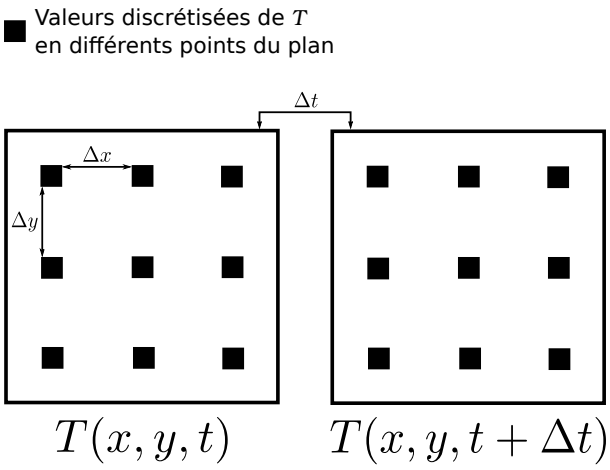

FIG. 4.1 – Exemple de discrétisation dans le plan, grille 3×3

A Condition de stabilité

Pour que l'équation de la chaleur reste stable il faut respecter la condition suivante :

$$\lambda = \alpha \left[\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2} \right] < 0,5, \tag{4.10}$$

avec λ le coefficient de stabilité. Cette condition permet de choisir α, Δt, Δx et Δy de façon à ce que la physique soit respectée.

Dans la suite de ce chapitre nous nous focalisons sur l'étude des performances du *stencil* défini par la discrétisation de l'équation de la chaleur en deux dimensions. Ce *stencil* est caractérisé par un voisinage de VON NEUMANN 5-point d'ordre 1 (le point centré est utilisé pour le calcul des cellules). L'intensité arithmétique de ce *stencil* est de 4,5 (cf. Tab. 1.1) : le ratio calcul/données est faible.

II Visualisation de la diffusion de la chaleur

Afin de vérifier la bonne diffusion de la chaleur dans le plan nous avons utilisé *ParaView* (logiciel de visualisation de données). `stenCINES` étant capable d'écrire dans le format `vtk` (*Visualization Toolkit*) la visualisation des données a été facile.

Nous avons regardé la diffusion de la chaleur pour une petite grille de 128×128 cellules représentant $16,64$ cm^2 dans un milieu immobile isotrope homogène ayant pour coefficient thermodynamique $0,4$. La grille initiale (voir Fig. 4.2) est caractérisée par un point froid en bas à gauche (-30 degrés) et un point chaud en haut à droite ($+30$ degrés) et le pas de temps a été fixé à 1×10^{-6} seconde. Attention, ici la hauteur représente la température et la grille est bien calculée en deux dimensions. Les Fig. 4.3 et Fig. 4.4 montrent respectivement l'évolution de la chaleur après 600 et 1200 itérations suivant deux types de conditions aux limites : (a) de DIRICHLET et (b) périodiques. Quand les conditions aux limites sont de DIRICHLET alors on voit très nettement que les valeurs aux bords restent fixes (1 degré). Par contre quand la diffusion est périodique alors le *stencil* s'applique aussi sur les bords : la température change et impacte les bords opposés.

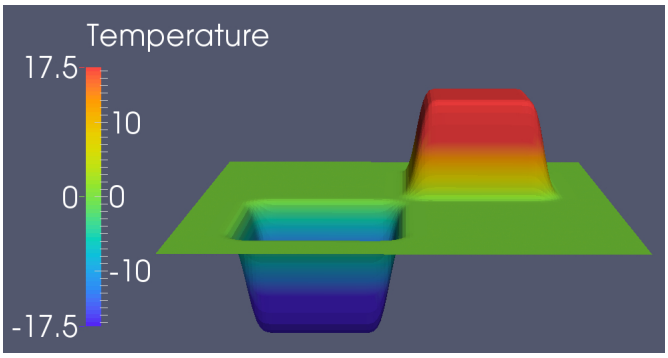

FIG. 4.2 – Visualisation de la chaleur avec *ParaView* (état initial)

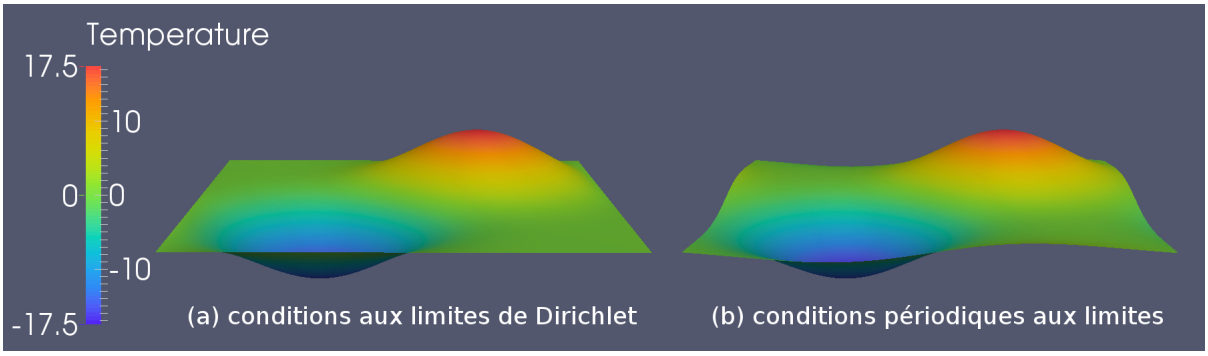

FIG. 4.3 – Visualisation de la chaleur avec *ParaView* (600 itérations)

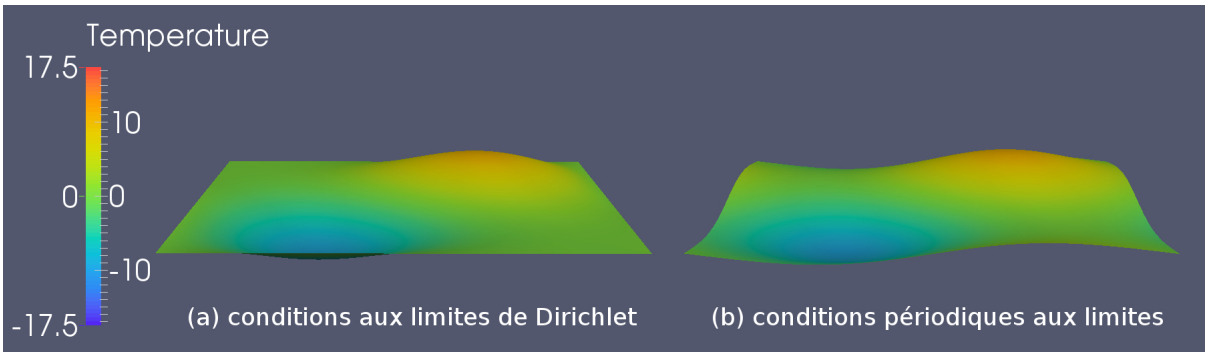

FIG. 4.4 – Visualisation de la chaleur avec *ParaView* (1200 itérations)

III Configurations de test

Plusieurs processeurs et accélérateurs ont été utilisés lors des tests : un CPU ARM S4 Pro, un CPU x86 Xeon E5-2650 et un GPU Tesla K20c. Ces trois processeurs et accélérateurs sont représentatifs des différentes architectures modernes disponibles en 2013/2014 sachant qu'il y a toujours un décalage entre la date de lancement d'un produit et la date réelle ou il est accessible. La table 4.1 résume les caractéristiques des différents modèles utilisés.

Modèle	**S4 Pro APQ8064**	**Xeon E5-2650**	**Tesla K20c**
Constructeur	*Qualcomm*	*Intel*	*Nvidia*
Architecture	*Krait 200*	*Sandy Bridge*	*Kepler*
Date de lancement	01/2012	01/2012	11/2012
Finesse de gravure	28 nm	32 nm	28 nm
Fréquence min.	0,384 GHz	2,0 GHz	0,706 GHz
Fréquence max.	1,670 GHz	2,8 GHz	0,706 GHz
Nombre de cœurs (ou SMX)	4	8	13
Taille du cache L1	32 Ko	64 Ko	64 Ko + 48 Ko
Taille du cache L2	2048 Ko	256 Ko	1536 Ko
Taille du cache L3	—	20 Mo	—
Type de mémoire vive (RAM)	DDR2	DDR3	GDDR5
Quantité de mémoire vive	2 Go	16 Go	5 Go
Bande passante mém. théo.	6,4 Go/s	50 Go/s	208 Go/s
Bande passante mém. mesurée	5,5 Go/s	35 Go/s	145 Go/s
Type d'instructions SIMD	NEON	AVX	—
Taille des instr. SIMD	128 bit	256 bit	—
Nb. d'instr. SIMD par cycle	1	2	—
Unités de calcul par SMX SP	—	—	192
Unités de calcul par SMX DP	—	—	64
Support des instructions FMA	Non	Non	Oui
Puissance crête SP	26.72 Gflop/s	256 Gflop/s	3524 Gflop/s
Puissance crête DP	—	128 Gflop/s	1175 Gflop/s
TDP max.	3 watts	95 watts	225 watts
Performance par watt SP	8,9 Gflops/W	2,7 Gflops/W	15.7 Gflops/W
Performance par watt DP	—	1,4 Gflops/W	5.2 Gflops/W

Tab. 4.1 – Récapitulatif des spécifications techniques des différents processeurs et accélérateur utilisés pour les tests

Dans la Tab. 4.1, le terme de cache L1 n'est pas tout à fait exact pour la Tesla K20c : il fait ici référence au cache L1, à la mémoire partagée (64 Ko) et à la mémoire cache en lecture seule (48 Ko). La bande passante mémoire a été mesurée avec l'ECC activé quand celui ci était disponible (Xeon et Tesla). Les abréviations SP et DP signifient respectivement simple précision et double précision. Les puissances crêtes de l'ARM et de la Tesla ont été calculées à partir de leur fréquence maximale alors que la performance crête du Xeon a été calculée à partir de la fréquence minimale : la technologie *Turbo*

Boost[1] était désactivée lors des tests. Le "TDP max." (*Thermal Design Power*, TDP) fait référence à la consommation énergétique maximale : les valeurs pour le Xeon et la Tesla sont celles données par les constructeurs alors que la valeur de l'ARM a été estimée suivant un article provenant d'AnandTech [Shi13] (*Qualcomm* ne donne pas ce type d'information). Attention, le TDP des processeurs (ARM et Xeon) ne prend pas en compte la consommation de RAM alors que sur accélérateur il comprend la consommation complète de la carte avec la mémoire globale (équivalent de la RAM) : nous considérerons ici que cela est négligeable.

La performance par watt est une estimation théorique du nombre d'opérations sur des flottants par seconde pour un watt. Cette valeur est calculée à partir de la crête et de la consommation maximale (TDP) du processeur. On remarque que la rentabilité énergétique théorique est bien meilleure sur GPU que sur CPU même si l'écart est moins marqué pour des nombres flottants en double précision. La performance énergétique de l'ARM semble bonne mais elle a été calculée à partir d'une consommation estimée : il est donc difficile de directement comparer cette performance à celle du Xeon et de la Tesla. Enfin, il faut garder en tête que cette analyse est uniquement basée sur des valeurs théoriques et il est très probable que les mesures différent.

IV Le modèle *Roofline*

Le modèle *Roofline* [WWP09] proposé par S. Williams et al. permet de borner de manière réaliste les performances maximales atteignables pour un noyau de calcul quelconque. Le principe est relativement simple et la *Roofline* est construite comme suit :

$$Gflop/s\ atteignables = min \begin{cases} Crête\ Gflop/s, \\ Bande\ passante\ mesurée \times intensité\ opérationnelle, \end{cases}$$

avec l'intensité opérationnelle (I_o) une grandeur dépendant directement de l'intensité arithmétique (I) du noyau de calcul : $I_o = \frac{I}{taille\ des\ données\ en\ octet}$.

La différence entre puissance crête et la bande passante mémoire (RAM) devenant de plus en plus importante sur les architectures de calcul modernes, il y a de plus en plus de noyau de calcul limités par la bande passante mémoire. Le modèle *Roofline* permet de déterminer une borne supérieure qu'il ne sera pas possible de dépasser. Cette borne est par définition inférieure ou égale à la puissance crête. C'est un outil précieux : grâce à elle il est plus facile de savoir quand s'arrêter dans le processus d'optimisation d'un code. Attention cependant, ce modèle n'est pas valable si la taille des données du problème est inférieure à la taille des caches du processeur : dans ce cas le problème n'est plus limité par la bande passante de la RAM mais soit par la bande passante des caches soit par la performance crête du processeur.

1. *Intel Tubo Boost* : www.intel.com/technology/turboboost/

V Performances sur ARM

V.1 Performances théoriques atteignables

La Fig. 4.5 propose la *Roofline* pour le S4 Pro (*Krait 200*) utilisé pour les tests. L'intensité opérationnelle du *stencil* 5-point étudié est de $\frac{4.5}{4} = 1,125$ en simple précision et de $\frac{4.5}{8} = 0,5625$ en double précision. On remarque alors que *stencil* est limité par la bande passante et que la performance maximale atteignable est de 6,2 Gflop/s en simple précision.

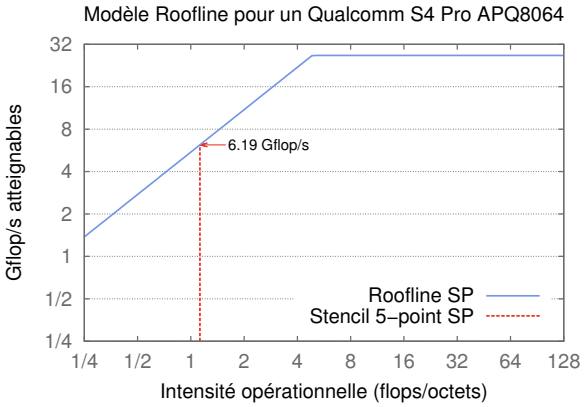

FIG. 4.5 – Modèle *Roofline* pour un *Qualcomm* S4 Pro APQ8064 (*Krait 200*)

V.2 Performances mesurées

Cette sous-section commente les performances obtenues sur ARM pour le code de l'équation de la chaleur en 2D. Les différentes versions du code ont été compilées avec le compilateur GNU (`g++`) dans sa version 4.8.2 avec les options de compilations suivantes : `-O3 -mfpu=neon -ftree-vectorize -funsafe-math-optimizations`.

A Performances en fonction de la taille du problème

La Fig. 4.6 montre l'évolution des performances du code (pour 4 cœurs @ 1,67 GHz) en fonction de la taille de la grille (grille carrée) et des différentes implémentations :

- Base : implémentation classique du code *stencil*,
- RB : implémentation du *Register Blocking*,
- DLT et DLTi : méthode *Dimension Lifted and Transposed*, DLTi pour l'utilisation de fonctions intrinsèques.

On remarque très nettement un effet de cache pour les grilles allant 32×32 à 512×512 cellules puis ensuite le code *stencil* devient limité par la bande passante mémoire. La méthode DLT est clairement avantageuse tant que la taille de la grille est inférieure à celle des caches : les performances sont plus que doublées avec l'utilisation des fonctions intrinsèques. Une fois sorti des caches l'implémentation classique donne de meilleures performances. Cela s'explique par des accès plus contigus en mémoire avec l'implémentation de base et donc une meilleure sollicitation de la bande passante. La méthode DLT

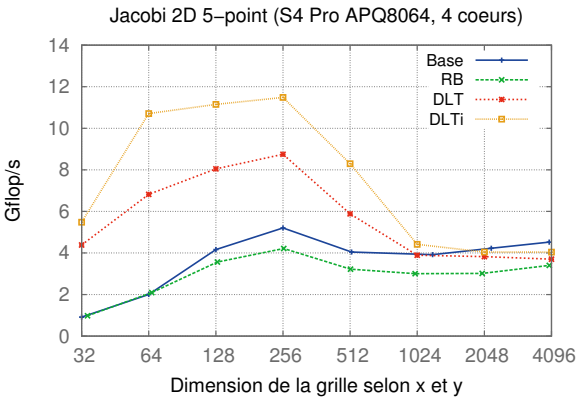

FIG. 4.6 – Gflop/s simple précision en fonction de la taille de la grille (*Krait 200*)

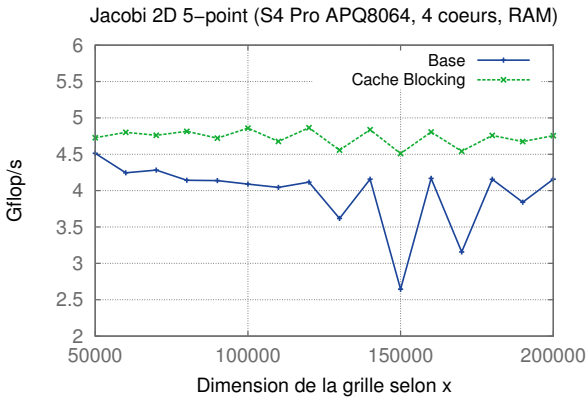

FIG. 4.7 – Gflop/s simple précision en fonction de la taille de la grille (*Krait 200*)

implique une réorganisation des données qui ne sont plus parfaitement contigus si l'on travaille sur des lignes plus petites que 128 bits (la taille des registres vectoriels NEON). Or, les accès à la mémoire DDR2 sont faits par lignes de 32 bits.

Dans les caches, le modèle *Roofline* ne s'applique pas et on obtient une performance maximale proche de 12 Gflop/s sur des nombres flottants simple précision. Cela correspond à 44 % de la performance crête de l'APQ8064. Le code *stencil* est contraint d'effectuer beaucoup de *load* et *store* dans le L1 car il y a peu de réutilisation des registres vectoriels : c'est pour cette raison qu'il n'est pas possible d'atteindre la performance crête du processeur sur ce type de code. Hors des caches le code atteint un maximum de 4,7 Gflop/s ce qui correspond à 85% de la performance atteignable selon le modèle *Roofline*.

La Fig. 4.7 illustre l'efficacité du *Cache Blocking* quand la dimension selon x dépasse la taille des caches (4 cœurs actifs @ 1,67 GHz). On remarque que la version avec *Cache Blocking* maintient la performance aux alentours de 4,7 Gflop/s alors que la version basique ne dépasse pas les 4,2 Gflop/s. Le gain est de l'ordre de 10%.

Enfin, le *Register Blocking* ne donne pas les résultats attendus : aujourd'hui, le compilateur est capable de correctement optimiser l'utilisation des registres sans que le code soit explicite. Les performances sont légèrement en dessous du code basique : cela est sûrement du au surcoût lié à l'utilisation d'un nombre plus élevé de boucles.

B Scalabilité faible dans les caches

La Fig. 4.8 montre l'évolution des performances du code *stencil* en fonction du nombre de cœurs pour une taille de grille qui tient dans les caches et qui augmente proportionnellement au nombre de cœurs (implémentation DLTi). La Fig. 4.9 se focalise sur l'efficacité lors de l'ajout de nouveaux cœurs en considérant qu'une efficacité de 100% équivaut à la performance séquentielle multipliée par le nombre de cœurs.

Pour chacune de ces deux figures (Fig. 4.8 et 4.9) nous avons aussi regardé l'évolution de la performance et de l'efficacité en fonction de la fréquence (par paliers de 430 MHz). Dans la Fig. 4.8 on remarque qu'il y a bien une évolution des performances en fonction de la fréquence mais le gain apporté à 1,67 GHz est plus faible que pour les autres fréquences. L'ARM utilisé pour les tests fonctionnait sur un jeune portage d'Ubuntu

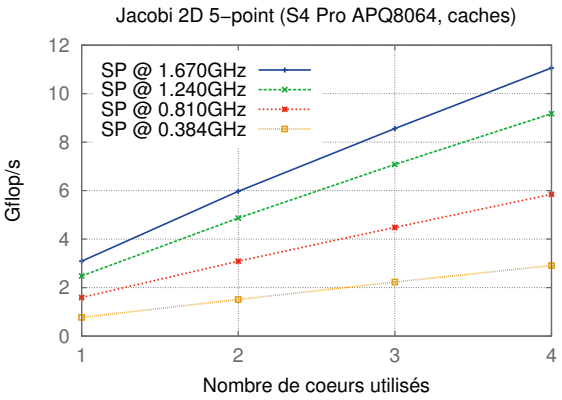

FIG. 4.8 – Gflop/s simple précision en fonction du nombre de cœurs pour une grille de taille inférieure aux caches (*Krait 200*)

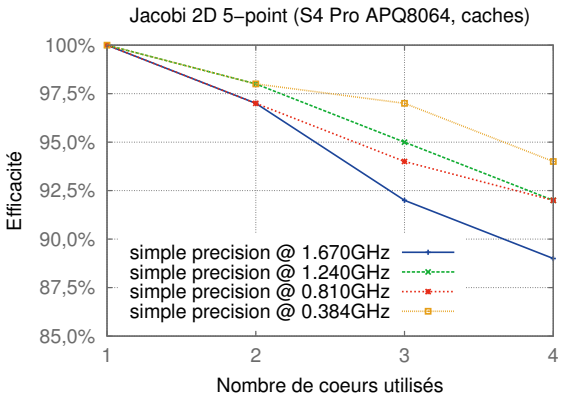

FIG. 4.9 – Efficacité en fonction du nombre de cœurs pour une grille de taille inférieure aux caches (*Krait 200*)

14.04 LTS (Linaro 14.06 [2]) et il y avait une politique énergétique très conservative : il y a de forte chance pour que la fréquence du processeur est été diminuée dynamiquement par le système.

La Fig. 4.9 montre un bon niveau d'efficacité quelle que soit la fréquence (presque toujours au dessus de 90%). Cependant on remarque que quand la fréquence augmente, l'efficacité diminue légèrement et si l'on ajoute à cela que la consommation énergétique augmente suivant le carré de la fréquence, il est plus intéressant (du point de vue énergétique) de calculer le *stencil* à basse fréquence. L'énergie consommée pour arriver à la solution sera moindre à basse fréquence (0,384 GHz) mais le temps de restitution des calculs sera plus élevé qu'à haute fréquence (1,670 GHz).

2. Linaro 14.06 : `http://releases.linaro.org/14.06/ubuntu/ifc6410`

VI Performances sur CPU (x86)

VI.1 Performances théoriques atteignables

La Fig. 4.10 propose la *Roofline* pour le Xeon E5-2650 (*Sandy Bridge*) utilisé pour les tests. On remarque que *stencil* est limité par la bande passante (comme pour l'ARM) et que la performance maximale atteignable est de 39,4 Gflop/s en simple précision et 19,7 Gflop/s en double précision.

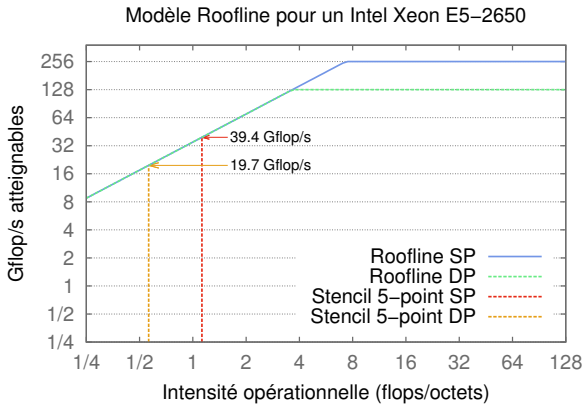

FIG. 4.10 – Modèle *Roofline* pour un *Intel Xeon* E5-2650 (*Sandy Bridge*)

VI.2 Performances mesurées

Cette sous-section commente les performances obtenues sur x86 pour le code de l'équation de la chaleur en 2D. Les différentes versions du code ont été compilées avec le compilateur *Intel* (`icpc`) dans sa version 14.0.3 avec les options de compilations suivantes : `-O3 -xHost` (et `-opt-streaming-stores always` pour les versions *Non-temporal stores*).

A Performances en fonction de la taille du problème

La Fig. 4.11 montre l'évolution des performances du code (pour 8 cœurs @ 2,00 GHz) en fonction de la taille de la grille (grille carrée) et des différentes implémentations (Base NTS correspond à l'implémentation classique compilé avec les *Non-temporal stores*).

La Fig. 4.12 compare les versions avec et sans *Cache Blocking* quand la dimension de la grille selon x dépasse la taille des caches.

On remarque qu'avec le compilateur *Intel* il n'y a pas de gain avec la version DLT du code. Depuis les instructions AVX, le coût des chargements non-alignés est bien moindre qu'avec les anciennes instructions SSE. La vectorisation automatique du code basique permet d'atteindre de bonnes performances dans les caches avec un maximum de 100 Gflop/s en simple précision (soit 40% de la performance crête). Hors des caches le code devient limité par la bande passante mémoire. Sans l'utilisation des *Non-temporal stores* le *stencil* atteint 26,7 Gflop/s alors qu'avec l'utilisation des *Non-temporal stores* il atteint 34,0 Gflop/s (soit environ 30% de mieux) ce qui représente 85% de la performance crête atteignable. Il est bon de noter que la désactivation de l'utilisation des caches en écriture

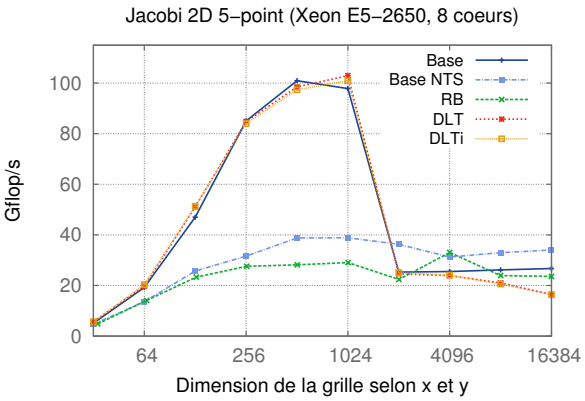

FIG. 4.11 – Gflop/s simple précision en fonction de la taille de la grille (*Sandy Bridge*)

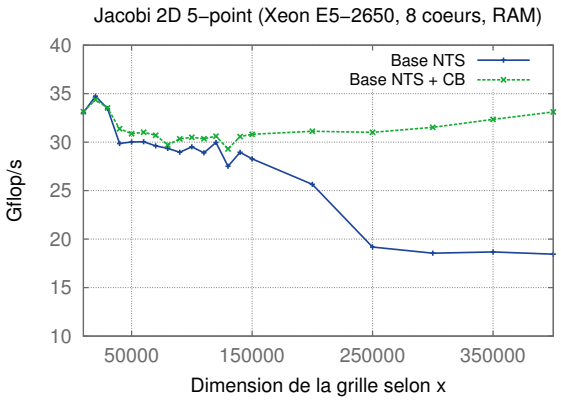

FIG. 4.12 – Gflop/s simple précision en fonction de la taille de la grille (*Sandy Bridge*)

empêche d'atteindre de bonnes performances pour de petites grilles car les accès mémoires en écriture sont toujours faits directement dans la RAM (voir courbe "Base NTS"). De plus, comme sur ARM, le *Register Blocking* ne donne pas les résultats attendus : le compilateur d'*Intel* est parfaitement capable de faire ce type d'optimisation tout seul.

L'étude de l'efficacité de *Cache Blocking* montre un gain de 50% quand la dimension de la grille selon x devient très élevée (Fig. 4.12). Cet écart de performance est très important et s'explique par la différence entre la bande passante mémoire et la capacité de calcul du processeur.

B Scalabilité faible dans les caches

Les Fig. 4.13 et Fig. 4.14 montrent respectivement l'évolution de la performance en simple et en double précision en fonction du nombre de cœurs pour une taille de problème qui augmente proportionnellement au nombre de cœurs (implémentation DLTi).

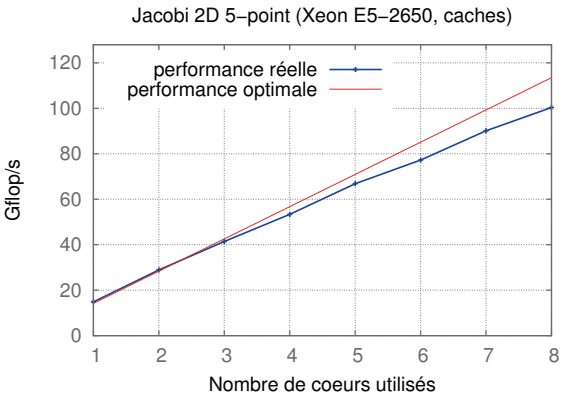

FIG. 4.13 – Gflop/s simple précision en fonction du nombre de cœurs pour une grille de taille inférieure aux caches (*Sandy Bridge*)

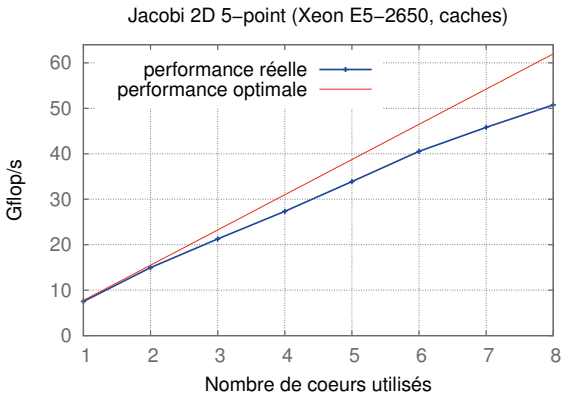

FIG. 4.14 – Gflop/s double précision en fonction du nombre de cœurs pour une grille de taille inférieure aux caches (*Sandy Bridge*)

La performance mesurée est relativement proche de la performance optimale et l'efficacité (cf. Fig. 4.15) est bonne jusqu'à 8 cœurs (elle ne descend jamais en dessous de 80%). Il n'y a pas de différence de comportement entre la version en simple et en double précision. L'étude de la scalabilité faible (calcul de l'efficacité) donne souvent de bons résultats puisque l'on reste dans des cas idéaux quel que soit le nombre de cœurs utilisés. Cependant on remarque que l'efficacité n'est pas parfaite et cela permet de mesurer le coût directement lié à l'utilisation d'une architecture parallèle. Plus il y a de cœurs plus le mécanisme de cohérence de cache devient cher et plus il y a de synchronisations entre les *threads*.

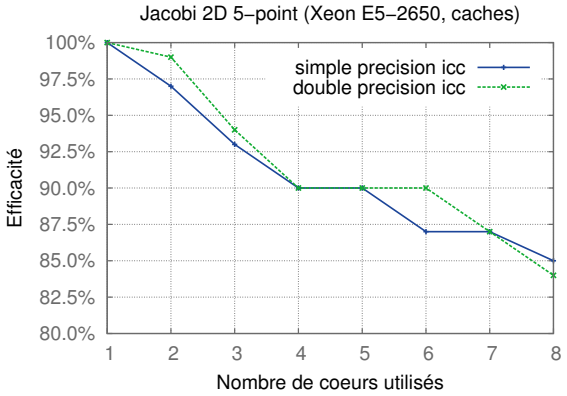

FIG. 4.15 – Scalabilité faible pour une grille de taille inférieure aux caches (*Sandy Bridge*)

VII Performances sur accélérateur

VII.1 Performances théoriques atteignables

La Fig. 4.16 propose la *Roofline* de la Tesla K20c (*Kepler*) utilisée pour les tests. Comme sur CPU, on remarque que *stencil* est limité par la bande passante et que la performance maximale atteignable est de 163,1 Gflop/s en simple précision et 81,6 Gflop/s en double précision.

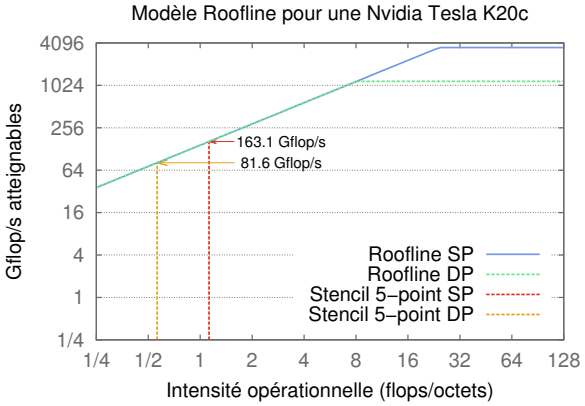

FIG. 4.16 – Modèle *Roofline* pour une *Nvidia Tesla K20c* (*Kepler*)

VII.2 Performances mesurées

Cette sous-section commente les performances obtenues sur GPU pour le *stencil* 2D. Les différentes versions du code ont été compilées avec le compilateur *Nvidia* (`nvcc`, CUDA *toolkit* 5.5) et avec l'option de compilation `-O3`.

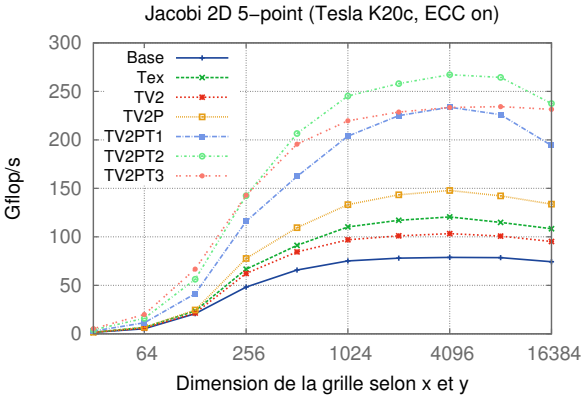

FIG. 4.17 – Gflop/s simple precision en fonction de la taille de la grille (*Kepler*)

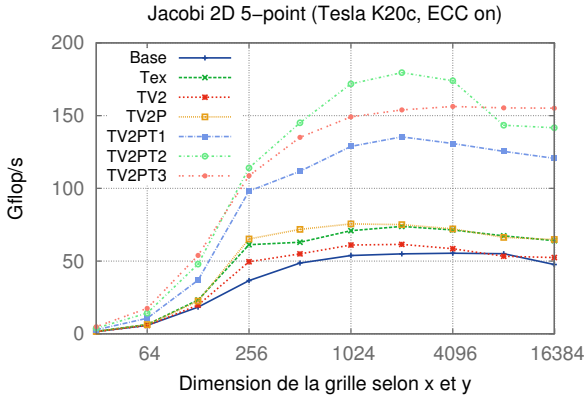

FIG. 4.18 – Gflop/s double precision en fonction de la taille de la grille (*Kepler*)

Les Fig. 4.17 et Fig. 4.18 montrent l'évolution des performances en simple et en double précision suivant la taille de la grille. Chaque courbe correspond à une implémentation différente (voir table. 4.2).

Base	Implémentation basique.
Tex	Utilisation de la mémoire texture (cache en lecture seule).
TV2	Optimisation précédente + chargements par vecteur de taille 2.
TV2P	Optimisations précédentes + *padding*.
TV2PT1	Optimisations précédentes + *Temporal Blocking* de taille 1.
TV2PT2	Optimisations précédentes + *Temporal Blocking* de taille 2.
TV2PT3	Optimisations précédentes + *Temporal Blocking* de taille 3.

TAB. 4.2 – Détail des optimisations pour les courbes Fig. 4.17 et Fig. 4.18

L'utilisation de la mémoire texture (cache en lecture seule) permet une bonne amélioration des performances que ce soit en simple ou en double précision. Par contre on remarque que les chargements par vecteur de taille 2 ne sont pas directement bénéfiques et il faut utiliser le *padding* pour réduire le nombre de branchements. En simple précision, la version avec *padding* est bien meilleure que la version qui n'utilise que les textures mais cela n'est pas vrai en double précision. Le chargement de nombres en double précision permet de saturer plus rapidement la bande passante mémoire puisque qu'ils occupent deux fois plus de place. La performance atteinte avec la version "TV2P" est de 150 Gflop/s en simple précision et de 75 Gflop/s en double précision soit environ 90% de la performance atteignable selon le modèle *Roofline*. Ces résultats sont proches des autres travaux effectuées sur GPU ([MA14], [HPS12]).

L'implémentation du *Temporal Blocking* permet de diminuer le nombre de lectures par itération dans la mémoire globale et de dépasser les performances précédentes. Le modèle *Roofline* présenté en sous-section VII.1 ne s'applique plus puisque le nombre d'accès à la mémoire par itération n'est plus le même. Le gain est assez impressionnant puisque avec un bloc de taille 2 ("TV2PT2") les performances atteintes sont quasiment doublées (267 Gflop/s en simple précision et 179 Gflop/s en double précision). Le *Temporal Blocking* est très efficace pour ce *stencil* 2D car l'intensité arithmétique est faible. Enfin, il est bon de noter que quand la taille des blocs devient trop grande (cf. "TV2PT3") le *Temporal Blocking* devient moins efficace : il faut donc trouver la bonne taille de bloc en fonction du problème et du matériel (ici un halo de taille 2).

Sur ce problème, la Tesla K20c est environ 8 fois plus rapide que le Xeon E5-2650 (si la taille de la grille dépasse celle des caches du processeur). Ce résultat est possible grâce à l'architecture des GPU qui permet de manipuler de la mémoire *on-chip* (la mémoire partagée). Sur CPU il n'est techniquement pas possible d'appliquer le même modèle de *Temporal Blocking* : les caches sont un mécanisme automatique et il l'utilisateur n'en a pas le contrôle direct.

VIII Comparaison énergétique

Cette section s'attache à la performance par watt suivant les différentes architectures étudiées (cf. Tab. 4.1). Il n'a pas été possible de relever la consommation énergétique réelle lors des calculs sur le *stencil* : nous considérons que la consommation est maximale (TDP max.). Cette assertion est sûrement fausse mais elle permet une comparaison entre les différentes architectures.

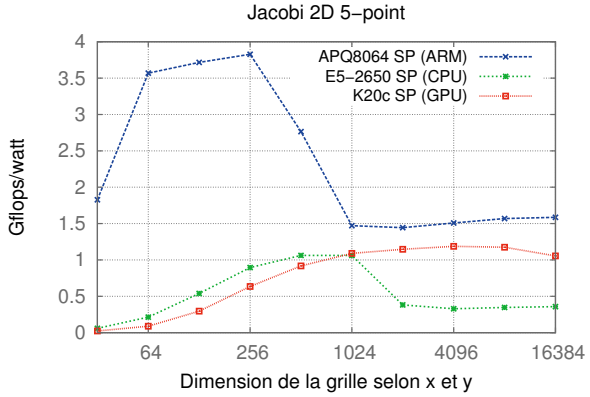

FIG. 4.19 – Performance simple précision par watt pour le Snapdragon S4 APQ8064 (*Krait 200*), le Xeon E5-2650 (*Sandy Bridge*) et la Tesla K20c (*Kepler*)

La Fig. 4.19 montre la performance en calcul simple précision par watt pour les différentes architectures de calcul. Pour chaque taille de grille, la meilleure performance en Gflop/s a été choisi toutes optimisations confondues. Sur l'ARM et le CPU on remarque une très nette chute de l'efficacité énergétique après une certaine dimension de la grille (256×256 pour l'ARM et 1024×1024 pour le CPU) : cela est dû à l'effet de cache. Sur GPU ce phénomène n'existe pas car il n'y a pas de cache (ou tout du moins ils sont très différents de ceux présents sur CPU). L'efficacité de l'ARM dépasse largement celle du CPU x86 et du GPU quand la grille tient dans les caches. Par contre, quand la taille du problème sort des caches alors l'efficacité se rapproche de celle sur GPU. Rappelons tout de même que le calcul de la performance par watt sur l'ARM est basé sur une consommation mesurée (cf. section III) et non sur la valeur donnée par le constructeur : il y a fort à parier que les consommations réelles du Xeon et de la Tesla soient très en dessous du TDP max. De plus, les unités vectorielles NEON ne sont pas compatibles avec la norme IEEE 754 et l'approximation dans les calculs est plus grande sur l'ARM que sur le Xeon et la Tesla.

L'ARM (APQ8064) mis à part, pour une taille de problème tenant dans les caches l'utilisation d'un CPU traditionnel (Xeon E5-2650) est légèrement plus avantageuse. Par contre quand l'utilisation de la mémoire globale (RAM) devient intensive alors l'efficacité énergétique sur GPU (Tesla K20c) est largement meilleure : elle est presque triplée par rapport au CPU.

Chapitre 5

Conclusion

I Bilan

Au travers de ce mémoire, nous avons étudié une méthode de calcul bien connue dans la simulation numérique : les algorithmes de type *stencil*. Nous avons essayé de l'adapter le mieux possible à différentes architectures modernes (CPU x86, CPU ARM et GPU) en utilisant plusieurs techniques (*Register Blocking*, *Cache Blocking*, *Dimension Lifted and Transposed*, *Temporal Blocking*, etc.) dans une optique de performance et d'efficacité énergétique.

Le code `stenCINES` est l'implémentation de cette méthode de calcul. Il a été conçu de manière à rester générique grâce aux possibilités du `C++` (utilisation des *templates*). `stenCINES` permet de calculer des *stencils* de tout ordres en 1D, 2D et en 3D pour des grilles structurées simple champ et multi champs. Deux types de conditions aux limites ont été considérées : de Dirichlet et périodiques.

Les différentes expérimentations ont été menées sur la discrétisation de l'équation de la chaleur en 2D. Cela se traduit pas un code *stencil* 5-point du premier ordre très commun aussi appelé Jacobi 2D 5-point. Les résultats montrent une bonne exploitation des différentes machines de calcul avec plus de 85% de la performance atteignable réellement atteinte (selon le modèle *Roofline*). Ces performances reflètent bien les différents travaux effectués par la communauté scientifique autour des Jacobi 5-point et confirment que `stenCINES` est un code de calcul efficace. De plus, la comparaison énergétique entre x86, ARM et GPU montre que les processeurs traditionnels (x86) sont de plus en plus remis en question pour la course vers l'exascale. Les GPUs et les ARMs sont de sérieux candidats pour dépasser les limites actuelles. Les résultats sur ARMs sont prometteurs mais ces processeurs ne sont pas encore complètement mur pour un véritable environnement de calcul : ils sont encore massivement en 32 bit (les premiers modèles 64 bit commencent à apparaître) et les instructions vectorielles ne garantissent pas une bonne précision dans les calculs. Du point de vue de l'architecture, les GPUs semblent être parfaitement adaptés aux codes massivement parallèles mais le seul point noir réside dans le modèle de programmation qui est différent du modèle traditionnel. Il est difficile de dire si la communauté scientifique l'adoptera complètement mais OpenACC essaie de répondre à cette problématique en proposant un langage par directive alternatif à CUDA et OpenCL. Ce nouveau modèle semble efficace sur des codes limités par la bande passante mémoire mais, aujourd'hui, il ne permet pas de rivaliser avec CUDA pour les autres types de codes.

Intel travaille actuellement sur un type d'accélérateur alternatif aux GPUs avec une architecture massivement parallèle à base de processeurs x86 : le Xeon Phi. Les premiers modèles (*Knights Corner*) commercialisés ont été beaucoup remis en question quant à leur efficacité réelle et il est difficile de dire si ils réussiront à devenir une nouvelle référence. Dans tous les cas, le monde du HPC est actuellement en pleine phase de transition et l'étude des différentes architectures est très importante afin de déterminer intelligemment les composants qui équiperont les supercalculateurs de demain.

II Pistes d'évolution

Le code `stenCINES` ne permet pas l'exploitation complète d'un *cluster* de calcul, il serait intéressant d'implémenter une version multi-nœuds avec MPI : les communications réseaux entre les différents nœuds posent de nouvelles problématiques de parallélisme. Il serait ensuite facile de proposer un code hybride combinant l'utilisation des *threads* sur un nœud et des processus entre les différentes machines.

Les *stencils* d'ordre élevé (*High-Order*) sont de plus en plus courant dans la simulation numérique : ils apportent une meilleure précision. L'intensité arithmétique de ces *stencils* est plus élevée et il est potentiellement possible de mieux se rapprocher de la performance crête des machines de calcul. En d'autres termes, ce type de code peut être doublement intéressant puisqu'il permet de mieux approximer les phénomènes pour un surcoût de temps qui n'est pas forcément proportionnel à l'ordre. `stenCINES` est nativement capable de calculer ce type de *stencils* et une étude approfondie est à effectuer.

Les accélérateurs de type Xeon Phi n'ont pas été traités et un portage du code est prévu afin des les comparer aux GPUs. Il en va de même pour la nouvelle architecture *Haswell* d'*Intel* (CPU). Cette dernière a été présentée dans le mémoire mais l'étude précise du comportement du code n'a pas été faite. Les premiers résultats ont quand même montré qu'il était plus difficile de se rapprocher de la puissance crête à cause de la spécialisation due aux opérations FMA.

Enfin, l'équation de la chaleur est un phénomène relativement simpliste puisqu'il n'y a qu'un champ à prendre en compte. L'implémentation d'un code de CFD (*Computational Fluid Dynamics*) est envisagée (équations d'Euler). Ce type de code fait intervenir plusieurs champs dont la densité, la vitesse et l'énergie et il est possible de calculer une multitude de phénomènes liés au comportement du fluide dans l'espace et dans le temps (tourbillon, canal, etc).

Bibliographie

[Chr11] Matthias-Michael Christen. *Generating and Auto-Tuning Parallel Stencil Codes*. PhD thesis, Université de Bâle, Suisse, 2011.

[CU10] José M. Cecilia, José M. García 0001, and Manuel Ujaldon. Cuda 2d stencil computations for the jacobi method. In Kristján Jónasson, editor, *PARA (1)*, volume 7133 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2010.

[DiNW+09] Hikmet Dursun, Ken ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. In-core optimization of high-order stencil computations. In *In PDPTA, pages 533–538, Las Vegas, NV, July13–16*, 2009.

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[HSP+11] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th International Conference on Compiler Construction : Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[JB12] Julien Jaeger and Denis Barthou. Automatic efficient data layout for multi-threaded stencil codes on cpus and gpus. *20th Annual International Conference on High Performance Computing*, 0 :1–10, 2012.

[Lui13] Justin Luitjens. Cuda pro tip : Increase performance with vectorized memory access. http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, December 2013.

[MA14] Naoya Maruyama and Takayuki Aoki. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In Armin Größlinger and Harald Köstler, editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 89–95, Vienna, Austria, January 2014.

[MK10] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1) :389–402, 2010.

[NVI14] NVIDIA. CUDA C Programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, february 2014.

[Ris06] Laurent Risser. Différences finies pour la résolution numérique des équations de la mécanique des fluides. http://laurent.risser.free.fr/THESE/coursM11_2006.pdf, february 2006.

[Shi13] Anand Lal Shimpi. The arm vs x86 wars have begun : In-depth power analysis of atom, krait and cortex a15. http://www.anandtech.com/show/6536/arm-vs-x86-the-real-showdown, January 2013.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2013.

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

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline : An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4) :65–76, April 2009.

Glossaire

CPU : *Central Process Unit* ou unité de calcul centrale.

CISC : *Complex Instruction Set Computer* ou ordinateur à jeu d'instructions complexe.

***Cluster* de calcul** : un supercalculateur.

Exascale : puissance d'au moins un exa flop par seconde (10^{18} flop/s).

Flop : *Floating point operation* ou opération sur nombre flottant.

Flop/s : nombre d'opérations sur des nombres flottants par seconde.

Fonction intrinsèque : fonction qui correspond à une instruction assembleur directement compréhensible par le processeur.

FMA : *Fused Multiply and Add* ou multiplication et addition fusionnées.

GPU : *Graphics Process Unit* ou unité de calcul graphique.

Granularité : définit le niveau de parallélisme utilisé.

HPC : *High Performance Computing* ou calcul haute performance.

Instruction : opération complète d'un processeur.

Maillage : discrétisation spatiale d'un milieu continu.

MPMD : *Multiple Programs Multiple Data* ou de multiples programmes pour de multiples données.

NUMA : *Non Uniform Memory Access* ou accès non uniformes à la mémoire.

Performance crête : performance maximale théorique d'une machine de calcul.

Pipeline : chaîne de traitement.

RAM : *Random Access Memory* ou mémoire à accès non séquentiel.

RISC : *Reduced Instruction Set Computer* ou ordinateur à jeu d'instructions réduit.

Scalabilité : signifie le passage à l'échelle dans le cadre de l'utilisation de plusieurs ressources.

SIMD : *Single Instruction Multiple Data* ou une instruction unique pour de multiples données.

SIMT : *Single Instruction Multiple Threads* ou une instruction unique pour de multiples *threads*.

SMX : *Streaming Multiprocessor neXt generation* ou multiprocesseur de flux de nouvelle génération.

SoC : *System on Chip* ou système dans la puce.

Supercalculateurs : ordinateur dédié au calcul construit à partir de plusieurs ordinateurs traditionnels.

Thread : fil d'exécution d'un programme informatique.

Vectorisation : calcul sur des vecteurs de données.