Vers des noyaux de calcul intensif pérennes

Wilfried Kirschenmann

UNIVERSITÉ DE LORRAINE

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

Vers des noyaux de calcul intensif pérennes

THÈSE

présentée et soutenue publiquement le 17 octobre 2012

pour l'obtention du

Doctorat de l'université de Lorraine

(spécialité informatique)

par

Wilfried KIRSCHENMANN

Composition du jury

Président :	Jocelyn SEROT	Professeur, Université Blaise Pascal, Clermont-Ferrand
Rapporteurs :	Denis BARTHOU	Professeur, Université de Bordeaux
	François BODIN	Professeur, IRISA, Rennes
Examinateurs :	Pascal BOUVRY	Professeur, Université du Luxembourg
	André SCHAFF	Professeur, Université de Lorraine, Nancy
	Stéphane VIALLE	Professeur, Supélec campus de Metz (directeur de thèse)
	Laurent PLAGNE	Docteur en physique, chercheur à EDF R&D, Clamart (co-encadrant de la thèse)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Supélec campus de Metz

EDF R&D

Loria Laboratoire lorrain de recherche en informatique et ses applications

Supélec

EDF R&D

Ce n'est pas tant l'aide de nos amis qui nous aide que notre confiance dans cette aide.

Epicure (-342 – -270)
Philosophe Grec

Remerciements

Aux randonneurs qui, croisant mon chemin, ont écarté les branches qui sur ma route...
Un simple sourire dans le métro – de compassion, de joie ou plus simplement de bonne humeur –, un encouragement, une discussion animée, un repas partagé ou une présence dans les moments de doute : autant d'empreintes imperceptibles traçant les sentiers de cette thèse.

Aux guides qui m'ont montré des voies invisibles entre les difficulté et m'ont indiqué le nord lors de mes égarements...
Martine et Raymond, pour tout ce que vous avez fait pour moi au cours de ces douze dernières années, je vous remercie ! Votre aide n'aurait probablement pas abouti au même résultat sans la participation active d'Anne-Cécile et Laurent. Cousin, Cousine merci pour tous les moments que nous avons partagés et pour votre soutien !
Vous avez contribué à façonner le chemin que j'ai emprunté et, si je ne vous dois pas tout, je vous dois beaucoup.

Aux éclaireurs qui m'ont aidé à dessiner mon chemin sur une carte et à lire celles des autres...
Jean-Philippe, nos nombreuses nuits entre gris clair et gris foncé, révisions ponctuées de débats de société, ont été autant d'occasion pour moi de confronter ma vision du monde et de comprendre ses fondements culturels. C'est également par ton intermédiaire que j'ai été amené à côtoyer Alexandre, Élodie, Jean, Kamel, Ludovic, Nicolas et Romain. AG1D, un nom de code, un cercle ouvert d'échanges et de rencontres, un lieu de traditions, d'engagements... D'amitié. Alexandre, tu m'as montré des chemins de travers et m'a amené à découvrir d'autres mondes.
Continuons à faire tomber les tours de guet de nos citadelles et à bâtir de nouveaux mondes.

Aux voyageurs éclairés qui m'ont montré des passerelles vers des univers différents...
Ariane, à cause de toi, je me disperse ! Aidée d'Alexandre, d'Alexis-Michel, d'Antoine, d'Augustin, de Bruno, de Christelle, de Daria, de Didier, de François, de Françoise, de Jean-Claude, de Joséphine, de Léa, de Patrice, de Pierre et de Philomène, tu m'a amené à m'intéresser à presque tous les sujets nécessaires pour se forger un regard sur notre monde.

Aux compagnons qui m'ont aidé à choisir une destination, nouveau point de départ...
Alia, Andréas, Angélique, Antoine, Bruno, Christophe, Christian, Daniel, David, Éléonore, Éric, Évelyne, Fannie, Flora, Frank, Franck, François, Gérald, Guillaume, Guy, Hugues, Ivan, Jean-Philippe, Laurent, Marc-André, Marie-Agnès, Mark, Matthieu, Maxime, Olivier, Sethy, Solène, Stéphane, Tanguy, Thierry et Véronique. Grâce à vous, le mot *collègue* ne signifie pas seulement *co-travailleur* : il comprend des notions d'amitié. Nos discussions sur l'égalité entre les vrais gens – hommes ou femmes – reprendront, je n'en doute pas, autour d'un verre en montagne.

Aux yeux affûtés qui, repassant derrière moi, m'ont aidé à bien indiquer mon chemin...
Laurent et Stéphane vous avez largement contribué à rendre ce document compréhensible. Secondés d'Alexandre, d'Alexis-Michel, de Léa, de Martine et de Raymond, vous m'avez même convaincu de le rendre lisible !

Aux oreilles attentives qui ont accepté d'entendre mon histoire, parfois en venant de loin...
Denis et François, vous avez accepté d'aider André et Pascal à juger, sous la présidence de Jocelyn, la qualité scientifique de ce parcours. Vous avez été assisté en cela de Laurent et Stéphane qui m'ont conseillé tout au long de cette thèse. Si les témoins de cette journée étaient trop nombreux pour être listés sur cette page, je ne les oublie pas pour autant.

... J'adresse mes remerciements les plus sincères.

Table des matières

Liste des figures

Liste des tableaux

Les experts sont formels et unanimes : en règle générale, l'ordinateur le mieux adapté à vos besoins est commercialisé environ deux jours après que vous ayez acheté un autre modèle !

Dave Barry (1947 – présent)
Prix Pulitzer du commentaire en 1988

Chapitre 1

Introduction

L'utilisation de logiciels de calcul scientifique permet aux acteurs industriels comme EDF de simuler des expériences qui seraient trop coûteuses, trop longues, trop compliquées voire impossibles à mettre en œuvre dans la pratique [1]. Dans le cas des problèmes trop spécifiques pour être traités par les logiciels disponibles sur le marché, l'industrie doit développer ses propres outils de simulation.

La complexité algorithmique et la taille de ces logiciels peuvent entraver la maintenance et la mise à jour de leurs modèles physiques et mathématiques. Lors de la mise au point d'un nouveau logiciel de simulation, une grande attention est donc portée aux choix influençant sa maintenance future. Afin de réduire le coût de cette maintenance, les techniques de génie logiciel privilégient les stratégies minimisant le nombre de branches dans le code. Cela signifie généralement la mise au point de modules génériques dépendant le moins possible des spécificités des ordinateurs ou des modèles physiques et mathématiques. Ces modules peuvent alors être utilisés plusieurs fois dans une ou plusieurs applications, dans des contextes différents. Les modifications apportées à un module sont alors visibles dans toute les applications. Au contraire, la multiplication de branches spécialisées pour différents cas particuliers augmente le travail de maintenance et multiplie les possibilités d'introduction d'erreurs.

Parallèlement à cette problématique de maintenance, l'augmentation des besoins en nombre et en précision de ces simulations conduit à porter un grand intérêt à la réduction des temps d'exécution de ces logiciels. Cette optimisation des performances d'un logiciel implique une spécialisation du code afin de l'adapter aux différents problèmes à résoudre et aux différentes architectures matérielles utilisées. Afin de pouvoir optimiser les différents problèmes sur les différentes architectures matérielles ciblées par l'application, différentes branches d'exécution doivent alors être mises en œuvre.

Les choix d'architecture logicielle et d'implémentation traduisent donc un compromis entre maintenabilité et performances. Une implémentation favorisant une plus grande réutilisabilité du code améliorera la maintenabilité mais peut s'opposer à l'optimisation des performances. Inversement, une implémentation permettant la spécialisation du code favorisera l'obtention de bonnes performances au détriment de sa maintenabilité.

Dans le cadre de cette thèse, nous cherchons à identifier les approches permettant d'améliorer la compatibilité entre les activités de maintenance et d'optimisation. Ceci permettra de mieux spécialiser les applications scientifiques pour différentes générations d'architectures matérielles sans augmenter les coûts de maintenance.

Sommaire

1.1 Les codes scientifiques : un compromis entre maintenabilité et performances

EDF R&D développe des logiciels de simulation pour de nombreux domaines scientifiques comme la mécanique des structures (Code-Aster [2]), la mécanique des fluides (Code-saturne [3, 4]), la thermique (Syrthes [5]) ou encore la neutronique (COCCINELLE [6] et COCAGNE [7]). Compte tenu de la complexité des problèmes à résoudre, les temps de résolution sont une très forte contrainte sur ces codes comme en témoigne le nombre de travaux effectués afin de réduire le temps de calcul des simulations [1, 8, 9, 10, 11, 12, 13, 14, 15]. Cette contrainte conduit à vouloir spécialiser fortement ces codes pour les machines sur lesquelles ils doivent s'exécuter.

La chaîne de calcul COCCINELLE est dédiée à la simulation du fonctionnement des cœurs de centrale nucléaire. Elle est utilisée aussi bien pour optimiser des paramètres d'exploitation que pour répondre aux exigences croissantes de l'Autorité de Sûreté Nucléaire (ASN). Pour l'exploitant EDF, COCCINELLE permet ainsi de déterminer des paramètres de pilotage, comme la concentration en bore nécessaire à l'équilibre de la réaction en chaîne, ou des paramètres d'optimisations comme la « longueur naturelle de campagne » qui correspond à la durée d'exploitation du combustible nucléaire avant qu'un rechargement du cœur ne soit nécessaire. COCCINELLE permet également à EDF de déterminer des éléments nécessaires aux dossiers de sûreté. Par exemple, lors d'un rechargement de combustible nucléaire, un dossier de sûreté doit être envoyé à l'ASN en vue d'obtenir l'autorisation de redémarrage de la centrale. Ce dossier nécessite entre autres la détermination du « point chaud », c'est-à-dire l'emplacement du réacteur qui atteint la température la plus élevée lors du fonctionnement de la centrale. Le code COCCINELLE est utilisé afin de calculer les températures maximales pouvant être obtenues dans les différentes parties du réacteur et permet donc l'identification de ce point chaud.

Afin de mieux prendre en compte certains phénomènes physiques, la chaîne de calcul COCAGNE a été mise au point. COCAGNE repose sur des modèles physiques plus complets que la chaîne COCCINELLE et permet donc d'effectuer des simulations plus proches de la réalité. Dans cette perspective, COCAGNE sert de code de référence afin d'estimer les erreurs liées au modèle physique de COCCINELLE [16]. COCAGNE a aussi pour vocation, à terme, de remplacer COCCINELLE. L'augmentation de la complexité des modèles induit une augmentation importante du nombre de calculs et donc, des contraintes sur les temps d'exécution. Par exemple, la validation par COCAGNE d'un calcul 3D COCCINELLE nécessite de traiter beaucoup plus d'inconnues. Un problème 3D COCCINELLE comportant 1 million d'inconnues sera validé par un calcul 3D COCAGNE sur un problème dit « crayon par crayon », correspondant à une discrétisation plus fine et comportant 100 millions d'inconnues [17]. Afin de pouvoir obtenir les résultats dans des délais raisonnables, il est important de réduire les temps de calcul de cette chaîne.

Pour connaître la puissance thermique dégagée dans un cœur de réacteur, il faut calculer le flux neutronique angulaire :

$$\Phi\left(\overrightarrow{r}, v, \overrightarrow{\Omega}, t\right) \equiv n\left(\overrightarrow{r}, v, \overrightarrow{\Omega}, t\right) v$$

où la quantité $n\left(\overrightarrow{r}, v, \overrightarrow{\Omega}, t\right) \mathrm{d}^3 r \mathrm{d}v \, \mathrm{d}\Omega$ représente le nombre de neutrons au point $\overrightarrow{r}$ à $\mathrm{d}^3 r$ près, ayant la vitesse v à $\mathrm{d}v$ près, dans la direction Ω à $\mathrm{d}\Omega$ d'angle solide près, et au temps t. Le livre *La physique des réacteurs nucléaires* de Serge Marguet [18] introduit la notion de flux neutronique angulaire en détails et explique comment l'évolution du flux neutronique est régie par l'équation de Boltzmann. La résolution de cette équation représente une part importante du temps d'exécution de la chaîne COCAGNE [19, 20]. Trois solveurs s'appuyant sur différentes

méthodes d'approximation sont disponibles dans la chaîne de calcul COCAGNE afin de résoudre cette équation. À chacune de ces approximations correspond un compromis particulier entre le temps de résolution et la précision des résultats.

- La méthode SP_N [21, 22, 18] correspond à l'approximation la plus grossière disponible au sein de la chaîne de calcul COCAGNE. Elle permet l'obtention de résultats plus précis que la méthode de diffusion utilisée dans COCCINELLE tout en nécessitant relativement peu de temps de calcul. C'est la méthode utilisée pour valider les calculs de COCCINELLE [17, 16]. La consommation mémoire et les temps de résolution induits par cette méthode permettent d'effectuer des calculs directement sur station de travail. Il n'est donc pas envisagé d'utiliser les résultats des différents travaux de parallélisation de cette méthode sur machine à mémoire distribuée [23, 24].
- La méthode S_n [25, 22, 18] correspond à une approximation plus précise mais plus coûteuse en temps de calcul.
- La méthode des caractéristiques [26, 22, 18] est en cours de développement. Cette méthode permettra l'obtention de résultats plus fidèles aux modèles physiques et à la structure géométrique des réacteurs [26]. Si la consommation mémoire et les temps de résolution induits limitaient cette méthode aux seuls problèmes 2D [27, 28, 29], l'augmentation au cours des dernières années de la puissance de calcul des ordinateurs ainsi que de leur capacité mémoire permet depuis de concevoir des solveurs visant à résoudre des problèmes 3D [30, 31, 32].

Afin d'améliorer les performances de COCAGNE, le groupe Analyse et Modèles Numériques d'EDF R&D a réalisé différents travaux d'optimisations pour réduire la consommation mémoire et le temps d'exécution de ces solveurs [33, 34, 35, 23, 36, 37].

Si le temps d'exécution des codes de calcul industriels est une contrainte importante, ces derniers sont également soumis à un autre impératif : la longueur de leur cycle de vie. Le code de mécanique Code-Aster a par exemple fêté ses 20 ans en 2009 tandis que la première version de COCCINELLE a été lancée en 1983 [38]. Durant toute cette période, le code doit être maintenu. Ce travail de maintenance inclut trois types de travaux :

- la correction des erreurs,
- l'ajout de nouvelles fonctionnalités afin de répondre aux nouveaux besoins,
- et l'adaptation du code aux nouvelles architectures matérielles.

Par exemple, en 2010, une nouvelle version de COCCINELLE est sortie afin de permettre de faire des calculs de cœur pour l'EPR [39].

Face à des durées de vie aussi longues, l'évolution des architectures matérielles des ordinateurs est extrêmement rapide. Deux ans s'écoulent en moyenne entre deux évolutions technologiques majeures des micro-processeurs centraux (*Central Processing Unit* ou CPU) comme l'illustre le *tableau* 1.1 représentant les évolutions des CPU X86 [1]. Ce tableau, qui ne concerne ni les autres familles de processeurs généralistes (PowerPC, ARM), ni les accélérateurs de calcul (Cell, GPUs), ne fait apparaître que les évolutions matérielles ayant un impact sur les méthodes d'optimisation des performances d'un code. Nous reviendrons plus longuement au chapitre 2 sur ces éléments ainsi que sur les différentes technologies présentées dans ce tableau. Nous verrons alors que, depuis 2002, l'augmentation du nombre d'opérations de calcul effectuées en une seconde s'appuie uniquement sur le parallélisme. En conséquence, un code non parallèle ne peut exploiter qu'une proportion toujours plus faible de la puissance de calcul disponible. Ainsi, sur notre machine de

1. Les processeurs de la famille X86 correspondent aux processeurs « INTEL et compatibles ». Les autres fabricants de processeurs X86 sont AMD et VIA.

Tableau 1.1 : *Évolutions technologiques dans les processeurs X86*

Date	Technologie	Degré de parallélisme				Conditions d'utilisation
		Apporté		Cumulé		
		Précision		Précision		
		simple	double	simple	double	
1997	3DNow !	2	-	2	-	Utilisation de fonctions très bas niveau Modification des structures de données
1999	SSE	4	-	4	-	Utilisation de fonctions très bas niveau Modification des structures de données
2001	SSE2	4	2	4	2	Utilisation de fonctions très bas niveau Modification des structures de données
2003	X86-64	-				Modification des tailles des types Recompilation des bibliothèques
2005	Multicœur[1]	n_c		$4n_c$	$2n_c$	Écriture de code multithreadé
2005	Disparition du FSB[2]	-				Temps d'accès mémoire non uniformes Modification des structures de données
2010	Cœurs GPU	Variable				Prise en compte de l'hétérogénéité
2011	AVX	8	4	$8n_c$	$4n_c$	Utilisation de fonctions très bas niveau Modification des structures de données

[1] n_c définit le nombre de cœurs d'un processeur multicœur. n_c est typiquement compris entre 2 et 16.

[2] En 2005, AMD remplace le Front Side Bus par sa technologie *Direct Connect* (DC)afin de supporter les accès mémoire non uniformes. INTEL attendra le lancement de l'architecture Nehalem en 2008 pour introduire un changement équivalent en introduisant la technologie *QuickPath Interconnect* (QPI). En 2011, QPI permet l'obtention de meilleurs débits de données que DC.

test [2] ${}_{2\times4}^{32}$Nehalem comportant 2 processeurs quadri-cœurs Xeon E5504 (2.0 GHz) embarquant la technologie SSE (cf. section 2.3), un code purement séquentiel (ni parallélisé, ni vectorisé) ne peut exploiter plus de $1/32^{\text{è}}$ de la puissance de calcul disponible.

L'optimisation d'un code de calcul pour une génération de processeurs ne peut donc garantir l'obtention d'un code capable d'exploiter efficacement les générations suivantes. Les coûts de mise en œuvre pour le développement, la maintenance et la validation étant très importants, il n'est pas envisageable de ré-optimiser les différents codes de calcul pour chaque génération d'ordinateurs. Un compromis doit donc être trouvé entre les coûts de maintenance d'un côté et l'optimisation des performances de l'autre.

Bien que cette recherche de compromis soit commune à tous les grands codes, nous nous intéresserons dans cette thèse aux solutions permettant de réduire les coûts d'optimisation des performances de la chaîne de calcul COCAGNE. Dans ce but, nous allons nous focaliser sur les solutions permettant d'améliorer l'adaptabilité du solveur SP_N aux nouvelles architectures. Ce solveur résout un problème d'algèbre linéaire et différentes approches existent aujourd'hui afin de mettre au point des applications d'algèbre linéaire pouvant exploiter les évolutions technologiques des processeurs dès leur apparition. Nous étudierons ces solutions dans la prochaine section.

2. ${}_{2\times4}^{32}$Nehalem se lit : machine disposant de 2 processeurs d'architecture Nehalem comportant 4 cœurs chacun. Chaque cœur comporte des unités SIMD à 4 voies. Le degré de parallélisme explicite disponible (multicœur × SIMD) sur cette machine est donc de 32.

1.2 Recensement des solutions pour le domaine de l'algèbre linéaire

L'utilisation de bibliothèques externes est le moyen traditionnel le plus efficace pour sous-traiter une partie du développement, de l'optimisation et de la maintenance de portions de codes ne correspondant pas au « cœur de métier » de l'application.

Nous allons dans cette section présenter les limites de l'externalisation de ces portions de codes vers des bibliothèques externes procédurales fondées sur des langages informatiques impératifs classiques (FORTRAN, C, ...). Deux types de limitations seront étudiées :

- le manque d'expressivité des opérations vectorielles puis,
- le manque d'expressivité des structures de données complexes.

Nous étudierons successivement ces deux limitations ainsi que les moyens disponibles pour les dépasser.

1.2.1 Limitation des bibliothèques procédurales

Dans un certain nombre de cas, mutualiser une partie du travail permet de faire des économies d'échelle. En informatique, la mutualisation de fonctions vers des bibliothèques permet de mettre en commun, entre plusieurs applications, les coûts de mise au point, de maintenance et d'optimisation de ces fonctions. Naturellement, pour que cela soit efficace, les fonctions de ces bibliothèques doivent être largement utilisées. Lorsque les fonctions d'une bibliothèque ont un champ d'utilisation suffisamment large, une interface finit généralement par devenir un standard *de facto*.

Au début des années 1970, les ordinateurs et les systèmes d'exploitation disponibles pour le calcul scientifique étaient très différents les uns des autres. Pour faciliter le développement d'applications scientifiques efficaces, chaque constructeur d'ordinateur fournissait une bibliothèque permettant d'effectuer les opérations les plus courantes. Chaque bibliothèque disposait alors de sa propre interface qui n'était pas compatible avec les bibliothèques des autres constructeurs. Pour pouvoir développer des codes portables entre les ordinateurs de différents constructeurs, la communauté scientifique a cherché à standardiser l'interface des principales opérations d'algèbre linéaire. Une étude menée sur différents codes de calcul scientifique a identifié les opérations mettant en œuvre des vecteurs comme les opérations les plus courantes [40]. Une proposition de standardisation de ces interfaces est faite en 1973 sous le nom de *Basic Linear Algebra Subprograms* (BLAS) [41, 42].

L'architecture des ordinateurs évoluant, la décomposition des algorithmes en opérations vectorielles n'est plus suffisante pour permettre l'obtention de bonnes performances sur toutes les architectures [43]. Afin de combler ce manque, Jack Dongarra, en 1985, propose à la communauté scientifique une extension des BLAS pour prendre en compte les produits matrice-vecteur [44]. Cette extension est finalisée et publiée en 1988 [45]. À cette occasion, l'interface BLAS originale est rebaptisée BLAS de « niveau 1 » (*Level 1 BLAS*) ou BLAS1 en référence à la complexité algorithmique des opérations proposées par cette interface : les opérations vectorielles sur des vecteurs de taille N ont une complexité en $\mathcal{O}(N^1)$. Pour des matrices denses de taille $N \times N$, les opérations matrice-vecteur introduites par Jack Dongarra ont une complexité en $\mathcal{O}(N^2)$ et sont donc dites de niveau 2 ou BLAS2. En 1987, Jack Dongarra propose d'étendre l'interface BLAS pour prendre en compte les opérations entre matrices comme les produits de matrices [46]. Ces opérations ont une complexité algorithmique en $\mathcal{O}(N^3)$ et forment donc les BLAS3 ; elles sont publiées en 1990 [47]. Depuis, un forum technique dédié aux BLAS s'est mis en place afin de faire évoluer l'interface en fonction des besoins de la communauté [48, 49].

Grâce à cette standardisation, les applications construites autour de l'interface BLAS deviennent portables sur les différentes plate-formes matérielles. Afin de gagner des parts de marché, les constructeurs d'ordinateurs et de processeurs cherchent à fournir à leurs clients les meilleures performances possibles. Pour y parvenir, ces derniers proposent donc leurs propres implémentations des BLAS. Actuellement, la majorité des constructeurs proposent leur implémentation des BLAS, comme par exemple :

- la *Math Kernel Library* (MKL) [50], fournie par INTEL ;
- la *AMD Core Math Library* (ACML) [51], fournie par AMD ;
- la *Engineering and Scientific Subroutine Library* (ESSL) [52], fournie par IBM.

Outre ces implémentations fournies par les constructeurs, différents travaux portés par la communauté du calcul intensif existent, comme par exemple les projets ouverts ATLAS et GOTO :

- ATLAS [53] (*Automatically Tuned Linear Algebra Software*) est une bibliothèque dont le processus d'installation effectue des mesures de performances afin de générer une implémentation optimisée pour l'ordinateur ;
- GOTO [54, 55] est une bibliothèque mise au point en assembleur par Kazushige Goto et qui est réputée pour être une des implémentations les plus performantes pour processeurs X86.

Des standards *de facto* existent également pour d'autres domaines scientifiques. Par exemple, les bibliothèques actuelles de transformées de Fourier implémentent la même interface que la version 3 de *Fastest Fourier Transform in the West* (FFTW) [56]. Depuis 2001, la *GNU Scientific Library* (GSL) [57] propose d'unifier les interfaces des bibliothèques de calcul utilisées dans de nombreux domaines scientifiques (algèbre linéaire, transformée de Fourier, interpolations, ...).

Une application dont la plus grande partie du temps d'exécution correspond à l'appel à de telles bibliothèques de calcul bénéficie automatiquement des performances apportées par les nouvelles architectures en changeant de version de bibliothèque. Le compromis entre maintenance et performance est alors trivial : le développeur de l'application scientifique prend en charge sa maintenance tandis que l'optimisation des performances est externalisée vers les bibliothèques scientifiques.

Cependant, il n'est pas toujours possible d'utiliser efficacement les bibliothèques disponibles. En effet, la majorité des bibliothèques orientées performances fournissent un jeu de fonctions fixe et fini. Ce jeu de fonctions peut s'avérer être trop rigide pour répondre efficacement aux besoins des applications clientes. Ainsi, dans son analyse de l'optimisation des performances sur le CRAY I, Jack Dongarra montre que l'utilisation des BLAS1 conduit à deux limitations :

- BLAS1 ne permet pas d'exprimer des opérations présentant deux niveaux de parallélisme ;
- BLAS1 ne définit pas de matrices.

Ceci illustre les deux catégories de limitations inhérentes à ce type de bibliothèque : un manque d'expressivité des opérations et un manque de moyen de description pour les objets mathématiques complexes. L'ajout des interfaces BLAS2 et BLAS3 n'a fait que repousser ces deux limitations. Ces limitations, qui existent toujours en 2011 comme nous allons le montrer, sont inhérentes à la nature des interfaces BLAS. Ces interfaces définissent des fonctions ou procédures, nous parlerons donc d'interfaces procédurales. Nous illustrerons dans la prochaine section l'impact des limitations des bibliothèques à interfaces procédurales sur les performances des codes.

1.2.2 Une plus grande expressivité des opérations d'algèbre linéaire

Afin de bien cerner le problème d'expressivité propre aux interfaces procédurales, nous allons étudier un exemple simple d'algèbre linéaire et la réponse apportée par les BLAS. Nous présenterons ensuite différentes approches favorisant l'expressivité et nous en étudierons les performances.

Soient W, X et Y trois vecteurs de taille n, et a, b et c trois scalaires. Nous souhaitons calculer la norme

$$norm = ||aW + bX + cY||. \tag{1.1}$$

Une implémentation informatique représentant les vecteurs W, X et Y par des tableaux de nombres flottants simple précision peut être [3] :

```
1 float *W, *X, *Y;
2 ...
3 float norm = 0;
4 for (int i=0; i<n; ++i) {
5   float tmp = a*W[i]+b*X[i]+c*Y[i];
6   norm+=tmp*tmp;
7 }
8 norm = sqrtf(norm);
```

Sur notre machine de tests ${}^{32}_{2\times4}$Nehalem (cf. section 2.3), cette implémentation séquentielle ne permet pas d'obtenir des performances optimales. En effet, afin d'exploiter efficacement le processeur, cette implémentation doit être parallélisée. Le degré et le type de parallélisme dépendent de l'architecture matérielle ciblée, comme nous le verrons au chapitre 2.

Nous souhaitons externaliser cette charge d'optimisation vers une bibliothèque externe. Afin de juger l'efficacité des différentes approches sur le plan des performances, nous allons comparer les performances obtenues par ces dernières avec les performances obtenues avec l'implémentation de référence ci-dessus ainsi qu'avec une implémentation optimisée en C.

La *figure* 1.1 représente le nombre d'opérations flottantes effectuées en une seconde en fonction de la taille des vecteurs. Les essais sont effectués sur notre machine de tests ${}^{32}_{2\times4}$Nehalem [4]. La courbe « C » montre les performances du code C présenté précédemment. Notons qu'elle est automatiquement vectorisée par le compilateur INTEL icc 12.0.2. Il est en outre possible de paralléliser cette implémentation à l'aide d'OpenMP[59]. Pour cela, il suffit d'ajouter la ligne

```
#pragma omp parallel for reduction(+:norm)
```

au dessus de la boucle `for` (ligne 4). La courbe « C optimisé » montre les performances alors obtenue [5].

Nous pouvons remarquer que pour les vecteurs de plus de trois millions d'éléments, le nombre d'opérations effectuées en une seconde est constant pour les deux implémentations qui atteignent respectivement 3,6 et seulement 6,3 GFlops. Les performances de ce type d'opérations sont limitées par la vitesse des accès à la mémoire RAM [60] : le processeur ne peut effectuer des opérations de calcul que si les données sont disponibles en registre. L'émergence des processeurs parallèles tend à augmenter la proportion de logiciels qui sont limités par la bande passante de la RAM [61]. En effet, un processeur capable d'effectuer deux opérations simultanément a

3. Cette implémentation a été choisi à titre d'exemple pour sa simplicité. Différentes stratégies permettant de minimiser la perte de précision des résultats existent, comme celle utilisée par netlib dans l'implémentation de référence [58].

4. Deux processeurs quadri-cœur INTEL Xeon E5504 (2.0 GHz) et 12 Go de RAM DDR3 1066.

5. Une vectorisation manuelle a été mise au point et permet d'obtenir les mêmes performances.

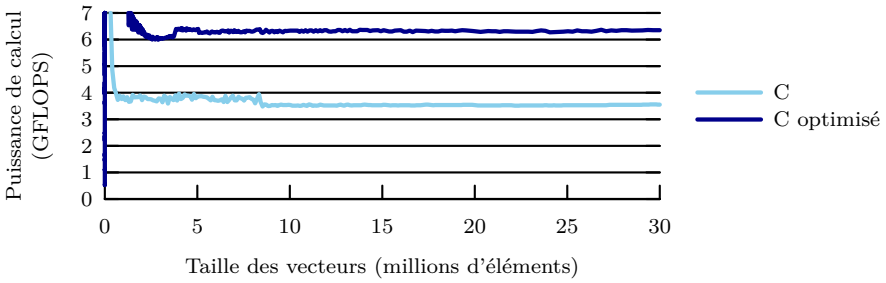


Figure 1.1 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem.*

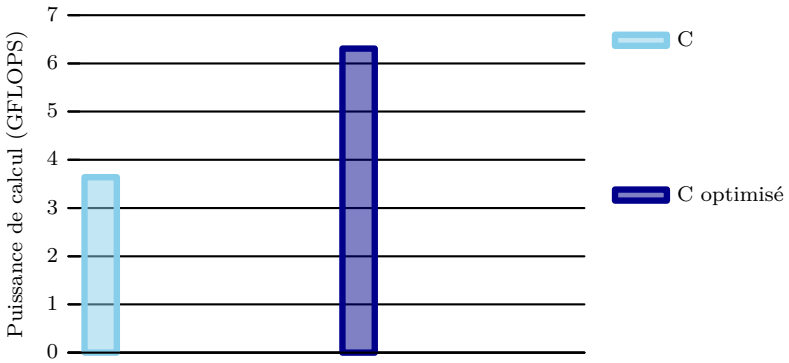


Figure 1.2 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

besoin d'un débit de données deux fois supérieur. Or, ces dernières années, la puissance de calcul des processeurs a cru plus rapidement que la bande passante. Lors de nos analyses des autres approches, nous présenterons les performances moyennes observées pour des tailles de vecteurs comprises entre 3 et 30 millions d'éléments en complétant le graphique de la *figure* 1.2.

1.2.2.1 Les interfaces procédurales : des interfaces de programmation peu expressives

Nous allons utiliser les BLAS pour effectuer l'opération 1.1. Les performances de notre implémentation dépendront alors des performances de l'implémentation BLAS utilisée, celle d'INTEL dans notre cas. Comme l'interface BLAS ne fournit pas de fonction permettant de calculer la norme d'une expression vectorielle quelconque, nous devons créer un vecteur temporaire contenant cette expression vectorielle puis en calculer la norme. Les différentes étapes de notre implémentation seront donc :

1. allouer de la mémoire pour un vecteur temporaire Z ;
2. calculer $Z = aW + bX + cY$;
3. calculer la norme de Z ;
4. désallouer Z.

La décomposition de notre problème en opérations prises en charge par les BLAS n'est pas achevée. En effet, la seconde opération, calculant Z, n'est également pas prise en charge. Cette opération doit, elle aussi, être décomposée. Les fonctions correspondant aux opérations élé-

mentaires nous permettant d'implémenter cette opération sont listées ci-dessous ainsi que leur descriptions mathématiques :

cblas_scopy(N, X, INCX, Y, INCY) :
$\forall i \in [\![0; \mathtt{N}-1]\!],\ \mathtt{Y}[i \times \mathtt{INCY}] \leftarrow \mathtt{X}[i \times \mathtt{INCX}]$;

cblas_sscal(N, ALPHA, X, INCX) :
$\forall i \in [\![0; \mathtt{N}-1]\!],\ \mathtt{X}[i \times \mathtt{INCX}] \leftarrow \mathtt{ALPHA} \times \mathtt{X}[i \times \mathtt{INCX}]$;

cblas_saxpy(N, ALPHA, X, INCX, Y, INCY) :
$\forall i \in [\![0; \mathtt{N}-1]\!],\ \mathtt{Y}[i \times \mathtt{INCY}] \leftarrow \mathtt{ALPHA} \times \mathtt{X}[i \times \mathtt{INCX}] + \mathtt{Y}[i \times \mathtt{INCY}]$.

À l'aide de ces fonctions, il est possible de calculer Z. L'implémentation avec les BLAS est donc :

```
float *W, *X, *Y;
...
float norm = 0;
float *Z = malloc(n*sizeof(float)); // allocation memoire pour Z
cblas_scopy(n, W, 1, Z, 1);         // Z=W
cblas_sscal(n, a, Z, 1);            // Z=a*Z
cblas_saxpy(n, b, X, 1, Z, 1);      // Z=b*X+Z
cblas_saxpy(n, c, Y, 1, Z, 1);      // Z=c*Y+Z
norm = cblas_snrm2(n, Z, 1);        // calcul de la norme
free(Z);                            // desallocation memoire
```

Selon la version de la bibliothèque choisie, cette implémentation pourra être parallèle et vectorisée. L'utilisation des BLAS permet donc l'exploitation des spécificités des processeurs.

Cependant, deux problèmes limitent les performances de cette implémentation. Premièrement, l'allocation de mémoire et sa désallocation sont des opérations coûteuses, en particulier pour des vecteurs de grande taille. Deuxièmement, la transformation de la boucle `for` présentée (page 8) en quatre appels de fonctions distincts revient à couper cette boucle en quatre, ce qui conduit à augmenter le nombre d'accès à la mémoire. L'implémentation à l'aide des BLAS est en effet équivalente de ce point de vue aux boucles suivantes :

```
float *X, *Y, *Z;
...
float norm = 0;
float *Z = malloc(n*sizeof(float));          // allocation memoire pour Z
for (int i=0 ; i<n ; ++i) Z[i]=W[i];         // Z=W
for (int i=0 ; i<n ; ++i) Z[i]=a*Z[i];       // Z=a*Z
for (int i=0 ; i<n ; ++i) Z[i]=b*X[i]+Z[i]; // Z=b*X+Z
for (int i=0 ; i<n ; ++i) Z[i]=c*Y[i]+Z[i]; // Z=c*Y+Z
for (int i=0 ; i<n ; ++i) norm+=Z[i]*Z[i];   // calcul de la norme
norm = sqrtf(norm);
free(Z);                                     // desallocation memoire
```

Le nombre d'accès mémoire est passé de $3.n$ avec l'implémentation en C à $11.n$ en utilisant les BLAS. Les performances de ce type d'opérations étant dominées par le nombre d'accès à la mémoire, nous pouvons donc nous attendre à ce que les performances soient dégradées d'un facteur 11/3. Dans ce cas, l'utilisation des BLAS devrait donc être contre-productive. Nous avons utilisé la bibliothèque MKL développée par INTEL et implémentant l'interface BLAS. Conformément à nos attentes, cette implémentation permet d'obtenir 1,0 GFlops en utilisant la version séquentielle de la MKL et 1,6 GFlops en utilisant la version parallèle de cette bibliothèque. Dans les deux cas, l'exécutable obtenu est environ quatre fois moins performant que celui optimisé en C comme l'illustre la *figure* 1.3.

Dans son analyse de l'optimisation des performances sur le CRAY I [43], Jack Dongarra décrit une étude analogue. Le problème auquel il s'intéressait pouvait mathématiquement se

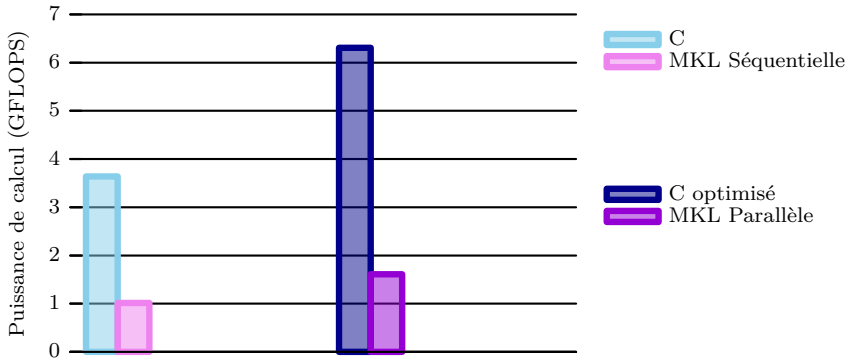


Figure 1.3 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

généraliser en regroupant les vecteurs dans une matrice. Il a donc étendu l'interface BLAS pour prendre en compte les matrices et ainsi résoudre son problème.

Dans notre cas de figure, l'extension de l'interface pour prendre en compte la norme de toutes les expressions vectorielles est impossible. Afin de comprendre cette impossibilité, supposons que l'on étende l'interface des BLAS avec les fonctions fictives `cblas_s2nm2` et `cblas_s3nm2` calculant la norme des expressions vectorielles comportant respectivement deux et trois vecteurs.

L'extension de l'interface BLAS pour prendre en charge la fonction `cblas_s3nm2` permet de résoudre le problème que nous nous étions posé : calculer $||aW + bX + cY||$ en mutualisant les travaux d'optimisation et de portage sur les nouvelles architectures. Cependant, si l'application évolue et nécessite l'introduction d'un quatrième vecteur, nous nous retrouverons face au même problème : il faudra de nouveau étendre l'interface BLAS. La prise en compte de toutes les expressions vectorielles est donc impossible car cet ensemble est infini [6].

1.2.2.2 La composition de fonctions avec les langages procéduraux

Afin de répondre de manière plus pérenne aux problèmes soulevés par notre exemple, il conviendrait de pouvoir composer les fonctions entre elles. Le rôle du vecteur Z est d'établir un lien entre les différents appels BLAS. La composition des fonctions entre elles permettrait de supprimer ce besoin et donc de conserver une seule boucle comme dans l'implémentation en C. La seule possibilité offerte par les langages procéduraux pour composer des fonctions entre elles passe par l'utilisation de fonctions d'ordre supérieur acceptant comme argument des pointeurs sur fonction. Outre la complexité d'utilisation induite par l'utilisation des pointeurs sur fonction, ces derniers introduisent un surcoût du fait de l'indirection et de l'impossibilité d'effectuer des optimisations interprocédurales. Sur notre ${}_{2\times4}^{32}$Nehalem, ce surcoût est d'environ 30 cycles par appel de fonction. Lorsque le corps de cette fonction est suffisamment important, le surcoût peut être négligé. Ce n'est plus le cas lorsque le corps de la fonction de rappel est petit. Dans notre exemple, le corps de ces fonctions de rappel correspondrait à l'itération élémentaire des boucles équivalentes à l'implémentation BLAS et contiendrait donc très peu d'instructions. Par exemple, le corps de la fonction équivalent à la fonction `cblas_saxpy` contiendrait deux instructions flottantes (`y=a*x+y`), traitées en un cycle. Le surcoût d'une fonction de rappel étant

6. En C, l'utilisation de fonctions variadiques permet une implémentation de la norme des expressions vectorielles. Cependant, aucune solution équivalente n'existe dans les autres langages procéduraux comme le FORTRAN.

d'environ 30 cycles, cette approche conduirait à une implémentation 30 fois plus lente que la version en C.

En résumé, dans le contexte du calcul numérique avec un langage procédural, le seul compromis entre maintenabilité et optimisation des performances est obtenu via l'utilisation de bibliothèques externes. Cependant, dans les cas où il n'existe pas de bibliothèque répondant exactement aux besoins de l'application, l'adaptation de l'application pour utiliser les bibliothèques existantes peut être contre-productive comme dans notre exemple. Il convient donc de trouver d'autres approches permettant de dépasser cette limite dans l'expressivité des opérations à réaliser.

1.2.2.3 Les langages fonctionnels

L'expressivité limitée des bibliothèques classiques ne permet pas de trouver un compromis intéressant entre maintenabilité et performances pour les solveurs d'algèbre linéaire de COCAGNE. En effet, de nombreuses opérations nécessaires à la mise au point de ces solveurs ne sont pas fournies par les bibliothèques d'algèbre linéaire. Outre les opérations de normes d'expressions vectorielles, nous pouvons par exemple citer les permutations d'éléments de vecteur [7] ou la résolution de systèmes linéaires dont le membre de droite est une expression vectorielle. Ces opérations doivent donc soit être implémentées et optimisées par les développeurs des applications scientifiques (COCAGNE dans notre cas), soit être implémentées de manière sous-optimale en utilisant l'interface BLAS. Dans ce dernier cas, nous avons vu que la principale limitation réside dans l'impossibilité de composer entre elles les différentes fonctions. Une façon de résoudre notre problème d'expressivité consiste donc à permettre les compositions de fonctions. Les langages fonctionnels reposent sur ce principe et permettent l'écriture de fonctions d'ordre supérieur acceptant d'autres fonctions comme argument. Par exemple, le code suivant correspond à l'implémentation de notre boucle avec le langage fonctionnel Haskell [62] et son module Repa (*Regular Parallel Arrays*) [63], qui est le module de référence concernant les opérations sur les tableaux :

```
import Data.Array.Repa as R
...
norm <- sqrt (
  R.foldAll
  (+)
  0
  (R.map
    (\x->x*x)
    (R.map(a*) w +^ R.map(b*) x +^ R.map(c*) y)
  )
)
```

La fonction `R.map` prend deux arguments. Le premier argument est une fonction unaire tandis que le second est un tableau. La fonction R.map applique alors la fonction unaire à tous les éléments du tableau. Dans l'exemple ci-dessus, `\x->x*x` définit la fonction unaire qui à chaque nombre, associe son carré. L'opérateur `+^` effectue une addition élément par élément de deux tableaux. La fonction `R.foldAll` effectue une accumulation et prend trois arguments. Le premier argument est la fonction binaire d'accumulation, le second argument est la valeur initiale de l'accumulateur et le troisième est le tableau dont les éléments doivent être accumulés.

7. Ces opérations correspondent à la multiplication d'un vecteur avec une matrice de passage orthonormée. L'interface BLAS fournit des fonctions pour effectuer ce type d'opérations dans certains cas très particuliers.

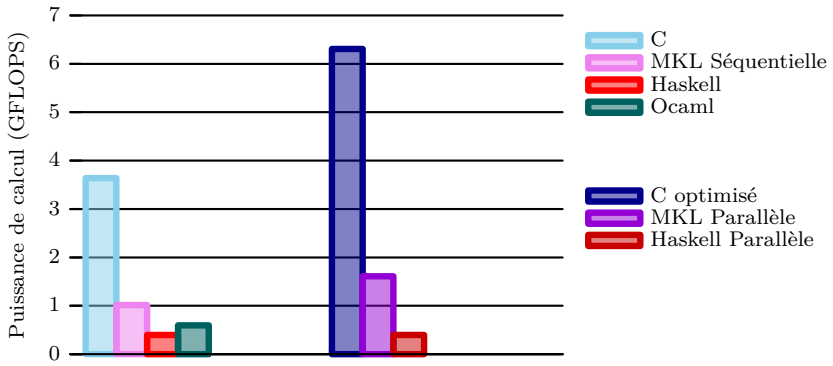


Figure 1.4 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

L'extension pour prendre en charge un troisième vecteur est simple : il suffit d'utiliser l'opérateur `+^` ainsi que la fonction `R.map`. Cependant, cela ne permet pas l'obtention de bonnes performances comme l'illustre la *figure* 1.4. En effet, les versions séquentielles et parallèles de ce programme atteignent 0,40 GFlops. Notons que les performances parallèles sont identiques aux performances séquentielles bien que plusieurs *threads* soient actifs. D'après Ben Lippmeier, coauteur de Repa, cela est dû à un manque de maturité des réductions dans le module Repa [8]. En Avril 2011, l'implémentation Haskell est environ 15 fois plus lente que notre version C optimisée.

Les performances obtenues avec le langage fonctionnel OCaml [64] (0,60 GFlops) sont également présentées sur la *figure* 1.4. Notons cependant que ce langage ne prend pas en charge le type flottant simple précision et que ces mesures sont donc effectuées sur des tableaux d'éléments double précision. Cette implémentation est environ dix fois plus lente que notre version C optimisée utilisant des nombres flottants simple précision.

En 2011, l'utilisation des langages fonctionnels actuels ne permet donc pas aisément de trouver directement un compromis intéressant entre performance et maintenabilité pour les applications qui nous intéressent.

1.2.2.4 Les langages spécialisés à un domaine

Sans nécessiter la prise en charge de la composition de fonctions, certains langages procéduraux permettent toutefois d'exprimer simplement la norme d'expression vectorielle. Par exemple, les langages MATLAB [65] ou Scilab [66] permettent d'exprimer les opérations matricielles et vectorielles très simplement :

```
result = norm(a*W+b*X+c*Y)
```

Afin de permettre la définition de telles opérations, ces langages prennent en charge le concept de vecteur. Les opérateurs `+` et `*` ont dans ces langages une sémantique bien adaptée, directement importée du langage mathématique. De la même manière, MATLAB et Scilab sont des langages restreints, efficaces pour implémenter des algorithmes mathématiques, mais mal adaptés à la mise au point d'autres applications. On parle d'un langage spécialisé à un domaine (*Domain Specific Language* (DSL) dans la littérature [67]). Généralement, la mise au point d'un DSL requiert la conception d'un environnement de développement relativement coûteux. En effet, cela implique le plus souvent le développement d'un compilateur ou d'un interpréteur, d'une bibliothèque d'entrées-sorties et d'outils de débogage.

8. http://www.haskell.org/pipermail/haskell-cafe/2011-April/090998.html

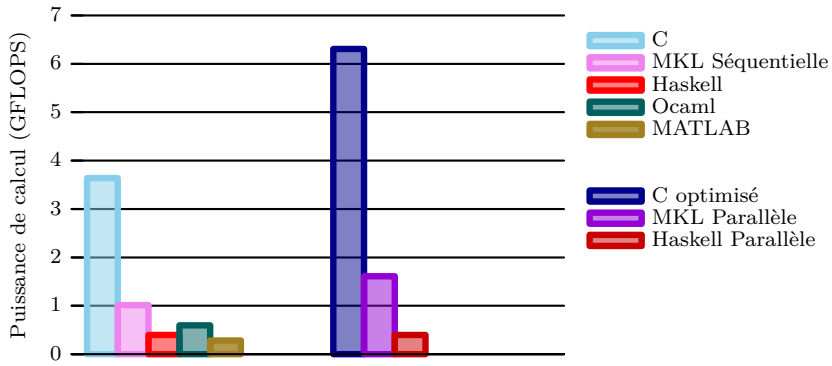


Figure 1.5 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

Cependant, la mise au point d'un environnement offrant de bonnes performances est complexe. Dans le cas de MATLAB, le code présenté ci-dessus offre de très mauvaises performances, comme l'illustre la *figure* 1.5. La version MATLAB plafonne à 0.28 GFlops, soit près de treize fois moins que la version C séquentielle (3.6 GFlops). Une implémentation sous MATLAB qui décompose cette opération en opérations élémentaires BLAS conduit aux mêmes performances. Nous supposons donc que la version de MATLAB disponible sur notre machine de tests (R2009b) ne fusionne pas les différentes boucles et n'utilise pas toutes les possibilités d'optimisation des processeurs modernes. Notons que nous ne disposons pas sur cette machine d'une version parallèle de MATLAB.

La proximité entre la spécification mathématique et son implémentation avec les DSL dédiés à l'algèbre linéaire permet de bénéficier d'une maintenabilité très importante. Cependant, les DSLs disponibles en 2011 ne fournissent pas des implémentations suffisamment performantes pour convenir aux besoins du calcul intensif.

1.2.2.5 La programmation orientée objet

La Programmation Orientée Objet (POO) est une autre approche permettant d'augmenter l'expressivité d'une bibliothèque. En effet, la possibilité de définir des objets possédant des fonctions membres et de passer ces objets en argument d'autres fonctions permet *in fine* d'écrire des fonctions qui prennent d'autres fonctions comme arguments. Un objet est une structure de données valuées et cachées qui répond à un ensemble de messages. Cette structure de données définit son état tandis que l'ensemble des messages qu'il accepte décrit son interface. Nous nous intéresserons aux langages orientés objet qui permettent la modification des données en place, ce qui permettrait d'implémenter efficacement les opérations d'algèbre linéaire.

Des objets de natures différentes (instances de classes différentes) peuvent présenter la même interface mais se comporter différemment. En effet, deux objets peuvent répondre aux mêmes sollicitations mais avec des implémentations différentes dépendant éventuellement de leur état interne. Du point de vue de l'utilisateur d'une Bibliothèque Orientée Objet (BOO), un même extrait de code conduira donc à exécuter des instructions différentes selon l'état interne des différents objets manipulés. Les détails de la conception d'une Bibliothèque Orientée Objet (BOO) C++ pour répondre à notre problème sont détaillées en annexe A.1 (page 143).

Le principe de cette conception consiste à identifier les fonctionnalités communes aux différentes expressions vectorielles. Dans le cadre de la BOO présentée en annexe A.1, il existe trois types d'expression vectorielles :

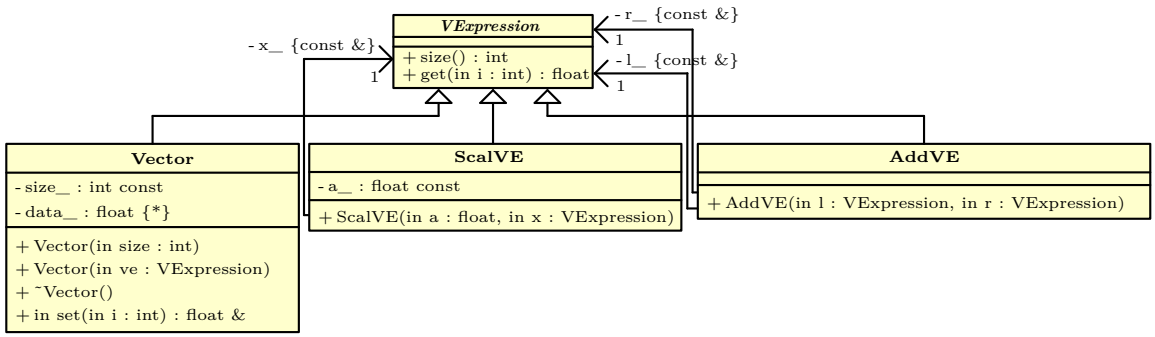


Figure 1.6 : *Diagramme de classes de la bibliothèque orientée objet.*

– les vecteurs (classe `Vector`),
– le produit d'une expression vectorielle par un scalaire (classe `ScaledVectorExpression`),
– la somme de deux expressions vectorielles (classe `AddVectorExpression`).

La *figure* 1.6 représente le diagramme de classes de cette BOO. Le fait que toutes les expressions vectorielles héritent d'une même classe permet d'écrire une fonction unique calculant la norme de n'importe quelle expression vectorielle :

```
1 float norm(const VectorExpression & ve){
2   float out = 0;
3   for (int i=0; i<ve.size() ; ++i) {
4     tmp = ve.get(i);
5     out+=tmp*tmp;
6   }
7   return sqrtf(out);
8 }
```

Selon que `ve` représente un simple vecteur ou une expression plus complexe, l'expression `ve.get(i)` prendra un sens différent. On dit de la fonction

```
float VectorExpression::get(int);
```

qu'elle est polymorphique.

La surcharge des opérateurs `+` et `*` permet alors à l'utilisateur de la bibliothèque d'écrire le code suivant :

Source *1.1 :* *Implémentation du calcul de la norme d'une expression vectorielle avec une BOO*

```
Vector w(n),x(n),y(n);
float  a,b,c;
...
float result = norm(a*w+b*x+c*y);
```

Du point de vue de l'utilisateur, l'utilisation de cette BOO permet d'obtenir une expressivité et une maintenabilité comparables à ce que propose MATLAB pour les vecteurs. En effet, en surchargeant les opérateurs `+` et `*` des expressions vectorielles, nous leur avons ajouté des informations sémantiques et défini une grammaire. Cette BOO fournit donc un DSL. Ce DSL héritant de la grammaire du C++, son langage hôte, il est qualifié de langage enfoui spécialisé à un domaine (*Domain Specific Embedded Languages* ou DSELs dans la littérature) [68, 69, 70].

Selon les possibilités du langage hôte, du compilateur choisi et des techniques employées, cette approche peut fournir les mêmes possibilités d'optimisation que les langages dédiés possédants

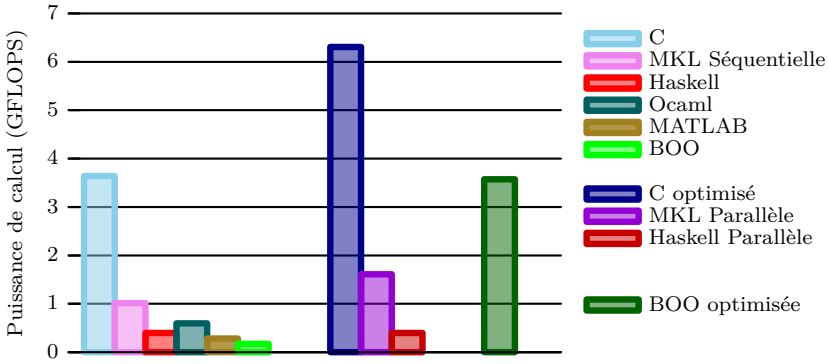


Figure 1.7 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

leurs propres compilateurs. Par conséquent, la mise au point d'un DSEL ne nécessite pas toujours la mise au point d'un nouvel environnement de développement.

Comme les langages orientés objet, les langages fonctionnels permettent aussi de définir des DSEL. Cependant, l'obtention d'implémentations efficaces requiert alors généralement deux étapes dans la génération du programme. Lors de l'exécution du programme écrit avec le DSEL, un arbre syntaxique abstrait (*Abstract Syntax Tree* ou AST) est construit. Cet AST est alors transformé, pour générer un exécutable optimisé. Le traitement de l'AST lors de l'exécution permet d'utiliser des informations qui ne sont pas toujours disponibles lors de la compilation comme la taille des jeux de données ou le problème à résoudre. Certaines optimisations impossibles à réaliser lors de la compilations sont alors permises. Cependant, si l'exécutable devient trop spécialisé, cela peut devenir contreproductif : optimiser un exécutable peut prendre un temps non-négligeable. Cette approche n'est efficace que pour les transformations apportant un gain suffisamment important pour en amortir le coût. Dans le cas où toutes les informations sont connues à la compilation, il est possible de générer et d'optimiser l'AST lors de la génération du programme [71, 72].

La *figure* 1.7 illustre les performances obtenues en utilisant le DSEL fournit par la BOO présentée dans l'annexe A.1 (0,17 GFlops pour la version séquentielle et 3,6 GFlops pour la version parallélisée et vectorisée). Si elles sont meilleures que celles fournies par la MKL, elles restent cependant très inférieures à la version optimisée en C (6.3 GFlops). Ceci est dû au coût des fonctions virtuelles. En effet, les fonctions virtuelles sont implémentées avec des pointeurs de fonction. L'utilisation d'une BOO implique les mêmes limitations de performances qu'une bibliothèque utilisant les pointeurs de fonction. Cette technique est donc à réserver pour les cas où le corps des fonctions virtuelles est suffisamment gros. De plus, l'utilisation de fonctions virtuelles limite les optimisations interprocédurales effectuées par le compilateur. Dans cet exemple, cela se traduit par le fait que le compilateur a été incapable de vectoriser automatiquement cette implémentation.

1.2.2.6 La programmation générative et les bibliothèques actives

Dans l'exemple précédent, toutes les fonctions utilisées peuvent être déterminées à la compilation, il est donc regrettable d'utiliser des pointeurs de fonction, un mécanisme dynamique impliquant un surcoût important lors de l'exécution. Un outil capable d'analyser le code et d'effectuer la composition des opérations lors du processus de compilation permettrait d'obtenir l'expressivité souhaitée sans impliquer de surcoût sur les performances lors de l'exécution. En

C, le mécanisme de macros permet d'effectuer un certain nombre de transformations du code lors de la compilation. Il est cependant trop limité pour répondre à notre besoin. Plus récent, le C++ a généralisé ce système avec les patrons (*templates*) de classe ou de fonction. Beaucoup plus expressifs et fournissant des mécanismes beaucoup plus élaborés, les patrons permettent l'introduction de la programmation générative.

La programmation générique permet de décrire des algorithmes abstraits pouvant s'appliquer à différents conteneurs de données. La bibliothèque de patrons standard du C++ [73, 74] (*C++ Standard Template Library* ou STL) propose ainsi un large ensemble de conteneurs génériques ainsi que de nombreux algorithmes opérant sur ces conteneurs. Par exemple l'algorithme `std::accumulate` permet une implémentation générique des différentes normes. Elle permet ainsi de passer de la norme L_2 aux normes $L_{-\infty}$, L_1 ou L_∞ en modifiant seulement la fonction qui permet d'effectuer l'accumulation des valeurs (dans la fonction `norm` présentée précédemment, cette opération est effectuée ligne 5).

La programmation générative est un style de programmation qui automatise la création de code source grâce à l'utilisation de patrons de classes et de fonctions. La programmation générative permet de mettre au point des bibliothèques qui prennent un rôle actif dans la compilation. Ces dernières sont dites « actives » par opposition aux simples collections passives de routines de calcul [75, 76, 77]. Dans sa thèse [77], Todd Veldhuizen en fournit la définition suivante :

> *Une bibliothèque active est une bibliothèque qui fournit à la fois des abstractions propres à un domaine et les connaissances requises pour les optimiser et vérifier leurs exigences sémantiques. En outre, les bibliothèques actives sont composables : un même fichier source peut combiner l'utilisation de plusieurs d'entre elles.*

Au contraire des bibliothèques traditionnelles ou génériques, les bibliothèques actives (BA) comme Blitz++ [78] ne sont donc pas un simple agrégat de fonctions, de classes ou d'objets. Elles fournissent un haut niveau d'abstraction et sont capables d'optimiser ces abstractions en fonction du contexte dans lequel elles sont utilisées. Elle prennent donc un rôle « actif » lors de la compilation et influencent la génération de code par le compilateur. La possibilité d'opérer des transformations dans le code permet à ces bibliothèques de se rapprocher des compilateurs. Dans l'article *Active Libraries : Rethinking the roles of compilers and libraries* [75] et dans sa thèse [77], Todd Veldhuizen présente en détails les fondements des bibliothèques actives et explique comment l'utilisation de connaissances issues d'un domaine particulier permet d'effectuer des optimisations que le compilateur ne peut effectuer seul.

Appliquées à notre exemple, ces techniques permettent de supprimer les appels aux fonctions virtuelles. En C++, il est possible d'utiliser un patron de conception permettant de résoudre le polymorphisme à la compilation. Cette technique, appelée *Curiously Recurring Template Pattern* (CRTP) [79, 80] permet de résoudre lors de la compilation les liens d'héritage et donc de supprimer les différents pointeurs de fonction par des appels directs aux fonctions de la classe réelle de l'objet, voire de remplacer ces appels par le corps de ces fonctions. Nous dirons alors de ces fonctions qu'elles sont *inlinées*. L'annexe A.2 présente de manière détaillée ces techniques et leur application à la conception d'un DSEL C++ permettant le calcul des expressions vectorielles. L'utilisation de cette bibliothèque est identique à celle de la BOO précédemment introduite. Notons que les possibilités de programmation générative du C++ peuvent être exploitées pour générer des instructions vectorielles lorsque c'est possible comme le fait la bibliothèque d'algèbre linéaire NT^2 [71, 72] ou pour mettre en place une parallélisation ciblant le CPU ou le GPU comme dans le module TPetra de Trilinos [81, 82].

Les mesures « BA » représentées sur la *figure* 1.8 montrent les performances obtenues pour l'implémentation présentée en annexe A.2. Notons tout d'abord que le compilateur a été ca-

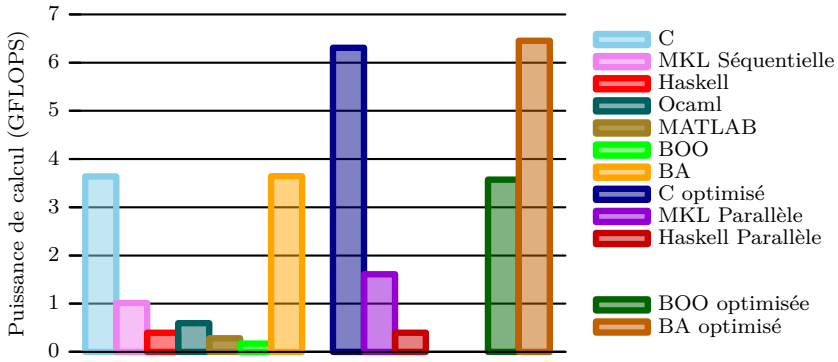


Figure 1.8 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

pable de vectoriser cette implémentation, ce qui montre bien que les fonctions ont pu être *inlinées* pour donner un code équivalent au code C introduit en début de chapitre. Les performances obtenues sont identiques à celles obtenues pour les implémentations en C, en séquentiel (3,6 GFlops) comme en parallèle (6.5 GFlops).

Afin de faciliter la maintenance d'une application scientifique, l'utilisation d'outils externes est une solution intéressante. Cependant, la plupart de ces bibliothèques fournissent des interfaces fixes qui ne permettent pas d'exprimer tous les problèmes de manière pertinente. Nous avons présenté un certain nombre d'approches permettant de surmonter les limitations dans l'expressivité des bibliothèques procédurales. Parmi ces approches, l'utilisation des langages dédiés à la problématique ciblée nous parait la plus prometteuse. En effet, cette approche permet de concilier à la fois une grande expressivité des opérations (cf. source 1.1) et des implémentations efficaces (cf. *figure* 1.8). Afin de ne pas devoir mettre au point un nouvel environnement de développement, nous préférerons leur implémentation sous forme de bibliothèques actives permettant d'utiliser les outils de développement associés au langage hôte.

Dans la prochaine section, nous allons nous intéresser aux méthodes de description des structures de matrices, second point de limitation des bibliothèques procédurales d'algèbre linéaire. Nous verrons alors que le peu d'expressivité des langages procéduraux limite la définition des structures de données tout comme elle limite la définition des opérations de calcul. Nous présenterons alors différentes approches permettant de dépasser cette nouvelle limitation et d'exprimer des structures de données complexes.

1.2.3 Vers une description plus avancée de la structure des matrices

1.2.3.1 Les interfaces procédurales : des structures de données peu évoluées

L'interface BLAS permet de transmettre des matrices denses stockées par lignes ou par colonnes en passant un pointeur et les dimensions du tableau en mémoire. La description du stockage d'une matrice dense pour les BLAS nécessite déjà cinq arguments [83]. Par exemple, pour définir le stockage d'une matrice **A**, les éléments suivants doivent être définis :

- `TRANS` définit si **A** est stockée par ligne ou par colonne,
- `M` définit le nombre de lignes de **A**,
- `N` définit le nombre de colonnes de **A**,
- `LDA` définit la distance en mémoire entre deux lignes consécutives de **A**,
- `A` définit l'emplacement en mémoire de **A**.

Les stockages plus complexes, comme le stockage de Morton [84] qui offre une bonne localité de données [85] ne sont pas pris en charge. Il serait envisageable d'étendre l'interface BLAS pour prendre en compte ce format de stockage mais le problème serait simplement reporté : les autres formats de stockage [86] ne seraient toujours pas supportés. Nous avons vu précédemment que l'extension des BLAS pour prendre en charge le calcul des normes de toutes les expressions vectorielles est impossible car il faudrait ajouter une infinité de fonctions. Ce constat s'étend aux formats de stockage des matrices denses.

Les matrices denses sont des objets mathématiques relativement simples. Pourtant, leur description est déjà complexe et ne nécessite pas moins de cinq arguments pour les formats les plus classiques [83]. Avec les matrices creuses qui contiennent un grand nombre d'éléments nuls, cette description se complique encore (cf. annexe B).

Les formats de stockage CRS (Compressed Row Storage) et CCS (Compressed Column Storage) sont les plus généraux et conviennent aux cas où l'on ne dispose pas d'information concernant la structure des matrices creuses. Dans le cas contraire, ces formats ne permettent pas l'implémentation d'algorithmes efficaces [87] car elles introduisent des indirections lors de l'accès à chaque élément. De plus, leur empreinte mémoire n'est pas négligeable car il faut stocker l'emplacement des éléments non nuls. Par exemple pour une matrice de taille $m \times n$ contenant k éléments et stockée selon les formats CRS ou CCS, la taille cumulée des différents tableaux mis en œuvre est de $2k+n$, soit un surcoût de $k+n$ par rapport au nombre d'éléments. Lorsque les éléments de la matrice sont répartis selon une structure connue à l'avance, il est possible de définir des formats de stockage moins onéreux. Pour des raisons que nous avons déjà évoqué, il est impossible de prendre en charge toutes les structures de matrices : il en existe une infinité. Seules les structures de matrice les plus courantes bénéficient donc de formats de stockage adaptés.

Dans *Templates for the solution of algebraic eigenvalue problems : a practical guide* [88], Jack Dongarra explique comment stocker efficacement quelques structures de matrices récurrentes [87] :

matrices creuses générales (cf. *figure* 1.9(a)) : nous l'avons vu, toutes les matrices creuses peuvent être stockées aux formats CRS (*Compressed Row Storage*) ou CCS (*Compressed Column Storage*) ;

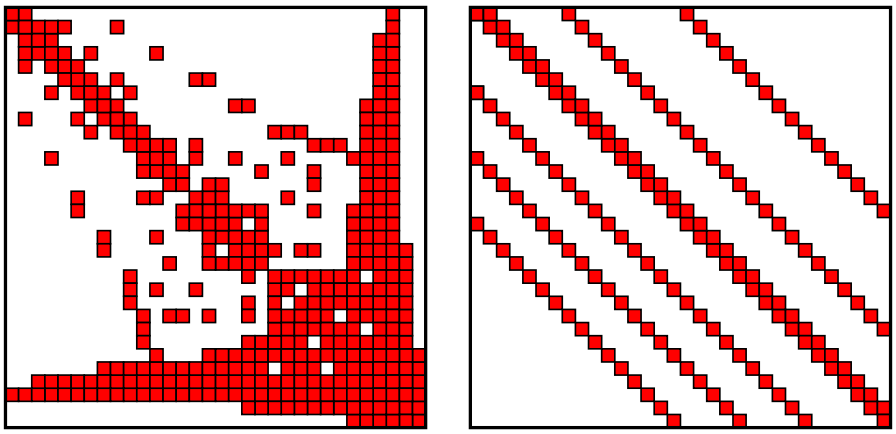
matrices multidiagonales (cf. *figure* 1.9(b)) : ces matrices sont nulles à l'exception de quelques diagonales. Ces matrices pourront être stockées aux formats CDS (*Compressed Diagonal Storage*) ou JDS (*Jagged Diagonal Storage*) ;

matrices à profil (cf. *figure* 1.9(c)) : ces matrices, aussi appelées matrices à bande variable, sont des matrices dont les éléments sont répartis dans une bande de largeur variable autour de la diagonale. Ces matrices peuvent être stockées au format SKS (*SKyline Storage*) ;

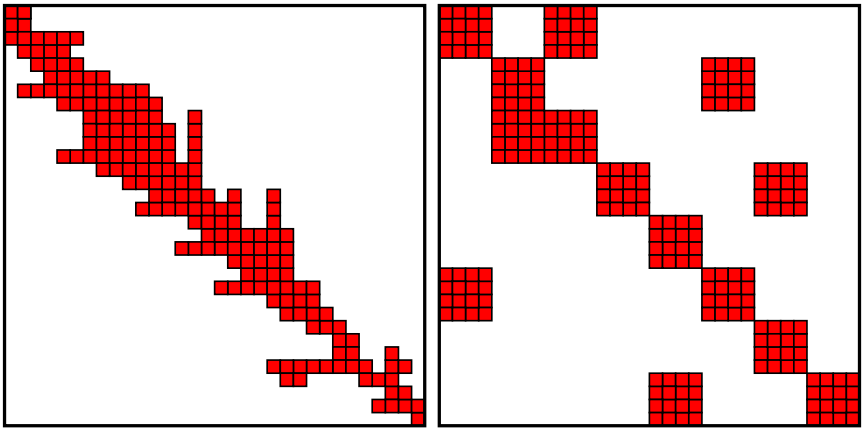
matrices creuses avec blocs denses (cf. *figure* 1.9(d)) : les éléments non-nuls de ces matrices sont regroupés dans des zones de forme rectangulaires. Ces matrices peuvent être stockées aux formats BCRS (*Block Compressed Row Storage*) ou BCCS (*Block Compressed Column Storage*).

L'utilisation des formats spécialisés permet de diminuer l'empreinte mémoire et facilite l'utilisation d'algorithmes plus efficaces. Par exemple, la résolution d'un système à profil est grandement simplifiée et devient naturellement parallélisable si l'on sait que la matrice est en réalité diagonale par blocs. Si ces blocs sont de même taille, une vectorisation des différentes opérations peut également être envisagée (cf. chapitre 2).

Les matrices des solveurs S_N et SP_N ne peuvent être représentées efficacement par les formats de stockage existants. La *figure* 1.10 représente un exemple de matrice issue du SP_N. La structure



(a) matrice creuse non structurée (b) matrice multidiagonale



(c) matrice à profil (d) matrice creuse par blocs

Figure 1.9 : *Différentes structures de matrices*

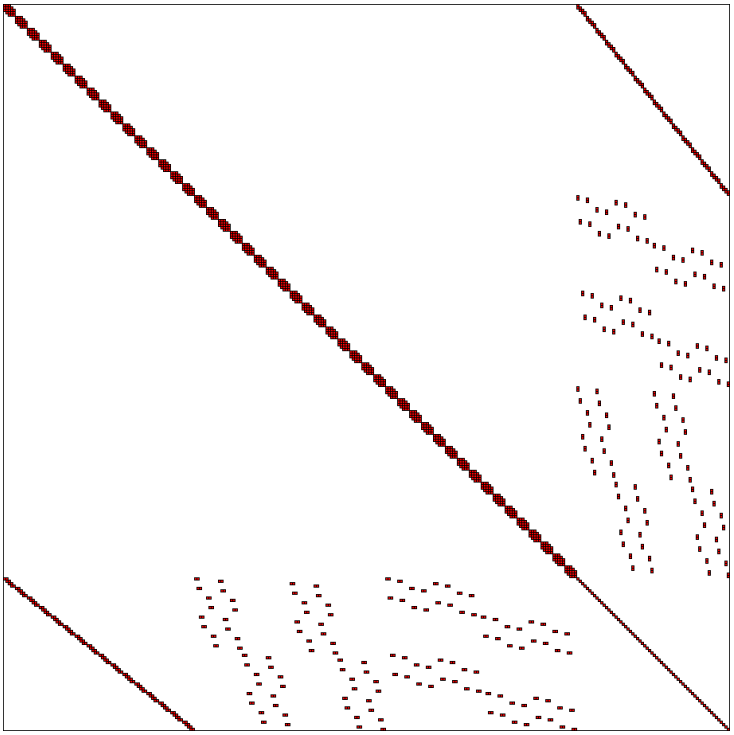


Figure 1.10 : *Matrice creuse issue du code SP_N.*

de cette matrice ne correspond pas aux structures habituellement prises en charges par des formats de stockage spécifiques disponibles. Comme l'emplacement de tous les éléments non nuls de cette matrice est connu lors de l'écriture du code, nous devons utiliser un autre type d'approche permettant d'exprimer l'ensemble des connaissances dont nous disposons concernant la structure de cette matrice. Cela permettra de limiter, voire de supprimer les indirections.

1.2.3.2 Autres solutions pour exploiter les structures de données

Dans la section précédente, nous avons vu qu'il est impossible de décrire précisément les structure de toutes les matrices creuses avec les langages procéduraux. Avec ces langages, la seule solution permettant d'exploiter la structure d'une matrice consiste à implémenter les différentes opérations souhaitées manuellement.

Pour contourner cette limitation, différentes approches ont été explorées. Nous allons en détailler quelques-unes ici.

Certains algorithmes peuvent être optimisés en modifiant la structure des matrices. Cependant, cette restructuration des données a un coût important qui doit être amorti par le gain en performance induit. Les solveurs creux directs comme PaStiX [89] utilisent par exemple des outils d'analyse de la structure de la matrice comme Scotch [90] ou Metis [91]. Ces outils permettent de déterminer les échanges de lignes et de colonnes permettant d'aboutir à une structure plus adaptée pour l'algorithme de résolution utilisé.

Dans les cas où un tel réordonnancement ne permet pas d'améliorer les performances, la question de l'exploitation de la structure de la matrice se pose. Une approche intéressante consiste à laisser l'utilisateur implémenter les opérations élémentaires sur ses matrices (ex : produit,

somme, accès aux éléments) sous la forme de fonctions de rappel [9] (*callbacks*). En s'appuyant sur ces fonctions de rappel, il est possible de mettre au point une couche logicielle implémentant des algorithmes de plus haut niveau. Le solveur PetSc [92, 93, 94] propose ce type de fonctionnalité. Cela peut également permettre de mettre en place des matrices non stockées dont les éléments sont calculés « à la volée ». Cette approche permet à l'utilisateur de choisir de manière générique parmi un catalogue d'algorithmes tout en exploitant une partie des spécificités des structures de ses matrices. Cependant, cela suppose encore une fois de disposer, pour chaque opération élémentaire, d'une implémentation optimisée pour chaque type de matrice et pour chaque architecture matérielle ciblée. Tout ce travail est laissé à la charge de l'utilisateur de la bibliothèque et ne permet pas toujours d'aboutir à un compromis satisfaisant entre performances et maintenabilité des codes.

Enfin, une dernière solution consiste, comme pour le problème des expressions vectorielles étudié précédemment, à définir un langage. En effet, la définition d'un langage permettant de décrire de manière abstraite la structure des matrices (denses, bande, ...) devrait permettre de dissocier les algorithmes de haut niveau, les structures de matrices et leur stockage. Cette structure pourrait donc être utilisée pour déterminer les structures de données et donc les formats de stockage optimaux.

1.2.4 Notre objectif : la maintenabilité des langages dédiés et les performances des bibliothèques optimisées

Dans les deux sections précédentes, nous avons présenté les limites des bibliothèques procédurales. Limites dans l'expressivité des opérations à réaliser d'une part et limites dans la complexité des structures des données descriptibles d'autre part. Différentes solutions pour dépasser ces limites ont été proposées et analysées.

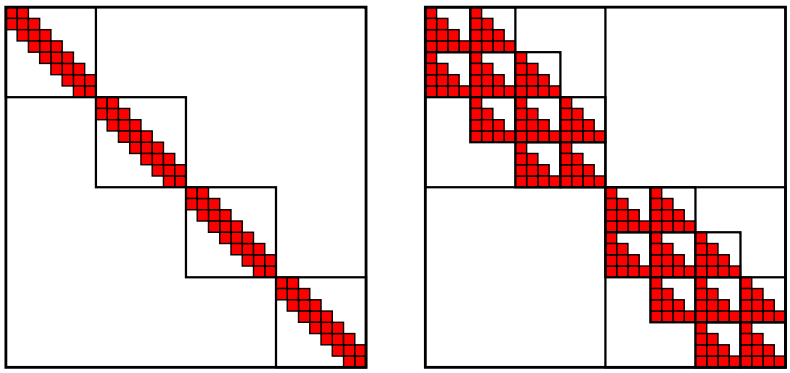
Nous retenons le choix consistant à définir un DSEL, encapsulé sous la forme d'une bibliothèque active. Cette encapsulation s'appuie sur la programmation générative. Elle permet de proposer une expressivité aussi importante que dans le cas des DSL, mais sans sacrifier pour autant les performances de l'application. Afin de parvenir à ce résultat, il convient de définir un DSEL par domaine. Ces domaines doivent être les plus restreints possibles. En effet, plus les domaines sont restreints, et plus nous disposerons de connaissances permettant de guider la génération de l'application.

Dans la prochaine section nous allons présenter Legolas++. Legolas++ est une bibliothèque active proposant deux DSEL interagissant entre eux. Le premier DSEL permet de décrire et d'exploiter des structures de matrices de manière récursive. Le seconde DSEL définit un langage spécialisé pour l'écriture d'algorithmes d'algèbre linéaire.

1.3 Legolas++ : une bibliothèque dédiée aux problèmes d'algèbre linéaire creux et structurés par blocs

Développée à EDF R&D, Legolas++ est une bibliothèque permettant la description et la manipulation de matrices creuses structurées pour la mise au point de solveurs d'algèbre linéaire. GLASS, une précédente version de la bibliothèque, a été présentée dans [33].

9. Une fonction de rappel est une fonction qui est passée en argument à une autre fonction. Cette dernière peut alors faire usage de cette fonction de rappel comme de n'importe quelle autre fonction, alors qu'elle ne la connaît pas par avance.



(a) Structure de la matrice M_2 à deux niveaux de blocs

(b) Structure de la matrice M_3 à trois niveaux de blocs

Figure 1.11 : *Structures de matrices à plusieurs niveaux*

La décomposition d'une matrice en blocs est une opération classique en algèbre linéaire. En effet, de nombreuses méthodes de résolution s'expriment sous la forme de matrices blocs. Par exemple, la matrice bloc suivante :

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix}$$

est utilisée pour exprimer les problèmes de point-selle [95]. Nous avons par ailleurs présenté dans la section précédente le format de stockage (BCSR) pour les matrices dont la structure creuse contient des blocs denses.

Legolas++ propose de réunifier et de généraliser ces approches. En effet, Legolas++ permet de préciser la structure des blocs sous-jacents. La structure d'une matrice est décrite comme la combinaison de différentes sous-structures de matrices. Par exemple, la *figure* 1.11(a) représente une matrice à deux niveaux : un premier niveau de structure diagonale avec des blocs tridiagonaux. Lorsqu'un bloc peut de nouveau être décomposé en blocs, on parle de matrice bloc multiniveaux. La *figure* 1.11(b) présente une matrice à trois niveaux : un premier niveau de structure diagonale, un second niveau de structure tridiagonale, un troisième niveau triangulaire inférieur. Dans le reste de ce chapitre, nous nommerons ces matrices M_2 et M_3 en référence à leur nombre de niveaux.

Legolas++ est une bibliothèque active. L'exemple de la section 1.2.2 utilisait un AST des opérations vectorielles afin d'optimiser le calcul de norme correspondant à l'équation 1.1. Le domaine d'application de Legolas++ est bien plus vaste et nécessite la construction de deux AST différents qui interagissent entre eux. Le premier AST est dédié à la description de la structure des matrices tandis que le second est dédié aux opérations de calcul.

Legolas++ permet de définir de manière récursive des « matrices structurées de matrices structurées » [10]. L'AST d'une structure de matrice est un arbre dont les nœuds sont des blocs et les feuilles des données scalaires. Le type de M_3 est :

```
Diagonal< Tridiagonal< LowerTriangular< float > > >.
```

Dans le cas général, la taille des différents niveaux n'est pas connue à la compilation. L'AST ne peut donc être complètement défini à la compilation. Il convient donc de dissocier la partie

10. L'expression « matrice de matrice » est employée ici de manière abusive car les éléments d'une matrice doivent être dans un corps, ce qui est trop limitant pour décrire les matrices issues du solveur SP_N.

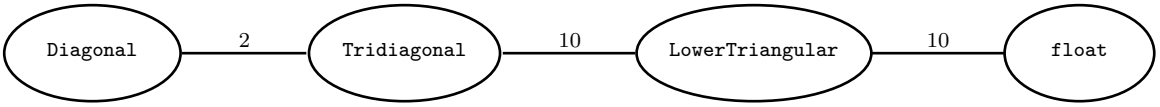


Figure 1.12 : *AST correspondant à la structure de la matrice M_3.*

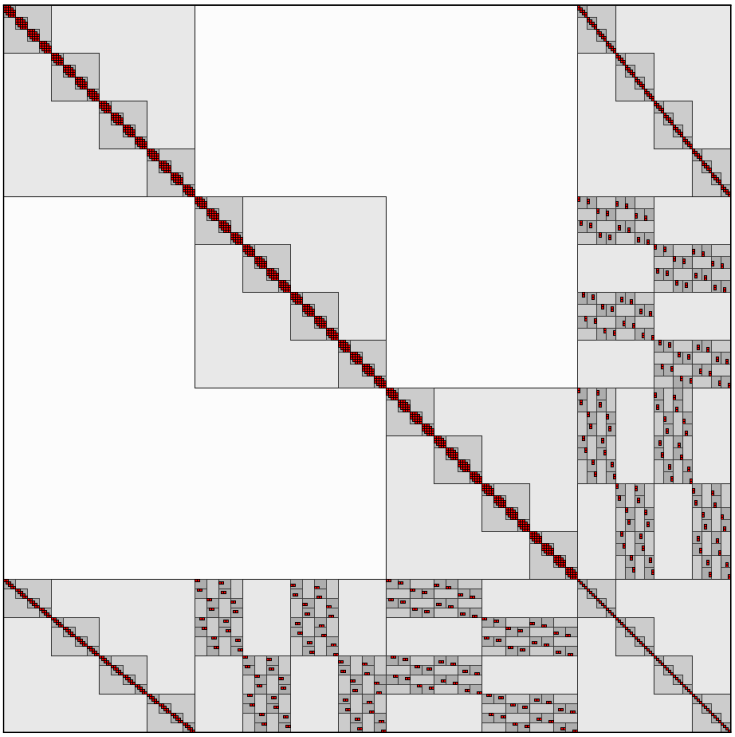


Figure 1.13 : *Structure d'une matrice à cinq niveaux décrite avec Legolas++ dans le solveur SP_N [33]. La densité du gris indique le niveau dans la structure ; les éléments scalaires non-nuls sont représentés en rouge.*

statique, connue à la compilation, de la partie dynamique, connue à l'exécution. Par exemple, le nombre de blocs présents à chaque niveaux est défini lors de l'exécution. L'AST de M_3 est présenté sur la *figure* 1.12. Les nombres qui figurent près des arrêtes de l'arbre représentent le nombre de blocs présents à ce niveau. Cela correspond au nombre de fois que la branche correspondante doit être dupliquée dans l'AST dynamique. Ils correspondent à la taille de chaque niveau. Cet arbre signifie que la matrice de la *figure* 1.11(b) a une structure de type `Diagonal` et contient deux blocs. Ces deux blocs sont de type `Tridiagonal` et contiennent dix blocs chacun (quatre dans la diagonale et six dans les sur- et sous-diagonales). Ces dix blocs sont de type `LowerTriangular` et contiennent dix éléments de type `float`. La *figure* 1.13 explicite la structure de la matrice représentée sur la *figure* 1.10.

Legolas++ permet d'accéder à un bloc de matrice à l'aide de l'opérateur parenthèses `()` comme si c'était un élément de matrice scalaire. Ainsi, `A(i,j)` retournera soit une référence sur un scalaire si `A` n'est pas une matrice bloc, soit un bloc de matrice dans le cas contraire. Dans ce dernier cas, ce bloc est considéré comme une sous-matrice et propose la même interface. L'expression `A(i,j)(k,l)(m,n)` permet donc d'accéder à un élément scalaire de la matrice M_3.

Naturellement, une matrice à plusieurs niveaux opère sur des vecteurs multiniveaux qui doivent donc également être supportés par Legolas++. Pour accéder à un bloc de vecteur, Lego-

las++ fournit l'opérateur crochets `[]`. L'utilisation des opérateurs parenthèses `()` et crochets `[]` permet de différencier la nature mathématique de ces deux types de structures. En effet, l'accès à un élément de matrice nécessite deux coordonnées tandis que l'accès à un élément de vecteur ne nécessite qu'une seule coordonnée. L'expression `X[i][j][k]` permet ainsi d'accéder à un élément d'un vecteur à trois niveaux.

Concrètement, Legolas++ généralise le mécanisme d'*Expression Templates* introduit précédemment pour prendre en charge les matrices et les vecteurs à plusieurs niveaux. Legolas++ permet ainsi de spécialiser l'implémentation des différentes opérations pour chaque structure. Prenons par exemple l'expression matricielle $Y = A \times X + Y$ où A est une matrice et où X et Y sont des vecteurs. Ci-dessous se trouvent les implémentations pour trois types de matrices :

A est une matrice diagonale

```
int nrows = A.nrows();
for (int i=0 ; i < nrows ; ++i)
  Y[i]+=A(i,i)*X[i];
```

A est une matrice tridiagonale (de taille minimale 3×3)

```
int nrows = A.nrows();
for (int i=0 ; i<2 ; ++i)
  Y[0]+=A(0,i)*X[i];
for (int i=1 ; i < nrows-1 ; ++i){
  Y[i]+= A(i,i-1)*X[i-1];
  Y[i]+= A(i,i  )*X[i  ];
  Y[i]+= A(i,i+1)*X[i+1];
}
for (int i=nrows-2 ; i<nrows ; ++i)
  Y[nrows-1]+=A(nrows-1,i)*X[i];
```

A est une matrice triangulaire inférieure

```
int nrows = A.nrows();
for (int i=0 ; i < nrows ; ++i)
  for (int j=0 ; j <= i ; ++j)
    Y[i]+=A(i,j)*X[j];
```

Il est aisé d'allonger cette liste et de fournir un opérateur de multiplication pour chaque structure élémentaire. Si A est une matrice par bloc, les différentes implémentations spécifiques doivent être composées afin de fournir une implémentation spécialisée pour la nouvelle structure. Prenons l'exemple de M_2 représentée sur la *figure* 1.11(a). Cette matrice est diagonale par blocs et ses blocs sont tridiagonaux. Analysons les étapes de compilation qui transforment le produit matrice-vecteur Legolas++ suivant :

```
Y2+=M2*X2;
```

en pseudo-code C équivalent.

Les opérateurs `+=` et `*` permettent de construire un AST correspondant aux opérations de calcul. Cet arbre donne ensuite lieu à une implémentation selon le procédé suivant : pour chaque niveau de la matrice, l'implémentation spécialisée est mise en œuvre et conduit à la construction de nouveaux AST correspondant aux opérations des niveaux inférieurs. Dans notre exemple, la structure de plus haut niveau de M_2 est la structure diagonale. Le compilateur remplacera donc `Y2+=M2*X2` par l'implémentation ci-dessous :

```
int nrows = M2.nrows();
for (int i=0 ; i < nrows ; ++i)
  Y2[i]+=M2(i,i)*X2[i];
```

Les blocs `M2(i,i)` ont une structure tridiagonale. La compilateur remplacera donc l'expression de la ligne 3 par le code spécialisé pour cette structure et générera l'implémentation suivante :

```
int nrows = M2.nrows();
for(int i=0 ; i < nrows ; ++i){
  int nrows2 = M2(i,i).nrows();
  for (int j=0 ; j<2 ; ++j)
    Y2[i][0]+=M2(i,i)(0,j)*X2[i][j];
  for (int j=1 ; j < nrows2-1 ; ++j){
    Y2[i][j]+=M2(i,i)(j,j-1)*X2[i][j-1];
    Y2[i][j]+=M2(i,i)(j,j  )*X2[i][j  ];
    Y2[i][j]+=M2(i,i)(j,j+1)*X2[i][j+1];
  }
  for (int j=nrows2-1 ; j<nrows2 ; ++j)
    Y2[i][nrows2-1]+=M2(i,i)(nrows2-1,j)*X2[i][j];
}
```

La matrice M_3 comporte un niveau supplémentaire et la transformation de l'expression `Y2+=M2*X2` nécessite une étape de plus. Cette dernière étape introduit des boucles spécialisées pour les matrices triangulaires inférieures :

```
int nrows = M3.nrows();
for(int i=0 ; i < nrows ; ++i){
  int nrows2 = M3(i,i).nrows();
  for (int j=0 ; j<2 ; ++j){
    int nrows3 = M3(i,i)(j,0).nrows();
    for (int k=0 ; k < nrows3 ; ++k)
      for (int l=0 ; l <= k ; ++l)
        Y3[i][0][k]+=M3(i,i)(0,j)(k,l)*X2[i][j][l];
  }
  for(int j=1 ; j < nrows2-1 ; ++j){
    {
      int nrows3 = M3(i,i)(j-1,j).nrows();
      for (int k=0 ; k < nrows3 ; ++k)
        for (int l=0 ; l <= k ; ++l)
          Y3[i][j][k]+=M2(i,i)(j,j-1)(k,l)*X3[i][j-1][l];
    }
    {
      int nrows3 = M3(i,i)(j,j).nrows();
      for (int k=0 ; k < nrows3 ; ++k)
        for (int l=0 ; l <= k ; ++l)
          Y3[i][j][k]+=M2(i,i)(j,j)(k,l)*X3[i][j][l];
    }
    {
      int nrows3 = M3(i,i)(j+1,j).nrows();
      for (int k=0 ; k < nrows3 ; ++k)
        for (int l=0 ; l <= k ; ++l)
          Y3[i][j][k]+=M3(i,i)(j,j+1)(k,l)*X3[i][j+1][l];
    }
  }
  for(int j=nrows2-1 ; j<nrows2 ; ++j){
    int nrows3 = M3(i,i)(j,nrows2-1).nrows();
    for (int k=0 ; k < nrows3 ; ++k)
      for (int l=0 ; l <= k ; ++l)
        Y3[i][nrows2-1][k]+=M3(i,i)(nrows2-1,j)(k,l)*X3[i][j][l];
```

répéter
  pour $i = 0$ **a** $n-1$ **faire**
    $S = B_i$
    pour $j = 0$ **a** $i-1$ **faire**
      $S = S - A_{ij}X_j$
    pour $j = i+1$ **a** $n-1$ **faire**
      $S = S - A_{ij}X_j$
    Résoudre : $A_{ii}X_i = S$
jusqu'à *convergence atteinte*;

Alg. 1.1 : Gauss-Seidel

```
do {
  for (int i=0;i<n;i++){
    s=B[i];
    for (int j=0;j<i;j++)
      s-=A(i,j)*X[j];
    for (int j=i+1;j<n;j++)
      s-=A(i,j)*X[j];
    X[i]=s/A(i,i);}
} while (!iter.end(X));
```

Source *1.2* : *Gauss-seidel implémenté avec Legolas++*

```
  }
}
```

Ce mécanisme permet l'écriture d'algorithmes complexes de manière concise et évolutive tout en permettant une optimisation relativement poussée de l'implémentation. En effet, cette approche propose un DSL offrant un haut niveau d'abstraction et permet de décrire simplement des algorithmes par blocs, avec une notation proche du langage mathématique. Par exemple, l'utilisateur peut choisir d'utiliser un algorithme de Gauss-Seidel [96, 97] pour résoudre un système linéaire $A.X = B$ (cf. : **Alg.** 1.1). L'implémentation de cet algorithme avec Legolas++ est donné dans l'extrait de code 1.2 qui est valable quel que soit le nombre de niveaux de la matrice A. Ainsi, si la matrice A est de niveau 1, la variable s sera un élément scalaire ; si la matrice A est de niveau plus élevé, la variable s sera un vecteur de même dimension que B_i.

En outre, le niveau d'abstraction de Legolas++ permet de masquer le parallélisme à l'utilisateur. Dans l'exemple de l'extrait de code 1.2, chaque opération peut éventuellement être parallélisée sans que l'utilisateur ayant écrit ce code n'en soit conscient. La version du solveur SP_N que nous avons développé et présenté dans [36] contient ainsi une seule fonction explicitement parallélisée, Legolas++ masquant tous les autres niveaux de parallélisme.

1.4 Objectif de la thèse : conception d'une version multicible de Legolas++

Dans les années 1990, l'architecture des supercalculateurs a subi une évolution majeure. L'industrie du calcul scientifique a petit à petit abandonné le calcul vectoriel pour se tourner vers des grappes d'ordinateurs superscalaires. Aujourd'hui, nous sommes en train de vivre le même genre de révolution au sein même des processeurs. Le processeur purement superscalaire a disparu du monde du calcul scientifique. La difficulté supplémentaire réside dans le fait que l'architecture superscalaire n'est pas remplacée par une autre architecture, mais par une famille d'architectures. Nous présenterons quelques éléments caractéristiques de cette évolution dans le prochain chapitre. Nous verrons alors que les outils actuels ne sont pas adaptés pour faire face à une telle diversité architecturale.

Nous avons vu dans la section précédente que Legolas++ permet la mise au point de solveurs spécialisés selon la structure des matrices. Les abstractions fournies par Legolas++ permettent en outre d'abstraire le parallélisme présent dans un solveur. La version actuelle de Legolas++ est parallélisée à l'aide de la bibliothèque INTEL TBB. Si cette dernière permet l'exploitation des

différents cœurs présents dans les processeurs, elle ne permet ni d'utiliser les unités vectorielles intégrées au processeur (cf. *tableau* 1.1), ni les accélérateurs de calcul GPU de plus en plus courants dans nos stations de travail.

Cette limitation de Legolas++ ne permet donc pas d'utiliser toutes les ressources matérielles disponibles pour nos simulations. En effet, une étude préliminaire menée en 2008 de parallélisation du solveur SP_N sur GPU, présentée dans [34], a permis d'atteindre de très bonnes performances (facteur d'accélération de 30 comparé à une exécution séquentielle sur le CPU). Cette parallélisation utilisait la structure des matrices mais ne s'appuyait pas pour autant sur Legolas++ : les noyaux de calculs ont été écrits et optimisés manuellement. Cependant, les coûts de maintenance liés à l'intégration de ces travaux dans le code industriel sont prohibitifs. Nous souhaitons étudier la faisabilité d'une extension de Legolas++ pour utiliser automatiquement les différentes architectures spécialisées pour le calcul scientifique.

L'objectif de cette thèse est d'identifier les verrous à lever pour mettre au point une bibliothèque permettant d'exploiter efficacement différentes familles de processeurs de manière transparente à l'utilisateur. Nous proposerons une solution pour lever ces verrous et validerons l'approche ainsi définie avec un démonstrateur multicible de Legolas++.

1.5 Plan de lecture

Dans le prochain chapitre, nous allons introduire un exemple simple d'utilisation de Legolas++ qui nous servira d'exemple de référence dans la suite du manuscrit. Après un bref historique concernant l'évolution des processeurs, nous nous intéresserons aux différences entre les implémentations de cet exemple, optimisées pour les différentes architectures qui nous intéressent. Cette étude nous conduira à introduire la nécessité pour Legolas++ de s'appuyer sur une bibliothèque prenant en charge l'abstraction de l'architecture matérielle des processeurs. Cette bibliothèque devra permettre d'unifier l'expression du parallélisme et d'adapter le format de stockage des données à la cible matérielle choisie.

Le chapitre 3 présente différents environnements de développement parallèles. Cette étude nous permettra d'identifier les caractéristiques clés que devra avoir la bibliothèque de parallélisation de Legolas++ afin de pouvoir prendre en charge différentes architectures matérielles.

Le chapitre 4 expose la bibliothèque *MultiTarget Parallel Skeletons* (MTPS), une bibliothèque de programmation parallèle et multicible. MTPS permet en particulier d'adapter les formats de stockages conformément à l'analyse effectuée au chapitre 2. Nous comparerons les performances obtenues sur notre exemple de référence avec les implémentations optimisées du chapitre 2.

Le chapitre 5 introduit Legolas++ et son domaine d'application. Les concepts introduit par MTPS seront ensuite utilisés pour concevoir une prototype multicible de Legolas++.

Le chapitre 6 synthétise les performances obtenues sur notre exemple de référence avec les différentes approches présentées dans cette thèse. Un second exemple illustre les performances obtenues avec notre prototype Legolas++ multicible sur un cas d'application plus complexe.

À cause du lien étroit entre les concepts introduits dans les chapitres 2, 4 et 5, nous préconisons une lecture de ces chapitres dans l'ordre. Le chapitre 3 peut être lu indépendamment des chapitre 4 et 5, après avoir lu le chapitre 2. Enfin, le chapitre 6 synthétisant les résultats des chapitres 2, 4 et 5, il est cohérent de le lire à la suite de la lecture de ces chapitres. Le lecteur intéressé par les résultats de performance pourra cependant lire ce chapitre indépendamment des autres.

La fonction de la mémoire est aussi importante que celle du calcul

Jacques Le Goff (1924 – présent)
Historien médiéviste français

Chapitre 2

Programmation multicible : une structure de données par cible ?

Dans l'introduction, nous avons vu que le suivi des évolutions des architectures de calcul est un enjeu important pour les codes industriels de calcul scientifique. Afin de faciliter ce suivi et de minimiser le coût d'adaptation aux nouvelles architectures, nous proposons de concentrer au sein de Legolas++ l'adaptation et l'optimisation des codes pour les différentes architectures.

Afin de définir les stratégies d'optimisation adaptées à chaque architecture matérielle, nous allons présenter les évolutions architecturales subies par les processeurs au cours des quinze dernières années. Nous nous intéresserons tout particulièrement aux différentes architectures que nous souhaitons cibler avec Legolas++.

Après avoir proposé des implémentations optimisées pour une opération simple sur ces différentes cibles matérielles, nous étudierons comment leur architecture impacte l'optimisation des codes. Cette étude nous permettra de mettre en avant la nécessité d'utiliser une couche d'abstraction chargée d'unifier l'expression du parallélisme présent dans l'application.

Nous définirons ensuite un cas test que nous utiliserons dans la suite du document pour mesurer l'efficacité des différentes approches. Ce cas test, issu du solveur neutronique SP_N, est un exemple simple d'utilisation de Legolas++. Nous nous appuierons sur ce cas test pour mettre en avant l'importance du format de stockage des matrices et la nécessité de l'adapter à chaque architecture ciblée.

Nous conclurons ce chapitre en spécifiant le besoin d'une couche logicielle intermédiaire entre les différentes cibles et Legolas++. Cette couche doit permettre une implémentation de Legolas++ n'imposant pas de structures de données et exprimant le parallélisme d'une manière unifiée.

Sommaire

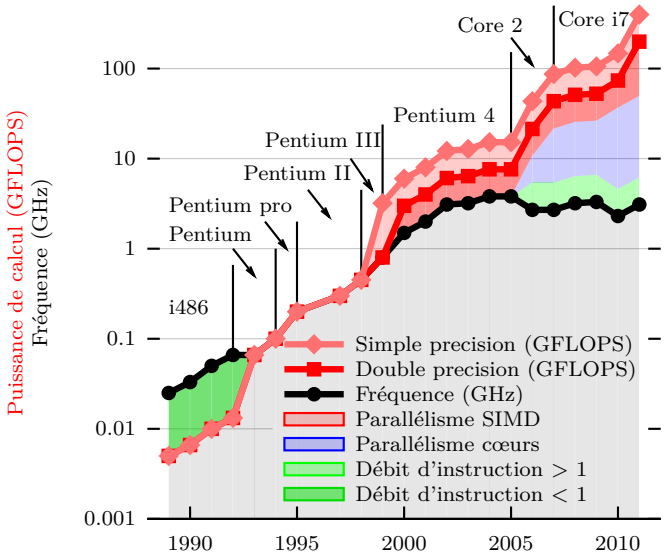


Figure 2.1 : *Approches permettant l'augmentation de la puissance de calcul des processeurs INTEL*

2.1 Présentation des différentes architectures cibles

La guerre économique que se livrent les différents fabricants de processeurs est une course : à la puissance de calcul, à la fréquence, au nombre de cœurs ou plus récemment à la satiété énergétique. Dans le monde du calcul scientifique, le principal critère de sélection d'un processeur reste la puissance de calcul du système qu'il permet de construire. De ce point de vue, diminuer la consommation des processeurs est un critère secondaire, pertinent uniquement s'il permet de mettre plus de processeurs dans le système. Dans cette section, nous allons étudier dans un premier temps l'évolution des processeurs afin de comprendre les choix qui ont été faits dans le but d'augmenter la puissance des processeurs. Nous présenterons ensuite quelques architectures qui nous semblent intéressantes dans le cadre de la mise au point des codes de calculs industriels. Enfin, nous comparerons les stratégies d'optimisation dédiées à ces différentes architectures. Nous déduirons de cette comparaison la nécessité d'unifier l'expression du parallélisme afin de masquer à l'utilisateur l'hétérogénéité des architectures et des modèles de programmation.

2.1.1 Les processeurs : des machines parallèles

L'amélioration des procédés de fabrication permet de multiplier par deux le nombre de transistors par unité de surface tous les deux ans [98], et ce depuis 1971. Cette réduction permet d'augmenter la fréquence et donc la puissance des processeurs. De plus, les transistors supplémentaires disponibles permettent l'ajout de nouvelles fonctionnalités augmentant le nombre d'opérations effectuées par cycle. Entre autres améliorations apportées aux processeurs au cours des dernières années, mentionnons ici les unités de pré-chargement des données (*prefetching*), de prédiction de branchement, ou encore l'augmentation de la taille de la mémoire cache. Le graphique de la *figure* 2.1 montre les choix effectués par INTEL afin d'améliorer la puissance de calcul de ses processeurs [11] [12]. Nous avons distingué différents éléments permettant l'amélioration de la puissance de calcul des processeurs.

11. D'après une première étude menée par SpiralGen et disponible sur `http://spiralgen.com/technology.html` (accédé le 22 mars 2011).
12. Sources de données : spécifications, manuels, guides d'optimisations et site internet `www.Intel.com`.

L'augmentation de la fréquence de fonctionnement f.

Exprimée en Hertz (Hz), la fréquence représente le nombre de cycles d'horloge par seconde. L'augmentation de la fréquence de fonctionnement est rendue possible par la diminution de la taille des transistors : des transistors plus petits et plus proches les uns des autres peuvent commuter plus rapidement [99]. Entre 1993 et 2002, la principale source d'augmentation de la puissance de calcul des processeurs est la fréquence.

L'augmentation du débit d'instructions D_i.

Exprimé en instructions par cycle, le débit d'instructions représente le nombre d'instructions dont le traitement peut être commencé à chaque cycle. Il y a deux possibilités pour améliorer le débit d'instruction : l'amélioration des unités de calcul d'une part ou l'augmentation du nombre d'unités de calcul d'autre part. En effet, il est possible de diminuer le temps de traitement d'une instruction donnée en complexifiant les unités de calcul concernées. Par exemple, l'unité de calcul flottant du co-processeur INTEL 80287 nécessitait entre 70 et 100 cycles pour effectuer l'addition de deux nombres flottants [12]. Le co-processeur INTEL 80387 disposait d'une unité de calcul plus complexe qui requierait entre 23 et 34 cycles par addition [12]. Depuis la sortie du processeur INTEL Pentium, une opération peut être initialisée à chaque cycle [12]. La multiplication des unités de calcul permet d'augmenter le nombre d'instructions pouvant s'exécuter simultanément. Par exemple, tous les processeurs INTEL depuis le Pentium III disposent de deux unités de calcul flottant « SSE » : une première dédiée aux sommes de nombres flottants et une seconde dédiée aux produits de nombres flottants. Il est donc possible d'exécuter simultanément des additions et des multiplications sur ces processeurs [12].

L'augmentation du nombre N_v de voies des unités de calcul.

Exprimé en opérations par instruction, le nombre de voies des unités de calcul représente le nombre d'opérations identiques pouvant être exécutées simultanément sur des données différentes. Par généralisation des travaux de Flynn [100, 101], les unités de calcul dont le nombre de voies est supérieur à un sont dites vectorielles ou SIMD (*Single Instruction Multiple Data*). Ces unités opèrent sur des vecteurs ou « paquets » contenant N_v éléments de données et appliquent la même fonction à ces différents éléments. Les technologies 3DNow !, SSE, SSE2 et AVX introduites dans le *tableau* 1.1 correspondent à des unités de calcul SIMD. Par exemple, l'introduction des unités SSE (*Streaming SIMD Extension*) dans le Pentium III en 1999 permet d'effectuer quatre opérations flottantes simple précision identiques en une seule instruction [102]. Sorti en 2000, le Pentium 4 introduit le jeu d'instruction SSE2 qui permet d'effectuer deux opérations flottantes double précision identiques en une seule instruction [102]. En 2011, les Core i7 de génération Sandy Bridge possèdent des unités AVX permettant d'effectuer 8 opérations simple précision ou 4 opérations double précision en une seule instruction [103].

L'augmentation du nombre de cœurs N_c.

Lorsque plusieurs processeurs sont gravés sur la même puce, ils sont appelés « cœurs de calcul » ou plus simplement « cœurs ». D'abord appelées « puces multiprocesseur » [104], ces puces sont aujourd'hui couramment appelées « processeurs multicœurs ». Tous les cœurs d'un tel processeur bénéficient des améliorations mentionnées ci-dessus. Par exemple, tous les cœurs d'un processeur peuvent comporter des unités SIMD ou voir leur fréquence varier d'une génération à l'autre. Depuis la sortie du Core 2 en 2006, le nombre de cœurs double en moyenne tous les 2 ans.

Au final, nous pouvons définir la puissance de calcul P_c d'un processeur par la formule suivante :

$$P_c = fD_iN_vN_c.$$

Comme N_v dépend du type des données manipulées (flottants simple ou double précision), P_c peut également en dépendre. C'est le cas sur les processeurs X86_64. Les différentes zones de couleur de la *figure* 2.1 représentent les contributions de ces différentes composantes dans l'obtention de la puissance de calcul pour les nombre flottants simple et double précision. Rappelons que le débit d'instructions flottantes est de 0,2 pour le proceseur i486 [12], ce qui justifie que les courbes de performances soient sous la courbe de fréquence pour ce processeur. Notons par ailleurs que les instructions SSE s'exécutent en deux cycles sur les processeurs Pentium III et Pentium 4 [12]. L'utilisation simultanée des deux unités SSE disponibles ne permet donc d'obtenir qu'un débit d'instructions flottantes égal à 1. À partir de la génération core 2, ces instructions s'exécutent en 1 cycle [12]. Les deux unités de calcul étant toujours disponibles, le débit d'instruction passe donc à 2.

Parmi ces quatre approches permettant l'amélioration des performances des processeurs, deux n'impactent pas la génération des exécutables (ni le code, ni la chaîne de compilation) : l'augmentation de la fréquence f ou du débit d'instructions D_i. Ces améliorations sont totalement transparentes pour l'application et permettent donc une accélération « gratuite » des applications lors d'un changement de processeur. Les deux autres approches imposent une transformation du code de l'application afin d'exploiter le parallélisme disponible. Dans le cas des unités vectorielles, on parle de parallélisme SIMD ou de parallélisme « à grain fin ». Pour le développeur d'une application, cela se traduit par l'utilisation de fonctions « intrinsèques » correspondant à une instruction assembleur. Nous verrons dans la section 2.1.3.3 un exemple de parallélisation de code mettant en œuvre les unités SSE. Enfin, l'augmentation du nombre de cœurs se traduit pour le développeur par la nécessité d'exposer des fils d'exécutions parallèles s'exécutant sur chacun des cœurs. Ces fils d'exécutions peuvent prendre la forme de processus ou de *threads* (processus légers).

L'introduction du parallélisme explicite nécessitant une intervention du développeur, les fabricants de processeurs ont longtemps favorisé l'augmentation de la puissance de calcul au travers de l'augmentation de la fréquence. Cependant, l'augmentation de la fréquence implique une très forte augmentation de la consommation énergétique et donc de la chaleur à dissiper. Ces problématiques énergétiques ont finalement conduit les fabricants de processeurs à favoriser les autres approches après avoir atteint la limite des 4,0 GHz. Depuis 2006, les processeurs les plus puissants ont une fréquence comprise entre 2,5 et 3,5 GHz.

Entre 1989 et 2002, les performances des applications ont ainsi été améliorées par la simple augmentation du débit d'instruction et de la fréquence, alors que depuis 2002, la fréquence des processeurs a tendance à baisser. Les performances des applications ne s'améliorent donc plus avec la sortie des nouveaux processeurs. Pour être exploitées efficacement, les dernières évolutions des processeurs nécessitent une modification logicielle mettant en œuvre leurs différents niveaux de parallélisme comme nous le verrons dans la section 2.1.3.3.

2.1.2 Du processeur au processeur assisté : les accélérateurs de calcul

Les besoins de performances des applications ont conduit les concepteurs de processeurs à mettre au point des puces répondant spécifiquement aux besoins des applications utilisées par leurs clients : des « accélérateurs ». En 1967, IBM produit le co-processeur IBM System/360 Model 91, un des premiers accélérateurs de calcul flottant (*Floating Point Unit* ou FPU) [105]. Les FPUs sont maintenant intégrés dans les processeurs centraux mais différentes

applications continuent à justifier l'existence de puces dédiées. Ainsi, estil très commun d'utiliser des DSPs (*Digital Signal Processor*) pour traiter des signaux numériques. Dans les années 1970 les cartes graphiques sont apparues pour faire de l'affichage en temps réel [106]. À cette époque où les processeurs étaient relativement simples, de nombreux accélérateurs ont fait leur apparition. Une grande partie de ces accélérateurs a été absorbée par d'autres processeurs, à l'instar du FPU, intégré dans les processeurs X86 depuis la sortie du 486 par INTEL en 1989. Les processeurs de rendu graphique ont suivi une évolution parallèle et sont également formés de différents accélérateurs : en 1995, ATI commercialise la carte Ati Rage 3D, la première carte graphique intégrant un accélérateur de rendu 3D [13].

Plus récemment, des accélérateurs dédiés aux calculs physiques (*Physics Processing Units* ou PPU) ont commencé à apparaître. Disponible depuis 2000, le système GRAPE-6 [107] permet d'accélérer la résolution du problème des N-corps en astrophysique. SPARTA [108] puis HELLAS [109] sont des architectures de processeurs dédiés à accélérer les calculs de déformations de matériaux. En 2006, Ageia propose les premières cartes PhysiX, comprenant des accélérateurs physiques à destination du grand public afin d'accélérer les traitements physiques dans les jeux vidéos. Depuis l'achat d'Ageia par NVIDIA et l'intégration de PhysX à leurs produits, les GPU NVIDIA sont également des PPUs.

L'évolution des techniques de conception et de production des processeurs ainsi que l'évolution des besoins ont rendu ces différents accélérateurs de plus en plus programmables. En effet, les coûts de conception d'un accélérateur performant sont aujourd'hui tellement élevés qu'il n'est plus économiquement possible de concevoir et de fabriquer une puce pour chaque application. Rendre les accélérateurs plus génériques et plus programmables permet de réutiliser le même accélérateur pour différentes familles de problèmes.

Aujourd'hui, plusieurs types d'accélérateurs de calcul sont couramment disponibles pour effectuer des calculs scientifiques.

Les FPGAs (*Field-Programmable Gate Array*) [110] sont des puces comportant un circuit logique programmable. Les FPGA permettent de programmer une architecture et d'obtenir, ainsi, une puce spécialisée pour le problème que l'on veut traiter. Certains PPU, comme SPARTA, sont construits autour d'un FPGA. Les FPGA sont couramment utilisés pour le décodage de séquences ADN [111]. Mais pour les calculs nécessitant l'utilisation de nombres flottants, les FPGA semblent aujourd'hui peu adaptés car l'implémentation d'unités de calcul flottant sur FPGA ne laisse pas beaucoup de place pour ajouter d'autres unités de calcul. Par la suite, nous ne nous intéresserons donc plus à cette catégorie d'accélérateurs.

Le processeur IBM Cell [112, 113], conçu comme une puce multicœur hétérogène, est composé d'un cœur de type PowerPC 970 chargé de contrôler 8 cœurs SIMD, les SPEs (*Synergistic Processing Elements*). Ces 8 cœurs sont des cœurs contenant une unité vectorielle Altivec (équivalent IBM du SSE) et embarquant très peu de mémoire locale. Dix fois plus puissant que les processeurs classiques disponibles à la même époque [114], le Cell est réputé très difficile à programmer. En effet, l'absence de mémoire cache, les fortes contraintes d'alignement mémoire et la sensibilité des SPEs à l'ordre des instructions impose généralement une programmation proche de l'assembleur et une gestion manuelle des transferts de données entre SPEs. De ce fait, la prise en main du Cell est difficile. No-

13. http://en.wikipedia.org/wiki/ATI_Rage

tons qu'en 2009, IBM a annoncé l'arrêt du développement du processeur Cell [14], sans pour autant renoncer à sortir de nouveaux processeurs dans cette gamme [15]

Les GPUs (*Graphic Processing Unit*) [115, 116, 117] reprennent les grandes lignes des processeurs vectoriels. Conçus à l'origine pour appliquer le même traitement informatique à chaque pixel d'un écran, ils sont de moins en moins spécialisés afin de pouvoir répondre aux besoins de plus en plus variés de l'industrie vidéoludique. Aujourd'hui, ce sont des processeurs comportant un grand nombre d'unités de calcul (jusqu'à 512 dans les GPUs NVIDIA).

Dans la suite de ce document, nous ne nous intéresserons qu'aux processeurs couramment disponibles dans les station de travail, c'est-à-dire aux processeurs X86_64 multicœurs avec unités SSE ou AVX commercialisés par INTEL, AMD et VIA [16] et aux GPUs commercialisés par NVIDIA. En effet, ces processeurs ont l'avantage d'être déjà bien répandus dans les environnements de calcul intensif.

L'étude de ces architectures permettra de déterminer les stratégies à employer afin de réduire les temps d'exécution des codes de calcul scientifiques.

2.1.3 Introduction à l'architecture des processeurs X86_64 et à leur programmation

Nous avons précédemment présenté quelques caractéristiques de l'évolution des processeurs. Nous allons maintenant nous intéresser aux processeurs X86_64. L'analyse de l'influence des réseaux d'interconnexion entre les processeurs et la RAM sur les performances d'un code a été analysé par Mathieu Faverge dans [118, 119]. Il propose différentes méthodes afin d'exploiter plus efficacement ces réseaux. D'après les mesures effectuées sur notre machine d'essai ${}^{32}_{2\times4}$Nehalem (cf. section 2.3), nous pouvons espérer des facteurs de gains de performances maximaux de l'ordre de 1,2 en utilisant ses méthodes d'optimisation. Ce chiffre est à comparer au facteur 32 attendu (8 cœurs $\times$ 4 voies SSE) pour une utilisation optimale du parallélisme. Nous caractériserons donc simplement les réseaux d'interconnexion par leur bande passante et par le débit de données maximal observé et nous nous intéresserons donc dans la suite à l'organisation et l'exploitation des unités de calcul. Notons toutefois que les techniques d'optimisations proposées par Mathieu Faverge sont complémentaires à l'utilisation du parallélisme et qu'il pourra être intéressant dans un second temps d'étudier la manière de les appliquer à notre problème.

Plusieurs techniques permettent de traiter différentes instructions simultanément dans un processeur. Ces technologies peuvent être classées en deux catégories en fonction de leur impact dans l'écriture des codes de calcul. La première catégorie contribue à l'augmentation du débit d'instructions et comprend par exemple le *pipelining* d'instructions [120] et leur exécution *out of order* [120]. L'utilisation de ces technologies est mise en œuvre au sein du processeur et elles ont généralement un impact négligeable sur l'écriture des codes. La seconde catégorie regroupe les technologies de parallélisation nécessitant une transformation du code. Depuis 2002, les processeurs X86_64 présentent différents niveaux de parallélisme. Du point de vue de l'utilisateur, deux types de parallélismes sont à prendre en compte : le parallélisme SIMD et le parallélisme multicœur.

14. http://www.hpcwire.com/hpcwire/2009-10-27/will_roadrunner_be_the_cells_last_hurrah.html, accédé le 14/11/2011.
15. http://www.hardwareheaven.com/news.php?newsid=344, accédé le 14/11/2011.
16. VIA est un fabricant de processeurs X86_64 spécialisés dans les processeurs pour systèmes embarqués.

2.1.3.1 Les unités SIMD

Michael J. Flynn a établi en 1966 une classification des ordinateurs [100, 101]. Cette classification peut aujourd'hui être utilisée au niveau du processeur, du cœur voire de l'unité de calcul. Dans la taxinomie de Flynn, un ordinateur SIMD (*Single Instruction Multiple Data*) dispose d'un décodeur d'instructions et d'une unité de calcul appliquant cette opération sur plusieurs données simultanément.

Avant 1997 et l'introduction d'unités SIMD directement dans les processeurs centraux, chaque processeur disposait d'un décodeur d'instructions et d'une unité de calcul scalaire. Afin de comprendre l'intérêt des unités SIMD dans la conception de processeurs disposant d'une capacité de calcul maximale, introduisons un modèle simplifié du coût en nombre de transistors d'un calcul.

Notons n_d et n_u le nombre de transistors respectivement nécessaires à l'élaboration d'un décodeur d'instructions et d'une unité de calcul scalaire. Nous définissons n_i le nombre de transistors requis pour effectuer i calculs simultanément sur i unités de calcul. Pour une architecture scalaire, chaque unité de calcul est accompagnée de son décodeur d'instructions :

$$n_i = i(n_\mathrm{d} + n_\mathrm{u}).$$

Dans un processeur SIMD à i voies, les i calculs sont identiques, il est possible de mutualiser le décodeur d'instructions entre les i unités de calcul :

$$n_i = n_\mathrm{d} + in_\mathrm{u}.$$

L'augmentation de la finesse de gravure des processeurs a permis d'augmenter le nombre de transistors et l'introduction d'unités de calcul SIMD a permis de consacrer une part plus importante de ces transistors à l'obtention de meilleures performances. Comme illustré dans le *tableau* 1.1 et la *figure* 2.1, l'augmentation du nombre d'opérations réalisables par seconde dans les processeurs récents repose de plus en plus sur les unités SIMD.

Afin d'exploiter pleinement une unité SIMD, il faut que l'algorithme implémenté soit vectorisable, c'est-à-dire qu'il soit décomposable en une suite d'opérations identiques appliquées à des données différentes. Un développeur qui dispose d'une unité SIMD opérant simultanément sur k données mais qui n'exploite pas cette spécificité ne pourra exploiter plus de $\frac{1}{k}$ de la puissance de calcul disponible.

À la différence des machines SIMD définies par Michael J. Flynn, les unités SIMD des processeurs X86_64 définissent un parallélisme SIMD au niveau des registres [121] (*SIMD Within A Register* ou SWAR). Dans les CPUs X86_64, les registres font 128 bits (SSEx [102]) ou 256 bits (AVX [103]). Ces registres travaillent sur des *paquets* de données scalaires. Le nombre de voies varie donc naturellement avec la taille des données scalaires. Ainsi, les unités SSE permettent d'effectuer simultanément quatre opérations flottantes simples précision mais seulement deux opérations flottantes double précision.

Il existe trois familles d'instructions opérant sur les paquets SIMD :

- les instructions permettant d'effectuer des transferts de données avec la mémoire ;
- les instructions vectorielles appliquant la même opération mathématique sur tous les éléments scalaire d'un paquet ;
- les instructions dites de « repaquetage » permettant de construire un nouveau paquet contenant des éléments provenant de deux autres paquets.

Si dans certains cas particuliers ces instructions peuvent être générées par le compilateur, ce n'est en général pas le cas. En particulier, les différents essais que nous avons effectués avec

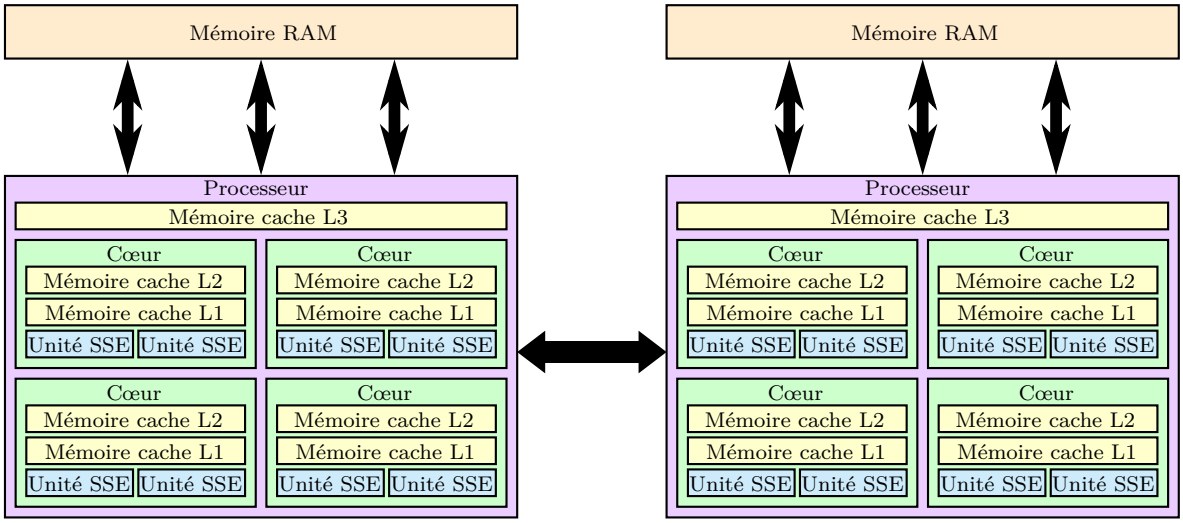


Figure 2.2 : *Architecture d'une station de calcul comportant deux processeurs quadri-cœurs.*

la version 12.0.2 du compilateur INTEL, la version 4.5.3 de g++ et la version 2.9 de clang montrent que les compilateurs actuels ne sont capables de vectoriser que dans le cas où les instructions à vectoriser sont dans un nid de boucle contenant moins de trois niveaux. Dans le cas général, ces instructions doivent explicitement être appelées par le développeur à l'aide de fonctions intrinsèques du compilateur [17] [102, 103]. La prochaine section (page suivante) présente un exemple de code utilisant ces fonctions intrinsèques. Nous verrons également section 2.4 que l'utilisation de ces instructions suppose d'avoir des structures de données adaptées.

2.1.3.2 Les processeurs multicœurs

Comme nous l'avons vu précédemment, « processeur multicœur » est le nom donné aux « puces multiprocesseurs ». Du point de vue du développeur, un ordinateur équipé d'un processeur multicœur est donc très semblable à un système composé de plusieurs processeurs disposés sur des supports différents. La *figure* 2.2 présente l'architecture d'une station de calcul comportant deux processeurs quadri-cœur.

Afin de tirer parti d'un processeur multicœur, les applications doivent être décomposées en tâches pouvant s'exécuter de manière concurrente. Pour les unités SIMD, le parallélisme est exprimé au niveau le plus bas possible : la tâche parallélisée correspond à une opération arithmétique ou logique.

S'appuyant sur des outils coûteux à utiliser, le parallélisme multicœur est exprimé à un niveau plus élevé. En effet, il s'appuie sur la mise en place de plusieurs processus (*processes*) ou de plusieurs fils d'exécutions (*threads*) dont le coût de lancement est important. Par exemple, le lancement d'un *thread* prend environ 7000 cycles sur notre machine de tests $_{2\times4}^{32}$Nehalem. Pour les processus systèmes, ce chiffre est encore plus important. Amortir ces coûts nécessite donc que chaque *thread* exécute une tâche suffisamment longue à traiter.

17. Une fonction intrinsèque est une fonction dont l'implémentation est assurée par le compilateur même. Typiquement, une séquence d'instructions générées automatiquement remplace l'appel de fonction original un peu à la manière d'une fonction *inline*. Cependant, à la différence d'une fonction inline, le compilateur a une connaissance approfondie de la fonction intrinsèque, et par conséquent peut mieux intégrer celle-ci et l'optimiser pour la situation donnée. Les fonctions intrinsèques sont aussi appelées *built-in functions* lorsqu'elle font partie de la définition d'un langage. http://fr.wikipedia.org/wiki/Fonction_intrinsèque

La multiplication du nombre de cœurs au sein des processeurs se traduit par la mise en place d'une hiérarchie du parallélisme. La *figure* 2.2 représente de manière simplifiée l'architecture de notre machine de tests $_{2\times4}^{32}$Nehalem. Celle-ci dispose de deux processeurs (en violet sur la figure), comportant chacun quatre cœurs (en vert sur la figure). Chacun de ces huit cœurs dispose de deux unités SSE (en bleu sur la figure). Dans l'écriture des codes, cela se traduit par la nécessité de mettre en place différents niveaux de parallélisme.

2.1.3.3 Optimisation de l'implémentation du calcul d'une expression vectorielle $||aW + bX + cY||$

Optimiser un code de calcul pour un CPU X86 multicœur impose d'exprimer deux niveaux de parallélisme dans le code. Un premier niveau de parallélisme (*threads*) permet d'exploiter les différents cœurs disponibles. Un second niveau de parallélisme (SIMD) permet d'exploiter les différentes unités de calcul disponibles dans chaque cœur. Dans cette section, nous présentons les transformations induites au sein du code présenté (page 8) et rappelé ci-après :

```
float *W, *X, *Y;
...
float norm = 0;
for (int i=0; i<n; ++i) {
  float tmp = a*W[i]+b*X[i]+c*Y[i];
  norm+=tmp*tmp;
}
norm = sqrtf(norm);
```

Cette opération est trivialement parallélisable : la boucle `for` correspond à une opération de réduction classique. Nous allons utiliser OpenMP [59] pour effectuer le premier niveau de parallélisation et les fonctions intrinsèques SSE [102] pour la vectorisation.

Paralléliser cette boucle avec OpenMP est trivial : il suffit d'ajouter une directive de parallélisation en précisant que la boucle `for` correspond à une réduction. Le code résultant est donc :

```
1 float *W, *X, *Y;
2 ...
3 float norm = 0;
4 #pragma omp parallel for reduction(+: result)
5 for (int i=0; i<n; ++i) {
6   float tmp = a*W[i]+b*X[i]+c*Y[i];
7   result+=tmp*tmp;
8 }
9 norm = sqrtf(norm);
```

Le mot clé `parallel` de la ligne 4 préconise le lancement automatique d'un *thread* par cœur vu par l'environnement d'exécution OpenMP [18]. Ces *threads* seront terminés à la sortie du bloc `for`, ligne 8. Le mot clé `for`, ligne 4, précise que chacun de ces *threads* devra pendre en charge une partie des itérations de la boucle `for` de la ligne suivante (ligne 5). Enfin, le mot clé `reduction`, ligne 4 précise que la boucle `for` définit une opération de réduction dont l'opérateur est l'addition (`+`) et la variable d'accumulation `result`.

L'utilisation des fonctions intrinsèques est plus complexe :

– les pointeurs `W`, `X` et `Y` doivent respecter des contraintes d'alignement ;

18. Ce nombre peut ne pas correspondre au nombre de cœurs. Par exemple, un cœur *hyperthreadé* [122, 104] expose deux cœurs au lieu de un. Il est également possible d'influencer le nombre de *threads* vu par l'environnement d'exécution *via* des variables d'environnement [59].

– `n` doit être un multiple de 4.

Nous supposons que ces contraintes sont respectées pour effectuer ce travail d'optimisation.

Les fonctions intrinsèques SSE correspondent à une programmation de très bas niveau : une fonction intrinsèque représente généralement une seule instruction assembleur [102]. Afin d'implémenter le corps de la boucle, nous allons utiliser le type de données `__m128` qui correspond à un paquet de 4 éléments de type `float`. Dans les commentaires de l'extrait de code suivant, nous noterons `m128[i]` le $i^{ème}$ élément du paquet de flottants `m128`. Notre code optimisé devient finalement :

```
1  #include <smmintrin.h>  // SSE 4.1 intrinsic header
2  ...
3  float *W, *X, *Y;       // W, X and Y are 128bits aligned
4  ...
5  float norm = 0;
6  #pragma omp parallel for reduction(+: result)
7  for (int i=0; i<n; i+=4) {// n%4 = 0
8
9    //wi ={W[i],W[i+1],W[i+2],W[i+3]}
10   __m128 wi = _mm_load_ps(W+i);
11   //a_ ={a,a,a,a}
12   __m128 a_ = _mm_set1_ps(a);
13   //awi={a_[0]*wi[0],a_[1]*wi[1],a_[2]*wi[2],a_[3]*wi[3]}
14   __m128 awi= _mm_mul_ps(a_,wi);
15
16   __m128 xi = _mm_load_ps(X+i);
17   __m128 b_ = _mm_set1_ps(b);
18   __m128 bxi= _mm_mul_ps(b_,xi);
19
20   __m128 yi = _mm_load_ps(Y+i);
21   __m128 c_ = _mm_set1_ps(c);
22   __m128 cyi= _mm_mul_ps(c_,yi);
23
24   //tmp={awi[0]+bxi[0],awi[1]+bxi[1],awi[2]+bxi[2],awi[3]+bxi[3]}
25   __m128 tmp = __mm_add_ps(awi, bxi);
26   tmp = __mm_add_ps(tmp, cyi);
27
28   //result+=tmp[0]*tmp[0]+tmp[1]*tmp[1]+tmp[2]*tmp[2]+tmp[3]*tmp[3]
29   result+=_mm_dp_ps(tmp, tmp, 0xff);
30 }
31 norm = sqrtf(norm);
```

Ce code est bien moins explicite que le code non vectorisé ci-contre. Un code complet devrait en outre contenir différentes branches pour les cas suivants :

– `W`, `X` ou `Y` ne sont pas correctement alignés,
– `n` n'est pas un multiple de 4,
– le processeur ciblé ne supporte pas l'instruction `DPPS` [19] apparue avec le jeu d'instruction SSE 4.1 ; par exemple, les processeurs INTEL Atom ne supportent pas cette instruction.

Il semble donc naturel de déléguer ce travail d'optimisation et de spécialisation à des outils dédiés. Dans les cas simples comme celui-ci, le compilateur génère des instructions SSE vectorisées à partir du code non vectorisé. Cependant, comme nous le verrons dans le chapitre 6, aucun compilateur disponible ne parvient à vectoriser les boucles plus complexes. Notre expérience sur différents cas montre en particulier que lorsqu'il y a plus de deux niveaux de boucle, aucun compilateur ne parvient à générer des opérations vectorielles automatiquement.

19. Cette instruction correspond à l'appel de la fonction `_mm_dp_ps`, ligne 29.

2.1.4 Introduction à l'optimisation pour GPUs NVIDIA

Dans la section précédente, nous avons brièvement présenté l'architecture des processeurs X86_64. Afin d'exploiter efficacement les cœurs disponibles, le code applicatif doit exposer deux niveaux de parallélisme correspondant à une hiérarchie des unités de calcul. Un premier niveau de parallélisme (multicœur) permet d'exploiter les différents cœurs disponibles. Un second niveau de parallélisme (SIMD) permet d'exploiter les différentes unités de calcul disponibles dans chaque cœur. Dans cette section, après avoir présenté un historique de l'architecture des accélérateurs graphiques, nous verrons que l'architecture des GPUs modernes conduit également le code applicatif à exposer deux niveaux de parallélisme. Nous verrons que pour exploiter efficacement un GPU, les contraintes d'optimisations conduisent à envisager l'architecture du GPU comme un processeur multicœur disposant de larges unités vectorielles, voire comme un processeur vectoriel.

2.1.4.1 Historique

Dans cette section, nous allons présenter l'évolution de l'architecture des accélérateurs graphiques afin de comprendre l'émergence des architectures actuelles. Le lecteur intéressé pourra également se référer à [123].

En 1988 le studio d'animation Pixar publie *RenderMan Interface Specification* [124, 125], une interface de bibliothèque permettant d'unifier l'interface de programmation pour les différentes bibliothèques de rendu photoréaliste de scènes 3D. RenderMan permet de dissocier les activités de modélisation et de rendu graphique. Afin de combler le manque d'expressivité des bibliothèques procédurales, RenderMan inclut un DSL dédié au rendu photoréaliste : *RenderMan Shading Language* [126]. L'utilisation de ce DSL permet aux utilisateurs de définir leurs propres effets visuels. RenderMan est une interface de rendu graphique photoréaliste et ne cible pas les applications de rendu en temps-réel : le rendu des 77 minutes du film d'animation *Toy Story* a pris près de 4 mois sur 300 processeurs [127].

En 1992, Silicon Graphic Inc (SGI) propose le standard OpenGL [128] afin de faciliter le développement et la portabilité des applications scientifiques et industrielles nécessitant un rendu de scènes 3D en temps réel. Afin de permettre un rendu des images en temps réel, OpenGL ne peut pas prendre en charge toutes les fonctionnalités de RenderMan. En particulier, OpenGL ne propose pas de DSL. Les différents types de traitement sont prédéfinis par le standard. Ceci permet en contrepartie des implémentations très efficaces capables d'afficher des scènes 2D en temps réel sur les stations de travail disponibles à cette époque.

À partir de 1995, le rendu de scène 3D en temps réel trouve un débouché auprès du grand public grâce aux jeux vidéos. L'ouverture de ce marché, très important en volume, permet de financer le développement d'accélérateurs matériels dédiés au support d'OpenGL. Afin d'améliorer la qualité graphique et les performances des jeux vidéo, des accélérateurs 3D supportant matériellement une partie des traitements OpenGL sont alors intégrés aux consoles de salon et aux ordinateurs grand public [129, 130].

En 1999, NVIDIA commercialise la première carte graphique comportant une « unité de calcul graphique » (*Graphic Processing Unit* ou GPU) : la GeForce 256 [20] est la première carte graphique capable d'accélérer matériellement l'ensemble des fonctionnalités d'OpenGL 1.2. Les GPUs sont alors des processeurs vectoriels avec des décodeurs d'instructions très pauvres : les traitements à appliquer sur chaque élément de donnée sont prédéterminés. OpenGL laisse en

20. http://en.wikipedia.org/wiki/GeForce_256

effet peu de liberté aux développeurs d'applications. En particulier, il ne permet pas la définition de nouveaux effets de rendu.

En 2000, une équipe de SGI montre comment réaliser une implémentation de RenderMan basée sur OpenGL [131]. Dans ces travaux, OpenGL est considéré comme une machine virtuelle SIMD : les fonctions OpenGL faisaient office d'assembleur pour la machine virtuelle. Afin de faciliter ce travail et d'étendre les possibilités de cette machine virtuelle, un assembleur minimal a été proposé comme extension à OpenGL. Bien que les performances de l'implémentation proposée aient été limitées [21], ces travaux ont permis de montrer que les GPUs pouvaient être utilisés comme des accélérateur SIMD pour mettre en œuvre des effets de rendu qui ne sont pas proposés nativement par OpenGL.

En 2002, L'OpenGL Architecture Review Board [22] (ARB) publie *ARB assembly*, un assembleur standard pour programmer les GPU. Si cet assembleur est très limité (il ne propose par exemple pas d'instructions permettant de contrôler le flot d'exécution), il permet cependant aux développeurs de concevoir et d'ajouter de nouveaux effets de rendu. Afin d'accélérer l'application de ces effets, les fabricants de GPUs vont peu à peu augmenter la programmabilité de leurs processeurs.

En 2004, la version 2.0 de OpenGL introduit *OpenGL Shading Language* (GLSL), un DSL dédié au rendu de scènes 3D. Les GPU deviennent alors des accélérateurs graphiques facilement programmables. Les développeurs de codes de calcul intensif commencent alors à exploiter la puissance de calcul disponible sur les GPUs pour effectuer des calculs généralistes (*General-Purpose computing on Graphics Processing Units* ou GPGPU) [132]. En effet, la puissance de calcul affichée des GPUs était bien supérieure à celle des CPUs. Comme l'illustre la *figure* 2.3(a), en valeur absolue, cet écart n'a cessé de croître depuis.

En 2011, le GPU AMD Radeon HD 6970, le GPU le plus puissant disponible sur le marché, dispose d'une puissance de calcul de 2703 GFlops, contre 316 GFlops pour le CPU le plus puissant : le processeur INTEL Core i7-3960X. Dans la version 4.0 du *CUDA Programming guide* [133], NVIDIA l'explique par le fait qu'une part plus grande des transistors est consacrée aux unités de calcul comme l'illustre la *figure* 2.3(b). Héritiers d'accélérateurs très spécialisés, les GPUs modernes ont vu leur programmabilité s'améliorer avec le temps au fur et à mesure que leur puissance de calcul et les besoins de l'industrie vidéo-ludique ont augmenté. Les premiers GPUs étaient des processeurs dont tous les transistors étaient consacrés à l'exécution de fonctions OpenGL pré-câblées. Les GPUs modernes héritent de cette historique de larges unités vectorielles et un très petit cache. À ces éléments se sont ajoutées des unités d'instruction plus évoluées permettant de contrôler plus finement les unités de calcul.

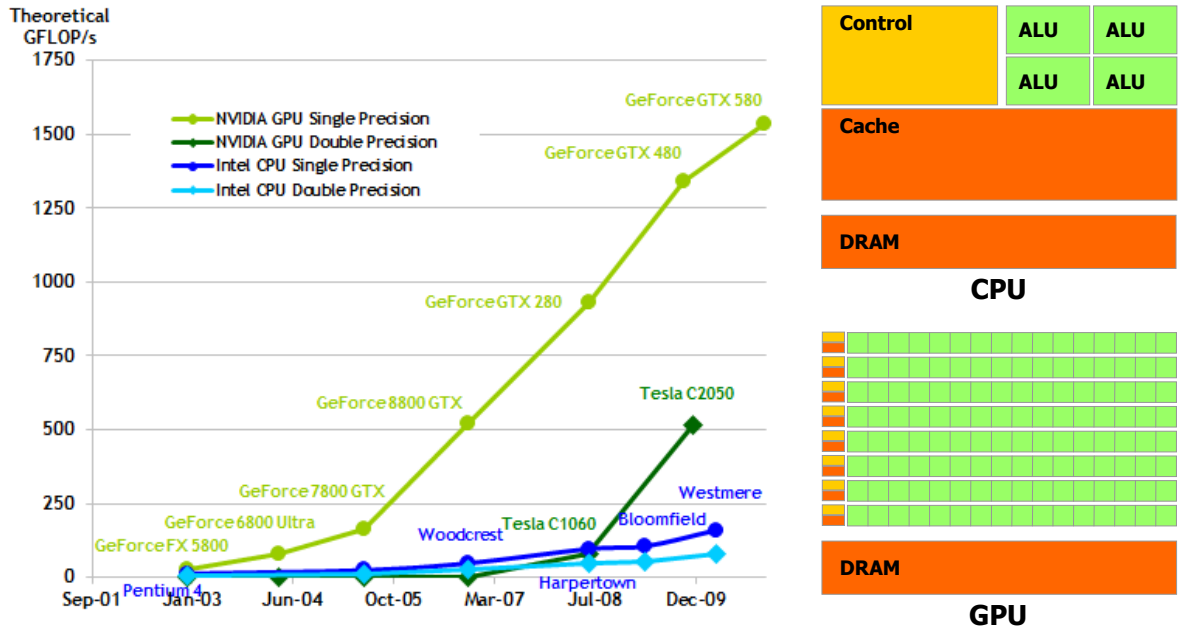
2.1.4.2 Introduction à l'architecture des GPUs NVIDIA

Dans cette section, nous allons présenter les principes de l'architecture des GPUs GF100 commercialisée par NVIDIA en 2011 et utilisée dans les produits dédiés au calcul scientifique (ligne de produit Tesla 20 [23]). Issus de l'architecture Fermi [117], les GPUS GF100 peuvent dans une première approche être assimilés à des processeurs multicœurs disposant d'unités SIMD à 32 voies. Cette section a pour objectif de donner au lecteur les éléments permettant de mieux

21. D'après les auteurs, les faibles performances de cette implémentation sont largement dûes au manque de maturité du compilateur mis au point pour la prise en charge du *RenderMan Shading Language*.

22. L'ARB est le consortium qui encadre alors les spécifications d'OpenGL. Depuis 2006, c'est le Kronos Group (`http://www.khronos.org/`) qui est chargé de faire évoluer OpenGL.

23. Les principes exposés ici s'étendent à tous les GPU NVIDIA GeForce 4XX et 5XX. Cependant, les données chiffrées peuvent varier d'un modèle à l'autre.



(a) Évolution de la puissance de calcul théorique.

(b) Utilisation des transistors.

Figure 2.3 : *Comparaison des caractéristiques des CPUs et des GPUs.*
Source : CUDA Programming guide *[133]*

comprendre les stratégies d'optimisation et les analyses de performances effectuées dans la suite du document. Le lecteur intéressé par les détails de l'architecture des GPUs Tesla 20 pourra se référer au *CUDA Programming guide* [133], à la documentation technique de Fermi [117] ou à la thèse de Sylvain Collange [134] qui effectue également une comparaison de différentes architectures GPU.

Les GPUs GF100 sont commercialisés sous forme de cartes périphériques embarquant de la mémoire RAM et un GPU GF100 comme l'illustre la *figure 2.4*. La carte est reliée au système hôte par un lien PCI Express permettant de transférer des données et de lancer des calculs sur la carte. Bien que les GPUs de la série Tesla 20 ne disposent que de 448 cœurs, ils exposent 21 504 *threads*. En effet, sur un GF100, plusieurs *threads* partagent les mêmes unités de calcul. Ce principe est appelé *Simultaneous Multi Threading* (SMT) [135] dans la littérature. Dans les processeurs INTEL, l'implémentation du SMT est appelée *Hyper-Threading*. La gestion matérielle des *threads* est effectuée de manière hiérarchique. Les trois niveaux de hiérarchie existant sont le *warp*[24], le bloc et la grille.

Un *warp* est un regroupement de 32 *threads*.

Un bloc est un regroupement de *warps*. Un bloc peut contenir entre 1 et 32 *warps*.

La grille regroupe l'ensemble des blocs. Le nombre maximal de blocs pris en charge est de 65535.

24. *Warp* (chaîne en français) est un terme technique du tissage. Sur un métier à tisser, la chaîne correspond à l'ensemble des fils (*threads* en anglais) tendus qui servent de support à la trame. Sur un métier à tisser, tous les fils d'une même chaîne montent ou descendent simultanément ; ce qui correspond au comportement des *threads* d'un même *warp*.

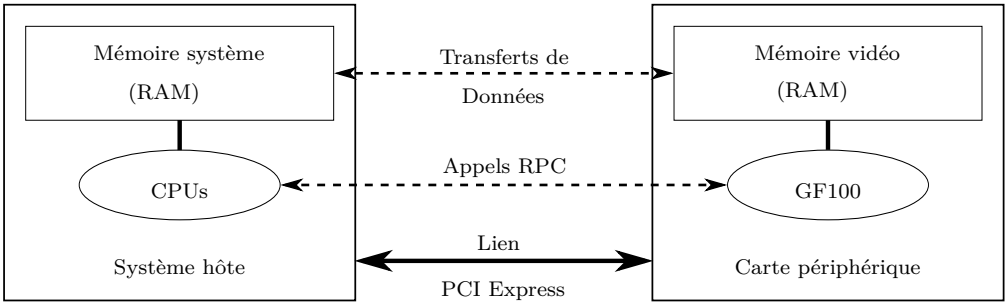


Figure 2.4 : *Les GPUs NVIDIA dédiés au calcul intensif sont commercialisés sous forme de cartes périphériques reliées au système hôte par un lien PCI Express.*

À cette hiérarchie logique correspond une hiérarchie matérielle : les cœurs sont assemblés par groupes de 32 au sein de multiprocesseurs de flux (*Streaming Multiprocessors* ou SM), comme l'illustre la *figure* 2.5. Un GF100 possède 14 SM.

La grille est ordonnancée au niveau du GPU : un ordonnanceur (*GigaThread* dans la documentation technique de Fermi [117]) est chargé de distribuer les blocs aux différents SM. Chaque SM peut prendre en charge simultanément jusqu'à huit blocs correspondant à un total maximal de 1536 *threads*. Les autres blocs sont stockés dans une file d'attente.

Au sein d'un SM, deux *warps* peuvent s'exécuter simultanément. Les deux ordonnanceurs de *warps* (*Warp Scheduler* sur la *figure* 2.5) et les deux décodeurs d'instruction (*Instruction Dispatch Unit* sur la *figure* 2.5) sont chargés de décoder deux instructions : une pour chacun des deux *warps* actifs. Chaque *warp* pourra exécuter son instruction sur une des unités fonctionnelles de calcul suivante :

- deux unités fonctionnelles regroupant chacune 16 cœurs de calcul,
- une unité fonctionnelle regroupant 16 unités d'accès mémoire,
- une unité fonctionnelle regroupant 4 unités de calcul dédiées aux fonctions transcendantes (*sinus*, *cosinus*, *exponentielle*, *logarithme*) et représentées par les quatre SFU (pour *Special Function Unit*) sur la *figure* 2.5.

La description du fonctionnement SMT d'un SM est décrite en détails par Matsushita dans [136]. La principale différence est la gestion par *warps* des unités fonctionnelles dans un SM alors que l'architecture de Matsushita traite de simples *threads*. La gestion des unités fonctionnelles par *warp* implique qu'à chaque cycle, tous les *threads* d'un même *warp* exécutent la même instruction. Si les 32 *threads* d'un *warp* doivent exécuter la même instruction de calcul non transcendante, cette instruction sera exécutée en un cycle par les 16 cœurs du groupe fonctionnel [25]. Au contraire, si les 32 *threads* doivent exécuter des instructions différentes, ces dernières ne pourront être décodées simultanément et leur exécution sera sérialisée. Un SM est une unité de calcul dite *Single Instruction Multiple Threads* (SIMT) [133]. Si un processeur SIMT est plus souple d'utilisation qu'un processeur SIMD, les performances maximales ne pourront être obtenues que dans le cas où la même instruction est exécutée par tous les *threads* d'un *warp*. Pour qu'un algorithme puisse s'exécuter de manière optimale sur un GF100, il faut donc qu'il puisse être implémenté pour un processeur multicœur avec des unités de calcul SIMD à 32 voies.

Sur une carte Tesla C2050, le ratio entre la puissance de calcul théorique et la bande passante théorique implique qu'il faut un minimum de 57 opérations de calcul flottant simple précision par accès mémoire pour exploiter au mieux la puissance de calcul disponible. Une attention par-

25. Les cœurs de calcul ayant une fréquence deux fois supérieure à celle du décodeur d'instructions, chaque cœur peut exécuter deux fois la même instruction par cycle de ce décodeur d'instructions.

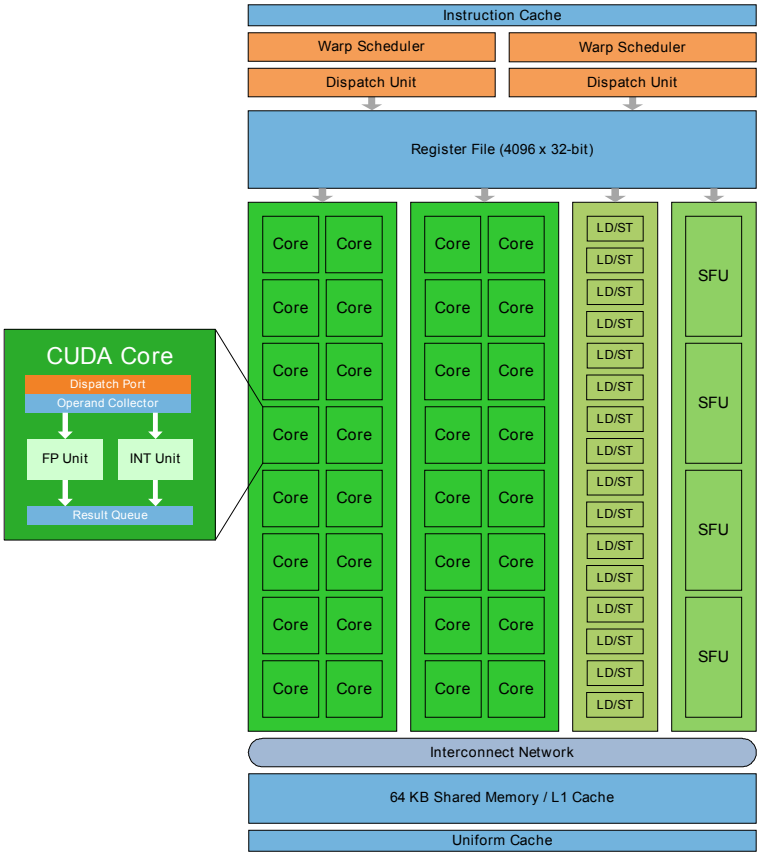


Figure 2.5 : *Architecture des multiprocesseurs de flux du GF100.*
Source : documentation technique de Fermi [117]

ticulière doit donc être portée à la réutilisation des données. Afin de faciliter ce travail, NVIDIA a inclus une hiérarchie de mémoire caches dans le GF100. Comme pour le CPU, l'exploitation de ce cache repose sur la localité spatio-temporelle des données. Cependant, comparé au CPU, le GF100 dispose de très peu de cache : le cache de niveau L2, partagé entre tous les SM, n'est que de 768 Ko pour 21 504 *threads* (1024 *threads* pour chacun des quatorze SMs), soit en moyenne 36 octets par *threads*. Ce chiffre est à comparer aux plusieurs Mo disponibles par *threads* sur un CPU. Pour exploiter efficacement un GF100, il faut donc que les jeux de données réutilisés soient petits ou communs à plusieurs *threads* afin de rester dans les différents niveaux de cache.

2.1.4.3 Implémentation optimisée du calcul d'une expression vectorielle $||aW+bX+cY||$ avec CUDA

Optimiser un code de calcul pour un GPU impose d'exprimer deux niveaux de parallélisme dans le code. Un premier niveau de parallélisme, entre les blocs de *threads*, permet d'exploiter les différents SM disponibles. Un second niveau de parallélisme (SIMT) permet d'exploiter les différentes cœurs de calculs. Nous allons dans cette section présenter les transformations induites au sein du code présenté page 8 et rappelé ci-après :

```
float *W, *X, *Y;
...
float norm = 0;
for (int i=0; i<n; ++i) {
  float tmp = a*W[i]+b*X[i]+c*Y[i];
  norm+=tmp*tmp;
}
norm = sqrtf(norm);
```

CUDA C/C++ est une extension des langages C et C++ proposée par NVIDIA afin de programmer les GPU en ajoutant un certain nombre de mots-clés, de fonctions et de variables. Nous allons utiliser CUDA C/C++ [133] pour paralléliser le calcul de la norme. Nous allons dans cette section présenter uniquement les concepts nécessaires à la compréhension des optimisations effectuées dans la suite du document. Le lecteur plus intéressé pourra se référer au *CUDA Programming guide* [133].

Avec CUDA C/C++, l'implémentation de ce calcul se divise en quatre parties.

1. Nous devons tout d'abord implémenter le noyau de calcul CUDA qui sera exécuté par chaque *thread* GPU,
2. Nous devons ensuite nous assurer que les vecteurs sont présents dans la mémoire GPU en implémentant les transferts de données entre le système hôte et le GPU,
3. Nous devons implémenter l'appel permettant de lancer l'exécution du noyau de calcul CUDA sur le GPU. Cet appel est asynchrone, ce qui signifie que CUDA permet au programme CPU de continuer son déroulement pendant que le noyau de calcul s'exécute sur GPU.
4. Nous devons enfin synchroniser le CPU sur le GPU avant d'effectuer une copie du résultat depuis le GPU vers le système hôte. Nous utiliserons pour cela une fonction de recopie contenant une synchronisation.

NVIDIA préconise d'utiliser les techniques de méta-programmation C++ pour générer les versions du noyau optimisées pour différentes tailles de blocs sur ce type d'application [137]. Afin de simplifier l'écriture du noyau de calcul de notre exemple, nous allons nous limiter au cas où les blocs contiennent 512 *threads* ; chaque bloc contient alors 16 *warps*.

Nous allons maintenant introduire quelques mots-clés CUDA nécessaires à l'implémentation de notre noyau de calcul.

__device__ qualifie les variables globales et les fonctions résidentes dans la mémoire du GPU.

__global__ déclare un noyau de calcul CUDA qui pourra être appelé depuis le CPU et qui s'exécutera sur GPU.

__shared__ permet de déclarer une variable résidant dans un espace mémoire partagé par tous les *threads* appartenant à un même bloc.

Cette implémentation nécessitera également l'utilisation de plusieurs fonctions définies par CUDA.

__syncthreads()) est une fonction qui permet de synchroniser tous les *threads* appartenant à un même bloc.

__device__ float atomicAdd(float*, float) est une fonction qui permet d'additionner un nombre flottant à un autre résidant en mémoire de manière atomique [26].

__device__ uint atomicInc(uint*, uint) est une fonction qui permet d'incrémenter un entier résidant en mémoire de manière atomique [26].

Enfin, CUDA définit un certain nombre d'objets permettant de définir la configuration de la grille ainsi que la position du *thread* et du bloc courant dans la grille.

gridDim contient la dimension de la grille.

blockIdx contient la position du bloc dans lequel se trouve le *thread* exécutant le code.

blockDim contient la dimension des blocs.

threadIdx contient la position, dans son bloc, du *thread* exécutant le code.

Dans chacun de ces objets, la dimension est accessible dans le champs **x** [27].

Le code suivant correspond à l'implémentation du noyau permettant de calculer la norme d'une expression vectorielle :

```
1  #define NB_THREADS_PER_BLOCK 512  // Nombre de threads par bloc
2  ...
3  // accumulateur des resultats des differents blocs
4  __device__ float result = 0.f;
5  // compteur du nombre de resultats partiels accumules
6  __device__ unsigned int count = 0;
7  // noyau de calcul CUDA execute par tous les threads
8  __global__ void normKernel(float a, float * W, float b, float * X
9                  , float c, float * Y, const int n // n~: taille des vecteurs
10                 , float* returnResult)  //adresse de retour du resultat
11 {
12   // nombre total de threads executes
13   int globalNbThreads= gridDim.x*NB_THREADS_PER_BLOCK;
14   // index du thread courant au sein de la grille
15   int globalThreadIndex= blockIdx.x*NB_THREADS_PER_BLOCK+threadIdx.x;
16
17   //scratchpad est un espace memoire partage entre les threads d'un bloc
18   __shared__ float scratchpad[512];
19
20   //chaque thread accumule une norme partielle dans une variable privee
```

26. L'utilisation d'une opération atomique permet de garantir qu'il n'y a pas eu d'accès à ce nombre entre la lecture de la valeur en mémoire et l'écriture du résultat.

27. Ces objets contiennent d'autres champs (**y** et **z**) utilisés lorsque la grille ou les blocs possèdent plus d'une dimension.

```
21  float partialNorm= 0.f;
22  // Les threads d'un warp accedent a des donnees spatialement proches
23  for (int i= globalThreadIndex ; i<n ; i+=globalNbThreads){
24    float tmp= a*W[i]+b*X[i]+c*Y[i];
25    partialNorm+= tmp*tmp;
26  }
27
28  //chaque thread du bloc partage son resultat partiel
29  scratchpad[threadIdx.x]= partialNorm;
30  __syncthreads();
31
32  //Les threads effectuent une reduction binaire :
33  //  a chaque etape, le nombre de threads actifs est divise par 2
34  if (threadIdx.x<256) // la moitie des 512 threads actifs continuent
35    scratchpad[threadIdx.x]+=scratchpad[threadIdx.x+256];
36  else return; // Les threads inactifs ne doivent plus etre synchronises
37  __syncthreads();
38  if (threadIdx.x<128) // la moitie des 256 threads actifs continuent
39    scratchpad[threadIdx.x]+=scratchpad[threadIdx.x+128];
40  else return; // Les threads inactifs ne doivent plus etre synchronises
41  __syncthreads();
42  if (threadIdx.x<64)) // la moitie des 128 threads actifs continuent
43    scratchpad[threadIdx.x]+=scratchpad[threadIdx.x+64];
44  else return; // Les threads inactifs ne doivent plus etre synchronises
45  __syncthreads();
46  // la moitie des 64 threads actifs continuent
47  // Il n'y a plus qu'un warp actif (32 threads) :
48  //  execution synchrone des threads
49  if (threadIdx.x<32){
50    // Pour empecher le compilateur de supprimer les acces a la memoire
51    volatile float * scratchpad_ = scratchpad;  // on utilise volatile
52    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+32];
53    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+16];
54    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+8];
55    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+4];
56    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+2];
57    scratchpad_[threadIdx.x]+= scratchpad_[threadIdx.x+1];
58  }
59
60  //le thread 0 du bloc possede le resultat partiel du bloc
61  if (threadIdx.x==0){
62    // le resultat du bloc est accumule dans l'accumulateur global
63    atomicAdd(&result, scratchpad[0]);
64
65    //si tous les resultats partiel ont ete accumules
66    if(atomicInc(&count, gridDim.x)==gridDim.x-1){
67      //le noyau retourne le resultat
68      *returnResult= sqrtf(result);
69      //les variables globales sont reinitialises pour
70      //  les prochains appels
71      count=0; result=0.f;
72    }
73  }
74 }
```

La boucle de la ligne 23 permet de calculer une norme partielle de l'expression vectorielle. Cette implémentation n'attribue pas à chaque *thread* un intervalle continu de l'expression comme nous aurions eu tendance à le faire sur CPU. Ce choix permet de maximiser la réutilisation des

données. En effet, sur GPU, lorsqu'un *thread* accède à un élément en mémoire, une ligne de cache de seize octets (quatre nombres flottants) est mise en cache. Sur CPU, il faut minimiser le nombre de *threads* accédant à une même ligne de cache afin de limiter les *false sharing* [138]. Cependant, une telle approche nécessite que le cache puisse contenir quatre itérations de calcul pour chaque *thread*. Une itération de calcul nécessite douze octets de données correspondant à `W[i]`, `X[i]` et `Y[i]`. Sur GPU, stocker les données pour quatre itérations, soit 48 octets par *thread* en mémoire cache est impossible : les caches L1 et L2 sont trop petits et ne peuvent stocker que 32 octets par *thread*. Cette implémentation utilise une particularité du cache L1 du GF100 : il est partagé au sein d'un SM. Il n'y a donc pas de protocole de cohérence mis en œuvre si deux *threads* appartenant à un même bloc accèdent à la même ligne de cache. Cette solution permet donc de minimiser le nombre d'accès à la RAM en garantissant que toutes les données mises en cache sont immédiatement utilisées.

La portion de code comprise entre les lignes 32 et 58 (page précédente) correspond à l'opération de réduction effectuée au sein de chaque bloc. Cette opération est effectuée selon un arbre binaire comme illustré sur la *figure* 2.6. À chaque itération, le nombre de *threads* actifs est divisé par deux. Ces *threads* vont ajouter à leur norme partielle la norme partielle des autres *threads*. Ces échanges de données se font en mémoire partagée : il s'agit d'un espace mémoire accessible par tous les *threads* d'un bloc. Afin de garantir la validité des valeurs lues, l'exécution des *threads* est synchronisée à chaque itération, après les écritures. Ces synchronisations étant coûteuses, leur nombre doit être minimisé. À partir du moment où seuls 32 *threads* sont actifs (ligne 49), ils appartiennent tous au même *warp* et sont donc exécutés par le SM de manière synchrone. Il est alors contreproductif d'ajouter des synchronisations. Les itérations restantes de la réduction sont donc simplement enchaînées.

Le compilateur CUDA suppose cependant que les échanges de données entre *threads* donnent toujours lieu à des synchronisations entre les *threads*. Dans le cas contraire, il peut supprimer certaines écritures en mémoire et conserver les données correspondantes en registre. Les lignes :

```
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+32];
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+16];
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+8];
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+4];
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+2];
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+1];
```

sont en effet considérées par le compilateur comme équivalentes à

```
scratchpad[threadIdx.x]+= scratchpad[threadIdx.x+32]
                        + scratchpad[threadIdx.x+16]
                        + scratchpad[threadIdx.x+8]
                        + scratchpad[threadIdx.x+4]
                        + scratchpad[threadIdx.x+2]
                        + scratchpad[threadIdx.x+1];
```

L'utilisation d'un pointeur qualifié de `volatile` permet de prévenir la suppression des accès en mémoire partagée par le compilateur. Nous redéfinissons donc ligne 51 un pointeur `scratchpad_`.

Enfin, les résultats partiels de chaque bloc sont accumulés dans une variable globale `result` initialisée à 0. Le dernier bloc à avoir effectué son accumulation écrit alors le résultat dans l'espace mémoire donné par l'utilisateur et réinitialise les variables globales.

Nous venons de voir comment implémenter efficacement un noyau permettant de calculer la norme d'une expression vectorielle. Afin d'utiliser ce noyau, nous devons maintenant copier les tableaux `W`, `X` et `Y` depuis la mémoire du système vers la mémoire de la carte périphérique. Nous pourrons ensuite lancer l'exécution du noyau de calcul `normKernel` avant de transférer le résultat depuis la mémoire de la carte vers la mémoire système.

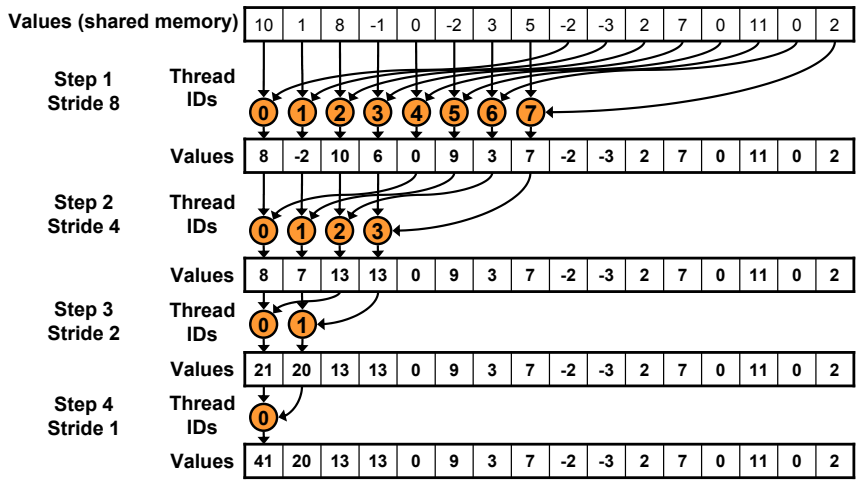


Figure 2.6 : *Arbre de réduction binaire d'une réduction efficace sur processeur GF100.*
Source : tutoriel d'optimisation CUDA présenté par Mark Harris à SuperComputing 2007 [137]

```
1  float *W, *X, *Y;
2  ...
3  float *W_gpu, *X_gpu, *Y_gpu;
4  /*Allocation sur le GPU a l'aide de CudaMalloc*/
5  ...
6  float norm;
7  const uint memSize = n*4; // 4 = sizeof(float)
8  // Les donnees sont transferees depuis la memoire systeme vers le GPU
9  cudaMemcpy((void*)W_gpu, (void*)W, memSize,  cudaMemcpyHostToDevice);
10 cudaMemcpy((void*)X_gpu, (void*)X, memSize,  cudaMemcpyHostToDevice);
11 cudaMemcpy((void*)Y_gpu, (void*)Y, memSize,  cudaMemcpyHostToDevice);
12 dim3 Dg(n/(512*10),1,1); //chaque thread traitera 10 elements de vecteur
13 dim3 Db(512);
14 // Le noyau de calcul est parametre avec Dg et Db puis lance sur le GPU
15 normKernel<<< Dg, Db>>>(a, W_gpu, b, X_gpu, c, Y_gpu, n, result_gpu);
16 // Le resultat est transfere depuis le GPU vers la memoire systeme
17 cudaMemcpy((void*)&norm, (void*)result_gpu, 4, cudaMemcpyDeviceToHost);
18 }
```

Les lignes 9 à 11 permettent de copier les tableaux `W`, `X` et `Y` depuis la mémoire du système vers la mémoire de la carte périphérique.

Les variables `Dg` et `Db` déclarées lignes 12 et 13 définissent la configuration de la grille et des blocs. Elles sont passées comme paramètres d'exécution au noyau de calcul CUDA `normKernel` entre triples chevrons `<<< >>>` ligne 15. `Dg` et `Db` sont accessibles dans le noyau de calcul dans les variables `gridDim` et `blockDim`.

Enfin, la ligne 17 permet de copier le résultat calculé sur la carte périphérique vers la mémoire système.

Ces deux extraits de code (le noyau de calcul CUDA, et celui pilotant son exécution depuis la CPU) doivent ensuite être compilés avec nvcc, le compilateur CUDA fourni par NVIDIA.

2.1.5 Comparaison des stratégies d'optimisation CPU et GPU

Dans les deux sections précédentes, nous avons présenté deux implémentations optimisées du calcul de la norme de l'expression vectorielle $||aX + bW + cZ||$ pour les processeurs X86_64 (page 39) et une pour les GPUs GF100 (page 46). Afin d'analyser les différences d'implémenta-

tion, nous allons tout d'abord décomposer cette opération en deux opérations mettant en œuvre deux types de parallélisme différents.

1. La première est une opération vectorielle correspondant au calcul des valeurs de l'expression vectorielle $aX + bW + cZ$ et où toutes les itérations sont indépendantes les unes des autres.

```
float* tmp;
for (int i=0; i<n; ++i) {
  tmp[i] = a*W[i]+b*X[i]+c*Y[i];
}
```

2. La seconde est une opération de réduction pour calculer la norme de cette expression.

```
float norm = 0;
for (int i=0; i<n; ++i) {
  norm+=tmp[i]*tmp[i];
}
norm = sqrtf(norm);
```

La fusion de ces deux boucles permet de réduire le tableau `tmp` à une simple variable et donc d'obtenir le code présenté page 8.

Nous allons maintenant comparer l'expression de ces deux types de parallélisme dans les deux implémentations optimisées (page 39 et 46).

2.1.5.1 Comparaison des implémentations de l'opération vectorielle

L'opération vectorielle est définie par la suite d'opérations composant le corps de la boucle

```
tmp[i] = a*W[i]+b*X[i]+c*Y[i]
```

ainsi que la borne maximale de la boucle (`n`).

Dans le code optimisé pour les architeures X86_64 (page 39) le corps de la boucle se retrouve entre les lignes 9 et 26. L'expression du parallélisme SIMD s'effectue donc par des appels à des fonctions intrinsèques spécifiques aux unités SSE. Le parallélisme multicœur est effectué avec la directive OpenMP `#pragma parallel for` ligne 6. Aux deux niveaux de parallélisme correspondent donc des paradigmes de programmation différents.

Dans le code CUDA (page 46), seule la parallélisation par *threads* est apparente de manière explicite car la vectorisation est effectué matériellement par les unités SIMT (voir section 2.1.4.2, page 41). Cette opération est exprimée dans la boucle de la ligne 23 et son expression est conforme au code original. Le fonctionnement SIMT a cependant impliqué une modification dans la stratégie d'accès aux données. Les accès sont effectués de la même manière que si le GPU était un processeur vectoriel. En effet, l'incrémentation de la boucle `i+=globalNbThreads` (ligne 23) correspond à l'incrément de la boucle `i+=4` de la ligne 7 (page 39). Dans cet exemple, le GF100 peut être considéré comme un processeur vectoriel dont le nombre de voies est décidé par le nombre total de *threads* lors du lancement du noyau de calcul.

2.1.5.2 Comparaison des implémentations de l'opération de réduction

L'opération de réduction est définie par différents éléments listés ci-dessous.

La fonction de réduction est une fonction binaire associative possédant un élément neutre. Dans notre cas, il s'agit de l'addition et son élément neutre est 0.

L'ensemble des éléments à réduire afin de calculer le résultat de l'opération souhaitée. Dans notre cas, il s'agit de l'ensemble des résultats de l'opération vectorielle.

Dans le code, (page 39), optimisé pour les architectures X86_64, la fonction de réduction n'est pas exprimée de la même manière pour les deux niveaux de parallélisme. Pour le parallélisme multicœur, cette opération est définie en rajoutant la directive OpenMP `reduction(+: result)` à la ligne 6. L'élément neutre est simplement utilisé lors de l'initialisation de `result`. La fonction `__mm_dp_ps` (ligne 29) effectue à la fois la dernière opération vectorielle et l'opération de réduction. L'expression de cette fonction n'est pas directement apparente dans les fonctions intrinsèques SSE. Cette opération de réduction pourrait tout de même être exprimée de manière isolée en utilisant une autre fonction intrinsèque (`__mm_hadd_ps`). Aux deux niveaux de parallélisme sur CPU correspondent donc, ici aussi, des paradigmes de programmation différents.

Dans le code optimisé pour le GF100, l'opération de réduction est effectuée entre les lignes 32 et 73 (page 47). Nous pouvons dans ce cas distinguer les deux niveaux de parallélisme :

- au sein des blocs de *threads*, cette implémentation de la réduction accède aux résultats partiels calculés par les différents *threads* en utilisant la mémoire partagée disponible sur chaque SM (lignes 32 à 58) ;
- entre les différents blocs, cette réduction utilise un accumulateur en mémoire RAM et des opérations atomiques pour synchroniser les communications (lignes 61 à 73).

L'élément neutre de la réduction est précisé pour chacun de ces niveaux (lignes 4 et 21). L'implémentation de chacun de ces deux niveaux pourrait être abstrait dans une fonction équivalente à une fonction intrinsèque SIMD comme `__mm_hadd_ps`. Pour un développeur, le GF100, et plus généralement tout processeur SIMT, peut donc aussi être considéré comme un processeur vectoriel pour les opérations de réduction.

Les deux opérations parallèles que nous avons identifiées se programment différemment sur les deux architectures cibles qui nous intéressent : les CPUs X86_64 multicœurs et les GPU GF100. Le *tableau* 2.1 récapitule les différentes approches utilisées pour exprimer l'opération vectorielle et l'opération de réduction sur ces deux architectures. Nous pouvons constater que la gestion des différentes cibles matérielles a un impact très important sur la maintenance des codes. En effet, le code C à l'origine de notre problème (page 8) comporte six lignes. Si une application utilisant ce code devait disposer de différentes branches pour différentes architectures cibles, il faudrait ajouter à ces 6 lignes 30 lignes pour la version X86_64 optimisée et 82 lignes pour la version CUDA. Le nombre de lignes serait donc multiplié par 25 malgré la simplicité de cette fonction. Cette augmentation du nombre de lignes de code s'ajoute aux autres problèmes que nous avons listés dans le chapitre d'introduction. Cette étude montre donc une nouvelle fois la nécessité d'utiliser des outils permettant d'abstraire l'architecture matérielle et en particulier l'expression du parallélisme.

Dans la prochaine section, nous allons étudier l'optimisation d'un exemple plus complexe, mettant en œuvre des structures de données également plus complexes que le simple vecteur utilisé jusqu'à présent. Nous constaterons alors que selon l'architecture matérielle ciblée, le format de stockage optimal diffère.

2.2 Optimisation pour CPU et pour GPU d'un exemple plus complexe

Dans la section précédente, nous avons pu constater les différences entre les paradigmes de programmation de certains outils dédiés à la parallélisation sur CPU et sur GPU. Nous avons également pu étudier quelques principes d'optimisation, en particulier concernant les accès

Tableau 2.1 : *Comparaison de l'expression des différentes formes de parallélisme.*

Cible	Niveau de parallélisme	Technologie	Operation vectorielle	Operation de réduction
CPU	multicœur	OpenMP	directive OpenMP `parallel for`	directives OpenMP `reduction(+:_)`
	SIMD	SSE	fonctions intrinsèques explicites	fonction intrinsèque explicite
GPU	multiSM	CUDA	lancement du noyau cuda	utilisation des opérations atomiques
	SIMT	CUDA	implicite	utilisation de la mémoire partagée

mémoires sur GPU. Dans cette section, nous allons étudier la parallélisation et l'optimisation d'opérations plus complexes. Nous nous intéresserons tout particulièrement aux adaptations qui doivent être apportées aux structures de données afin de permettre l'utilisation efficace des unités SIMD et SIMT.

Notre expérience d'utilisation de Legolas++ [34] pour mettre au point des solveurs d'algèbre nous a conduit à identifier quatre familles d'opérations récurrentes dans les solveurs d'algèbre linéaire creux et structurés :

les opérations vectorielles comme $Y = aX + Y$ où X et Y sont des vecteurs et a un scalaire,

les opérations de réduction pour calculer des produits scalaires ou des normes,

les opérations de permutations des éléments d'un tableau en mémoire,

les opérations de calcul sur des collections de matrices.

Les deux premières familles d'opérations sont relativement simples à traiter et correspondent au « Hello world » de toutes les approches de programmation parallèles. Elle font en effet partie des exemples classique de programmation sur toutes les architectures matérielles [137, 139, 140, 141, 142]. Tous les exemples de parallélisations que nous avons présentés jusqu'à présent correspondent à ces deux familles d'opérations.

La troisième famille d'opérations correspond à la généralisation de la transposition de tableaux bidimensionnels pour les tableaux comportant plus de deux dimensions et dispose également d'une abondante littérature sur les différentes architectures matérielles [143, 144, 145, 146, 137, 147]. Les opérations de permutations se traitent d'une manière relativement semblable sur les différentes architectures : l'idée principale est de décomposer l'opération globale en un ensemble de permutations de tuiles qui seront transposées dans une mémoire « proche » (mémoire locale ou mémoire cache). L'article de Michael Bader [147] explique très clairement cette idée et montre son implémentation pour GPU.

La dernière famille d'opérations est en revanche moins bien décrite dans la littérature. Dans cette section, nous présenterons un exemple appartenant à cette famille d'opérations ainsi que des implémentations optimisées pour CPU et GPU. Nous étudierons ensuite l'impact de ces optimisations sur les structures de données multidimensionnelles. Nous déduirons de cette étude les transformations à apporter aux structures de données afin d'exploiter efficacement les unités SIMD et SIMT.

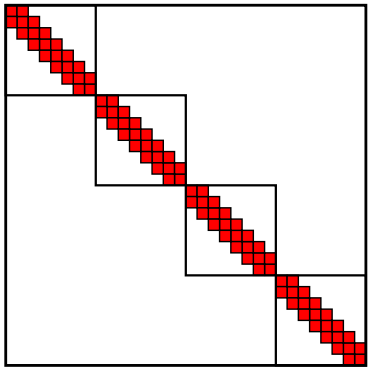


Figure 2.7 : *Matrice diagonale à blocs bandes et symétriques. Cette matrice contient 4 blocs tridiagonaux de taille 8×8*

2.2.1 Présentation du problème

Nous allons étudier les performances sur différentes cibles matérielles de la résolution de l'équation $\mathbf{AX} = \mathbf{B}$ où $\mathbf{A}$ est une matrice diagonale à blocs bandes et symétriques comme celle de la *figure* 2.7. La matrice $\mathbf{A}$ est définie par :

- le nombre de blocs ;
- la taille des blocs ;
- la demi largeur de bande des blocs qui correspond au nombre de diagonales inférieures.

Par exemple, la *figure* 2.7 représente une matrice diagonale par blocs comportant quatre blocs de taille 8×8 et avec une demi largeur de bande égale à un.

Notons respectivement A_{ii}, X_i et B_i les éléments de $\mathbf{A}$, $\mathbf{X}$ et $\mathbf{B}$. La matrice $\mathbf{A}$ étant diagonale par bloc, la résolution du système $\mathbf{AX} = \mathbf{B}$ est équivalente à la résolution de l'ensemble des sous-problèmes indépendants $A_{ii}X_i = B_i$.

Afin de résoudre ces problèmes, nous utiliserons une forme LDL^{T} [148, 149], présentée dans l'**Alg.** 2.1, de la décomposition de Cholesky suivie d'une opération dite de « descente-remontée », présentée dans l'**Alg.** 2.2. Cette méthode de résolution a la particularité de pouvoir être appliquée « en place ». Autrement dit, les résultats L_{ii} et D_{ii} de la décomposition peuvent être stockés directement dans A_{ii}. Nous noterons A^d_{ii} la forme décomposée de A_{ii}. De même, le résultat X_i de la descente-remontée peut être écrit directement dans B_i. Dans la suite, nous considérerons simplement l'opération de descente-remontée opérant sur la matrice $\mathbf{A}$ et le vecteur $\mathbf{X}$ (que nous aurons préalablement initialisé pour être égal à $\mathbf{B}$).

2.2.2 Implémentations optimisées

Pour implémenter ce problème sur les différentes architectures dont nous disposons, nous allons utiliser des classes représentant la matrice $\mathbf{A}$ et le vecteur 2D $\mathbf{X}$. Nous spécialiserons l'implémentation de ces classes pour chaque architecture. Cela nous permettra d'écrire les algorithmes de factorisation et de descente-remontée une seule fois pour toutes les architectures.

Nous présenterons dans un premier temps l'interface d'utilisation des matrices et des vecteurs ainsi que la fonction correspondant à la descente-remontée. Nous présenterons ensuite les implémentations de ces classes pour les différentes cibles. Pour chaque cible matérielle, nous proposerons différents formats de stockage de la matrice $\mathbf{A}$ et du vecteur $\mathbf{X}$ et nous étudierons leur influence sur les performances.

Données

A : une matrice bande, symmetrique et définie positive
A^d : la matrice A décomposée sous la forme LDL^{T}
D : une matrice diagonale
L : une matrice bande inférieure avec une diagonale unitaire
L^{T} : la matrice transposée de L
hbw : la demi largeur de bande (Half BandWidth) de A, correspond au nombre de diagonales de L
n : nombre de lignes de A
B : le vecteur membre de droite
X : le vecteur inconnu

```
// Coin supérieur
pour i ← 0 a hbw faire
    pour j ← 0 a i faire
        A^d_ij ← A_ij
        pour k ← 0 a j faire
            A^d_ij ← A^d_ij − A^d_ik A^d_jk
    pour j ← 0 a i faire
        A^d_ii ← A^d_ij − A^d_jj (A^d_ij)^2
        A^d_ij ← A^d_ij A^d_jj
    A^d_ii ← 1 / A^d_ii
// Partie centrale
pour i ← hbw a n faire
    pour j ← i − hbw a i faire
        A^d_ij ← A_ij
        pour k ← i − hbw a j faire
            A^d_ij ← A^d_ij − A^d_ik A^d_jk
    pour j ← i − hbw a i faire
        A^d_ii ← A^d_ij − A^d_jj (A^d_ij)^2
        A^d_ij ← A^d_ij A^d_jj
    // A^d_ii contient 1/D_ii afin
       d'éviter des divisions lors
       de la descente remontée
    A^d_ii ← 1 / A^d_ii
```

Alg. 2.1 : Décomposition de Cholesky (forme LDL^{T}).

```
/* Étape 1 : LX = B                    */
// Descente
// Coin supérieur
pour i ← 0 a hbw faire
    X_i ← B_i
    pour j ← 0 a i faire
        X_i ← X_i − A^d_ij B_j
// Partie centrale
pour i ← hbw a n faire
    X_i ← B_i
    pour j ← i − hbw a i faire
        X_i ← X_i − A^d_ij B_j
/* Étape 2 : DX = X                    */
pour i ← hbw a n faire
    // A^d_ii contient 1/D_ii afin
       d'éviter des divisions
    X_i ← A^d_ii X_i
/* Étape 3 : L^T X = X                 */
// Remontée
// Partie centrale
pour i ← n − 1 a hbw faire
    pour j ← i − hbw a i faire
        X_j ← X_j − A^d_ij X_i
// Coin supérieur
pour i ← hbw a −1 faire
    pour j ← 0 a i faire
        X_j ← X_j − A^d_ij X_i
```

Alg. 2.2 : Descente-remontée

2.2.2.1 Principe général

Dans cette section, nous allons présenter les concepts de vecteur et de matrice ainsi que leur utilisation pour l'implémentation de la descente-remontée.

```
class Vector {
  public:
    typedef RealType; // le type de reel utilise, generalement float
    INLINE Vector(...);       // un ou plusieurs constructeurs
    INLINE int size() const;  // retourne la taille du vecteur

    // deux accesseurs pour lire et ecrire les donnees
    INLINE const RealType & operator[](int i);
    INLINE RealType & operator[](int i);
};
```

La classe `Vector` dispose d'une interface classique : les opérateurs crochets `[]` permettent d'accéder aux éléments du vecteur. Nous verrons lors de l'implémentation SSE qu'il n'est pas possible de laisser le type des données `float` dans l'interface. Ce type est donc défini par `RealType`.

La macro `INLINE` correspondra à des mots-clé différents selon les implémentations. Dans les implémentations X86_64, elles définiront le mot-clé C++ `inline`. Dans l'implémentation CUDA, elle définira les mots-clés `__device__`, `__host__` et `inline` afin que ces fonctions soient aussi compilées pour le GPU. Notons qu'une instance de la classe `Vector` peut être construite sur CPU ou sur GPU.

Nous allons maintenant présenter la classe `Vector2D`. Cette classe ne dispose pas de constructeur, ce qui est relativement inhabituel pour une classe C++. Il s'agit d'une des contraintes à respecter pour permettre aux instances de cette classe d'être copiées comme des structures C (avec `memcpy` par exemple). En effet, nous devrons passer une instance de cette classe aux noyaux de calcul CUDA. Les contraintes imposées par CUDA pour ce type de classe sont assez proches des contraintes du C++ pour la définition des POD [28] (*Plain Old Data*) [74]. À notre connaissance, ces contraintes ne sont pas complètement formalisées dans la documentation CUDA [29]. Notre expérience nous permet cependant de définir les contraintes suivantes :

- la classe ne doit pas définir de constructeur,
- la classe ne doit pas contenir de données membres statiques [30],
- la classe ne doit pas contenir de fonctions virtuelles,
- tous les membres données de la classe doivent respecter ces mêmes conditions ou être des types scalaires au sens de la norme du C++ [74].

La classe `Vector2D` comportera donc deux fonctions `initialize` et `copy` qui remplacent les constructeurs et allouent la mémoire GPU tandis qu'une fonction `finalize` fait office de destructeur et désalloue la mémoire. Notons que ces trois fonctions n'ont pas vocation à être exécutées sur le GPU et ne prennent que le mot-clé `inline` (et pas `INLINE`). L'opérateur crochets de la ligne 13 construit à la volée les instances de la classe `Vector` représentant un bloc X_i de vecteur. Les instances de cette classe ne font l'objet d'aucun transfert entre la mémoire système et la mémoire du GPU. C'est ce qui permet à la classe `Vector` de ne pas être soumise à ces contraintes.

28. Les POD sont des types compatibles avec C. Il s'agit soit de types scalaires au sens de la norme, soit de classes POD. Une classe POD ne contient pas de constructeurs, n'a pas de données non statiques privées et ces dernières doivent être des POD ou des types scalaires au sens de la norme.

29. Seules les contraintes pesant sur les classes pouvant être utilisées dans les noyaux CUDA sont spécifiées dans l'appendice D du *CUDA programming guide* [133]

30. Les POD peuvent contenir des données statiques. C'est une des différences entre les contraintes POD et les contraintes CUDA.

```
class Vector2D {
  public :
    // initialize() remplace le constructeur
    // alloue l'espace memoire pour tous les vecteurs 1D
    inline void initialize(int nbBlocks, int sizeOfBlocks);
    // finalize() remplace le destructeur
    // desalloue la memoire
    inline void finalize();
    // copy() remplace le constructeur par recopie
    inline void copy(const Vector2D & rhs);
    // operator[] cree a la volee une instance de Vector
    // cette instance dispose d'un espace memoire propre fonction de son indice
    INLINE Vector operator[](int block);
    INLINE int nbBlocks() const;
};
```

Les classes `BandSymmetricMatrix` et `BlocDiagonalMatrix_BSM` représentant respectivement les blocs A_{ii} et la matrice $\mathbf{A}$ sont définies selon les mêmes principes.

```
class BandSymmetricMatrix{
  public:
    typedef RealType; // le type de reel utilise, generalement float

    INLINE BandSymmetricMatrix(int BlockSize, int hbw,
                               int blockId, int nbBlocks,
                               RealType * data); // un constructeur
    INLINE int size() const;          // la taille du bloc
    INLINE int hbw() const;           // la demi largeur de bande
    ...
    // deux accesseurs pour lire et ecrire les donnees
    INLINE const RealType & operator()(int i, int j) const;
    INLINE RealType & operator()(int i, int j);
};

class BlocDiagonalMatrix_BSM {
  public :
    // initialize remplace le constructeur
    inline void initialize(int nbBlocks, int sizeOfBlocks, int hbw);
    // finalize remplace le destructeur
    inline void finalize();
    // copy remplace le constructeur par recopie
    inline void copy(const BlocDiagonalMatrix_BSM & rhs);
    // operator[] cree a la volee une instance de BandSymmetricMatrix
    // cette instance dispose d'un espace memoire propre fonction de son indice
    INLINE BandSymmetricMatrix operator[](int block);
    INLINE int nbBlocks() const;
    ...
};
```

Nous pouvons utiliser ce jeu de classes pour implémenter la fonction effectuant la descente remontée en place (c.f. **Alg.** 2.2). Cette fonction prend en argument un bloc de matrice A^d_{ii} préalablement factorisée et un bloc de vecteur X_i préalablement initialisé pour être égal à B_i. En sortie de la fonction, X_i contient la solution du système $A^d_{ii}X_i = B_i$.

```
INLINE void solveProblemBlock(
    const BandSymmetricMatrix & A_factorized
  , Vector & X)
{
  const int hbw=A_factorized.hbw();
```

```
6   const int nrows=A_factorized.size();
7   /******* ETAPE1 : L X = X *********/
8   // A_factorized(i,j) = L_ij
9   for (int i=0 ; i < hbw-1 ; i++){
10    for (int j=0 ; j < i ; j++){
11      X[i]-=A_factorized(i,j)*X[j];
12    }
13  }
14  for (int i=hbw-1 ; i < nrows ; i++){
15    for (int j=i-hbw+1 ; j < i ; j++){
16      X[i]-=A_factorized(i,j)*X[j];
17    }
18  }
19  /******* ETAPE2 : D X = X *********/
20  // A_factorized(i,i) = D_ii
21  for (int i=0 ; i < nrows ; i++){
22    //ON STOCKE L INVERSE DE D DANS L(I,I)
23    X[i]*=A_factorized(i,i);
24  }
25  /******* ETAPE3 : Lt X = X *********/
26  // A_factorized(i,j) = L^T_ji
27  for (int i=nrows-1 ; i >= hbw-1 ; i--){
28    for (int j=i-hbw+1 ; j < i ; j++){
29      X[j]-=A_factorized(i,j)*X[i];
30    }
31  }
32  for (int i=hbw-2 ; i > -1 ; i--){
33    for (int j=0 ; j < i ; j++){
34      X[j]-=A_factorized(i,j)*X[i];
35    }
36  }
37 }
```

Cette fonction ne laisse apparaître aucune particularité liée à l'architecture matérielle : ni fonction intrinsèque ni objet CUDA. Nous montrerons dans la suite comment implémenter les classes de vecteurs et de matrices permettant de tirer profit du niveau SIMD/SIMT des processeurs. Notons que dans l'implémentation utilisée pour effectuer nos mesures, cette fonction commence par une série de tests. Lorsque `hbw` est inférieur à seize, des implémentations spécialisées sont appelées. Toutes les boucles sur `j` (lignes 10, 15, 28 et 33) de ces implémentations sont déroulées afin d'en éviter le surcoût. Lorsque `hbw` est supérieur à seize, cette optimisation n'apporte plus de gain de performances.

2.2.2.2 Différents formats de stockage : entrelacement des deux niveaux de la structure

Avant de nous focaliser sur les implémentations optimisées de ces classes, nous allons nous pencher le temps de cette section sur les formats de stockage des données.

Les données de la matrice **A** et du vecteur **X** peuvent être représentées sous la forme d'un tableau à deux dimensions tel que représenté sur la *figure* 2.8 : chaque ligne de couleur contient alors les données d'un bloc A_{ii} ou X_i. Nous avons vu dans la section 1.2.3.1 qu'il existe différents formats de stockage pour les tableaux bidimensionnels : par ligne, par colonne, selon un ordre de Morton, etc. La *figure* 2.9 représente différents formats de stockage pour le tableau de la *figure* 2.8. Pour chaque sous-figure, nous avons représenté l'ordre de parcours des éléments dans le tableau et représenté l'espace mémoire occupé par ce tableau. Nous avons représenté

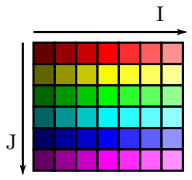


Figure 2.8 : *Représentation d'une structure de données bidimensionnelle.*

différents formats de stockage entrelaçant les lignes et les colonnes. La *figure* 2.9(a) représente le cas où il n'y a pas d'entrelacement : les données sont stockées lignes par lignes. À l'opposé, la *figure* 2.9(e) représente le cas où l'entrelacement est complet : les données sont stockées colonne par colonne. Entre les deux, les figures 2.9(b) à 2.9(d) représentent des situations intermédiaires. Nous appellerons facteur d'entrelacement le nombre de lignes dont les éléments sont entrelacés.

L'entrelacement de facteur quatre (ou entrelacement quatre par quatre) représenté sur la *figure* 2.9(d) est particulièrement intéressant. En effet, dans ce cas, il n'est pas possible d'entrelacer parfaitement les lignes et les colonnes car le nombre de lignes n'est pas multiple de quatre. Dans ce cas, il convient d'ajouter des élément de remplissage (*padding*), représentés en rouge dans le parcours. Cela implique qu'une partie de l'espace mémoire utilisé est perdue (les espaces blancs). Un facteur d'entrelacement trop important peut donc avoir un coût élevé sur la consommation mémoire d'une application.

Nous souhaitons vectoriser la descente-remontée du problème $\mathbf{AX} = \mathbf{B}$ sur une unité SIMD à deux voies. Les sous problèmes $A_{ii}X_i = B_i$ étant indépendants, une solution serait que chaque voie de l'unité SIMD effectue la descente-remontée d'un bloc. L'utilisation des unités SIMD suppose de travailler avec des « paquets » de données scalaires. Prenons l'exemple de la ligne 11 (page précédente) :

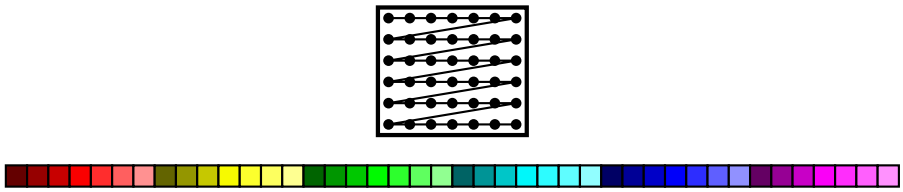
```
x[i]-=A_factorized(i,j)*X[j];
```

Afin de pouvoir effectuer simultanément cette opération sur les deux voies de l'unité SIMD, il faut que l'élément `x[i]` appartenant à un vecteur bloc X_{2k} et celui appartenant au vecteur bloc X_{2k+1} soient adjacents en mémoire. De même, les éléments `A_factorized(i,j)` des bloc $2k$ et $2k+1$ doivent être stockés de manière contiguë. Ce qui suppose que le facteur d'entrelacement soit un multiple de deux.

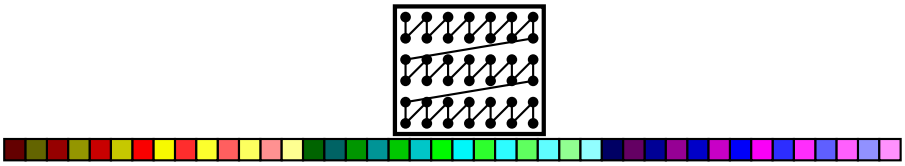
Nous allons maintenant proposer des implémentations parallèles des classes de vecteurs et de matrices. Nous ferons varier le facteur d'entrelacement de ces différentes implémentations et mesurerons l'impact de ce paramètre sur les performances. Nous constaterons alors que pour chaque architecture, un facteur d'entrelacement optimal permet d'obtenir les meilleures performances.

2.2.2.3 Implémentation pour processeurs X86_64

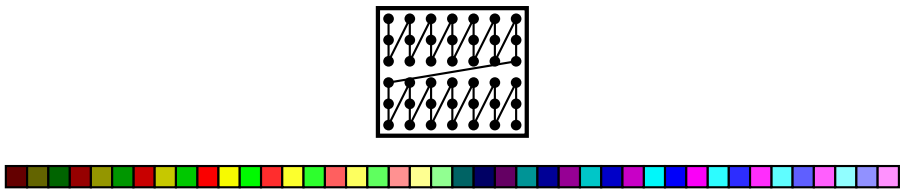
Dans cette section, nous allons détailler l'implémentation de la version X86_64 des classes de vecteurs et de matrices. Comme nous l'avons vu dans la section 2.1.3, deux niveaux de parallélisme sont disponibles dans ces processeurs. Le niveau le plus bas correspond aux unités SIMD tandis que le niveau le plus haut correspond aux différents cœurs disponibles. Nous nous intéresserons dans un premier temps à l'utilisation des unités SIMD. Nous verrons alors comment il est possible de masquer l'utilisation des fonctions intrinsèques SSE en utilisant des classes adéquates. Dans un second temps, nous nous intéresserons à l'exploitation du parallélisme multicœur.



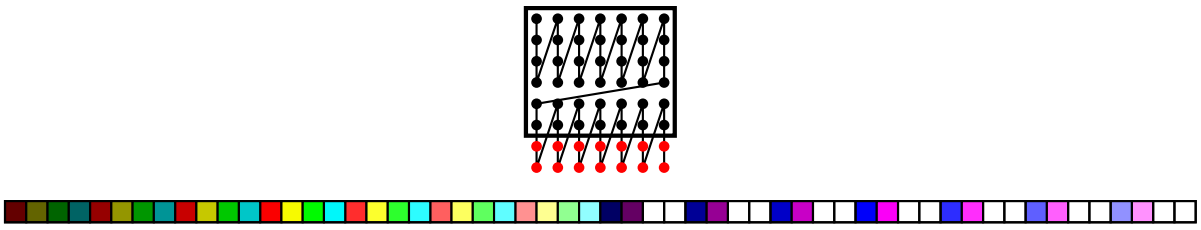
(a) Entrelacement de facteur 1 : stockage par ligne



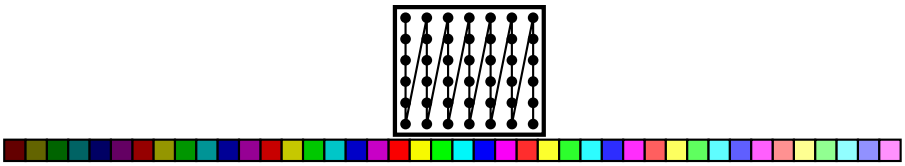
(b) Entrelacement de facteur 2



(c) Entrelacement de facteur 3



(d) Entrelacement de facteur 4



(e) Entrelacement complet : stockage colonne par colonne

Figure 2.9 : *Différents formats de stockage pour une structure de données rectangulaire.*

Parallélisation SIMD L'utilisation de classes C++ permet d'alléger l'usage des fonctions intrinsèques. Il s'agit pour cela de définir des types adaptés permettant par exemple de masquer les fonctions intrinsèques dans les opérateurs de la classe. Différentes implémentations de telles classes C++ existent [150, 151]. Nous allons ici simplement en rappeler le principe et montrer l'impact que l'utilisation de telles classes peut avoir sur l'implémentation de notre problème.

La classe `SSEData` définie ci-après représente un paquet de quatre valeurs flottantes simple précision (`float`) dont les opérateurs sont surchargés pour utiliser les unités SSE.

```
struct SSEData{
  // le type __m128 possede la meme representation que float[4]
  union{
  __m128 data_; // type « paquet » defini pour les unites SSE
  float f_[4];  // type equivalent pour l'acces aux scalaires
  };
  ...
  inline const SSEData & operator+= (const SSEData & b){
    return *this = _mm_add_ps(data_, b.data_);
  }
  ...
};
//d'autres surcharge d'operateurs
inline SSEData operator*(const float & a, const SSEData & b);
...
```

L'union anonyme définie entre les lignes 3 et 6 représente le paquet de données scalaires. Le type de données `__m128` est utilisé par les fonctions intrinsèques et sa définition varie selon les compilateurs. Afin d'accéder aux différents éléments scalaires, nous devons réinterpréter ce type comme un tableau de quatre nombres flottants (`float f_[4]`). C'est ce qui nous permet par exemple d'initialiser les matrices et vecteurs.

Afin de vectoriser la descente-remontée (cf. **Alg.** 2.2) et de traiter plusieurs blocs simultanément, nous allons définir le type de réel (`RealType`) comme étant `SSEData` :

```
1  // code defini selon l'architecture ciblee
2  inline int interleaving(int I, int i, int J, int j){
3    return i*J+j; // pas d'entrelacement
4    // return ((i/N)*J+j)*N+i%N; // entrelacement de facteur N
5    // return j*I+i; // entrelacement complet
6  }
7
8
9  class BandSymmetricMatrix{
10   public:
11     // RealType correspond a SSEData afin
12     //    de beneficier de la surcharge des operateurs
13     typedef SSEData RealType;
14
15     // le troisieme argument (data) est l'adresse de la zone
16     //    memoire deja allouee par BlocDiagonalMatrix_BSM
17     inline BandSymmetricMatrix( int size, int hbw, RealType * data,
18                                 int nbBlocks, int block
19                               );
20     ... // definition de hbw_, block_, size_, data_ et nbBlocks_
21     inline RealType & operator()(int i, int j){
22       // conversion des indices 2D (i,j) en un indice 1D I
23       const int I = i*(hbw_)+i-j;
24       const int J = block_;
```

```
25        // interleaving est utilisee pour changer le facteur d'entrelacement
26        const int idx = interleaving(I, size_*hbw_, J, nbBlocks_);
27        return data_[idx];
28      }
29      ...
30  };
```

Les nombres `I` et `J` lignes 23 et 24 représentent la position de l'élément dans la représentation 2D définie dans la section précédente et représentée sur la *figure* 2.8.

Lorsque l'on accède à un élément $A_{ii_{kl}}$ de matrice, l'appel `A_ii(k,l)` renvoie un objet de type `SSEData`. Ce type comporte, comme nous l'avons vu, quatre éléments scalaires. Ces quatre éléments correspondent respectivement à $A_{4i,4i_{kl}}$, $A_{4i+1,4i+1_{kl}}$, $A_{4i+2,4i+2_{kl}}$ et $A_{4i+3,4i+3_{kl}}$. De ce fait, une instance de la classe `BandSymmetricMatrix` ne représente pas une *vue* sur le bloc A_{ii}, mais une *vue* sur les quatre blocs $A_{4i,4i}$ à $A_{4i+3,4i+3}$.

Parallélisation multicœur La parallélisation sur différents cœurs de la descente-remontée est triviale. Comme dans l'exemple précédent, il suffit de paralléliser la boucle `for` itérant sur les différents blocs A_{ii} et X_i. L'ajout d'une directive OpenMP `#pragma parallel for` permet donc de paralléliser cette opération. Il ne faut cependant pas oublier de modifier les bornes de la boucle : les problèmes sont maintenant résolus quatre par quatre.

```
BlocDiagonalMatrix_BSM Afacto;
Vector2D X;
...
#pragma omp parallel for
for(int block = 0 ; block < Afacto.nbBlocks()/4 ; ++block ){
  solveProblemBlock(Afacto.getBlock(block), X.getBlock(block));
}
```

Nous venons de présenter une implémentation parallèle de la descente-remontée adaptée aux processeurs multicœurs X86_64. Nous allons maintenant présenter la parallélisation de cette opération sur GPU à l'aide de CUDA.

2.2.2.4 Implémentation pour GPUs

Dans cette section, nous allons décrire la parallélisation sur GPU de la résolution de l'équation $\mathbf{AX} = \mathbf{B}$. Comme nous l'avons vu section 2.1.4.2, le GPU possède deux niveaux de parallélisme : un premier niveau de parallélisme SIMT et un second niveau de parallélisme multiSM.

Cependant, sur cet exemple, les deux niveaux de parallélisme se confondent. En effet, la programmation d'unités SIMT ne nécessite pas une vectorisation explicite des opérations. En effet, cette étape est faite matériellement : si tous les *threads* d'un *warp* doivent exécuter la même opération, alors cette opération est automatiquement vectorisée. Dans le cas contraire, l'exécution des différentes instructions demandées par les différents *threads* est sérialisée.

L'approche retenue consiste à faire en sorte que chaque *thread* effectue une descente-remontée sur un problème $A_{ii}X_i = B_i$. L'implémentation de la classe de matrices (`BandSymmetricMatrix`) est semblable à celle adoptée pour les processeurs X86_64 (page précédente) : les deux seules différences sont que dans cette implémentation, `RealType` correspond au type `float` et la fonction `interleaving` correspond à l'entrelacement complet. Nous ne représenterons donc pas cette implémentation.

Nous nous intéressons en revanche à l'étape de parallélisation multi*threads* devant exploiter à la fois les différents SM et les différents cœurs de chaque SM. Comme pour le CPU, nous devons paralléliser une simple boucle `for`. Pour ce faire, nous devons écrire un noyau de calcul qui sera exécuté de manière concurrente par les différents *threads* lancés.

```
__global__ void solveProblemBlock_Kernel(BlocDiagonalMatrix_BSM Afacto,
                                         Vector2D x)
{
  const int nbThreads = blockDim.x*gridDim.x;
  const int begin = threadIdx.x + blockIdx.x*blockDim.x;
  const int end = Afacto.nbBlocks();
  for (int i = begin ; i < end ; i+=nbThreads){
    solveProblemBlock(Afacto.getBlock(block), x.getBlock(block));
  }
}
```

Finalement, l'appel à ce noyau permet d'exécuter la descente-remontée en parallèle sur le GPU. Pour cela, il faut définir la configuration de la grille des blocs puis les passer en argument CUDA (entre triple chevrons <<< >>>) au noyau de calcul.

```
BlocDiagonalMatrix_BSM Afacto;
Vector2D X;
...
const int nbElts = Afacto.nbBlocks();
const int bx = 256; // nombre de threads par bloc
// nombre de blocs theorique
int gx = (nbElts+bx-1)/bx;  //=nbElts/bx arrondi au superieur
while (gx > 65535) gx /= 2; // le nombre de blocs est limite a 65535
const dim3 Db(bx,1,1);
const dim3 Dg(gx, 1,1);
solveProblemBlock_Kernel<<< Dg, Db >>>(Afacto, x);
```

Dans les sections précédentes, nous avons exposé un problème d'algèbre linéaire dont l'optimisation est plus intrusive que pour les opérations vectorielles présentées section 2.1. Nous avons décrit deux implémentations optimisées de la résolution de ce problème : une pour CPU X86_64 et l'autre pour GPU GF100. Nous allons dans les deux prochaines sections introduire les différentes configurations matérielles dont nous disposons puis les performances obtenues sur ces différentes architectures. Nous déduirons de cette étude le format de stockage des données optimal pour chaque architecture.

2.3 La plate-forme de tests : les configurations matérielles

Dans la section précédente, nous avons présenté la parallélisation de l'opération de descente-remontée pour les CPUs X86_64 et pour les GPUs GF100. Dans cette section, nous allons introduire les différentes configurations matérielles que nous utiliserons pour mesurer les performances de ces implémentations.

Le *tableau* 2.2 montre les caractéristiques techniques des CPUs utilisés pour effectuer nos mesures. La nomenclature adoptée pour nommer les machines indique les différents niveaux de parallélisme disponible en simple précision. Par exemple, ${}_{2\times4}^{32}$Nehalem se lit : machine disposant de deux processeurs d'architecture Nehalem comportant quatre cœurs chacun. L'architecture Westmere dispose d'unités SSE permettant d'appliquer une instruction sur quatre jeux de données différents. Le degré de parallélisme disponible (multicœur et SIMD) sur cette machine est donc de 32 ($2 \times 4 \times 4$). Outre les performances théoriques, nous avons indiqué les meilleures performances mesurées pour chacune de ces machines. Les puissance de calcul (GFlops) s'entendent en simple précision et ont été mesurées en effectuant différents produits de matrices fournis par la bibliothèque INTEL MKL. Les débits de données (GB/s) ont été mesurés avec un test de performances inspiré de STREAM [61, 152]. STREAM est un test de performances de bande passante destiné à mesurer le débit de données entre le processeur et la mémoire système.

Tableau 2.2 : *Caractéristiques matérielles des machines de tests CPU.*

Nom	Processeur	SIMD	Mémoire	Cœurs	GFlops		Go/s	
		Date		utilisés	th.	obs.	th.	obs.
${}^{32}_{1\times4}$SandyBridge	Core i5-2500 3.30 GHz	AVX	2×2 Go	1	49.2	47.0	19.9	16.9
		01/2011	DDR3 1333	4	196.7	184.2	19.9	17.8
${}^{32}_{2\times4}$Nehalem	2×Xeon E5504 2.00 GHz	SSE 4.2	6×2 Go	1	14.9	14.3	23.8	9.2
		03/2009	DDR3 1066	8	119.2	103.5	47.6	21.9

Tableau 2.3 : *Caractéristiques matérielles des machines de tests GPU.*

Nom	Processeur	Date de sortie	Mémoire	GFlops		Go/s	
				th.	obs.	th.	obs.
Fermi	Tesla C2050	04/2010	4 Go	959,6	608,4	134,1	90,0

Développé par John D. McCalpin, STREAM utilise trois opérations mettant en œuvre deux ou trois vecteurs de données. Nous avons étendu ce test en ajoutant des opérations comportant jusqu'à huit vecteurs. Notons que selon la configuration matérielle, ce n'est pas la même opération qui offre le meilleur débit de données.

Notre machine de référence, utilisée pour la grande majorité des mesures, est la machine ${}^{32}_{2\times4}$Nehalem. Celle-ci comporte deux processeurs quadri-cœurs INTEL Xeon E5504 de type Nehalem EP [153, 154, 155]. Il s'agit de la configuration par défaut des postes scientifiques déployés à EDF R&D en 2010 et 2011. La machine ${}^{32}_{1\times4}$SandyBridge comporte un processeur quadri-cœurs INTEL Core i5-2500 de génération Sandy-Bridge [156, 154, 155]. Il s'agit de notre seule configuration disposant d'unité AVX.

Nous avons testé différents compilateurs pour effectuer nos mesures (INTEL icpc 12.0.2, clang 2.9, g++ 4.3 à 4.6). Le compilateur offrant les meilleures performances sur la majorité de nos cas tests est g++ 4.6. Nous utiliserons donc ce compilateur pour toutes nos mesures.

Le *tableau* 2.3 décrit les caractéristiques de la carte GPU NVIDIA TESLA C2050 que nous avons utilisée. La puissance de calcul a été mesurée à l'aide de la bibliothèque CUBLAS [157], l'implémentation des BLAS fournie par NVIDIA. Le débit de données maximal a été mesuré à l'aide du test *bandwidthTest* du kit de développement logiciel CUDA. Nous utilisons la version 4.0 du compilateur CUDA nvcc.

2.4 Analyse des performances : à chaque architecture sa structure de données

Cette section présente, pour différents facteurs d'entrelacements, les performances atteintes par les différentes implémentations.

Nous avons effectué nos mesures de performances en faisant varier quatre paramètres :

- le nombre de blocs de la matrice **A**,
- la taille de ces blocs,
- la demi largeur de bande de ces blocs (`hbw`),
- le facteur d'entrelacement.

Les trois premiers paramètres caractérisent la matrice **A**, tandis que le dernier paramètre caractérise le format de stockage de **A** et **X**. À l'issue de cette étude, nous pourrons ainsi déterminer pour chaque architecture le format de stockage optimal pour la matrice **A** et le vecteur **X**.

Tableau 2.4 : *Paramètres utilisés pour effectuer les tests de performances.*

Paramètre	Valeurs
Nombre de blocs	10 000, 20 000, 30 000, 40 000, 50 000, 75 000, 100 000
Taille des blocs	25, 50, 100, 200, 500
`hbw`	2, 3, 4, 6, 8, 10, 15
Entrelacements	1[1], 4[1], 8, 16, 32, complet

[1] Valeurs utilisées si compatibles avec l'architecture ciblée.

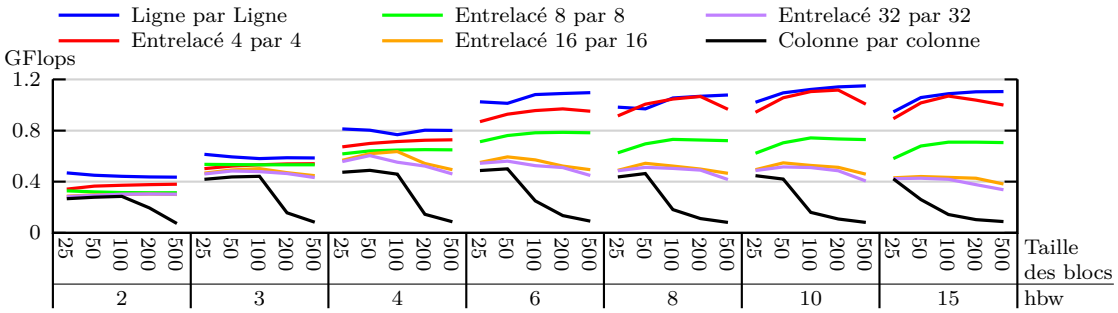


Figure 2.10 : *Influence de l'entrelacement des données sur les performances. Machine test : ${}^{32}_{2\times4}$Nehalem – exécution séquentielle CPU*

Le *tableau* 2.4 indique pour chaque paramètre l'ensemble des valeurs utilisées pour effectuer nos mesures. Plusieurs de nos machines (${}^{32}_{1\times4}$SandyBridge et Fermi) ne disposent que de 4 Go de mémoire RAM ce qui limite les combinaisons de paramètres utilisés dans nos tests de performances.

Nous nous intéresserons uniquement au temps nécessaire pour effectuer l'étape de descente-remontée et ignorons celui nécessaire à la factorisation. En effet, il est classique que la matrice factorisée soit utilisée plusieurs fois afin de résoudre des systèmes correspondant à des seconds membres $\mathbf{B}$ différents. Dans le solveur SP$_N$, le résultat de la factorisation peut être réutilisé une fois par itération, soit environ 70 fois dans les résultats présentés dans notre article [36]. Les performances seront analysées et comparées de manière détaillée dans le chapitre 6. Nous verrons alors que le nombre de blocs a relativement peu d'influence sur les performances obtenues. Pour chaque combinaison des trois autres paramètres, nous décrirons donc les performances moyennes observées quelque soit le nombre de blocs.

Les figures 2.10 à 2.15 représentent pour chaque configuration de bloc le nombre d'opérations réalisées par seconde pour différents formats de stockage.

Les figures 2.10 et 2.11 présentent les performances obtenues lors d'exécutions séquentielles sur ${}^{32}_{2\times4}$Nehalem et ${}^{32}_{1\times4}$SandyBridge. Ces performances correspondent à ce que nous pouvons attendre de machines dotées d'un processeur mono-cœur et ne disposant pas d'unités vectorielles. Dans cette configuration, le format de stockage ligne par ligne offre les meilleures performances dans presque tous les cas. Ce format de stockage parait donc le mieux adapté pour une utilisation purement séquentielle des processeurs X86_64 dotés de mémoire cache. Ce résultat n'est pas surprenant : c'est le format qui conserve le mieux la localité spatiale des données. En effet, plus le facteur d'entrelacement est petit, plus le stockage d'un bloc est compact, c'est-à-dire qu'il réduit l'espace compris entre le premier élément du bloc et le dernier. De ce fait, les données nécessaires à la descente-remontée peuvent être stockées dans les plus petits niveaux de cache. Lors de la « remontée », les données sont donc directement accessibles. Les unités de calcul

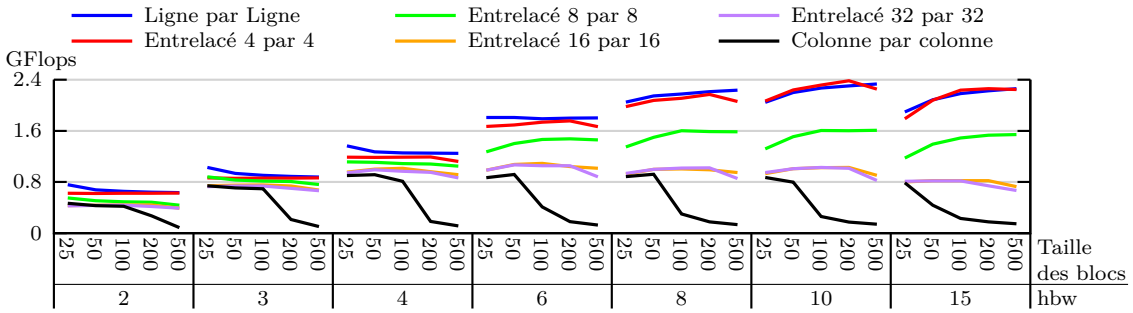


Figure 2.11 : *Influence de l'entrelacement des données sur les performances. Machine test : ${}_{1\times4}^{32}$SandyBridge – exécution séquentielle CPU*

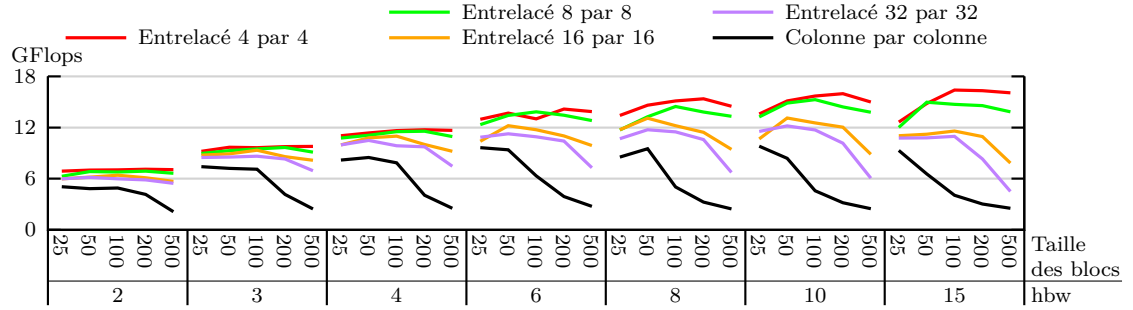


Figure 2.12 : *Influence de l'entrelacement des données sur les performances. Machine test : ${}_{2\times4}^{32}$Nehalem – exécution parallèle CPU : SSE et OpenMP*

peuvent alors commencer la remontée sans attendre que les données nécessaires retraversent les différents niveaux de cache. À l'opposé, le stockage colonne par colonne augmente la probabilité d'obtenir des défauts de cache et offre les plus mauvaises performances.

Les figures 2.12 et 2.13 présentent les performances obtenues lors d'exécutions parallèles à deux niveaux utilisant les unités SSE de tous les cœurs du processeur sur les machines ${}_{2\times4}^{32}$Nehalem et ${}_{1\times4}^{32}$SandyBridge. Comme nous l'avons vu dans la section 2.2.2.2, l'entrelacement minimal pour utiliser les unités SSE est alors de 4. Nous pouvons constater que les formats correspondant à des facteurs d'entrelacement de 4 et de 8 offrent les meilleures performances sur les différentes machines. Les performances sont en moyenne améliorées d'un facteur 17 sur ${}_{2\times4}^{32}$Nehalem et d'un facteur 7 sur ${}_{1\times4}^{32}$SandyBridge. Ce qui est inférieur au facteur d'accélération 32 attendu sur les deux machines.

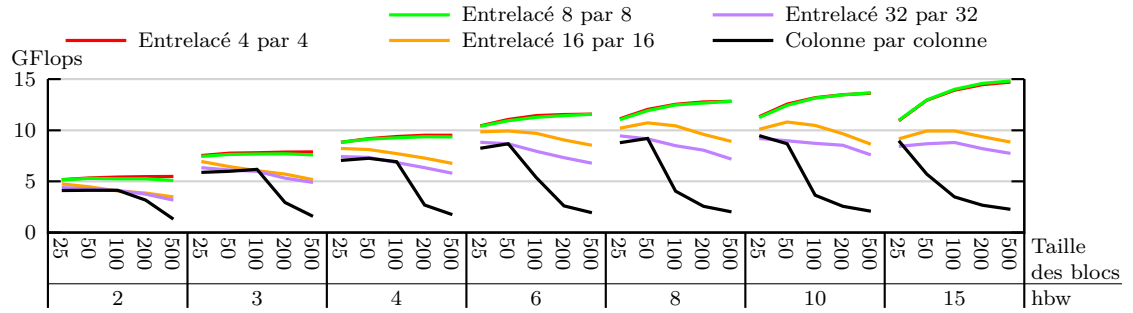


Figure 2.13 : *Influence de l'entrelacement des données sur les performances. Machine test : ${}_{1\times4}^{32}$SandyBridge – exécution parallèle CPU : SSE et OpenMP*

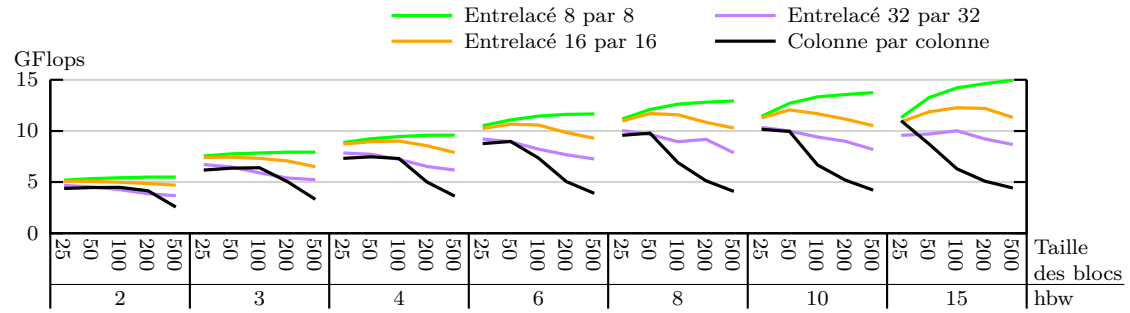


Figure 2.14 : *Influence de l'entrelacement des données sur les performances. Machine test : ${}^{32}_{1\times4}$SandyBridge – exécution parallèle : AVX et OpenMP*

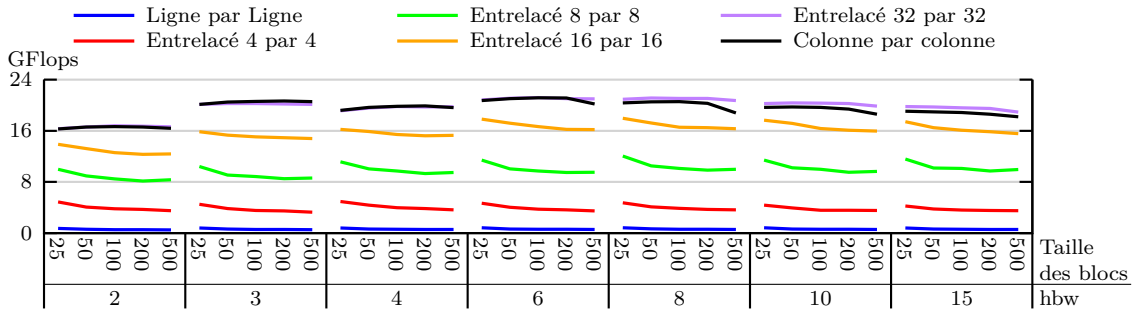


Figure 2.15 : *Influence de l'entrelacement des données sur les performances. Machine test : Fermi – exécution parallèle*

L'analyse des performances parallèles qui sera effectuée au chapitre 6 nous montrera que les performances parallèles de cette opération sont ici limitées par la vitesse de transfert des données entre le processeur et la mémoire RAM. Les formats de stockage permettant de minimiser le nombre d'accès à la mémoire et la latence de ces derniers offrent alors les meilleures performances. Le seul format de stockage qui ne permet pas à un problème $A_{ii}X_i = X_i$ de tenir dans le cache L3 et qui génère donc plus d'accès mémoire que les autres est le format de stockage colonne par colonne. Pour tous les autres, les données nécessaires à la résolution d'un problème tiennent dans ce cache. Cependant, lorsque le facteur d'entrelacement dépasse 8, les données nécessaires à la résolution d'un problème $A_{ii}X_i = X_i$ ne tiennent plus dans le cache L2. En effet, une analyse de l'exécution du code à l'aide du logiciel INTEL *VTune Amplifier XE 2011* [158] montre que si le nombre de requêtes d'accès à la mémoire RAM reste comparable, le nombre de défauts du cache L2 augmente fortement lorsque le facteur d'entrelacement dépasse 8. La latence des accès mémoire est donc augmentée, grevant ainsi les performances.

La *figure* 2.14 présente les performances obtenues sur la machine ${}^{32}_{1\times4}$SandyBridge lorsque l'on utilise les unités AVX et OpenMP. Nous pouvons observer sans surprise que les meilleures performances sont obtenues avec le facteur d'entrelacement 8 : lorsque le facteur d'entrelacement est plus grand, les données nécessaires à la résolution d'un problème $A_{ii}X_i = X_i$ ne tiennent plus dans le cache L2. L'utilisation d'instructions AVX plutôt que SSE multiplie par 2 la puissance de calculs disponible. Ces instructions n'ont cependant pas d'effet sur la bande passante disponible et ne permettent pas d'améliorer les performances des codes dont les performances sont limitées par la vitesse des accès à la mémoire RAM. Les performances obtenues avec les instructions AVX sont très proches de celles obtenues avec les instructions SSE (*figure* 2.14), ce qui confirme que la descente-remontée est limitée par la vitesse des accès à la mémoire RAM.

Tableau 2.5 : *Format de stockage optimal pour chaque type d'unité de calcul.*

Type d'unité de calcul	Format de stockage optimal
Scalaire	Ligne par ligne
SSE	Entrelacé 4 par 4
AVX	Entrelacé 8 par 8
GPU	Colonne par colonne

Enfin, la *figure* 2.15 présente les performances obtenues sur notre GPU Tesla C2050 (machine Fermi). Sur cette machine, il est possible d'utiliser tous les formats de stockage qui nous intéressent : le fonctionnement SIMT des SM n'impose pas d'entrelacement minimal pour fonctionner. Cependant, le principe SIMT implique aussi que le respect de l'entrelacement de l'unité SIMD équivalente permet de minimiser la séquentialisation de l'exécution des différents *threads*. Il est donc logique que les formats entrelacés 32 par 32 et en colonne permettent l'obtention des meilleures performances. Notre expérience sur d'autres exemples montre que selon les cas, l'un ou l'autre de ces deux formats permet l'obtention des meilleures performances. Nous privilégierons dans la suite le format de stockage colonne par colonne car il a l'avantage de ne pas nécessiter de remplissage supplémentaire des structures de données (*padding*), ce qui est important compte tenu de la relativement faible capacité de la mémoire RAM du GPU (4 Go).

Finalement, nous pourrons retenir de cette expérience que pour chaque architecture, un format de stockage particulier permet d'obtenir les meilleures performances. Le *tableau* 2.5 récapitule les formats de stockage optimaux pour les différentes architectures.

2.5 Programmation multicible : un code source unique, différents exécutables optimisés

Dans ce chapitre, nous avons observé de nombreuses différences dans l'optimisation d'une application pour un processeur X86_64 ou un GF100. Ces deux architectures exposent deux niveaux de parallélisme avec une hiérarchie semblable. Le niveau le plus haut correspond à des flots d'instructions différents (parallélisme multicœur sur CPU et parallélisme multiSM sur GPU). Le niveau le plus bas correspond à l'application d'une même instruction sur des données différentes (parallélisme SIMD sur CPU et parallélisme SIMT sur GPU). Si ces similitudes se retrouvent au niveau de l'architecture des processeurs, elles ne se retrouvent pas dans les codes optimisés correspondants : les paradigmes de programmation exposés par CUDA d'une part et le couple OpenMP/fonctions intrinsèques SSE d'autre part sont très différents. Les différences de paradigmes ne sont pas les seules difficultés à affronter pour mettre au point différentes versions optimisées d'une application : nous avons vu qu'à chaque architecture correspond un stockage optimal, résultat d'un compromis entre localité des données et bonne utilisation des unités de calcul.

La programmation multicible vise à générer, à partir d'un unique code source donné, différents exécutables optimisés pour les différentes architectures ciblées. La conception d'outil de programmation multicible nécessite d'une part la définition d'un modèle de programmation parallèle indépendant des différentes architectures ciblées et d'autre part la définition des règles de transformation du code.

Afin d'obtenir une version multicible de Legolas++ dotée de performances approchant les limites matérielles, nous devons être capables d'adapter automatiquement les structures de données ainsi que l'expression du parallélisme disponible dans les algorithmes choisis par l'utilisateur.

Pour parvenir à cet objectif, nous souhaitons ajouter une couche logicielle intermédiaire entre Legolas++ et les interfaces proposées par les constructeurs de matériel. Ceci permettra la mise au point de Legolas++ et de ses extensions sans se soucier des différences entre les architectures matérielles. En particulier, cette couche intermédiaire devra se charger d'interfacer les différents environnements de développement (OpenMP, CUDA, *etc.*) et d'adapter le format de stockage des données.

Dans le prochain chapitre, nous étudierons les différents environnements de développement multicible pour machines parallèles. Nous nous intéresserons tout particulièrement aux abstractions offertes concernant les structures de données afin de les adapter à l'architecture matérielle.

À l'issue de cette étude, nous présenterons dans le chapitre 4, la conception d'une couche logicielle prenant en charge le réordonnancement des données et le parallélisme multicible. Nous déduirons de cette étude les conditions permettant l'adaptation automatique du format de stockage des données. Nous en déduirons plus généralement les contraintes d'écriture des codes multicibles.

Enfin, le chapitre 5 présentera la conception d'un démonstrateur montrant comment adapter Legolas++ afin de mettre au point une version multicible.

La bibliographie se fait après et non avant d'aborder un sujet de recherche.

Jean Baptiste Perrin (1870 – 1942)
Prix Nobel de physique de 1926

Chapitre 3

État de l'art des environnements de développement parallèle multicible

Dans le but de trouver le meilleur compromis possible entre maintenabilité et performances des codes de calcul, une solution classique consiste à mutualiser l'optimisation vers des bibliothèques externes. Dans le cadre de la mise au point de la chaîne de calcul COCAGNE, le choix a été fait de concentrer le travail d'optimisation des solveurs d'algèbre linéaire au sein de la bibliothèque Legolas++. Afin de pouvoir exploiter efficacement les machines de calcul dont nous disposons, nous souhaitons mettre au point une version multicible de Legolas++.

Dans le chapitre 2, nous avons comparé les implémentations optimisées de deux exemples pour différentes architectures matérielles. Nous avons alors identifié deux verrous devant être levés afin de mettre au point une version multicible de Legolas++ :

1. Legolas++ doit s'appuyer sur une expression unifiée du parallélisme de l'application afin de ne pas faire apparaître les spécificités du matériel dans le code source de l'application,
2. Legolas++ doit s'appuyer sur une structure de données capable d'adapter son format de stockage à l'architecture matérielle.

Afin de mettre au point une version multicible de Legolas++, nous voulons insérer une couche logicielle chargée de lever ces deux verrous.

De nombreux travaux s'attaquent au premier verrou et visent déjà à abstraire le parallélisme et les différences entre les plates-formes matérielles. L'objet de ce chapitre est donc de comparer les différentes familles de solutions disponibles pour mettre au point des applications fonctionnant efficacement sur différentes architectures matérielles.

Sommaire

3.1 Vers un niveau d'abstraction plus élevé

L'expérience d'optimisation d'un cas d'utilisation simple de Legolas++ nous a montré que la parallélisation efficace d'une grande partie des cas d'utilisation de Legolas++ repose sur l'abstraction de deux éléments :

– l'expression du parallélisme
– l'expression du stockage des données en mémoire

La STL [73, 74] offre un bon niveau d'abstraction concernant le stockage des données. Des approches similaires couvrent différents domaines applicatifs (images, éléments d'algèbre, graphes, *etc.*). Nous nous intéresserons donc ici aux approches permettant d'assurer la portabilité d'un code parallèle sur différentes plates-formes matérielles et serons attentifs aux mécanismes proposés afin d'abstraire le stockage en mémoire.

Les *POSIX threads* (ou *Pthreads*) [159] constituent la primitive de parallélisme la plus communément disponible et donc la référence naturelle pour le parallélisme portable. Il s'agit malheureusement aussi de la primitive considérée comme la plus difficile à maîtriser à cause de la (trop grande) liberté offerte au développeur [160, 161]. Le nombre de travaux dédiés à l'aide à la détection des erreurs de programmation [162, 163, 164, 165, 166] illustre cette difficulté.

L'augmentation du niveau d'abstraction permet de masquer ces primitives au prix de la généralité. À l'opposé des Pthreads, nous pouvons trouver les langages dédiés (ou DSLs). Ces langages ne permettent pas de répondre à tous les besoins, mais il sont conçus pour répondre de manière optimale aux besoins existants dans un domaine particulier. De ce fait, le concepteur du langage peut utiliser des connaissances issues du domaine afin d'effectuer un certain nombre d'optimisations sans intervention de l'utilisateur. *Structured Query Language* (SQL), dont la première version publiée en 1974 s'appelait SEQUEL [167], est un langage extrêmement spécialisé et restreint aux seules requêtes dans les bases de données relationnelles. Cette forte spécialisation va de pair avec une forte abstraction. En effet, l'utilisateur doit uniquement décrire les données qu'il veut récupérer et non plus décrire la manière de récupérer ces données. Cette approche déclarative permet à l'environnement d'exécution (le serveur SQL) d'éventuellement paralléliser l'exécution de la requête. Finalement, un utilisateur n'ayant aucune compétence en programmation parallèle peut utiliser relativement efficacement une machine parallèle sans le savoir. En 2007, pour répondre à la généralisation des architectures parallèles dans les machines grand public, Microsoft étend le langage C# avec une extension inspirée de SQL : LINQ [168, 169]. LINQ permet d'accéder de manière uniforme à différentes sources de données, éventuellement hétérogènes (par exemple, une base de données SQL, des fichiers XML et encore des structures de données en mémoire). L'intégration de LINQ dans un langage généraliste comme le C# permet d'étendre l'expressivité des requêtes. Par exemple, il est possible d'écrire un lanceur de rayon comme une seule requête LINQ [31]. Officiellement disponible en 2010, *Parallel LINQ* (PLINQ) parallélise automatiquement les requêtes LINQ sur les processeurs multicœurs. DryadLINQ [170, 171] est un projet en cours en 2011 visant à exécuter automatiquement les commandes LINQ sur des fermes de calcul en s'appuyant sur Dryad, le framework de développement parallèle sur cluster mis au point par Microsoft [172, 173]. D'autres projets visent actuellement à intégrer cette approche dans d'autres langages [174, 175].

D'autres exemples de DSL permettent de fournir une abstraction sur le parallélisme. Ainsi, NT^2 [71, 72] est une bibliothèque C++ proposant un DSL inspiré de MATLAB et qui parallélise automatiquement le code selon l'architecture ciblée. NT^2 s'appuie pour cela sur le moteur de parallélisation SIMD EVE [150]

31. http://blogs.msdn.com/b/lukeh/archive/2007/10.aspx

Notons que l'objectif de cette thèse est exactement de répondre à cette problématique. En proposant un DSL restreint à la mise au point de solveurs pour les problèmes d'algèbre linéaires creux structurés, l'idée est bien entendu d'utiliser les propriétés de ce domaine afin de permettre une exécution la plus performante possible sur différentes architectures matérielles.

3.2 Éléments de discrimination

Notre objectif de développement de codes parallèles multicibles nous amène à considérer des critères spécifiques lors de l'évaluation d'un environement de développement et à bâtir une classification particulière. Dans cette section, nous proposons un certain nombre de critères de comparaison que nous allons utiliser dans la suite de ce chapitre.

3.2.1 Type de conception de l'environnement

Il existe différentes façons de proposer de nouvelles fonctionnalités à l'utilisateur :

la conception d'un langage *ad hoc* a l'avantage de laisser une totale liberté au concepteur de l'outil. Cependant, cela suppose de mettre au point un nouvel environnement de développement comprenant un compilateur, un débogueur, une bibliothèque standard, *etc.* ; ce qui peut être extrêmement coûteux ;

les extensions ou les annotations de langage permettent de réduire le travail de conception associé à la mise au point d'un nouveau langage. L'idée est ici d'ajouter de nouveaux mots-clés dans un langage et d'utiliser ces derniers pour définir de nouvelles fonctionnalités ou de nouvelles propriétés. Ceci demande encore la mise au point d'un compilateur, mais ce dernier pourra généralement faire appel au compilateur du langage hôte. La bibliothèque standard du langage hôte peut également être réutilisée dans un certain nombre de cas. Cela ne dispense cependant pas de mettre au point un débogueur ;

la conception d'une bibliothèque a l'avantage de ne nécessiter aucun autre développement que celui nécessaire à la mise au point de la bibliothèque. Les possibilités sont *a priori* plus limitées qu'avec la conception d'un nouveau langage puisque les règles d'utilisation dépendent du langage hôte. Cependant, certains langages permettent de mettre au point bibliothèques, *actives*, qui fournissent un langage spécialisé à un domaine enfouis dans un langage hôte (DSEL). Par exemple, la bibliothèque active NT^2 [71] fournit un DSEL extrêmement proche du langage MATLAB.

3.2.2 Niveau d'abstraction du parallélisme

Les différents outils disponibles fournissent des niveaux d'abstraction du parallélisme variable. Les *threads* fournissent par exemple un niveau d'abstraction très faible du parallélisme quand au contraire SQL ou LINQ parviennent à l'abstraire et à le masquer complètement. Nous choisissons de définir trois niveaux d'abstraction :

abstraction faible : l'outil propose des primitives de bas niveau où l'utilisateur doit spécifier les portions de code à exécuter en parallèle ainsi que les mécanismes de synchronisation (communication) à utiliser ;

abstraction moyenne : l'utilisateur spécifie les portions de code à exécuter en parallèle, les éléments de synchronisation sont automatiquement mis en place par l'outil ;

abstraction élevée : l'utilisateur ne précise rien, la parallélisation peut automatiquement être mise en œuvre par l'outil.

3.2.3 Support des accélérateurs de calcul

Parmi les différentes architectures matérielles disponibles aujourd'hui, les accélérateurs de calcul ont un degré de parallélisme important mais assez contraint (SIMT) et une capacité mémoire limitée (en 2011, jusqu'à 6 Go pour la carte Tesla C2070). Ce type d'accélérateurs de calcul offre de très bonnes performances et est disponible sous forme de carte périphérique. Nous chercherons donc à identifier les éléments qui permettent ou empêchent de supporter les accélérateurs de calcul.

3.2.4 Adaptation du stockage à l'architecture matérielle

Différentes solutions existent pour représenter des données en mémoire. Les pointeurs permettent d'accéder directement à la mémoire tandis que les conteneurs de la STL fournissent une abstraction et des interfaces d'utilisation indépendantes du stockage effectif en mémoire. Nous définissons trois niveaux d'adaptation :

nul : l'outil ne permet pas l'adaptation du stockage sans intervention de l'utilisateur, par exemple parce que les tableaux sont manipulés directement par pointeurs ;

partiel ou théorique : rien dans la conception de l'outil n'empêche d'adapter le stockage à l'architecture cible. Cette fonctionnalité n'a cependant pas été implémentée ;

complet : l'outil adapte automatiquement le stockage selon l'architecture cible.

À notre connaissance, aucune approche ne permet en 2012 d'adapter automatiquement le stockage selon l'architecture cible. Les travaux les plus avancés prennent en charge le partitionnement et la distribution de donnée pour les architectures à mémoire distribuée.

3.3 Différentes approches pour permettre le développement d'applications parallèles multicibles

3.3.1 Les approches basées sur la programmation événementielle

Utiliser la programmation événementielle consiste à écrire un programme en définissant les réactions de ses *composants* aux différents événements qui peuvent se produire [176]. Dans le cas qui nous intéresse, les évènements sont des messages envoyés par les *composants*. L'application est donc composée de deux types d'éléments : l'environnement d'exécution qui a en charge la transmission des messages et les *composants* définis par l'utilisateur qui sont chargés de traiter ces messages et d'en générer de nouveaux.

D'une certaine manière, cette approche correspond à ce qu'auraient été les *threads* s'ils avaient été conçus pour la programmation orienté objet : à un *composant* correspondrait un *thread*. L'état interne est entièrement encapsulé (aucune donnée n'est partagée) et l'interface de communication entre les composants se fait par le biais de messages. Le type et le contenu des messages définissent le traitement qui suit leur réception. L'environnement d'exécution doit donc prendre en charge le transfert des messages ainsi que l'ordonnancement de l'exécution des différents *composants*. En effet, la taille des *composants* pouvant être très petite, l'environnement d'exécution contient généralement un ordonnanceur chargé d'affecter les *composants* (*threads* logiques) ayant des opérations en attente aux *threads* système disponibles.

Le passage des messages et l'ordonnancement de l'exécution étant à la charge de l'environnement d'exécution, l'utilisateur n'a besoin de connaître l'architecture matérielle sous-jacente que pour des raisons de performances.

Erlang [177, 178] est un langage fonctionnel mis au point par la société Ericson en 1987 afin de faciliter la mise au point des applications temps-réel dans le monde des télécommunications. Il supporte les environnements à mémoire partagée ou distribuée et s'exécute sur les nombreuses architectures matérielles disponibles dans ce contexte. Cependant, il ne s'exécute pas sur les accélérateurs de calcul habituels (Cell, GPU). En effet, la mémoire disponible pour le code des SPE dans le Cell est trop petite pour héberger un environnement d'exécution Erlang tandis que l'architecture SIMD des GPU ne permet pas une bonne utilisation de ceux-ci avec Erlang.

Charm++ [179, 180] est une extension de C++ disponible depuis 1993 et permettant de développer une application parallèle basée sur l'envoi de messages asynchrones. Avec Charm++, l'utilisateur définit des composants (appelés *chares*) dont les fonctions sont activées par l'arrivée de messages envoyés par d'autres *chares*. Charm++ permet de cibler les architectures en mémoire partagée et en mémoire distribuée. Charm++ permet aussi de cibler le processeur Cell [181, 182] ou d'encapsuler des *chares* s'exécutant sur GPU [183]. Dans ce dernier cas, l'utilisateur doit fournir les noyaux de calcul optimisés pour le GPU.

Finalement, cette approche permet à l'utilisateur de décrire le graphe de dépendances entre les tâches : à chaque message correspond un arc du graphe. Ce graphe peut ensuite être utilisé pour obtenir une exécution parallèle de l'application.

3.3.2 Les approches basées sur les patrons parallèles

L'objectif des approches basées sur les patrons parallèles est comme dans le cas de la programmation événementielle d'aboutir à une description du graphe de dépendance entre les tâches. Ces approches se distinguent cependant par la volonté d'abstraire la description de ce graphe. Cette abstraction part de l'identification des schémas récurrents dans ces graphes. Ainsi, il n'est pas rare qu'un nœud du graphe génère plusieurs nœuds correspondant à des tâches indépendantes et qu'un dernier nœud dépende de la fin de toutes ces tâches indépendantes. Un exemple caractéristique conduisant à ce type de graphe est une boucle *for* dont les itérations sont indépendantes.

3.3.2.1 Différentes écoles concurrentes conduisent aux mêmes patrons

En 1988, Murray Cole propose d'abstraire ces schémas par des fonctions d'ordre supérieur (fonctions de fonctions) et de les fournir en tant qu'outils pour le développeur [184]. De cette manière, l'utilisateur peut se concentrer sur son application et fournir ce qui va aller autour du squelette : l'outil prend en charge la parallélisation. Cole a pour objectifs de :

- simplifier la programmation parallèle en augmentant le niveau d'abstraction ;
- améliorer la portabilité et la réutilisabilité des codes parallèles en épargnant au développeur l'écriture des détails de mise en œuvre des schémas parallèles ;
- améliorer les performances des codes parallèles en fournissant des implémentations optimisées pour l'architecture ciblée ;
- fournir la possibilité de mettre en place des optimisations statiques et dynamiques qui reposent sur une description de haut niveau de la structure algorithmique.

De nombreux travaux ont fait suite à ceux de Cole, une revue récente et assez approfondie du domaine est disponible dans [185].

Parallèlement à l'émergence de cette communauté scientifique, celle des patrons de conception [186] (*design Patterns*) s'est formée avec des objectifs assez proches :

- fournir de bonnes solutions génériques pour des problèmes récurrents ;
- établir une terminologie commune pour faciliter la communication entre les personnes autour de problèmes récurrents ;

– rehausser le niveau de réflexion des développeurs pour des problèmes récurrents ;
– faciliter l'apprentissage des bonnes pratiques pour des problèmes récurrents.

Aux alentours des années 2000, cette communauté a commencé à s'intéresser au problème récurent du parallélisme [187, 188] et a abouti à des outils ressemblant beaucoup pour l'utilisateur aux squelettes algorithmiques : les fonctions d'ordre supérieur (fonctions de fonctions) sont remplacées par des patrons de classes (classes de classes).

Ces deux approches font partie de ce que l'on nomme la programmation parallèle structurée [189, 185] par analogie à la distinction faite par Dijkstra entre la programmation structurée et l'utilisation de `goto` [190]. La programmation parallèle structurée apporte un niveau d'abstraction plus élevé [191] et supporte au moins en théorie le parallélisme imbriqué.

Le problème du parallélisme imbriqué met au jour la différence la plus importante entre les squelettes et les patrons parallèles. L'approche « puriste » par squelettes algorithmiques vise à construire une représentation globale de la structure algorithmique de l'application afin de pouvoir utiliser cette structure pour effectuer des optimisations de haut niveau. L'approche par patrons de conception parallèle cherche au contraire à rendre ces problèmes et les différents niveaux de parallélisme indépendants les uns des autres afin de faciliter la mise au point de modules facilement réutilisables [185].

Nous allons maintenant présenter les caractéristiques de quelques outils disponibles afin de comparer les bénéfices des deux approches.

3.3.2.2 Exemples de squelettes algorithmiques

Comme nous l'avons vu précédemment, un environnement de programmation par squelettes algorithmiques fournit un ensemble de fonctions d'ordres supérieurs correspondant aux différents squelettes proposés. L'utilisateur doit alors définir les éléments correspondant à son application particulière. *MapReduce* [192, 193], un environnement de squelette algorithmique mis au point par Google, en est un exemple simple : cet environnement propose à l'utilisateur de définir :

1. le calcul à effectuer sur chaque sous-ensemble de données ;
2. la manière de rassembler les données.

Le cas des squelettes algorithmiques est trop compliqué pour être traité par une bibliothèque procédurale classique. Encore une fois, c'est l'intégration des concepts dans le langage qui permet de proposer des environnements de programmation et d'exécution. Ces langages peuvent être créés de manière *had oc* avec leur environnement de programmation comme par exemple Skil [194] ou P^3L [195]. Ces derniers se veulent être des langages de coordination entre des noyaux de calcul écrits en C. Ils permettent ainsi à l'utilisateur d'exprimer le parallélisme d'une application sous la forme d'un graphe de modules. Les squelettes algorithmiques élémentaires correspondent alors à des graphes simples. Eden [196] propose de coordonner des noyaux écrits en Haskell. Dans ces trois cas (Skil, P^3L ou Eden), un environnement d'exécution est chargé de la mise en œuvre du parallélisme et des communications. Quaff [188] est une bibliothèque C++ fournissant un DSEL de coordination de calculs. Les squelettes algorithmiques élémentaires forment les mots de ce DSEL dont la particularité réside dans le fait qu'il est résolu à la compilation, supprimant ainsi le besoin d'environnement d'exécution.

3.3.2.3 Exemples de mise en œuvre de patrons de conception parallèles

Dans la communauté du développement parallèle en C++, de nombreux projets visent à proposer une version parallèle de la STL. Ces projets fournissent des bibliothèques proposant des conteneurs et des algorithmes parallèles. L'objectif est de permettre à un utilisateur de la

STL de paralléliser son code simplement en échangeant ses algorithmes et ses conteneurs avec leurs équivalents parallèles.

Par exemple, STAPL [197, 198, 199] est une bibliothèque conçue en 1998 comme un sur-ensemble de la STL. Lorsque les algorithmes de STAPL sont utilisés, l'environnement d'exécution choisit automatiquement l'implémentation en fonction des performances obtenues sur des cas précédents [200]. Cet environnement permet aussi à STAPL de cibler les architectures à mémoire partagée et à mémoire distribuée. Outre ce niveau « utilisateur », deux autres niveaux d'utilisation de la bibliothèque permettant une plus grande liberté d'utilisation et d'optimisation sont proposés. Le niveau « développeur » permet l'extension des fonctionnalités de STAPL en ajoutant de nouveaux conteneurs et de nouveaux algorithmes tandis que le niveau « spécialiste » permet de modifier l'environnement d'exécution.

STAPL introduit les concepts de *pView* [201] et de *pRange* qui généralisent les itérateurs de la STL et permettent de manipuler les données indépendamment de leur stockage effectif. Un espace mémoire correspondant à une matrice pourra ainsi être *vu* comme une matrice par colonnes ou comme une matrice par lignes. Cependant, STAPL ne prend pas en charge le réordonnancement automatique des données en mémoire pour permettre par exemple la vectorisation automatique des algorithmes.

Mise au point par INTEL en 2006, la bibliothèque INTEL TBB [139] cible les architectures programmables avec des *threads* (processeurs X86, X86_64, PowerPC) sous différents OS. Tout comme STAPL, INTEL TBB utilise un concept d'itérateur généralisé. D'une certaine manière, la bibliothèque INTEL TBB peut être considérée comme une restriction de STAPL aux seules architectures à mémoire partagée.

D'autres projets, comme Thrust [202], s'inspirent de la STL et ciblent directement les GPU NVIDIA. SkePU [203] va plus loin et propose une solution multicible personnalisable : outre les algorithmes déjà proposés, l'utilisateur peut en définir de nouveaux qui pourront être exécutés sur CPU, sur GPU avec CUDA, ou sur une cible OpenCL [32]. Pour parvenir à ce résultat, SkePU combine l'utilisation des macros et des templates afin de générer les différentes implémentations. La principale limite de ces approches est le manque de support pour les structures de données complexes et pour les fonctions définies par l'utilisateur. La limitation aux structures de données unidimensionnelle équivalentes aux `std::vector`, permet de s'affranchir du besoin de réordonnancement exposé dans la section 2.4.

Enfin, certains travaux se concentrent sur l'abstraction du matériel au service d'un outil de plus haut niveau. Ainsi, le module TPetra de Trilinos [81] s'appuie sur une couche dédiée à la parallélisation sur CPU à l'aide d'INTEL TBB ou sur GPU avec CUDA [82]. Contrairement aux travaux précédents, cet outil est extrêmement restreint. Afin de répondre aux besoins de parallélisation de TPetra, seuls deux constructions sont nécessaires : `parallel_for` et `parallel_reduce`. Ces constructions sont également prises en charge dans la couche de parallélisation de la bibliothèque HONEI [204].

3.3.2.4 Les patrons parallèles par annotation de code source

Outre les approches précédentes pour l'expression des formes récurrentes de parallélisme, des travaux ont été fait afin de permettre l'expression des structures parallèles dans les codes séquentiels existants. L'annotation de source permettant de ne pas ou de peu modifier le code séquentiel, elle limite le degré d'intrusion dans le code existant. Elle permet aussi de limiter l'ajout d'erreurs lors de la parallélisation et permet donc un développement en plusieurs phases.

32. OpenCL est une extension du C permettant de mettre au point un code portable entre différentes cibles matérielles (cf. page 79).

Une première phase de développement séquentiel est effectuée par un spécialiste du domaine applicatif. Dans un second temps, un spécialiste du parallélisme peut mettre en place des patrons parallèles par simple annotation des sources. Notons que cette famille d'approche n'est pertinente que pour les cas où la parallélisation de l'application ne nécessite pas des modifications algorithmiques importantes.

L'outil s'appuyant sur des annotations de code source le plus connu est OpenMP [59]. Défini pour le langage FORTRAN en 1997 et pour les langages C et C++ en 1998, il permet à l'utilisateur de paralléliser des nids de boucles ou des sections complètes de code. Si OpenMP est bien adapté aux cas simples comme celui présenté au chapitre 2, différentes limitations de ce standard font que son adoption n'a pas été aussi large qu'attendue :

- la parallélisation d'un algorithme non restreint à des boucles parallèles peut être compliquée et introduire de lourds changements algorithmiques (ex : tri [205]) ;
- si OpenMP offre la portabilité du parallélisme sur différentes plates-formes à mémoire partagée, il ne permet de cibler efficacement ni les architectures à mémoire distribuée, malgré un certain nombre de tentatives [206, 207, 208, 209, 210], ni les accélérateurs de calcul, même si des travaux en cours tentent de combler ce vide [211, 212, 213] ;
- pour les architectures à mémoire distribuée, la nécessité d'utiliser MPI et donc d'exprimer deux niveaux de parallélisme a découragé une partie des utilisateurs qui ont préféré utiliser uniquement MPI. Ce choix est d'autant plus légitime qu'OpenMP ne permet pas toujours d'obtenir des performances meilleures que MPI sur une machine multicœur [214, 215].

OpenHMPP (HMPP pour *Hybrid Multicore Parallel Programming*) [216], généralisation d'OpenMP aux architecture hybrides, est développé depuis 2007 par CAPS entreprise. Ici, les boucles sont généralisées aux « codelets » : des portions de code dont les données d'entrée et de sortie sont bien identifiées. Une fois les codelets extraits, OpenHMPP permet d'exécuter ces derniers sur une architecture matérielle donnée (par exemple un GPU) après avoir automatiquement effectué les transferts de données requis. Afin de permettre l'optimisation du code, OpenHMPP incorpore un certain nombre de directives permettant de guider la génération de code pour l'architecture cible et permet en outre la modification du code généré. Cependant, le manque de standardisation et donc de visibilité concernant sa pérennité, ne favorise pour l'instant pas l'adoption d'OpenHMPP. Un processus de standardisation est alors initié [217] et conduit en novembre 2011 à l'émergence du standard OpenACC [218]. Des travaux sont actuellement en cours afin d'unifier ce standard avec OpenMP.

Au final, les solutions par annotation de sources se veulent peu intrusives. Cependant, cela n'est possible qu'au détriment d'une élévation du niveau d'abstraction pour le réordonnancement des données. En effet, ne pas s'introduire dans le code existant ne permet pas de s'affranchir du pointeur C qui correspond à la fois aux adresses mémoire et aux structures de données. Dès lors, il est impossible de masquer à l'utilisateur des modifications dans le stockage des données en mémoire.

3.3.3 Les approches basées sur la programmation par tableaux

Le principe de base de la programmation par tableaux est d'appliquer la même opération sur des collections de valeurs. Bien que l'expression soit anachronique, on pourrait *a posteriori* considérer cette approche comme un cas très particulier de patron parallèle. En effet, cette approche consiste à rendre implicite deux formes très courantes de parallélisme : le parallélisme de données (boucle `for` parallèle sur tous les éléments d'un tableau) et la réduction (boucle `for` avec accumulateur pour diminuer la dimensionnalité du tableau de sortie). C'est en pratique le seul cas où la parallélisation automatique donne de bons résultats, y compris pour les langages

fonctionnels comme Haskell où l'analyse est pourtant bien simplifiée grâce à l'absence d'effets de bord [219, 220, 221].

Ainsi, NESL [222, 223] est un langage de programmation fonctionnel conçu en 1993 pour traiter les opérations en parallèle sur des listes. Le principe consiste à fournir à NESL une fonction à appliquer sur tous les éléments d'une liste. NESL supporte les structures multidimensionnelles et plusieurs niveaux de parallélisme imbriqués. NESL permet à l'utilisateur de ne pas se soucier de la manière dont les opérations sont effectuées, mais simplement de déclarer quelles sont les opérations souhaitées. De ce fait, NESL permet une parallélisation implicite, y compris sur des unités de calcul SIMD. NESL adapte de fait les structures de données pour permettre une vectorisation efficace. Cependant, NESL n'adapte pas le stockage en fonction de l'architecture : seul le stockage vectoriel (correspondant à un entrelacement complet dans notre exemple du chapitre 2) est possible.

Héritier de NESL, *Data Parallel Haskell* (DPH) [224, 219] intègre depuis 2007 les fonctionnalités de NESL au langage Haskell sous forme de DSEL. *Accelerate* [225] est un autre DSEL de programmation par tableaux en Haskell dont le but est d'ajouter le support des accélérateurs de calcul ; aujourd'hui, seules les cartes graphiques de NVIDIA sont ciblées au travers de CUDA.

Outre les langages fonctionnels, NESL a également eu des héritiers dans les langages impératifs : Brook [226, 227] et RapidMind [228, 229] sont deux exemples d'intégration de telles fonctionnalités dans des extensions du langage C.

Entre 2002 et 2007, les GPUs n'étaient pas programmables au sens actuel du terme. Cependant, il était possible en passant par des fonctionnalités de traitement graphique d'effectuer un certain nombre d'opérations de calcul. Brook [226, 227] fut mis au point à l'université de Virginie entre 2002 et 2007 afin de permettre l'utilisation des cartes graphiques en évitant cette forme de programmation extrêmement compliquée. Brook était une extension de C permettant de prendre en compte le parallélisme de donnée. Brook ciblait le GPU en s'appuyant sur les bibliothèques graphiques DirectX9 et OpenGL. Brook a été peu utilisé en comparaison avec CUDA car son modèle de programmation était plus contraint. Brook a fini par disparaître avec l'annonce de la sortie d'OpenCL.

RapidMind [228, 229] est une évolution commerciale de Sh [230]. RapidMind permet à l'utilisateur de cibler les processeurs multicœurs, les GPU [229] et le processeur Cell [228]. RapidMind est une extension du C basée sur un mécanisme de macros et de templates C++ : le compilateur C++ est donc utilisé pour compiler le « C étendu » défini par RapidMind. Afin de pouvoir masquer le stockage des données à l'utilisateur, RapidMind introduit de nouveaux types de valeurs et de tableaux dont les données ne sont accessibles qu'à travers l'interface fournie. Le principe est relativement proche de celui des *pViews* de STAPL puisque l'utilisateur n'a aucun moyen de savoir comment les données sont effectivement stockées. Afin d'être parallélisées, les fonctions devant s'appliquer à ces nouveaux types de données subissent un certain nombre de contraintes. La plus notable de ces contraintes est l'absence d'effets de bords puisque les données qui ne sont pas définies comme valeur de retour ne peuvent être modifiées et que les données non déclarées localement ne sont pas accessibles. Ceci permet entre autre l'utilisation de la compilation à la volée par l'environnement de développement afin de s'adapter à l'architecture matérielle. Au final, si RapidMind arrive à apporter les fonctionnalités de NESL dans un langage impératif, c'est en utilisant un modèle de programmation proche de la programmation fonctionnelle.

Relativement proche de RapidMind sur les principes, INTEL Ct [231] se voulait être une plate-forme d'essai d'INTEL pour mettre au point un environnement de programmation permettant de cibler aussi bien ses processeurs que ses cartes graphiques *Larabee* [232]. Suite au rachat de RapidMind par INTEL en 2009, les deux projets ont fusionné pour donner naissance à INTEL *Array Building Blocks* (ArBB) [233]. ArBB se compose de quatre éléments :

1. une machine virtuelle abstraite ;
2. un modèle de programmation pour cette machine ;
3. la traduction de ce modèle sous la forme d'un langage parallèle ;
4. une bibliothèque implémentant la machine virtuelle et fournissant le langage parallèle sous la forme d'un DSEL C++.

Les différentes versions publiques d'ArBB ne ciblaient que les processeurs X86. Cependant, avant d'arrêter ce projet le 11 novembre 2011 [33], dans ses communication INTEL précisait qu'ArBB devait permettre de paralléliser les applications pour ses futurs processeurs massivement multi-cœurs (*Many Integrated Core*) [234]. Aujourd'hui, ce projet fait partie des projets de recherche d'Intel.

3.3.4 Les autres approches

CUDA C/C++ [133] est une extension des langages C et C++ proposée en 2007 par NVIDIA pour programmer ses cartes graphiques (GPU) répondant à l'architecture CUDA (cf section 2.1.4 pour une définition plus précise). CUDA C/C++ se compose de deux parties. Une première partie, en C standard, permet la gestion du matériel (initialisation de cartes graphiques, allocations mémoire, transferts de données, ...). La seconde partie étend les langages C et C++ avec de nouveaux mots-clés afin de définir des concepts propres à la programmation sur GPU. CUDA C/C++ permet à l'utilisateur de programmer des noyaux (*kernels*) en C ou en C++ et de les exécuter sur GPU. CUDA C/C++ est une interface d'assez bas niveau puisque l'utilisateur doit effectuer les allocations mémoire sur le GPU ainsi que les transferts de données depuis la mémoire du système hôte. Le modèle de programmation utilisé par le langage CUDA C peut en principe permettre la parallélisation sur un grand nombre d'architectures parallèles à mémoire partagée, y compris les processeurs multicœurs [235, 236, 237, 238]. C'est pourquoi l'interface de CUDA C a pu servir de base à la mise au point d'OpenCL.

OpenCL [239] est un standard qui a vu le jour en 2008 sous l'impulsion d'Apple qui souhaitait unifier les modèles de programmation pour les différentes cartes graphiques disponibles. OpenCL définit une extension du langage C ressemblant beaucoup à CUDA C/C++ proposé quelques mois plus tôt par NVIDIA. Conçu dès l'origine pour cibler les accélérateurs parallèles ainsi que les processeurs multicœurs, son interface permet de choisir à l'exécution l'architecture cible parmi celles disponibles. Cependant, si OpenCL permet d'obtenir un code portable entre les différentes architectures matérielles disponibles sur le marché, il ne garantit en rien la portabilité des performances [240]. Dans le but d'obtenir de bonnes performances, OpenCL ne résout donc pas grand chose seul. OpenCL propose néanmoins un langage et une interface commune pour tous les accélérateurs et constitue par conséquent un outil pertinent pour les outils multicibles qui génèrent du code source.

3.3.5 Synthèse

Le *tableau* 3.1 liste l'ensemble des environnements cités dans ce chapitre en rappelant leur situation par rapport aux critères sélectionnés en section 3.2.

Parmi tous ceux que nous avons étudié, seuls les outils basés sur la programmation par tableau proposent un niveau d'abstraction de parallélisme élevé. De surcroît, cette approche a l'avantage de permettre l'expression de la localité des données : en deux dimensions avec la convention C, les éléments `x[0][0]` et `x[0][1]` sont plus proches que les éléments `x[0][0]`

33. http://software.intel.com/en-us/forums/showthread.php?t=105738&o=a&s=lr

Tableau 3.1 : *Synthèse des différentes approches étudiées*

	approche	nom	type de conception	niveau abst.	support accél.	adaptation stockage
	événementielle	Erlang	langage	moyen	non	N/A
		Charm++	bibliothèque		Cell	partiel
Parallélisme structuré	squelette	P^3L	ext. de lang.	moyen	non	non
		Eden	ext. de lang.			non
		Quaff	bibliothèque		Cell	partiel
	patrons de conception	STAPL	bibliothèque	moyen	non	partiel
		INTEL TBB				non
		TPetra			oui	partiel
		HONEI				non
		Thrust				non
		CuPP				non
		SkePU				non
	annotations	OpenMP	annot. de code	moyen	non	non
		HMPP	annot. de code		oui	non
	tableaux	NESL	langage	élevé	non	théorique
		DPH	ext. de lang.			non
		INTEL Ct	ext. de lang.			non
		INTEL ArBB	bibliothèque		initialement prévu	théorique
		accelerate	bibliothèque		non	non
		RapidMind	bibliothèque			théorique
		Brook	ext. de lang.			non
	autres	CUDA C	ext. de lang.	faible	non	non
		OpenCL	ext. de lang.	faible	non	non

et `x[1][0]`. Lorsque le stockage des données est pris en charge par l'outil (c'est le cas pour Nesl, ArBB et RapidMind), il est potentiellement possible d'adapter le format de stockage des données.

L'approche qui nous semble la plus pertinente pour utiliser les accélérateurs matériels est l'approche par patrons de conception. Cette famille d'approches unifie l'expression du parallélisme et adapte le code généré pour les différentes architectures matérielles. D'autre part, l'exemple de STAPL montre qu'il est alors possible de bénéficier des avantages de la programmation par tableaux. En effet, afin de pouvoir prendre en charge l'adaptation des structures de données, STAPL s'appuie sur la notion de *pview*. C'est l'utilisation de ce concept qui permet à ArBB de prendre en charge le format de stockage des données.

3.4 Choix stratégique : positionnement de MTPS

L'objectif de notre étude était de trouver les concepts permettant de mettre au point un outil de parallélisation multicible. Cet outil doit répondre à deux exigences :

1. il doit permettre d'exprimer le parallélisme disponible dans l'application indépendamment de l'architecture matérielle ciblée ;
2. il doit prendre en charge l'adaptation du stockage des données en fonction de l'architecture, tel que nous l'avons vu au chapitre 2.

Notre volonté de supporter les accélérateurs de calcul et les unités de calcul SIMD impose de se diriger vers un outil de parallélisme structuré ou vers la programmation par tableaux. En effet, leur support nécessite de pouvoir adapter le format de stockage (cf. chapitre 2). Dans les deux cas, nous avons des exemples de stratégies pour abstraire le stockage des éléments en reprenant le concept de *vues* présent dans STAPL ou INTEL ArBB. Les deux approches sont également compatibles avec une implémentation sous la forme d'un DSEL encapsulé dans une bibliothèque active.

Dans le prochain chapitre, nous étudierons la conception de MTPS (multiTarget Parallel Skeleton), une bibliothèque que nous avons mis au point afin de répondre à ces besoins. MTPS est un outil de parallélisme structuré mettant en œuvre des patrons de conception. Comme HONEI, Thrust ou SkePU, MTPS doit permet l'utilisation d'accélérateurs de calcul. À la différence de NESL ou INTEL ArBB, MTPS ne prend en charge que les structures de données bidimensionnelles. Cette limitation nous paraît faible : la majorité des outils que nous avons étudié ne supportent que les structures à une dimensions. De plus, il est toujours possible de sérialiser une structure comportant plus de dimensions afin de se ramener à une structure bidimensionnelle En contrepartie, le format de stockage de ces structures est adapté à la cible matérielle comme nous l'avons vu au chapitre 2. Afin de masquer ces différents formats de stockage à l'utilisateur, MTPS reprend le concept de *vues* présent dans STAPL et INTEL ArBB.

À l'issue de cette étude, nous effectuerons un retour d'expérience du développement d'une bibliothèque multicible et identifierons les contraintes devant être respectées pour permettre l'implémentation multicible d'une application ou d'une bibliothèque. Le chapitre 5 présentera comment adapter Legolas++ à ces contraintes pour permettre la vectorisation automatique des problèmes implémentés. Nous verrons alors qu'il pourrait être judicieux de définir une couche intermédiaire permettant de généraliser MTPS aux structures de données comportant plus de deux niveaux.

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).

David Small (1958 – présent)
Concepteur de Spectre GCR, un émulateur MAC pour atari ST.

Chapitre 4

MTPS : MultiTarget Parallel Skeleton

Dans le premier chapitre, nous avons montré l'importance de la mutualisation de code entre les applications ou bibliothèques. Afin de mutualiser les travaux d'optimisation de plusieurs solveurs d'algèbre linéaire, une première version de Legolas++ a été mise au point. Nous souhaitons étendre Legolas++ afin d'utiliser les architectures matérielles disponibles actuellement. Pour mutualiser le développement des fonctionnalités et des algorithmes de Legolas++, nous souhaitons mettre au point une couche logicielle chargée d'abstraire l'architecture matérielle et sur laquelle pourra s'appuyer une version multicible de Legolas++.

Dans le second chapitre, nous avons présenté les architectures matérielles que nous voulons utiliser. Nous avons ensuite introduit les optimisations adaptées à ces architectures. Nous avons alors identifié deux verrous à lever pour mettre au point une application multicible exploitant efficacement ces architectures. Le premier verrou concerne l'hétérogénéité de l'expression du parallélisme. En effet, les différentes plate-formes matérielles dédiées au calcul scientifiques sont livrées avec des outils de parallélisation correspondant à des paradigmes généralement incompatibles comme CUDA et les fonctions intrinsèques SSE. La mise au point d'un code capable de s'adapter à ces paradigmes suppose donc d'abstraire l'expression du parallélisme. Cette abstraction devra être composée de constructions dont une implémentation peut être fournie pour chaque architecture. Le second verrou concerne l'expression des structures de données. Nous avons en effet montré que le format de stockage des données influe fortement sur les performances de l'application et peut même interdire l'utilisation de certaines unités de calcul.

Dans le chapitre 3, nous avons comparé plusieurs approches permettant de mettre au point des applications parallèles et portables sur différentes architectures matérielles. Si la majorité des approches étudiées permettent de lever le premier verrou, à ce jour, aucune n'adapte la structure de données selon l'architecture cible.

Nous présentons donc dans ce chapitre notre conception de MTPS [241, 242], une bibliothèque prenant en charge l'adaptation des structures de données bidimensionnelles et du parallélisme pour les CPUs X86_64 et pour les GPUs compatibles avec CUDA.

À l'issue de cette présentation, nous analyserons les diverses contraintes d'utilisation de MTPS. Dans le prochain chapitre, nous présenterons comment l'implémentation de Legolas++ a été étendue pour respecter une partie de ces contraintes.

Sommaire

4.1 Modèle de programmation : introduction aux contextes vectoriels

Dans cette section nous allons présenter le modèle de programmation de MTPS. Ce modèle doit permettre d'unifier les modèles des architecture des CPUs X86_64 et des GPUs GF100. Nous définirons également le modèle de structure de données et le modèle d'exécution associé. Dans la prochaine section, nous montrerons comment ces concepts sont manipulés pour définir une application parallèle.

4.1.1 Modèle d'architecture matérielle

Comme prérequis à la définition du modèle de programmation, nous devons définir le type de machine qu'il doit permettre de cibler. Nous avons vu au chapitre 2 que les architectures des CPUs et des GPUs comportaient de nombreuses similitudes. En effet, les deux architectures présentent chacune deux niveaux de parallélisme. Le premier correspond à des flots d'instructions distincts (parallélisme multicœur pour le CPU et parallélisme multi-SM pour le GPU) tandis que le second correspond à l'application d'un même flot d'instructions sur des jeux de données différents (parallélisme SIMD sur CPU et parallélisme SIMT sur GPU).

Si l'utilisation du premier niveau de parallélisme sur les exemples de codes applicatifs que nous avons étudié est assez peu contraint sur CPU, elle l'est d'avantage sur GPU. En particulier parce que les allocations de la mémoire GPU et les transferts de données entre la RAM CPU et la RAM GPU doivent être effectués explicitement.

Le second niveau a un impact beaucoup plus important sur le code. En effet, sur CPU comme sur GPU, le second niveau de parallélisme matériel impacte le stockage des données en mémoire. Afin de tirer profit des unités vectorielles, nous proposons dans le chapitre 2 d'utiliser une structure de données bidimensionnelle permettant un stockage entrelacé. Nous avons vu que pour chaque type d'architecture, nous pouvions définir un entrelacement optimal. Cet entrelacement peut être « nul » (cf. *figure* 2.9(a), page 59), « total » (cf. *figure* 2.9(e)) ou « entrelacé d'un facteur e » (cf. figures 2.9(b) et 2.9(c)).

L'utilisation du second niveau de parallélisme implique en outre de connaître le nombre N_d de données sur lesquelles s'applique chaque instruction. Pour une architecture X86_64, N_d correspond simplement au nombre de voies des unités SIMD tel que défini dans la section 2.1.1. L'utilisation d'un processeur X86_64 nécessite un entrelacement d'un facteur e où e est un multiple de N_d. Pour l'architecture NVIDIA GF100, dédiée au calcul scientifique, N_d correspond au nombre de *threads* fonctionnant de manière synchrone, c'est-à-dire 32, le nombre de *threads* dans un *warp*. Sur les unités SIMT, les performances maximales sont atteintes soit lorsque le facteur d'entrelacement e est un multiple de 32, soit lorsque l'entrelacement est total.

Le modèle de programmation présenté dans la suite de cette section permet de cibler l'ensemble des architectures comportant un ou plusieurs cœurs identiques. Ces cœurs peuvent contenir des unités de calcul SIMD ou SIMT opérant sur N_d données.

4.1.2 Rappel du cas d'application : résolution du système $\mathbf{AX} = \mathbf{B}$

Dans cette section, nous allons rappeler l'exemple de la section 2.2. Nous utiliserons cet exemple dans la suite pour illustrer les concepts de notre modèle.

Considérons donc une matrice $\mathbf{A}$ définie positive et diagonale par blocs avec des blocs bandes symétriques et de même taille (c.f. : *figure* 4.1). Considérons également $\mathbf{X}$ et $\mathbf{B}$, deux vecteurs blocs de même taille qu'une colonne de $\mathbf{A}$. Une méthode classique pour résoudre le problème

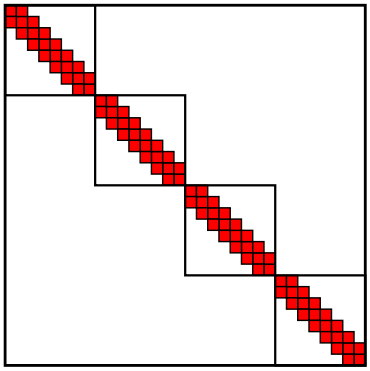


Figure 4.1 : *Matrice diagonale à blocs bandes et symétriques. Cette matrice contient 4 blocs tridiagonaux de taille 8*

A : une matrice diagonale par blocs avec des blocs symétriques et définis positifs
$\mathbf{A}^d$: La forme factorisée de la matrice **A**
B : le vecteur membre de droite
X : le vecteur inconnu, initialisé pour être égal à **B**

Résultat : **A** est factorisée et devient $\mathbf{A}^d$
pour tous les $A_{ii} \in \mathbf{A}$ **faire**
  `factorisation_bloc`(A_{ii})

Alg. 4.1 : Factorisation

Résultat : **X** contient la solution du système $\mathbf{AX} = \mathbf{B}$
pour tous les $({\mathbf{A}^d}_{ii}, X_i) \in (\mathbf{A}^d, \mathbf{X})$ **faire**
  `descente_remontee_bloc`(${\mathbf{A}^d}_{ii}, X_i$)

Alg. 4.2 : Descente-remontée

$\mathbf{AX} = \mathbf{B}$ consiste à décomposer la matrice **A** selon un algorithme de Cholesky (nous avons choisi une variante $\mathbf{LDL}^{\mathrm{T}}$ de cette décomposition). Une opération dite de « descente-remontée » permet ensuite de résoudre le système $\mathbf{AX} = \mathbf{B}$. Nous effectuerons cette dernière opération en place, c'est-à-dire qu'après avoir initialisé le vecteur **X** pour qu'il soit égal à **B**, nous résoudrons le système $\mathbf{AX} = \mathbf{X}$. La matrice **A** est diagonale par bloc, les opérations de factorisation (`factorisation_bloc`) et de descente-remontée (`descente_remontee_bloc`) peuvent donc être appliquées indépendamment sur chaque bloc de la matrice comme le montrent les **Alg.** 4.1 et 4.2

Nous allons maintenant montrer comment MTPS permet d'adapter le stockage de $\mathbf{A}^d$ et de **X** afin de paralléliser efficacement cet algorithme. Pour cela, nous allons tout d'abord introduire le modèle de la structure de données de MTPS. Puis, nous présenterons les opérateurs permettant de paralléliser les opérations s'appliquant aux éléments de cette structure.

4.1.3 Introduction au modèle de données

Les objets mathématiques **A** et **X** ont des structures de données comparables : ce sont des structures à deux niveaux. Ils possèdent des éléments dont la structure de données contient elle même un niveau (A_{ii}, X_i et B_i) accessibles avec un (pour les blocs X_i) ou deux indices (pour les blocs A_{ii}). Nous allons maintenant nous doter des outils permettant de définir ces structures de données ainsi que les contraintes qui s'y appliquent.

Tableau 4.1 : *Quelques exemples de classes appartenant à* $\mathbb{W}_n$.

Objet mathématique	données du *Profil*	n	Taille des *vecteurs*
Vecteur de taille N	N	1	N
matrice de taille $N_0 \times N_0$[1]	$\varnothing$	N_0	N_0
matrice bande de taille $N \times N$ et de largeur `bw`	N `bw`	1	`bw` $\times N$ avec des éléments fantômes[2]
matrice symétrique bande de taille $N \times N$ et de demi-largeur `hbw`	N `hbw`	1	`hbw` $\times N$ avec des éléments fantômes[2]
matrice tridiagonale de taille $N \times N$	N	3	$N-1$, N, $N-1$
matrice tridiagonale symétrique de taille $N \times N$	N	2	N, $N-1$

[1] Dans le cadre d'une classe définissant uniquement des matrices de taille $N_0 \times N_0$ fixée « en dur ».

[2] Ces éléments fantômes permettent de simplifier les calculs d'indices. Ces calculs convertissent la position d'un élément dans la matrice en une position dans le *vecteur* stockant effectivement les données.

Definition 1. Un *vecteur* définit une bijection entre un ensemble d'éléments distincts de même type et un intervalle $[\![0; n]\!]_{n\in\mathbb{N}}$.

Soit $\mathbb{V}$ l'ensemble des classes définissant des *vecteurs*. Soit `V<T>` $\in \mathbb{V}$ une classe de *vecteur* dont les éléments sont de type `T`. Soit $\mathtt{v_T}$ une instance de la classe `V<T>`. Nous noterons $\mathtt{v_T[i]}$ le $\mathtt{i}^{\text{è}}$ élément de $\mathtt{v_T}$ et $\mathtt{v_T.size()}$ le nombre d'éléments. Par exemple, un bloc A_{ii} de matrice ou X_i de vecteur peut stocker ses données dans des `V<float>`. Le nombre d'éléments peut varier entre deux instances $\mathtt{v_{T_1}}$ et $\mathtt{v_{T_2}}$ de la même classe `V<T>`. Nous dirons que le nombre d'éléments d'un vecteur est défini *dynamiquement*.

Definition 2. Une *n-séquence* définit un *vecteur* de n éléments.

Soit $\mathbb{S}_n$ l'ensemble des classes définissant des *n-séquences*. Soit `S<T>` $\in \mathbb{S}_n$ une classe de *n-séquences* dont les éléments sont de type `T`.

Soit $\mathbb{W}_n$ l'ensemble des classes dont les membres données peuvent être implémentés comme une unique *n-séquence* de *vecteurs* de flottants en simple précision (`float`). Prenons par exemple `W`, une classe élément de $\mathbb{W}_n$:

```
struct W {
  V<float> fields_[n];
};
```

Si `W` représente un bloc de vecteur X_i, n vaut 1 et `fields_[0]` stocke les éléments du vecteur. Le *tableau* 4.1 présente quelques exemples de classes issues du domaine de l'algèbre linéaire. Dans l'exemple que nous traitons ici, les blocs de matrice ont une structure bande symétrique dont les différents paramètres sont déterminés à l'exécution. À un bloc doit donc correspondre un *vecteur* unique dont la taille (`hbw` $\times N$) pourra être déterminée elle aussi lors de l'exécution. Les blocs de matrice, comme les blocs de vecteurs, sont donc stockés dans des instances de `W` pour lesquelles n vaut 1. Les articles [241, 242] introduisent un exemple où les blocs A_{ii} ont une structure tridiagonale et symétrique. La largeur de bande est définie dans le type de la structure et il est alors possible de stocker la diagonale et la sous-diagonale dans des champs séparés et n vaut 2.

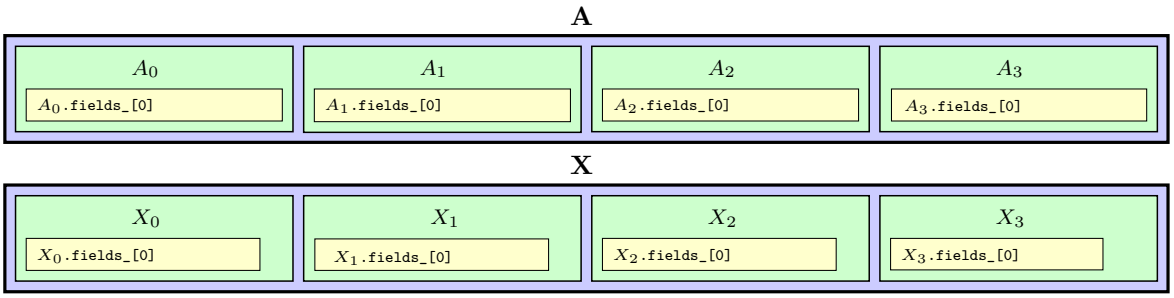


Figure 4.2 : *structure de données de la matrice* **A** *et du vecteur* **X** *sous formes de classes appartenant à* $\mathbb{V}<\mathbb{W}_n>$ *telles que définies dans le tableau 4.1.*

Definition 3. Le *profil* d'une classe `W` est l'ensemble des données permettant de calculer le nombre d'éléments de chacun des *vecteurs*. La seconde colonne du tableau 4.1 expose pour chaque exemple les données définissant le *profil*.

Soit $\mathbb{V}<\mathbb{W}_n>$ l'ensemble des classes de *vecteurs* dont les éléments sont des instances de classes de $\mathbb{W}_n$. Par exemple, les objets mathématiques **A** et **X** peuvent être implémentés à l'aide de classes appartenant à $\mathbb{V}<\mathbb{W}_n>$.

La figure 4.2 montre comment ces concepts se traduisent dans l'exemple que nous avons introduit. Les rectangles jaunes représentent des *vecteurs* de flottants simple précision. Les rectangles verts représentent des n-*séquences* de *vecteurs*. Enfin, les rectangles bleus représentent des *vecteurs* de n-*séquences*.

Les deux rectangles bleus représentent **A** et **X**. Les rectangles verts représentent les blocs A_{ii} et X_i. La matrice **A** représentée par le rectangle bleu du haut est un *vecteur* contenant quatre blocs A_{ii} représentés par quatre rectangles verts. Chaque bloc A_{ii} est une 1-*séquence* contenant un *vecteur* de `float` représenté par un rectangles jaunes.

4.1.4 Un parallélisme restreint aux opérations *vectorisables*

Nous allons maintenant présenter les conditions sous lesquelles MTPS permettra d'adapter le stockage des éléments contenus dans les classes de $\mathbb{V}<\mathbb{W}_n>$ (dont les instances sont des « *vecteurs* de n-*séquences* de *vecteurs* de `float` ») et ainsi générer des codes optimisés pour différentes architectures parallèles. Pour ce faire, nous allons tout d'abord redéfinir plus formellement le concept de *collection*. Nous utiliserons ensuite ce concept pour définir deux squelettes parallèles : `vectorized_for` et `vectorized_reduce`.

4.1.4.1 Les *collections*

Definition 4. Un m-*tuple* est un ensemble ordonné contenant m éléments pouvant être égaux. Nous noterons les m-*tuple* entre parenthèses : (a_m) est un m-*tuple* contenant les m éléments a_0, a_1, …, a_{m-1}. Contrairement aux n-*sequences*, les m-*tuple* peuvent contenir des types de classes ou des instances de classes différentes.

Soit $(\mathtt{T}_m)$ un m-*tuple* de m classes appartenant toutes à $\mathbb{W}_n$.

Soit $(\mathtt{t}_m)$ un m-*tuple* d'instances des classes du m-*tuple* $(\mathtt{T}_m)$.

Soit f une fonction prenant un m-*tuple* $(\mathtt{t}_m)$ comme argument et qui ne renvoie rien. Remarquons que dans le contexte de cette explication, le m-*tuple* $(\mathtt{t}_m)$ est mutable et peut être modifié par f.

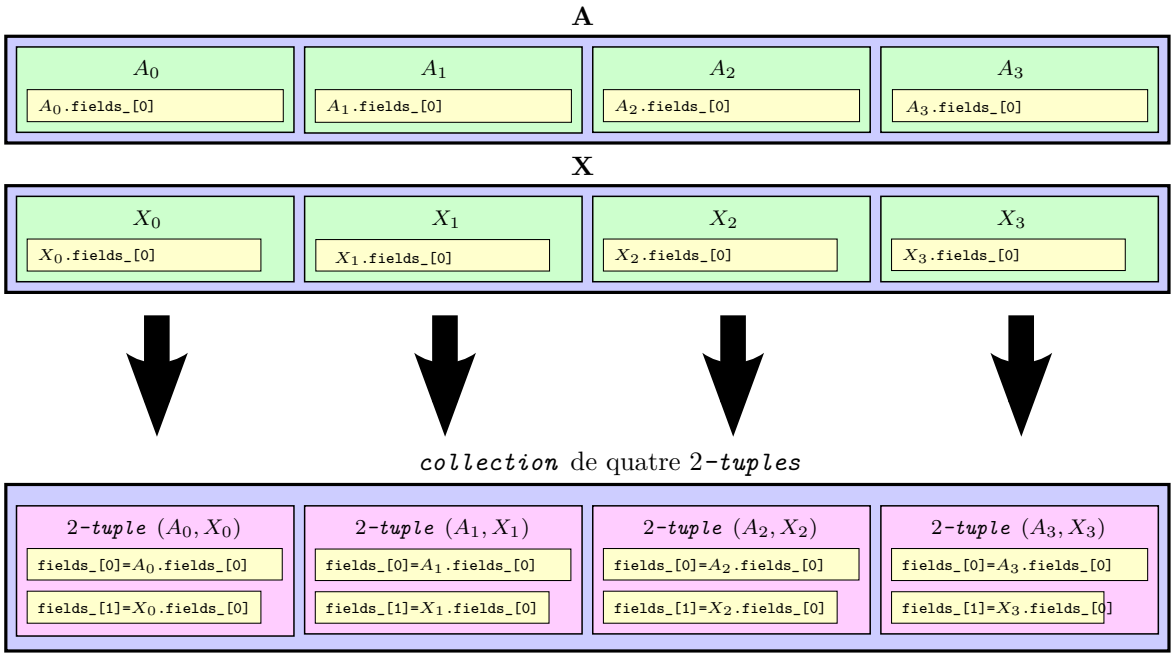


Figure 4.3 : *La matrice* **A** *et le vecteurs* **X** *sont regroupés en un* `vecteur` *de* `2-tuples`*.*

La *figure* 4.3 montre comment la matrice **A** et le vecteur **X** sont transformés pour construire quatre *2-tuples* (représentés en roses). Ces *2-tuples* forment les arguments de la fonction `descente_remontee_bloc`. Chacun de ces *2-tuples* contient deux *vecteurs* (représentés en jaune) issus respectivement de **A** et **X**. Ces quatre *2-tuples* sont rassemblés au sein d'un *vecteur* représentée en bleu.

Definition 5. Une *collection* c_f est un *vecteur* instance d'une classe appartenant à $\mathbb{V}<\mathbb{W}_n>$ et dont tous les éléments sont indépendants vis à vis d'une fonction f, c'est-à-dire que l'ensemble des résultats de la fonction f appliquée aux éléments de la *collection* c_f ne dépend pas de l'ordre dans lequel le vecteur c_f est parcouru.

Le fait qu'un *vecteur* soit ou non une *collection* dépend donc à la fois de la représentation des données et de la fonction qui lui sera appliquée. Considérons par exemple le cas d'une matrice dense. Elle peut être considérée comme un *vecteur* de lignes ou comme un *vecteur* de colonnes. Supposons que la fonction f que l'on souhaite appliquer à cette matrice consiste à calculer, pour chaque ligne, la somme de tous les éléments. Dans ce cas, la matrice, vue comme un *vecteur* de lignes est une *collection* vis à vis de f : pour chaque ligne, la somme peut être effectuée indépendamment des autres lignes. Si en revanche la matrice est vue comme un *vecteur* de colonnes, la matrice n'est pas une *collection* vis à vis de f.

En bas de la *figure* 4.3, le *vecteur* bleu représente une *collection* de *2-tuples* vis à vis de la fonction `descente_remontee_bloc`. Nous nommerons c_{drb} cette collection dans la suite de ce chapitre.

Definition 6. Deux éléments d'une *collection* sont dits *semblables* lorsqu'ils possèdent le même *profil*. Nous définissons une *collection homogène* comme une *collection* dont tous les éléments sont *semblables*.

La matrice **A** comporte des blocs ayant exactement la même structure et la même taille ; elle définit donc une *collection homogène* de blocs A_{ii}. Tous les blocs de vecteur X_i ont la même

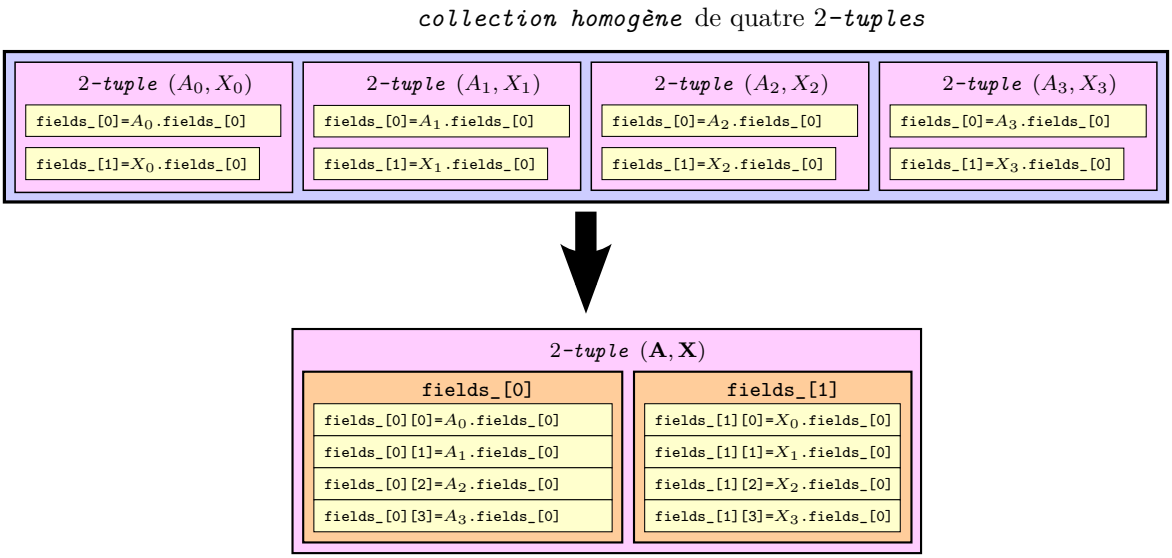


Figure 4.4 : *La* `collection` *de* `2-tuples` *est transformée en un* `2-tuple` *de structures de données bidimensionnelles.*

taille et sont donc naturellement ***semblables*** eux aussi. Le vecteur $\mathbf{X}$ définit donc également une ***collection homogène***. Composés d'éléments ***semblables***, les ***2-tuples*** (A_{ii}, X_i) sont naturellement ***semblables***. Par composition, la ***collection*** c_{drb} est donc ***homogène***.

MTPS permet de transformer une ***collection homogène*** en une structure de données bidimensionnelle semblable à celle présentée dans la section 2.2.2.2. Le format de stockage de ce type de structure peut facilement être adapté pour exploiter efficacement différentes architectures. La *figure* 4.4 illustre cette transformation : les rectangles roses représentent des ***2-tuples***, les rectangles oranges représentent des structures de données bidimensionnelles (telles que définies section 2.2.2.2) et les rectangles jaunes représentent un ***vecteur*** appartenant à un bloc. À chaque ligne de la structure bidimensionnelle correspond un indice de bloc différent. MTPS adaptera automatiquement l'entrelacement des données appartenant à ces différents blocs en fonction de l'architecture choisie comme présenté dans la section 2.2.2.2. Un mécanisme de *vues* comparables aux *pViews* de STAPL [201] permet à l'utilisateur de manipuler ces différentes données indépendamment du format de stockage sélectionné.

Nous allons maintenant voir comment la notion de ***collection*** permet de définir des opérations parallélisables sur les différentes plate-formes matérielles. Les deux prochaines sections introduiront les contraintes permettant la parallélisation efficace de ces opérations et proposeront deux squelettes parallèles permettant de les exprimer : `vectorized_for` et `vectorized_reduce`.

4.1.4.2 Le squelette parallèle `vectorized_for`

Soit F une fonction qui applique une fonction f à tous les éléments d'une ***collection*** c_f supposés mutables :

$$\begin{array}{lcll} F & : & (\mathtt{W}_n)^{N_{\text{Elts}}} \mapsto \varnothing & \\ F(c_f) & \equiv & f(c_f[i]) & \forall i \in [0, N_{\text{Elts}} - 1] \end{array}$$

Soit $\mathcal{F}$ l'ensemble des fonctions F qui appliquent une fonction f à tous les éléments d'une collection c_f.

Par définition de la *collection*, il est trivial de paralléliser une implémentation de F. En C, il suffirait de mettre une directive OpenMP `#pragma omp parallel for` au dessus de la boucle `for` itérant sur les éléments de la collection (c.f. : section 2.1.3.3). Par contre, la parallélisation de cette fonction sur une unité SIMD est moins évidente. Afin de comprendre les contraintes que cela induit sur f et c_f, nous allons nous intéresser au cas des unités SIMT.

Nous avons vu au chapitre 2 que ces unités comportent différents cœurs sur chacun desquels un *thread* exécute ses instructions. Le principe de fonctionnement des unités SIMT veut que si les *threads* de tous les cœurs doivent exécuter une instruction différente, elles ne s'exécuteront pas simultanément : les différentes instructions à exécuter seront sérialisées et exécutées uniquement sur les cœurs concernés. Les différents cœurs ne peuvent effectuer leurs traitement en parallèle que s'ils doivent exécuter la même instruction. Afin de garantir une exécution optimale, il faut donc s'assurer que les instructions devant être exécutées par les différents *threads* soient les mêmes pour chaque application de f.

Definition 7. Une fonction *non-divergente* est une fonction ne comportant pas de tests sur la valeur de ses arguments. La suite d'instruction générée par une fonction *non-divergente* est donc unique.

Definition 8. Une fonction F appliquant une fonction *non-divergente* f à tous les éléments d'une *collection* c_f est dite *régulière*.

Definition 9. Une opération *vectorisable* est une opération pouvant être décrite comme l'application d'une fonction *régulière* F à une *collection homogène* c_f. MTPS fournit le squelette `vectorized_for` afin d'exécuter cette opération en tirant profit des unités SIMT ou SIMD présentes dans les processeurs :

$$F(c_f) \equiv \texttt{vectorized\_for}(f, c_f).$$

L'**Alg.** 2.2 (page 54) présente l'opération de descente remontée implémentée par la fonction `descente_remontee_bloc`. Cet algorithme ne comporte pas de tests sur les valeurs des éléments des blocs. La fonction `descente_remontee_bloc` peut donc être implémentée de façon *non-divergente*.

Résoudre le problème $\mathbf{A}^{\mathbf{d}}\mathbf{X} = \mathbf{X}$ à l'aide d'une descente-remontée revient donc à appliquer une fonction *regulière* sur la *collection homogène* de 2-*tuples* c_{drb}. Ce résultat ne doit pas nous surprendre : nous avons déjà présenté une implémentation vectorisée de cette opération dans le chapitre 2.

4.1.4.3 Le squelette parallèle `vectorized_reduce`

Dans cette section, nous allons introduire les éléments nécessaires pour définir une réduction. Après avoir présenté formellement cette famille d'opérations, nous appliquerons les concepts introduits sur l'exemple de la norme d'un vecteur.

L'opération prise en charge par le squelette `vectorized_reduce` est un cas particulier de l'opérateur *fold*[34] [243] disponible dans la majorité des langages fonctionnels. Dans le cas général,

34. L'opérateur *fold* correspond aux opérations implémentables à l'aide de *MapReduce* [192]

fold prend trois arguments[35] :

$$\begin{array}{lcl} \mathit{fold} & : & ((\alpha,\beta)\mapsto\beta,\beta,[\alpha])\mapsto\beta \\ \mathit{fold}(f_{\text{fold}},v,[]) & = & v \\ \mathit{fold}(f_{\text{fold}},v,[h:t]) & = & f_{\text{fold}}\left(h,\mathit{fold}\left(f_{\text{fold}},v,t\right)\right) \end{array}$$

Le premier argument, noté f_{fold} ci-dessus est la fonction binaire de réduction. Le second argument v représente la valeur initiale de la réduction. Enfin, le troisième argument, que nous noterons l, correspond à une liste d'éléments à réduire.

La parallélisation de l'opérateur *fold* sur des unités SIMD ou SIMT impose un certain nombre de contraintes sur f_{fold}, v et l :

- f_{fold} doit être décomposée en deux fonctions ***non divergentes*** f_m et f_r telles que :

$$\begin{array}{lcl} f_m & : & \alpha\mapsto\beta \\ f_r & : & (\beta,\beta)\mapsto\beta \\ f_{\text{fold}}(a,b) & = & f_r(f_m(a),b) \end{array}$$

La fonction f_r doit en outre être associative et posséder un élément neutre b_0,

- v doit être égal à b_0,
- l doit être une ***collection homogène*** vis à vis de f_m.

Finalement, `vectorized_reduce` prend quatre arguments et est défini par :

$$\begin{array}{lcl} \texttt{vectorized\_reduce} & : & ((\alpha\mapsto\beta),(\beta,\beta)\mapsto\beta,\beta,[\alpha]) \\ \texttt{vectorized\_reduce}(f_m,f_r,v,[]) & = & v \\ \texttt{vectorized\_reduce}(f_m,f_r,v,[h:t]) & = & f_r\left(f_m\left(h\right),\texttt{vectorized\_reduce}\left(f_m,f_r,v,t\right)\right) \end{array}$$

Afin d'illustrer le fonctionnement de ce squelette, nous allons maintenant définir le calcul de la norme d'un vecteur à l'aide du squelette `vectorized_reduce`. Pour cela nous allons définir les deux fonctions que nous passerons comme arguments à `vectorized_reduce` :

- **square** :

$$\begin{array}{lcl} \text{square} & : & \texttt{float}\mapsto\texttt{float}_{\text{positif}} \\ \text{square}(a) & = & a*a \end{array}$$

- **add** :

$$\begin{array}{lcl} \text{add} & : & (\texttt{float}_{\text{positif}},\texttt{float}_{\text{positif}})\mapsto\texttt{float}_{\text{positif}} \\ \text{add}(a,b) & = & a+b \end{array}$$

Finalement, nous pouvons définir la norme d'un vecteur V par :

$$\text{norme}(V)=\texttt{vectorized\_reduce}(\text{square},\text{add},0,V).$$

35. Nous présentons ici la forme non-curryfiée [244] de la fonction *fold*.

4.1.5 Extension de la structure de données

Soit $\mathbb{I}$ l'ensemble des types arithmétiques intégrés au langage C++ tels que définis par Bjarne Stroustrup dans la section 4.4.1 de *The C++ Programming Language* [245].
$\mathbb{I} = \{\texttt{float}, \texttt{double}, \texttt{int}, \texttt{bool}, \texttt{long}, \texttt{char}...\}$

Soit $\mathbb{M}$ l'ensemble des classes dont les attributs de données peuvent être écrits comme des ***séquences*** de ***vecteurs*** d'éléments de $\mathbb{I}$. Prenons par exemple `M`, une classe de $\mathbb{M}$:

```
struct M{
  V<float>  floatFields_[nFloat];
  V<double> doubleFields_[nDouble];
  V<int>    intFields_[nInt];
  V<bool>   boolFields_[nBool];
  ...
  // un vecteur pour chaque
  // type arithmetique
};
```

L'ensemble $\mathbb{W}$ est donc une restriction de $\mathbb{M}$ aux classes dont toutes les données sont des flottants simple précision.

Les définitions vues précédemment se généralisent aux ***collections*** de ***m-tuples*** d'objets dont le type appartient à $\mathbb{M}$ à condition que tous les types arithmétiques utilisés aient la même empreinte mémoire. En effet, cette condition permet de s'assurer que le nombre de voies des unités SIMD utilisées est le même pour tous les types employés. Par exemple, si les éléments de **A** étaient des nombres flottants double précision alors que les éléments de **X** étaient des nombres flottants simple précision, il serait impossible de vectoriser la descente remontée comme nous l'avons fait au chapitre 2. *A contrario*, si les éléments de **A** étaient des entiers de 32 bits, alors cette vectorisation serait possible : dans un registre SIMD (un registre SSE par exemple), il est possible de paqueter autant d'entiers de 32 bits que de flottants simple précision (4 pour les registres SSE).

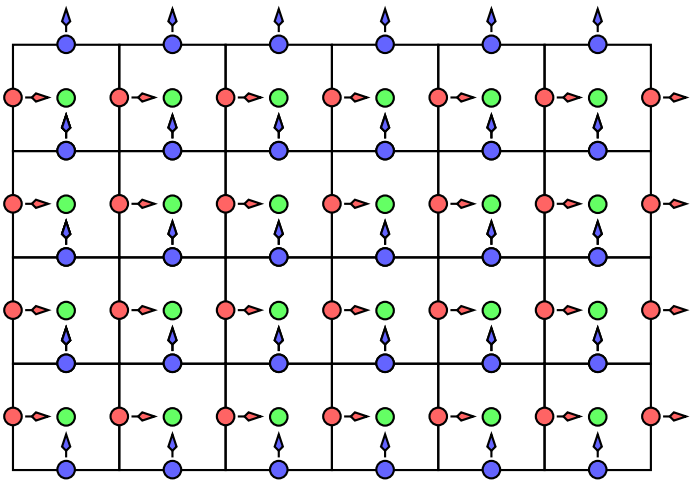
4.1.6 Les *contextes vectoriels*

Le concept clé de MTPS est le concept de ***collections homogène***. MTPS prend en charge l'adaptation du stockage de ces ***collections*** selon l'architecture ciblée. MTPS fournit en outre deux squelettes parallèles permettant d'écrire des fonctions opérant sur des ***collections*** : `vectorized_for` et `vectorized_reduce`.

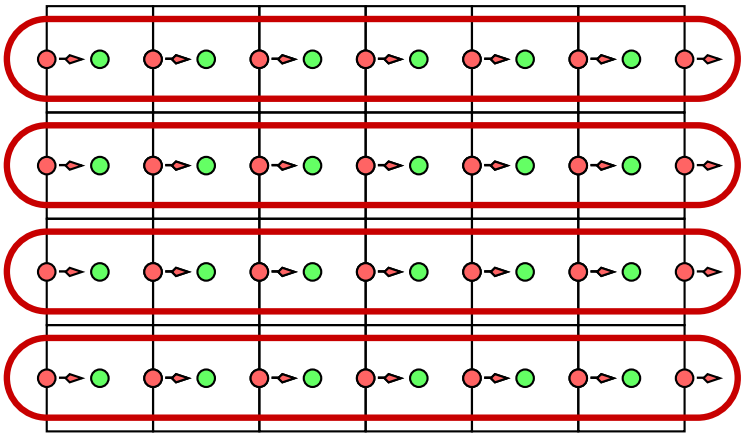
Cependant, nous l'avons vu, un ***vecteur*** ne devient une ***collection*** qu'en regard d'une ou plusieurs fonctions. Deux fonctions ***vectorisables*** avec la même ***collection*** appartiennent au même ***contexte vectoriel***. Deux fonctions qui sont vectorisables avec les mêmes données mais des définitions différentes de ***collection*** appartiennent à deux contextes différents.

Nous allons illustrer ceci avec un exemple issu du solveur SP_N de COCAGNE. La *figure* 4.5(a) représente un maillage spatial 2D. Les Degrés De Liberté (DDLs) de flux neutronique sont représentés par les points verts tandis que les flèches bleues et rouges représentent les DDLs de courants (dérivé du flux neutronique). Nous ne nous attarderons pas ici sur les méthodes de résolutions employées. Le lecteur intéressé pourra se référer aux articles suivants [21, 246, 33, 23]. Nous retiendrons ici que la méthode de résolution retenue dans le solveur SP_N conduit à résoudre un ensemble de problèmes indépendants pour chaque ligne de DDLs suivant x puis un ensemble de problèmes indépendants pour chaque colonne suivant y comme illustré sur les *figure* 4.5(b) et 4.5(c).

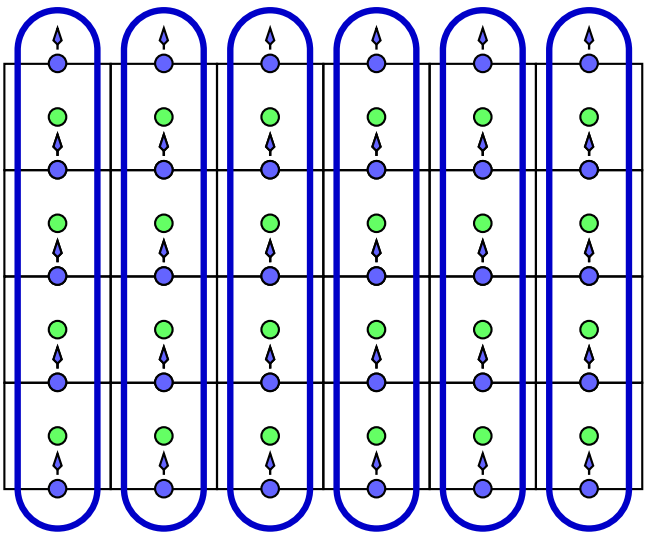
Lorsque l'on traite les problèmes suivant la direction x, chaque ligne entourée en rouge sur la *figure* 4.5(b) représente un problème indépendant dont la résolution peut être implémentée



(a) Degrés de liberté de flux et de courants.



(b) Suivant y, chaque ligne conduit à un problème indépendant.



(c) Suivant x, chaque colonne conduit à un problème indépendant.

Figure 4.5 : *Un maillage spatial 2D issu de la chaîne COCAGNE.*

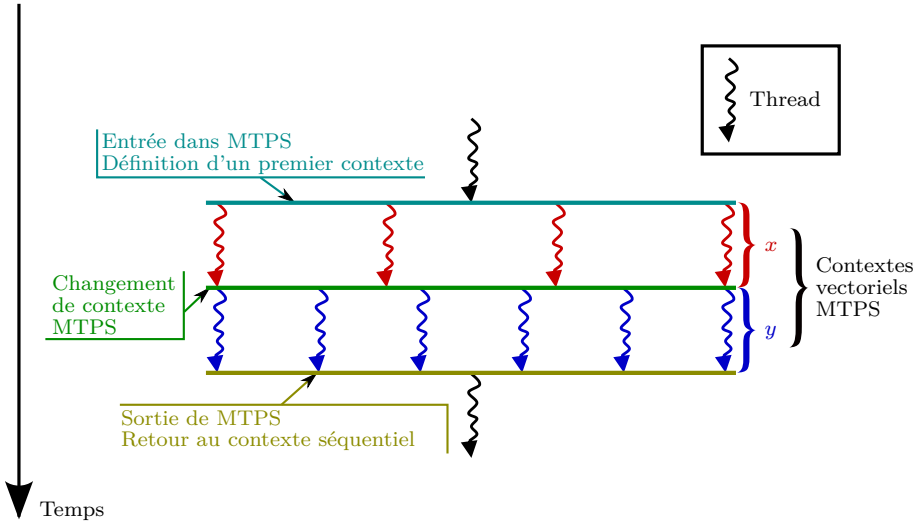


Figure 4.6 : *Une application enchaîne différents contextes d'exécution.*

de manière `vectorisable`. Pour ce faire, la fonction de résolution d'une ligne doit prendre en argument (entre autres) un `vecteur` de DDLs de courant et un `vecteur` de DDLs de flux. Nous représentons ainsi l'ensemble des DDLs de flux par un `vecteur` de `vecteurs`. Le premier niveau contient les lignes de DDLs. Chaque ligne étant à son tour représentée par un `vecteurs` de `float`. Autrement dit, l'ensemble des DDLs de flux est représenté par un `vecteur` 2D correspondant à un stockage ligne par ligne.

Lorsque l'on traite le problème suivant y, les mêmes causes menant aux mêmes conclusions, l'ensemble des DDLs de flux est représenté par un `vecteur` 2D correspondant à un stockage colonne par colonne.

Ces deux opérations sont donc `vectorisables`, mais dans des `contextes vectoriels` différents. Entre ces deux opérations, un `changement de contexte` doit donc être effectué. Dans ce cas, ce changement de contexte consiste à réordonnancer le `vecteur` de DDLs de flux représenté par les points verts. Ce changement de contexte est également l'occasion de redéfinir le nombre de tâches indépendantes présentes à ce stade de l'exécution (4 suivant x puis 6 suivant y). La *figure* 4.6 illustre les différentes étapes de la résolution du problème représenté sur la *figure* 4.5 en utilisant MTPS :

1. Au départ, seul le *thread* principal du programme s'exécute et permet par exemple d'initialiser les données.
2. Un premier `contexte` MTPS est créé pour traiter la direction x. Le stockage des différentes `collections` est adapté selon l'architecture ciblée.
3. Chaque *thread* résout un problème correspondant à une colonne de DDLs.
4. Avant de pouvoir traiter la direction y, un `changement de contexte` doit avoir lieu. À cette occasion, les données sont réordonnancées et l'environnement d'exécution est reconfiguré.
5. Chaque *thread* résout un problème correspondant à une ligne de DDLs.
6. Enfin, l'exécution terminée, le programme sort de MTPS et revient dans un `contexte` séquentiel.

4.2 Principes de fonctionnement de MTPS

Dans la section précédente, nous avons introduit les différents concepts constituant le modèle de programmation proposé par MTPS. Dans cette section, nous allons présenter les principes du fonctionnement de MTPS. MTPS vise à produire un exécutable spécialisé et optimisé pour l'architecture matérielle ciblée, l'algorithme utilisé et les structures de données décrites par l'utilisateur.

La mise en œuvre de ces principes s'effectue en deux temps. Dans un premier temps, les types des structures de données correspondant aux *collections* doivent être générés. Ces *collections* peuvent ensuite être instanciées et utilisées lors de l'exécution du programme. Ces principes impliquent donc une compilation en deux étapes. La première étape, partant du code source original de l'utilisateur doit permettre de générer les structures de données et de spécialiser les squelettes parallèles. Ces éléments étant définis, ils peuvent alors être utilisés pour générer un exécutable optimisé.

Cette présentation est divisée en cinq parties. Après avoir introduits nos choix technologiques dans une première partie, les trois parties suivantes présentent les trois principes clés utilisés pour permettre la spécialisation des structures de données et des squelettes parallèles. La dernière partie présente une vue d'ensemble et détaille le travail devant être fourni par un utilisateur de MTPS pour paralléliser une opération.

4.2.1 Choix technologiques d'implémentation

Différentes méthodes d'implémentation de MTPS sont naturellement possibles. Il serait par exemple possible d'utiliser des outils de génération de code. Cela permettrait, partant du code source original de générer un code source intermédiaire mettant en œuvre des structures de données et des squelettes parallèles spécialisés. Ce code intermédiaire pourrait ensuite être compilé dans un environnement de développement classique. Nous avons choisi d'utiliser les possibilités de programmation générative offerte par le C++ pour effectuer ces deux étapes en un seul temps. Fondées sur les patrons (*templates*) de classes ou de fonctions, cette approche permet de générer successivement le code intermédiaire spécialisé puis l'exécutable optimisé lors de la compilation avec un compilateur C++ standard. Si ce choix limite les transformations possibles, il a l'avantage de ne pas nécessiter la mise au point d'outils de déboguage spécifique.

4.2.2 Définition et instanciation de la structure de données

MTPS représente les différentes cibles matérielles sous la forme de classes. Ces classes fournissent un ensemble de fonctions d'entrelacement. Ces fonctions définissent le format d'entrelacement adapté à l'architecture considérée. Leur rôle est donc équivalent à celui de la fonction `interleaving` utilisée à la ligne 26 du code page 61. L'utilisateur choisit, lors de la compilation, la cible visée et donc les fonctions d'entrelacement associées.

L'utilisateur doit en outre fournir une description des éléments de la *collection*. Cette description définit la taille n et la nature de la n*-séquence* de *vecteurs* qui permet de contenir les données correspondant à un élément.

Le conteneur de *collection* proposé par MTPS s'appuie sur ces deux éléments pour définir, lors de la compilation, une structure de données optimisée selon l'architecture cible visée. Les flèches verticales représentées sur la *figure* 4.7 représentent ainsi les dépendances entre ces concepts.

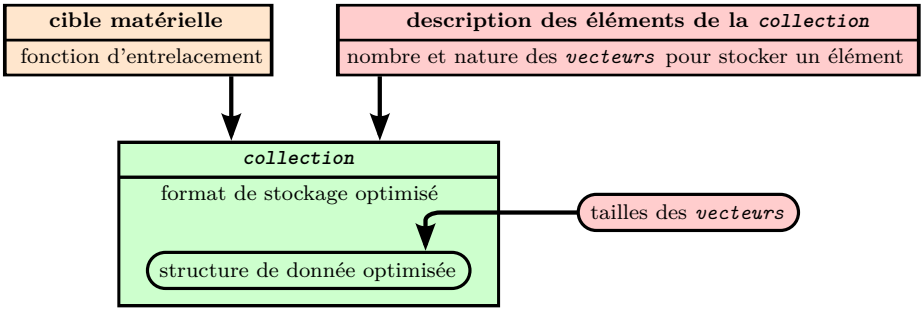


Figure 4.7 : *L'implémentation du concept de* `collection` *s'appuie sur la description des éléments fournie par l'utilisateur et sur les fonctions d'entrelacement fournies par la classe décrivant l'architecture matérielle afin de générer une structure de données optimisée.*

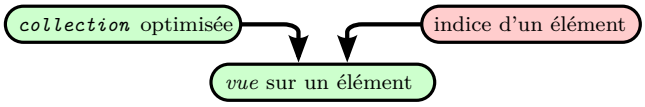


Figure 4.8 : *MTPS fournit des* vues *permettant d'accéder aux différents éléments d'une* `collection` *dont le stockage est optimisé.*

Lors de la création de la *collection*, l'utilisateur doit fournir le nombre d'éléments de cette *collection* ainsi que la taille des *n-vecteurs*. La flèche horizontale entre les deux rectangles arrondis représente ce passage d'information lors de l'exécution. C'est à ce moment que le conteneur de *collection* alloue la mémoire et crée la structure de données optimisée.

4.2.3 Abstraction du parallélisme et accès aux données

Dans la section 2.2.2, nous avons montré comment réutiliser une seule implémentation de la fonction `solveProblemBlock` (définie page 56) sur les différentes cibles matérielles. L'idée était de paramétrer cette fonction par le type de ses données d'entrée. Cette fonction était alors automatiquement spécialisée lors de la compilation selon l'architecture pour laquelle étaient adaptées les données. Les *vues* fournies par MTPS permettent de générer automatiquement des implémentations spécialisées des blocs de matrice A_{ii} ou des vecteur X_i indépendamment du format de stockage retenu. De ce point de vue, nous avons simplement généralisé dans MTPS les techniques présentées dans la section 2.2.2. Comme l'illustre la *figure* 4.8, une *vue* est crée à partir d'une *collection* et de l'indice d'un élément de cette *collection*. Les *vues* s'appuient sur les fonctions d'entrelacement de la cible visée pour accéder aux données de chaque élément. Notons qu'une *vue* ne correspond pas forcément à un seul élément de la *collection*. En effet, si l'architecture ciblée contient des unités SIMD [36], une *vue* correspond à plusieurs éléments. Par exemple, si l'architecture ciblée comporte des unités SIMD à quatre voies, une *vue* correspondra à quatre éléments de la *collection*. Nous avons présenté ce principe dans la section 2.2.2.3, page 58. Nous avions alors utilisé des types de données vectorisés (`SSEData`) pour effectuer simultanément les opérations de calcul correspondant à quatre blocs différents. Les squelettes parallèles de MTPS sont chargés de créer les *vues* sur les éléments de la *collection* et d'appliquer la même fonction *non-divergente* sur ces *vues*. La flèche pointillée de la *figure* 4.9 illustre ces appels.

36. seules les unités vectorielles SSE sont supportées dans la version de MTPS de la fin 2011

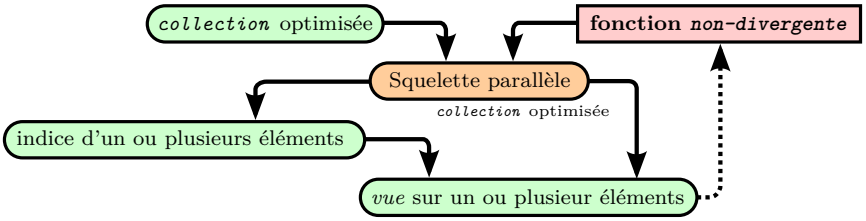


Figure 4.9 : *Les opérateurs parallèles utilisent des* `vues` *pour appliquer les fonctions définies par l'utilisateur aux différents éléments de la* `collection`.

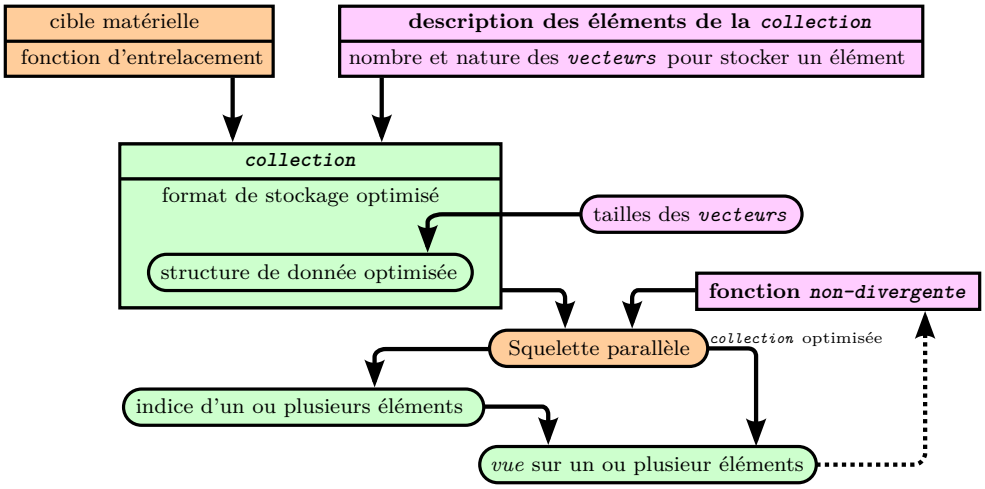


Figure 4.10 : *Vue d'ensemble du fonctionnement de MTPS.*

4.2.4 Vue d'ensemble

La *figure* 4.10 donne une vue d'ensemble sur les différents éléments introduits jusqu'à présent. Les éléments représentés en roses doivent être fournis par l'utilisateur. L'utilisateur doit en outre choisir la cible matérielle ainsi que le squelette parallèle à utiliser. Ces éléments sont représentés en orange sur la *figure* 4.10. Les éléments représentés en vert sont quant à eux fournis ou générés par MTPS.

4.3 Contraintes d'utilisation et architectures matérielles

Dans ce chapitre, nous avons présenté MTPS, une bibliothèque dédiée à la parallélisation multicible. Nous avons présenté dans la section 4.1 le formalisme auquel doivent se soumettre les algorithmes pour pouvoir être parallélisés avec MTPS. Ce formalisme impose de pouvoir décrire les données sous forme de *collections*.

Les contraintes pesant sur ces *collections* sont nécessaires au support des différentes architectures que nous avons considérées. Dans cette section, nous allons préciser le lien entre ces contraintes et le support de ces différentes architectures matérielles. Pour cela, nous allons définir, pour chaque architecture, les contraintes permettant son utilisation.

Le support d'un espace mémoire séparé, nécessaire pour le GPU, implique de pouvoir copier facilement les données d'un espace mémoire vers un autre. Pour cela, MTPS impose que les éléments d'une *collection* soient décrits comme des *n-séquences* de *vecteurs*. Cela permet en effet d'implémenter la structure de données générée comme n tableaux en mémoire

qu'il est aisé de copier d'un espace mémoire vers un autre. En pratique, MTPS alloue un espace mémoire par tableau et manipule n pointeurs vers ces espaces mémoire.

Le support efficace des unités SIMT est rendu possible grâce à deux choses. Premièrement, les éléments d'une *collection* doivent être *semblables*. Deuxièmement, les fonctions appliquées doivent être *non-divergentes*. La première contrainte permet d'adapter le format de stockage des données tandis que la seconde permet de vectoriser les opérations en appliquant, en même temps, la même fonction à différents éléments de la *collection* à l'aide des unités SIMT. Le respect de ces deux contraintes permet donc l'obtention de bonnes performances.

Le support des unités SIMD implique le respect des contraintes permettant l'utilisation des unités SIMT. À ces contraintes s'ajoutent le fait que tous les types de données élémentaires doivent avoir la même taille. Cela permet de définir des *vues* qui manipulent différents objets simultanément.

Le respect de ces contraintes permet d'automatiser la parallélisation et la vectorisation d'un même code source pour différentes architectures matérielles. Nous verrons dans le chapitre 6 que les performances alors observées sont comparables aux performances obtenues pour des implémentations optimisées « à la main ».

Nous avons conçu MTPS comme une couche de bas niveau destinée à être utilisée pour mettre au point des outils de plus haut niveau. Dans le prochain chapitre, nous introduirons Legolas++, une bibliothèque C++ dédiée à la résolution de système linéaires mettant en oeuvre des matrices creuses structurées. Nous verrons alors comment Legolas++ se positionne vis à vis de ces contraintes. Nous présenterons également un démonstrateur de Legolas++ capable de paralléliser et de vectoriser automatiquement un solveur sur les processeur X86_64.

Il faut savoir ce que l'on veut. Quand on le sait, il faut avoir le courage de le dire ; quand on le dit, il faut avoir le courage de le faire.

Georges Clémenceau (1841 – 1929)
Président du conseil

Chapitre 5

Conception et réalisation d'un démonstrateur multicible de Legolas++

Le chapitre 1 introduit l'importance de la séparation entre le développement des fonctionnalités des codes de calcul et leur optimisation pour les différentes plate-formes matérielles. Le chapitre 2 illustre l'intérêt de différentes architectures matérielles et les deux principales difficultés liées à la maintenance de différentes branches de codes dédiées à ces dernières. Premièrement, l'expression du parallélisme utilise des paradigmes différents entre les différentes architectures ce qui complique la réutilisation du code. Deuxièmement, les structures de données doivent être adaptées pour chaque architecture, rendant ainsi compliquées les communications entre les différentes branches du code.

Dans le chapitre 3, étudions et comparons donc différentes approches permettant d'écrire des applications parallèles portables sur différentes architectures matérielles. Si plusieurs d'entre elles permettent d'abstraire les différentes formes de parallélisme, aucune à ce jour ne prend en charge l'adaptation des structures de données.

Le chapitre 4 introduit MTPS, notre bibliothèque permettant une adaptation automatique des structures de données à l'architecture matérielle. Cette possibilité est cependant offerte au détriment de la généralité de l'approche. En effet, MTPS permet de paralléliser efficacement sur différentes architectures l'ensemble des opérations qui sont effectuées dans un même *contexte vectoriel*. À ces contraintes fonctionnelles s'ajoutent des contraintes de programmation liées à la prise en charge des différents paradigmes et modèles de programmation. Par exemple, les réordonnancements de données nécessitent de travailler avec des *collections* d'objets ne comportant que des champs de données unidimensionnels contenant un même nombre d'éléments.

Dans ce chapitre, nous présentons dans un premier temps les principes et le fonctionnement de Legolas++, une bibliothèque d'algèbre linéaire développée à EDF R&D. Dans un second temps, nous expliquons comment étendre Legolas++ vers une version multicible. Pour cela, Legolas++ doit respecter les contraintes de fonctionnement de MTPS. L'implémentation d'un démonstrateur multicible de Legolas++ permet de valider notre approche. Nous analyserons les performances obtenues avec notre démonstrateur dans le chapitre 6. Nous verrons alors qu'il permet de mettre au point des solveurs parallèles portables sur des architectures disposant d'unités SIMD différentes et dont les performances sont comparables aux performances obtenues avec une implémentation optimale.

Sommaire

5.1 Legolas++ : présentation de l'existant

Legolas++ est une bibliothèque active [37] écrite en C++ qui fournit un DSEL dédié à la mise au point de solveurs d'algèbre linéaire mettant en œuvre des matrices creuses structurées sur plusieurs niveaux. Une première introduction à Legolas++ a été effectuée dans la section 1.3. Nous introduisons dans cette section les différents éléments permettant la mise au point d'un solveur. Nous commençons par les éléments les plus simples : les vecteurs. Nous continuons avec les matrices avant de terminer avec les solveurs. Nous verrons ensuite comment étendre ces concepts afin de mettre au point un démonstrateur multicible de Legolas++.

5.1.1 Les vecteurs

Legolas++ fournit des vecteurs multidimensionnels. Contrairement aux vecteurs proposés par Blitz++ [78], ces vecteurs peuvent être ***non-rectangulaires***. Différents éléments permettent de caractériser un vecteur Legolas++ :

- le type des données (ex. : `float` ou `double`),
- le nombre de dimensions du vecteur,
- les tailles du vecteur et de ses sous-vecteurs.

Les deux derniers éléments permettent de définir le ***profil*** d'un vecteur tel que nous l'avons défini dans la section 4.1.3. Dans Legolas++, la description de la taille des éléments (vecteurs ou matrices) est appelée ***shape***. Dans le cas des vecteurs Legolas++, ces deux notions sont équivalentes. Nous verrons que cela n'est pas le cas pour les matrices.

La création et l'utilisation de vecteurs Legolas++ est simple : Legolas++ fournit le patron de classe `Legolas::MultiVector`. Il est paramétré par le type des données et la dimension du vecteur. L'extrait de code suivant définit par exemple le type d'un vecteur à deux dimensions contenant des nombres flottants simples précisions [38] :

```
typedef Legolas::MultiVector<float,2> Vector2D;
```

Pour créer un vecteur, un utilisateur doit seulement fournir sa ***shape*** en sus de son type. Le type permettant de définir la ***shape*** d'un vecteur dépend naturellement du type de vecteur considéré. Ce type est fournit par la classe `Legolas::MultiVector` sous le nom `Shape`. L'extrait de code suivant construit un vecteur à deux niveaux comportant n_1 vecteurs unidimensionnels de taille n_2 :

```
Vector2D::Shape vectorShape(n1,n2);
Vector2D X(vectorShape);
```

Le vecteur X peut alors être utilisé comme suit :

```
for (int i1 = 0 ; i1 < n1 ; ++i1)
  for (int i2 = 0 ; i2 < n2 ; ++i2)
    X[i1][i2] = ...
```

Dans Legolas++, l'implémentation des vecteurs s'appuie sur le mécanisme d'*expression templates* présenté dans l'annexe A.2. Cette implémentation est en outre parallélisée à l'aide de la bibliothèque INTEL TBB [139] et conduit à l'obtention de performances équivalentes à celles obtenues avec les meilleures implémentations présentées à la section 1.2.2 :

```
Vector2D Y(vectorShape), Z(vectorShape);
float a, b, c;
...
```

37. cf. section 1.2.2.6
38. Les types fournis par Legolas++ apparaissent en rouge dans les extraits de code.

```
Vector2D V = a*X+b*Y+c*Z;
```

5.1.2 Les matrices

Dans la section précédente, nous avons introduit les vecteurs Legolas++ ainsi que leur utilisation. Dans cette section, nous allons présenter les matrices proposées par Legolas++. Ces matrices ont la particularité de posséder une structure. Cette structure, à plusieurs niveaux est décrite comme la combinaison récursive de structures élémentaires comme nous l'avons décrit dans la section 1.3. Comme pour les vecteurs, la création d'une matrice Legolas++ nécessite plusieurs étapes. Elle doit tout d'abord être définie avec ses données avant de pouvoir être instanciée et manipulée. Nous présentons dans cette section les principes de chacune des ces étapes ainsi que les possibilités de configurations offertes par Legolas++.

5.1.2.1 Définition d'une matrice

La première étape pour créer une matrice consiste à fournir à Legolas++ une définition de cette matrice. Une définition de matrice Legolas++ est un objet définissant complètement la matrice et ses éléments. Elle doit définir les éléments suivants :

- le type de ses données réelles (par exemple `float` ou `double`),
- les données permettant de calculer les éléments de la matrice,
- sa structure élémentaire (par exemple `Legolas::Diagonal` ou `Legolas::Banded`),
- le nombre de niveaux définissant sa structure,
- sa taille et celles de ses blocs (sa ***shape***),
- la définition de ses niveaux inférieurs.

Les quatre derniers points définissent le ***profil*** de la matrice, c'est à dire l'ensemble des donnés permettant de déterminer le nombre d'éléments non nuls de la matrice ainsi que leur emplacement. Le ***profil*** diffère de la ***shape*** Legolas++ car cette dernière ne permet de décrire que la taille de matrice. Par exemple, le nombre de diagonales est un élément du ***profil*** d'une matrice bande, mais il ne fait pas partie de sa ***shape***.

Les différentes structures élémentaires définissent le schéma de remplissage de la matrice. Elles forment une hiérarchie. En effet, la structure `Legolas::Diagonal` est un cas particulier de la structure `Legolas::Banded` qui est elle-même une spécialisation de la structure `Legolas::Sparse`. À chacune de ces structures correspond un ensemble d'accesseurs optimisés. Par exemple, la structure `Legolas::SymmetricBanded` définit les accesseurs suivants :

- `lowerBandedGetElement(int i, int j)`,
- `int hbw()`

et hérite les accesseurs suivants des structures de sa hiérarchie :

- `BandedGetElement(int i, int j)`,
- `SparseGetElement(int i, int j)`,
- `int nrows()`,
- `int ncols()`.

L'annexe C.1 présente un exemple de définition de matrice. La classe `MDefinition` représente la définition Legolas++ d'une matrice bande symétrique.

5.1.2.2 création et manipulation d'une matrice

La définition d'une matrice Legolas++ n'est pas instanciable directement. Une classe représentant une définition de matrice peut ne pas définir tous les éléments permettant de caractériser

une matrice. En particulier, la taille d'une matrice n'est généralement pas connue lors de l'écriture de sa définition : elle est souvent calculée à l'exécution à partir d'autres données. Nous allons ici montrer comment créer une matrice Legolas++ représentant M à partir d'une définition de matrice `MDefinition`[39].

Pour cela, la première étape consiste à construire une classe de matrice correspondant à cette définition :

```
typedef Legolas::GenericMatrixInterface<MDefinition>::Matrix Matrix;
```

Le patron de classe `Legolas::GenericMatrixInterface` met en place l'ensemble des accesseurs spécialisés correspondant à la structure définie par `MDefinition` et à sa hiérarchie.

Pour créer une matrice Legolas++, il faut créer une instance de la classe `MDefinition::Data` :

```
MDefinition::ShapeType shape(10,10); //M est une matrice 10 × 10
MDefinition::Data initializer(shape, 3); //la demi-largeur de bande de M : 3
```

et de l'utiliser pour instancier la matrice Legolas++ :

```
Matrix m(initializer);
```

La matrice Legolas++ `m` peut maintenant être utilisée conjointement avec les vecteurs Legolas++ :

```
m.getElement(i,j); //acces generique (inefficace) a un element de m
m.lowerBandedGetElement(i,j); //acces efficace a un element de m

Legolas::MultiVector<float, 1> v1,v2, v3;
... // initialisation de  v1

//Legolas++ prend en charge les operations matrice-vecteur
v2 = m*v1; //produit matrice-vecteur
v3 = v2/m; //resolution de l'equation m*v3=v2 (cf. section 5.1.3)
double error = Legolas::norm(v1-v3);
assert(error < 1e-12);
```

5.1.2.3 Les options de configuration des matrices

Afin d'optimiser les performances de l'application, Legolas++ permet de définir des options modifiant le comportement des matrices. Dans cette section, nous allons étudier les cinq options disponibles pour spécialiser le comportement d'une matrice Legolas++.

Le mode de stockage de la matrice

Par défaut, les matrices Legolas++ ne prennent pas en charge le stockage des éléments : à chaque accès aux élément de la matrice correspond un appel à une fonction définie par l'utilisateur dans sa définition. On parle alors de matrice « virtuelle ».

Dans certain cas, l'accès aux différents éléments peut être coûteux, Par exemples, si les données d'une matrice sont définies dans un fichier. Il est alors préférable de déléguer à Legolas++ la prise en charge du stockage de ces éléments en utilisant des matrices « stockées ». Dans ce cas, la construction d'une matrice est effectuée en trois temps par Legolas++. Dans un premier temps, Legolas++ crée une matrice « virtuelle » temporaire. Dans un second temps, une matrice « stockée » est instanciée. Enfin, dans un troisième temps, Legolas++ recopie les éléments de la matrice « virtuelle » dans la matrice « stockée ».

Selon l'origine des données ou les contraintes pesant sur l'application, l'utilisateur de Legolas++ peut choisir l'une ou l'autre de ces solutions (matrice « virtuelle » ou « stockée »).

39. L'annexe C.1 présente un exemple d'implémentation pour la classe `MDefinition`.

L'utilisation de matrices « virtuelles » implique généralement une empreinte mémoire plus faible. Cependant, recalculer à la demande les éléments de la matrice n'est pas toujours aussi performant que le simple accès à une donnée stockée en mémoire. De plus, seules les matrices « stockées » peuvent être modifiées après leur création.

L'algorithme de résolution de systèmes

Legolas++ permet de définir pour chaque matrice l'algorithme à utiliser pour résoudre les systèmes linéaires $AX = B$. Par défaut, l'algorithme utilisé est l'algorithme de Gauss-Seidel présenté page 27. Naturellement, pour les matrices de niveau supérieur à un, seuls des algorithmes par blocs peuvent être utilisés.

L'algorithme des produits matrice-vecteur

Cette option permet de définir l'algorithme utilisé lors des produits matrice-vecteurs. Choisir un algorithme spécialisé permet d'éviter des tests ou des boucles inutiles. Par exemple, lorsque la structure de la matrice est diagonale, il est inutile d'utiliser deux boucles imbriquées : il n'y a qu'une opération à effectuer par ligne. Une implémentation générique est fournie par défaut.

Le type de conteneurs d'éléments

Si le mode de stockage réel est utilisé, Legolas++ utilise par défaut des vecteurs STL [73] pour stocker les éléments de la matrice. Il est cependant possible de fournir d'autres conteneurs de données.

Le type de conteneur de matrice

À chaque structure de matrice fournie par Legolas++ correspond un conteneur spécifique. Ce conteneur définit le format de stockage de la matrice et peut utiliser un ou plusieurs conteneurs d'éléments. Pour la structure de matrice dense, il pourrait par exemple y avoir des conteneurs effectuant un stockage colonne par colonne, ligne par ligne ou encore selon un ordre de Morton [84].

Afin de pouvoir être passées à Legolas++, ces options doivent être regroupées dans une construction spécifique fournie par Legolas++ :

```
typedef Legolas::InputMatrixOptions<  StorageMode
                                    , InverseAlgorithm
                                    , ProductAlgorithm
                                    , ElementsContainer
                                    , MatrixContainer>  MOptions;
```

Ces options peuvent ensuite être passées à Legolas++ pour obtenir une classe de matrice disposant des fonctionnalités correspondantes :

```
typedef Legolas::GenericMatrixInterface<MDefinition, MOptions>::Matrix Matrix;
```

L'usage reste inchangé par rapport à une matrice utilisant une configuration par défaut. Il est ainsi possible de mettre au point une version de référence de l'application avant de l'optimiser en spécifiant les options des matrices.

5.1.3 Les solveurs

Pour qu'il soit intéressant d'utiliser une bibliothèque comme Legolas++ pour concevoir un solveur, il faut que les algorithmes utilisés puissent tirer partie de la structure des matrices, ce qui n'est généralement pas le cas des algorithmes de résolutions directs car l'étape de factorisation peut impliquer un remplissage important de la matrice. Il est donc préférable de n'utiliser ce type d'algorithmes que pour traiter les niveaux les plus bas de la matrice. L'utilisation d'algorithmes

itératifs aux niveaux supérieurs conduit cependant à multiplier les appels aux algorithmes utilisés pour résoudre les problèmes des niveaux inférieurs. Par exemple, l'algorithme de résolution utilisé par défaut dans Legolas++ est l'algorithme itératif dit de Gauss-Seidel [96, 97] que nous avons présenté page 27. À chaque itération, cet algorithme conduit à résoudre les systèmes linéaires correspondants aux blocs diagonaux de la matrice, et il est important de préserver les résultats partiels qui peuvent être réutilisés d'une itération sur l'autre. Par exemple, dans le solveur de neutronique SP_N implémenté à l'aide de Legolas++ [33], le premier niveau de la matrice est résolu à l'aide d'une factorisation des blocs se trouvant sur la diagonale. Cette factorisation étant très coûteuse, il est nécessaire pour obtenir de bonnes performances de ne pas refaire cette opération à chaque itération.

Les solveurs Legolas++ ont pour objectif de répondre à cette problématique en préservant les états internes des algorithmes entre leurs différentes exécutions. Les solveurs suivent la même construction hiérarchique que les matrices : chaque solveur correspond à un bloc de la matrice et contient d'une part les données nécessaires à son algorithme et d'autre part les sous-solveurs correspondants aux niveaux inférieurs de la matrice. Ces sous-solveurs utilisent l'algorithme défini dans les options des sous-matrices. De cette manière, il est possible de construire récursivement un algorithme de résolution complexe, capable d'exploiter de manière optimale la structure d'une matrice. Il est également possible de changer simplement l'algorithme utilisé à un niveau sans toucher aux autres niveaux ; ce qui peut être extrêmement compliqué avec d'autres approches.

L'annexe C.2 présente l'implémentation de la classe `SymmetricBandedGaussSeidelAlgorithm`, un exemple d'algorithme Legolas++.

5.2 Contraintes pour une version multicible de Legolas++

Dans la section précédente, nous avons présenté Legolas++ et ses principes d'utilisation. Legolas++ permet de mettre au point des solveurs d'algèbre linéaire mettant en œuvre des matrices creuses structurées hiérarchiquement. La structure de ces matrices est décrite en combinant récursivement des structures élémentaires. Legolas++ permet d'associer à chacune de ces structures élémentaires deux algorithmes spécialisés. Le premier algorithme est utilisé pour effectuer les produits matrice-vecteur tandis que le second est utilisé pour résoudre les systèmes d'algèbre linéaire. Les algorithmes correspondant aux structures élémentaires d'une matrice se combinent alors entre eux pour générer un algorithme spécialisé pour cette matrice. Les principes de cette génération sont exposés dans la section 1.3.

Dans cette section, nous présenterons comment concevoir une version multicible de Legolas++. Nous commencerons par rappeler les contraintes pesant sur un code multicible avant de décrire comment adapter Legolas++ pour respecter ces contraintes.

5.2.1 L'expérience MTPS : rappel des contraintes pour un code multicible

Dans le chapitre 4, nous avons présenté MTPS, une bibliothèque permettant d'écrire un code dont les performances sont portables sur différentes architectures matérielles. Pour parvenir à ce résultat, nous avons dû définir un champs d'application restreint pour MTPS. Par exemple, MTPS ne prend en charge que les algorithmes *`vectorisables`* qui appliquent une fonction *`non-divergente`* à une *`collection homogène`* d'éléments. Ces éléments doivent pouvoir être contenus dans une structure contenant un unique tableau par type de données scalaires (`float`, `int`, `double` ...). Les tableaux de données correspondants aux différents éléments de la *`collection homogène`* doivent être de même taille. Cela permet de représenter cette collection sous la forme de structures bidimensionnelles (une par type scalaire), ce qui permet l'adaptation

du format de stockage pour répondre aux particularité des architectures matérielles ciblées. Nous rappelons dans cette section les contraintes à respecter pour exploiter efficacement les capacités des processeurs X86_64 (processeurs fabriqués par INTEL, AMD ou VIA) et des processeurs GF100 (NVIDIA). Ces contraintes sont issues du modèle de programmation de MTPS.

5.2.1.1 Processeur X86_64

Ces processeurs disposent d'unités SIMD (SSE ou AVX). Leur utilisation automatique impose l'utilisation d'algorithmes *vectorisables* qui appliquent une fonction *non-divergente* à une *collection homogène* d'éléments. MTPS fournit des *vues parallèles* sur les éléments d'une *collection homogène*. Ces *vues parallèles* permettent de manipuler plusieurs éléments simultanément et de manière transparente. Elles reposent sur une surcharge des opérateurs pour les types SIMD proposés par les compilateurs (cf. le type `SSEData`, section 2.2.2.3). Afin de permettre l'utilisation des unités SIMD, il faut que les données manipulées soient de taille identique. Ainsi MTPS ne prend-elle pas en charge la vectorisation d'algorithmes mettant en œuvre à la fois des données de type `float` et des données de type `double`.

5.2.1.2 Processeur GF100

À la différence des processeurs X86_64, les processeurs GF100 ne disposent pas d'unités SIMD mais d'unités SIMT. L'obtention de bonnes performances sur ces dernières nécessitent également un réordonnancement des données et l'utilisation d'algorithmes *vectorisables*. En revanche, elles n'imposent pas que les types de données aient la même taille. D'autre part, ces processeurs sont généralement disponibles sous forme de cartes périphériques disposant de leurs propres modules de mémoire RAM. L'utilisation de ces processeurs implique alors de pouvoir copier efficacement les données d'un espace mémoire vers un autre espace mémoire. La latence observée lors de copies entre ces espaces mémoires est importante. De ce fait, les données doivent être sérialisées dans un unique espace mémoire contigu afin de limiter le temps requis par ces copies. Enfin, les différents types de données manipulés doivent également respecter un certain nombre de contraintes proches des contraintes définissant les POD (*Plain Old Data*) [74] en C++. Nous listons une partie de ces contraintes dans la section 2.2.2.1. Les structures de données MTPS respectent ces contraintes.

5.2.2 Famille de problèmes compatibles avec une implémentation multicible

Tous les problèmes qui peuvent être résolus avec Legolas++ ne peuvent pas être multicibles. Certains algorithmes ne peuvent par exemple pas être parallélisés. MTPS ne peut donc pas permettre d'implémenter l'ensemble des cas d'application de Legolas++. Dans cette section, nous allons montrer sous quelles conditions nous pouvons considérer que Legolas++ applique indépendamment une même fonction *non-divergente* à tous les éléments d'une *collection homogène*.

C'est en particulier le cas d'une famille de structures pour lesquelles l'algorithme de résolution du problème $AX = B$ consiste à résoudre des sous-problèmes $A_{i,j}X_i = B_j$ indépendants. Si les différents problèmes correspondant à des sous-matrices peuvent être résolus indépendamment, alors ils ne sont pas couplés. Cela signifie donc que pour chaque ligne et chaque colonne de la matrice, il y a exactement un bloc non-nul [40]. En effet, dans ce cas, la matrice inverse ne fait

40. Il s'agit d'une généralisation des matrices de passage.

pas apparaître de couplage entre les blocs comme l'illustre l'exemple suivant où A,B,C,D et E sont cinq matrices inversibles :

$$\begin{bmatrix} & A & & & \\ & & & B & \\ C & & & & \\ & & D & & \\ & & & E & \end{bmatrix}^{-1} = \begin{bmatrix} & & C^{-1} & & \\ A^{-1} & & & & \\ & & & D^{-1} & \\ & & & & E^{-1} \\ & B^{-1} & & & \end{bmatrix}.$$

Donc, l'ensemble de ces problèmes peut être écrit de sorte à mettre en oeuvre des matrices diagonales par blocs : il suffit pour cela de changer l'ordre des matrices et des vecteurs de manière appropriée.

Or, nous avons vu dans le chapitre précédent que pour qu'une fonction ne soit pas ***divergente*** vis à vis d'un ensemble d'éléments, il faut que le parcours des données soit le même pour chacun de ces éléments. Appliqué à notre problème, cela implique :

- que la structure des différents blocs soit exactement la même (même type de structure et même ***shape***),
- que la suite d'instructions exécutée par les différents algorithmes impliqués ne dépende que de la structure de la matrice.

Supposons que toutes les matrices Legolas++ puissent être sérialisées [41]. Lorsque les différents blocs d'une matrice possèdent exactement la même structure, cette matrice peut être considérée comme une ***collection homogène*** vis à vis des algorithmes permettant de résoudre un sous-problème correspondant à un élément de la diagonale.

Au final, notons $\mathcal{E}$ l'ensemble des problèmes $AX = B$ vérifiant les propriétés suivantes :

- A est une matrice diagonale par blocs,
- la structure de tous ces blocs est exactement la même,
- l'algorithme choisi pour résoudre les problèmes-blocs $A_{ii}X_i = B_i$ n'est pas ***divergent***, c'est à dire que la suite d'instructions correspondante à son exécution ne dépend pas des données.

$\mathcal{E}$ représente l'ensemble des problèmes pour lesquels il est possible de mettre au point une implémentation multicible en appliquant les principes étudiés au chapitre 4.

Dans la suite de ce chapitre, nous ne considérerons cependant aucune restriction sur la structure des blocs de la diagonale. La *figure* 5.1 montre un exemple de matrice correspondant à ces contraintes. Bien que cette classe de problèmes soit très réduite vis à vis de l'ensemble des problèmes pouvant être traités avec Legolas++, elle n'en constitue pas moins une classe de problèmes importante. Par exemple, ce type de structure apparaît à chaque fois que l'on souhaite utiliser une méthode itérative des directions alternées [247] pour résoudre un problème discrétisé suivant une grille cartésienne. Les problèmes résultants d'une discrétisation suivant un maillage irréguliers peuvent toutefois également faire apparaître des structures diagonales par bloc en utilisant un maillage à deux niveaux : le premier niveau de maillage est un maillage irrégulier classiques, mais chacune de ses mailles est à son tour discrétisé suivant un maillage régulier [248, 249].

5.3 Un démonstrateur de Legolas++ multicible

Dans la section précédente, nous avons identifié la classe de problèmes pour laquelle il est possible de fournir une implémentation multicible. Nous allons maintenant nous intéresser aux

41. Cette hypothèse est plus que réaliste : elles sont au moins sérialisées dans la mémoire RAM.

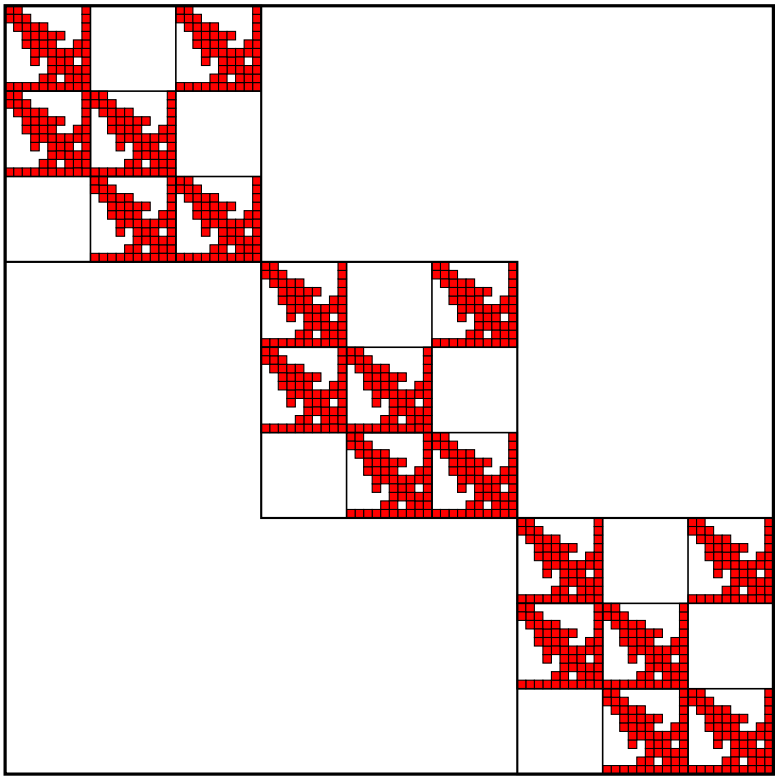


Figure 5.1 : *Exemple d'une matrice pouvant être représentée par une* `collection homogène`*.*

moyens permettant de générer des implémentations adaptées aux différentes cibles conformément aux principes énoncés au chapitre 4.

5.3.1 Structures de données Legolas++ et *vues* parallèles

Dans MTPS, nous parvenons à rendre transparent le format de stockage des ***collections*** en utilisant un mécanisme de ***vues*** permettant de masquer les données à l'utilisateur. Lorsque l'architecture ciblée possède des unités SIMD/SIMT, une ***vue*** peut correspondre à plusieurs objets (un par voie SIMD/SIMT). On parle alors de ***vue parallèle***. Dans cette section, nous présentons comment créer des ***vues*** sur les conteneurs de données Legolas++. Pour cela, nous étudierons comment transformer les structures de données Legolas++ en structures bidimensionnelles puis comment créer des ***vues parallèles***. Nous présenterons successivement dans le cas des vecteurs, des matrices puis des solveurs.

5.3.1.1 Les vecteurs

Un vecteur unidimensionnel est sérialisé par nature. Sérialiser un vecteur multidimensionnel est relativement simple : il suffit d'allouer un tableau suffisamment grand pour contenir tous les éléments du vecteur et d'attribuer à chaque bloc de vecteur un partie de ce tableau. La *figure* 5.2 montre les différences entre des représentations sérialisées et non sérialisées d'un vecteur bidimensionnel. Dans cette section, après avoir illustré les principes de la sérialisation, nous présenterons comment cela permet de transformer des structures de données multidimensionnelles en structure de donnée bidimensionnelles, ce qui permettra de généraliser le principe des ***vues*** de MTPS pour prendre en charge les structures de données multidimensionnelles.

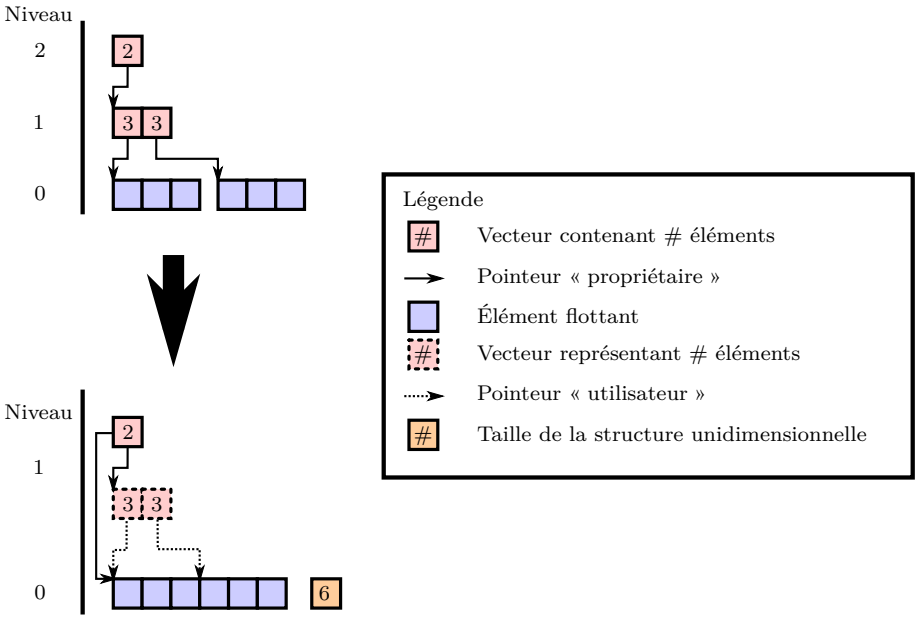


Figure 5.2 : *Sérialisation des données d'une structure à deux niveau.*

Le haut de la *figure* 5.2 représente une implémentation non sérialisée. L'objet représentant le vecteur de niveau deux contient un pointeur vers un tableau de vecteurs de niveau un. Chacun de ces objets contient un pointeur vers un tableau d'éléments flottants (ces éléments sont dits de niveau zéro dans la terminologie Legolas++).

La représentation du bas correspond à une implémentation dont les données sont sérialisées : tous les éléments flottants sont dans un unique tableau. Pour ce faire, l'allocation de ce tableau doit être pris en charge par le vecteur de niveau deux. L'idée est la suivante : lors de la construction du vecteur bidimensionnel, le nombre d'éléments de chaque niveau est calculé à partir du *profil* du vecteur. Un tableau est alors alloué pour chaque niveau. Ensuite, les différents niveaux sont initialisés et pointent vers la portion du tableau de niveau inférieur qui leur est allouée. Ce principe est aisément généralisable à des vecteurs de dimensions supérieures. Afin de limiter l'empreinte mémoire, il est également possible de n'allouer que le tableau contenant les éléments flottants et de générer à la demande les vecteurs de niveaux intermédiaires. C'est la stratégie que nous avons suivi pour implémenter la classe `Vector2D` dans la section 2.2.2.

En appliquant ce principe, un vecteur à N dimensions peut être ramené à une structure bidimensionnelle. Il suffit par exemple de sérialiser les $N-1$ derniers niveaux du vecteur. Pour que cette structure soit rectangulaire, il faut que les différents vecteurs de dimension $N-1$ aient le même profil. La *figure* 5.3 illustre ces concepts en prenant l'exemple d'un vecteur V de dimension 3. Sur cette figure, les couleurs sont utilisées pour identifier les données appartenant aux différents vecteurs de niveau 1. À chaque vecteur de niveau 2 correspond une ligne de la structure bidimensionnelle. Pour que la structure bidimensionnelle soit rectangulaire et compatible avec les principes énoncés dans le chapitre 4, il est nécessaire que tous ces vecteurs de niveau 2 aient le même *profil*. Dans l'exemple de cette figure, le *profil* de chacun de ces vecteurs peut être énoncé comme suit : vecteur de dimension 2 contenant deux vecteurs, le premier contenant trois éléments, le second en contenant quatre. En généralisant la définition présentée section 4.1.3, nous dirons de ces vecteurs de dimension 2 qu'ils sont *semblables*. Soit f une fonction prenant pour argument un vecteur de dimension 2. Le vecteur V peut alors être considéré comme une *collection homogène* vis à vis de f.

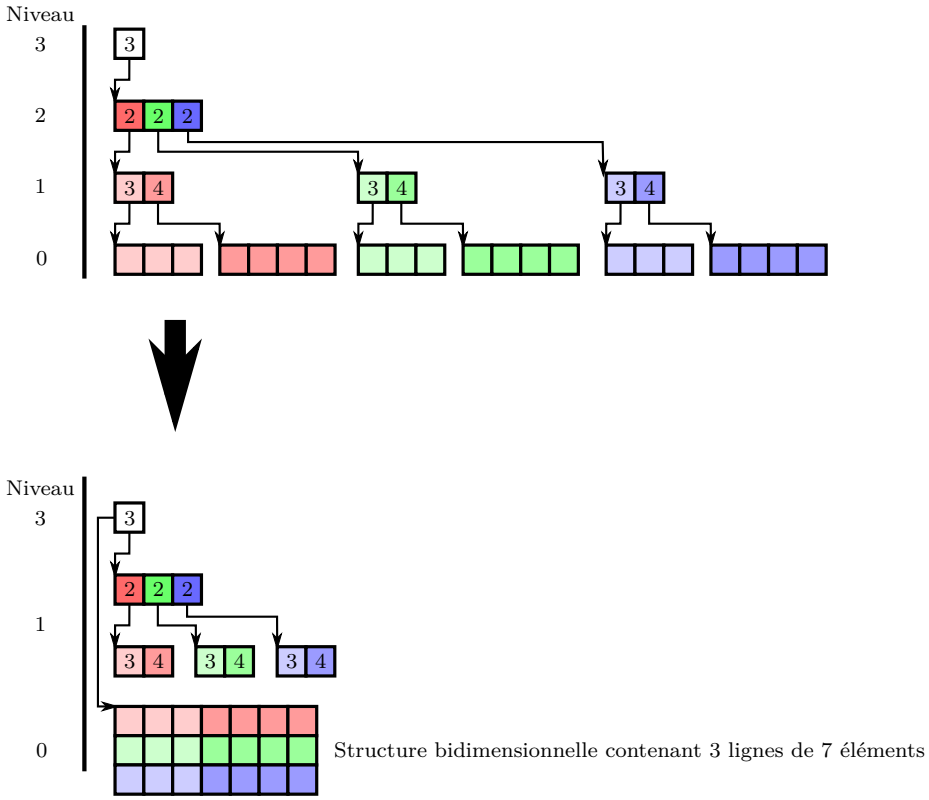


Figure 5.3 : *Transformation du stockage d'un vecteur 3D en structure bidimensionnelle.*

Si f est ***non-divergente***, il est possible d'implémenter f et la ***collection homogène*** à l'aide de MTPS afin d'obtenir un code multicible. Cependant, notre implémentation de MTPS ne peut pas être utilisée directement dans Legolas++ car dans Legolas++, les passages d'objets se font par références tandis que notre implémentation de MTPS repose sur l'idée de générer à la demande des ***vues*** sur ces objets. Nous avons donc ré-implémenté pour les besoins de Legolas++ des conteneurs multicibles spécifiques reposant sur les principes et reprenant le modèle de programmation de MTPS. Une autre solution aurait été de ré-implémenter Legolas++ afin de permettre le passage par ***vues*** générées à la volée. Cette solution étant complexe à mettre en œuvre, nous avons fait le choix de ré-implémenter une partie de MTPS pour les besoins de Legolas++.

Nos vecteurs Legolas++ multicibles sont paramétrés par le type de réels et leur niveau (comme pour les vecteurs Legolas++ actuels), mais également par le niveau où se situent les ***collections homogènes*** : Dans l'exemple de la *figure* 5.3, le vecteur V comporte trois niveaux. Le troisième niveau correspondant à une ***collection homogène*** de vecteurs de niveau deux.

```
const int NumberOfLevels = 3;
const int CollectionLevel=3;
typedef Legolas::MultiVector<float,NumberOfLevels,CollectionLevel> XXXXX;
```

Naturellement, les vecteurs de niveau deux représentant les éléments de la ***collection*** doivent avoir le même ***profil***.

Pour être multicible, notre implémentation de vecteurs multidimensionnels doit permettre d'opérer simultanément sur différents éléments. Ainsi selon l'architecture choisie, l'accès aux éléments de la ***collection homogène*** renverra une ***vue*** représentant soit un vecteur unique, soit plusieurs vecteurs (k si l'architecture cible comporte des unités SIMD/SIMT à k voies).

Tableau 5.1 : *Exemples pour différentes structures de matrices de* `linéarisation` *des indices* (i, j)

Structure	Données de *profil*	$(i, j) \rightarrow [0; T-1]$	T
Dense	$n_{\text{row}}, n_{\text{col}}$	$in_{\text{col}} + j$	$n_{\text{row}} n_{\text{col}}$
Dense symétrique	$n\ (= n_{\text{row}} = n_{\text{col}})$	$\frac{(i-1)(i-2)}{2} + j \qquad (i \geq j)$	$\frac{n(n-1)}{2}$
Bande	$n, l_{\text{inf}}, l_{\text{sup}}$	$i(1 + l_{\text{inf}} + l_{\text{sup}}) + j - i + l_{\text{inf}}$	$n(l_{\text{inf}} + l_{\text{sup}})$
Bande symétrique	n, hbw	$i\text{hbw} + i - j \qquad (i \geq j)$	$n\text{hbw}$

Afin de ne pas stocker les deux types de *vues* (parallèles ou séquentielles), nous avons choisi de stocker des *vues* génériques qui sont transtypées à la demande pour être soit séquentielles, soit parallèles. Dans l'exemple de la *figure* 5.3, sur une machine disposant d'unités SIMD à 3 voies, les *vues* sur les éléments rouges peuvent être transtypées en *vues parallèles* sur les 3 éléments de la *collection*. De ce fait, il n'est pas nécessaire de stocker à la fois les *vues* séquentielles et parallèles. Le comportement par défaut de notre implémentation des vecteurs multidimensionnels est identique à celui des vecteurs Legolas++ actuels. Les *vues* parallèles sont typiquement crées par des algorithmes Legolas++ (correspondant aux squelettes parallèles de MTPS) ou par des utilisateurs avertis.

Dans cette section, nous avons montré comment concevoir des vecteurs multidimensionnels multicibles. Pour y parvenir, nous avons dans un premier temps conçu des vecteurs multidimensionnels dont les données étaient représentées sous la forme de structures bidimensionnelles avant de nous appuyer sur les principes de MTPS pour transformer les vecteurs Legolas++ en *collections homogènes*. Contrairement à MTPS qui génère les *vues* à la demande, nos vecteurs stockent ces *vues*, ce qui permet d'assurer une plus grande compatibilité avec le reste de la bibliothèque Legolas++.

5.3.1.2 Les matrices

Notre démarche pour concevoir des matrices Legolas++ multicibles est la même que pour les vecteurs. Nous commencerons par présenter comment sérialiser les données des matrices à plusieurs niveaux. Nous appliquerons alors le même principe que pour les vecteurs : nous utiliserons des *vues* génériques capables de représenter soit une matrice (la *vue* est dite séquentielle), soit plusieurs (la *vue* est dite parallèle).

La sérialisation des données d'une matrice comportant un niveau revient à établir une correspondance entre l'ensemble des indices (i, j) des éléments non nuls et un intervalle d'entiers $[0; T-1]$. Nous parlerons alors de *linéarisation* des indices (i, j). Les éléments de cette matrice peuvent alors être stockés dans un vecteur de taille T. Le *tableau* 5.1 présente quelques exemples de *linéarisation* pour différentes structures de matrices. Parfois, l'obtention de formules de *linéarisation* simples ne peut se faire sans que T ne soit supérieur au nombre d'éléments non nuls de la matrice. Par exemple, les formules présentées pour les structures « bande » et « bande symétrique » impliquent le stockage d'éléments « inutiles ».

Appliqué à tous les niveaux d'une matrice, ce principe permet de transformer le problème du stockage d'une matrice comportant N niveaux, en un problème de stockage d'un vecteur comportant N niveaux. La *figure* 5.4 représente une matrice à deux niveaux et le vecteur correspondant. La matrice est une matrice de structure « bande symétrique ». Les blocs de la matrice possèdent la même structure. Les éléments noirs représentent les éléments « inutiles » introduits par la formule de *linéarisation* choisie.

En réutilisant le travail effectué précédemment sur les vecteurs, nous avons ainsi défini des conteneurs de matrices sérialisés : pour chaque structure de matrice, un conteneur prend en

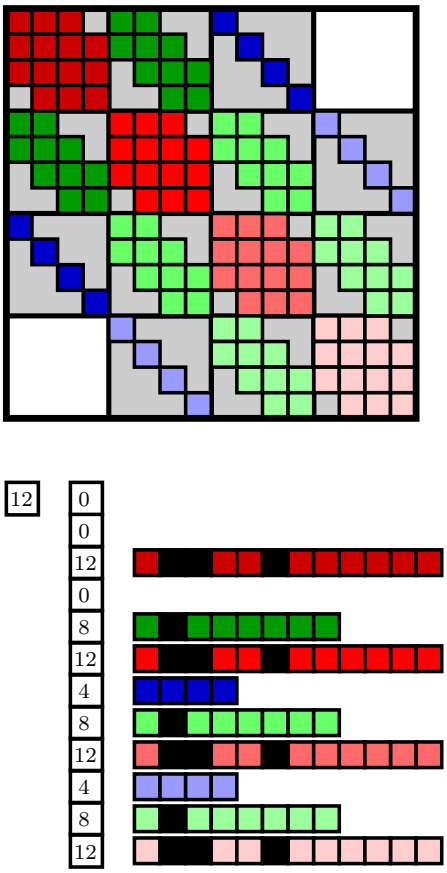


Figure 5.4 : *Transformation d'une matrice à deux niveaux en vecteur à deux niveaux.*

charge la sérialisation des éléments. Cependant, pour pouvoir réutiliser nos travaux sur les vecteurs, nous avons du limiter les possibilités offertes par Legolas++ : il est impossible d'utiliser un conteneur sérialisé à un seul niveau de la matrice. Pour pouvoir utiliser ces conteneurs, il faut que la matrice soit définie comme « stockée à tous les niveaux et que des conteneurs sérialisés soient utilisés à tous les niveaux.

Dans le cadre de cette thèse, nous avons implémenté un conteneur sérialisé permettant de générer des *vues parallèles* : `Legolas::BandedSymmetricFlatMatrixContainer` est spécialisé pour les structures bandes symétriques.

Afin de générer des *vues parallèles*, il faut que la matrice comporte un niveau dont la structure est diagonale et dont les blocs possèdent le même *profil*. Le conteneur linéarisé `Legolas::DiagonalFlatMatrixContainer` peut alors être utilisé. Il définit un stockage optimisé pour les matrices diagonales par blocs et assimilables à des *collections homogènes* de blocs. Ce conteneur crée la structure de donnée bidimensionnelle permettant le réordonnancement des données en fonction de l'architecture ciblée. Il est alors possible de créer des *vues parallèles* permettant d'opérer de manière transparente sur plusieurs blocs de manière simultanée. Lorsque le conteneur `Legolas::DiagonalFlatMatrixContainer` est défini pour plusieurs niveaux de la matrice, la conteneur de plus haut niveau est utilisé pour la parallélisation.

Dans cette section, nous avons montré que le problème posé par la conception de matrices multicibles est équivalent au problème posé par la conception de vecteurs multicibles. Nous avons ainsi étendu les conteneurs parallèles des vecteurs Legolas++ afin de prendre en charge les matrices Legolas++. Comme nous l'avons indiqué précédemment, cela a été rendu possible en limitant l'ensemble des cas pouvant être traités (uniquement les problèmes mettant en œuvre des matrices diagonales par blocs homogènes) et en désactivant certaines possibilités offertes par Legolas++ (les matrices doivent être « stockées » à tous les niveaux). Malgré ces restrictions, l'ensemble des problèmes couverts justifie le travail effectué. En effet, la résolution parallèle de problèmes physique discrétisés suivant un maillage cartésien conduit souvent à la mise en œuvre de matrices diagonales par blocs.

5.3.1.3 Les solveurs

Dans les deux sections précédentes, nous avons vu comment construire des conteneurs capables d'adapter les structures de données selon l'architecture matérielle choisie. Dans cette section, nous expliquons pourquoi nous ne pouvons pas utiliser le même principe et quelles pistes nous proposons pour permettre la mise au point de *vues* parallèles sur les solveurs.

Pour les matrices, nous nous sommes appuyés sur la possibilité offerte par Legolas++ de prendre en charge directement le stockage des éléments des matrices « stockées [42] ». Legolas++ ne fournit actuellement pas de fonctionnalité équivalente pour les solveurs et l'implémentation des solveurs est totalement libre : il peut contenir l'adresse d'une bibliothèque externe à utiliser pour résoudre le système. Afin de pouvoir définir un conteneur de solveur sérialisé, nous devons donc standardiser les données que peut contenir un solveur. À ce jour, nous n'avons pas trouvé de manière simple de standardiser les solveurs sans pour autant limiter fortement les possibilités de l'utilisateur.

Plaçons nous dans le cas d'un solveur ne contenant que des matrices et des vecteurs Legolas++. Ce solveur pourra par exemple contenir la forme factorisée d'une matrice ou une direction de descente dans le cas d'un gradient. Sérialiser ce solveur revient alors à stocker les éléments des matrices et des solveurs dans le même espace mémoire. Comme la sérialisation des matrices peut

42. Les matrices virtuelles utilisent directement les données fournies par l'utilisateur dans la définition de la matrice. Voir la section 5.1.2.3 pour plus de détails.

se ramener au problème de la sérialisation des vecteurs, le problème posé par la sérialisation de ce solveur est équivalent au problème posé par la sérialisation d'un vecteur multidimensionnel. Cependant, pour de nombreux algorithmes, les matrices et les vecteurs nécessaires pour définir l'état du solveur n'ont pas tous le même niveau. Notre implémentation des conteneurs multicibles n'est malheureusement pas compatible avec les structures dont le nombre de niveaux n'est pas constant.

Une solution consisterait à implémenter dans chaque classe de solveur des fonctions permettant de calculer le nombre et le type des éléments à stocker à chaque niveau. Les allocations pourraient dans ce cas être effectuées par le solveur le plus élevé. Chaque solveur disposerait alors d'un espace réservé. Afin de préserver les possibilités de réordonnancement, les solveurs devraient utiliser des *vues* sur ces espaces mémoire.

Nous n'avons pas eu le temps de valider cette approche. De ce fait, il nous a été impossible de prendre en charge les solveurs Legolas++ sur GPU. Ce qui empêche notre démonstrateur de Legolas++ de cibler les processeurs GF100. Nous avons toutefois pu prendre en charge les différents types de processeurs X86_64 (disposant d'unités SSE ou AVX). Dans la prochaine section, nous présentons l'état d'avancement de notre démonstrateur Legolas++ sur ces processeurs.

5.3.2 Un démonstrateur capable de cibler les différentes générations de processeurs X86_64

Afin de pouvoir supporter les processeurs X86_64, notre démonstrateur doit permettre la création de *vues* parallèles permettant de manipuler plusieurs objets à la fois. Jusqu'à présent, nous avons construit des *vues* travaillant sur les éléments d'une *collection homogène*. N'ayant pas pu sérialiser les solveurs, nous n'avons pas été en mesure de créer de telles *vues* parallèles sur des solveurs. Créer de telles *vues* permettrait de manipuler plusieurs solveurs (un par voie SIMD) de manière transparente.

Afin de contourner ce problème, nous avons choisi de définir des solveurs capables d'utiliser des *vues* parallèles sur les vecteurs et les matrices. Ceci permet à ces solveurs de posséder un « état parallèle », c'est à dire un état correspondant à plusieurs problèmes (un par voie SIMD). Notons toutefois que cette approche n'est possible que pour les solveurs dont l'intégralité du code est fourni à Legolas++. En effet, dans les autres cas, il nous est impossible de modifier les données définissant l'état du solveur. Cette vérification doit être effectuée par l'utilisateur.

De cette manière, nous avons pu obtenir un démonstrateur de Legolas++ capable de cibler les différentes architectures de la famille X86_64. Le choix de l'architecture cible se fait simplement en utilisant les options de compilation. En effet, les compilateurs définissent des macros correspondant à l'architecture ciblée. Par exemple, l'option `-mavx` de g++ permet de cibler les architecture disposant d'unités AVX. Cette option définit également la macro `__AVX__`. Nous utilisons ces macros pour définir le format de stockage et le type de *vues* adaptés à l'architecture.

5.3.3 Utilisation et limitations du démonstrateur

Nous l'avons vu, la spécialisation du format de stockage d'une matrice passe par l'utilisation de conteneurs spécialisés prenant en charge la sérialisation des données. L'utilisation de tels conteneurs à un niveau donné implique que des conteneurs compatibles soient utilisés pour tous les niveaux inférieurs et que le mode de stockage des matrices soit le mode « réel ».

Le second point nécessaire pour une utilisation efficace de différentes architecture est la propriété de *non-divergence* des algorithmes appliqués. En effet, cette condition permet de

cibler les unités SIMD (processeurs X86_64) et garantit un fonctionnement optimal des unités SIMT (processeurs GF100). La vérification de cette propriété n'est pas prise en charge par notre démonstrateur : c'est à l'utilisateur de s'assurer que les algorithmes ne sont pas *divergents*.

Lorsque l'utilisateur a vérifié que ces conditions étaient réunies, il peut utiliser les algorithmes `Legolas::SIMDDiagInverse` et `Legolas::SIMDDiagonalMatrixVectorProduct` et le conteneur `Legolas::DiagonalFlatMatrixContainer` au niveau de la *collection*. Ces deux algorithmes correspondent aux squelettes parallèles de MTPS et prennent alors en charge le fait de générer des *vues* parallèles. Notons que l'utilisation du conteneur `Legolas::DiagonalFlatMatrixContainer` nécessite l'utilisation de conteneurs linéarisés pour les niveaux inférieurs de la matrice.

Finalement, notre démonstrateur permet de paralléliser automatiquement sur les processeurs X86_64 les deux solveurs présentés dans la section 5.1. Dans le prochain chapitre, nous analyserons de manière détaillée les performances obtenues par notre démonstrateur.

Ne me parlez pas de vos efforts. Parlez-moi de vos résultats.

James J. Ling (1922 – 2004)
Industriel américain et ancien dirigeant de Ling-Temco-Vought

Chapitre 6

Analyse des performances du démonstrateur

Dans le chapitre 1, nous avons présenté les intérêts et les difficultés de séparer les activités de développement des fonctionnalités des codes de calcul d'un part et les activités d'optimisation du code pour un ensemble de plates-formes matérielles données.

Dans le chapitre 2, après avoir rappelé l'apport des différentes architectures matérielles, nous avons présenté les difficultés liées au développement d'un code multicible. Nous avons identifié deux difficultés majeures. La première concerne l'expression du parallélisme : selon l'architecture ciblée, les paradigmes de programmation diffèrent, ce qui rend difficile la mise en commun de portions de codes. La seconde difficulté que nous avons identifié concerne l'adaptation des structures de données. En effet, afin de pouvoir exploiter au mieux les caractéristiques des architectures matérielles, les structures de données doivent être modifiées. Cela peut rendre complexe l'interopérabilité entre des portions de code s'exécutant sur des architectures différentes.

Dans le chapitre 3, nous avons étudié différentes approches permettant de développer des applications parallèles portables sur différentes architectures. À ce jour, aucun outil disponible ne prend en charge l'adaptation des structures de données pour les cibles matérielles que nous ciblons.

Dans le chapitre 4, nous avons décrit le modèle de programmation de MTPS, une bibliothèque C++ permettant la parallélisation des opérations `vectorisables` appartenant au même `contexte vectoriel`. Nous avons ensuite présenté comment MTPS prend en charge l'adaptation des structures de données et unifie l'expression du parallélisme.

Dans le chapitre 5, nous avons présenté Legolas++, une bibliothèque permettant de décrire et de manipuler des matrices creuses et structurées. Après avoir remis en cause certains choix de conception de MTPS, Nous avons ensuite présenté comment nous avons adapté Legolas++ au modèle de programmation de MTPS afin d'aboutir à une version parallèle capable de cibler automatiquement les processeurs disposant d'unités vectorielles.

Dans ce chapitre, nous présentons les performances obtenues avec les différentes approches présentées dans ce document. Les performances des implémentations Legolas++ et MTPS seront comparées aux performances des implémentations optimisées à la main introduites dans le chapitre 2.

Sommaire

Dans les deux exemples qui suivent, nous allons résoudre un système linéaire $\mathbf{AX} = \mathbf{B}$, avec deux structures de matrices différentes pour $\mathbf{A}$, impliquant deux algorithmes de résolution distincts. La matrice $\mathbf{A}$ est dans les deux cas une matrice diagonale par blocs. Elle présentera deux niveaux dans le premier exemple et trois dans le second exemple. Dans le premier cas, les blocs auront la structure de type bande symétrique présentée dans les chapitres 2 et 4 (cf. *figure* 6.1). Dans le second exemple, la structure sera plus complexe : la structure d'un bloc comportera deux niveaux, chacun de type bande symétrique (cf. *figure* 6.19). Dans les deux cas, les sous-problèmes $A_{ii}X_i = B_i$ peuvent être traités indépendamment et parallèlement aux autres. Le problème $\mathbf{AX} = \mathbf{B}$ peut en effet être considéré comme une *collection*[43] vis à vis de la fonction de résolution d'un problème $A_{ii}X_i = B_i$.

6.1 Premier cas : les blocs ont une structure bande symétrique

Le premier exemple que nous allons traiter est issu du solveur de neutronique SP_N et correspond à l'exemple introduit dans la section 2.2. Les blocs de $\mathbf{A}$ ont une structure bande symétrique. Comme dans la section 2.2, l'algorithme utilisé pour résoudre un sous-problème $A_{ii}X_i = B_i$ consiste à décomposer le bloc A_{ii} selon une forme LDL^T de la factorisation de Cholesky puis à effectuer une opération dite de « descente-remontée ». Seule cette dernière opération nous intéresse pour effectuer nos analyses de performances. Il est en effet classique de ré-utiliser plusieurs fois le résultat de la décomposition pour résoudre des problèmes possédant des seconds membres différents. Ce sera le cas pour résoudre le problème présenté dans la section 6.2.

Dans cette section, nous allons tout d'abord présenter la structure de la matrice $\mathbf{A}$ ainsi que les différents paramètres permettant de la définir précisément. Nous déterminerons ensuite les performances maximales théoriques attendues en fonction des caractéristiques des machines de tests. Finalement, nous comparerons les performances obtenues par les différentes approches avec les performances théoriques.

6.1.1 Structure de la matrice A et paramètres du problème

La matrice $\mathbf{A}$ est une matrice diagonale par bloc dont les blocs possèdent une structure bande symétrique. Trois paramètres permettent de caractériser cette structure de matrice :

- le nombre de blocs n_b,
- la taille des blocs t_b,
- la demi largeur de bande des blocs hbw.

La matrice représentée sur le *figure* 6.1 est caractérisée par les valeurs suivantes :

n_b	t_b	hbw
3	20	6

Dans le solveur SP_N, n_b est généralement compris entre 1×10^4 et 3×10^5. La *figure* 6.2 représente le nombre d'opérations réalisées par seconde sur ${}^{32}_{2\times4}$Nehalem en fonction de n_b lorsque t_b et hbw valent respectivement 25 et 2. Nous avons utilisé l'implémentation séquentielle présentée section 2.2. Nous pouvons constater que pour un nombre de blocs compris entre 1×10^4 et 1×10^5 les performances ne dépendent pas du nombre de blocs. Le même constat peut se généraliser aux autres valeurs des ces deux paramètres, aux autres implémentations (parallèles et vectorisées) et aux autres machines de tests. Par conséquent, nous ignorerons ce paramètre dans la suite.

43. voir définition section 4.1.4.1

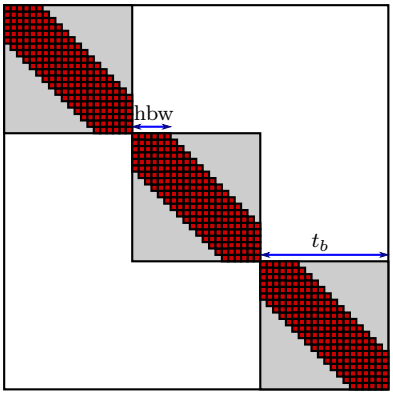


Figure 6.1 : *Structure de la matrice* **A** *utilisée dans le premier cas test.*

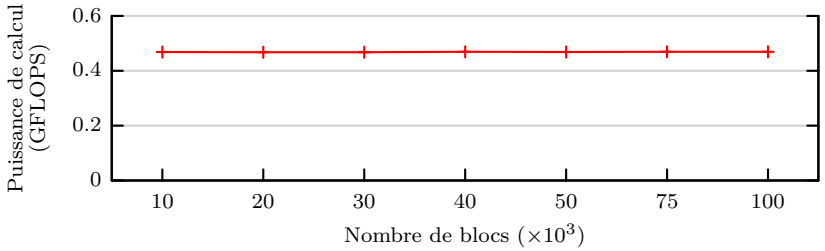


Figure 6.2 : *Performances de la descente-remontée séquentielle en fonction du nombre de blocs de la matrice* **A** *sur la machine* $_{2\times4}^{32}$*Nehalem.*

La section suivante présente un modèle donnant les performances d'une implémentation idéale de la descente-remontée sur les différentes architectures.

6.1.2 Performances d'une implémentation idéale

Une unité de calcul ne peut effectuer une opération que sur des données disponibles en registres. Si le débit de données n'est pas suffisant pour alimenter les unités de calcul, ces dernières seront inactives une partie du temps. Dans ce cas, les performances sont limitées par la bande passante de la RAM.

À l'inverse, les transferts de données ne peuvent être initiés que si des instructions de transferts sont exécutées. S'il s'écoule trop de temps entre deux instructions d'accès à la mémoire, le réseau d'interconnexion RAM/processeur sera sous-exploité. Les performances sont alors limitées par la puissance de calcul du processeur.

Partant de ce constat, nous établissons un modèle de performance simplifié. Ce modèle repose sur les notions d'*intensité arithmétique* et d'*intensité arithmétique critique*.

Definition 10. L'*intensité arithmétique* i_a est définie par NVIDIA comme le ratio entre le nombre d'opérations et le nombre d'accès mémoire d'une portion d'application [133].

Definition 11. L'*intensité arithmétique critique* i_c caractérise l'équilibre entre la puissance de calcul d'un processeur et la bande passante de la RAM. Cette intensité correspond au ratio entre le nombre d'opérations réalisables en une seconde et la quantité de données transférables entre la RAM et le processeur durant ce même temps.

Afin de déterminer l'élément limitant les performances d'un noyau de calcul pour une machine de calcul donnée, il suffit alors de comparer l'*intensité arithmétique* du noyau de calcul avec l'*intensité arithmétique critique* de la machine de calcul.

- Si i_a est inférieur à i_c, alors les performances sont limitées par la bande passante mémoire. L'obtention de meilleures performances nécessite alors de minimiser le nombre d'accès à la mémoire RAM. Pour cela, il est possible de choisir des algorithmes permettant une meilleure réutilisation des données. Une autre approche consiste à recalculer à la volée des éléments plutôt que de les lire en mémoire. C'est dans ce but que la possibilité de ne pas stocker les matrices Legolas++ a été offerte.
- Si au contraire i_a est supérieur à i_c, les performances de l'application sont limitées par la puissance de calcul. Notons que dans ce cas, des efforts supplémentaires d'implémentation sont requis pour garantir une utilisation optimale du processeur. En effet, l'utilisation d'algorithmes maximisant l'intensité arithmétique (comme les algorithmes dits « *cache oblivious* » [250]) n'est pas suffisant pour garantir des performances optimales. La bibliothèque MTL4 [251] permet l'utilisation d'un algorithme récursif minimisant le nombre d'accès à la mémoire et maximisant donc l'intensité arithmétique. Cependant, cela n'est pas suffisant pour garantir des performances optimales [252]. Dans son article détaillant les choix mis en œuvre dans son implémentation des BLAS, Kazushige GOTO donne divers éléments à prendre en compte pour exploiter pleinement les processeurs [54]. Par exemple, notre modèle de performance ne prend en compte qu'un seul des trois niveaux de cache, il est donc optimiste par nature. De plus, notre implémentation ne prend pas en compte ni le TLB [44] ni l'ordre de parcours des données. Ces deux éléments induisent des latences non prises en compte par notre modèle lors des accès aux données. De plus, au contraire des opérations mettant en œuvre des matrices denses, l'algorithme que nous étudions nécessite plusieurs boucles imbriquées ; ce qui implique des ruptures du pipeline d'instructions [45] et donc une augmentation de la latence dans l'exécution des opérations de calcul.

Nous allons donc maintenant calculer l'*intensité arithmétique* de la descente-remontée. Pour cela, nous devons distinguer l'architecture X86_64 de l'architecture GF100. En effet, la présence d'un cache de grande capacité sur l'architecture X86_64 permet d'éviter certains accès mémoire. Nous allons donc distinguer ces deux cas pour le nombre d'accès à la mémoire. Pour l'architecture X86_64, nous supposons que l'ensemble des données pour un bloc tient en cache et ne comptabilisons donc que les accès correspondants à une seule lecture du couple matrice-vecteur [46] et à l'ecriture du résultat dans le même vecteur. Pour l'architecture GF100, tous les accès sont comptabilisés car les différents niveaux de mémoire cache sont trop petits pour avoir un impact.

44. Le Translation Lookaside Buffer, ou TLB, est une mémoire cache du processeur utilisé par l'unité de gestion mémoire dans le but d'accélérer la traduction des adresses virtuelles en adresses physiques.
Source : http://fr.wikipedia.org/wiki/Translation_Lookaside_Buffer

45. Un pipeline est un élément d'un circuit électronique dans lequel les données avancent les unes derrière les autres, au rythme du signal d'horloge. Dans la microarchitecture d'un microprocesseur, c'est plus précisément l'élément dans lequel l'exécution des instructions est découpée en étapes. Ce découpage permet d'exécuter la première étape d'une instruction avant que l'instruction précédente ait fini de s'exécuter. Le premier ordinateur à utiliser cette technique fut l'IBM Stretch, conçu en 1958.
Source : http://fr.wikipedia.org/wiki/Pipeline_(informatique)

46. Nous comptabilisons dans ce cadre les accès non explicites correspondant aux éléments fantômes de la matrice. Ces éléments permettent de simplifier les calculs d'indices mais entraînent quelques accès supplémentaires.

Notons n_{Ops} le nombre d'opérations flottantes et $n_{a,\text{architecture}}$ le nombre d'accès à la RAM.

$$
\begin{aligned}
n_{\text{Ops}} &= n_b(4t_b\text{hbw} - 2\text{hbw}^2 - 3t_b + 2\text{hbw}) \\
n_{a,\text{X86\_64}} &= n_b(t_b\text{hbw} + 2t_b) \\
n_{a,\text{GF100}} &= n_b(6t_b\text{hbw} - 3\text{hbw}^2 - 4t_b + 3\text{hbw}) \\
i_{a,\text{X86\_64}} &= \frac{4t_b\text{hbw} - 3t_b - 2\text{hbw}^2 + \text{hbw}}{t_b\text{hbw} + 2t_b} \\
i_{a,\text{GF100}} &= \frac{4t_b\text{hbw} - 3t_b - 2\text{hbw}^2 + 2\text{hbw}}{6t_b\text{hbw} - 4t_b - 3\text{hbw}^2 + 3\text{hbw}}
\end{aligned}
$$

Nous pouvons aisément démontrer que $i_{a,\text{X86\_64}}$ et $i_{a,\text{GF100}}$ croissent avec t_b et hbw. De plus, lorsque ces grandeurs tendent vers l'infini et que t_b est grand devant hbw, $i_{a,\text{X86\_64}}$ et $i_{a,\text{GF100}}$ tendent respectivement vers 4 et 2/3. La *figure* 6.3 représente $i_{a,\text{X86\_64}}$ et $i_{a,\text{GF100}}$ pour différentes valeurs de t_b et hbw.

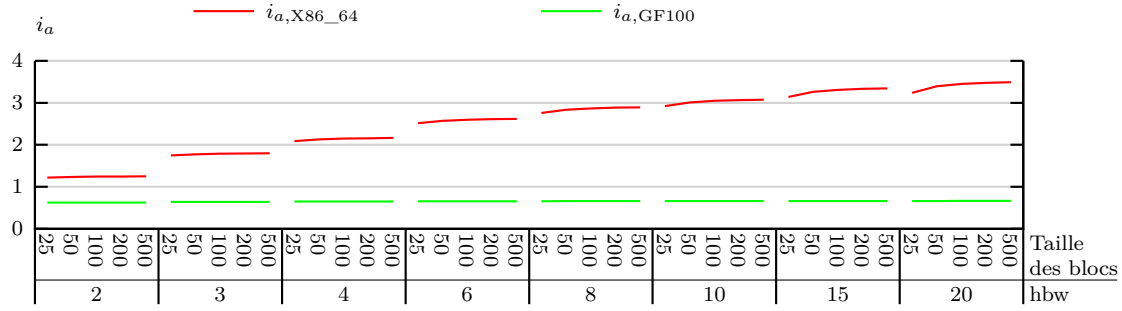


Figure 6.3 : *Intensité arithmétique sur CPU pour différentes configurations du problème.*

Les figures 6.4 à 6.6 comparent i_a et i_c pour chaque architecture testée. Pour le calcul de i_c, nous avons utilisé les meilleures performances de calcul et de débit de données mesurées sur la machine selon la méthodologie détaillée section 2.3. Ces mesures sont présentées dans les tableaux 2.2 et 2.3. Pour les deux machines à base d'architecture X86_64 (figures 6.4 et 6.5), deux valeurs de i_c sont présentées. La plus petite, tracée en bleu clair, correspond à la valeur de i_c calculée pour une exécution séquentielle tandis que la seconde correspond à la valeur de i_c calculée pour une exécution complètement parallélisée (multicœur et SIMD). Pour ces deux machines, la valeur de i_c correspondant à une exécution séquentielle de la descente-remontée est inférieure ou très proche de i_a. Dans ce cas, c'est donc la puissance de calcul du processeur qui limite les performances. Lorsque la descente-remontée est parallélisée, i_c est supérieure à i_a et les performances sont limitées par la bande passante mémoire. Sur la *figure* 6.6, nous n'avons représenté i_c que pour une exécution parallèle (sur GPU, le faire pour une exécution séquentielle n'a aucun sens). Sur cette architecture, i_c est supérieure à i_a et les performances sont limitées par la bande passante mémoire.

Nous pouvons déduire de cette analyse des performances atteignables sur chaque architecture par une implémentation idéale. Notons $F_{\text{architecture}}$ et $D_{\text{architecture}}$ la puissance de calcul maximale et le débit de données maximal mesurés sur une architecture donnée. Le temps d'exécution théorique $t_{\text{architecture}}$ peut alors être calculé par :

$$
t_{\text{architecture}} = \begin{cases} \frac{n_{\text{Ops}}}{F_{\text{architecture}}} & \text{si } i_a > i_c \\ \frac{n_{a,\text{architecture}}}{D_{\text{architecture}}} & \text{si } i_a < i_c \end{cases}
$$

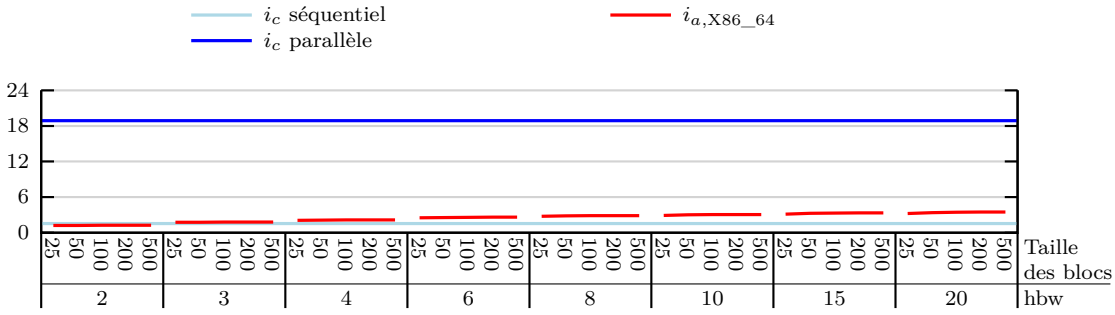


Figure 6.4 : *Comparaison entre i_a et les intensités arithmétiques critiques de ${}^{32}_{2\times4}$Nehalem.*

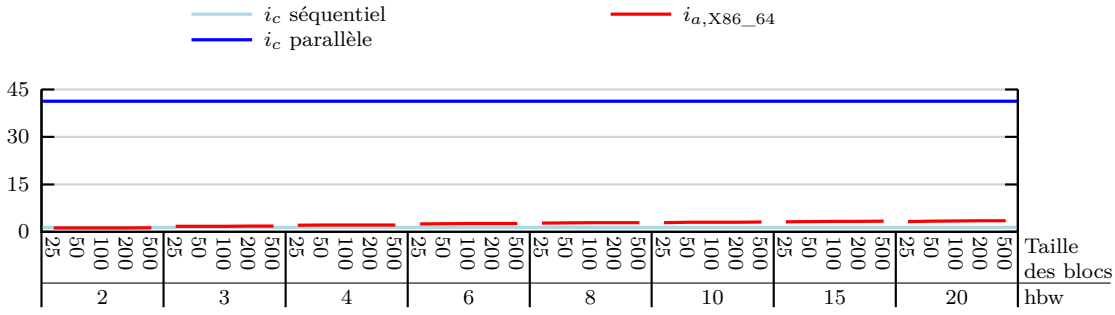


Figure 6.5 : *Comparaison entre i_a et les intensités arithmétiques critiques de ${}^{32}_{1\times4}$SandyBridge.*

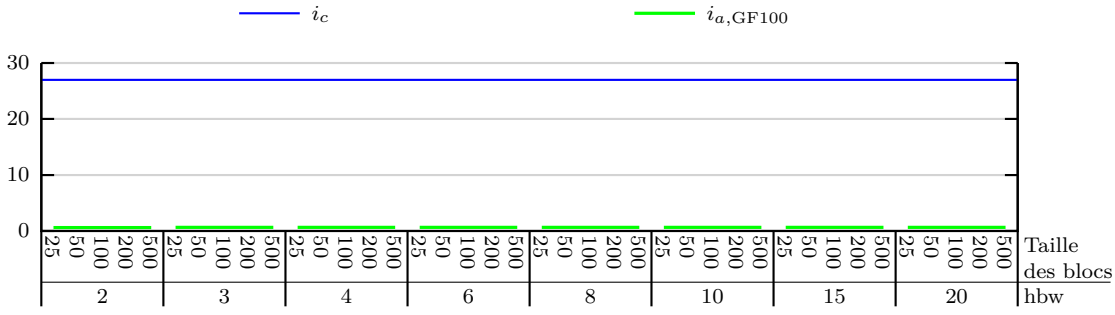


Figure 6.6 : *Comparaison entre i_a et les intensités arithmétiques critiques de Fermi.*

Ce modèle permet donc d'estimer le temps requis pour résoudre le problème pour une implémentation idéale aussi bien optimisée que les bibliothèques constructeur utilisées pour effectuer nos mesures de performances maximales (cf. section 2.3). Nous utiliserons ce modèle de performances dans la prochaine section afin d'estimer l'optimalité de notre implémentation optimisée « à la main ».

6.1.2.1 Performances de l'implémentation optimisée « à la main »

Dans cette section, nous étudierons les performances obtenues avec l'implémentation présentée dans la section 2.2. L'implémentation X86_64 utilise OpenMP pour le parallélisme multicœur et les fonctions intrinsèques du compilateur pour générer des instructions SIMD (SSE ou AVX). L'implémentation GF100 utilise CUDA C/C++.

Les figures 6.7 et 6.8 représentent les performances séquentielles obtenues sur nos différentes machines. Les performances sont exprimées en GFlops et obtenus d'après la formule suivante :

$$\text{performances} = \frac{1}{1024^3}.\frac{\text{nombre d'opérations}}{\text{temps d'exécution}}$$

Dans les deux cas, les performances de l'implémentation idéale sont supposées limitées par la puissance de calcul quand hbw est supérieur à 2 (et par la bande passante de la mémoire RAM si hbw vaut 2). Notre modèle ne prend en compte que deux facteurs de limitation des performances (la bande passante de la RAM et la puissance de calcul du processeur). À cause de l'ignorance des autres facteurs de limitation des performances (latence des accès mémoire, indisponibilité de registres, suite d'instructions mal prise en charge par le pipeline, défauts des niveaux inférieurs du cache ou du TLB...) les performances mesurées sont comprises entre 20% et 60% des performances attendues, ce qui est très satisfaisant : ces résultats sont à comparer aux 67% obtenus par MTL4 dans sa comparaison par rapport à une implémentation optimale pour un produit de matrices denses [252].

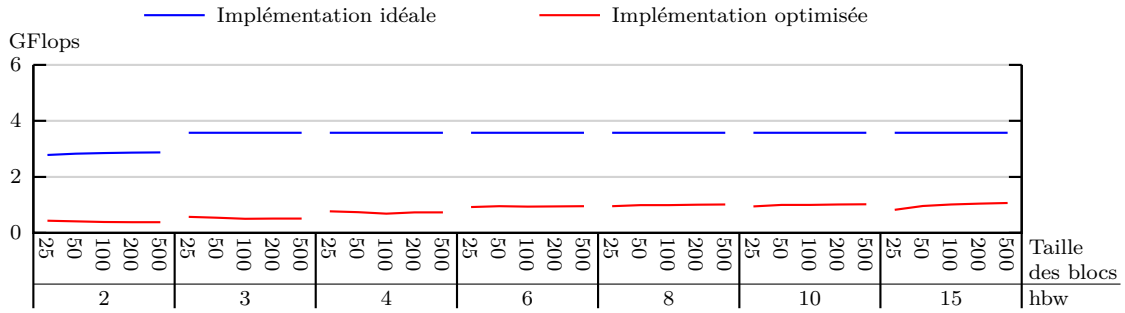


Figure 6.7 : *Performances de l'implémentation optimisée sur* $_{2\times4}^{32}$*Nehalem (exécution séquentielle).*

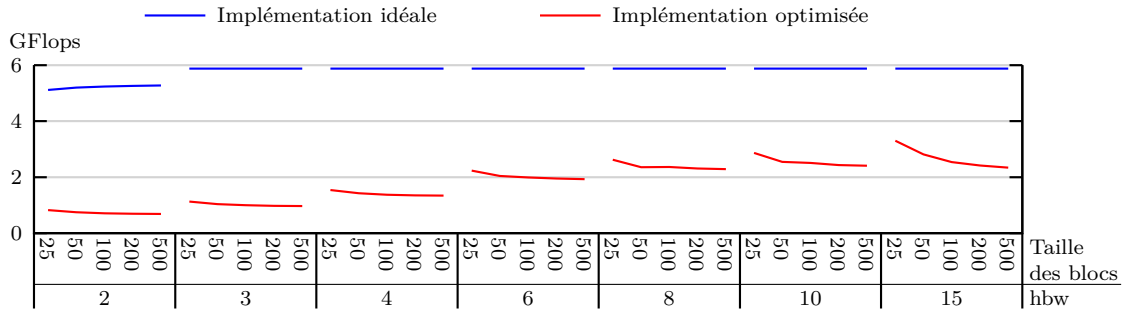


Figure 6.8 : *Performances de l'implémentation optimisée sur* ${}_{1\times4}^{32}$*SandyBridge (exécution séquentielle).*

Les figures 6.9 à 6.11 représentent les performances mesurées pour des exécutions parallèles. Les performances de l'implémentation idéale sont cette fois limitées par la bande passante de la RAM. Notre implémentation optimisée est alors plus proche de l'implémentation idéale : les performances mesurées correspondent à plus de 80% des performances idéales. Nous pouvons donc raisonnablement considérer que cette implémentation optimisée constitue une bonne référence afin de comparer l'efficacité des autres approches présentées dans ce manuscrit.

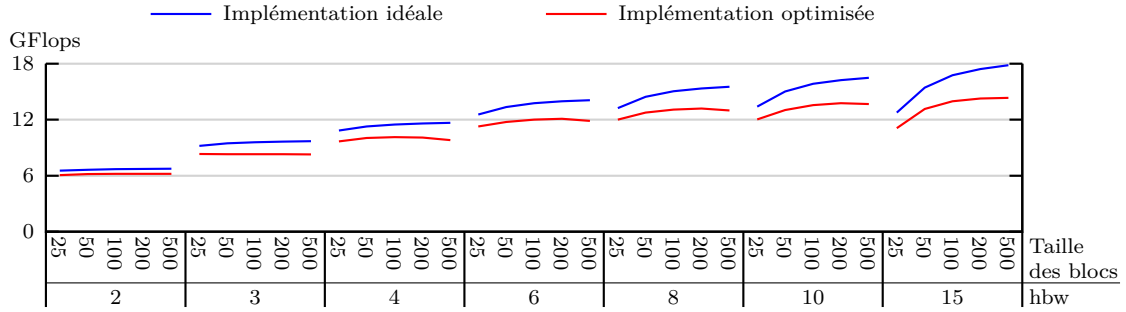


Figure 6.9 : *Performances de l'implémentation optimisée sur* ${}_{2\times4}^{32}$*Nehalem (exécution parallèle).*

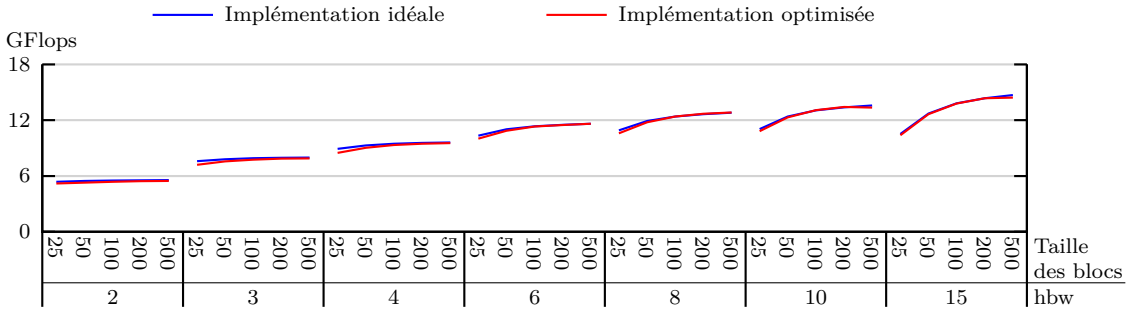


Figure 6.10 : *Performances de l'implémentation optimisée sur* ${}_{1\times4}^{32}$*SandyBridge (exécution parallèle).*

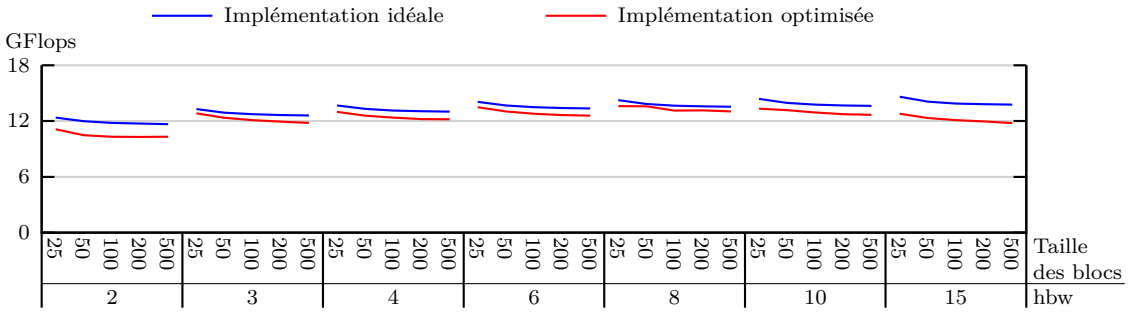


Figure 6.11 : *Performances de l'implémentation optimisée sur Fermi.*

6.1.3 Performances de MTPS

Dans le chapitre 4, nous avons présenté comment MTPS permet de générer des exécutables optimisés pour différentes architectures matérielles à partir d'une même description des structures de données manipulées et de la structure du parallélisme. Nous allons dans cette section comparer les performances obtenues avec MTPS à celles de nos implémentations de référence.

Les figures 6.12 à 6.14 représentent les performances mesurées avec MTPS et avec nos implémentations de référence. Les performances obtenues avec MTPS sont légèrement moins bonnes dans les cas où les blocs sont petits. Cela s'explique par le fait que le temps nécessaire pour effectuer une descente-remontée augmente linéairement avec la taille des blocs. Or, la tâche élémentaire parallélisée par MTPS correspond à cette descente-remontée. Plus le temps nécessaire à l'exécution de ces tâches est important, plus les surcoûts introduits par MTPS sont amortis. Néanmoins, les surcoûts de MTPS comparés à notre implémentation optimisée sont minimes.

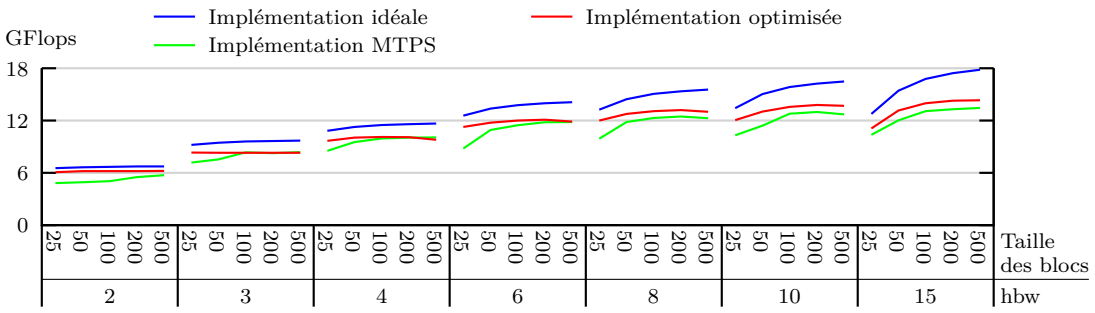


Figure 6.12 : *Performances de MTPS sur ${}_{2\times4}^{32}$Nehalem (parallèle).*

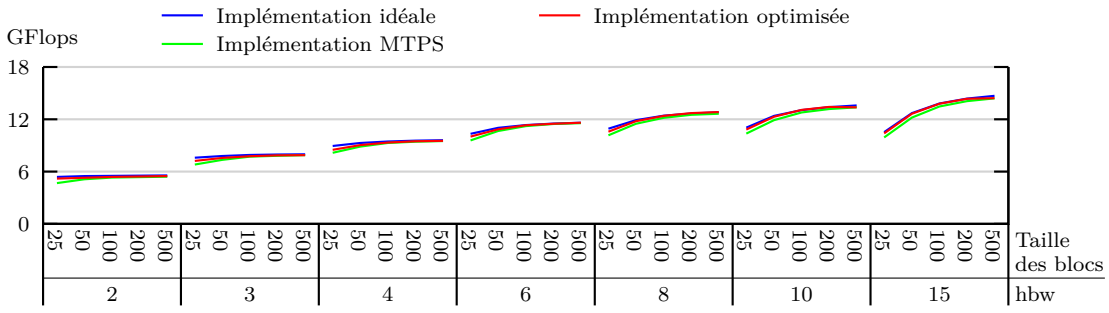


Figure 6.13 : *Performances de MTPS sur ${}_{1\times4}^{32}$SandyBridge (parallèle).*

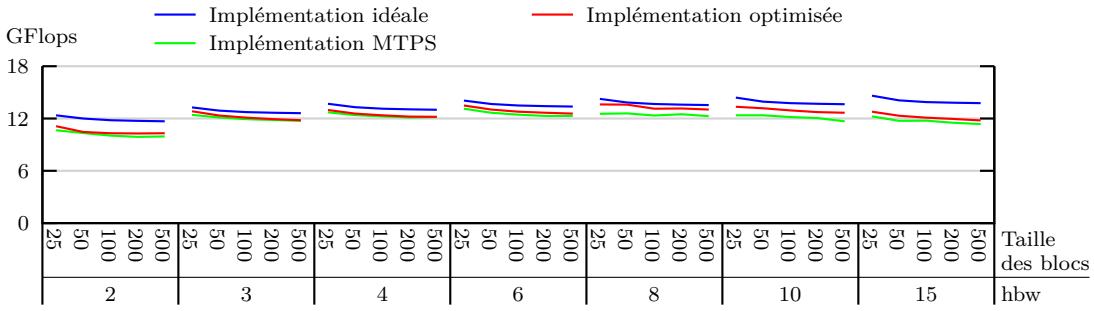


Figure 6.14 : *Performances de MTPS sur Fermi.*

- Sur la machine ${}^{32}_{2\times4}$Nehalem, les écarts de performances entre MTPS et notre implémentation optimisée sont de 7% en moyenne (20% dans les cas les plus défavorables correspondant aux problèmes les plus petits). Finalement, les performances obtenues avec MTPS correspondent à plus de 70% des performances qui auraient été obtenues avec une implémentation idéale et plus de 79% des performances de notre implémentation optimisée à la main.
- Sur la machine ${}^{32}_{1\times4}$SandyBridge, les écarts sont beaucoup plus faibles sans que nous soyons capables d'interpréter cette différence de comportement. Les écarts de performances entre MTPS et notre implémentation optimisée sont en moyenne de 2.3% (10% dans les cas les plus défavorables correspondant aux problèmes les plus petits). Finalement, les performances obtenues avec MTPS correspondent à plus de 87% des performances qui auraient été obtenues avec une implémentation idéale et plus de 90% des performances de notre implémentation optimisée à la main.
- Sur la machine Fermi, les écarts de performances entre MTPS et notre implémentation optimisée sont de 4% en moyenne (8% dans les cas les plus défavorables correspondant aux problèmes les plus grands). Finalement, les performances obtenues avec MTPS correspondent à plus de 83% des performances qui auraient été obtenues avec une implémentation idéale et plus de 92% des performances de notre implémentation optimisée à la main.

6.1.4 Performances du démonstrateur Legolas++ multicible

Legolas++ est une bibliothèque d'algèbre linéaire spécialisée pour les problèmes mettant en œuvre des matrices creuses structurées. Dans le chapitre 5, nous avons présenté la conception d'un démonstrateur de Legolas++ capable de spécialiser automatiquement les structures de données et le type de parallélisme en fonction de l'architecture matérielle. Cette implémentation reprend une partie des concepts de MTPS et permet de cibler des processeurs X86_64 comportant des unités vectorielles de générations différentes (SSE ou AVX). Ce démonstrateur permet donc de paralléliser et de vectoriser automatiquement les opérations d'algèbre linéaire écrites avec Legolas++.

Les figures 6.15 et 6.16 représentent les performances obtenues sur les machines ${}^{32}_{2\times4}$Nehalem et ${}^{32}_{1\times4}$SandyBridge. Nous pouvons constater que ces performances sont très proches de celles obtenues avec les implémentations optimisées. Sur la machine ${}^{32}_{2\times4}$Nehalem, le démonstrateur Legolas++ permet même l'obtention de meilleures performances que MTPS. Nous pensons que cela s'explique par le fait que Legolas++ ne reprend pas l'intégralité des concepts introduits par MTPS. Legolas++ contient par conséquent moins d'indirections que MTPS, ce qui pourrait expliquer que les surcoûts soient moindres.

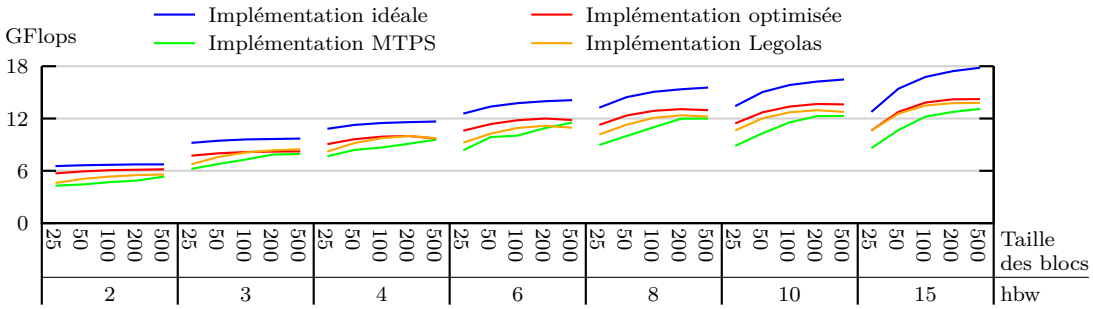


Figure 6.15 : *Performances du démonstrateur Legolas++ sur* ${}_{2\times4}^{32}$*Nehalem (parallèle).*

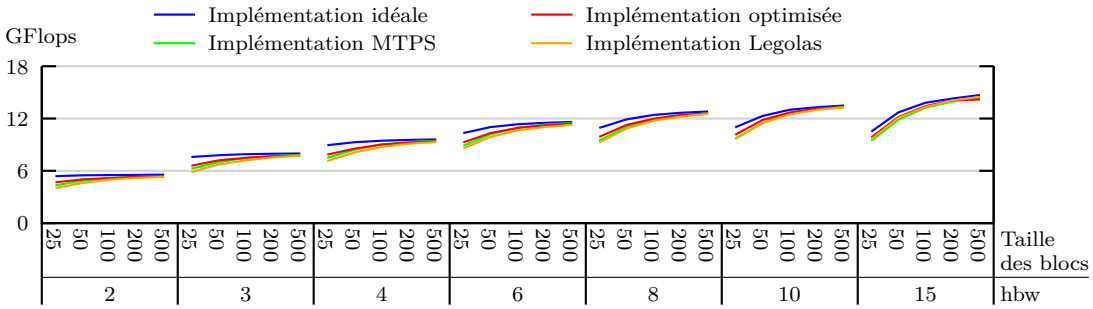


Figure 6.16 : *Performances du démonstrateur Legolas++ sur* ${}_{1\times4}^{32}$*SandyBridge (parallèle).*

6.1.5 Bilan : accélérations obtenues

Parler des facteurs d'accélération (*speedup*) obtenus est toujours discutable. Non seulement, un facteur d'accélération peut être artificiellement amélioré en partant d'une implémentation de référence inefficace, mais il est également impossible de généraliser les gains d'accélérations obtenus pour un problème donné sur une machine donnée à d'autres problèmes ou d'autres machines.

Néanmoins, nous nous sommes placés dans un contexte où une telle analyse peut être menée raisonnablement. En effet, nous avons montré dans les sections précédentes que notre implémentation séquentielle de référence est raisonnablement performante. Elle servira de référence pour le calcul des facteurs d'accélérations obtenus par les autres approches. Nous avons, en outre, défini un modèle permettant de prédire l'efficacité idéale d'une parallélisation pour différentes architectures et pour différents problèmes.

Dans ce contexte, nous pouvons nous permettre une analyse des facteurs d'accélération obtenus grâce aux différentes approches présentées par rapport à l'implémentation séquentielle optimisée à la main. Les figures 6.17 et 6.18 représentent les facteurs d'accélération obtenus avec les différentes approches sur les deux machines ${}_{2\times4}^{32}$Nehalem et ${}_{1\times4}^{32}$SandyBridge. Encore une fois, nous pouvons constater que les performances de MTPS et du démonstrateur Legolas++ ont des performances très proches de l'implémentation optimisée.

Outre les différentes approches présentées précédemment, ces figures présentent les facteurs d'accélération obtenus avec une parallélisation OpenMP de notre implémentation séquentielle. Cette implémentation correspond à ce qu'il est possible de faire sans modifier les structures de données. À la différence des autres approches, cette parallélisation n'exploite donc pas les unités vectorielles et apporte naturellement une accélération limitée aux nombre de cœurs disponibles. Sur ${}_{2\times4}^{32}$Nehalem, notre démonstrateur Legolas++ est entre 1,3 et 1,9 fois plus rapide que l'implémentation de référence uniquement parallélisée avec OpenMP. Sur ${}_{1\times4}^{32}$SandyBridge, l'accéléra-

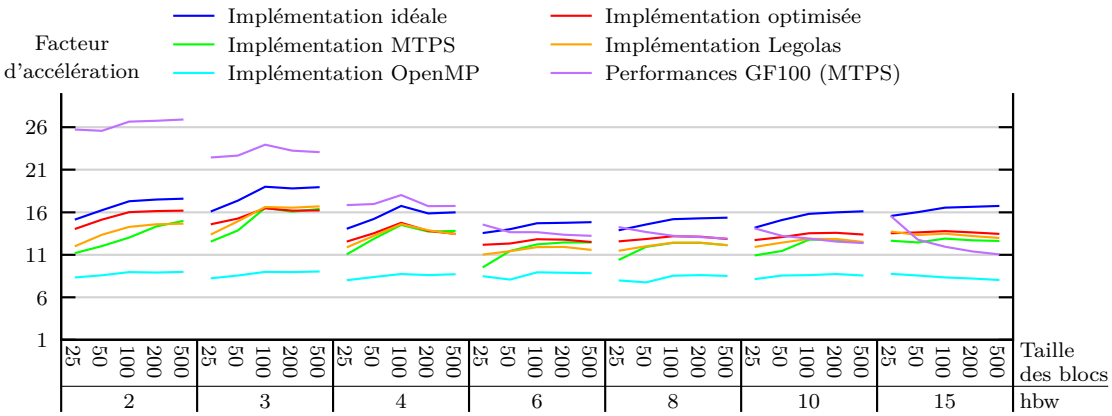


Figure 6.17 : *Accélérations apportées par les différentes approches sur* ${}^{32}_{2\times4}$*Nehalem.*

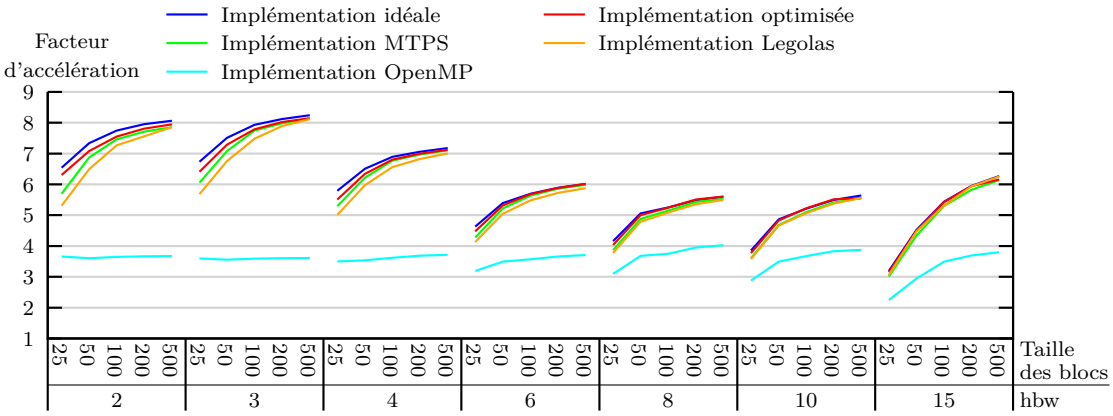


Figure 6.18 : *Accélérations apportées par les différentes approches sur* ${}^{32}_{1\times4}$*SandyBridge.*

tion apportée par ce démonstrateur par rapport à cette implémentation de référence parallélisée avec OpenMP est comprise entre 1,4 et 2,3. Ces accélérations sont apportées par l'utilisation des unités SIMD (SSE pour ${}_{2\times4}^{32}$Nehalem et AVX pour ${}_{1\times4}^{32}$SandyBridge). C'est la spécialisation du format de stockage lors de la compilation en fonction de l'architecture matérielle ciblée qui permet à Legolas++ d'exploiter ces unités SIMD.

Sur la *figure* 6.17, le facteur d'accélération apporté par l'utilisation de la machine Fermi est également représenté. Les deux machines ${}_{2\times4}^{32}$Nehalem et Fermi ayant été mises sur le marché à la même époque (mars 2009) et correspondant toutes deux à des gammes de produits équivalentes (station de travail scientifique), cette comparaison est la plus neutre possible. Nous pouvons donc comparer les gains apportés par **une implémentation unique basée sur MTPS** sur deux architectures différentes. Nous constatons sur cette figure que lorsque hbw est inférieur à 10, les performances sur Fermi sont meilleures que celles obtenues sur ${}_{2\times4}^{32}$Nehalem. L'écart maximal étant obtenu pour hbw égal à 2. Lorsque hbw est égal ou supérieur à 10, il devient plus intéressant d'utiliser ${}_{2\times4}^{32}$Nehalem que Fermi. Cela s'explique par le fait que sur ${}_{2\times4}^{32}$Nehalem, l'intensité arithmétique augmente avec hbw tandis qu'elle reste presque constante sur Fermi (cf. figures 6.4 à 6.6). En effet, le cache du GF100 est top petit pour permettre une réutilisation des données lorsque hbw augmente.

Finalement, les facteurs d'accélération obtenus sur ${}_{2\times4}^{32}$Nehalem et ${}_{1\times4}^{32}$SandyBridge pour ces cas tests varient entre 4 et 16. Ces performances sont donc éloignées du facteur 32 auquel nous aurions pu nous attendre d'après les caractéristiques des seuls processeurs (8 cœurs $\times$ 4 voies SIMD sur ${}_{2\times4}^{32}$Nehalem ou 4 cœurs $\times$ 8 voies SIMD sur ${}_{1\times4}^{32}$SandyBridge). Comme nous l'avons expliqué dans la section 6.1.2, cela s'explique par le fait que les performances sont limitées par les accès à la mémoire. Dans la prochaine section, nous étudierons les facteurs d'accélérations obtenus sur des problèmes présentant une intensité arithmétique plus grande et n'étant pas limités par la bande passante de la mémoire RAM. Nous pourrons alors juger la qualité de la parallélisation mise en œuvre par MTPS et le démonstrateur Legolas++.

6.2 Deuxième cas : les blocs ont une structure bande symétrique sur deux niveaux

Dans la section précédente, nous avons étudié les performances obtenues par notre démonstrateur de Legolas++ multicible sur des cas relativement simples. En effet, la structure de données associée à ces cas correspond à une structure bidimensionnelle rectangulaire, c'est-à-dire une structure bidimensionnelle dont les tailles sont constantes pour chaque dimension. Nous avions d'ailleurs utilisé ce cas pour introduire les différents formats de stockage optimisés dans la section 2.2.

Dans cette section, nous nous intéressons à un cas plus complexe : la structure de données associée à ce problème n'est plus bidimensionnelle rectangulaire, mais tridimensionnelle (une dimension par niveau de structure de la matrice) et non rectangulaire. Notre démonstrateur Legolas++ transforme automatiquement lors de la compilation cette structure de données tridimensionnelle en structure bidimensionnelle rectangulaire. Legolas++ peut ainsi utiliser les formats de stockage spécialisés pour les différentes architectures (cf. section 2.2). Grâce à cette adaptation du stockage, Legolas++ peut ensuite utiliser les connaissances issues du domaine mathématique pour paralléliser et vectoriser automatiquement l'algorithme de résolution choisi.

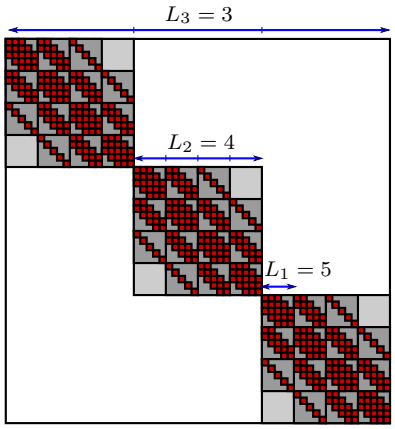


Figure 6.19 : *Structure de la matrice* **A** *utilisée dans le second cas test.*

6.2.1 Structure de la matrice A et paramètres du problème

Dans ce second problème, nous avons choisi d'utiliser une structure plus complexe. La structure que nous considérons ici possède trois niveaux :

- un niveau de structure diagonale, de taille $L_3 \times L_3$ et représenté en gris clair ;
- un premier niveau de structure bande symétrique, de taille $L_2 \times L_2$ et représenté en gris foncé ;
- un second niveau de structure bande symétrique, de taille $L_1 \times L_1$ et représenté en rouge.

La *figure* 6.19 donne une représentation graphique de cette structure. Dans le cadre de cette étude, nous ne ferons pas varier la structure de la matrice. Le premier niveau (gris clair) comportera donc toujours 2 sous-diagonales (hbw=3). Les blocs du dernier niveau comprendront 0,1 ou 2 sous-diagonales selon leur position dans la structure supérieure.

À cette structure à trois niveaux correspond un algorithme à trois niveaux pour résoudre le système $\mathbf{AX} = \mathbf{B}$.

- Pour la structure diagonale, l'algorithme est trivial et consiste simplement à résoudre les différents sous-problèmes. Le problème $\mathbf{AX} = \mathbf{B}$ peut ainsi être considéré comme une *collection* de sous problèmes *semblables*.
- Pour la première structure bande symétrique, nous avons choisi d'utiliser un algorithme de Gauss-Seidel (cf. **Alg.** 1.1). Cet algorithme nécessite d'effectuer des produits matrice-vecteurs pour les blocs extra-diagonaux et de résoudre des systèmes matriciels pour les blocs sur la diagonale. Notons n_i le nombre d'itérations effectuées. Nous ferons varier n_i afin de faire varier l'*intensité arithmétique* de l'algorithme de résolution.
- Pour la seconde structure bande symétrique, nous avons choisi de réutiliser la forme LDL^T de l'algorithme de Cholesky utilisé dans la section précédente.

Finalement, quatre paramètres caractérisent notre problème : L_1, L_2, L_3 et n_i.

Dans le cadre de notre étude, nous fixons les valeurs suivantes :

- $L_3 = 5000$
- $L_2 = L_1 = L$

En fonction de ces paramètres, nous pouvons déterminer l'*intensité arithmétique* i_a en comptant simplement le nombre d'accès mémoire et le nombre d'opérations :

$$i_a = n_i \frac{54L^2 - 80L + 16}{9L^2 - 4L}.$$

Nous nous plaçons dans le cas où n_i et L sont supérieurs à 10. Dans ce cas, i_a est toujours supérieur à 50. Nous nous plaçons ainsi dans un contexte où i_a est supérieur à i_c, quelque soit la configuration matérielle envisagée : sur ${}^{32}_{2\times4}$Nehalemet ${}^{32}_{1\times4}$SandyBridge, i_c vaut respectivement 18.9 et 41.3. D'après notre modèle introduit section 6.1.2, les performances de cet algorithme doivent alors être limitées par la puissance de calcul et non pas par la bande passante de la mémoire RAM.

6.2.2 Performances du démonstrateur

Les figures 6.20 et 6.21 représentent les performances obtenues pour différentes configurations du problème et du démonstrateur Legolas++. Sauf pour les cas pour lesquels L vaut 100,

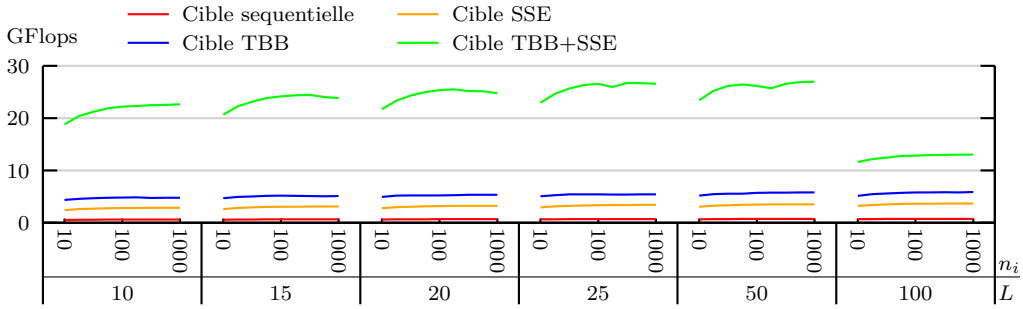


Figure 6.20 : *Performances du démonstrateur Legolas++ sur ${}^{32}_{2\times4}$Nehalem.*

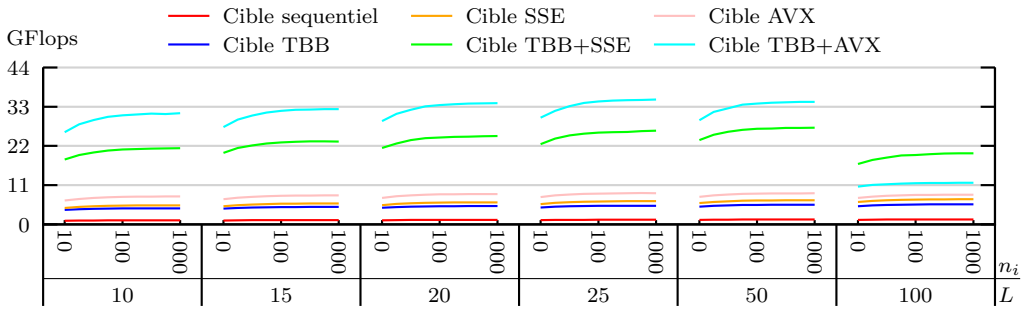


Figure 6.21 : *Performances du démonstrateur Legolas++ sur ${}^{32}_{1\times4}$SandyBridge.*

les performances sont relativement stables. Lorsque L vaut 100, les données nécessaires pour résoudre un problème correspondant au second niveau de la matrice prennent environ 450 ko. Afin de ne pas effecteur d'accès redondants à la mémoire RAM, il faut donc que le processeur dispose d'au moins 450 ko de cache par problème résolu simultanément. Sur ${}^{32}_{2\times4}$Nehalem, chacun des deux processeurs dispose de 4 Mo de cache de niveau 3. Chacun de ces processeurs peut donc résoudre efficacement jusqu'à neuf problèmes simultanément. Dans le cas où Legolas++ est configuré pour utiliser les unités SSE disponibles sur tous les cœurs, 16 problèmes sont résolus simultanément sur chaque processeur. La baisse de performances pour les cas où L vaut 100 est donc due à des défauts de cache. Sur ${}^{32}_{1\times4}$SandyBridge, le même raisonnement est valable en considérant que la taille du cache est cette fois-ci de 6 Mo.

À l'exception des cas où le démonstrateur Legolas++ est configuré pour tirer profit des unités AVX, les performances obtenues correspondent à plus de 20% des performances crêtes mesurées avec le même degré de parallélisme ; ce qui correspond à plus de 17% des performances théoriques des machines. Ceci est principalement dû à l'algorithme de résolution choisi et à son

implémentation séquentielle. Nous montrerons dans la suite de ce chapitre pourquoi ce résultat nous semble très satisfaisant. D'autant plus que ces résultats sont réguliers comme le montrent les figures 6.22 et 6.23. Elles représentent les facteurs d'accélération obtenus pour différentes configurations du problème et du démonstrateur Legolas++. Nous avons pris comme référence séquentielle les performances obtenues avec la cible séquentielle car cela correspond à ce qui aurait été obtenu sans utiliser Legolas++. Comme nous pouvions nous y attendre en observant les performances obtenues, tant que L reste inférieur à 100, les accélérations obtenues ne varient pas selon la configuration du problème. Afin d'en faciliter l'analyse, le *tableau* 6.1 synthétise ces

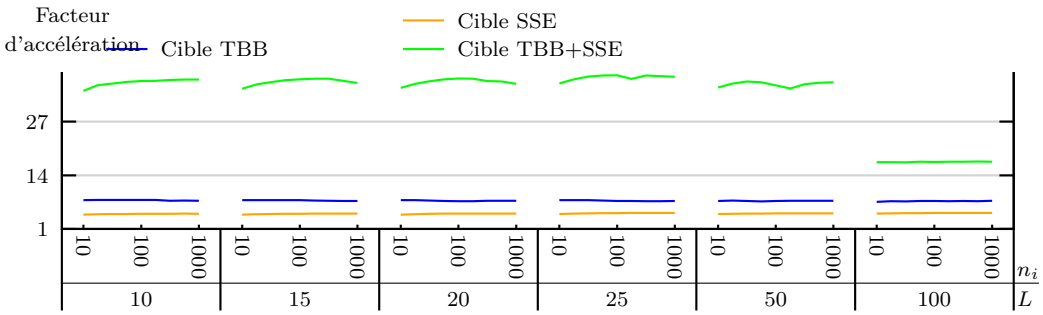


Figure 6.22 : *Accélérations apportées par Legolas++ sur* ${}^{32}_{2\times4}$*Nehalem.*

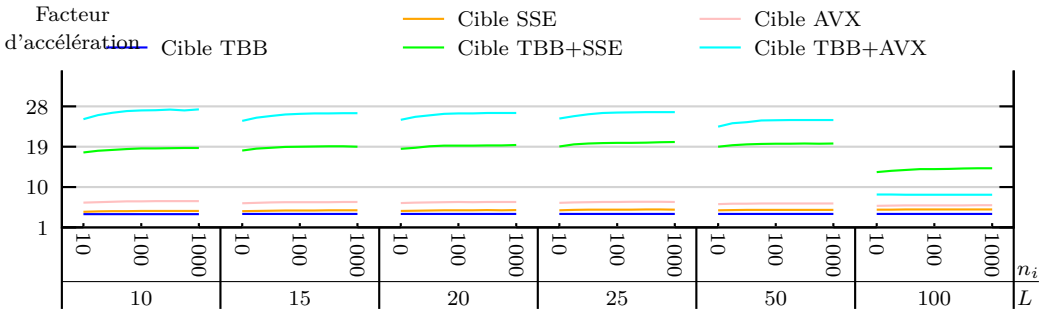


Figure 6.23 : *Accélérations apportées par Legolas++ sur* ${}^{32}_{1\times4}$*SandyBridge.*

résultats en prenant pour chaque donnée indiquée la valeur moyenne observée pour les différentes configurations du problème.

Afin d'analyser les facteurs d'accélération obtenus, il convient de les comparer au nombre de voies parallèles utilisées. Nous pouvons tout d'abord noter que l'accélération apportée par l'utilisation des différents cœurs disponibles est parfaitement linéaire avec le nombre de voies disponibles.

Lorsque les unités SSE sont utilisées, l'accélération est super-linéaire. Ceci s'explique par le fait que *pour cette famille de problèmes*, il est plus performant de choisir un format de stockage correspondant au format de stockage spécialisé pour les unités SSE (avec un entrelacement des éléments 4 par 4). Afin de mieux comprendre ce résultat, nous avons mesuré les performances obtenues sur nos deux machines avec des implémentations « dé-vectorisées ». c'est-à-dire que nous avons remplacé les fonctions intrinsèques du compilateur correspondant aux instructions vectorielles par des boucles appliquant l'opération souhaitée sur les différents éléments du vecteur SIMD. Notons que dans ce cas, notre compilateur (la version 4.6.1 de g++) ne « re-vectorise » pas la boucle ainsi générée. Nous avons pris comme référence séquentielle les performances obtenues avec le format de stockage le plus simple car il correspond à ce qui aurait été obtenu sans utiliser

Tableau 6.1 : *Synthèse des performances obtenues*

	Parallélisme exploité	Voies parallèles	Performances crêtes mesurées (GFlops)	Performances obtenues (GFlops)	Efficacité (%)	Accélération observée
$^{32}_{2\times4}$Nehalem	Aucun (séquentiel)	1	3.6	0.8	20.8	1
	SSE	4	14.3	3.7	25.7	4.9
	TBB	8	25.9	5.9	22.7	7.8
	TBB+SSE	32	103.5	27.0	26.1	36.0
$^{32}_{1\times4}$SandyBridge	Aucun (séquentiel)	1	5.9	1.4	23.8	1
	SSE	4	23.5	7.1	30.0	5.0
	AVX	8	47.0	8.8	18.6	6.3
	TBB	4	23.0	5.6	24.4	4.0
	TBB+SSE	16	92.1	27.1	29.5	19.4
	TBB+AVX	32	184.2	35.0	19.0	25.0

Legolas++. Les performances obtenues avec une implémentation séquentielle utilisant le format de stockage dédié aux unités SSE sont environ 1.4 fois plus élevées que celles obtenues avec le format de stockage le plus simple. Il serait cependant absurde de mettre au point un code de calcul s'appuyant sur un tel format de stockage sans aller jusqu'au bout de la démarche, c'est-à-dire jusqu'à l'utilisation des unités SSE. Deux éléments permettent d'expliquer ce phénomène. Premièrement, cela permet de diviser par 4 le nombre de tests et d'opérations de calculs d'indices : ces tests et calculs sont communs aux différents éléments d'un bloc SSE. Deuxièmement, les accès à la mémoire ne sont pas effectués de manière continue dans l'algorithme de résolution choisi. Le choix d'un format de stockage non entrelacé peut générer des accès conflictuels entre les bancs du cache L1 [47]. Au contraire, les formats de stockage entrelacés minimisent l'apparition de ces conflits. À partir d'un entrelacement 8 par 8, ces conflits ne peuvent plus apparaître. Cependant, comme nous l'avons évoqué dans la section 2.2.2.2, ceci est obtenu au détriment de la localité de données et donc de l'utilisation optimale du cache. Finalement, à format de stockage équivalent, l'utilisation des unités SSE apporte un facteur d'accélération d'environ 3.5.

Lorsque les unités AVX sont utilisées sur $^{32}_{1\times4}$SandyBridge, l'accélération obtenue est inférieure à 8. À format de stockage équivalent, l'utilisation des unités AVX n'apporte qu'un gain de 4.5. Nous pensons que ceci est dû aux limitations du bus de données entre les registres AVX et le cache L1. En effet, sur l'architecture Sandy Bridge [156], trois bus de transferts de 128 bits sont disponibles entre ces registres et le cache L1 : deux permettent de charger des données depuis le cache L1 vers les registres tandis que le dernier permet de stocker des données en registres vers le cache L1. De ce fait, un seul jeu de données de 256 bits peut transiter depuis le cache L1 vers les registres AVX à chaque cycle tandis que deux cycles sont nécessaires au stockage d'un élément. Le taux de réutilisation des données à l'échelle d'un registre est relativement faible dans notre algorithme, nous supposons donc que l'exécution est limitée par le débit de données entre les registres AVX et le cache L1. Dans le cas d'un produit de matrices dense (type `sgemm` [46]), les données déjà en registres sont réutilisées. Cela permet d'éviter ce goulet d'étranglement. Dans le cas de l'algorithme de résolution choisi, cela parait difficile à mettre en œuvre. Ce devrait de toute façon être fait dans la version séquentielle de l'algorithme : la paral-

47. Voir pages 2-22 et 3-57 du guide d'optimisation d'INTEL [253] pour plus de détails concernant les conflits d'accès au cache L1.

lélisation est automatiquement effectuée par Legolas++ sur la base de l'algorithme de résolution séquentiel choisi.

Finalement, les efficacités obtenues correspondent aux résultats obtenus sur de nombreuses machines avec la version FORTRAN de l'implémentation NETLIB des BLAS [254]. Le travail effectué par Kazushige Goto pour mettre au point la bibliothèque Goto BLAS [54, 55] consiste justement à prendre en compte les effets que nous venons de citer. Contrairement au modèle que nous avons introduit et qui repose sur l'existence d'un cache parfait, unique, uniforme et de taille infinie, Kazushige Goto s'appuie sur un modèle prenant en compte les registres et le fonctionnement réel des différents niveaux de caches. Il parvient ainsi à maximiser le nombre d'opérations pouvant être effectuées par seconde. Parmi les optimisations citées, notons la mise en place d'accès continus à la mémoire et la maximisation de la réutilisation des données. Ces transformations de l'algorithme original sont rendues possibles grâce à la simplicité de cet algorithme. Effectuer le même travail d'optimisation sur l'algorithme que nous étudions paraît extrêmement compliqué : cela suppose tout d'abord d'écrire cet algorithme « à plat », c'est-à-dire sans utiliser Legolas++. La complexité de la structure de la matrice choisie présage d'un algorithme tout aussi compliqué. Une fois cet algorithme écrit, il serait alors possible de réordonner les opérations de façon à pouvoir maximiser la réutilisation des données dans les registres. La difficulté serait alors de trouver les ordonnancements valables pour toutes les configurations de matrices.

6.3 Bilan

Notre démonstrateur permet, à partir d'un code relativement simple et concis, de définir des matrices structurées et les algorithmes de résolutions correspondant à ces matrices. Le code des solveurs ainsi écrit est portable sur différentes générations de processeurs. Les performances obtenues sur ces différentes générations de processeurs sont excellentes : bien que notre implémentation séquentielle ne soit pas mauvaise (environ 20% des performances théoriques), les accélérations obtenues sont quasi-linéaires en fonction du nombre de voies parallèles lorsque l'algorithme employé offre une *intensité arithmétique* suffisante. Nous devons maintenant étudier comment améliorer les performances de l'implémentation séquentielle avant d'envisager de la généraliser automatiquement.

Ne t'attarde pas à l'ornière des résultats
René Char (1907 – 1988)
Capitaine Alexandre pendant la Résistance
Poète

Chapitre 7

Conclusions et perspectives

Dans ce chapitre, nous rappelons les résultats obtenus au cours de cette thèse et proposons des pistes pour poursuivre ces recherches.

7.1 Bilan

Dans cette thèse, nous abordons la problématique de la maintenance des codes de calcul intensifs dans un contexte industriel. Nous avons vu dans l'introduction que la maintenance de ces codes est complexe et coûteuse. À ces difficultés s'ajoutent celles de la maintenance des performances sur les différentes générations de matériel sur lesquelles ces codes de calcul devront s'exécuter pendant leur durée de vie. En effet, les approches favorisant la maintenabilité d'un code sont souvent en opposition avec celles qui optimisent les performances. Lors de la mise au point d'un code de calcul intensif, un compromis doit donc être trouvé entre maintenabilité et performances.

Développée au sein du département SINETICS d'EDF R&D, la bibliothèque Legolas++ vise à établir un compromis intéressant. En effet, elle propose une approche de haut niveau dans un domaine restreint, ce qui permet à ses utilisateur une expressivité proche du langage mathématique pour décrire des solveurs d'algèbre linéaire mettant en œuvre des matrices creuses structurées. Ce haut niveau d'abstraction permet de faciliter la maintenance des code de calculs s'appuyant sur Legolas++ en permettant aux utilisateurs de se focaliser sur les aspects mathématiques du problème sans se soucier des aspects informatiques. Afin de compléter cet objectif, il est important de pouvoir proposer une implémentation optimisée de Legolas++ sur différentes architectures matérielles. Dans cette thèse, nous avons exploité les connaissances *a priori* issues du domaine d'application de Legolas++ afin d'atteindre cet objectif.

Pour cela, dans le chapitre 2, nous avons étudié les implémentations optimisées (parallèles et vectorisées) pour CPU et pour GPU d'un problème d'algèbre linéaire mettant en œuvre des matrices creuses structurées. Nous sommes parvenus à la conclusion que deux types de différences séparent les codes optimisés pour ces architectures matérielles :

- les modèles de programmation utilisés et donc l'expression du parallélisme ne sont pas identiques,
- les formats de stockage des données conduisant à l'obtention de performances optimales varient d'une architecture à l'autre.

Dans le but de masquer ces différences, nous avons étudié dans le chapitre 3 les différentes approches permettant la mise au point d'un code parallèle portable sur différentes architectures matérielles. Nous avons identifié plusieurs solutions pour masquer les différences entre les modèles

de programmation liées aux différentes architectures matérielles. Cependant aucun des outils que nous avons étudié ne permet aujourd'hui d'adapter automatiquement le format de stockage des données.

Afin de répondre à ce besoin, nous avons conçu MTPS (Multi-Target Parallel Skeleton), une bibliothèque C++ visant à adapter automatiquement le format de stockage des données sur différentes architectures matérielles. MTPS utilise des squelettes parallèles afin de masquer les différences entre les modèles de programmation et un système de *vues* sur les données pour permettre une manipulation de celles-ci indépendamment de leur format de stockage effectif. Les performances obtenues avec MTPS sont proches des performances optimales atteignables sur nos différentes machines de test. Cependant, pour parvenir à ce résultat, nous avons été contraints de restreindre l'ensemble des cas d'application de MTPS : cette bibliothèque ne permet de traiter que des problèmes conduisant à appliquer une même fonction à tous les éléments d'une collection homogène.

Dans le chapitre 5, après avoir introduit Legolas++ et son utilisation, nous avons présenté comment aboutir à une version multicible de Legolas++ en nous appuyant sur notre expérience MTPS. Pour cela, nous avons réimplémenté dans notre démonstrateur les concepts permettant de cibler les processeurs multicœurs disposant d'unités vectorielles différentes. Ce démonstrateur nous permet de montrer qu'il est possible, avec Legolas++, de générer automatiquement, à partir d'un même code source, des exécutables optimisés pour différentes architectures matérielles. Naturellement, les contraintes permettant à MTPS d'adapter le format de stockage restent valables avec ce démonstrateur de Legolas++ multicible et seules les opérations mettant en œuvre des matrices diagonales par bloc et dont les blocs possèdent la même structure peuvent être optimisées automatiquement pour différentes architectures. De très nombreux cas d'application de Legolas++ mettant en œuvre de telles matrice, cela ne nous est pas apparu comme une limitation.

Dans le chapitre 6, après avoir introduit un modèle simplifié des performances d'une application sur une architecture donnée, nous avons analysé les performances obtenues par différentes approches sur un premier exemple de problème. Nous avons alors montré que MTPS et Legolas++ introduisent un faible surcoût comparé à des implémentations optimisées : dans tous les cas, les performances obtenues sont très proches des performances prédites par notre modèle. Nous avons ensuite introduit un second exemple plus complexe que nous n'avons implémenté qu'avec notre démonstrateur Legolas++ : l'implémentation de cet exemple aurait été très complexe à mettre en œuvre avec une autre approche. Les implémentations obtenues conduisaient à des performances comprises entre 20% et 30% textbfdes performances théoriques sur nos différentes machines de test, ce qui est très satisfaisant à nos yeux compte tenu du type d'algorithme que nous avons choisis de mettre en œuvre.

7.2 Perspectives

Naturellement, la première suite qui peut être donnée à ces travaux consiste à intégrer les résultats précédents dans une version stable et industrialisable de Legolas++. Cela pose la question technique du choix du langage. Les différents outils conçus dans le cadre de cette thèse ont été implémentés en C++. Une autre solution aurait été de définir un langage *ad hoc* et de concevoir son compilateur. Cette solution aurait permis de simplifier la syntaxe, en particulier pour MTPS, mais cette stratégie implique des investissements importants dans l'environnement de développement. Cependant, la complexité atteinte dans les mécanismes de métaprogrammation peuvent laisser penser que nous sommes proches des limites de l'approche dans la mesure où il

est aujourd'hui relativement compliqué de déboguer un métaprogramme C++ en raison de l'absence d'outils dédiés. Aujourd'hui, seules les vérifications lexicales et syntaxiques sont effectuées par le compilateur et il est par exemple impossible de suivre pas à pas les différentes étapes de la génération du code optimisé, ni d'explorer l'ensemble des éléments constituants une classe donnée. Une approche basée sur un langage dédié disposant de son compilateur aurait en outre l'avantage de permettre la génération de code OpenCL plutôt que CUDA, ce qui permettrait alors de cibler tous les accélérateurs matériels disponibles aujourd'hui (et vraisemblablement demain).

Au cours de cette thèse, nous avons montré l'importance du format de stockage des données dans l'obtention de performances optimales. Nous avons en outre introduit un mécanisme de transformation permettant d'adapter automatiquement le format de stockage des structures de données bidimensionnelles. Il nous semble important de généraliser cette réflexion. En effet, dans notre approche, le fait de nous ramener à une structure de donnée bidimensionnelle conduit à laisser à l'utilisateur le soin d'exprimer la localité des données : les données d'une même ligne sont supposées être plus proches que les données d'une même colonne. Implicitement, l'utilisateur établit alors un lien entre l'algorithme qu'il choisit d'utiliser et la structure de données qu'il exprime. La question de savoir s'il est possible d'exprimer cette localité de données sans imposer une structure de données reste ouverte. Cela offrirait plus de libertés dans l'adaptation du format de stockage des données. Suite à cette question, il conviendrait alors d'étudier comment calculer la localité des données à partir de l'algorithme défini par l'utilisateur.

Les langages de programmation impératifs obligent l'utilisateur à définir le format de stockage de ses données. Cela lui permet de définir des structures adaptées à son problème et à son environnement matériel. Au contraire, les langages déclaratifs ne permettent généralement de définir ni le format de stockage des données, ni les relations de proximité entre ces données. Un plus juste milieu consisterait à fournir à l'environnement d'exécution ou de compilation les informations lui permettant de générer les structures de données optimales. La programmation orientée objet est un pas dans cette voie, mais ne permet d'exprimer qu'une localité des données sémantiques. En effet, les membres données d'une classes sont proches en mémoire car ils contribuent à un même but : définir l'état d'un objet. Cependant, pour les meilleures performances possibles, cette analyse devrait être faite en fonction de l'algorithme à appliquer.

En attendant de trouver une réponse générale, les langages dédiés (DSL ou DSEL) peuvent apporter une réponse partielle pour des domaines précis. En effet, leur interface peut être déclarative et, s'agissant d'un domaine applicatif restreint, il est possible de déterminer les règles de génération des structures de données optimisées à partir des connaissances *a priori* issues du domaine. La démarche qui nous a conduit à concevoir une version multicible de Legolas++ peut ainsi être appliquée pour d'autres domaines que l'algèbre linéaire.

Annexe A

Du polymorphisme dynamique au polymorphisme statique

La Programmation Orientée Objet (POO) est une approche permettant d'augmenter l'expressivité d'une bibliothèque. En effet, la possibilité de définir des objets possédant des fonctions membres et de passer ces objets en argument d'autres fonctions permet *in fine* d'écrire des fonctions qui prennent comme argument des fonctions. Un objet est une structure de données valuées et cachées qui répond à un ensemble de messages correspondant à des fonctions « membres ». Cette structure de données définit son état tandis que l'ensemble des messages qu'il accepte décrit son comportement.

Des objets de différentes classes peuvent définir la même interface, c'est-à-dire accepter les mêmes messages. L'implémentation des fonctions permettant le traitement de ces messages sera cependant généralement différente d'une classe d'objet à l'autre. La POO permet l'encapsulation des fonctions et des données : en fonction de leur état interne, deux objets peuvent se comporter différemment face aux mêmes sollicitations. Du point de vue de l'utilisateur d'une Bibliothèque Orientée Objet (BOO), un même extrait de code pourra donc générer des exécutions différentes selon le type et l'état interne des différents objets manipulés.

Nous allons dans cette annexe étudier la conception d'une BOO afin de comprendre comment cette approche permet d'améliorer l'expressivité d'une bibliothèque. Nous allons nous intéresser au cas présenté dans la section 1.2.2. Nous présenterons dans un premier temps la conception et l'implémentation de cette bibliothèque en C++ en utilisant les techniques de programmation habituelles de ce langage. Dans un second temps, nous présenterons une implémentation alternative permettant l'obtention de meilleures performances mais dont l'utilisation est plus contrainte.

A.1 Programmation orientée objet

Une expression vectorielle est soit un vecteur, soit une combinaison linéaire d'expressions vectorielles. Les différentes expressions vectorielles possibles ne possèdent donc pas la même structure de données interne. Par exemple, certaines expressions comportent un seul vecteur tandis que d'autres en comportent trois. Cependant, elles possèdent toutes (le simple vecteur inclus) un comportement commun : elles peuvent communiquer une valeur scalaire pour chaque indice compris entre zéro et leur taille. Il est possible de formaliser cette propriété dans une classe abstraite virtuelle `VectorExpression` dont hériteront toutes les expressions vectorielles :

```
1 class VectorExpression{
```

```
2   virtual int size() const =0;
3   virtual float get(const int i) const =0;
4 };
```

En C++, une classe abstraite contient au moins une fonction purement virtuelle. Afin de définir une fonction comme purement virtuelle, il suffit de la déclarer virtuelle (avec le mot-clé `virtual`) et de la définir comme étant égale à zéro (`=0`). La ligne 2 déclare que les classes héritant de `VectorExpression` doivent définir une fonction virtuelle appelée `size` qui ne prend aucun argument et qui renvoie un entier. La ligne 3 déclare la fonction `get` qui prend en argument un entier `i` et renvoie un nombre flottant.

D'autres informations sémantiques non explicitement formulées s'ajoutent à ces définitions :

– l'entier retourné par la fonction `size` représente la taille des vecteurs de l'expression,

– le nombre flottant retourné par la fonction `get` est égal au $\texttt{i}^{\text{ème}}$ élément de l'expression.

Avec la définition de cette classe, nous disposons de suffisamment d'informations pour pouvoir mettre au point une fonction qui calcule la norme des expressions vectorielles :

```
float norm(const VectorExpression & ve){
  float out = 0;
  for (int i=0; i<ve.size() ; ++i) {
    tmp = ve.get(i);
    out+=tmp*tmp;
  }
  return sqrtf(out);
}
```

Naturellement, nous ne pouvons pas utiliser directement cette fonction. En effet, la classe `VectorExpression` est abstraite et ne peut donc pas être instanciée. Pour pouvoir utiliser cette fonction, il nous faut donc définir des classes héritant de `VectorExpression`. La première classe que nous allons implémenter est la classe `Vector` qui représente un vecteur :

```
1  class Vector : public VectorExpression{
2    public:
3      Vector(const int size): size_(size), data_(new float[size_]){}
4      ~Vector(){delete[] data_;}
5  ...
6      virtual int size() const { return size_; };
7      virtual float get(int i) const { return data_[i]; };
8      float & set(int i) { return data_[i]; };
9  ...
10   private:
11     const int size_;
12     float * data_;
13 };
```

La ligne 1 déclare que la classe `Vector` hérite de la classe `VectorExpression`. Afin de pouvoir être instanciée, cette classe ne doit plus contenir de fonction purement virtuelle. Les fonctions `size` et `get` sont donc respectivement surchargées aux lignes 6 et 7.

Contrairement au paradigme fonctionnel, la programmation orientée objet autorise la modification des données. La fonction `set`, définie à la ligne 8, permet donc de modifier les données du vecteur.

Comme la classe `Vector` hérite de la classe `VectorExpression`, nous pouvons maintenant calculer la norme d'un vecteur. Afin de prendre en charge les expressions vectorielles plus complexes, nous devons définir les autres classes correspondant aux expressions vectorielles. Une expression vectorielle est une combinaison linéaire d'expressions vectorielles. Deux types d'opérations doivent donc être exprimables : la somme de deux expressions vectorielles d'une part et le

produit d'une expression vectorielle avec un scalaire d'autre part. Nous allons donc définir deux classes `AddVectorExpr` et `ScalVectorExpr` correspondant respectivement à ces deux opérations.

Naturellement, `AddVectorExpr` hérite de la classe abstraite `VectorExpression` :

```
1  class AddVectorExpr : public VectorExpression{
2    public:
3      AddVectorExpr(const VectorExpression & l, const VectorExpression & r)
4        : left_(l), right_(r){ assert(left_.size()==right_.size()); }
5      virtual int size() const { return left_.size(); };
6      virtual float get(const int i) const {
7        return left_.get(i)+right_.get(i);
8      };
9    private:
10     const VectorExpression & left_;
11     const VectorExpression & right_;
12 };
```

Afin de pouvoir calculer les éléments à retourner avec la fonction `get`, la classe `AddVectorExpr` doit contenir des références vers les deux expressions à ajouter : `left_` et `right_`, respectivement définies aux lignes 10 et 11. Le calcul de la somme des éléments de `left_` et `right_` est déporté dans la fonction `get` définie à la ligne 6. Nous pouvons ainsi éviter l'allocation d'un vecteur temporaire comme le vecteur Z que nous avions utilisé avec les BLAS (cf. section 1.2.2.1).

La classe `ScalVectorExpr` ressemble beaucoup à la classe `AddVectorExpr` :

```
1  class ScalVectorExpr : public VectorExpression{
2    public:
3      ScalVectorExpr(const float & a, const VectorExpression & x)
4        : a_(a), x_(x){}
5      virtual int size() const { return x_.size(); };
6      virtual float get(const int i) const { return a_*x_.get(i); };
7    private:
8      const float a_;
9      const VectorExpression & x_;
10 };
```

La principale différence réside dans la fonction `get` (cf. ligne 6) qui multiplie l'élément de l'expression et le scalaire. Notons que `x` (cf. ligne 9) peut représenter n'importe quelle classe héritant de `VectorExpression`. Les classes `AddVectorExpr` et `ScalVectorExpr` permettent donc de représenter n'importe quelle expression vectorielle, aussi longue soit-elle.

L'utilisateur peut maintenant écrire un code comme celui-ci :

```
1  Vector w(n),x(n),y(n);
2  float  a,b,c;
3  ...
4  float result = norm(
5    AddVectorExpr(
6      AddVectorExpr(
7        ScalVectorExpr(a,w),
8        ScalVectorExpr(b,x)
9      ),
10     ScalVectorExpr(c,y)
11   )
12 )
```

Il est aisé de regrouper les classes `Vector`, `ScalVectorExpr`, `AddVectorExpr` ainsi que la fonction `norm` dans une bibliothèque. Afin de faciliter l'utilisation de cette bibliothèque, le C++ permet de surcharger les opérateurs et ainsi de définir des langages spécialisés à un domaine.

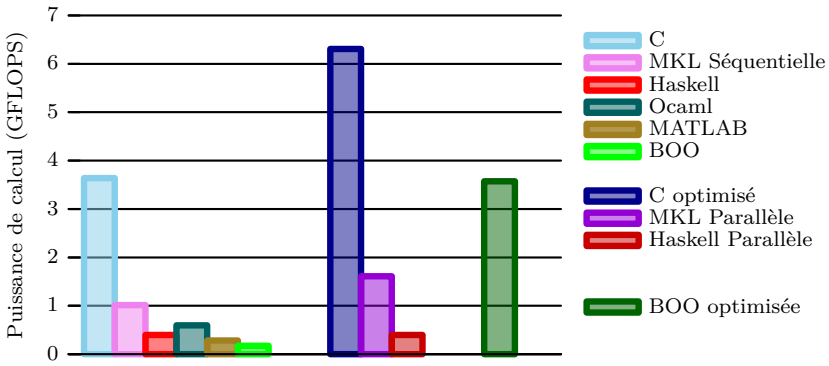


Figure A.1 : *Performances obtenues pour le calcul de la norme* $||aW + bX + cY||$.

```
1  AddVectorExpr operator+ ( const VectorExpression & left
2                          , const VectorExpression & right
3                          ){
4    return AddVectorExpr(left, right);
5  }
6
7  ScalVectorExpr operator*  ( const float & a
8                             , const VectorExpression & x
9                             ){
10   return ScalVectorExpr(a, x);
11 }
```

Cela permet *in fine* à l'utilisateur d'une telle bibliothèque d'implémenter la calcul de la norme de la manière suivante :

```
1 Vector w(n),x(n),y(n);
2 float  a,b,c;
3 ...
4 float result = norm(a*w+b*x+c*y);
```

L'utilisateur de cette bibliothèque est alors en droit d'espérer que leurs implémentations soient optimisées. Ce droit est d'autant plus légitime qu'il est relativement aisé de fournir une implémentation parallèle et vectorisée de cette bibliothèque. Le graphique « BOO » de la *figure* A.1 reprend le code présenté ci-dessus tandis que le graphique « BOO optimisé » est obtenu à partir d'une implémentation parallèle et vectorisée par nos soins. Les autres approches mentionnées correspondent aux approches présentées dans la section 1.2.2

Les performances fournies par cette approche (0,17 GFlops pour la version ci-dessus et 3,6 GFlops pour la version optimisée) sont meilleures que celles fournies par la MKL. Elles restent cependant très inférieures à la version optimisée en C. Ceci est dû au coût des fonctions virtuelles. En effet, les fonctions virtuelles sont implémentées avec des pointeurs de fonction et impliquent donc un surcoût d'indirection. Sur la machine $_{2\times4}^{32}$Nehalem, ce surcoût est estimé à 30 cycles. Ce chiffre est à comparer au nombre de cycles requis pour exécuter le corps de nos fonctions virtuelles (1 ou 2 selon les cas). De plus, l'utilisation de fonctions virtuelles empêche le compilateur d'effectuer des optimisations interprocédurales. Dans cet exemple, cela se traduit par le fait que le compilateur est incapable de vectoriser cette implémentation.

Toutes les informations étant disponibles à la compilation, il est regrettable d'utiliser un mécanisme impliquant un tel surcoût lors de l'exécution. Un outil qui serait capable d'analyser le code et d'effectuer la composition des opérations lors du processus de compilation permettrait d'obtenir l'expressivité souhaitée et n'impliquerait pas de surcoût sur les performances. En C,

le mécanisme de macros permet d'effectuer un certain nombre de transformations du code lors de la compilation. Cependant, il est trop limité pour répondre à notre besoin. Plus récent, le C++ a généralisé ce système avec les patrons (*templates*) de classe ou de fonction. Beaucoup plus expressifs et fournissant des mécanismes beaucoup plus élaborés, les patrons permettent l'introduction de la programmation générative.

A.2 La programmation générative

La programmation générique permet de décrire des algorithmes abstraits pouvant s'appliquer à différents conteneurs de données. Ces conteneurs correspondent aux différentes structures de données. La bibliothèque de patrons standard du C++ [73, 74] (*C++ Standard Template Library* ou STL) propose ainsi un large ensemble de conteneurs génériques ainsi que de nombreux algorithmes opérant sur ces conteneurs. Ainsi l'algorithme `std::accumulate` est une implémentation générique des différentes normes. Elle permet par exemple de passer de la norme L_2 aux normes $L_{-\infty}$, L_1 ou L_∞ en modifiant simplement la fonction qui permet d'effectuer l'accumulation des valeurs (pour la fonction `norm` présentée précédemment, cette opération est effectuée ligne 5, page 144).

Les bibliothèques génériques qui prennent un rôle actif dans la compilation sont dites « actives » par opposition aux simples collections passives de routines de calcul [75, 76, 77]. Dans sa thèse [77], Todd Veldhuizen fournit la définition suivante :

> *Une bibliothèque active est une bibliothèque qui fournit à la fois des abstractions propres à un domaine et les connaissances requises pour les optimiser et vérifier leurs exigences sémantiques. En outre, les bibliothèques actives sont composables : un même fichier source peut combiner l'utilisation de plusieurs d'entre elles.*

Au contraire des bibliothèques traditionnelles ou génériques, les bibliothèques actives, comme Blitz++ [78], ne sont donc pas un simple agrégat de fonctions, de classes ou d'objets. Elles fournissent un haut niveau d'abstraction et sont capables d'optimiser ces abstractions en fonction du contexte. Elle prennent donc un rôle « actif » lors de la compilation et influencent la génération de code par le compilateur. La possibilité d'opérer des transformations dans le code permet à ces bibliothèques de se rapprocher des compilateurs. Dans l'article *Active Libraries : Rethinking the roles of compilers and libraries* [75], Todd Veldhuizen présente en détails les fondements des bibliothèques actives et explique comment l'utilisation de connaissances issues d'un haut niveau d'abstraction dans un domaine particulier permet d'effectuer des optimisations que le compilateur ne peut effectuer seul.

Appliquées à notre exemple, ces techniques permettent de supprimer les appels aux fonctions virtuelles. Nous allons pour cela utiliser un patron de conception permettant de résoudre le polymorphisme à la compilation. Cette technique, appelée *Curiously Recurring Template Pattern* (CRTP) [79, 80] permet de résoudre lors de la compilation les liens d'héritage et donc de supprimer les différents pointeurs de fonction par des appels directs aux fonctions de la classe réelle de l'objet, voire de remplacer ces appels par le corps de ces fonctions. Nous dirons alors de ces fonctions qu'elles sont *inlinées*.

Le principe du CRTP consiste à paramétrer la classe abstraite par ses classes dérivées :

```
1 template <class DERIVED> class Base{
2 ...
3   private:
4     inline const DERIVED & convert() const {
5       return *static_cast<const DERIVED* >(this);
```

```
6       }
7   };
8   ...
9   class Derived : public Base<Derived>{
10  ...
11  };
```

La ligne 9 signifie que la classe `Derived` hérite de la classe `Base<Derived>`. Le CRTP fonctionne sur l'hypothèse qu'il n'y a que la classe `Derived` qui hérite de `Base<Derived>`. Grâce à cette hypothèse, toute instance de la classe `Base<Derived>` peut être convertie en une instance de la classe `Derived`. Dans l'exemple ci-dessus, la fonction `convert` (ligne 4) convertit l'instance courante de `Base<DERIVED>` en une instance de la classe `DERIVED`. Cette conversion permet ensuite aux autres méthodes de la classe `Base<DERIVED>` de faire directement appel aux fonctions de la classe `DERIVED`. Le mot-clé `inline` précédent la déclaration de la fonction `convert` encourage le compilateur à inliner cette fonction. L'appel à cette fonction a donc généralement un coût nul.

Nous allons maintenant appliquer ceci aux classes d'expression vectorielles :

```
1  template <class VECTOR_EXPRESSION> class VectorTemplateExpression{
2    public:
3      inline int size() const { return convert().size(); }
4      inline float get(const int i) const { return convert().get(i); }
5    private:
6      inline const VECTOR_EXPRESSION & convert() const {
7        return *static_cast<const VECTOR_EXPRESSION* >(this);
8      }
9  };
```

Nous reconnaissons ici la classe `VectorExpression` qui a été adaptée pour permettre l'utilisation du CRTP. Comme dans l'exemple introductif de la classe `Base`, une instance de la classe `VectorTemplateExpression<VECTOR_EXPRESSION>` peut être convertie en une instance de la classe `VECTOR_EXPRESSION`. Cette conversion permet d'appeler directement les fonctions `size` et `get` (lignes 3 et 4) qui ne sont plus virtuelles mais font directement appel aux implémentations de la classe `VECTOR_EXPRESSION`. Ces fonctions peuvent maintenant être inlinées par le compilateur. Appeler une de ces fonctions est donc strictement équivalent à appeler directement l'implémentation de la classe `VECTOR_EXPRESSION`.

Nous pouvons maintenant écrire une version générique de la fonction `norm` afin de prendre en compte ces modifications :

```
1  template <class VE>
2  float norm(const VectorTemplateExpression<VE> & ve){
3    float out = 0;
4    for(int i=0; i<ve.size() ; ++i){
5      float tmp = ve.get(i);
6      out+=tmp*tmp;
7    }
8    return sqrtf(out);
9  }
```

Cette fonction est paramétrée par le type réel `VE` de l'expression vectorielle `ve`. Lors de l'appel à cette fonction, le compilateur instanciera automatiquement `norm` en fonction du type de `ve`. Le typage spécifique `VectorTemplateExpression<VE>` de `ve` permet d'empêcher le compilateur d'instancier cette fonction avec un argument incompatible. Le reste de la fonction n'a pas besoin d'être modifiée.

Pour les classes, les modifications à apporter sont légèrement plus importantes. Nous allons prendre l'exemple de la classe `ScalVectorTemplateExpr` :

```
1  template <class VE> class ScalVectorTemplateExpr
2  : public VectorTemplateExpression<ScalVectorTemplateExpr<VE> >{
3    public:
4      inline ScalVectorTemplateExpr(  const float & a
5                                    , const VectorTemplateExpression<VE>& x)
6      : a_(a), x_(x) {}
7      inline int size() const { return x_.size(); };
8      inline float get(const int i) const { return a_*x_.get(i); };
9    private:
10     const float a_;
11     const VectorTemplateExpression<VE> & x_;
12 };
```

La classe `ScalVectorTemplateExpr` prend en charge le produit d'une expression vectorielle par un scalaire. Elle doit donc contenir une référence vers cette expression vectorielle d'une part et ledit scalaire d'autre part. Dans la version précédente, il suffisait de contenir une référence vers une instance de la classe abstraite `VectorExpression`. Cependant, nos différentes expressions vectorielles n'héritent plus de la même classe. Il faut donc paramétrer `ScalVectorTemplateExpr` par le type réel de l'expression vectorielle, que nous nommons `VE` à la ligne 1. Naturellement, la classe `ScalVectorTemplateExpr<VE>` représente une expression vectorielle et hérite donc de la classe `VectorTemplateExpression<ScalVectorTemplateExpr<VE> >`.

Lorsque le patron de fonction `norm` reçoit comme argument une instance `ve` de la classe `ScalVectorTemplateExpr`, le compilateur spécialise ce patron. L'objet `ve` est alors perçu comme une instance de la classe `VectorTemplateExpression<ScalVectorTemplateExpr<VE> >`. L'appel à la fonction `VectorTemplateExpression<ScalVectorTemplateExpr<VE> >::get(i)` est alors remplacé par le compilateur par un appel à la fonction `ScalVectorTemplateExpr::get(i)`. Cet appel de fonction sera à son tour remplacé par le code équivalent. Tous les appels de fonction intermédiaires étant inlinés, le code résultant est donc équivalent au code C.

Comme pour la bibliothèque présentée dans la section précédente, la surcharge des opérateurs permet de simplifier l'utilisation de la bibliothèque. Des patrons de fonctions sont donc mis en œuvre afin de surcharger les opérateurs :

```
1  template <class L, class R>
2  AddVectorTemplateExpr<L, R> operator+
3    ( const VectorTemplateExpression<L> & left
4    , const VectorTemplateExpression<R> & right
5    )
6  {
7    return AddVectorTemplateExpr<L, R>(left, right);
8  }
9
10 template <class VECTOR_EXPRESSION>
11 ScalVectorTemplateExpr<VECTOR_EXPRESSION> operator*
12   ( const float & a
13   , const VectorTemplateExpression<VECTOR_EXPRESSION> & x
14   )
15 {
16   return ScalVectorTemplateExpr<VECTOR_EXPRESSION>(a, x);
17 }
```

Le compilateur prendra en charge la détermination de ces types et instanciera la spécialisation de ces deux fonctions. L'interface utilisateur est donc la suivante :

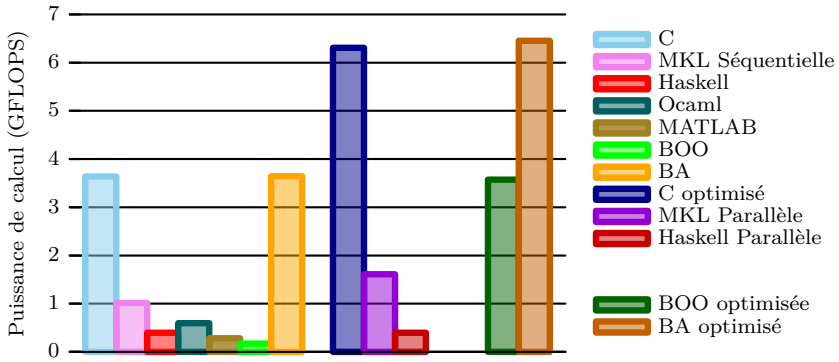


Figure A.2 : *Performances obtenues pour le calcul de la norme $||aW + bX + cY||$ sur notre machine de tests ${}_{2\times4}^{32}$Nehalem pour des vecteurs de plus de 3×10^6 éléments.*

```
1 VectorTemplate w(n),x(n),y(n);
2 float  a,b,c;
3 ...
4 float result = norm(a*w+b*x+c*y);
```

Lorsque le compilateur arrive à la ligne 4, il doit analyser l'expression `a*w+b*x+c*y`. L'ordre de précédence des opérateurs le conduit appeler trois fois la fonction `operator*` spécialisée avec le type `VectorTemplate` pour construire `aW`, `bX` et `cY`. Il va ensuite appeler deux fois la fonction `operator+` pour construire successivement `aW_bX` et `aW_bX_cY`. Ce mécanisme d'interprétation des opérateurs pour générer un AST de haut niveau lors de la compilation est communément appelé *expression templates* [255, 256]. L'objet `aW_bX_cY` étant construit, la fonction `norm` pourra être appelée sur cet objet. Le compilateur connaissant toutes les fonction appelées, ces dernières peuvent être *inlinées*. Cette ligne de code est donc équivalente à la boucle C présentée en début de chapitre. L'article [60] montre comment paralléliser les *expression templates* en s'appuyant sur la bibliothèque INTEL TBB [139].

La mesure « BA » de la *figure* A.2 montre les performances obtenues par cette implémentation. Notons tout d'abord que le compilateur a été capable de vectoriser cette implémentation. L'implémentation présentée ci-dessus obtient les mêmes performances que l'implémentation C en séquentiel (3,6 GFlops) comme en parallèle (6.5 GFlops). Nous avons mis au point deux versions optimisées : une version vectorisée par le compilateur tandis que nous avons vectorisé la seconde nous-même. Dans les deux cas, les performances obtenues sont identiques.

A.3 Avantages de la programmation générative

Dans ce chapitre, nous avons présenté deux approches pour implémenter une bibliothèque orientée objet. Dans un premier temps, nous avons montré que la programmation orientée objet permet la définition de langages spécialisées à un domaine. L'utilisation d'un tel langage permet de simplifier l'écriture du code utilisateur de la bibliothèque. Cependant, notre première implémentation s'appuyant sur des fonctions virtuelles ne permet pas l'obtention de performances comparables avec une implémentation optimisée. Dans un second temps, nous avons montré comment la programmation générative permet au compilateur d'interpréter un langage spécialisé lors de la compilation. Les fonctions virtuelles ne sont alors plus nécessaires et il est possible d'obtenir des implémentations aussi efficaces que des implémentations optimisées en C.

Annexe B

Introduction aux formats de stockage creux *Compressed Row Storage*

Le format de stockage de matrices creuses CRS (*Compressed Row Storage*) vise à limiter l'empreinte mémoire des matrices creuses. Il permet aussi de limiter le nombre d'opérations à effectuer pour les calculs mettant en œuvre des matrices (ex : produit matrice-vecteur). Le format CRS nécessite le stockage de plusieurs tableaux. Un premier tableau, `val`, contient les éléments non-nuls stockés ligne par ligne. Un second tableau, `col_ind`, contient les indices de colonnes des éléments qui sont dans le tableau `val`. Le dernier tableau, `row_ptr`, contient les indices de début de ligne dans le tableau `col_ind`. Nous allons illustrer ceci par un exemple. Soit A une matrice de taille 5×4 :

$$A = \begin{bmatrix} a & 0 & 0 & b & 0 \\ 0 & 0 & c & 0 & d \\ 0 & e & 0 & 0 & 0 \\ 0 & 0 & f & 0 & 0 \end{bmatrix}.$$

Cette matrice contient vingt éléments dont quatorze sont nuls. Afin de limiter à la fois l'empreinte mémoire et le nombre d'opérations à effectuer, il est pertinent de définir un format de données adapté à de telles matrices. Dans le format de stockage CRS, le tableau `val` contient l'ensemble des éléments non nuls tandis que le tableau `col_ind` contient l'indice de colonne de ces éléments :

$$\begin{aligned} \texttt{val} &= [\ a\ \ b\ \ c\ \ d\ \ e\ \ f\ ] \\ \texttt{col\_ind} &= [\ 1\ \ 4\ \ 3\ \ 5\ \ 2\ \ 3\ ]. \end{aligned}$$

Par exemple, l'élément c est dans la troisième colonne, comme l'élément f. Le tableau `row_ptr` contient les indices de début de ligne dans le tableau `col_ind` :

$$\texttt{row\_ptr} = [\ 1\ \ 3\ \ 5\ \ 6\ ].$$

Par exemple, les troisième et quatrième éléments de `row_ptr`, valent 5 et 6. La différence est de 1, ce qui signifie que la troisième ligne de la matrice A contient 1 élément. Cet élément (e) et son numéro de colonne (2) sont stockés en cinquième position des tableaux `val` et `col_ind`.

Annexe C

Définitions Legolas++

Dans ce chapitre, nous présentons quelques exemples de classes à implémenter pour utiliser Legolas++. Nous présentons tout d'abord comment implémenter la définition d'une matrice avant de présenter comment ajouter un algorithme de résolution à Legolas++. Ce chapitre s'appuie sur les éléments introduits section 5.1.

C.1 Définition d'une matrice Legolas++

Dans cette section, nous présentons les différents éléments composant la définition d'une matrice Legolas++. Nous introduisons successivement la liste des éléments qui composent une définition puis deux exemples de définitions. Le premier exemple portera sur la définition d'une matrice à un niveau tandis que le second mettra en œuvre une matrice à deux niveaux.

C.1.1 Concept C++ de définition de matrice Legolas++

La définition d'une matrice Legolas++ est un objet dont la classe doit respecter les contraintes imposées par le concept de `MATRIX_DEFINITION`[48]. Les différents éléments requis par ce concept figurent en orange dans l'extrait de pseudo-code C++ suivant :

```
1  concept MatrixDefinitionConcept
2    // REAL_TYPE definit le type de donnees reelles (float ou double)
3    // LEVEL indique le niveau de la structure
4    : public Legolas::DefaultMatrixDefinition<REAL_TYPE, LEVEL>{
5      public :
6        // MatrixStructure definit la structure de la matrice
7        typedef ... MatrixStructure;
8        // ConstGetElement definit le type de retour des accesseurs
9        typedef ... GetElement;
10
11       // Data encapsule les donnees de la matrice
12       //    Data doit heriter de Legolas::MatrixShape<1>
13       struct Data : Legolas::MatrixShape<LEVEL>
14       {
15         // un constructeur defini par l'utilisateur
16         Data(const Legolas::MatrixShape<LEVEL> & ms, ...);
17         // un constructeur par recopie
18         Data(const Data & data);
19         ...
```

48. L'équivalent en JAVA ou en C# d'un concept C++ est une interface.

```
20        };
21
22        // un accesseur optimise selon la structure de matrice choisie ligne 7
23        static inline GetElement XXXXX (int i , int j, const Data & data);
24        // differentes fonctions permettant de recuperer les parametres
25        //  variables de la structure de matrice choisie ligne 7
26        static inline int XXXXX(const Data & data);
27        static inline int XXXXX(const Data & data);
28 };
```

Ce concept permet d'accéder à l'ensemble des données définissant une matrice Legolas++.

- Le type des données réelles contenues dans la matrice est défini par `REAL_TYPE` (ligne 4).
- Le nombre de niveau de la matrice est défini par `LEVEL` (ligne 4).
- La description de la structure élémentaire de la matrice est accessible d'une part par le type de structure `MatrixStructure` (ligne 7) et d'autre part par les accesseurs définis à partir de la ligne 26. Les différentes structure élémentaires définissent le schéma de remplissage de la matrice. Elles définissent une hiérarchie. En effet, la structure `Legolas::Diagonal` est un cas particulier de la structure `Legolas::Banded` La liste des accesseurs à implémenter dépend de la structure `MatrixStructure` choisie.
- La taille de la matrice et de ses blocs sont accessibles *via* des accesseurs hérités de la classe `Legolas::DefaultMatrixDefinition<REAL_TYPE, LEVEL>` (ligne 4).
- Les données de la matrice sont accessibles *via* l'accesseur optimisé ligne 23. La signature de l'accesseur dépend de la structure de matrice choisie.

C.1.2 Matrice Legolas++ à un niveau

La classe `MDefinition` fournit la définition Legolas++ d'une matrice bande symétrique dont les éléments sont de type `float`. Cette classe utilise la structure `Legolas::BandedSymmetric` fournie la Legolas++ :

```
1  class MDefinition : public Legolas::DefaultMatrixDefinition<float,1>{
2    public:
3      typedef float                       RealType;
4      typedef Legolas::BandedSymmetric    MatrixStructure;
5      typedef RealType                    GetElement;
6      // ShapeType est fourni par DefaultMatrixDefinition
7      typedef MDefinition::ShapeType      ShapeType;
8
9      struct Data : ShapeType{
10       Data(const ShapeType & ms, const int hbw)
11         : ShapeType(ms), hbw_(hbw){}
12       Data(const Data & data)
13         : ShapeType(data), hbw_(data.hbw_){}
14       const int hbw_;
15     };
16
17     // Les fonctions requises par la structure Legolas::BandedSymmetric
18     static inline GetElement lowerBandedGetElement( int i, int j
19                                                     , const Data & data){
20       // i et j designent ici un element non nul
21       return XXXXXXX; // Data herite nrows() de MatrixShape<1>
22     }
23     static inline int linf(const Data & data){
24       return data.hbw_;
25     }
26 };
```

La classe imbriquée `MDefinition::Data` (lignes 9 à 15) contient la largeur de bande de la matrice (`hbw_`). Cette donnée peut être utilisée par les fonctions `lowerBandedGetElement` et `linf` dont la signature est imposée par le choix de la structure `Legolas::BandedSymmetric`, ligne 4. Aucune autre contrainte ne pèse sur ces fonctions : elles pourraient par exemple lire des valeurs dans un fichier. Notons ici que l'utilisation de `lowerBandedGetElement` implique d'avoir préalablement vérifié que l'élément désigné par les coordonnées `i` et `j` est défini par la structure. Legolas++ fournit un ensemble d'accesseurs capables de faire ces vérifications avant de retourner le résultat de `lowerBandedGetElement`.

C.1.3 Création d'une matrice à plusieurs niveaux

Dans la section précédente nous avons vu comment définir une matrice à un niveau. Dans cette section, nous montrons comment construire une matrice à deux niveaux. Nous illustrons cette démarche en nous appuyant sur la création de la matrice Legolas++ correspondant à la matrice **A** introduite section 2.2 et dont nous rappelons la structure sur la *figure* C.1.

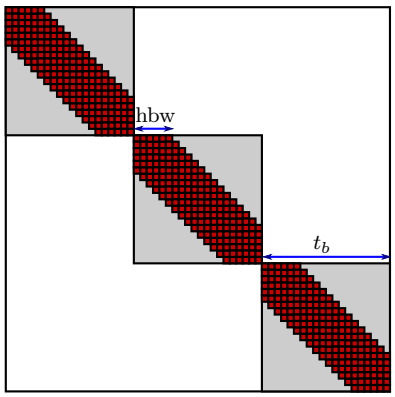


Figure C.1 : *Structure de la matrice* **A**.

Au premier niveau de la matrice se trouvent des blocs de matrice possédant une structure bande symétrique. Pour ce niveau, nous reprendrons la définition de matrice `MDefinition` introduite section 5.1.2.1. Il nous reste donc à créer `ADefinition`, la définition Legolas++ du second niveau de matrice.

Pour le second niveau, le type de réels ne change pas : il est nécessairement le même à tous les niveaux. Par contre, les éléments de la matrice sont de type `MDefinition::Data`. La classe imbriquée `ADefinition::Data` doit donc contenir les données nécessaires à l'instanciation d'éléments de type `MDefinition::Data`, c'est à dire qu'elle doit contenir la demi-largeur des blocs. Ceci montre qu'il est impossible de créer une définition générique ne dépendant que de la structure de niveau courante : la définition d'un niveau de matrice dépend de la définition des niveaux de matrices inférieurs.

```
class ADefinition : public Legolas::DefaultMatrixDefinition<float,2>{
  public:
    typedef Legolas::Diagonal           MatrixStructure;
    typedef typename MDefinition::Data  GetElement;
    // ShapeType est fourni par DefaultMatrixDefinition
    typedef MDefinition::ShapeType     ShapeType;

    struct Data : ShapeType
    {
```

```
      Data(const ShapeType & ms, const int blocksHbw)
        : ShapeType(ms), blocksHbw_(blocksHbw){}
      Data(const Data & data)
        : ShapeType(data),blocksHbw_(data.blocksHbw_){}
      const int blocksHbw_;
    };

    // accesseur optimise pour la structure MatrixStructure
    static inline GetElement diagonalGetElement(int i ,const Data & data) {
      return GetElement(data.getSubMatrixShape(i,i), data.blocksHbw_);
    }
};
```

C.2 Les solveurs

Pour qu'il soit intéressant d'utiliser une bibliothèque comme Legolas++ pour concevoir un solveur, il faut que les algorithmes utilisés puissent tirer partie de la structure des matrices. Nous présentons dans ce chapitre l'implémentation de l'algorithme de résolution utilisé par défaut dans Legolas++ : l'algorithme itératif dit de Gauss-Seidel [96, 97] que nous avons présenté page 27. À chaque itération, cet algorithme conduit à résoudre les systèmes linéaires correspondants aux blocs diagonaux de la matrice, et il est important de préserver les résultats partiels qui peuvent être réutilisés d'une itération sur l'autre. Les solveurs Legolas++ ont pour objectif de répondre à cette problématique en préservant les états internes des algorithmes entre leurs différentes exécutions. Les solveurs suivent la même construction hiérarchique que les matrices : chaque solveur correspond à un bloc de la matrice et contient d'une part les solveurs correspondants aux niveaux inférieurs de la matrice et d'autre part les données nécessaires à son algorithme.

Comme dans le cas des définitions de matrices vues précédemment, les classes définissant des solveurs sont relativement libres et peuvent par exemple faire appel à des bibliothèques externes. La seule contrainte pèse sur l'interface : les solveurs doivent répondre au concept `ALGORITHM_INV` défini ci-après en pseudo-code C++.

```
concept ALGORITHM_INV{
  public:
    template <class TA, class TX, class TB>
    class Engine : public Legolas::LinearSolver<TA,TX,TB> {
      public:
        inline Engine(const TA & A, TX & X, const TB & B );
        inline void solve(const TA & A, TX & X, const TB & B);
        inline void transposeSolve(const TA & A, TX & X, const TB & B);
    };
};
```

Ce concept introduit un patron de classe imbriqué nommé `Engine` [49]. Ce patron est paramétré par le type `TA` de la matrice, le type `TX` du vecteur inconnu et le type `TB` [50] du vecteur membre de droite de l'équation $AX = B$. Ce patron de classe comporte un constructeur et deux méthodes permettant de résoudre le problème. Le constructeur sert typiquement à allouer les sous-solveurs et les variables d'état du solveur courant. La fonction `solve` permet de résoudre le système $AX = B$ tandis que la fonction `transposeSolve` permet de résoudre le système

49. Le fait de paramétrer une sous-classe `Engine` plutôt que la classe `ALGORITHM_INV` permet de simplifier l'écriture et le débogage de Legolas++.

50. Les types `TX` et `TB` peuvent être différents puisque `TB` peut par exemple être une expression vectorielle.

$A^T X = B$. À titre d'exemple, la classe `SymmetricBandedGaussSeidelAlgorithm` qui suit contient une implémentation de l'algorithme de Gauss-Seidel spécialisé pour les matrices ayant une structure bande symétrique. Aucune contrainte ne pèse ici sur la structure des niveaux inférieurs de la matrice.

```
1  class SymmetricBandedGaussSeidelAlgorithm{
2    public :
3      template <class TA, class TX, class TB>
4      class Engine : public Legolas::LinearSolver<TA,TX,TB> {
5        private:
6          typedef typename TX::Element          XElement;
7          typedef typename TB::Element          BElement;
8          typedef typename TA::ConstGetElement  AElement;
9          typedef typename AElement::template SolverEngine<
10                             XElement,BElement>::Solver BlockSolver;
11         XElement accu_; // accumulateur utilise par l'algorithme
12         int nbSolvers_; // nombre de blocs sur la diagonale de A
13         BlockSolver * blockSolvers_; // tableau de sous-solveurs
14         TX B_estimate_; // utilise pour calculer l'erreur relative
15
16       public:
17         inline Engine(const TA & A, TX & X, const TB & B )
18           : ... //allocation et initialisation des membres donnees
19         { ... }
20
21         ~Engine(){ ... /* desallocation des membres donnees */ }
22
23         inline void solve(const TA & A, TX & X, const TB & B)
24         {
25           double relative_error;
26           do{
27             for(int i = 0 ; i < A.nrows() ; ++i){
28               int minj=std::max(0, i-A.lsup());
29               int maxj=std::min(nrows, i+A.lsup()+1);
30               accu_ = B[i];
31               for(int j = minj ; j<i ; ++j)
32                 accu_ -= A.lowerBandedGetElement(i,j)*X[j];
33               for(int j = i+1 ; j<maxj ; ++j)
34                 accu_ -= A.lowerBandedGetElement(j,i)*X[j];
35               // resolution du systeme A_{i,i}X_i = accu_
36               blockSolvers_[i].solve(A.lowerBandedGetElement(i,i),X[i],accu_);
37             }
38             B_estimate_ = A*X;
39             relative_error = dot(B_estimate_-B,B_estimate_-B)/dot(B,B);
40           } while (relative_error > 5.e-6);
41         }
42
43         inline void transposeSolve(const TA & A, TX & X, const TB & B)
44         { solve(A, X, B); } // A est symetrique
45     };
46 };
```

Le type des sous-solveurs est déterminé à la ligne 10. Ces sous-solveurs sont ensuite initialisés dans le constructeur et utilisés à la ligne 36 pour résoudre les sous-systèmes correspondant aux termes diagonaux de la matrice A. Ces sous-solveurs utilisent l'algorithme défini dans les options des sous-matrices. De cette manière, il est possible de construire récursivement un algorithme de résolution complexe, capable d'exploiter de manière optimale la structure d'une matrice. Il

est également possible de changer simplement l'algorithme utilisé à un niveau sans toucher aux autres niveaux ; ce qui peut être extrêmement compliqué avec d'autres approches.

Glossaire

AVX (*Advanced Vector eXtensions*) : extension SIMD du jeu d'instructions X86 qui succède aux unités SSE. Les unités AVX traitent des paquets de données de 256 bits. elles permettent donc d'exécuter simultanément huit opérations flottantes en simple précision ou quatre en double précision.

COCAGNE : nouvelle chaîne de calcul visant à simuler le cœur des réacteurs nucléaire d'EDF. COCAGNE doit succéder à COCCINELLE.

COCCINELLE : chaîne de calcul de cœur actuellement utilisée pour simuler le cœur des réacteurs nucléaire d'EDF.

CUDA C/C++ : extension aux langages C et C++ proposée par NVIDIA et permettant la programmation de GPUs.

GPU : *Graphic Processing Unit* ou processeur graphique.

INTEL TBB (*Threading Building Blocks*) : bibliothèque générique C++ proposée par INTEL et permettant la parallélisation d'une application sur plusieurs *threads*.

OpenCL : extension du langage C par le Kronos Group et permettant la programmation de processeurs parallèles (CPU, GPUs, Cell).

OpenMP : extension aux langages FORTRAN, C et C++ permettant de paralléliser une application sur plusieurs *threads*.

S_N : méthode de résolution de l'équation de Boltzmann et utilisée dans la chaîne COCAGNE. Cette méthode est plus exacte que le méthode SP_N mais bien plus coûteuse en temps de calcul.

SIMD (*Single Instruction Multiple Data*) : catégorie d'ordinateur définie par Flynn [100] et aujourd'hui utilisée pour caractériser les unités de calcul ou les processeurs qui appliquent une même instruction à des données différentes.

SP_N : méthode de résolution de l'équation de Boltzmann et utilisée dans la chaîne COCAGNE. Cette méthode de résolution offre un bon compromis entre temps de résolution et précision des résultats.

SSE (*Streaming SIMD Extensions*) : extension SIMD du jeu d'instructions X86. Les unités SSE traitent des paquets de données de 128 bits. elles permettent donc d'exécuter simultanément quatre opérations flottantes en simple précision ou deux en double précision.

Thread : suite d'instructions s'appliquant à un jeu de données et pouvant s'exécuter de manière concurrente à un autre *thread* sur les différents processeurs d'un ordinateur à mémoire partagée.

Bibliographie

[1] B. Thomas, C. Domain, Y. Souffez et P. Eon-Duval, « Parallèle ? je dirais même plus : massivement parallèle ! », *Épure*, octobre 1999, p. 3–13. Cité pages 1 et 3.

[2] EDF R&D. « Site internet de code-aster ». http://www.code-aster.org. Cité page 3.

[3] F. Archambeau, M. Sakiz et N. Mehitoua, « Code saturne : A finite volume code for turbulent flows », *International Journal On Finite Volumes*, vol. 1, n° 1, février 2004, p. 62. Cité page 3.

[4] EDF R&D. « Site internet de code-saturne ». http://www.code-saturne.org. Cité page 3.

[5] EDF R&D. « Site internet de syrthes ». http://rd.edf.com/syrthes. Cité page 3.

[6] J. Texeraud, M. Hypolite, S. Marguet *et al.*, « Coccinelle v3.9 : manuel d'utilisation ». Note Interne n° HI27-2010-00010, EDF R&D, 2010. Cité page 3.

[7] M. Guillo, Y. Pora et D. Couyras, « Documentation utilisateur de la version 1.0 de cocagne ». Note Interne n° HI27-2009-03967, EDF R&D, 2009. Cité page 3.

[8] A. Arnaud, E. Derocquigny, J.-p. Gaussot et C. Duquennoy, « Calcul d'incertitudes : illustration des opportunités de réduction de conservatismes grâce au calcul haute performance ». Note Interne n° H-T56-2008-00373-FR, EDF R&D, 2008. Cité page 3.

[9] P. Vezolle, S. Vialle et X. Warin, « Large Scale Experiment and Optimization of a Distributed Stochastic Control Algorithm. Application to Energy Management Problems. », dans *International workshop on Large-Scale Parallel Processing (LSPP 2009)*, p. 8 pages, Rome, Italy, May 29 2009. Cité page 3.

[10] C. Makassikis, S. Vialle et X. Warin, « Distribution of a Stochastic Control Algorithm Applied to Gas Storage Valuation », dans *The 7th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT).*, p. 493–498, Cairo, Egypt, dec 2007. IEEE. Cité page 3.

[11] O. Boiteau, « Indicateurs de performance d'un calcul (temps/mémoire) ». Documentation Code_Aster n° U1.03.03 r4223, EDF R&D, 2010. Cité page 3.

[12] O. Boiteau, « Notice d'utilisation du parallélisme ». Documentation Code_Aster n° U2.08.06 r5328, EDF R&D, 2011. Cité page 3.

[13] A. Assire, O. Boiteau, T. Desoza *et al.*, « Point sur le calcul hpc avec aster ». Compte-rendu du Comité Technique Aster, EDF R&D, février 2008. Cité page 3.

[14] C. Moulinec, C. Denis, C.-T. Pham *et al.*, « Telemac : An efficient hydrodynamics suite for massively parallel architectures », *Computers & Fluids*, vol. 51, n° 1, 2011, p. 30 – 34. Cité page 3.

[15] A. Bretault et T. Jouhanique, « Code coccinelle : Implication du module de thermohydrolique et optimisation du temps de calcul ». Note Interne n° HT-10/92/015/B, EDF R&D, 1992. Cité page 3.

[16] M. Guillo, Y. Pora, F. Hoareau et D. Couyras, « Note de vérification et de validation de la version 1.1 du code cocagne ». Note Interne n° HI27-2010-03511-FR, EDF R&D, 2010. Cité pages 3 et 4.

[17] M. Guillo, F. Hoareau et D. Couyras, « Ètude de faisabilité d'un calcul 3d de cœur crayon par crayon avec le code de cœur cocagne ». Note Interne n° HI27-2008-01726-FR, EDF R&D, 2008. Cité pages 3 et 4.

[18] S. Marguet, *La physique des réacteurs nucléaires*. Lavoisier, 2011. Cité pages 3 et 4.

[19] F. Hoareau, « COCAGNE : impact des éléments finis RTk différents par direction sur les calculs crayon par crayon 3D ». Note Interne n° CR-I27-2009-77, EDF R&D, 2009. Cité page 3.

[20] M. Barrault, « Performances de la version 1.0.3 de cocagne (séquentiel) sur deux stations de travail ». Compte Rendu interne n° CR-I23-2010-033, EDF R&D, 2010. Cité page 3.

[21] E. Gelbard, « Simplified Spherical Harmonics Equations and Their Use in Shielding Problems ». Rapport technique n° WAPD-T-1182 (Rev. 1), Westinghouse Electric Corp. Bettis Atomic Power Lab., Pittsburgh, 1961. Cité pages 4 et 93.

[22] E. Lewis et W. Miller Jr, *Computational methods of neutron transport*. American Nuclear Society, 1993. Cité page 4.

[23] B. Lathuilière, *Méthode de décomposition de domaine pour les équations du transport simplifié en neutronique*. Thèse de doctorat, LaBRI, Université Bordeaux I, Talence, France, janvier 2010. Cité pages 4 et 93.

[24] P. Guérin, *Méthodes de décomposition de domaine pour la formulation mixte duale du problème critique de la diffusion des neutrons*. Thèse de doctorat, Université Paris VI, 2007. Cité page 4.

[25] B. Carlson, « Solution of the transport equations by s_n approximations ». Rapport technique n° LA-1599, Los Alamos, 1953. Cité page 4.

[26] J. Askew, « A charasteristic formulation of the neutron transport equation in complicated geometries », n° AEEW-M 1108, 1972. Cité page 4.

[27] M. Halsall, « CACTUS, a charasteristic solution to the neutron transport equation in complicated geometries ». Rapport technique n° AEEW-R 1291, Atomic Energy Establishment, Winfrith, Dorchester, Dorcet, United Kingdom, 1980. Cité page 4.

[28] S. Hong et N. Cho, « Crx : A code for rectangular and hexagonal lattices based on the method of characteristics », *Annals of Nuclear Energy*, vol. 25, n° 8, 1998, p. 547–565. Cité page 4.

[29] F. Févotte, *Techniques de traçage pour la méthode des caractéristiques appliquée à la résolution de l'équation du transport des neutrons en domaines multi-dimensionnels*. Thèse de doctorat, Université Paris Sud - Paris XI, Orsay, France, 2009. Cité page 4.

[30] H. Joo, J. Cho, K. Kim *et al.*, « Methods and performance of a three-dimensional whole-core transport code decart », dans *Proceedings of PHYSOR 2004 - The Physics of Fuel Cycles and Advanced Nuclear Systems : Global Developments*, Chicago, Illinois, USA, April 2009. Cité page 4.

[31] M. Dahmani et R. Roy, « Parallel solver based on the three-dimensional characteristics method : Design and performance analysis », *Nuclear Science and Engineering*, vol. 150, n° 2, 2005. Cité page 4.

[32] M. Smith, A. Marin-Lafleche, W. Yang *et al.*, « Method of Characteristics Development Targeting the High Performance Blue Gene/P Computer at Argonne National Laboratory », dans *Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011)*, Rio de Janeiro, RJ, Brazil, May 2011. Cité page 4.

[33] L. Plagne et A. Ponçot, « Generic Programming for Deterministic Neutron Transport Codes », dans *Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*, Palais des Papes, Avignon, France, September 2005. Cité pages 4, 22, 24, 93 et 107.

[34] W. Kirschenmann, L. Plagne, S. Ploix *et al.*, « Massively Parallel Solving of 3D Simplified P_N Equations on Graphic Processing Units », dans *Proceedings of Mathematics, Computational Methods & Reactor Physics*, Saratoga Springs, New York, USA, May 2009. Cité pages 4, 28 et 52.

[35] M. Barrault, B. Lathuilière, P. Ramet et J. Roman, « A Non Overlapping Parallel Domain Decomposition Method Applied to The Simplified Transport Equations », dans *Proceedings of Mathematics, Computational Methods & Reactor Physics*, Saratoga Springs, New York, USA, May 2009. Cité page 4.

[36] W. Kirschenmann, L. Plagne et S. Vialle, « Parallel SP_N on Multi-Core CPUs and Many-Core GPUs », *Transport Theory and Statistical Physics*, vol. 39, n° 2, 2010, p. 255–281. Cité pages 4, 27 et 64.

[37] M. Barrault, B. Lathuilière, P. Ramet et J. Roman, « Efficient parallel resolution of the simplified transport equations in mixed-dual formulation », *Journal of Computational Physics*, vol. 230, n° 5, 2011, p. 2004 – 2020. Cité page 4.

[38] F. Blanchon et J. Planchard, « Méthodes numériques utilisées dans le code de cinétique neutronique coccinelle ». Note Interne n° HI/4109-07, EDF R&D, mars 1982. Cité page 4.

[39] J. Texeraud et M. Hypolite, « Cahier des charges négocié contractuel coccinelle v3.10 ». Note Interne n° HI27-2010-097, EDF R&D, 2010. Cité page 4.

[40] C. L. Lawson, « Background, Motivation, and a Retrospective View of the BLAS ». Rapport technique, SIAM, 1999. Cité page 6.

[41] C. L. Lawson, R. J. Hanson et F. T. Krogh, « A proposal for standard linear algebra subprograms », *ACM SIGNUM Newsletter*, vol. 8, 1973. Cité page 6.

[42] C. L. Lawson, R. J. Hanson, D. R. Kincaid et F. T. Krogh, « Basic linear algebra subprograms for fortran usage », *ACM Trans. Math. Softw.*, vol. 5, September 1979, p. 308–323. Cité page 6.

[43] J. J. Dongarra et S. C. Eisenstat, « Squeezing the most out of an algorithm in cray fortran », *ACM Trans. Math. Softw.*, vol. 10, August 1984, p. 219–230. Cité pages 6 et 10.

[44] J. J. Dongarra, J. D. Croz, S. Hammarling et R. J. Hanson, « A proposal for an extended set of fortran basic linear algebra subprograms », *SIGNUM Newsl.*, vol. 20, January 1985, p. 2–18. Cité page 6.

[45] J. J. Dongarra, J. Du Croz, S. Hammarling et R. J. Hanson, « An extended set of fortran basic linear algebra subprograms », *ACM Trans. Math. Softw.*, vol. 14, March 1988, p. 1–17. Cité page 6.

[46] J. J. Dongarra, J. Du Croz, I. Duff et S. Hammarling, « A proposal for a set of level 3 basic linear algebra subprograms », *SIGNUM Newsl.*, vol. 22, July 1987, p. 2–14. Cité pages 6 et 136.

[47] J. J. Dongarra, J. Du Croz, S. Hammarling et I. S. Duff, « A set of level 3 basic linear algebra subprograms », *ACM Trans. Math. Softw.*, vol. 16, March 1990, p. 1–17. Cité page 6.

[48] J. J. Dongarra, « Preface : Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I », *Int. J. High Performance Computing Applications*, vol. 16, n° 1, Spring 2002, p. 1–111. Cité page 6.

[49] J. J. Dongarra, « Preface : Basic Linear Algebra Subprograms Technical (Blast) Forum Standard II », *Int. J. High Performance Computing Applications*, vol. 16, n° 2, Summer 2002, p. 115–199. Cité page 6.

[50] INTEL, *Math Kernel Library (MKL) Documentation, version 10.3.* Cité page 7.

[51] AMD, *Math Kernel Library (ACML) Documentation – Version 4.4.0.* Cité page 7.

[52] IBM, *ESSL for AIX V5.1 – ESSL for Linux on POWER V5.1 – Guide and Reference.* Cité page 7.

[53] R. Whaley et J. J. Dongarra, « Automatically tuned linear algebra software », dans *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, p. 1–27, San Jose, CA, 1998. IEEE Computer Society. Cité page 7.

[54] K. Goto et R. A. van de Geijn, « Anatomy of high-performance matrix multiplication », *ACM Trans. Math. Softw.*, vol. 34, n° 3, 2008. Cité pages 7, 123 et 137.

[55] K. Goto et R. A. van de Geijn, « High-performance implementation of the level-3 blas », *ACM Trans. Math. Softw.*, vol. 35, n° 1, 2008. Cité pages 7 et 137.

[56] M. Frigo et S. G. Johnson, « The design and implementation of FFTW3 », *Proceedings of the IEEE*, vol. 93, n° 2, 2005, p. 216–231. Special issue on “Program Generation, Optimization, and Platform Adaptation”. Cité page 7.

[57] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., édition 3rd, 2009. Cité page 7.

[58] Netlib. « Site internet de netlib ». http://www.netlib.org/. Cité page 8.

[59] OpenMP Architecture Review Board, *OpenMP Application Program Interface, version 3.0*, 2008. Cité pages 8, 38 et 77.

[60] L. Plagne, F. Hülsemann, D. Barthou et J. Jaeger, « Parallel expression template for large vectors », dans *POOSC '09 : Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, p. 8, Genova, Italy, 2009. ACM. Cité pages 8 et 150.

[61] J. D. McCalpin, « Memory Bandwidth and Machine Balance in Current High Performance Computers », *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, dec 1995, p. 19–25. Cité pages 8 et 62.

[62] S. P. Jones, éditeur, *Haskell 98 Language and Libraries : The Revised Report*. http ://haskell.org/, September 2002. Cité page 12.

[63] G. Keller, M. M. Chakravarty, R. Leshchinskiy *et al.*, « Regular, shape-polymorphic, parallel arrays in haskell », dans *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming, ICFP '10*, p. 261–272, Baltimore, Maryland, USA, 2010. ACM. Cité page 12.

[64] P. Narbel, *Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml*. Vuibert, 2005. Cité page 13.

[65] MathWorks, *MATLAB - The Language Of Technical Computing Documentation, version 2009b*. Cité page 13.

[66] S. Consortium, *Scilab 5.2 Documentation*. Cité page 13.

[67] A. van Deursen, P. Klint et J. Visser, « Domain-specific languages : an annotated bibliography », *SIGPLAN Not.*, vol. 35, June 2000, p. 26–36. Cité page 13.

[68] P. Hudak, « Building domain-specific embedded languages », *ACM Comput. Surv.*, vol. 28, December 1996. Cité page 15.

[69] K. Czarnecki, J. T. Odonnell, J. Striegnitz *et al.*, « DSL Implementation in MetaOCaml, Template Haskell, and C++ », *LNCS : Domain-Specific Program Generation*, vol. 3016, n° 2, 2004, p. 51–72. Cité page 15.

[70] P. Hudak, « Modular domain specific languages and tools », dans *Proceedings of the 5th International Conference on Software Reuse*, coll. « ICSR '98 », p. 134–, Washington, DC, USA, 1998. IEEE Computer Society. Cité page 15.

[71] J. Falcou, *Un Cluster pour la Vision Temps Réel : Architecture, Outils et Applications*. Thèse de doctorat, l'Université Blaise Pascal (Clermont II), Clermont-Ferrand, France, décembre 2006. Cité pages 16, 17, 71 et 72.

[72] J. Falcou, J. Sérot, L. Pech et J.-T. Lapresté, « Meta-programming applied to automatic smp parallelization of linear algebra code », dans E. Luque, T. Margalef et D. Benítez, éditeurs, *Euro-Par 2008 – Parallel Processing*, vol. 5168 (coll. *Lecture Notes in Computer Science*), p. 729–738. Springer Berlin / Heidelberg, 2008. Cité pages 16, 17 et 71.

[73] P. Plauger, M. Lee, D. Musser et A. A. Stepanov, *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, édition 1st, 2000. Cité pages 17, 71, 106 et 147.

[74] ISO, *ISO/IEC 14882 :2003 : Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003. Cité pages 17, 55, 71, 108 et 147.

[75] T. L. Veldhuizen et D. Gannon, « Active libraries : Rethinking the roles of compilers and libraries », dans *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM, 1998. Cité pages 17 et 147.

[76] K. Czarnecki, U. W. Eisenecker, R. Glück *et al.*, « Generative programming and active libraries », dans *Selected Papers from the International Seminar on Generic Programming*, p. 25–39, London, UK, 2000. Springer-Verlag. Cité pages 17 et 147.

[77] T. L. Veldhuizen, *Active libraries and universal languages*. Thèse de doctorat, Indianapolis, IN, USA, 2004. AAI3134053. Cité pages 17 et 147.

[78] T. L. Veldhuizen, « Arrays in Blitz++ », dans *ISCOPE '98 : Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, p. 223–230, London, UK, 1998. Springer-Verlag. Cité pages 17, 103 et 147.

[79] J. O. Coplien, « Curiously recurring template patterns », *C++ Rep.*, vol. 7, February 1995, p. 24–27. Cité pages 17 et 147.

[80] J. O. Coplien, *Curiously recurring template patterns*, p. 135–144. New York, NY, USA, SIGS Publications, Inc., 1996. Cité pages 17 et 147.

[81] M. A. Heroux, R. A. Bartlett, V. E. Howle *et al.*, « An overview of the Trilinos project », *ACM Trans. Math. Softw.*, vol. 31, n° 3, 2005, p. 397–423. Cité pages 17 et 76.

[82] C. G. Baker, H. Carter Edwards, M. A. Heroux et A. B. Williams, « A light-weight API for Portable Multicore Programming », dans *PDP 2010 : Proceedings of The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Washington, DC, USA, 2010. IEEE Computer Society. Cité pages 17 et 76.

[83] S. Blackford, G. Corliss, J. Demmel *et al.*, « Basic linear algebra subprograms technical forum standard », *International Journal of High Performance Applications and Supercomputing*, vol. 16, n° 1, Spring 2002. Cité pages 18 et 19.

[84] G. M. Morton, « A computer oriented geodetic data base ; and a new technique in file sequencing ». Rapport technique, IBM Canada Ltd., Ottawa, Canada, 1966. Cité pages 19 et 106.

[85] P. Gottschling, D. S. Wise et M. D. Adams, « Representation-transparent matrix algorithms with scalable performance », dans *Proceedings of the 21st annual international conference on Supercomputing*, coll. « ICS '07 », p. 116–125, Seattle, Washington, 2007. ACM. Cité page 19.

[86] J. Herrero et J. Navarro, « Using non-canonical array layouts in dense matrix operations », dans B. Kågström, E. Elmroth, J. J. Dongarra et J. Wasniewski, éditeurs, *Applied Parallel Computing. State of the Art in Scientific Computing*, vol. 4699 (coll. *Lecture Notes in Computer Science*), p. 580–588. Springer Berlin / Heidelberg, 2007. Cité page 19.

[87] J. J. Dongarra, « Sparse Matrix Storage Formats (section 10.2) », dans Bai *et al.* [88], p. 315–319. Cité page 19.

[88] Z. Bai, J. Demmel, J. J. Dongarra *et al.*, éditeurs, *Templates for the solution of algebraic eigenvalue problems : a practical guide*. SIAM, Philadelphia, 2000. Cité pages 19 et 165.

[89] P. Hénon, P. Ramet et J. Roman, « PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems », *Parallel Computing*, vol. 28, n° 2, janvier 2002, p. 301–321. Cité page 21.

[90] F. Pellegrini et J. Roman, « Sparse matrix ordering with Scotch », dans *Proceedings of HPCN'97, Vienna, LNCS 1225*, p. 370–378, avril 1997. Cité page 21.

[91] G. Karypis et V. Kumar, « A parallel algorithm for multilevel graph partitioning and sparse matrix ordering », *J. Parallel Distrib. Comput.*, vol. 48, n° 1, 1998, p. 71–95. Cité page 21.

[92] S. Balay, W. D. Gropp, L. C. McInnes et B. F. Smith, « Efficient management of parallelism in object oriented numerical software libraries », dans E. Arge, A. M. Bruaset et H. P. Langtangen, éditeurs, *Modern Software Tools in Scientific Computing*, p. 163–202. Birkhäuser Press, 1997. Cité page 22.

[93] S. Balay, J. Brown, *et al.*, « PETSc users manual ». Rapport technique n° ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010. Cité page 22.

[94] S. Balay, J. Brown, K. Buschelman *et al.* « PETSc Web page », 2011. http ://www.mcs.anl.gov/petsc. Cité page 22.

[95] M. Benzi, G. H. Golub et J. Liesen, « Numerical solution of saddle point problems », *Acta Numerica*, vol. 14, 2005, p. 1–137. Cité page 23.

[96] G. H. Golub et C. F. Van Loan, *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. Cité pages 27, 107 et 156.

[97] W. H. Press, S. A. Teukolsky, W. T. Vetterling et B. P. Flannery, *Numerical Recipes 3rd Edition : The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3ème édition, 2007. Cité pages 27, 107 et 156.

[98] G. E. Moore, « Progress in digital integrated electronics. », dans *IEEE International Electron Devices Meeting 21*, coll. « Technical Digest », p. 11–13. IEEE, 1975. Cité page 31.

[99] J. M. Rabaey, A. Chandrakasan et B. Nikolic, *Digital Integrated Circuits, second edition*. Prentice-Hall, Upper Saddle River, New Jersey, USA, 2003. Cité page 32.

[100] M. J. Flynn, « Some computer organizations and their effectiveness », *IEEE Trans. Comput.*, vol. 21, September 1972, p. 948–960. Cité pages 32, 36 et 159.

[101] R. Duncan, « A survey of parallel computer architectures », *Computer*, vol. 23, February 1990, p. 5–16. Cité pages 32 et 36.

[102] INTEL, *INTEL C++ Intrinsics Reference*, 2007. 312482-002US. Cité pages 32, 36, 37, 38 et 39.

[103] INTEL, *INTEL Advanced Vector Extensions – Programming Reference*, June 2011. 319433-011. Cité pages 32, 36 et 37.

[104] K. Olukotun, B. A. Nayfeh, L. Hammond *et al.*, « The case for a single-chip multiprocessor », *SIGPLAN Not.*, vol. 31, September 1996, p. 2–11. Cité pages 32 et 38.

[105] S. F. Anderson, J. G. Earle, R. E. Goldschmidt et D. M. Powers, « The IBM system/360 model 91 : Floating-point execution unit », *IBM J. Res. Dev.*, vol. 11, n° 1, 1967, p. 34. Cité page 33.

[106] G. R.A., « The challenge of computer graphics in continental western europe », dans *Proceedings of the IEEE*, p. 421–428. IEEE, April 1974. Cité page 34.

[107] J. Makino, T. Fukushige et M. Koga, « A 1.349 tflops simulation of black holes in a galactic center on grape-6 », dans *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, coll. « Supercomputing '00 », Dallas, Texas, United States, 2000. IEEE Computer Society. Cité page 34.

[108] B. Bishop, T. P. Kelliher et M. J. Irwin, « Sparta : Simulation of physics on a real-time architecture », dans *Proceedings of the 10th Great Lakes symposium on VLSI*, coll. « GLSVLSI '00 », p. 177–182, Chicago, Illinois, United States, 2000. ACM. Cité page 34.

[109] S. Yardi, B. Bishop et T. Kelliher, « Hellas : a specialized architecture for interactive deformable object modeling », dans *Proceedings of the 44th annual Southeast regional conference*, coll. « ACM-SE 44 », p. 56–61, Melbourne, Florida, 2006. ACM. Cité page 34.

[110] S. Hauck et A. DeHon, *Reconfigurable Computing : The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. Cité page 34.

[111] S. Benner, R. J. A. Chen, N. A. Wilson *et al.*, « Sequence-specific detection of individual DNA polymerase complexes in real time using a nanopore », *Nature Nanotechnology*, vol. 2, novembre 2007, p. 718–724. Cité page 34.

[112] J. A. Kahle, M. N. Day, H. P. Hofstee *et al.*, « Introduction to the cell multiprocessor », *IBM J. Res. Dev.*, vol. 49, July 2005, p. 589–604. Cité page 34.

[113] T. Chen, R. Raghavan, J. N. Dale et E. Iwata, « Cell broadband engine architecture and its first implementation : a performance view », *IBM J. Res. Dev.*, vol. 51, September 2007, p. 559–572. Cité page 34.

[114] S. Williams, J. Shalf, L. Oliker *et al.*, « The potential of the cell processor for scientific computing », dans *Proceedings of the 3rd conference on Computing frontiers*, coll. « CF '06 », p. 9–20, New York, NY, USA, 2006. ACM. Cité page 34.

[115] E. Kilgariff et R. Fernando, « The geforce 6 series gpu architecture », dans *ACM SIGGRAPH 2005 Courses*, coll. « SIGGRAPH '05 », New York, NY, USA, 2005. ACM. Cité page 35.

[116] E. Lindholm, J. Nickolls, S. Oberman et J. Montrym, « Nvidia tesla : A unified graphics and computing architecture », *IEEE Micro*, vol. 28, March 2008, p. 39–55. Cité page 35.

[117] NVIDIA, « NVIDIA's Next Generation CUDA Compute Architecture : Fermi ». Whitepaper, NVIDIA corp., 2009. Cité pages 35, 41, 42, 43 et 44.

[118] M. Faverge, *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs*. Thèse de doctorat, LaBRI, Université Bordeaux I, Talence, Talence, France, 2009. Cité page 35.

[119] M. Faverge et P. Ramet, « A NUMA aware scheduler for a parallel sparse direct solver », *Parallel Computing*, 2009. Submitted. Cité page 35.

[120] S. Weiss et J. E. Smith, « Instruction issue logic for pipelined supercomputers », *SIGARCH Comput. Archit. News*, vol. 12, January 1984, p. 110–118. Cité page 35.

[121] R. J. Fisher et H. G. Dietz, « Compiling for simd within a register », dans *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, coll. « LCPC '98 », p. 290–304, London, UK, 1999. Springer-Verlag. Cité page 36.

[122] D. Koufaty et D. Marr, « Hyperthreading technology in the netburst microarchitecture », *Micro, IEEE*, vol. 23, n° 2, 2003, p. 56–65. Cité page 38.

[123] J. Nickolls et W. J. Dally, « The gpu computing era », *IEEE Micro*, vol. 30, March 2010, p. 56–69. Cité page 40.

[124] S. Upstill, *RenderMan Companion : A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. Cité page 40.

[125] Pixar, *The RenderMan Interface*, édition version 3.1 specification, September 1989. Cité page 40.

[126] A. A. Apodaca et M. W. Mantle, « Renderman : Pursuing the future of graphics », *IEEE Comput. Graph. Appl.*, vol. 10, July 1990, p. 44–49. Cité page 40.

[127] M. Henne, H. Hickel, E. Johnson et S. Konishi, « The making of Toy Story », dans *COMPCON '96. Technologies for the Information Superhighway. Forty-First IEEE Computer Society International Conference.*, p. 463–468, Seattle, Washington, 1996. IEEE Comput. Soc. Press. Los Alamitos, CA, USA. Cité page 40.

[128] OpenGL Architecture Review Board, *OpenGL reference manual : the official reference document for OpenGL, release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992. Cité page 40.

[129] H. Yoshizawa, T. Otsuka et S. Sasaki, « High-Performance Architecture of a Next-Generation 3D-CG Rendering Processor. », dans *Proceedings of ICSPAT'95*, p. 195–199, 1995. Cité page 40.

[130] T. Mitra et T.-c. Chiueh, « A breadth-first approach to efficient mesh traversal », dans *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, coll. « HWWS '98 », p. 31–38, New York, NY, USA, 1998. ACM. Cité page 40.

[131] M. S. Peercy, M. Olano, J. Airey et P. J. Ungar, « Interactive multi-pass programmable shading », dans *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, coll. « SIGGRAPH '00 », p. 425–432, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. Cité page 41.

[132] M. Pharr et R. Fernando, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. Cité page 41.

[133] NVIDIA, *NVIDIA CUDA C Programming Guide 4.0*, 2011. Cité pages 41, 42, 43, 45, 55, 79 et 122.

[134] S. Collange, *Enjeux de conception des architectures GPGPU : unités arithmétiques spécialisées et exploitation de la régularité*. Thèse de doctorat, Université de Perpignan Via Domitia, Perpignan, France, novembre 2010. Cité page 42.

[135] D. M. Tullsen, S. J. Eggers et H. M. Levy, « Simultaneous multithreading : maximizing on-chip parallelism », *SIGARCH Comput. Archit. News*, vol. 23, May 1995, p. 392–403. Cité page 42.

[136] H. Hirata, K. Kimura, S. Nagamine *et al.*, « An elementary processor architecture with simultaneous instruction issuing from multiple threads », *SIGARCH Comput. Archit. News*, vol. 20, April 1992, p. 136–145. Cité page 43.

[137] Mark Harris. « Optimizing CUDA », Novembre 2007. Tutoriel CUDA présenté à SuperComputing 2007 (SC'07). Cité pages 45, 49 et 52.

[138] W. J. Bolosky et M. L. Scott, « False sharing and its effect on shared memory performance », dans *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, p. 3–3, Berkeley, CA, USA, 1993. USENIX Association. Cité page 48.

[139] J. Reinders, *INTEL threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007. Cité pages 52, 76, 103 et 150.

[140] Y. Magda, *Visual C++ Optimization with Assembly Code*. A-List Publishing, 2004. Cité page 52.

[141] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien *et al.*, « Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture », *IBM Syst. J.*, vol. 45, January 2006, p. 59–84. Cité page 52.

[142] M. Hassaballah, S. Omran et Y. B. Mahdy, « A review of simd multimedia extensions and their usage in scientific and engineering applications », *Comput. J.*, vol. 51, November 2008, p. 630–649. Cité page 52.

[143] C. H. Q. Ding, « An optimal index reshuffle algorithm for multidimensional arrays and its applications for parallel architectures », *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, March 2001, p. 306–315. Cité page 52.

[144] D. Fraser, « Array permutation by index-digit permutation », *J. ACM*, vol. 23, April 1976, p. 298–309. Cité page 52.

[145] A. Edelman, S. Heller et S. L. Johnsson, « Index transformation algorithms in a linear algebra framework », *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, December 1994, p. 1302–1309. Cité page 52.

[146] S. Lennart Johnsson et C.-T. Ho, « Algorithms for matrix transposition on boolean n-cube configured ensemble architecture », *SIAM J. Matrix Anal. Appl.*, vol. 9, November 1988, p. 419–454. Cité page 52.

[147] M. Bader, H.-J. Bungartz, D. Mudigere *et al.*, « Fast GPGPU data rearrangement kernels using CUDA », *ArXiv e-prints*, novembre 2010. Cité page 52.

[148] P. Lascaux et R. Théodor, *Analyse numérique matricielle appliquée à l'art de l'ingénieur vol 1 méthodes directes*. Masson, 2ème édition, 1998. Cité page 53.

[149] G. H. Golub et C. F. Van Loan, *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 3ème édition, 1996. Cité page 53.

[150] J. Falcou et J. Sérot, « EVE, an Object Oriented SIMD Library », dans M. Bubak, G. D. v. Albada, P. M. A. Sloot et J. J. Dongarra, éditeurs, *Computational Science - ICCS 2004*, vol. 3038 (coll. *Lecture Notes in Computer Science*), p. 314–321. Springer Berlin / Heidelberg, 2004. Cité pages 60 et 71.

[151] INTEL, *INTEL C++ Compiler 12.0 User and Reference Guides*. Cité page 60.

[152] J. D. McCalpin, « Stream : Sustainable memory bandwidth in high performance computers ». Rapport technique, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http ://www.cs.virginia.edu/stream/. Cité page 62.

[153] M. E. Thomadakis, « The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms ». Rapport technique, Texas A&M University, 2010. Cité page 63.

[154] INTEL, « Intel 64 and ia-32 architectures software developer's manual volume 1 : Basic architecture ». Rapport technique, 2010. Cité page 63.

[155] INTEL, « Intel 64 and ia-32 architectures software developer's manual volumes 2a and 2b : Instruction set reference, a-z ». Rapport technique, 2010. Cité page 63.

[156] Linley Gwennap, « Sandy Bridge Spans Generations », *Processor Watch*, n° 328, 2010. Cité pages 63 et 136.

[157] NVIDIA, *CUDA CUBLAS library 4.0*, 2011. Cité page 63.

[158] INTEL, *INTEL VTune Amplifier XE*, 2011. Cité page 66.

[159] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Cité page 71.

[160] E. A. Lee, « The Problem with Threads ». Rapport technique n° UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5) :33-42, May 2006. Cité page 71.

[161] H.-J. Boehm, « Threads cannot be implemented as a library », *SIGPLAN Not.*, vol. 40, June 2005, p. 261–268. Cité page 71.

[162] S. Savage, M. Burrows, G. Nelson *et al.*, « Eraser : a dynamic data race detector for multi-threaded programs », *ACM Trans. Comput. Syst.*, vol. 15, November 1997, p. 391–411. Cité page 71.

[163] H. Jula, D. Tralamazza, C. Zamfir et G. Candea, « Deadlock immunity : enabling systems to defend against deadlocks », dans *Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08*, p. 295–308, San Diego, California, 2008. USENIX Association. Cité page 71.

[164] K. Serebryany et T. Iskhodzhanov, « Threadsanitizer : data race detection in practice », dans *Proceedings of the Workshop on Binary Instrumentation and Applications*, coll. « WBIA '09 », p. 62–71, New York, New York, 2009. ACM. Cité page 71.

[165] A. Jannesari et W. F. Tichy, « On-the-fly race detection in multi-threaded programs », dans *Proceedings of the 6th workshop on Parallel and distributed systems : testing, analysis, and debugging*, coll. « PADTAD '08 », p. 6 :1–6 :10, Seattle, Washington, 2008. ACM. Cité page 71.

[166] A. Mühlenfeld et F. Wotawa, « Fault Detection in Multi-Threaded C++ Server Applications », *Electronic Notes in Theoretical Computer Science*, vol. 174, n° 9, 2007, p. 5 – 22. Cité page 71.

[167] D. D. Chamberlin et R. F. Boyce, « Sequel : A structured english query language », dans *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, coll. « SIGFIDET '74 », p. 249–264, Ann Arbor, Michigan, 1974. ACM. Cité page 71.

[168] E. Meijer, B. Beckman et G. Bierman, « Linq : reconciling object, relations and xml in the .net framework », dans *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, coll. « SIGMOD '06 », p. 706–706, New York, NY, USA, 2006. ACM. Cité page 71.

[169] P. Pialorsi et M. Russo, *Introducing microsoft linq*. Microsoft Press, Redmond, WA, USA, édition first, 2007. Cité page 71.

[170] Y. Yu, M. Isard, D. Fetterly *et al.*, « DryadLINQ : a system for general-purpose distributed data-parallel computing using a high-level language », dans *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, coll. « OSDI'08 », p. 1–14, San Diego, California, 2008. USENIX Association. Cité page 71.

[171] M. Isard et Y. Yu, « Distributed data-parallel computing using a high-level programming language », dans *Proceedings of the 35th SIGMOD international conference on Management of data*, coll. « SIGMOD '09 », p. 987–994, Providence, Rhode Island, USA, 2009. ACM. Cité page 71.

[172] M. Isard, M. Budiu, Y. Yu *et al.*, « Dryad : distributed data-parallel programs from sequential building blocks », *SIGOPS Oper. Syst. Rev.*, vol. 41, March 2007, p. 59–72. Cité page 71.

[173] M. Isard, M. Budiu, Y. Yu *et al.*, « Dryad : distributed data-parallel programs from sequential building blocks », dans *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, coll. « EuroSys '07 », p. 59–72, Lisbon, Portugal, 2007. ACM. Cité page 71.

[174] Querydsl. « Site internet de querydsl ». http://www.querydsl.com/. Cité page 71.

[175] LINQ++. « LINQ++ : An embeded dsl for c++ (site internet de LINQ++) ». https://github.com/hjiang/linqxx/wiki. Cité page 71.

[176] T. Faison, *Event-Based Programming : Taking Events to the Limit*. Apress, Berkely, CA, USA, 2006. Cité page 73.

[177] J. Armstrong et S. Virding, « Erlang - an experimental telephony programming language », dans *XIII International Switching Symposium*, p. 43 – 48, 1990. Cité page 74.

[178] R. Virding, C. Wikström et M. Williams, *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. Cité page 74.

[179] L. V. Kale et S. Krishnan, « Charm++ : a portable concurrent object oriented system based on C++ », dans *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, coll. « OOPSLA '93 », p. 91–108, Washington, D.C., United States, 1993. ACM. Cité page 74.

[180] L. V. Kale et S. Krishnan, « Charm++ : a portable concurrent object oriented system based on C++ », *SIGPLAN Not.*, vol. 28, October 1993, p. 91–108. Cité page 74.

[181] D. Kunzman, « Charm++ on the Cell Processor ». Rapport de DÉA, Dept. of Computer Science, University of Illinois, 2006. Cité page 74.

[182] D. Kunzman, G. Zheng, E. Bohm et L. V. Kalé, « Charm++, Offload API, and the Cell Processor », dans *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006. Cité page 74.

[183] L. Wesolowski, « An application programming interface for general purpose graphics processing units in an asynchronous runtime system ». Rapport de DÉA, Dept. of Computer Science, University of Illinois, 2008. Cité page 74.

[184] M. Cole, *Algorithmic skeletons : structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991. Cité page 74.

[185] H. González-Vélez et M. Leyton, « A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers », *Softw. Pract. Exper.*, vol. 40, November 2010, p. 1135–1160. Cité pages 74 et 75.

[186] E. Gamma, R. Helm, R. Johnson et J. Vlissides, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. Cité page 74.

[187] S. MacDonald, D. Szafron, J. Schaeffer et S. Bromling, « Generating parallel program frameworks from parallel design patterns », dans *Euro-Par '00 : Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, p. 95–104, London, UK, 2000. Springer-Verlag. Cité page 75.

[188] J. Falcou, J. Sérot, T. Chateau et J.-T. Lapresté, « Quaff : efficient C++ design for parallel skeletons », *Parallel Computing*, vol. 32, n° 7-8, 2006, p. 604–615. Cité page 75.

[189] M. D. McCool, « Structured parallel programming with deterministic patterns », dans *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, coll. « HotPar'10 », p. 5–5, Berkeley, CA, 2010. USENIX Association. Cité page 75.

[190] E. W. Dijkstra, « Go To statement considered harmful », *Comm. ACM*, vol. 11, n° 3, 1968, p. 147–148. Letter to the Editor. Cité page 75.

[191] L. Bougé, « The data-parallel programming model : A semantic perspective », dans A. Darte et G.-R. Perrin, éditeurs, *The Data-Parallel Programming Model*, vol. 1132 (coll. *LNCS Tutorial*), p. 4–26. Springer Verlag, juin 1996. Invited Conference. Cité page 75.

[192] J. Dean et S. Ghemawat, « MapReduce : simplified data processing on large clusters », dans *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, p. 10–10, San Francisco, CA, 2004. USENIX Association. Cité pages 75 et 91.

[193] J. Dean et S. Ghemawat, « MapReduce : simplified data processing on large clusters », *Commun. ACM*, vol. 51, January 2008, p. 107–113. Cité page 75.

[194] G. H. Botorog et H. Kuchen, « Efficient high-level parallel programming », *Theoretical Computer Science*, vol. 196, n° 1-2, 1998, p. 71 – 107. Cité page 75.

[195] B. Bacci, M. Danelutto, S. Orlando *et al.*, « p^3l : A structured high-level parallel language, and its structured support », *Concurrency - Practice and Experience*, vol. 7, n° 3, 1995, p. 225–255. Cité page 75.

[196] R. Loogen, Y. Ortega-mallén et R. Peña marí, « Parallel functional programming in Eden », *J. Funct. Program.*, vol. 15, May 2005, p. 431–475. Cité page 75.

[197] L. Rauchwerger, F. Arzu et K. Ouchi, « Standard Templates Adaptive Parallel Library (stapl) », dans *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, coll. « LCR '98 », p. 402–409, London, UK, 1998. Springer-Verlag. Cité page 76.

[198] P. An, A. Jula, S. Rus *et al.*, « STAPL : an adaptive, generic parallel C++ library », dans *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, coll. « LCPC'01 », p. 193–208, Cumberland Falls, KY, USA, 2003. Springer-Verlag. Cité page 76.

[199] A. Buss, Harshvardhan, I. Papadopoulos *et al.*, « STAPL : Standard Template Adaptive Parallel Library », dans *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, coll. « SYSTOR '10 », p. 14 :1–14 :10, Haifa, Israel, 2010. ACM. Cité page 76.

[200] N. Thomas, G. Tanase, O. Tkachyshyn *et al.*, « A framework for adaptive algorithm selection in STAPL », dans *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, coll. « PPoPP '05 », p. 277–288, Chicago, IL, USA, 2005. ACM. Cité page 76.

[201] A. Buss, A. Fidel, Harshvardhan *et al.* « The STAPL pView », 2010. Cité pages 76 et 90.

[202] J. Hoberock et N. Bell. Thrust : *http ://code.google.com/p/thrust/*. Cité page 76.

[203] J. Enmyren et C. W. Kessler, « SkePU : a multi-backend skeleton programming library for multi-GPU systems », dans *Proceedings of the fourth international workshop on High-level parallel programming and applications*, coll. « HLPP '10 », p. 5–14, Baltimore, Maryland, USA, 2010. Cité page 76.

[204] D. V. Dyk, M. Geveler, S. Mallach *et al.*, « HONEI : A collection of libraries for numerical computations targeting multiple processor architectures », *Computer Physics Communications*, vol. 180, n° 12, 2009, p. 2534–2543. Cité page 76.

[205] M. Süss et C. Leopold, « A user's experience with parallel sorting and OpenMP », dans *Proceedings of the Sixth European Workshop on OpenMP - EWOMP'04*, p. 23–38, Stockholm, 2004. Cité page 77.

[206] A. Basumallik, S.-J. Min et R. Eigenmann, « Towards OpenMP execution on software distributed shared memory systems », dans H. Zima, K. Joe, M. Sato *et al.*, éditeurs, *High Performance Computing*, vol. 2327 (coll. *Lecture Notes in Computer Science*), p. 357–362. Springer Berlin / Heidelberg, 2006. Cité page 77.

[207] D. Millot, A. Muller, C. Parrot et F. Silber-Chaussumier, « Step : A distributed OpenMP for coarse-grain parallelism tool », dans R. Eigenmann et B. de Supinski, éditeurs, *OpenMP in a New Era of Parallelism*, vol. 5004 (coll. *Lecture Notes in Computer Science*), p. 83–99. Springer Berlin / Heidelberg, 2008. Cité page 77.

[208] M. Sato, S. Satoh, K. Kusano et Y. Tanaka, « Design of OpenMP compiler for an SMP cluster », dans *Proceedings of First European Workshop on OpenMP (EWOMP)*, Lund, Sweden, 1999. Cité page 77.

[209] A. Basumallik et R. Eigenmann, « Towards automatic translation of OpenMP to MPI », dans *Proceedings of the 19th annual international conference on Supercomputing*, coll. « ICS '05 », p. 189–198, Cambridge, Massachusetts, 2005. ACM. Cité page 77.

[210] D. Margery, G. Vallée, R. Lottiaux *et al.*, « Kerrighed : A SSI Cluster OS Running OpenMP », dans *Proceeding of the Fifth European Workshop on OpenMP (EWOMP'03)*, 2003. Cité page 77.

[211] S. Lee, S.-J. Min et R. Eigenmann, « Openmp to gpgpu : a compiler framework for automatic translation and optimization », *SIGPLAN Not.*, vol. 44, February 2009, p. 101–110. Cité page 77.

[212] E. Ayguadé, R. M. Badia, F. D. Igual *et al.*, « An extension of the starss programming model for platforms with multiple gpus », dans *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, coll. « Euro-Par '09 », p. 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. Cité page 77.

[213] E. Ayguade, R. M. Badia, D. Cabrera *et al.*, « A proposal to extend the openmp tasking model for heterogeneous architectures », dans *Proceedings of the 5th International Workshop on OpenMP : Evolving OpenMP in an Age of Extreme Parallelism*, coll. « IWOMP '09 », p. 154–167, Berlin, Heidelberg, 2009. Springer-Verlag. Cité page 77.

[214] F. Cappello et D. Etiemble, « Mpi versus mpi+openmp on ibm sp for the nas benchmarks », dans *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, coll. « Supercomputing '00 », Washington, DC, USA, 2000. IEEE Computer Society. Cité page 77.

[215] G. Krawezik, « Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors », dans *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, coll. « SPAA '03 », p. 118–127, New York, NY, USA, 2003. ACM. Cité page 77.

[216] F. Bodin et S. Bihan, « Heterogeneous multicore parallel programming for graphics processing units », *Sci. Program.*, vol. 17, December 2009, p. 325–336. Cité page 77.

[217] OpenHMPP Consortium, *OpenHMPP Concepts & Directives, version 1.0*, 06 2011. Cité page 77.

[218] OpenACC, *The OpenACC^{TM} Application Programming Interface*, 11 2011. Cité page 77.

[219] S. Peyton Jones, « Harnessing the multicores : Nested data parallelism in haskell », dans G. Ramalingam, éditeur, *Programming Languages and Systems*, vol. 5356 (coll. *Lecture Notes in Computer Science*), p. 138–138. Springer Berlin / Heidelberg, 2008. Cité page 78.

[220] S. Marlow, S. Peyton Jones et S. Singh, « Runtime support for multicore haskell », dans *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, coll. « ICFP '09 », p. 65–78, Edinburgh, Scotland, 2009. ACM. Cité page 78.

[221] S. Marlow, S. Peyton Jones et S. Singh, « Runtime support for multicore haskell », *SIGPLAN Not.*, vol. 44, August 2009, p. 65–78. Cité page 78.

[222] G. E. Blelloch, « NESL : A nested data-parallel language (version 2.6) ». Rapport technique n° CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, avril 1993. Cité page 78.

[223] G. E. Blelloch, S. Chatterjee, J. C. Hardwick *et al.*, « Implementation of a portable nested data-parallel language », *Journal of Parallel and Distributed Computing*, vol. 21, n° 1, avril 1994, p. 4–14. Cité page 78.

[224] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones *et al.*, « Data Parallel Haskell : a status report », dans *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, coll. « DAMP '07 », p. 10–18, Nice, France, 2007. ACM. Cité page 78.

[225] M. M. Chakravarty, G. Keller, S. Lee *et al.*, « Accelerating Haskell array codes with multicore GPUs », dans *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, coll. « DAMP '11 », p. 3–14, Austin, Texas, USA, 2011. ACM. Cité page 78.

[226] I. Buck, T. Foley, D. Horn *et al.*, « Brook for GPUs : stream computing on graphics hardware », dans *ACM SIGGRAPH 2004 Papers*, coll. « SIGGRAPH '04 », p. 777–786, Los Angeles, California, 2004. ACM. Cité page 78.

[227] I. Buck, T. Foley, D. Horn *et al.*, « Brook for GPUs : stream computing on graphics hardware », *ACM Trans. Graph.*, vol. 23, August 2004, p. 777–786. Cité page 78.

[228] M. D. McCool et B. D'Amora, « M08 - programming using RapidMind on the Cell BE », dans *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, p. 222. ACM Press, 11 2006. Cité page 78.

[229] M. D. McCool, K. W. andBrent Henderson et H.-Y. Lin, « Poster reception - performance evaluation of GPUs using the rapidmind development platform », dans *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, p. 181. ACM Press, 11 2006. Cité page 78.

[230] M. McCool, S. Du Toit, T. Popa *et al.*, « Shader algebra », *ACM Trans. Graph.*, vol. 23, August 2004, p. 787–795. Cité page 78.

[231] A. Ghuloum, E. Sprangle, J. Fang *et al.*, « Ct : A flexible parallel programming model for tera-scale architectures ». Rapport technique, INTEL, October 2007. Cité page 78.

[232] R. Ronny Ronen, « Larrabee : a many-core intel architecture for visual computing », dans *Proceedings of the 6th ACM conference on Computing frontiers*, coll. « CF '09 », p. 225–225, New York, NY, USA, 2009. ACM. Cité page 78.

[233] C. J. Newburn, M. McCool, B. So *et al.*, « Intel array building blocks : A retargetable, dynamic compiler and embedded language », dans *The International Symposium on Code Generation and Optimization (CGO)*, Chamonix, France, 2011. Cité page 78.

[234] INTEL, « Parallel programming. multicore processors today, many-core co-processors ready ». Rapport technique, INTEL, Juin 2011. Cité page 79.

[235] J. A. Stratton, S. S. Stone et W.-M. W. Hwu, « MCUDA : An efficient implementation of CUDA kernels for multi-core CPUs », dans J. N. Amaral, éditeur, *Languages and Compilers for Parallel Computing*, p. 16–30. Springer-Verlag, Berlin, Heidelberg, 2008. Cité page 79.

[236] N. Farooqui, A. Kerr, G. Diamos *et al.*, « A framework for dynamically instrumenting GPU compute applications within GPU Ocelot », dans *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, coll. « GPGPU-4 », p. 9 :1–9 :9, Newport Beach, California, 2011. ACM. Cité page 79.

[237] G. F. Diamos, A. R. Kerr, S. Yalamanchili et N. Clark, « Ocelot : a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems », dans *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, coll. « PACT '10 », p. 353–364, Vienna, Austria, 2010. ACM. Cité page 79.

[238] R. Domínguez, D. Schaa et D. Kaeli, « Caracal : dynamic translation of runtime environments for GPUs », dans *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, coll. « GPGPU-4 », p. 5 :1–5 :7, New York, NY, USA, 2011. ACM. Cité page 79.

[239] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.1*, 2008. Cité page 79.

[240] S. Rul, H. Vandierendonck, J. D'Haene et K. De Bosschere, « An experimental study on performance portability of OpenCL kernels », dans *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010. Cité page 79.

[241] W. Kirschenmann, L. Plagne et S. Vialle, « Multi-target C++ implementation of parallel skeletons », dans *POOSC '09 : Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, Genova, Italy, 2009. ACM. Cité pages 83 et 87.

[242] W. Kirschenmann, L. Plagne et S. Vialle, « Multi-Target Vectorization With MTPS C++ Generic Library », dans *Proceedings of PARA 2010 conference : State of the Art in Scientific and Parallel Computing PARA 2010 : State of the Art in Scientific and Parallel Computing*, Reykjavik Islande, 06 2010. Cité pages 83 et 87.

[243] G. Hutton, « A tutorial on the universality and expressiveness of fold », *J. Funct. Program.*, vol. 9, July 1999, p. 355–372. Cité page 91.

[244] C. Strachey, « Fundamental concepts in programming languages », *Higher-Order and Symbolic Computation*, vol. 13, 2000, p. 11–49. Cité page 92.

[245] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. Cité page 93.

[246] J. Lautard, D. Schneider et A. Baudron, « Mixed Dual Methods for Neutronic Reactor Core Calculations in the CRONOS System », dans *Proc. Int. Conf. Mathematics and Computation, Reactor Physics and Environmental Analysis of Nuclear Systems, Madrid, Spain, Senda International, SA, Madrid*, p. 814–826, 1999. Cité page 93.

[247] J. Douglas et J. E. Gunn, « A general formulation of alternating direction methods », *Numerische Mathematik*, vol. 6, 1964, p. 428–453. 10.1007/BF01386093. Cité page 109.

[248] T. Boubekeur et C. Schlick, « Generic mesh refinement on gpu », dans *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, coll. « HWWS '05 », p. 99–104, New York, NY, USA, 2005. ACM. Cité page 109.

[249] M. Lefebvre, J.-M. L. Gouez et C. Basdevant, « Generic Refinement and Block Partitioning for Unstructured GPU Simulations », dans *Parallel CFD 2012*, Atlanta, USA, May 2012. Cité page 109.

[250] M. Frigo, C. E. Leiserson, H. Prokop et S. Ramachandran, « Cache-oblivious algorithms », dans *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, coll. « FOCS '99 », p. 285–, Washington, DC, USA, 1999. IEEE Computer Society. Cité page 123.

[251] P. Gottschling, D. S. Wise et M. D. Adams, « Representation-transparent matrix algorithms with scalable performance », dans *ICS '07 : Proceedings of the 21st annual international conference on Supercomputing*, p. 116–125, Seattle, Washington, 2007. ACM. Cité page 123.

[252] P. Gottschling, D. S. Wise et A. Joshi, « Generic support of algorithmic and structural recursion for scientific computing », *The International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, vol. 24, n° 6, December 2009, p. 479–503. Cité pages 123 et 126.

[253] INTEL, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2011. 248966-025. Cité page 136.

[254] J. J. Dongarra, « Performance of various computers using standard linear equations software », *SIGARCH Comput. Archit. News*, vol. 20, June 1992, p. 22–44. Updated version (June 20, 2011). Cité page 137.

[255] T. L. Veldhuizen, *Expression templates*, p. 475–487. New York, NY, USA, SIGS Publications, Inc., 1996. Cité page 150.

[256] T. L. Veldhuizen et M. E. Jernigan, « Will C++ be faster than fortran ? », dans *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, coll. « ISCOPE '97 », p. 49–56, London, UK, 1997. Springer-Verlag. Cité page 150.

Résumé

Cette thèse aborde les difficultés de mise au point de codes multicibles – c'est-à-dire de codes dont les performances sont portables entre différentes cibles matérielles. Nous avons identifié deux principales difficultés à surmonter : l'unification de l'expression du parallélisme d'une part et la nécessité d'adapter le format de stockage des données d'autre part.

Afin de mettre au point une version multicible de la bibliothèque d'algèbre linéaire Legolas++ mise au point à EDF R&D, nous avons conçu MTPS (MultiTarget Parallel Skeleton), une bibliothèque dédiée à la mise au point de codes multicible. MTPS permet d'obtenir une implémentation multicible pour les problèmes appliquant une même fonction aux différents éléments d'une collection. MTPS prend alors en charge l'adaptation du format de stockage des données en fonction de l'architecture ciblée.

L'intégration des concepts de MTPS dans Legolas++ a conduit à l'obtention d'un prototype multicible de Legolas++. Ce prototype a permis de mettre au point des solveurs dont les performances sont proches de l'optimal sur différentes architectures matérielles.

Mots-clés : Legolas++, programmation multicible, parallélisation, vectorisation, structures de données

Abstract

This thesis addresses the challenges of developing multitarget code – that is to say, codes whose performance is portable across different hardware targets. We identified two key challenges : the unification of the the parallelism expression and the need to adapt the format for storing data according to the target architecture.

In order to develop a multitarget version of Legolas++, a linear algebra library developed at EDF R&D, we designed MTPS (Multi-Tatget Parallel Skeleton), a library dedicated to the development of multitarget codes. MTPS allows for multitarget implementations of problems that apply the same function to all the elements of a collection. MTPS then handles the adaptation of the format for storing data according to the targeted architecture.

Integrating the concepts of MTPS in Legolas++ has led to the production of a multitarget prototype of Legolas++. This prototype has allowed the development of solvers whose performances near the harware limits on different hardware architectures.

Keywords: Legolas++, multitarget programming, parallelization, vectorization, data structures