République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de Batna 2

Faculté de mathématiques et d'informatique

Département d'Informatique

THÈSE

En vue de l'obtention du diplôme de

Doctorat en Sciences en Informatique

présentée et soutenue publiquement par

Toufik BAROUDI

le: 09 Juillet 2020

OPTIMISATION DES ACCÈS MÉMOIRE POUR LES ARCHITECTURES MULTI-COEURS

Jury

Pr. Azzedine BILAMI	Professeur	Université de Batna 2	Président
Pr. Rachid SEGHIR	Professeur	Université de Batna 2	Rapporteur
Dr. Moumen HMOUMA	M.C.A	Université de Batna 2	Examinateur
Pr. Allaoua CHAOUI	Professeur	Université de Constantine 2	Examinateur
Dr. El Kamel MERAH	M.C.A	Université de Khenchela	Examinateur
Pr. Hammadi BENNOUI	Professeur	Université de Biskra	Examinateur
Dr. Vincent LOECHNER	M.C	Université de Strasbourg France	Invité

THESE

Remerciements

Je tiens à prosterner remerciant Allah le tout puissant de m'avoir donné le courage et la patience pour terminer ce travail.

Mes vifs remerciements vont tout d'abord à Monsieur **Rachid SEGHIR**, Professeur à l'Université Mostefa Benboulaid, Batna 2, mon directeur de thèse pour m'avoir donné la possibilité d'effectuer cette thèse sous sa direction. Je tiens aussi à le remercier pour ses précieux conseils durant mes travaux de thèse et la préparation de la soutenance.

Je tiens également à remercier vivement Monsieur **Vincent LOECHNER**, Maître de conférences à l' Université de Strasbourg, France pour l'accueil et l'aide précieuse qu'il m'a prodigué tout au long de cette thèse.

Je veux également remercier tous les membres de jury : Monsieur **Azzedine BILAMI**, Professeur à l'Université de BATNA 2, pour m'avoir fait l'honneur de présider mon jury de thèse, ainsi que Monsieur **Moumen HMOUMA**, Maître de conférences à l'Université de Batna 2, Monsieur **Allaoua CHAOUI**, Professeur à l'Univerité de Constantine 2, Monsieur **El Kamel MERAH**, Maître de conférences à l'Université de Khenchela, Monsieur **Hammadi BENNOUI**, Professeur à l'Université de Biskra, pour m'avoir fait l'honneur d'accepter de juger ce travail.

D'autre part, je tiens à remercier chaleureusement le laboratoire ICube de l'université de Strasbourg et tout particulièrement les membres de l'équipe ICPS pour m'avoir acceuilli dans le cadre du Programme National Exceptionnel (P.N.E) durant l'année universitaire 2018-2019.

Je profite également de cette opportunité pour remercier mes très chers parents qui m'ont toujours soutenu , mes frères et mes sœurs pour leur soutien sans faille, leurs encouragements et sans lesquels rien n'aurait été possible.

Je remercie plus particulièrement ma femme pour sa présence et son soutien indéfectible même dans les moments difficiles au cours de mon parcours, sans oublier les fruits de ma vie mes belles filles Maissa et Israa.

Je voudrais dire aussi un très grand merci à mes collègues de l'institut d'hygiène et sécurité avec qui j'ai partagé bien plus qu'un lieu de travail.

Résumé

Dans l'objectif d'augmenter les performances, l'architecture des processeurs a évolué vers des plate-formes multi-coeurs et many-coeurs composées de multiples unités de traitements. Cependant, la difficulté de la programmation de ces architectures entrave souvent de tirer pleinement avantage du potentiel de ses unités de traitement. Ce qui a motivé la communauté de recherche en compilation à développer des solutions alternatives dont l'objectif est de décharger le programmeur des détails des architectures cibles, tout en générant des programmes aussi efficaces que possible. C'est le domaine de recherche en parallélisation et optimisation automatique de codes. Dans la quasi-totalité des programmes, les nids de boucles représentent l'essentiel du temps de calcul. Pour optimiser l'exécution de ces partis de codes, les compilateurs leur appliquent des transformations afin d'améliorer la localité spatiale et temporelle des accès mémoire. La majorité de ces transformations sont basées sur le modèle polyédrique qui est un formalisme permettant de représenter les itérations et les références à des tableaux par des points à coordonnées entières de polyèdres bornés. L'optimisation des programmes d'algèbre linéaire est l'un des problèmes qui a attiré l'attention de la communauté de recherche en optimisation de codes depuis de nombreuses années. En particulier, les opérations sur les matrices creuses font partie des codes de calcul clés dans de nombreuses applications scientifiques et d'ingénierie. Par conséquent, plusieurs formats de stockage alternatifs ont été proposés afin de stocker ces matrices et de calculer uniquement les éléments non nuls. Dans cette thèse, nous introduisons, dans un premier temps, une nouvelle approche pour optimiser les opérations matricielles sur des matrices creuses particulières en utilisant une structure de données dense appelée le format 2d-Packed. L'idée de base est que les opérations matricielles utilisant cette nouvelle structure de données peuvent être optimisées et parallélisées automatiquement à l'aide des compilateurs source à source basés sur le modèle polyédrique, comme Pluto. Ce travail a permis d'améliorer les performances d'un bon nombre de benchmarks de l'algèbre linéaire. Dans un deuxième temps, nous avons présenté une étude sur l'effet des types d'allocation, statique et dynamique, sur les performances de benchmarks utilisant la structure de données proposée et certains noyaux d'algèbre linéaire de la suite PolyBench.

Mots clés :
Architecture multi-cœurs, Optimisation et parallélisation de codes, Matrices creuses, Modèle polyédrique, Format 2d-Packed.

Abstract

With the aim of increasing performance, the architecture of processors has evolved into multicore and manycore platforms composed of multiple processing units. However, the difficulty of programming these architectures often hinders taking full advantage of the potential of its processing units. This has motivated the compilation research community to develop alternative solutions, the goal of which is to free the programmer from the details of target architectures, while generating programs as efficient as possible. This is the area of research in parallelization and automatic code optimization. In almost all programs, loop nests account for most of the computing time. To optimize the execution of these code parties, the compilers apply transformations to them in order to improve the spatial and temporal locality of the memory accesses. The majority of these transformations are based on the polyhedral model which is a formalism allowing to represent iterations and references to arrays by points with integer coordinates of bounded polyhedra. Optimizing linear algebra programs has been one of the issues that has caught the attention of the code optimization research community for many years. In particular, operations on sparse matrices are part of the key calculation codes in many scientific and engineering applications. Consequently, several alternative storage formats have been proposed in order to store these matrices and to calculate only the non-zero elements. In this thesis, we first introduce a new approach to optimize matrix operations on particular sparse matrices using a dense data structure called the 2d-Packed format. The basic idea is that matrix operations using this new data structure can be automatically optimized and parallelized using source-to-source compilers based on the polyhedral model, such as Pluto. This work has improved the performance of a number of benchmarks in linear algebra. Second, we presented a study on the effect of allocation types, static and dynamic, on benchmark performance using the proposed data structure and certain linear algebra kernels of the PolyBench suite.

Keywords :
Multicore architecture, Optimization and parallelization of codes, Sparse matrices, Polyhedral model, 2d-Packed format.

الملخص

بهدف زيادة الأداء ، تطورت بنية المعالجات إلى منصات متعددة النواة والعديد من النواة تتكون من وحدات معالجة متعددة. ومع ذلك ، فإن صعوبة برمجة هذه البنى في كثير من الأحيان تجعل من الصعب الاستفادة الكاملة من إمكانات وحدات المعالجة الخاصة بها. وقد حفز ذلك مجتمع البحوث التجميعية على تطوير حلول بديلة هدفها تحرير المبرمج من تفاصيل البنى المستهدفة مع توليد برامج بأقصى قدر ممكن من الكفاءة. هذا هو مجال البحث في التوازي والتحسين التلقائي للكود. في كل البرامج تقريباً مثل أعشاش الحلقات معظم وقت الحوسبة. لتحسين تنفيذ أطراف الكود هذه ، يقوم المترجمون بتطبيق التحويلات عليها من أجل تحسين الأداءالمكاني والزمني للذاكرة التي تصل إليها. تعتمد معظم هذه التحولات على نموذج متعدد السطوح وهو شكل رسمي يسمح بتمثيل التكرارات والإشارات إلى المصفوفات بواسطة نقاط مع إحداثيات عدد صحيح من متعدد الوجوه. يعد تحسين برامج الجبر الخطي أحد المشكلات التي لفتت انتباه مجتمع أبحاث تحسين الكود لسنوات عديدة. على وجه الخصوص ، تعتبر العمليات على المصفوفات المجوفة جزءًا من أكواد الحساب الرئيسية في العديد من التطبيقات العلمية والهندسية. وبالتالي ، تم اقتراح العديد من تنسيقات التخزين البديلة لتخزين هذه المصفوفات ولحساب العناصر غير الصفرية فقط. في هذه الرسالة ، قدمنا أولاً مقاربة جديدة لتحسين عمليات المصفوفة على مصفوفات مجوفة معينة باستخدام بنية بيانات كثيفة تسمى التنسيق ثنائي الأبعاد. الفكرة الأساسية هي أن عمليات المصفوفة التي تستخدم بنية البيانات الجديدة هذه يمكن تحسينها وموازنتها تلقائيًا باستخدام المجمعين من مصدر إلى مصدر بناءً على نموذج متعدد السطوح ، مثل بلوتو. حسن هذا العمل أداء عدد كبير من معايير الجبر الخطي. ثانياً ، قدمنا دراسة حول تأثير أنواع التخصيص ، ثابتة وديناميكية ، على الأداء القياسي باستخدام بنية البيانات المقترحة وبعض برامج الجبر الخطية في مجموعة بوليبانش.

الكلمات المفتاحية :
بنية متعددة النواة ، تحسين وموازنة الرموز ، مصفوفات مجوفة ، نموذج متعدد السطوح ، تنسيق ثنائي الأبعاد.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Au cours de la dernière décennie, le monde de l'informatique a connu une orientation de plus en plus prononcée vers la multiprogrammation. Depuis la fin de la montée en fréquence des processeurs programmables, les concepteurs de circuits intégrés ont cherché à améliorer leurs performances en augmentant le nombre des unités de calcul (appelées cœurs) intégrés dans une même puce. Cependant, les architectures multi-cœurs ne supportent pas l'accélération du rendu graphique, celui-ci reste le domaine réservé des processeurs graphiques. La difficulté de la programmation des architectures multi-cœurs a souvent entravé l'exploitation efficace du potentiel de ses unités de traitement. En effet, plusieurs solutions ont été proposées pour résoudre ce problème, l'une des approches les plus prometteuses est l'optimisation et la parallélisation automatique [62]. Cette approche ne nécessite aucun effort de la part du programmeur dans le processus de parallélisation d'un programme. Alternativement, les compilateurs devraient être en mesure de paralléliser et d'ajuster automatiquement le code séquentiel écrit dans les langages de programmation standards afin d'exploiter une architecture multi-cœurs donnée. Le plus souvent, les différents cœurs des processeurs multi-cœurs se partagent le même espace mémoire (mémoire partagée). Cela veut dire que les cœurs du processeur accèdent à la mémoire partagée d'une manière concurrente, ce qui influe considérablement sur les performances globales du système. Ceci est dû au fait que les unités de calcul se trouvent souvent en attente de données provenant de la mémoire, augmentant ainsi leur temps d'inactivité. Les architectures multi-cœurs sont ainsi utilisées dans des domaines tels que les applications mobiles, le transcodage réseau, le multimédia, la vision, etc.

La quasi-totalité des programmes intègrent de plus en plus des nids de boucles impliquant une augmentation des temps d'exécution des implémentations. De ce fait, plusieurs travaux de recherche ont été proposés pour exploiter les potentialités offertes par les architectures dans le but d'augmenter le niveau du parallélisme des nids de boucles, afin de réduire leurs temps d'exécution. Ces travaux décrivent des techniques de ré-ordonnancement des traitements des structures itératives, permettant de sélectionner les traitements indépendants pour les exécuter dans le même espace temporel. Ces techniques de parallélisme procèdent à la modification de la structure algorithmique du nid de boucles tout en conservant le comportement global de l'application. La forme de représentation des nids de boucles dans une structure matricielle est appelée le modèle polyédrique [16]. Ce modèle assure la formulation de l'aspect itérative des boucles ainsi que les dépendances des données. Il permet de représenter les nids de boucles sous la forme de polyèdres. A l'intérieur de ces polyèdres les instances des instructions sont représentées par des points entiers. Dans le modèle polyédrique, des transformations affines sur les codes (particulièrement sur les nids de boucles) peuvent être appliquées automatiquement afin d'améliorer les performances par la parallélisation ainsi que l'amélioration de la localité des données.

De nombreuses applications en calcul scientifique, telles que la simulation de la mécanique des structures, l'étude de la dynamique des fluides et le traitement d'image/signal sont modélisés mathématiquement à l'aide d'équations aux dérivées partielles. La discrétisation de ces équations aboutit à résoudre un système linéaire représenté par l'équation algébrique Ax=b. Ces systèmes linéaires sont en général exprimés sous la forme de matrice avec un caractère symétrique, creux et de très grande taille. Le traitement de ce type de matrice en parallèle nécessite un bon choix du format de stockage facilitant les opérations demandées. L'optimisation de ce type de calcul est primordiale, et la grande taille de la structure de données implique le recours au parallélisme et à des méthodes appropriées. Dans la discipline de calcul matricielle, de nombreux problèmes font intervenir des matrices de grandes tailles qui contiennent beaucoup de coefficients nuls, on parle de matrices creuses (sparse matrices) [65, 35, 60]. Afin d'économiser l'espace mémoire et éviter les calculs inutiles, il est souhaitable de ne stocker que les éléments non nuls d'une matrice. Il est alors indispensable de stocker également d'autres informations pour retrouver les lignes et les colonnes de la matrice.

Dans cette thèse nous avons cherché à améliorer les performances des systèmes multicœurs à travers l'optimisation des accès à la mémoire en utilisant des opérations sur des matrices creuses particulières. L'idée est de faire adapter les références d'un programme à la structure de la hiérarchie mémoire de l'architecture multi-cœurs. Ceci pourrait se faire en transformant des parties du programme original (sans toucher à sa sémantique) en vue d'aboutir à un programme cible dans lequel la latence mémoire est réduite le plus possible. Il s'agit plus précisément de proposer des mécanismes pour améliorer la localité spatiale et temporelle d'un programme.

Les travaux présentés dans la première partie de contribution de cette thèse se concentrent sur la proposition d'une nouvelle approche pour optimiser les opérations matricielles sur des matrices triangulaires et en bandes en utilisant une structure de données dense à deux dimensions appelée 2d-Packed Format pour le stockage de matrices creuses [15]. L'idée de base est que les opérations matricielles utilisant ces structures de données peuvent être automatiquement optimisées et parallélisées au moyen du modèle polyédrique. D'une part, les matrices triangulaires et en bandes sont compactées, ce qui permet de l'économie importante de mémoire, et d'autre part, le code sous-jacent peut être optimisé et parallélisé en utilisant les outils d'optimisation polyédriques existants pour obtenir les meilleures performances. Nous démontrons par une étude expérimentale l'efficacité de notre approche en parallélisant et en optimisant plusieurs codes de calcul matriciel en utilisant le compilateur polyédrique Pluto [34] , et en comparant leurs performances aux versions séquentielles et parallèles non-denses, la version LPF [43], et avec des routines de la bibliothèque MKL.

Dans la fin de cette première partie, nous avons appliqué notre approche proposée sur un code très intéressant dans le calcul haute performance (multiplication matrice-vecteur) sur des matrices creuses générées par la méthode de dissection emboîtée en utilisant une nouvelle structure de données 2d-Packed-Nested Dissection (2d-P-ND).

Nous nous sommes par la suite intéressés, dans la deuxième partie de la contribution, de mesurer l'importance du type d'allocation dans la structure de données 2d-Packed proposée. A cet effet, nous avons comparé les performances de différents noyaux d'algèbre linéaire en utilisant les deux modes d'allocation, statique et dynamique, de tableaux. Nous présentons notre étude détaillée sur les raisons possibles de la différence de performances de nos noyaux et de certains noyaux d'algèbre linéaire de la suite PolyBench, sur deux architectures différentes. Nous avons constaté que, dans ce contexte, le mode d'allocation des matrices (dynamique ou statique) a une influence notable sur les temps d'exécution mesurés. L'étude que nous avons menée nous a permet d'identifier des compteurs de performances différents qui sont responsables de ces différences dans le temps d'exécution entre l'allocation statique et dynamique [14]. Les contributions de ce travail de thèse sont structurées en six chapitres suivis d'une conclusion générale et perspectives.

Le premier chapitre est dédié à la présentation des architectures parallèles. Nous présentons tout d'abord les différents types d'architectures parallèles selon la taxonomie de Flynn. Ensuite, nous introduisons la hiérarchie mémoire et les différents niveaux de mémoire cache. Enfin nous détaillons les architectures multi-cœurs et les architectures many-cœurs.

Le deuxième chapitre consiste en une présentation du modèle polyédrique. En premier lieu, nous introduisons les différentes notions liées à ce modèle, notamment les nids de boucles, les fonctions affines et les fonctions de transformations et d'optimisation. Dans un deuxième temps, nous montrons les dépendances de données et leurs différents types.Le troisième volet est consacré à la présentation des outils polyédriques existants. Enfin, nous introduisons les différentes techniques d'optimisation.

Dans le troisième chapitre, nous introduisons l'optimisation et la parallélisation automatique. Nous traitons initialement la nécessité de parallélisation et l'optimisation automatique. Ensuite, nous détaillons les langages de parallélisation automatique. Nous terminons ce chapitre par la présentation de différents outils de la parallélisation automatique, où nous décrivons avec plus de détails l'outil Pluto que nous utilisons dans nos travaux.

Le quatrième chapitre présente les matrices creuses et leurs formats de stockage existants. Dans la première partie de ce chapitre, nous présentons les notions et les notations sur les matrices et leurs types. Nous détaillons ensuite les matrices creuses et leurs différents formats de stockage existants.

Le cinquième chapitre est dédié à la présentation de notre approche proposée. Dans ce chapitre nous introduisons au début les travaux de recherche liés à notre sujet. Deuxièmement, nous présentons notre approche 2d-Packed Format pour les matrices triangulaires et les matrices en bande. Troisièmement, nous faisons une description détaillée des résultats expérimentaux en comparant les performances de notre approche aux versions séquentielles et parallèles non-denses, la version LPF, et avec des routines de la bibliothèque MKL. Nous terminons ce chapitre par une application de notre approche sur des matrices générées par la méthode de dissection emboitée.

Dans le dernier chapitre, nous nous intéressons à l'étude de l'effet du type d'allocation (statique ou dynamique) dans la structure de données 2d-Packed proposée. Nous faisons une étude comparative en parallélisant et en optimisant plusieurs opérations de calcul matriciel, à l'aide du compilateur source à source Pluto. Enfin, nous terminons cette thèse par une conclusion suivie d'un aperçu de nos travaux en cours et futurs.

Chapitre 1

Les architectures parallèles

Sommaire

1.1 Introduction

De nos jours, les architectures parallèles se sont progressivement généralisées au sein de tous les matériels informatiques. À l'origine, ces architectures étaient réservées à des utilisations très spécifiques de calcul haute performance, tandis que la majorité des développements avaient lieu sur les architectures séquentielles. Ces dernières ont offert durant de nombreuses années une amélioration des performances sans qu'il soit nécessaire de remettre en question leur mode de programmation grâce à l'augmentation rapide des fréquences de fonctionnement. Certains formes de parallélisme ont été intégrées dans les processeurs pour en améliorer l'efficacité (pipelining des instructions, calcul simultané dans des unités de calcul séparées, . . .) mais celles-ci restaient implicites et cachées aux programmeurs. Cependant, l'augmentation de la fréquence des processeurs a atteint il y a quelques années un pic difficilement franchissable à cause de limitations physiques. Les architectures parallèles se sont donc immiscées dans les utilisations courantes afin de pouvoir continuer à fournir un accroissement des performances. Dans ce chapitre nous avons présenté les différents types d'architectures parallèles selon la taxonomie de Flynn, Nous avons décrit aussi les types de la quatrième catégorie de machine parallèle MIMD selon le type de mémoire : partagé, distribué et hybride. Nous avons également détaillé les différents catégories de l'architecture MIMD selon le type de mémoire. L'étude des ces architectures nous a conduit de montrer l'importance de la hiérarchie mémoire et la mémoire cache ainsi que leurs fonctionnalités. Dans la suite de ce chapitre, nous avons présenté les différentes fonctionnalités de deux exemples d'architectures multi-cœurs Intel et AMD. Enfin, Deux exemples de processeurs many-cœurs ont été discutés dans la dernière partie de chapitre (Intel Xeon Phi et les GPU).

1.2 Introduction aux architectures parallèles

Le parallélisme informatique consiste essentiellement à utiliser plus de processeurs (nœuds informatiques) qui coopèrent via un réseau d'interconnexion pour la résolution d'un problème informatique. Tout ordinateur, séquentiel ou parallèle, exécute des instructions sur les données. De nombreuses applications scientifiques présentent beaucoup de concurrence, car le monde réel lui-même est extrêmement parallèle. Bien que les systèmes multicœurs offrent un débit de calcul supérieur à celui des systèmes monocœurs, il est parfois difficile de tirer parti des performances. Le gain de performance que l'on peut obtenir en améliorant certaines parties d'une application est limité par la loi d'Amdahl. Cette loi indique que la vitesse d'accélération parallèle est intrinsèquement limitée par le code séquentiel dans un programme conformément à l'équation 1.1. [3, 66, 95, 26].

$$\mathrm{S}peed\mathrm{U}p = \frac{1}{1-p+\frac{p}{\mathrm{N}}}. \tag{1.1}$$

Où p représente la fraction de code parallèle et N le nombre de cœurs de calcul. La vitesse augmente avec le nombre de cœurs comme indiqué à la figure 1.1. Si la partie séquentielle d'une application est volumineuse (c'est-à-dire 50%), l'accélération parallèle se sature avec un petit nombre de cœurs (c.-à-d. 16 cœurs)

Le nombre d'applications qui offrent des performances adaptées au nombre de processeurs est limité. Cela se produit généralement lorsqu'il n'y a pas de dépendance de données entre différents flux. Cependant, la plupart des applications nécessitent une communication entre les cœurs de calcul. Dans ce scénario, le trafic de données et la synchronisation entre les nœuds peuvent être coûteux.

La taxonomie de Flynn est une méthode largement utilisée pour classifier les architectures informatiques (prendre en compte le flux d'instructions et le flux de données) : [55, 57].

1. SISD (Single Instruction, Single Data) ;
2. MISD (Multiple Instruction, Single Data) ;

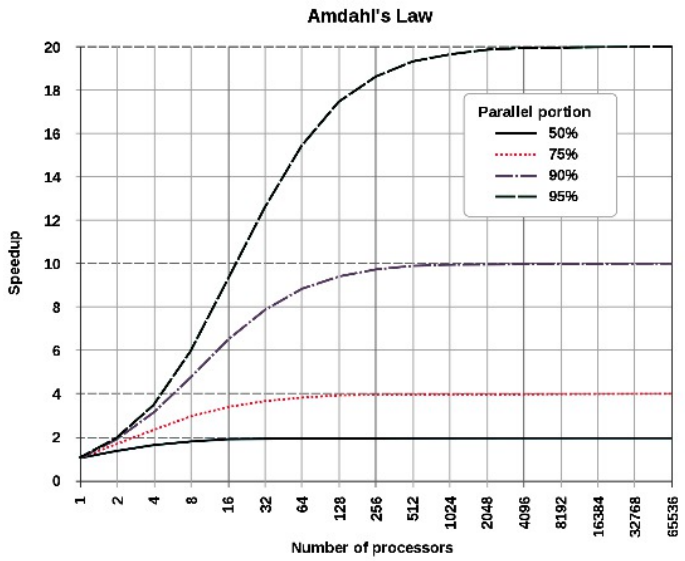


FIGURE 1.1 – Loi d'Amdhal [116].

3. SIMD (Single Instruction, Multiple Data) ;
4. MIMD (Multiple Instruction, Multiple Data).

Nous énumérons les principaux types d'architectures parallèles à la figure 1.2

1.2.1 L'architecture SISD

En informatique, SISD est un terme faisant référence à une architecture informatique dans laquelle le processeur (un mono-processeur) exécute un seul flot d'instruction avec des données stockées dans une seule mémoire. L'architecture SISD est illustrée dans la figure 1.3. Dans ce système, les classifications sont basées sur le nombre d'instructions simultanées et de flux de données présents dans l'architecture de l'ordinateur. L'architecture SISD peut avoir des caractéristiques de traitement simultanées. La plupart des ordinateurs SISD modernes sont basés sur la récupération d'instructions. Le mode d'exécution des architectures SISD est montré dans la figure 1.4 [57].

1.2.2 L'architecture SIMD

Une machine parallèle de cette classe est composée de N processeurs identiques. La figure 1.5 montre l'architecture SIMD. Chaque processeur dispose d'une mémoire locale dans laquelle il conserve les données sur lesquelles il travaillera. Sur les machines SIMD, tous les processeurs exécutent simultanément la même instruction, émise par le contrôleur, sur leurs données locales, le mode d'exécution des instructions sur les machines SIMD est illustré dans la figure 1.6. Les processeurs peuvent communiquer les uns avec les autres afin d'effectuer des décalages et d'autres opérations.

Cette approche peut réduire à la fois la complexité matérielle et logicielle, mais ne convient que pour des problèmes spécialisés caractérisés par un degré élevé de régularité, tels que le traite-

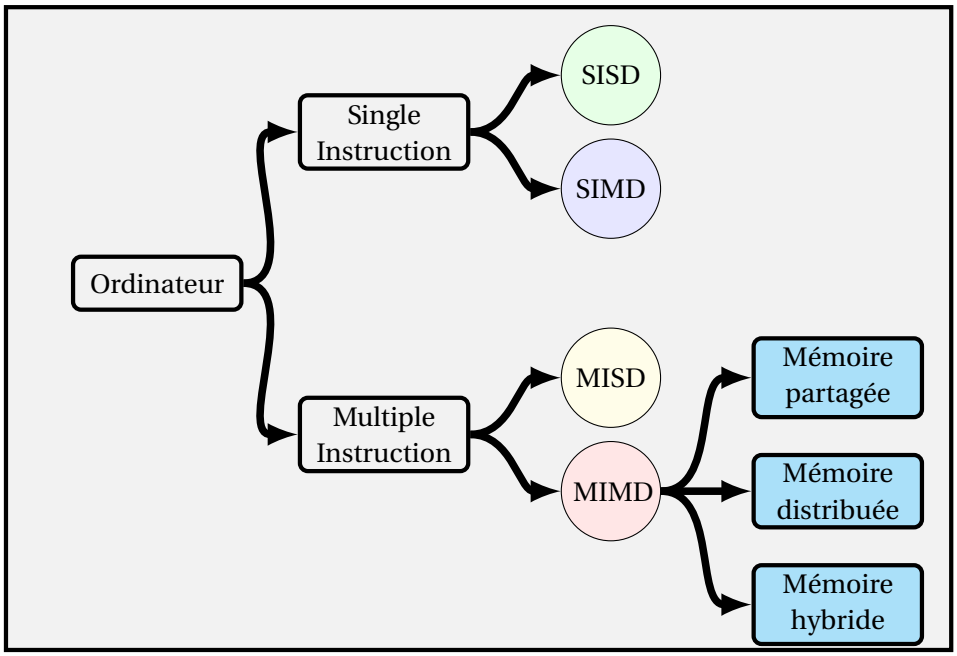


FIGURE 1.2 – La taxonomie de Flynn.

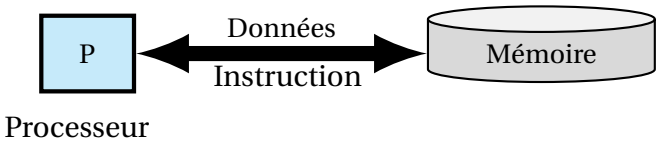


FIGURE 1.3 – L'architecture SISD.

ment d'images et certaines simulations numériques. Si les charges sur les processeurs ne sont pas équilibrées, les performances sont médiocres (car l'exécution est synchronisée à chaque étape, tout attend le processeur le plus lent). Par exemple, lorsque vous utilisez des graphiques, vous pouvez charger simultanément N pixels dans la mémoire principale au lieu de charger un seul pixel. Le pixel suivant sera ensuite chargé dans la mémoire principale. De même, SIMD n'inclut que les points de données d'un jeu d'instructions sur l'opération pouvant être exécutées simultanément. Parmi les machine utilisant l'architecture SIMD on a : IBM 9000, connection machine CM-2 et Cray X-MP, etc [41].

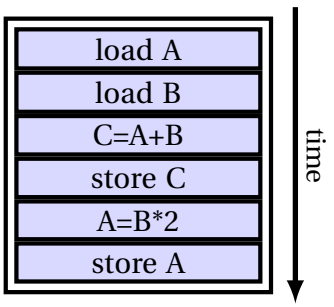


FIGURE 1.4 – Mode d'exécution des instructions avec L'architecture SISD.

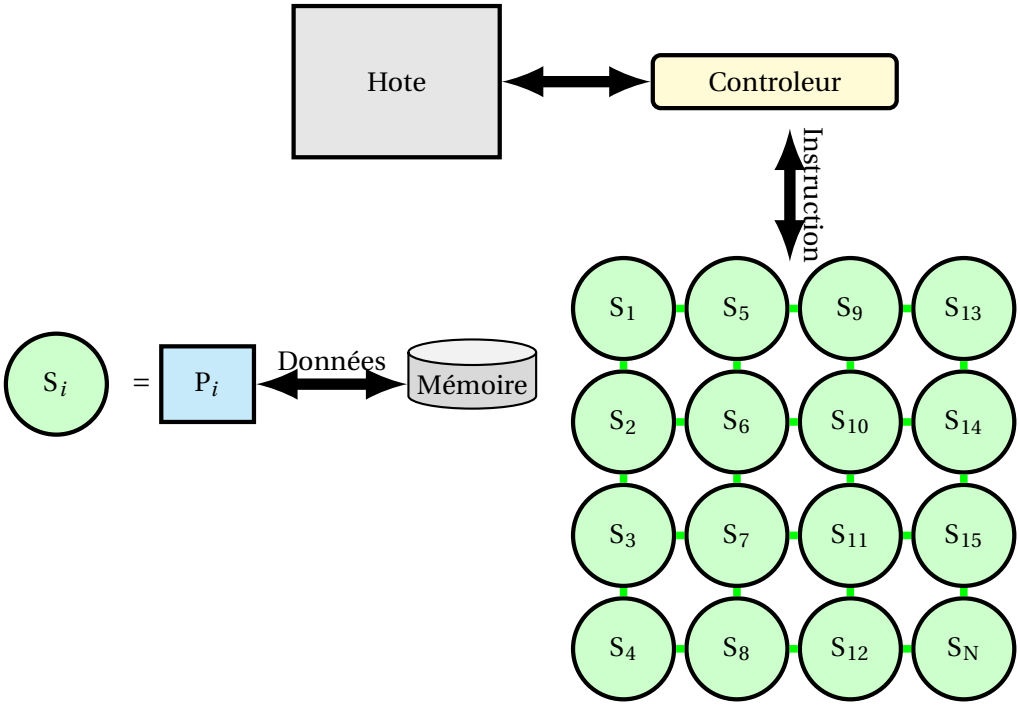


FIGURE 1.5 – L'architecture SIMD.

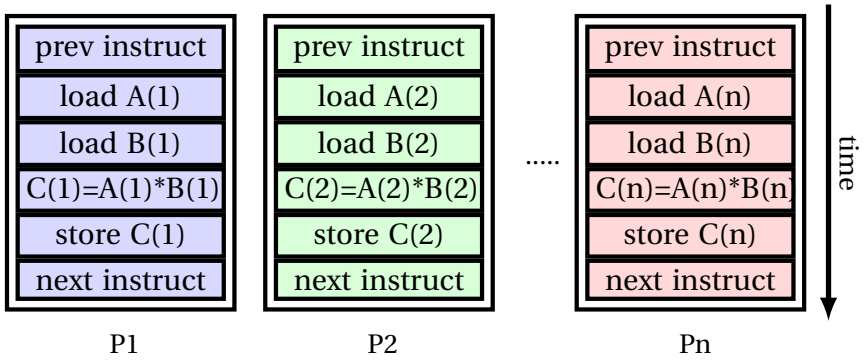


FIGURE 1.6 – Mode d'exécution des instructions avec L'architecture SIMD.

1.2.3 L'architecture MISD

MISD est un type d'architecture de calcul parallèle où de nombreuses unités fonctionnelles effectuent différentes opérations sur les mêmes données. Cette architecture est présenté sur la figure 1.7. Les architectures de pipeline est l'architecture principale de ce type. Le mode d'exécution des instructions avec l'architecture MISD est montré sur la figure 1.8. Il n'existe pas beaucoup d'instances de cette architecture. Cependant, un exemple remarquable de MISD en informatique est l'ordinateur de contrôle de vol de la navette spatiale.

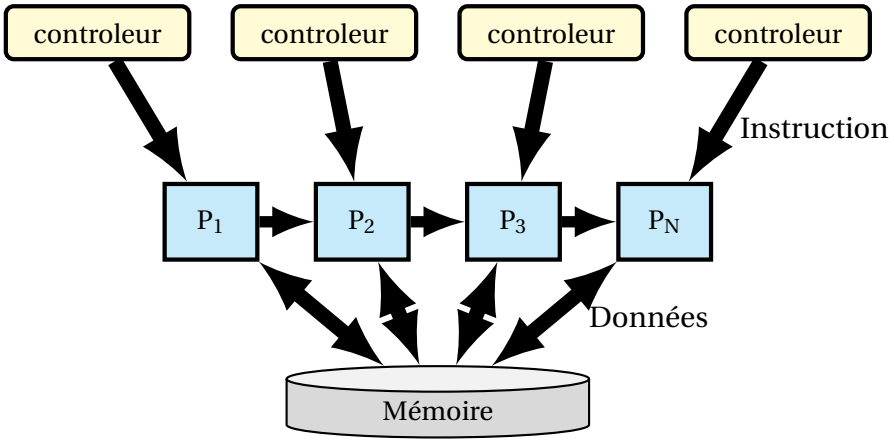


FIGURE 1.7 – L'architecture MISD.

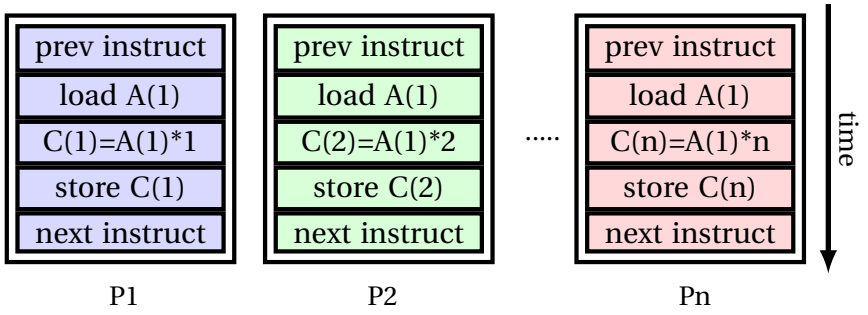


FIGURE 1.8 – Mode d'exécution des instructions avec L'architecture MISD.

1.2.4 L'architecture MIMD

Cette classe d'ordinateurs est le paradigme le plus général et le plus puissant de Flynn. Les machines utilisant MIMD ont un certain nombre de processeurs qui fonctionnent de manière asynchrone et indépendamment. A tout moment, différents processeurs peuvent exécuter différentes instructions sur différentes données. La figure 1.9 montre le mode d'exécution avec l'architecture MIMD. Dans cette classe d'architectures, la méthode courante pour les subdiviser est la relation entre les processeurs et la mémoire. Ce type de subdivision conduit à trois types principaux d'architectures MIMD : mémoire partagée, mémoire distribuée et mémoire hybride [41].

1.2.4.1 Les machines MIMD à mémoire partagée

Dans l'architecture à mémoire partagée, illustrée à la figure 1.10, tous les processeurs possèdent l'accès à une mémoire commune, généralement via un bus ou une hiérarchie de bus. Les processeurs communiquent les uns avec les autres par la méthode : un processeur écrit les données dans un emplacement en mémoire et un autre processeur lit les données. Avec ce type de communication, le temps d'accès aux données est le même, car toutes les communications

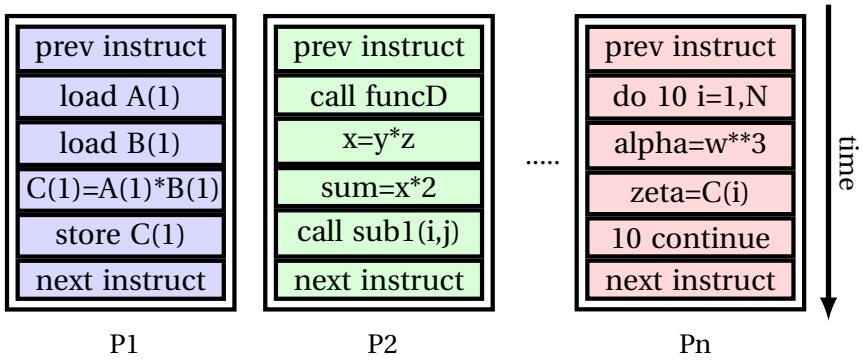


FIGURE 1.9 – Mode d'exécution des instructions avec L'architecture MIMD.

passent par le bus. L'avantage de ce type d'architecture est qu'il est facile à programmer car il n'existe aucune communication explicite entre les processeurs, les communications étant gérées via la mémoire globale. L'accès à cette mémoire peut être contrôlé à l'aide de techniques développées à partir d'ordinateurs multitâches, par exemple des sémaphores. Cependant, l'architecture à mémoire partagée ne s'adapte pas bien. Le problème principal survient lorsque plusieurs processeurs tentent d'accéder simultanément à la mémoire globale. Une méthode permettant d'éviter ce conflit d'accès à la mémoire consiste à diviser la mémoire en plusieurs modules de mémoire, chacun étant connecté aux processeurs via un réseau de commutation hautes performances. Cependant, cette approche tend simplement à déplacer le problème vers le réseau de communication. Parmi Les exemples de cette architecture on cite : SGI PowerChallenge, Sequent Balance et Symmetry [99].

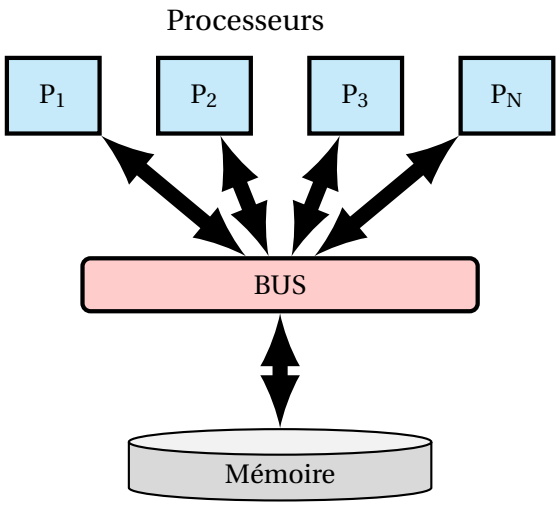


FIGURE 1.10 – Exemple d'une architecture à mémoire partagée.

1.2.4.2 Les machines MIMD à mémoire distribuée

L'architecture à mémoire distribuée présentée sur la figure 1.11 contourne les inconvénients de l'architecture de la mémoire partagée en attribuant à chaque processeur sa propre mémoire. Un processeur ne peut accéder qu'à la mémoire qui lui est directement rattachée. Si un processeur a besoin de données contenues dans la mémoire d'un processeur distant, il doit alors envoyer un message au processeur distant lui demandant de lui envoyer les données [128].

L'accès à la mémoire locale peut être nettement plus rapide que l'accès aux données d'un processeur distant. En addition, plus la distance physique du processeur distant est grande, plus l'ac-

cès aux données distantes peut être long. Ce temps d'accès non uniforme peut être affecté par la façon dont les processeurs sont connectés. Bien que, connecter un processeur à un autre soit une possibilité pour un petit nombre de processeurs, cela devient rapidement impraticable à mesure que le nombre de connexions augmente. Une solution au problème de la connexion des processeurs consiste à connecter un processeur à un petit sous-ensemble de ses voisins. Chacun des voisins du sous-ensemble de communication serait connecté à un sous-ensemble différent de processeurs, permettant ainsi à des messages d'être envoyés d'un processeur à un autre via un certain nombre de processeurs intermédiaires. Cela peut être fait de différentes manières. Une option serait d'utiliser des puces de commutation qui permettent à l'utilisateur d'adapter la topologie de la machine à leurs propres besoins. Une autre possibilité courante consiste à connecter les processeurs dans un arrangement hypercube. Cela présente l'avantage de ne pas augmenter radicalement le nombre de connexions à mesure que le nombre de processeurs augmente, tout en offrant un certain nombre de chemins d'acheminement de messages différents. En outre, il existe de nombreux algorithmes et logiciels parallèles pour les hypercubes. Les machines à mémoire distribuée ont été construites en utilisant toutes les méthodes décrites. Plutôt que de connecter directement les processeurs ensemble, la pratique actuelle consiste à les connecter à un réseau de puces de routage. Les mêmes problèmes de topologie s'appliquent dans ce cas, mais les processeurs ne jouent plus aucun rôle dans le transfert de message. Comme il peut y avoir un nombre différent de processeurs et de puces de routage, cela permet une plus grande liberté lors de la construction du réseau. Un exemple de machines ayant une architecture à mémoire distribuée : the Meiko Computing Surfaces [68].

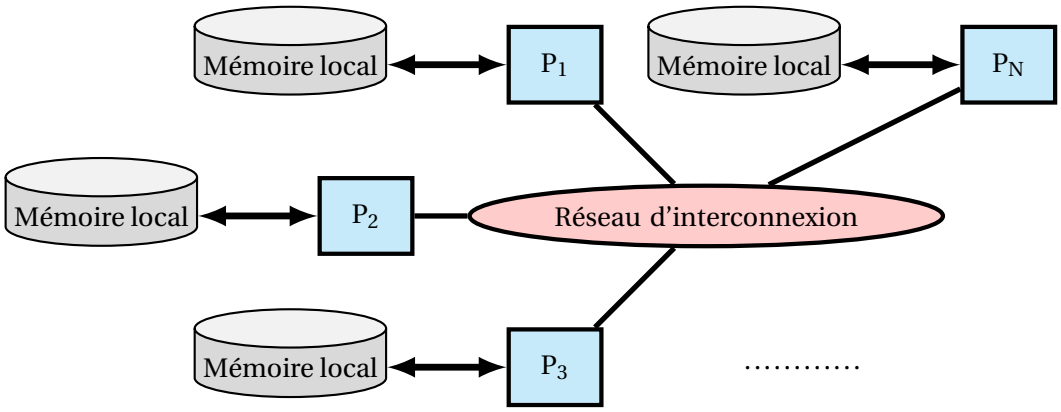


FIGURE 1.11 – Exemple d'une architecture à mémoire distribuée.

1.2.4.3 Les machines MIMD à mémoire hybride

Pour combiner l'avantage de la rapidité de communication inter-processeurs dans une machine à mémoire partagée, et celui du nombre élevé de processeurs de l'architecture à mémoire distribuée, les constructeurs ont pensé à une nouvelle architecture hybride à mémoire partagée-distribuée montré dans la figure 1.12. Les machines de ce type sont constituées de nœuds, appelés grappes (clusters), dont chacun est composé de plusieurs processeurs partageant une mémoire commune, l'ensemble de ces nœuds étant relié par un réseau d'interconnexion. La composition d'un noeud est présenté sur la figure 1.13. Ici, l'architecture à mémoire partagée en intra-nœud et l'architecture à mémoire distribuée en inter-nœuds sont assemblées.

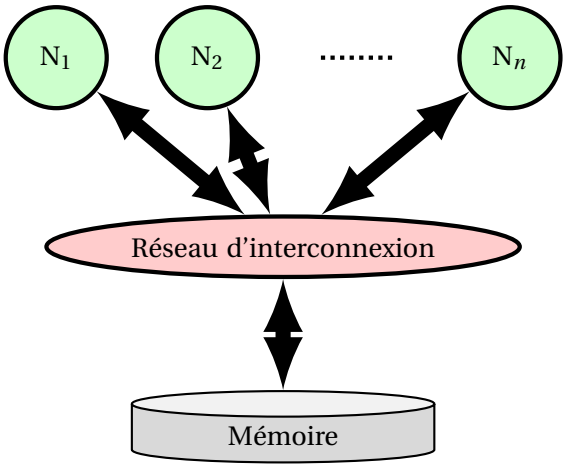


FIGURE 1.12 – Exemple d'une architecture à mémoire hybride.

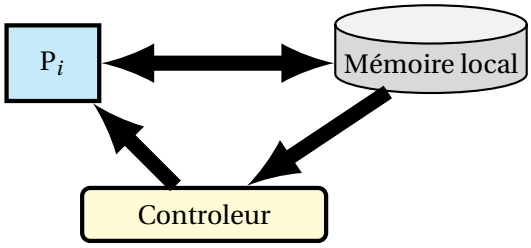


FIGURE 1.13 – Noeud d'une architecture à mémoire hybride.

1.3 La hiérarchie mémoire et la mémoire cache

Dans cette partie, nous expliquons pourquoi une mémoire à un seul niveau n'est pas pratique dans les machines modernes et donc le besoin d'une hiérarchie de mémoire. Les accès mémoire sont très courants dans les programmes. Le temps nécessaire pour charger les données de la mémoire dans le processeur s'appelle la latence de l'opération en mémoire. Il est généralement mesuré en cycles d'horloge du processeur ou en nanoseconde. Les processeurs modernes sont rapides en ce sens qu'ils peuvent fonctionner normalement à des vitesses d'horloge de plusieurs GHz et peuvent exécuter plus d'une instruction par cycle d'horloge. Ainsi, une mémoire doit être rapide afin de correspondre à la vitesse du processeur. De plus, tous les microprocesseurs attendent une RAM. En d'autres termes, toute donnée particulière est nécessaire à tout moment et il n'y a aucune contrainte quant à l'ordre de placement des instructions ou des données dans la mémoire. Une application logicielle moderne s'attend à des centaines de mégaoctets ou gigaoctets de stockage pour les données. Par exemple, un système de contrôle de caméra a besoin de mémoire pour stocker les images enregistrées. Par conséquent, une mémoire doit être volumineuse pour répondre aux besoins de stockage. En outre, il doit également prendre en charge le stockage permanent. Toutes les exigences ci-dessus peuvent être satisfaites avec une seule technologie de mémoire, mais le coût est énorme et considéré comme non pratique. Outre les exigences techniques, une mémoire doit être abordable pour les consommateurs. Cette dernière exigence est considérée comme mutuellement exclusive avec les autres. Par conséquent, une solution consistant en une mémoire à un niveau n'est pas pratique et une hiérarchie de mémoire est introduite afin de résoudre ce problème [109, 24, 117].

1.3.1 La hiérarchie mémoire

Un principe fondamental qui a trouvé l'intérêt de la hiérarchie de la mémoire et de la mémoire cache est la localité. Il existe deux types de localité :

Localité temporelle : Si un programme utilise un bloc de mémoire, ce bloc de mémoire est sus-

ceptible d'être réutilisé . La localité temporelle est aussi appelée locality in time. Par exemple, la plupart des programmes ont des boucles simples telles que les instructions et les données sont référencées de manière répétée. Dans la boucle for ***(i=0; i<N; i++)*** l'emplacement de mémoire qui stocke la variable i sera référencé de manière répétée.

Localité spatiale : Si un programme utilise un bloc de mémoire, il est probable que des blocs de mémoire proches de cette mémoire seront utilisés. La localité spatiale s'appelle également locality in space. Par exemple, les instructions de programme sont généralement consultées de manière séquentielle, si aucune branche ou aucun saut ne se produit. De plus, la lecture ou l'écriture de matrices entraîne généralement l'accès séquentiel à la mémoire.

Selon la localité temporelle, les blocs de mémoire situés dans la mémoire de niveau supérieur (par exemple, la mémoire principale) doivent être chargés dans la mémoire cache pour tirer parti de la latence. En fonction de la localisation spatiale, les blocs de mémoire plus proches des blocs auxquels vous avez accédé doivent également être lus au préalable dans le cache.
En raison du principe de localité, un système de mémoire rapide, volumineux et coûteux à un seul niveau est inutile. Dans un court intervalle de temps, un programme n'a pas besoin de toutes ses données accessibles immédiatement. Par conséquent, nous pouvons avoir un stockage multi-niveaux. Le premier niveau de stockage, rapide, petit et coûteux, fournit un accès immédiat à un sous-ensemble des données du programme. Le reste des données est stocké dans des niveaux de stockage plus élevés, plus lents mais plus volumineux et moins cher que la mémoire de premier niveau. La quasi-totalité des architectures modernes sont dotées d'une hiérarchie de mémoire constituée de plusieurs niveaux de stockage. Chaque niveau de stockage est optimisé pour un but. La figure 1.14 fournit une illustration de la hiérarchie d'une mémoire moderne.

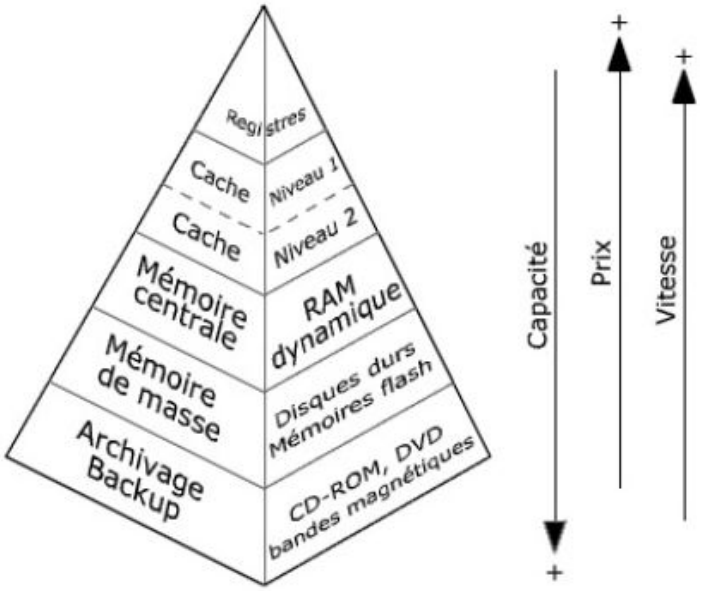


FIGURE 1.14 – Hiérarchie de la mémoire [90].

Disque : le disque offre un stockage permanent à un coût très faible par bit mais très lent ;

Mémoire principale : La mémoire principale est généralement constituée de DRAM (Dynamic Random Access Memory). Il fournit un stockage à accès aléatoire relativement volumineux, rapide et peu cher. La vitesse de la mémoire principale est assez lente comparée à la vitesse du processeur. Si un processeur peut exécuter 10 instructions par ns, il peut exécuter plus de 100 instructions dans l'attente de l'exécution d'un seul accès aux données sur la mémoire principale. La latence d'accès à la mémoire sur la mémoire principale est élevée par rapport au temps d'exécution d'une instruction ;

Mémoire cache : La mémoire cache est généralement constituée de SRAM (Static Random Access Memory). C'est une mémoire petite, mais extrêmement rapide, située entre le processeur et

la mémoire principale. La mémoire cache est introduite afin de réduire la latence d'accès à la mémoire. Les données fréquemment utilisées sont automatiquement chargées dans le cache. La capacité de la mémoire cache est souvent limitée et beaucoup plus petite que la mémoire principale pour la raison : coût et taille de la puce. Il est considéré coûteux d'avoir une mémoire cache volumineuse. En outre, la mémoire cache de premier niveau est généralement intégrée dans la puce de processeur et la taille de la puce est limitée.

1.3.2 Concepts de base sur la mémoire cache

Dans cette section, nous présentons d'abord la classification de la mémoire cache. Deuxièmement, nous détaillons l'organisation de la mémoire cache et expliquons comment un bloc de mémoire dans la mémoire principale est mappé dans la mémoire cache. De plus, nous présentons les opérations liés à ce composant matériel [61, 121].

1.3.2.1 Classification du cache

La mémoire cache est classée en fonction de la taille, de la latence d'accès à la mémoire et de la proximité avec le processeur. Il existe trois couches de cache sur la majorité des processeurs modernes comme montré dans la figure 1.15.

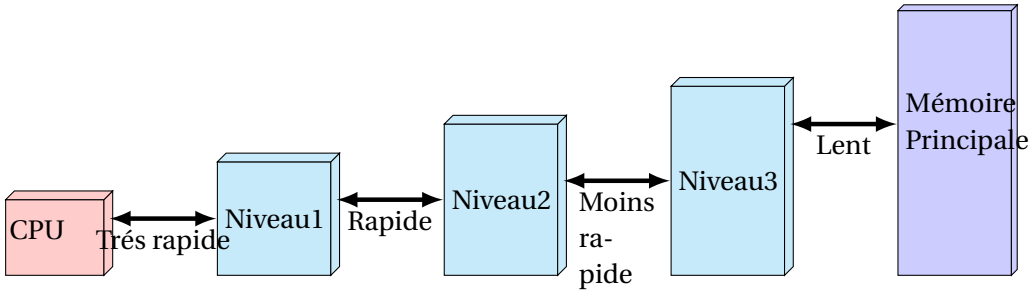


FIGURE 1.15 – Les niveaux de cache L1, L2, L3.

1-Le cache de niveau 1 (L1) : Le cache L1 est une mémoire cache extrêmement rapide mais relativement petite. Le cache L1 est généralement intégré à la puce du processeur. Il est très proche du processeur et il est accessible sur chaque accès en mémoire. Par conséquent, du point de vue architectural, ce cache doit disposer de nombreux ports en lecture/écriture et d'une bande passante d'accès très élevée. Il est considéré comme impossible ou extrêmement coûteux de créer un cache de grande taille L1 avec ces propriétés.

2-Le cache de niveau 2 (L2) : Le cache L2 est un peu plus lent mais plus grand que le cache L1. Le cache L2 peut être intégré dans la puce de processeur ou situé sur une puce séparée avec un bus alternatif à grande vitesse (séparé du bus système principal) interconnectant le cache au processeur. L'accès au cache L2 n'est possible que lorsqu'un manquement sur le cache L1 se produit. Ainsi, il peut avoir une latence d'accès mémoire plus élevée, moins de ports et une bande passante d'accès plus faible. Ces propriétés nous permettent d'agrandir le cache L2.

3-Le cache de niveau 3 (L3) : Le cache L3 est nettement plus lent et plus volumineux que les caches L1 et L2. L'accès au cache L3 n'est possible que lorsqu'un manquement sur le cache L2 se produit.

La mémoire cache peut également être classée en fonction des données stockées dans la mémoire cache.

a-Cache d'instruction : le cache d'instruction ne contient que des instructions de programme. Le processeur ne lit que dans le cache d'instructions et n'effectue aucune opération d'écriture.

b-Cache de données : le cache de données ne contient que des données de programme. Le processeur lit et écrit dans le cache de données.

c-Cache unifié : le cache unifié stocke les instructions et les données du programme.

Un accès à un bloc de mémoire dans la mémoire principale peut être classé en tant que succès de cache (cache hit) ou défaut de cache (cache miss) , définis comme suit :

Définition 1 (cache hit) : est un accès à un bloc de mémoire présent dans le cache ;

Définition 2 (cache miss) : est un accès à un bloc de mémoire qui ne se trouve pas dans le cache.

La figure 1.16 montre les deux type d'accès.

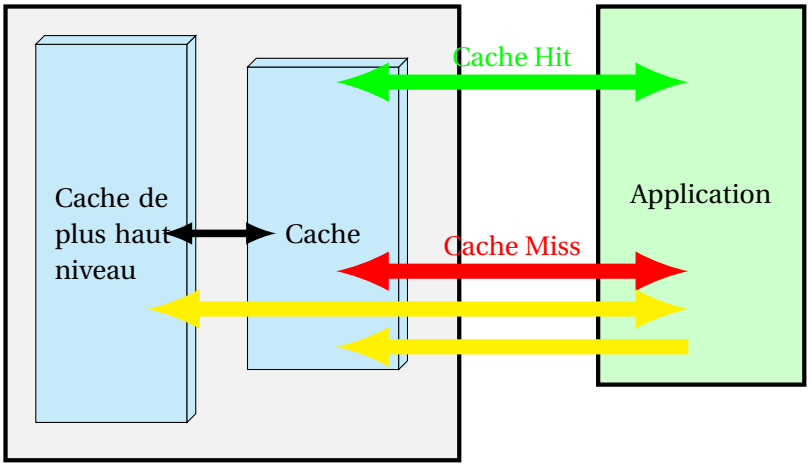


FIGURE 1.16 – Succès et défaut de cache.

Nous procédons en présentant l'organisation de cache et les opérations traitées sur ce composant matériel.

1.3.2.2 Organisation de cache

Pour comprendre l'organisation de la mémoire cache, nous présentons la définition du terme ligne de cache.

Définition 3 (ligne de cache) : La ligne de cache est la plus petite unité de données qu'un cache peut gérer.

Un cache est subdivisé en lignes de cache. La taille d'une ligne de cache est déterminée à la fois par le processeur et par la conception du cache. L'emplacement physique dans la mémoire cache où une ligne est stockée est appelé un bloc de cache. En fait, pour des raisons de simplicité, nous considérons que les deux termes sont équivalents. Maintenant, nous détaillons comment un bloc de mémoire dans la mémoire principale est mappé dans la mémoire cache. Le terme schéma de mappage mémoire à cache est défini comme suit :

Définition 4 (schéma de mappage mémoire à cache) : Le schéma de mappage mémoire à cache est un ensemble de règles qui spécifient comment un bloc de mémoire de la mémoire principale est mappé dans la mémoire cache.

L'implémentation matérielle de la mémoire cache peut être vue comme une table de hachage. La colonne clé est alors l'adresse d'un bloc de mémoire dans la mémoire principale. Il existe trois types de schéma de mappage mémoire à cache :

1-Mappage complètement associatif : un bloc de mémoire dans la mémoire principale peut être placé n'importe où dans le cache. Un schéma de mappage complètement associatif offre de meilleures performances. Étant donné que tout bloc de mémoire dans la mémoire principale peut être stocké sur n'importe quel bloc de cache, le nombre de défauts de cache est

inférieur. L'inconvénient de ce schéma est sa complexité. Si nous voulons déterminer si un bloc de mémoire dans la mémoire principale est dans le cache ou non, nous devons vérifier tous les blocs de mémoire présents dans le cache. En pratique, cela nécessite un grand nombre de comparateurs, ce qui augmente la complexité et le coût de la mise en œuvre de caches volumineux. Par conséquent, ce type de cache est généralement utilisé uniquement pour les petits caches, généralement inférieurs à 4 Ko. la figure 1.17 montre le mappage complètement associatif.

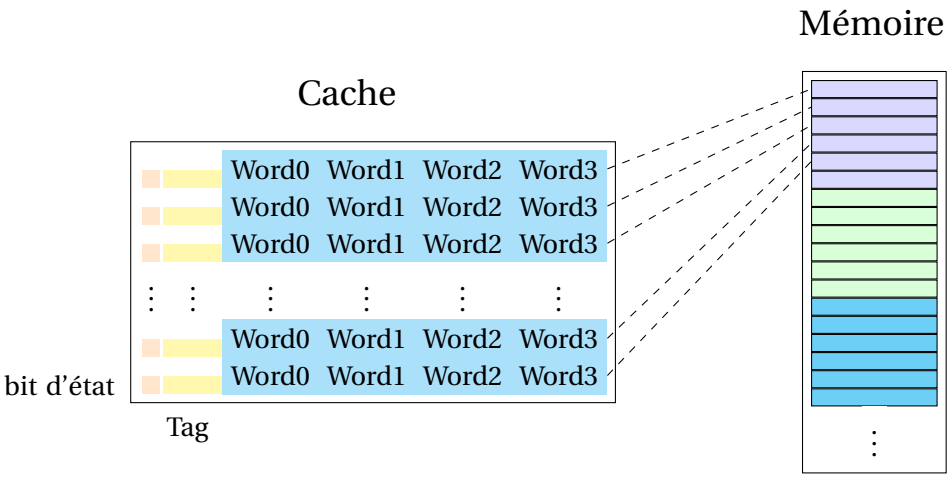


FIGURE 1.17 – Mappage complètement associatif.

2-Mappage Direct : un bloc de mémoire dans la mémoire principale ne peut être mappé que sur un seul bloc de cache unique dans le cache. Le mappage est généralement calculé comme suit : (Adresse de bloc) MOD (Nombre de blocs dans le cache). Le mappage direct est le schéma de mappage le plus simple. Cela nous oblige uniquement à comparer l'adresse d'un bloc dans la mémoire principale à l'adresse d'un bloc de cache. L'avantage de ce schéma de mappage est qu'il est simple et peu coûteux à mettre en œuvre. Cependant, l'inconvénient est qu'un cache mappé directement n'est pas flexible et offre souvent de faibles performances en raison du nombre élevé de conflits de cache. Le schéma de mappage direct est présenté dans la figure 1.18.

3-Mappage N-Associatif : un bloc de mémoire dans la mémoire principale peut être placé dans un ensemble de blocs de cache. Le cache est appelé cache n-way set-associative. Le cache est organisé de différentes manières. les plus courants sont 2, 4 et 8. En fait, nous pouvons considérer le cache mappé directement comme un cache 1-associatif. Un cache N-associatif est une combinaison d'un cache mappé direct et d'un cache complètement associatif. Il présente les avantages et les inconvénients moyens des deux schémas de mappage La figure 1.19 illustre le mappage N-associatif.

1.3.2.3 Les opérations sur la mémoire cache

Nous présentons deux opérations concernant le bloc de mémoire dans la mémoire cache : lecture (reading) et écriture (writing).

1-Lecture : Un bloc peut être identifié dans le cache ou non en fonction de deux informations : un indicateur (flag : valide ou non valide) et une balise (tag) pour chaque bloc. Lorsque l'ordinateur démarre, la mémoire cache est vidée et tous les blocs sont marqués comme non valides. L'indicateur devient valide lorsque les données sont écrites dans le set de cache.

L'accès aux données du cache se fait dans l'ordre suivant :

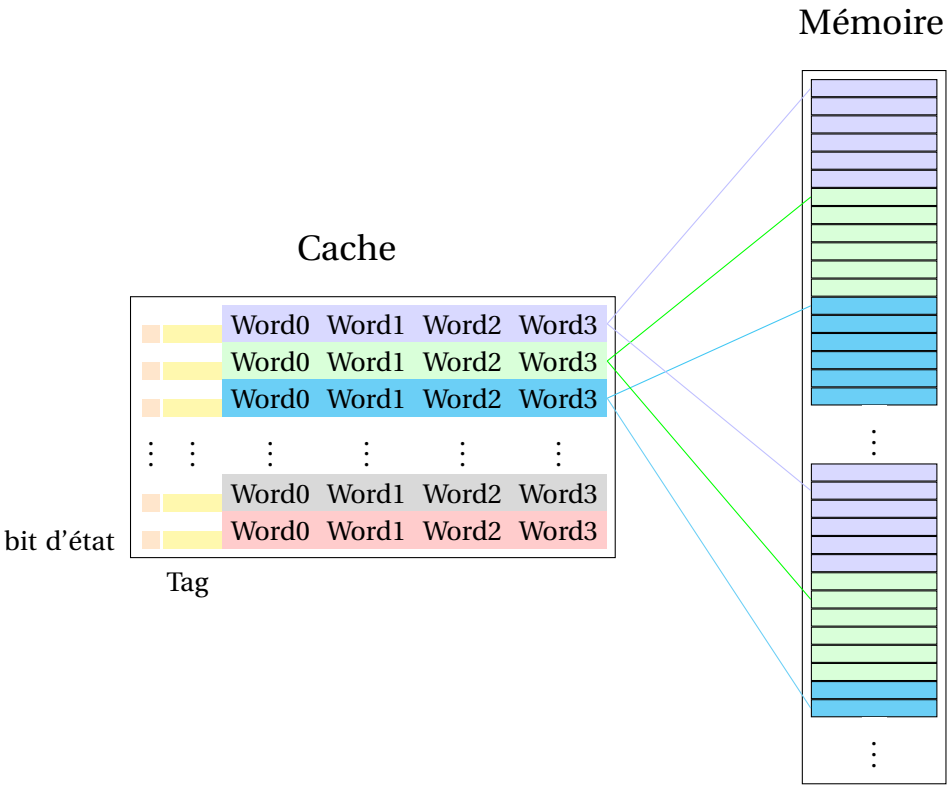


FIGURE 1.18 – Mappage Direct

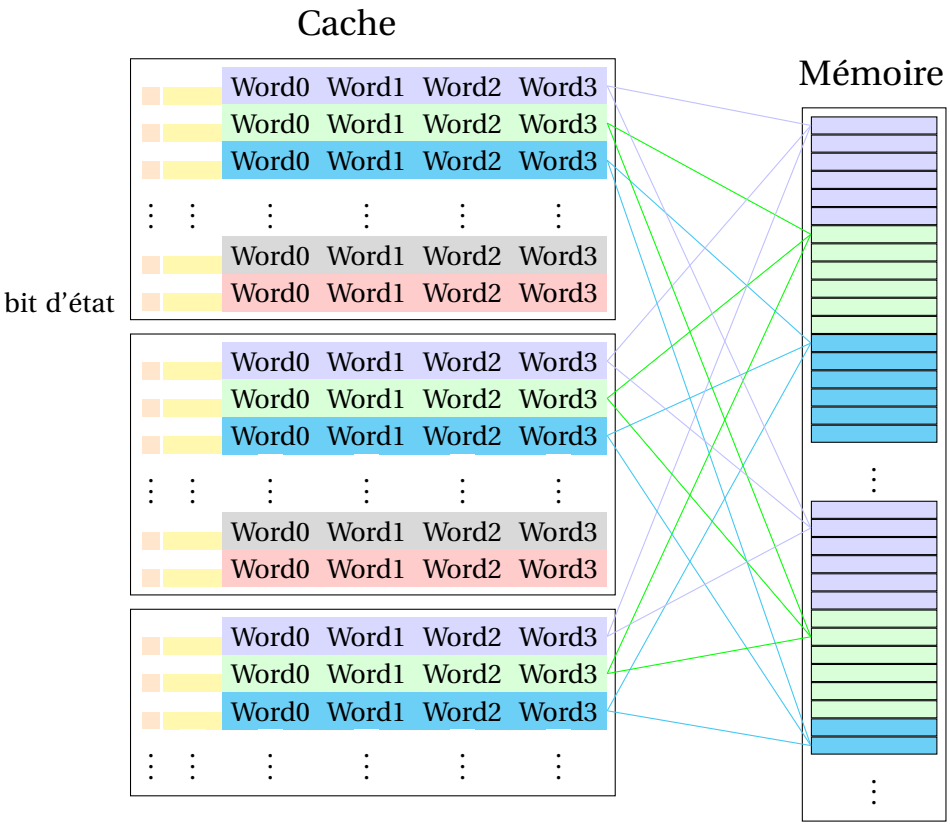


FIGURE 1.19 – Mappage n-associatif.

1. le champs set est utilisé pour trouver le set de cache ;
2. Tous les champs tags valides de set sont comparés au champ de tag de l'adresse. Si le résultat est égal, nous obtenons un succès (hit). Si ce n'est pas le cas, nous obtenons un échec (miss) et le bloc correct doit être chargé à partir de la mémoire de niveau inférieur ;
3. Le champ word est utilisé pour trouver la position du mot dans le bloc.

Les défauts de cache (miss) peuvent se produire pour trois raisons :

1. **Obligatoire** : la ligne n'est pas dans le cache car les blocs associés sont vide.
2. **Conflit** : la ligne n'est pas dans le cache et tous les blocs associés à un set sont utilisés.
3. **Capacité** : la mémoire cache est pleine.

2-Ecriture : il existe deux stratégies pour écrire sur le hit et deux pour écrire sur le miss. Lorsqu'un accès au cache survient (hit), l'écriture peut se faire de deux manières différentes :

1. **Write-through** : l'écriture sur le cache est également effectuée au niveau de la mémoire inférieure.
2. **Write-back** : l'écriture est seulement faite sur le cache, l'écriture sur la mémoire inférieure est faite quand le bloc est remplacé.

Il existe également deux stratégies pour écrire sur la politique Write-through dans le cas de miss :

1. **Write allocate** : le bloc est écrit dans la mémoire inférieure puis chargé dans le cache.
2. **No-Write allocate** : le bloc n'est modifié que dans la mémoire inférieure.

1.4 Les architectures à mémoire partagée

L'absence de mise à l'échelle des fréquences a motivé l'augmentation du nombre de cœurs par processeur. Les architectures à mémoire partagée actuelles visent donc à fournir une puissance de calcul supérieure en agrégeant des unités de traitement plus petites. Dans cette partie, afin d'analyser les architectures à mémoire partagée utilisées dans cette thèse, nous les avons classées en processeurs multicœurs et many-cœurs. Bien que le nombre de cœurs qui déterminent la division ne soit pas clair, nous avons considéré comme multi-cœurs les processeurs autonomes, c'est-à-dire ceux fournissant des fonctionnalités parallèles par la combinaison de cœurs polyvalents, et autant many-cœurs, les accélérateurs ou les coprocesseurs avec une agrégation de cœurs qui peut être utilisés pour des usages spécifiques dans le domaine de parallélisation.

1.4.1 Les architectures multicœurs

La tendance actuelle du calcul haute performance à obtenir des performances évolutives consiste à augmenter le nombre de cœurs disponibles par processeur sur des machines à mémoire partagée. La conception de la puce multicœur et les efforts déployés pour surmonter les limitations matérielles des systèmes parallèles multiprocesseurs symétriques (SMP) classiques ont conduit à l'émergence des architectures hiérarchiques. Ces architectures présentent une topologie complexe et un sous-système de mémoire hiérarchique. Dans cette section, nous présentons une description plus détaillée de cette architecture et ses composants. Nous discutons également de l'évolution de l'architecture monocœur à l'architecture multi-cœurs [56, 48, 64, 87].

1.4.1.1 Les architectures à mémoire partagée hiérarchique

Dans cette thèse, nous considérons comme une architecture à mémoire partagée hiérarchique tout plate-forme multiprocesseur comprenant : (i) des unités de traitement partageant une mémoire globale et (ii) des unités de traitement et des composants de mémoire organisés selon une topologie hiérarchique. Dans ce contexte, nous citons les deux architectures à mémoire partagée hiérarchique : les machines UMA (Uniform Memory Access), les machines NUMA (Non-Uniform Memory Access).

a-Architecture UMA : Sur les plates-formes UMA, toutes les unités de traitement ont des coûts d'accès similaires à la mémoire partagée globale. Cela est dû au fait que la mémoire partagée globale est connectée à un seul bus utilisé par les unités de traitement pour accéder à la mémoire. De plus, dans cette architecture, les éléments de traitement partagent les périphériques, qui sont également connectés au bus unique. Le principal problème de cette conception est que le bus devient un goulot d'étranglement, car toutes les unités de traitement doivent l'utiliser pour accéder à la mémoire globale et aux périphériques. Par conséquent, un tel bus limite l'évolutivité de l'architecture UMA. Nous considérons cette architecture comme un multiprocesseur à mémoire partagée hiérarchique en raison de la topologie de base actuelle conçue au sein de ces machines. Les plates-formes UMA actuelles présentent une topologie complexe, avec plusieurs processeurs, puces multicœurs et mémoires cache. La figure 1.20 illustre une plate-forme multicœur avec une conception UMA. Sur la figure, nous pouvons observer que la machine dispose de quatre processeurs et chacun de six cœurs. Compte tenu du sous-système de mémoire, la machine dispose de deux niveaux de mémoire cache partagée. Chaque paire de cœurs partage une mémoire cache L2 et chaque processeur dispose d'une mémoire cache L3 partagée. La mémoire principale est partagée entre tous les cœurs de la machine et accessible par tous les cœurs via un seul bus. Même si ces machines ont un accès uniforme à la mémoire partagée, il est important de prendre en compte la topologie lors du mappage du processus / threads de l'application. En raison de l'organisation hiérarchique des cœurs, des processeurs et des mémoires cache, le temps de communication entre les unités de traitement peut varier en fonction de la distance qui les sépare. Par exemple, dans la machine présentée à la figure 1.20, la hiérarchie du cache peut être explorée pour réduire le temps de communication entre un groupe de threads. Ce groupe peut être placé dans le même processeur, en évitant la communication entre processeurs [109].

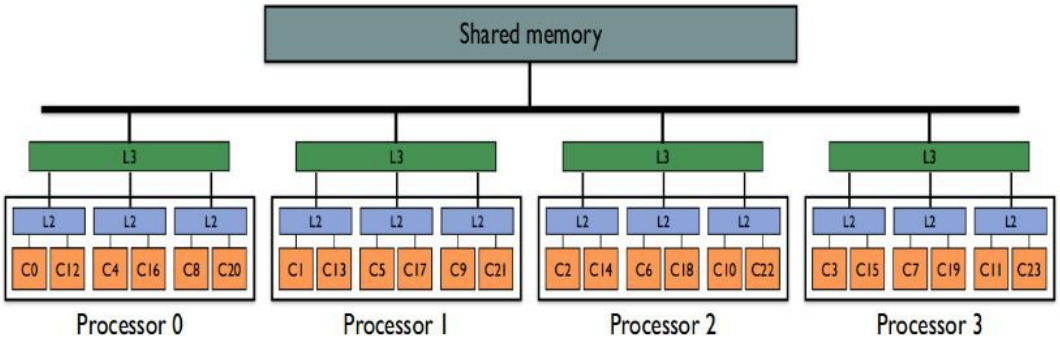


FIGURE 1.20 – L'architecture UMA [109].

b-Architecture NUMA : Une plate-forme NUMA est un système multiprocesseur dans lequel les éléments de traitement sont desservis par plusieurs banques de mémoire, distribuées physiquement via la plate-forme. Bien que la mémoire soit physiquement distribuée, les unités de traitement de la machine la voient comme une mémoire partagée unique. Dans ces machines, le temps passé à accéder aux données est conditionné par la distance entre le processeur et la banque de

mémoire dans laquelle les données sont physiquement allouées. Les architectures NUMA sont généralement conçues avec des mémoires cache, afin de réduire les pénalités d'accès à la mémoire. Pour cette raison, certaines mesures visant à garantir la cohérence du cache pour les unités de traitement sont mises en œuvre dans les plates-formes NUMA actuelles, ce qui aboutit à des plates-formes NUMA cohérentes avec un cache (cache coherent NUMA -ccNUMA-). L'un des avantages de l'architecture NUMA est qu'elle associe une bonne évolutivité de la mémoire à une caractéristique de programmation simple. Dans ces machines, un réseau d'interconnexion efficace et spécialisé prend en charge le nombre élevé d'unités de traitement et de très grandes mémoires. La mémoire étant considérée comme une mémoire partagée globale, les programmeurs peuvent utiliser des modèles de programmation à mémoire partagée pour développer des applications parallèles sur ces machines. Les figures 1.21 et 1.22 montrent des schémas représentant deux machines NUMA. Celui représenté à la figure 1.21 rapporte les machines NUMA classiques des années 80 alors que la figure 1.22 décrit une machine NUMA actuelle avec des puces multicœurs. Nous pouvons observer dans les deux figures que les machines NUMA sont organisées en plusieurs nœuds connectés par un réseau d'interconnexion. Chaque nœud est généralement composé de plusieurs unités de traitement (mono-cœur ou multi-cœur) et de banques de mémoire.

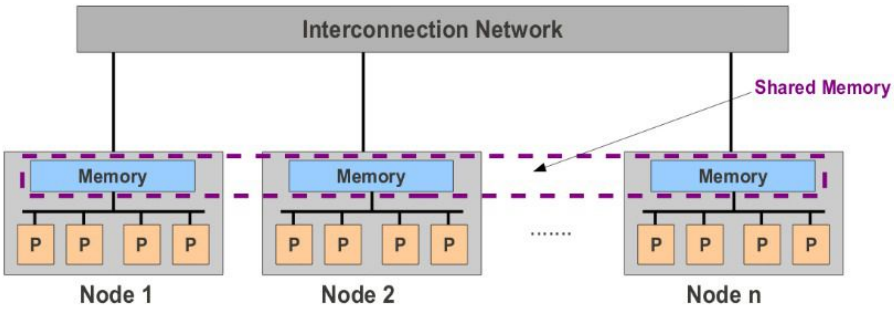


FIGURE 1.21 – L'ancienne architecture NUMA [109].

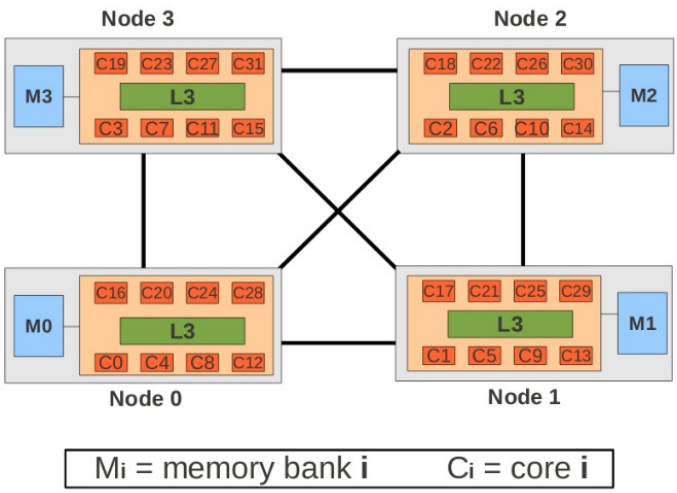


FIGURE 1.22 – La nouvelle architecture NUMA [109].

1.4.1.2 Architecture monocœur à multicœurs

Au cours des deux dernière décennies, le principal défi des recherches en informatique dans le domaine de l'architecture informatique a été de maintenir la célèbre loi de Moore présenté par la figure 1.23 [26, 44]. Cette loi stipule que la puissance du processeur est dupliquée tous les 18 mois. Cependant, à la fin des années 90, les chercheurs en architecture informatique avaient déjà du mal à confirmer cette loi lors de la conception de processeurs monocœur. Un processeur monocœur est caractérisé par une seule unité de traitement (core) responsable du calcul de toutes les instructions. L'architecture d'un processeur mono-cœur est illustré dans la figure 1.24a. Cette unité de traitement unique a généralement une unité arithmétique et logique et des composants de mémoire pour stocker des données. Dans l'architecture de processeur monocœur, une nouvelle conception était nécessaire pour augmenter sa fréquence. En raison de la demande pour des processeurs plus puissants, les chercheurs ont utilisé des mécanismes de parallélisme au niveau des instructions et au niveau des threads pour améliorer les performances des processeurs.

Le parallélisme au niveau des instructions permet d'exécuter plusieurs instructions à la fois. Afin de prendre en charge le parallélisme au niveau des instructions, les processeurs mono-cœur implémentent des techniques de pipeline et de plusieurs unités arithmétiques. La technique du pipeline divise l'exécution d'une instruction en plusieurs étapes, permettant à certaines d'entre elles d'être exécutées en parallèle. La technique des unités arithmétiques multiples permet l'exécution de plusieurs opérations simultanément. Le mécanisme de parallélisme au niveau des threads prend en charge l'exécution de plusieurs threads dans le même processeur monocœur. Ce mécanisme est possible grâce à la réplication de certains composants du processeur. Cependant, même ces mécanismes n'ont pas permis l'amélioration nécessaire des performances des machines parallèles. Depuis que les processeurs mono-cœur ont été utilisés pour construire des machines SMP à grande échelle, les groupes de chercheurs se sont efforcés d'améliorer leur puissance en intégrant la conception multi-cœur. Un processeur multicœur est composé de deux cœurs indépendants ou plus, partageant certains composants matériels.

Les deux figures 1.24b et 1.24c illustrent deux architectures multi-cœurs. Le concept principal de l'architecture multicœur est l'utilisation d'unités de traitement avec des fréquences plus courtes qui fonctionnent ensemble pour accomplir un travail. De cette manière, les processeurs multicœurs peuvent faire plus de travail que les processeurs mono-cœur, en raison de leur parallélisme implicite. De plus, la distance entre les unités de traitement et les mémoires est plus petite dans les processeurs multicœurs. Par conséquent, la latence nécessaire pour effectuer une communication peut être plus courte, ce qui améliore les performances globales du processeur.

1.4.1.3 Exemples d'architecture multicœurs

Le reste de cette section décrit deux architectures multicœurs que nous avons utilisées pour évaluer les performances des approches étudiées dans cette thèse. La première est basé sur le Intel Ivy Bridge (Xeon E5) et la deuxième est un processeur AMD MagnyCours. Les caractéristiques architecturales sont détaillées. D'une part, le processeur Xeon E5 présente une micro-architecture assez récente (Intel Ivy Bridge) pour des systèmes de mémoire partagée hautes performances, permettant d'atteindre des performances maximales par cœur de processeur dans notre approche proposée, ce qui représente : une des meilleures performances par cœur parmi les systèmes actuellement disponibles. D'autre part, le système basé sur Magny-Cours à 48 cœurs a l'un des chiffres de performances agrégées les plus élevés dans un système à mémoire partagée , bien que ses performances par cœur soient assez réduites . Le système Xeon E5 nous a donc permis d'analyser une micro-architecture Intel assez récente, tandis que le système Magny-Cours présente les problèmes liés à l'intégration de plusieurs processeurs multicœurs.

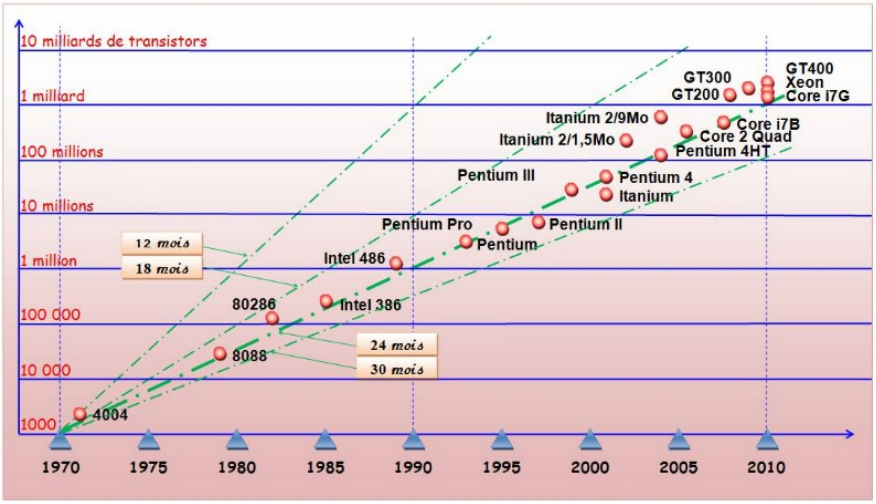


FIGURE 1.23 – Loi de Moore [26].

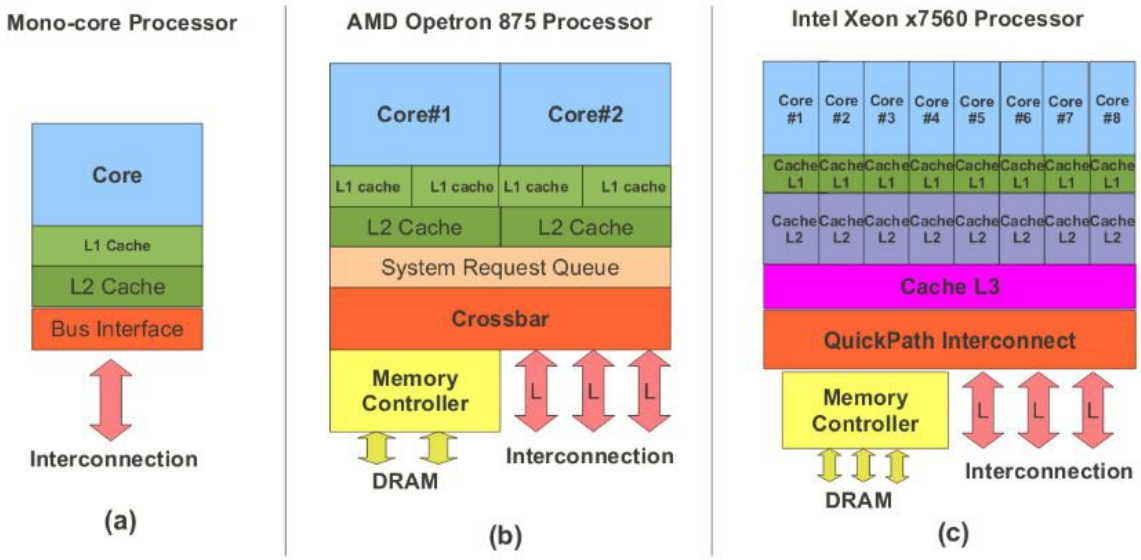
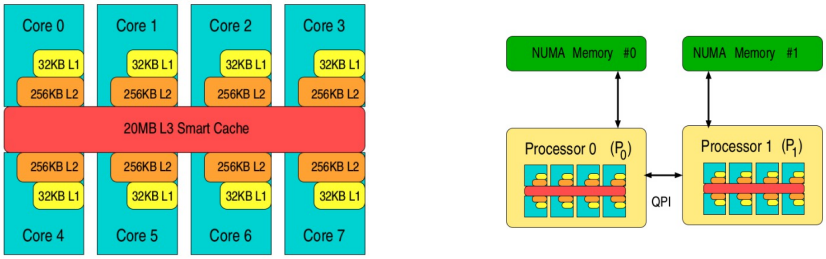


FIGURE 1.24 – L'évolution monocœur à multicœurs [109].

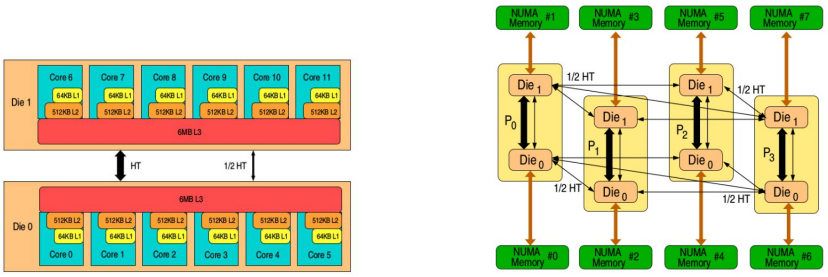
a-Intel Xeon E5 La figure 1.25a présente le schéma d'un processeur Xeon E5 à 8 cœurs, basé sur l'architecture Ivy Bridge, dans lequel jusqu'à 16 threads peuvent s'exécuter simultanément grâce à l'hyperthreading. Les huit cœurs de ce processeur partagent le cache L3 (appelé LLC ou cache de dernier niveau), implémenté en tant que Intel Smart Cache, où chaque cœur peut accéder à la totalité du cache lorsque le reste des cœurs est inactif. Ce cache est divisé en tranches physiques connectées à un bus en anneau interne. La figure 1.25b illustre la configuration de l'interconnexion dans un système Intel Xeon E5-2670 à double socket où les processeurs et la mémoire sont liés par un QuickPath Interconnect (QPI). Ce système NUMA prend en charge la mémoire DDR3-1600 MHz [119, 70].



(a) Processeur Intel Xeon E5-2650. (b) Système Dual-socket Intel Xeon E5-2650.

FIGURE 1.25 – Architecture d'un système Intel Xeon E5 basé sur Ivy Bridge [119].

b-AMD Opteron Le second système, un gros nœud du cluster DAS-4, possède 48 cœurs dans 4 processeurs AMD Opteron 6172 (Magny-Cours), chacun avec 12 cœurs et 128 Go de RAM. La figure 1.26a présente la structure du processeur Magny-Cours à 12 cœurs, composé de deux sockets AMD Opteron à 6 cœurs (les 6 cœurs partagent le cache L3) interconnectés par des liaisons HyperTransport (HT). La figure 1.26b montre les interconnexions HT entre les différentes puces ainsi que l'accès direct de chaque puce à sa région de mémoire prenant en charge la technologie DDR3-1333 MHz. Les flèches minces représentent les demi-liaisons HT (8 bits), tandis que les épaisses représentent les liaisons HT complètes (16 bits). Comme chaque processeur à 12 cœurs est un système NUMA avec 2 régions NUMA, ce système à quatre sockets possède huit régions NUMA [87, 126].



(a) Processeur AMD Opteron 1672 . (b) Système Quad-socket AMD Opteron 6172.

FIGURE 1.26 – Architecture d'un système AMD Opteron basé sur magny-Cours [87].

1.4.2 Les architectures many-cœurs

Les processeurs multicœurs fournissent des performances de calcul agrégées élevées, mais lorsque le nombre de cœurs est grand, les probabilités de tirer parti de tous les cœurs en même

temps sont réduites, car les codes parallèles ont également des parties séquentielles et la majorité d'entre eux ne peuvent pas exploiter toutes les ressources fournies par un processeur multicœur. Une situation plus courante est un code parallèle avec des parties non équilibrées et hétérogènes qui ne tireront pas avantage de plusieurs cœurs similaires. Dans ce scénario, avoir un grand nombre de cœurs est très inefficace en termes de consommation d'énergie, et même en termes de coût, car cela pourrait être résolu en disposant d'un ensemble moins coûteux de cœurs plus petits offrant une fonctionnalité spécifique. C'est l'objectif des accélérateurs ou des coprocesseurs many-cœurs. Ils fournissent des architectures massivement parallèles avec des unités de calcul simples ou des cœurs pour accélérer les sections d'une application exécutée dans un processeur à usage général. Parmi ces architectures many-cœurs, cette section s'articule autour de deux d'entre elles extrêmement intéressantes : l'Intel Xeon Phi (un accélérateur à architecture x86) et les GPU (Fermi, Kepler, Maxwell, Pascal et Volta), actuellement présents dans les supercalculateurs les plus puissants.

1.4.2.1 Intel ® Xeon Phi

Le processeur Intel Xeon Phi est le dernier et le plus évolutif des coprocesseurs many-cœurs x86 . Le premier produit commercial de l'architecture MIC (Many Integrated Cores), il reflète les efforts déployés par Intel pour développer un coprocesseur à plusieurs cœurs. L'objectif principal était de fournir un accélérateur à programmer avec les paradigmes et les langages traditionnels, évitant ainsi le besoin à l'informatique GPGPU et à des modèles de programmation complexes. Cette solution a également été explorée par IBM Cell Broadband Engine , qui a mis en œuvre une nouvelle architecture également utilisée dans les consoles de jeu vidéo populaires telles que la PlayStation3 de Sony et il a même atteint la 1ère position dans la liste des top 500 en juin 2008 avec IBM Roadrunner, mais il a terminé sa production en 2009 [39, 115, 107].

a-Architecture Hardware Le coprocesseur Intel® Xeon Phi comprend jusqu'à soixante-et-un (61) cœurs sur une puce connectés via une interconnexion bidirectionnelle. Outre les cœurs, 8 contrôleurs de mémoire et autres dispositifs spéciaux y sont également installés, tels qu'un contrôleur de mémoire (GBOX), une logique client PCI Express (SBOX) ou un moteur d'affichage (DBOX). Comme le montre la figure 1.27 , chaque noyau contient :

1. Unité de traitement vectoriel de 512 bits (VPU) ;
2. Le Core Ring Interface (CRI) ;
3. Le cache L2 ;
4. Le répertoire des balises (TD) ;
5. Contrôleur d'interruption de processeur asynchrone (APIC).

Le noyau récupère et décode les instructions à partir de quatre contextes d'exécution de threads matériels. L'unité de traitement vectorielle (VPU) exécute des opérations à virgule flottante et en nombre entier. Le Core Ring Interface héberge le cache L2, APIC, TD et connecte chaque core à Ring Stop. TD maintient la cohérence entre les caches L2 du noyau. Le contrôleur de mémoire est composé de deux canaux de mémoire indépendants, chaque canal ayant une largeur de 32 bits et fournissant une connexion à la RAM. SBOX inclut la logique client Express PCI et les fonctionnalités de gestion de l'alimentation. DBOX prend en charge le débogage du Xeon Phi.

b-Architecture software Le Xeon Phi est implémenté en tant que collection étroitement intégrée de cœurs de processeur sur la carte d'extension PCI Express. La carte est conforme aux spécifications PCI Express pour le point de terminaison PCI Express. Cela implique qu'il y ait trois espaces d'adressage (configuration, mémoire, E / S). Chaque carte représente un domaine de multitraitement symétrique (SMP) symétriquement couplé au domaine de calcul représenté par la plateforme hôte. De nombreuses API prennent en charge de nombreuses applications de calcul haute

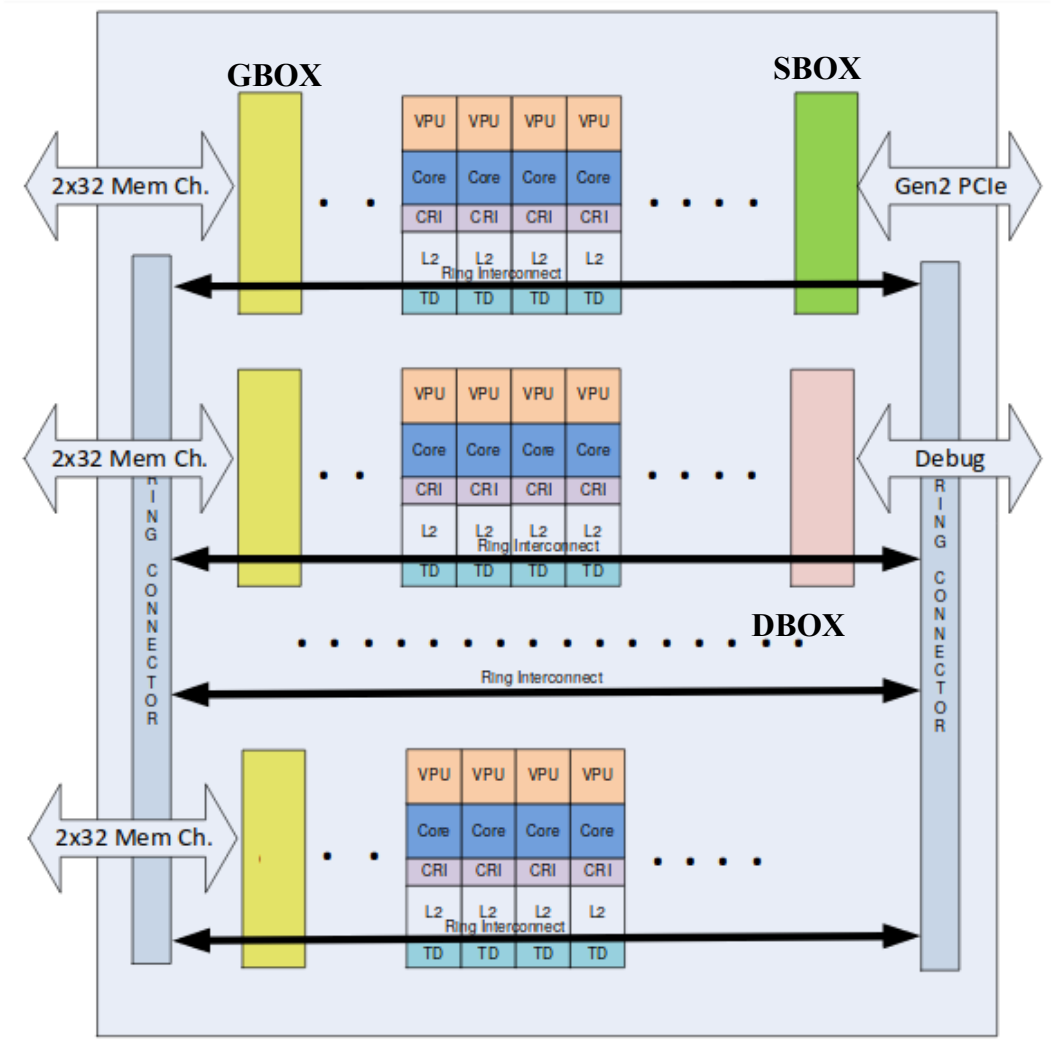


FIGURE 1.27 – Architecture de base d'un processeur Intel Xeon Phi [107].

performance (HPC). Il existe également des API TCP / IP, MPI, OpenCL et certaines autres interfaces Intel, qui créent des couches d'abstraction. La couche de communication entre l'hôte et la carte s'appelle SCIF. Tous ensemble, ils créent une pile de logiciels pour la plateforme Many-cœurs (MPSS). MPSS est le nom collectif de tous les logiciels de l'hôte et de la carte prenant en charge le coprocesseur. La figure 1.28 illustre les relations entre les API et les autres composants. Le côté gauche de l'image montre la pile d'hôte exécutée sur un noyau Linux standard. Le côté droit montre une pile similaire qui appartient au coprocesseur et qui s'exécute au-dessus du noyau Linux avec les ajustements Xeon Phi.

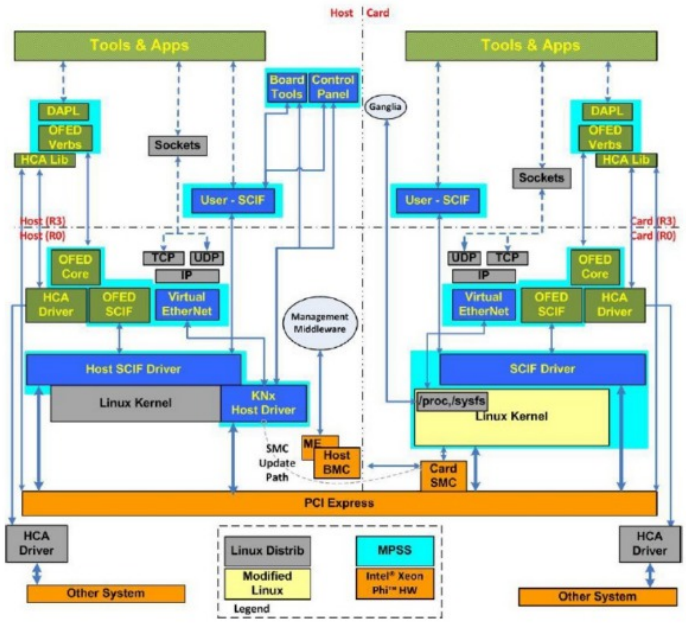


FIGURE 1.28 – Architecture software d'un processeur Intel Xeon Phi [107].

L'architecture logicielle prend en charge trois modèles de programmation – déchargement, symétrique et natif. Dans le modèle de déchargement, l'application principale est lancée sur le processeur hôte. Le processus de déchargement est contrôlé par certaines directives du compilateur et fait partie du domaine d'activité du compilateur. Le modèle symétrique est composé de plusieurs processus, où le calcul et la communication doivent être effectués explicitement en utilisant un mécanisme standard, par exemple MPI. Le dernier modèle natif est la variante, où l'application s'exécute exclusivement sur le coprocesseur Xeon Phi. Contrairement aux ordinateurs standard, le Xeon Phi ne contient pas de système de stockage permanent. Sa base de données est maintenue dans la RAM et peut être montée à distance, par exemple via NFS.

1.4.2.2 Graphics Processing Unit (GPU)

L'architecture du GPU est optimisée pour le parallélisme au niveau des données. Le processeur graphique GPU est conçu à l'origine comme un coprocesseur (device) spécialisé et dédié à accélérer les calculs intervenant dans le rendu interactif d'images de synthèse. Il prend place au sein d'un système hôte (host), constitué d'un ou plusieurs CPU, d'un espace mémoire partagé et d'autres périphériques. le GPU utilise un environnement d'exécution et un pilote de périphérique. Un programme utilisant le GPU sera divisé en deux parties : un programme principal exécuté par le processeur hôte, dont une des tâches consistera à configurer le GPU, et éventuellement un ou

plusieurs noyaux de calcul exécutés par le GPU. Pour exécuter un programme, le CPU exécute ses instructions séquentiellement dans le host. Le device et le host n'ont pas l'accès à leur mémoire. Donc, nous devons allouer la mémoire que le device doit utiliser pour l'exécution des instructions sur le GPU. Une fois la mémoire allouée, il faut transférer les données du host vers le device pour l'exécution d'une manière parallèle. La figure 1.29 montre le modèle d'exécution d'un processeur GPU [101, 80, 98].

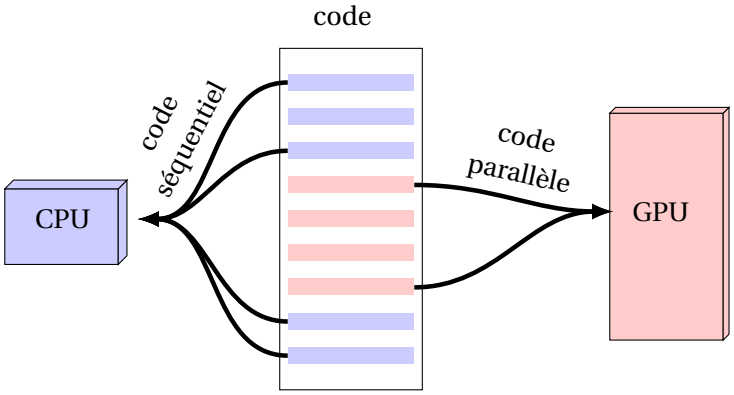


FIGURE 1.29 – Mode d'exécution de processeur GPU.

La majorité des GPU sont développé par la société NVDIA. Le GPU NVIDIA révèle que la puce est organisée en un ensemble de multiprocesseurs de flux hautement threadés (SM). Chaque multiprocesseur de flux contient plusieurs processeurs de flux (SP). Les multiprocesseurs de flux (SM) sont regroupés dans des clusters de traitement de thread (TPC). Le nombre de SP par SM dépend de la génération de la puce.

Depuis la sortie de la puce Fermi en 2010, NVIDIA a développé et produit deux autres puces GPU : Kepler et Maxwell.
Maxwell est le successeur de l'architecture Kepler. Débutant en février 2014, Maxwell a été conçu dans un souci d'efficacité énergétique. L'objectif était d'améliorer la performance par watt et la performance par zone. Des fonctionnalités ont également été ajoutées pour améliorer l'utilisation du GPU dans cette architecture, telles que le parallélisme dynamique, Hyper-Q et la création de l'unité de gestion de la grille. La figure 1.30 montre L'architecture globale du GM200, un des processeurs GPU basés sur l'architecture Maxwell. Un ensemble évolutif de six GPC (Graphic processor Cluster) constitue le GPU, avec quatre SM Maxwell (SMM) par GPC. un cache L2 de taille 3 072 Ko est partagée entre les six GPC.

La puce GM200 possède 3072 processeurs de streaming (SP).Chaque processeur de flux (SM) est threadé et peut exécuter des milliers de threads par application. Les GPU sont optimisés via le débit d'exécution d'un grand nombre de threads. Le matériel en tire parti en passant à différents threads pendant que d'autres threads attendent des accès mémoire à latence longue. Cette méthodologie permet une logique de contrôle très minimale pour chaque thread d'exécution. Chaque thread est très léger et nécessite très peu de temps de création. Du point de vue de la mémoire, l'architecture du GPU est très différente de celle d'un CPU. Le GPU GM200 est fourni avec jusqu'à vingt quatre gigaoctets de mémoire vive dynamique (GDDR). Comme illustré à la figure 1.31, chaque module SMM est divisé en quatre blocs de traitement, chacun contenant son propre tampon d'instructions, son propre scheudleur et ses unités de répartition, 32 SP, 8 unités LD / ST (load/store) et 8 SFU (Special Function Unit). Cette division simplifie la logique de conception et d'ordonnancement en réduisant les zones, la puissance et la latence des calculs. Cette réduction de puissance, avec une réduction minimale des performances, permet de placer davantage de SMM dans le GPU. Les quatre blocs partagent 96 Ko de mémoire partagée dédiée, ce qui est

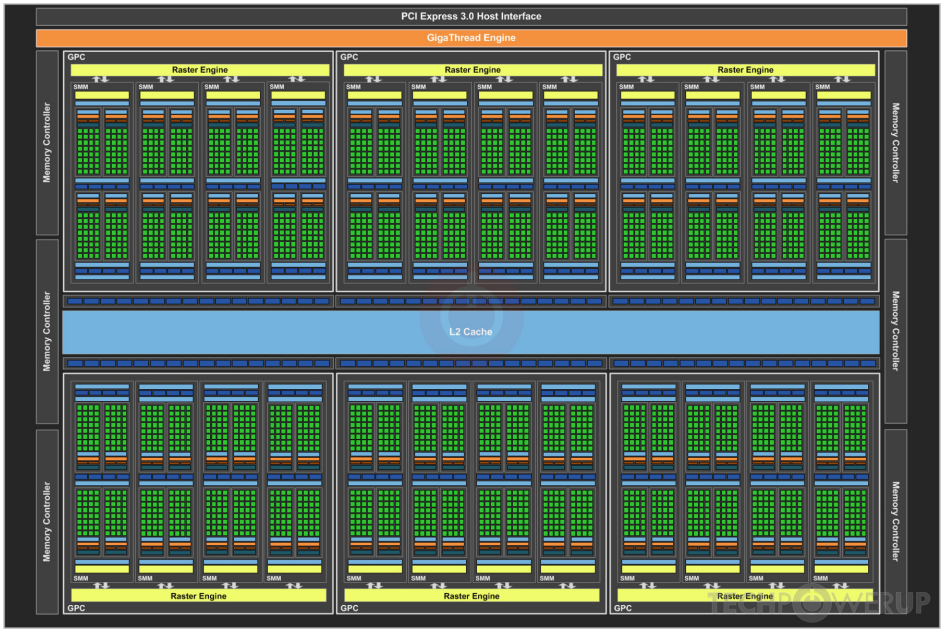


FIGURE 1.30 – L'architecture de GPU GM200 [98].

différent des architectures précédentes. La fonctionnalité de mise en cache L1 a été déplacée pour être partagée avec le cache de texture.

Les processeurs graphiques ont traditionnellement été conçus pour des tâches spécialisées très spécifiques. La plupart de leurs transistors effectuent des calculs liés au rendu graphique 3D. En général, les GPU effectuent un travail gourmand en mémoire, tel que le mappage de texture et le rendu de polygones. Le GPU effectue également des calculs géométriques tels que la rotation et la translation des sommets dans différents systèmes de coordonnées. Les processus de décodage vidéo spécialisés sont optimisés sur le GPU moderne. Ces processus incluent :

1. Compensation de mouvement (mocomp) ;
2. Transformée en cosinus discrète inverse (iDCT) ;
3. Correction ascendante du télécinéma ;
4. Transformée en cosinus discrète modifiée inverse (iMDCT) ;
5. filtre de déblocage à n boucles ;
6. Prédiction intra-imageQuantification Inverse (QI) ;
7. Décodage à longueur variable (VLD) ;
8. Traitement de flux binaire (CAVLC / CABAC).

NVIDIA a développé une architecture informatique parallèle appelée CUDA (Compute Unified Device Architecture). Ce moteur informatique, qui constitue le cœur des GPU NVIDIA modernes, est accessible aux développeurs de logiciels via des extensions de langages de programmation standard. Le développement de CUDA a permis aux développeurs d'accéder au jeu d'instructions virtuel et à la mémoire du GPU. Cela a permis d'exploiter les éléments de calcul parallèle natifs du GPU NVIDIA. La figure 1.32 illustre de manière générale les étapes nécessaires au transfert de données et à l'exécution de code sur le processeur graphique.

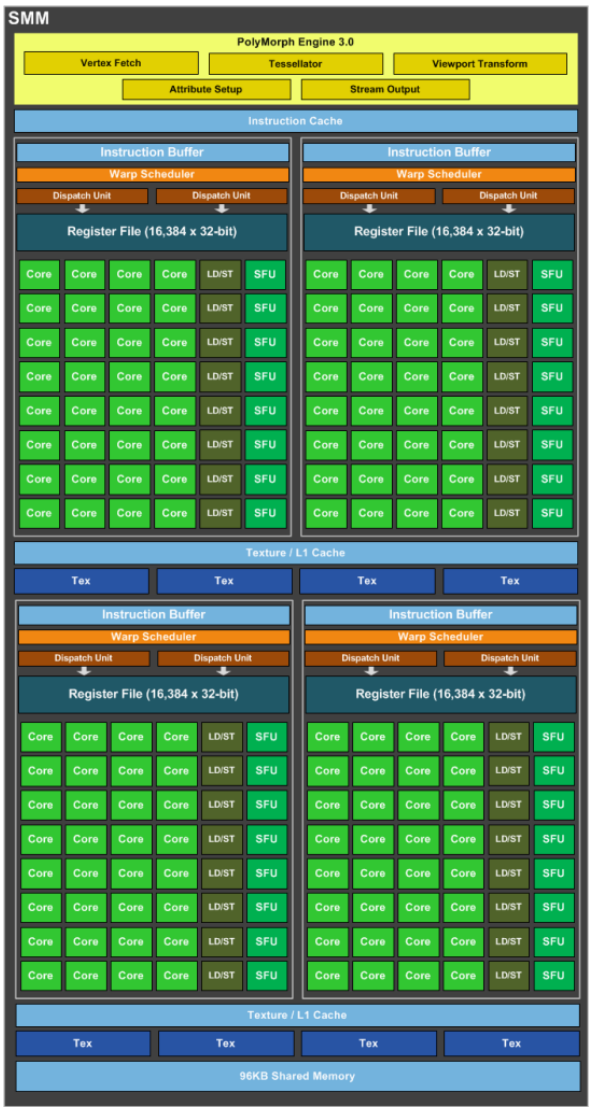


FIGURE 1.31 – L'architecture de SM Maxwell [98].

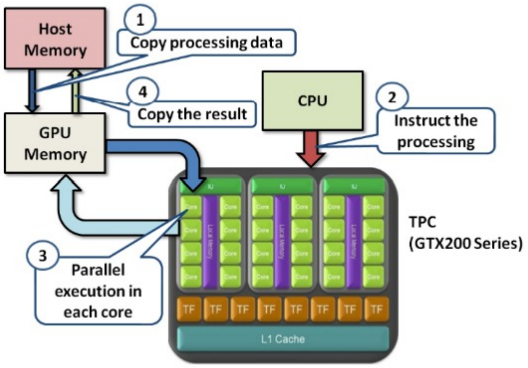


FIGURE 1.32 – Exemple de processus CUDA [98].

1.5 Conclusion

Dans ce chapitre, les architectures parallèles ont été décrites d'une manière générale. Les quatre catégories d'architectures parallèles ont été introduites et leurs principes de fonctionnement ont été expliqués. Nous avons décrit aussi les types de la quatrième catégorie de machine parallèle MIMD selon le type de mémoire : partagé, distribué et hybride. L'étude des ces architectures nous a permis de montrer l'importance de la hiérarchie mémoire et la mémoire cache ainsi que leurs fonctionnalités. La configuration et les performances de la mémoire cache intégrée sont à prendre en considération au moment du choix d'un processeur car elles détermineront les performances générales d'un ordinateur.

Ensuite, les différents types d'architectures à mémoire partagée ont été introduits. Nous avons également vu que la tendance architecturale des nouveaux processeurs réside dans l'augmentation du nombre de cœurs. On a présenté les différentes fonctionnalités de deux exemples d'architectures multi-cœurs (Intel Xeon E5 et AMD Opteron 6172). Après les architectures multi-cœurs, les fabricants ont introduit des processeurs many-cœurs avec plusieurs centaines de cœurs, voire des milliers. Deux exemples de processeurs many-cœurs ont été discutés en détail dans la dernière partie de chapitre : le processeur Intel Xeon Phi et les proesseurs graphique (GPU).

Dans le deuxième chapitre nous allons introduire le modèle polyédrique et on va discuter les différents techniques de transformation et d'optimisation de code.

Chapitre 2

Modèle Polyédrique

Sommaire

2.1 Introduction

Le modèle polyédrique [17] est un environnement puissant pour la parallélisation et l'optimisation automatique. Il est basé principalement sur une représentation algébrique des programmes permettant la construction et la recherche des séquences complexes d'optimisation. Ce modèle permet de représenter des boucles imbriquées (dites nid de boucles) sous la forme de polyèdres. A l'intérieur de ces polyèdres les instances des instructions sont représentées par des points à coordonnées entières. Dans le modèle polyédrique, des transformations affines sur les codes (particulièrement sur les nids de boucles) peuvent être appliquées automatiquement afin d'améliorer les performances par la parallélisation ainsi que l'amélioration de la localité des données.
L'utilisation de plateforme de compilation dans le modèle polyédrique se fait en quatre étapes : La première étape consiste à représenter le code original dans le modèle polyédrique en utilisant des outils tel que Clan [21, 20] ou Pet [103, 104]. La deuxième étape est l'analyse de dépendances de données. Cette opération peut être faite en utilisant l'outil Candl [25, 105]. Ensuite, dans la troisième étape en fait l'appelle à un optimiseur automatique basé sur le modèle polyédrique comme Pluto [34, 33] pour paralléliser et optimiser le code polyédrique. Finalement, la dernière étape est la génération du code à partir la représentation polyédrique en utilisant des outils comme Cloog [20, 32] ou Isl Codegen [122]. Dans ce chapitre nous avons introduit des notations et des définitions algébriques, ensuite nous avons détaillé les axes principaux du modèle polyédriques : Les nids de boucles, la transformation dans ce modèle, l'analyse de dépendances, les outils polyédriques, et enfin nous avons montré les différents techniques de transformations.

2.2 Définitions et notations

Définition 1 (Linéarité) Une fonction k-dimensionnellle est dite Linéaire ssi on peut l'exprimer sous la forme suivante [34] :

$$f(\vec{v}) = \mathrm{M}_f \cdot \vec{v}$$

telque : $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ et $\mathrm{M}_f \in \mathbb{R}^{k \times d}$ est une matrice de k lignes et d colonne.

Définition 2 (Fonction Affine) Une fonction k-dimensionnellle est dite Affine ssi on peut l'exprimer sous la forme suivante [34] :

$$f(\vec{v}) = \mathrm{M}_f \cdot \vec{v} + f_0$$

telque : $f_0 \in \mathbb{R}^k$

Définition 3 (Sous-espace Affine) Un ensemble de vecteurs est un sous-espace affine d'un espace vectoriel ssi il est fermé sous une combinaison affine, c'est à dire, si $\vec{x}$ et $\vec{y}$ sont dans l'espace, tous les points situés sur la ligne joignant $\vec{x}$ et $\vec{y}$ appartiennent à cet espace [34].

Définition 4 (Hyperplan Affine) Un hyperplan affine est un sous-espace $(n-1)$-dimensionnel affine d'un espace n-dimensionnel affine [34]. Pour $\vec{c} \in \mathrm{K}^n$ avec $\vec{c} \neq \vec{0}$ et un scalaire $b \in \mathrm{K}$.
L'hyperplan affine est l'ensemble de tous les vecteurs $\vec{x} \in \mathrm{K}^n$ tel que : $\vec{c} \cdot \vec{x} = b$. Il généralise la notion de plans : par exemple, un point, une ligne, un plan sont des hyperplans dans les espaces à 1, 2 et 3 dimensions respectivement.

Définition 5 (demi-espace Affine) Un hyperplan divise l'espace en deux demi-espaces H1 et H2, de sorte que :

$$H1 = \{\vec{x} \in K^n / \vec{c} \cdot \vec{x} \leq b\}$$

et

$$H2 = \{\vec{x} \in K^n / \vec{c} \cdot \vec{x} \geq b\}$$

telque : $\vec{c} \in K^n$, $\vec{c} \neq \vec{0}$, $b \in K$.

Définition 6 (Polyèdre) Un ensemble $S \in K_m$ est un polyèdre s'il existe un système d'un nombre fini d'inégalités affines $A \cdot \vec{x} \leq b$ tel que

$$P = \{\vec{x} \in K_m / A \cdot \vec{x} \leq b\}$$

De manière équivalente, c'est l'intersection finie de demi-espaces [34].

2.3 Nid de boucles

Définition 7 (Boucle) Une boucle dans un programme est un bloc de code qui est exécuté plusieurs fois. Ce bloc est de la forme :

```
for i= L, U, P do
Begin
   {
     Ins_1;
     Ins_2;
     ...
     Ins_N;
   }
end
```

Où L, U et P sont des expressions arithmétiques entières, tel que L est la borne inférieure de la boucle, U est la borne supérieure de la boucle et S est le pas de la boucle. Dans le cas où P = 1 on parle de la boucle régulière. La variable entière i est l'indice de la boucle. L'ensemble ordonné des instructions $S = Ins_1, Ins_2, \cdots Ins_N$ compose le corps de la boucle. Chaque répétition de l'ensemble d'instructions S est une itération.

Définition 8 (Nid de Boucles) Un nid de boucles est un ensemble fini de N boucles emboîtées contenant des instructions exécutables tel que N > 1. Cette structure est de la forme :

```
for i1= L1,U1,P1  do
 for i2= L2,U2,P2  do
 ...
  for iN=LN,UN,PN do
    Begin
     {
       Ins_1;
       Ins_2;
       ...
       Ins_M;
     }
    end
```

N est la dimension du nid de boucles. La boucle B_1 (ayant l'indice i_1) est la boucle la plus externe et la boucle B_N est la boucle la plus interne. La profondeur d'une boucle B_j est le nombre de boucles externes qui l'entourent. Un nid de boucles est dit parfait si toutes les instructions sont à la profondeur maximale (la boucle la plus interne).

Définition 9 (Nid de Boucles Affine) est une séquence de boucles arbitrairement imbriquées avec des bornes de boucle et des accès aux tableaux qui sont des fonctions affines de paramètres de programme et des indices de boucles externes . Les paramètres de programme sont des constantes symboliques qui apparaissent dans les limites de boucle ou des fonctions d'accès. Ils représentent très souvent la taille du problème. Par exemple dans le code suivant N et b sont les paramètres du programme.

```
1 for (i=0; i<N; i++)
2   for (j=0; j<N; j++)
3     S1: x[i] = x[i]+ b *A[i][j]*y[j];
```

2.4 Modèle Polyédrique

Le modèle polyédrique est une représentation sémantique, algébrique qui combine la puissance d'analyse et la flexibilité de la transformation pour concevoir des heuristiques d'optimisation sophistiquées. Le modèle polyédrique est plus proche de l'exécution du programme que les représentations syntaxiques et qui fonctionne sur des instructions individuelles ou des instances d'instructions dans les itérations. Pour chaque instance, l'algorithme optimisé calcule un mappage qui déterminera à quel moment (mappage temporel ou ordonnancement) et/ou sur quel processeur (mappage spatial ou emplacement) cette instance doit être exécutée. Il a été la base pour des initiatives majeures dans le domaine d'optimisation et de parallélisation automatique des programmes. Ce modèle est basé sur les systèmes d'équations récurrentes affines, définies sur des domaines polyédriques. Une équation récurrente est une fonction qui détermine comment une valeur d'une variable est définie (récursivement), en un point de l'espace des indices, en fonction de sa valeur en d'autres points [9, 25, 16, 118]. Dans ce modèle, des transformations de programme peuvent être appliquées et le code peut être généré à partir du modèle. La puissance du modèle réside dans sa capacité à exprimer une séquence potentiellement complexe de transformations de boucles, telles que la distribution, fusion de boucles, Skewing et Tiling, etc.

Définition 10 (vecteur d'itération) Les coordonnées d'une instance d'instruction sont définies par un vecteur $\vec{s} \in \mathrm{K}^n$ avec n la profondeur du nid de la boucle. Un vecteur d'itération d'une instruction est un vecteur constitué des valeurs de tous ses itérateurs de boucle englobantes. Ainsi, une occurrence d'une instruction à la profondeur de boucle n peut être représentée par un vecteur d'itération de taille n. Chaque instruction est exécutée une fois pour chaque valeur possible du vecteur d'itération. Ainsi, le vecteur d'itération représente chaque instance dynamique d'une instruction. Ils peuvent être exprimés comme suit :

$$\vec{s} = (i_1, i_2, i_3, \cdot\cdot, i_n)^{\mathrm{T}}$$

2.4.1 Domaine d'itération

Définition 11 (Domaine d'itération) Un domaine d'itération est l'ensemble des points entiers correspondant aux instances réelles d'une instruction.

Le domaine d'itération est donc une façon compacte de représenter toutes les instances d'une instruction donnée est de considérer l'ensemble de toutes les valeurs possibles de son vecteur d'itération. Considérons le segment de code (nid de boucles) suivant :

```
1 for (i=0;i<N;i++)
2 {
3   for (j=0;j<i;j++)
4     S(i,j);
5   for (j=i+2;j<N;j++)
6     T(i,j);
7 }
```

Les domaines d'itérations pour les deux instructions S et T sont les suivants :

$$D_S(N) = \{i, j / 0 \leq i < N \wedge 0 \leq j < i\}$$

$$D_T(N) = \{i, j / 0 \leq i < N \wedge i+2 \leq j < N\}$$

La représentation des deux polyèdres $D_S(N)$ et $D_T(N)$ sous forme de matrices est comme suit :

$$\begin{cases} 0 \leq i \\ i < N \\ 0 \leq j \\ j < i \end{cases} = \begin{cases} i \geq 0 \\ N - i - 1 \geq 0 \\ j \geq 0 \\ i - j - 1 \geq 0 \end{cases} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \cdot (N) + \begin{bmatrix} 0 \\ -1 \\ 0 \\ -1 \end{bmatrix} \geq 0$$

$$\begin{cases} 0 \leq i \\ i < N \\ i+2 \leq j \\ j < N \end{cases} = \begin{cases} i \geq 0 \\ N - i - 1 \geq 0 \\ j - i - 2 \geq 0 \\ N - j - 1 \geq 0 \end{cases} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \cdot (N) + \begin{bmatrix} 0 \\ -1 \\ -2 \\ -1 \end{bmatrix} \geq 0$$

La figure 2.1 montre la représentation de notre nid de boucles dans le modèle polyédrique pour N=6.

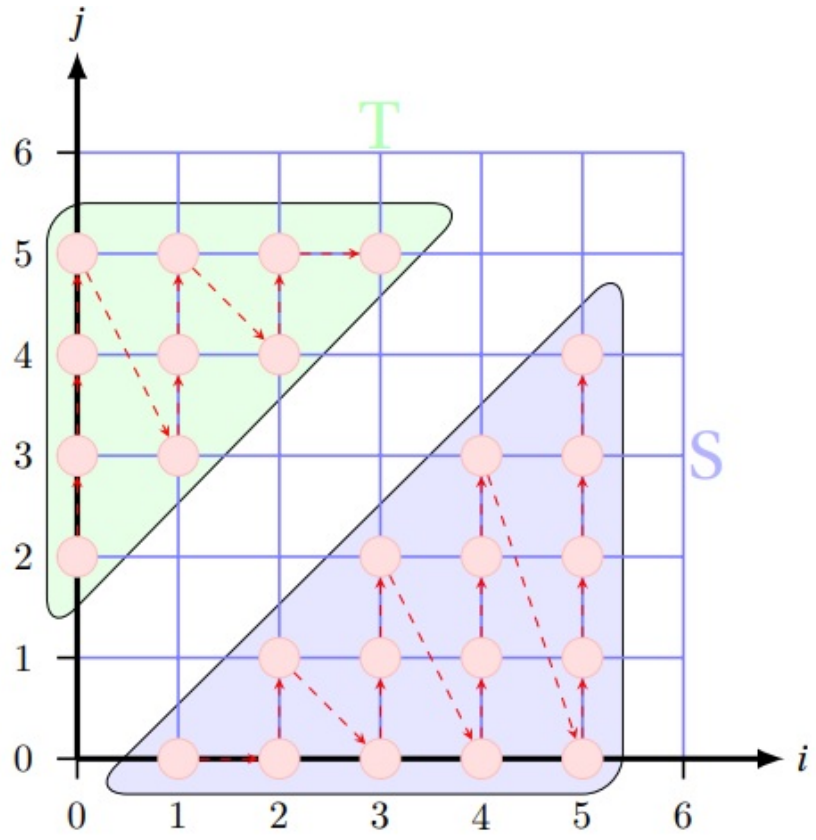


FIGURE 2.1 – Représentation de nid de boucles dans le modèle polyédrique.

2.4.2 Static Control Part

Un SCoP est défini comme un ensemble maximal d'instructions consécutives, où les bornes de boucle et les conditions sont des fonctions affines sur le domaine d'itération. Le domaine d'itération de ces boucles peut toujours être spécifié grâce à un ensemble d'inégalités linéaires définissant un polyèdre. Le terme polyèdre sera utilisé pour désigner un ensemble de points dans un

espace vectoriel Z_n borné par des inégalités affines :

$$D = \{x / x \in Z_n; Ax + a \geq 0\}$$

où x est le vecteur d'itération (le vecteur des valeurs des indices des boucles), A est une matrice constante et a est un vecteur constant, éventuellement paramétrique. Le domaine d'itération est un sous-ensemble de l'espace d'itération possible : $D \subset Z_n$. La correspondance entre le contrôle sur l'exécution de l'instruction S et le domaine polyédrique est comme suit :

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= N; j++)
    if (i <= N+2-j)
      S(i,j);
```

$$D_S(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} | \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} -1 \\ N \\ -1 \\ N \\ N+2 \end{bmatrix} \geq 0 \right.$$

Définition 12 (ordre lexicographique) On dit qu'un vecteur $a \in Z_n$ est lexicographiquement plus petit que $b \in Z_n$ si pour la première position i dans laquelle a et b diffèrent, on a :

$$(a \prec b) \equiv \exists i, 1 \leq i \leq n, (a_1 ... a_i) = (b_1 ... b_i) \wedge a_{i+1} < b_{i+1}$$

2.4.3 Fonction de scattering

Le domaine d'itération ne contient aucune information d'ordre : il ne décrit que l'ensemble des instances d'instructions mais pas l'ordre dans lequel elles doivent être exécutées les unes par rapport aux autres. Dans le passé, l'ordre lexicographique du domaine d'itération a été considéré, ce qui n'est pas suffisant (par exemple lors de la transformation du code). Si aucune information d'ordre n'est donnée, cela signifie que les instances d'instructions peuvent être exécutées dans n'importe quel ordre (ceci est utile, par exemple, pour spécifier le parallélisme), mais certaines instances d'instructions peuvent dépendre d'autres et il peut être important d'imposer un ordre donné. Par conséquent, des informations supplémentaires sont nécessaires. Nous appelons " scattering " tout type d'information d'ordre dans le modèle polyédrique. Il existe en effet de nombreux types d'ordre, comme l'allocation, l'ordonnancement, le découpage, etc. Néanmoins, ils sont tous exprimés de la même manière, en utilisant des tampons logiques qui peuvent avoir diverses sémantiques [34]. Un exemple très utile de fonctions de scattering multidimensionnelles est l'ordonnancement (scheduling) du programme original. La méthode pour le calculer est assez simple. L'idée est de construire un arbre abstrait de syntaxe du programme et de lire le schedule pour chaque énoncé. Par exemple, considérons l'implémentation suivante :

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
 {
 S1: a = 0 ;
   for (k = 0; k < j-1; k++)
   {
    S2:   C[k][j] += A[k][i] * B[i][j] ;
    S3:   a += B[k][j] * A[k][i] ;
    }
 S4:   C[i][j] += A[i][i] * B[i][j] + a ;
 }
```

$$\begin{cases} \theta_{S_1}(i,j) = (0,i,0,j,0) \\ \theta_{S_2}(i,j,k) = (0,i,0,j,1,k,0) \\ \theta_{S_3}(i,j,k) = (0,i,0,j,1,k,1) \\ \theta_{S_4}(i,j) = (0,i,0,j,2) \end{cases}$$

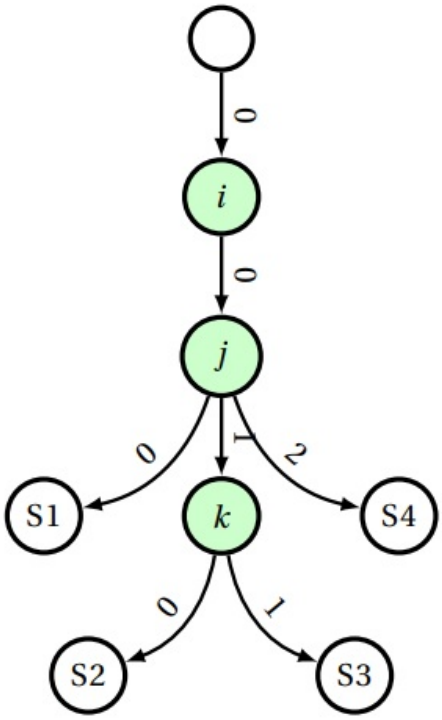


FIGURE 2.2 – Arbre abstrait de syntaxe.

L'arbre abstrait de syntaxe est donné par la figure 2.2 . Il donne directement les fonctions de scattering (schedules) pour toutes les instructions du programme. Les dimensions impaires du schedule représentent l'ordre textuel, et les dimensions paires expriment la relation avec les indices d'itérations. Ces schedules dépendent des indices d'itérations et donnent pour chaque instance de chaque instruction une date d'exécution unique. Le scattering d'une instruction S, notée par θ_S, peut être défini comme suit :

$$\theta_S(\vec{x}) = A \cdot \vec{x}$$

Où : $\vec{x}$ est le vecteur d'itération ; A est la matrice de scattering.

Définition 13 (la fonction de scheduling) la fonction de scheduling d'une instruction S, également connue sous le nom de schedule de S, est une fonction qui ordonne chaque instance dynamique de S à une date logique, exprimant l'ordre d'exécution entre les instructions :

$$\forall \vec{x} \in D_S, \theta_S(\vec{x}) = T \cdot \vec{x} + \vec{t}$$

A titre d'exemple, considérons notre exemple précédent, en modifiant le schedule de S_3 de cette façon : $\theta_S(i,j,k) = (j+1, 2i+j, 3k+1)$

$$T_S \cdot \begin{pmatrix} i \\ j \\ k \\ 1 \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 3 & 1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ 1 \end{pmatrix}$$

Définition 14 (le schedule affine) Étant donné une instruction S, un schedule p-dimensionnel affine θ_S est une forme affine sur l'indice de boucle externe $\vec{x}_S$ et le paramètre $\vec{n}$. Il est écrit comme suit :

$$\theta_S(\vec{x}) = T_S \cdot \begin{pmatrix} \vec{x}_s \\ \vec{n} \\ 1 \end{pmatrix} : T_S \in K^{p \cdot dim(\vec{x}_s) + dim(\vec{n}) + 1}$$

2.4.4 Analyse de dépendance de données

Définition 15 (Dépendance de données) On dit qu'il y a une dépendance des données chaque fois que deux instructions $S_i(\vec{x}_i)$ et $S_j(\vec{x}_j)$ accèdent au même emplacement mémoire, et qu'au moins un accès est un **Write**.

Les tests de dépendance de données sont requis pour toute forme de détection de la parallélisation automatique. Les relations de dépendance de données sont utilisées pour déterminer quand deux opérations, instructions ou itérations d'une boucle peuvent être exécutées en parallèle [74, 59]. Par exemple, dans le code :

```
1    I1 : A = B + C ;
2    I2 : D = A + 1 ;
3    I3 : E = A * 2 ;
```

Les deux instructions I1 et I2 ne peuvent pas être exécutées au même temps puisque I2 utilise la valeur de A qui est calculée par l'instruction I1, C'est ce qu'on appelle **dépendance de flot** puisque la valeur des données passe de I1 à I2, et il est noté (I1 σ^f I2), I3 dépend aussi de I1, (I1 σ^f I3), donc I1 doit être exécutée avant I2 et I3 à la fois. Les relations de dépendance de données sont souvent représentées dans un graphe de dépendance de données, avec des arcs représentant les relations, comme il est montré dans la figure 2.3.

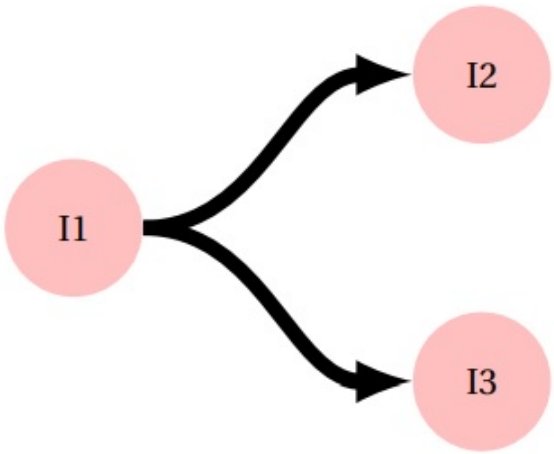


FIGURE 2.3 – Graphe de dépendance de données

Notez que I2 et I3 ne sont pas connectés par des arcs de dépendance de données et peuvent donc être exécuté en parallèle si deux processeurs sont disponibles. Il existe deux autres types de dépendance des données. Dans le segment de programme :

```
1    I1 : A = B + C ;
2    I2 : B = D * 3 ;
```

I1 utilise la valeur de B avant que I2 lui attribue une nouvelle valeur. Puisque I1 utilise l'ancienne valeur de B, elle doit être exécuté avant I2, c'est ce qu'on appelle **anti dépendance**. Ce type de dépendance est noté I1 σ^a I2.

Le troisième type de dépendance est montré dans le segment de programme ci-dessous :

```
1     I1 : A = B + C ;
2     I2 : D = A + 1 ;
3     I3 : A = E + F ;
```

Ici I3 assigne une nouvelle valeur à A après que celle ci soit modifiée par I1. Si I1 est exécuté après I3, alors A contiendra la mauvaise valeur après ce segment de programme. Ainsi, I1 doit précéder I3 ; ceci est appelé **dépendance de sortie**, noté I1 σ^o I3 . Le flux de contrôle doit également être pris en compte lors de la construction de relations de dépendance aux données. Par exemple, dans le segment de programme :

```
I1:  A = B + C
     if ( X >= 0 ) then
I2:     A = 0
     else
I3:     D = A
     end if
```

Dans ce morceau de code, on voit qu'il y a une dépendance de sortie entre I1 et I2 (I1 σ^o I2) et une dépendance de flot entre I1 et I3 (I1 σ^f I3), mais il n'y a aucune dépendance entre I2 et I3 puisque elles sont sur des branches différentes de la même instruction if, la valeur de A utilisée dans I3 ne proviendra jamais de I2. Puisque le flux d'exécution réel d'un programme n'est pas connu avant l'exécution, une relation de dépendance de données n'implique pas toujours une communication de données ou un conflit de mémoire. Par exemple, dans ce segment de programme :

```
I1:  A = B + C
C1:  if ( X >= 0 ) then
I2:     A = A + 1
     end if
I3:  D = A * 2
```

Les relations de dépendance de données I1 σ^f I2 et I2 σ^f I3 seront tous les deux calculés par le compilateur, même si I3 prendra en fait la valeur de A parmi I1 ou I2, selon la valeur de X. Il existe un nombre important de compilateurs qui utilisent également le concept de dépendance aux instructions sous le contrôle du if. Ceci est appelé **dépendance de controle** et il est noté σ^c . Dans le segment de programme ci-dessus, par exemple, la dépendance de controle I1 σ^c I2 est maintenu. Les relations de dépendance de controle sont souvent ajoutées au graphe de dépendance de données afin de trouver quelles instructions peuvent être réorganisées dans le programme.

2.4.5 Dépendance de données dans les boucles

A l'intérieur des boucles, nous nous intéressons aux relations de dépendance des données entre les instructions et aux relations de dépendances entre les instances d'instructions. Nous distinguons les différentes instances d'exécution d'une instruction en exposant l'étiquette de l'instruction avec les itérations de la boucle. Par exemple, dans la boucle :

```
for I=1 to 3 do
I1:  A(I) = B(I)
  for J=1 to 2 do
I2:  C(I,J) = A(I) + B(J)
  end for
end for
```

l'instruction I1 est exécutée trois fois : $I1^1$,$I1^2$, $I1^3$ et I2 est exécuté six fois : $I2^{1,1}$,$I2^{1,2}$, $I2^{2,1}$, $I2^{2,2}$,$I2^{3,1}$, $I2^{3,2}$ (nous mettons le numéro d'itération de la boucle externe en premier). Nous pouvons également dessiner les itérations comme des points avec des coordonnées cartésiennes, comme il est montré dans la figure 2.4.

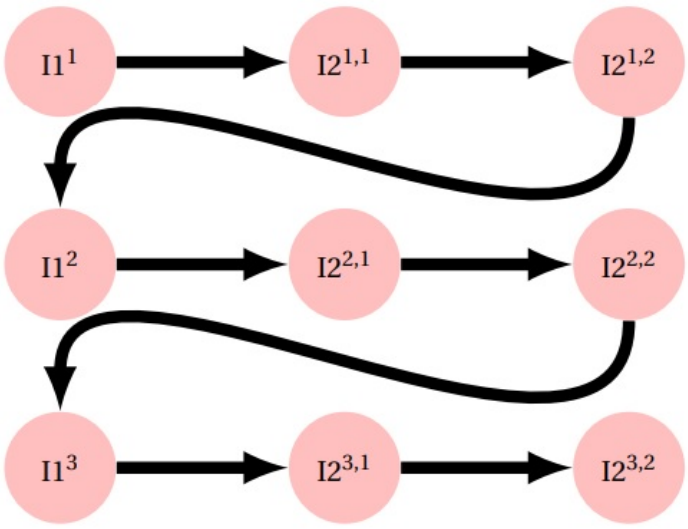


FIGURE 2.4 – Diagramme de dépendances de données d'une boucle.

Ce diagramme illustre l'espace d'itération; les flèches indiquent l'ordre dans lequel les instances des instructions sont exécutées. Pour trouver les relations de dépendance de données dans les boucles, les tableaux et les indices sont examinés. Dans la boucle :

```
1     for I=1 to 3 do
2     I1:  A(I) = B(I)+ C(I)
3     I2:  D(I) = A(I)
4     end for
```

la relation I1 σ^f I2 est maintenu, puisque pour toute itération i, I1^i assignera A(i) et I2^i utilisera A(i) sur la même itération de la boucle. Puisque la dépendance reste dans la même itération de la boucle, cette relation est notée I1 $\sigma_=$ I2

Dans la boucle similaire suivante :

```
1     for I=1 to 3 do
2     I1:  A(I) = B(I)+ C(I)
3     I2:  D(I) = A(I-1)
4     end for
```

la relation I1 σ^f I2 est toujours valable, mais pour toute itération i, I2^i utilisera un élément de A affecté à l'itération précédente de la boucle par I1^{i-1}, (sauf I2^i, qui utilise une ancienne valeur de A(1)). Puisque la dépendance passe de l'itération $i-1$ à l'itération i , nous disons que I1 $\sigma_<$ I2 (la relation est $<$ parce que $i-1<i$).

2.5 Transformation polyédrique

Les transformations de programme dans le modèle polyédrique peuvent être spécifiées par des fonctions de scheduling bien choisies. Elle modifient les polyèdres sources en des polyèdres cibles contenant les mêmes points entiers mais dans un nouveau système de coordonnées, donc avec un ordre lexicographique différent.

Définition 11 (Transformation) Une transformation planifie une date logique de l'instance d'instruction à une date dans l'espace transformé, par rapport à la sémantique du programme

d'origine. Les transformations sont généralement appliquées pour améliorer la localité des données, réduire le code de contrôle et exposer ou améliorer le degré de parallélisme. La nouvelle planification Φ_S pourrait exprimer un nouvel ordre d'exécution sur l'instruction S.

Définition 11 (Transformation affine unidimensionnelle) Une transformation affine unidimensionnelle pour l'instruction S est une fonction affine définie par :

$$\Phi_S(\vec{i}) = (c_1^S c_2^S .... c_{m_S}^S) \cdot (\vec{i}_s) + c_0^S$$

$$\Phi_S(\vec{i}) = (c_1^S c_2^S .... c_{m_S}^S) \cdot \begin{pmatrix} \vec{i}_S \\ 1 \end{pmatrix}$$

Où : c_1, c_2,..., $c_{m_S} \in Z$, $\vec{i} \in Z^{m_S}$.

Ainsi, une transformation affine unidimensionnelle pour chaque instruction peut être interprétée comme un hyperplan de partition avec un vecteur normal (c_1, c_2,..., c_{m_S}).
Une transformation affine multidimensionnelle peut être représentée comme une séquence de Φ pour chaque instruction. Nous utilisons un exposant pour désigner l'hyperplan pour chaque niveau. Φ_k^S représente l'hyperplan au niveau k pour l'instruction S. Si $1 \leq k \leq d$, tous les Φ_k^S peuvent être représentés par une seule fonction affine d-dimensionnelle donnée par :

$$\tau_S(\vec{i}_S) = M_S \cdot \vec{i}_S + \vec{t}_S$$

Où : $M_S \in \mathbb{Z}^{d \cdot m_S}$, $\vec{t}_S \in \mathbb{Z}^d$

$$\tau_S(\vec{i}) = \begin{pmatrix} \Phi_1^S(\vec{i}) \\ \Phi_1^S(\vec{i}) \\ \vdots \\ \Phi_d^S(\vec{i}) \end{pmatrix} = \begin{pmatrix} c_{11}^S & c_{12}^S & \dots & c_{1m_S}^S \\ c_{21}^S & c_{22}^S & \dots & c_{2m_S}^S \\ \vdots & \vdots & \vdots & \vdots \\ c_{d1}^S & c_{d2}^S & \dots & c_{dm_S}^S \end{pmatrix} \cdot \vec{i}_S + \begin{pmatrix} c_{10}^S \\ c_{20}^S \\ \vdots \\ c_{d0}^S \end{pmatrix}$$

Définition 12 (Légalité d'une transformation) Une transformation est légale si l'exécution du programme modifié produit le même résultat que l'exécution du programme initial. Autrement dit, le nouvel ordre d'exécution des itérations et des instructions doit conserver toutes les dépendances initiales.

Définition 13 (Transformation Unimodulaire) Une transformation de boucle unimodulaire est un changement de base dans ($\mathbb{Z}^{n_S}$) appliqué sur le domaine d'itération D_S. Cette transformation assure que si le domaine d'itération original est un polyèdre convexe, alors le domaine d'itération transformé est un polyèdre convexe. Les calculs sont décrits à travers un nouveau vecteur d'itération $\vec{I'}$ tel que $\vec{I'} = U \cdot \vec{I}$ où U est une matrice de déterminant 1 ou -1. Les transformations de boucle unimodulaires sont des combinaisons d'échange de boucle, d'inversion de boucle et de skewing de boucle. Une transformation unimodulaire U est valide si et seulement si $0 \prec U_d$ pour chaque vecteur de distance non nulle d.

Exemple : Considérons le segment de code suivant :

```
1 for (i = 1; i <= N; i++)
2   for (j = 1; j <= i; j++)
3   S: a[i] = a[i] + b[i][j];
```

Le domaine d'itération pour l'instruction S est :

$$D_S(N) = \{(i, j) / 1 \leq i \leq N \wedge 1 \leq j \leq i\}$$

La transformation inverse de boucle :

$$\tau = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Le domaine d'itération du nid de boucles après la transformation :

$$D(N) = \{(J,I) / 1 \leq I \leq N \wedge 1 \leq J \leq I\}$$

Donc : $J \leq I \leq N$ et $1 \leq J \leq N$
Alors le segment de code devient :

```
1 for (J = 1; J <= N; J++)
2   for (I = J; I <= N; I++)
3   S: a[J] = a[J]+b[J][I];
```

2.6 Outils polyédriques

2.6.1 OpenScop

OpenScop [18] est une spécification ouverte qui définit un format de fichier et un ensemble de structures de données pour représenter le SCoP (Static Control Part), c'est-à-dire une partie de programme qui peut être représentée dans le modèle polyédrique. L'objectif d'OpenScop est de fournir une interface commune aux différents outils de compilation polyédriques afin de simplifier leur interaction. OpenScop s'appuie sur les anciens formats de structure de données et de fichiers polyédriques courants, tels que les structures de données .cloog et CLooG [51], pour fournir un format unique et extensible à la plupart des outils de compilation polyédriques. Il est composé de deux parties (voir figure 2.5).

La première partie, appelée partie centrale, est consacrée à la représentation polyédrique d'un SCoP. Il contient ce qui est strictement nécessaire pour construire un framework source-à-source complet dans le modèle polyédrique et pour produire un code sémantiquement équivalent pour le SCoP, de l'analyse à la génération de code. La deuxième partie du format est la partie extension, contient des extensions pour fournir des informations supplémentaires à certains outils.
Comme le montre la figure 2.5 , la partie centrale est composée du ***context*** et des ***Statements*** :

— ***context*** représente les informations globales du SCoP. Il se compose du langage cible, des contraintes globales sur les paramètres et éventuellement des noms de paramètres qui peuvent être nécessaires pour le processus de génération de code ;

— ***Statements*** représente les informations sur les instructions.

Nb_statements est le nombre d'instructions dans le SCoP, c'est-à-dire le nombre d'éléments ***Statement*** dans la ***Statement_List***. ***Statement*** représente l'information sur une instruction donnée. A chaque instruction est associée une liste de relations et, facultativement, une liste d'extensions d'instructions. La liste des relations peut inclure un domaine d'itération, une relation de diffusion et plusieurs relations d'accès. Les extensions d'instructions sont des informations facultatives. Ils sont commencés par un nombre entier qui spécifie le nombre d'extensions fournies. ***Extension_List*** représente la partie extension et il peut contenir un nombre arbitraire d'informations génériques.

2.6.2 Clan (Chunky Loop Analyzer)

Clan [21, 20] est une bibliothèque gratuite qui traduit certaines parties de programmes de haut niveau écrits en C, C++, C# ou Java en une représentation polyédrique, à savoir OpenScop . Cette représentation peut être manipulée par d'autres outils pour réaliser des analyses complexes ou

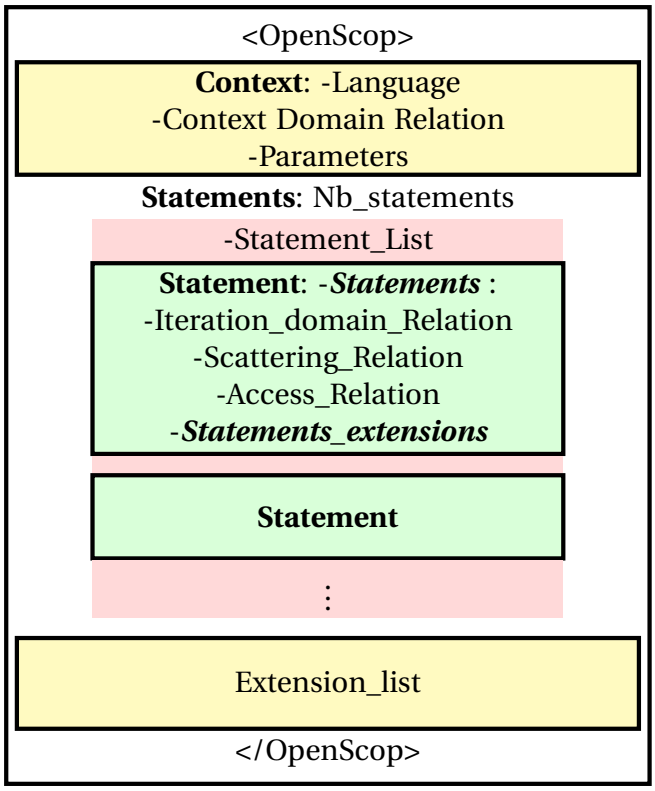


FIGURE 2.5 – La structure de données OpenScop.

des restructurations de programme (pour l'optimisation, la parallélisation ou tout autre type de manipulation).
Il a été créé pour éviter l'écriture fastidieuse et sujette aux erreurs des fichiers d'entrée pour les outils polyédriques (tels que CLooG, LeTSeE, Candl, etc.). En utilisant Clan, l'utilisateur doit traiter les codes source basés sur la grammaire C seulement (comme C, C++, C# ou Java).
Clan fait partie du projet Chunky, un outil de recherche pour l'amélioration de la localité des données. Il est conçu pour être le front-end de tout optimiseur et/ou paralléliseur automatique de source à source. Le format de sortie OpenScop a été choisi pour être indépendant de la bibliothèque polyédrique, donc Clan peut facilement intégrer n'importe quelle structure de compilation polyédrique. Clan a été intégré avec succès aux compilateurs de haut niveau de PoCC [124] , Pluto, etc.

2.6.3 Clay (Chunky Loop Alteration wizardrY)

Clay [19] est un logiciel gratuit et une bibliothèque dédiée à l'optimisation semi-automatique utilisant le modèle polyédrique. Il prend en entrée un programme de haut niveau ou sa représentation polyédrique et le transformer selon un script de transformation. Des primitives de transformations de boucles classiques sont fournies (fusion, fission, skewing, strip-mining, tiling, etc.).
Clay est capable de vérifier la légalité de la séquence complète de transformation et de suggérer des corrections à l'utilisateur si la sémantique originale n'est pas conservée.

2.6.4 CLooG (Chunky Loop Generator)

CLooG [20, 32] est un logiciel libre et une librairie pour générer du code pour scanner des Z-polyèdres. C'est-à-dire qu'il trouve un code (par exemple dans C, FORTRAN ...) qui atteint chaque point d'intégrale d'un ou plusieurs polyèdres paramétrés. CLooG a été écrit à l'origine pour résoudre le problème de génération de code pour optimiser les compilateurs basés sur le modèle polyédrique. Néanmoins, il est utilisé maintenant dans divers domaines, par exemple construire

des automates de contrôle pour la synthèse de haut niveau ou pour trouver la meilleure approximation polynomiale d'une fonction.
CLooG peut nous aider dans toutes les situations où l'analyse des polyèdres est importante. Alors que l'utilisateur a un contrôle total sur la qualité du code généré, CLooG est conçu pour éviter les coûts de contrôle et pour produire un code très efficace.
CLooG est largement utilisé (y compris par les compilateurs GCC et LLVM), disséminé (il est installé par défaut par les principales distributions Linux) et considéré comme l'état de l'art dans la génération de code polyédrique [51].

2.6.5 Candl (Chunky ANalyzer for Dependencies in Loops)

Candl [22] est un logiciel libre et une bibliothèque consacrée au calcul des dépendances de données. À partir d'une représentation polyédrique d'une partie de contrôle statique d'un programme, il est capable de calculer exactement l'ensemble des instances d'instructions dans les relations de dépendance. Par conséquent, sa sortie est utile pour construire des transformations de programme en respectant la sémantique du programme original.
Cet outil a été conçu pour être robuste et précis. Il implémente certaines techniques habituelles pour la suppression des dépendances de données, telles que la privatisation ou l'expansion de tableaux, etc.

2.6.6 Clint (Chunky Loop INTeraction)

Clint [129] est une interface de manipulation directe conçue pour (1) aider les programmeurs à paralléliser des programmes de calcul haute performance (2) faciliter l'exploration des transformations possibles; et (3) garantir l'exactitude du code final. Clint tire parti de la nature géométrique du modèle polyédrique en présentant des instructions de code, leurs instances et leurs dépendances dans une visualisation par points de dispersion de domaines d'itération similaires à ceux utilisés dans la littérature sur la compilation polyédrique.
En rendant la visualisation interactive, elle réduit les tâches d'extraction de parallélisme et de transformation de code à la reconnaissance de formes visuelles et aux manipulations géométriques, permettant ainsi de gérer la complexité du modèle sous-jacent.

2.7 Les techniques de transformation de code

2.7.1 Fusion de boucles

Cette transformation fusionne les boucles adjacentes en une boucle unique. La fusion est utile pour réduire le nombre de boucles et réduire les distances de réutilisation des données. L'ordre des instructions dans la boucle fusionnée est le même qu'avant la fusion. la figure 2.6 illustre un exemple de fusion de boucles.

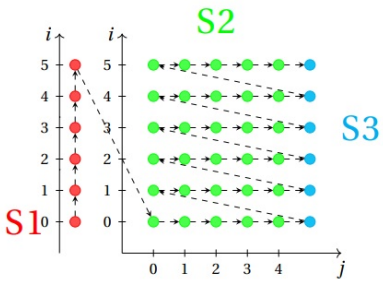
2.7.2 Permutation de boucles

La permutation de boucle échange simplement la position de deux boucles dans un nid de boucles. L'une des principales utilisations est d'améliorer le comportement des accès à un tableau. Il est également connu comme l'échange de boucle. la permutation de boucles est montrée dans la figure 2.7.

2.7.3 Distribution de boucles

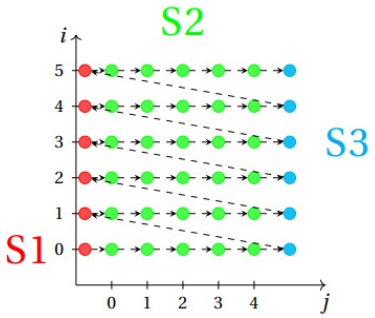
Cette transformation est également appelée fission ou splitting de boucles. Il consiste à diviser le contrôle de boucle sur différentes instructions dans le corps de la boucle. La distribution de boucle peut être appliquée à n'importe quelle boucle tant que les dépendances sont conservées. La figure 2.8 montre un exemple de distribution de boucles.

```
for (i=0; i<n; i++)
 {
    S1(i);
 }
for (i=0; i<n; i++)
 {
    for (j=0; j<n; j++)
     {
       S2(i,j);}
    S3(i);
  }
```



(a) La représentation du code original.

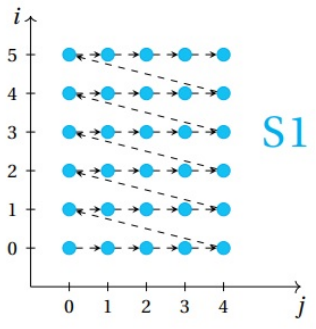
```
for (i=0; i<n; i++)
 {
    S1(i);
    for (j=0; j<n; j++)
     {
       S2(i,j);
       }
    S3(i);
  }
```



(b) La représentation du code transformé.

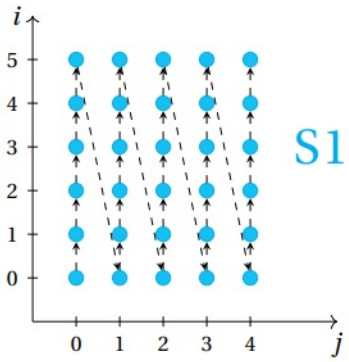
FIGURE 2.6 – Exemple de fusion de boucles (n=6).

```
for (i=0; i<n; i++)
 {
    for (j=0; j<m; j++)
     {
      S1(i,j);
      }
 }
```



(a) La représentation du code original.

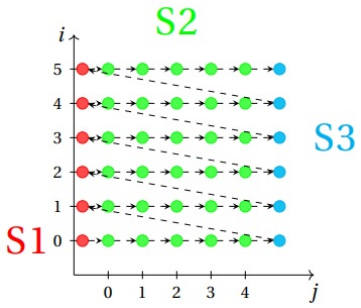
```
for (j=0; j<m; j++)
 {
    for (i=0; i<n; i++)
     {
      S1(i,j);
      }
 }
```



(b) La représentation du code transformé.

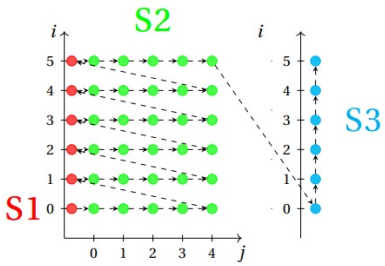
FIGURE 2.7 – Exemple de permutation de boucles (n=6, m=5).

```
for (i=0; i<n; i++)
 {
    S1(i);
    for (j=0; j<m; j++)
     {
      S2(i,j);
      }
    S3(i);
  }
```



(a) La représentation du code original.

```
for (i=0; i<n; i++)
 {
    S1(i);
    for (j=0; j<m; j++)
     {S2(i,j);}
 }
for (i=0; i<n; i++)
 {
    S3(i);
 }
```



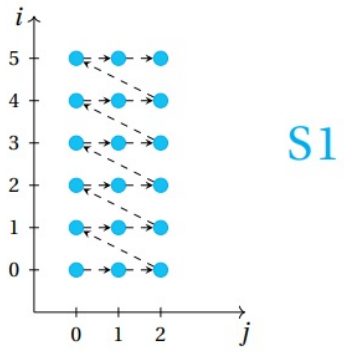
(b) La représentation du code transformé.

FIGURE 2.8 – Exemple de distribution de boucles (n=6, m=5).

2.7.4 Inclinaison de boucles (Skewing)

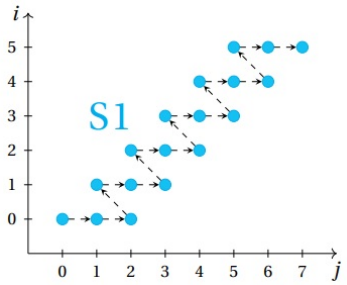
L'inclinaison de boucle consiste à décaler un indice de boucle par une fonction affine d'autres indices de boucle. Il est particulièrement utilisé pour afficher des boucles parallèles en supprimant les dépendances entre itérations pour certaines boucles. La figure 2.9 montre un exemple de Skewing.

```
for (i=0; i<n; i++)
 {
    for (j=0; j<m; j++)
     {
      S1(i,j);
      }
 }
```



(a) La représentation du code original.

```
for (i=0; i<n+m-1; i++)
 {
    for (j=0; j<m+i; j++)
     {
      S1(i,j-i);
      }
  }
```



(b) La représentation du code transformé.

FIGURE 2.9 – Exemple de Skewing (n=6, m=3).

2.7.5 Peeling de boucles

Peeling de boucles consiste à supprimer certaines premières ou dernières itérations d'une boucle dans un code séparé en dehors de la boucle. C'est toujours légal, à condition qu'aucune itération supplémentaire ne soit introduite. Cette transformation est utile pour appliquer un alignement de mémoire initial particulier sur des références de tableau avant la vectorisation en boucle (voir figure 2.10).

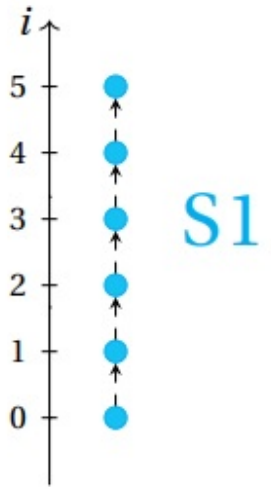
2.7.6 Réorganisation des instructions

Les instructions appartenant au même indice sont réorganisées de la même manière que dans les boucles classiques. Un exemple de réorganisation de boucles est montré dans la figure 2.11.

2.7.7 Strip-mining

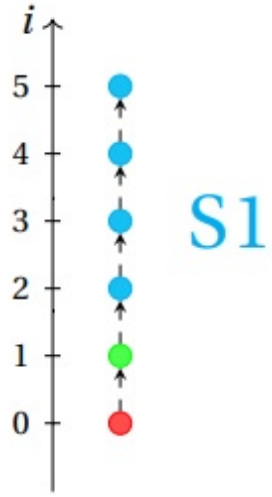
Strip-mining est une méthode permettant d'ajuster la granularité d'une opération en divisant une boucle unique (généralement la boucle la plus interne) en une boucle imbriquée. La boucle interne résultante itère sur des blocs de taille constante de la boucle d'origine, et la nouvelle boucle externe exécute suffisamment de fois la boucle interne pour couvrir toutes les bandes, obtenant ainsi le nombre total d'itérations nécessaire. Le partitionnement de l'espace d'itération de boucle conduit au partitionnement d'un grand tableau en blocs plus petits, ajustant ainsi les éléments de tableau accédés à la taille du cache, améliorant la réutilisation du cache et éliminant les exigences de taille de cache. La figure 2.12 montre un exemple de Strip mining.
Remarque : La fonction floord(a, b) renvoie l'entier inférieur du résultat de la division entre a et b.

```
for (i=0; i<n; i++)
 {
    S1(i);
 }
```



(a) La représentation du code original.

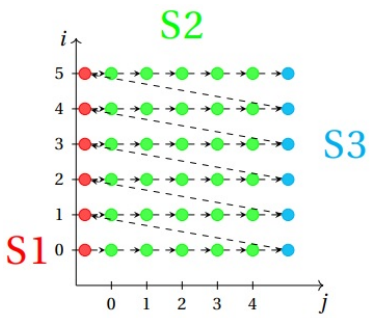
```
S1(0);
S1(1);
for (i=2; i<n; i++)
 {
    S1(i);
  }
```



(b) La représentation du code transformé.

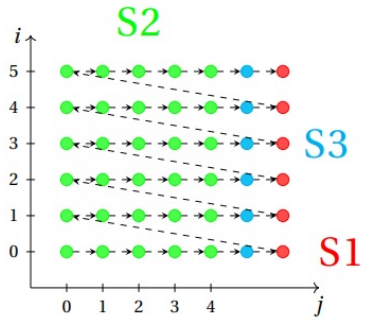
FIGURE 2.10 – Exemple de Peeling de boucles (n=6).

```
for (i=0; i<n; i++)
 {
    S1(i);
    for (j=0; j<m; j++)
     {
      S2(i,j);
      }
    S3(i);
  }
```



(a) La représentation du code original

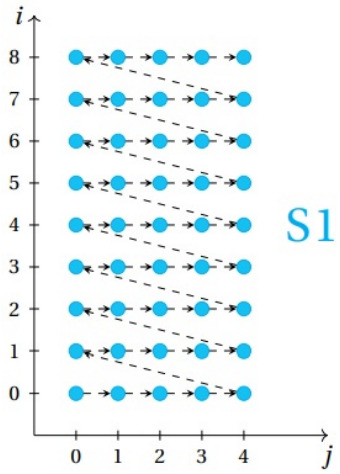
```
for (i=0; i<n; i++)
 {
   for (j=0; j<m; j++)
     {S2(i,j);}
   S3(i);
   S1(i);
 }
```



(b) La représentation du code transformé.

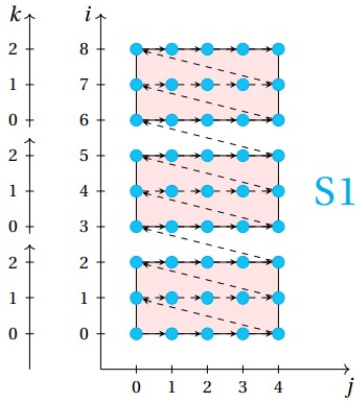
FIGURE 2.11 – Exemple de Réorganisation de boucles (n=6, m=5).

```
for (i=0; i<n; i++)
 {
    for (j=0; j<m; j++)
     {
       S1(i,j);
       }
 }
```



(a) La représentation du code original.

```
  for (k=0; k<floord(n-1,3); k
++)
   {
   for (i=3*k; i<=min(n-1,3*k+2)
; i++)
    {
      for (j=0; j<m; j++)
       { S1(i,j);}
    }
   }
```



(b) La représentation du code transformé.

FIGURE 2.12 – Exemple de Strip mining (n=9, m=5).

2.7.8 Tiling

Le Tiling de boucle est une transformation clé dans l'optimisation de la localité des données et du parallélisme.

— Le Tiling pour la localité de données nécessite de regrouper les points dans un espace d'itération en blocs plus petits (tuiles) permettant une réutilisation dans plusieurs directions lorsque le bloc s'inscrit dans une mémoire plus rapide pouvant être des registres, cache L1, cache L2 ou cache L3. Lorsque l'exécution se déroule tuile par tuile, les distances de réutilisation ne sont pas en fonction de la taille du problème mais en fonction de la taille de tuile. On peut appliquer le Tiling plusieurs fois, une fois pour chaque niveau de la hiérarchie de la mémoire.

— Le Tiling pour le parallélisme implique de partitionner l'espace d'itération en tuiles qui peuvent être exécutées simultanément sur différents processeurs avec une fréquence et un volume de communication inter-processeur réduits : une tuile est exécutée atomiquement sur un processeur avec une communication requise seulement avant et après l'exécution.

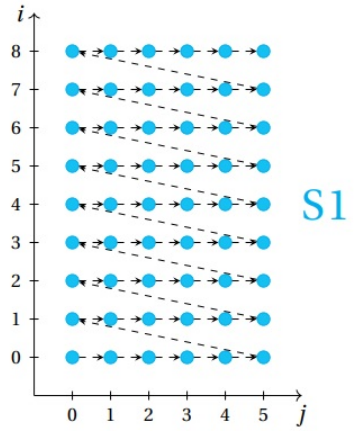
Un exemple de tiling est illustré dans la figure 2.13.

Le Tiling commence par un espace d'itération et partitionne l'espace d'itération en tuiles uniformes d'une taille et d'une forme données. Les tuiles peuvent être des formes quelconques telles que des triangles, des carrés, des rectangles, des parallélogrammes, des hexagones ou leurs équivalents de dimension supérieure. En pratique, cependant, les carrés, les rectangles et les parallélogrammes sont communs. En conséquence, deux types de techniques de Tiling sont distingués dans la littérature : le Tiling rectangulaire et le Tiling parallélépipédique (voir figure 2.14). Dans le premier cas, tous les tuiles sont des carrés, des rectangles ou leurs équivalents de dimension supérieure. Dans le deuxième cas, les tuiles peuvent également être des parallélépipèdes (appelés parallélogrammes dans l'espace 2D) .

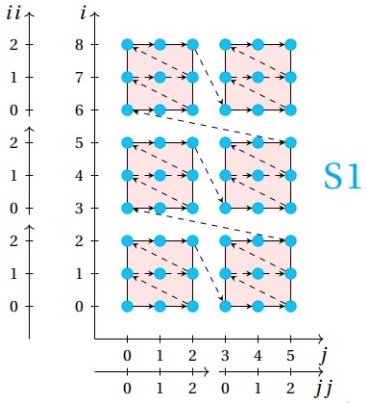
Le Tiling est caractérisé par les propriétés suivantes :

```
for (i=0; i<n; i++)
 {
    for (j=0; j<m; j++)
     {
      S1(i,j);
      }
 }
```

```
for(ii=0;ii<floord(n-1,3);k++)
for(jj=0;ii<floord(m-1,3);k++)
 for(i=3*ii;i<=min(n-1,3*ii+2);i
  ++)
 for(j=3*jj;j<=min(m-1,3*jj+2);j
  ++)
      {
        S1(i,j);
      }
```

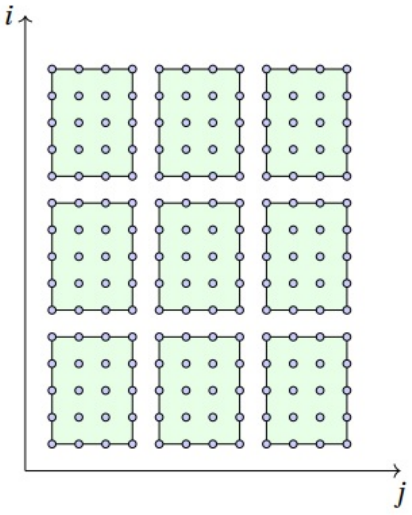


(a) La représentation du code original.

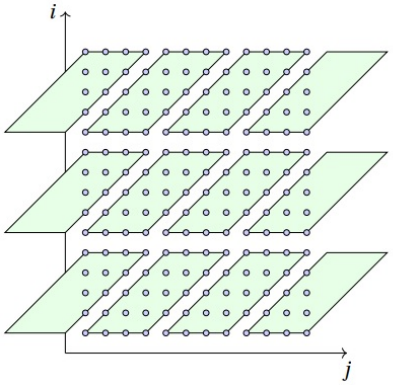


(b) La représentation du code transformé.

FIGURE 2.13 – Exemple de Tiling (n=9, m=6).



(a) Tiling rectangulaire.



(b) Tiling parallélépipédique.

FIGURE 2.14 – Techniques de Tiling.

— Le Tiling est similaire du Strip-mining (Strip-mining fonctionne pour des boucles simples).

— Le Tiling est utilisé pour les nids de boucles.

— L'objectif est d'interchanger les boucles de Tiling vers l'extérieur et les boucles simple vers l'intérieur.

— Le Tiling est caractérisée par la taille des tuiles (Ts) et le décalage des tuiles (To) tel que (0 < To < Ts).

— Chaque tuile commence par l'itération i telle que : i mod Ts = To. Chaque tuile itère de Tk-Ts+To à (Tk+1)-Ts+To-1 , où : Tk est le tuile numéro k.

— Le compilateur doit déterminer les nombres de tuiles minimum et maximum.

— La formule générale pour le Tiling d'une boucle de la forme (For i= i0 , N) est :

```
for it = floord (i0-T0,Ts)* Ts +T0 , floord (N-T0,Ts)* Ts +T0 , Ts
  for i= max (i0, it), min (N, it+Ts-1)
```

2.8 Conclusion

Pour optimiser l'exécution des boucles, les compilateurs leur appliquent des transformations afin d'améliorer la localité spatiale et temporelle des accès mémoire. La majorité de ces transformations sont basées sur le modèle polyédrique. La première section de ce chapitre présente le modèle polyédrique qui permet de représenter les nids de boucles sous la forme de polyèdres afin de construire et rechercher des séquences complexes d'optimisation, ainsi que les notions de base dans ce modèle. Ensuite, les transformations polyédriques et les dépendances de données ont été expliqués dans la deuxième section. Nous avons vu que pour vérifier la légalité d'une transformation pour un nid de boucles, il faut vérifier la légalité de la transformation pour toutes les dépendances présentes dans le nid de boucles original. On a introduit et discuté les différents outils polyédriques existants dans la troisième section. Enfin, dans la dernière section de ce chapitre, les techniques de transformation et d'optimisation polyédrique ont été présentées. Le tiling est l'une des techniques puissantes d'optimisation de programmes. Il permet d'augmenter le niveau du parallélisme en incrémentant les itérations collectées dans le même groupe. On a également expliqué les caractéristiques importantes de cette technique à la fin de cette section.

Le chapitre suivant sera consacré à la discution de la parallélisation et l'optimisation automatique et leurs différents outils.

Chapitre 3

Parallélisation et Optimisation automatique

Sommaire

3.1 Introduction

L'industrie des machines parallèles en et en plein expansion pour répondre à la demande croissante de haute performance pour les applications parallèles. Les architectures multi-cœurs et many-cœurs deviennent un sujet important dans les domaines de l'architecture informatique et de l'informatique haute performance. La conception du processeur n'est plus pilotée par des fréquences d'horloge élevées. Au lieu de cela, de plus en plus de cœurs programmables deviennent disponibles. Au cours des dernières années, il y a beaucoup de travaux de passage généralisé de l'industrie informatique à l'informatique parallèle. Presque tous les ordinateurs grand public sont livrés avec des processeurs multi-cœurs. L'informatique parallèle ne sera plus seulement reléguée aux superordinateurs. De plus, les appareils électroniques tels que les téléphones portables et autres appareils portables ont commencé à introduire des capacités de calcul parallèle. Par conséquent, de plus en plus de développeurs de logiciels devront faire face à une variété de plates-formes et de technologies de calcul parallèle afin de fournir des performances optimales pour utiliser toute la puissance du processeur. Cependant, la programmation parallèle nécessite une connaissance approfondie des systèmes sous-jacents et des paradigmes parallèles, ce qui complique l'ensemble du processus. Il serait souhaitable de laisser le programmeur programmer en séquentiel et que les outils de parallélisation aident à paralléliser automatiquement le programme. Dans ce troisième chapitre, nous présentons la nécessité et les différentes étapes et les techniques de parallélisation automatique, ensuite nous décrivons quelques outils de parallélisation automatique avec un plus de détail sur l'outil PLuTo qui est basé sur le modèle polyédrique et que nous avons utilisé dans nos travaux.

3.2 La nécessité de parallélisation automatique

Le calcul haute performance devient de plus en plus courant. Pour utiliser pleinement lesquelles machines, les programmeurs doivent paralléliser efficacement leurs applications. Cette tâche est loin d'être triviale, d'où la nécessité d'automatiser ce processus. Les techniques passées ont fourni la solution pour des langages de programmation comme FORTRAN et C. Cependant, cela ne suffit pas. Ces techniques traitaient des sections de parallélisation avec des systèmes spécifiques comme la boucle ou une section particulière de code. L'identification des possibilités de parallélisation est une étape cruciale pour générer une application parallèle. Ce besoin de paralléliser les applications est partiellement traité par des outils qui analysent le code pour exploiter le parallélisme. Ces outils utilisent des techniques de compilation ou des techniques d'exécution. Certaines techniques sont intégrées dans certains outils de parallélisation, mais l'utilisateur doit identifier le parallélisme du code et marquer le code avec des constructions de langage spéciales. Le compilateur identifie ces constructions de langage et analyse le code marqué pour la parallélisation. Certains outils ne font que paralléliser une forme spéciale de code comme les boucles. Par conséquent, un outil entièrement automatique pour convertir le code séquentiel en un code parallèle est toujours requis [81, 89, 62, 91, 10] .

3.3 Les étapes de parallélisation automatique

Le processus commence par l'identification des sections de code qui, selon le programmeur, ont des possibilités de parallélisme. Cette tâche est difficile puisque souvent le programmeur qui veut paralléliser le code n'est pas celui qui a écrit le code original. Par conséquent, cette première étape du processus de parallélisation semble facile au début, mais elle peut ne pas être simple.
L'étape suivante consiste à identifier les relations de dépendance de données d'un programme séquentiel donné. C'est une étape cruciale pour trier les sections de code parmi celles identifiées qui ont réellement besoin d'une parallélisation. Cette étape est la plus importante et difficile car elle implique beaucoup d'analyse. La plupart des travaux dans le domaine de parallélisation automatique considèrent les programmes Fortran, car au cours de cette étape d'identification des

dépendances de données, Fortran donne des garanties sur l'aliasing que les langages tels que C et C ++ n'ont pas. L'aliasing peut être décrit comme une situation dans laquelle un emplacement de données en mémoire peut être accédé par différents noms symboliques dans le programme. Parfois, les dépendances sont supprimées en changeant le code c'est la prochaine étape de la parallélisation. Le code est transformé de telle sorte que la fonctionnalité et, par conséquent, la sortie ne sont pas modifiées, mais la dépendance sur une autre section de code est supprimée.
La dernière étape consiste à générer le code parallèle avec un modèle de programmation parallèle approprié. Fonctionnellement, ce code devrait être le même que le code séquentiel d'origine. De plus, il a des constructions supplémentaires et des appels de fonctions parallèles pour lui permettre de fonctionner sur plusieurs threads, processus ou les deux.

3.4 Outils de parallélisation

Nous présentons dans cette section en détail les différents langages de parallélisation proposés.

3.4.1 OpenMP (Open multi processing)

OpenMP [45, 113, 75] est une API (Application Programming Interface) permettant d'écrire une application parallèle pour des machines à mémoire partagée en C, C ++ et FORTRAN. OpenMP est facile à mettre en œuvre et permet également un parallélisme incrémental qui est basé sur les threads. OpenMP utilise le modèle (Fork - Join) de l'exécution parallèle. Ce modèle fonctionne comme suit :

— Le thread maître crée des threads travailleurs et forme une équipe avec eux.

— Les threads travailleurs se terminent avec la région parallèle.

— Le thread maître continue son exécution dans une région séquentielle.

Open MP utilise des directives de compilateur pour la parallélisation. Ces directives sont des directives pour les opérations vectorielles et pour les accélérateurs matériels. Il n'est pas utilisé pour vérifier la dépendance des données, les conflits de données ou les interblocages. OpenMP est conçu pour prendre en charge les programmes qui fonctionneront correctement en tant que programmes parallèles et séquentiels. Le principe de OpenMP est basé sur l'ajout des directives pour indiquer au compilateur : (1)Quelles sont les instructions à exécuter en parallèle, et (2) Comment distribuer les instructions et les données entre les différents threads. Dans la structure de l'API OpenMP on a aussi des bibliothèques pour des fonctionnalités spécifiques (informations dynamiques, actions sur le runtime,...), et on a des variables d'environnement pour influer sur le programme à exécuter (nombre de threads, stratégies d'ordonnancement,...). Nous présentons la structure de la technique OpenMP pour un programme écrit en langage C ou C++ dans la figure 3.1.

3.4.2 CUDA (Compute Unified Device Architecture)

CUDA [97, 40, 96] est une plate-forme de calcul parallèle à usage général et un modèle de programmation qui exploite le moteur de calcul parallèle dans les unités de traitement graphique (GPU) NVIDIA [38] pour résoudre de nombreux problèmes de calcul complexes de manière plus efficace. L'utilisateur de CUDA peut accéder au GPU pour le calcul, comme cela a été fait traditionnellement sur le CPU. La plate-forme CUDA est accessible via des bibliothèques accélérées CUDA, des directives de compilation, des interfaces de programmation d'applications et des extensions aux langages de programmation standard tels que C, C ++, Fortran et Python. CUDA C est une extension de la norme ANSI C [7] avec une poignée d'extensions de langage pour permettre une programmation hétérogène, ainsi que des API simples pour gérer les périphériques, la mémoire et d'autres tâches. CUDA est également un modèle de programmation évolutif qui permet

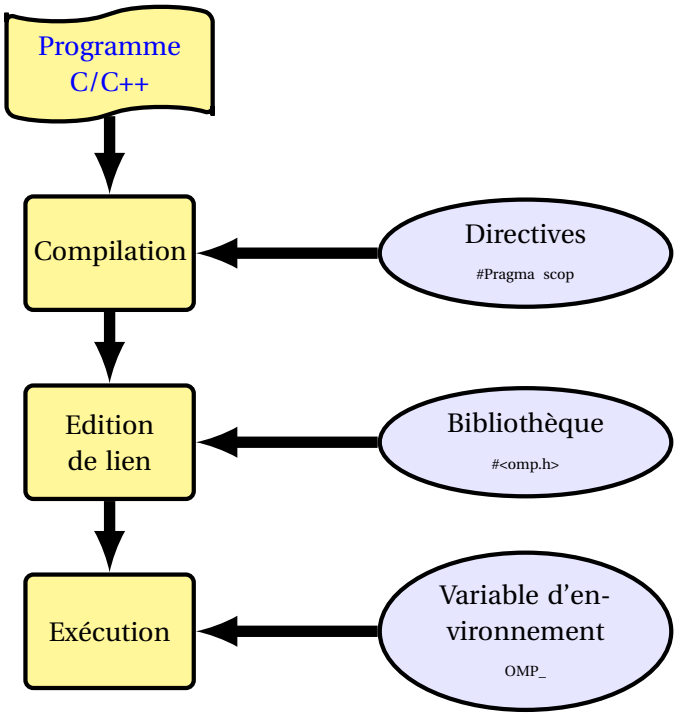


FIGURE 3.1 – Structure de l'API OpenMP pour un programme C/C++.

aux programmes d'adapter de manière transparente leur parallélisme aux GPU avec un nombre variable de cœurs, tout en conservant une courbe d'apprentissage superficielle pour les programmeurs familiarisés avec le langage de programmation C. CUDA fournit deux niveaux d'API pour la gestion du périphérique GPU et l'organisation des threads : API Driver et API Runtime comme montré dans la Figure 3.2. Le driver CUDA se charge du rôle d'intermédiaire entre le code compilé et le GPU. Le runtime CUDA est lui un intermédiaire entre le développeur et le driver qui facilite le développement en masquant certains détails. CUDA propose soit de passer par l'API runtime soit d'accéder directement à l'API driver. Il est possible de voir l'API runtime comme un langage de haut niveau et l'API driver comme un intermédiaire entre le haut et le bas niveau qui permet d'optimiser manuellement le code plus en profondeur.

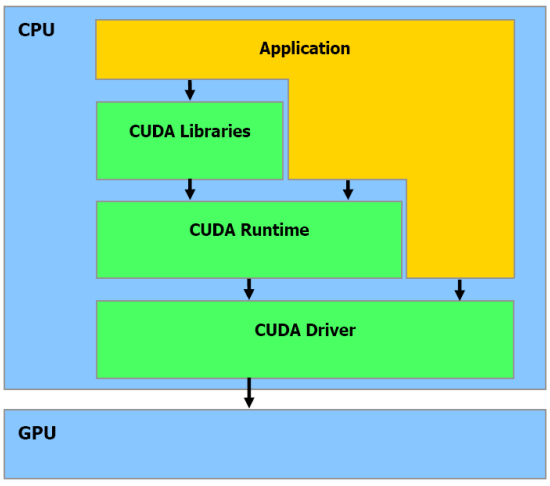


FIGURE 3.2 – Architecture de l'API CUDA.

Un programme CUDA consiste en un mélange des deux parties suivantes :

— Host code qui s'exécute sur le CPU.

— Device code qui s'exécute sur le GPU.

Le compilateur NVIDIA CUDA nvcc sépare le code du périphérique du Host code pendant le processus de compilation. Le Host code est un code C standard et il est en outre compilé avec des compilateurs C. Le Device code est écrit en utilisant CUDA C étendu avec des mots-clés pour l'étiquetage des fonctions parallèles aux données, appelées noyaux. Le Device code est en outre compilé par nvcc. Au cours de la phase de liaison, des bibliothèques d'exécution CUDA sont ajoutées pour les appels de procédure du noyau et la manipulation explicite du périphérique GPU.

3.4.3 PIPS (Paralllel Image Processing System)

Le système de traitement d'images en parallèle PIPS [36] a été développé en tant que plateforme de recherche dans le domaine du traitement d'images et du calcul parallèle. Les algorithmes de traitement d'image sont bien adaptés aux systèmes parallèles en raison de la grande quantité de données et de la possibilité de distribuer les calculs sur de nombreux processeurs de manière naturelle.

La structure de PIPS est hautement modulaire et hiérarchique. La portée des modules de PIPS s'étend des services bas niveau de base jusqu'aux interfaces utilisateur haut de gamme. L'interaction des modules représentés sur la figure 3.3 peut être vue comme une chaîne ou une pile où chaque module ne repose que sur l'interface fournie par le module du niveau inférieur suivant. À son tour, chaque module fournit une interface à l'utilisateur et / ou à la couche suivante de modules. En montant dans la hiérarchie, l'utilisateur est de plus en plus ignorant des propriétés du matériel sous-jacent. Les interfaces interactives haut de gamme masquent même tous les problèmes de parallélisation de l'utilisateur. Descendre dans la hiérarchie permet à l'utilisateur de prendre le contrôle des stratégies de parallélisation, de traitement des données, des concepts de communication ou même du matériel.

L'interface matérielle avec le système et la couche inférieure du système est donnée par le Message Passer (MEPS) qui gère toutes les demandes de communication survenant pendant le calcul. Des mécanismes de synchronisation du système et de diffusion de données sont également mis en œuvre dans cette couche. De nouvelles stratégies de routage et de communication peuvent être intégrées dans MEPS. La portabilité de l'ensemble du logiciel de traitement d'image est garantie en adaptant la partie dépendant du matériel du Message Passer à d'autres architectures. L'intelligence principale concernant le traitement de l'image et sa parallélisation est fournie par le module suivant plus haut dans la hiérarchie. Le module PIPS peut gérer le stockage distribué de parties d'image et fournit des services tels que la distribution et la collecte de données et de commandes.

Dans cette couche, les algorithmes peuvent être optimisés en choisissant la distribution de données la plus adaptée, en minimisant les communications et les calculs ou en superposant des parties de calcul et de communication de l'algorithme. En tant qu'interface de programmation, une bibliothèque C ++ a été implémentée pour former le niveau supérieur suivant du système PIPS. Au moment de l'exécution, le système parallèle est contrôlé par des flux de commande qui sont envoyés au système via le serveur de réseau. Les flux de commandes sont générés par les fonctions de bibliothèque sur le poste de travail.

En utilisant cette bibliothèque, on peut mettre en œuvre des algorithmes de traitement d'image complexes et les intégrer dans n'importe quel système capable d'appeler des fonctions fournies par l'utilisateur. Cela a été fait avec le système bien connu Cantata/Khoros [6]. L'interface utilisateur «PipsLab» a été aussi conçue. Elle donne accès à presque toutes les fonctionnalités de la bibliothèque C ++ via un interpréteur. La syntaxe et l'utilisation de 'PipsLab' ressemble assez à

MatLab. Par exemple. il est possible d'avoir des sous-routines et des boucles. En haut de 'Pips-Lab' se trouve l'interface X-Windows 'Action'. Il donne un accès intuitif à toutes les fonctionnalités 'PipsLab'.

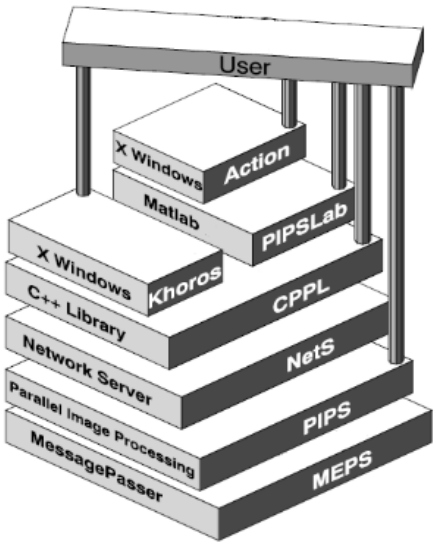


FIGURE 3.3 – L'infrastructure de la technique PIPS [36].

3.4.4 OpenCL (Open Computing Language)

OpenCL [114] est une norme ouverte pour la programmation parallèle à usage général sur les CPU, les GPU et autres processeurs, donnant aux développeurs de logiciels un accès portable et efficace à la puissance de ces plates-formes de traitement hétérogènes. OpenCL prend en charge un large éventail d'applications, allant des logiciels embarqués et grand public aux solutions HPC, en passant par une abstraction portable de haut niveau et peu performante. OpenCL est particulièrement adapté pour jouer un rôle de plus en plus important dans les applications graphiques interactives émergentes qui combinent des algorithmes de calcul parallèles généraux avec des pipelines de rendu graphique. OpenCL se compose d'une API pour coordonner le calcul parallèle entre des processeurs hétérogènes ; et un langage de programmation multiplateforme avec un environnement de calcul bien spécifié.
Pour décrire les idées fondamentales derrière OpenCL, nous utiliserons une hiérarchie de modèles :

— Modèle de plateforme ;
— Modèle d'exécution ;
— Modèle de mémoire ;
— Modèle de programmation.

Le Modèle de plateforme : Le modèle consiste en un Host connecté à un ou plusieurs Devices OpenCL. Un Device OpenCL est divisé en une ou plusieurs Compute Units qui sont ensuite divisées en un ou plusieurs Processing Elements. Les calculs sur un Device se produisent dans les Processing Elements.

Le Modèle d'exécution L'exécution d'un programme OpenCL se déroule en deux parties : Les Kernels qui s'exécutent sur un ou plusieurs Devices OpenCL et un programme Host qui s'exécute sur le Host. Le programme Host définit le contexte des Kernels et gère leur exécution.

Le Modèle de mémoire : Dans ce modèle

- Chaque work-item (thread) invoque un kernel (code compilé puis exécuté). Chaque work-item dispose de sa mémoire privée. Cette mémoire privée n'est accessible que par le work-item.
- Des work-items peuvent être groupés et former ainsi un work group. Chaque work group partage une mémoire locale qui joue le rôle de mémoire partagée. La cohérence assurée au sein de cette mémoire est faible.
- La mémoire globale ou constante (en lecture seule) est quand à elle partagée par tous les work groups.

Le Modèle de programmation : Le modèle d'exécution OpenCL prend en charge les modèles de programmation : Data parallel et task parallel, ainsi que les hybrides de ces deux modèles.

Dans un programme Host OpenCL de base on a 5 étapes simples :

1. Définir la plateforme (platforme = devices + context + queues)
2. Créer et construir le programme (bibliothèque dynamique pour les Kernels)
3. Installer memory objects
4. Définir le Kernel (attacher des arguments à la fonction du Kernel)
5. Soumettre des commandes, transférer des objets mémoire et exécuter les Kernels.

La figure 3.4 illustre les API de base de la plateforme d'exécution dans OpenCL.

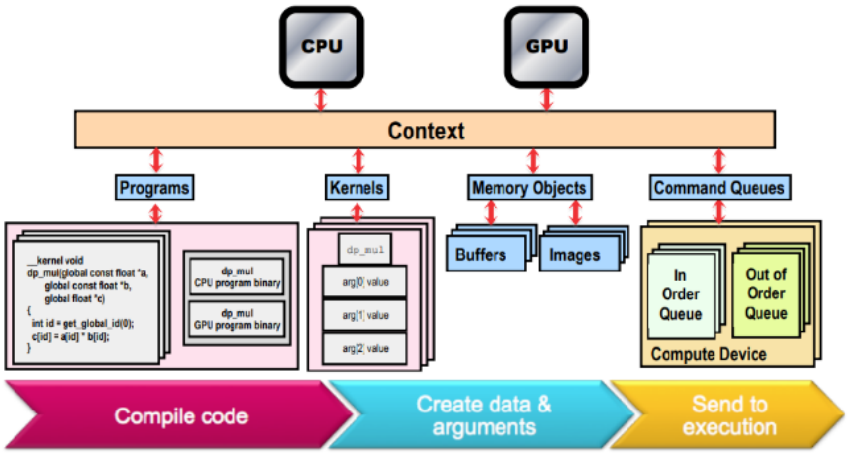


FIGURE 3.4 – Les API de base de la plateforme d'exécution dans OpenCL.

3.4.5 OpenACC (Open Accelerators)

Avec l'émergence des architectures GPU et many-cœurs dans l'informatique haute performance, les programmeurs souhaitent pouvoir programmer en utilisant un modèle de programmation familier de haut niveau qui offre à la fois hautes performances et portabilité à un large éventail d'architectures informatiques. OpenACC est apparu en 2011 en tant que modèle de programmation utilisant des directives de haut niveau pour compiler le parallélisme dans le code et en parallélisant les compilateurs pour générer le code pour divers accélérateurs parallèles.
Afin de garantir la compatibilité d'OpenACC avec toutes les architectures informatiques disponibles au moment de son lancement et dans l'avenir, OpenACC définit un modèle abstrait pour l'informatique accélérée. Ce modèle expose plusieurs niveaux de parallélisme pouvant apparaître

sur un processeur, ainsi qu'une hiérarchie de mémoires à différents degrés de vitesse et d'adressabilité. L'objectif de ce modèle est de garantir qu'OpenACC sera applicable à plus d'une architecture particulière, voire même des architectures largement disponibles à ce moment-là, mais de garantir que OpenACC puisse également être utilisé sur des futurs périphériques.

À la base, OpenACC prend en charge le déchargement des calculs et des données d'un périphérique hôte (Host Device) vers un accélérateur (Accelerator Device). En fait, ces périphériques peuvent être identiques ou avoir une architecture complètement différente, comme dans le cas d'un hôte de processeur (CPU) et d'un accélérateur de processeur graphique (GPU). Les deux dispositifs peuvent également avoir des espaces mémoire séparés ou un seul espace mémoire. Dans le cas où les deux périphériques ont des mémoires différentes, le compilateur et le moteur d'exécution OpenACC analysent le code et gèrent toute gestion de la mémoire d'accélérateur et le transfert des données entre la mémoire de l'hôte (Host Memory) et celle du périphérique (Device Memory). La figure 3.5 présente un diagramme de haut niveau de l'accélérateur abstrait OpenACC (les périphériques et les mémoires peuvent être physiquement les mêmes sur certaines architectures).

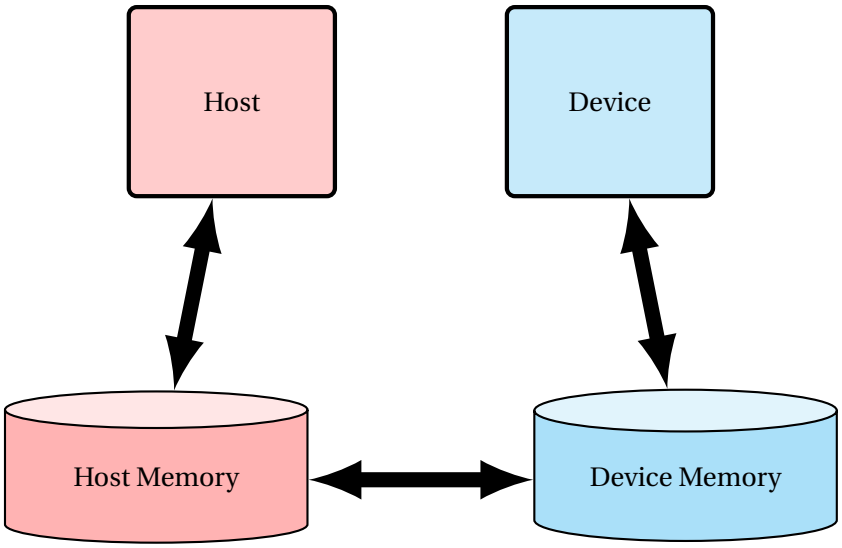


FIGURE 3.5 – Modèle abstrait de OpenACC.

3.5 Les outils de parallélisation automatique

Dans ce qui suit nous présentons les différents outils de parallélisation automatique.

3.5.1 Suif (Stanford University Intermediate Format)

Suif [125] est un compilateur de parallélisation automatique fait à l'université de Stanford. Ce compilateur lit les programmes ordinaires C ou FORTRAN et génère des codes intermédiaires parallélisés (du Stanford University Intermediate Format), des codes exécutables parallélisés pour certains systèmes, ou un programme C contenant des constructions parallèles comme doall, doacross, etc [72]. Il est capable de détecter et de générer les doall, avec le paquet librement disponible. Ce compilateur est composé de plusieurs passes. Chaque passe joue son rôle et est indépendant l'un de l'autre.

Le noyau SUIF remplit trois fonctions majeures :

— **Il définit la représentation intermédiaire des programmes** : cette représentation prend en charge à la fois les transformations de haut niveau de restructuration de programme ainsi que les analyses et optimisations de bas niveau.

— **Il fournit des fonctions pour accéder et manipuler la représentation intermédiaire** : masquer les détails de bas niveau de la mise en œuvre, rend le système plus facile à utiliser et aide à maintenir la compatibilité si la représentation est modifiée.

— **Il structure l'interface entre les passes du compilateur** : les passes SUIF sont des programmes distincts qui communiquent via des fichiers. Le format de ces fichiers est le même pour toutes les étapes d'une compilation. Le système prend en charge l'expérimentation en autorisant des données définies par l'utilisateur dans les annotations.

3.5.2 Polaris (Automatic Parallelization of Conventional Fortran Programs)

L'objectif du compilateur Polaris [29] est de faire progresser l'état de l'art dans la parallélisation automatique des programmes. Créer et maintenir un compilateur d'optimisation qui représente cet état de l'art et qui peut être utilisé comme une infrastructure solide par des groupes de recherche et développement de compilateurs. Polaris inclura toutes les techniques d'optimisation nécessaires pour transformer un programme séquentiel donné en une forme qui s'exécute efficacement sur la machine cible. Cela inclut des techniques telles que la détection automatique du parallélisme et la distribution des données. Les machines cibles prévues sont des ordinateurs parallèles haute performance avec un espace d'adressage global ainsi que des multiprocesseurs traditionnels à mémoire partagée.

Le compilateur Polaris prend en entrée un programme Fortran77, transforme ce programme pour qu'il s'exécute efficacement sur un ordinateur parallèle , et sort cette version du programme dans l'un des nombreux dialectes Fortran parallèles possibles. Le langage d'entrée comprend plusieurs directives qui permettent à l'utilisateur de Polaris de spécifier explicitement le parallélisme dans le programme source. Le langage de sortie de Polaris est généralement sous la forme de Fortran77 plus les directives parallèles. Par exemple, un ensemble de directives parallèles génériques comprend les directives «CSRD $ PARALLEL» et «CSRD $ PRIVATE a, b», spécifiant que les itérations de la boucle suivante doivent être exécutées simultanément et que les variables a et b doivent être déclarées «private» à la boucle courante, respectivement. Un autre langage de sortie que Polaris peut générer est le Fortran plus le langage de directive disponible sur la machine SGI [86].
Polaris effectue ses transformations en plusieurs «passes de compilation». En plus de nombreuses passes couramment connues, Polaris comprend des capacités avancées exécutant les tâches suivantes : privatisation de réseaux, tests de dépendance de données, reconnaissance de variables d'induction, analyse inter procédurale et analyse de programme symbolique. Un ensemble complet d'options permet à l'utilisateur et au développeur de Polaris d'expérimenter l'outil de manière flexible. L'implémentation de Polaris comprend 170 000 lignes de code C ++. Une infrastructure de base fournit une hiérarchie de classes C ++ que les développeurs de la compilation individuelle peuvent utiliser pour manipuler et analyser le programme d'entrée.

3.5.3 Cetus

Cetus [47] est un outil qui effectue la transformation source-à-source des programmes. Il fournit également une infrastructure de base pour écrire des outils de parallélisation automatique ou des compilateurs. Cetus peut fonctionner sur n'importe quel système qui prend en charge Oracle, environnement d'exécution Java. Il a été développé à l'origine par Purdue University. Il fournit un analyseur C interne et il est écrit en Java. L'architecture de Cetus est similaire à Polaris, la principale différence est que Polaris traduit les codes FORTRAN et Cetus est conçu pour les programmes écrits en langage C. Les techniques de parallélisation de base que Cetus implémente actuellement sont la privatisation, la reconnaissance des variables de réduction et la substitution des variables d'induction.
Cetus permet la parallélisation automatique en utilisant l'analyse de dépendance de données avec les inégalités Banerjee-Wolfe [106], le tableau et la privatisation scalaire. Il utilise GNU CPP pour

le prétraitement, Yacc [76] et Bison [84] pour la compilation, et Antlr [100] pour la génération de code. Nous montrons les composants de base de compilateur Cetus dans la figure 3.6.

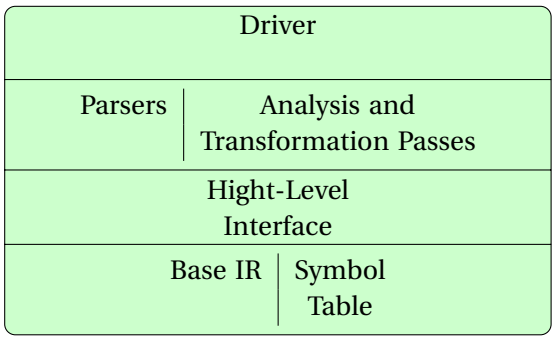


FIGURE 3.6 – L'infrastructure de Cetus.

3.5.4 Par4All

Par4All [120] est un compilateur de parallélisation et d'optimisation automatique pour les program mes séquentiels C et Fortran financés par le démarrage du projet HPC. Le but de ce compilateur source-à-source est d'intégrer plusieurs outils de compilation dans un compilateur facile à utiliser mais puissant qui transforme automatiquement des programmes existants pour cibler différentes plates-formes matérielles.
L'hétérogénéité est partout aujourd'hui, des superordinateurs au monde mobile, et l'avenir semble être promis à une hétérogénéité de plus en plus grande. Ainsi, l'adaptation automatique de programmes sur des cibles telles que les systèmes multicœurs, les systèmes embarqués, les ordinateurs haute performance et les GPU est un défi majeur.
Par4All est principalement basé sur l'infrastructure du compilateur source-à-source PIPS (Interprocedural Parallelizer of Scientific Programs) et bénéficie de capacités inter-procédurales telles que les effets mémoire, la détection de réduction, la détection de parallélisme et les analyses polyédriques telles que les régions convexes et les conditions préalables.
La nature source-à-source de Par4All facilite l'intégration d'outils dans le flux de compilation. La combinaison des analyses de PIPS et l'insertion d'autres optimiseurs au milieu du flux de compilation est automatisée par Par4All en utilisant un gestionnaire de passes programmable pour effectuer une analyse complète du programme, repérer des boucles parallèles et générer principalement du code OpenMP, CUDA ou OpenCL. La figure 3.7 illustre l'architecture de compilateur Par4All.

3.5.5 ICU-PFC

ICU-PFC [69] est un compilateur de parallélisation automatique. Il reçoit des sources FORTRAN et génère des codes FORTRAN parallèles dans lesquels des directives OpenMP pour l'exécution parallèle sont insérées. ICU-PFC détecte la boucle parallèle DO ALL dans le programme et insère les directives OpenMP appropriées. Pour la détection de boucle parallèle, une matrice de dépendances est conçue pour stocker des informations de dépendance de données d'instructions dans une boucle. Enfin, le générateur de code parallélisé de ICU-PFC peut générer du code parallèle supporté par OpenMP.

Le ICU-PFC est composé de : front-end, un analyseur de base, un analyseur de dépendance de données et un générateur de code.
Le Front-end effectue l'analyse lexicale et l'analyse syntaxique du code source qui est écrit dans FORTRAN77. L'analyseur de base effectue l'analyse du flux de contrôle, l'analyse du flux de données et d'autres analyses. L'analyseur de dépendance de données vérifie les relations de dépendance de données entre les instructions. Cette relation est synthétisée à partir des dépendances

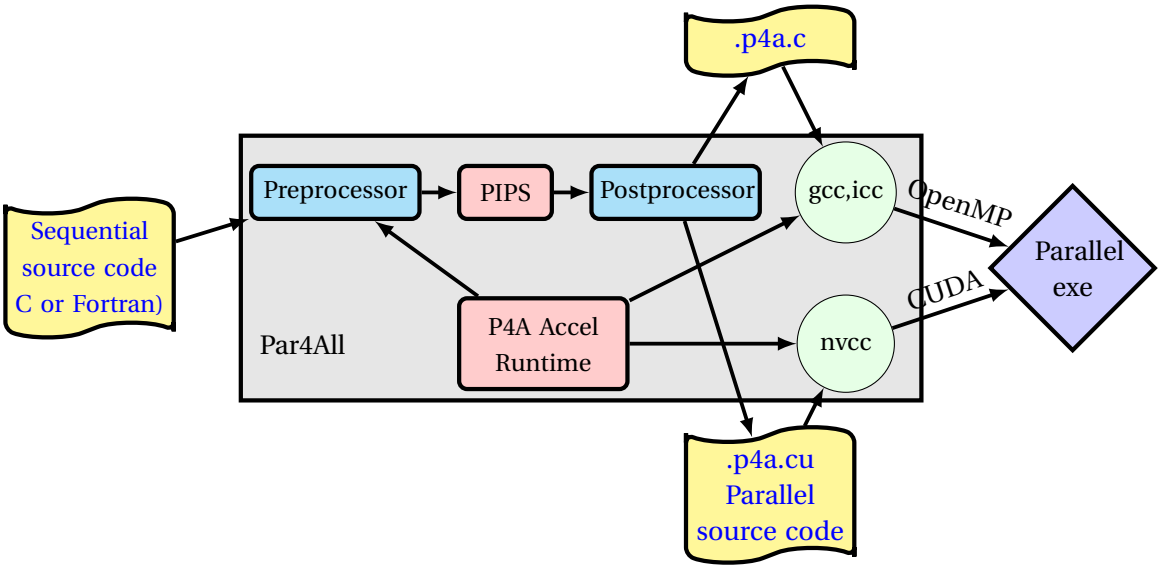


FIGURE 3.7 – L'architecture de l'outil de parallélisation automatique Par4All.

de données des variables utilisées dans les instructions. En se basant sur les données des deux analyseurs, le générateur de code produit le code FORTRAN parallélisé. Le code généré, qui peut inclure des directives OpenMP, est compilé par Omni OpenMP Compiler. L'architecture du compilateur ICU-PFC est illustrée par la figure 3.8.

3.5.6 iPat/OMP

iPat [73] est un outil interactif. C'est un outil de parallélisation pour OpenMP. L'outil aide l'utilisateur à paralléliser les codes séquentiel en version parallélisée en utilisant OpenMP. L'éditeur Emacs est requis pour la mise en œuvre de l'outil. Il utilise le compilateur Omni OpenMP.

iPat / OMP a quatre types de capacités :

1. **Analyse de parallélisme :** vérifier les dépendances, en conséquence décider quelle section peut être parallélisée.
2. **Création de la directive :** inclure les directives OpenMP pour la section de code pouvant être parallélisée.
3. **Restructuration du programme :** Restructuration du programme ou des sections afin d'accroître le parallélisme.
4. **Analyse du temps d'exécution :** y compris les appels qui mesurent le temps d'exécution.

Nous présentons l'architecture de l'outil iPat/OMP dans la figure 3.9.

3.5.7 CAPO (CAPTools-based Automatic Parallelizer using OpenMP)

CAPO [75] est un outil interactif de parallélisation et d'analyse des performances développé par NASA Ames Research Center pour insérer les directives OpenMP dans les codes FORTRAN. Paraver est un outil d'analyse de performance est développé par le Centre européen de parallélisme de Barcelone, Université de Catalogne (CEPBA-UPC) pour analyser la performance d'un programme parallèle.

CAPO-Paraver (voir figure 3.10) est un environnement de programmation parallèle assisté par ordinateur qui lie Capo avec Paraver. Cet outil aide les programmeurs dans les optimisations complexes de programmes parallèles, ce qui est très difficile manuellement. CAPO dispose d'un

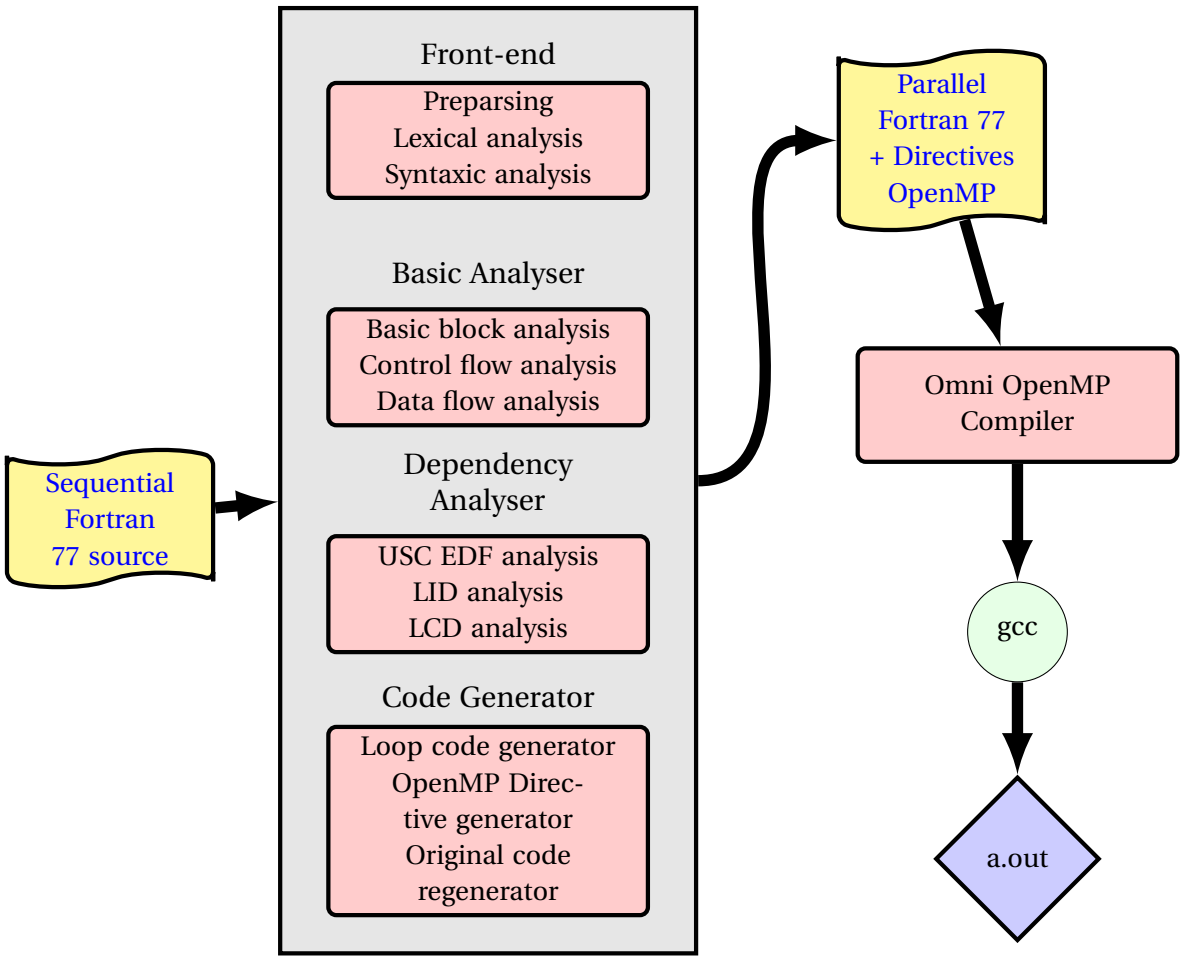


FIGURE 3.8 – L'architecture de l'outil ICU-PFC.

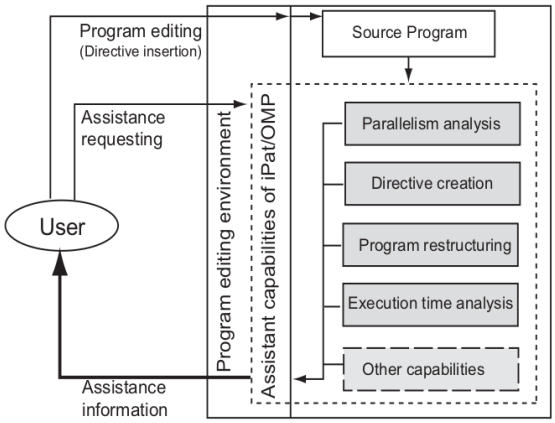


FIGURE 3.9 – L'architecture de l'outil iPat/OMP [73].

moteur d'analyse de dépendances intégré pour les boucles qui possède une fonction supplémentaire de stockage des informations de dépendance dans une base de données et d'amélioration de l'analyse des dépendances en utilisant les réponses aux questions posées à l'utilisateur.
Paraver se compose d'un paquet de traçage et d'une interface utilisateur graphique (GUI) pour examiner les traces. Paraver a la capacité d'analyser le niveau de thread, le niveau de tâche et les programmes parallèles hybrides. Par conséquent, cet environnement aide l'utilisateur à la fois dans l'analyse de la dépendance des données et dans l'analyse des performances. Il prend l'entrée de l'utilisateur, ce qui augmente l'efficacité du programme parallèle. Cependant, cet outil cible uniquement la parallélisation au niveau de la boucle et non la parallélisation au niveau de la tâche (analyse).

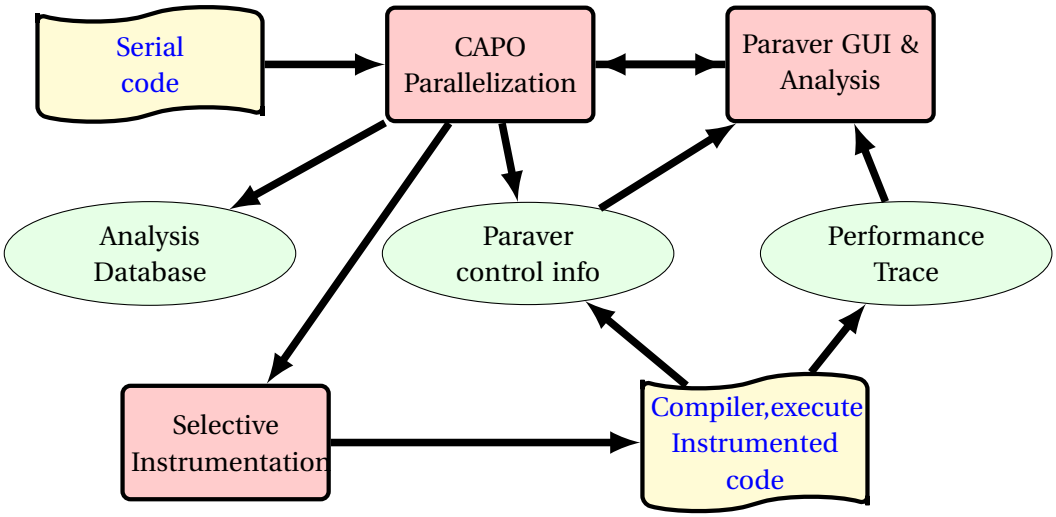


FIGURE 3.10 – La parallélisation automatique avec CAPO.

3.5.8 YUCCA (User Code Conversion and Analysis)

L'outil YUCCA anciennement connu sous le nom Serial to Parallel (S2P) est un outil de parallélisation automatique développé par KPIT Technologies qui considère les boucles comme des tâches de parallélisation [10]. YUCCA est applicable pour la parallélisation du programme C hérité sans aucune intervention manuelle.
L'outil YUCCA est un outil de conversion source-to-source ; c'est-à-dire lorsqu'une application C est donnée en entrée, YUCCA génère une application C parallélisée en sortie. Le code parallèle est un code multithread avec les constructions Pthreads et OpenMP insérées aux endroits appropriés. Tout au long de ce texte, les mots «code», «program» et «application» sont utilisés de manière interchangeable et ils se réfèrent aux entrées et aux sorties de l'outil YUCCA. YUCCA insère des API Pthreads en cas de parallélisation des tâches et insère des API OpenMP en cas de parallélisation de boucles.
L'outil YUCCA se compose d'un front-end de type compilateur qui peut prétraiter et analyser le code d'application et d'un back-end intelligent qui effectue une analyse de dépendance statique pour identifier les sections de code parallélisables. La figure 3.11 illustre la parallélisation automatique avec YUCCA.

3.5.9 Prospector

Prospector [78], un outil d'extraction et de conseil en parallélisme basé sur le profilage qui permet une programmation parallèle plus facile. Supposons un processus de parallélisation typique qui comprend les quatre étapes suivantes :

1. Trouver des candidats pour la parallélisation ;

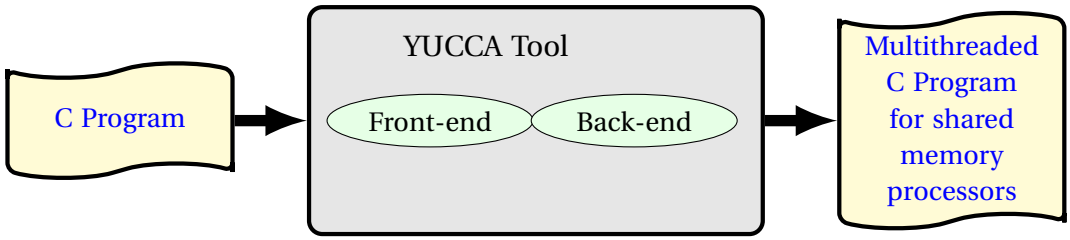


FIGURE 3.11 – La parallélisation automatique avec YUCCA.

2. Comprendre les candidats, principalement en analysant les dépendances de données dans les cibles ;
3. Paralléliser les cibles ;
4. Vérification et optimisation.

Prospector vise à combler l'écart entre la parallélisation automatique et la parallélisation manuelle. Prospector divise le travail entre les outils logiciels et les programmeurs pour maximiser le bénéfice global. Il fournit aux programmeurs des candidats de cibles parallélisables qui ont été découvertes par le profilage dynamique. Cependant, les décisions sur la façon de paralléliser les boucles identifiées sont laissées aux programmeurs, bien que Prospector fournisse des conseils. L'architecture de l'outil Prospector est montré dans La figure 3.12.

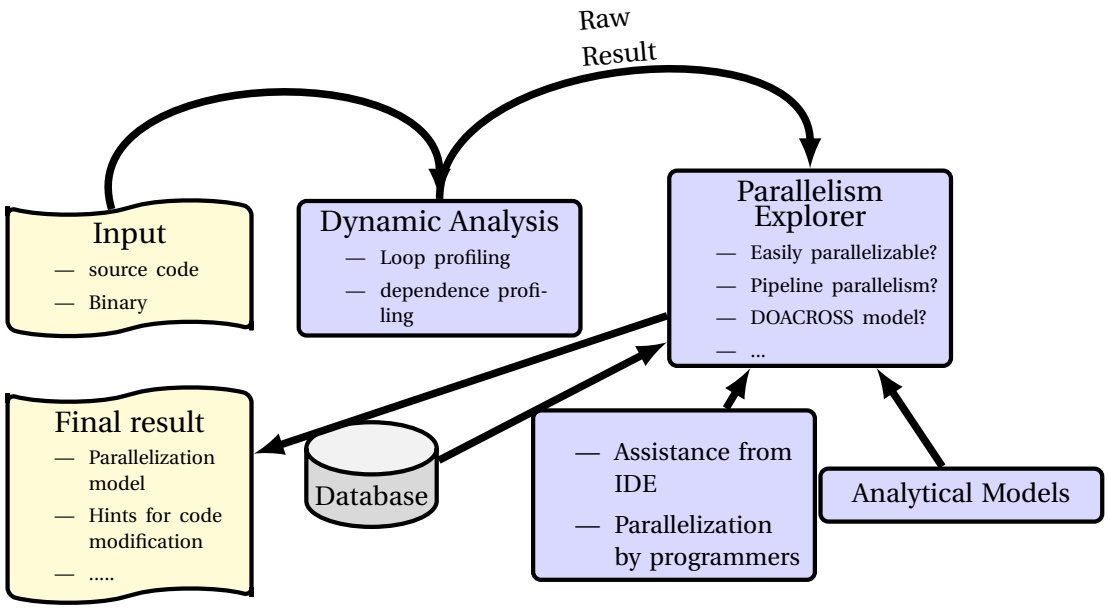


FIGURE 3.12 – L'architecture de l'outil Prospector.

3.5.10 PHiPAC

L'utilisation d'une interface de bibliothèque d'algèbre linéaire standard, telle que BLAS, permet au code d'application portable d'obtenir des performances élevées à condition qu'une bibliothèque optimisée soit disponible et abordable. J.Bilmes et al ont développé une méthodologie, nommée PHiPAC [27] , pour développer des bibliothèques d'algèbre linéaire haute performance portables en langage C. L'objectif est de produire, avec un minimum d'effort, des bibliothèques d'algèbre linéaire hautes performances pour une large gamme de systèmes. La méthodologie PHiPAC a trois composantes. Premièrement, J.Bilmes et al [27] ont développé un modèle générique de compilateurs C et de microprocesseurs actuels qui fournit des instructions pour la production

de code C portable hautes performances. Deuxièmement, plutôt que des routines particulières codées à la main, ils ont écrit des générateurs paramétrés qui produisent du code selon leurs besoins. Troisièmement, ils ont écrit des scripts qui ajustent automatiquement le code pour un système particulier en faisant varier les paramètres des générateurs.

les résultats montrent que l'écriture d'un générateur paramétré et de scripts de recherche pour une routine nécessite moins d'effort que le réglage manuel d'une version unique pour un système unique. De plus, avec l'approche PHiPAC, l'effort de développement peut être amorti sur un grand nombre de plates-formes. Et en recherchant automatiquement un grand espace de conception, on peut découvrir des combinaisons de paramètres gagnantes mais imprévues. En utilisant la méthodologie PHiPAC, le travail produit un générateur de multiplication matriciel portable, compatible avec la bibliothèque BLAS ainsi que le code résultant peut atteindre plus de 90% des performances maximales sur une variété de stations de travail actuelles.

3.5.11 APOLLO (Automatic speculative POLyhedral Loop Optimizer)

Apollo [88] est un framework d'optimisation de boucles automatique et dynamique basé sur le modèle polyédrique. Apollo vise des nids de boucles de profondeur arbitraire, éventuellement imparfaite, et qui ne peuvent pas être analysées statiquement au moment de la compilation. Le concept de base d'Apollo repose sur le fait que, même s'ils ne sont pas statistiquement analysables, de nombreux programmes présentent un comportement linéaire au moment de l'exécution, du moins dans certaines phases. Le système vise à optimiser efficacement ces codes en utilisant le modèle polyédrique à la volée. Actuellement, Apollo cible les codes "C" et "C ++". Cependant, tous les traitements sont effectués dans LLVM IR [83] et, par conséquent, étendre Apollo à n'importe quel autre langage est aussi simple que d'étendre l'analyseur de langage LLVM correspondant pour reconnaître les directives # pragma, utilisé pour spécifier les nids de boucles cibles. L'aperçu général de l'architecture d'Apollo est présenté à la figure 3.13. L'utilisateur marque les nids de boucles cibles avec un pragma (# pragma apollo dcop). Le code attribué au pragma est transmis au compilateur Apollo (apollocc pour les programmes 'C' et apollo ++ pour les programmes 'C ++').
Au moment de la compilation, la phase statique d'Apollo permet de :

1. analyser avec précision les instructions en mémoire pouvant être désambiguïsées au moment de la compilation ;
2. générer une version instrumentée pour suivre les accès en mémoire qui ne peuvent pas être désambiguïsés au moment de la compilation. La version instrumentée fonctionnera sur un échantillon des itérations de la boucle la plus externe et les informations acquises de manière dynamique seront utilisées pour créer un modèle de prédiction des accès mémoire non analysables statiquement ;
3. générer des squelettes de code parallèles qui sont des versions incomplètes de l'imbrication de boucles d'origines et qui nécessitent une instanciation à l'exécution pour générer le code final. Chaque instanciation représente une nouvelle optimisation. Par conséquent, les squelettes de code peuvent être considérés comme des modèles hautement génériques prenant en charge un grand nombre de transformations d'optimisation et de parallélisation. En outre, les squelettes intègrent le support requis pour les spéculations (par exemple, le code de vérification et de récupération).

Ces étapes sont décrites dans la région intitulée «compilation time » à la figure 3.13.
Au moment de l'exécution, la phase dynamique d'Apollo permet de :

1. exécuter la version instrumentée sur un échantillon d'itérations de boucle ultérieurs consécutives ;
2. construire un modèle de prédiction linéaire pour les limites de la boucle et les accès à la mémoire ;

3. calculer les dépendances entre ces accès;
4. instancier un squelette de code et générer une version optimisée et parallèle du code séquentiel d'origine, sémantiquement correcte par rapport au modèle de prédiction;
5. lors de l'exécution du code multi-thread, chaque thread vérifie indépendamment si la prédiction est toujours valable. Si ce n'est pas le cas, une restauration est lancée et le système tente de créer un nouveau modèle de prédiction.

Ces étapes sont décrites dans la région intitulée «Run Time» dans la figure 3.13.

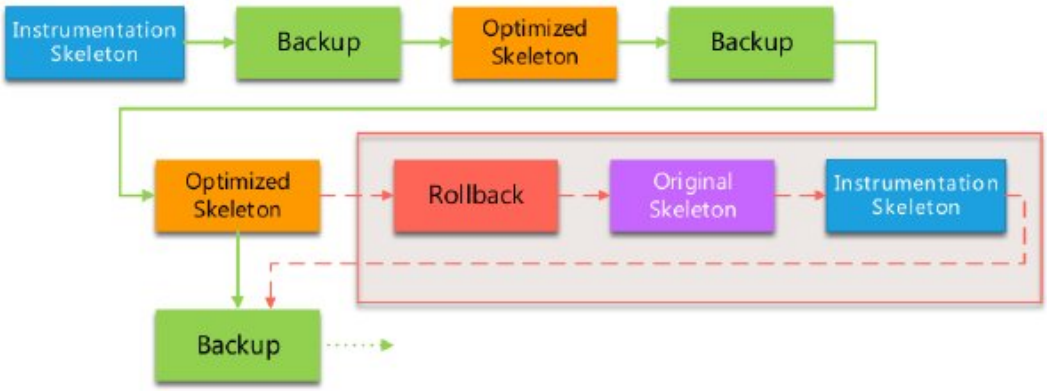


FIGURE 3.13 – L'execution d'une boucle avec Apollo [108].

3.5.12 PLuTo (An automatic parallelizer and locality optimizer for affine loop nests)

PLuTo [31, 33, 30]est un framework automatique de transformation source-à-source basé sur le modèle polyédrique qui peut optimiser simultanément des programmes réguliers pour le parallélisme et la localité des données. L'idée de base de ce framework est de transformer un code source C d'entrée en un code C de sortie sémantiquement équivalent qui permet d'obtenir un meilleur parallélisme et une meilleure localité des données. Les codes sources ciblés sont des séquences de boucles éventuellement imparfaitement imbriquées, et les transformations sont des fonctions affines qui correspondent au modèle polyédrique.

Pluto implémente le tiling de boucle, qui est connu pour ses bonnes performances pour les opérations sur des tableaux ou des matrices de grandes tailles pour le parallélisme et la localité de données. Il réalise également la SIMDization, en insérant des directives de compilation permettant la vectorisation des boucles internes. Plluto contient aussi plusieurs types de transformations de boucle, telles que la fusion de boucle, l'inversion, l'échange et le skewing, etc. De plus, Pluto est capable de générer automatiquement un code parallèle OpenMP optimisé pour les architectures multicœurs. La figure 3.14 montre l'ensemble d'outils qui sont utilisés dans la parallélisation avec PLuTo. Uday Bondhugula et Al [34] ont utilisé le scanner, le parser et le testeur de dépendance du projet LooPo. Le testeur de dépendance de LooPo peut fournir des dépendances de flux, les anti dépendance, les dépendances d'entrée et de sortie. Les développeurs de Pluto ont utilisé PipLib comme solveur pour trouver une solution minimale lexicographique. Cloog a été utilisé pour la génération de code. Le framework de transformation prend en entrée, les domaines polyédriques et les polyèdres de dépendance du testeur de dépendance de LooPo, calcule les transformations et les fournit à Cloog. Le code parallèle OpenMP compilable est finalement sorti après un Post-traitement syntaxique sur le code Cloog. Ce Post-traitement syntaxique est intégré pour effectuer des transformations syntaxiques sur le code généré à partir de Cloog. Le choix des boucles pour effectuer ces transformations est spécifié par le framework de transformation, et donc la légalité est garantie.

a-LooPo (Loop Parallelization in the Polytope Model) : LooPo [63] est un framework de parallélisation de boucles développé à l'Université de Passau qui est basé sur le modèle poly-

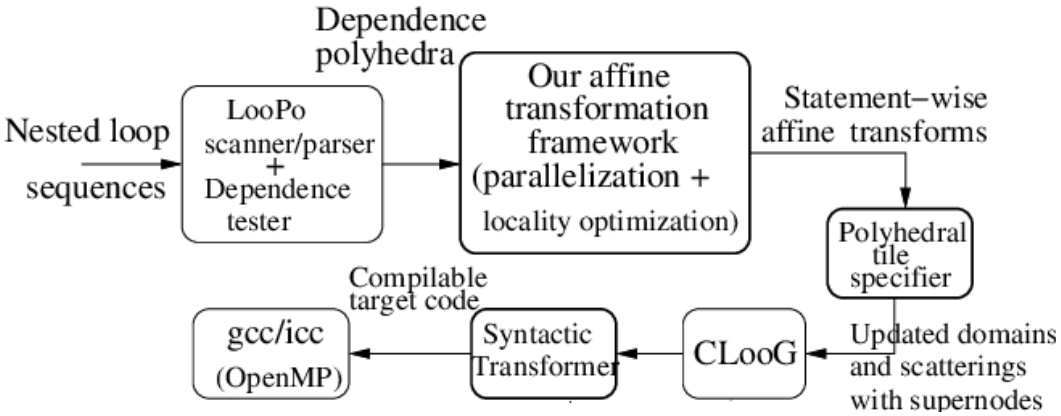


FIGURE 3.14 – La parallélisation automatique avec PLuTo [34].

édrique. Tout utilisateur intéressé par un aspect particulier de la parallélisation peut brancher son propre module dans LooPo et obtenir un environnement de compilation source-à-source personnalisé. LooPo se compose d'un certain nombre de modules, qui transforment le programme source en un programme cible parallèle exécutable :

— **L'entrée :** LooPo accepte des nids de boucle en C, Fortran ou un certain nombre de notations de boucles abstraites, et des déclarations de fonctions, de procédures et de constantes symboliques.

— **Les solveurs d'inégalité :** Nous avons considéré plusieurs méthodes pour la programmation linéaire paramétrique, qui est le problème mathématique central du modèle du polyèdre. L'implémentation de LooPo est basée sur PIpLib [54] et l'algorithme Fourier-Motzkin [77]. Fourier-Motzkin est la méthode standard de la projection de polytopes .

— **L'analyseur de dépendances :** LooPo présente une méthode d'analyse de dépendances décrite par Banerjee [11]. Il travaille sur la mise en œuvre de la méthode de Feautrier [52], qui permet de paralléliser un code, grâce à une analyse plus précise.

— **Les scheduleurs :** LooPo fournit trois scheduleurs automatiques :

1. La méthode hyperplan de Lamport [82] gère parfaitement les boucles imbriquées avec des dépendances uniformes.
2. Le scheduleur de Feautrier [53] détermine un code optimal pour des boucles imparfaitement imbriquées avec des dépendances.
3. Le scheduleur de Darte/Vivien [46] est rapide et produit des résultats raisonnablement pour des programmes en boucle arbitraires avec des dépendances uniformes et non-uniformes.

— **Les allocateurs :** LooPo fournit uniquement la méthode d'allocation de Feautrier qui détermine le placement des opérations sur les processeurs virtuels en fonction de la règle de calcul du propriétaire.

— **Le générateur de code cible :** La génération de code cible se déroule en trois phases :

1. Les instructions du programme source sont transformées individuellement.
2. Ces instructions transformées sont fusionnées en un seul programme cible.
3. L'arbre d'analyse cible est traduit en l'un de plusieurs langages de sortie possibles, par exemple C parallèle.

b-PipLib (Parametric Integer Programming Library) : PipLib [54] est le solveur de programmation linéaire en nombre entiers conçu par Paul Feautrier. PIpLib est un logiciel qui trouve le minimum lexicographique (ou maximum) dans l'ensemble des entiers appartenant à un polyèdre convexe. La différence entre PipLib et les autres outils de programmation en nombre entiers (comme lp-solve ou CPLEX) est que le polyèdre peut dépendre linéairement d'un ou de plusieurs paramètres entiers. Si l'utilisateur demande une solution non entière, PipLib peut donner la solution exacte en tant que quotient est entier. Le coeur du PIpLib est l'algorithme de coupe de Gomory paramétré suivi de la méthode dual simplex paramétrée. La bibliothèque PipLib a été implémentée pour permettre à l'utilisateur d'appeler PIpLib directement à partir de ses programmes, sans accès aux fichiers ni appels système. L'utilisateur n'a besoin que de lier ses programmes avec les bibliothèques C.

c-ISL (Integer Set Library) : ISL [122] est une bibliothèque C développé par Sven Verdoolaege sécurisée pour les threads permettant de manipuler des ensembles et des relations de points entiers délimité par des contraintes affines. Les descriptions des ensembles et des relations peuvent impliquer à la fois des paramètres et des variables quantifiées de manière existentielle. Tous les calculs sont effectués en arithmétique entière exacte en utilisant GMP (la bibliothèque de calcul sur les grands nombres) ou imath (bibliothèque pour les calculs entiers de précision arbitraire).
La bibliothèque ISL est principalement destinée à être utilisée dans le modèle polyédrique pour l'analyse et la transformation de programmes, mais certaines des nombreuses opérations qu'elle prend en charge ont été utilisées en dehors de ce modèle. Depuis le début, l'un des principaux objectifs à long terme était de fournir toutes les manipulations d'ensembles et de polynômes requises par la bibliothèque barvinok, qui, à cette époque, utilisait une combinaison de : PolyLib [85], PipLib, Omega et GiNaC [23]. ISL déjà atteint les objectifs à court terme de remplacement de PolyLib dans le générateur de boucle CLooG en produisant un meilleur code en éliminant les contraintes redondantes sur les entiers mais non sur les rationnels, et en constituant la base d'un vérificateur d'équivalence des programmes pouvant être représentés dans le modèle polyédrique. Les dernières versions de l'outil Pluto ont été basé sur cette bibliothèque.

3.6 Conclusion

la parallélisation automatique est une approche qui permet de faciliter l'utilisation des machines parallèles. Cette approche consiste à prendre un code développé pour une machine séquentielle et l'adapter pour une machine parallèle. Ce chapitre introduit tout d'abord la parallélisation automatique et ces étapes dans premier temps, ensuite il a décrit ses techniques (on a discuté comme techniques : OpenMP, CUDA, PIPS et OpenCL). La dernière section de ce chapitre présente les différents outils de parallélisation automatique. On a détaillé aussi dans cette section l'outil Pluto qui est un framework de transformation source-to-source basé sur le modèle polyédrique. Pluto utilise le parallélisme afin d'atteindre des meilleurs performances pour un programme. Cet outil est utilisé pour générer le code parallèle et optimisé dans nos études expérimentales (chapitre 5 et 6).

Dans le chapitre suivant, nous allons présenté les différents opérations sur les matrices et les différents formats de stockage des matrices creuses.

Chapitre 4

Les Matrices creuses

Sommaire

4.1 Introduction

Les informations scientifiques, commerciales et mathématiques sont souvent organisées en lignes et colonnes pour former des tableaux rectangulaires appelés «matrices». Les matrices apparaissent souvent sous forme de tableaux de données numériques issues d'observations physiques, mais elles se produisent également dans divers contextes mathématiques. Par exemple, toutes les informations sont nécessaires à la résolution d'un système d'équations et la solution de ce système peut être obtenue en effectuant des opérations appropriées sur ces informations. Ceci est particulièrement important dans le développement de programmes informatiques pour la résolution de systèmes d'équations car les ordinateurs sont bien adaptés à la manipulation de tableaux d'informations numériques. Dans la discipline de calcul matricielle, de nombreux problèmes font intervenir des matrices de grandes tailles qui contiennent beaucoup de coefficients nuls, on parle de matrices creuses (sparse matrices). Afin d'économiser l'espace mémoire, il est souhaitable de ne stocker que les éléments non nuls d'une matrice. Il est alors indispensable de stocker également d'autres informations pour retrouver les lignes et les colonnes de la matrice. Nous présentons dans ce chapitre les notions de base des matrices et les différents types de matrices creuses ainsi que leurs formats de stockage les plus connues.

4.2 Notions sur les matrices

Une matrice est définie comme un tableau rectangulaire d'éléments fonctionnels ou numériques, disposés en lignes ou en colonnes. Le plus important dans cette définition est que (au plus) deux indices sont nécessaires pour identifier un élément donné : un indice de ligne et un indice de colonne. C'est-à-dire qu'une matrice est un tableau à 2 dimensions. La définition inclut des tableaux dans lesquels la valeur maximale d'un, ou des deux indices, est l'unité. Par exemple, une seule "liste" d'éléments, organisée dans une seule ligne ou une seule colonne, est appelée matrice "ligne" ou "colonne". Même un seul élément peut être qualifié de matrice un par un (c'est-à-dire 1×1). A titre d'illustration, une matrice rectangulaire m par n peut être schématisée comme suit :

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

La matrice rectangulaire ci-dessus possède m lignes et n colonnes. En poursuivant cette illustration, nous écrivons un ensemble $m \times n$ équations algébriques linéaires sous la forme :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = c_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = c_2 \\ \quad \ldots \qquad \ldots \qquad \ldots \qquad \ldots \qquad \ldots \quad = \ldots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \ldots + a_{mn}x_n = c_m \end{cases} \tag{4.1}$$

Ce qui précède définit un ensemble de m équations à n inconnues, le point important est de comparer l'ensemble d'équations (4.1) avec la définition de la matrice m lignes par n colonnes ci-dessus. Ce chapitre se concentrera sur les règles de base des matrices, qui nous permettront, entre autres, d'écrire l'ensemble d'équations (4.1) sous la forme :

$$\mathrm{A} \cdot x = c \tag{4.2}$$

Dans lequel la matrice A a la forme schématisée ci-dessus. Dans (4.2), chacun des symboles littéraux représente une matrice. La matrice A est une matrice rectangulaire, avec m lignes et n colonnes. La matrice x a n lignes et une seule colonne. Il est généralement désignée sous le nom de «vecteur», de même que la matrice c, qui a aussi m lignes et une seule colonne. Comme mentionné précédemment, x et c peuvent également être appelés matrices de colonnes (ou vecteurs

de colonnes). On remarquera immédiatement que, bien que le système (4.2) soit compact, il ne transmet pas toutes les informations de (4.1). C'est-à-dire que (4.1) ne rend pas la «dimensionnalité» claire : il n'est pas évident que A soit m lignes par n colonnes. Si l'ensemble (4.2) est «carré» (c'est-à-dire que $m = n$), alors la matrice A aura un «déterminant», écrit |A|, ou $|a_{ij}|$. Cette notation est non seulement pratique, mais elle a également un sens, puisqu'un déterminant, bien qu'écrit sous forme de tableau, s'évalue à une seule valeur fonctionnelle ou numérique (mais on ne doit pas supposer que |A| est nécessairement positif).

$$\det \mathrm{A} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

4.3 Opérations de base sur les matrices

4.3.1 Transposition

La matrice transposée de A s'écrit A^{T}. A^{T} est obtenue en interchangeant les lignes et les colonnes de A , c'est-à-dire, $\mathrm{A}^{\mathrm{T}}_{ij} = \mathrm{A}_{ji}, i = 1, ..., m, j = 1, ..., n$ Ensuite, A($m \times n$) devient A($n \times m$) sous transposition. De même, $v \rightarrow [v]$, c'est-à-dire que la transposée d'une colonne est une ligne, et inversement. L'opération de transposition est très importante dans le domaine de calcul matriciel [2].

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \mathrm{A}^{\mathrm{T}} = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$$

4.3.2 Inverse

Par définition, la matrice inverse A^{-1} annule les effets de la matrice A. L'effet cumulatif de l'application de A^{-1} après A est la matrice Unitaire I :

$$\mathrm{A}^{-1} \cdot \mathrm{A} = \mathrm{I} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

L'inverse de la matrice est utile pour résoudre les équations matricielles. Chaque fois que nous voulons enlever de la matrice A dans certaines équations matricielles, nous pouvons «multiplier» A avec son inverse A^{-1} pour la faire disparaître. Par exemple, pour résoudre la matrice X dans l'équation $\mathrm{X} \cdot \mathrm{A} = \mathrm{B}$, multipliez les deux côtés de l'équation par A^{-1} à partir de la droite : $\mathrm{X} = \mathrm{B} \cdot \mathrm{A}^{-1}$. Pour résoudre X dans $\mathrm{A} \cdot \mathrm{B} \cdot \mathrm{C} \cdot \mathrm{X} \cdot \mathrm{D} = \mathrm{E}$, multipliez les deux côtés de l'équation par D^{-1} à droite et par $\mathrm{A}^{-1}, \mathrm{B}^{-1}$ et C^{-1} (dans cet ordre) en partant de la gauche : $\mathrm{X} = \mathrm{C}^{-1} \cdot \mathrm{B}^{-1} \cdot \mathrm{A}^{-1} \cdot \mathrm{E} \cdot \mathrm{D}^{-1}$ [2].

4.3.3 Multiplication des matrices

Comme les matrices rectangulaires sont composées de vecteurs, nous allons d'abord discuter des produits vectoriels, avant de définir le produit de ces matrices «plus grandes». Le produit le plus important de deux vecteurs est leur «produit scalaire». Ce produit donne un scalaire, tout comme le produit vectoriel point dans l'analyse vectorielle. De plus, le résultat numérique est le même aussi, puisqu'il s'agit de la somme des produits des éléments correspondants [2]. Le produit

vectoriels est défini par :

$$u \cdot v = [u_1 u_2 \dots u_n] \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{j=1}^{n} u_j v_j$$

Notez que les deux vecteurs doivent avoir le même nombre de termes (éléments). C'est-à-dire que les deux vecteurs doivent avoir les «mêmes dimensions». Si tel n'était pas le cas, les deux vecteurs ne seraient pas "Conforme, en multiplication". Le plus important est que le produit scalaire soit toujours vu comme le produit d'un vecteur ligne et d'un vecteur colonne. et son résultat est une matrice (1×1) (c'est-à-dire un scalaire). À cet égard, la notation la plus significative pour le produit vectoriel est $[u]v$ ou $[v]u$. Dans (4.2), le produit $A \cdot x$ est défini comme égal au vecteur c. Le produit d'une matrice rectangulaire et d'un vecteur est un autre vecteur. À partir de (4.1), on verra que (dans $A \cdot x = c$) chaque élément (scalaire) de c est la somme des produits élément par élément d'un vecteur de ligne de A par la colonne x : le premier vecteur de ligne de A est : $[a_1] = [a_{11}, a_{12}, ..., a_{1n}]$. Le produit $[a_1] \cdot x$ est c_1, le premier élément du vecteur c. C'est-à-dire :

$$a_{11}x1 + a_{12}x2 + \dots + a_{1n}xn = \sum_{j=1}^{n} a_{1j}x_j = c_1$$

Dans le cas général, C = A · B , chaque élément de C est le résultat d'un produit scalaire d'une ligne de A et d'une colonne de B. En particulier, l'élément général $c_{ij} = [a_i] \cdot b_j$. Le concept est schématisé dans la figure 4.1.

$$\left[\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & \cdots & & a_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & & & a_{mk} \end{array}\right] \left[\begin{array}{c|c|c|c} b_{11} & b_{12} & & b_{1n} \\ b_{21} & \cdots & & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{k1} & & & b_{kn} \end{array}\right] = \begin{bmatrix} [\mathbf{a}_1]\{\mathbf{b_1}\} & [\mathbf{a}_1]\{\mathbf{b}_2\} & \cdots & [\mathbf{a}_1]\{\mathbf{b}_n\} \\ [\mathbf{a}_2]\{\mathbf{b_1}\} & [\mathbf{a}_2]\{\mathbf{b}_2\} & \cdots & [\mathbf{a}_2]\{\mathbf{b}_n\} \\ \cdots & & & \cdots \\ [\mathbf{a}_m]\{\mathbf{b_1}\} & & \cdots & [\mathbf{a}_m]\{\mathbf{b}_n\} \end{bmatrix}.$$

FIGURE 4.1 – Multiplication des matrices.

4.4 Types des matrices

4.4.1 Matrice Unitaire

Une matrice carrée $(n \times n)$ dont les éléments a_{ij} sont nuls pour $i \neq j$ et dont les éléments a_{ii} sont égaux à 1 est définie comme la «matrice unitaire», I correspond à l'unité en mathématiques scalaires. Par exemple, s'ils sont conformes, $Ix = x$ ou $A \cdot I = A$. Tout comme en algèbre scalaire, la multiplication d'une matrice, $A(n \times n)$,par la matrice unitaire, $I(n \times n)$, laisse A inchangée. Notez également que I = I(I). Dans la matrice unitaire I, les éléments d'unité sont supposés se trouver dans la «diagonale principale». Les éléments «hors diagonale» sont nuls. La matrice d'unité peut également être écrite sous la forme $[\delta_{ij}]$. Le symbole δ_{ij} est appelé «delta de Kronecker» [13]. Par définition,$\delta_{ij} = 1$ pour $i = j$ et $\delta_{ij} = 0$, pour $i \neq j$.

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

4.4.2 Matrice Diagonale

la matrice est dite «matrice diagonale», si les éléments diagonaux principaux ne sont pas des unités, mais que tous les éléments hors de cette diagonale sont nuls. Les éléments diagonaux ne sont généralement pas égaux. Dans les cas où les principaux éléments diagonaux sont égaux, la matrice est appelée «matrice scalaire». Le produit de deux matrices diagonales est une autre matrice diagonale, dont les principaux éléments diagonaux sont les produits des éléments correspondants des deux matrices données. Toutefois, si A n'est pas une diagonale et B est une diagonale, le produit n'est pas commutatif [2].

$$\mathrm{D} = \begin{pmatrix} d_{11} & 0 & \dots & 0 \\ 0 & d_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & d_{nn} \end{pmatrix}$$

4.4.3 Matrice Orthogonale

Les lignes (et/ou) les colonnes d'une matrice orthogonale sont perpendiculaires (orthogonales), au même sens que dans l'analyse vectorielle. C'est-à-dire que le produit scalaire d'une ligne avec une autre est égal à zéro. Un exemple simple est :

$$\mathrm{I} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}$$

Clairement, les lignes et les colonnes du matrice ci-dessus sont orthogonales; leurs produits scalaires sont nuls. Dans le cas de cet exemple, la matrice est également dite «orthonormée», car les longueurs des lignes/colonnes sont normalisées à 1 (c'est-à-dire que le produit scalaire d'une ligne/colonne en elle-même est de 1). La matrice orthogonale a de fréquentes applications dans les problèmes d'ingénierie.

4.4.4 Matrice Complexe

Une matrice Z, dont les éléments sont des nombres complexes, peut être écrite $[z_{ij}]$, où $z_{ij} = x_{ij} + jy_{ij}$ ou $\mathrm{Z} = \mathrm{X} + j\mathrm{Y}$, (où «j» est la notation pour le nombre imaginaire : $j^2 = -1$). Cette dernière forme montre une «séparation» des parties réelle et imaginaire en matrices séparées. Dans cette notation, X et Y sont tous les deux des nombres réels. Une matrice, $\mathrm{W} = \mathrm{X} - j\mathrm{Y}$, est appelée le "conjugué" de Z. La somme, ou le produit, de deux matrices complexes peut être formée de manière simple, élément par élément, à l'aide d'arithmétique complexe ou à l'aide de la seconde notation ($\mathrm{Z} = \mathrm{X} + j\mathrm{Y}$). Par Exemple : $\mathrm{Z_1Z_2} = (\mathrm{X_1X_2} - \mathrm{Y_1Y_2}) + j(\mathrm{X_1Y_2} + \mathrm{Y_1X_2})$.

$$\mathrm{Z} = \begin{pmatrix} x_{11} + jy_{11} & x_{12} + jy_{12} & \dots & x_{1m} + jy_{1m} \\ x_{21} + jy_{21} & x_{22} + jy_{22} & \dots & x_{2m} + jy_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} + jy_{n1} & x_{n2} + jy_{n2} & \dots & x_{nm} + jy_{nm} \end{pmatrix}$$

4.4.5 Matrice Symétrique et matrice Skew-Symétrique

Une matrice qui est inchangée sous transposition est appelée «symétrique». Par exemple, la matrice A ci-dessous est symétrique ($\mathrm{A}^{\mathrm{T}} = \mathrm{A}$).

$$\mathrm{A} = \begin{pmatrix} a & e & f \\ e & b & g \\ f & g & c \end{pmatrix}$$

et nous notons que $a_i = [a_i]$, c'est-à-dire que les lignes et les colonnes correspondantes sont égales. Par exemple, la ligne 1 : $[a, e, f]$ est égal à la colonne 1 : a, e, f. Les matrices symétriques jouent un rôle important dans les problèmes d'ingénierie. Par exemple, les fonctions énergétiques sont généralement symétriques. Pour toute matrice réelle B, le produit $\mathrm{B}^{\mathrm{T}} \cdot \mathrm{B}$ est toujours une matrice carrée symétrique. En d'autres termes, B est $(m \times n)$ et le produit est $(n \times n)$, c'est-à-dire une matrice carrée. Il est évident que $(\mathrm{B}^{\mathrm{T}}\mathrm{B})^{\mathrm{T}} = \mathrm{B}^{\mathrm{T}}\mathrm{B}$, c'est-à-dire que la matrice de produit est symétrique [110].

Dans l'exemple ci-dessous, on a $\mathrm{W}^{\mathrm{T}} = -\mathrm{W}$, alors W est appelé une «matrice Skew-symétrique». Comme les éléments diagonaux principaux sont inchangés lors de la transposition, les éléments diagonaux principaux d'une matrice Skew-symétrique doivet être égaux à zéro.

$$\mathrm{W} = \begin{pmatrix} 0 & w1 & -w2 \\ -w1 & 0 & w3 \\ w2 & -w3 & 0 \end{pmatrix}$$

4.4.6 Matrice Triangulaire

Dans la discipline mathématique de l'algèbre linéaire, une matrice triangulaire est une sorte particulière de matrice carrée. Une matrice carrée est appelée triangulaire inférieure si toutes les entrées au-dessus de la diagonale principale sont nulles. De même, une matrice carrée est appelée triangulaire supérieure si toutes les entrées au-dessous de la diagonale principale sont nulles. Il existe deux types de matrices triangulaires inférieures (ou supérieures) :

Matrice triangulaire impaire : une matrice est dite triangulaire impaire si tous les éléments non nuls sont en dessous ou en dessus de la diagonale de la matrice, et l'ordre de la matrice est un nombre impair $\mathrm{N}(\mathrm{N} = 2k+1)$.

Matrice triangulaire paire : dans le cas d'une matrice triangulaire paire, nous avons la même définition que précédemment, mais l'ordre de la matrice est un nombre pair $\mathrm{N}(\mathrm{N} = 2k)$.

L'exemple suivant illustre deux instances de matrices triangulaires avec des ordres pair (la matrice A) et impair (la matrice B) respectivement.

$$\mathrm{A} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}, \mathrm{B} = \begin{pmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

Ces matrices sont très importantes car : (1) leurs déterminants sont faciles à calculer sous forme de produit de leurs termes diagonaux principaux et (2) leurs inverses sont également facile à déterminer. L'exemple suivant indique la facilité de résolution d'un système d'équations triangulaire :

$$\mathrm{D} = \begin{pmatrix} 1 & 2 & -1 \\ 0 & 3 & 3 \\ 0 & 0 & 5 \end{pmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \\ 5 \end{bmatrix}$$

Comme la dernière équation est «découplée», $x_3 = 1$ par inspection. Une fois que x_3 est connu, x_2 peut être résolu par la deuxième équation($x_2 = 2$), puis on peut déduire la valeur de x_1 à partir la première équation $x_1 = 4$. Il existe de nombreuses méthodes de résolution des déterminants, des systèmes d'équations et des inversions de la matrice qui intègrent la triangularisation de la matrice [110].

4.4.7 Matrice en Bande

Une matrice en bande A est une matrice dont les éléments non nuls sont situés dans une bande centrée le long de la diagonale principale. Pour de telles matrices, seule une petite proportion des éléments N^2 sont non nuls. Une matrice carrée A a une bande inférieure $W_l < N$ et une bande supérieure $W_u < N$ si W_l et W_u sont les plus petits entiers tels que : $A_{ij} = 0; \forall i > j + W_l$, et $A_{ij} = 0; \forall j > i + W_u$. Le nombre maximal d'éléments non nuls dans une ligne est $W = W_l + W_u + 1$. L'exemple suivant montre une matrice en bande d'ordre $N = 6$ avec une bande inférieure $W_l = 1$ et une bande supérieure $W_u = 2$.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{pmatrix}$$

4.4.8 Matrice creuse

Une matrice creuse est une matrice composée principalement de valeurs nulles. Les matrices creuses sont distinctes des matrices avec des valeurs généralement non nulles, appelées matrices denses. Une matrice est creuse si plusieurs de ses coefficients sont nuls. L'intérêt pour la creusité (sparsity) vient de ce que son exploitation peut générer d'énormes économies de calcul et que de nombreux problèmes matriciels importants qui se posent dans la pratique sont creuses [65, **?**, 60]. La sparsity d'une matrice peut être quantifiée avec un score, qui correspond au nombre de valeurs nulles dans la matrice divisé par le nombre total d'éléments de la matrice : .

$$sparsity = \frac{\#\ valeurs\ nulles}{\#\ elements\ de\ la\ matrice} \tag{4.3}$$

ci-dessous un exemple de petite matrice creuse de (3×4) :

$$A = \begin{pmatrix} 5 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix}$$

L'exemple présente 8 valeurs nulles sur les 12 éléments de la matrice, ce qui donne à cette matrice un score de sparsity de 0,66 environ 66 %.

4.5 Format de stockage des matrices creuses

En général, une matrice creuse est une matrice contenant principalement des éléments nuls. Une définition courante est qu'une matrice est creuse chaque fois qu'il est avantageux d'utiliser des formats de stockage spéciaux qui ne stockent que les éléments non nuls de la matrice. L'avantage peut donc être une demande de mémoire réduite de la matrice, mais également une réduction de l'effort de calcul lors de l'utilisation de la matrice pour des calculs mathématiques. Les opérations mathématiques, qui doivent être exécutées sur la matrice, sont également importantes lors de la sélection d'un format de stockage . Dans les sections suivantes, nous citons les formats de stockage des matrices creuses les plus connues .

4.5.1 Le Format Coordinate (COO)

Le format COO [112, 12, 94] est l'un des formats les plus simple de stockage matriciels rarement utilisée. Trois vecteurs sont nécessaires pour stocker la matrice de manière compressée. Deux vecteurs sont utilisés pour stocker les informations de ligne et de colonne de chaque élément non nul de la matrice. Le troisième vecteur stocke lui-même les valeurs non nulles. La figure

4.2 illustre un exemple de matrice et les données obtenues à l'aide du schéma de stockage COO. Le terme NNZ est utilisé pour décrire le nombre d'éléments non nuls dans la matrice. Il convient également de mentionner qu'il n'existe aucun ordre spécifique dans lequel les éléments doivent être stockés dans ce format. L'ordre indiqué dans la figure 4.2 est utilisé uniquement à des fins de présentation.

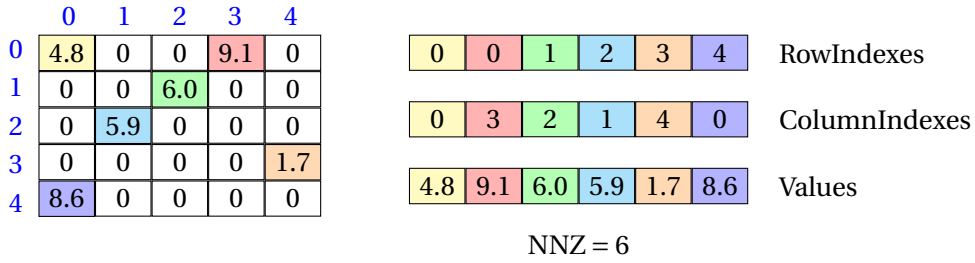


FIGURE 4.2 – Le format de stockage COO.

L'avantage le plus important du format COO est sa structure simple, ce qui le rend très facile à créer. Il est également possible d'ajouter de nouveaux éléments en développant les vecteurs existants. Mais le format présente également des inconvénients importants. Il n'est pas possible d'itérer sur les lignes ni sur les colonnes de la matrice stockée, ce qui peut empêcher des implémentations parallèles efficaces d'opérations de matrice. Il est également impossible de rechercher efficacement un élément spécifique de la matrice, car chaque élément peut être stocké à une position quelconque dans les vecteurs. Le format COO a moins d'importance pour les calculs réels et il est plus souvent utilisé comme format d'échange de matrice très simple.

4.5.2 Le format Compressed Sparse Row (CSR)

Le format CSR [112, 127, 94] est l'un des formats de stockage des matrices creuses les plus couramment utilisés. La figure 4.3 décrit une matrice représentée au format CSR. Très similaire au format COO, il nécessite trois vecteurs pour stocker les données de la matrice. Deux sont identiques à ceux utilisés par COO, l'un contient toujours les indices des colonnes et l'autre sert à stocker les valeurs non nulles. Les éléments sont stockés en utilisant un ordre de lignes majeur, ce qui est important pour comprendre la structure du dernier vecteur. Le n^{eme} élément du vecteur pointe vers le premier élément de la n^{eme} ligne. Cela permet un accès très rapide à tous les éléments d'une ligne donnée. De plus, le vecteur peut être utilisé pour calculer le nombre d'éléments non nuls par ligne. Cela peut être fait en soustrayant le décalage de la ligne du décalage de la ligne suivante. Le vecteur contient un élément supplémentaire à la fin, qui pointe directement derrière le dernier élément de la dernière ligne. De ce fait, le nombre d'éléments de la dernière ligne peut également être calculé. La variable supplémentaire N_{rows} est nécessaire pour le calcul de la demande en mémoire pour le format CSR. N_{rows} décrit le nombre de lignes de la matrice stockée. Le format CSR peut être traversé par des lignes. Le format CSC (Compressed Sparse Column) doit également être mentionnée ici, un schéma de stockage très similaire à celui de la CSR. Les éléments sont stockés en utilisant l'ordre colonne-majeur au lieu de l'ordre ligne-majeur. Au lieu des indices de colonne, les indices de ligne sont stockés. Cela permet au format CSC d'être parcouru par ses colonnes.

4.5.3 Le format Diagonale (DIA)

Le format DIA [112, 127, 94] est optimal lorsque les éléments non nuls sont limités à un petit nombre de diagonales de la matrice. Les avantages de cette méthode sont la réduction des besoins en mémoire et la réduction du transfert de données. Les inconvénients de ce format quand la matrice creuse n'a pas de motif diagonal et beaucoup d'éléments zéro sont stockés en mémoire. La figure 4.4 affiche la matrice d'origine à gauche, avec des éléments de couleur différente de zéro pour la visualisation et le vecteur de stockage. Le vecteur de stockage place les valeurs des

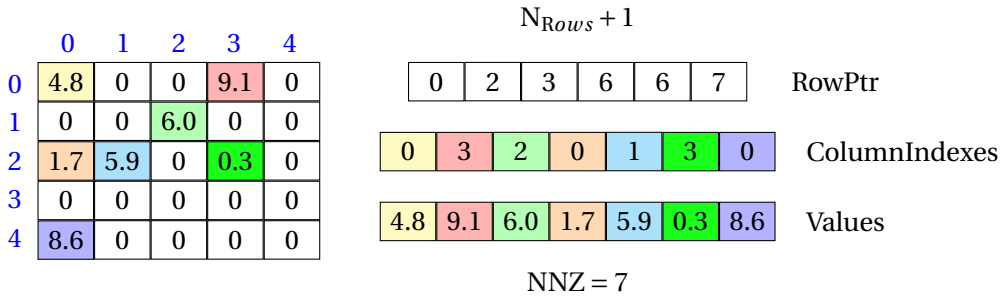


FIGURE 4.3 – Le format de stockage CSR.

éléments dans une matrice de taille 6 × 3 car il y a 6 lignes et 3 diagonales. Les * symbolisent le remplissage de la matrice.

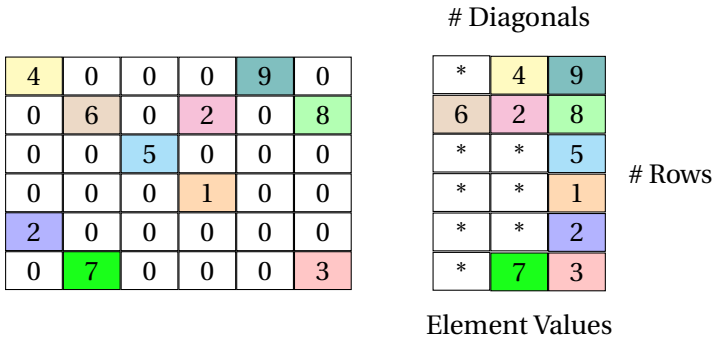


FIGURE 4.4 – Le format de stockage DIA.

4.5.4 Le format ELLPACK (ELL)

Dans Le format ELLPACK (ELL) [8, 127, 94] et pour l'exemple précédent on stocke les indications de colonne dans une matrice de taille 6 × 2 car il y a 6 lignes et un maximum de 2 entrées par ligne. Au total, il y a 6 colonnes, avec les indicateurs de colonne comptés (1,2, ..., 6). L'élément 1, par exemple, se trouve dans la deuxième colonne, ce qui donne une valeur de 2 dans la matrice des indications de colonne. De même, l'élément 3 est dans la sixième colonne, ce qui donne une valeur de 6 dans la matrice des indications de colonne. La seconde matrice, similaire au format DIA, stocke les valeurs des éléments. Contrairement au format DIA où le remplissage est placé à gauche, le format ELL est placé à droite. ELL est implémenté plus souvent que DIA, car les éléments de colonne non nuls ne doivent suivre aucun modèle. Ce format est illustré sur la figure 4.5.

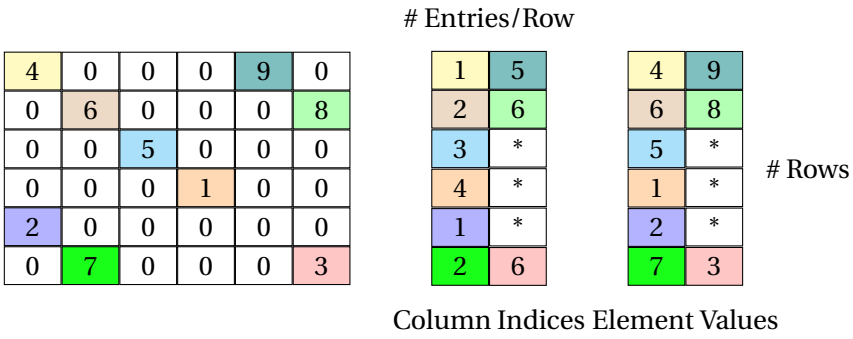


FIGURE 4.5 – Le format de stockage ELL.

4.5.5 Le format ELLPACK-R (ELL-R)

Actuellement, on utilise le plus souvent le format ELLpack-R (ELL-R) [79]. La figure 4.6 illustre une matrice stockée au format ELL-R. Trois vecteurs sont nécessaires pour stocker les données de la matrice. Le premier vecteur stocke le nombre d'éléments non nuls qui existent dans chaque ligne. Les deux autres vecteurs sont à nouveau utilisés pour stocker les indices de colonne et les valeurs non nulles. Un ordre spécifique est utilisé pour stocker les éléments de la matrice. Dans une première étape, la matrice est réduite à ses éléments non nuls uniquement . La taille de la matrice résultante dépend du nombre de lignes et de la ligne la plus longue (qui contient plus d'éléments non nuls). Tous les éléments sont décalés vers la gauche, les entrées manquantes sont remplies de zéro, appelé remplissage . Les éléments de cette matrice réduite sont ensuite stockés dans les vecteurs en utilisant l'ordre de la colonne principale. La seule différence entre le format ELL-R et le format ELL traditionnel est l'utilisation du tableau de largeur supplémentaire.

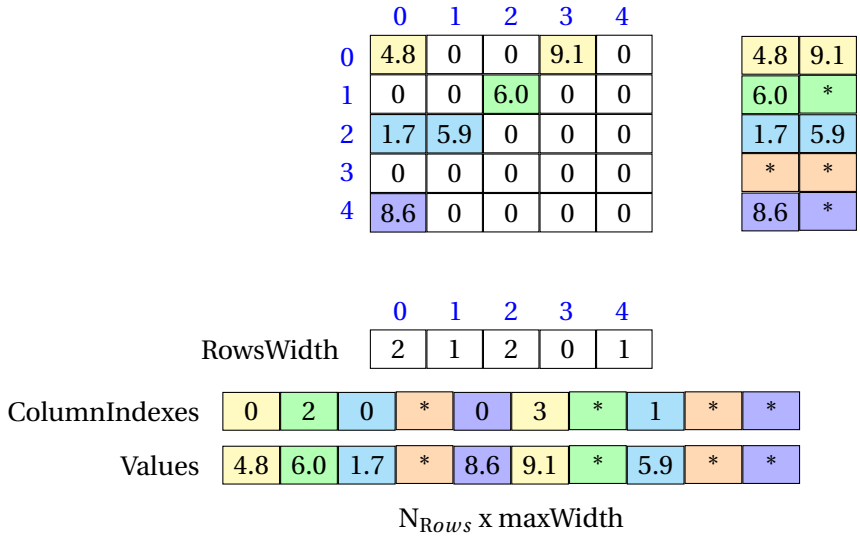


FIGURE 4.6 – Le format de stockage ELL-R.

4.5.6 Le format Jagged Diagonal (JAD)

Le format JAD (Jagged Diagonal) [92, 28] peut être considéré comme une généralisation du format Ellpack qui supprime l'hypothèse des lignes de longueur fixe. Pour construire la structure diagonale irrégulière, commencez par la structure de données CSR et triez les lignes de la matrice en diminuant le nombre d'éléments non nuls. Pour construire le premier "j-diagonal", extrayez le premier élément de chaque ligne de la structure de données CSR. Le second "j-diagonal" est constitué des seconds éléments de chaque ligne de la structure de données CSR. Le troisième, quatrième,. . ., "j-diagonales" peuvent alors être extraits de la même manière. Les longueurs des "j-diagonales" successives diminuent. Le nombre de "j-diagonales" pouvant être extraites est égal au nombre d'éléments non nuls de la première ligne de la matrice permutée, c'est-à-dire au plus grand nombre d'éléments non nuls par ligne. Pour stocker cette structure de données, trois tableaux sont nécessaires : un tableau DJ réel pour stocker les valeurs non nuls de la matrice, le tableau JDIAG qui stocke les positions de colonne de ces valeurs et un tableau de pointeurs IDIAG pointant au début de chaque j-diagonal dans le tableau JDIAG. Considérons la matrice A de la figure 4.7 et sa version triée PA. Les lignes de PA ont été obtenues à partir de celles de A en les triant par nombre d'éléments non nuls, du plus grand au plus petit. Ensuite, la structure de données JAD pour A est illustré dans la figure 4.8.

	0	1	2	3	4
0	4.8	0	0	9.1	0
1	0	0	6.0	0	0
2	1.7	5.9	0	3.4	0
3	0	0	0	0	0
4	8.6	0	0	0	0

A

	0	1	2	3	4
0	1.7	5.9	0	3.4	0
1	4.8	0	0	9.1	0
2	0	0	6.0	0	0
3	8.6	0	0	0	0
4	0	0	0	0	0

P_A

FIGURE 4.7 – Matrice permutée de format JAD.

IDIAG	0	4	6	7			
JDIAG	0	0	2	0	1	3	3
DJ	1.7	4.8	6.0	8.6	5.9	9.1	3.4

NNZ = 6

FIGURE 4.8 – Le format de stockage JAD.

4.5.7 Le Format Linear Packed (LP)

Les figures 4.9, 4.10 et 4.11 illustrent trois formes couramment utilisées en algèbre linéaire pour représenter des matrices creuses spéciales. Ces matrices nécessitent un traitement spécial car elles contiennent un grand nombre d'éléments nuls et que leur emplacement est connu avant l'exécution. Pour gagner de l'espace, seuls les éléments non nuls sont stockés pour ces matrices, on dit que les matrices sont stockées sous leur forme Linear Packed [43]. Dans ce qui suit, nous examinons les formes Linear Packed pour chacun des trois types spéciaux de matrices.

Matrice triangulaire : Dans ce cas, les éléments non nuls sont au-dessous de la diagonale de la matrice, appelée matrice triangulaire inférieure, ou au-dessus de la diagonale de la matrice, appelée triangulaire supérieure. Lorsqu'ils sont stockés dans la représentation Linear Packed, seuls les éléments non nuls d'un matrice triangulaire de taille $N \times N$ doivent être sauvegardées dans un tableau de taille ($\sum_{k=1}^{N} k = N \times N/2$). La figure 4.9 présente un exemple de stockage en format Linear Packed pour les matrices triangulaires.

A	0	0	0	0
B	C	0	0	0
D	E	F	0	0
G	H	I	P	0
K	L	M	N	O

Matrice triangulaire

A	B	C	D	E	F	G	H	I	P	K	L	M	N	O

Format Linear Packed

FIGURE 4.9 – Le format Linear Packed d'une matrice triangulaire.

Matrice en bande : Tous les éléments non nuls de ce type de matrices sont positionné dans une bande étroite bornée diagonalement par deux constantes $W_l, W_u \geq 0$ tel que $a_{ij} \neq 0$ si et

seulement si : $i - W_l \leq j \leq i + W_u$. Lorsqu'elle est stockée sous forme Linear Packed, une matrice en bande ne nécessite qu'un tableau de taille $N \times (W_l + W_u + 1)$, comme illustré à la figure 4.10.

A	B	0	0	0
C	D	E	0	0
F	G	H	I	0
0	R	K	L	M
0	0	N	O	P

Matrice en bande $W_l = 2$ et $W_u = 1$

0	0	A	B	0	C	D	E	F	G	H	I	R	K	L	M	N	O	P	0

Format Linear Packed $size = N \times (W_l + W_u + 1)$

FIGURE 4.10 – Le format Linear Packed d'une matrice en bande.

Matrice triangulaire en bande : C'est une forme spéciale de la matrice en bande dont tous les éléments non nuls sont situés au-dessus ou au-dessous de la diagonale principale, c'est-à-dire qu'il s'agit d'une matrice à bandes avec $W_l = 0$ ou $W_u = 0$ comme illustré à la figure 4.11.

A	0	0	0	0
B	C	0	0	0
D	E	F	0	0
0	G	H	I	0
0	0	K	L	M

Matrice triangulaire en bande $W_l = 2$ et $W_u = 0$

0	0	A	0	B	C	D	E	F	G	H	I	K	L	M

Format Linear Packed $size = N \times (W_l + 1)$

FIGURE 4.11 – Le format Linear Packed d'une matrice triangulaire en bande.

4.5.8 Le Format Rectangular Full Packed (RFPF)

Le format RFPF [67] était aussi proposé pour des matrice creuses particulière, c'est une nouvelle structure de données pour stocker les matrices triangulaires, symétriques et hermitiennes. Les tableaux bidimensionnels standard de Fortran et C (également appelé format complet) utilisés pour représenter des matrices triangulaires et symétriques consomment beaucoup d'espace de stockage, mais offrent des performances élevées grâce à l'utilisation de BLAS (Basic Linear Algebra Subprograms) de niveau 3. Les matrices de format Packed réduisent l'utilisation de l'espace de stockage mais offrent des performances faibles car il n'y a pas de BLAS Packed de niveau 3. Le format RFPF permet de combiner les bonnes caractéristiques de stockage Packed et complet pour obtenir des performances élevées en utilisant BLAS de niveau 3, car RFPF est une représentation standard en format complet. En outre, RFPF nécessite exactement le même stockage minimal que le format Packed. L'idée est de stocker seulement les éléments non nuls d'une matrice triangulaire ($(N \times (N+1))/2$ éléments) dans un tableau de deux dimension en utilisant les routines de la bibliothèque LAPACK selon la parité de l'ordre de la matrice N. Les deux figures 4.13 et 4.12 présentent la structure de format RFPF pour les matrices dans les deux cas N pair et N impair.

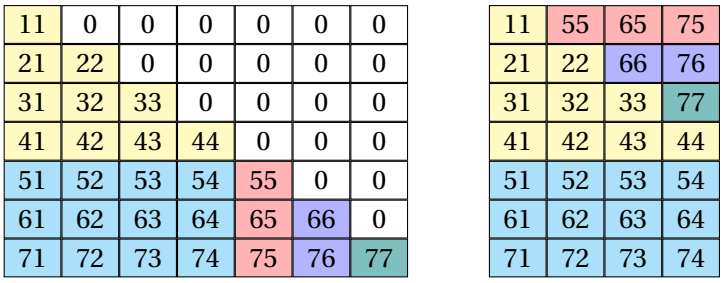


FIGURE 4.12 – Le format RFPF pour une matrice triangulaire d'ordre N impair

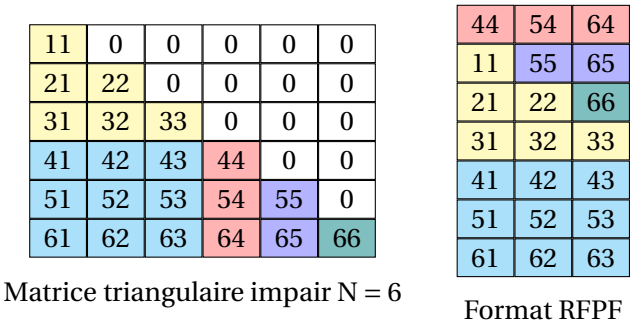


FIGURE 4.13 – Le format RFPF pour une matrice triangulaire d'ordre N pair.

4.6 Conclusion

L'algèbre linéaire est un langage universel qui sert à décrire de nombreux phénomènes en électronique, mécanique et économie. Le calcul matriciel est l'un des éléments les plus importants dans l'algèbre linéaire. Il est basé principalement sur la notion des matrices. Nous avons invoqué dans ce chapitre les différents opérations et les types de matrices. Parmi les type de matrice on a expliqué les matrices creuses. Une matrice creuse en simple mots est une matrice contenant beaucoup d'éléments nuls. On peut retrouver ce type de matrice dans plusieurs problématiques comme : Théorie des graphes, résolution d'équations aux dérivées partielles et Système de recommendations (netflix). Nous avons introduit dans ce chapitre aussi les formats élémentaires existants pour stocker les matrices creuses.

Dans le cinquième chapitre nous avons proposé une approche qui permet de stocker des matrices creuses particulières (matrices triangulaires et matrices en bande) dans des structures de données à deux dimension afin d'atteindre des meilleurs performances pour plusieurs codes d'algèbre linéaire, cette approche est appelé le format 2d-Packed.

Chapitre 5

L'approche proposée : Format 2d-Packed

Sommaire

5.1 Introduction

L'optimisation des programmes d'algèbre linéaire est l'un des problèmes qui a attiré l'attention de la communauté de recherche en optimisation de code depuis de nombreuses années. Presque toutes les techniques d'optimisation existantes visent des opérations sur des structures matricielles denses, dans lesquelles les références de tableau et les bornes d'indice de boucles sont des fonctions affines. Cependant, les opérations sur les matrices creuses sont les noyaux de calculs clés dans de nombreuses applications scientifiques et d'ingénierie. Il est bien connu que les structures matricielles denses et les algorithmes sont très inefficaces lorsqu'ils sont appliqués à des matrices creuses, car ils utilisent une grande quantité de mémoire pour stocker des éléments nuls et effectuer des calculs inutiles sur eux.

Par conséquent, plusieurs formats de stockage alternatifs ont été proposés afin de stocker et de calculer uniquement des éléments non nuls. Parmi ces formats, on peut citer par exemple des format pour représenter les matrices triangulaires : (a) Linear Packed Format (LPF) [43] où la matrice est stockée dans un tableau unidimensionnel utilisé dans la bibliothèque LINPACK [50], l'inconvénient de l'utilisation de ce format est que les références de tableau ne sont plus des fonctions affines, et donc que le modèle polyédrique ne s'applique plus à elles ; (b) Rectangular Full Packed Format (RFPF) [67] où une matrice symétrique ou triangulaire est enregistrée en format rectangulaire et d'autres routines de factorisation de Cholesky ont été proposées dans la bibliothèque LAPACK [5].

Dans ce chapitre, nous proposons une nouvelle approche pour optimiser les opérations matricielles triangulaires et en bande en utilisant une structure de données dense à deux dimensions pour le stockage de matrices creuses [15]. L'idée de base est que les opérations matricielles utilisant ces structures de données peuvent être automatiquement optimisées et parallélisées au moyen du modèle polyédrique. D'une part, les matrices triangulaires et en bande sont comprimées, ce qui permet une économie importante de mémoire, et d'autre part, le code sous-jacent peut être optimisé et parallélisé en utilisant les outils d'optimisation polyédriques existants pour obtenir les meilleures performances. Nous démontrons par des résultats expérimentaux l'efficacité de notre approche en parallélisant et en optimisant plusieurs codes de calcul matriciel en utilisant le compilateur polyédrique Pluto, et en comparant leurs performances aux versions séquentielles et parallèles non-denses, la version LPF, et avec la bibliothèque MKL.

5.2 Exemple de motivation

Dans cet exemple de motivation, nous nous intéressons à l'optimisation automatique et à la parallélisation de la routine sspfa (figure 5.1) de la bibliothèque LINPACK. Le type de données de base est double. Dans cet exemple, les dépendances de données complexes empêchent l'optimisation et la parallélisation automatiques des compilateurs standard, d'où la nécessité d'un compilateur polyédrique : des transformations de boucles complexes sont nécessaires pour exposer le parallélisme et le tiling.

```
1 for (j=0;j<n;j++)
2  for (k=0;k<j;k++)
3    for (i=0;i<k;i++)
4       A[j][k]= A[j][k] - A[j][i] . A[k][i] ;
```

FIGURE 5.1 – Le code sspfa.

Quand une matrice d'ordre n est triangulaire, il n'est évidemment pas approprié de la stocker dans une structure de n^2 éléments, puisqu'il y aura $\frac{n\cdot(n-1)}{2}$ éléments nuls qui ne sont pas requis. Les solutions existantes utilisent généralement une disposition linéaire, où les éléments non nuls

sont stockés dans un tableau de dimension unique (un vecteur) de taille $\frac{n\cdot(n+1)}{2}$. Le code sspfa correspondant pour une telle structure est représenté sur la figure 5.2. On peut remarquer que les fonctions d'accès au tableau de la figure 5.2 sont affines mais en fonction de variables qui ne le sont pas, et ne sont donc pas supportées par le modèle polyédrique. Par conséquent, les compilateurs statiques polyédriques, tels que Pluto, ne peuvent pas automatiquement optimiser et paralléliser les calculs matriciels en utilisant cette structure de données unidimensionnelle pour stocker des éléments non nuls. Cependant, nous aimerions exécuter un compilateur polyédrique sur ce code, car les dépendances de données complexes l'empêchent d'être facilement parallélisé, tilé, vectorisé et optimisé pour la localité mémoire. Dans notre travail, nous avons introduit une conversion simple d'une matrice triangulaire (non packed) en un format 2d-Packed et des transformations correspondante qui sont affines et adaptées au modèle polyédrique. En utilisant notre approche, la routine sspfa est réécrite comme indiqué sur la figure 5.3. Dans le code transformé, toutes les références de tableau, les limites de boucles et les tests sont affines, ce qui signifie que ce nouveau code peut être automatiquement optimisé et parallélisé par des compilateurs polyédriques.

```
1  //stocker la matrice A dans le vecteur AP
2  ii=0;
3  for (i=0; i<n; i++)
4    {
5     ii+=i;
6     for(j=0; j<i; j++)
7       AP[ii+j] = A[i][j];
8    }
9  //calcul
10 jj=0;
11 for (j=0;j<n;j++)
12  {
13    kk=0;
14    jj+=j;
15    for (k=0;k<j;k++)
16     {
17       kk+=k;
18       for (i=0;i<k;i++)
19        A[jj+k]= A[jj+k] - A[jj+i] . A[kk+i] ;
20     }
21   }
```

FIGURE 5.2 – Le code sspfa dans le format linéaire Packed.

Pour montrer l'efficacité de la nouvelle version 2d-Packed du code, nous avons lancé Pluto (avec les options –tile –parallel), puis nous avons compilé le code résultant en utilisant icc (version 18.0.0, avec les options - O3 -march = native). L'exécutable ainsi obtenu est ensuite exécuté sur un processeur Intel de 20 cœurs. Pour $n = 4000$, nous avons constaté que le temps d'exécution de la nouvelle version est environ douze fois plus rapide que la version LPF : il passe de 11.28 secondes à 0.95 secondes. Le compilateur icc n'a pas pu auto-paralléliser la version LPF - il a les mêmes performances que le séquentiel - . La sortie Pluto du code 2d-packed est de 140 lignes (voir l'annexe de la thèse) : le skewing et le tiling sont appliqués sur le code (en utilisant la taille de la tuile par défaut : $32 \times 32 \times 32$), et le résultat parallélisé en utilisant OpenMP ; aucune vectorisation supplémentaire n'a été détectée par Pluto. Le compilateur icc n'a pas pu auto-paralléliser le code original, ni le code 2d-packed, nous avons donc besoin d'un compilateur polyédrique pour transformer ce code. D'autres mesures pour cet exemple sont rapportées dans la section expérimentale, colonne sspfaTri de la figure 5.24a.

```
//stocker la matrice A dans la matrice A1
for (i=0; i<n; i++)
  for (j=0; j<=i; j++)
  {
    if (2*i>n && 2*j>n)
     A1[n-i-1][n-j]= A[i][j];
    else
     A1[i][j]=A[i][j];
   }
//calcul
for (j=0;j<n;j++)
  for (k=0;k<j;k++)
    for (i=0;i<k;i++)
     {
      if (2*i>n )
       A[n-j-1][n-k]= A[n-j-1][n-k] - A[n-j-1][n-i] . A[n-k-1][n-i] ;
      if (2*i<=n && 2*k>n)
       A[n-j-1][n-k]= A[n-j-1][n-k] - A[j][i] . A[k][i] ;
      if (2*i<=n && 2*k<=n)
       A[j][k]= A[j][k] - A[j][i] . A[k][i] ;
     }
```

FIGURE 5.3 – Le code sspfa dans le format 2d-Packed.

5.3 Autres travaux connexes

Les calculs matriciels sont la pierre angulaire de nombreuses applications scientifiques et d'ingénierie, et les performances de presque toutes les applications matricielles dépendent de celles des calculs matriciels. Par conséquent, un grand intérêt des chercheurs a été orienté vers l'optimisation de ce type de calculs. Cependant, la plupart des chercheurs en optimisation ont ciblé des calculs sur des matrices denses en raison de la régularité de leurs schémas de stockage de données et des calculs sous-jacents, en plus de leur large utilisation [49]. Au cours des trois dernières décennies, de nombreuses techniques d'optimisation manuelle et automatique ont été proposées.

Les techniques d'optimisation manuelles reposent sur l'effort du programmeur expert, après une analyse approfondie d'un problème donné, le programmeur propose un programme réglé manuellement dont la performance est la meilleure possible. De nombreuses techniques d'optimisation de calcul matriciel manuelles ont été conçues [37, 71, 123] .
Alternativement, les techniques d'optimisation automatique consistent à concevoir et implémenter des outils et des compilateurs capables de traduire un programme donné en un code optimisé et parallélisé. Le programmeur n'a pas besoin de s'inquiéter du processus d'optimisation, tout ce qu'il doit savoir, c'est comment utiliser ces outils. Les techniques d'optimisation automatique appliquées aux calculs matriciels sur les matrices denses incluent Pluto , EPOD [42] , APOLLO [88] et PHiPAC [27] .

Les techniques d'optimisation ciblant les calculs matriciels sur des matrices creuses, telles que les matrices triangulaires et en bandes, sont rares et presque toutes sont des techniques d'optimisation manuelles appliquées à des problèmes spécifiques [1, 4, 93]. Parmi ces travaux, Gustavson et al. [67] ciblent l'algorithme de Cholesky sur les matrices triangulaires, en remplaçant les calculs sur les matrices creuses par un ensemble de calculs sur des matrices denses de taille inférieure (égale à la moitié de la taille originale) , en appelant les routines BLAS de niveau 3. Notre format de représentation pour les matrices triangulaires est inspiré de ce travail, où la matrice est coupée en deux parties, le petit triangle étant déplacé pour couvrir les zéros dans l'autre partie (comme présenté ci-dessous dans la figure 5.6). Nous avons amélioré leur format de stockage, afin d'effectuer une transformation de données unique pour les matrices de tailles paires et impaire : dans leur proposition, les matrices impaires et paires sont stockées différemment, ce qui oblige le program-

meur à distinguer ces deux cas. Par rapport à leur travail, nous proposons également un nouveau format de stockage en bande.

Récemment, Cui et al. [43] ont proposé une technique automatique d'optimisation pour les calculs matriciels. A de nos connaissances, leur travail est le seul à utiliser un compilateur polyédrique pour optimiser les calculs matriciels triangulaires. Leur idée principale est l'isolation de la sémantique des opérations de haut niveau à partir des détails d'organisation des structures de données composées. De cette façon, une spécification abstraite simplifiée des opérations peut être dérivée, puis analysée et optimisée avec précision en utilisant les compilateurs optimiseurs tels que Pluto et EPOD .
Plus précisément, leur approche comporte trois étapes : la normalisation de la matrice, l'optimisation et la dénormalisation de la matrice. La première étape cherche à isoler la sémantique de haut niveau des opérations matricielles de l'implémentation interne des structures de données. Ceci à son tour consiste à dériver un nouveau code abstrait traitant les structures de données non Packed (matrices denses) à partir des structures de données empaquetées manipulant le code original. Dans la deuxième étape, le code abstrait résultant est transmis au compilateur source-to-source pour générer un code parallèle optimisé. Ce code optimisé est ensuite converti à travers l'étape de dénormalisation afin de l'adapter à l'organisation de la structure de données d'origine. Cela consiste principalement à convertir les accès groupés denses bidimensionnels en accès condensés unidimensionnels. Les auteurs proposent un langage d'annotation que le programmeur doit utiliser pour définir la sémantique voulue des structures de données dans leurs calculs matriciels, pour effectuer les étapes 1 et 3 de leur algorithme. Même si le framework de Cui et al. est bien conçu et s'avère efficace dans de nombreux cas, nous pensons que les programmeurs pourraient avoir des difficultés à l'utiliser. D'une part, parce qu'ils doivent écrire les calculs matriciels en bandes et en triangles (voir la figure 5.3 par exemple), qu'ils ne connaissent peut-être pas. Et d'autre part, parce qu'ils doivent exprimer la sémantique voulue des structures de données de leurs calculs matriciels dans le langage d'annotation, ce qui peut conduire à des erreurs.
Les auteurs affirment en effet dans leur article que le code optimisé est garanti si les annotations fournies par l'utilisateur peuvent être supposées correctes. Ils mentionnent également que leur approche est efficace pour optimiser uniquement les calculs matriciels condensés où la disposition matricielle peut être facilement exprimée avec leur langage d'annotation. Dans notre proposition, tout ce que les programmeurs ont à faire est d'écrire les calculs matriciels dans un format matriciel non compressé (tableaux bidimensionnels) et de les déclarer triangulaires ou en bandes. L'ensemble du processus d'optimisation est totalement transparent et automatique à partir de ce point.

Très récemment, Sampaio et al. [111] ont suggéré d'utiliser un framework hybride complexe (statique et dynamique) afin de surmonter les limites des techniques d'analyse de dépendance purement statiques. Contrairement à notre approche, dans laquelle nous proposons de modifier la disposition des données pour des cas spécifiques afin que les codes sous-jacents soient faciles à analyser, le travail de Sampaio et al. a tenté d'optimiser des programmes non affines plus généraux. Mais il se concentre dans sa forme actuelle, en démontrant la faisabilité et l'impact potentiel de l'utilisation de transformations polyédriques sur des programmes non-affines. L'évaluation de leur approche pour l'accès polynomial résultant du stockage compacté de matrices triangulaires est laissée à un travail futur.

5.4 La Technique d'optimisation pour le format 2d-Packed

Dans cette section, nous décrivons notre technique d'optimisation pour format 2d-packed. Cette technique consiste à prendre en entrée les matrices creuses à manipuler et un code dense original. Dans un premier temps, notre technique définit les matrices creuses dans des formats 2d-

Packed, où seuls les éléments non nuls sont stockés (les matrices 2d-packed sont allouées dynamiquement) [15]. Ensuite, le code original pour les matrices denses est transformé en un nouveau code en utilisant les structures 2d-packed. Le code résultant est finalement parallélisé et optimisé par le paralléliseur et l'optimiseur source-à-source Pluto. Un aperçu de notre approche est illustré à la figure 5.4.

5.4.1 L'approche 2d-Packed pour les matrices triangulaires

Gustavson et al. [67] proposent d'utiliser un format rectangulaire complet pour stocker une matrice triangulaire d'ordre N. La taille du rectangle est $N \times \frac{(N+1)}{2}$ lorsque N est impair, et $(N+1) \times \frac{N}{2}$ sinon (voir la figure 5.5). Même si cette représentation garantit l'obtention d'une structure rectangulaire complète dans les deux cas, elle présente l'inconvénient d'ajouter une ligne supplémentaire dans le cas où N est pair. Cela conduit à écrire deux versions du code en utilisant deux fonctions de transformation distinctes, selon que N est impair ou pair. Gustavson et al. proposent également des transformations matricielles pour la factorisation de Cholesky afin de s'adapter à la nouvelle structure de données et à leur implémentation dans la bibliothèque LAPACK [5].

Dans notre travail, nous proposons d'utiliser exactement la même structure de données lorsque N est impair. C'est-à-dire qu'une matrice de taille $N \times \frac{(N+1)}{2}$ (N lignes et $\frac{(N+1)}{2}$ colonnes) est requise. Lorsque N est pair, nous proposons une structure de données différente où une matrice de taille $N \times \frac{N}{2}$ (N lignes et $\frac{N}{2}$ colonnes) et une colonne supplémentaire de taille $\frac{N}{2}$ sont utilisées pour stocker les valeurs non nulles (voir Figure 5.6). Cette nouvelle structure possède l'avantage de conserver le même code que dans le cas impair et une seule fonction de transformation est requise.
L'algorithme 1 montre la façon dont nous stockons une matrice triangulaire dans le format 2d-packed.

Afin de pouvoir utiliser la nouvelle structure 2d-packed, nous avons proposé une fonction de transformation unimodulaire par morceaux, que l'on peut utiliser pour transformer toute opération matricielle manipulant des matrices triangulaires. Dans notre travail, nous avons appliqué cette transformation à de nombreux benchmarks de calcul matriciel. Les codes résultants sont ensuite optimisés et parallélisés par l'outil de parallélisation et d'optimisation automatique Pluto. L'algorithme 2 illustre la transformation de code, en utilisant la fonction unimodulaire, de toutes les déclarations dans lesquelles une matrice triangulaire est référencée. Pour simplifier, nous supposons, sans perte de généralité, que l'instruction ne contient qu'une seule référence (x, y) à une matrice triangulaire. Cette référence est transformée en format 2d-packed comme montré dans l'équation 5.1.

$$f_t(x,y) = \begin{cases} (-x+N-1, -y+N) & \text{si } y > \frac{N}{2} \\ (x,y) & \text{sinon} \end{cases} \tag{5.1}$$

Cette fonction de transformation unimodulaire peut être réécrite dans la représentation matricielle comme illustré dans l'équation 5.2

$$f_t(x,y) = \begin{cases} \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} N-1 \\ N \end{pmatrix} & \text{si } y > \frac{N}{2} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \text{sinon} \end{cases} \tag{5.2}$$

5.4.1.1 Représentation polyédrique

Soit l'exemple montré dans la figure 5.7 permettant de calculer le produit d'une matrice triangulaire par un vecteur.

Après l'application de notre transformation 2d-Packed pour les matrices triangulaires on a trouvé le code présenté dans la figure 5.9. La représentation polyédrique de ce code pour n=6 est illustrée dans la figure 5.8.

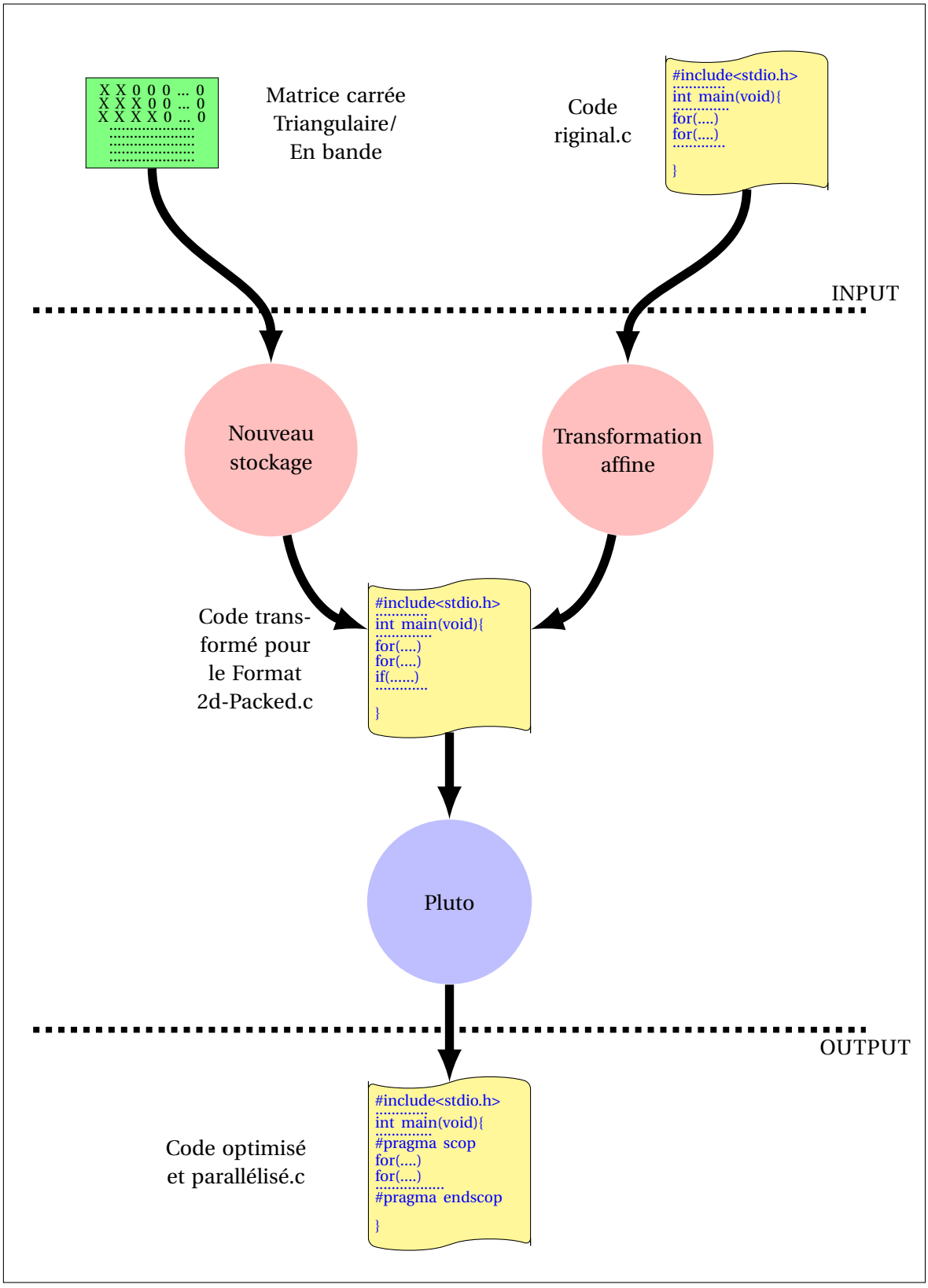
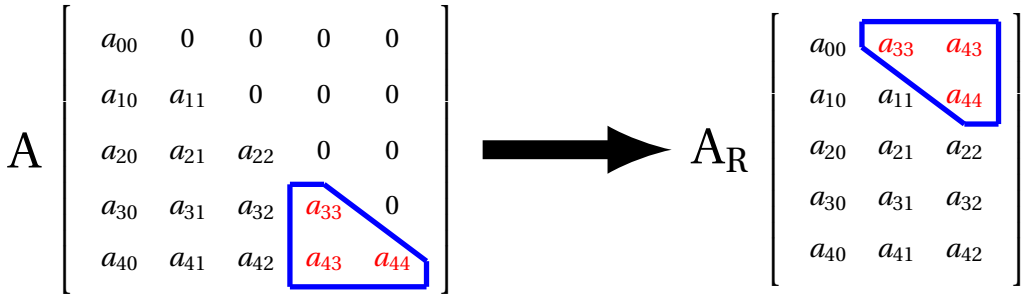
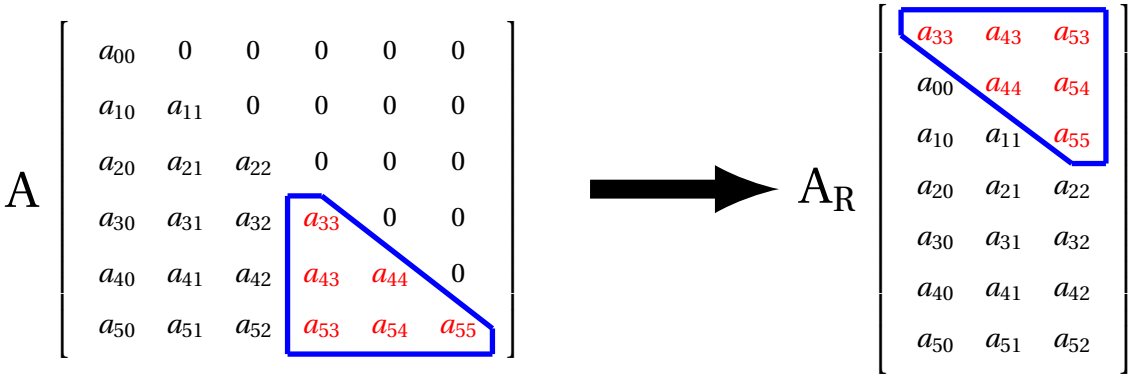


FIGURE 5.4 – L'approche 2d-packed.

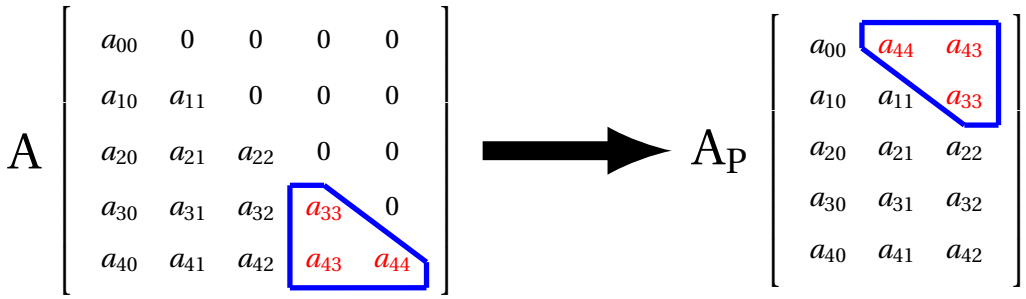


(a) Matrice triangulaire impair.

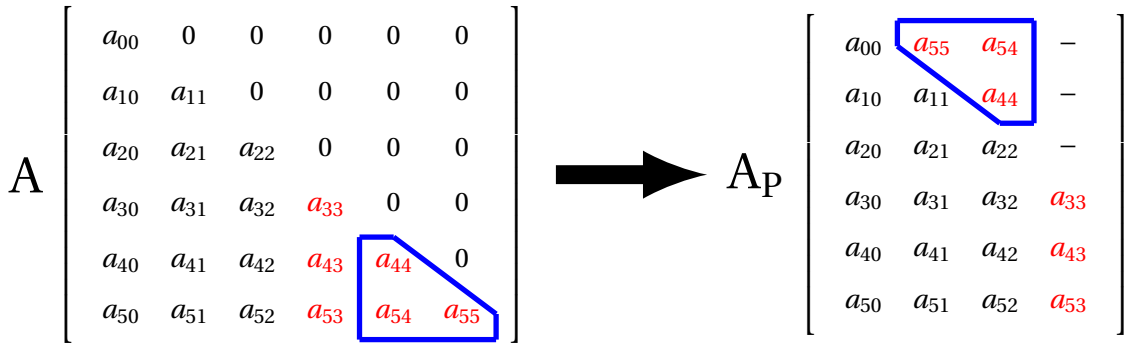


(b) Matrice triangulaire pair

FIGURE 5.5 – Le format de stockage Rectangular Full-Packed.



(a) Matrice triangulaire impair.



(b) Matrice triangulaire pair.

FIGURE 5.6 – Le format de stockage 2d-Packed.

```
1 //le code original produit matrice triangulaire vecteur
2 for(i=0; i<n; i++)
3          for(j=0; j<=i; j++){
4           S: C[i] = C[i] + A[i][j] * V[j];
5           }
```

FIGURE 5.7 – Le code original trimatvecmul.

Algorithm 1: Storage in 2d-packed format for triangular matrices.

Input: Triangular matrices Mat_i of size N

```
1 for RowInd ← 0 to N − 1 do
2     for ColInd ← 0 to RowInd − 1 do
3         if ColInd > N/2 then
4             StorMat_i(N − RowInd − 1, N − ColInd) ← Mat(RowInd, ColInd);
5         else
6             StorMat_i(RowInd, ColInd) ← Mat(RowInd, ColInd);
7         end
8     end
9 end
```

Algorithm 2: Triangular-matrix code transformation for the 2d-packed storage format.

Input: Triangular matrices Mat_i of order N

```
1 for any statement containing references Mat_i(x_j, y_j) to triangular matrices do
2     replace any reference Mat_i(x_j, y_j) with StorMat_i(z_j, w_j) where :
3     if y_j > N/2 then
4         (z_j, w_j) = (N − x_j − 1, N − y_j);
5     else
6         (z_j, w_j) = (x_j, y_j);
7     end
8 end
```

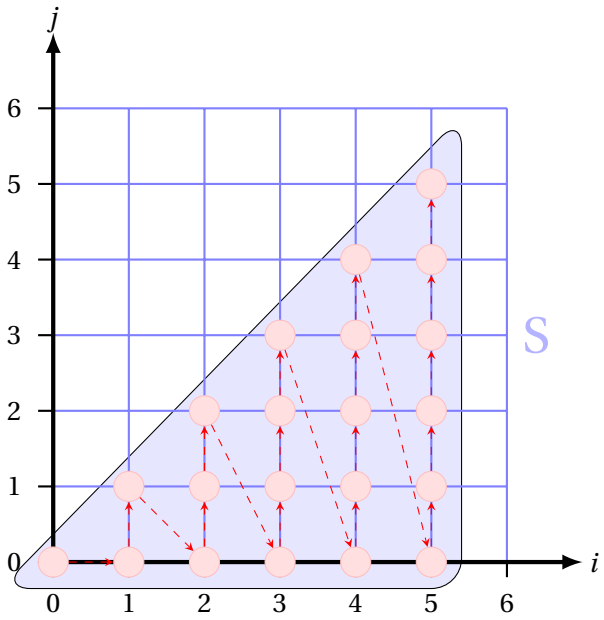


FIGURE 5.8 – Représentation polyédrique de code original trimatvecmul.

```
//le code 2d-Packed produit matrice triangulaire vecteur
for(i=0; i<n; i++)
          for(j=0; j<=i; j++) {
               if(2*j>n)
                S1: C[i] =C[i] + A[n-(i)-1][n-(j)] * V[j];

               if(2*j<=n)
                S2: C[i] = C[i] + A[i][j] * V[j];
                }
```

FIGURE 5.9 – Le code matvecmul transformé en 2d-Packed.

la figure 5.10 montre la représentation polyédrique du code transformé en 2d-Packed pour toujours n=6.

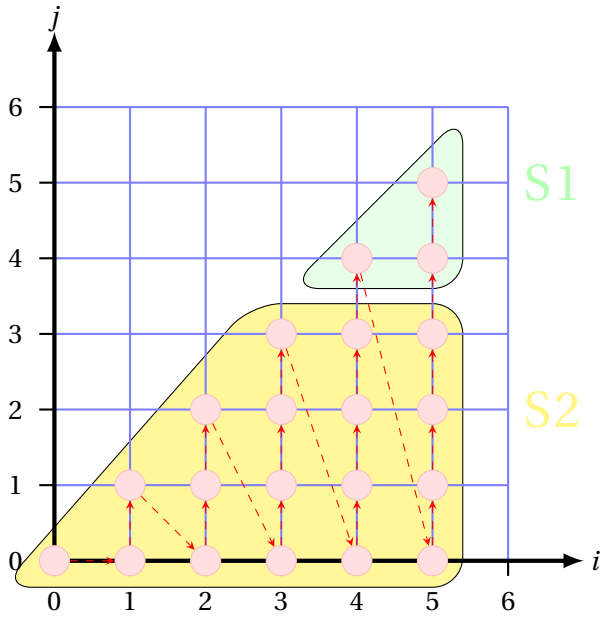


FIGURE 5.10 – Représentation polyédrique du code matvecmul transformé en 2d-Packed.

Le code généré par l'outil de parallélisation et d'optimisation automatique pluto en utilisant les options de compilation –tiled et –parallel est illustré dans la figure 5.11

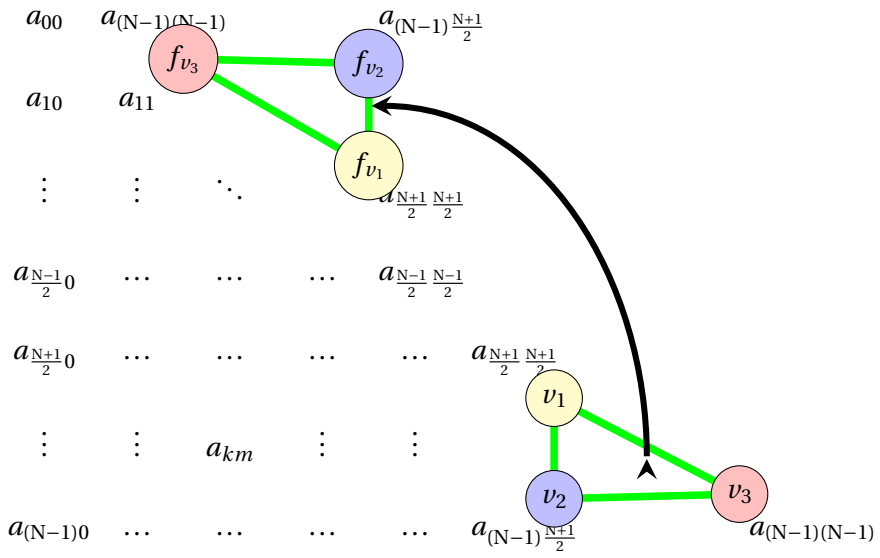
5.4.1.2 Démonstration

Afin de prouver que notre fonction de transformation pour les matrices triangulaires f_t est valide, nous devons démontrer qu'il s'agit d'une bijection c'est à dire tout élément appartenant au triangle inférieur droit de la figure 5.12a possède une et une seule image appartenant au triangle supérieur. Les autres éléments non nuls de la matrice restent non touchés. Puisque la transformation des premiers éléments est unimodulaire, il suffit de prouver que les images des sommets du triangle inférieur sont exactement les sommets du triangle supérieur (voir la figure 5.12a), où les coordonnées des sommets sont les indices des éléments matriciels correspondants. Lorsque N est impair, les sommets du triangle inférieur sont : $V1 = (\frac{N+1}{2}, \frac{N+1}{2})$,$V2 = (N-1, \frac{N+1}{2})$ et $V3 = (N-1, N-1)$. La transformation des sommets V1, V2 et V3 en utilisant la fonction unimodulaire 5.1 est donnée par $f_t(V1)$, $f_t(V2)$ et $f_t(V3)$, respectivement :

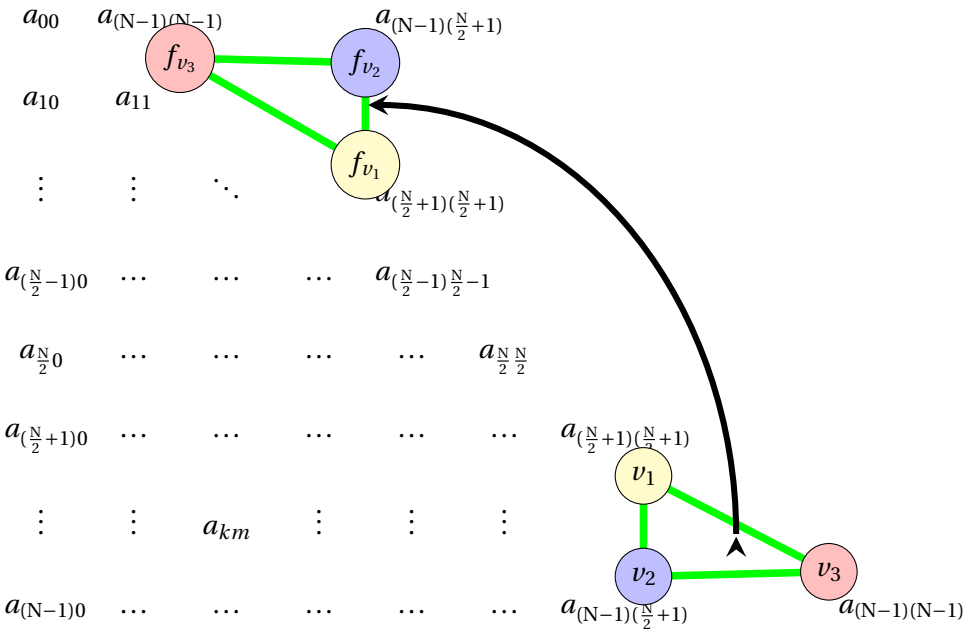
$$\begin{array}{lcll} f_t(V_1) & = & (-\frac{N+1}{2}+N-1, -\frac{N+1}{2}+N) & = & (\frac{N-3}{2}, \frac{N-1}{2}) \\ f_t(V_2) & = & (-(N-1)+N-1, -\frac{N+1}{2}+N) & = & (0, \frac{N-1}{2}) \\ f_t(V_3) & = & (-(N-1)+N-1, -(N-1)+N) & = & (0,1) \end{array}$$

```
////le code 2d-Packed produit matrice triangulaire vecteur apres l'
    utilisation de  pluto
/* Start of CLooG code */
if (n >= 1) {
  lbp=0;
  ubp=floord(n-1,32);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4)
  for (t1=lbp;t1<=ubp;t1++) {
    for (t2=0;t2<=t1;t2++) {
      if (t2 >= ceild(n+1,64)) {
        for (t3=32*t1;t3<=min(n-1,32*t1+31);t3++) {
          for (t4=32*t2;t4<=min(t3,32*t2+31);t4++) {
            C[t3] = C[t3] + A[n - (t3) - 1][n - (t4)] * V[t4];;
          }
        }
      }
      if ((t1 == t2) && (t1 >= ceild(n-61,64))) {
        for (t3=32*t1;t3<=floord(n,2);t3++) {
          for (t4=32*t1;t4<=t3;t4++) {
            C[t3] = C[t3] + A[t3][t4] * V[t4];;
          }
        }
      }
      if (t2 <= floord(n-62,64)) {
        for (t3=32*t1;t3<=min(n-1,32*t1+31);t3++) {
          for (t4=32*t2;t4<=min(t3,32*t2+31);t4++) {
            C[t3] = C[t3] + A[t3][t4] * V[t4];;
          }
        }
      }
      if ((t2 <= floord(n,64)) && (t2 >= ceild(n-61,64))) {
        for (t3=max(ceild(n+1,2),32*t1);t3<=min(n-1,32*t1+31);t3++) {
          for (t4=32*t2;t4<=floord(n,2);t4++) {
            C[t3] = C[t3] + A[t3][t4] * V[t4];;
          }
          for (t4=ceild(n+1,2);t4<=min(t3,32*t2+31);t4++) {
            C[t3] = C[t3] + A[n - (t3) - 1][n - (t4)] * V[t4];;
          }
        }
      }
    }
  }
}
/* End of CLooG code */
```

FIGURE 5.11 – Le code trimatvecmul au format 2d-Packed généré après la parallélisation avec PLuTo.



(a) Matrice triangulaire impair.



(b) Matrice triangulaire pair.

FIGURE 5.12 – Matrice de transformation pour les matrices triangulaires.

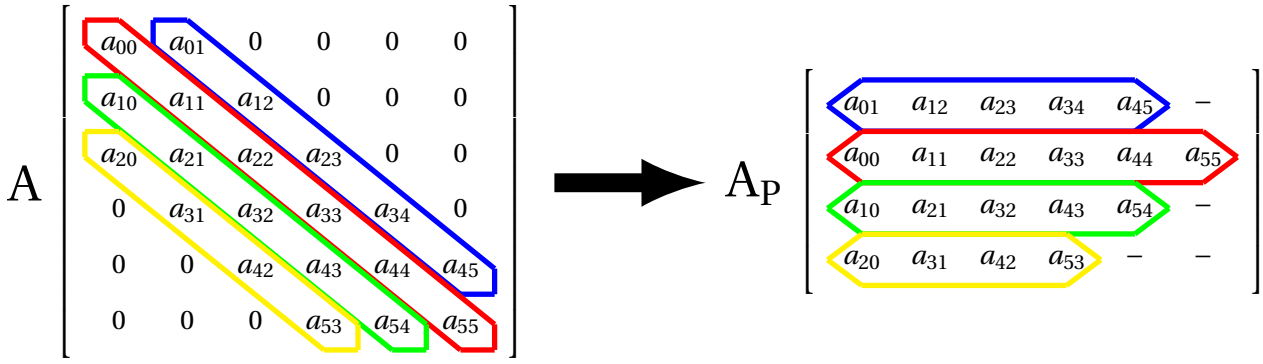


FIGURE 5.13 – Stockage des matrices en bande en format 2d-Packed.

On peut remarquer que ce sont exactement les sommets du triangle supérieur (figure 5.12a), qui est disjoint des autres éléments non nuls, de sorte que le cas impair est prouvé. Lorsque N est pair, les sommets du triangle inférieur sont : V1 = $(\frac{N}{2}+1, \frac{N}{2}+1)$,V2 = $(N-1, \frac{N}{2}+1)$ et V3 = $(N-1, N-1)$. La transformation de ces sommets en utilisant la même fonction 5.1 est donnée par $f_t(V1)$, $f_t(V2)$ et $f_t(V3)$, respectivement :

$$\begin{array}{lcll} f_t(V_1) & = & (-(\frac{N}{2}+1)+N-1, -(\frac{N}{2}+1)+N) & = & (\frac{N}{2}-2, \frac{N}{2}-1) \\ f_t(V_2) & = & (-(N-1)+N-1, -(\frac{N}{2}+1)+N) & = & (0, \frac{N}{2}-1) \\ f_t(V_3) & = & (-(N-1)+N-1, -(N-1)+N) & = & (0,1) \end{array}$$

Encore une fois, nous pouvons voir que ce sont les sommets du triangle supérieur (figure 5.12b), qui est disjoint des autres éléments non nuls non touchés.

5.4.2 L'approche 2d-Packed pour les matrices en bande

Nous proposons une approche d'optimisation similaire à appliquer aux matrices en bande. Dans ce cas, nous stockons les $N^2 - (\frac{(N-W_u)^2+(N-W_l)^2-2N-W_u-W_l}{2})$ éléments non nuls d'une matrice en bande carrée d'ordre N dans une structure de données 2d-packed de $W_l + W_u + 1$ lignes, ces lignes sont de tailles différentes (voir la figure 5.13). Cette structure est légèrement différente de celle implémentée dans la bibliothèque LAPACK.

En effet, tous les éléments de la bande supérieure sont décalés vers la gauche afin de ne stocker aucun élément nul (en utilisant l'allocation dynamique). La figure 5.14 montre un code C allouant dynamiquement de la mémoire pour cette nouvelle structure de données 2d-packed. Les algorithmes 3 et 4 illustrent, respectivement, la manière dont les éléments non nuls des matrices en bande sont stockées en format 2d-packed, et la transformation de code pour toutes les instructions dans lesquelles les matrices en bande sont référencées.

```
1 double **allocBandMat(int n, int Wu, int Wl){
2  double **mat; int i;
3   mat= malloc((Wu+Wl+1)*sizeof(double *));
4   for (i=0; i<=Wu; i++)
5      mat[Wu-i]=calloc((n-i),sizeof(double));
6   for(i=1; i<= Wl; i++)
7      mat[Wu+i]=calloc((n-i),sizeof(double));
8   return mat;
9 }
```

FIGURE 5.14 – Allocation dynamique des matrices en bande en format 2d-Packed.

Encore une fois, nous considérons, sans perte de généralité, que nous recevons une déclaration contenant une seule référence (x, y) à une matrice en bande. Cette référence est transformée comme montré par l'équation 5.3.

$$f_b(x,y) = \begin{cases} (\mathrm{W}_u + x - y, y) & \text{if } x > y \\ (\mathrm{W}_u + x - y, x) & \text{else} \end{cases} \tag{5.3}$$

La représentation matricielle de cette transformation unimodulaire par morceaux est donnée par l'équation 5.4.

$$f_b(x,y) = \begin{cases} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \mathrm{W}_u \\ 0 \end{pmatrix} & \text{if } x > y \\ \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \mathrm{W}_u \\ 0 \end{pmatrix} & \text{else} \end{cases} \tag{5.4}$$

Algorithm 3: Storage in 2d-packed format for banded matrices.

Input: Banded matrices Mat_i of order N and Upper bandwidth W_u and Lower bandwidth W_l

```
for RowInd ← 0 to N − 1 do
    for ColInd ← 0 to N − 1 do
        if ColInd <= (RowInd + W_u) and ColInd >= (RowInd − W_l) then
            if RowInd > ColInd then
                StorMat_i(W_u + RowInd − ColInd, ColInd) ← Mat(RowInd, ColInd);
            else
                StorMat_i(W_u + RowInd − ColInd, RowInd) ← Mat(RowInd, ColInd);
            end
        end
    end
end
```

Algorithm 4: Banded-matrix code transformation for the 2d-packed storage format.

Input: Banded matrices Mat_i of order N and Upper bandwidth W_u and Lower bandwidth W_l

```
for any statement containing references Mat_i(x_j, y_j) to banded matrices do
    replace any reference Mat_i(x_j, y_j) with StorMat_i(z_j, w_j) Where :
    if x_j > y_j then
        (z_j, w_j) = (W_u + x_j − y_j, y_j);
    else
        (z_j, w_j) = (W_u + x_j − y_j, x_j)
    end
end
```

5.4.2.1 Représentation polyédrique

De la mème manière présenté dans le cas des matrices triangulaires , on va montrer l'exemple du produit d'une matrice en bande par un vecteur, le code présenté dans la figure 5.15 calcule ce produit.

Après l'application de notre transformation 2d-Packed pour les matrices en bande on a trouvé le code présenté dans la figure 5.17. La représentation polyédrique de ce code pour n=6, k1=1, k2=2 est illustré dans la figure 5.16.

la figure 5.18 montre la représentation polyédrique du code transformé en 2d-Packed pour toujours n=6.

Le code généré par l'outil de parallélisation et d'optimisation automatique pluto en utilisant les options de compilation –tiled et –parallel est illustré dans la figure 5.19.

```
//le code original produit matrice en bande vecteur
for(i=0; i<n; i++)
          for(j=0; j<n; j++){
//k1 la largeur de bande superieur et k2 la largeur inferieur
          if ((j >= (i - k2)) && (j <= (i + k1)))
           S: C[i] = C[i] + A[i][j] * V[j];
           }
```

FIGURE 5.15 – Le code original bandmatvecmul.

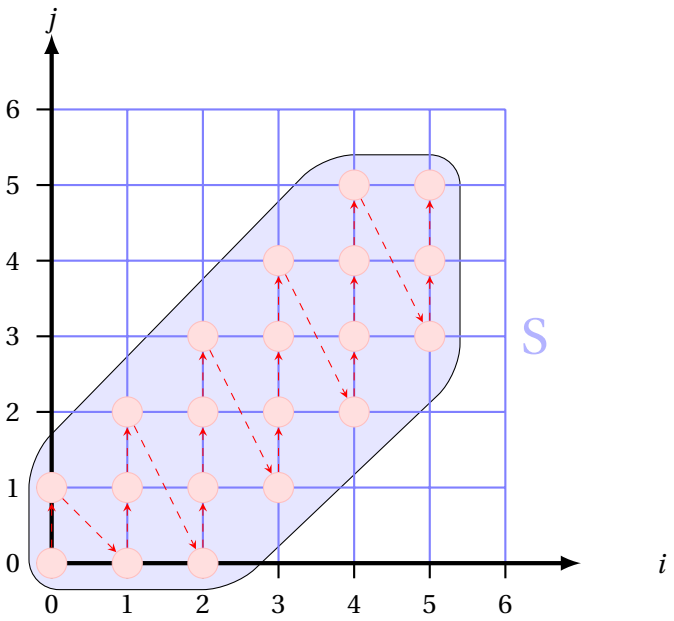


FIGURE 5.16 – Représentation polyédrique de code original bandmatvecmul.

```
//le code 2d-Packed produit matrice en bande vecteur
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
      if ((j >= (i - k2)) && (j <= (i + k1)))
      {
        if (i < j)
          S1: C[i] = C[i] + A[k1 + i - (j)][i] * V[j];
        else
          S2: C[i] = C[i] + A[k1 + i - (j)][j] * V[j];
      }
    }
```

FIGURE 5.17 – Le code bandmatvecmul transformé en 2d-Packed.

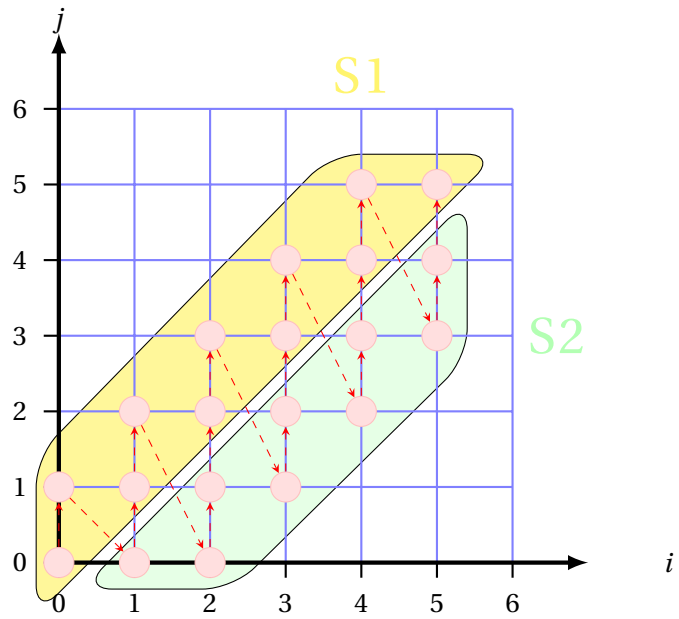


FIGURE 5.18 – Représentation polyédrique du code bandmatvecmul transformé en 2d-Packed.

5.4.2.2 Démonstration

Comme dans le cas des matrices triangulaires, nous pouvons prouver la validité de notre transformation pour les matrices en bandes f_b en démontrant que tout élément appartenant au trapèze supérieur de la Matrice A de la figure 5.20 possède une et une seule image appartenant au trapèze supérieur (respectivement inférieur) de la matrice A_P. En raison de l'unimodularité de nos deux transformations, il suffit de prouver que les images des deux trapèzes sont disjointes, ou équivalentes, que les sommets des trapèzes supérieurs et inférieurs de la matrice A correspondent exactement aux sommets des trapèzes disjoints supérieurs et inférieurs de la matrice A_p, qui se fait de la même manière que dans la preuve précédente. Les coordonnées des sommets sont données dans la figure 5.20.

```
//le code 2d-Packed produit matrice en bande vecteur apres l'utilisation de
    pluto
/* Start of CLooG code */
if ((k2 >= -k1) && (n >= 1) && (n >= -k1+1) && (n >= -k2+1)) {
  lbp=max(0,ceild(-k1-31,32));
  ubp=min(floord(n-1,32),floord(n+k2-1,32));
#pragma omp parallel for private(lbv,ubv,t2,t3,t4)
  for (t1=lbp;t1<=ubp;t1++) {
    for (t2=max(0,ceild(32*t1-k2-31,32));t2<=min(min(floord(n-1,32),floord(n+
    k1-1,32)),floord(32*t1+k1+31,32));t2++) {
      if ((k1 >= 1) && (k2 >= 0) && (t1 == t2)) {
        for (t3=32*t1;t3<=min(n-2,32*t1+30);t3++) {
          for (t4=max(32*t1,t3-k2);t4<=t3;t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t4] * V[t4];;
          }
          for (t4=t3+1;t4<=min(min(n-1,32*t1+31),t3+k1);t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t3] * V[t4];;
          }
        }
      }
      if (k2 >= 0) {
        for (t3=max(32*t1,32*t2-k1);t3<=min(32*t1+31,32*t2-1);t3++) {
          for (t4=32*t2;t4<=min(min(n-1,32*t2+31),t3+k1);t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t3] * V[t4];;
          }
        }
      }
      if (k2 <= -1) {
        for (t3=max(32*t1,32*t2-k1);t3<=min(min(32*t1+31,n+k2-1),32*t2+k2+31)
    ;t3++) {
          for (t4=max(32*t2,t3-k2);t4<=min(min(n-1,32*t2+31),t3+k1);t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t3] * V[t4];;
          }
        }
      }
      if ((k1 >= 1) && (k2 >= 0) && (t1 == t2) && (t1 >= ceild(n-31,32))) {
        for (t4=max(32*t1,n-k2-1);t4<=n-1;t4++) {
          C[(n-1)] = C[(n-1)] + A[k1 + (n-1) - (t4)][t4] * V[t4];;
        }
      }
      if (k1 >= 1) {
        for (t3=max(32*t1,32*t2+31);t3<=min(min(n-1,32*t1+31),32*t2+k2+31);t3
    ++) {
          for (t4=max(32*t2,t3-k2);t4<=32*t2+31;t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t4] * V[t4];;
          }
        }
      }
      if (k1 <= 0) {
        for (t3=max(32*t1,32*t2-k1);t3<=min(min(n-1,32*t1+31),32*t2+k2+31);t3
    ++) {
          for (t4=max(32*t2,t3-k2);t4<=min(32*t2+31,t3+k1);t4++) {
            C[t3] = C[t3] + A[k1 + t3 - (t4)][t4] * V[t4];;
          }
        }
      }
    }
  }
}
/* End of CLooG code */
```

FIGURE 5.19 – Le code bandmatvecmul en 2d-Packed généré après la parallélisation avec Pluto.

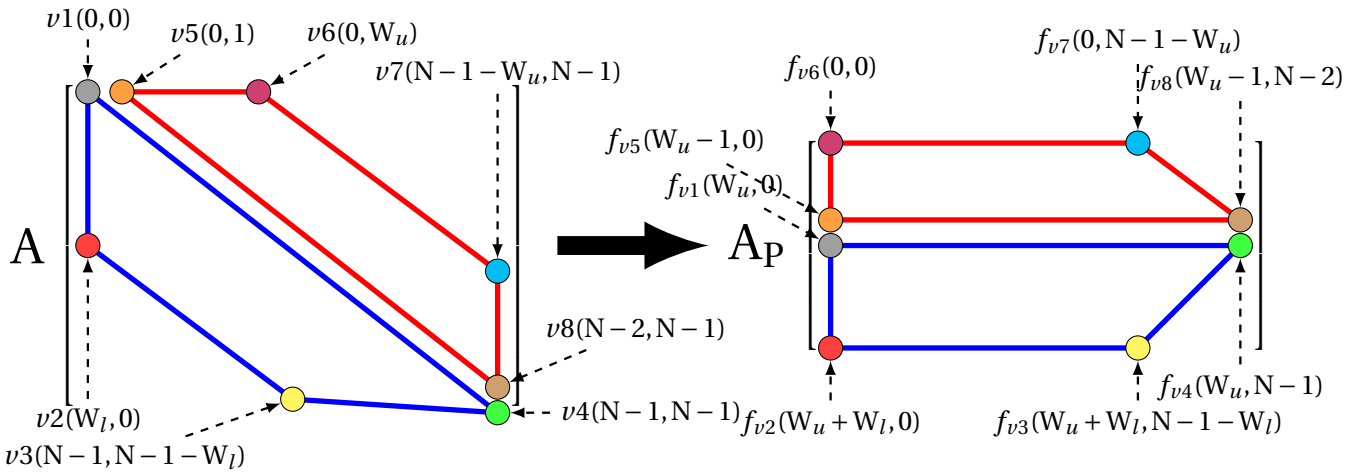


FIGURE 5.20 – Fonction de transformation des matrices en bande en format 2d-Packed.

5.5 Résultats Expérimentaux

Pour évaluer notre approche, nous avons combiné la nouvelle transformation de format 2d-Packed avec le compilateur source-à-source Pluto. Pluto est utilisé pour paralléliser et vectoriser automatiquement les programmes, en plus de l'optimisation des accès mémoire grâce à l'amélioration de la localité des données. Nous avons appliqué notre approche pour optimiser et paralléliser six codes de calcul d'algèbre linéaire à double précision avec les différents types de matrices présentés dans le Tableau 5.1. Notez que certains calculs matriciels, tels que Cholesky et sspfa, ne peuvent pas être appliqués à des matrices non triangulaires et n'apparaissent pas dans ce tableau. Ces codes sont les mêmes que ceux évalués par Cui et al. [43] , plus la factorisation de Cholesky qui a été évaluée par Gustavson et al. [67]. Pour montrer l'efficacité de notre approche, nous avons comparé ses performances à un ensemble de méthodes et de bibliothèques existantes traitant des calculs d'algèbre linéaire. Les comparaisons ont été effectuées entre des versions de code différentes, mais sémantiquement équivalentes, comme indiqué dans le Tableau 5.2. Tous les programmes ont d'abord été testés pour vérifier l'exactitude des matrices de sortie avant d'être exécutés pour des mesures de temps d'exécution.

Pour toutes nos expériences, nous avons utilisé un double socket 2x10 (et 40 threads) Intel® Xeon® CPU E5-2650 v3 @ 2.30 GHz sous Linux 4.4.0 (tous les détails de ce processeur ont été expliqué dans le chapitre 1). Nous avons compilé nos programmes en utilisant icc 17.0.0 et gcc 5.4.0, avec les options -O3 -march = natif. Les programmes appelant les routines MKL sont compilés en utilisant icc avec les flags supplémentaires -mkl -parallel. En outre, la version 0.11.4 de Pluto, avec les options –parallel –tile a été appelée pour paralléliser automatiquement, appliquer le tiling et optimiser les codes originals et 2d-packed. Les codes LPF (Linear Packed Format) sont séquentiels, car ils ne peuvent pas être parallélisés automatiquement par Pluto, car dans le stockage LPF, les fonctions d'accès sont non-linéaires. Chaque programme a été exécuté cinq fois et les temps d'exécution des codes ont été mesurés en utilisant gettimeofday (). Les deux mesures extrémales ont été éliminées et la moyenne des trois restants est rapportée.

Les figures 5.21, 5.22, 5.23, 5.24 et 5.25 illustrent l'exécution des différents points de référence pour les plus petites (N=500 ou 1000), petites (N = 1000 ou 2000), moyennes (N = 2000 ou 4000), grandes (N = 4000 ou 8000) et les plus grandes (N = 8000 ou 16000) matrices, respectivement. Dans les cinq figures, les sous-figures (a) et (b) montrent les temps d'exécution obtenus en utilisant les compilateurs icc et gcc, respectivement. Les figures d'exécution icc représentent six barres correspondant aux six versions de code indiquées dans le tableau 5.2, à l'exception de sspfa qui n'a pas de routine équivalente dans la bibliothèque MKL. Les chiffres d'exécution de gcc ne représentent

TABLEAU 5.1 – Noyaux de calcul matriciel

Computation	**Matrix Type**	**Designation**
Cholesky Factorization	Banded Triangular	`choleskyBT`
	Triangular	`choleskyTri`
Matrix Matrix Multiplication	Banded Triangular	`matmulBT`
	Banded	`matmulBand`
	Triangular	`matmulTri`
Matrix Vector Multiplication	Banded Triangular	`mvBT`
	Banded	`mvBand`
	Triangular	`mvTri`
Matrix Matrix Solver	Banded Triangular	`solvematBT`
	Triangular	`solvematTri`
Matrix Vector Solver	Banded Triangular	`solvevectBT`
	Triangular	`solvevectTri`
sspfa Factorisation	Banded Triangular	`sspfaBT`
	Triangular	`sspfaTri`

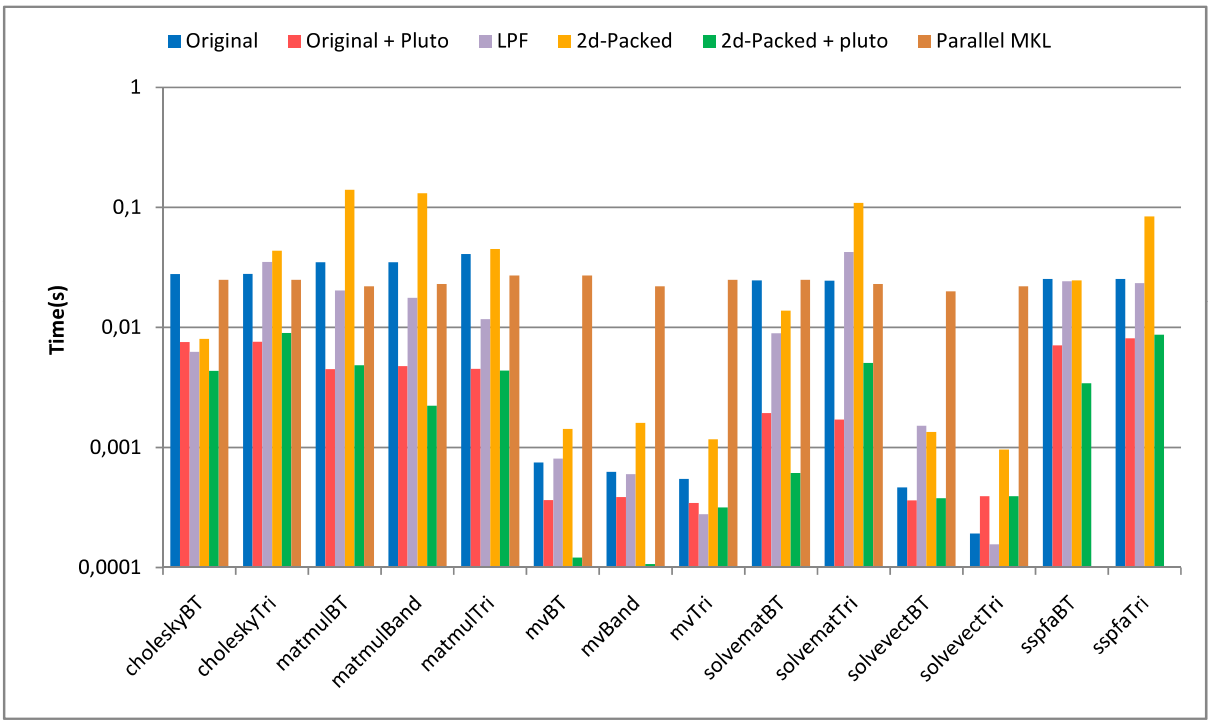
que cinq barres : il n'y a pas de bibliothèque MKL fournie avec gcc. Tous les axes Y sont en échelle logarithmique en raison de la variation importante de leurs valeurs. La première valeur de N est utilisée pour tous les algorithmes O(N3), la seconde pour les deux algorithmes.

Dans toutes les figures, on peut noter que les temps les plus élevés sont ceux correspondant aux codes original et 2d-packed. C'est parce qu'ils n'étaient pas parallélisés, et en raison du stockage d'une grande quantité de zéros inutiles dans le premier cas, et le second cas contient des tests et des opérations arithmétiques dans les indices . Lorsque ces deux codes sont optimisés et parallélisés par Pluto, leurs performances sont considérablement améliorées. Ils sont également meilleurs comparés au code LPF, qui ne peut pas être parallélisé automatiquement. Notez que, en plus de leur faible stockage, les performances de nos codes optimisés (2d-packed + pluto) sont meilleures ou équivalentes à celles des codes optimisés originaux (original + pluto) pour la quasi-totalité d'entre eux. La raison de ce delta est que, d'abord, les codes 2d-packed accèdent globalement à moins de mémoire, et par conséquent il y a moins de pression sur la hiérarchie de la mémoire et sur les caches en particulier ; deuxièmement, certains des codes d'origine effectuent des calculs sur des zéros inutiles, qui ont été supprimés dans les codes 2d-packed. En générale, sur toutes ces expériences, les codes parallélisés 2d-packed fonctionnent mieux que les codes parallélisés originaux avec un rapport de :

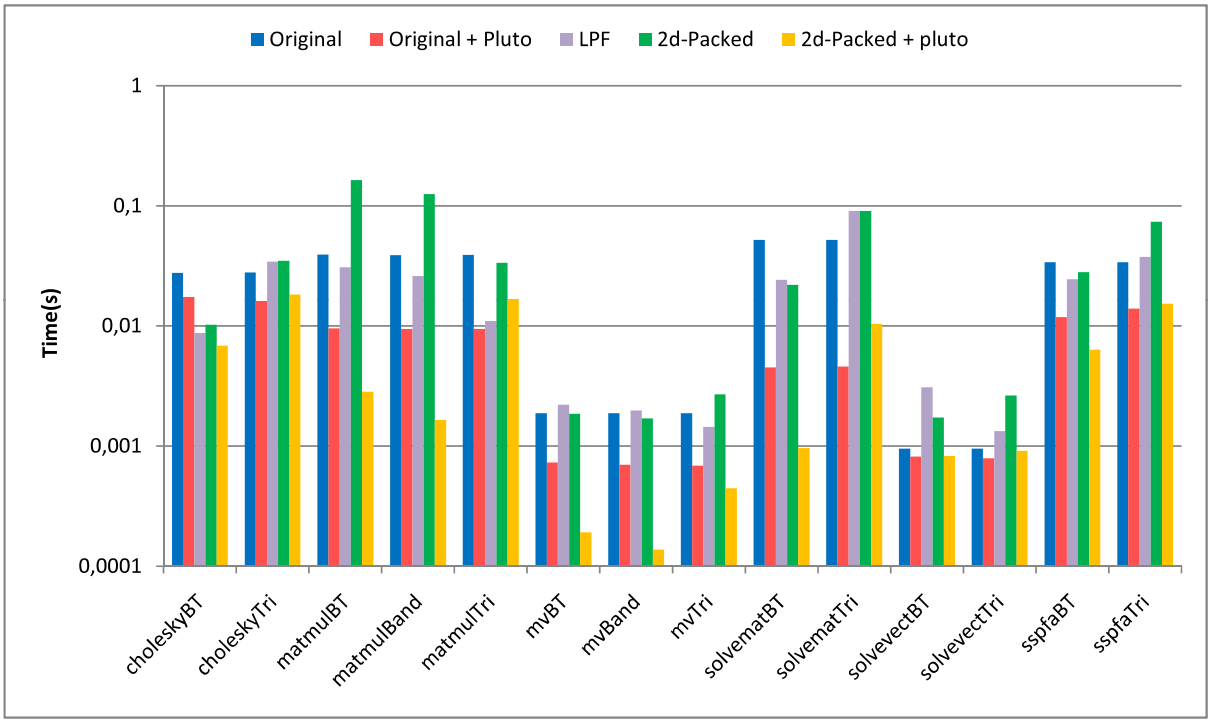
— 1.64x pour gcc,

— 1.78x pour icc.

La comparaison des codes optimisés 2d-packed avec les routines MKL parallélisées non-Packed avec le compilateur icc dans les cinq figures montre que notre méthode dépasse les routines de MKL dans dix cas sur douze pour les matrices : plus petites, petites et moyennes, et pour les matrices grandes et plus grandes, les performances sont similaire, avec une variance élevée. Les routines MKL consomment plus de mémoire que nos codes 2d-packed parallélisés, car ils traitent les matrices dans leur format carré complet. D'autre part, comme ils effectuent des calculs sur des matrices complètes ou triangulaires, nous devons indiquer la quantité de données supplémentaires qui ne sont pas traitées par la version 2d-packed, mais qui sont traitées par les routines MKL. Ceci est résumé dans le tableau 5.3, avec le nom de la routine qui a été utilisé pour chaque référence. Notez que nous rapportons 100% des données traitées par la version triangulaire 2d-packed dans certains cas, lorsque l'entrée de la routine MKL est une matrice triangulaire ou symétrique ; 50% dans les autres cas, lorsque l'entrée est une matrice carrée complète. Le résultat de cette expérience est que les codes 2d-packed sont souvent en concurrence avec le code MKL non optimisé

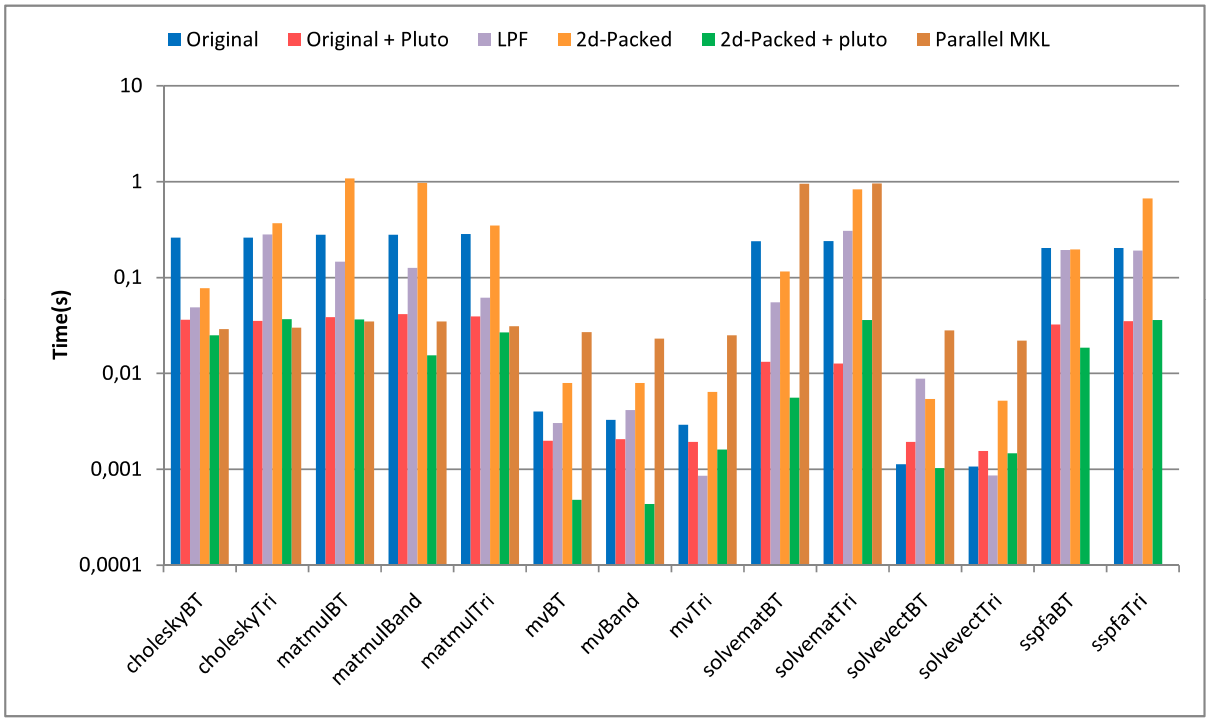


(a) Les codes compilés avec ICC.

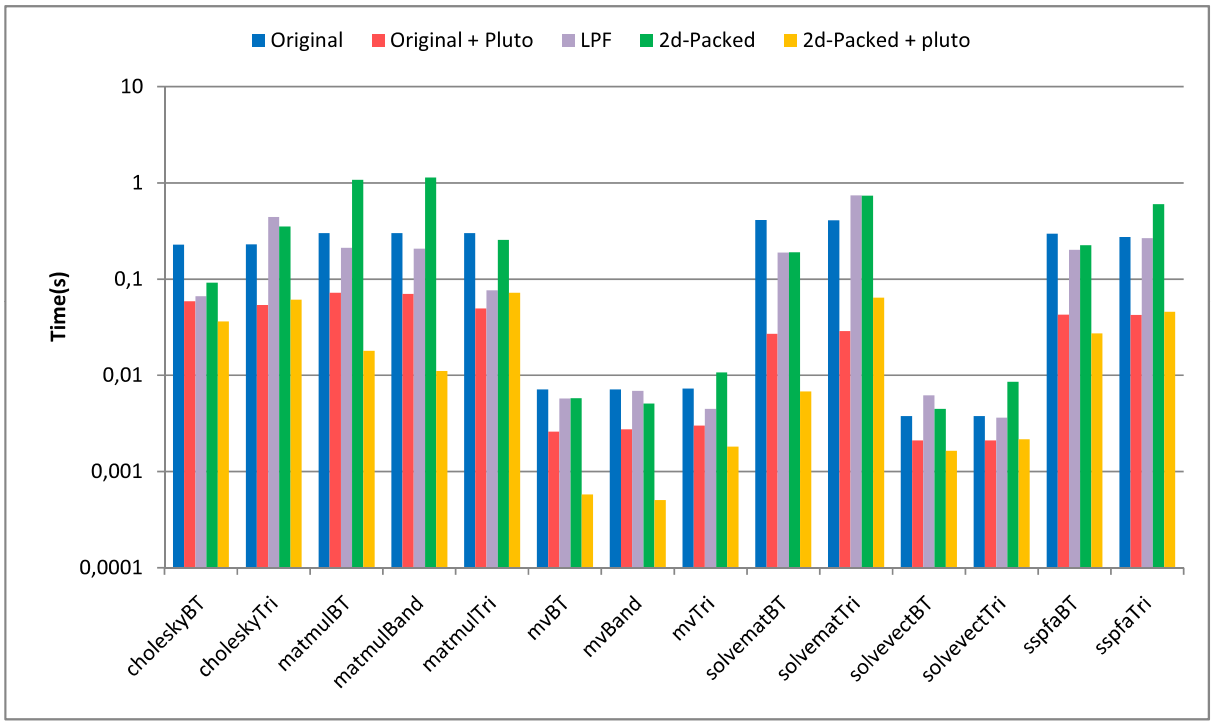


(b) Les codes compilés avec GCC.

FIGURE 5.21 – Temps d'exécution avec la dimension (N=500; 1000 pour les benchmarks O(N^2)).

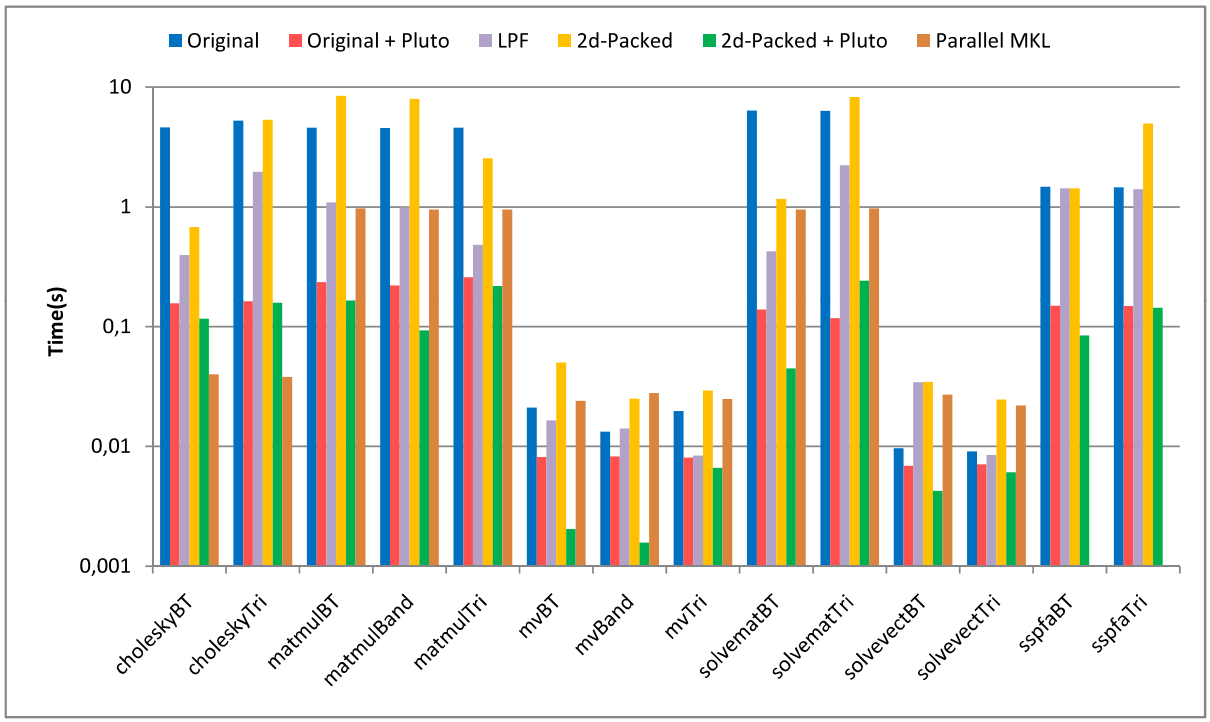


(a) Les codes compilés avec ICC.

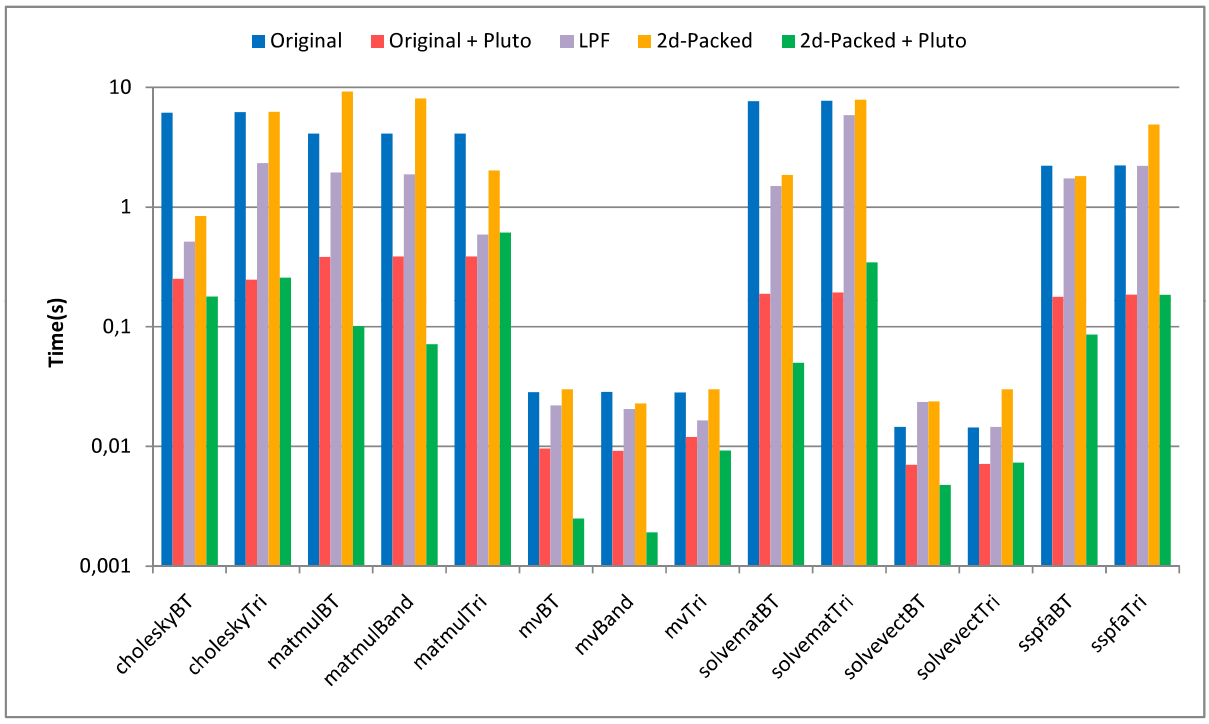


(b) Les codes compilés avec GCC.

FIGURE 5.22 – Temps d'exécution avec la dimension (N=1000 ; 2000 pour les benchmarks $O(N^2)$).

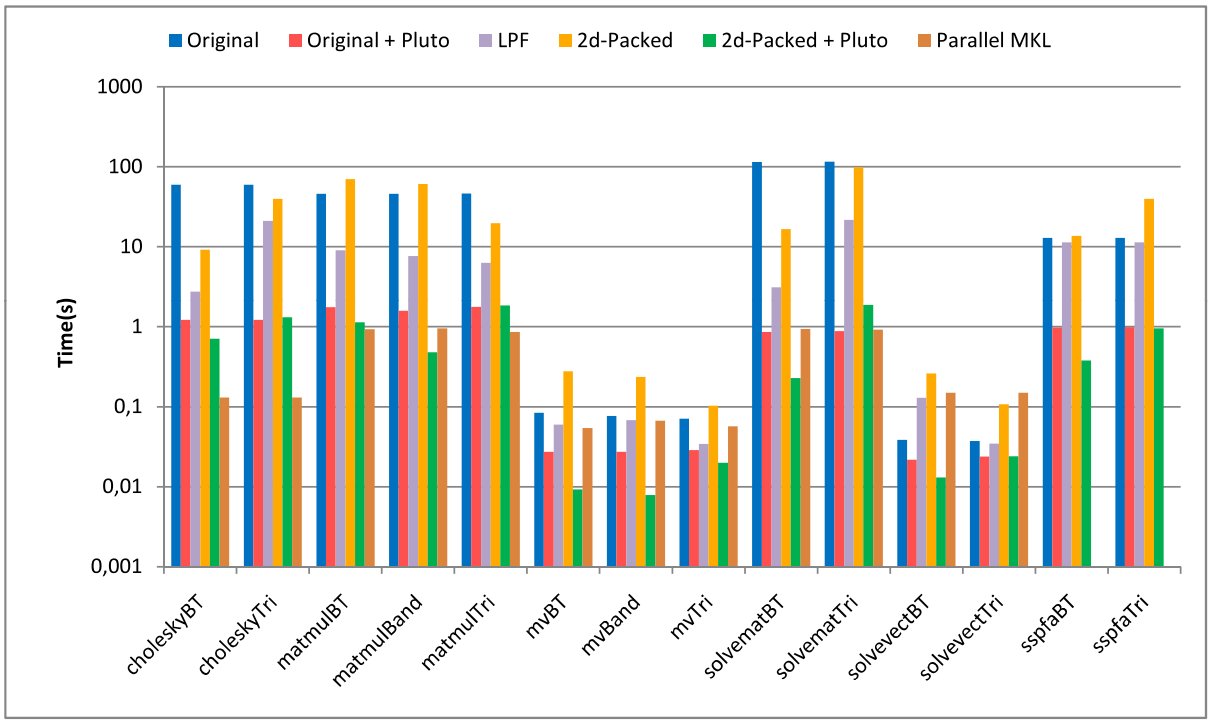


(a) Les codes compilés avec ICC.

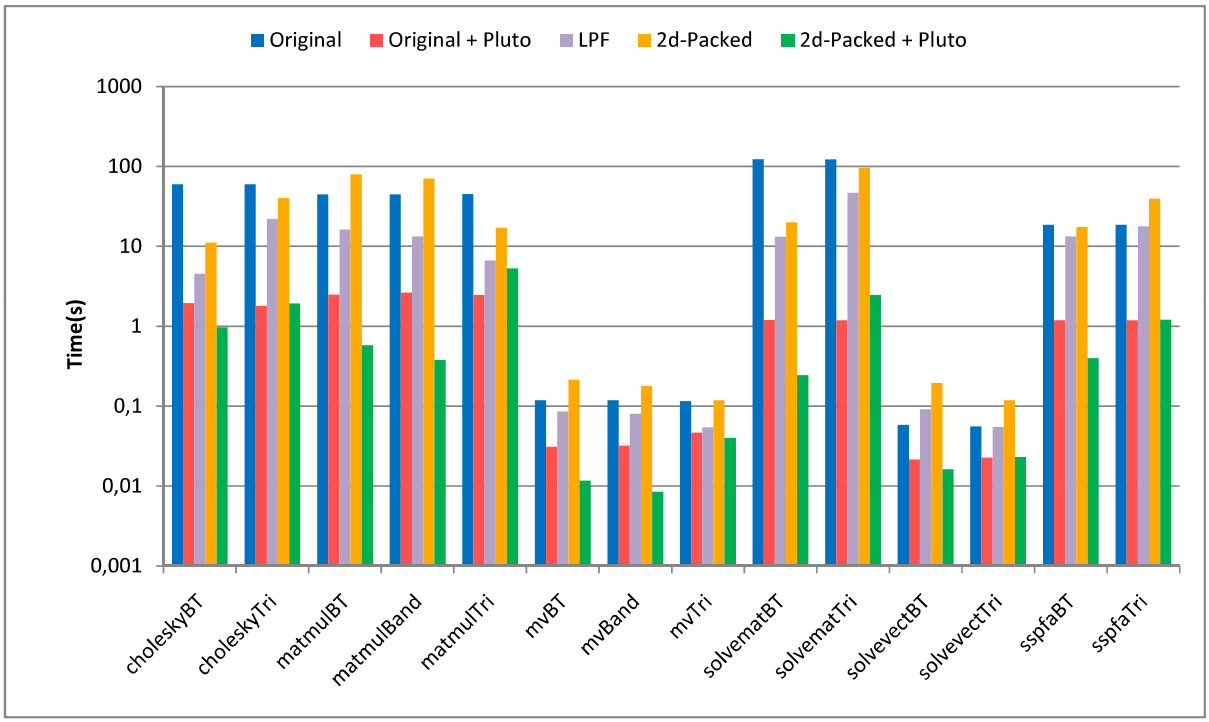


(b) Les codes compilés avec GCC.

FIGURE 5.23 – Temps d'exécution avec la dimension (N=2000 ; 4000 pour les benchmarks O(N^2)).

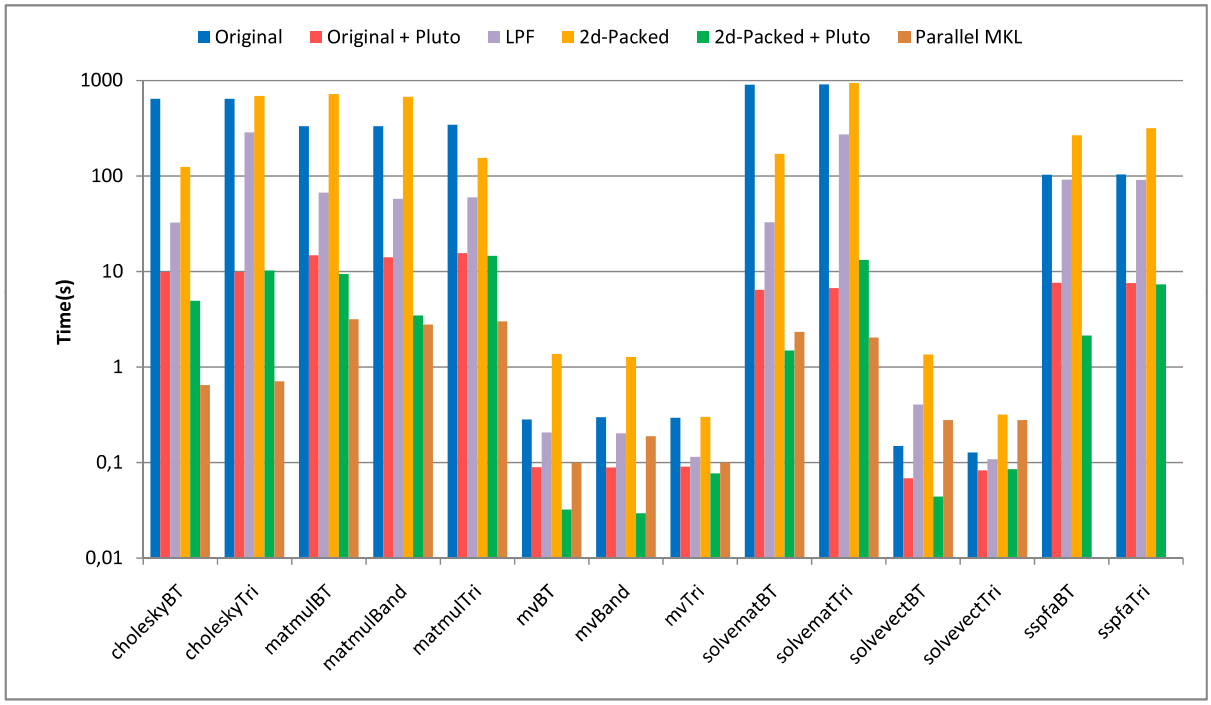


(a) Les codes compilés avec ICC.

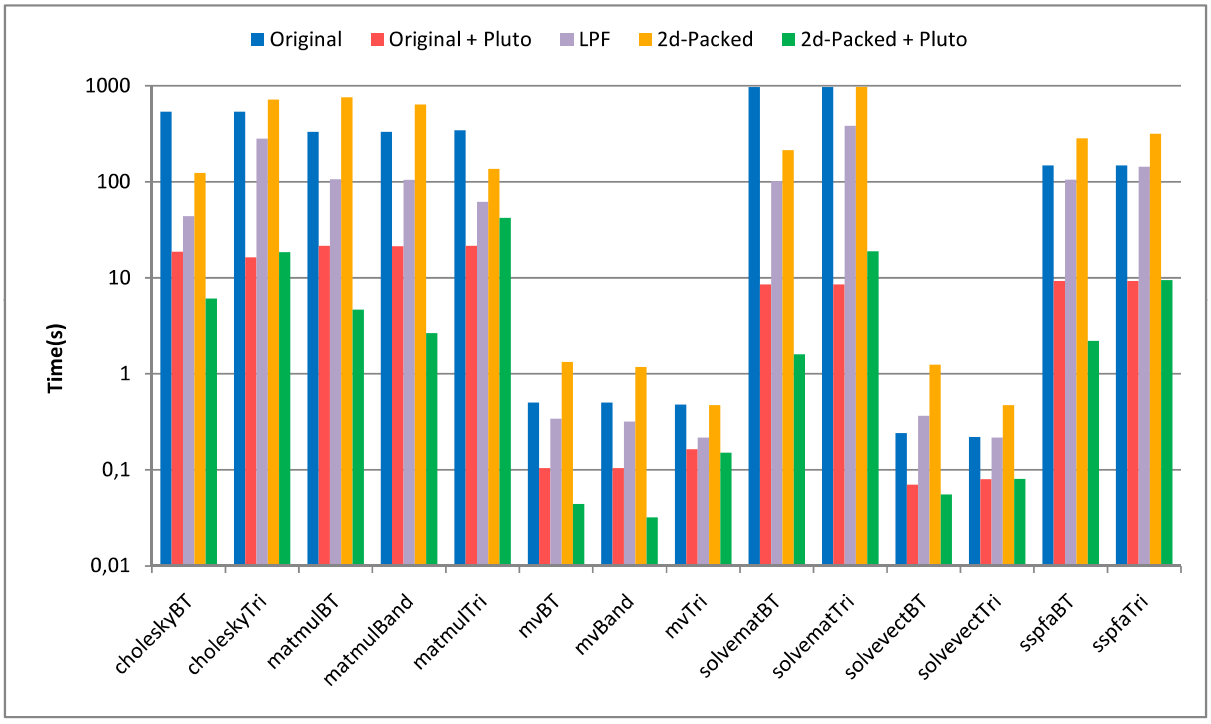


(b) Les codes compilés avec GCC.

FIGURE 5.24 – Temps d'exécution avec la dimension (N=4000 ; 8000 pour les benchmarks $O(N^2)$).



(a) Les codes compilé avec ICC.



(b) Les codes compilé avec GCC.

FIGURE 5.25 – Temps d'exécution avec la dimension (N=8000; 16000 pour les benchmarks O(N^2)).

TABLEAU 5.2 – Version des codes

Version de code	Description
Original	Code séquentiel original gérant des matrices carrées complètes
Original + Pluto	Le code original automatiquement optimisé et parallélisé par Pluto
LPF	Le code séquentiel gérant des matrices creusess dans le format Linear Packed
2d-packed	Le code séquentiel gérant des matrices creusess dans le format 2d-packed
2d-packed + Pluto	Le code 2d-packed automatiquement optimisé et parallélisé par Pluto
Parallel MKL	La routine MKL optimisé et parallélisé avec le compilateur icc gérant des matrices carrées complètes

TABLEAU 5.3 – Les appels des routines MKL

Benchmark	Nom de routine MKL	Percentage de données traités par la version 2d-packed
`choleskyBT`	`LAPACKE_dpotrf`	44%
`choleskyTri`	`LAPACKE_dpotrf`	100%
`matmulBT`	`cblas_dgemm`	22%
`matmulBand`	`cblas_dgemm`	19%
`matmulTri`	`cblas_dgemm`	50%
`mvBT`	`cblas_dgemv`	22%
`mvBand`	`cblas_dgemv`	19%
`mvTri`	`cblas_dgemv`	50%
`solvematBT`	`cblas_dtrsm`	44%
`solvematTri`	`cblas_dtrsm`	100%
`solvevectBT`	`cblas_dtrsv`	44%
`solvevectTri`	`cblas_dtrsv`	100%

à la main.

Nous avons également exécuté les trois noyaux proposés par Gustavson et al [67]. pour les matrices triangulaires (fatorization de Cholesky : LAPACKE_dpftrf, inverse : LAPACKE_dtftri, et résolution avec Cholesky : LAPACKE_dpftrs), et observé qu'ils fonctionnent à peu près d'une manière similaire que ceux utilisant le format matriciel complet dans MKL (LAPACKE_dpotrf, LAPACKE_-dpotri, LAPACKE_dpotrs). Puisque nous nous comparons directement à cette dernière pour la factorisation de Cholesky, nous n'avons pas présenté de comparaison supplémentaire avec les trois noyaux de Gustavson et al. Non rapporté dans ces figures, nous avons également essayé de paralléliser les codes LPF en utilisant l'auto-paralléliseur icc (avec les options -parallel -O3 -march = natif), et il s'avère que icc n'est pas capable de paralléliser ces codes, très probablement en raison des accès non linéaires aux tableaux LPF et de l'analyse de dépendances complexe qui en résulte. Les temps d'exécution utilisant l'option -parallel sont les mêmes que les temps d'exécution séquentiels rapportés, à l'exception de deux codes de produit matrice-vecteur très courts (mvBT et mvBand). Selon l'ensemble des expériences ci-dessus, nous concluons que notre transformation de format 2d-packed combinée avec Pluto est capable d'améliorer de manière significative les performances de ces calculs matriciels creuses. Ce n'est pas le cas pour le format compact linéaire, dont l'optimisation automatique et la parallélisation sont assez difficiles.

5.6 Application de l'approche 2d-Packed : la méthode de dissection emboîtée

Dans cette section nous avons appliqué notre méthodes d'optimisation sur un code très intéressant dans le calcul haute performance (Mulmatvec : multiplication matrice-vecteur) sur des matrices creuses générées par la méthode de dissection emboîtée [58] en utilisant une nouvelle structure de données 2d-Packed-Nested Dissection (2d-P-ND). Le principe de cette approche est de stocker seulement les éléments non nuls de la matrice creuse dans le format 2d-P-ND. Ensuite, nous transformons le code original en un nouveau code qui utilise la structure 2d-PND. Enfin, Le code résultant est optimisé et parallélisé en utilisant l'outil de parallélisation et d'optimisation automatique PLuTo.

5.6.1 La méthode de dissection emboîtée (éléments finis)

Les analyses techniques des systèmes mécaniques ont été traitées en déduisant des équations différentielles reliant les variables à travers des principes physiques de base tels que l'équilibre, conservation de l'énergie, conservation de la masse, lois de la thermodynamique, équations de Maxwell et lois du mouvement de Newton. Cependant, une fois formulés, la résolution des modèles mathématiques obtenus est souvent impossible, en particulier lorsque ceux-ci sont des équations aux dérivées partielles non linéaires. La méthode des éléments finis FEM (dissection emboîtée) consiste à remplacer la structure physique à étudier par un nombre finis d'éléments ou de composants discrets qui représentent un maillage. Ces éléments sont liés entre eux par un nombre de points appelés nœuds. On considère d'abord le comportement de chaque partie indépendante, puis on assemble ces parties de telle sorte qu'on assure l'équilibre des forces et la compatibilité des déplacements réels de la structure en tant qu'objet continu. l'illustration des exemples de maillage des structures mécaniques est présentée sur la figure 5.26. La méthode des éléments finis est donc une technique récente à caractère pluridisciplinaire car elle met en œuvre les connaissances de trois disciplines de base :

— La mécanique des structures : élasticité, résistance des matériaux, dynamique, plasticité, etc...

— L'analyse numérique : méthodes d'approximation, résolution des systèmes linéaires, des problèmes aux valeurs propres, etc.

— L'informatique appliquée : techniques de développement et de maintenance de grands logiciels.

Les étapes de l'étude d'un problème en utilisant la méthode des éléments finis sont toujours :

1. Diviser la structure en morceaux (éléments avec nœuds) (discrétisation/maillage)
2. Connecter (assembler) les éléments aux noeuds pour former un système approximatif d'équations de la structure entière.
3. Résolver le système d'équations impliquant des quantités inconnues au niveau des nœuds (déplacements par exemple)
4. Calculer les quantités souhaitées sur des éléments sélectionnés.

La méthode de dissection emboîtée ordonne les nœuds du maillage (nous avons utilisé un maillage carré $n \times n$ pour notre expérimentation) pour représenter la matrice A. Le graphe d'une matrice A de dimension $m (m = n \times n)$ est constitué d'un ensemble de m sommets dont certains sont reliés par des arêtes tel que :

— Chaque sommet représente une inconnue du système linéaire.

— Chaque arête existante entre deux sommets i et j indique que le coefficient A_{ij} est non nul.

— Les autres coefficients de la matrice A sont nuls.

Les deux figures 5.27 et 5.28 représentent le graphe et la matrice générée par la technique de dissection emboîtée pour $n = 3$ et $m = 9$ respectivement.

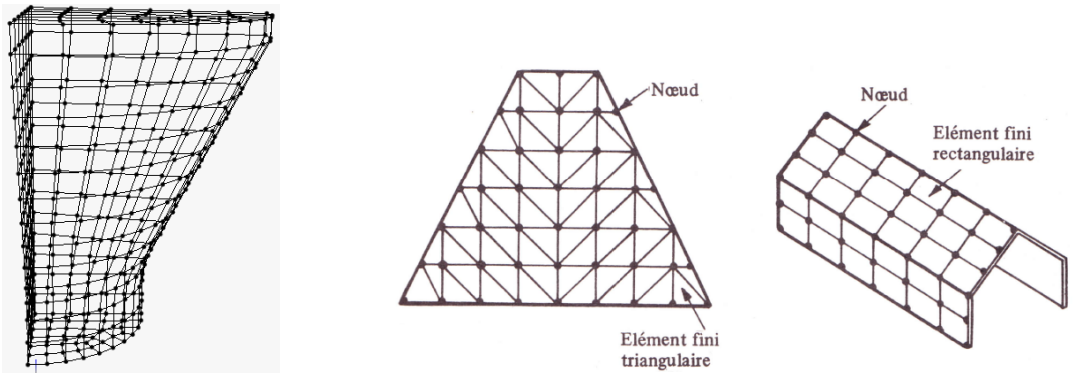


FIGURE 5.26 – Exemples de maillage des structures mécaniques.

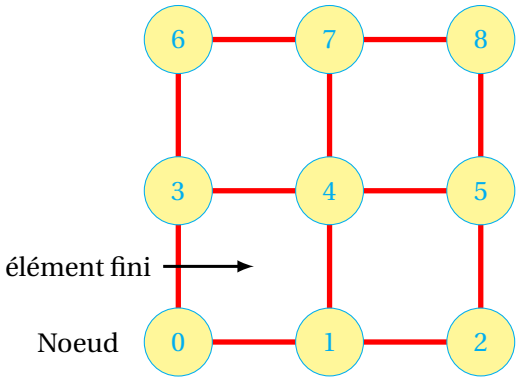


FIGURE 5.27 – le graphe de maillage.

	0	1	2	3	4	5	6	7	8
0	X	X	0	X	0	0	0	0	0
1	X	X	X	0	X	0	0	0	0
2	0	X	X	0	0	X	0	0	0
3	X	0	0	X	X	0	X	0	0
4	0	X	0	X	X	X	0	X	0
5	0	0	X	0	X	X	0	0	X
6	0	0	0	X	0	0	X	X	0
7	0	0	0	0	X	0	X	X	X
8	0	0	0	0	0	X	0	X	X

FIGURE 5.28 – La matrice générée par la méthode de dissection emboîtée.

5.6.2 La transformation 2d-P-ND

Dans notre approche 2d-P-ND, nous proposons une transformation permettant de stocker seulement les diagonales non-nuls de la matrice générée par la méthode de dissection emboîtée dans une nouvelle structure de données. Cette dernière, est une matrice de 5 lignes et m colonnes. La structure de données après la transformation 2d-P-ND de la matrice de l'exemple précédent est illustrée dans la figure 5.29.

	0	1	2	3	4	5	6	7	8
0	X	X	X	X	X	X			
1	X	X	0	X	X	0	X	X	
2	X	X	X	X	X	X	X	X	X
3	X	X	0	X	X	0	X	X	
4	X	X	X	X	X	X			

FIGURE 5.29 – La matrice résultante après la transformation 2d-P-Nd.

On peut représenter la référence (x, y) dans notre approche 2d-P-ND par l'équation 5.5.

$$f_t(x,y) = \begin{cases} (x-y+2,x) & \text{if } y = x \vee y = x+1 \\ (x-y+2,y) & \text{if } y = x-1 \\ (x-y+n,x) & \text{if } y = x+n \\ (x-y-n+4,y) & \text{if } y = x-n \end{cases} \tag{5.5}$$

Cette fonction de transformation unimodulaire peut être réécrite dans la représentation matricielle comme illustré dans l'équation 5.6

$$f_t(x,y) = \begin{cases} \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 2 \\ 0 \end{pmatrix} & \text{if } y = x \vee y = x+1 \\ \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 2 \\ 0 \end{pmatrix} & \text{if } y = x-1 \\ \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} n \\ 0 \end{pmatrix} & \text{if } y = x+n \\ \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 4-n \\ 0 \end{pmatrix} & \text{if } y = x-n \end{cases} \tag{5.6}$$

5.6.3 Résultats expérimentaux

La transformation 2d-P-ND est affine, et par conséquent, une transformation supportée par le modèle polyédrique (on peut paralléliser et optimiser automatiquement les codes transformés). Affin d'illustrer l'efficacité de notre approche sur les architectures multi-cœurs, nous avons utilisé une machine d'expérimentation équipée d'un processeur Intel i7-3770 qui contient 4 coeurs avec 8 threads. Nous avons aussi utilisé le compilateur gcc avec les options -openmp,-msse3 et -O3 sur un système d'exploitation Linux (version 3.16.0). La figure 5.30 montre les temps d'exécution obtenus pour le benchmark produit matrice vecteur (MulMatVec) avec déférentes tailles de matrices générées par l'application de la méthode de dissection emboîtée sur des éléments carré de taille n égale 200, 220, 250, 270, 290 et 300 (les matrices générées sont de taille M = $n \times n$). Le tableau 5.4 montre les six versions de codes utilisés dans la comparaison.

Pour les déférentes tailles de matrices, on peut remarquer que les temps d'exécution les plus élevés sont toujours ceux correspondant aux codes Naive, optimisé et 2d-P-Nd. On peut résumer les raisons de cette remarque dans les trois points suivants :

— Les trois codes Naive, Optimisé et 2d-P-ND n'étaient pas parallélisés par un outil de parallélisation automatique, donc un seul cœur de processeur a été utilisé.

— Le stockage d'une grande quantité d'éléments nuls inutiles dans les versions naive et optimisé.

TABLEAU 5.4 – Version de codes utilisés.

Version de code	Description
Naive	multiplication de tous les éléments de la matrice par les éléments du vecteur
Naive Par	multiplication de tous les éléments de la matrice par les éléments de vecteur en utilisant l'optimisation et la parallélisation par PLuTo
Optimisé	multiplier seulement les éléments non-zéros de la matrice par le vecteur
optimisé Par	multiplier seulement les éléments non-zéros de la matrice par le vecteur avec l'optimisation et la parallélisation par PLuTo.
2d-P-ND	multiplier la matrice transformée en 2d-P-ND par le vecteur.
2d-P-ND Par	multiplier la matrice transformée en 2d-P-ND par le vecteur en utilisant l'optimisation et la parallélisation par PLuTo

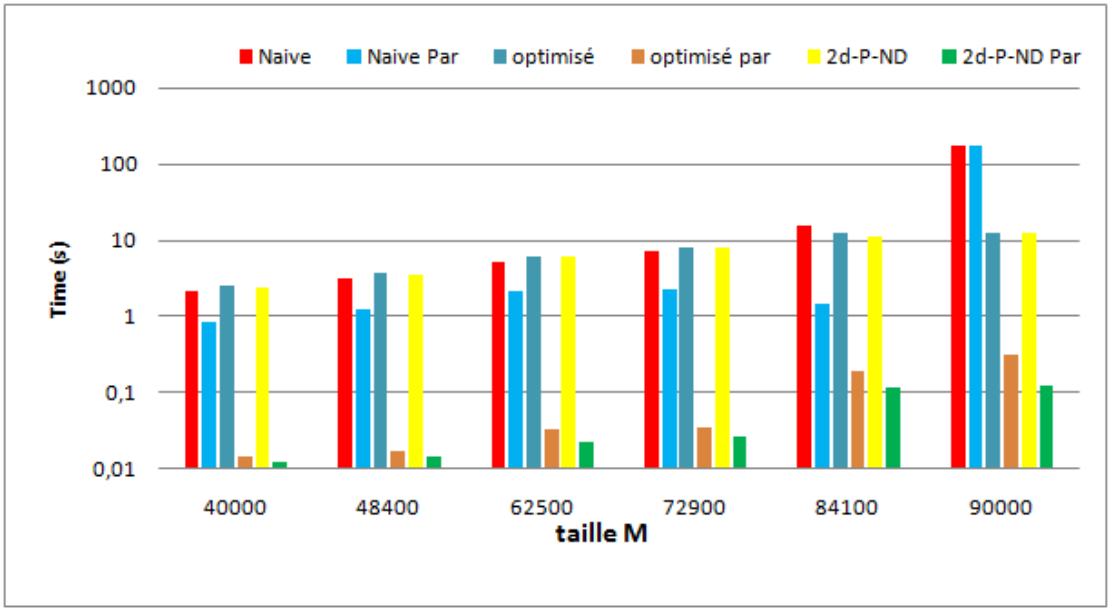


FIGURE 5.30 – Temps d'exécution avec les déférents taille de matrice pour le benchmark MulMatVec.

— la version 2d-P-ND contient plusieurs tests et opérations dans les indices de tableau .

Lorsque ces trois codes sont optimisés et parallélisés par Pluto, leurs performances sont considérablement améliorées pour les deux versions Optimisé et 2d-P-ND. L'amélioration est moins considérable pour la version naive. On peut noter aussi que les performances de notre codes 2d-P-ND parallélisé sont toujours meilleures que les deux autre versions parallèles. La raison de cette performance est que le code 2d-P-ND accède globalement à moins de mémoire, et par conséquent il y a moins de pression sur la hiérarchie de la mémoire et sur les caches en particulier, alors que le code original effectue des calculs sur des zéros inutiles. On peut conclure que l'approche 2d-P-ND combinée avec l'outil d'optimisation et de parallélisation automatique Pluto est capable d'améliorer la performance d'un code très intéressant dans le domaine de calcul haute performance (multiplication matrice vecteur) sur des matrices creuses générées par la méthode de dissection emboitée.

5.7 Conclusion

Dans ce chapitre, nous avons proposé une nouvelle structure de stockage de données qui permet de stocker les éléments non nuls des matrices triangulaires et des matrices en bande afin d'optimiser certains codes de calcul matriciel et d'atteindre des meilleurs performances dans les architectures multi-cœurs. Les fonctions d'accès aux nouvelles structures 2d-Packed sont des fonctions affines. Par conséquence, le code transformé peut être optimisé et parallélisé à l'aide d'outils polyédriques automatiques, tels que le framework source à source Pluto. Nos résultats expérimentaux montrent une amélioration significative des performances pour des codes de calcul matriciels triangulaires et en bande, par rapport à d'autres formats généralement utilisés pour le stockage de matrices creuses. Notre approche est plus générale que celles décrites dans l'état de l'art puisqu'elle peut être appliquée à n'importe quel calcul matriciel. Elle pourrait être implémenté comme un outil automatique qui transforme des programmes manipulant des matrices en bande et triangulaires en des programmes manipulant des structures de format 2d-Packed, ce qui permet de tirer pleinement avantage des compilateurs polyédriques bien établis. Dans la dernière partie de ce chapitre, on a appliqué une variante de notre transformation 2d-Packed sur des matrices générées par la méthode de dissection emboîtée. les résultats expérimentaux montrent l'efficacité de notre approche pour l'amélioration des performances dans les architectures multi-cœurs. Dans le dernier chapitre, on va faire une comparaison entre deux types d'allocation mémoire statique et dynamique en appliquant notre approche proposé 2d-Packed ainsi que sur un ensemble de benchmarks de la suite Polybench.

Chapitre 6

Effet du type d'allocation mémoire sur les performances des programmes optimisés

Sommaire

6.1 Introduction

Le modèle polyédrique améliore automatiquement les performances parallèles et la localité des données des noyaux d'algèbre linéaire réguliers. Dans les travaux précédents, nous avons utilisé une nouvelle structure de données, 2d-Packed, pour stocker uniquement les éléments non zéros de matrices creuses régulières (triangulaires et en bande) allouées dynamiquement pour différentes opérations d'algèbre linéaire de base, et utilisé Pluto pour les paralléliser et les optimiser. À notre grande surprise, il y avait d'énormes écarts dans nos mesures des temps d'exécution de ces noyaux qui n'étaient dus qu'au mode d'allocation : en tant que tableaux déclarés statiquement ou en tant que tableaux de pointeurs alloués dynamiquement. Dans ce chapitre, nous comparons les performances de différents noyaux d'algèbre linéaire en utilisant différents modes d'allocation de tableaux [14]. Nous présentons notre étude détaillée des raisons possibles de la différence de performances de nos noyaux et de certains noyaux d'algèbre linéaire de la suite PolyBench, sur deux architectures différentes : un double processeur AMD 2×12 coeurs (Magny-Cours) et un 2×10 coeurs hyperthreadé Intel Xeon (Haswell-EP). Nous concluons que l'allocation de mémoire statique ou dynamique a un impact sur les performances dans de nombreux cas, et que l'architecture du processeur et les décisions du compilateur gcc peuvent provoquer des variations importantes et parfois surprenantes, en faveur de l'un ou l'autre mode d'allocation.

6.2 Allocation dynamique Vs Statique dans le format 2d-Packed

La différence fondamentale entre les allocations statiques et dynamiques est que l'allocation statique se compose de tailles fixes qui ne peuvent pas changer tout au long du programme, tandis que l'allocation dynamique est infiniment flexible. Afin de connaître la meilleure façon d'allocation dans notre approche de transformation 2d-Packed avec des matrices triangulaires, nous avons utilisé les deux types d'allocation et nous avons fait une comparaison.Dans l'allocation dynamique, nous avons stocké la matrice triangulaire au format 2d-Packed en utilisant un tableau de pointeurs de taille $N \times (N+1)/2$ lorsque N est impair, et nous l'avons stockée dans un tableau de pointeurs de taille $N \times (N/2)$, en plus d'une colonne supplémentaire de taille $N/2$ lorsque N est pair. Le code d'allocation pour les matrices triangulaires au format 2d-Packed est illustré dans la figure 6.1. Les caractéristiques de cette allocation sont :

1. La mémoire est allouée "à la volée" pendant l'exécution.
2. La quantité exacte d'espace ou le nombre d'éléments ne doivent pas être connus à l'avance par le compilateur.
3. elle rend utilisation intensive de pointeurs.

```
1  //allocation dynamique de la structure 2d-Packed pour les matrices
       triangulaires
2  double **allocTriPackMat(int n)
3  {
4    double **mat;
5    mat=malloc(n*sizeof(int *));
6    for (int i=0; i<n/2; i++)
7        mat[i]=calloc((n+1)/2,sizeof(double));
8    for (; i<n; i++)
9        mat[i]=calloc(((n/2)+1),sizeof(double));
10   return mat;
11 }
```

FIGURE 6.1 – Le code d'allocation dynamique.

Dans l'allocation statique, nous avons stocké notre matrice triangulaire au format 2d-Packed

en utilisant un tableau de taille N × ((N/2) + 1) dans les deux cas (N est impair ou pair). Les caractéristiques de cette allocation sont :

1. les stockage des variables dans la mémoire est effectué par le compilateur.
2. La taille exacte du tableau doit être connue au moment de la compilation.

6.2.1 Temps d'exécution et performance

Nous avons appliqué notre approche pour optimiser et paralléliser quatre noyaux d'algèbre linéaire double précision à l'aide de matrices triangulaires : factorisation de Cholesky, multiplication de deux matrices (MatMul), solveur matrice-matrice (SolvMat) et factorisation sspfa. Nous avons comparé les temps d'exécution des noyaux optimisés et parallélisés automatiquement au format 2d-Packed utilisant l'allocation statique à ceux utilisant l'allocation dynamique. Tous les résultats rapportés sont obtenus à partir d'une moyenne de cinq exécutions de chaque noyau en utilisant des matrices de taille N = 8000. Le temps d'allocation n'était pas inclus dans les mesures de temps. Les expériences ont été exécutées sur deux ordinateurs différents :

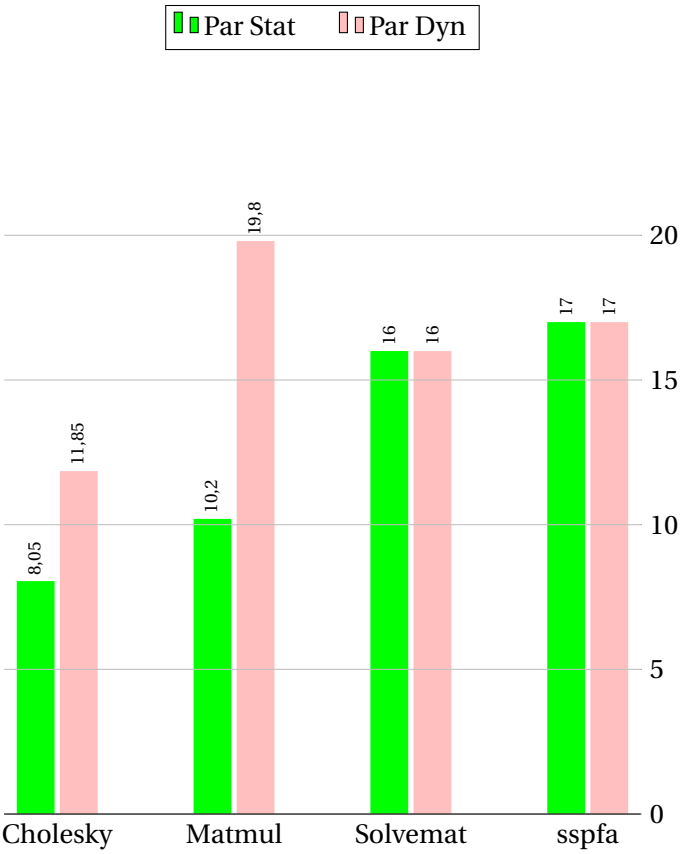
— **config1** est un Intel Xeon E5-2650 v3 à double socket (Haswell-EP) de 2 × 10 cœurs hyper-threadés, avec prise en charge AVX2 (256 bits) ;

— **config2** est un AMD Opteron 6172 (Magny-Cours) à double socket de 2×12 cœurs, avec prise en charge SSE (128 bits).

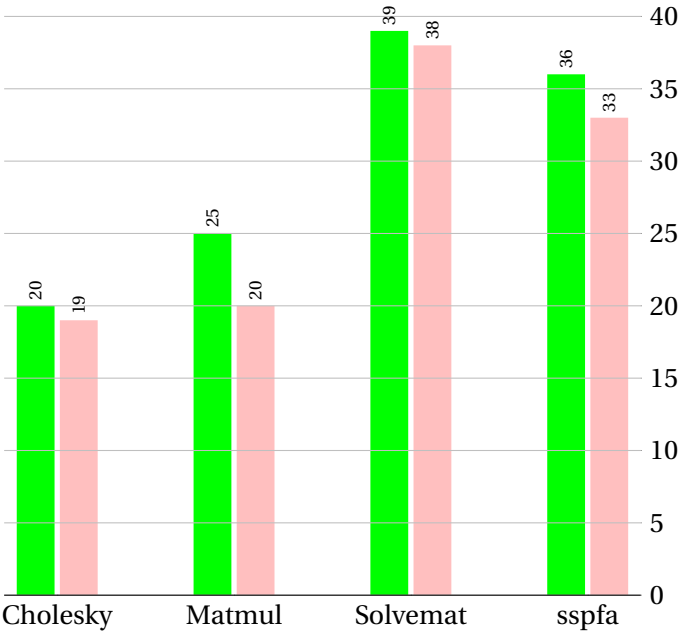
Les deux configurations exécutent exactement le même système : Ubuntu (bionic) avec un noyau standard Linux 4.0.15. Nous avons compilé nos codes sur les deux configurations en utilisant exactement la même version de gcc (7.3.0), avec les options `-O3 -march = native -fopenmp`. Nous avons utilisé Pluto 0.11.4 avec les options `-tile -parallel` pour optimiser et paralléliser les programmes avec des tailles de tuiles par défaut.

La figure 6.2 montre les temps d'exécution obtenus sur nos machines config1 et config2. Dans les deux sous-figures, chaque couple de barres représente respectivement les temps d'exécution avec l'allocation statique et dynamique. Dans la première configuration (config1, figure 6.2a), on peut remarquer que les temps d'exécution dans les deux cas sont presque les mêmes pour sspfa et Solvemat, tandis que pour Cholesky et MatMul l'allocation statique fonctionne mieux que l'allocation dynamique. Dans la deuxième configuration (config2, figure 6.2b) l'allocation dynamique conduit à de meilleures performances par rapport à l'allocation statique pour tous ces benchmarks.

Il existe des écarts notables entre ces temps d'exécution selon que les tableaux sont alloués statiquement ou dynamiquement. Le compilateur est le même, le noyau est le même, alors quelles pourraient être les raisons de ces différences ? Pour répondre à cette question, nous avons effectué un ensemble d'expériences supplémentaires sur le benchmark MatMul, car c'est celui qui présente des différences importantes dans les temps d'exécution et un effet opposé complet du type d'allocation de mémoire sur nos deux plates-formes de test. La section suivante présente une analyse détaillée des performances de la référence MatMul à l'aide des deux modes d'allocation de mémoire.



(a) Temps d'exécution pour la machine config1 (secondes).



(b) Temps d'exécution pour la machine config2 (secondes).

FIGURE 6.2 – Comparaison de Temps d'exécution entre les allocations statique et dynamique

6.3 Etude expérimentale de la multiplication matricielle en format 2d Packed

L'algorithme original de multiplication de deux matrices carrées triangulaires inférieures du même ordre N est donné par l'algorithme 5

Algorithm 5: Multiplication des matrices triangulaires inferieures

Input: Deux matrices triangulaires inferieures A et B d'ordre N

```
1 for i ← 0 to N − 1 do
2     for k ← 0 to i do
3         for j ← 0 to k do
4             C[i][j] ← C[i][j] + A[i][k] ∗ B[k][j];
5         end
6     end
7 end
```

Dans notre étude, nous avons utilisé deux variantes de la multiplication matricielle dans le format 2d-packed, où les matrices d'entrée A et B ainsi que la matrice de sortie C sont stockées dans des structures de données 2d-packed. La première variante du code de multiplication matricielle est obtenue en appliquant l'algorithme de transformation brut 2d-packed. Le code résultant, nommé C1, est illustré dans la figure 6.3. Ce code n'est composé que d'un nid de boucle parfait dans lequel la boucle la plus interne contient trois tests. La deuxième variante, nommée C2, est un code équivalent dans lequel les tests ont été supprimés grâce à la fission de boucles. Le code résultant, illustré sur la figure 6.4, est un nid de boucle non parfait ne contenant aucun test. Dans les sections suivantes, nous comparerons les performances des allocations statiques et dynamiques pour les codes C1 et C2, en mesurant leur temps d'exécution, le nombre d'instructions exécutées, le nombre de cache loads, le nombre de défauts cache et le nombre d'instructions vectorisées. Ces comparaisons sont illustrées sur les figures 6.5, 6.6, 6.7 et 6.8. Dans toutes ces expériences, les quatre barres représentent respectivement le code d'origine avec allocation statique, le code original avec allocation dynamique, le code optimisé et parallélisé par Pluto avec allocation statique et le code optimisé et parallélisé par Pluto avec allocation dynamique.

Dans la figure 6.5, en regardant individuellement chaque groupe de barres et en comparant la première à la troisième et la seconde à la dernière, nous voyons que l'optimisation et la parallélisation de nos codes en utilisant Pluto est toujours bénéfique. Pour le code C1 (avec tests), les figures 6.6 et 6.7 confirment que l'application de Pluto réduit le nombre d'instructions exécutées, les cache loads L1 et les défauts de cache. L'application de Pluto au code C2 (sans tests) réduit également le nombre de défauts de cache, mais cela augmente le nombre total d'instructions exé-

```
const int nn=n/2;
for (i=0 ; i<n ; i++) {
  for (k=0 ; k<=i ; k++) {
    for (j=0 ; j<=k ; j++) {
      if (i>nn && j>nn && k>nn)
        C[n-i-1][n-j] += A[n-i-1][n-k] * B[n-k-1][n-j];
      if (i>nn && j<=nn && k>nn)
        C[i][j] += A[n-i-1][n-k] * B[k][j];
      if (j<=nn && k<=nn)
        C[i][j] += A[i][k] * B[k][j];
    }
}
}
```

FIGURE 6.3 – Le code C1 : multiplication de deux matrices triangulaires dans le format 2d-packed.

```
const int nn=n/2;
for(i=0; i<nn+1; i++) {
  for(k=0; k<=i; k++) {
    for(j=0; j<=k; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
for(i=nn+1; i<n; i++) {
  for(k=0; k<=i && k<nn+1; k++) {
    for(j=0; j<=k; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
  for(k=nn+1; k<=i; k++) {
    for(j=0; j<=k && j<nn+1; j++) {
      C[i][j] += A[n-(i)-1][n-(k)] * B[k][j];
    }
    for(j=nn+1; j<=k; j++) {
      C[n-(i)-1][n-(j)] += A[n-(i)-1][n-(k)] * B[n-(k)-1][n-(j)];
    }
  }
}
```

FIGURE 6.4 – Le code C2 : multiplication de deux matrices triangulaires après la fission de boucles dans le format 2d-packed.

cutées et le nombre de cache loads, probablement en tant qu'effet de tiling et de parallélisation. Les sous-sections suivantes présentent une analyse détaillée de ces graphiques pour comparer les allocations statiques et dynamiques.

6.3.1 Temps d'exécution

En comparant chaque couple de barres de la figure 6.5, nous observons que la différence de temps d'exécution entre la version statique et la version dynamique varie de 0.80× à 1.95×. Dans le code C2 d'origine (séquentiel), l'allocation statique ou dynamique n'influence pas les temps d'exécution (deux premières barres dans les deux groupes de barres de droite). Dans les autres versions (originales et parallèles) et configurations (config1 ou config2), l'allocation statique fonctionne mieux que l'allocation dynamique, à l'exception du code optimisé pour Pluto sur config2 : la version dynamique fonctionne mieux que la version statique (par exemple 20s *vs* 25s pour le code C1).

6.3.2 Nombre total d'instructions :

Dans la figure 6.6a, on peut remarquer que le nombre d'instructions exécutées du code C1 est similaire sur les deux configurations. Par exemple : Instr(orig-static, config1, C1) ≃ Instr (orig-static, config2, C1) ≃ 1,950 milliards d'instructions. Pour le code C2, il est plus élevé sur config2 que sur config1, ce qui signifie que le compilateur a certainement généré des codes différents de C2 sur ces deux architectures différentes.

Pour le code C1, l'allocation dynamique augmente le nombre d'instructions exécutées dans le code d'origine, tandis qu'elle est réduite sur le code Pluto. Mais ces résultats ne sont pas corrélés au temps d'exécution.

Pour le code C2, le nombre d'instructions exécutées augmente lorsqu'il est optimisé et parallélisé par Pluto. Le code original est composé de trois boucles imparfaites imbriquées, et Pluto génère un code tuilé et parallélisé plus complexe de boucles imparfaites de profondeur six. Le nombre d'instructions exécutées dans l'allocation dynamique est supérieur à celui de l'allocation

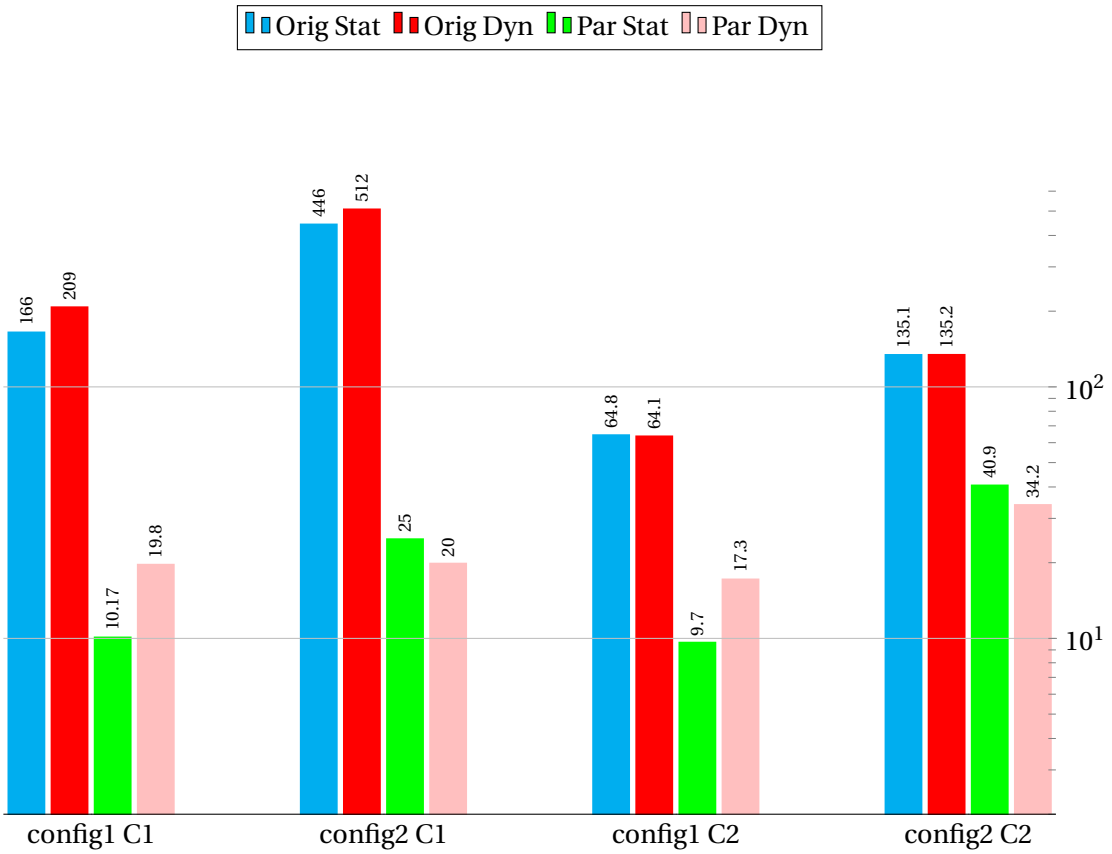


FIGURE 6.5 – Temps d'execution (en secondes, échelle logarithmique) pour les allocations statique et dynamique, avec les machines config1 et config2, pour les codes C1 et C2.

statique sur config1, probablement en raison du contrôle supplémentaire et de la pression supplémentaire sur l'allocation des registres de ce code complexe. Cet effet est le contraire sur les autres versions.

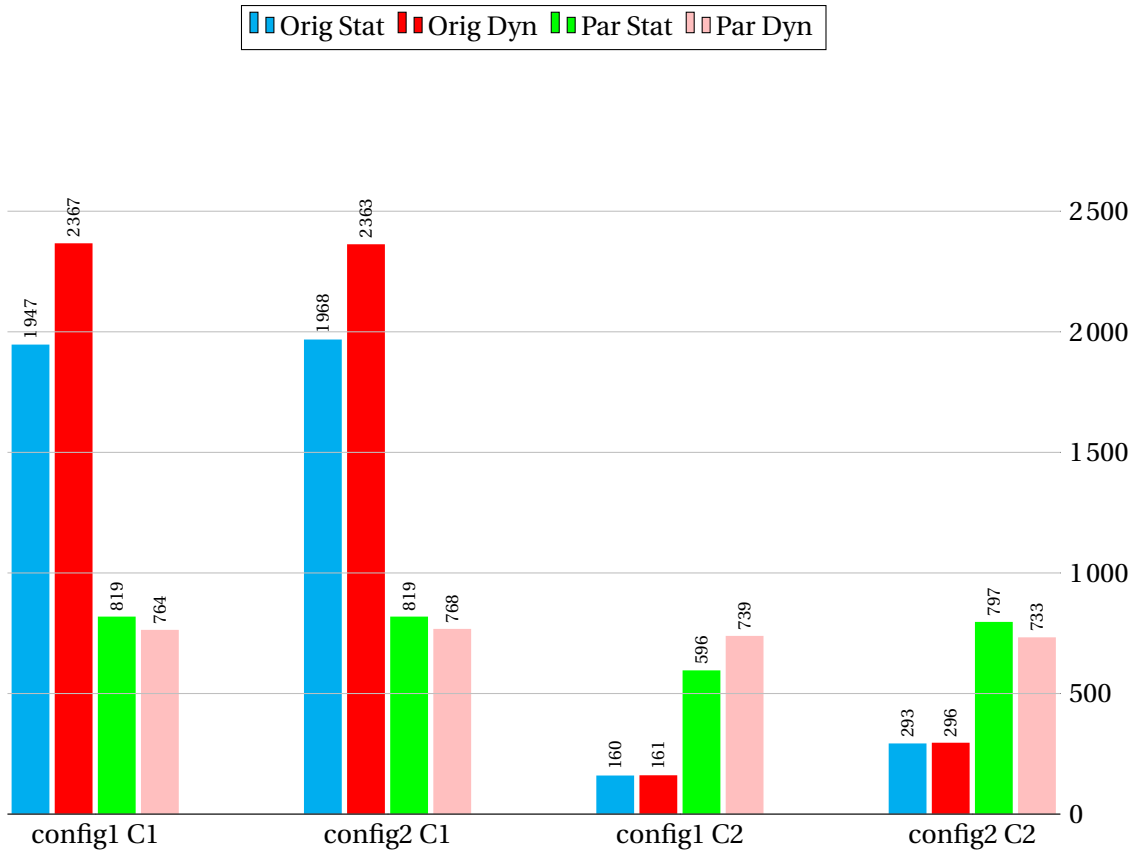
Globalement, ces mesures montrent que le compilateur semble prendre des décisions différentes sur nos deux architectures de test. Il existe une certaine corrélation entre la différence du nombre d'instructions exécutées et la différence de performances entre l'allocation statique et l'allocation dynamique illustrée sur la figure 6.5 à l'exception de la version optimisée de config1 pour le code C1.

6.3.3 Nombre de L1-dcache-loads

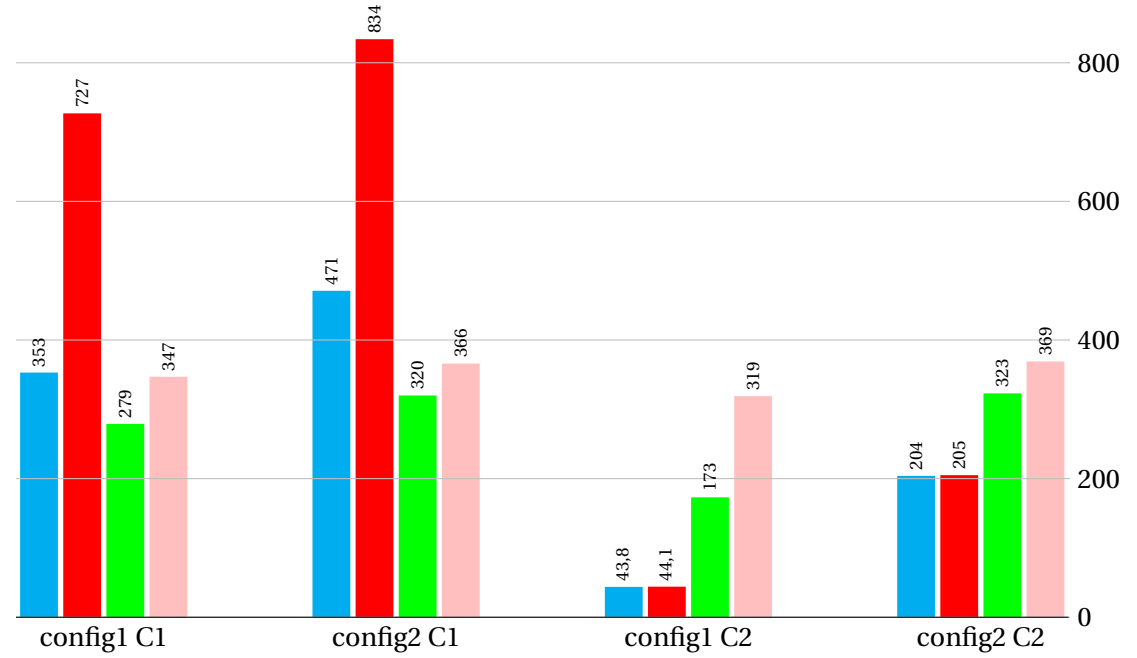
La figure 6.6b montre le nombre de L1-dcache-loads : cette fonction peut être utilisée pour déterminer combien de fois les ports de chargement sont utilisés. Dans tous les cas de cette expérience, le nombre de L1-dcache-loads dans l'allocation dynamique est supérieur à celui de l'allocation statique. La raison en est l'utilisation de pointeurs qui conduit évidemment à davantage d'accès mémoire pour accéder aux données. Pour la comparaison entre l'allocation de mémoire statique et dynamique, le nombre de L1-dcache-loads peut être en quelque sorte corrélé au temps d'exécution sur config1 mais pas pour les versions parallèles config2, où le temps d'exécution en allocation dynamique est meilleur que celui de l'allocation statique.

6.3.4 Nombre de défauts de cache L1 et L3

Le nombre de défauts de cache L1 et L3 est indiqué dans les figures 6.7a et 6.7b. Il y a parfois un rapport très important de défauts de cache par rapport au nombre total de loads (jusqu'à 19,7 /43,8 = 45% L1-dcache-misses pour config1/C2 avec allocation statique). Comme prévu, dans les codes optimisés de Pluto, les défauts de cache sont considérablement réduites dans la plupart des

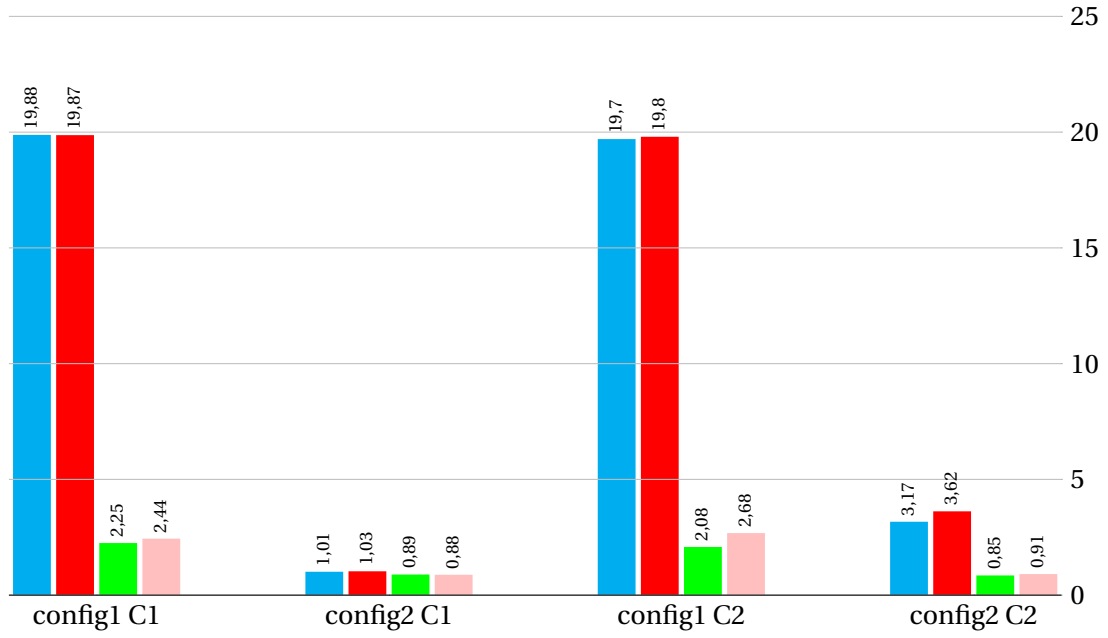


(a) Nombre d'instructions exécutées (Milliards).

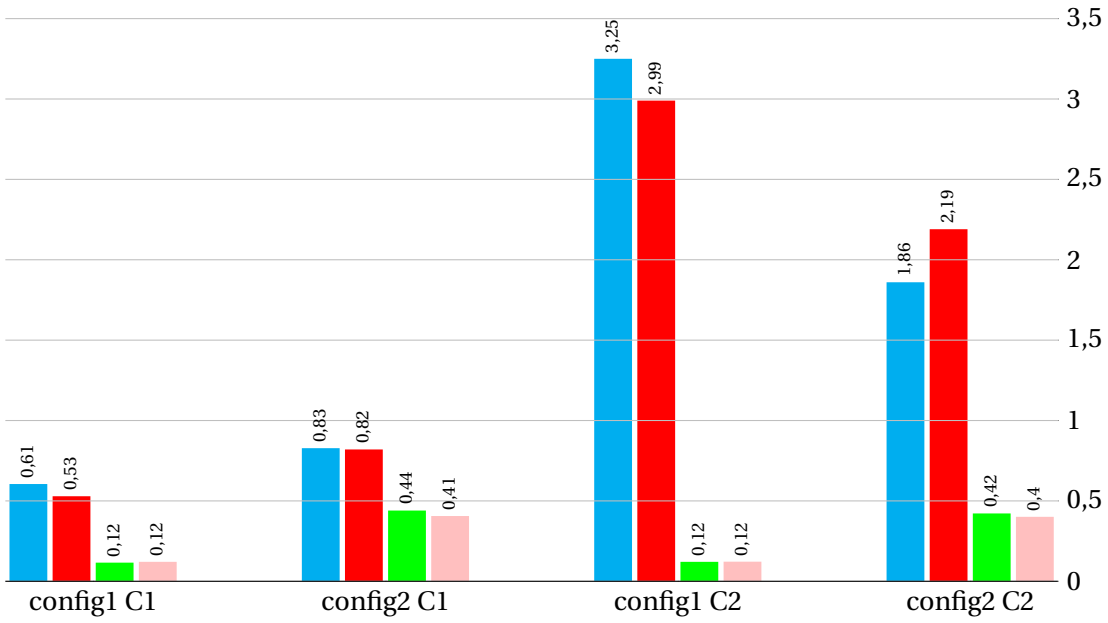


(b) Nombre de L1-dcache-loads (Milliards).

FIGURE 6.6 – Nombre d'instructions et L1-dcache-loads pour les allocations statique et dynamique, avec les machines config1 et config2, pour les codes C1 et C2.



(a) Nombre de L1-dcache-misses (Milliards).



(b) Nombre de L3-cache-misses (Milliards).

FIGURE 6.7 – Nombre de défauts de cache L1 et L3 pour les allocations statique et dynamique, avec les machines config1 et config2, pour les codes C1 et C2.

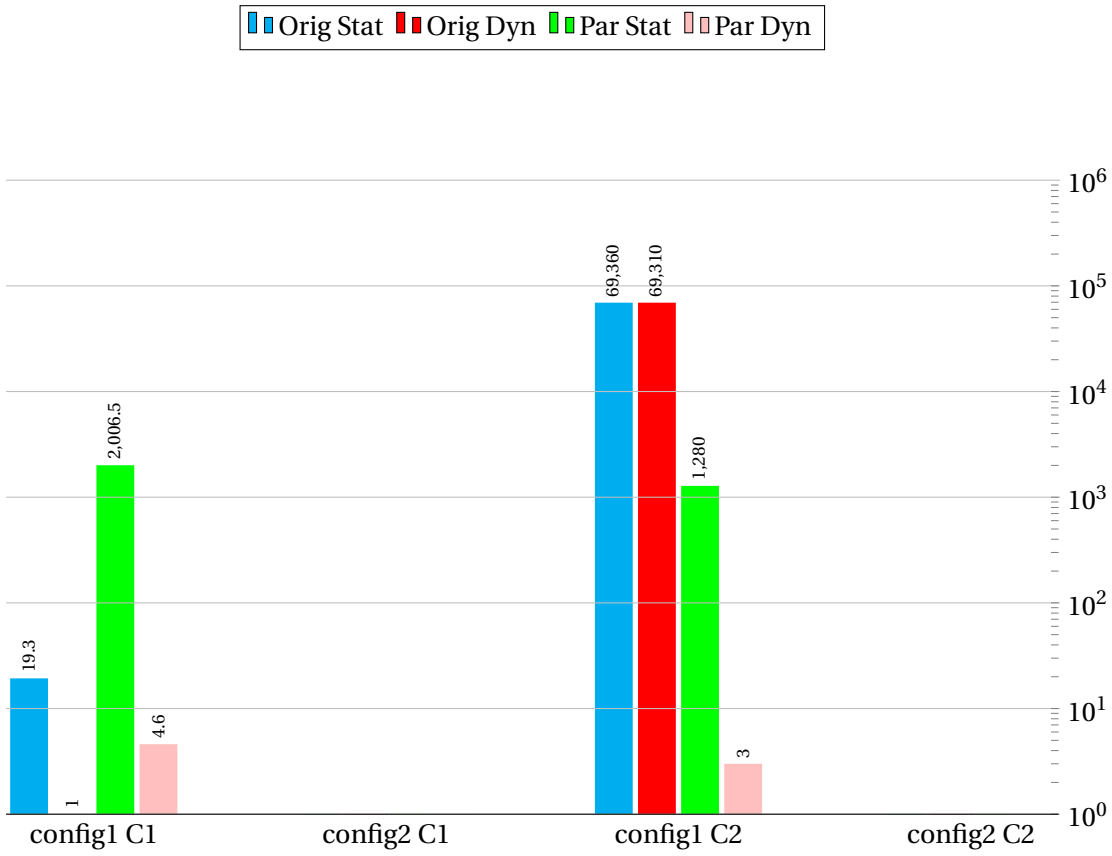


FIGURE 6.8 – Nombre d'instructions vectorisés (en Millions, échelle logarithmique) pour les allocations statique et dynamique, avec les machines config1 et config2, pour les codes C1 et C2.

cas (par exemple, 2,08 /173 = 1,2% dans le même exemple). La deuxième observation est que la somme du nombre de défauts de cache L1 et L3 est légèrement corrélée avec les temps d'exécution dans les versions optimisées de Pluto, à l'exception de la version config2/C2. Cependant, dans les versions originales des codes, ce n'est pas vrai.

6.3.5 Nombre d'instructions vectorisées

Dans cette dernière expérience, nous avons mesuré le nombre d'instructions vectorisées sur config1 uniquement, car ce compteur de performances n'est pas disponible sur notre ancien processeur config2. On peut clairement remarquer à partir de la figure 6.8 que le code original C2 est bien vectorisé, alors que les autres versions ne sont au mieux que partiellement vectorisées. Étant donné que le code C1 contient des tests dans les boucles les plus internes, il était plus difficile pour le compilateur de découvrir les opportunités de vectorisation dans ce code. Dans les deux cas, il y a plus d'instructions vectorisées dans les versions d'allocation statiques que dans les versions dynamiques. Nous pensons que c'est l'une des raisons des différences signalées précédemment, et cela explique pourquoi l'allocation statique sur config1 est nettement meilleure que sa version allouée dynamiquement. Cependant, sur config2, l'activation de l'option `fopt info vec` dans gcc n'a montré aucune différence entre l'allocation de mémoire statique et dynamique, de sorte que ces codes sont vectorisés de la même manière probablement sans beaucoup d'impact sur les performances.

6.3.6 Synthèse

La conclusion de ces mesures est qu'il est très difficile d'obtenir une explication simple sur la raison pour laquelle certains codes avec allocation statique fonctionnent mieux que ceux avec allocation dynamique, ou l'inverse. Notre meilleure hypothèse est que les performances globales

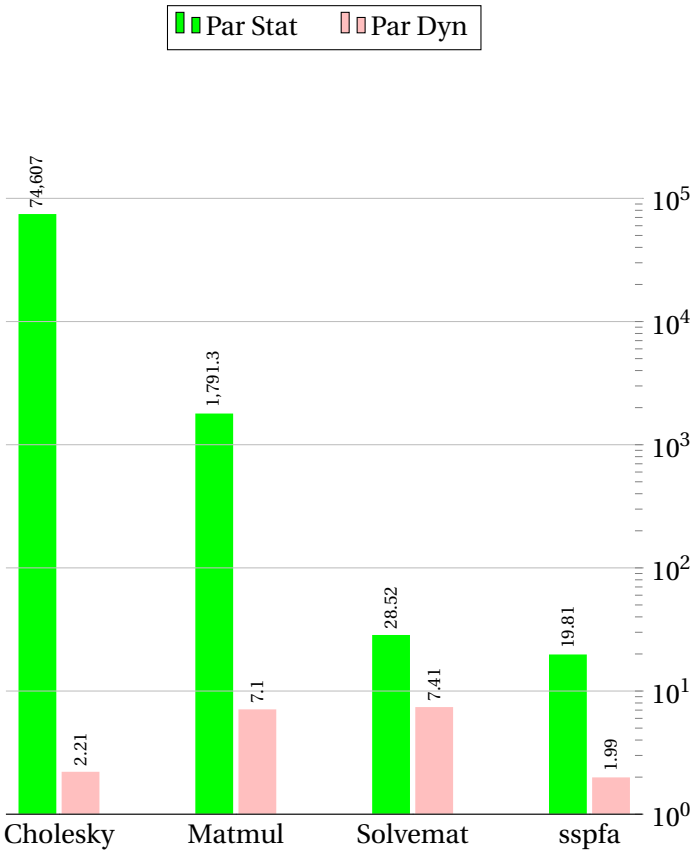


FIGURE 6.9 – Nombre d'instructions vectorisées (Millions, échelle logarithmique) pour les allocations statique et dynamique avce la machine config1 pour les benchmarks en format 2d-packed.

(figure 6.5) peuvent être déduites d'une combinaison de (1) Nombre total d'accès à la mémoire (2) Nombre de défauts de cache et (3) Nombre d'instructions vectorisées.

Cependant, sur config1 le nombre de boucles vectorisées est probablement le principal facteur affectant les performances.

6.4 Nombre d'instructions vectorisées dans tous les benchmarks 2d-Packed

Pour confirmer l'hypothèse proposée précédement, nous avons comparé le nombre d'instructions vectorisées sur config1 en utilisant les deux types d'allocations statique et dynamique pour les quatre benchmarks présentés précédemment (figure 6.2a). Les résultats sont présentés dans la figure 6.9. On voit bien que le nombre d'instructions vectorisées en allocation statique est toujours supérieur à celui en allocation dynamique (ex : Nbr_vec_instr (Cholesky, statique) = 74 607 millions d'instructions, Nbr_vec_instr (Cholesky, dynamique) = 2,21 millions d'instructions). Mais il est très significatif dans les deux premiers benchmarks (Cholesky et Matmul). Ces derniers sont ceux qui présentent le plus de variation de performances sur la figure 6.2a. De toutes les expériences précédentes, nous concluons que sur config1 le nombre d'instructions vectorisées est probablement l'un des principaux facteurs affectant les performances des benchmarks étudiés jusqu'à présent.

Dans la section suivante, nous nous concentrerons sur la suite Polyédrique des Benchmarks PolyBench [102] et vérifierons si le nombre d'instructions vectorisées affecte également les performances de ces benchmarks dans les allocations de mémoire statiques et dynamiques.

6.5 Allocation dynamique Vs allocation statique dans PolyBench

Nous avons étudié la différence de performance des deux modes d'allocation de mémoire en considérant huit noyaux d'algèbre linéaire de la suite PolyBench : `2mm`, `3mm`, `atax`, `bicg`, `mvt`, `trisolv`, `lu` et `cholesky`. Toutes les expériences sont réalisées sur notre machine config1, basée sur l'Intel Xeon (Haswell-EP). Les tailles de matrice utilisées dans ces expériences sont de 2 000 pour les algorithmes $O(N^3)$ (`2mm`, `3mm`, `cholesky` et `lu`), et 20 000 pour les algorithmes $O(N^2)$ (`atax`, `bicg`, `mvt` et `trisolv`).

PolyBench fournit un macro C pour déclarer ses données en tant que tableaux alloués par pile (déclarés dans la fonction principale) ou en tant que tableaux multidimensionnels alloués en tas (avec un appel `posix_mem -align`, par défaut). Nous avons compilé tous les programmes avec gcc version 7.3.0 dans les deux mode d'allocations statique et dynamique.

6.5.1 Les mesures :

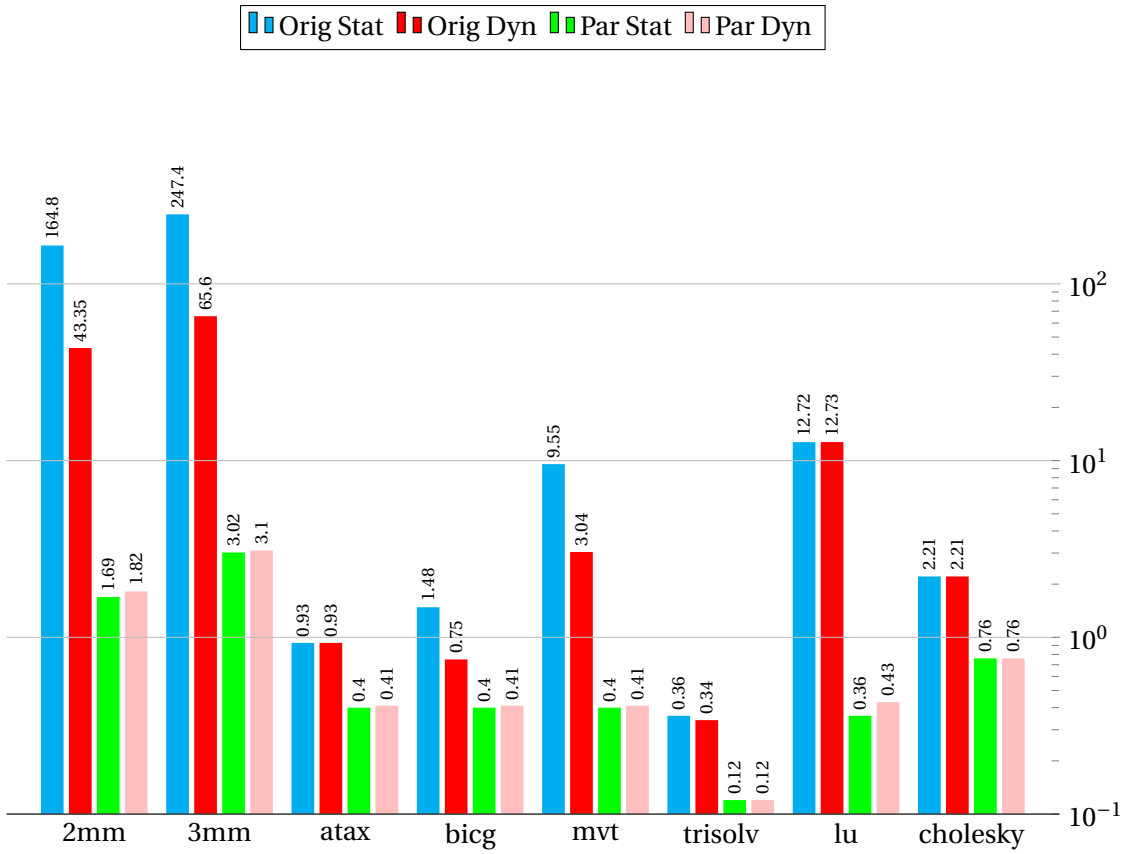
Nous avons montré les temps d'exécution des différents benchmarks dans la figure 6.10a, et nous avons tracé le nombre d'instructions vectorisées dans la figure 6.10b.

Les temps d'exécution de tous les benchmarks optimisés et parallélisés (couples de barres de droite) sont très similaires, que les tableaux soient alloués statiquement ou dynamiquement. Une exception notable est `lu` : 0,36s pour la version statique contre 0,43s pour la version dynamique, un ralentissement de 20%. Sur la figure 6.10b, la différence entre le nombre d'instructions vectorisées pour ces deux exécutions n'est que de 3%, donc la principale raison de cet écart n'est probablement pas la vectorisation. L'autre exception est `2mm`, où la différence de 8% du temps d'exécution peut être attribuée à une différence de 12% du nombre d'instructions vectorisées.

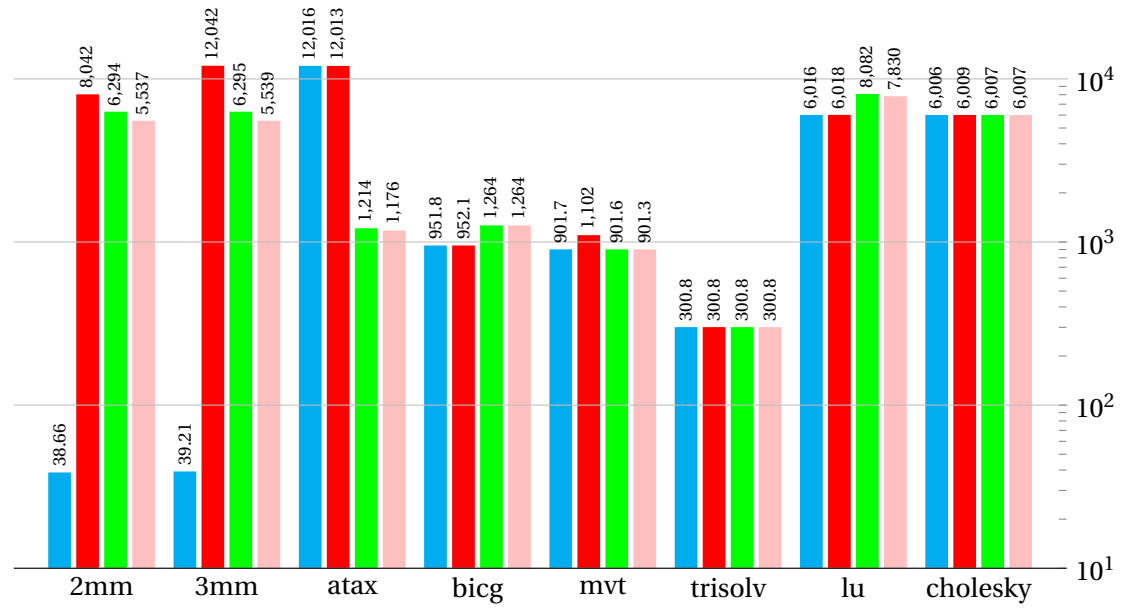
Pour les codes originaux séquentiels (couples de barres de gauche), les temps d'exécution sont très proches pour `atax`, `trisolv`, `lu` et `cholesky`. Mais les quatre autres repères (`2mm`, `3mm`, `bicg` et `mvt`) montrent des différences significatives : l'allocation dynamique fonctionne mieux que l'allocation statique. Le nombre d'instructions vectorisées est bien meilleur pour les versions allouées dynamiquement de `2mm` et `3mm`. Cela peut clairement expliquer la différence pour ces deux repères. La différence du nombre d'instructions vectorisées dans `mvt` est de 22%, ce qui n'est pas suffisant pour expliquer une différence de temps d'exécution de 3×. Pour `bicg`, il n'y a pas de différence. Donc, pour `bicg` et `mvt`, nous devons trouver une autre explication que la vectorisation pour ces accélérations. À ce stade, il n'y a aucune explication pour la différence de performances entre l'allocation statique et l'allocation dynamique de trois benchmarks : la version Pluto de `lu` et les versions originales de `bicg` et `mvt`.

Les autres compteurs de performances sont présentés dans les figure 6.11 et 6.12. La différence de performances pour les versions originales de `bicg` et `mvt` peut être corrélée au nombre de L1-dcache-loads de la figure 6.11b et au nombre de défauts de cache de figure 6.12a et 6.12b. Le nombre total de loads est supérieur à 64% pour `bicg` et 38% pour `mvt` dans les versions d'allocation statique, et le même genre de différence est notable sur les deux repères d'origine `2mm` et `3mm`.

Pour le code parallèle `lu` cette différence est de 12,5%, ce qui pourrait aussi expliquer en partie la différence de temps d'exécution pour ce benchmark. Mais les autres benchmarks parallélisés montrent également une différence significative dans leur nombre de loads qui n'a pas eu d'impact sur leurs temps d'exécution. Les versions d'allocation dynamique ont environ 10% de loads mémoire en plus que les versions statiques, jusqu'à 17% pour `2mm`. Cependant, c'est probablement une conséquence des versions d'allocation statiques ayant des instructions un peu plus vectorisées que les instructions dynamiques, comme on le voit sur la figure 6.10b. Ainsi, pour les versions d'allocation dynamique de Pluto, il existe des instructions vectorisées plus exécutées, mais la pression sur la mémoire est plus élevée, ce qui, combiné, donne les mêmes performances globales. Il n'y a pas d'explication satisfaisante dans ces mesures pour la différence de performance entre l'allocation statique et dynamique du benchmark `lu` optimisés par Pluto .

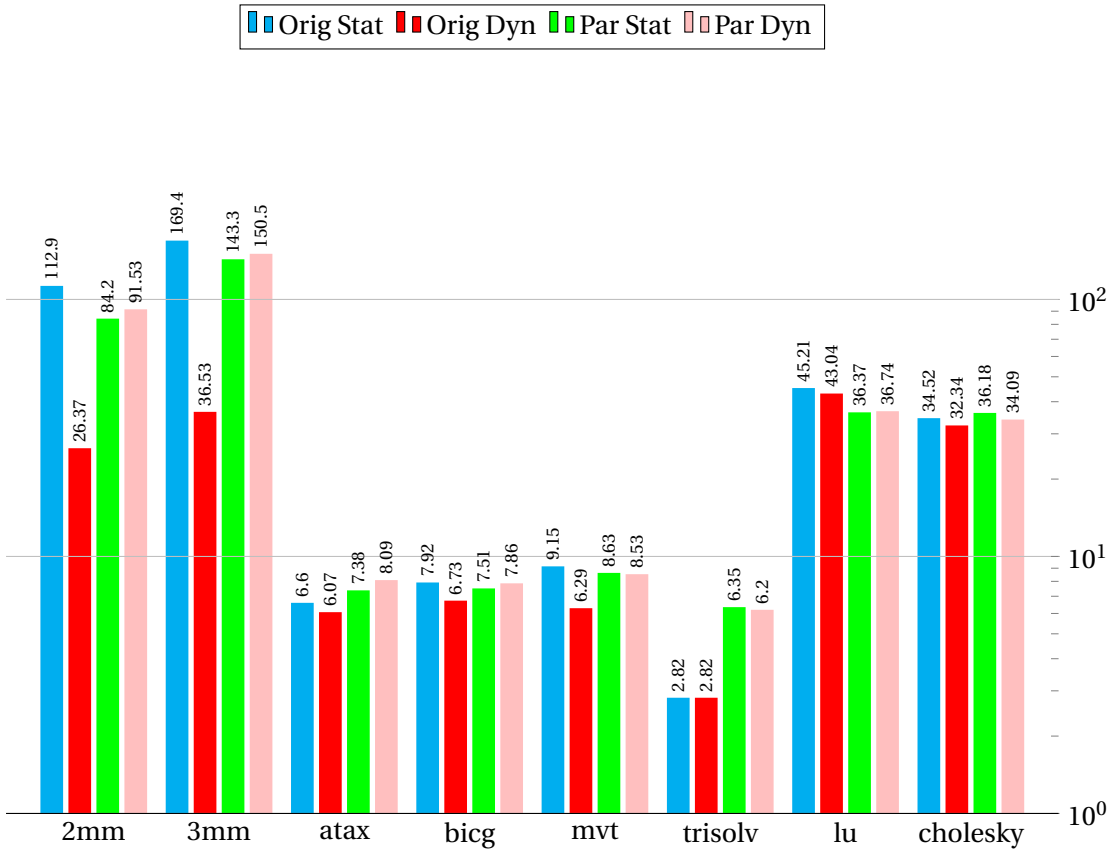


(a) Temps d'exécution (secondes, échelle logarithmique).

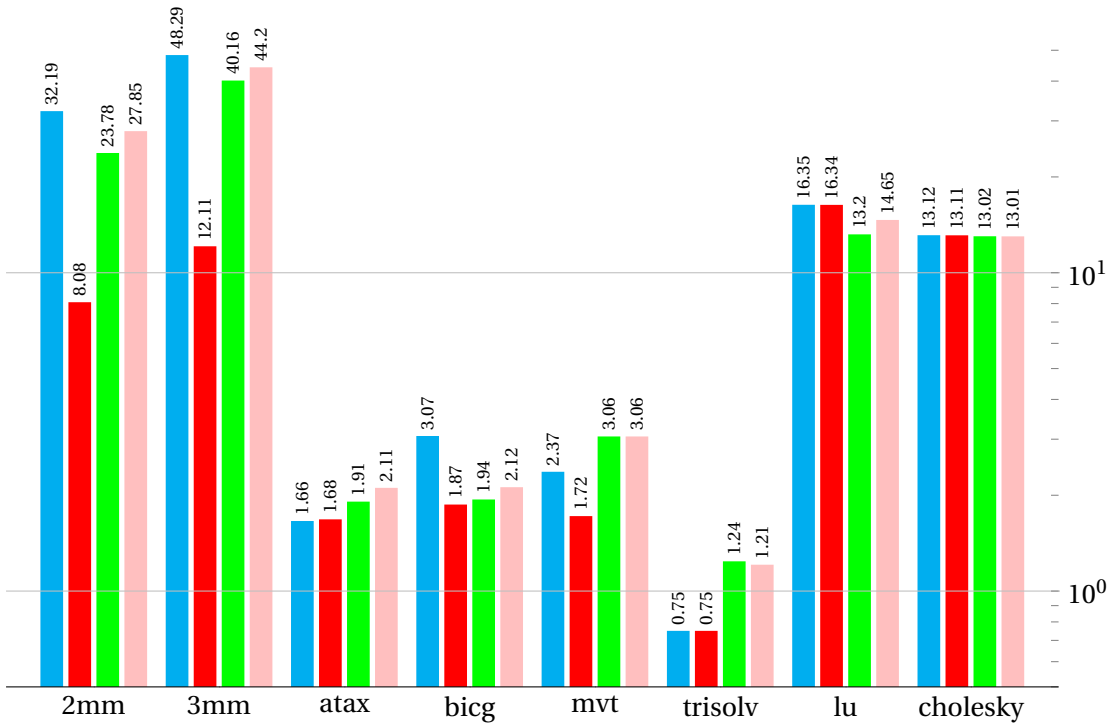


(b) Nombre d'instructions vectorisées (Millions, échelle logarithmique).

FIGURE 6.10 – Temps d'exécution et nombre d'instructions vectorisées pour les allocations statique et dynamique dans PolyBench.

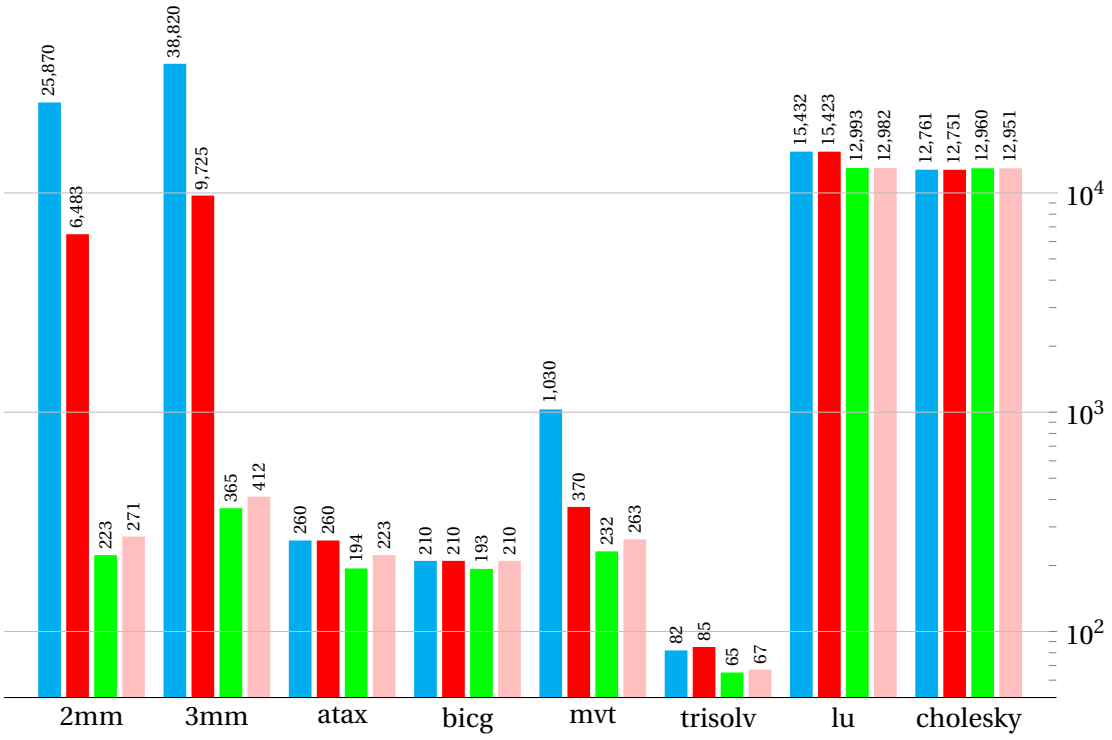


(a) Nombre d'instructions exécutées (Milliards).

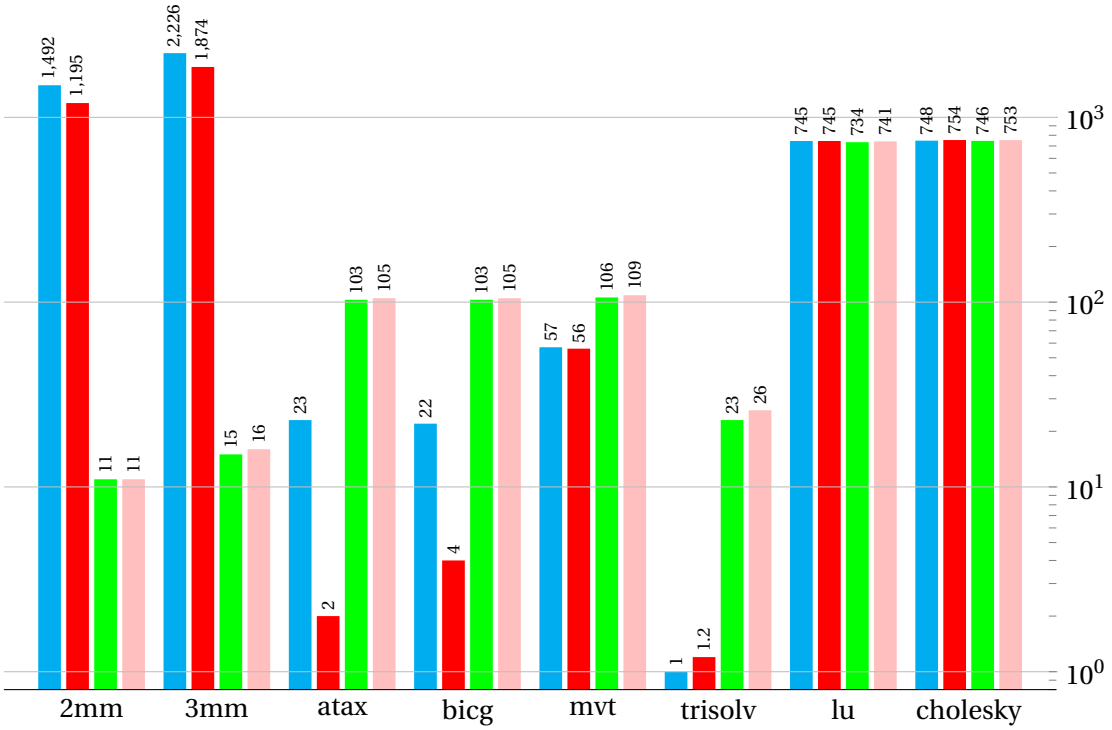


(b) Nombre de L1-dcache-loads (Milliards).

FIGURE 6.11 – Comparaison de nombre d'instructions et L1-dcache-loads entre les allocations statique et dynamique pour les benchmarks de PolyBench.



(a) Nombre de L1-dcache-misses (Millions).



(b) Nombre de L3-cache-misses (Millions).

FIGURE 6.12 – Comparaison de défauts de cache L1 et L3 entre les allocations statique et dynamique pour les benchmarks de PolyBench.

6.5.2 Analyse

Sur huit benchmarks sous formes originales et optimisées par Pluto, six présentent des variations de performances significatives, soit sous forme originale, soit sous forme optimisée Pluto, entre l'allocation statique et l'allocation dynamique. Les différences sont principalement dues au nombre d'instructions vectorisées et au nombre de loas et de défauts de cache.

Quelque chose qui est difficile à expliquer est la raison pour laquelle l'accès aux matrices allouées statiques provoque plus d'accès (et défauts) à la mémoire totale que les accès dynamiques dans les versions originales des codes, pour 4 des 8 benchmarks. Dans la version allouée dynamiquement, il existe un accès mémoire supplémentaire potentiel pour chaque accès au tableau effectué ; il pourrait être mis en cache par réutilisation temporelle via un registre si les tableaux sont accessibles ligne par ligne, mais il pourrait difficilement avoir un effet opposé sur le nombre total d'accès à la mémoire. La vectorisation pourrait l'expliquer, car les accès à la mémoire sont regroupés lorsqu'ils sont consultés en tant que vecteur, mais cela ne peut se produire que dans 2 des 4 de ces benchmarks. Une autre possibilité est que cela pourrait être un effet secondaire d'une pression plus élevée sur l'allocation des registres, résultant en une décision non optimale du compilateur gcc et finalement en plus d'accès à la mémoire. Cet effet pourrait s'inverser dans les versions Pluto des codes puisque les versions optimisées Pluto montrent une localité temporelle plus évidente qui pourrait être exploitée par le compilateur. Des recherches supplémentaires devraient être effectuées pour évaluer la pression sur l'allocation des registres et éventuellement valider cette hypothèse. En conclusion, dans les versions originales des codes, les versions statiques fonctionnent mieux que les versions dynamiques dans 50% de nos benchmarks sur notre plateforme expérimentale config1. Sur config2, nous avons observé le contraire. En plus de notre première série de benchmarks sur les codes 2d-Packed, ces résultats sur PolyBench montrent que le code le plus performant après optimisation par Pluto semble imprévisible : parfois l'allocation statique fonctionne mieux grâce à la vectorisation (comme dans `Cholesky` et `Matmul` sur config1 dans la première série d'expériences) et parfois l'allocation dynamique est plus performante (comme sur config2 ou dans `lu` ou `2mm` sur config1 dans la deuxième série d'expériences).

6.6 Conclusion :

Nous avons montré dans ce chapitre que le type d'allocation de mémoire, en tant que tableau déclaré statique ou tableau alloué dynamiquement, a un impact significatif sur les performances de certains noyaux d'algèbre linéaire. Cela se produit à la fois sur les codes séquentiels et sur les codes automatiquement parallélisés et optimisés. Dans notre étude expérimentale, les causes potentielles identifiées de ces variations de performances sont (1) la capacité du compilateur à vectoriser les noyaux, et (2) la variation du nombre d'accès à la mémoire et de défauts de cache effectués dans le code généré. La variation des performances peut être alternativement en faveur du mode d'allocation statique ou du mode dynamique, et peut même basculer sur différentes architectures de processeurs cibles de manière imprévisible.

D'autres expériences pourraient être menées dans les travaux futurs sur différentes architectures de processeur, différents systèmes d'exploitation, en utilisant différentes options de compilation et différents compilateurs pour confirmer les causes possibles de la variation des performances due à l'allocation de mémoire.

Conclusion générale et perspectives

Notre objectif majeur est de contribuer à l'amélioration des performances des systèmes multi-cœurs. Ceci pourrait se faire à travers des transformations sur des parties du code original en vue d'aboutir à un programme cible dans lequel les délais de réponse sont réduits le plus possible. Nous nous sommes intéressés dans ce travail à l'optimisation des codes de l'algèbre linéaires sur des matrices creuses particulières en utilisant une nouvelle structure de données qui permet de ne pas stocker une grande partie des coefficients nuls et de ré duire considérablement la complexité des calculs sur les systèmes linéaires. L'idée est que les opérations matricielles utilisant ces structures de données, allouées dynamiquement ou statiquement, peuvent être automatiquement optimisées et parallélisées au moyen du modèle polyédrique. Plusieurs formats de stockage similaires ont été proposés afin de stocker et de calculer uniquement des éléments non nuls. Parmi ces formats, Nous avons cité :

— **Linear Packed Format (LPF) :** dans ce format, la matrice triangulaire ou en bande est stockée dans un tableau unidimensionnel. Ce qui fait que les outils de parallélisation et d'optimisation basés sur le modèle polyédrique ne peuvent pas optimiser les transformations correspondantes qui ne sont pas affines.

— **Rectangular Full Packed Format (RFPF) :** dans ce format, la matrice symétrique ou triangulaire est stockée dans un format rectangulaire. Dans cette proposition, les matrices triangulaires impaires et paires sont stockées différemment, ce qui oblige le programmeur à distinguer entre ces deux cas au moment de l'écriture de son code.

Notre format de représentation 2d-Packed pour les matrices triangulaires est inspiré du format RFPF. Nous avons amélioré ce format de stockage, afin d'effectuer une transformation de données unique pour les matrices de tailles paires et impaires. Nous avons proposé aussi une structure de données 2d-Packed pour représenter les matrices en bande.

La première partie de notre thèse est consacré à l'état de l'art de notre contribution. Cette partie est composée de quatre chapitres : Dans le premier chapitre, Nous avons décrit les architectures parallèles d'une manière générale. Nous avons détaillé les quatre catégories d'architectures parallèles et leurs principes de fonctionnement selon la taxonomie de Flynn. Nous avons aussi montré l'importance de la hiérarchie mémoire et la mémoire cache ainsi que leurs fonctionnalités. Nous avons vu en détail dans la dernière partie de ce chapitre les architectures multi-cœurs et les architectures many-cœurs ainsi que des exemples sur les deux types d'architecture.

Dans le deuxième chapitre, nous nous sommes focalisés sur le modèle polyédrique qui permet de représenter les nids de boucles sous forme de polyèdres afin de construire et de rechercher des séquences complexes d'optimisation ainsi que les différentes notions autour de ce modèle. Nous avons introduit et discuté les différents outils polyédriques existants ainsi que les techniques de transformation et d'optimisation polyédrique. Le troisième chapitre consiste à introduire le domaine de l'optimisation et la parallélisation automatique ainsi que ses langages et techniques. On a détaillé aussi dans ce volet l'outil Pluto qui est un framework de transformation source-to-source, basé sur le modèle polyédrique, permettant d'utiliser le parallélisme afin d'atteindre de meilleures performances pour un programme. A la fin de la partie état de l'art, on a introduit les différents types de matrices et opérations sur ces dernières. Parmi les types de matrices, il y a les matrices creuses qu'on peut trouver dans plusieurs problématiques comme : la théorie des graphes, la résolution d'équations aux dérivées partielles et les systèmes de recommandation. On a introduit aussi les formats élémentaires existants pour le stockage des matrices creuses.

La deuxième partie de notre thèse est la partie contribution. Dans un premier temps on a proposé une nouvelle approche pour optimiser les opérations matricielles triangulaires et en bandes

en utilisant une structure de données dense à deux dimensions pour le stockage de matrices creuses. Donc à ce stade, uniquement les éléments non nuls des matrices triangulaires et en bande sont stockés dans des structures à deux dimensions, ce qui permet une économie importante de la mémoire. Ainsi, le code sous-jacent de l'opération matricielle peut être optimisé et parallélisé en utilisant les outils d'optimisation polyédriques existants pour obtenir les meilleures performances. On a démontré par des résultats expérimentaux l'efficacité de notre approche en parallélisant et en optimisant plusieurs codes de calcul matriciel en utilisant le compilateur polyédrique Pluto, et en comparant leurs performances aux versions séquentielles et parallèles avec d'autre approches proposées antérieurement.

Dans un deuxième temps, on a comparé les performances de plusieurs codes de calcul matriciel en utilisant les deux types d'allocation mémoire, statique et dynamique, sous forme originale et sous forme optimisée et parallélisée. On a montré que le type d'allocation de mémoire, en tant que tableau déclaré statique ou tableau alloué dynamiquement, a un impact significatif sur les performances de certains noyaux d'algèbre linéaire. Cela se produit à la fois sur les codes séquentiels et sur les codes automatiquement parallélisés et optimisés. D'autres expériences pourraient être menées dans les travaux futurs sur différentes architectures de processeur, différents systèmes d'exploitation, en utilisant différentes options de compilation et différents compilateurs pour confirmer les causes possibles de la variation des performances due à l'allocation de mémoire.

En perspectives, nous envisageons à court terme de publier l'automatisation de notre approche 2d-Packed et son intégration dans la bibliothèque CLAPCK. Cette bibliothèque fournit notamment des fonctions pour la résolution de systèmes d'équations linéaires, le calcul de valeurs propres et les décompositions de matrices. L'appel de notre approche avec les différents types de données (réel, double, complexe et double complexe) à travers la bibliothèque CLAPACK permettra d'augmenter les performances des systèmes multi-coeurs. Nous envisageons aussi d'étendre notre approche ou de proposer de nouvelles approches permettant la transformation automatique d'un programme séquentiel en un programme équivalent accéléré pour des architectures et des systèmes multi-cœurs en utilisant de nouveaux outils d'optimisation et de parallélisation automatique comme APOLLO. Nous prévoyons également de proposer d'autres structures de données pour optimiser les opérations de l'algèbre linéaire en utilisant des matrices creuses plus générales sur différentes architectures comme les GPU.

Annexe :Le code sspfa dans le format 2d-Packed Pluto

```
void kernel(double **mtri, int n)
{  int              i, j, k;
 int t1, t2, t3, t4, t5, t6;
 int lb, ub, lbp, ubp, lb2, ub2;
 register int lbv, ubv;
/* Start of CLooG code */
if (n >= 3) {
  for (t1=0;t1<=floord(n-2,16);t1++) {
    lbp=max(0,ceild(32*t1-n+1,32));
    ubp=floord(t1,2);
#pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
    for (t2=lbp;t2<=ubp;t2++) {
      for (t3=0;t3<=min(floord(1952*t1-3712*t2+3477*n-8516,111456),t2);t3++)
    {
        if (t3 >= ceild(n+1,64)) {
          for (t4=max(max(32*t1-32*t2,32*t2+1),32*t3+2);t4<=min(n-1,32*t1-32*
    t2+31);t4++) {
            for (t5=max(32*t2,32*t3+1);t5<=min(32*t2+31,t4-1);t5++) {
              for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
                mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[n - (t4) - 1][n - (t6)] * mtri[n - (t5) - 1][n - (t6)];;
              }
            }
          }
        }
        if ((t1 == 2*t2) && (t1 == 2*t3) && (t1 <= floord(n,32)) && (t1 >=
    ceild(n-1,32))) {
          if (t1%2 == 0) {
            mtri[n - ((16*t1+2)) - 1][n - ((16*t1+1))] = mtri[n - ((16*t1+2))
     - 1][n - ((16*t1+1))] - mtri[(16*t1+2)][16*t1] * mtri[(16*t1+1)][16*t1];;
          }
        }
        if ((t2 >= ceild(n+1,64)) && (t3 <= floord(n-62,64))) {
          for (t4=max(32*t1-32*t2,32*t2+1);t4<=min(n-1,32*t1-32*t2+31);t4++)
    {
            for (t5=32*t2;t5<=min(32*t2+31,t4-1);t5++) {
              for (t6=32*t3;t6<=32*t3+31;t6++) {
                mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[t4][t6] * mtri[t5][t6];;
              }
            }
          }
        }
        if ((t2 <= floord(n-60,64)) && (t2 >= ceild(n-61,64))) {
          for (t4=max(ceild(n+3,2),32*t1-32*t2);t4<=min(n-1,32*t1-32*t2+31);
    t4++) {
            for (t5=max(32*t2,32*t3+1);t5<=floord(n,2);t5++) {
              for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
                mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
              }
            }
            for (t6=32*t3;t6<=min(floord(n,2),32*t3+31);t6++) {
              mtri[n - (t4) - 1][n - ((32*t2+31))] = mtri[n - (t4) - 1][n -
    ((32*t2+31))] - mtri[t4][t6] * mtri[(32*t2+31)][t6];;
            }
          }
        }
        if ((t1 == 2*t2) && (t1 >= ceild(n-61,32))) {
          for (t4=max(16*t1+1,32*t3+2);t4<=floord(n+2,2);t4++) {
            for (t5=max(16*t1,32*t3+1);t5<=t4-1;t5++) {
              for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
                if (t1%2 == 0) {
                  mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
```

```
55                }
56              }
57            }
58          }
59        }
60        if (t2 <= floord(n-62,64)) {
61          for (t4=max(max(32*t1-32*t2,32*t2+1),32*t3+2);t4<=min(n-1,32*t1-32*
    t2+31);t4++) {
62            for (t5=max(32*t2,32*t3+1);t5<=min(32*t2+31,t4-1);t5++) {
63              for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
64                mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
65              }
66            }
67          }
68        }
69        if ((t2 == t3) && (t2 <= floord(n,64)) && (t2 >= ceild(n-1,64))) {
70          for (t4=max(ceild(n+5,2),32*t1-32*t2);t4<=min(n-1,32*t1-32*t2+31);
    t4++) {
71            mtri[n - (t4) - 1][n - ((32*t2+1))] = mtri[n - (t4) - 1][n -
    ((32*t2+1))] - mtri[t4][32*t2] * mtri[(32*t2+1)][32*t2];;
72            for (t5=ceild(n+3,2);t5<=min(32*t2+31,t4-1);t5++) {
73              mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
    mtri[t4][32*t2] * mtri[t5][32*t2];;
74              for (t6=ceild(n+1,2);t6<=t5-1;t6++) {
75                mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[n - (t4) - 1][n - (t6)] * mtri[n - (t5) - 1][n - (t6)];;
76              }
77            }
78          }
79        }
80        if ((t2 >= ceild(n+1,64)) && (t3 <= floord(n,64)) && (t3 >= ceild(n
    -61,64))) {
81          for (t4=max(32*t1-32*t2,32*t2+1);t4<=min(n-1,32*t1-32*t2+31);t4++)
    {
82            for (t5=32*t2;t5<=min(32*t2+31,t4-1);t5++) {
83              for (t6=32*t3;t6<=floord(n,2);t6++) {
84                mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[t4][t6] * mtri[t5][t6];;
85              }
86              for (t6=ceild(n+1,2);t6<=32*t3+31;t6++) {
87                mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[n - (t4) - 1][n - (t6)] * mtri[n - (t5) - 1][n - (t6)];;
88              }
89            }
90          }
91        }
92        if ((t1 == 2*t2) && (t1 == 2*t3) && (t1 <= floord(n-2,32))) {
93          for (t4=ceild(n+3,2);t4<=min(min(floord(n+4,2),n-1),16*t1+31);t4++)
     {
94            for (t5=16*t1+1;t5<=floord(n,2);t5++) {
95              for (t6=16*t1;t6<=t5-1;t6++) {
96                if (t1%2 == 0) {
97                  mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
98                }
99              }
100           }
101           for (t6=16*t1;t6<=floord(n,2);t6++) {
102             if (t1%2 == 0) {
103               mtri[n - (t4) - 1][n - ((t4-1))] = mtri[n - (t4) - 1][n - ((
    t4-1))] - mtri[t4][t6] * mtri[(t4-1)][t6];;
104             }
105           }
106         }
107       }
108       if ((t2 <= floord(n,64)) && (t2 >= ceild(n-59,64)) && (t3 <= floord(n
```

```
    -62,64))) {
109           for (t4=max(ceild(n+3,2),32*t1-32*t2);t4<=min(n-1,32*t1-32*t2+31);
    t4++) {
110             for (t5=32*t2;t5<=floord(n,2);t5++) {
111               for (t6=32*t3;t6<=32*t3+31;t6++) {
112                 mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
113               }
114             }
115             for (t5=ceild(n+1,2);t5<=min(32*t2+31,t4-1);t5++) {
116               for (t6=32*t3;t6<=32*t3+31;t6++) {
117                 mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[t4][t6] * mtri[t5][t6];;
118               }
119             }
120           }
121         }
122         if ((t2 == t3) && (t2 <= floord(n-2,64)) && (t2 >= ceild(n-59,64))) {
123           for (t4=max(ceild(n+5,2),32*t1-32*t2);t4<=min(n-1,32*t1-32*t2+31);
    t4++) {
124             for (t5=32*t2+1;t5<=floord(n,2);t5++) {
125               for (t6=32*t2;t6<=t5-1;t6++) {
126                 mtri[t4][t5] = mtri[t4][t5] - mtri[t4][t6] * mtri[t5][t6];;
127               }
128             }
129             t5 = floord(n+2,2);
130             for (t6=32*t2;t6<=floord(n,2);t6++) {
131               mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
    mtri[t4][t6] * mtri[t5][t6];;
132             }
133             for (t5=ceild(n+3,2);t5<=min(32*t2+31,t4-1);t5++) {
134               for (t6=32*t2;t6<=floord(n,2);t6++) {
135                 mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[t4][t6] * mtri[t5][t6];;
136               }
137               for (t6=ceild(n+1,2);t6<=t5-1;t6++) {
138                 mtri[n - (t4) - 1][n - (t5)] = mtri[n - (t4) - 1][n - (t5)] -
     mtri[n - (t4) - 1][n - (t6)] * mtri[n - (t5) - 1][n - (t6)];;
139               }
140             }
141           }
142         }
143       }
144     }
145   }
146 }
147 /* End of CLooG code */
148 }
```

Listing 1 – Le code sspfa dans le format 2d-Packed généré par Pluto

Bibliographie

[1] R. C. Agarwal, F. G. Gustavson, M. V. Joshi, and M. Zubair. A scalable parallel block algorithm for band cholesky factorization. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1995, San Francisco, California, USA, February 15-17, 1995*, pages 430–435, 1995. 86

[2] Äke Björck. *Numerical Methods in Matrix Computations*. Springer International Publishing, 2015. 72, 74

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. 5

[4] B. S. Andersen, J. Waśniewski, and F. G. Gustavson. A recursive formulation of cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Softw.*, 27(2) :214–244, June 2001. 86

[5] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. 84, 88

[6] D. Argiro, K. Farrar, and S. Kubica. Cantata : visual programming environment for the khoros system. In *COMG*, 1995. 56

[7] K. Arnold and J. Peyton, editors. *A C User's Guide to ANSI C*. Prentice-Hall, Inc., USA, 1992. 54

[8] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 273–282, New York, NY, USA, 2014. Association for Computing Machinery. 78

[9] A. Athanasios Konstantinidis and P. H. J. Kelly. More definite results from the Pluto scheduling algorithm. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Chamonix, France, 2011. 34

[10] A. Athavale, P. Randive, and A. Kambale. Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel codes. *URL http ://www. kpit. com/downloads/research-papers/automatic-parallelization-sequential-codes. pdf*, 2011. 53, 64

[11] U. Banerjee. Loop transformations for restructuring compilers : The foundations. In *Springer US*, 1993. 68

[12] B. Bani-Ismail and G. Kanaan. Comparing different sparse matrix storage structures as index structure for arabic text collection. *Int. J. Inf. Retr. Res.*, 2(2) :52–67, Apr. 2012. 76

[13] D. Banković. Equations in bounded lattices with kronecker delta. *Journal of Multiple-Valued Logic and Soft Computing*, 12, 01 2006. 73

[14] T. Baroudi, V. Loechner, and R. Seghir. Static versus Dynamic Memory Allocation : a Comparison for Linear Algebra Kernels. In *IMPACT 2020, in conjunction with HiPEAC 2020*, Bologna, Italy, Jan. 2020. 2, 114

[15] T. Baroudi, R. Seghir, and V. Loechner. Optimization of triangular and banded matrix operations using 2d-packed layouts. *ACM Trans. Archit. Code Optim.*, 14(4), Dec. 2017. 2, 84, 88

[16] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. 1, 34

[17] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, page 7–16, USA, 2004. IEEE Computer Society. 32

[18] C. Bastoul. Openscop : A specification and a library for data exchange in polyhedral compilation tools. Technical report, Paris-Sud University, France, September 2011. 42

[19] C. Bastoul. Clay : the chunky loop alteration wizardry. Technical report, Paris-Sud University, France, 2012. 43

[20] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 209–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 32, 42, 43

[21] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, page 320–334, Berlin, Heidelberg, 2003. Springer-Verlag. 32, 42

[22] C. Bastoul and L.-N. Pouchet. Candl : the chunky analyzer for dependences in loops. Technical report, LRI, Paris-Sud University, France, 2012. 44

[23] C. Bauer, A. Frink, and R. Kreckel. Introduction to the ginac framework for symbolic computation within the c++ programming language. *Journal of Symbolic Computation*, 33 :1–12, 01 2000. 69

[24] H. Belhadj Amor. *Memory hierarchy in embedded multiprocessor system built around networks on chip*. Theses, University Grenoble Alpes, Oct. 2017. 12

[25] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303, Paphos, Cyprus, Mar. 2010. Springer Verlag. 32, 34

[26] R. Bias, C. Lewis, and D. Gillan. The tortoise and the (soft)ware : Moore's law, amdahl's law, and performance trends for human-machine systems. *J. Usability Studies*, 9(4) :129–151, Aug. 2014. viii, 5, 21, 22

[27] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac : A portable, high-performance, ansi c coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, page 253–260, New York, NY, USA, 1997. Association for Computing Machinery. 65, 86

[28] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical report, Carnegie Mellon University, USA, 1993. 79

[29] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris : The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994. 60

[30] U. Bondhugula, A. Acharya, and A. Cohen. The pluto+ algorithm : A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.*, 38(3), Apr. 2016. 67

[31] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–5, April 2008. 67

[32] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In L. Hendren, editor, *Compiler Construction*, pages 132–146, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 32, 43

[33] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6) :101–113, June 2008. 32, 67

[34] U. K. R. Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. PhD thesis, Ohio State University, USA, 2008. AAI3325799. ix, 2, 32, 33, 36, 67, 68

[35] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 503–510, Washington, DC, USA, 2008. IEEE Computer Society. 1

[36] H. Burkhardt, A. Bieniek, R. Klaus, M. Nolle, G. Schreiber, and H. Schulz-Mirbach. The parallel image processing system pips. Technical report, Universitaet Hamburg, DEU, 1994. ix, 56, 57

[37] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1) :38–53, Jan. 2009. 86

[38] N. Capodieci, R. Cavicchioli, and M. Bertogna. Nvidia gpu scheduling details in virtualized environments : Work-in-progress. In *Proceedings of the International Conference on Embedded Software*, EMSOFT '18. IEEE Press, 2018. 54

[39] G. Chrysos. Intel® xeon phi coprocessor (codename knights corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31, Aug 2012. 24

[40] S. Cook. *CUDA Programming : A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. 54

[41] P. Cremonesi, C. Gennaro, and R. Marega. I/o performance in hybrid mimd+simd machines. In P. Sloot, M. Bubak, and B. Hertzberger, editors, *High-Performance Computing and Networking*, pages 688–697, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 7, 9

[42] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. *ACM Trans. Archit. Code Optim.*, 9(3) :14 :1–14 :37, Oct. 2012. 86

[43] H. Cui, Q. Yi, J. Xue, and X. Feng. Layout-oblivious compiler optimization for matrix computations. *ACM Trans. Archit. Code Optim.*, 9(4) :35 :1–35 :20, Jan. 2013. 2, 80, 84, 87, 100

[44] M. A. Cusumano and D. B. Yoffie. Extrapolating from moore's law. *Commun. ACM*, 59(1) :33–35, Dec. 2015. 21

[45] L. Dagum and R. Menon. Openmp : An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1) :46–55, Jan. 1998. 54

[46] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling, 1994. 68

[47] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus : A source-to-source compiler infrastructure for multicores. *Computer*, 42(12) :36–42, Dec. 2009. 60

[48] G. Dharb, N. Mansoor, S. Shahriat, and A. Ganguly. Pase : A parallel speedup estimation framework for network-on-chip based multicore systems. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, Oct 2017. 18

[49] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1) :1–17, Mar. 1990. 86

[50] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. Stewart. *LINPACK Users' Guide*. pub-SIAM, 1979. 84

[51] I. Fassi. *XFOR (Multifor) : A New Programming Structure to Ease the Formulation of Efficient Loop Optimizations*. Theses, Université de Strasbourg, Nov. 2015. 42, 44

[52] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1) :23–53, Feb 1991. 68

[53] P. Feautrier. Some efficient solutions to the affine scheduling problem : I. one-dimensional time. *Int. J. Parallel Program.*, 21(5) :313–348, Oct. 1992. 68

[54] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22, 08 1996. 68, 69

[55] M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1) :67–70, Mar. 1996. 5

[56] S. Geneves. *Performance studies on multicore processors : efficient event-driven runtime and programming models comparison*. Theses, University of Grenoble, Apr. 2013. 18

[57] C. Gennaro. *Models for Simd, Mimd and Hybrid Parallel Architectures Models for Simd, Mimd and Hybrid Parallel Architectures*. PhD thesis, Dipartimento di Elettronica e Informazione, POLITECNICO DI MILANO, 2004. 5, 6

[58] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2) :345–363, 1973. 108

[59] C. Gerth. *Dependency Analysis*, pages 107–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 38

[60] J. R. Gilbert, S. Reinhardt, and V. B. Shah. High-performance graph algorithms from parallel sparse matrices. In *Proceedings of the 8th International Conference on Applied Parallel Computing : State of the Art in Scientific Computing*, PARA'07, pages 260–269, Berlin, Heidelberg, 2007. Springer-Verlag. 1, 76

[61] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, page 124–131, New York, NY, USA, 1983. Association for Computing Machinery. 14

[62] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, University of Passau, 2004. 1, 53

[63] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004. 67

[64] J. Guo, K. Dai, and Z. Wang. A heterogeneous multi-core processor architecture for high performance computing. In C. Jesshope and C. Egan, editors, *Advances in Computer Systems Architecture*, pages 359–365, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 18

[65] A. Gupta, S. Koric, and T. George. Sparse matrix factorization on massively parallel computers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1 :1–1 :12, New York, NY, USA, 2009. ACM. 1, 76

[66] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5) :532–533, May 1988. 5

[67] F. G. Gustavson, J. Waśniewski, J. J. Dongarra, and J. Langou. Rectangular full packed format for cholesky's algorithm : Factorization, solution, and inversion. *ACM Trans. Math. Softw.*, 37(2) :18 :1–18 :21, Apr. 2010. 81, 84, 86, 88, 100, 107

[68] A. Holman. The meiko computing surface : A parallel and scalable open systems platform for oracle. In *Proceedings of the 10th British National Conference on Databases : Advanced Database Systems*, BNCOD 10, page 96–114, Berlin, Heidelberg, 1992. Springer-Verlag. 11

[69] Hong-Soog Kim, Young-Ha Yoon, Sang-Og Na, and Dong-Soo Han. Icu-pfc : an automatic parallelizing compiler. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, volume 1, pages 243–246 vol.1, May 2000. 61

[70] F. Hoseini, A. Atalar, and P. Tsigas. Modeling the performance of atomic primitives on modern architectures. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery. 23

[71] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao. *Optimization of Dense Matrix Multiplication on IBM Cyclops-64 : Challenges and Experiences*, pages 134–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 86

[72] A. Hurson, J. Lim, K. Kavi, and K. Lee. Parallelization of doall and doacross loops-a survey. *Advances in Computers*, 45 :53–103, 12 1997. 59

[73] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for openmp : Ipat/omp. *IEICE - Trans. Inf. Syst.*, E89-D(2) :399–407, Feb. 2006. ix, 62, 63

[74] N. B. Jensen and S. Karlsson. Improving loop dependence analysis. *ACM Trans. Archit. Code Optim.*, 14(3), Aug. 2017. 38

[75] H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, and X. Martorell. Automatic multilevel parallelization using openmp. *Sci. Program.*, 11(2) :177–190, Apr. 2003. 54, 62

[76] S. C. Johnson and R. Sethi. *Yacc : A Parser Generator*, page 347–374. W. B. Saunders Company, USA, 1990. 61

[77] C. W. Keundefinedler. Parallel fourier-motzkin elimination. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, page 66–71, Berlin, Heidelberg, 1996. Springer-Verlag. 68

[78] M. Kim, H. Kim, and C.-K. Luk. Prospector : A dynamic data-dependence profiler to help parallel programming. In *International Conference on Parallel Architectures and Compilation Techniques*, Berkeley, CA, June 2010. USENIX Association. 64

[79] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. Sparse matrix-vector multiplication on gpgpu clusters : A new storage format and a scalable implementation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, IPDPSW '12, page 1696–1702, USA, 2012. IEEE Computer Society. 79

[80] M. Kruliš, J. Lokoč, and T. Skopal. Efficient extraction of feature signatures using multi-gpu architecture. In S. Li, A. El Saddik, M. Wang, T. Mei, N. Sebe, S. Yan, R. Hong, and C. Gurrin, editors, *Advances in Multimedia Modeling*, pages 446–456, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 27

[81] H. T. Kung and J. Subhlok. A new approach for automatic parallelization of blocked linear algebra computations. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 122–129, New York, NY, USA, 1991. ACM. 53

[82] L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2) :83–93, Feb. 1974. 68

[83] C. Lattner and V. Adve. Llvm : A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. 66

[84] J. Levine and L. John. *Flex and Bison*. O'Reilly Media, Inc., 1st edition, 2009. 61

[85] V. Loechner. Polylib : A library for manipulating parameterized polyhedra, 1999. 69

[86] T. Loos and R. Bramley. Mpi performance on the sgi power challenge. In *Proceedings of the Second MPI Developers Conference*, MPIDC '96, page 203, USA, 1996. IEEE Computer Society. 60

[87] J.-P. Lozi. *Towards more scalable mutual exclusion for multicore architectures*. Theses, Universite Pierre et Marie Curie - Paris VI, July 2014. viii, 18, 23

[88] J. Manuel, M. Caamaño, A. Sukumaran Rajam, A. Baloian, M. Selva, and P. Clauss. Apollo : Automatic speculative polyhedral loop optimizer, 01 2017. 66, 86

[89] G. M. Megson and X. Chen. *Automatic parallelization for a class of regular computations*. World Scientific, 1997. 53

[90] L. D. Mey, 2019. viii, 13

[91] S. Midkiff. *Automatic Parallelization : An Overview of Fundamental Compiler Techniques*, volume 7. Synthesis Lectures on Computer Architecture, 01 2011. 53

[92] E. Montagne and A. Ekambaram. An optimal storage format for sparse matrices. *Inf. Process. Lett.*, 90(2) :87–92, Apr. 2004. 79

[93] A. P. Mullhaupt and K. S. Riedel. Banded matrix fraction representation of triangular input normal pairs. *IEEE Transactions on Automatic Control*, 46(12) :2018–2022, Dec 2001. 86

[94] P. Nandy, M. Hall, E. C. Davis, C. Olschanowsky, M. S. Mohammadi, W. He, and M. Strout. Abstractions for specifying sparse matrix data transformations. In *Proceedings of the Eighth International Workshop on Polyhedral Compilation Techniques*, 2018. 76, 77, 78

[95] R. Nelson. Including queueing effects in amdahl's law. *Commun. ACM*, 39(12es) :231–es, Dec. 1996. 5

[96] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2) :40–53, Mar. 2008. 54

[97] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2. 54

[98] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96 :879–899, 05 2008. viii, 27, 28, 29

[99] Parhami.B. *Shared-Memory MIMD Machines*, pages 439–457. Springer US, Boston, MA, 2002. 10

[100] T. J. Parr and R. W. Quong. Antlr : A predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7) :789–810, July 1995. 61

[101] L. Polok and P. Smrz. Increasing double precision throughput on nvidia maxwell gpus. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, San Diego, CA, USA, 2016. Society for Computer Simulation International. 27

[102] L. Pouchet. Polybench/c 4.1 : The polyhedral benchmark suite, 2015. 123

[103] L.-N. Pouchet, C. Bastoul, and AlbertCohen. Letsee : the legal transformation space explorator. Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007. Extended abstract, pp 247–251. 32

[104] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model : Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press. 32

[105] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010. IEEE Computer Society Press. 32

[106] K. Psarris. The banerjee-wolfe and gcd tests on exact data dependence information. *J. Parallel Distrib. Comput.*, 32(2) :119–138, Feb. 1996. 60

[107] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools : The Guide for Application Developers*. Apress, USA, 1st edition, 2013. viii, 24, 25, 26

[108] A. S. Rajam, L. E. Campostrini, J. M. M. Caamaño, and P. Clauss. Speculative runtime parallelization of loop nests : Towards greater scope and efficiency. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 245–254, May 2015. ix, 67

[109] C. P. Ribeiro. *Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms*. PhD thesis, Grenoble Alpes University, France, 2011. viii, 12, 19, 20, 22

[110] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003. 75

[111] D. N. Sampaio, L.-N. Pouchet, and F. Rastello. Simplification and runtime resolution of data dependence constraints for loop transformations. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 10 :1–10 :11, New York, NY, USA, 2017. ACM. 87

[112] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 99–108, New York, NY, USA, 2015. Association for Computing Machinery. 76, 77

[113] S. Shah and M. Bull. Openmp. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 13–es, New York, NY, USA, 2006. Association for Computing Machinery. 54

[114] D. Singh and P. Yiannacouras. *OpenCL*, pages 97–114. Springer International Publishing, Cham, 2016. 57

[115] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.Liu. Knights landing : Second-generation intel xeon phi product. *IEEE Micro*, 36(2) :34–46, Mar 2016. 24

[116] U. Steinberg and B. Kauer. Towards a scalable multiprocessor user-level environment, 04 2010. viii, 6

[117] H. N. Tran. *Cache memory aware priority assignment and scheduling simulation of real-time embedded systems*. Theses, Universite de Bretagne occidentale - Brest, Jan. 2017. 12

[118] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society. 34

[119] A. Varma, W. J. Bowhill, J. Crop, C. Gough, B. Griffith, D. Kingsley, and K. Sistla. Power management in the intel xeon e5 v3. *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 371–376, 2015. viii, 23

[120] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell. Sesam/par4all : A tool for joint exploration of mpsoc architectures and dynamic dataflow code generation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation : Methods and Tools*, RAPIDO '12, page 9–16, New York, NY, USA, 2012. Association for Computing Machinery. 61

[121] X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.*, 26(2) :263–300, Mar. 2004. 14

[122] S. Verdoolaege. Isl : An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, page 299–302, Berlin, Heidelberg, 2010. Springer-Verlag. 32, 69

[123] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31 :1–31 :11, Piscataway, NJ, USA, 2008. IEEE Press. 86

[124] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.*, 11(1), Feb. 2014. 43

[125] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and et al. The suif compiler system : A parallelizing and optimizing research compiler. Technical report, Stanford, CA, USA, 1994. 59

[126] C. Ye, C. Ding, H. Luo, J. Brock, D. Chen, and H. Jin. Cache exclusivity and sharing : Theory and optimization. *ACM Trans. Archit. Code Optim.*, 14(4), Nov. 2017. 23

[127] L. Yuan, Y. Zhang, X. Sun, and T. Wang. Optimizing sparse matrix vector multiplication using diagonal storage matrix format. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, page 585–590, USA, 2010. IEEE Computer Society. 77, 78

[128] H.-K. Yun and H. F. Silverman. A distributed memory mimd multi-computer with reconfigurable custom computing capabilities. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, ICPADS '97, page 8–13, USA, 1997. IEEE Computer Society. 10

[129] O. Zinenko, S. Huot, and C. Bastoul. Clint : A direct manipulation tool for parallelizing compute-intensive program parts. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 109–112, July 2014. 44